Creational Design Patterns

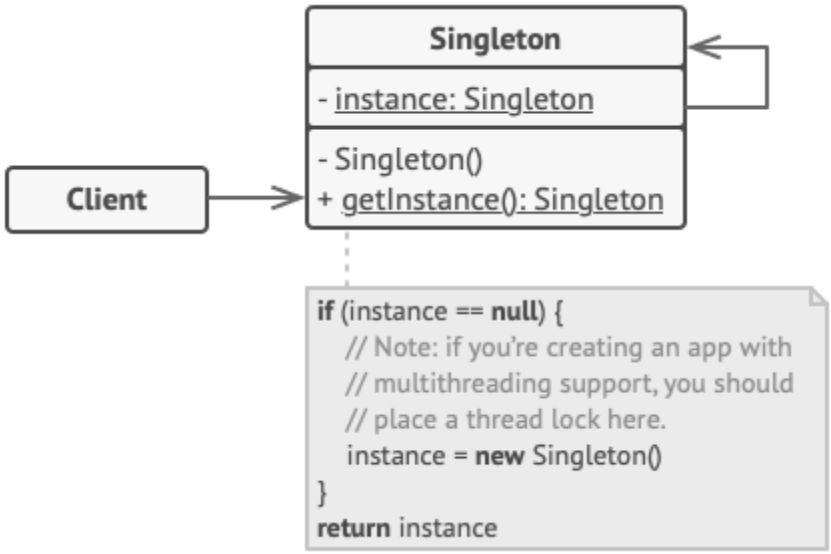
Singleton

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. It's ideal for scenarios requiring centralized control, like managing database connections or configuration settings.

Implementation:

1. Add a private static field to the class for storing the singleton instance.
2. Declare a public static creation method for getting the singleton instance.
3. Implement "lazy initialization" inside the static method. It should create a new object on its first call and put it into the static field. The method should always return that instance on all subsequent calls.
4. Make the constructor of the class private. The static method of the class will still be able to call the constructor, but not the other objects.
5. Go over the client code and replace all direct calls to the singleton's constructor with calls to its static creation method.



1. The **Singleton** class declares the static method getInstance that returns the same instance of its own class.

The Singleton's constructor should be hidden from the client code. Calling the getInstance method should be the only way of getting the Singleton object.

Pros

- You can be sure that a class has only a single instance.
- You gain a global access point to that instance.
- The singleton object is initialized only when it's requested for the first time.

Cons

- Violates the *Single Responsibility Principle*. The pattern solves two problems at the time.
- The Singleton pattern can mask bad design, for instance, when the components of the program know too much about each other.
- The pattern requires special treatment in a multithreaded environment so that multiple threads won't create a singleton object several times.
- It may be difficult to unit test the client code of the Singleton because many test frameworks rely on inheritance when producing mock objects. Since the constructor of the singleton class is private and overriding static methods is impossible in most languages, you will need to think of a creative way to mock the singleton. Or just don't write the tests. Or don't use the Singleton pattern.

Factory Method

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.

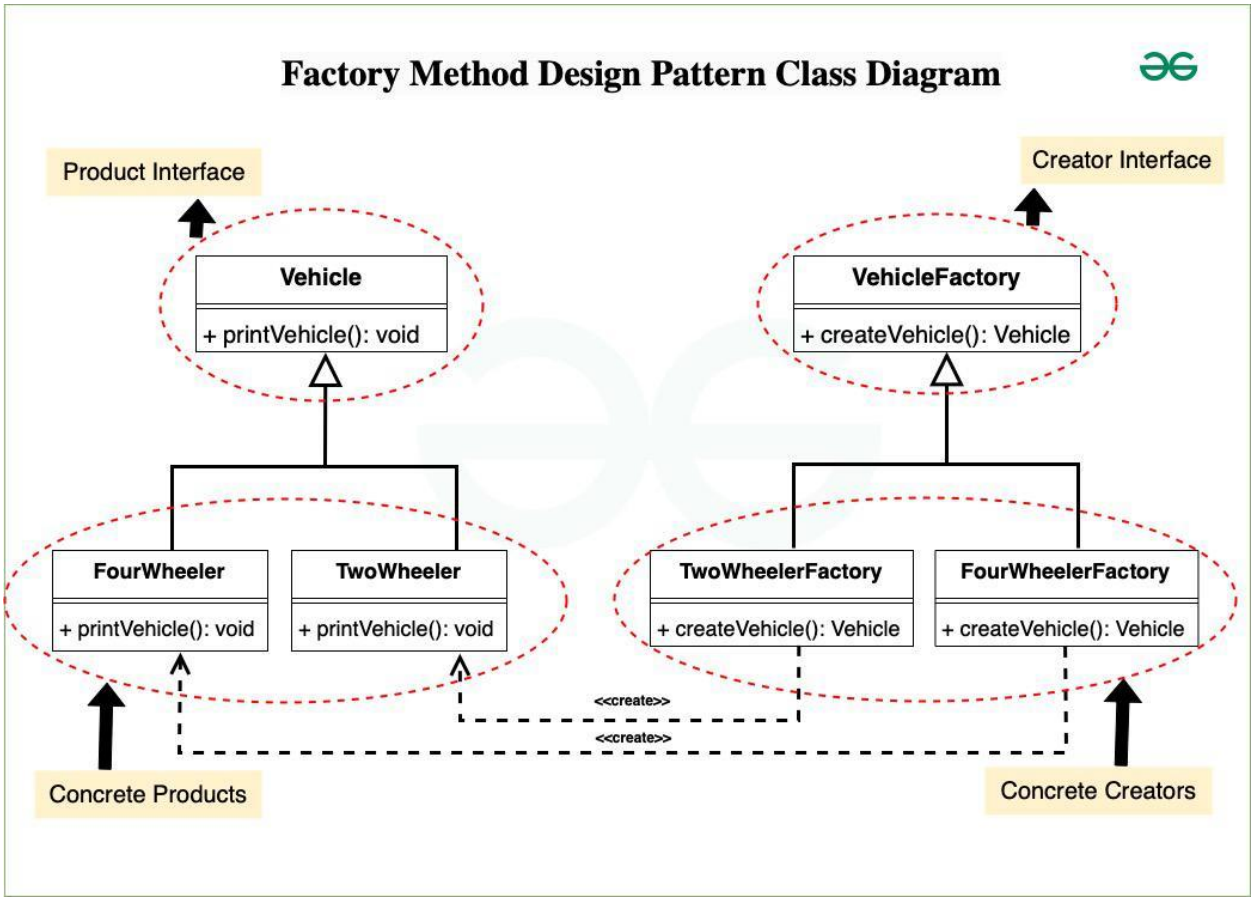
Below are the main components of Factory Design Pattern:

- **Creator**: This is an abstract class or an interface that declares the factory method. The creator typically contains a method that serves as a factory for creating objects. It may also contain other methods that work with the created objects.

- **Concrete Creator**: Concrete Creator classes are subclasses of the Creator that implement the factory method to create specific types of objects. Each Concrete Creator is responsible for creating a particular product.

- **Product**: This is the interface or abstract class for the objects that the factory method creates. The Product defines the common interface for all objects that the factory method can create.

- **Concrete Product**: Concrete Product classes are the actual objects that the factory method creates. Each Concrete Product class implements the Product interface or extends the Product abstract class



The **factory design pattern** is a creational design pattern that provides an interface or method to create objects without specifying their exact classes. Let's break down the pros and cons mentioned:

Pros

1. **Avoid tight coupling between the creator and the concrete products**

- Tight coupling occurs when a class is directly dependent on a specific implementation, making it hard to change or replace the implementation later.

- The factory pattern decouples the client code from concrete product classes by delegating object creation to a factory, enabling flexibility and easier maintenance.

- **Example**: Instead of calling `new ConcreteProduct()` in the client code, you use `factory.createProduct()`. If the product changes, only the factory needs updating.

2. **Single Responsibility Principle (SRP)**

- SRP states that a class should have only one reason to change. By centralizing the creation logic in a factory, the rest of the program doesn't deal with it directly.

- This reduces redundancy and makes the code easier to maintain since changes to object creation logic happen in one place.

3. **Open/Closed Principle (OCP)**

- OCP states that software entities should be open for extension but closed for modification.

- By using a factory, you can add new product types without modifying the existing client code. This is achieved by extending the factory or introducing new subclasses.

- **Example**: If you introduce a new `ProductD`, you only need to update the factory and not every client using the factory.

Cons

1. **Increased code complexity due to additional subclasses**

- The factory pattern introduces additional layers of abstraction (e.g., factories, interfaces, and subclasses for each product type).

- This complexity might feel unnecessary in simpler applications, especially if only a few product types exist.

- **Best use case**: When there's already an existing hierarchy or when you anticipate needing scalability in creating new products.

2. **Best for existing hierarchies of creator classes**

- The factory pattern shines in systems with well-established base classes or interfaces for products.

- If the hierarchy doesn't exist, you might need to define multiple new classes/interfaces, which can feel over-engineered for simpler needs.

- For a brand-new system with no existing hierarchy, the cost of setting up the pattern may outweigh its benefits unless extensibility is a key goal.

The factory design pattern helps decouple object creation, adheres to key design principles like SRP and OCP, and simplifies maintenance. However, it can introduce unnecessary complexity, especially in smaller or simpler systems without an existing class hierarchy. Use it where scalability and flexibility are priorities, and avoid it for trivial use cases.

Abstract Factory

Abstract Factory is a creational design pattern that lets you produce families of related objects without specifying their concrete classes.

The Abstract Factory Pattern is one of the creational design patterns that provides an interface for creating families of related or dependent objects without specifying their concrete classes and implementation, in simpler terms the Abstract Factory Pattern is a way of organizing how you create groups of things that are related to each other.

