Contents

Abstract Factory Design Pattern

Introduction: The Abstract Factory Design Pattern is a creational design pattern that provides an interface for creating families of related or dependent objects without specifying their concrete classes. This pattern allows for the creation of a suite of related products without knowing their specific classes.

Which SOLID Principle Followed by This Pattern: The Abstract Factory Pattern adheres to the Dependency Inversion Principle (DIP), which is part of the SOLID principles. DIP states that high-level modules should not depend on low-level modules. Both should depend on abstractions.

Type of This Pattern: The Abstract Factory Pattern is a Creational pattern. Creational patterns are concerned with the way objects are created, providing mechanisms to make the creation process more adaptable and dynamic.

Components of This Pattern

- **AbstractFactory:** Declares a set of methods for creating abstract product objects.
- **ConcreteFactory:** Implements the methods to create concrete products.
- **AbstractProduct:** Declares an interface for a type of product object.
- **ConcreteProduct:** Implements the AbstractProduct interface.
- **Client:** Uses only the interfaces declared by AbstractFactory and AbstractProduct classes.

What Is This Pattern: The Abstract Factory Pattern provides a way to encapsulate a group of individual factories that have a common theme. It defines an interface for creating families of related or dependent objects without specifying their concrete classes.

Why Do We Need This Pattern: We need this pattern to ensure that a system is independent of how its products are created, composed, and represented. It promotes consistency among products and allows the system to be easily extended to produce new kinds of products.

How Does This Pattern Work

- **Define Abstract Factories:** Create interfaces for families of related products.
- **Implement Concrete Factories:** Implement these interfaces for specific product variants.
- **Client Uses Abstract Factories:** The client interacts with the abstract factory to create objects.

When to Use This Pattern

- When a system should be independent of how its products are created, composed, and represented.
- When a system should be configured with one of multiple families of products.
- When you want to provide a library of products, revealing only their interfaces, not their implementations.

When Not Recommended to Use This Pattern

- When you don't need to manage or create multiple families of products.
- When the variations in the products do not require a consistent set of interfaces.

Layman Example

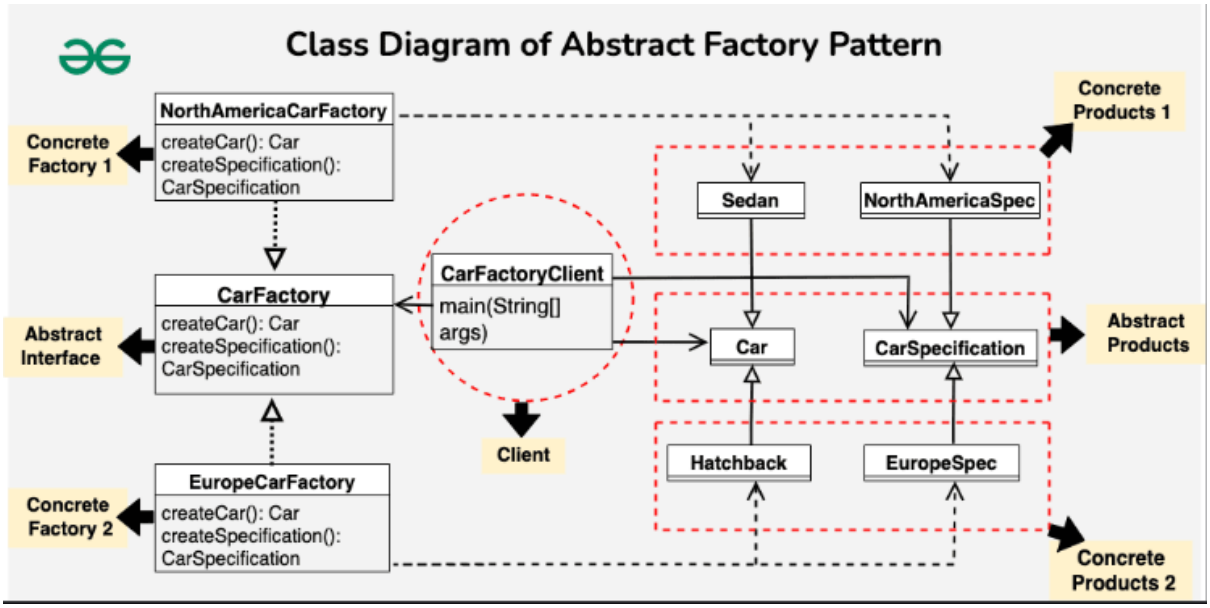
Furniture Store: Imagine a furniture store selling different types of furniture sets like Victorian, Modern, and Art Deco. Each set includes a chair, a sofa, and a coffee table. The Abstract Factory pattern can be used to create a family of related products (furniture set) without specifying their concrete classes.

Vehicle Factory: Think about a vehicle factory that produces different types of vehicles like cars and trucks. Each type of vehicle might need different sets of parts such as engines, tires, and seats. The Abstract Factory can help create a family of related parts for each vehicle type.

Technical Example

Imagine you're managing a global car manufacturing company. You want to design a system to create cars with specific configurations for different regions, such as North America and Europe. Each region may have unique requirements and regulations, and you want to ensure that cars produced for each region meet those standards.

Flow Diagram



- **Client**: Uses the Abstract Factory interface.
- **Abstract Factory**: Interface with methods to create products.
- **Concrete Factories:** Implement the Abstract Factory interface.
- **Abstract Products:** Interfaces for product types.
- **Concrete Products:** Implementations of the product interfaces.

Code: (Visit Below link)

https://www.geeksforgeeks.org/abstract-factory-pattern/

Advantages

- **Encapsulation:** Encapsulates the creation of a set of related products.
- **Consistency:** Ensures that related products are created together.
- **Scalability:** Makes it easy to introduce new variants of products.

Disadvantages

- **Complexity:** Adds more classes and complexity to the codebase.
- **Rigidity:** Can be more difficult to extend a family of products without modifying the factory interfaces.
- **Increased Number of Classes:** As you introduce more abstract factories and product families, the number of classes in your system can grow rapidly. This can make the code harder to manage and understand, particularly for smaller projects.
- **Dependency Inversion Principle Violation**: In some cases, the Abstract Factory pattern may lead to a violation of the Dependency Inversion Principle, especially if client code directly depends on concrete factory implementations rather than the abstract interfaces.
- **Limited Extensibility**: Extending the abstract factory hierarchy or introducing new product families might require modifications to multiple parts of the code, potentially leading to cascading changes and making the system less extensible.
- **Not Ideal for Simple Systems**: The Abstract Factory pattern may be overkill for smaller, less complex systems where the overhead of defining abstract factories and products outweighs the benefits of the pattern.

Use Case

- GUI toolkits where different look-and-feel standards are applied based on the platform (Windows, MacOS, Linux).
- Creating product families in software like automotive manufacturing systems where different models require different sets of parts.

Conclusion: The Abstract Factory Design Pattern is a powerful tool for creating families of related objects without being tied to their concrete implementations. It promotes flexibility and scalability in your codebase, making it easier to maintain and extend. However, it's important to weigh its benefits against the added complexity to determine if it's the right solution for your particular problem.

Builder Design Pattern

Introduction: The Builder Design Pattern is a creational design pattern that allows you to construct complex objects step by step. Unlike other creational patterns, which focus on the instantiation process, the Builder pattern focuses on the construction process.

Which SOLID Principle Followed by This Pattern: The Builder Design Pattern follows the Single Responsibility Principle (SRP). It decouples the construction of a complex object from its representation, allowing different builders to create different representations.

Type of This Pattern: The Builder Pattern is a Creational pattern. Creational patterns deal with object creation mechanisms, trying to create objects in a manner suitable to the situation.

Components of This Pattern

- **Builder:** Declares interface for creating parts of a product.
- **ConcreteBuilder:** Implements the Builder interface and provides specific implementations for the construction steps.
- **Director**: Constructs an object using the Builder interface.
- **Product:** Represents the complex object being built.

What Is This Pattern: The Builder Pattern provides a way to construct a complex object step by step. It allows you to produce different types and representations of an object using the same construction process.

Why Do We Need This Pattern

- Simplify the creation of complex objects.
- Allow the same construction process to create different representations of objects.
- Improve readability and maintainability of the code by isolating the construction logic.

How Does This Pattern Work

- **Define the Builder Interface**: Declare methods for creating parts of a Product object.
- **Implement Concrete Builders:** Provide specific implementations for the Builder interface.
- **Use a Director:** The Director class constructs the Product using the Builder interface.
- **Get the Product:** The Client uses the Director to get the constructed product.

When to Use This Pattern

- When the construction process of an object is complex.
- When the construction process should allow different representations.
- When you want to isolate the construction code from the representation.

When Not Recommended to Use This Pattern

- When the object can be created in a single step.
- When the object doesn’t require multiple configurations or representations.

Layman Example

- **Burger Builder:** Imagine a fast-food restaurant where you can customize your burger with different ingredients. The Builder pattern allows you to build a burger step by step with choices like bun type, meat, vegetables, and condiments.
- **House Builder:** Think about constructing a house. You can choose different types of foundations, walls, roofs, and interiors. The Builder pattern helps in creating a house with different features using the same construction process.

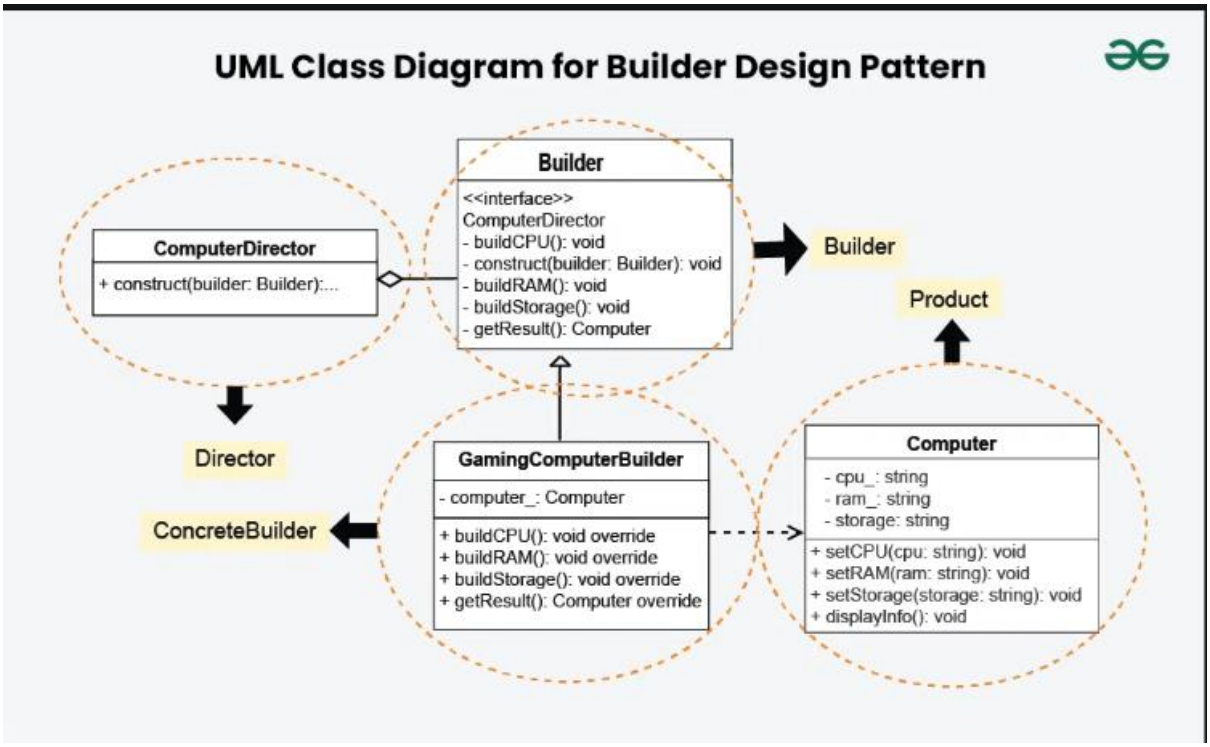
Technical Example

You are tasked with implementing a system for building custom computers. Each computer can have different configurations based on user preferences. The goal is to provide flexibility in creating computers with varying CPUs, RAM, and storage options.

