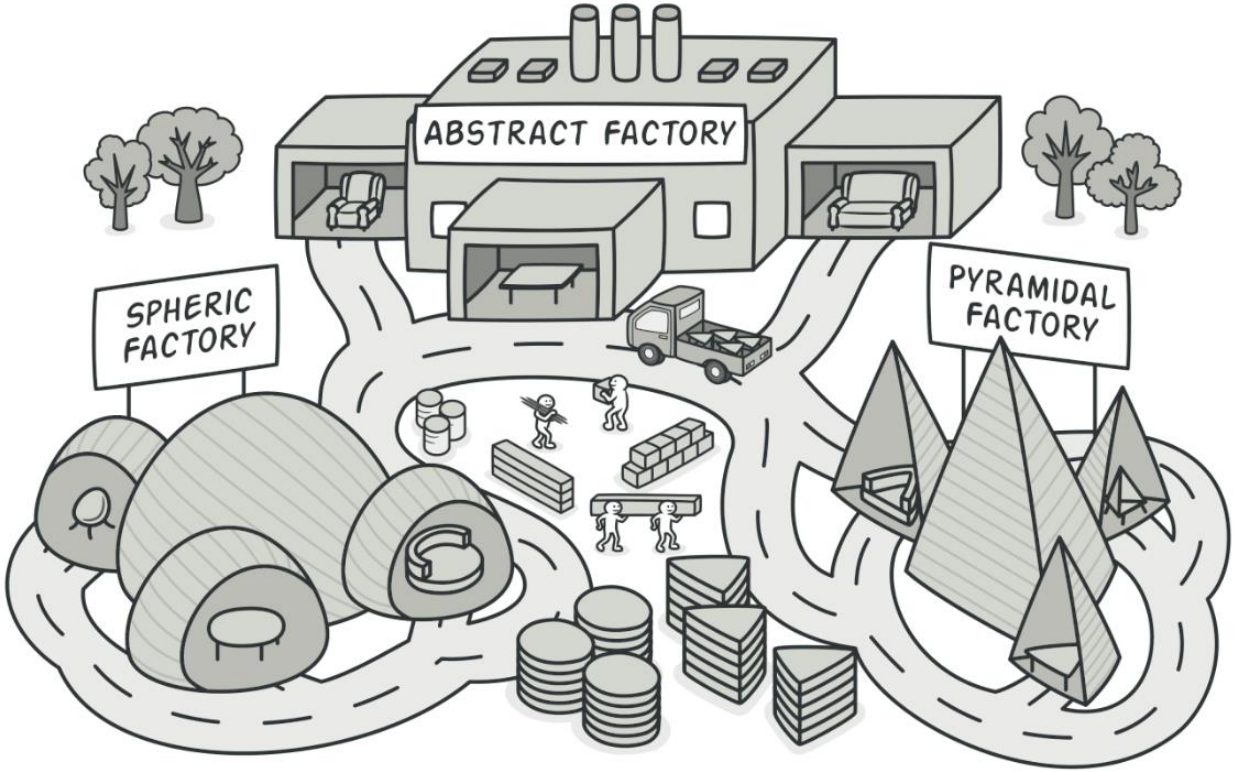
CSci 3081W: Program Design and Development

Lecture 9 – Abstract Factory Pattern
Façade Pattern

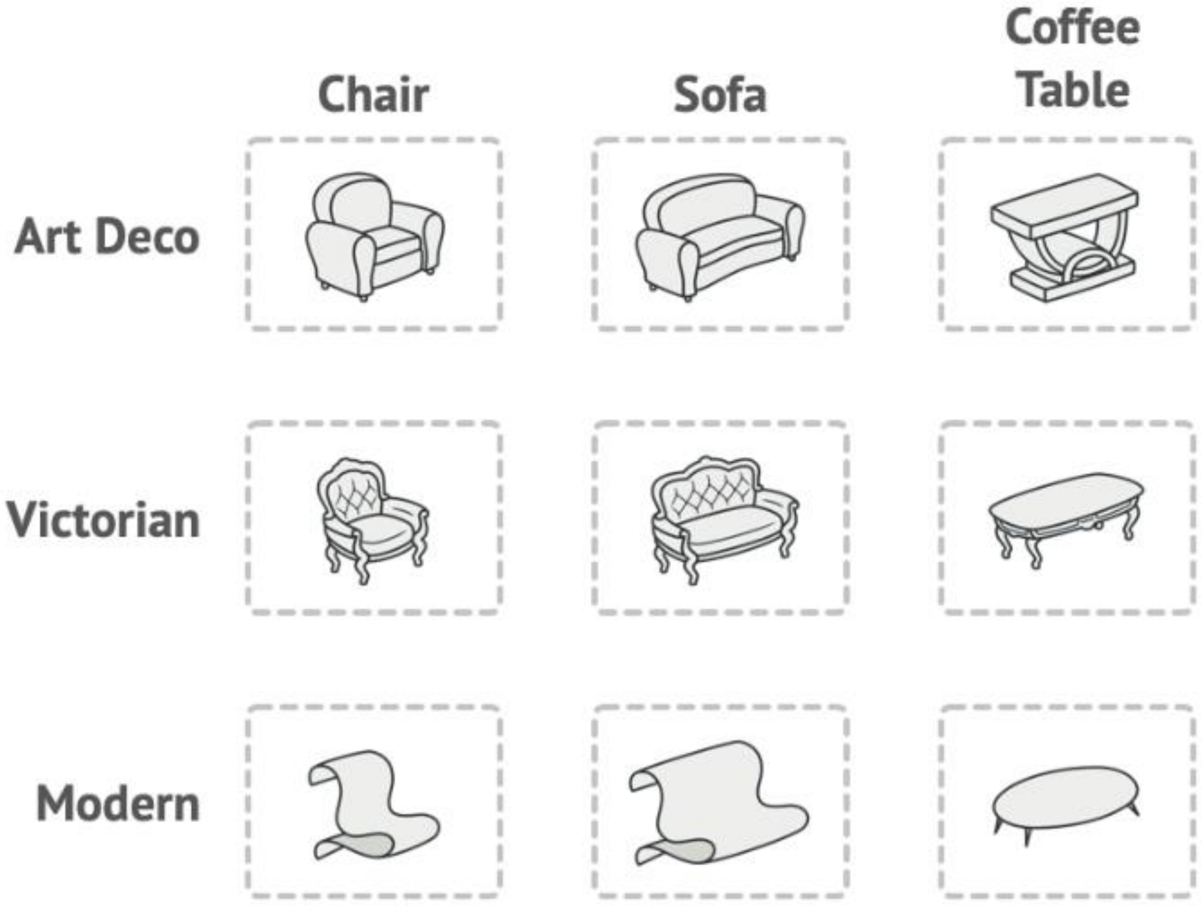
Abstract Factory Pattern

It lets you produce families of related objects without specifying their concrete classes



