Basic Concepts In The Software Industry

By

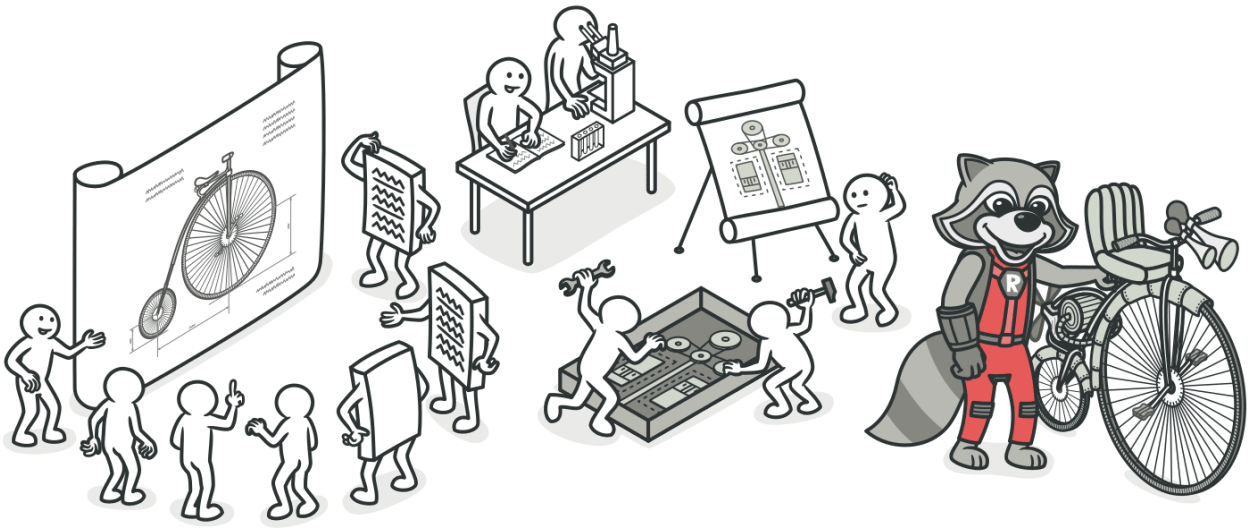
Mohamed El-Fateh Sabry

Mohamed Hesham

Mostafa Ahmed

Ahmed Sultan

Design Pattern



The design pattern is a solution for solving common software or application development problems.

The pattern is not a specific piece of code, but a general concept for solving a particular problem.

Patterns are often confused with algorithms because both concepts describe typical solutions to some known problems. While an algorithm always defines a clear set of actions that can achieve some goal, a pattern is a more high-level description of a solution. The code of the same pattern applied to two different programs may be different.

Pattern Consist Of?

- The intent of the pattern briefly describes both the problem and the solution.
- Motivation further explains the problem and the solution the pattern makes possible.
- The structure of classes shows each part of the pattern and how they are related.
- The code example in one of the popular programming languages makes it easier to grasp the idea behind the pattern.

Patterns are not invented but discovered as recurring solutions to problems in various fields. Patterns were first described in Christopher Alexander's book "A Pattern Language: Towns, Buildings, Construction" for designing urban environments. The idea was then applied to software design by Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm in their book "Design Patterns: Elements of Reusable Object-Oriented Software". This book popularized the concept of design patterns in software engineering.

There are **two main reasons** to learn design patterns even though you might already be writing some code that uses them unknowingly:

- **Better Problem Solving**: Design patterns provide a collection of proven solutions to common design problems. Learning them gives you a toolbox of techniques to tackle these challenges effectively.
- **Communication and Collaboration**: Design patterns establish a common language among programmers. Knowing the names and concepts behind these patterns allows you to communicate design ideas efficiently with your teammates.

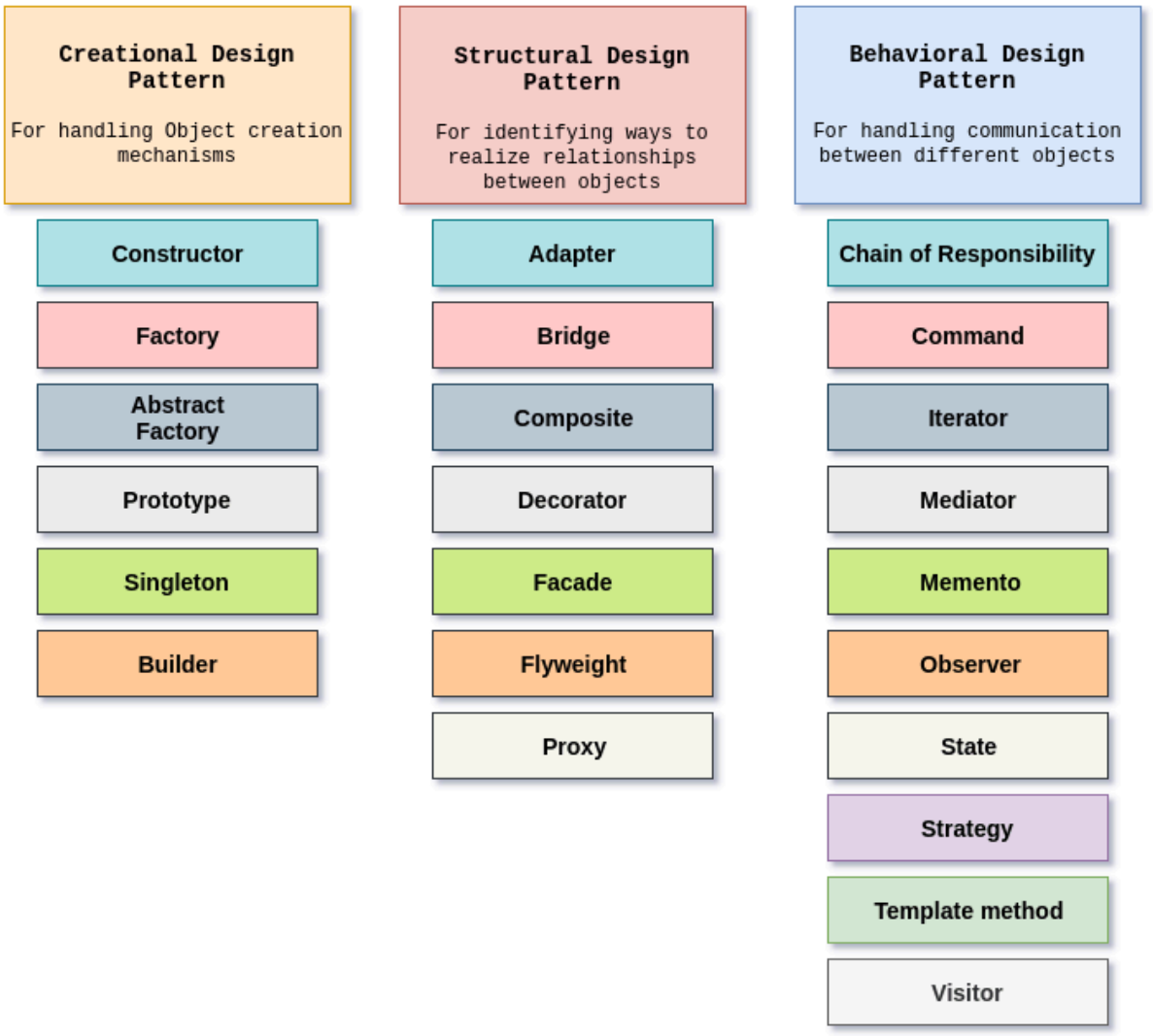
Design patterns are not without criticism. Here are some of the most common complaints:

1. Kludges for weak languages: Some argue that patterns are a workaround for limitations in certain programming languages. For instance, the Strategy pattern might be unnecessary in a language with built-in lambda functions.
2. Inefficient solutions: Critics say patterns can become rigid if applied too strictly. Overly focusing on following a pattern exactly can lead to overly complex code that doesn't fit the specific situation.
3. Overuse: New programmers, excited about learning patterns, might try to use them everywhere, even in simple scenarios where a more straightforward approach would be better.

Design patterns can be classified in two ways:

- By Scope:
    a. Idioms: These are the simplest patterns, often specific to a single programming language.
    b. Architectural Patterns: The most complex patterns, used to design the entire application architecture and work in almost any language.
- By Intent:
    a. Creational Patterns: Focus on creating objects in a flexible and reusable way.
    b. Structural Patterns: Deal with how to assemble objects and classes into larger structures.
    c. Behavioral Patterns: Address communication and responsibility assignment between objects.

Design Patterns In Python

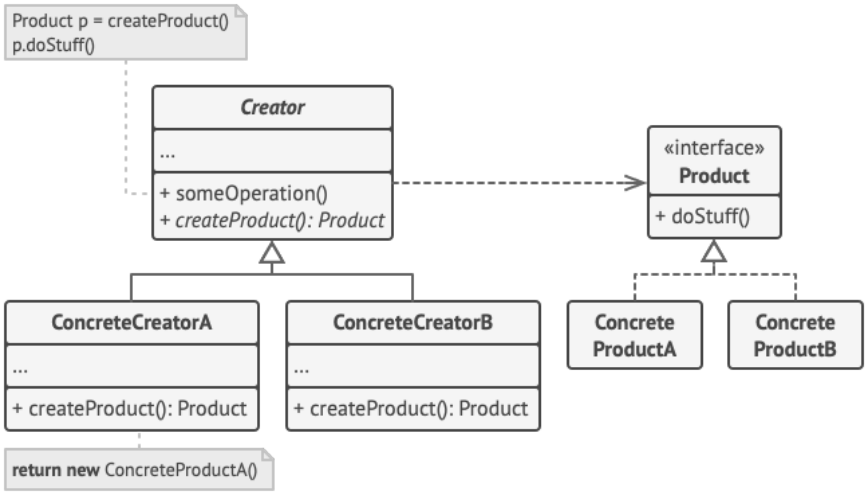


REMINDER!