Code

https://www.geeksforgeeks.org/builder-design-pattern/

Flow Diagram



- **Client:** Requests the Director to construct an object.
- **Director:** Uses the Builder interface to construct the object.
- **Builder:** Provides methods to build parts of the product.
- **Concrete Builder**: Implements the Builder methods to construct the product.
- **Product:** The final constructed object.

Advantages

- **Control:** Provides control over the construction process.
- **Isolation:** Isolates complex construction code from the product's representation.
- **Flexibility:** Allows construction of different representations.

Disadvantages

- **Complexity**: Adds more classes and complexity to the codebase.
- **Overhead:** Might be overkill for simple objects.

Use Case

- Constructing complex objects like documents, vehicles, or UI components.
- When an object requires multiple steps or configurations for creation.
- In web apps, the Builder pattern constructs user profiles with optional fields (bio, picture, contact info) for flexible and scalable object creation.

Conclusion

The Builder Design Pattern is a powerful way to construct complex objects by breaking down the construction process into manageable steps. It enhances flexibility and maintainability by isolating the construction logic from the product's representation, making it easier to produce different types and configurations of a product.

Prototype Design Pattern

Introduction: The Prototype Design Pattern is a creational design pattern that allows you to create new objects by copying existing objects, known as prototypes. This pattern is useful when the process of creating a new object is expensive or complex.

Which SOLID Principle Followed by This Pattern: The Prototype Pattern follows the Single Responsibility Principle (SRP). It delegates the cloning process to the prototype objects themselves, thereby keeping the creation logic encapsulated within the object.

Type of This Pattern: The Prototype Pattern is a Creational pattern. Creational patterns deal with the process of object creation, aiming to make it more efficient and adaptable.

Components of This Pattern

- Prototype Interface: Declares the cloning method.
- Concrete Prototype: Implements the cloning method to create a copy of itself.
- Client: Uses the Prototype interface to create a new object by cloning an existing one.

What Is This Pattern: The Prototype Pattern allows you to create new objects by copying existing objects. This pattern involves creating a prototype interface with a clone method, which is implemented by concrete prototype classes. The client can then create new objects by cloning existing prototypes.

Why Do We Need This Pattern

- Avoid the cost of creating objects through the standard means (e.g., constructors).
- Create new objects from existing instances, which can be especially useful when the creation process is resource-intensive.
- Simplify the creation of objects with complex configurations or states.

How Does This Pattern Work

- Define Prototype Interface: Declare a cloning method.
- Implement Concrete Prototypes: Implement the cloning method to return a copy of the object.
- Client Clones Prototypes: The client creates new objects by cloning existing prototype instances.

When to Use This Pattern

- When the process of creating new instances is expensive or complex.
- When you need to create objects that are similar to existing ones.
- When you want to reduce the number of subclasses used for creating complex objects.

When Not Recommended to Use This Pattern

- When creating a new object is simple and inexpensive.
- When each object is unique and does not benefit from cloning.
- When there are simpler alternatives like using a factory pattern.

Layman Example

- Cookie Cutter: Imagine a cookie cutter in the shape of a star. Instead of manually shaping each cookie, you use the cookie cutter to quickly produce multiple cookies with the same shape. The cookie cutter acts as the prototype.

- Rubber Stamp: Think of a rubber stamp with a specific design. Instead of drawing the design repeatedly, you use the stamp to create identical impressions. The stamp is the prototype.

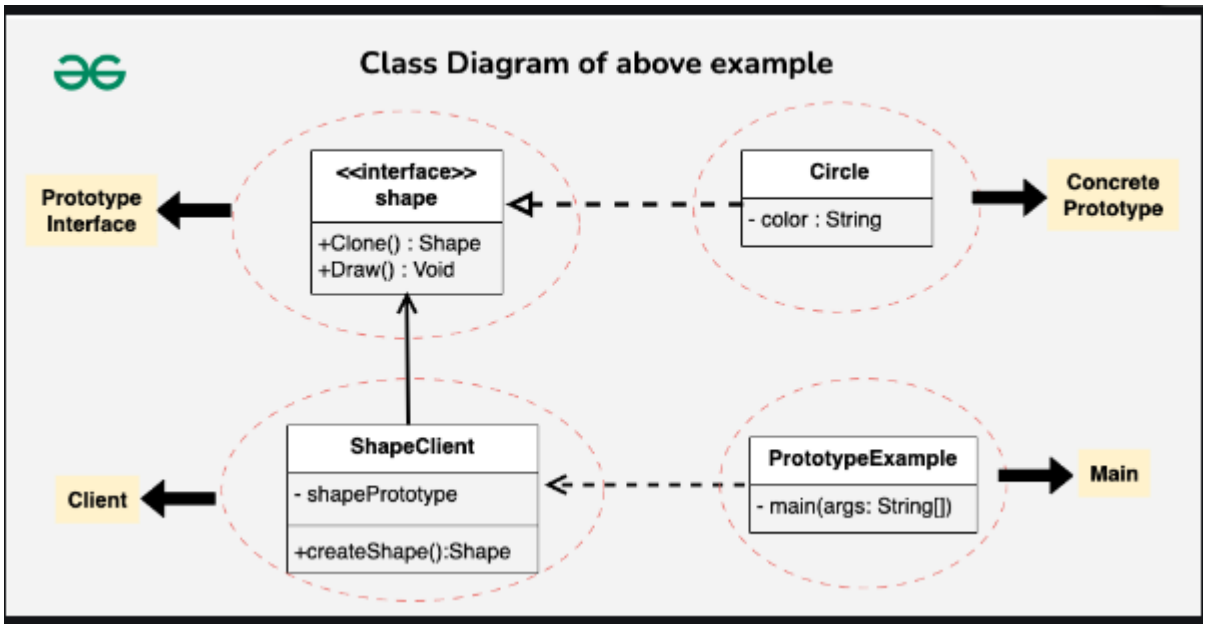
Technical Example:

Imagine you're working on a drawing application, and you need to create and manipulate various shapes. Each shape might have different attributes like color or size. Creating a new shape class for every variation becomes cumbersome. Also, dynamically adding or removing shapes during runtime can be challenging.

Code

https://www.geeksforgeeks.org/prototype-design-pattern/

Flow Diagram



- Client: Requests a clone from the prototype.
- Prototype Interface: Defines the clone method.
- Concrete Prototype: Implements the clone method to return a copy of itself.
- Cloned Object: The resulting new object created by the clone method.

Advantages

- Efficiency: Reduces the cost of creating objects.
- Simplicity: Simplifies the creation of complex objects.
- Flexibility: Allows for dynamic and flexible object creation.

Disadvantages

- Complexity: Requires implementing a cloning method in each prototype class.
- Memory: Cloning can be memory-intensive if objects are large or deeply nested.

Real-Life Scenarios and Use Cases

- Graphic Editors: Creating copies of graphical objects (shapes, lines) with the same properties.
- Game Development: Cloning game characters or objects with similar attributes.
- Document Management: Creating templates for documents and generating new documents by cloning these templates.

Conclusion: The Prototype Design Pattern is a useful creational pattern that enables efficient and flexible object creation by cloning existing objects. It is particularly beneficial when creating new objects is resource-intensive or when objects share common configurations. By leveraging the Prototype pattern, you can improve the performance and maintainability of your application, although it requires careful implementation of the cloning method.

Factory Method Design Pattern

Introduction: The Factory Method Design Pattern 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. This pattern delegates the instantiation of objects to subclasses, promoting flexibility and extensibility.

Which SOLID Principle Followed by This Pattern: The Factory Method Pattern follows the Open/Closed Principle (OCP). This principle states that software entities should be open for extension but closed for modification. By using the Factory Method, new types of products can be introduced without altering existing code.

Type of This Pattern: The Factory Method Pattern is a Creational pattern. Creational patterns deal with object creation mechanisms, trying to create objects in a manner suitable to the situation.

Components of This Pattern

- **Product:** Defines the interface of objects the factory method creates.
- **ConcreteProduct**: Implements the Product interface.
- **Creator:** Declares the factory method that returns an object of type Product. The Creator may also provide a default implementation of the factory method.
- **ConcreteCreator**: Overrides the factory method to return an instance of a ConcreteProduct.

What Is This Pattern: The Factory Method Pattern defines an interface for creating an object, but lets subclasses alter the type of objects that will be created. This allows a class to defer instantiation to its subclasses.

Why Do We Need This Pattern

- Provide a way to delegate the instantiation process to subclasses.
- Promote code flexibility and reusability.
- Avoid tight coupling between the code that needs to instantiate objects and the classes that are instantiated.

How Does This Pattern Work

- Define a Product Interface: Specify the operations that all concrete products must implement.
- Implement Concrete Products: Create concrete classes that implement the Product interface.
- Create a Creator Class: Define the factory method in the creator class, which returns a Product object.
- Override Factory Method: Subclasses of the Creator override the factory method to instantiate specific concrete products.

When to Use This Pattern

- When a class can't anticipate the type of objects it needs to create.
- When a class wants its subclasses to specify the objects it creates.
- To localize the knowledge of which class to instantiate.

When Not Recommended to Use This Pattern

- When the object creation is simple and doesn't require a separate factory method.
- When the application doesn't need flexibility in the instantiation process.

Layman Example

Bakery: Imagine a bakery that makes different types of bread (e.g., baguette, sourdough). The bakery has a general method to make bread, but the specific type of bread is determined by the subclass of the bakery.

Vehicle Factory: Consider a vehicle factory that produces various types of vehicles (e.g., cars, bikes). The factory has a method to manufacture a vehicle, but the specific type of vehicle is decided by the subclass.

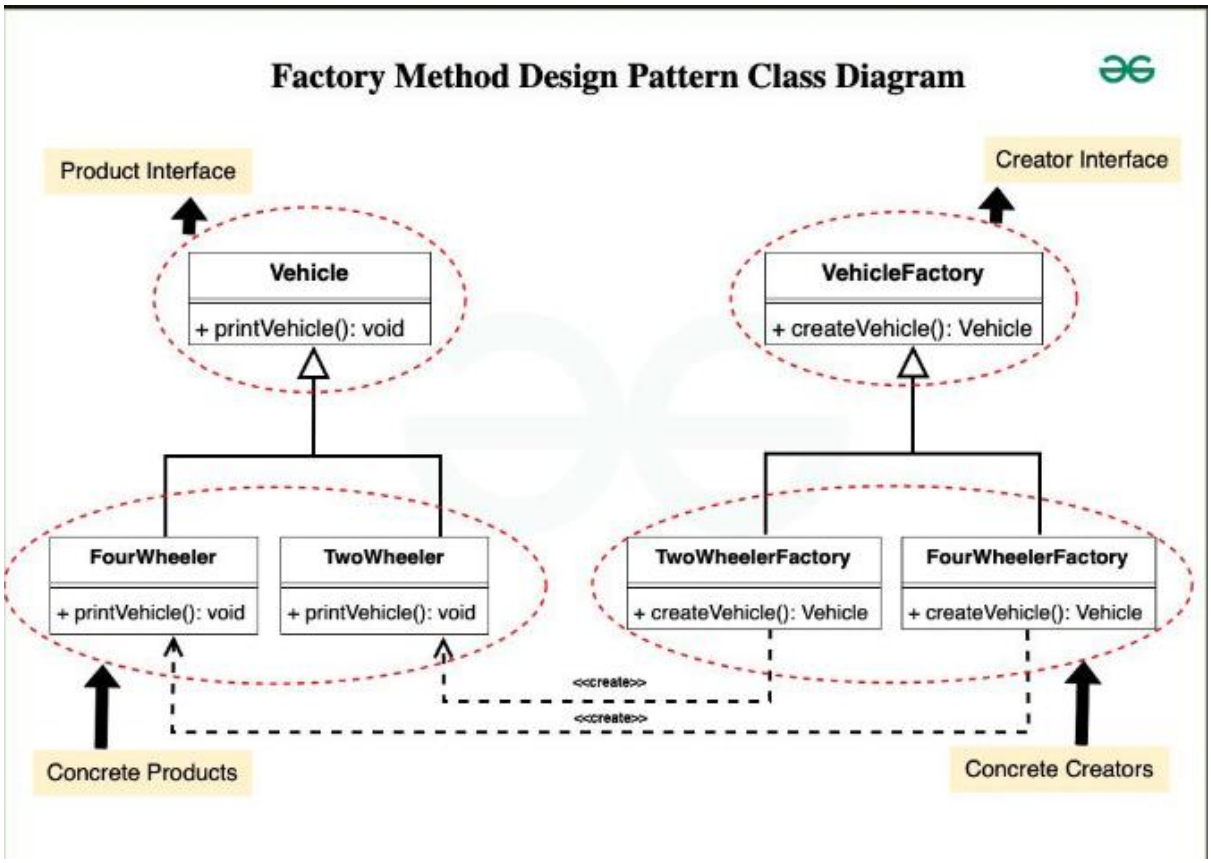
Technical Example

Consider a software application that needs to handle the creation of various types of vehicles, such as Two Wheelers, Three Wheelers, and Four Wheelers. Each type of vehicle has its own specific properties and behaviors.