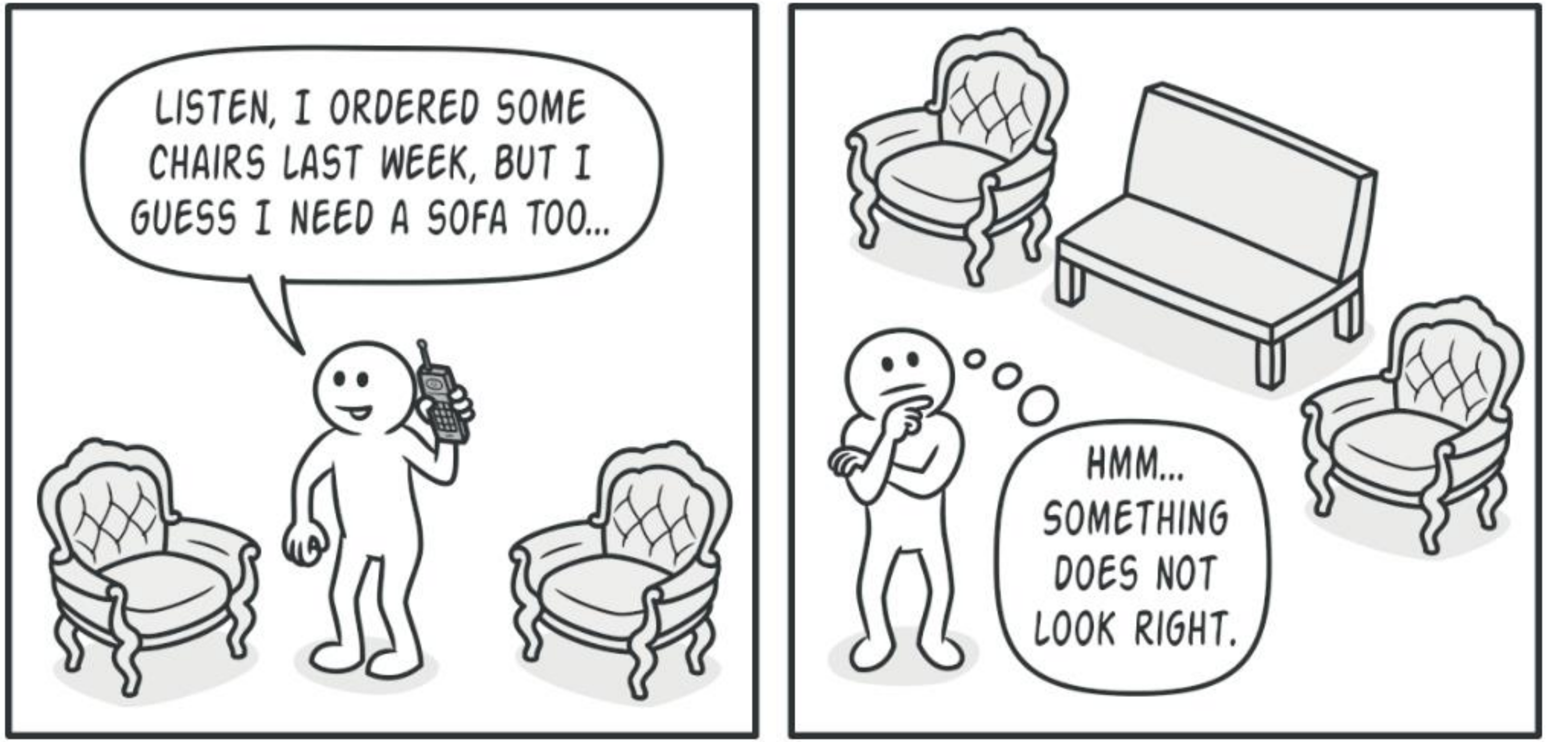
Problem

You have moved on from simulating Road and Ship Logistics to simulating a furniture store



Product families and their variants.

Problem


LISTEN, I ORDERED SOME CHAIRS LAST WEEK, BUT I GUESS I NEED A SOFA TOO...
HMM... SOMETHING DOES NOT LOOK RIGHT.

A Modern-style sofa doesn't match Victorian-style chairs.

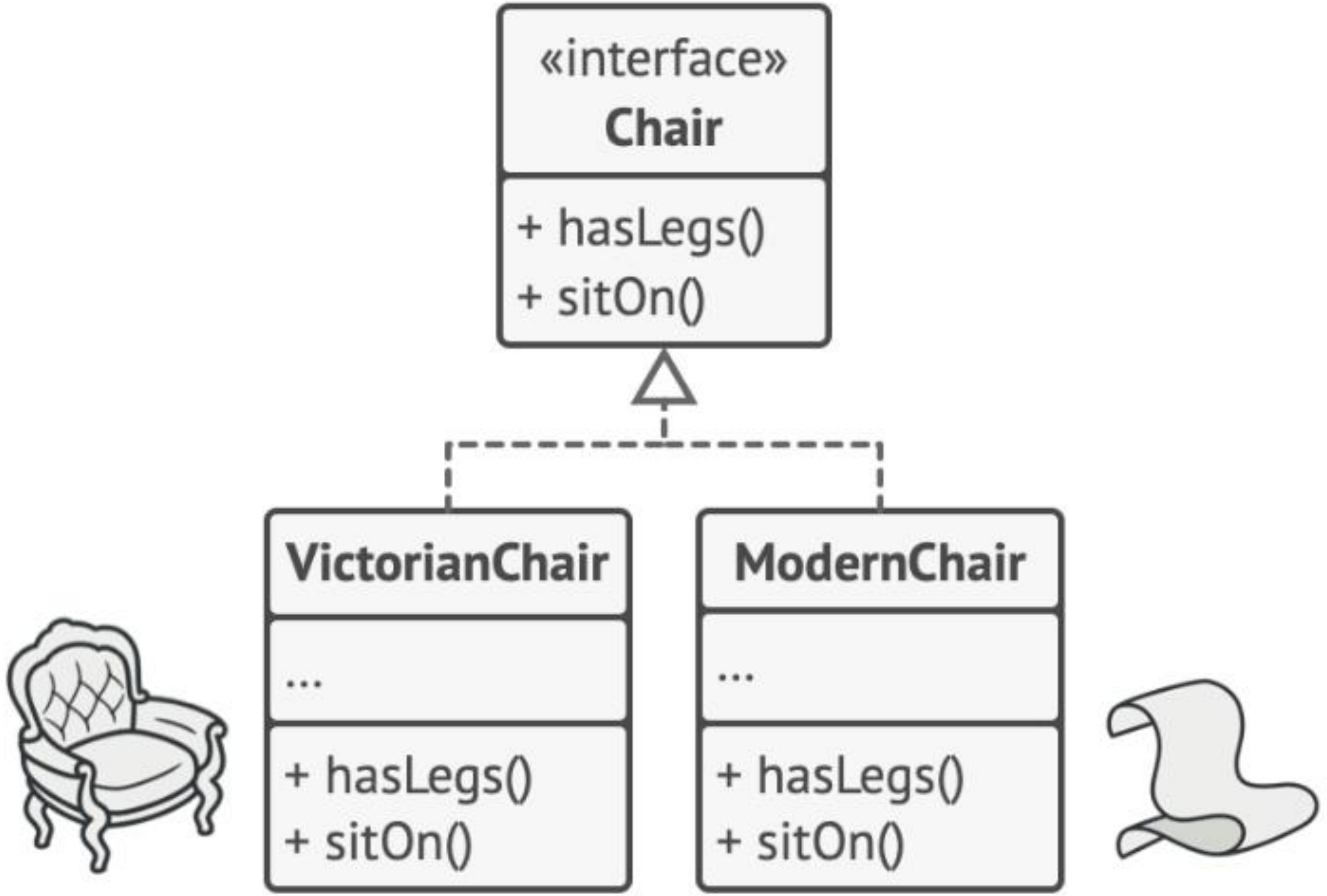
Problem

O from SOLID:

Don’t ever want to change existing code when adding new pieces of furniture

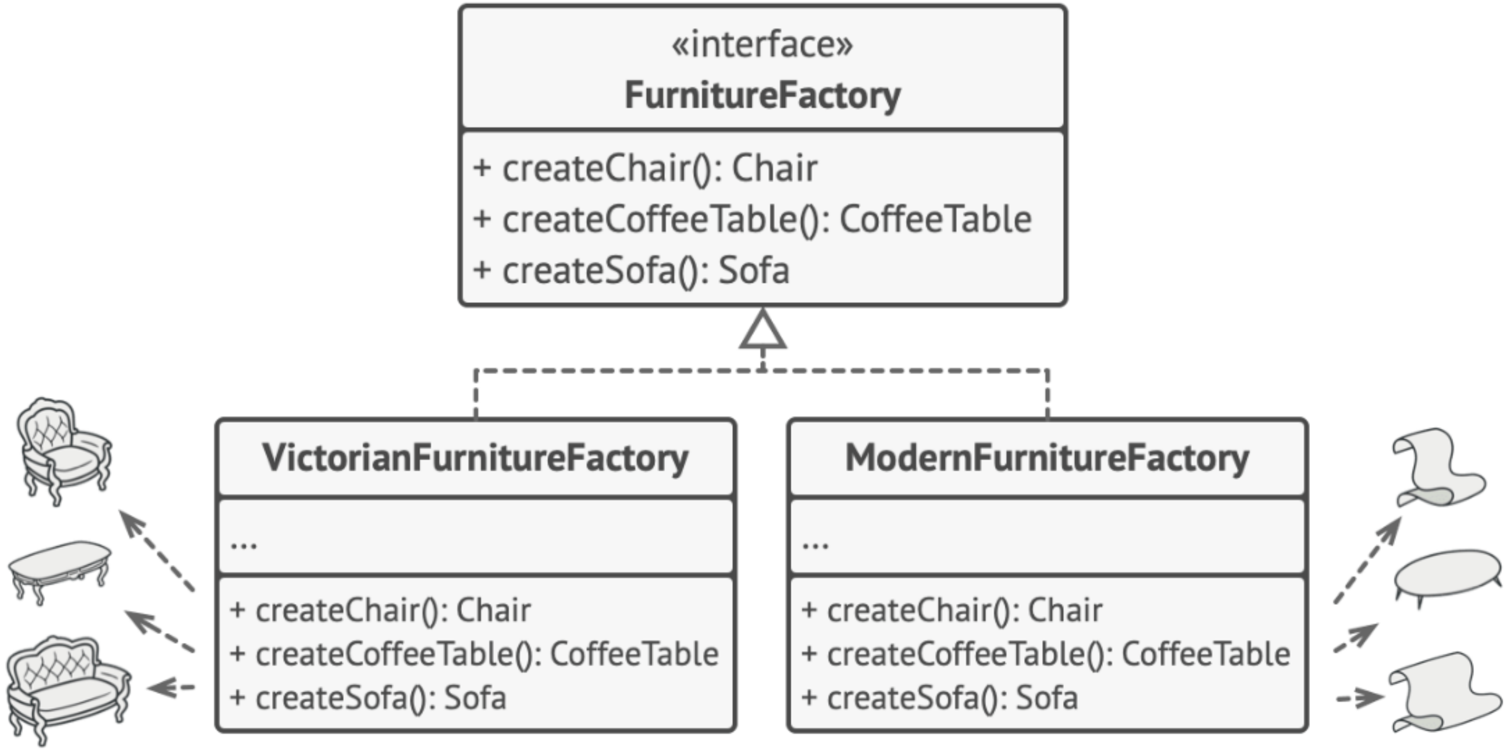
Imagine how often you’d have to update the core code every time a new piece of furniture releases