Components of Abstract Factory Pattern

To understand abstract factory pattern, we have to understand the components of it and relationships between them.

- **Abstract Factory**:
    - Abstract Factory provides a high-level blueprint that defines rules for creating families of related object without specifying their concrete classes.
    - It provides a way such that concrete factories follow a common interface, providing consistent way to produce related set of objects.
- **Concrete Factories**:
    - Concrete Factories implement the rules specified by the abstract factory. It contain the logic for creating specific instances of objects within a family.

    - Also multiple concrete factories can exist, each produce a distinct family of related objects.

- **Abstract Products**:

    - Abstract Products represents a family of related objects by defining a set of common methods or properties.

    - It acts as an abstract or interface type that all concrete products within a family must follow to and provides a unified way for concrete products to be used interchangeably.

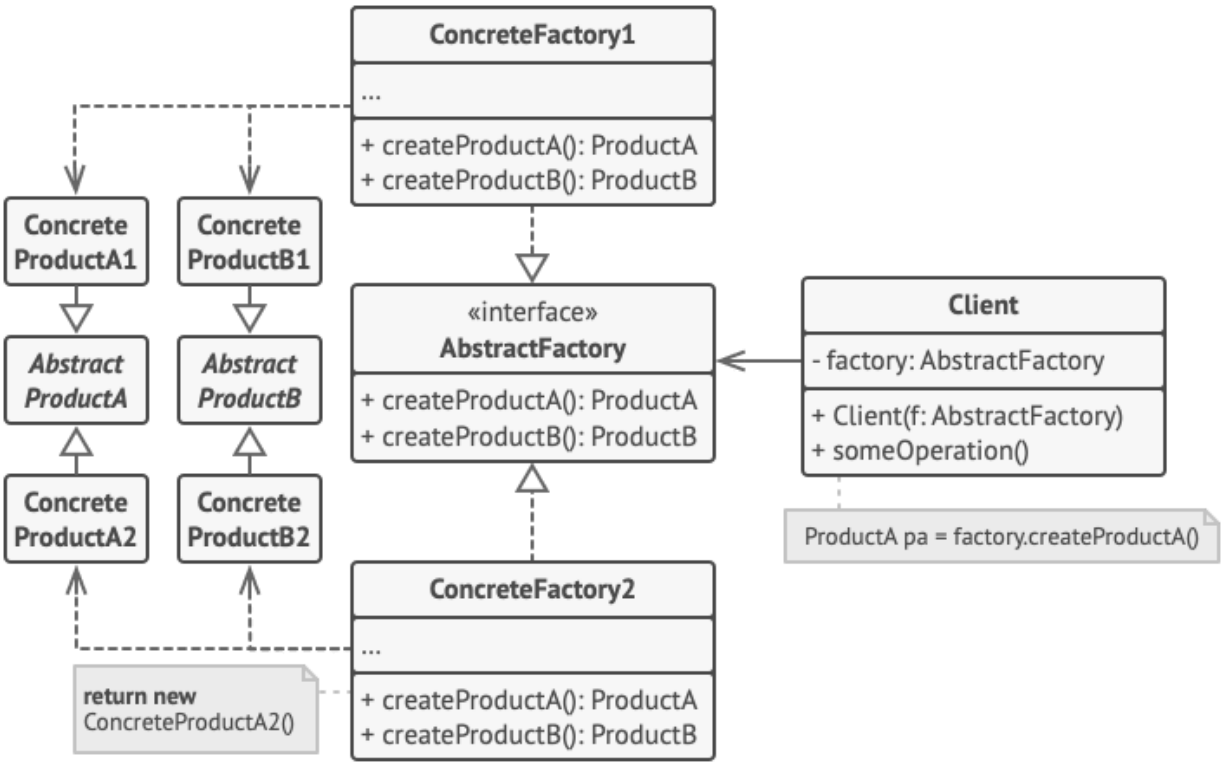
- **Concrete Products**:

    - They are the actual instances of objects created by concrete factories.

    - They implement the methods declared in the abstract products, ensuring consistency within a family and belong to a specific category or family of related objects.

- **Client**:

    - Client utilizes the abstract factory to create families of objects without specifying their concrete types and interacts with objects through abstract interfaces provided by abstract products.



- Abstract Factory pattern is almost same as Factory Pattern and is considered as another layer of abstraction over factory pattern.
- Abstract Factory patterns work around a super-factory which creates other factories.
- At runtime, the abstract factory is coupled with any desired concrete factory which can create objects of the desired type.

Pros

- You can be sure that the products you're getting from a factory are compatible with each other.
- You avoid tight coupling between concrete products and client code.
- *Single Responsibility Principle*. You can extract the product creation code into one place, making the code easier to support.
- *Open/Closed Principle*. You can introduce new variants of products without breaking existing client code.

Cons

- The code may become more complicated than it should be, since a lot of new interfaces and classes are introduced along with the pattern.

Builder

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.

