

# Structural Patterns: Proxy, Facade, Bridge

---

COMP3522 OBJECT ORIENTED PROGRAMMING 2  
WEEK 10

<https://refactoring.guru/design-patterns>

# 结构型模式：代理、外观、桥接

---

COMP3522 面向对象程序设计2 第10周

<https://refactoring.guru/design-patterns>

# Categorizing Design Patterns

## Behavioural

Focused on communication and interaction between objects. How do we get objects talking to each other while minimizing coupling?

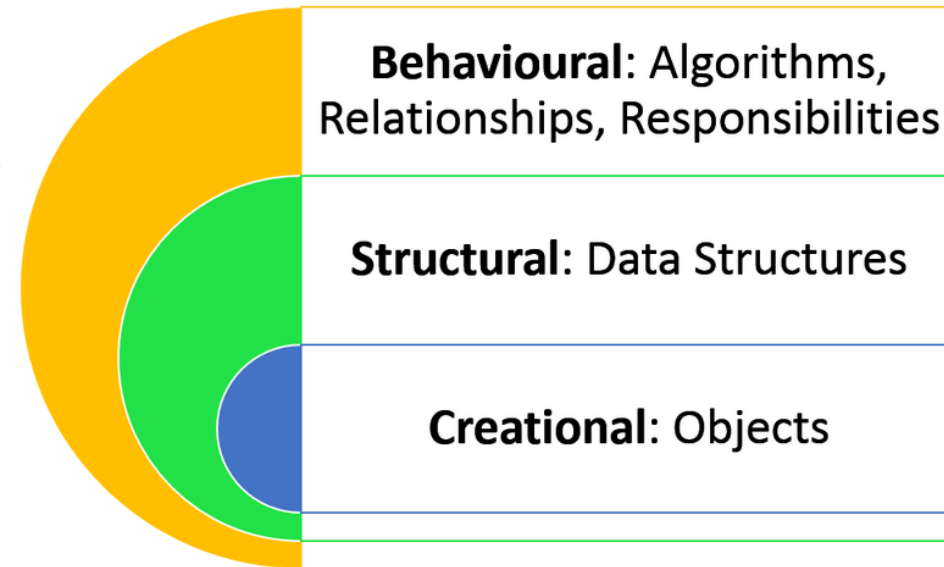
## Structural (We are looking at these!)

How do classes and objects combine to form structures in our programs? Focus on architecting to allow for maximum flexibility and maintainability.

## Creational

All about class instantiation. Different strategies and techniques to instantiate an object, or group of objects

Design Patterns



# 设计模式的分类

## 行为型

关注对象之间的通信与交互。我们如何在最小化耦合的同时让对象之间进行对话？

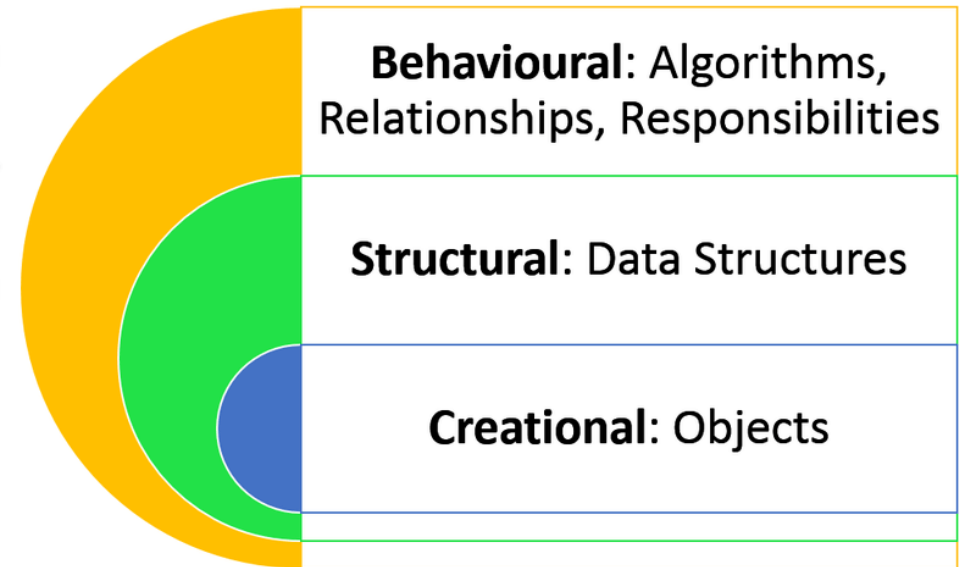
## 结构型 (我们正在研究这些!)

类和对象如何组合以形成程序中的结构？重点在于架构设计，以便实现最大的灵活性和可维护性。

## 创建型

关于类实例化的全部内容。不同的策略和用于实例化对象或一组对象的技术

Design Patterns



## Structural patterns

### Proxy

- When you want to wrap around an object and control access to it.

### Facade

- A simple interface to a complex API/Subsystem

### Bridge

- Breaking down a large class or a large set of coupled classes into abstractions and implementations.

## 结构型模式

### 代理

- 当你想要包装一个对象并控制对其的访问时。

### 外观

- 为复杂 API/子系统提供一个简单的接口

### 桥接

- 将一个大型类或一组紧密耦合的类拆分为抽象和实现。

# Wrapper

---

# 包装器

---

# What is a wrapper?

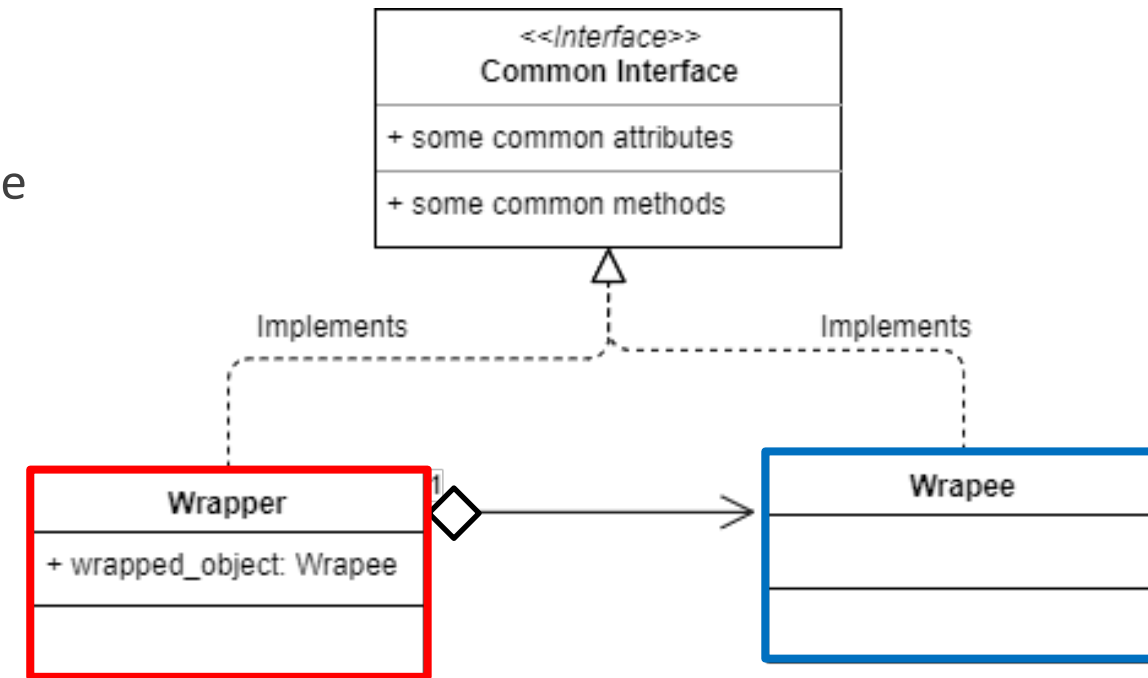
A class that ‘wraps’ around another class.

The **wrapper** has the same interface as the ‘**wrapee**’.

**Remember, this is interface in the OOP sense.** The **wrapper** has the same public method and attributes as the **wrapee**. We can implement this using an abstract base class

All calls to the **wrapper** call the subsequent functions in the **wrapee** or the wrapped object.

Wrapping an object, with another object of the same interface allows us the flexibility to do some extra processing before and after we make the call to the **wrapee**



# 什么是包装器？

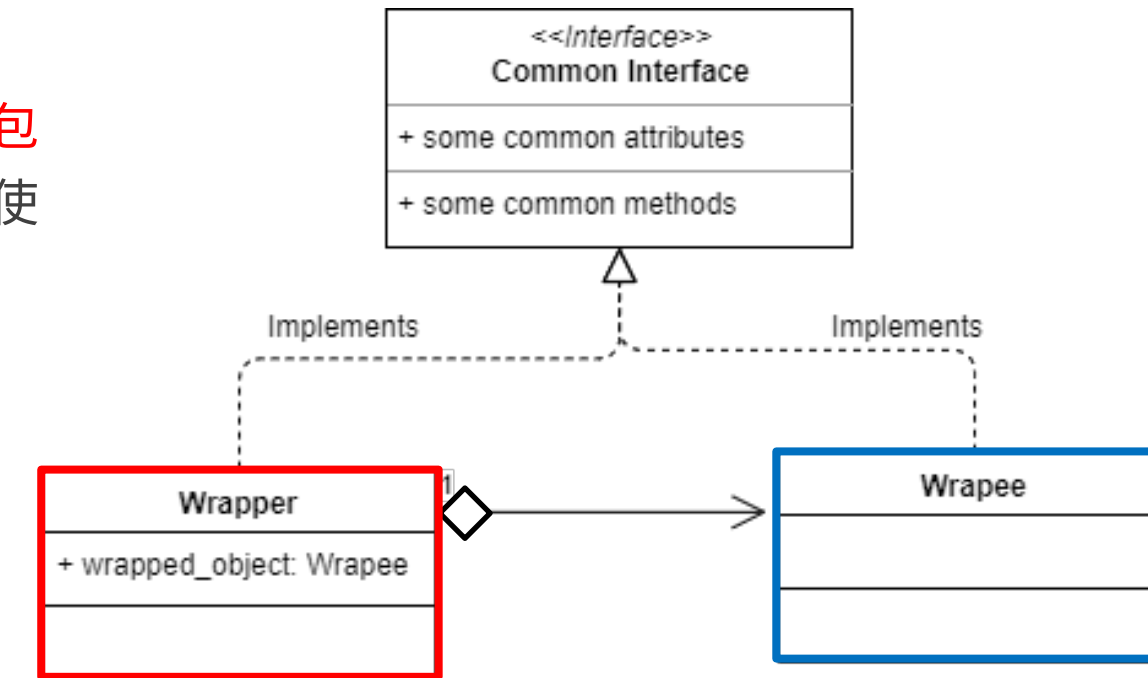
一个“包装”另一个类的类。

该**包装器**具有与“**被包装者**”相同的接口。

请注意，此处的接口是面向对象编程意义上的接口。该**包装器**具有与**被包装者**相同的公共方法和属性。我们可以使用抽象基类来实现这一点

对**包装器**的所有调用都会进一步调用**被包装者**或被包装对象中的相应函数。

使用具有相同接口的另一个对象包装一个对象，使我们在调用**被包装者**之前和之后可以灵活地执行一些额外处理



# What is a wrapper?

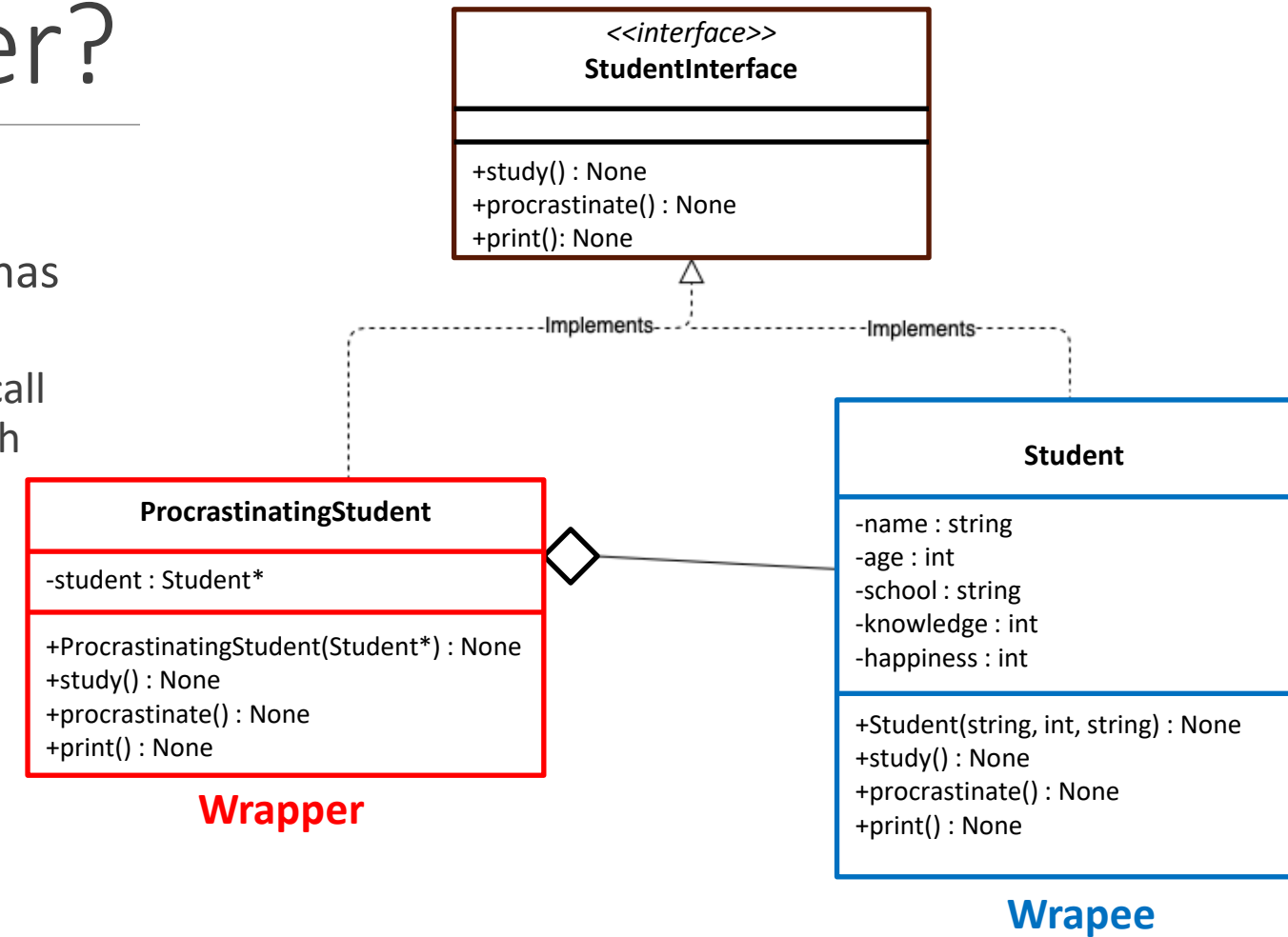
**Student** is the **wrapee**

**ProcrastinatingStudent** is a **wrapper**. It has an instance of **Student**

- Any calls to its functions will eventually call the **Student** instance's functions, but with possibly additional behavior

Both classes implement the functions defined in the **StudentInterface**

Wrapper.cpp



# 什么是包装器？

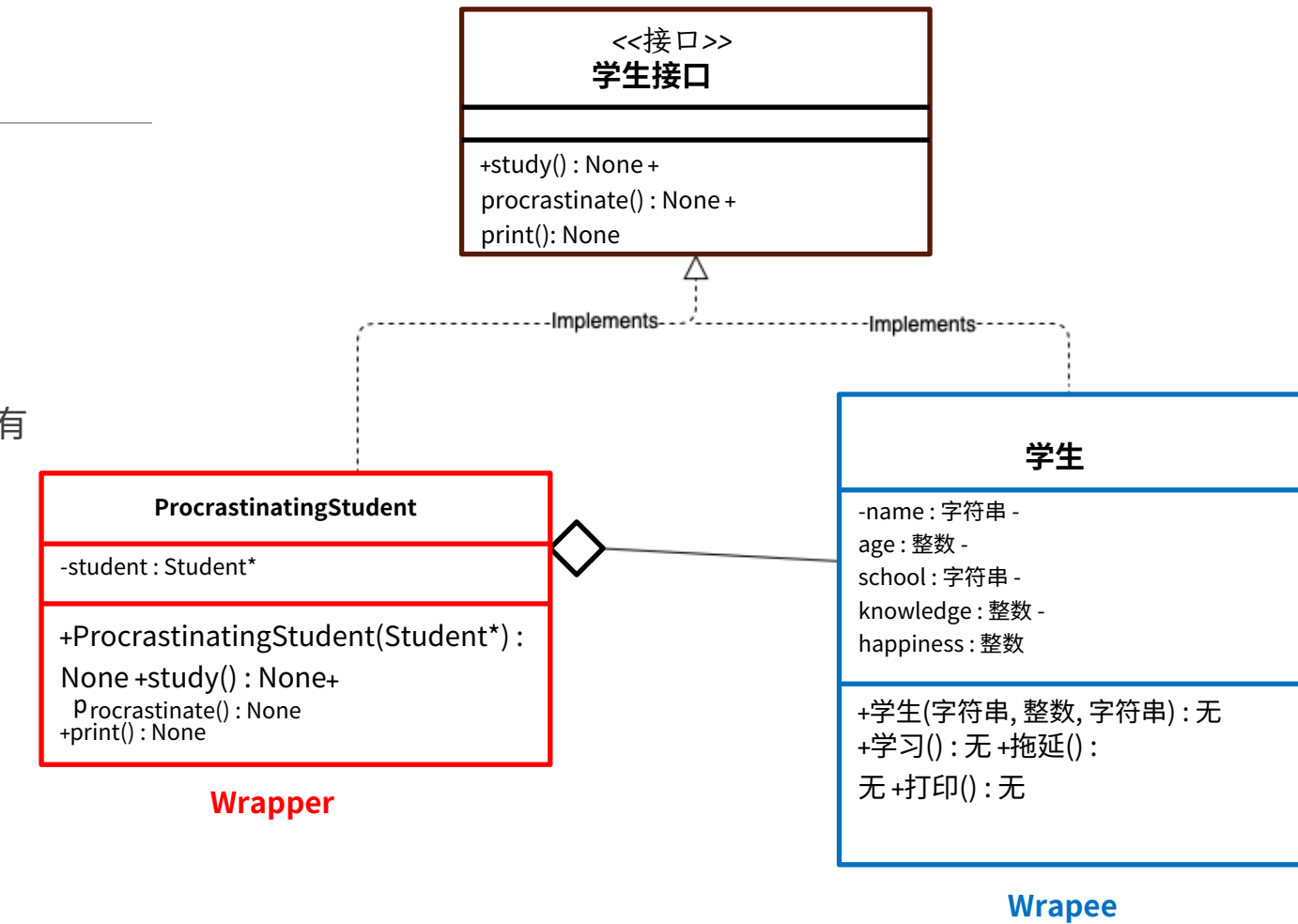
**学生** 是 **被包装者**

**拖延的学生** 是一个 **包装器**。它拥有一个 **学生** 的实例

- 对其函数的任何调用最终都会调用该 **学生** 实例的函数，但是带有可能附加的行为

这两个类都实现了在 **StudentInterface** 中定义的函数

Wrapper.cpp

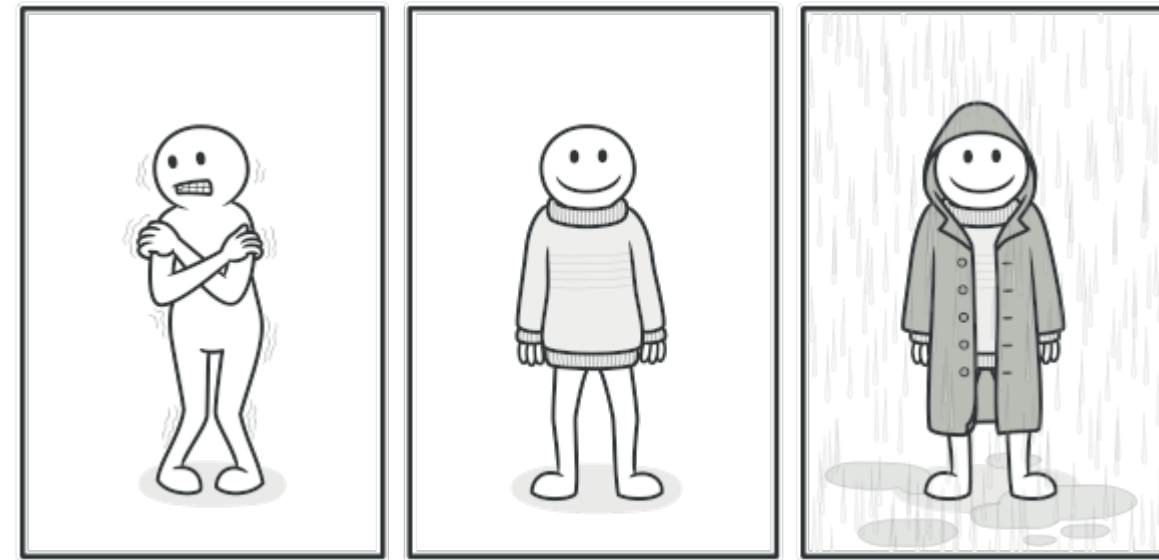


# Wearing many layers (Multiple Wrappers)

We can wrap a **base component** in a **wrapper** to add some extra behaviour or functionality to it.