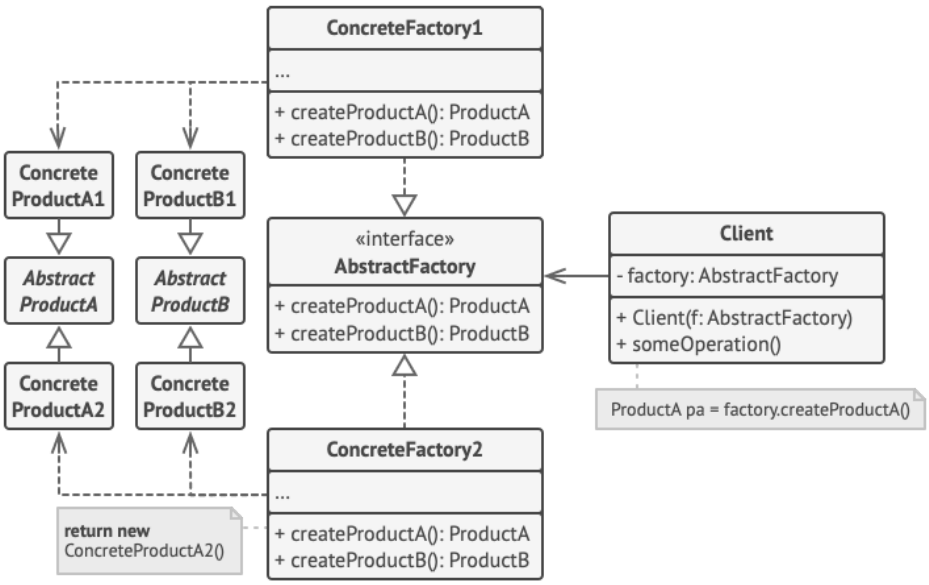
Pattern Consist Of

1. **The Intent**
2. **Motivation**
3. **Classes Structure**
4. **Code Example**

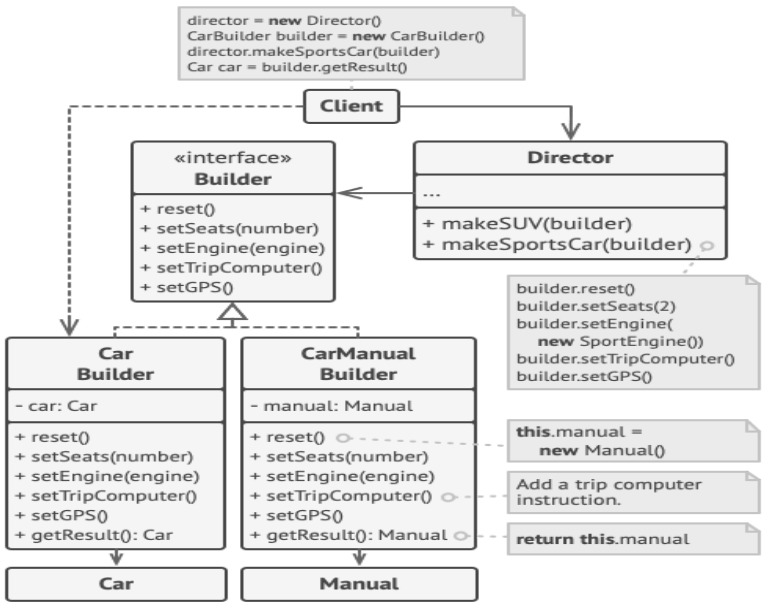
Factory Method

The Intent	Factory Method is a creational design pattern that provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created.
Motivation	= Use the Factory Method when you don't know beforehand the exact types and dependencies of the objects your code should work with. = Use the Factory Method when you want to provide users of your library or framework with a way to extend its internal components. = Use the Factory Method when you want to save system resources by reusing existing objects instead of rebuilding them each time.
Classes Structure	
Code Example	https://refactoring.guru/design-patterns/factory-method/python/example#lang-features

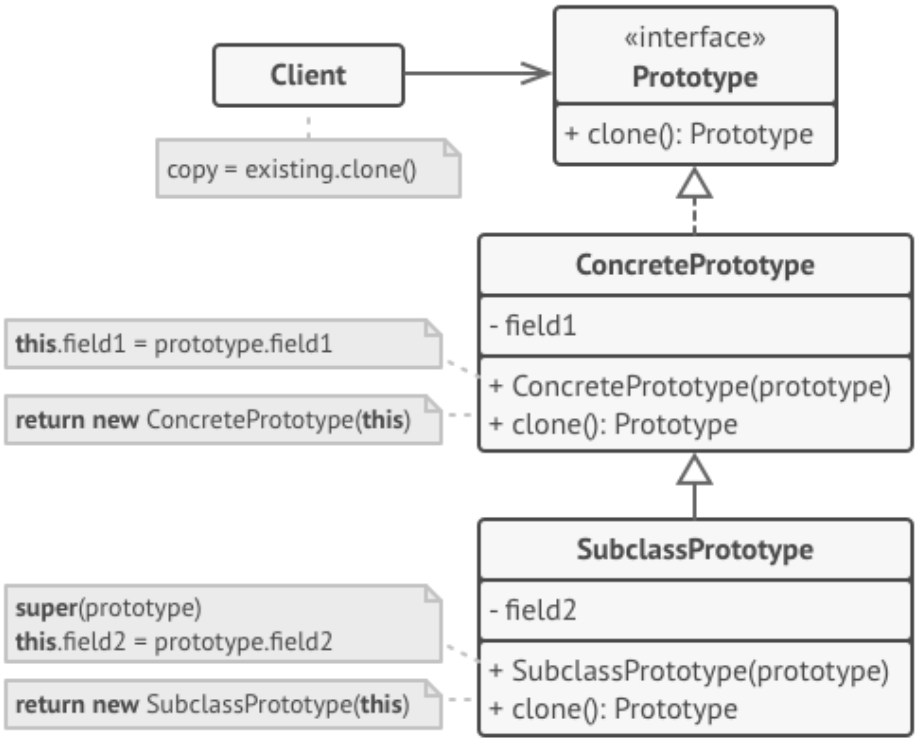
Abstract Factory

The Intent	Abstract Factory is a creational design pattern that lets you produce families of related objects without specifying their concrete classes.
Motivation	= Use the Abstract Factory when your code needs to work with various families of related products, but you don't want it to depend on the concrete classes of those products—they might be unknown beforehand or you simply want to allow for future extensibility. = Consider implementing the Abstract Factory when you have a class with a set of Factory Methods that blur its primary responsibility.
Classes Structure	
Code Example	https://refactoring.guru/design-patterns/abstract-factory/python/example#example-0

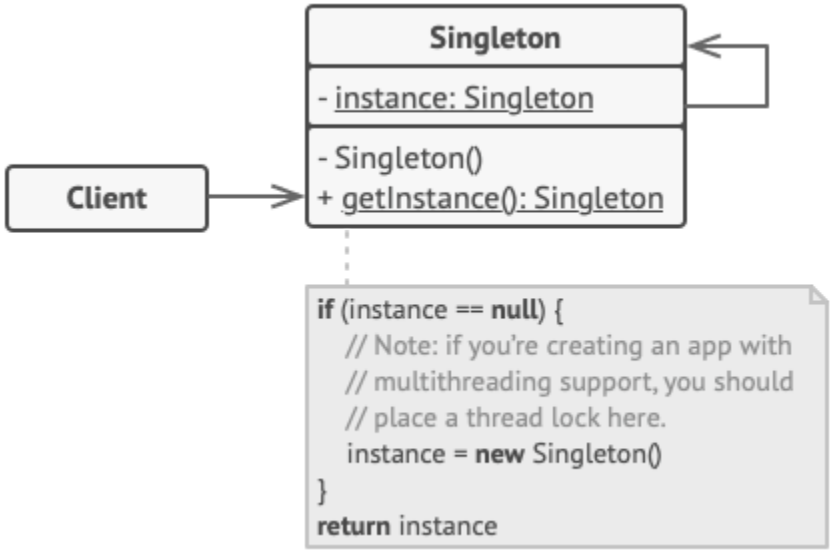
Builder

The Intent	Builder is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.
Motivation	= Use the Builder pattern to get rid of a “telescoping constructor”. = Use the Builder pattern when you want your code to be able to create different representations of some product (for example, stone and wooden houses). = Use the Builder to construct Composite trees or other complex objects.
Classes Structure	
Code Example	[Builder in Python / Design Patterns]

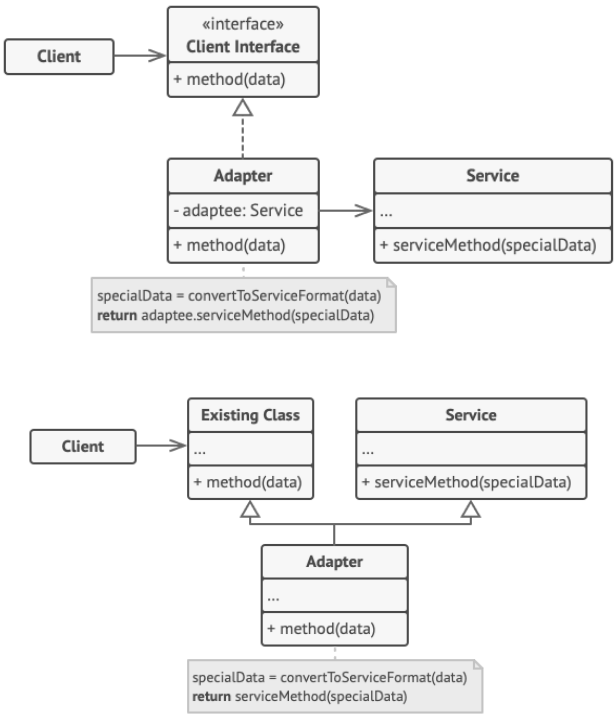
Prototype

The Intent	Prototype is a creational design pattern that lets you copy existing objects without making your code dependent on their classes.
Motivation	= Use the Prototype pattern when your code shouldn't depend on the concrete classes of objects that you need to copy. = Use the pattern when you want to reduce the number of subclasses that only differ in the way they initialize their respective objects.
Classes Structure	
Code Example	[Prototype in Python / Design Patterns]

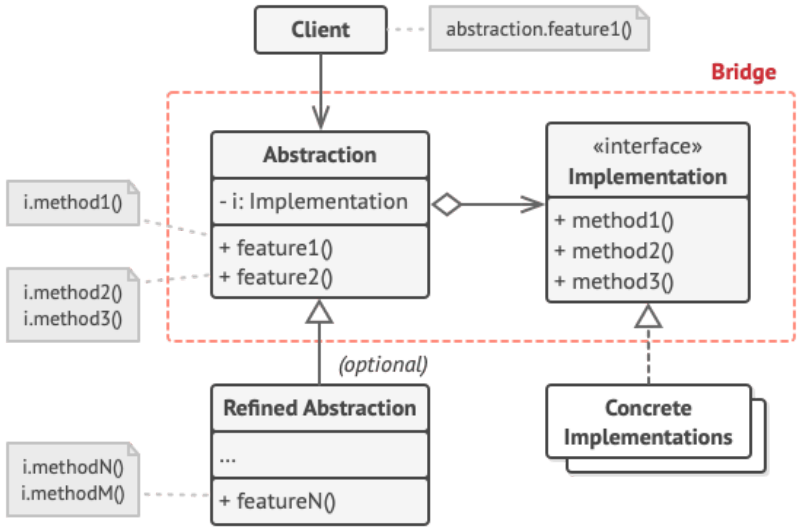
Singleton

The Intent	Singleton is a creational design pattern that lets you ensure that a class has only one instance while providing a global access point to this instance.
Motivation	= Use the Singleton pattern when a class in your program should have just a single instance available to all clients; for example, a single database object shared by different parts of the program. = Use the Singleton pattern when you need stricter control over global variables.
Classes Structure	
Code Example	Singleton in Python / Design Patterns