Solution



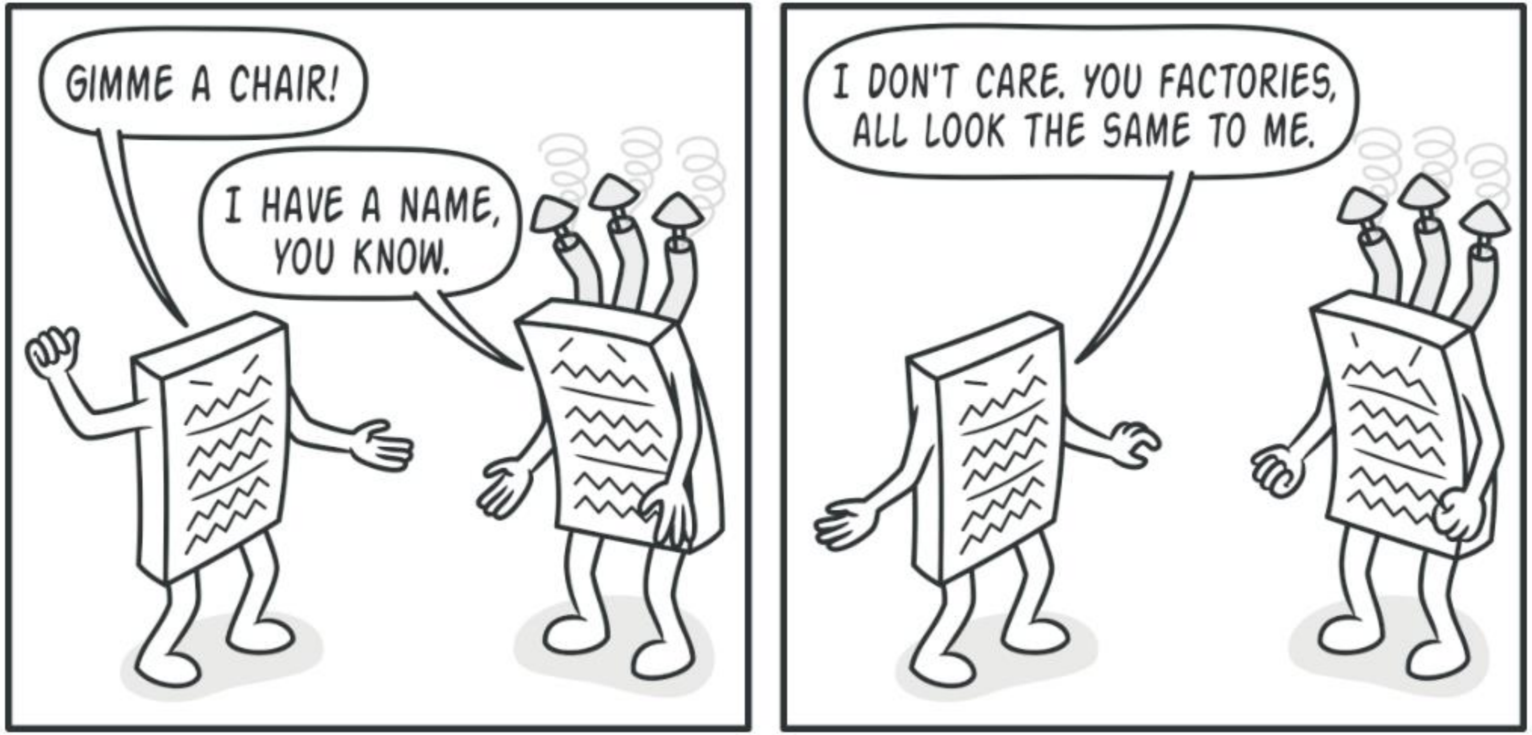
All variants of the same object must be moved to a single class hierarchy.

Solution



Each concrete factory corresponds to a specific product variant.

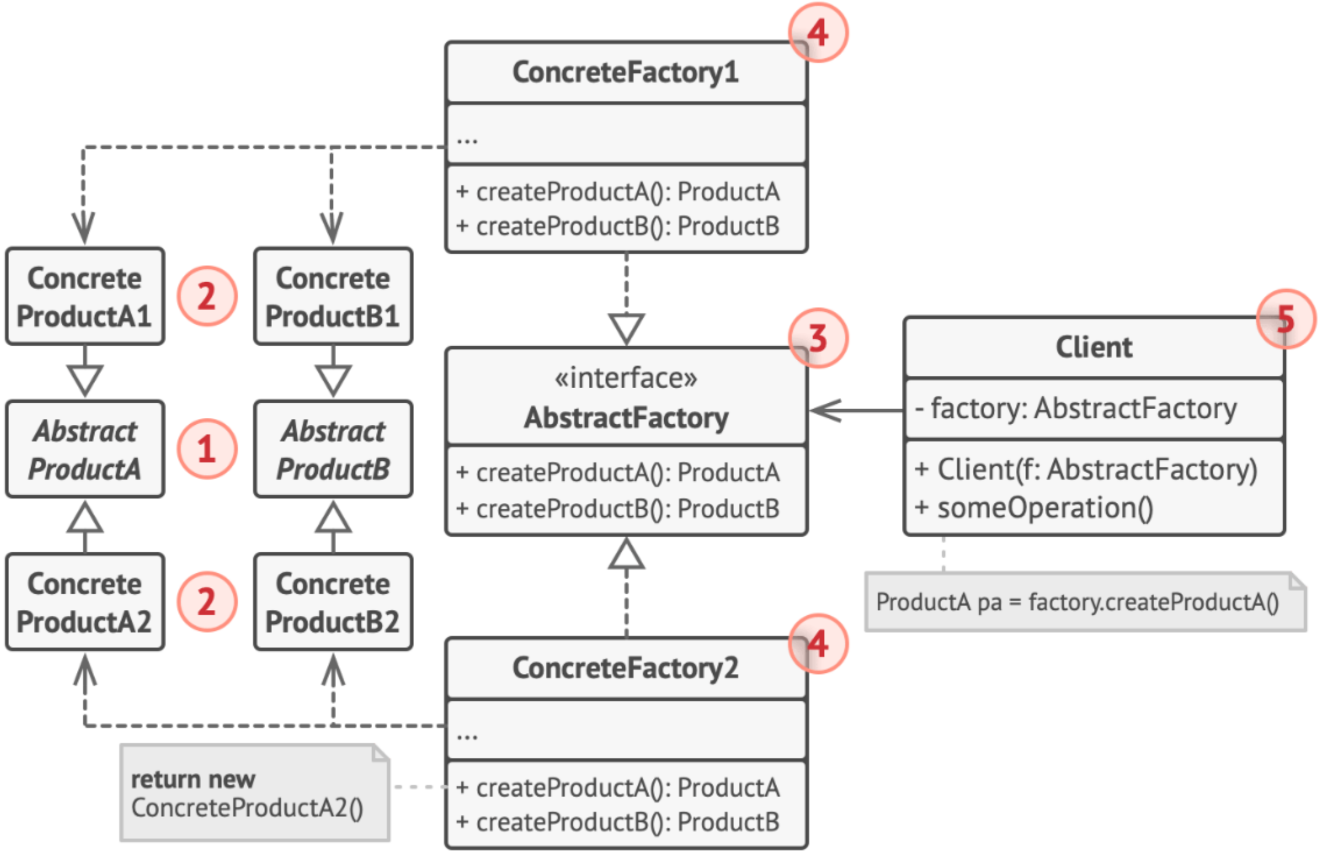
Solution



The client shouldn't care about the concrete class of the factory it works with.