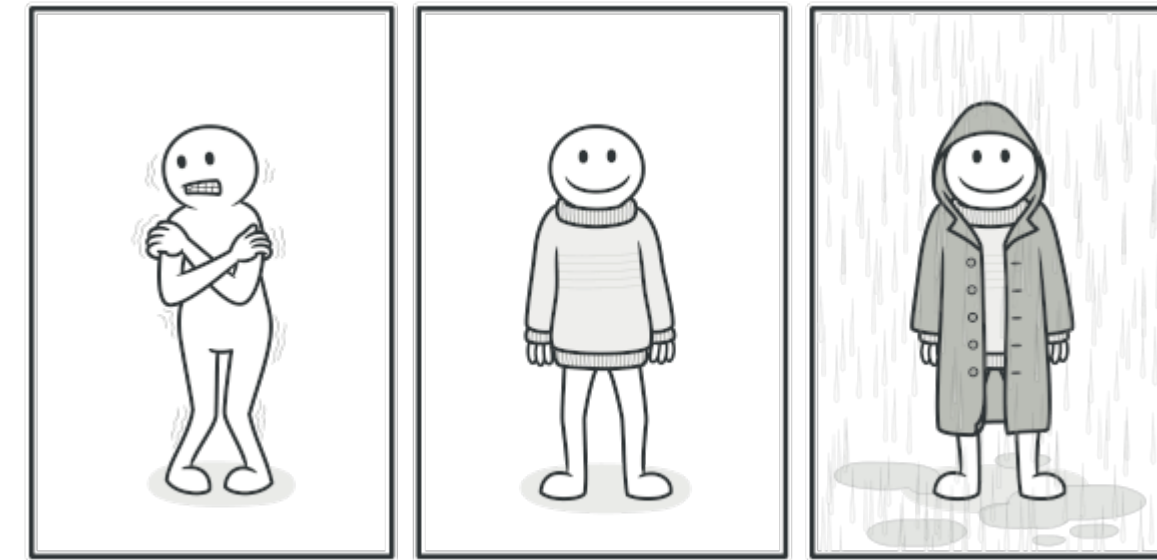
We can also wrap that **wrapper** in another **wrapper** to add more functionality

This can go on infinitely. (*We'll learn about this later...Decorator pattern*)



# 穿戴多层（多个包装器）

我们可以将一个 **基础组件** 用 **包装器** 包裹起来，为其添加一些额外的行为或功能。我们还可以将这个 **包装器** 再用另一个 **包装器** 包裹，以添加更多功能。这个过程可以无限进行下去。（我们稍后会学习这一点……装饰器模式）



# Proxy

---

# 代理

---

# Proxy

A Proxy is a **wrapper**.

It controls access to the **wrapee**.

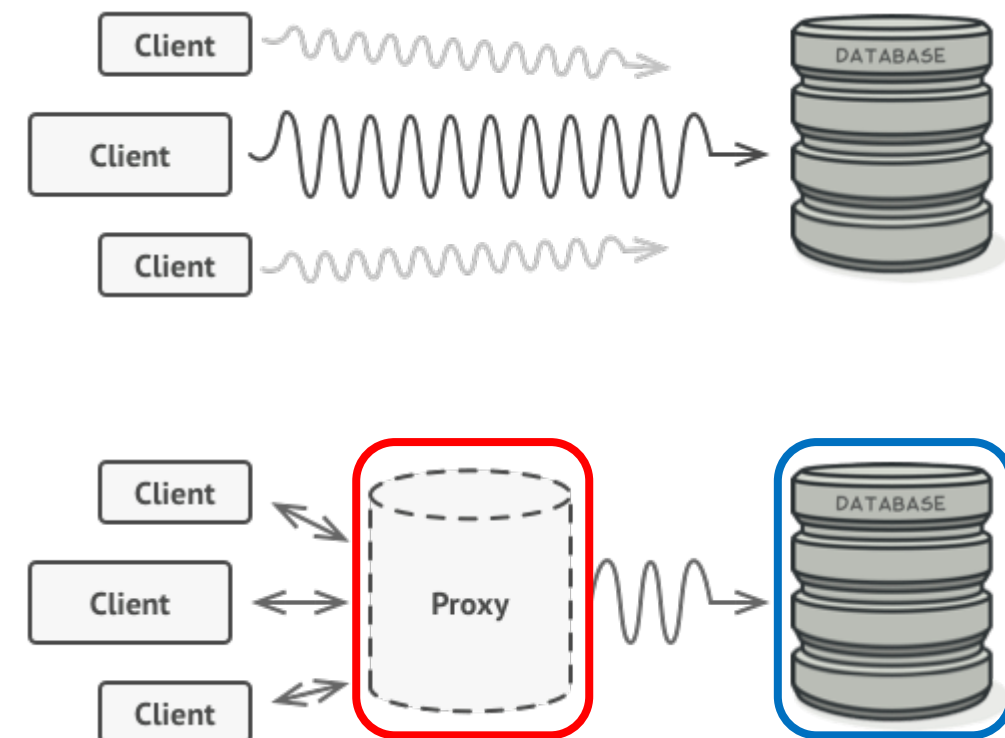
For example:

Database queries can be slow.

We might want to not only speed this up but also restrict access to the database. It's better to allow only objects with authorization.

A **proxy** in this case, is an object that has the **same interface** as the **database** and has access to it.

It can handle **authorization**, result **caching** and other **optimizations**. The client doesn't even need to know.



# 代理

代理是一个**包装器**。

它控制对**被包装对象**的访问。

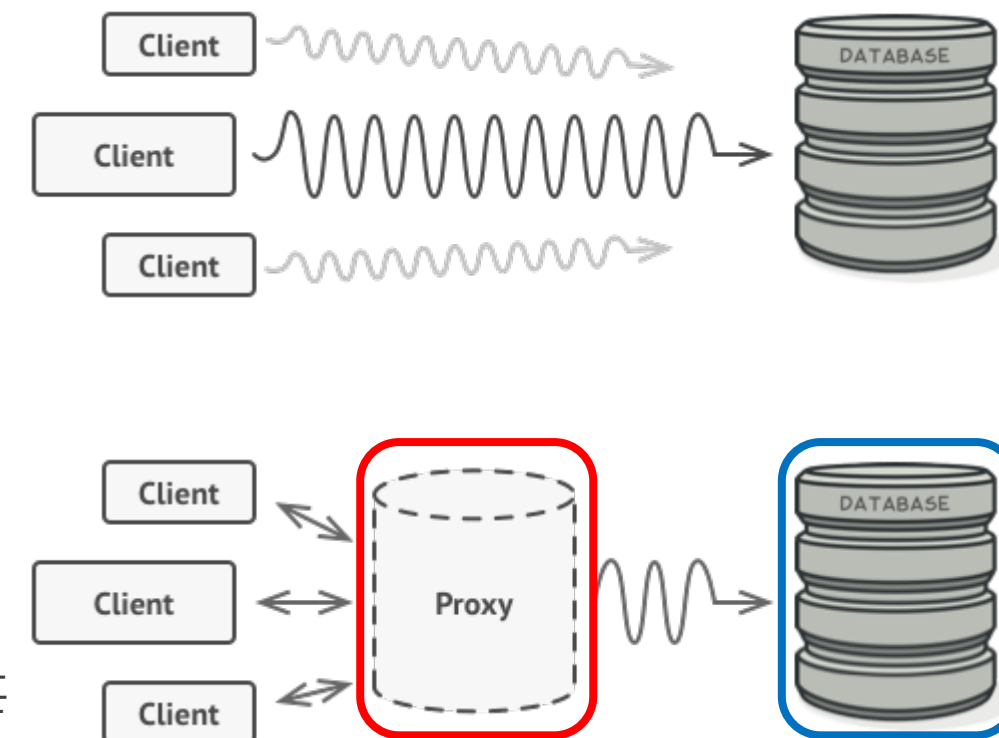
例如：

数据库查询可能很慢。

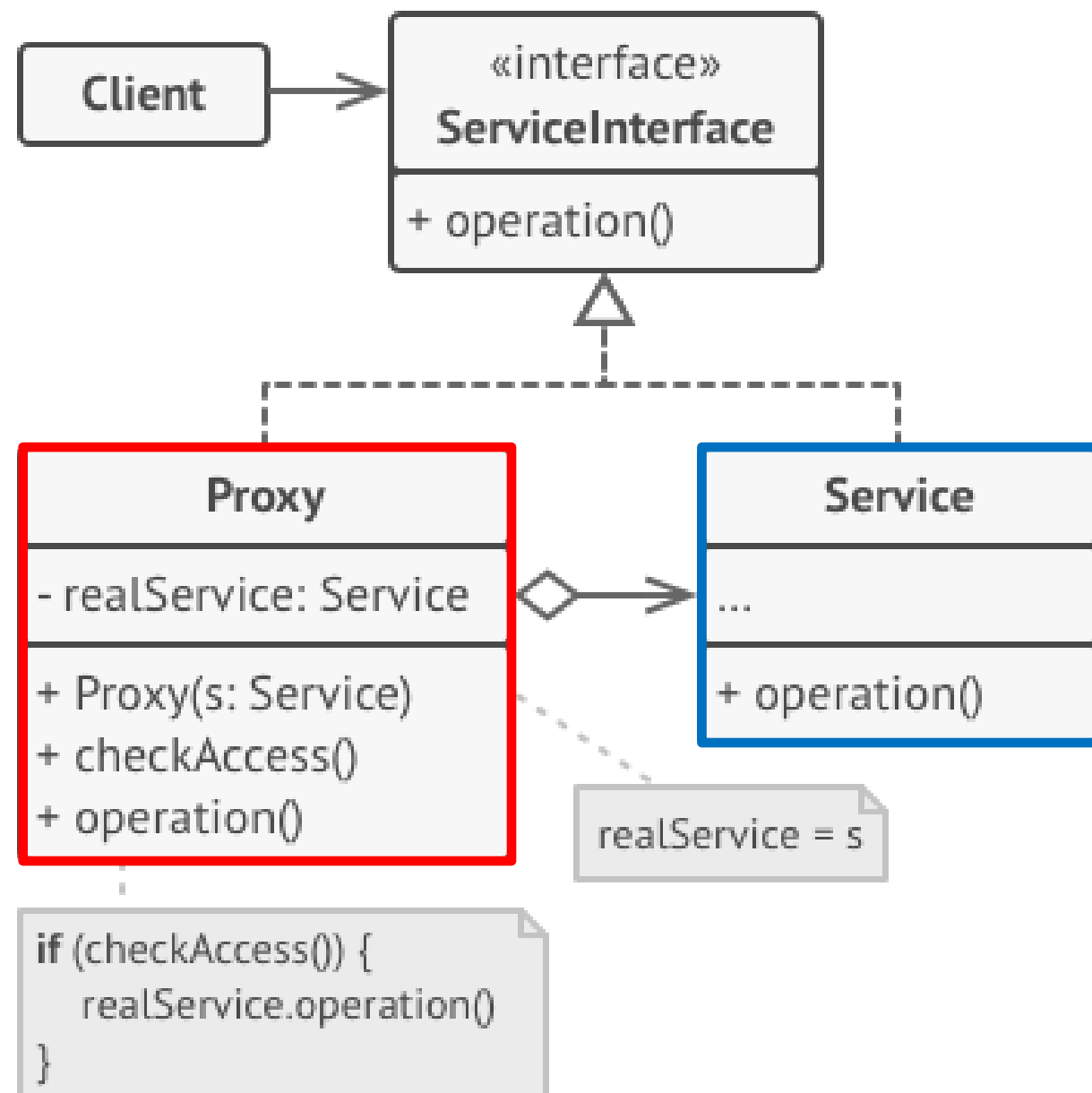
我们可能不仅希望加快查询速度，还希望限制对数据库的访问。最好只允许具有授权的对象进行访问。

在这种情况下，**代理** 是一个具有与 **数据库** 相同的 **接口** 并能够访问它的对象。

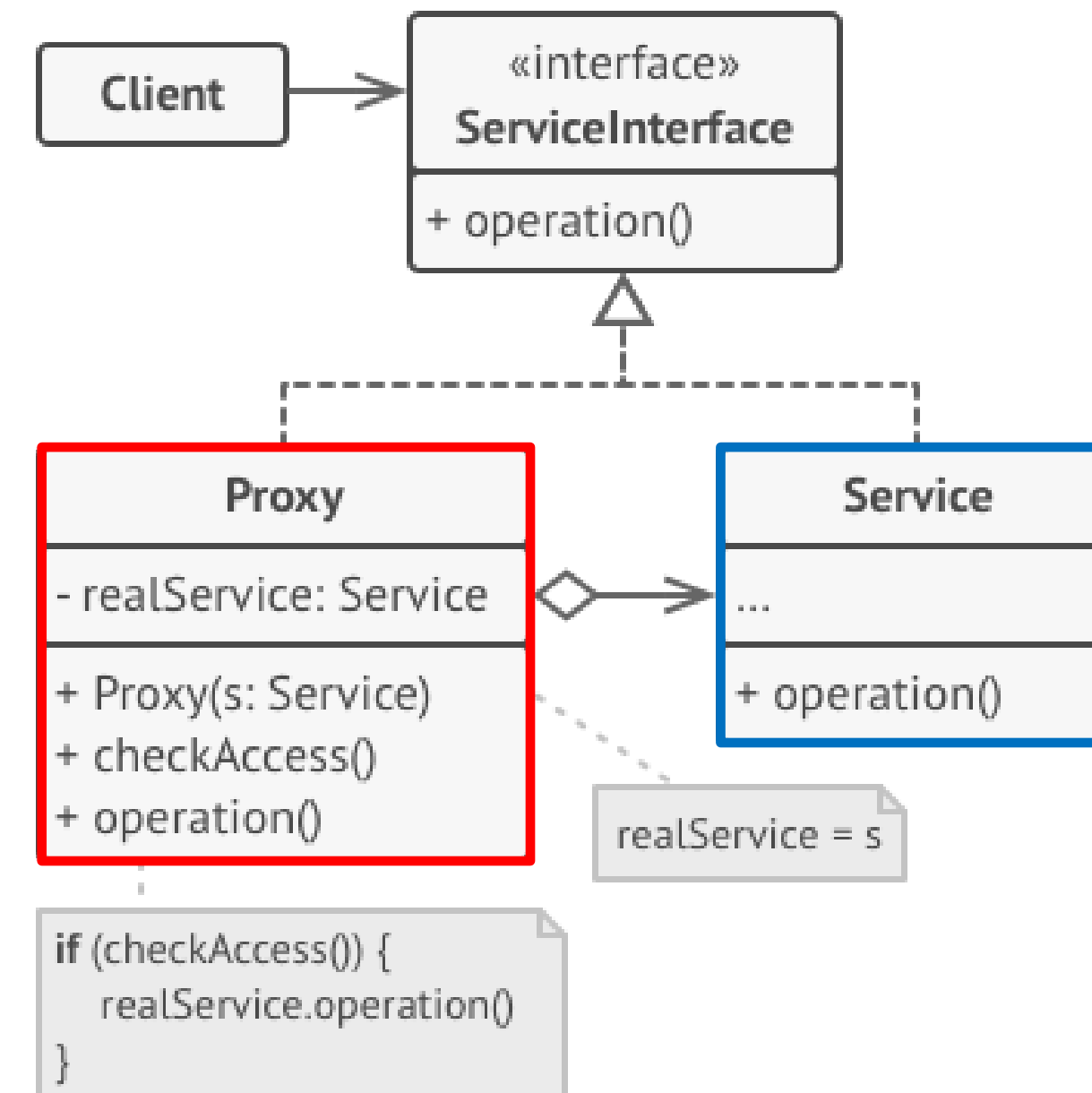
它可以处理 **授权**、结果 **缓存** 以及其他 **优化**。客户端甚至无需知晓。



# The Proxy Pattern

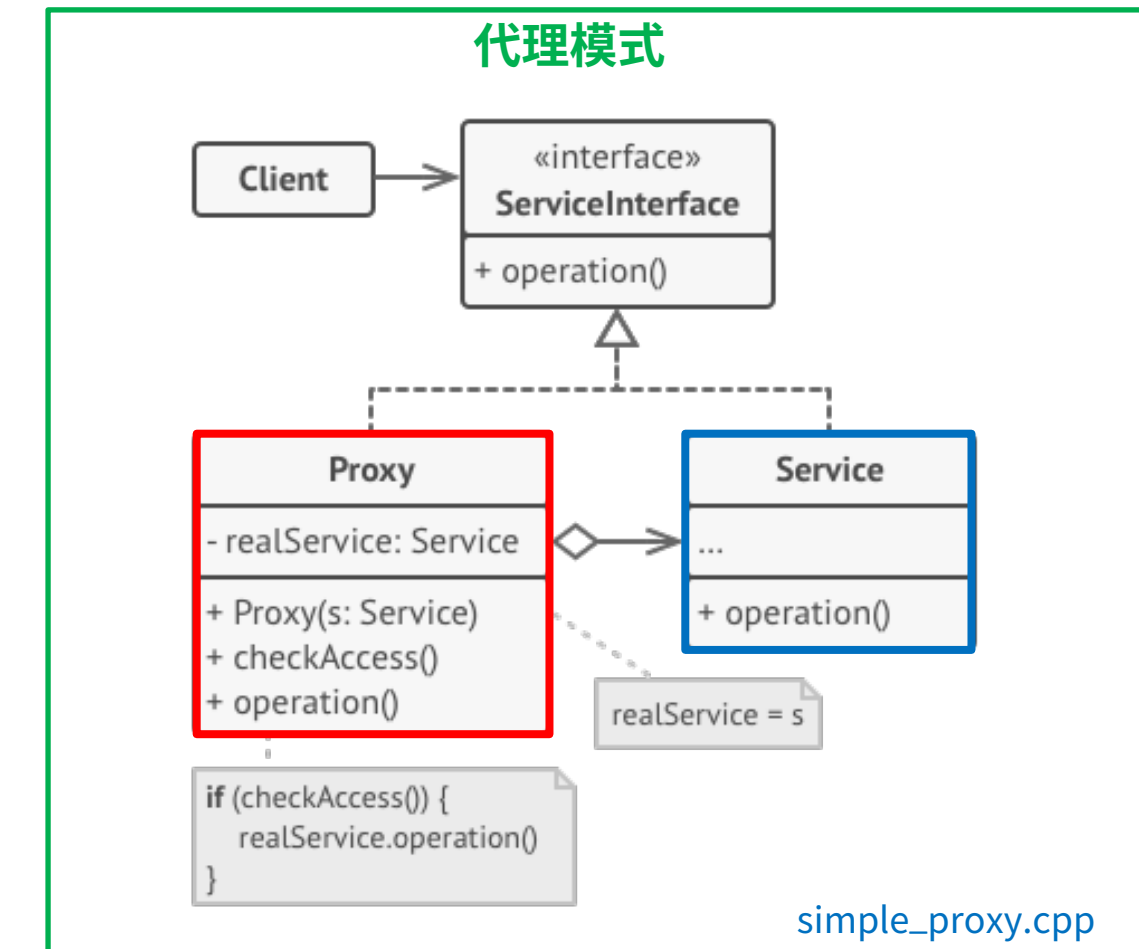
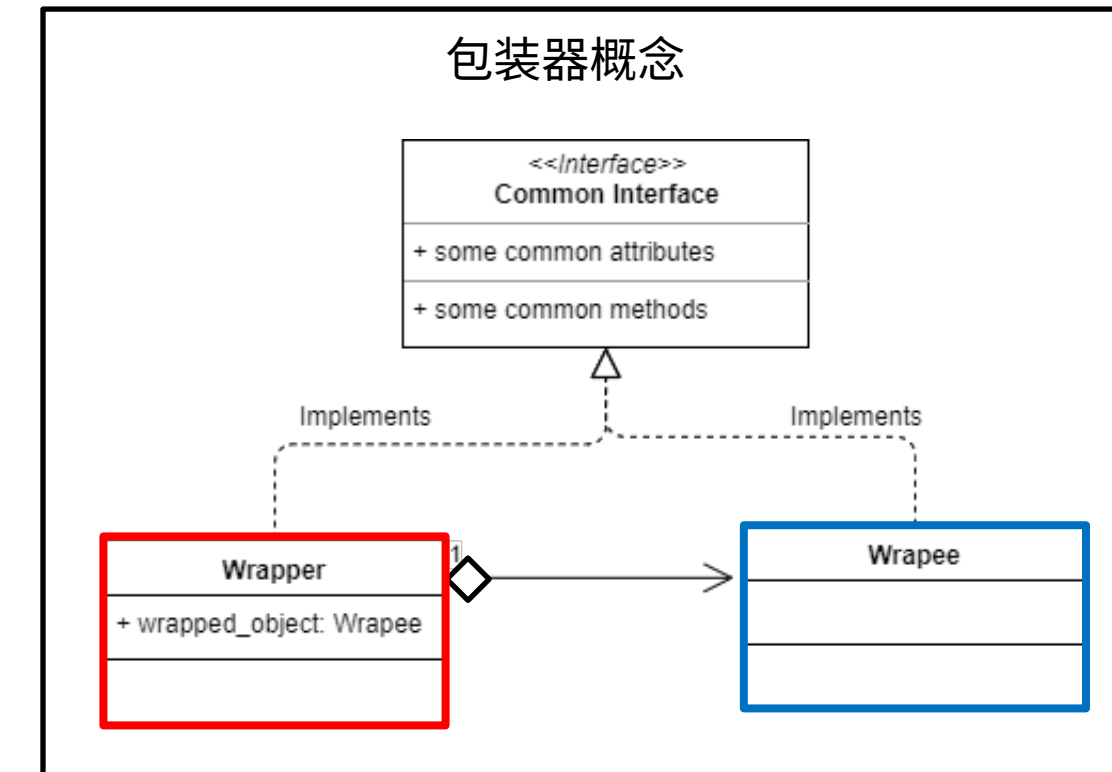
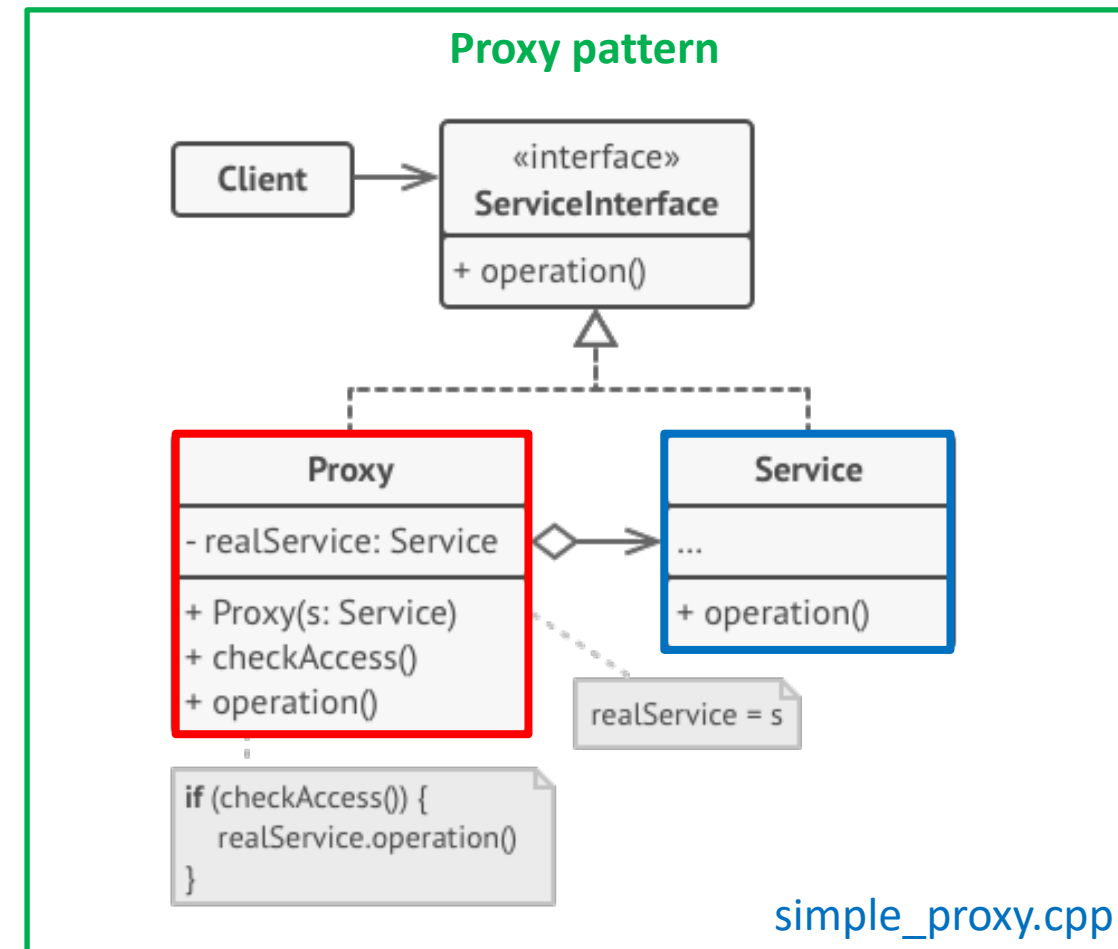
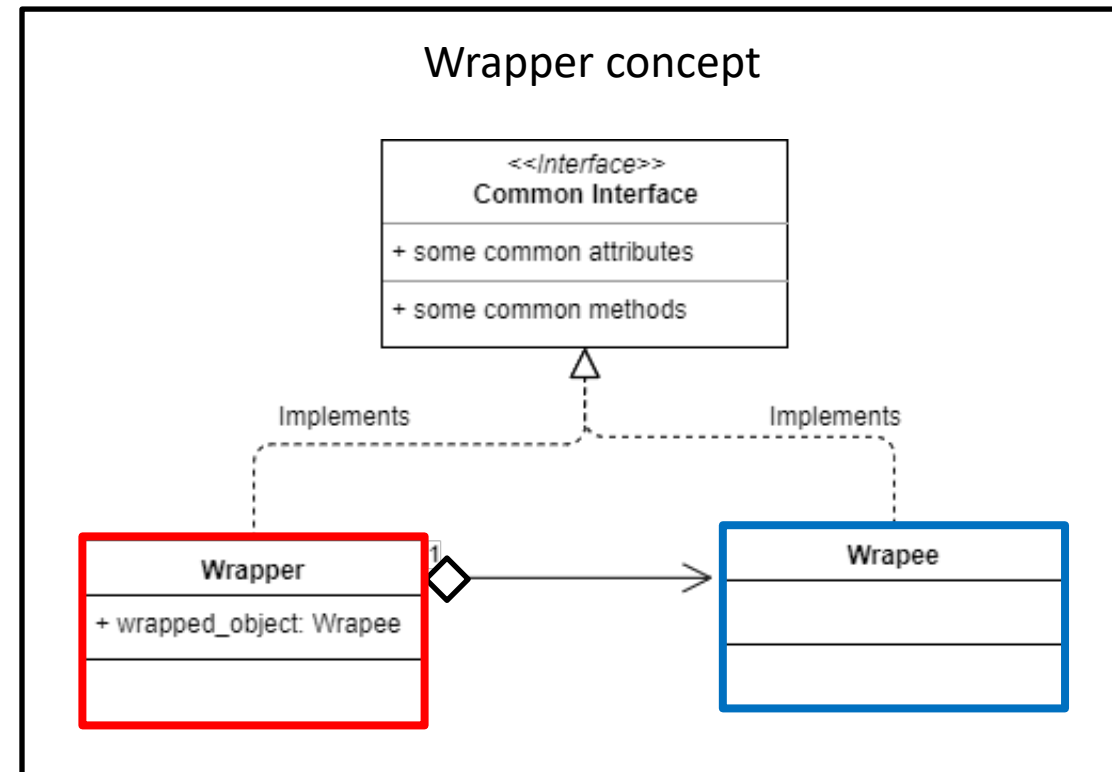