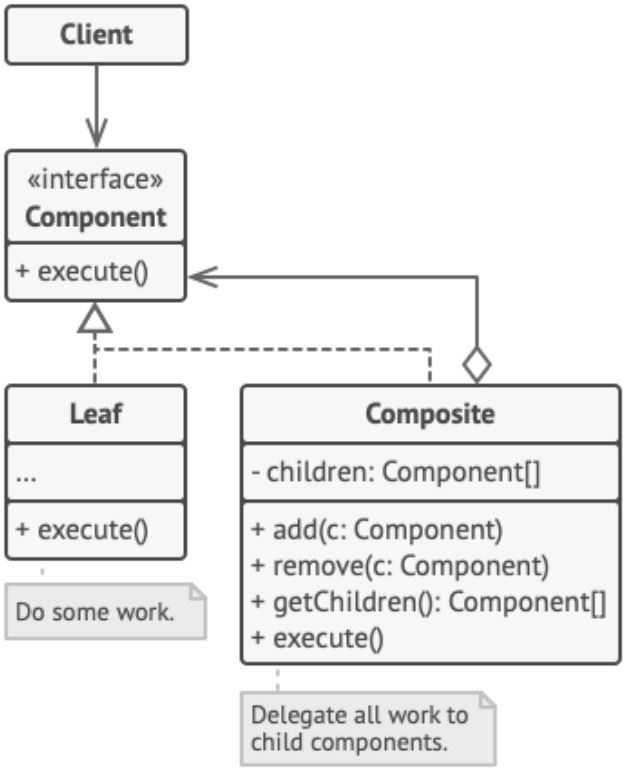
Adapter

The Intent	The adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate.
Motivation	= Use the Adapter class when you want to use some existing class, but its interface isn't compatible with the rest of your code. = Use the pattern when you want to reuse several existing subclasses that lack some common functionality that can't be added to the superclass.
Classes Structure (Object adapter) (Class adapter)	
Code Example	[Adapter in Python / Design Patterns]

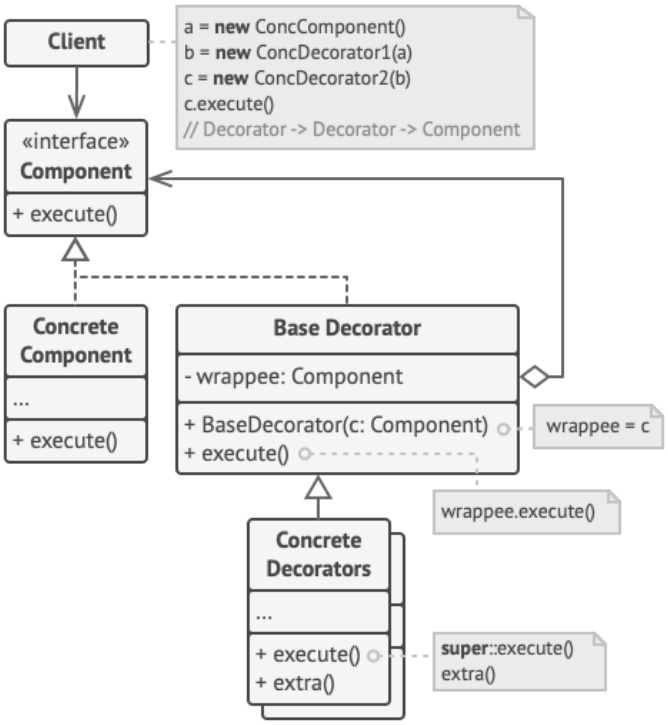
Bridge

The Intent	Bridge is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies: abstraction and implementation which can be developed independently of each other.
Motivation	= Use the Bridge pattern when you want to divide and organize a monolithic class that has several variants of some functionality (for example, if the class can work with various database servers). = Use the pattern when you need to extend a class in several orthogonal (independent) dimensions. = Use the Bridge if you need to be able to switch implementations at runtime.
Classes Structure	
Code Example	[Bridge in Python / Design Patterns]

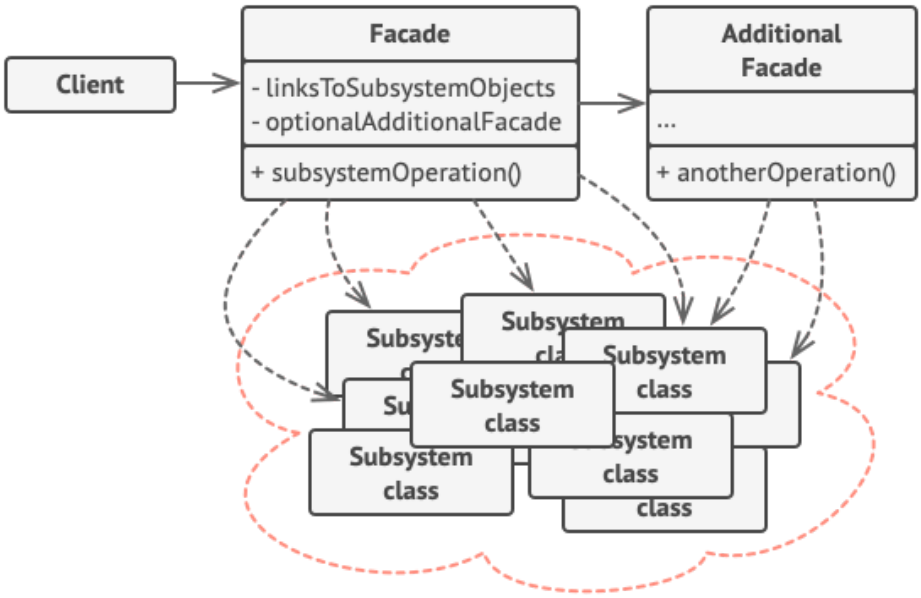
Composite

The Intent	Composite is a structural design pattern that lets you compose objects into tree structures and then work with these structures as if they were individual objects.
Motivation	= Use the Composite pattern when you have to implement a tree-like object structure. = Use the pattern when you want the client code to treat both simple and complex elements uniformly.
Classes Structure	
Code Example	[Composite in Python / Design Patterns]

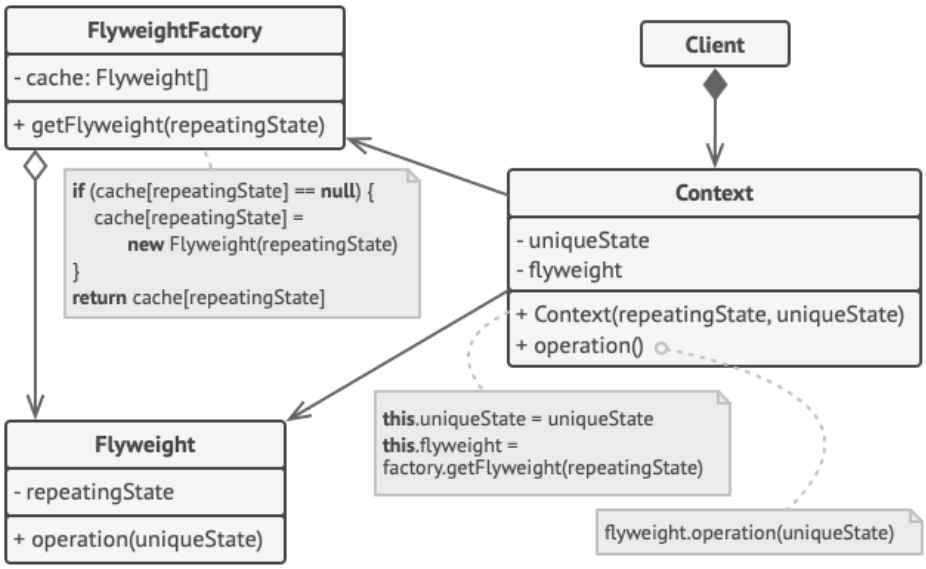
Decorator

The Intent	A decorator is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.
Motivation	= Use the Decorator pattern when you need to be able to assign extra behaviors to objects at runtime without breaking the code that uses these objects. = Use the pattern when it's awkward or not possible to extend an object's behavior using inheritance.
Classes Structure	
Code Example	[Decorator in Python / Design Patterns]

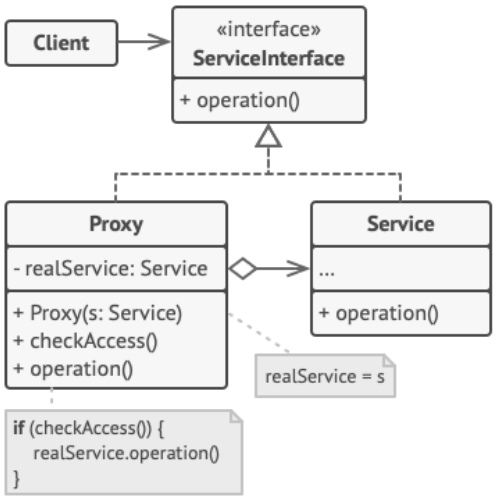
Facade

The Intent	A facade is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes.
Motivation	= Use the Facade pattern when you need to have a limited but straightforward interface to a complex subsystem. = Use the Facade when you want to structure a subsystem into layers.
Classes Structure	
Code Example	[Facade in Python / Design Patterns]

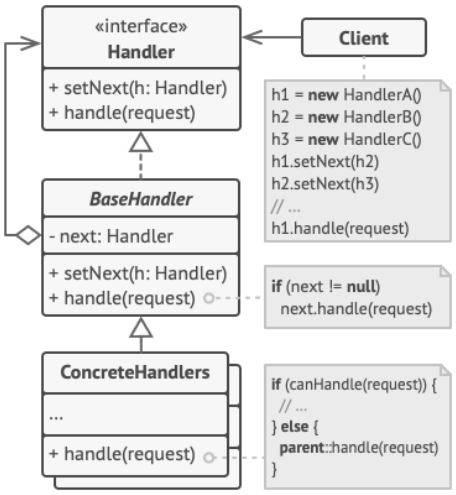
Flyweight

The Intent	Flyweight is a structural design pattern that lets you fit more objects into the available amount of RAM by sharing common parts of the state between multiple objects instead of keeping all of the data in each object.
Motivation	= Use the Flyweight pattern only when your program must support many objects that barely fit into available RAM.
Classes Structure	
Code Example	[Flyweight in Python / Design Patterns]