Code

https://www.geeksforgeeks.org/factory-method-for-designing-pattern/

Flow Diagram



- **Client:** Requests a product creation from the Creator.
- **Creator:** Declares the factory method and may provide a default implementation.
- **Concrete Creator:** Implements the factory method to create specific Concrete Products.
- **Product:** The interface of objects created by the factory method.
- **Concrete Product**: Specific implementations of the Product interface.

Advantages

- **Flexibility:** Promotes the addition of new product types without modifying existing code.
- **Reusability:** Encourages code reuse by defining common product interfaces.
- **Encapsulation:** Keeps the instantiation code separate from the business logic.

Disadvantages

- **Complexity:** Adds extra classes and can make the codebase more complex.
- **Overhead:** May introduce unnecessary overhead if not used judiciously.

Real-Life Scenarios and Use Cases

- **GUI Frameworks:** Creating different types of buttons, windows, and dialogs for different operating systems.
- **Document Processing:** Generating various types of documents (PDF, Word, HTML) using a common interface.
- **Logging Frameworks**: Supporting multiple logging mechanisms (e.g., file, console, network) using the same interface.

Conclusion: The Factory Method Design Pattern is a powerful creational pattern that provides flexibility and extensibility in object creation. By delegating the instantiation process to subclasses, it adheres to the Open/Closed Principle, making it easier to introduce new product types without modifying existing code. While it adds complexity, the benefits of flexibility and reusability often outweigh the costs, especially in large and complex systems.

Facade Design Pattern

Introduction: The Facade Design Pattern is a structural design pattern that provides a simplified interface to a complex system of classes, libraries, or APIs. It acts as a unified interface to a set of interfaces in a subsystem, making it easier to use.

Which SOLID Principle Followed by This Pattern: The Facade Pattern often aligns with the Single Responsibility Principle (SRP). It encourages separating client code from complex subsystems, promoting a clean separation of concerns.

Type of This Pattern: The Facade Pattern is a Structural pattern. Structural patterns focus on the composition of classes or objects, simplifying the structure of a system.

Components of This Pattern

- **Facade:** Provides a simple interface to the complex subsystem. It delegates client requests to appropriate objects in the subsystem.
- **Subsystem Classes**: Classes that implement the functionality of the subsystem. These classes are hidden from the client by the facade.

What Is This Pattern: The Facade Pattern provides a unified interface to a set of interfaces in a subsystem. It defines a higher-level interface that makes the subsystem easier to use, by providing a single entry point to its functionality.

Why Do We Need This Pattern

- Simplify the usage of a complex system by providing a high-level interface.
- Encapsulate the complexity of a subsystem, hiding its implementation details from clients.
- Promote loose coupling between clients and the subsystem, making the system easier to maintain and modify.

How Does This Pattern Work

- **Create a Facade Class**: Define a facade class that provides a simplified interface to the subsystem.
- **Define Subsystem Classes**: Implement classes that represent the functionality of the subsystem.
- **Delegate Client Requests:** The facade class delegates client requests to appropriate objects in the subsystem.
- **Client Uses Facade:** Clients interact with the facade class without needing to know about the subsystem's implementation details.

When to Use This Pattern

- When you need to provide a simple interface to a complex subsystem.
- When you want to encapsulate the complexity of a subsystem, hiding its implementation details.
- When you want to promote loose coupling between clients and the subsystem.

When Not Recommended to Use This Pattern

- When the system is simple and does not have a complex structure.
- When the overhead of creating a facade outweighs the benefits.

Layman Example

- Home Theater System: Imagine you have a home theater system with multiple devices like a TV, DVD player, and sound system. Instead of managing each device separately, you have a remote control that serves as a facade, allowing you to perform actions like "watch movie" or "listen to music" with a single button press.
- Online Shopping: Consider an online shopping website where you can place orders, check order status, and manage your account. Behind the scenes, there are several subsystems handling inventory management, order processing, and customer accounts. The website acts as a facade, providing a simple interface for users to interact with the complex backend systems.

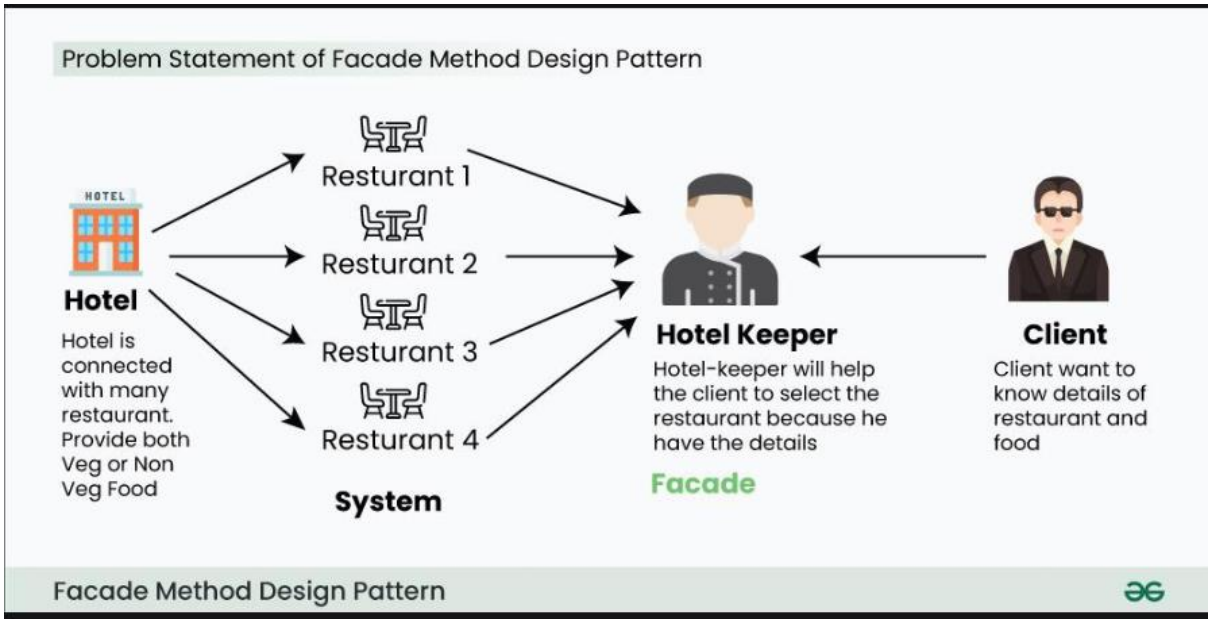
Technical Example

Let's consider a hotel. This hotel has a hotel keeper. There are a lot of restaurants inside the hotel e.g. Veg restaurants, Non-Veg restaurants, and Veg/Non Both restaurants. You, as a client want access to different menus of different restaurants. You do not know what are the different menus they have. You just have access to a hotel keeper who knows his hotel well. Whichever menu you want, you tell the hotel keeper and he takes it out of the respective restaurants and hands it over to you.

Code

https://www.geeksforgeeks.org/facade-design-pattern-introduction/

Flow Diagram



- **Client:** Interacts with the facade to perform operations.
- **Facade:** Provides a simplified interface to the subsystem.
- **Subsystem Classes**: Implement the functionality of the subsystem.
- **Interaction:** The facade delegates client requests to appropriate subsystem classes.

Advantages

- **Simplification:** Simplifies the usage of a complex system by providing a unified interface.
- **Encapsulation:** Hides the complexity of a subsystem from clients, promoting loose coupling.
- **Maintenance:** Makes it easier to maintain and modify the system by encapsulating changes within the facade.

Disadvantages

- **Limited Functionality**: The facade may not expose all the functionality of the subsystem, limiting flexibility.
- **Abstraction Overhead**: Introducing a facade adds an additional layer of abstraction, which may introduce overhead.

Real-Life Scenarios and Use Cases

- **Operating Systems:** The shell acts as a facade to interact with the underlying system resources like file management and process execution.
- **APIs**: APIs provide a facade for developers to interact with complex systems or services like cloud platforms or payment gateways.

Conclusion: The Facade Design Pattern is a valuable tool for simplifying the usage of complex systems by providing a unified interface. By encapsulating the complexity of a subsystem, it promotes loose coupling and enhances maintainability. While it adds an additional layer of abstraction, the benefits of simplification and encapsulation often outweigh the drawbacks, especially in large and complex systems.

Adapter Design Pattern

Introduction: The Adapter Design Pattern is a structural design pattern that allows incompatible interfaces to work together. It acts as a bridge between two incompatible interfaces by converting the interface of a class into another interface that a client expects.

Which SOLID Principle Followed by This Pattern: The Adapter Pattern often aligns with the Interface Segregation Principle (ISP). It allows clients to use specific interfaces without being affected by the complexities of the underlying classes.

Type of This Pattern: The Adapter Pattern is a Structural pattern. Structural patterns focus on the composition of classes or objects, simplifying the structure of a system.

Components of This Pattern

- **Target Interface:** Defines the interface expected by the client.
- **Adaptee:** Represents the existing interface that needs to be adapted.
- **Adapter:** Implements the Target interface and wraps the Adaptee, translating requests from the client into a format that the Adaptee can understand.
- **Client:** Consumes the Target interface, unaware of the Adapter's presence.

What Is This Pattern: The Adapter Pattern allows objects with incompatible interfaces to work together. It involves creating a middleman (the Adapter) that converts the interface of a class into another interface that a client expects.

Why Do We Need This Pattern

- Enable the integration of existing classes with incompatible interfaces into new systems.
- Facilitate code reusability by allowing existing classes to be reused in new contexts.
- Promote interoperability between different components of a system.

How Does This Pattern Work

- **Define Target Interface**: Create an interface that the client expects.
- **Implement Adaptee:** Define existing classes with incompatible interfaces (the Adaptee).
- **Create Adapter:** Implement a class that implements the Target interface and wraps the Adaptee, translating requests from the client into a format that the Adaptee can understand.
- **Client Uses Adapter:** The client interacts with the Target interface provided by the Adapter, unaware of the Adapter's presence.

When to Use This Pattern

- When you need to integrate existing classes with incompatible interfaces into new systems.
- When you want to reuse existing classes in new contexts without modifying their source code.
- When you need to provide a unified interface to multiple classes with different interfaces.

When Not Recommended to Use This Pattern

- When the Adaptee interface is simple and can be directly implemented by the client.
- When the overhead of introducing an adapter outweighs the benefits of using it.

Layman Example

- **Electrical Adapter:** Imagine you have a device with a European plug, but you need to use it in a country with a different type of socket. You use an electrical adapter that converts the European plug into the appropriate type, allowing you to use the device.
- **Language Translator:** Consider a scenario where you need to communicate with someone who speaks a different language. You use a language translator who understands both languages and acts as an adapter, translating your words into the language the other person understands.