## 代理模式



This is the exact same structure as a wrapper.

这与包装器的结构完全相同。



# Proxy – Why and When do we use it

- If you want to execute something before or after the primary service logic
- Since the proxy implements the same interface as the service, it can be passed to any object that expects a ServiceInterface.
- Encapsulate and control access to **expensive objects**
- A Proxy can work even if the original service is not available.
- Follows the open closed principle. Can introduce **new proxies that extend behavior** instead of modifying the service.



# 代理——我们为什么以及何时使用它

- 如果你想在主要服务逻辑执行之前或之后运行某些操作
- 由于代理实现了与服务相同的接口，因此可以将其传递给任何期望 ServiceInterface 的对象。
- 封装并控制对 **昂贵对象** 的访问
- 即使原始服务不可用，代理仍然可以工作。
- 遵循开闭原则。可以引入 **新的代理扩展行为** 而不是修改服务。



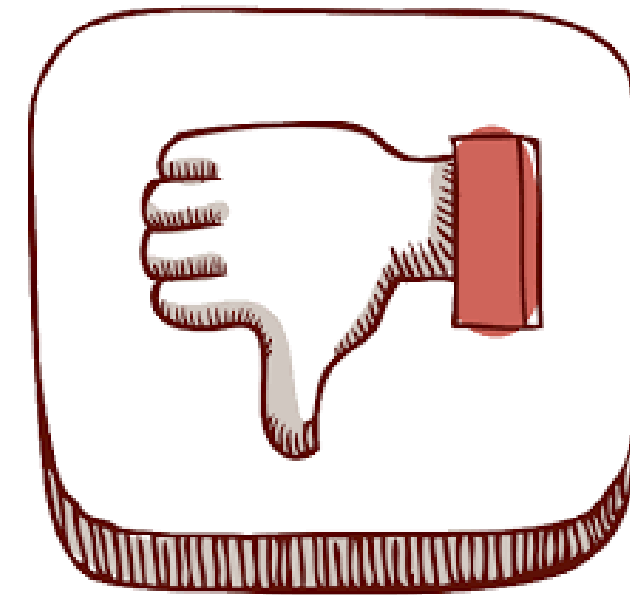
## Proxy – Disadvantages

- Can complicate the codebase if multiple proxies exist.
- Service **response might be delayed** if the proxy is carrying out extensive work before or after invoking the service.



## 代理 – 缺点

- 如果存在多个代理，可能会使代码库变得复杂。
- 服务 **响应可能延迟**，如果代理在调用服务之前或之后执行大量工作。



# Facade

---

WHEN YOU WANT TO HIDE THE MESS RIGHT BEFORE YOUR FRIENDS  
COME OVER FOR BOARD GAMES...

# 外观

---

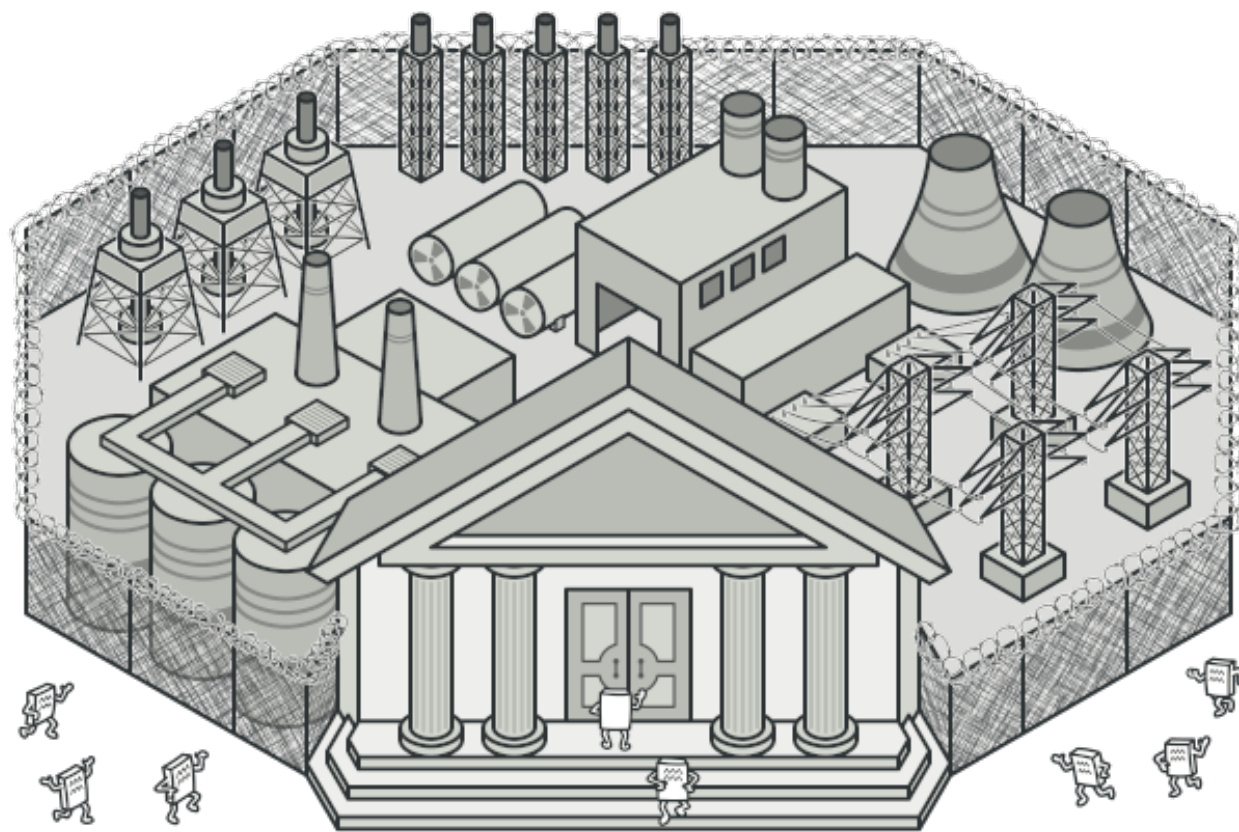
当你的朋友要来家里玩桌游时，你想要藏起那一团乱麻……

# Facade

The Facade pattern **encapsulates** a complex system or API

Used when we need to provide a **simple interface to a complex system**

Abstracts away the details of the complex system or API

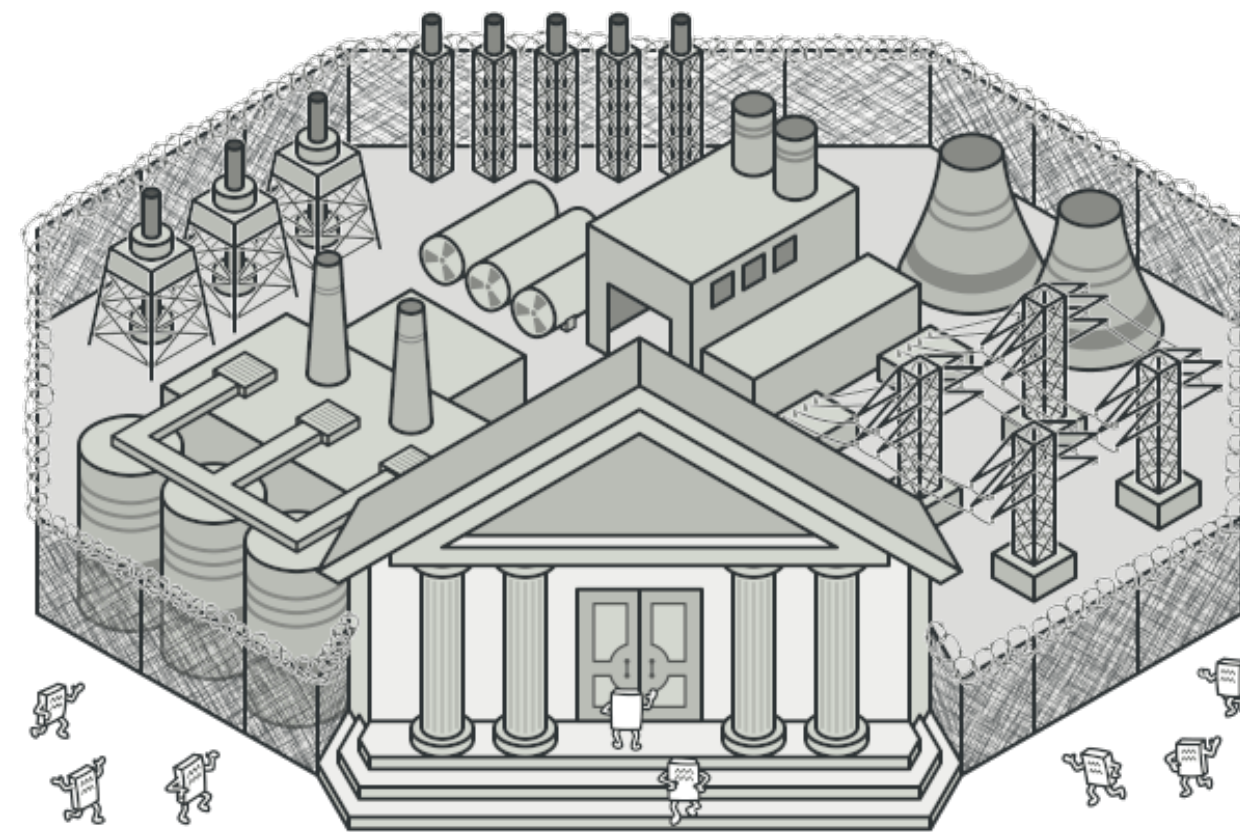


# 外观

外观模式 **封装** 了一个复杂的系统或 API

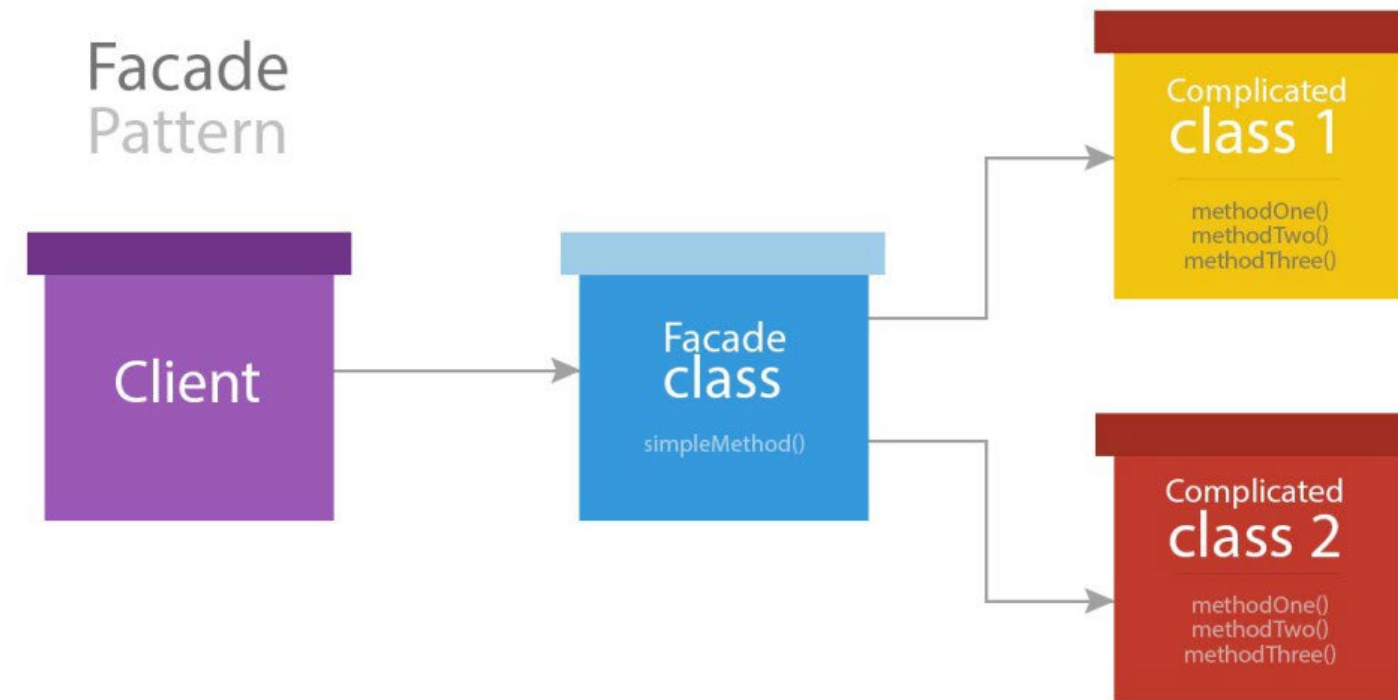
当我们需要为复杂系统提供一个**简单接口**时使用

将复杂系统或 API 的细节抽象化



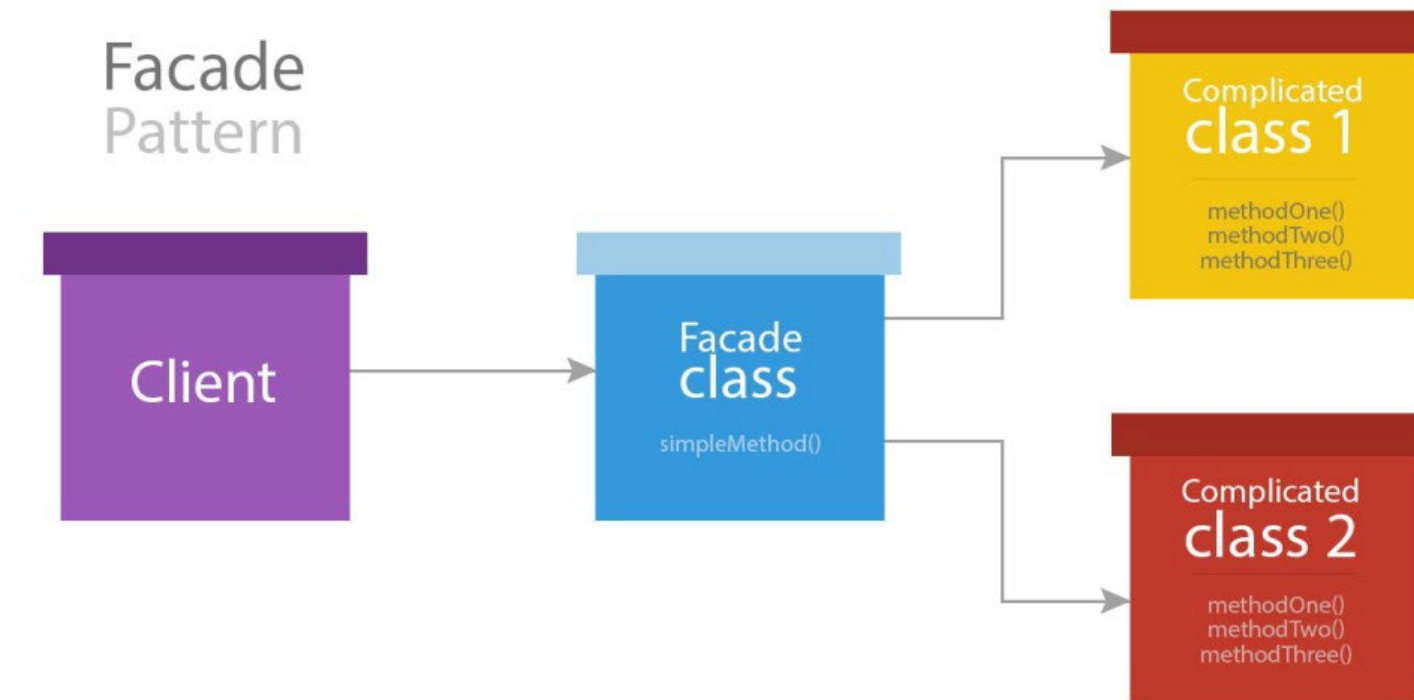
# Facade

- Often we need to implement a complex API in our program
- This complex API / system has a lot of classes and moving parts
- Often we import such libraries to only use a subset of those features and to solve a specific need.
- Façade encapsulate these complex systems and hides their complex behaviour.
- The rest of our code only needs to be aware of the **simple interface** provided by the **Façade**.