This pattern separates the construction of a complex object from its representation, allowing the same construction process to create different representations.

The Builder pattern suggests that you extract the object construction code out of its own class and move it to separate objects called *builders*. The pattern organizes object construction into a set of steps (`buildWalls`, `buildDoor`, etc.). To create an object, you execute a series of these steps on a builder object. The important part is that you don't need to call all of the steps. You can call only those steps that are necessary for producing a particular configuration of an object.

Components of the Builder Design Pattern

1. Product

The Product is the complex object that the Builder pattern is responsible for constructing.

- It may consist of multiple components or parts, and its structure can vary based on the implementation.
- The Product is typically a class with attributes representing the different parts that the Builder constructs.

2. Builder

The Builder is an interface or an abstract class that declares the construction steps for building a complex object.

- It typically includes methods for constructing individual parts of the product.
- By defining an interface, the Builder allows for the creation of different concrete builders that can produce variations of the product.

3. ConcreteBuilder

ConcreteBuilder classes implement the Builder interface, providing specific implementations for building each part of the product.

- Each ConcreteBuilder is customized to create a specific variation of the product.
- It keeps track of the product being constructed and provides methods for setting or constructing each part.

4. Director

The Director is responsible for managing the construction process of the complex object.

- It collaborates with a Builder, but it doesn't know the specific details about how each part of the object is constructed.
- It provides a high-level interface for constructing the product and managing the steps needed to create the complex object.

5. Client

The Client is the code that initiates the construction of the complex object.

- It creates a Builder object and passes it to the Director to initiate the construction process.
- The Client may retrieve the final product from the Builder after construction is complete.

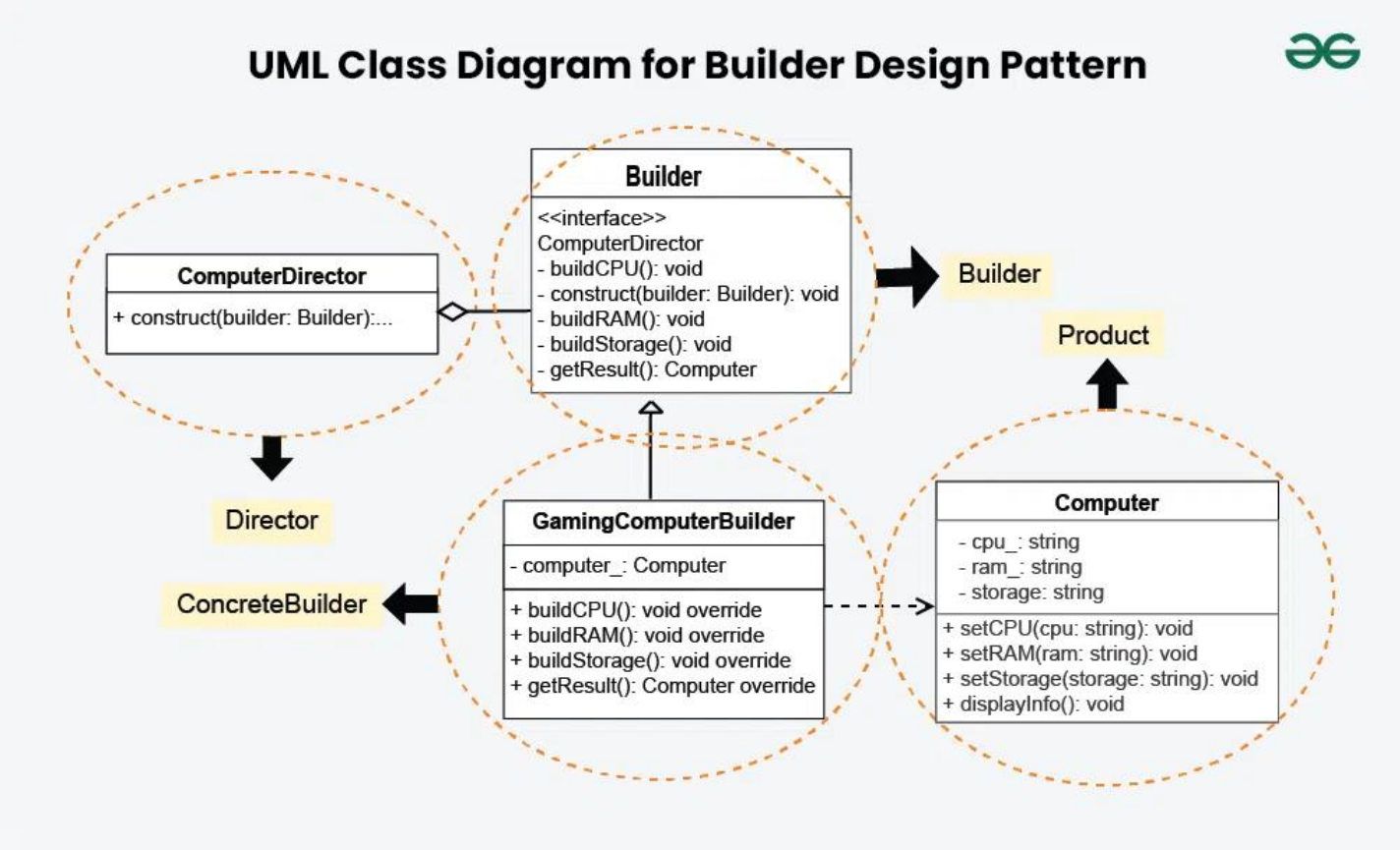
Steps to implement Builder Design Pattern