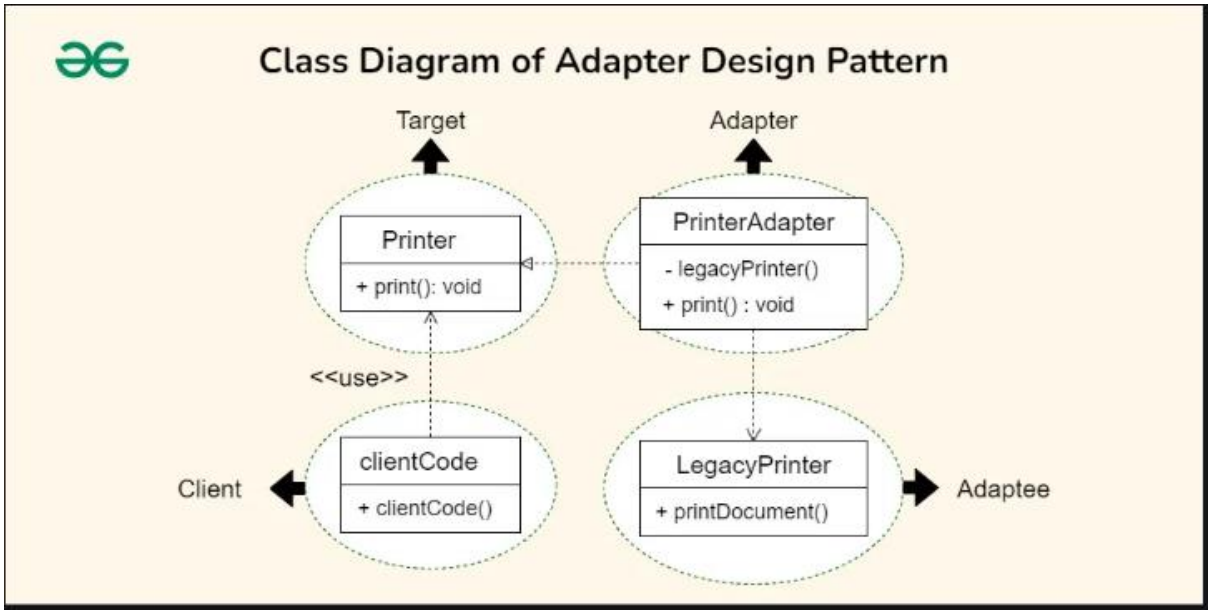
Technical Example

Let's consider a scenario where we have an existing system that uses a LegacyPrinter class with a method named printDocument() which we want to adapt into a new system that expects a Printer interface with a method named print(). We'll use the Adapter design pattern to make these two interfaces compatible.

Code

https://www.geeksforgeeks.org/adapter-pattern/

Flow Diagram



- **Client:** Requests a service through the Target interface.
- **Adapter:** Implements the Target interface and wraps the Adaptee.
- **Adaptee:** Provides the functionality needed by the Adapter, but with an incompatible interface.

Advantages

- **Flexibility:** Allows integration of classes with incompatible interfaces.
- **Reusability:** Enables reuse of existing classes in new contexts without modifying their source code.
- **Interoperability:** Promotes interoperability between different components of a system.

Disadvantages

- **Complexity**: Introducing an additional layer (the Adapter) can add complexity to the system.
- **Overhead:** There may be overhead associated with the translation between interfaces.

Real-Life Scenarios and Use Cases

- **Library Integration**: Integrating legacy libraries with outdated interfaces into modern applications.
- **Hardware Compatibility**: Using adapters to connect devices with different interfaces (e.g., USB to HDMI adapter).
- **API Wrappers:** Creating adapters to translate requests between different versions of an API.

Conclusion: The Adapter Design Pattern is a useful tool for integrating classes with incompatible interfaces into new systems. By providing a bridge between different interfaces, it enables code reuse and promotes interoperability. While it adds an additional layer of complexity, the benefits of flexibility and reusability often outweigh the drawbacks, especially in scenarios where integration is necessary.

Composite Design Pattern

Introduction: The Composite Design Pattern is a structural design pattern that allows you to compose objects into tree-like structures to represent part-whole hierarchies. This pattern enables clients to treat individual objects and compositions of objects uniformly.

Which SOLID Principle Followed by This Pattern: The Composite Pattern adheres to the Single Responsibility Principle (SRP) by ensuring that composite objects manage their children and the Liskov Substitution Principle (LSP) by allowing clients to interact with individual objects and compositions in the same way.

Type of This Pattern: The Composite Pattern is a Structural pattern. Structural patterns deal with object composition and relationships, facilitating the creation of complex structures.

Components of This Pattern

- Component: The base interface or abstract class for all objects in the composition, declaring methods for managing child components.
- Leaf: Represents individual objects in the composition that do not have any children.
- Composite: Represents objects that have children and implement methods to manage these children.
- Client: Interacts with objects through the Component interface, treating leaf and composite objects uniformly.

What Is This Pattern: The Composite Pattern allows you to build complex structures by composing objects into tree structures. Both individual objects (leaves) and groups of objects (composites) are treated uniformly through a common interface.

Why Do We Need This Pattern

- Simplify client code by allowing it to treat individual objects and compositions of objects uniformly.
- Facilitate the creation of complex hierarchical structures.
- Provide flexibility in adding or removing components dynamically.

How Does This Pattern Work

- Define Component Interface: Create an interface or abstract class that defines operations applicable to both leaf and composite objects.
- Implement Leaf Class: Create a class that represents individual objects, implementing the component interface.
- Implement Composite Class: Create a class that represents composite objects, managing child components and implementing the component interface.
- Client Uses Component Interface: The client interacts with objects through the component interface, treating both leaf and composite objects uniformly.

When to Use This Pattern

- When you need to represent part-whole hierarchies of objects.
- When you want to allow clients to treat individual objects and compositions uniformly.
- When you need to create a structure that can be dynamically composed of objects.

When Not Recommended to Use This Pattern

- When the structure of objects is simple and does not involve hierarchical relationships.
- When the overhead of managing the composite structure is not justified.

Layman Example

File System: Imagine a file system where you have files and folders. Files are individual objects (leaves) while folders can contain multiple files or other folders (composites). The Composite Pattern allows you to perform operations like "open" or "delete" on both files and folders uniformly.

Organization Structure: Consider an organization where there are employees and departments. Employees are individual objects (leaves) while departments can contain multiple employees or other departments (composites). The Composite Pattern enables you to manage employees and departments uniformly, such as calculating total salaries.

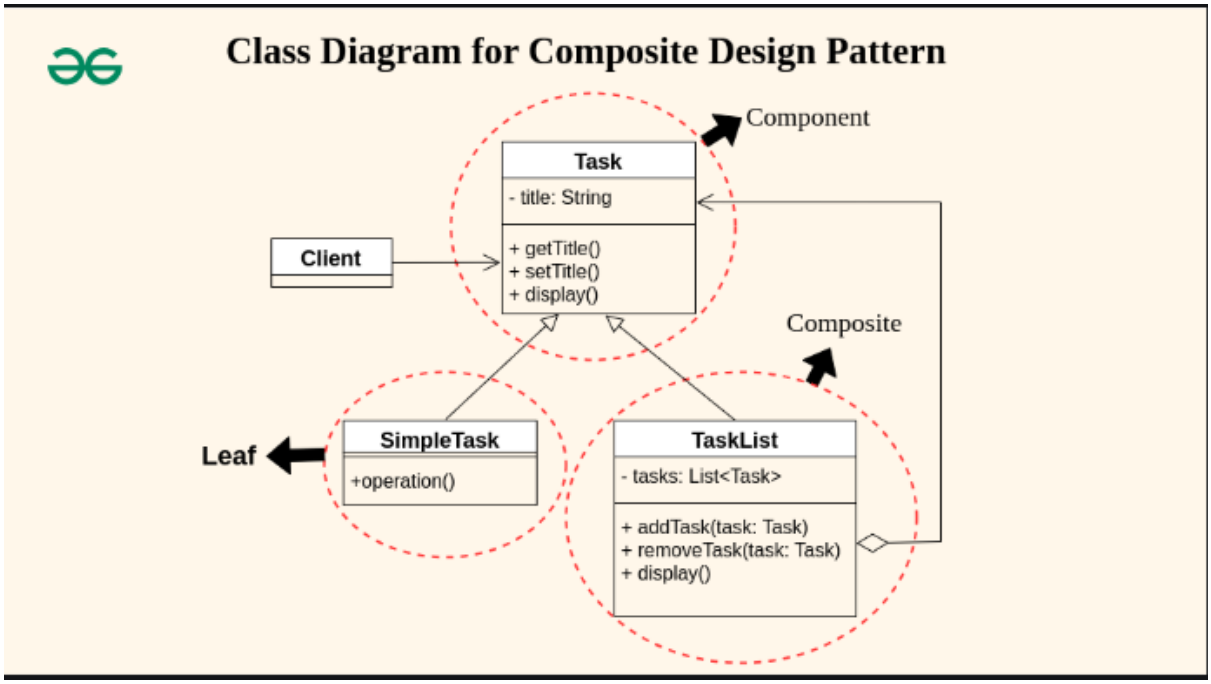
Technical Example

Imagine you are building a project management system where tasks can be either simple tasks or a collection of tasks (subtasks) forming a larger task.

Code

https://www.geeksforgeeks.org/composite-design-pattern-in-java/

Flow Diagram



- Component: The base interface for leaf and composite objects.
- Leaf: Implements the component interface for individual objects.
- Composite: Implements the component interface for composite objects, managing children.
- Client: Interacts with the component interface, treating both leaves and composites uniformly.

Advantages

- Uniformity: Allows clients to treat individual objects and compositions uniformly.
- Flexibility: Facilitates dynamic composition of objects into complex structures.
- Simplicity: Simplifies client code by reducing the need for complex conditionals.

Disadvantages

- Complexity: Introduces complexity in managing the composite structure.
- Overhead: May incur performance overhead due to additional abstraction.

Real Life Scenarios Use Cases

- GUI Libraries: Representing GUI components where windows contain panels, buttons, text fields, etc.
- Document Management Systems: Managing documents where sections contain subsections, paragraphs, and sentences.

Conclusion: The Composite Design Pattern is a powerful tool for managing part-whole hierarchies in software design. By enabling uniform treatment of individual objects and compositions, it simplifies client code and promotes flexibility. While it introduces some complexity and overhead, its benefits in terms of maintainability and scalability make it a valuable pattern in many contexts.

Decorator Design Pattern

Introduction: The Decorator Design Pattern is a structural design pattern that allows behavior to be added to individual objects, dynamically, without affecting the behavior of other objects from the same class. It provides a flexible alternative to subclassing for extending functionality.

Which SOLID Principle Followed by This Pattern: The Decorator Pattern adheres to the Open/Closed Principle (OCP), which states that software entities should be open for extension but closed for modification. This pattern allows for the addition of new functionalities without altering existing code.

Type of This Pattern: The Decorator Pattern is a Structural pattern. Structural patterns deal with object composition and the arrangement of classes or objects to form larger structures.

Components of This Pattern

- Component: An interface or abstract class defining the operations that can be dynamically augmented.
- Concrete Component: A class that implements the component interface and represents the core functionality.
- Decorator: An abstract class that implements the component interface and contains a reference to a component object.
- Concrete Decorator: A class that extends the decorator and adds functionality to the component.

What Is This Pattern: The Decorator Pattern allows for the dynamic addition of behavior to objects by wrapping them in an object of a decorator class. This wrapping can be done multiple times, providing a flexible way to enhance object behavior.

Why Do We Need This Pattern

- Add functionality to objects without modifying their code.
- Combine multiple behaviors flexibly and dynamically.
- Avoid the complexity and rigidity of subclassing.

How Does This Pattern Work

- Define Component Interface: Create an interface or abstract class that declares operations.
- Implement Concrete Component: Create a class that implements the component interface, representing the core functionality.
- Create Decorator Class: Create an abstract decorator class that also implements the component interface and has a reference to a component object.
- Implement Concrete Decorators: Create concrete decorator classes that extend the decorator class and add functionality to the component.

When to Use This Pattern

- When you need to add responsibilities to individual objects dynamically and transparently.
- When you need to add responsibilities that can be withdrawn.
- When extension by subclassing is impractical due to a large number of extensions or when extensions should be applied selectively.

When Not Recommended to Use This Pattern

- When there are simpler ways to add functionality, such as straightforward subclassing.
- When the added complexity of managing many decorator classes is not justified.