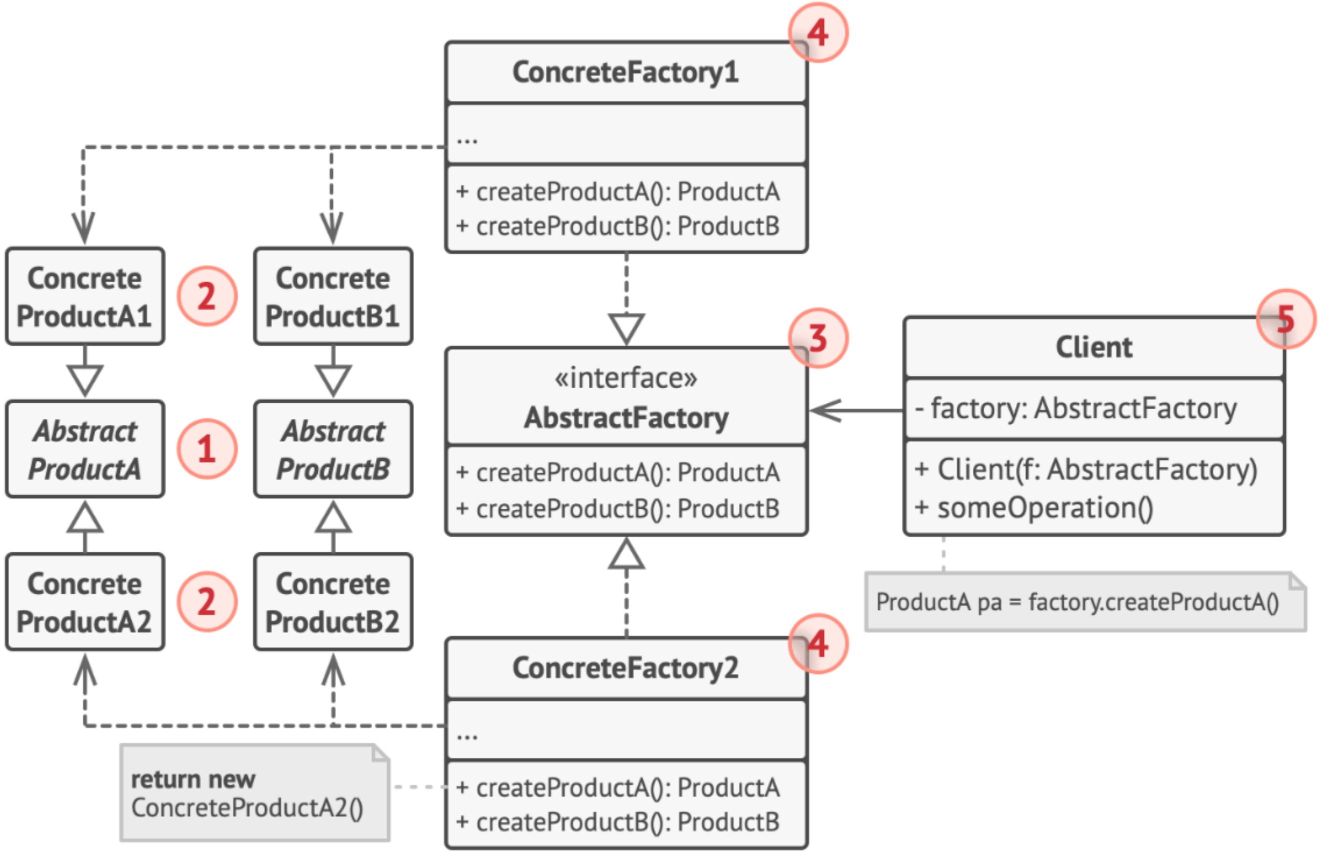
UML Class Diagram

1. **Abstract Products** declare interfaces for a set of distinct but related products which make up a product family.


ConcreteFactory1
...
+ createProductA(): ProductA
+ createProductB(): ProductB
4
Concrete ProductA1
2
Concrete ProductB1
Abstract ProductA
1
Abstract ProductB
Concrete ProductA2
2
Concrete ProductB2
3
«interface» AbstractFactory
+ createProductA(): ProductA
+ createProductB(): ProductB
5
Client
- factory: AbstractFactory
+ Client(f: AbstractFactory)
+ someOperation()
ProductA pa = factory.createProductA()
4
ConcreteFactory2
...
+ createProductA(): ProductA
+ createProductB(): ProductB
return new ConcreteProductA2()

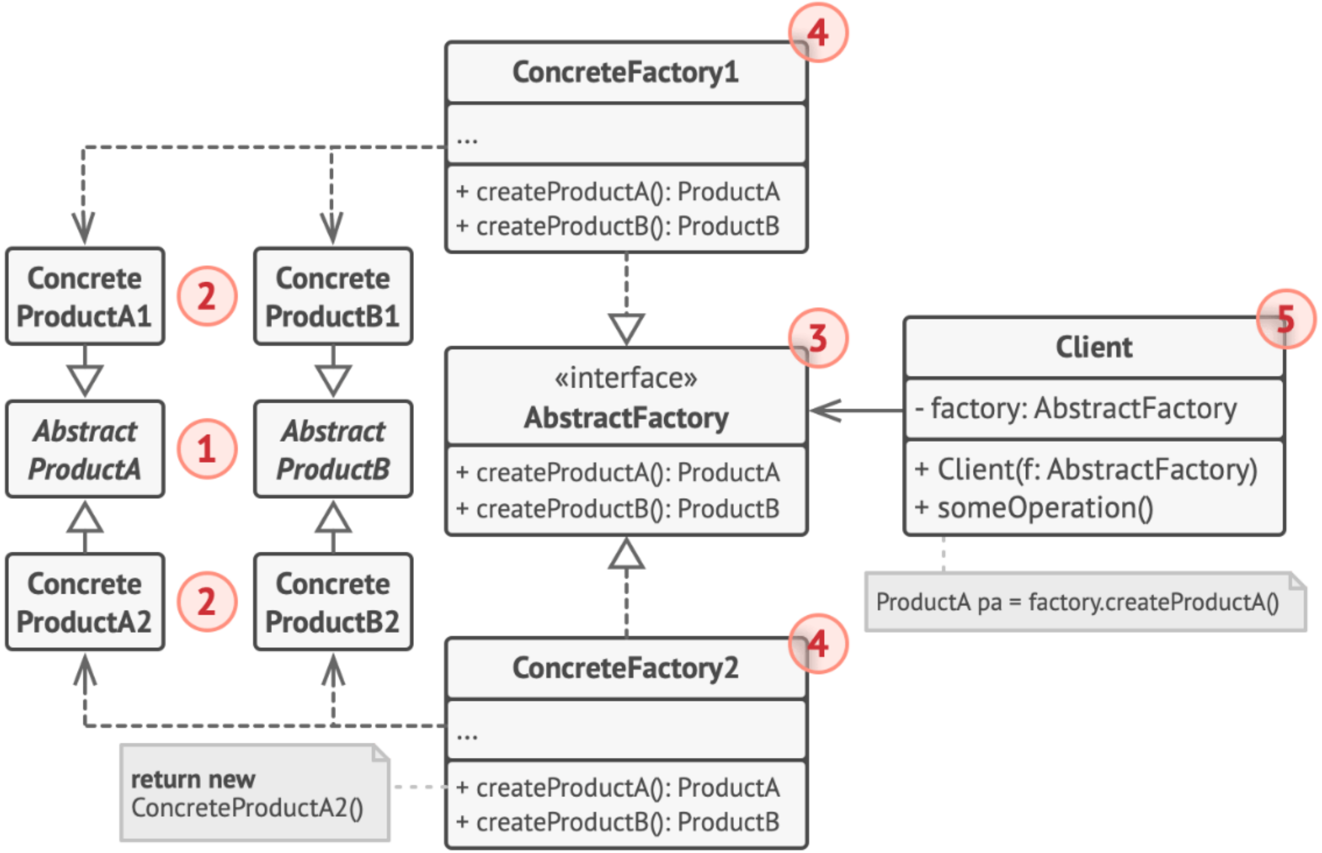
UML Class Diagram

2. **Concrete Products** are various implementations of abstract products, grouped by variants. Each abstract product (chair/sofa) must be implemented in all given variants (Victorian/Modern).


4
ConcreteFactory1
...
+ createProductA(): ProductA
+ createProductB(): ProductB
Concrete ProductA1
2
Concrete ProductB1
Abstract ProductA
1
Abstract ProductB
Concrete ProductA2
2
Concrete ProductB2
3
«interface» AbstractFactory
+ createProductA(): ProductA
+ createProductB(): ProductB
5
Client
- factory: AbstractFactory
+ Client(f: AbstractFactory)
+ someOperation()
ProductA pa = factory.createProductA()
4
ConcreteFactory2
...
+ createProductA(): ProductA
+ createProductB(): ProductB
return new ConcreteProductA2()