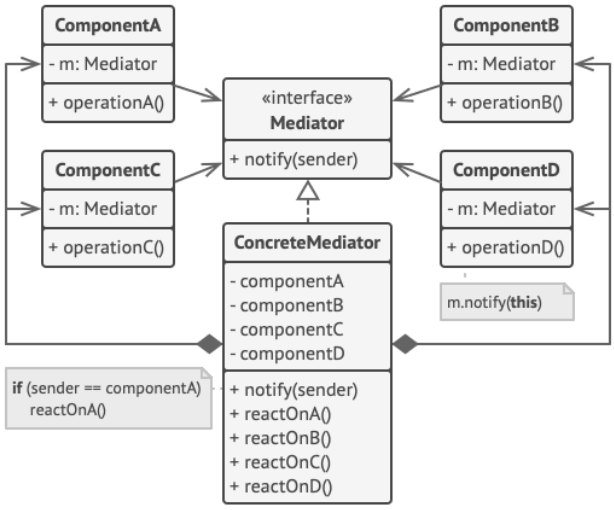
Proxy

The Intent	Proxy is a structural design pattern that lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.
Motivation	= Lazy initialization (virtual proxy). This is when you have a heavyweight service object that wastes system resources by being always up, even though you only need it from time to time. = Access control (protection proxy). This is when you want only specific clients to be able to use the service object; for instance, when your objects are crucial parts of an operating system and clients are various launched applications (including malicious ones). = Logging requests (logging proxy). This is when you want to keep a history of requests to the service object. = Smart reference. This is when you need to be able to dismiss a heavyweight object once there are no clients that use it.
Classes Structure	
Code Example	[Proxy in Python / Design Patterns]

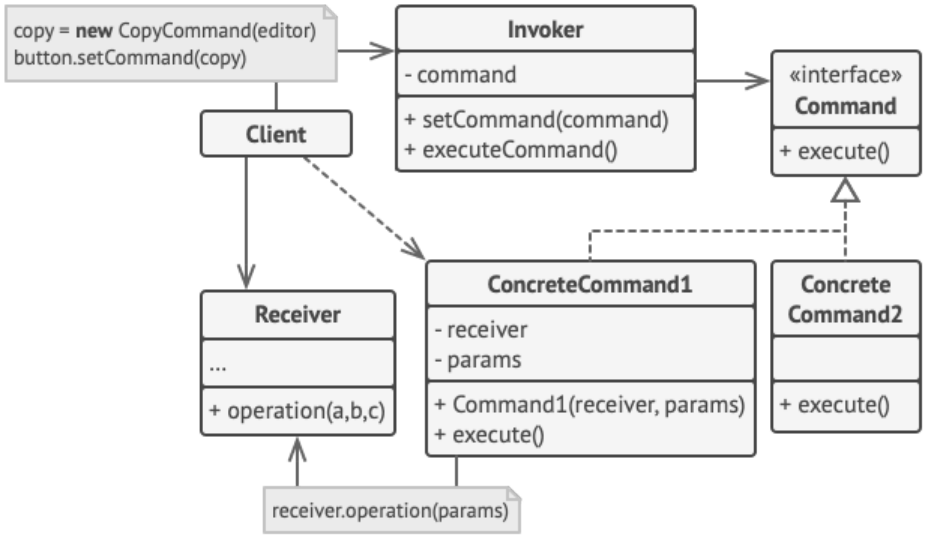
Chain of Responsibility

The Intent	Chain of Responsibility is a behavioral design pattern that lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.
Motivation	= Use the Chain of Responsibility pattern when your program is expected to process different kinds of requests in various ways, but the exact types of requests and their sequences are unknown beforehand. = Use the CoR pattern when the set of handlers and their order are supposed to change at runtime.
Classes Structure	
Code Example	[Chain of Responsibility in Python / Design Patterns]

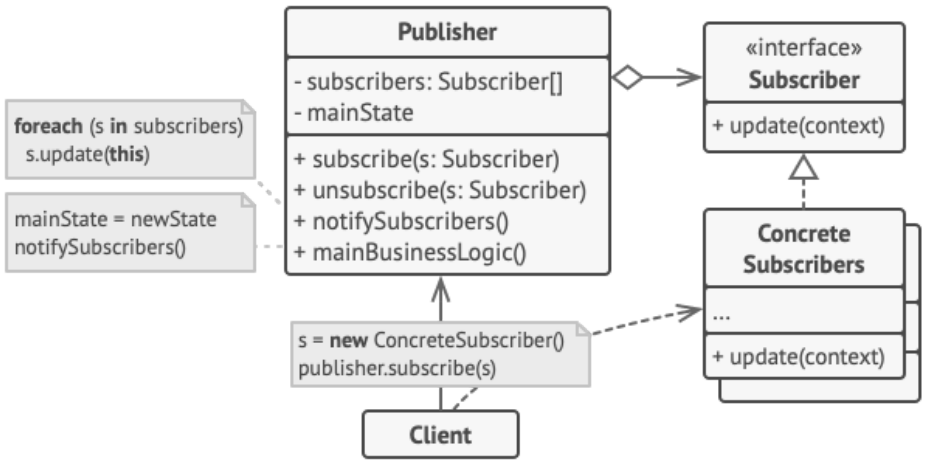
Mediator

The Intent	Mediator is a behavioral design pattern that lets you reduce chaotic dependencies between objects. The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object.
Motivation	= Use the Mediator pattern when it's hard to change some of the classes because they are tightly coupled to a bunch of other classes. = Use the pattern when you can't reuse a component in a different program because it's too dependent on other components. = Use the Mediator when you find yourself creating tons of component subclasses just to reuse some basic behavior in various contexts.
Classes Structure	
Code Example	[Mediator in Python / Design Patterns]

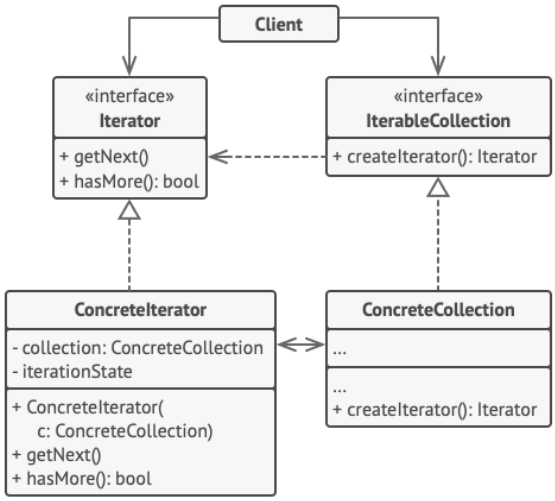
Command

The Intent	The command is a behavioral design pattern that turns a request into a stand-alone object that contains all information about the request. This transformation lets you pass requests as method arguments, delay or queue the request's execution, and support undoable operations.
Motivation	= Use the Command pattern when you want to parametrize objects with operations. = Use the Command pattern when you want to queue operations, schedule their execution, or execute them remotely. = Use the Command pattern when you want to implement reversible operations.
Classes Structure	
Code Example	[Command in Python / Design Patterns]

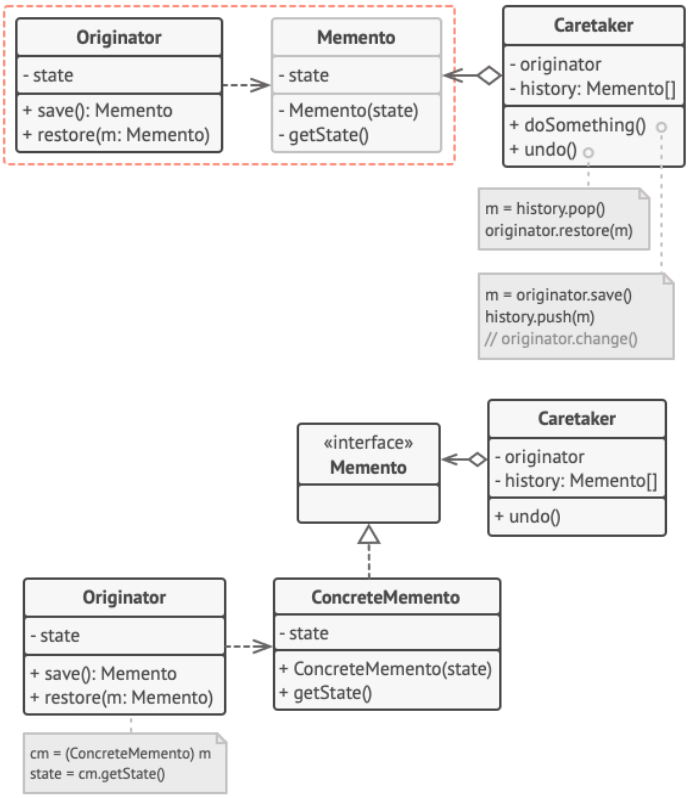
Observer

The Intent	Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.
Motivation	= Use the Observer pattern when changes to the state of one object may require changing other objects, and the actual set of objects is unknown beforehand or changes dynamically. = Use the pattern when some objects in your app must observe others, but only for a limited time or in specific cases.
Classes Structure	
Code Example	[Observer in Python / Design Patterns]

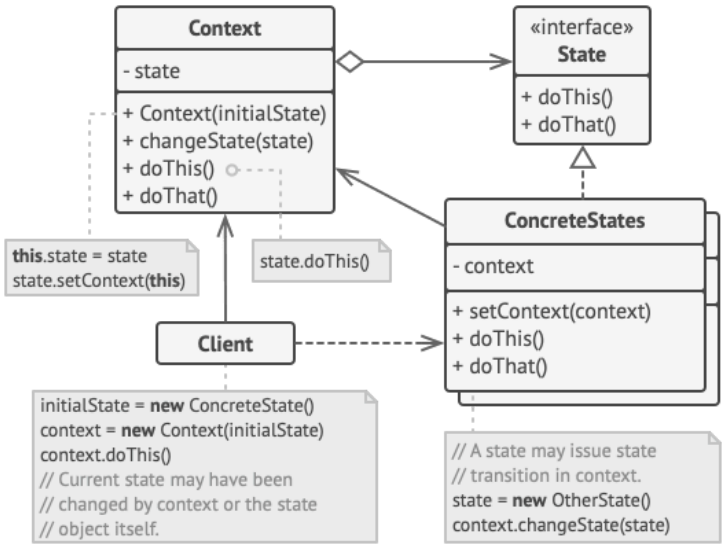
Iterator

The Intent	Iterator is a behavioral design pattern that lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).
Motivation	= Use the Iterator pattern when your collection has a complex data structure under the hood, but you want to hide its complexity from clients (either for convenience or security reasons). = Use the pattern to reduce duplication of the traversal code across your app. = Use the Iterator when you want your code to be able to traverse different data structures or when types of these structures are unknown beforehand.
Classes Structure	
Code Example	[Command in Python / Design Patterns]