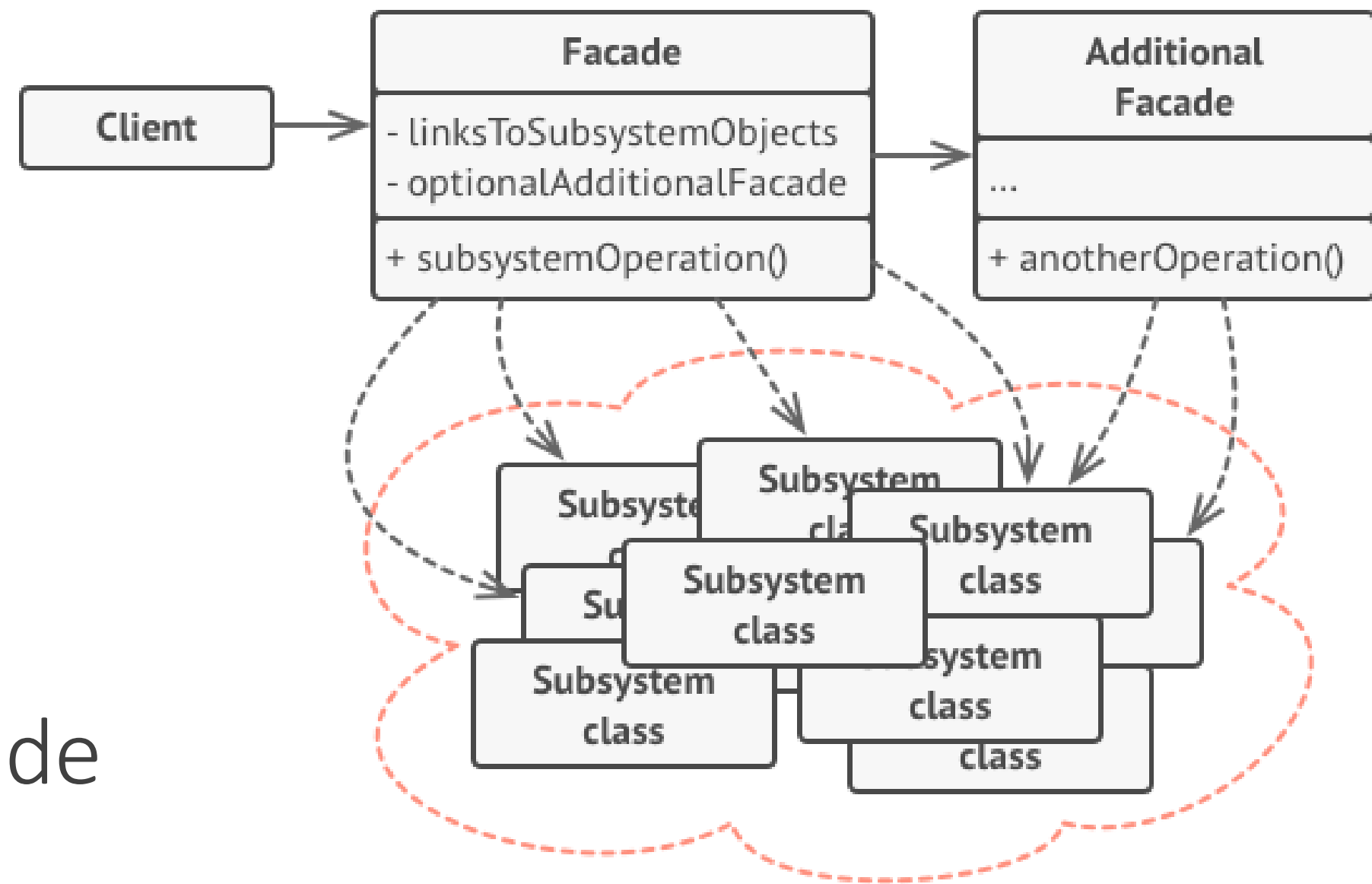


# 外观

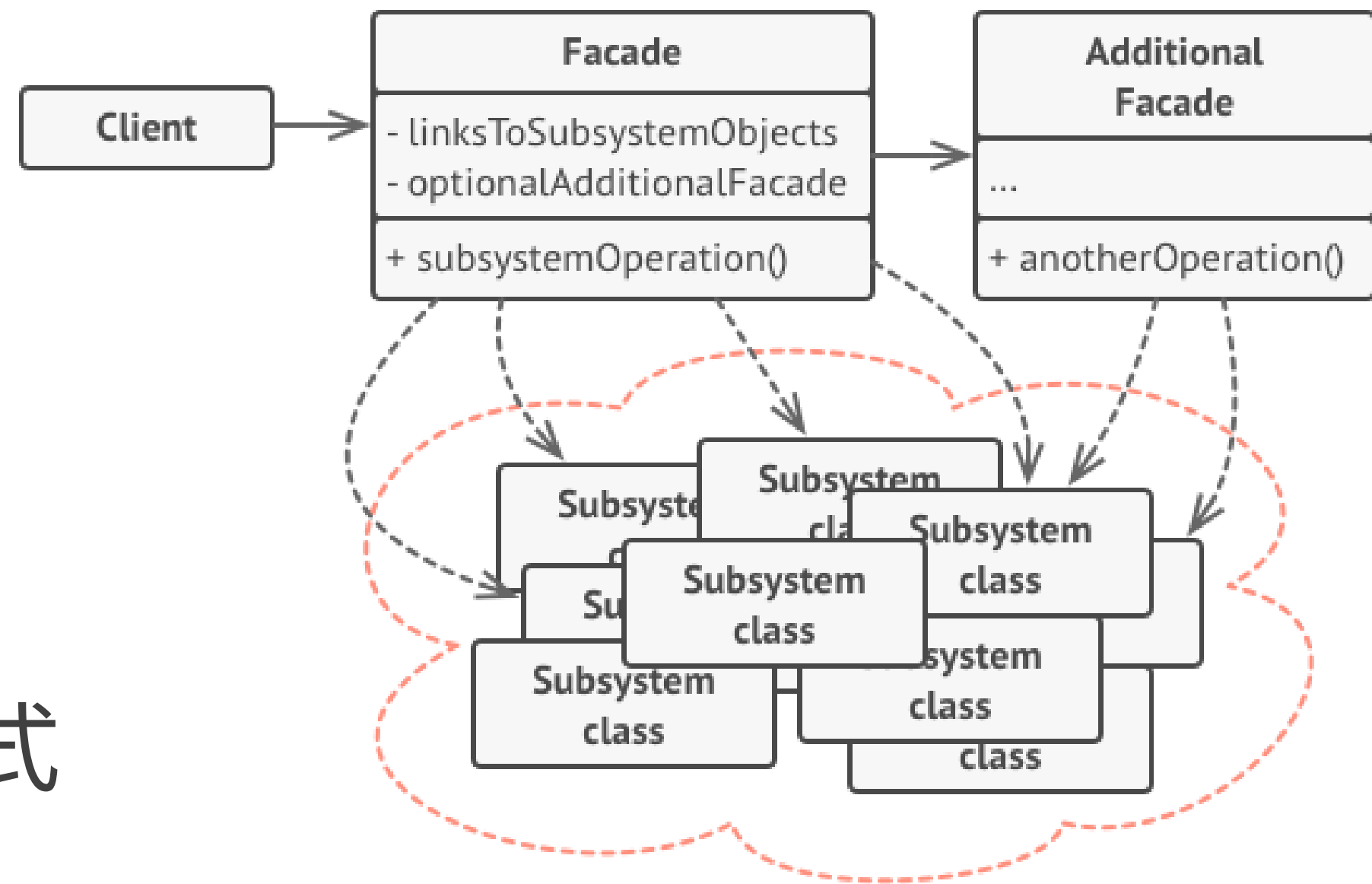
- 通常我们需要在程序中实现一个复杂的 API
- 这个复杂的 API / 系统包含大量类和可变部分
- 我们经常导入此类库只是为了使用其中一部分功能，以解决特定需求。
- 外观模式封装了这些复杂系统，并隐藏了它们的复杂行为。
- 我们其余的代码只需了解 **外观** 提供的 **简单接口**。



# The Facade Pattern

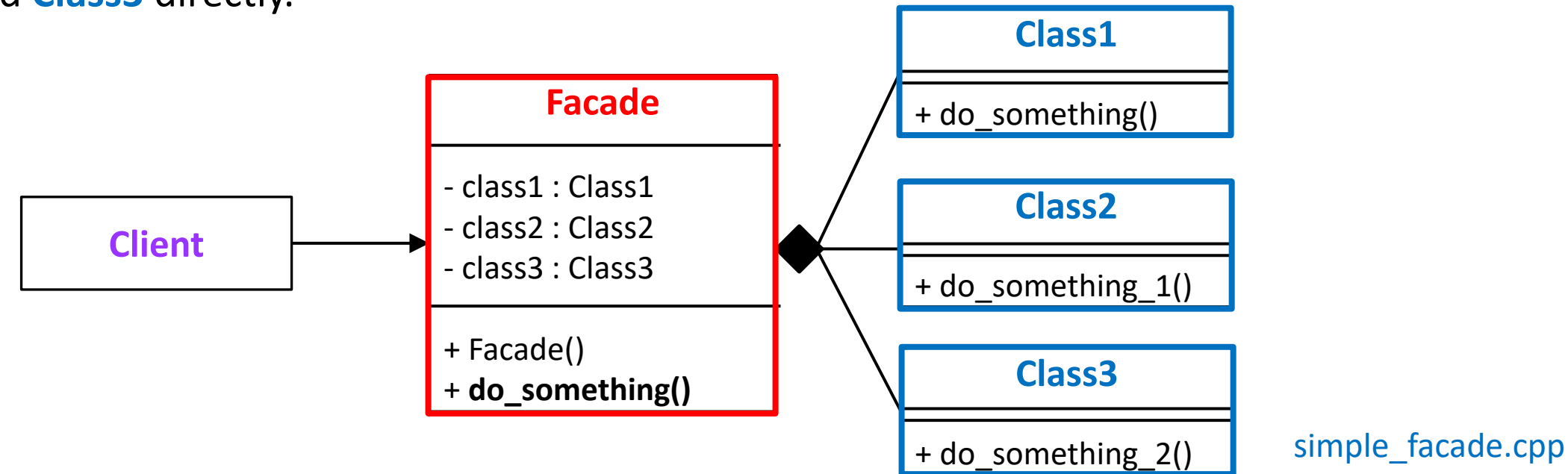


# 外观模式



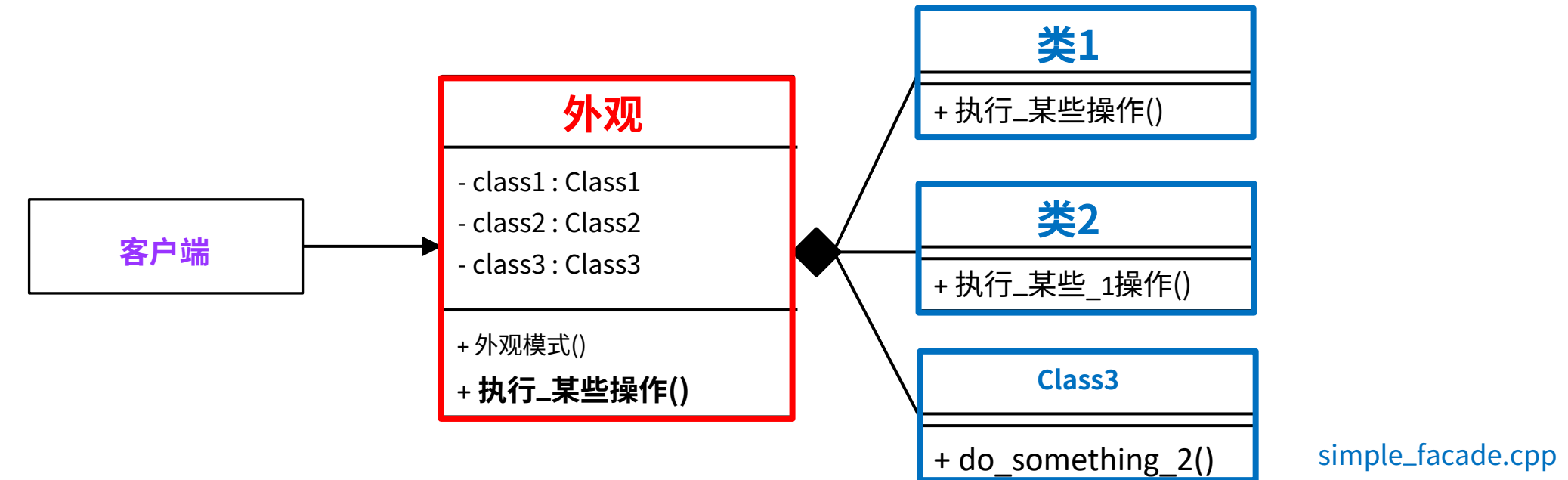
# Simple Facade

- **Problem:** **Client** wants to perform an action that only a sequence of calls to **Class1**, **Class2**, and **Class3** will complete. Don't want to couple **Client** with the **classes**
- Hide the sequence of calls in **Facade**'s **do\_something** function
- **Client** now only needs to call **Facade**'s **do\_something** function instead of calling **Class1**, **Class2**, and **Class3** directly.



# 简单外观

- **问题:** **客户端** 想要执行一个操作，该操作只能通过调用 **Class1**、**Class2**和**Class3** 将完成。不希望将**Client**与**classes**耦合
- 将一系列调用隐藏在 **Facade** 的 **do\_something** 函数中
- **客户端**现在只需调用 **Facade** 的 **do\_something** 函数，而无需直接调用 **Class1**、**Class2** 和 **Class3** 。

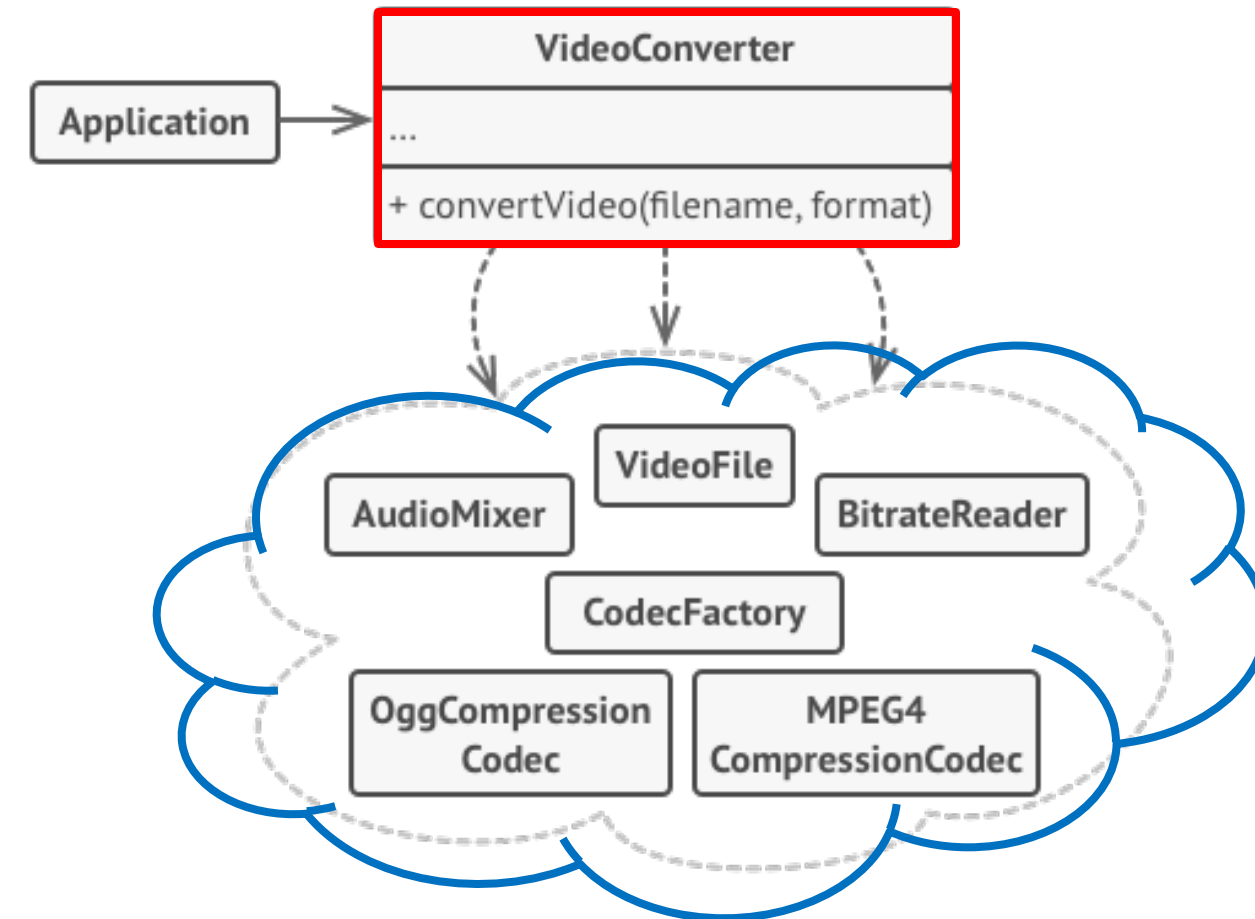


# Facade Example

Say you imported a video conversion package that offered different ways to convert a video file with a number of customizations.

We can create our own **VideoConverter** file that accesses and uses this package while providing our system with a **simple interface** and **hiding the complexity**.

[video\\_package\\_facade\\_example.cpp](#)