Layman Example

- Coffee Shop: Imagine a coffee shop where you can order a basic coffee and then add various extras like milk, sugar, and whipped cream. Each extra can be added independently, and you can combine them in any way you like. The coffee is the component, and the extras are decorators.
- Christmas Tree: Think of decorating a Christmas tree. You start with a plain tree and add lights, tinsel, ornaments, and a star. Each decoration can be added or removed independently, enhancing the tree's appearance. The tree is the component, and the decorations are decorators.

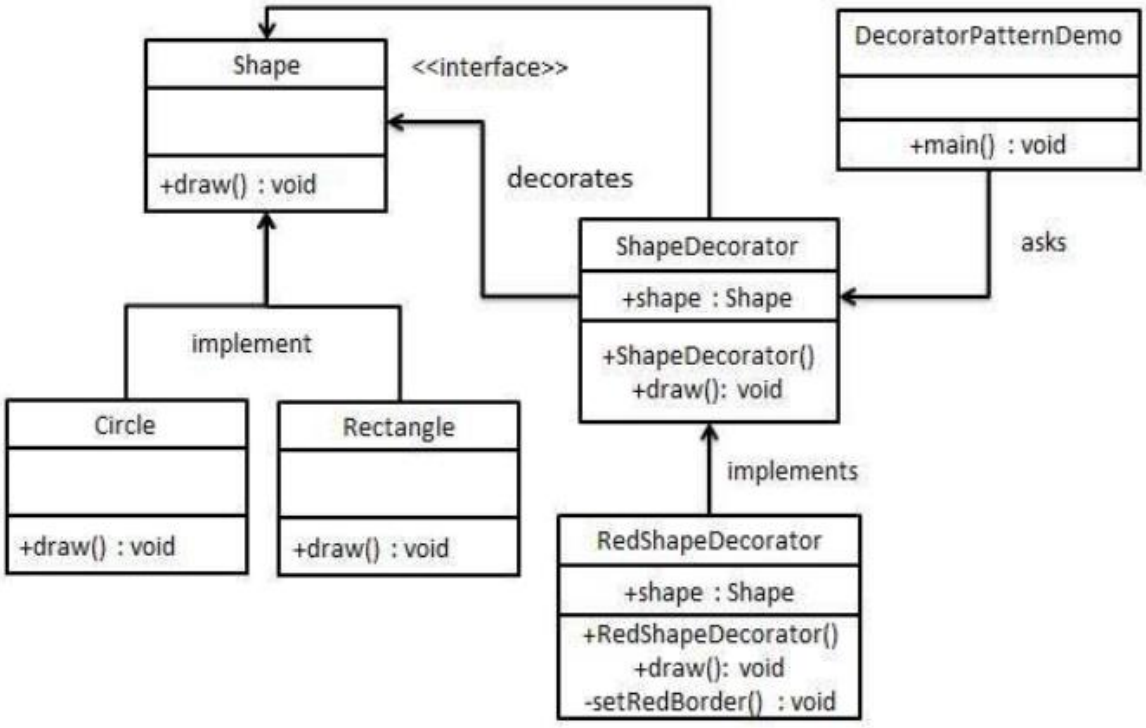
Technical Example

We're going to create a Shape interface and concrete classes implementing the Shape interface. We will then create an abstract decorator class ShapeDecorator implementing the Shape interface and having Shape object as its instance variable.

Code

https://www.tutorialspoint.com/design_pattern/decorator_pattern.htm

Flow Diagram



- Component: The interface or abstract class for the core functionality.
- Concrete Component: Implements the core functionality.
- Decorator: Implements the component interface and wraps a component object.
- Concrete Decorator: Extends the decorator to add specific functionality.

Advantages

- Flexibility: Allows for dynamic and flexible addition of behavior to objects.
- Reusability: Encourages composition over inheritance, promoting code reuse.
- Single Responsibility: Each decorator class has a single responsibility, making the system more modular and easier to understand.

Disadvantages

- Complexity: Can lead to a large number of small classes that are difficult to manage.
- Performance: May introduce performance overhead due to the multiple layers of wrapping.

Real-Life Scenarios Use Cases

- Graphical User Interfaces: Adding features to GUI components like borders, scroll bars, and shadows.
- File I/O: Java's I/O classes use decorators to add functionalities like buffering, data conversion, and encryption to file streams.

Conclusion: The Decorator Design Pattern provides a flexible way to add behavior to objects without altering their code. By using composition over inheritance, it allows for the dynamic extension of functionalities, promoting modularity and reusability. While it can introduce complexity and performance overhead, its advantages in terms of flexibility and adherence to the Open/Closed Principle make it a valuable pattern in many software design contexts.

Interpreter Design Pattern

Introduction: The Interpreter Design Pattern is a behavioral design pattern that provides a way to evaluate sentences in a language. It is used to define a grammatical representation for a language and an interpreter to interpret sentences in that language.

Which SOLID Principle Followed by This Pattern: The Interpreter Pattern follows the Single Responsibility Principle (SRP) by defining a class that interprets a specific type of sentence, thereby maintaining a single responsibility. It also adheres to the Open/Closed Principle (OCP) by allowing new grammar rules to be added without modifying existing code.

Type of This Pattern: The Interpreter Pattern is a Behavioral pattern. Behavioral patterns focus on how objects interact and communicate with each other.

Components of This Pattern

- Abstract Expression: Declares an abstract interpret method.
- Terminal Expression: Implements the interpret method for terminal symbols in the grammar.
- Non-Terminal Expression: Implements the interpret method for non-terminal symbols in the grammar.
- Context: Contains information that's global to the interpreter.

What Is This Pattern: The Interpreter Pattern defines a grammatical representation for a language and provides an interpreter to deal with this grammar. It is used to interpret sentences or expressions written in a particular language.

Why Do We Need This Pattern

- Define a clear and extensible grammar for a language.
- Interpret expressions within the language in a consistent manner.
- Simplify the parsing and evaluation of expressions.

How Does This Pattern Work

- Define Grammar: Create classes for each grammar rule, implementing the interpret method.
- Context: Pass context information to the interpret method.
- Interpret Expressions: Use the interpreter classes to interpret expressions in the language.

When to Use This Pattern

- When you have a simple grammar and need to interpret expressions.
- When you need to define a language and provide an interpreter for it.
- When you can represent the grammar as a class hierarchy.

When Not Recommended to Use This Pattern

- When the grammar is too complex or changes frequently.
- When performance is a critical concern, as the pattern may introduce inefficiencies.

Layman Example

- Mathematical Expressions: Imagine a simple calculator that can evaluate expressions like "3 + 5". The interpreter pattern can be used to parse and evaluate this expression.
- Drawing Commands: Consider a drawing program where commands like "draw circle" or "draw square" are interpreted and executed. The interpreter pattern helps in translating these commands into actual drawing operations.

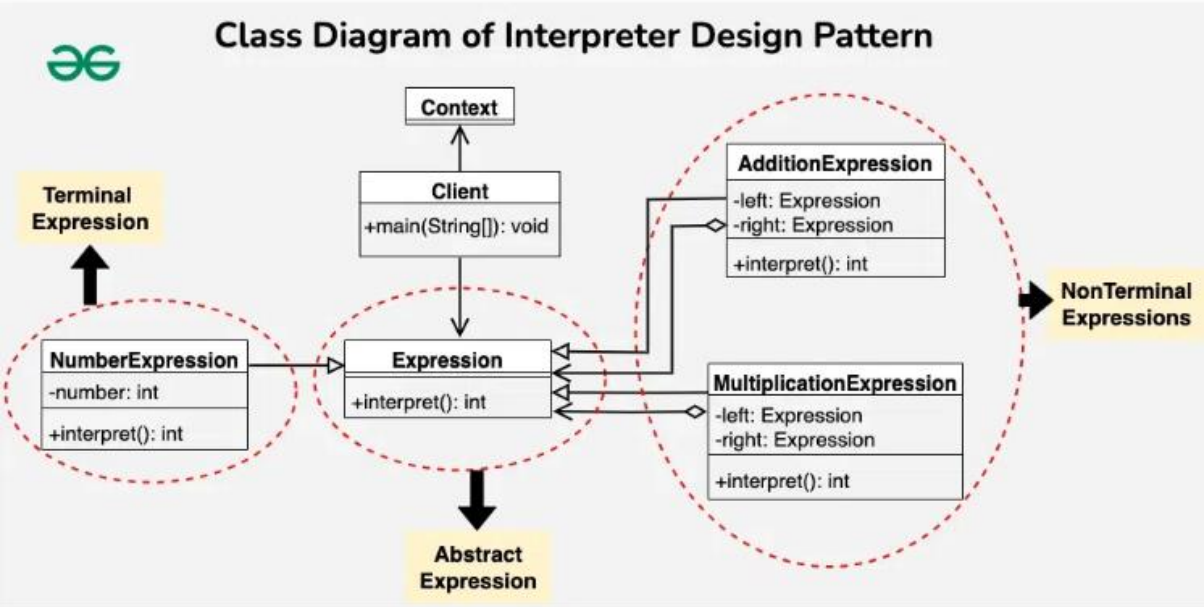
Technical Example

Suppose we have a simple language that supports basic arithmetic operations, such as addition (+), subtraction (-), multiplication (*), and division (/). We want to create a calculator program that can interpret and evaluate arithmetic expressions written in this language.

Code

https://www.geeksforgeeks.org/interpreter-design-pattern/

Flow Diagram



- Context: Holds global information for interpretation.
- Abstract Expression: Interface for interpreting.
- Terminal Expression: Represents leaf nodes.
- Non-Terminal Expression: Represents composite nodes.

Advantages

- Extensibility: Easy to extend the grammar by adding new expressions.
- Clear Grammar Representation: Provides a clear and structured representation of the grammar.
- Reusability: Reusable grammar rules.

Disadvantages

- Complexity: Can become complex for large grammars.
- Performance: Interpretation may be slow for complex expressions.

Real Life Scenarios Use Cases

- Configuration Files: Interpreting custom configuration file formats.
- Query Languages: Interpreting queries in domain-specific languages.
- Command Interpreters: Interpreting and executing commands in a command-line interface.

Conclusion: The Interpreter Design Pattern provides a way to define and interpret a language's grammar. It is best suited for simple grammars and expressions, offering a clear and extensible way to interpret sentences. While it has its advantages in terms of structure and reusability, it may not be suitable for complex or performance-critical scenarios. Understanding when and how to use the Interpreter Pattern is crucial for effectively managing and interpreting domain-specific languages.

Command Design Pattern

Introduction: The Command Design Pattern is a behavioral design pattern that turns a request into a stand-alone object containing all information about the request. This transformation allows you to parameterize methods with different requests, delay or queue a request's execution, and support undoable operations.

Which SOLID Principle Followed by This Pattern: The Command Pattern adheres to the Single Responsibility Principle (SRP) by encapsulating a request as an object, and the Open/Closed Principle (OCP) by allowing new commands to be added without changing existing code.

Type of This Pattern: The Command Pattern is a Behavioral pattern, focusing on how objects interact and communicate to perform tasks.

Components of This Pattern

- Command Interface: Declares an interface for executing an operation.
- Concrete Command: Implements the command interface, binding a receiver object to an action.
- Receiver: Knows how to perform the operations associated with carrying out a request.
- Invoker: Asks the command to carry out the request.
- Client: Creates a concrete command and sets its receiver.

What Is This Pattern: The Command Pattern encapsulates a request as an object, thereby allowing for the parameterization of clients with queues, requests, and operations. It also provides support for undoable operations and can facilitate logging changes, transaction management, etc.

Why Do We Need This Pattern

- Decouple the sender and receiver of a request.
- Allow for the easy addition of new commands without altering existing code.
- Enable the queuing of requests, logging of request histories, and supporting undo operations.

How Does This Pattern Work

- Define Command Interface: Create an interface with an execute method.
- Implement Concrete Commands: Create concrete classes implementing the command interface, encapsulating actions and receivers.
- Receiver: Implement the actions to be performed.
- Invoker: Store and execute commands.
- Client: Instantiate and configure commands and their receivers.

When to Use This Pattern

- When you need to parameterize objects with operations.
- When you need to support undo functionality.
- When you need to log changes or queue requests.

When Not Recommended to Use This Pattern

- When the complexity added by creating command classes is not justified by the simplicity of the operation.
- When the operations are simple and unlikely to change, making the pattern overkill.