Below are the steps to implement Builder Design Pattern:

1. **Create the Product Class**: Define the object (product) that will be built. This class contains all the fields that make up the object.
2. **Create the Builder Class**: This class will have methods to set the different parts of the product. Each method returns the builder itself to allow method chaining.
3. **Add a Build Method**: In the builder class, add a method called build() (or similar) that assembles the product and returns the final object.

4. **Use the Director (Optional)**: If needed, you can create a director class to control the building process and decide the order in which parts are constructed.

5. **Client Uses the Builder**: The client will use the builder to set the desired parts step by step and call the build() method to get the final product.



Pros

- You can construct objects step-by-step, defer construction steps or run steps recursively.
- You can reuse the same construction code when building various representations of products.
- *Single Responsibility Principle*. You can isolate complex construction code from the business logic of the product.

Cons

- The overall complexity of the code increases since the pattern requires creating multiple new classes.

Prototype

Intent

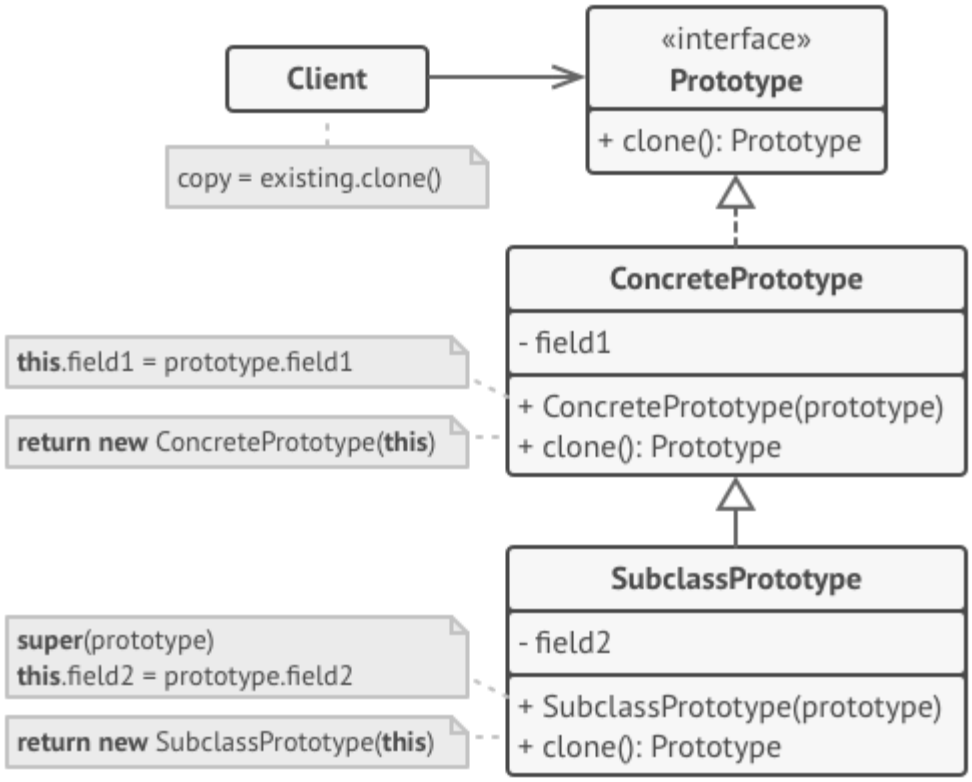
Prototype is a creational design pattern that lets you copy existing objects without making your code dependent on their classes. Prototype allows us to hide the complexity of making new instances from the client. The existing object acts as a prototype and contains the state of the object.

Components of Prototype Design Pattern

The Prototype Design Pattern's components include the prototype interface or abstract class, concrete prototypes and the client code, and the clone method specifying cloning behavior. These components work together to enable the creation of new objects by copying existing ones.

- **Prototype Interface or Abstract Class**
    - This defines the method for cloning objects and sets a standard that all concrete prototypes must follow. Its main purpose is to serve as a blueprint for creating new objects by outlining the cloning contract.
    - It includes a clone method that concrete prototypes will implement to create copies of themselves.
- **Concrete Prototype**
    - This class implements the prototype interface or extends the abstract class. It represents a specific type of object that can be cloned.
    - The Concrete Prototype details how the cloning process should work for instances of that class and provides the specific logic for the clone method.
- **Client**
    - The Client is the code or module that requests new object creation by interacting with the prototype.
    - It initiates the cloning process without needing to know the specifics of the concrete classes involved.