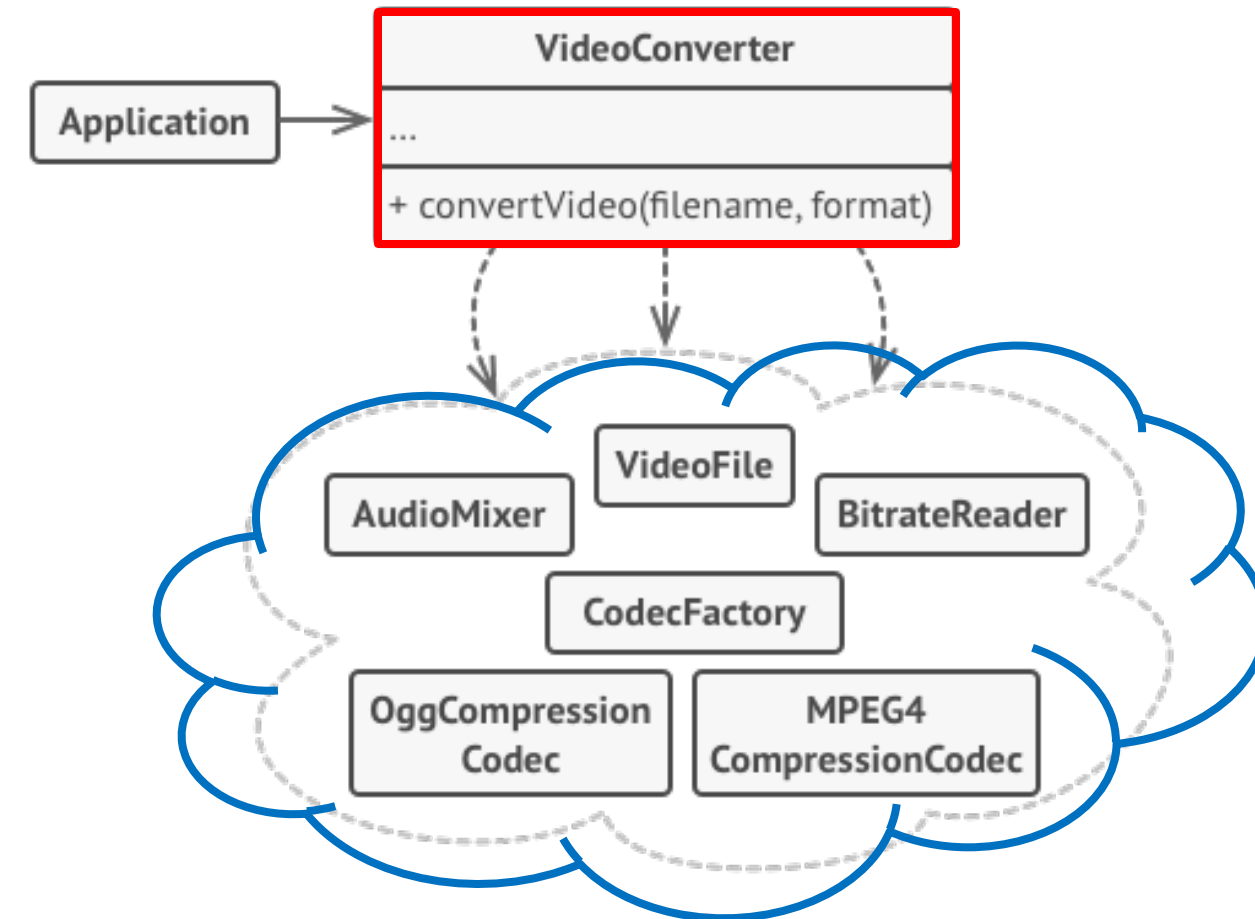


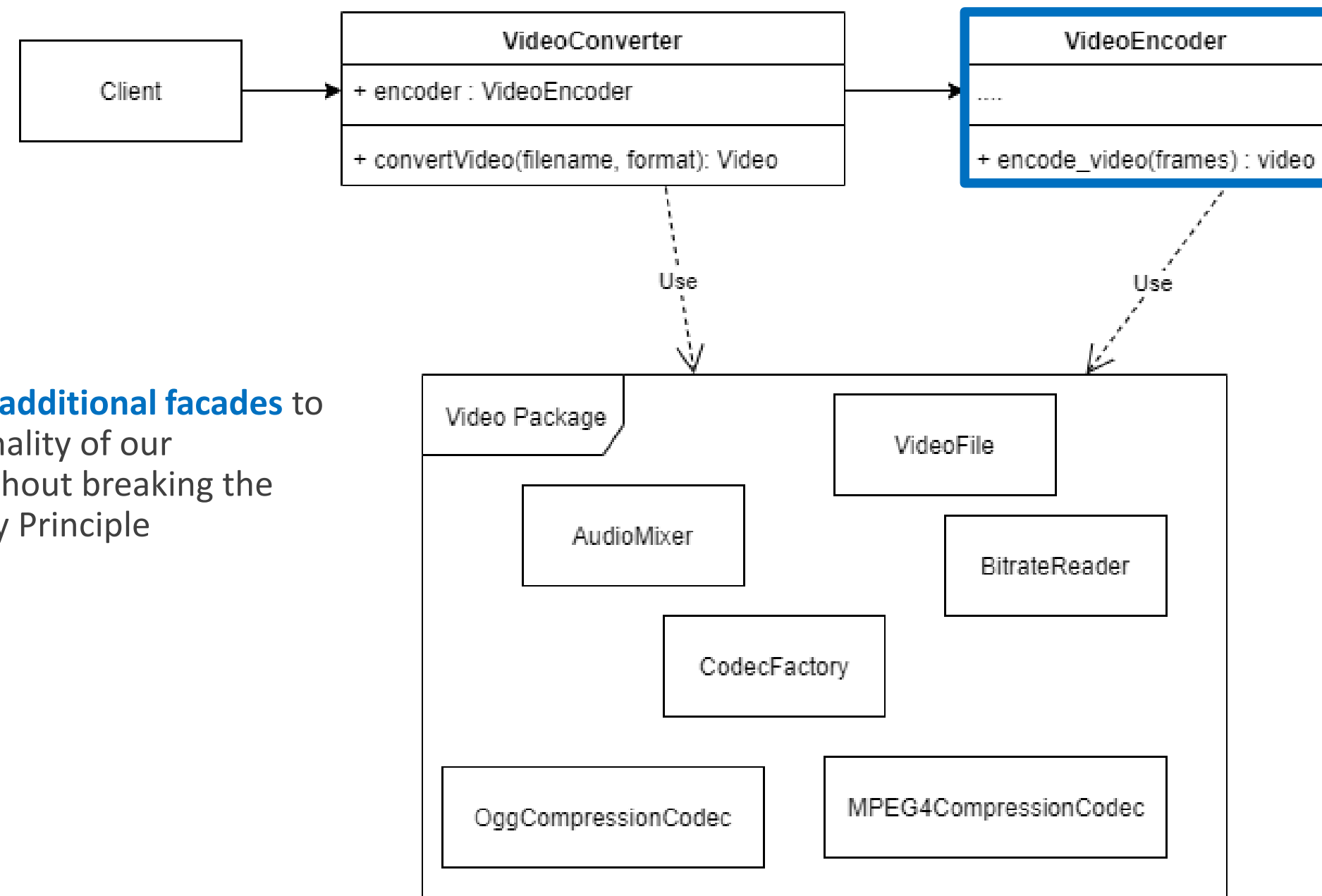
# 外观模式示例

假设你导入了一个视频转换包，该包提供了多种方法来转换视频文件，并支持许多自定义选项。

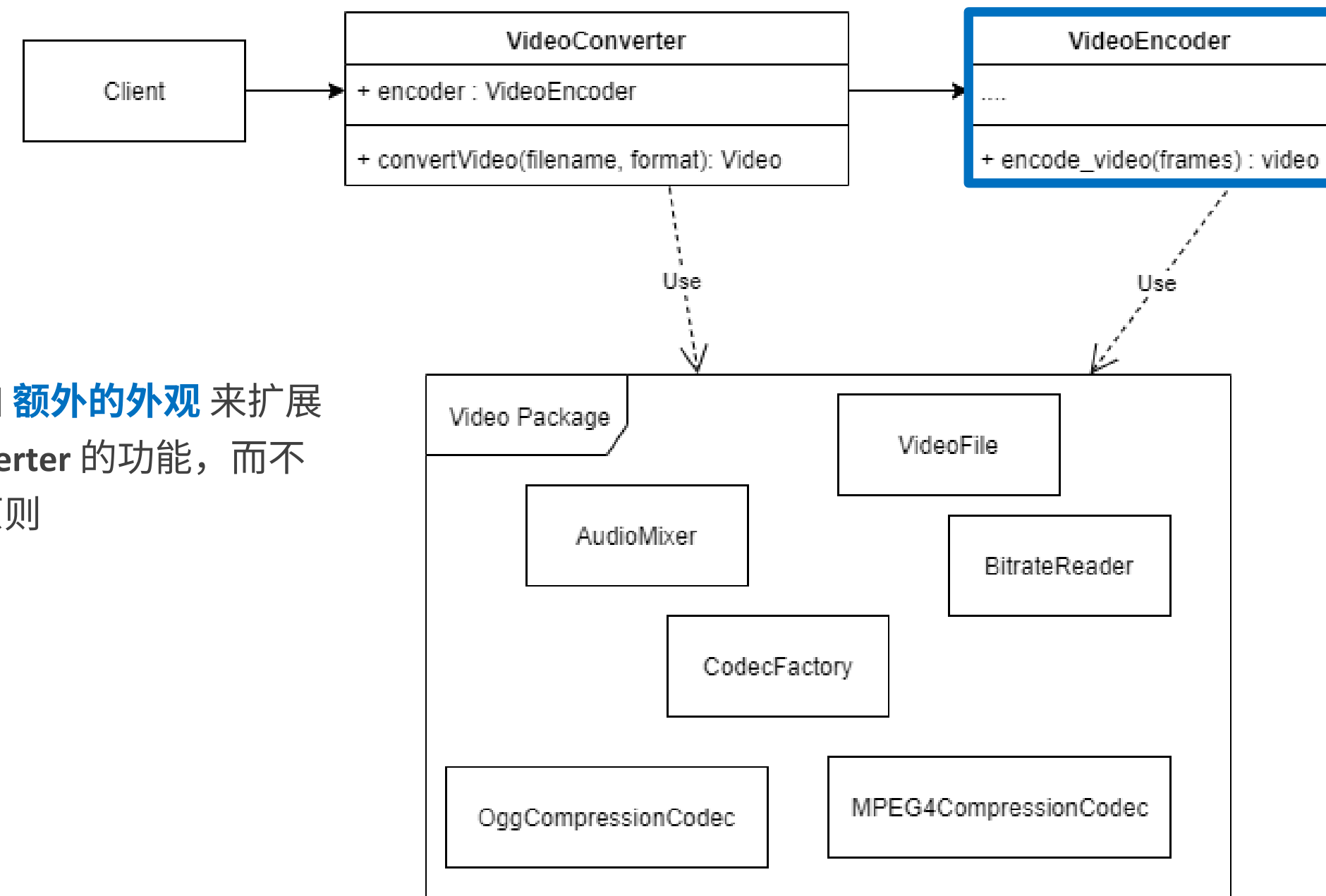
我们可以创建自己的 **VideoConverter** 文件，该文件访问并使用此包，同时为我们的系统提供一个 **简单接口** 和 **隐藏复杂性**。

[video\\_package\\_facade\\_example.cpp](#)





We could even add **additional facades** to expand the functionality of our **VideoConverter** without breaking the Single Responsibility Principle



我们甚至可以添加 **额外的外观** 来扩展我们的 **VideoConverter** 的功能，而不会破坏单一职责原则

# Facade – Why and When do we use it

- Good use of **Encapsulation** and **Data Hiding**
- When we want to **avoid complex architectures** if different parts of your code were **dependent on complex tools** / libraries / sub-systems
- If we only want to use a **small part of a larger more complex system**.
- Can be used to **organize a system into layers**.



# 外观模式——我们为何以及何时使用它

- 良好地运用了**封装和数据隐藏**
- 当我们希望 **避免复杂的架构**时，如果代码的不同部分 **依赖于复杂的工具 / 库 / 子系统**
- 如果我们只想使用一个**更大、更复杂系统的一小部分**。
- 可用于 **将系统组织成分层结构**。



# Facade - Disadvantages

---

- A Facade can become an **extremely complex and large class to maintain**.
- It can become an **epicentre of coupling**



# 外观模式 - 缺点

---

- 外观（Facade）可能变成一个 **极为复杂且庞大的类，难以维护**。
- 可能成为耦合的**中心**



# Bridge

---

WHEN COMPOSITION SAVES THE DAY

# 桥梁

---

当构图拯救了一天

# Bridge

---

A pattern that lets us split a large class, or a set of closely related classes into separate hierarchies.

The class can usually be split into “***Abstractions***” and “***Implementations***”.

Yes. I *Italicized*, **bolded**, increased the sizes and put quotation marks around those words.

No. I am not a monster for doing that. Those words mean something a little different when it comes to the Bridge Pattern.

# 桥接

---

一种设计模式，允许我们将一个大型类或一组紧密相关的类拆分为独立的层次结构。

该类通常可被拆分为“**抽象**”和“**实现**”。

是的。我将这些词用斜体、加粗、放大字号并加上了引号。

不。我那样做并不是怪物。**这些词在谈及桥接模式时，含义略有不同。**

# Bridge example

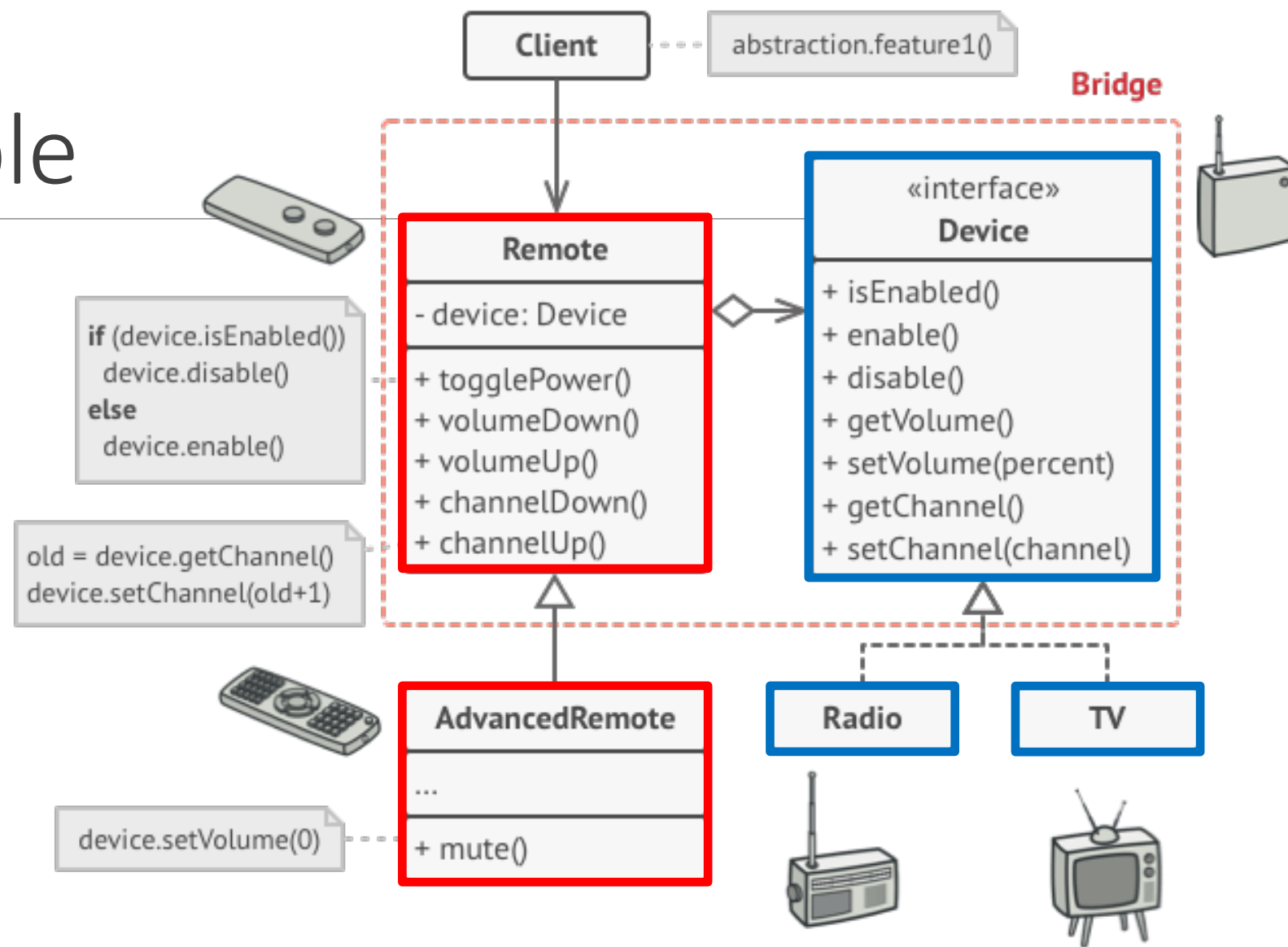
A **Remote** is an **abstraction** of a **device**.

The **Remote** doesn't do anything by itself. It just acts like a **control layer** controlling some **device** implementation.

The Bridge Pattern lets us manage two sets of classes that define an entity where one dimension of it (**Remote**) is coupled with another dimension (**Device**).

Let's implement this example.

device remote bridge example.cpp



# 桥接示例

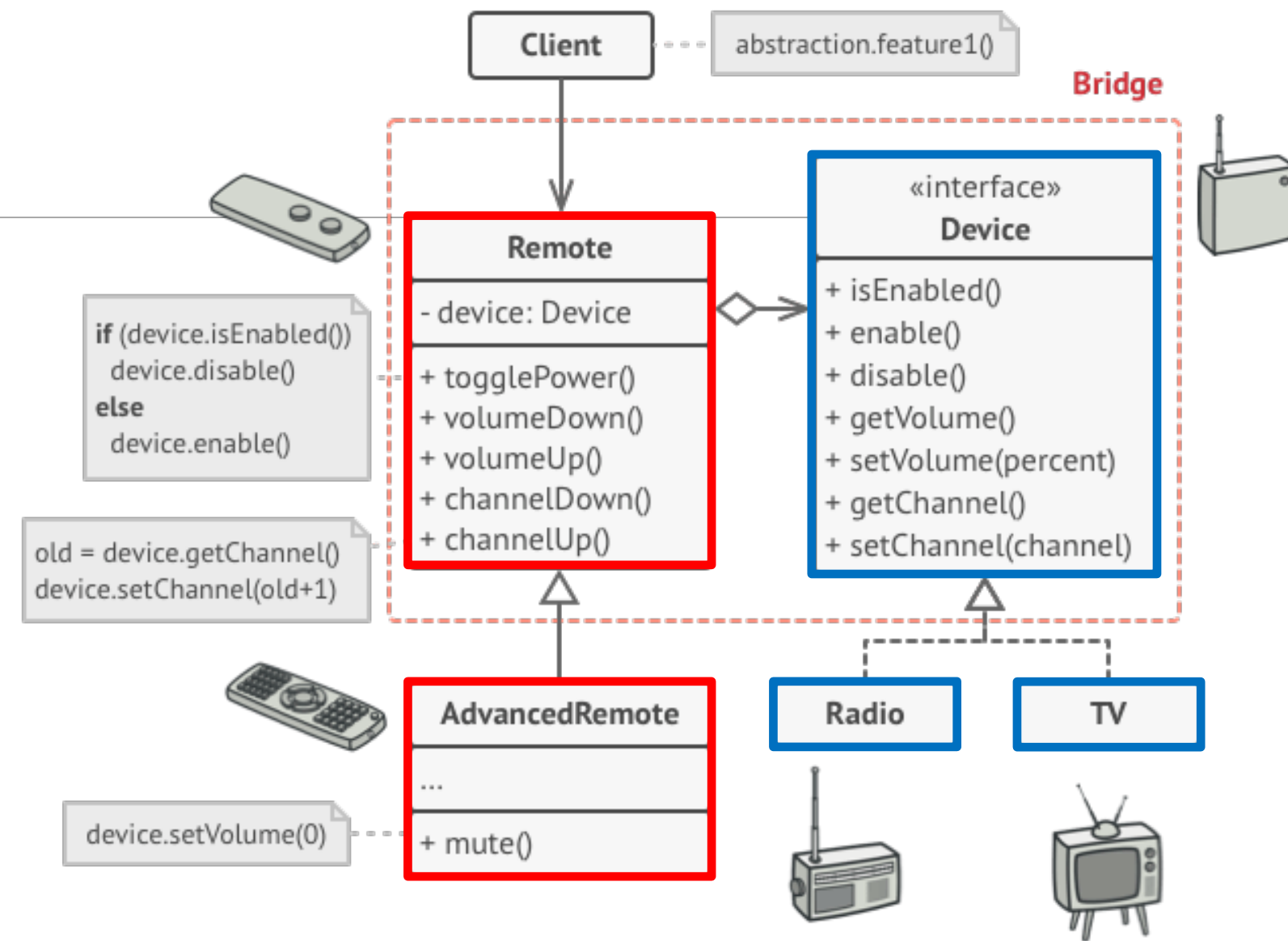
远程对象 是对 抽象 的一种 设备。

远程控制**Remote**本身并不执行任何操作。它只是作为一个**控制层**，用于控制某个**device**的实现。

桥接模式使我们能够管理两组定义实体的类，其中一组维度（Remote）与另一组维度（Device）相耦合。

让我们实现这个示例。

device\_remote\_bridge\_example.cpp

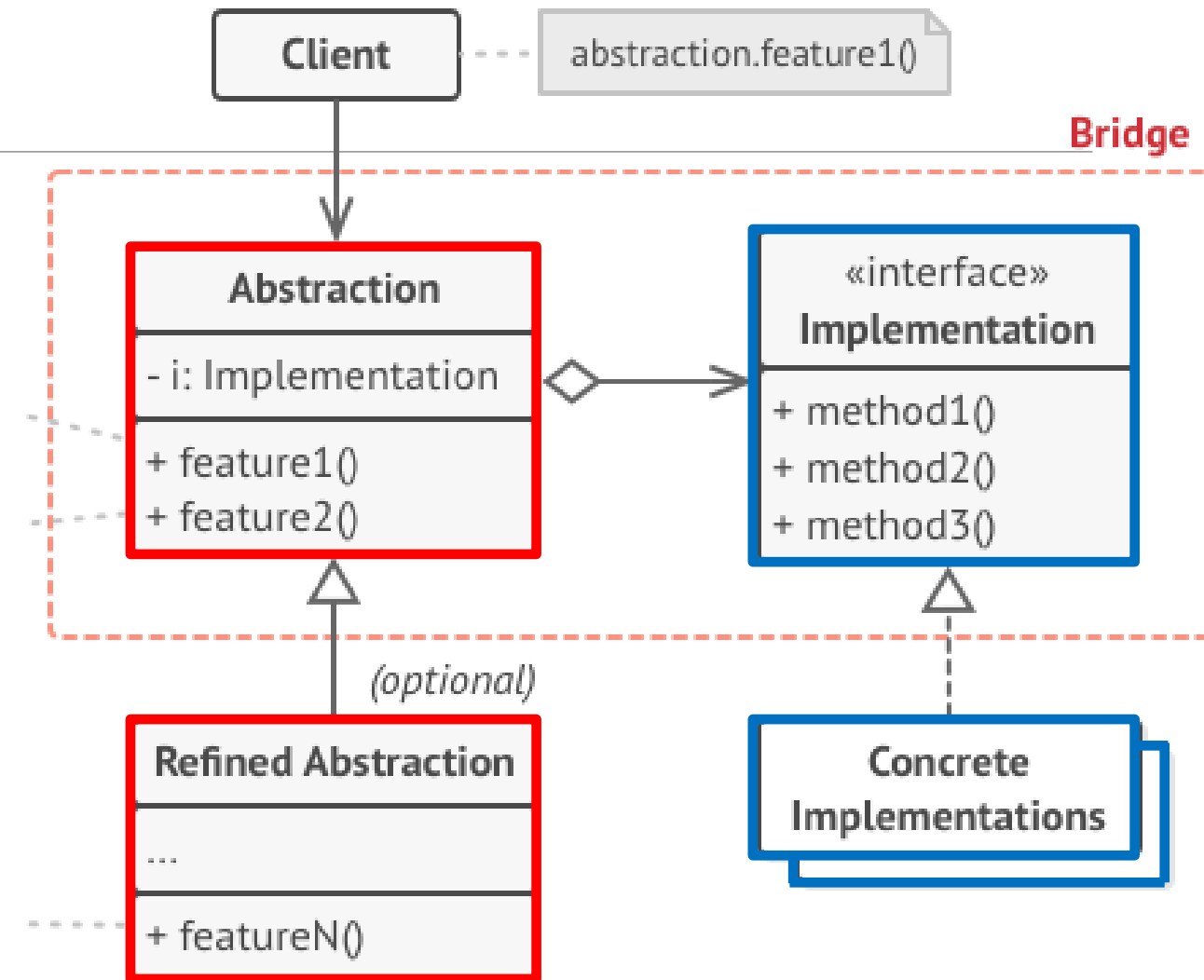


# Bridge

This is what the bridge pattern's UML Diagram looks like.

We have 2 hierarchies. The **Abstraction hierarchy** and an **Implementation Interface hierarchy**.

We say that the **Abstractions** control the **Implementation**. The **Remote** just controls the **Device**. It doesn't do anything else.