Layman Example

- Remote Control: Imagine a remote control for a TV where each button (command) performs a different operation like turning the TV on/off or changing channels. The remote (invoker) sends the command, the TV (receiver) executes it.
- Restaurant Order: Think of a restaurant where a waiter takes an order (command), places it in the kitchen (invoker), and the chef (receiver) prepares the meal. Each order can be processed independently.

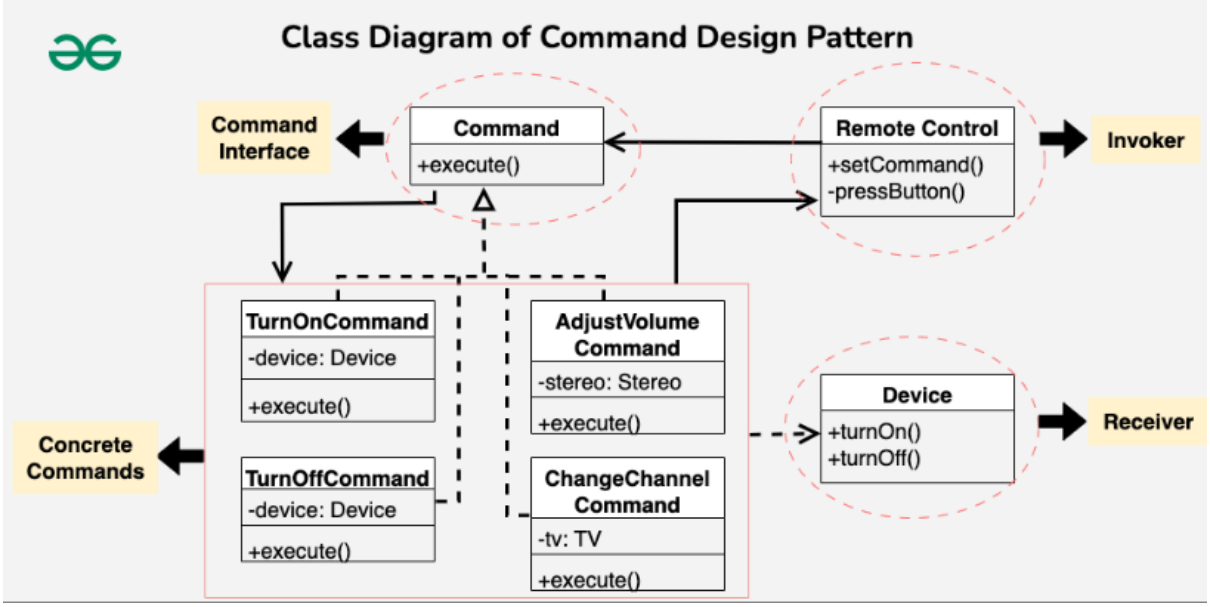
Technical Example

Imagine you are tasked with designing a remote control system for various electronic devices in a smart home. The devices include a TV, a stereo, and potentially other appliances. The goal is to create a flexible remote control that can handle different types of commands for each device, such as turning devices on/off, adjusting settings, or changing channels.

Code

https://www.geeksforgeeks.org/command-pattern/

Flow Diagram



- Client: Creates concrete command objects.
- Invoker: Stores commands and executes them.
- Command Interface: Defines the execute method.
- Concrete Commands: Implement the execute method.
- Receiver: Executes the actual logic.

Advantages

- Decouples Sender and Receiver: Allows the sender to issue requests without knowing the receiver's implementation.
- Extensible: Easy to add new commands without changing existing code.
- Support for Undo/Redo: Commands can store state for undo/redo functionality.
- Command Queuing and Logging: Commands can be queued and logged for later execution or review.

Disadvantages

- Increased Complexity: Adds additional layers of complexity with multiple command classes.

- Overhead: Might be overkill for simple operations.

Real Life Scenarios Use Cases

- GUI Buttons: Each button press can be mapped to a command.
- Transaction Management: Commands can be used to log and replay transactions.
- Macro Recording: Recording a series of actions and replaying them as a macro.

Conclusion: The Command Design Pattern is an effective way to encapsulate requests as objects, allowing for extensibility, decoupling, and the ability to manage operations such as undo, redo, and logging. While it introduces additional complexity, the benefits it provides in terms of flexibility and maintainability often outweigh the costs, especially in systems that require versatile request handling and operations management. Understanding and implementing the Command Pattern can significantly enhance the design and functionality of complex applications.

Chain of Responsibility Design Pattern

Introduction: The Chain of Responsibility Design Pattern is a behavioral design pattern that allows an event to be processed by one of many handlers. Handlers are arranged in a chain, and each handler decides either to process the request or to pass it to the next handler in the chain.

Which SOLID Principle Followed by This Pattern: The Chain of Responsibility Pattern follows the Single Responsibility Principle (SRP) by decoupling the request sender from the receiver, and the Open/Closed Principle (OCP) by allowing new handlers to be added to the chain without modifying existing code.

Type of This Pattern: The Chain of Responsibility Pattern is a Behavioral pattern, as it is concerned with the assignment of responsibilities between objects.

Components of This Pattern

- Handler (Abstract): Defines an interface for handling requests and optionally sets the next handler.
- Concrete Handler: Handles requests it is responsible for, and forwards others along the chain.
- Client: Initiates the request to a handler in the chain.

What Is This Pattern: The Chain of Responsibility Pattern creates a chain of handler objects. Each handler decides whether to process the request or pass it to the next handler in the chain.

Why Do We Need This Pattern

- Avoid coupling the sender of a request to its receiver.
- Allow multiple objects to handle the request.
- Enable dynamic addition and removal of handlers.

How Does This Pattern Work

- Define Handlers: Create abstract and concrete handler classes.
- Set Up Chain: Link handlers together.
- Process Request: Pass requests along the chain until a handler processes it.

When to Use This Pattern

- When multiple objects might handle a request, but the specific handler isn't known beforehand.
- When you want to issue a request to one of several objects without specifying the receiver explicitly.
- When the set of handlers and their order can be dynamically changed.

When Not Recommended to Use This Pattern

- When every handler is required to process a request (chain might add unnecessary complexity).
- When request processing doesn't need to be distributed across multiple handlers.

Layman Example

- Customer Service: Think of a customer service system where an issue can be resolved by a frontline representative, escalated to a supervisor, and finally to a manager. Each level in the chain can handle the issue or pass it along.

- Approval Process: Consider a document approval process where a document must be approved by a team leader, then a department head, and finally the CEO. Each person in the chain can approve or escalate it.

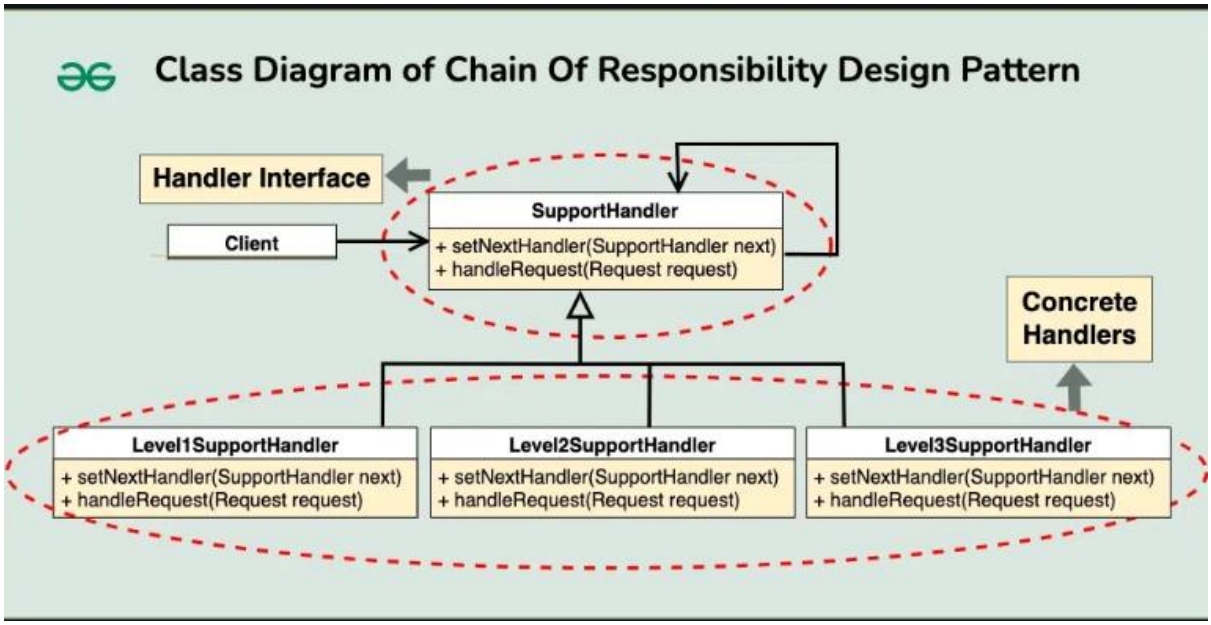
Technical Example

Imagine a customer support system where customer requests need to be handled based on their priority. There are three levels of support: Level 1, Level 2, and Level 3. Level 1 support handles basic requests, Level 2 support handles more complex requests, and Level 3 support handles critical issues that cannot be resolved by Level 1 or Level 2.

Code

https://www.geeksforgeeks.org/chain-responsibility-design-pattern/

Flow Diagram



- Client: Sends a request to the first handler.
- Handler Interface: Defines a method to handle requests.
- Concrete Handlers: Implement the method, handling specific requests or passing them along the chain.
- Chain: Represents the sequence of handlers through which the request passes.

Advantages

- Decoupling: Reduces the coupling between the sender and receiver.
- Responsibility Sharing: Distributes responsibility among handlers.
- Flexibility: Allows dynamic addition/removal of handlers.

Disadvantages

- Uncertain Handling: No guarantee that the request will be handled.
- Complexity: Can add complexity with long chains of handlers.

Real Life Scenarios Use Cases

- Technical Support: Handling customer support tickets through various levels of support.
- Event Handling: Event processing systems where events pass through multiple stages of handling.

Conclusion: The Chain of Responsibility Pattern provides a flexible way to pass requests along a chain of handlers. It allows different parts of an application to handle requests without coupling the sender to a specific handler. While it offers significant advantages in terms of decoupling and flexibility, it can introduce complexity, especially if the chain becomes too long or handlers are not appropriately managed. Understanding when and how to use the Chain of Responsibility Pattern can significantly improve the design and maintainability of software systems.

State Design Pattern

Introduction: The State Design Pattern is a behavioral design pattern that allows an object to alter its behavior when its internal state changes. This pattern involves defining a state interface, concrete state classes representing various states, and a context class that maintains an instance of the current state.

Which SOLID Principle Followed by This Pattern: The State Design Pattern adheres to the Single Responsibility Principle (SRP) by encapsulating state-specific behavior in separate state classes. It also aligns with the Open/Closed Principle (OCP) by allowing the addition of new states without modifying existing code.

Type of This Pattern: The State Design Pattern is a Behavioral pattern as it deals with the behavior of objects.

Components of This Pattern

- State Interface: Defines methods that encapsulate behavior based on the state.
- Concrete State Classes: Implement the state interface and represent specific states.
- Context Class: Maintains a reference to the current state and delegates state-specific behavior to the current state object.

What Is This Pattern: The State Design Pattern allows an object to change its behavior when its internal state changes. It encapsulates the behavior related to different states in separate classes, making the object's behavior more modular and flexible.

Why Do We Need This Pattern

- Simplify complex conditional logic by encapsulating it in state classes.
- Allow an object's behavior to change dynamically based on its internal state.
- Improve code readability and maintainability by organizing state-specific behavior into separate classes.

How Does This Pattern Work

- Define State Interface: Create an interface that declares methods representing state-specific behavior.
- Implement Concrete State Classes: Create concrete classes implementing the state interface, each representing a different state and its behavior.
- Define Context Class: Create a class that maintains a reference to the current state and delegates state-specific behavior to the current state object.
- Change State: Allow the context object to switch between different states dynamically based on its internal state.

When to Use This Pattern

- When an object's behavior depends on its state and must change dynamically at runtime.
- When there are multiple conditional statements based on an object's state, leading to complex code.