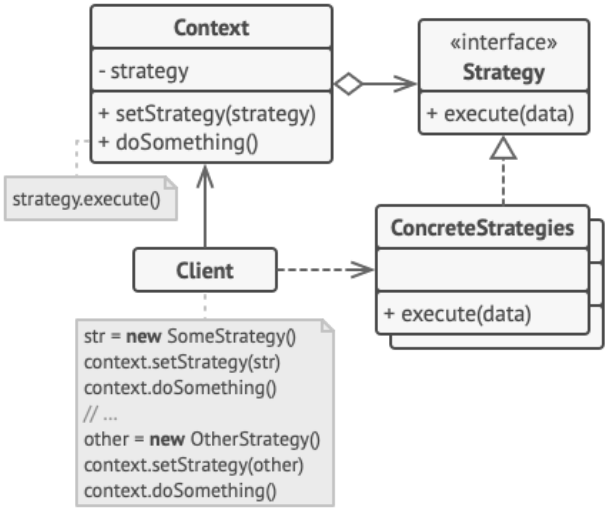
Memento

The Intent	Memento is a behavioral design pattern that lets you save and restore the previous state of an object without revealing the details of its implementation.
Motivation	= Use the Memento pattern when you want to produce snapshots of the object's state to be able to restore a previous state of the object. = Use the pattern when direct access to the object's fields/getters/setters violates its encapsulation.
Classes Structure **"Nested Classes"** **"intermediate interface"**	
Code Example	Memento in Python / Design Patterns

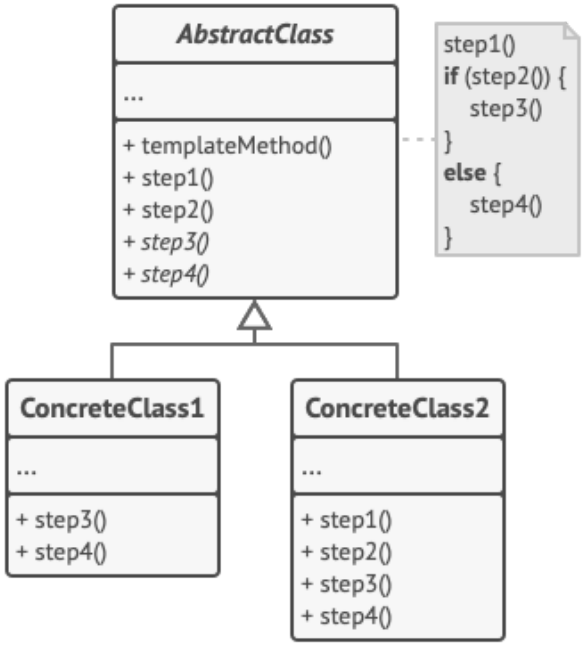
State

The Intent	State is a behavioral design pattern that lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.
Motivation	= Use the State pattern when you have an object that behaves differently depending on its current state, the number of states is enormous, and the state-specific code changes frequently. = Use the pattern when you have a class polluted with massive conditionals that alter how the class behaves according to the current values of the class's fields. = Use State when you have a lot of duplicate code across similar states and transitions of a condition-based state machine.
Classes Structure	
Code Example	[State in Python / Design Patterns]

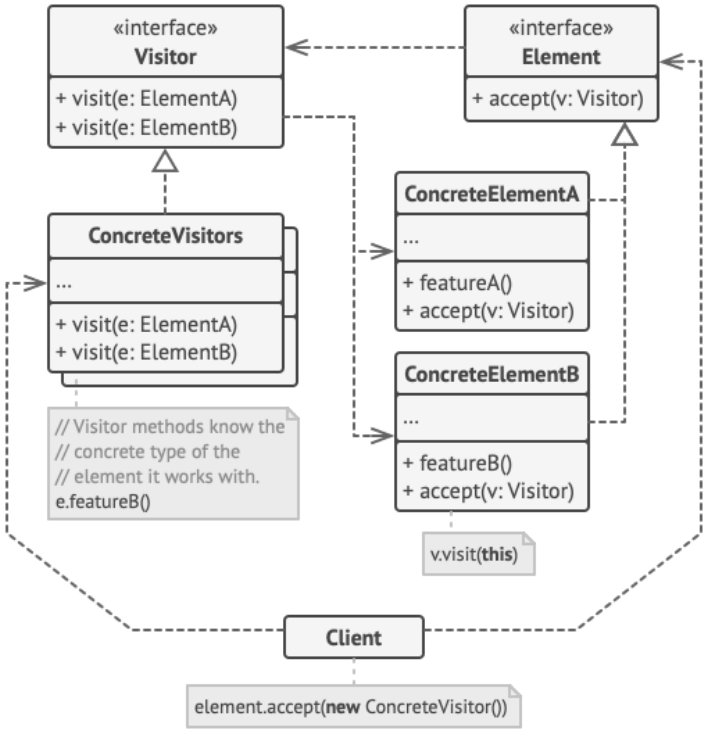
Strategy

The Intent	Strategy is a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.
Motivation	= Use the Strategy when you have a lot of similar classes that only differ in the way they execute some behavior. = Use the Strategy pattern when you want to use different variants of an algorithm within an object and be able to switch from one algorithm to another during runtime. = Use the pattern when your class has a massive conditional statement that switches between different variants of the same algorithm.
Classes Structure	
Code Example	Strategy in Python / Design Patterns

Template Method

The Intent	The Template Method is a behavioral design pattern that defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.
Motivation	= Use the Template Method pattern when you want to let clients extend only particular steps of an algorithm, but not the whole algorithm or its structure. = Use the pattern when you have several classes that contain almost identical algorithms with some minor differences. As a result, you might need to modify all classes when the algorithm changes.
Classes Structure	
Code Example	Template Method in Python / Design Patterns

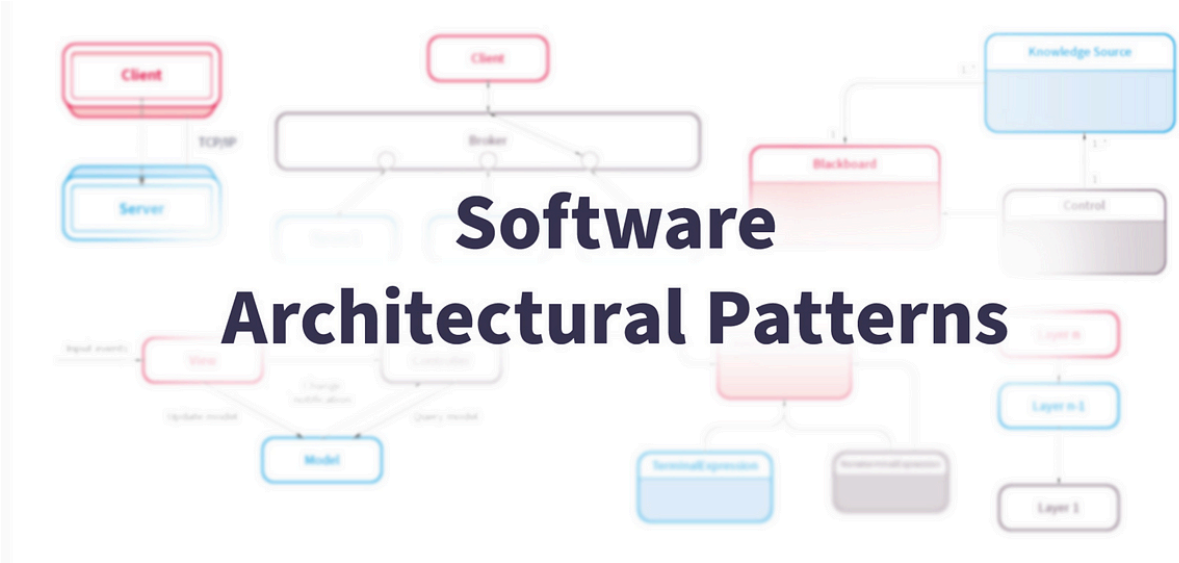
Visitor

The Intent	Visitor is a behavioral design pattern that lets you separate algorithms from the objects on which they operate.
Motivation	= Use the Visitor when you need to operate on all elements of a complex object structure (for example, an object tree). = Use the Visitor to clean up the business logic of auxiliary behaviors. = Use the pattern when a behavior makes sense only in some classes of a class hierarchy, but not in others.
Classes Structure	
Code Example	[Visitor in Python / Design Patterns]

Software Architecture Pattern

A software architecture pattern defines a system's high-level structure and organization. It outlines the fundamental components, their interactions, and the overall layout of the system.

Architectural patterns guide decisions about the system’s scalability, performance, and maintainability. They focus on the system’s macro-level aspects and establish a framework for the design and implementation of the entire application.

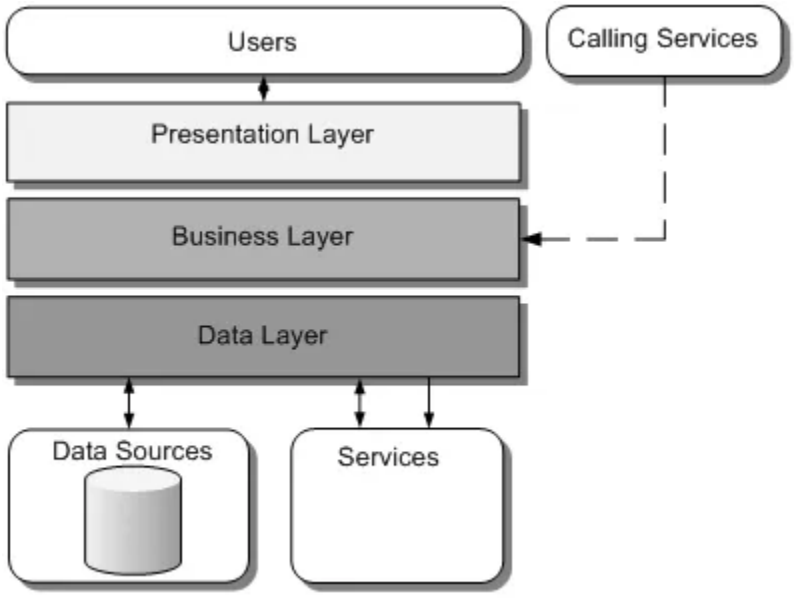


N-Tier Pattern

N-tier architecture, also known as multi-tier architecture, separates software into distinct layers for processing, data management, and presentation. These layers are logically and structurally differentiated to promote modularity, maintainability, and scalability.

Key Concepts:

- **Separation of Concerns**: Each layer has specific responsibilities, making the system easier to understand, maintain, and extend.
- **Layer Interaction**: Layers interact with each other in a defined manner, typically with each layer only communicating with the layers directly adjacent to it.