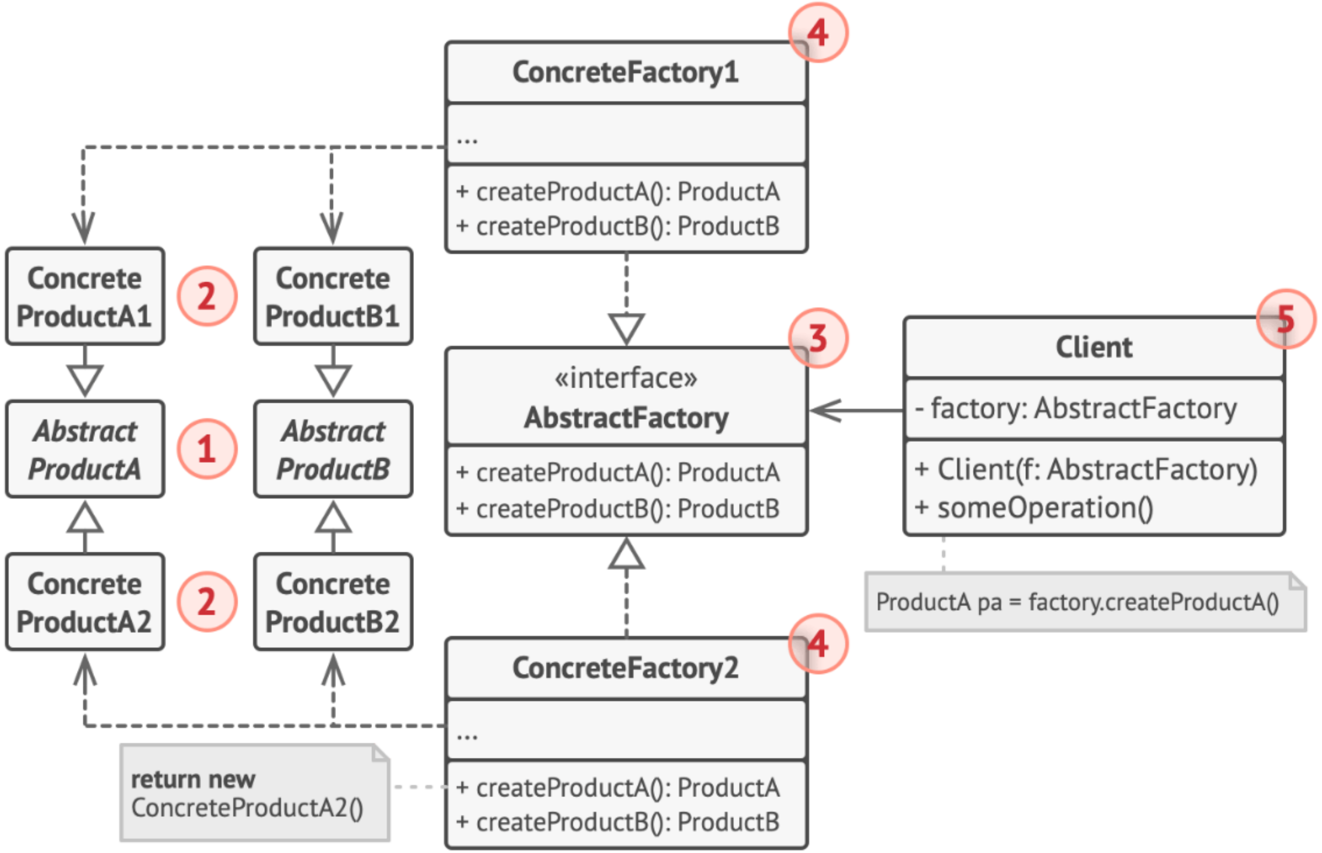
UML Class Diagram

3. The **Abstract Factory** interface declares a set of methods for creating each of the abstract products.


ConcreteFactory1
...
+ createProductA(): ProductA
+ createProductB(): ProductB
4
Concrete ProductA1
2
Concrete ProductB1
Abstract ProductA
1
Abstract ProductB
Concrete ProductA2
2
Concrete ProductB2
«interface» AbstractFactory
+ createProductA(): ProductA
+ createProductB(): ProductB
3
Client
- factory: AbstractFactory
+ Client(f: AbstractFactory)
+ someOperation()
5
ProductA pa = factory.createProductA()
ConcreteFactory2
...
+ createProductA(): ProductA
+ createProductB(): ProductB
4
return new ConcreteProductA2()

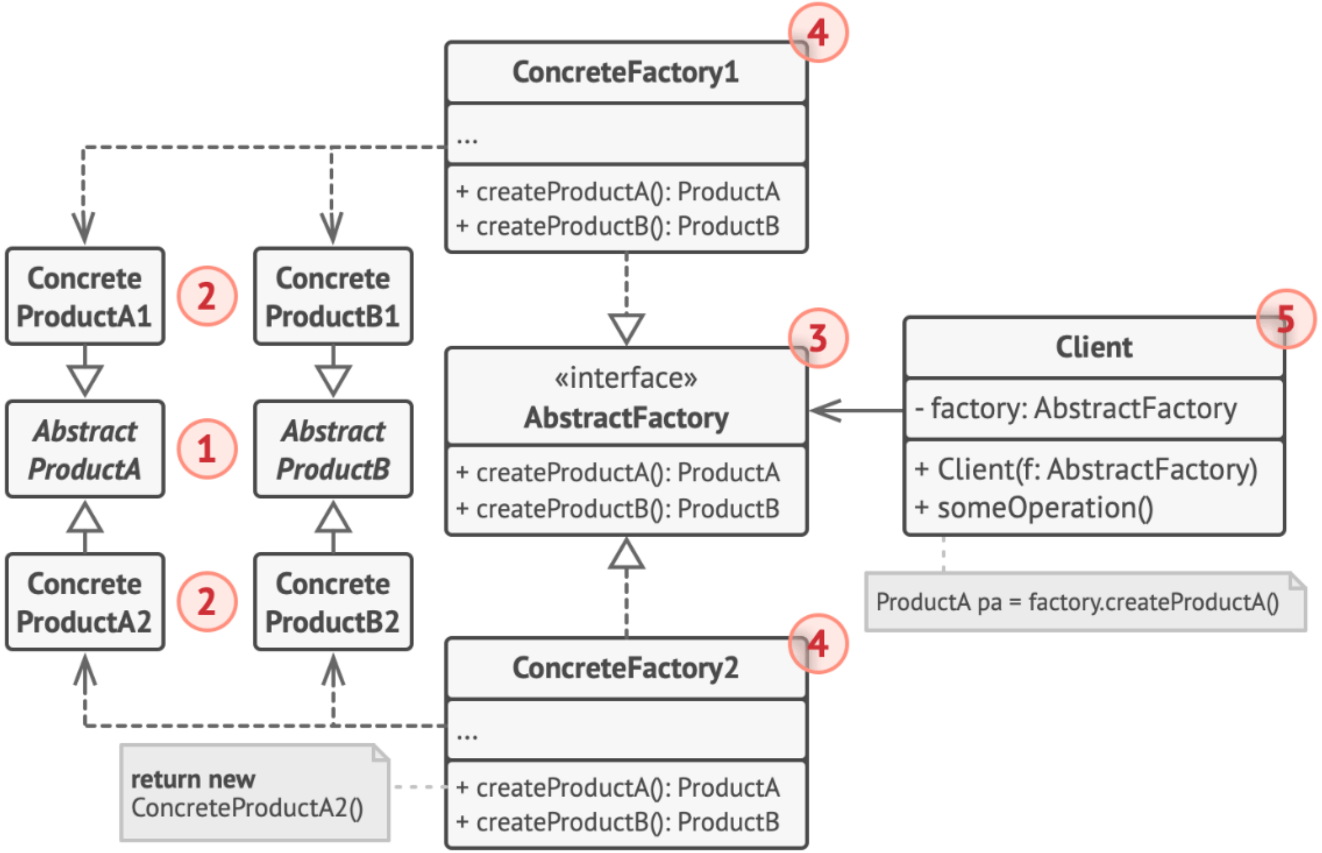
UML Class Diagram

4. **Concrete Factories** implement creation methods of the abstract factory. Each concrete factory corresponds to a specific variant of products and creates only those product variants.