When Not Recommended to Use This Pattern

- When the behavior of an object doesn't depend on its internal state.
- When there are only a few states, and the overhead of implementing the pattern outweighs its benefits.

Layman Example

- Traffic Light: Think of a traffic light that transitions between different states (red, yellow, green) based on the traffic conditions. Each state has its behavior (e.g., stopping, slowing down, moving) dictated by the traffic light's current state.
- Vending Machine: Consider a vending machine that changes its behavior based on its current state (e.g., idle, selecting item, dispensing item). The machine's actions (e.g., accepting money, dispensing products) vary depending on its state.

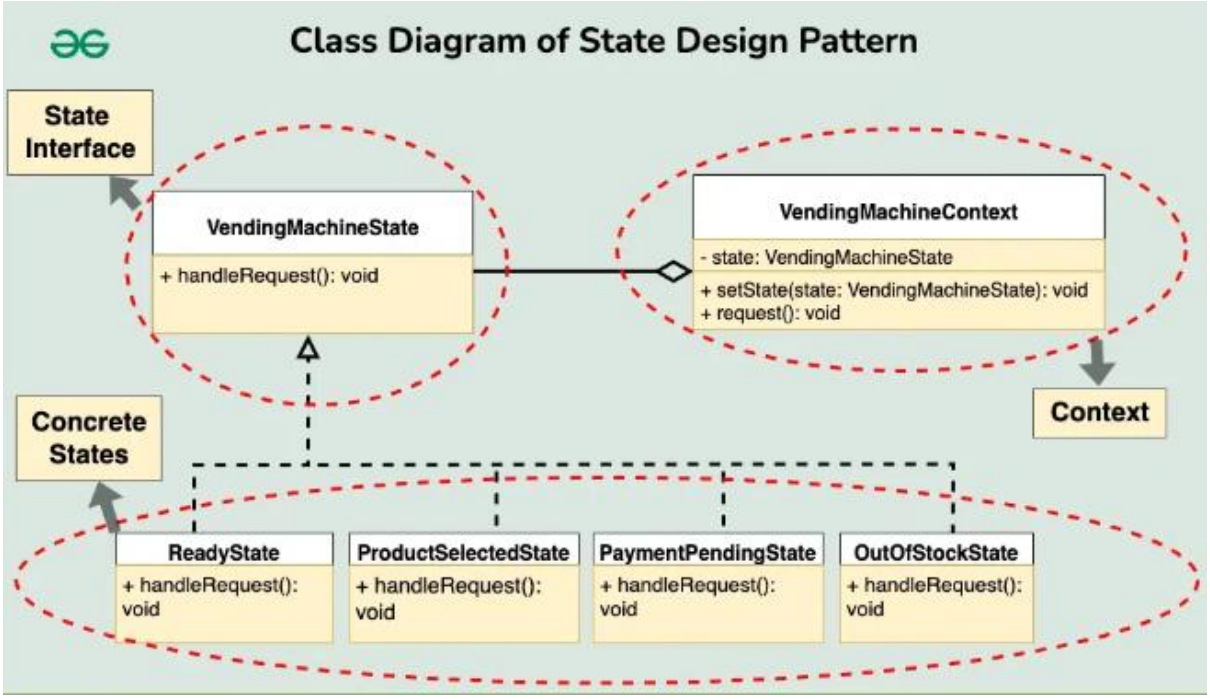
Technical Example

Imagine a vending machine that sells various products. The vending machine needs to manage different states such as ready to serve, waiting for product selection, processing payment, and handling out-of-stock situations. Design a system that models the behavior of this vending machine efficiently.

Code

https://www.geeksforgeeks.org/state-design-pattern/

Flow Diagram



- A flow diagram would illustrate how the context object delegates state-specific behavior to the current state object, allowing the object's behavior to change dynamically.

Advantages

- Simplifies Complex Logic: Encapsulates state-specific behavior, making code easier to understand and maintain.
- Promotes Loose Coupling: States are independent of each other, promoting modular and flexible code.
- Enhances Extensibility: Easy to add new states without modifying existing code.

Disadvantages

- Potential Overhead: Introducing multiple state classes can increase code complexity, especially for simple applications.

- Increased Number of Classes: Requires the creation of separate state classes, which may lead to a larger codebase.

Real Life Scenarios Use Cases

- Order Processing System: Handling different order states (e.g., pending, processing, shipped) in an e-commerce application.
- User Authentication System: Managing user login states (e.g., logged in, logged out, session expired) in a web application.

Conclusion: The State Design Pattern provides an elegant solution for managing an object's behavior dynamically based on its internal state. By encapsulating state-specific behavior in separate classes, the pattern promotes code reusability, maintainability, and extensibility. While introducing additional classes, the benefits of improved code organization and flexibility often outweigh the associated overhead. Understanding and applying the State Design Pattern can significantly enhance the design and architecture of software systems.

Strategy Design Pattern

Introduction: The Strategy Design Pattern is a behavioral design pattern that enables an object to alter its behavior at runtime by encapsulating different algorithms within separate classes and allowing the client to choose the desired algorithm dynamically.

Which SOLID Principle Followed by This Pattern: The Strategy Design Pattern adheres to the Open/Closed Principle (OCP) by allowing new algorithms (strategies) to be added without altering the client code.

Type of This Pattern: The Strategy Design Pattern is a Behavioral pattern, as it deals with the assignment of behaviors to objects.

Components of This Pattern

- Strategy Interface: Defines a common interface for all concrete strategies.
- Concrete Strategies: Implements the strategy interface and encapsulates different algorithms.
- Context: Contains a reference to a strategy object and allows clients to set the desired strategy dynamically.

What Is This Pattern: The Strategy Design Pattern defines a family of algorithms, encapsulates each one into a separate class, and makes them interchangeable. This pattern allows the client to choose the appropriate algorithm dynamically at runtime.

Why Do We Need This Pattern

- Encapsulate algorithms in separate classes, promoting code reusability and maintainability.
- Enable the selection of different algorithms at runtime without modifying the client code.
- Simplify complex conditional statements by delegating algorithm selection to strategy objects.

How Does This Pattern Work

- Define Strategy Interface: Create an interface or abstract class to define a common method for all algorithms.
- Implement Concrete Strategies: Implement the interface with different algorithms in separate concrete strategy classes.
- Create Context: Create a context class that holds a reference to a strategy object and defines a method to set or change the strategy.
- Client Interaction: Clients interact with the context class, which delegates the algorithm execution to the current strategy object.

When to Use This Pattern

- When you have multiple related algorithms and want to encapsulate each one separately.
- When you need to switch between different algorithms dynamically at runtime.
- When you want to avoid conditional statements for selecting algorithms.

When Not Recommended to Use This Pattern

- When there is only one algorithm and it's unlikely to change.
- When the overhead of creating multiple strategy classes outweighs the benefits.

Layman Example

- Travel Planning: Consider a travel planning application where users can choose different modes of transportation (e.g., car, train, plane) for their trips. Each mode of transportation represents a different strategy with its own cost calculation and route planning algorithms.
- Payment Methods: Think of an online shopping application where users can choose between different payment methods (e.g., credit card, PayPal, bank transfer). Each payment method represents a different strategy for processing payments and handling transaction details.

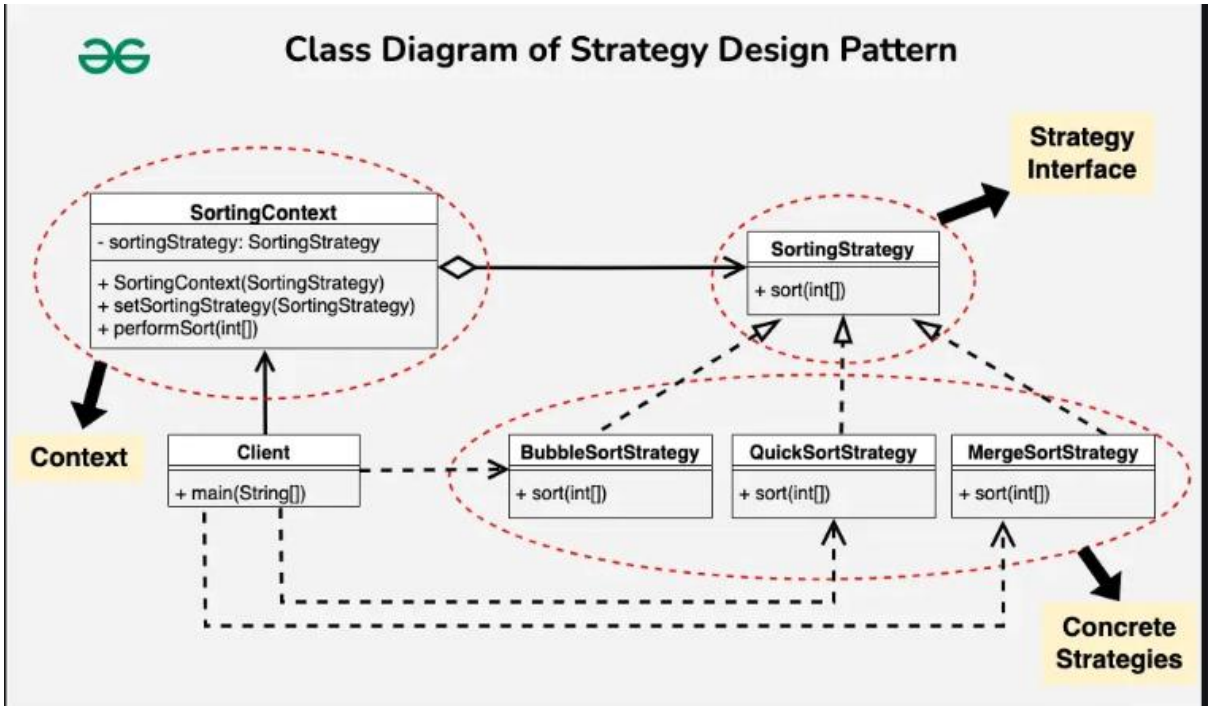
Technical Example

Let's consider a sorting application where we need to sort a list of integers. However, the sorting algorithm to be used may vary depending on factors such as the size of the list and the desired performance characteristics.

Code

https://www.geeksforgeeks.org/strategy-pattern-set-1/

Flow Diagram



- A flow diagram would illustrate how the context delegates the sorting task to the current strategy object, allowing the algorithm to be chosen dynamically.

Advantages

- Flexibility: Allows clients to choose different algorithms at runtime.
- Modularity: Encapsulates algorithms into separate classes, promoting code reuse and maintainability.
- Easy Testing: Each strategy can be tested independently.

Disadvantages

- Increased Number of Classes: Introducing multiple strategy classes may lead to a larger codebase.
- Complexity: Managing multiple strategies and their interactions can add complexity to the code.

Real Life Scenarios Use Cases

- Data Compression: Choosing different compression algorithms based on data characteristics.
- Route Planning: Selecting different route planning algorithms for navigation applications.

Conclusion: The Strategy Design Pattern provides an effective way to encapsulate algorithms and make them interchangeable at runtime. By separating algorithms into separate classes, this pattern promotes code reusability, maintainability, and flexibility. While introducing additional classes, the benefits of modularizing algorithms and enabling dynamic algorithm selection often outweigh the associated overhead. Understanding and implementing the Strategy Design Pattern can greatly improve the design and extensibility of software systems.

Observer Design Pattern

Introduction: The Observer Design Pattern is a behavioral design pattern where an object, known as the subject, maintains a list of its dependents, called observers, and notifies them of any state changes, usually by calling one of their methods.

Which SOLID Principle Followed by This Pattern: The Observer Design Pattern adheres to the Single Responsibility Principle (SRP) by separating the concerns of subject and observer objects. It also aligns with the Open/Closed Principle (OCP) by allowing new observers to be added without modifying the subject.

Type of This Pattern: The Observer Design Pattern is a Behavioral pattern, as it's primarily concerned with the communication between objects.

Components of This Pattern

- Subject: Maintains a list of observers, provides methods to add and remove observers, and notifies observers of state changes.
- Observer: Defines an interface for objects that should be notified of changes in the subject's state.
- Concrete Subject: Implements the subject interface and sends notifications to observers when its state changes.
- Concrete Observer: Implements the observer interface and receives notifications from the subject.