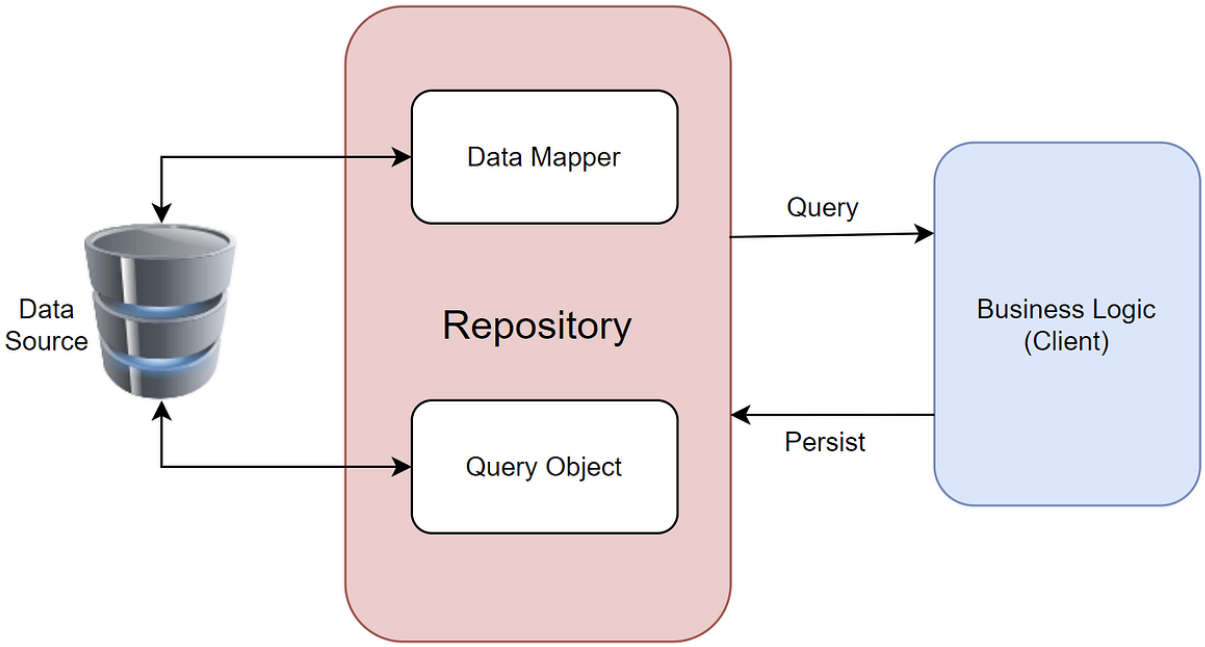


Repository Pattern

The repository architecture pattern is a design pattern that abstracts data access and persistence, providing a clean separation between business logic and data access logic. This pattern promotes maintainability, flexibility, and testability of the codebase.

Key Concepts:

- **Repository**: Acts as an intermediary between the domain and data layers, offering a simple interface for data operations.
- **Separation of Concerns**: Separates business logic from data access logic.
- **Abstraction**: Hides the implementation details of data access, making it easier to change the data source without affecting the business logic.

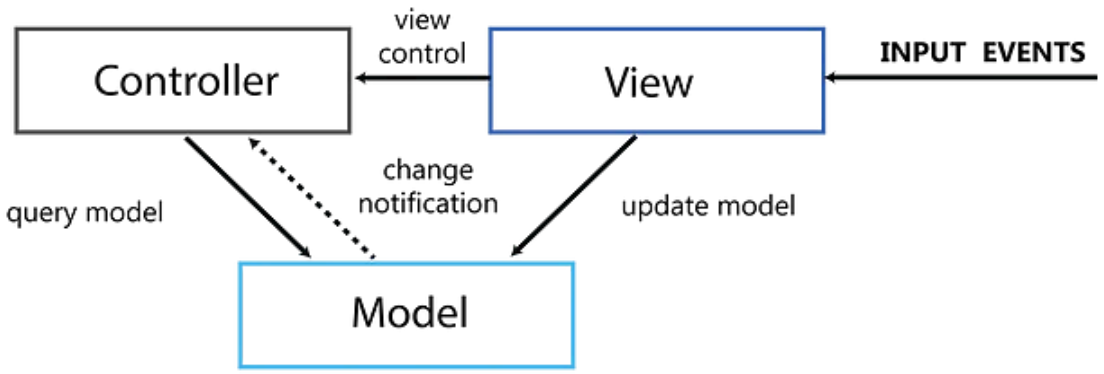


Model View Controller (MVC) Pattern

The Model-View-Controller (MVC) pattern is a design pattern used to separate application logic into three interconnected components: the Model, the View, and the Controller. This separation of concerns allows for a more modular, maintainable, and testable codebase.

Key Components:

- **Model**:
  - Responsibility: Represents the application data and business logic.
  - Example: Database interactions, data manipulation, business rules.
- **View**:
  - Responsibility: Represents the user interface and displays data to the user.
  - Example: HTML/CSS for web apps, UI components for desktop/mobile apps.
- **Controller**:
  - Responsibility: Acts as an intermediary between the Model and the View. It handles user input, updates the Model, and refreshes the View.
  - Example: Handling form submissions, routing requests, updating UI based on user actions.

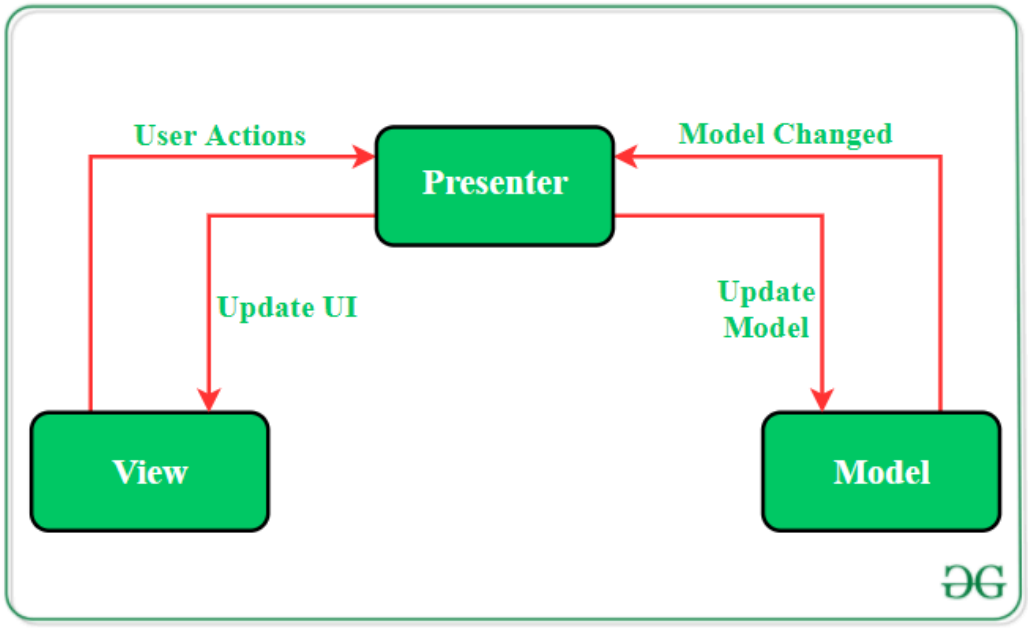


Model View Presenter (MVP) Pattern

The Model-View-Presenter (MVP) pattern is an architectural pattern that overcomes the challenges of the Model-View-Controller (MVC) pattern by providing a more modular, testable, and maintainable structure for project codebases. The MVP pattern separates concerns into three components: Model, View, and Presenter.

Key Components:

- **Model:**
  - Responsibility: Manages the data layer, handling domain logic (real-world business rules) and communication with the database and network layers.
  - Example: Database queries, data manipulation, network requests.
- **View:**
  - Responsibility: Represents the UI layer, displaying data to the user and capturing user interactions.
  - Example: UI elements, data visualization, user input handling.
- **Presenter:**
  - Responsibility: Acts as an intermediary between the Model and the View, fetching data from the Model and applying UI logic to update the View. It manages the state of the View and responds to user input.
  - Example: Data fetching, updating UI elements, handling user actions.

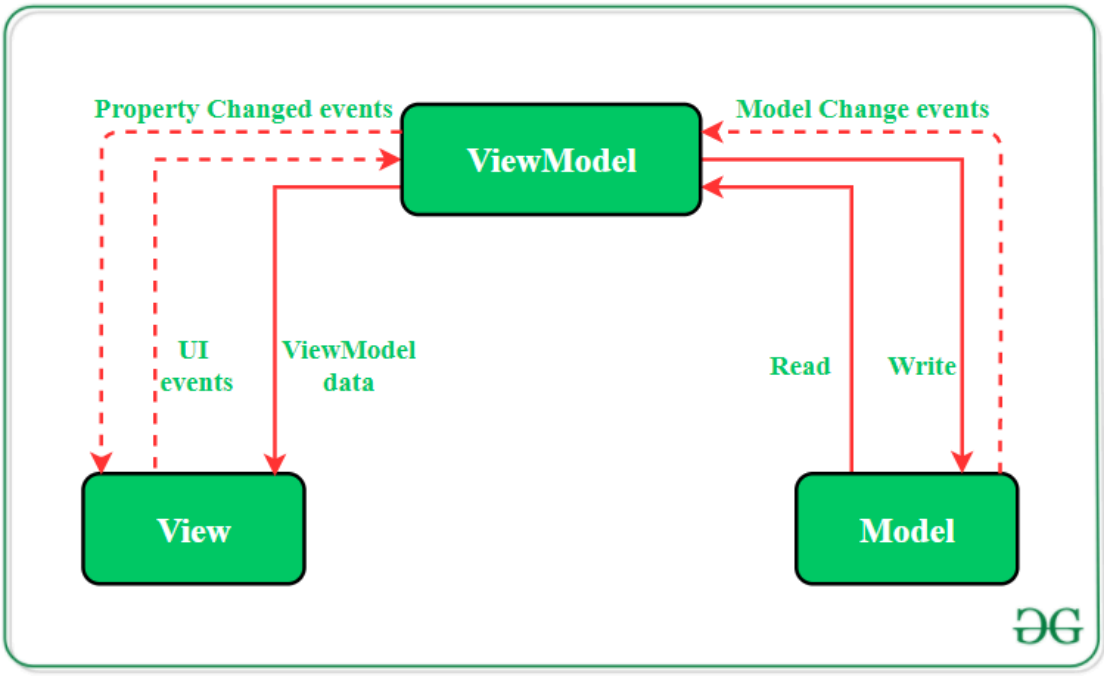


Model View ViewModel (MVVM) Pattern

Model-View-ViewModel (MVVM) is an industry-recognized software architecture pattern that addresses the drawbacks of the MVP and MVC patterns by providing a more structured approach to separating data presentation logic (Views or UI) from the core business logic of an application.