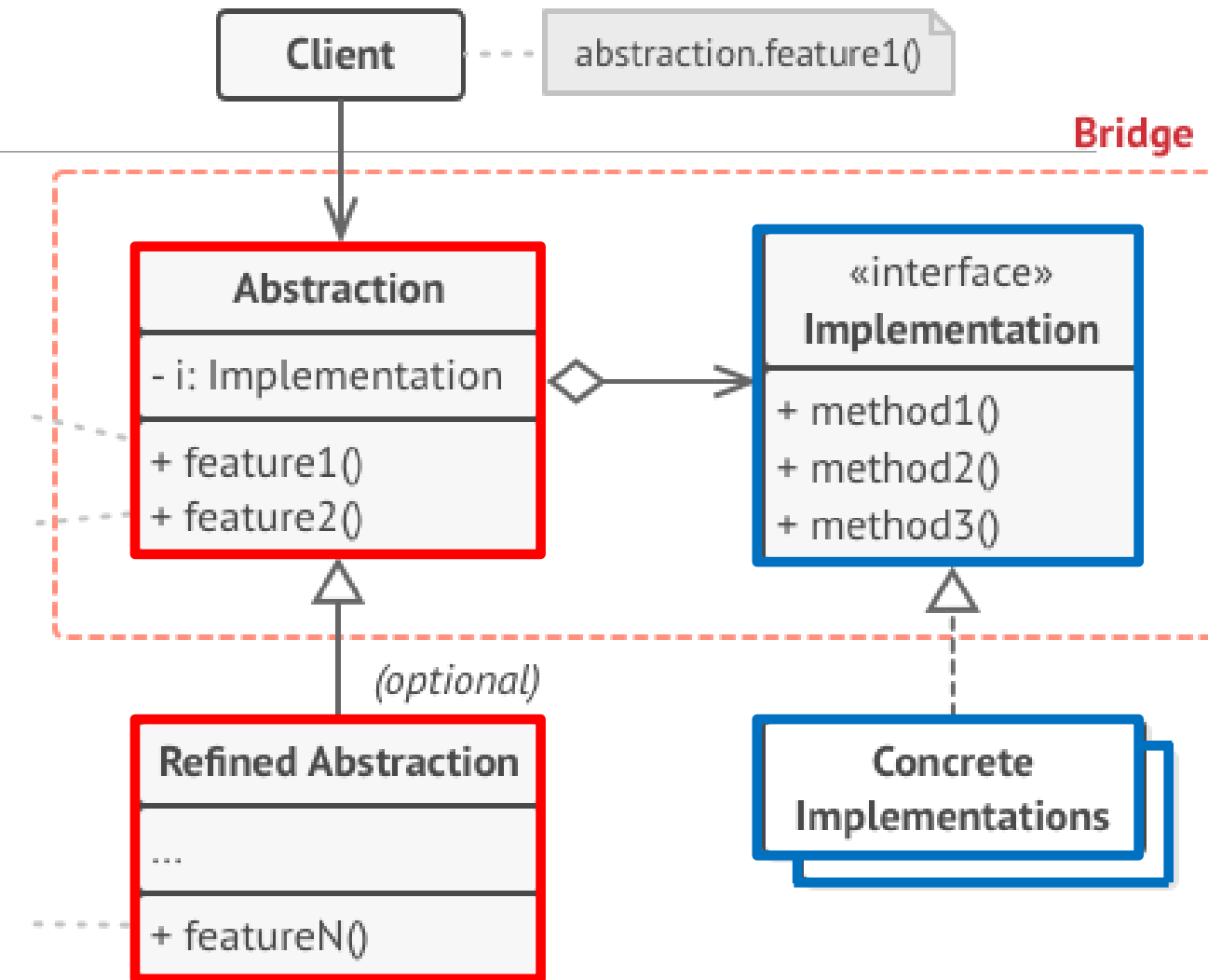


# 桥接

这就是桥接模式的 UML 图表的样子。

我们有 两个层次结构。抽象层次结构 和一个 实现接口层次结构。

我们说 抽象 控制着 实现。远程 只控制 设备，它不做其他任何事情。



# Bridge – When do we use it and Why?

- When we want to divide and conquer a huge class that has many variations.
- This pattern allows us to manage entities that have **multiple orthogonal dimensions** that can each be **extended separately** independent of each other.
- We can use the Bridge Pattern if we want to **switch implementations at run-time**. For example if we want to provide a DataSource a different input/output stream.
- Implements the Open/Closed Principle. Introduce new abstraction and implementations independently.
- Actually it manifests all the SOLID principles.



# 桥接模式——我们何时使用它，为什么使用它？

- 当我们想要分解并处理一个具有多种变化的庞大类时。
- 该模式使我们能够管理具有**多个正交维度**的实体，每个维度都可以**独立扩展**，互不影响。
- 如果我们希望在运行时切换实现，则可以使用桥接模式。例如，为数据源提供不同的输入/输出流。
- 符合开闭原则。可独立引入新的抽象和实现。
- 实际上它体现了所有的 SOLID 原则。



## Bridge – Disadvantages

Sometimes the code can get really complex if we have a highly cohesive class where the different dimensions are not really independent of each other.

The idea of what our “**abstraction**” and our “**implementation**” is can get vague and artificial sometimes. Making the design difficult to understand.



## 桥接——缺点

有时，如果一个类的内聚性很高，并且各个维度之间并非真正独立，代码可能会变得非常复杂。

我们对“**抽象**”和“**实现**”的理解有时会变得模糊且不自然，从而使设计难以理解。



# Let's Recap...

---

## Proxy

A wrapper (can be one of many) that let's us **control access to a service** that has some limitations.

## Facade

A class that encapsulates a complex system or API and hides its implementation details. Provides a **simple interface that makes it easy to use**. Usually we only want to use 1 or 2 features anyway.

## Bridge

When we want to **separate a large class** or a set of highly closely related classes **into two separate hierarchies**, an “Abstraction” and an “Implementation”.

# 让我们回顾一下...

---

...

## 代理

一个包装器（可以是多个中的一个），它使我们能够**控制对具有某些限制的服务的访问**。

## 外观

一个封装了复杂系统或 API 并隐藏其实现细节的类。提供一个**简单的接口，使其易于使用**。通常我们 anyway 只想使用其中的一两个功能。

## 桥接

当我们想要将一个大型类或一组高度相关的类分离为两个独立的层次结构时，即“抽象”和“实现”。

# Factory pattern

---

COMP3522 OBJECT ORIENTED PROGRAMMING 2  
WEEK 10

# 工厂模式

---

COMP3522 面向对象程序设计 2 第10周

# Categorizing Design Patterns

## ❑ Behavioural

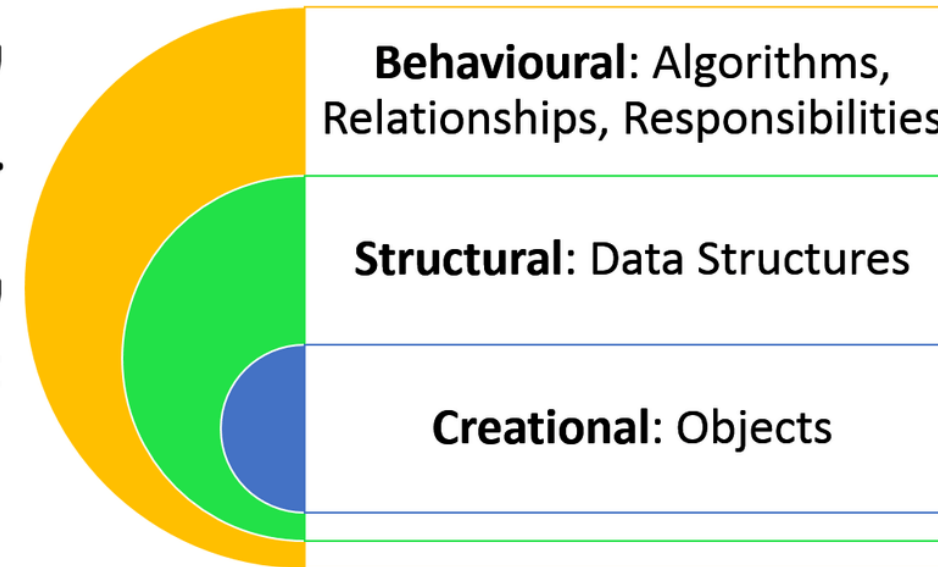
Focused on communication and interaction between objects. How do we get objects talking to each other while minimizing coupling?

❑ **Structural** How do classes and objects combine to form structures in our programs? Focus on architecting to allow for maximum flexibility and maintainability.

## ❑ **Creational (We are looking at these!)**

All about class instantiation. Different strategies and techniques to instantiate an object, or group of objects

Design Patterns



# 设计模式分类

## ❑ 行为型

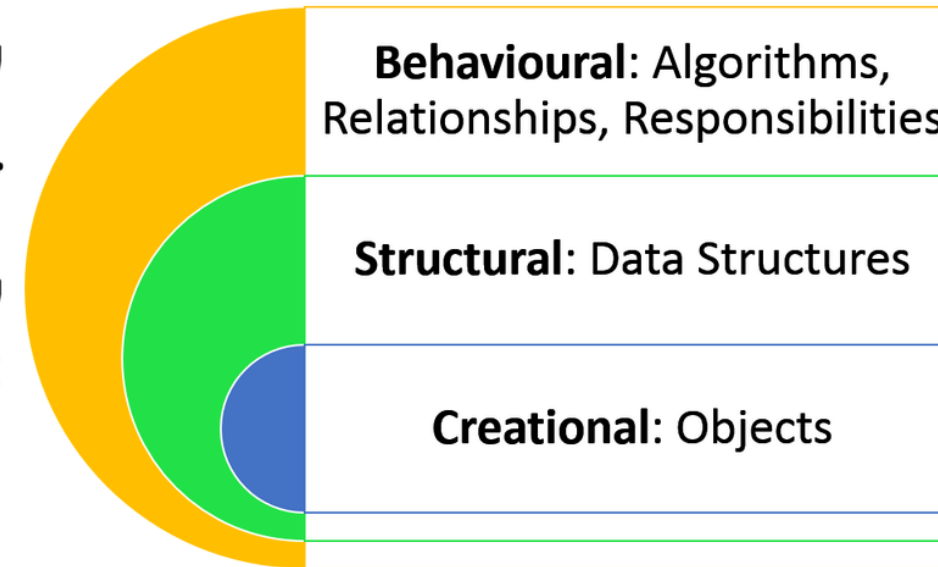
关注对象之间的通信与交互。我们如何让对象相互通信的同时最小化耦合？

❑ **结构型** 类和对象如何在程序中形成结构？重点在于架构设计，以实现最大的灵活性和可维护性。

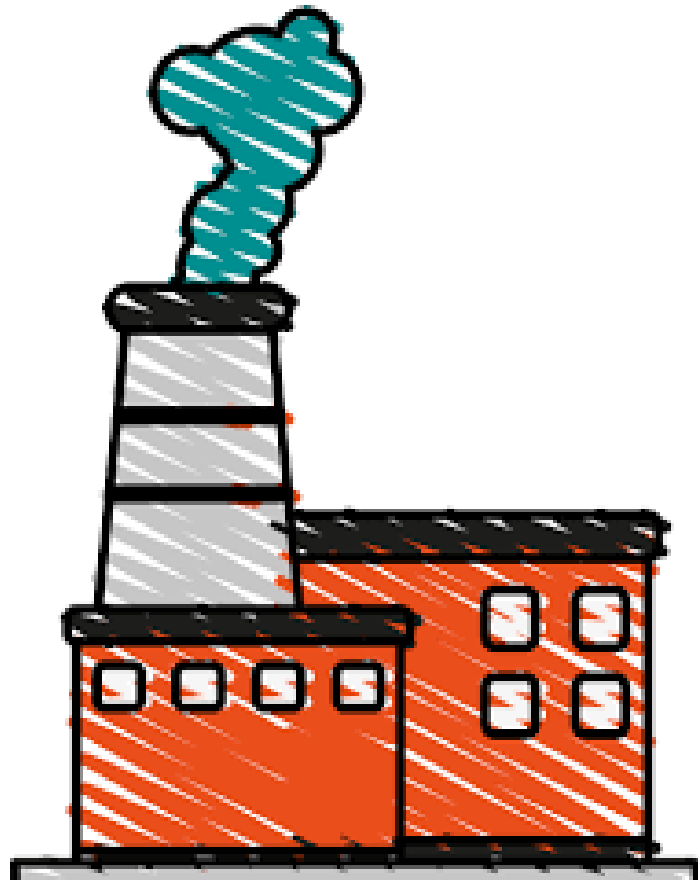
## ❑ **创建型 (我们正在研究这些!)**

关于类实例化的所有内容。实例化一个对象或一组对象的不同策略和技术

Design Patterns

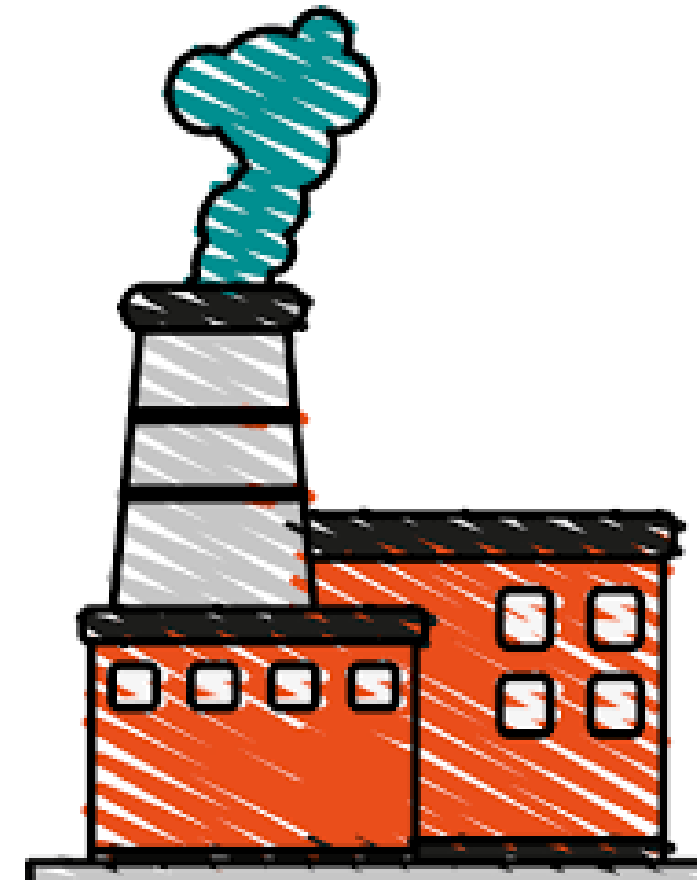


## First let's clarify the word Factory...



- Something that is responsible for making things
- In this case objects.
- The word **factory** is used a lot, don't mix it up with the **Factory Pattern** or the **Abstract Factory Pattern**. They are also responsible for making objects but are different.
- A Factory could refer to:
  - The pattern(s)
  - A class that creates objects
  - A static/class function that generates objects.

## 首先让我们澄清一下“工厂”这个词……



- 负责制造事物的某物
- 在这种情况下指的是对象。
- “**factory**”这个词使用得很频繁，不要将其与“**Factory Pattern**”或“**Abstract Factory Pattern**”混淆。它们也负责创建对象，但有所不同。
- 工厂可能指：
  - 该模式（们）
  - 一个创建对象的类
  - 一个生成对象的静态/类函数。

## Some more lingo – Creation Method

- Any **method or function** that is responsible for creating an object is a **factory**.
- Different from the constructor, although you can argue that a constructor is a creation method too. A **creation method** can be a wrapper around a **constructor call**. (Check the example below)
- A **creation method** doesn't have to make new instances of a class, it can return a cached object or even re-use objects from a collection (Object Pool Pattern).
- Resist the urge to call it a Factory Method. The Factory Pattern is also known as the Factory Method Pattern and it can cause all sorts of confusion.

```
static Asteroid generate_random_asteroid(int min_speed, int max_speed){  
    int speed = rand()%(max_speed - min_speed + 1) + min_speed;  
    Asteroid asteroid;  
    asteroid.speed = speed;  
    return asteroid;  
}
```

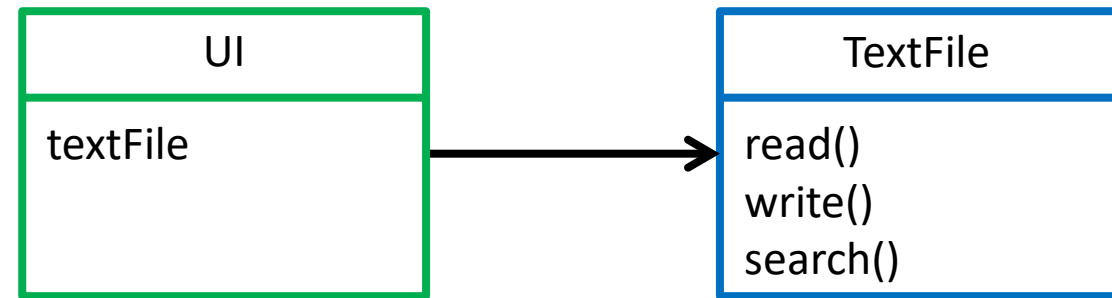
## 更多术语——创建方法

- 任何 **负责创建对象的方法或函数都是一个工厂**。
- 不同于构造函数，尽管你也可以说构造函数本身也是一种创建方法。一个**创建方法**可以是**构造函数调用**的封装。（请查看下方示例）
- 一个**创建方法**不一定需要创建类的新实例，它可以返回一个缓存的对象，甚至可以从集合中复用对象（对象池模式）。
- 避免将其称为工厂方法。工厂模式也被称为工厂方法模式，这样称呼可能会引起各种混淆。

```
静态小行星生成_随机_小行星(整型 最小_速度, 整型 最大_速度){整型 速度 = rand()%  
(最大_速度 - 最小_速度 + 1) + 最小_速度; 小行星 小行星; 小行星.速度 = 速度; 返回  
小行星;  
}
```

# Remember the dependency inversion principle?

---

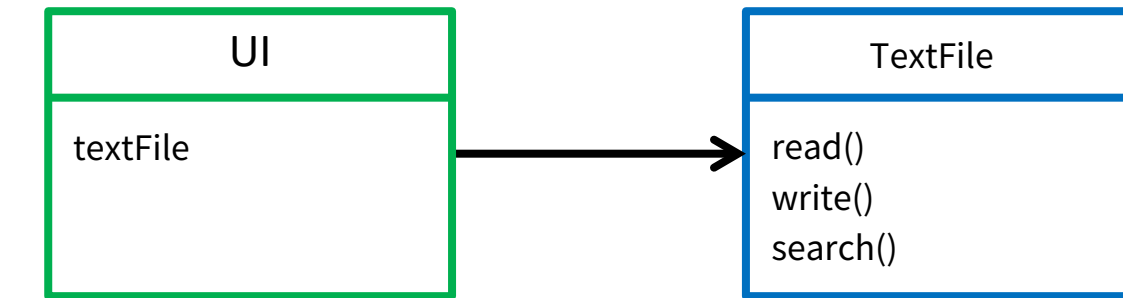


Say we were developing an app with a **UI** that was populated with some data from a **text file**.

After a few releases and years of development, for some reason we decide to switch out to a more secure source of data, perhaps an encrypted **JSON** file or even perhaps a **SQLite** Database. We would have to edit all the modules/classes that dealt with our app's **UI**!

# 还记得依赖倒置原则吗？

---



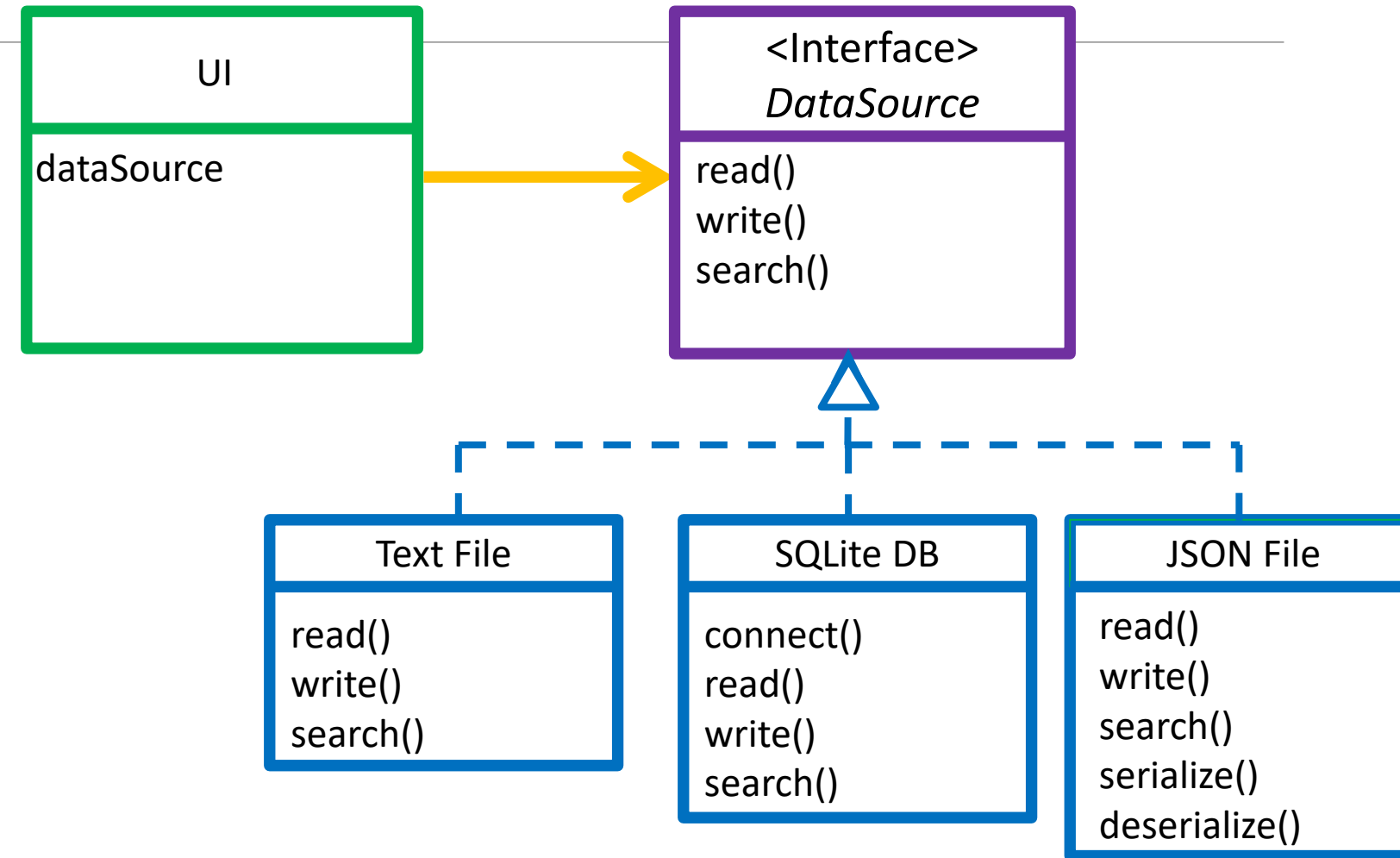
假设我们正在开发一个应用程序，其 **UI** 界面由来自某个 **文本文件** 的数据填充。

经过几次发布和数年的开发，由于某些原因，我们决定切换到更安全的数据源，也许是一个加密的 **JSON** 文件，甚至可能是 **SQLite** 数据库。我们将不得不修改所有处理应用程序 **UI** 的模块/类！

# Remember the dependency inversion principle?

We can decouple our system to instead depend on a **data source** which is an **abstraction** that **hides** different **data sources**.