- **Clone Method**
    - This method is declared in the prototype interface or abstract class and outlines how an object should be copied.
    - Concrete prototypes implement this method to define their specific cloning behavior, detailing how to duplicate the object's internal state to create a new, independent instance.



1. The **Prototype Registry** provides an easy way to access frequently-used prototypes. It stores a set of pre-built objects that are ready to be copied. The simplest prototype registry is a `name → prototype` hash map. However, if you need better search criteria than a simple name, you can build a much more robust version of the registry.

Pros

- You can clone objects without coupling to their concrete classes.
- You can get rid of repeated initialization code in favor of cloning pre-built prototypes.
- You can produce complex objects more conveniently.

- You get an alternative to inheritance when dealing with configuration presets for complex objects.

Cons

- Cloning complex objects that have circular references might be very tricky.

Structural Design Pattern

Adapter

Intent

Adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate. Because of incompatible interfaces, it serves as a bridge between two classes that would not otherwise be able to communicate. The adapter approach is very helpful when attempting to incorporate third-party libraries or legacy code into a new system.

Components of Adapter Design Pattern

Below are the components of adapter design pattern:

- **Target Interface**: Defines the interface expected by the client. It represents the set of operations that the client code can use. It's the common interface that the client code interacts with.

- **Adaptee**: The existing class or system with an incompatible interface that needs to be integrated into the new system. It's the class or system that the client code cannot directly use due to interface mismatches.

- **Adapter**: A class that implements the target interface and internally uses an instance of the adaptee to make it compatible with the target interface. It acts as a bridge, adapting the interface of the adaptee to match the target interface.

- **Client**: The code that uses the target interface to interact with objects. It remains unaware of the specific implementation details of the adaptee and the adapter. It's the code that benefits from the integration of the adaptee into the system through the adapter.

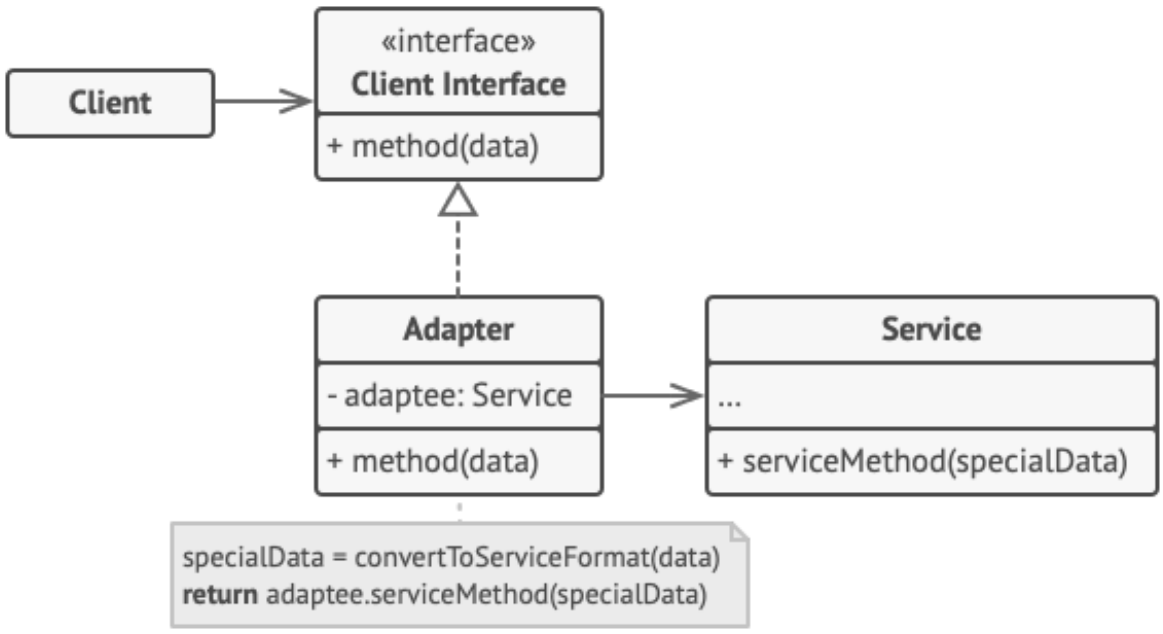
Different implementations of Adapter Design Pattern

1. Class Adapter (Inheritance-based)

- In this approach, the adapter class inherits from both the target interface (the one the client expects) and the adaptee (the existing class needing adaptation).
- Programming languages that allow multiple inheritance, like C++, are more likely to use this technique.
- However, in languages like Java and C#, which do not support multiple inheritance, this approach is less frequently used.

2. Object Adapter (Composition-based)

- The object adapter employs composition instead of inheritance. In this implementation, the adapter holds an instance of the adaptee and implements the target interface.
- This approach is more flexible as it allows a single adapter to work with multiple adaptees and does not require the complexities of inheritance.
- The object adapter is widely used in languages like Java and C#.



