The **Factory Design Pattern** is a creational pattern used to create objects without specifying the exact class of object that will be created. It provides a method to create objects based on certain input or conditions, making it easier to add new types without changing the client code.

**Scenario**

Imagine you have an e-commerce application where users can make different types of payments, such as credit card, PayPal, or bank transfer. You could use the Factory Design Pattern to generate payment processors dynamically based on the payment type requested by the user.

**Steps:**

1. **Define the abstract product (Payment Processor interface)**: This is the common interface that all payment processors will implement.
2. **Concrete products (Payment Processor implementations)**: These are the actual implementations for each type of payment (e.g., CreditCardPaymentProcessor, PayPalPaymentProcessor).
3. **Factory class**: The factory class will create instances of the correct payment processor based on the payment method requested in the API.

The **Abstract Factory Pattern** is a creational design pattern that provides an interface for creating families of related or dependent objects without specifying their concrete classes. It is particularly useful when you need to create products that are part of a family of related products, and the exact type of product isn't known until runtime.

**Real-World Scenario: Abstract Factory for Sending Notifications**

Let's imagine you are building a system that sends notifications (e.g., emails, SMS, or push notifications) to users. Based on the user's preferred notification channel, the API should choose the appropriate method of sending the notification. The Abstract Factory Pattern can be used to create a family of notification services (e.g., Email Notification, SMS Notification, and Push Notification).

**Steps:**

1. **Abstract Factory**: Defines an interface for creating a family of related notification objects.
2. **Concrete Factories**: Implement the Abstract Factory to create specific notification products (e.g., EmailNotificationFactory, SMSNotificationFactory).
3. **Abstract Products**: Defines the interface for the notification product (e.g., INotification).
4. **Concrete Products**: Implement the abstract notification interface (e.g., EmailNotification, SMSNotification, PushNotification).
5. **Client**: The REST API client that calls the abstract factory to send notifications based on user preference.

The **Singleton Pattern** is a creational design pattern that ensures a class has only one instance and provides a global point of access to that instance. It is typically used when you need to control access to shared resources, such as a configuration manager, a logging system, or a database connection, ensuring that only one instance of the resource exists throughout the application's lifetime.

**Real-Life Example: Database Connection**

Let's consider a real-life example of a **Database Connection** class in a web application. A **Singleton** pattern is ideal for managing database connections because you only want one instance of the connection object throughout the entire application, ensuring that multiple database connections are not created unnecessarily.

The **Builder Pattern** is a design pattern that is used to separate the construction of a complex object from its representation. It allows you to create different representations of the same type of object by using the same construction process.

Let's walk through an example of using the Builder Pattern in a **real-life scenario** involving a REST API in Python. For this, we'll simulate building an API request with various optional fields using the Builder Pattern.

**Scenario:**

When ordering a pizza, customers can customize their pizza with different toppings, sizes, crust types, sauces, etc. We can use the **Builder Pattern** to allow for a flexible way to construct a pizza order.

**Steps:**

1. **Define the Product (Pizza Order):** This is the object that will be constructed using the builder. It will contain all the details about the pizza.
2. **Create the Builder Class:** This class will have methods to set different attributes for the pizza (e.g., size, crust, toppings, etc.).
3. **Director Class (Optional):** This class can be used to streamline the pizza creation process, offering predefined pizza combinations like "Margherita" or "Pepperoni."
4. **Customer Class (Optional):** A customer can use the builder to create their custom pizza order.

The **Prototype Pattern** is a design pattern where you create new objects by copying an existing object, known as the prototype. This is useful when creating new objects is costly or complex, and cloning an existing one is more efficient.

**Steps:**

1. **Create a Prototype Interface**: This will define the method for cloning objects.
2. **Concrete Prototypes**: These represent different types of objects (e.g., Book, Electronics).
3. **Client API**: The REST API will allow clients to fetch, clone, and modify prototypes.

The **Decorator Pattern** is a structural design pattern that allows behavior to be added to an individual object dynamically, without affecting the behavior of other objects of the same class. It is useful when you want to add responsibilities to objects without altering their structure, and it is often used for logging, validation, caching, and other cross-cutting concerns.

The **Proxy Design Pattern** is a structural design pattern that provides an object representing another object. A proxy controls access to the original object, allowing you to add additional behavior or logic before delegating requests to the real object. This pattern is useful for scenarios like lazy loading, access control, logging, monitoring, and caching.

**Example Scenario for Proxy Pattern in a REST API:**

Imagine you're building an API that fetches data from a remote resource (e.g., a third-party service or a database). Instead of directly calling the remote service every time, you use a **Proxy** to control access to the resource. The proxy can:

* **Cache results** to reduce the number of remote calls.
* **Log requests** for monitoring.
* **Check access permissions** before forwarding the request to the real service

**Advantages of the Proxy Pattern in This Example:**

1. **Caching**: The proxy pattern allows caching the results of expensive operations (like database calls or API requests), improving performance.
2. **Access Control**: The proxy can check if the request is authorized before delegating it to the real service.
3. **Logging/Monitoring**: The proxy can log or monitor requests before passing them to the real service, providing a way to gather metrics or debugging information.
4. **Lazy Initialization**: The proxy can delay the creation or initialization of the real object until it is actually needed.

**Real-World Applications of Proxy Pattern:**

* **Remote Proxies**: Accessing a remote object, such as an object in a different process or on a different server.
* **Virtual Proxies**: Delaying the instantiation of a resource-intensive object until it's really needed (e.g., a large file or a database connection).
* **Protective Proxies**: Controlling access to an object by verifying credentials, such as protecting sensitive data or APIs.
* **Caching Proxy**: Storing the result of a function call and returning the cached value on subsequent requests to improve performance, as shown in the example.

The **Adapter Pattern** is a structural design pattern that allows two incompatible interfaces to work together. The adapter serves as a bridge between the original interface and the expected interface. In real-world scenarios, the adapter pattern is used when you need to integrate third-party libraries, APIs, or services with a system that expects a different interface.

**Scenario:**

You need to integrate two different types of logging systems into a REST API:

1. **File Logger**: This logger writes logs to a file.
2. **Database Logger**: This logger saves logs to a database.

You want to create a unified interface for logging in your API, regardless of whether the actual logging is done via file or database. The **Adapter Pattern** can be used to provide a consistent interface to the rest of the system.

**Steps:**

1. **Create the Existing Logging Systems**: Two separate logging systems with different interfaces.
2. **Define a Unified Logger Interface**: An abstract class or interface that all adapters will implement.
3. **Implement the Adapter for Each Logging System**: Adapters for both the file logger and the database logger.
4. **Create the REST API**: The API will use the unified adapter interface to perform logging, without worrying about which logging system is being used.

This time, we'll model a **Menu System** for a restaurant, where menus can contain individual items (like burgers or drinks) as well as other menus (like "Appetizers" or "Desserts").

**Scenario:**

In a restaurant application, you have individual menu items (like dishes, drinks, etc.) as well as composite menus that group multiple items or other submenus. The composite pattern will help us manage both individual menu items and complex menu structures in a consistent way.

**Steps:**

1. **Define a common interface** for all menu items (both individual and composite).
2. **Create individual menu items** (like MenuItem for a single dish or drink).
3. **Create a composite menu** (like Menu) that can contain other menu items or even submenus.
4. **Use the uniform interface** to print out the entire menu, whether it consists of individual items or nested submenus.