What Is This Pattern: The Observer Design Pattern establishes a one-to-many dependency between objects, where changes in one object (the subject) are reflected in its dependent objects (observers) automatically.

Why Do We Need This Pattern

- Enable objects to be notified of changes in another object's state without tight coupling.
- Support the principle of loose coupling between objects.
- Allow multiple objects to react to changes in a single object's state.

How Does This Pattern Work

- The subject maintains a list of observers.
- When the subject's state changes, it notifies all observers by calling their update method.
- Each observer implements the update method to handle the notification appropriately.

When to Use This Pattern

- When multiple objects need to be notified of changes in another object's state.
- When the number of objects that need to be notified is unknown or may vary.
- When an object needs to notify other objects without making assumptions about who these objects are.

When Not Recommended to Use This Pattern

- When there's only a single observer or a fixed number of observers, and their interaction is straightforward.
- When the subject and observers are tightly coupled and changing one requires modifying the other.

Layman Example

- News Agency: Think of a news agency as the subject and subscribers as observers. When the news agency publishes a new article, it notifies all subscribers (observers) by sending them the latest news.
- Stock Market: Consider a stock market application where users are interested in specific stocks. The stock market acts as the subject, and users are observers. When the price of a stock changes, all users interested in that stock are notified.

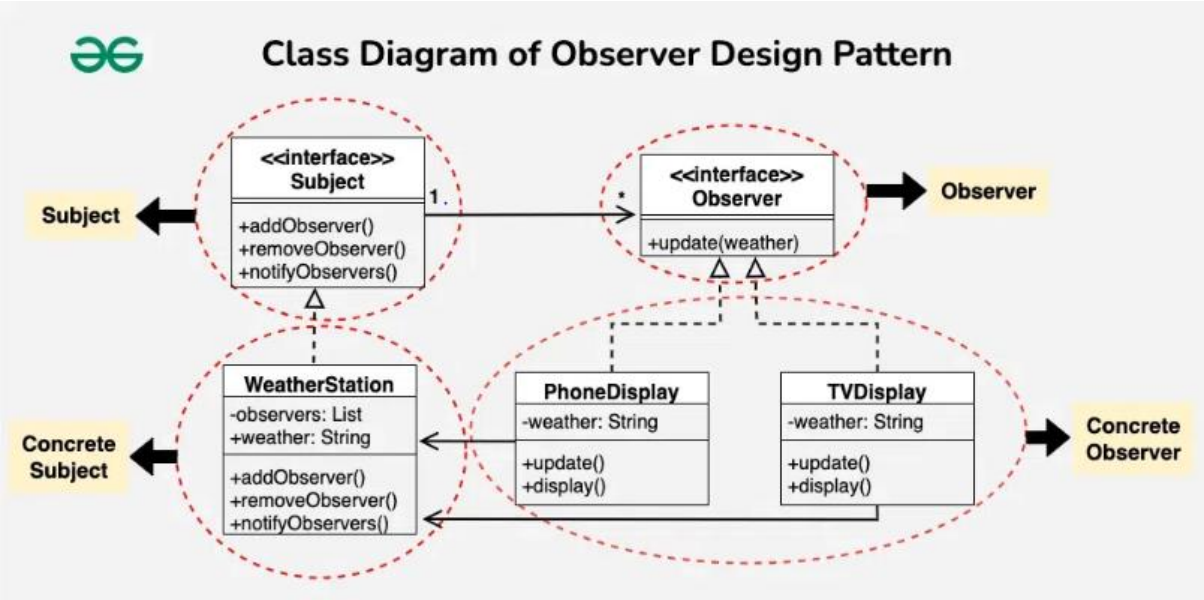
Technical Example

Consider a scenario where you have a weather monitoring system. Different parts of your application need to be updated when the weather conditions change.

Code

https://www.geeksforgeeks.org/observer-pattern-set-1-introduction/

Flow Diagram



- A flow diagram would illustrate the flow of notifications from the subject to observers when the subject's state changes.

Advantages

- Decoupling: Enables subjects and observers to interact without tightly coupling them together.
- Flexibility: Allows for dynamic addition and removal of observers at runtime.
- Reusability: Promotes reuse of observer and subject components in different contexts.

Disadvantages

- Unexpected Updates: Observers may receive unnecessary or unexpected notifications.
- Complexity: Introducing multiple observers and subjects may increase code complexity.

Real Life Scenarios Use Cases

- Social Media Notifications: Users receive notifications when someone likes or comments on their posts.
- UI Updates: User interface components are updated in response to changes in underlying data models.

Conclusion: The Observer Design Pattern provides a flexible solution for implementing one-to-many relationships between objects, allowing objects to communicate changes in state without tight coupling. By separating concerns and enabling dynamic interactions, this pattern promotes code reusability, maintainability, and flexibility. While introducing additional classes and complexity, the benefits of loosely coupled components and dynamic interactions often outweigh the associated overhead. Understanding and applying the Observer Design Pattern can greatly enhance the design and scalability of software systems.

Template Design Pattern

Introduction: The Template Design Pattern 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.

Which SOLID Principle Followed by This Pattern: The Template Design Pattern primarily follows the Open/Closed Principle (OCP) by allowing subclasses to extend the behavior of the superclass without modifying its code.

Type of This Pattern: The Template Design Pattern is a Behavioral pattern as it deals with the behavior of objects.

Components of This Pattern

- Abstract Class: Defines the skeleton of the algorithm with abstract methods representing steps.
- Concrete Classes: Implement the abstract methods to define specific behavior for each step.

What Is This Pattern: The Template Design Pattern defines the basic steps of an algorithm in a superclass and allows subclasses to override specific steps to provide custom implementations while keeping the overall algorithm structure intact.

Why Do We Need This Pattern

- Define a common algorithm structure while allowing subclasses to provide their own implementations for specific steps.
- Promote code reuse by encapsulating common behavior in a superclass.
- Ensure consistency in algorithms across subclasses.

How Does This Pattern Work

- The abstract class defines a template method that contains the algorithm's skeleton, including calls to abstract methods.
- Concrete subclasses extend the abstract class and provide implementations for the abstract methods, customizing the behavior of specific steps.
- Clients interact with concrete subclasses, invoking the template method to execute the algorithm.

When to Use This Pattern

- When you have an algorithm with common steps but varying implementations for specific steps.
- When you want to avoid code duplication by encapsulating common behavior in a superclass.

When Not Recommended to Use This Pattern

- When the algorithm structure is unlikely to change, and all steps have fixed implementations.
- When the algorithm becomes too complex, and subclassing may lead to confusion or an overly large hierarchy.

Layman Example

- Baking a Cake: Think of a recipe for baking a cake. The recipe outlines the general steps (e.g., mixing ingredients, baking) but allows for variations in ingredients and decorations (e.g., chocolate vs. vanilla, frosting vs. icing).
- Building a House: Consider a construction blueprint for building a house. The blueprint provides a general outline of the building process (e.g., foundation, framing, finishing), but contractors can customize materials and design details based on client preferences.

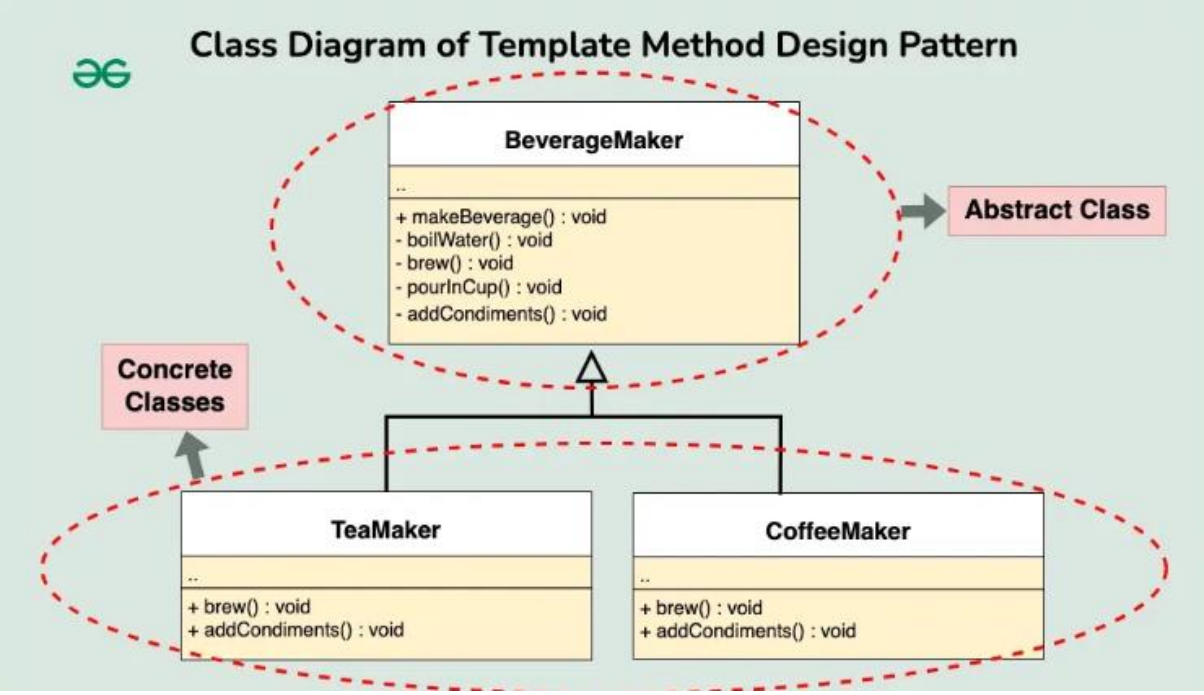
Technical Example

Let's consider a scenario where we have a process for making different types of beverages, such as tea and coffee. While the overall process of making beverages is similar (e.g., boiling water, adding ingredients), the specific steps and ingredients vary for each type of beverage.

Code

https://www.geeksforgeeks.org/template-method-design-pattern/

Flow Diagram



- A flow diagram would illustrate how the template method calls the abstract methods defined in the superclass, and concrete subclasses provide custom implementations for these methods.

Advantages

- Code Reuse: Promotes reuse of common algorithm steps across subclasses.
- Flexibility: Allows subclasses to provide custom implementations for specific steps.
- Encapsulation: Encapsulates algorithm structure in a superclass, improving code organization and readability.

Disadvantages

- Inflexibility: Subclasses are limited to the structure defined by the superclass, which may not always be suitable.
- Complexity: Introducing too many abstract methods or subclasses can lead to a complex class hierarchy.

Real Life Scenarios Use Cases

- Report Generation: Generating reports with different formats (e.g., PDF, HTML) but following a similar structure.
- Data Processing Pipelines: Processing data with multiple stages (e.g., reading, parsing, analyzing) where some stages may vary.

Conclusion: The Template Design Pattern provides a flexible way to define the skeleton of an algorithm in a superclass while allowing subclasses to override specific steps. By encapsulating common behavior in a superclass and allowing for customization in subclasses, this pattern promotes code reuse, flexibility, and maintainability. While introducing additional classes and complexity, the benefits of reusability and flexibility often outweigh the associated overhead. Understanding and applying the Template Design Pattern can significantly improve the design and scalability of software systems.

Types of Creational Design Patterns:

Hint: “Abraham becomes First President of State”

- A: Abstract Factory
- B: Builder
- F: Factory Method
- P: Prototype
- S: Singleton

Types of Structural Design Patterns:

Hint: “Finding ABC of DP Fastly”

- F: Facade Pattern
- A: Adapter Pattern
- B: Bridge Pattern
- C: Composite Pattern
- D: Decorator Pattern
- P: Proxy Pattern
- F: Flyweight Pattern

Types of Behavioral Design Patterns:

Hint: “2MICS On TV (MMIICCSSOTV)”

- 2M: Mediator Pattern & Memento Pattern
- 2I: Iterator Pattern & Interpreter Pattern
- 2C: Command Pattern & Chain-Of-Responsibility Pattern
- 2S: State Pattern & Strategy Pattern
- O: Observer Pattern
- T: Template Method Pattern
- V: Visitor Pattern