Key Components:

- **Model**:
  - Responsibility: Abstracts data sources and provides data handling capabilities. It works with the ViewModel to retrieve and save data.
  - Example: Database queries, data manipulation, network requests.
- **View**:
  - Responsibility: Represents the user interface and observes the ViewModel for changes. It informs the ViewModel about user actions but contains no application logic.
  - Example: UI components, data bindings.
- **ViewModel**:
  - Responsibility: Acts as a link between the Model and the View, exposing data streams relevant to the View and handling user input logic.
  - Examples: Data transformations, command handling, and state management.



The S.O.L.I.D Principles

The SOLID principles are a set of design principles in object-oriented programming that were introduced to improve the design and architecture of software. These principles were popularized by Robert C. Martin, also known as "Uncle Bob," who is a well-known figure in the software engineering community.

Origins

- **1980s-1990s: Early Influences**
    - The roots of SOLID principles can be traced back to the early works on object-oriented design and programming. Concepts like encapsulation, inheritance, and polymorphism were being explored and formalized.
    - Barbara Liskov introduced the concept of the Liskov Substitution Principle (LSP) in her 1987 conference keynote address, which would later become one of the SOLID principles.
- **1990s: Emergence of Design Patterns**
    - The publication of "Design Patterns: Elements of Reusable Object-Oriented Software" by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (often referred to as the "Gang of Four" or GoF) in 1994 played a significant role in shaping modern software design. This book introduced 23 design patterns that solved common

design problems and influenced the development of principles like those in SOLID.

- 2000: Introduction of SOLID Principles
    - Robert C. Martin introduced the SOLID acronym in the early 2000s. He summarized and named five key principles of object-oriented design that he had been advocating through his books, articles, and lectures.
    - The acronym "SOLID" was coined by Michael Feathers, another influential software engineer, who used it to describe these principles more memorably.
- 2003: "Agile Software Development, Principles, Patterns, and Practices”
    - Robert C. Martin published his book "Agile Software Development, Principles, Patterns, and Practices" in 2003. This book brought widespread attention to the SOLID principles, as Martin discussed them in the context of agile methodologies and their importance in building maintainable and flexible software.

Impact

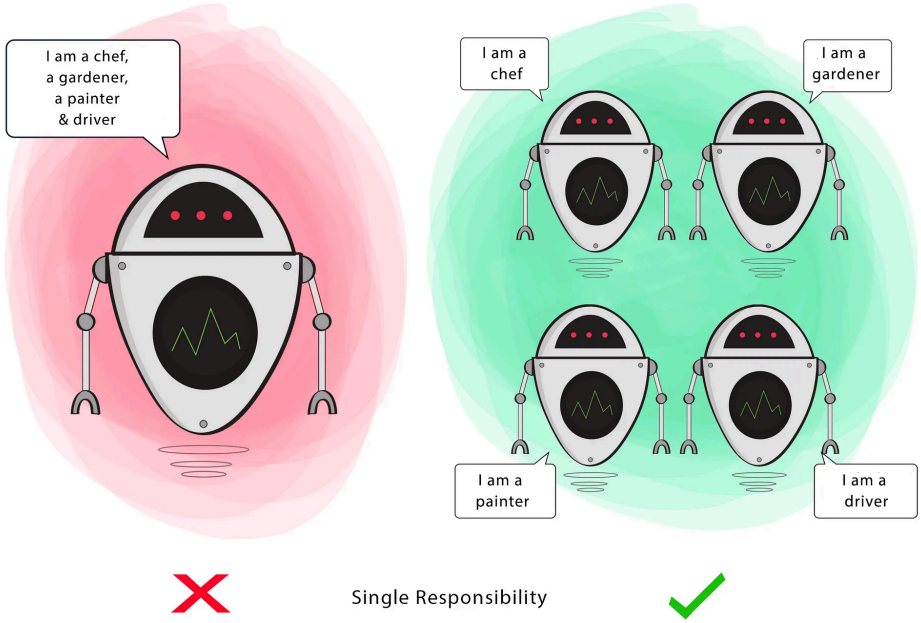
- **Software Design and Architecture**: The SOLID principles have become foundational concepts in software engineering, guiding developers in creating robust, scalable, and maintainable systems.
- **Agile Development**: The principles align well with agile methodologies, promoting flexibility and adaptability in software development.
- **Education and Training**: SOLID principles are widely taught in computer science and software engineering courses, influencing new generations of developers.
- **Industry Adoption**: Many software companies and development teams adopt SOLID principles as part of their best practices, leading to more reliable and maintainable software products.

S — Single Responsibility

The Single Responsibility Principle (SRP) is one of the five SOLID principles of object-oriented design. It states that a class should have only one reason to change, meaning it should have only one job or responsibility.

Explanation

- **Responsibility**: Responsibility is considered a reason for change. If a class has more than one responsibility, it has more than one reason to change. Having multiple reasons to change can make a class harder to understand, maintain, and modify because changes in one responsibility might affect the other responsibilities.
- **Focus**: SRP encourages keeping a class focused on a single task or functionality. This makes the class more cohesive and ensures that it is easier to understand and manage.
- **Modularity**: By adhering to SRP, the software design becomes more modular. Each class handles a specific piece of functionality, which can be developed, tested, and maintained independently.

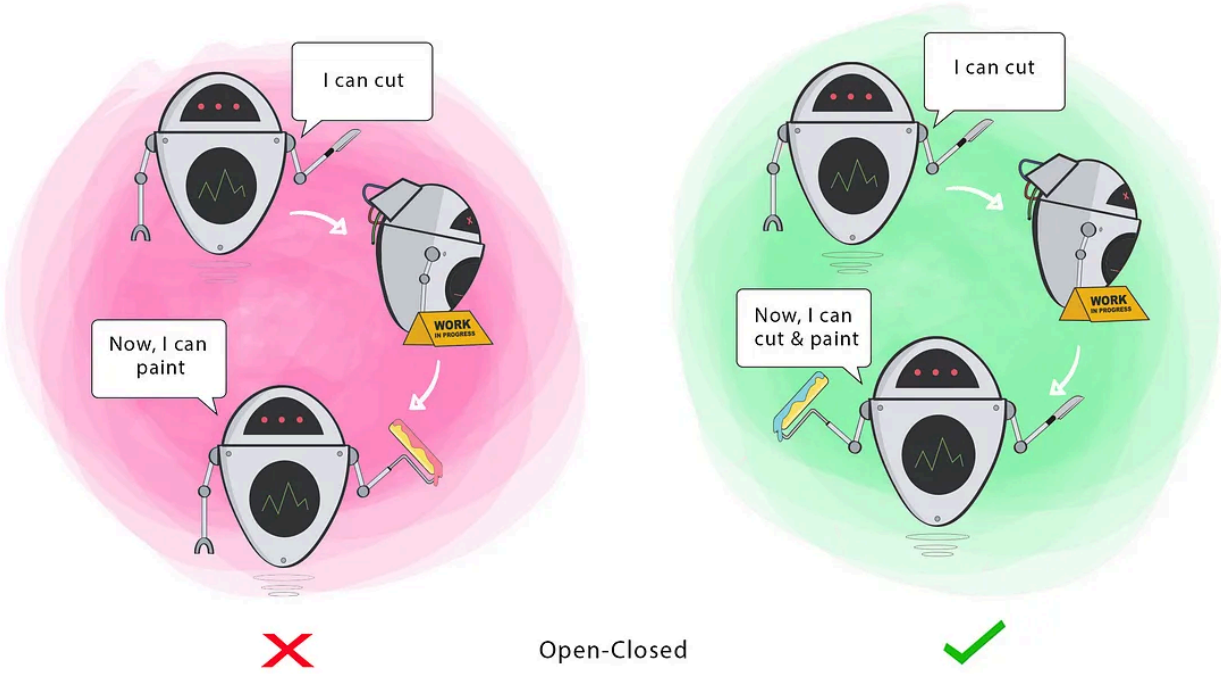


O — Open-Closed

The Open/Closed Principle (OCP) is one of the five SOLID principles of object-oriented design. It states that software entities (such as classes, modules, and functions) should be open for extension but closed for modification.

Explanation

- **Open for Extension**: This means that the behavior of a software module can be extended to accommodate new functionality.
- **Closed for Modification**: This means that the source code of a software module should not be changed to add new functions

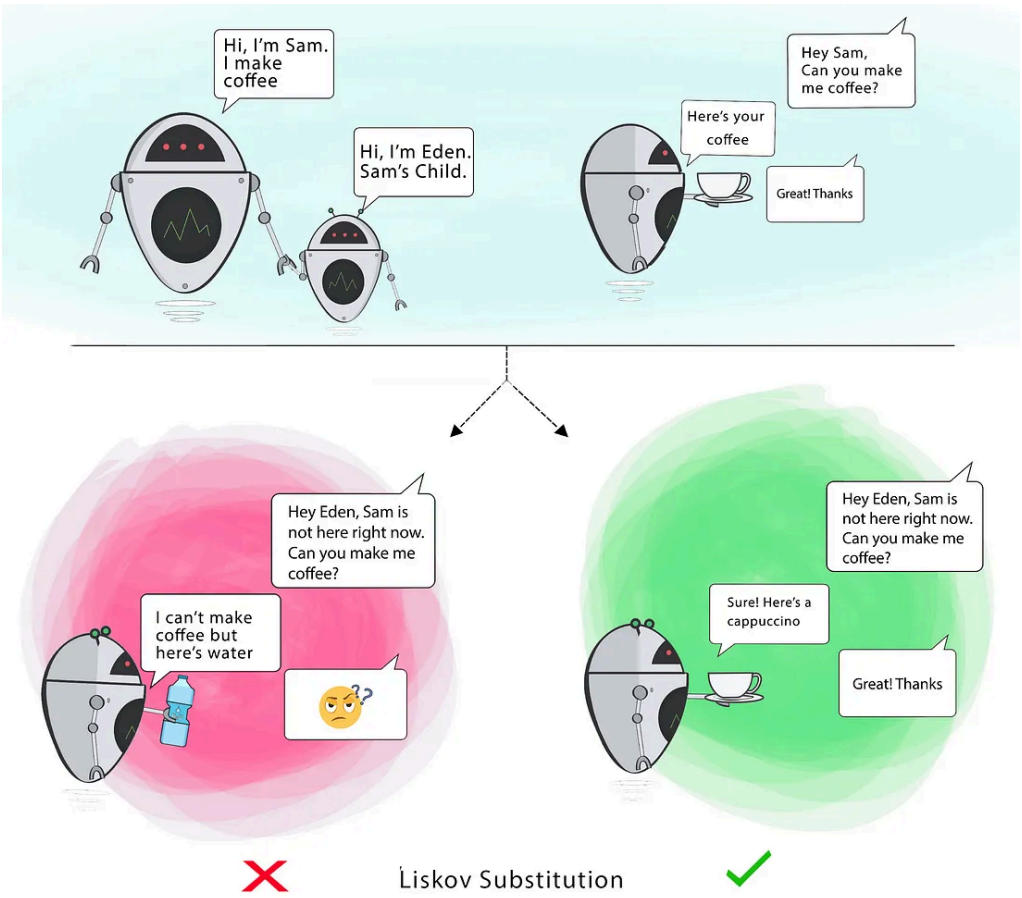


L — Liskov Substitution

The Liskov Substitution Principle (LSP) is one of the five SOLID principles of object-oriented design. It states that objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program.