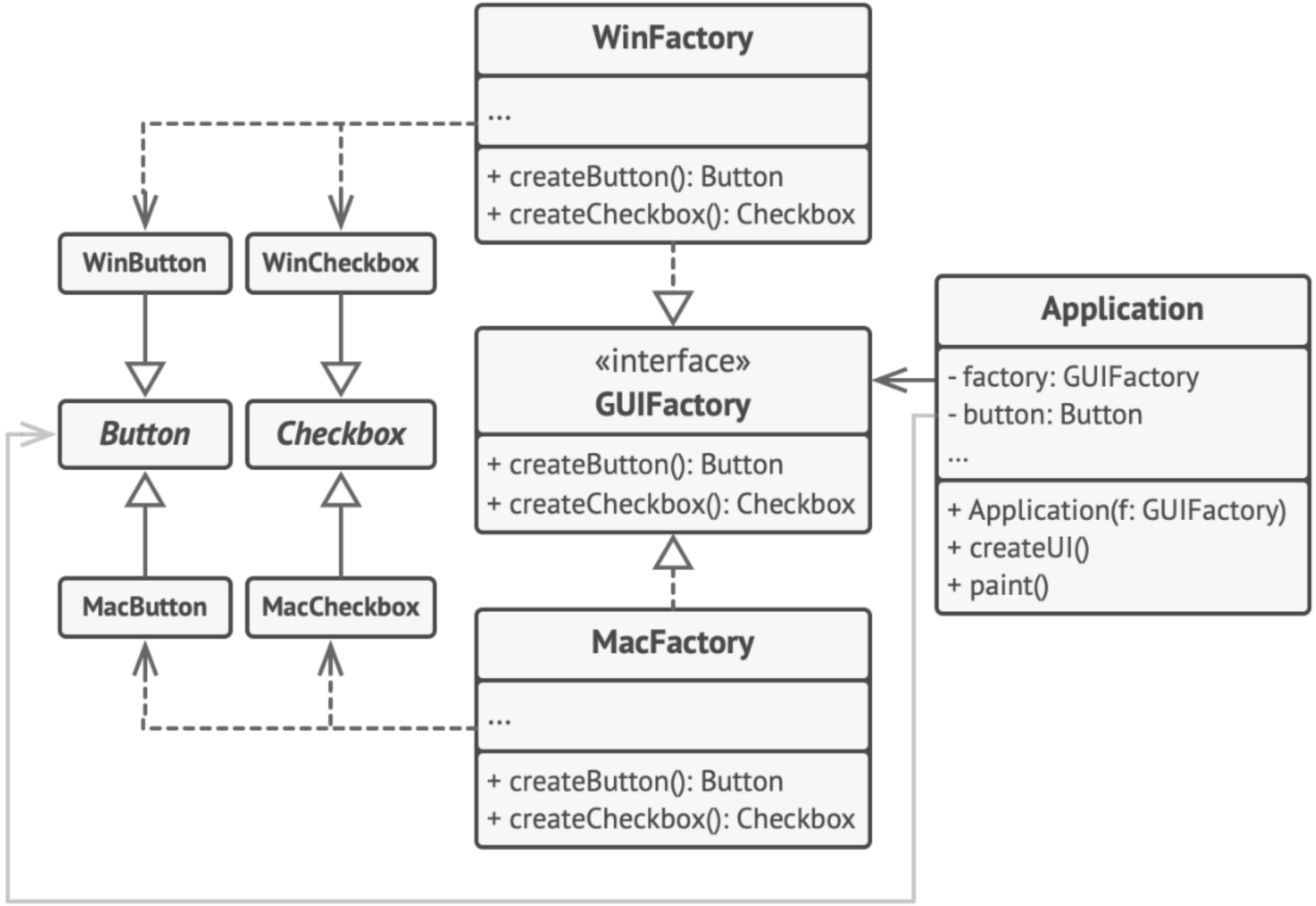

4
ConcreteFactory1
...
+ createProductA(): ProductA
+ createProductB(): ProductB
Concrete ProductA1
2
Concrete ProductB1
Abstract ProductA
1
Abstract ProductB
Concrete ProductA2
2
Concrete ProductB2
3
«interface» AbstractFactory
+ createProductA(): ProductA
+ createProductB(): ProductB
5
Client
- factory: AbstractFactory
+ Client(f: AbstractFactory)
+ someOperation()
ProductA pa = factory.createProductA()
4
ConcreteFactory2
...
+ createProductA(): ProductA
+ createProductB(): ProductB
return new ConcreteProductA2()

UML Class Diagram

5. The **Client** can work with any concrete factory/product variant, as long as it communicates with their objects via abstract interfaces.


ConcreteFactory1
...
+ createProductA(): ProductA
+ createProductB(): ProductB
4
Concrete ProductA1
2
Concrete ProductB1
Abstract ProductA
1
Abstract ProductB
Concrete ProductA2
2
Concrete ProductB2
3
«interface» AbstractFactory
+ createProductA(): ProductA
+ createProductB(): ProductB
Client
5
- factory: AbstractFactory
+ Client(f: AbstractFactory)
+ someOperation()
ProductA pa = factory.createProductA()
ConcreteFactory2
4
...
+ createProductA(): ProductA
+ createProductB(): ProductB
return new ConcreteProductA2()

Another Example



The cross-platform UI classes example.

Applicability

Use when your code needs to work with various families of related products – but you don’t want it to depend on the concrete classes of those products (allows future extensibility)

The big 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.

S from SOLID
O from SOLID

Relation between **Abstract Factory** and **Factory Method**

Many designs start by implementing the Factory Method

(less complicated and more customizable via derived classes)

then evolve toward Abstract Factory

(more flexible, but more complicated)

Relation between **Abstract Factory** and **Factory Method**

Abstract Factory classes are often based on a set of Factory Methods

Abstract Factories can be implemented as a Singleton (breaks the **S** of SOLID though)

Walkthrough and Demo of Abstract Factory code

Uploaded on Canvas

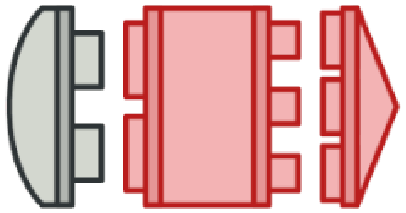
Find it in the course schedule for today

`abstract_factory.cpp`

Structural Design Patterns

Structural design patterns explain how to assemble objects and classes into larger structures, while keeping these structures flexible and efficient.

Structural Design Patterns

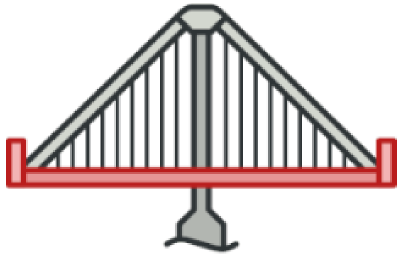


Adapter

Allows objects with incompatible interfaces to collaborate.

We are not going over this one

Structural Design Patterns

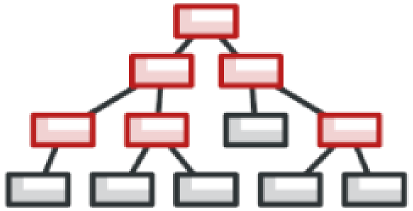


Bridge

Lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.

We are not going over this one

Structural Design Patterns

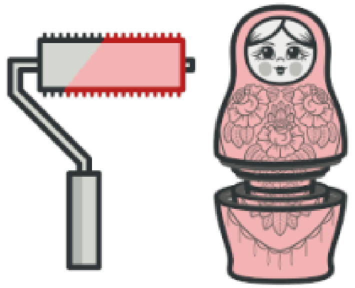


Composite

Lets you compose objects into tree structures and then work with these structures as if they were individual objects.

This one you will combine with a Factory to make a Composite Factory in Lab 7.

Structural Design Patterns



Decorator

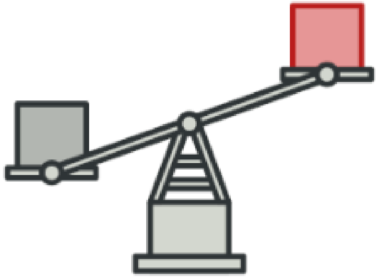
Lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

Structural Design Patterns

Facade

Provides a simplified interface to a library, a framework, or any other complex set of classes.

Structural Design Patterns

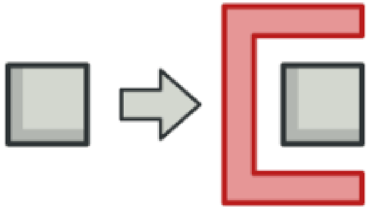


Flyweight

Lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object.

We are not going over this one

Structural Design Patterns



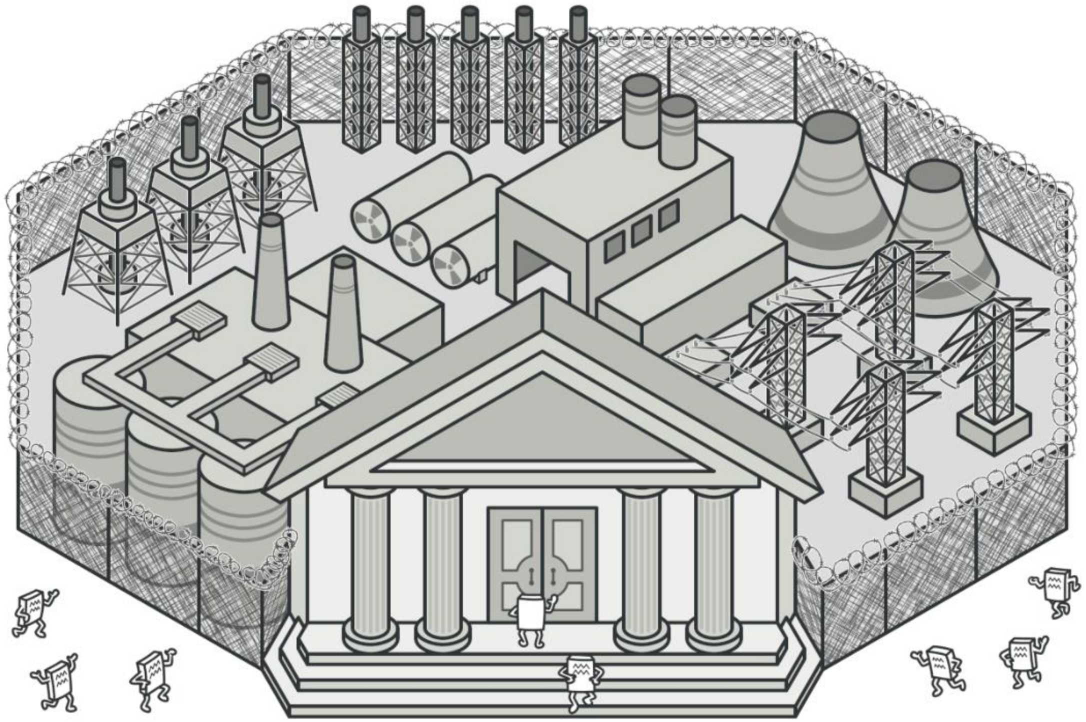
Proxy

Lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.