Pros

- *Single Responsibility Principle*. You can separate the interface or data conversion code from the primary business logic of the program.
- *Open/Closed Principle*. You can introduce new types of adapters into the program without breaking the existing client code, as long as they work with the adapters through the client interface.

Cons

- The overall complexity of the code increases because you need to introduce a set of new interfaces and classes. Sometimes it's simpler just to change the service class so that it matches the rest of your code.

Facade

Intent

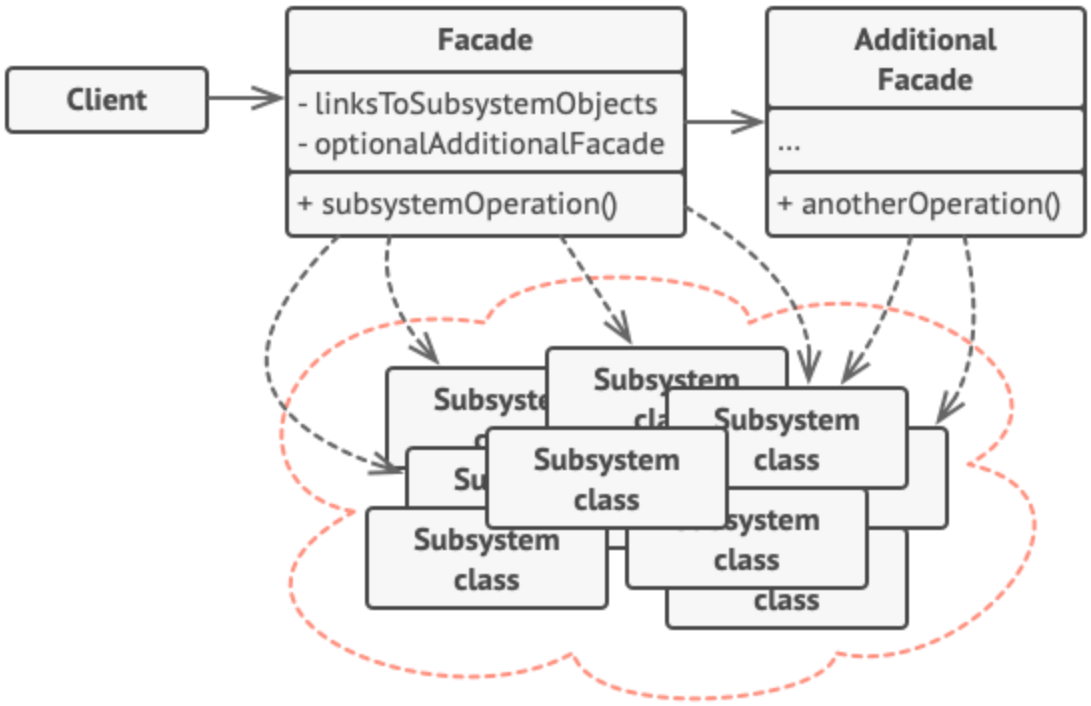
Facade is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes.

Facade Method Design Pattern provides a unified interface to a set of interfaces in a subsystem. Facade defines a high-level interface that makes the subsystem easier to use.

A facade is a class that provides a simple interface to a complex subsystem which contains lots of moving parts. A facade might provide limited functionality in comparison to working with the subsystem directly. However, it includes only those features that clients really care about.

Having a facade is handy when you need to integrate your app with a sophisticated library that has dozens of features, but you just need a tiny bit of its functionality.

For instance, an app that uploads short funny videos with cats to social media could potentially use a professional video conversion library. However, all that it really needs is a class with the single method `encode(filename, format)`. After creating such a class and connecting it with the video conversion library, you'll have your first facade.



1. The **Facade** provides convenient access to a particular part of the subsystem's functionality. It knows where to direct the client's request and how to operate all the moving parts.
2. An **Additional Facade** class can be created to prevent polluting a single facade with unrelated features that might make it yet another complex structure. Additional facades can be used by both clients and other facades.
3. The **Complex Subsystem** consists of dozens of various objects. To make them all do something meaningful, you have to dive deep into the subsystem's implementation details, such as initializing objects in the correct order and supplying them with data in the proper format. Subsystem classes aren't aware of the facade's existence. They operate within the system and work with each other directly.

Declare and implement this interface in a new facade class. The facade should redirect the calls from the client code to appropriate objects of the subsystem. The facade should be responsible for initializing the subsystem and managing its further life cycle unless the client code already does this.