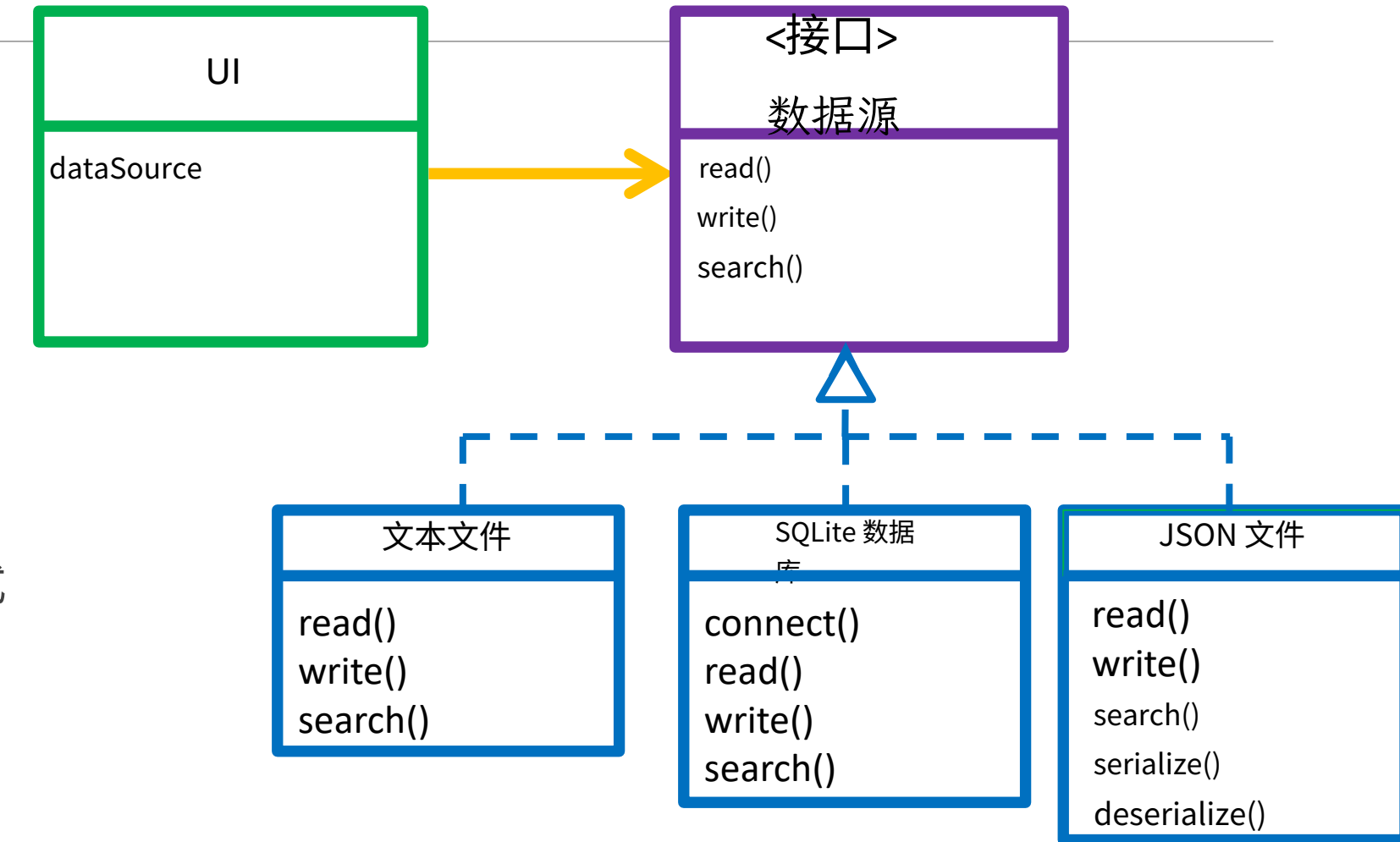
Our **UI** would just have to be **dependent on a common interface** provided by the **Data Source** and not be concerned with how that data source is implemented



# 还记得依赖倒置原则吗？

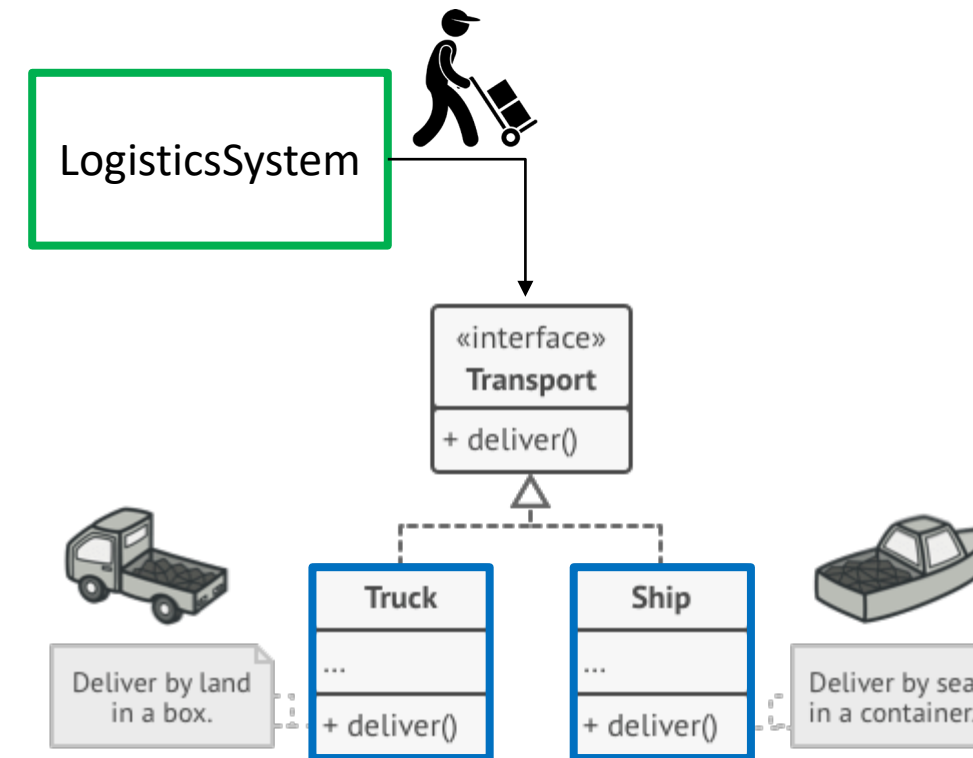
我们可以将系统解耦，转而依赖一个 **数据源**，它是一种 **抽象**，用于 **隐藏** 不同 **数据源**。

我们的 **UI** 只需依赖于 **依赖的通用接口**，该接口由 **数据源** 提供，而无需关心该数据源的具体实现方式



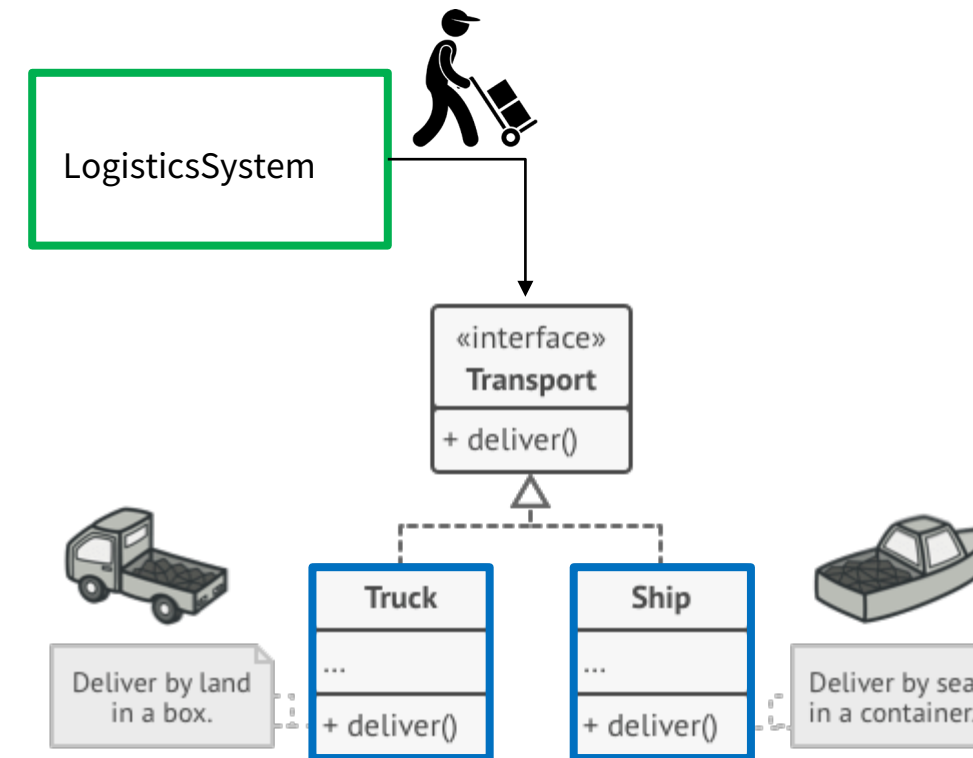
# Factory Pattern

- A way to **decouple client code** that **uses** an object or a service, from the code that **creates** an object or a service.
- Consider a scenario where a system is managing the logistics of delivering goods.
- If inheritance is done correctly, all a **Logistics System** cares about is depending on the **Transport** Interface, not the **concrete transports** (**Truck** or **Ship**). This follows **the Dependency Inversion Principle**.



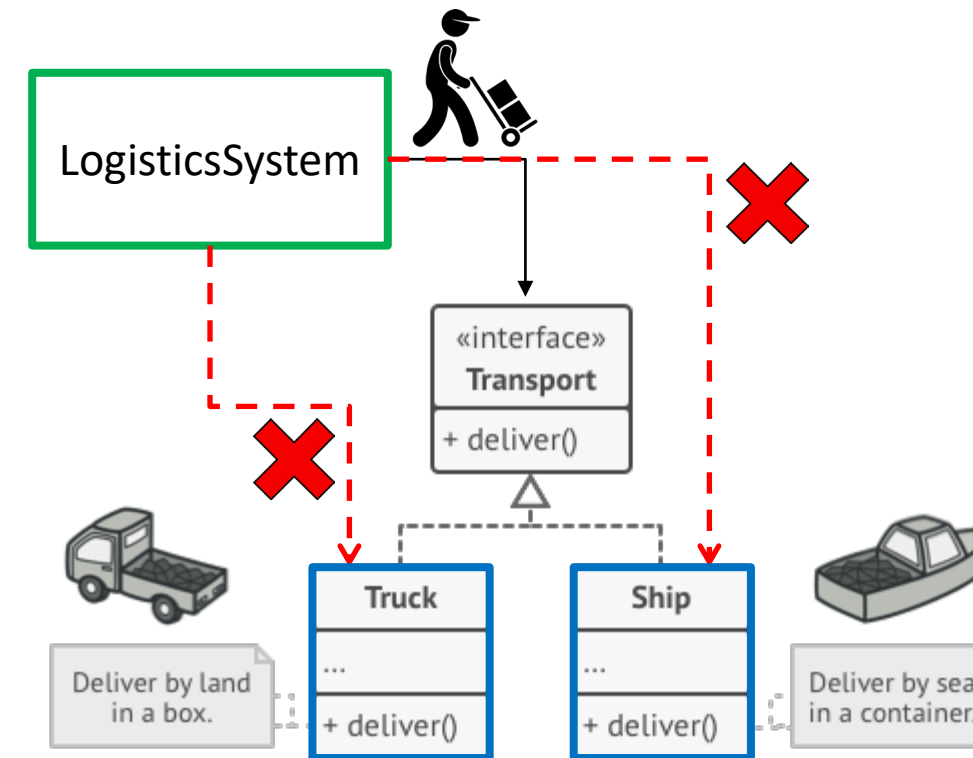
# 工厂模式

- 一种 **将使用** 对象或服务的客户端代码与 **使用** 该对象或服务的代码和 **创建** 该对象或服务的代码进行解耦的方法。
- 考虑一个系统正在管理货物运输物流的场景。
- 如果正确地实现了继承，**物流系统** 所关心的仅仅是依赖于**运输接口**，而不是具体的**运输工具**（**卡车**或**轮船**）。这遵循了**依赖倒置原则**。



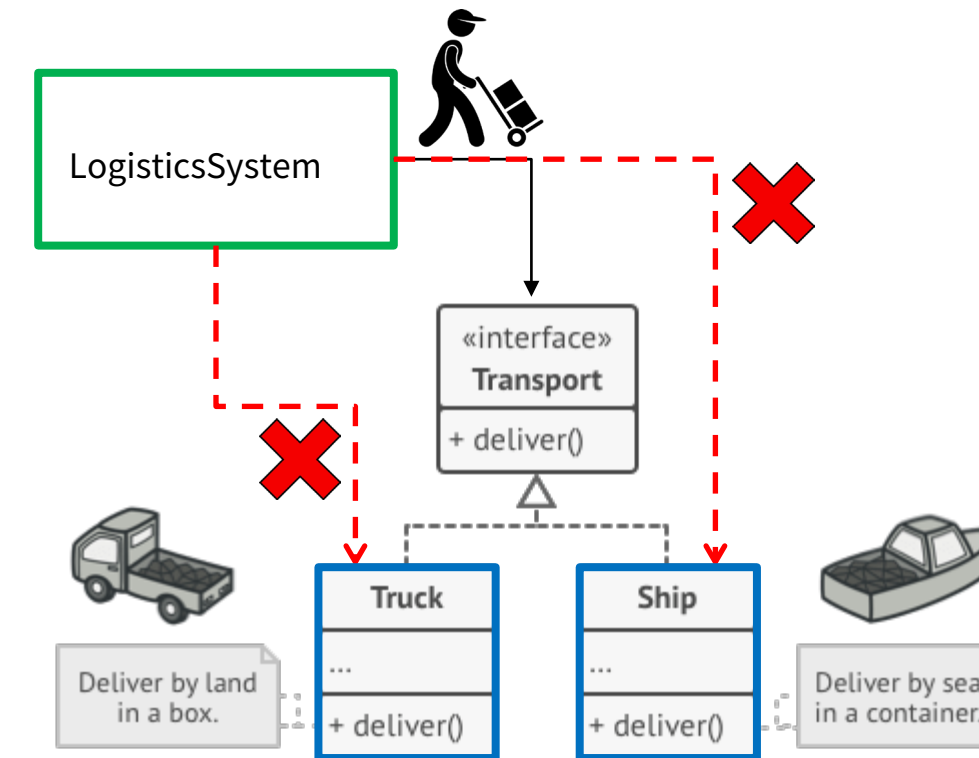
# Factory Pattern

- But at some point we need to create the **concrete class**. If the **Logistics System** did that, then it would need to call the **Truck()** or **Ship()** constructors.
- This would make **Logistics System dependent on the concrete classes** and break the Dependency Inversion Principles.
- A **high level module** is now dependent on a **low level module**



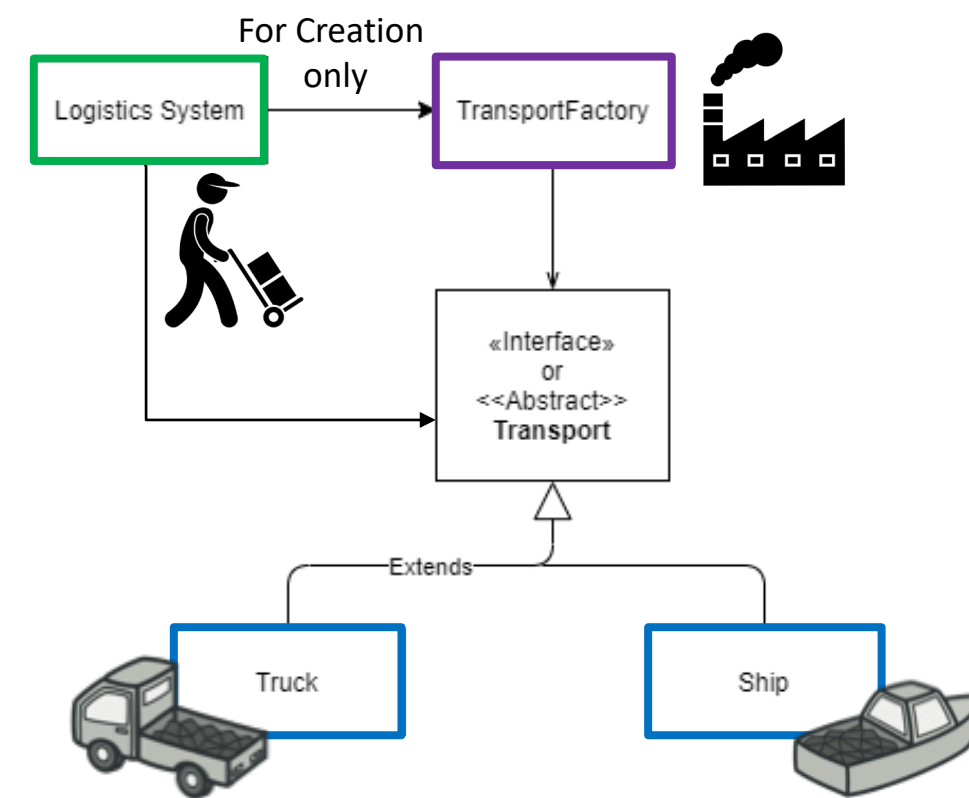
# 工厂模式

- 但在某个时刻，我们需要创建具体类。如果物流系统自行完成这一操作，那么它就需要调用**Truck()**或**Ship()**构造函数。
- 这将导致物流系统依赖于具体类，并违反依赖倒置原则。
- 一个高层模块现在依赖于一个低层模块



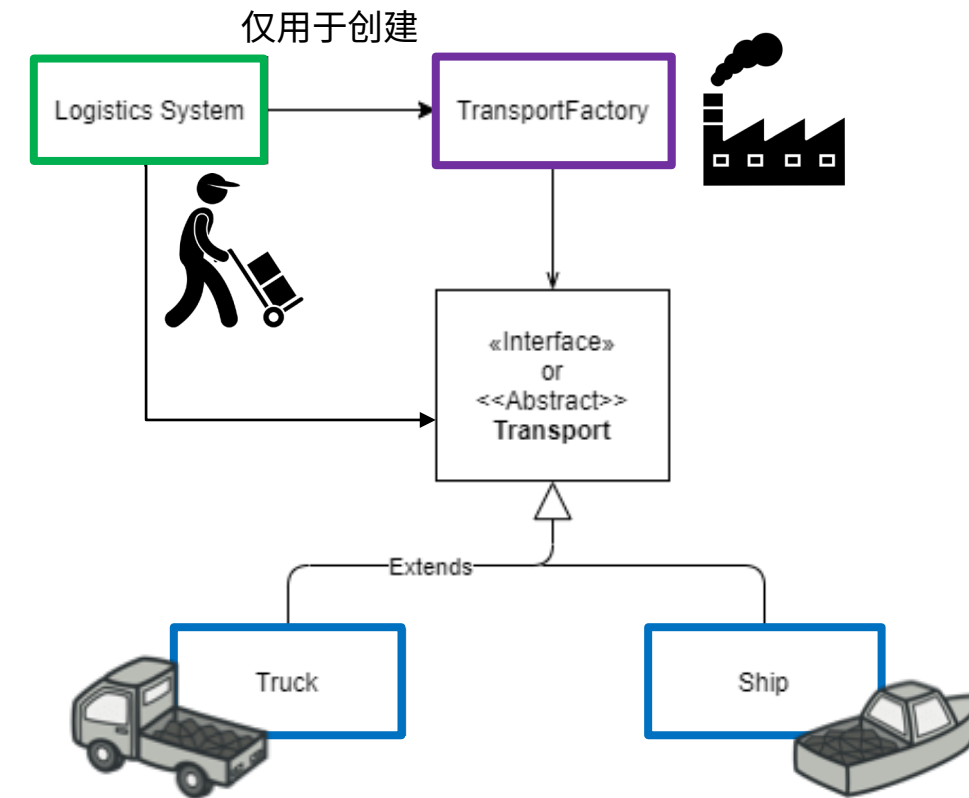
# Factory Pattern

- So when it comes to creating these objects we need another entity to handle the responsibility of creating a specific type of **Transport**.
- We need a **factory** which will create a concrete **Truck/Ship** and return it to the **Logistics System**.
- As far as the **Logistics System** is concerned. It doesn't necessarily know that it is receiving a **Truck/Ship**. All it needs to know is that it is receiving a **Transport**.