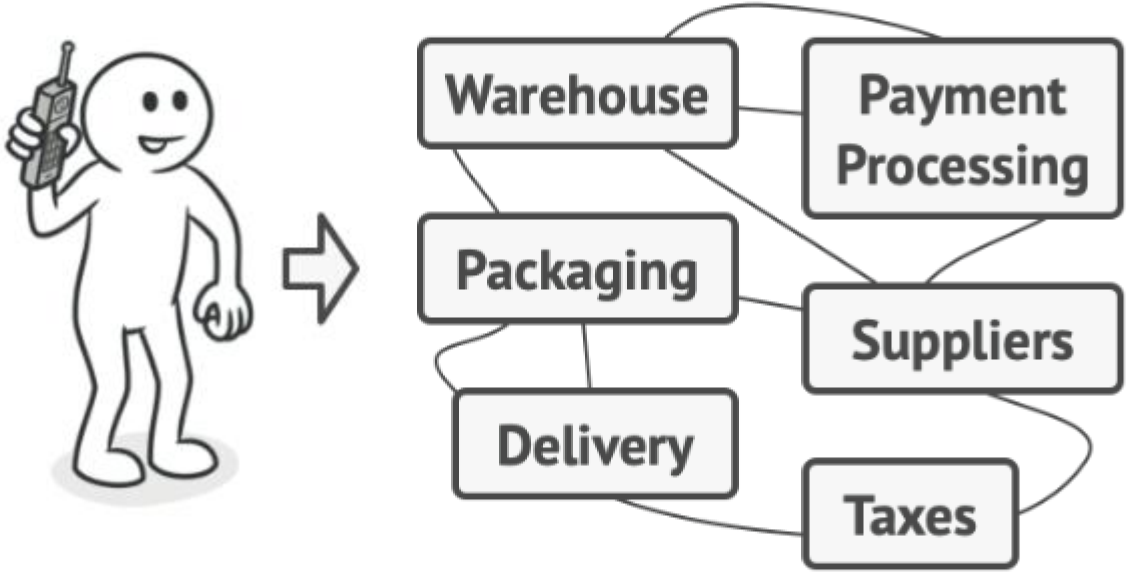
We are not going over this one

Façade Pattern

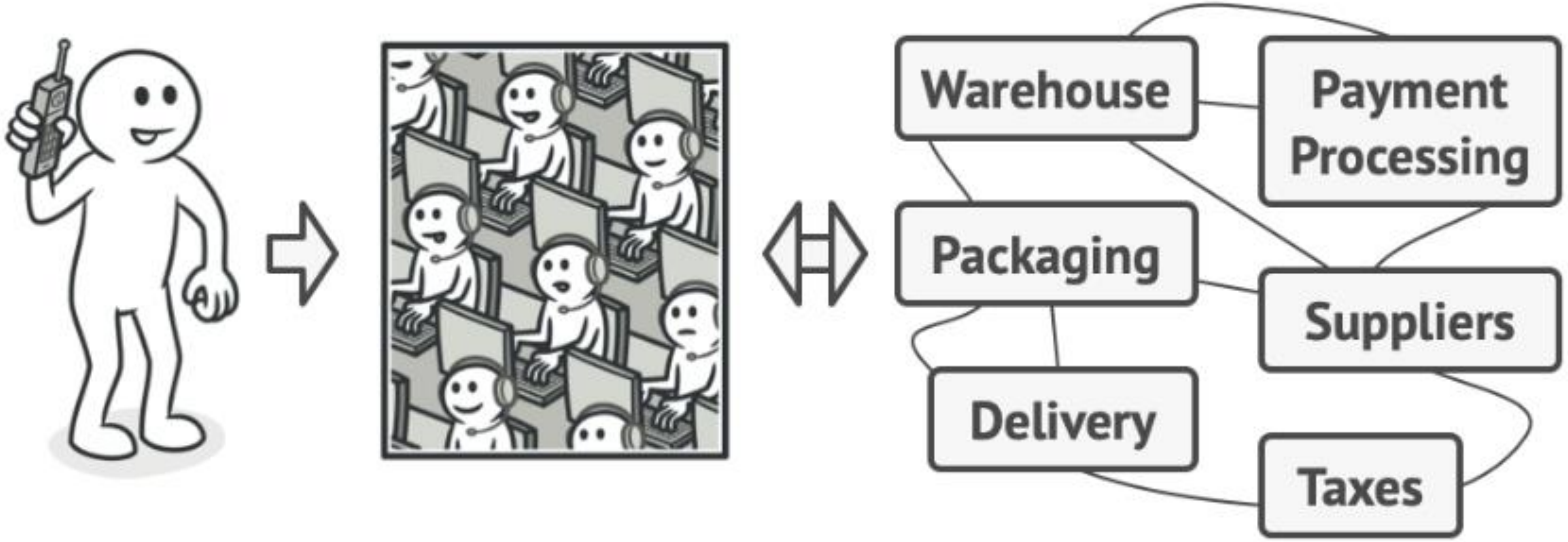
Provides a simplified interface to a library, a framework, or any other complex set of classes.



Problem


Warehouse
Payment Processing
Packaging
Suppliers
Delivery
Taxes

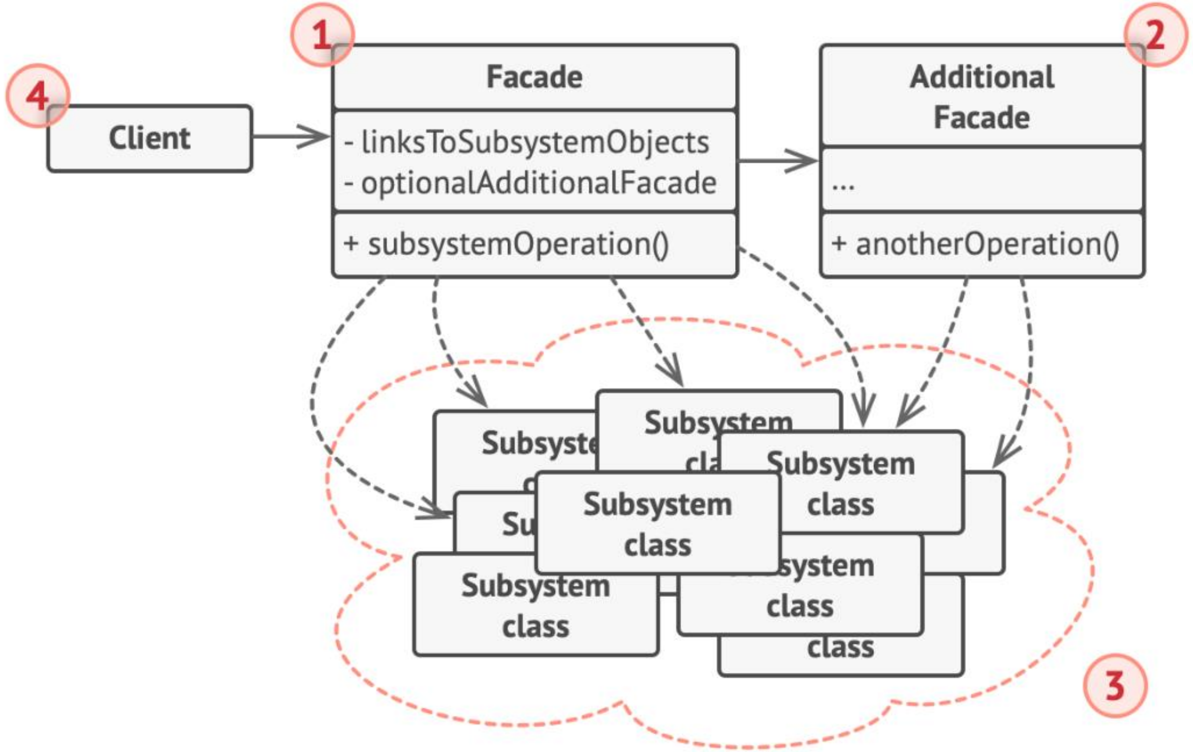
Solution



Placing orders by phone.