4. The **Client** uses the facade instead of calling the subsystem objects directly.

Advantages of Facade Method Design Pattern

- **Simplified Interface:**
    - Simplifies the use and understanding of a complex system by offering a clear and concise interface.
    - Hides the internal details of the system, reducing cognitive load for clients.
- **Reduced Coupling:**
    - Clients become less reliant on the internal workings of the underlying system when they are disconnected from it.
    - Encourages the reusability and modularity of code components.
    - Allows for the independent testing and development of various system components.
- **Encapsulation:**
    - Encapsulates the complex interactions within a subsystem, protecting clients from changes in its implementation.
    - Allows for changes to the subsystem without affecting clients, as long as the facade interface remains stable.
- **Improved Maintainability:**
    - Easier to change or extend the underlying system without affecting clients, as long as the facade interface remains consistent.
    - Allows for refactoring and optimization of the subsystem without impacting client code.

Disadvantages of Facade Method Design Pattern

- **Increased Complexity:**
    - Adding the facade layer in the system increases the level of abstraction.
    - Because of this, the code may be more difficult to understand and debug
- **Reduced Flexibility:**

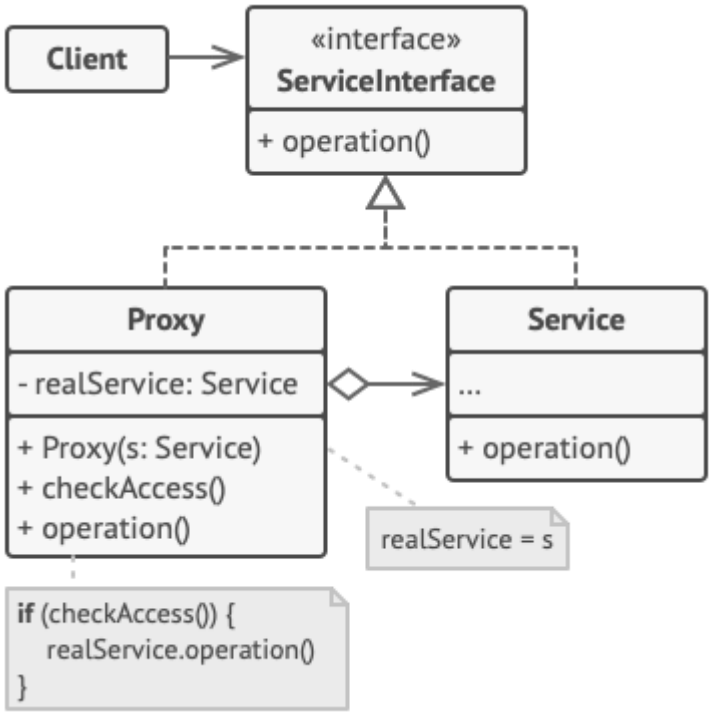
    - The facade acts as a single point of access to the underlying system.
    - This can limit the flexibility for clients who need to bypass the facade or access specific functionalities hidden within the subsystem.
- **Overengineering:**
    - Applying the facade pattern to very simple systems can be overkill, adding unnecessary complexity where it's not needed.
    - Consider the cost-benefit trade-off before implementing a facade for every situation.
- **Potential Performance Overhead:**
    - Adding an extra layer of indirection through the facade can introduce a slight performance overhead, especially for frequently used operations.
    - This may not be significant for most applications, but it's worth considering in performance-critical scenarios.

Proxy

Intent

Proxy is a structural design pattern that lets you provide a substitute or placeholder for another object. The Proxy Design Pattern a structural design pattern is a way to use a placeholder object to control access to another object. Instead of interacting directly with the main object, the client talks to the proxy, which then manages the interaction. This is useful for things like controlling access, delaying object creation until it's needed (lazy initialization), logging, or adding security checks.

The Proxy Design Pattern is a design pattern in which the client and the actual object are connected by a proxy object. The client communicates with the proxy, which manages access to the real object, rather than the real object directly. Before sending the request to the real object, the proxy can take care of additional tasks like caching, security, logging, and lazy loading.



Components of Proxy Design Pattern

1. Subject

The Subject is an interface or an abstract class that defines the common interface shared by the RealSubject and Proxy classes. It declares the methods that the Proxy uses to control access to the RealSubject.

- Declares the common interface for both RealSubject and Proxy.
- Usually includes the methods that the client code can invoke on the RealSubject and the Proxy.

2. RealSubject

The RealSubject is the actual object that the Proxy represents. It contains the real implementation of the business logic or the resource that the client code wants to access.

- It Implements the operations declared by the Subject interface.
- Represents the real resource or object that the Proxy controls access to.

3. Proxy

The Proxy acts as a surrogate or placeholder for the RealSubject. It controls access to the real object and may provide additional functionality such as lazy loading, access control, or logging.

- Implements the same interface as the RealSubject (Subject).
- Maintains a reference to the RealSubject.
- Controls access to the RealSubject, adding additional logic if necessary.

Pros

- You can control the service object without clients knowing about it.
- You can manage the lifecycle of the service object when clients don't care about it.
- The proxy works even if the service object isn't ready or is not available.
- *Open/Closed Principle*. You can introduce new proxies without changing the service or clients.

Cons

- The code may become more complicated since you need to introduce a lot of new classes.
- The response from the service might get delayed.