Explanation

- **Substitutability**: If class S is a subclass of class T, then objects of type T should be able to be replaced with objects of type S without altering the desirable properties of the program (correctness, task performed, etc.).
- **Behavioral Consistency**: Subclasses should behave in a manner consistent with the expectations set by the base class. This means that the subclass should adhere to the contract defined by the base class.

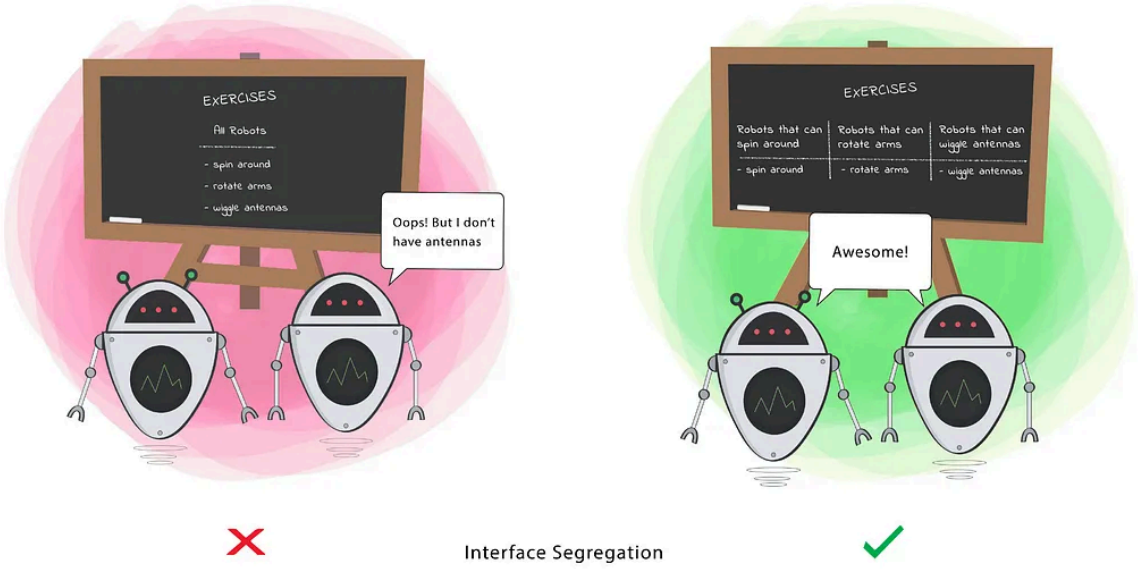


I — Interface Segregation

The Interface Segregation Principle (ISP) is one of the five SOLID principles of object-oriented design. It states that no client should be forced to depend on methods it does not use. This principle promotes the use of smaller, more specific interfaces rather than large, general-purpose ones.

Explanation

- **Client-Specific Interfaces**: Instead of having one large interface with many methods, create multiple smaller, more specific interfaces. This ensures that clients only need to know about the methods that are relevant to them.
- **Decoupling**: Smaller interfaces reduce the coupling between clients and the interfaces they use, making the system more flexible and easier to maintain.
- **Cohesion**: Interfaces with fewer methods are more cohesive, meaning the methods are more closely related to each other, which makes the interface easier to understand and use.

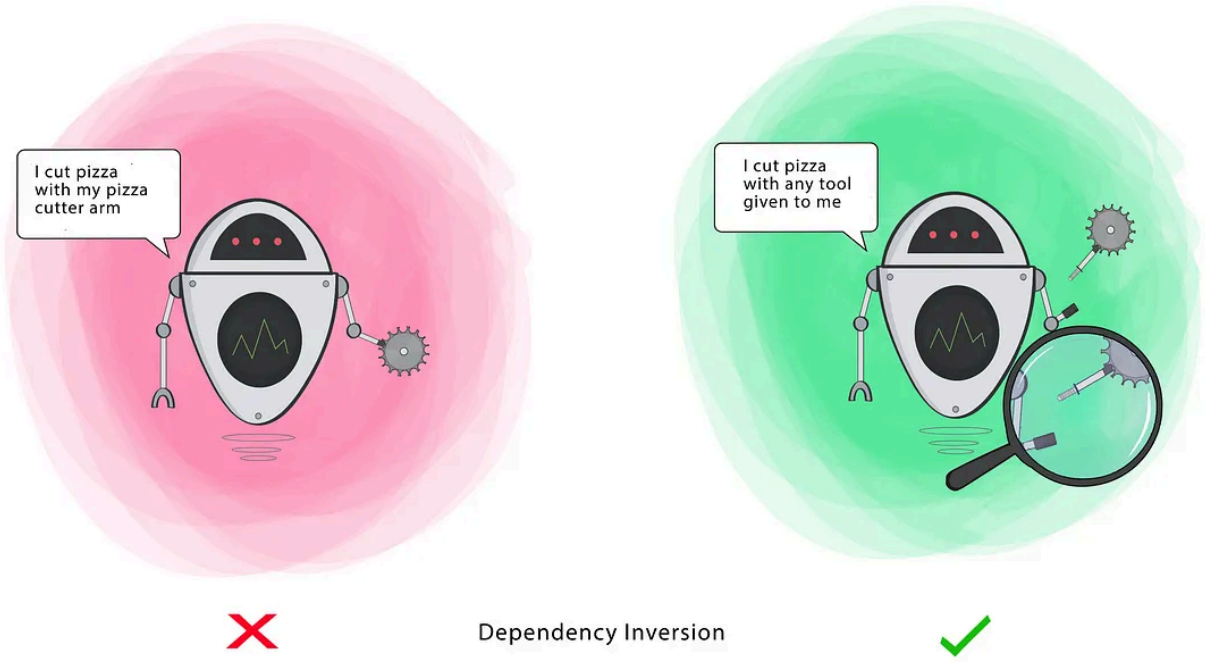


D — Dependency Inversion

The Dependency Inversion Principle (DIP) is one of the five SOLID principles of object-oriented design. It states that high-level modules should not depend on low-level modules. Both should depend on abstractions (e.g., interfaces). Furthermore, abstractions should not depend on details. Details should depend on abstractions.

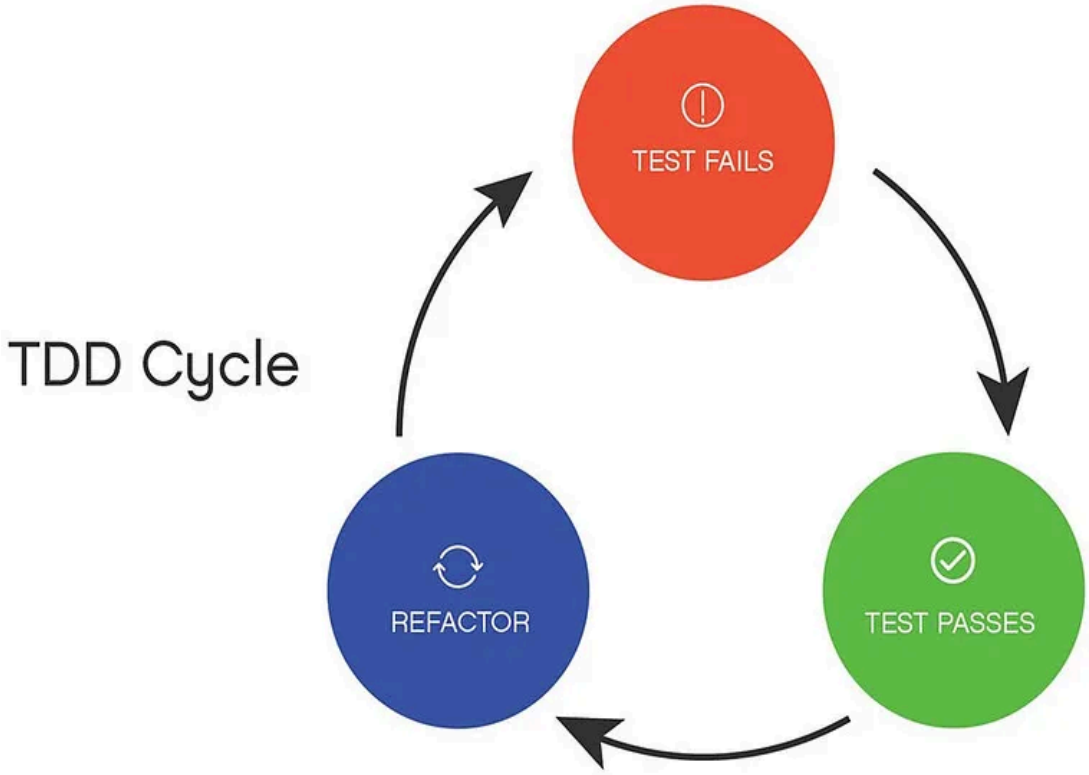
Explanation

- **High-Level Modules**: These are modules that contain complex logic and higher-order policies in the application.
- **Low-Level Modules**: These are modules that deal with more detailed operations such as reading data from databases, sending messages, etc.
- **Abstractions**: These are interfaces or abstract classes that define the contracts that both high-level and low-level modules should follow.



Test-Driven Development (TDD)

Test-driven development (TDD) is a software development approach that emphasizes writing tests before writing the actual code. The core idea behind TDD is to ensure that every piece of code is thoroughly tested, leading to higher code quality and reliability. TDD follows a simple and iterative cycle, often referred to as the “red-green-refactor” cycle.



The TDD Cycle:

- **Red:** In the initial “Red” phase, the developer writes a failing test case that highlights the desired behavior not yet implemented.
- **Green:** In the “Green” phase, the developer writes the minimum amount of code necessary to pass the test.
- **Refactor:** In the “Refactor” phase, the developer improves the code without changing its behavior. This step enhances the design, removes duplication,3, and improves maintainability.

References

- https://refactoring.guru/design-patterns
- https://sourcemaking.com/design_patterns
- https://freedium.cfd/https://python.plainenglish.io/solid-principles-made-simple-a-basic-guide-with-examples-15e5553aecec
- https://medium.com/@mahmoudIbrahimAbbas/software-architecture-patterns-558486f4c3aa
- https://medium.com/front-end-weekly/software-architecture-patterns-that-you-must-know-1b3a4cbab4ca
- https://medium.com/@dees3g/a-guide-to-test-driven-development-tdd-with-real-world-examples-d92f7c801607