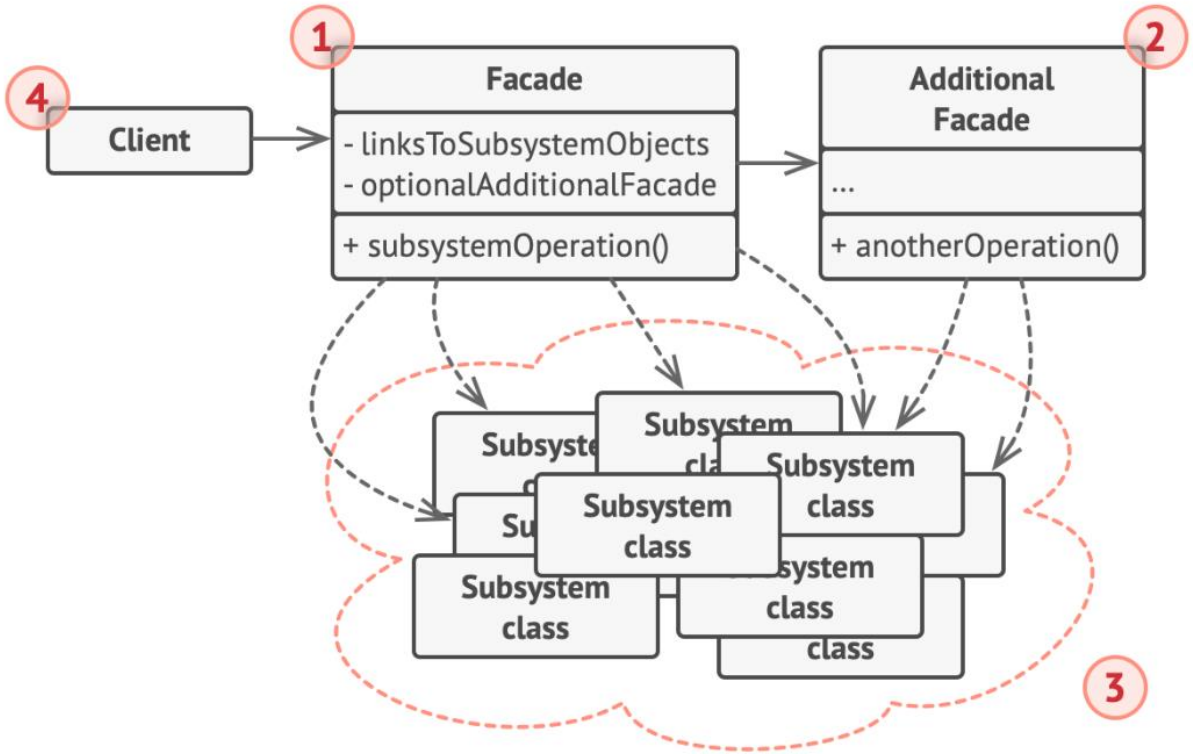
UML Class Diagram

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.



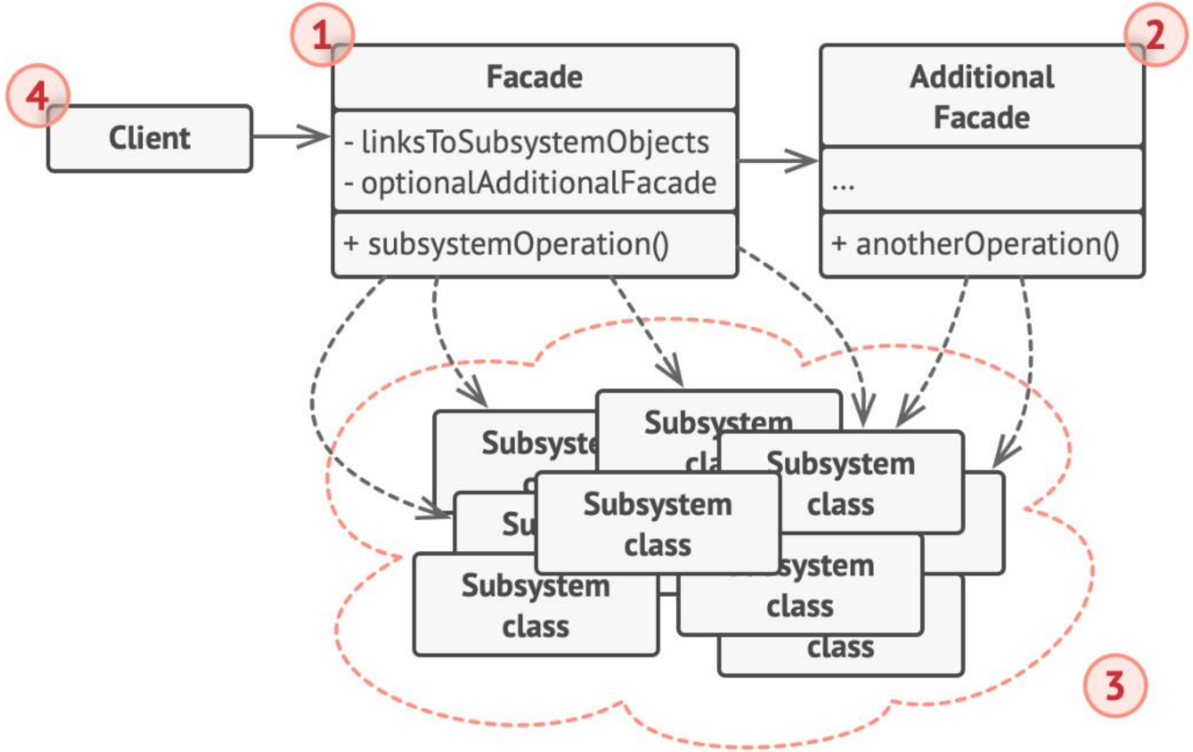
UML Class Diagram

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.



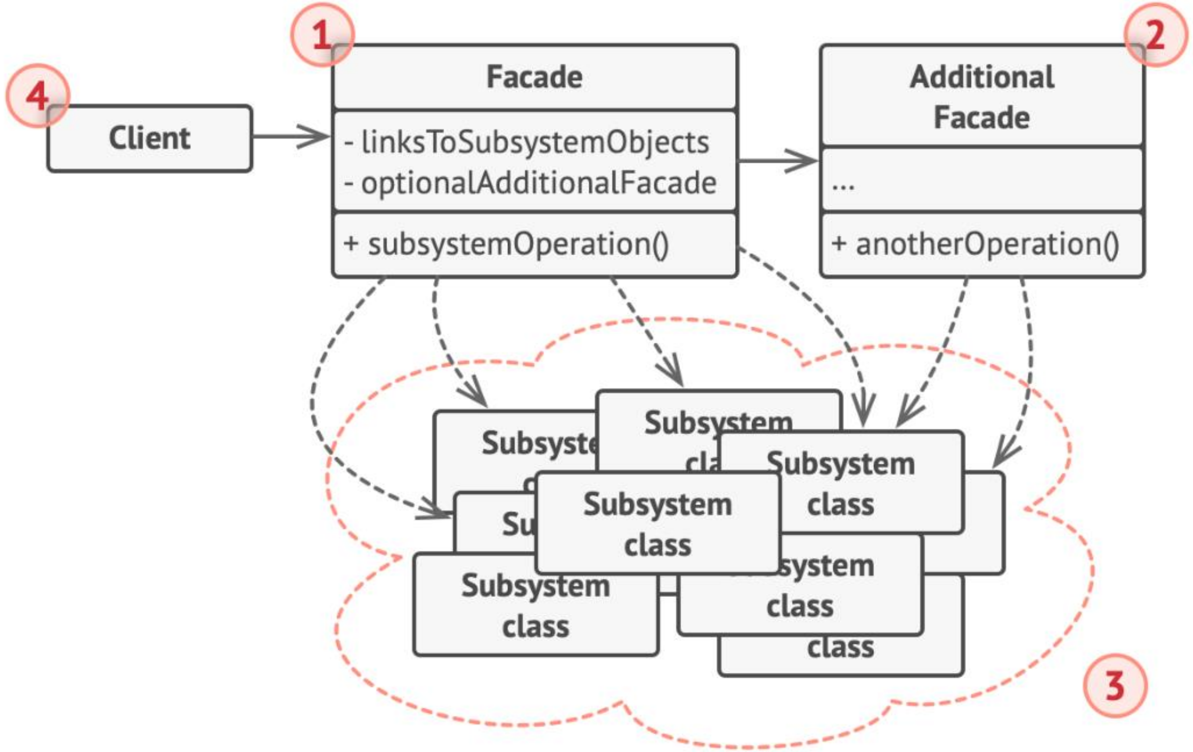
UML Class Diagram

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.



UML Class Diagram

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



Applicability

When you need to have a limited but straightforward interface to a complex subsystem

When you want to structure a subsystem into layers

The big pro

You can isolate your code from the complexity of a subsystem.

Potential con: extremely high coupling

Relationship with other patterns

Abstract Factory can serve as an alternative to **Facade** when you only want to hide the way the subsystem objects are created from the client code.

A **Facade** class can often be transformed into a **Singleton** since a single facade object is sufficient in most cases.

Façade Pattern

No code example

Those of you who have explored the project, which class is the façade?