# 工厂模式

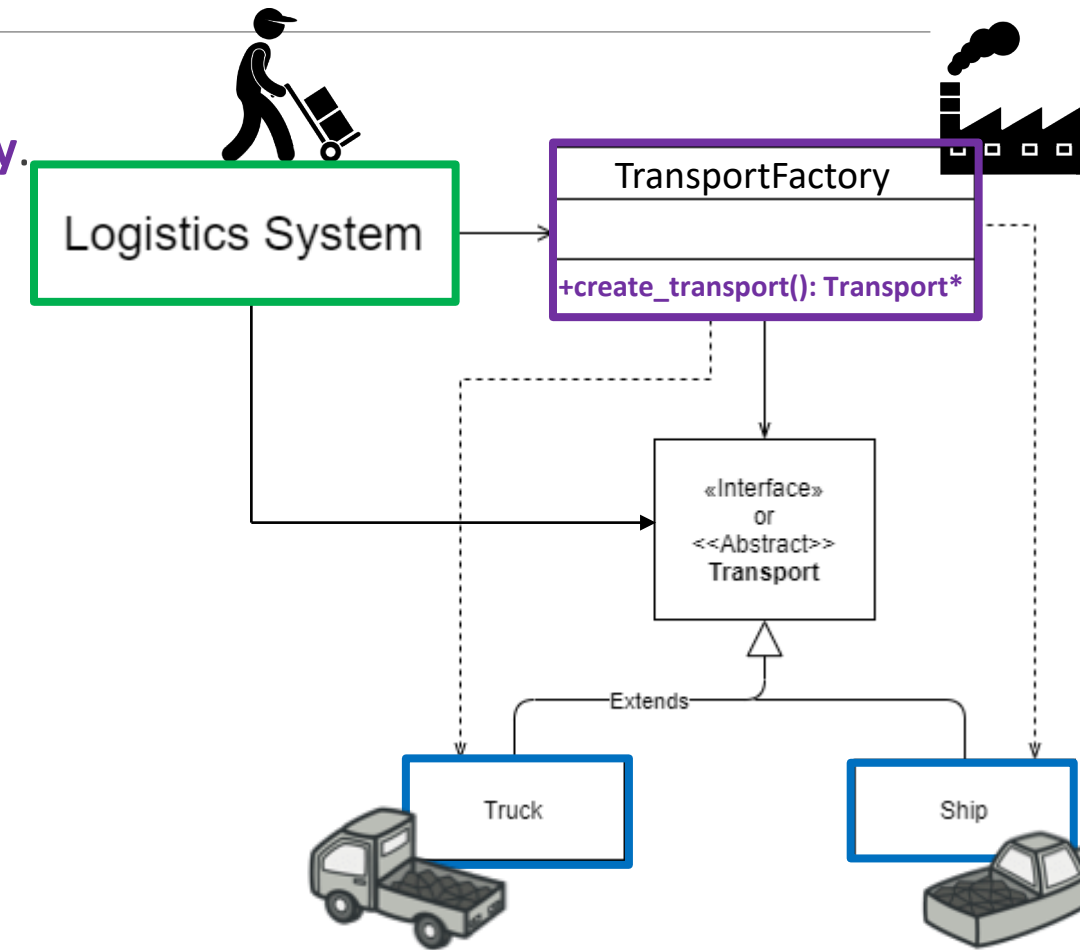
- 因此，在创建这些对象时，我们需要另一个实体来负责创建特定类型的 **Transport**。
- 我们需要一个 **工厂**，它将创建一个具体的 **卡车/船舶** 并将其返回给 **物流系统**。
- 就**物流系统**而言，它不一定知道正在接收的是**卡车/船舶**。它只需要知道正在接收的是一个**运输工具**即可。



# Factory Pattern

- This still isn't enough. All we have done is move the call to our **Truck()**, **Ship()** constructor to the **TransportFactory**. This isn't that useful.
- Say our **TransportFactory** has a **create\_transport()** method. Now this method will have a massive **if-else** or switch statement block where it will be dependent on the concrete Truck and ship Classes

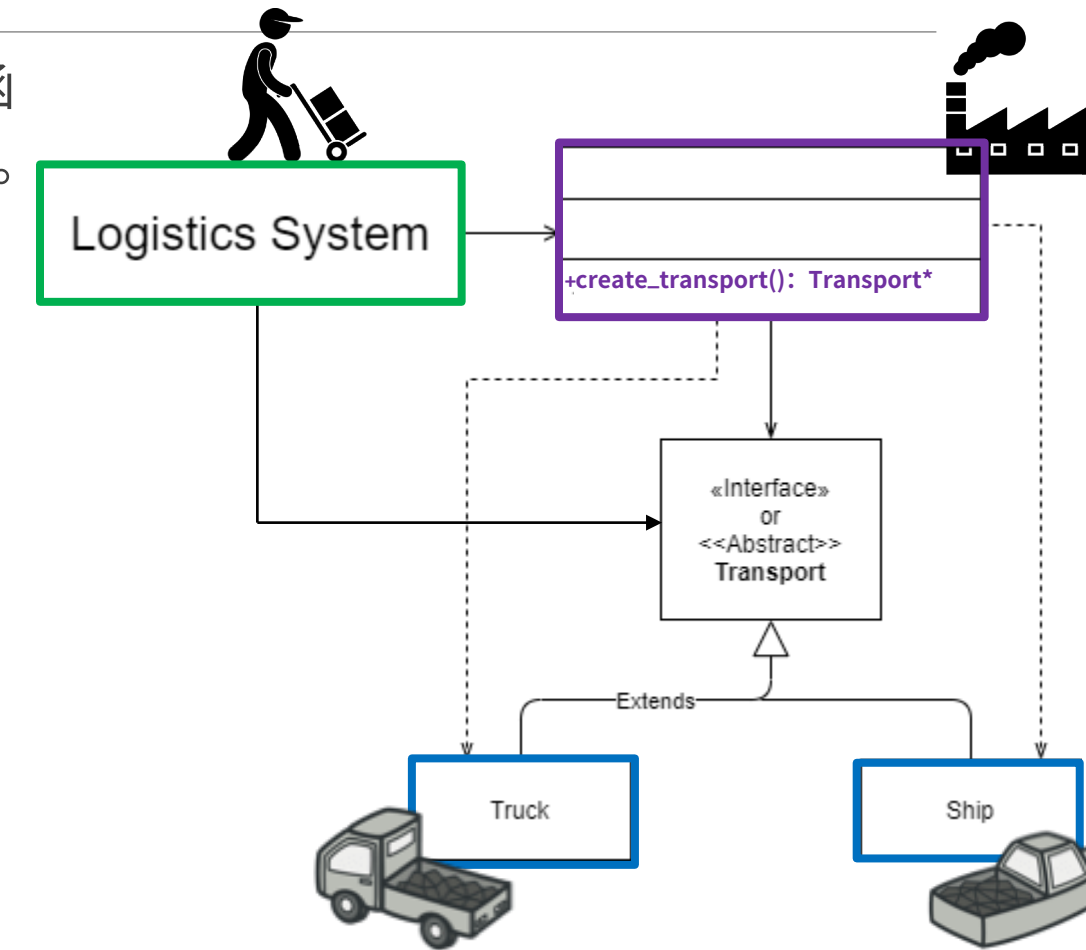
```
Transport* create_transport(){  
    if (current_type == Ship)  
        return new Ship();  
    else  
        return new Truck();  
}
```



# 工厂模式

- 这仍然不够。我们所做的只是将对**Truck()**、**Ship()**构造函数的调用移到了**TransportFactory**中。这并没有太大用处。
- 假设我们的 **TransportFactory** 拥有一个 **create\_transport()** 方法。现在，该方法将包含一个庞大的 **if-else** 或 **switch** 语句块，其具体实现依赖于 Truck 和 Ship 类

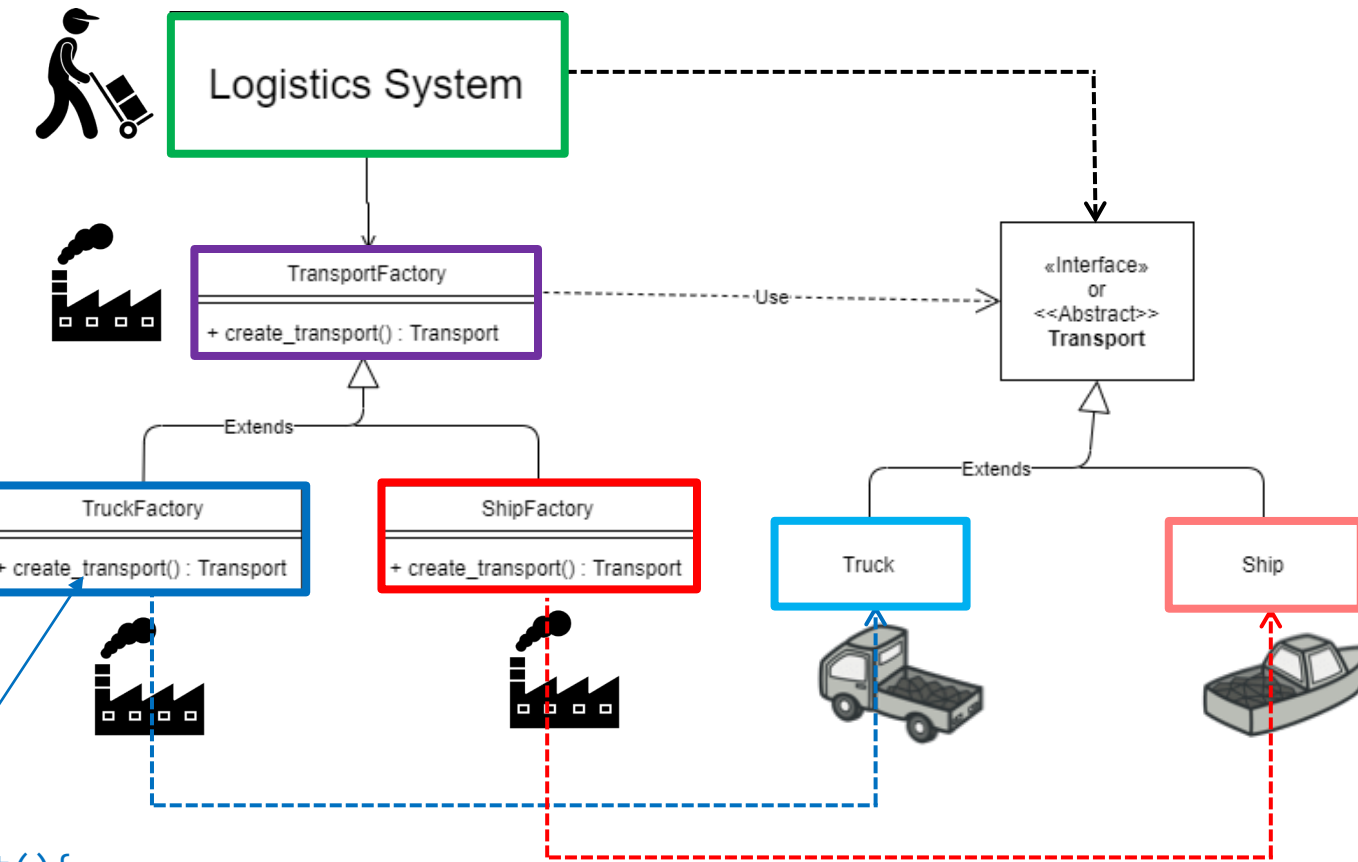
```
Transport* create_transport(){  
    if (current_type == Ship)  
        return new Ship();  
    else  
        return new Truck();  
}
```



# Factory Pattern

- That's better.
- We can simply inherit from **TransportFactory** and override the **create\_transport()** method.
- All we need to do is supply the **Logistics System** with the right factory. (Which can be done by a controller).
- As far as the **Logistics System** is concerned it needs an object with the **TransportFactory** interface.

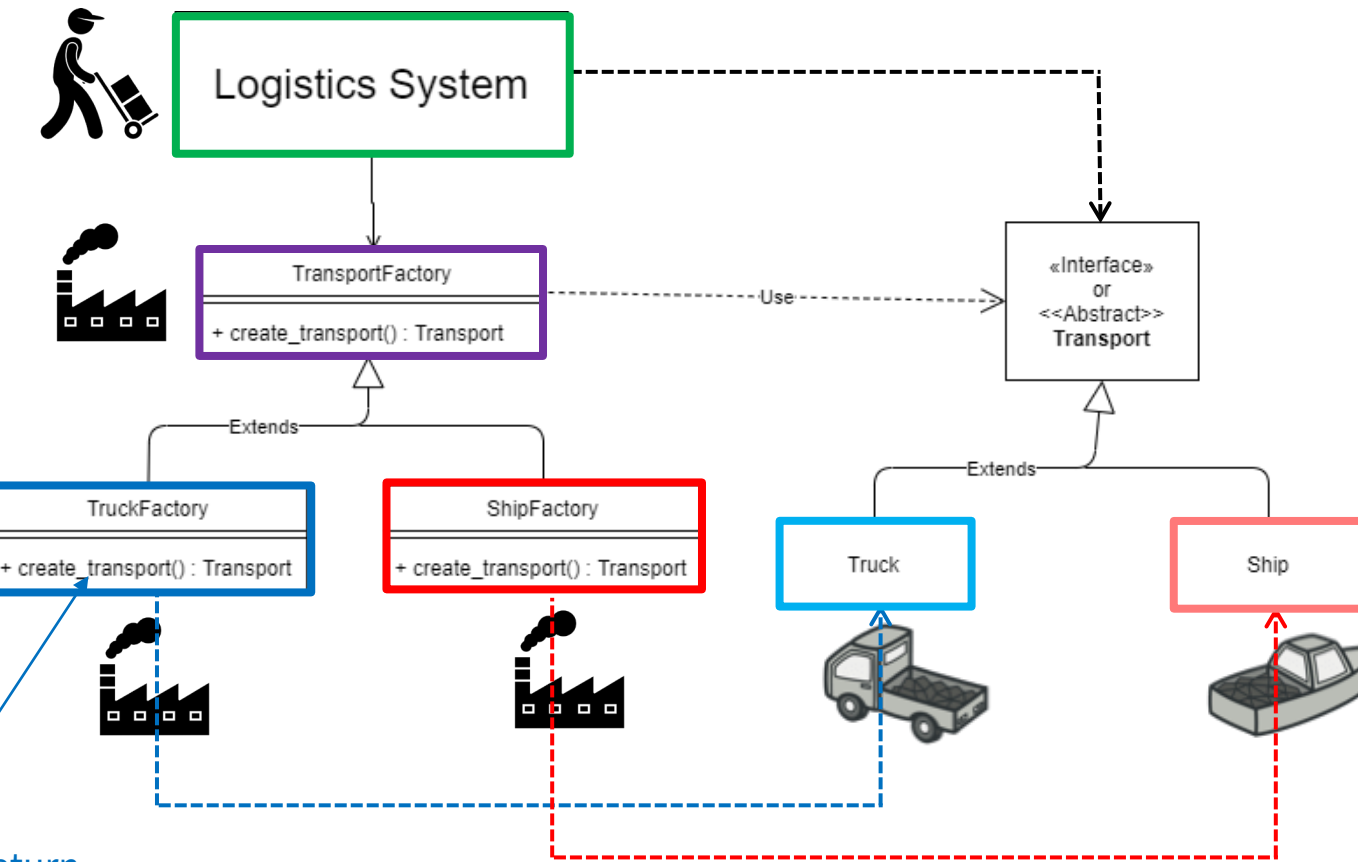
```
Transport create_transport(){  
    return Truck();  
}
```



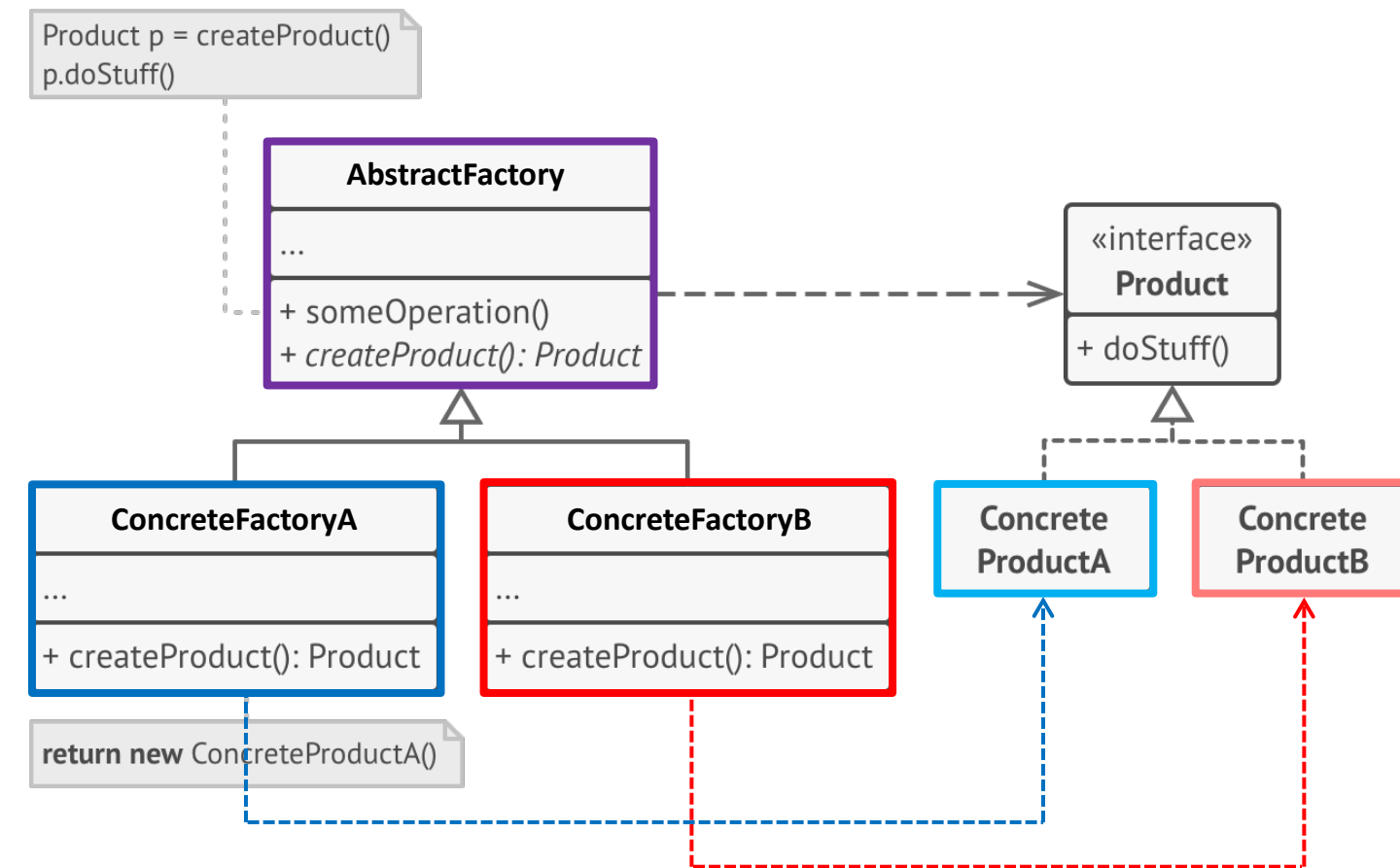
# 工厂模式

- 这样更好。
- 我们可以简单地继承 **TransportFactory** 并重写 **create\_transport()** 方法。
- 我们只需向 **Logistics System** 提供正确的工厂即可。（这可以通过控制器完成）。
- 就 **Logistics System** 而言，它只需要一个具有 **TransportFactory** 接口的对象。

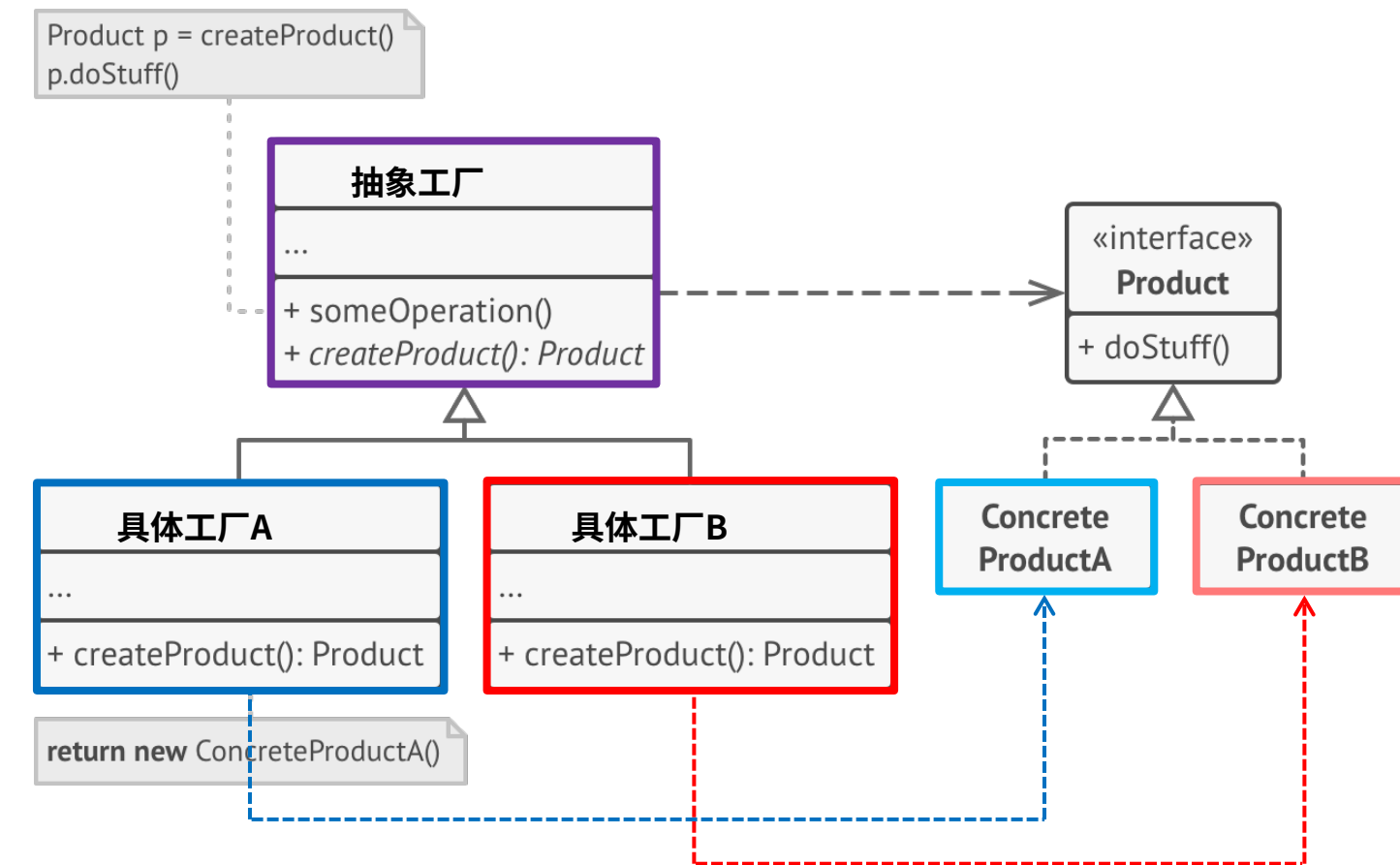
```
Transport create_transport(){ return  
Truck();  
}
```



# Factory Pattern



# 工厂模式



# Let's Build a Factory Pattern Together.

Let's say we need to implement a Factory Pattern for a website hosting a forum messaging board. This website creates a new user object when the user logs in, signs up to be a member or views the forum as a guest.

Our system can deal with two kinds of users. A **Guest** and a **Member**.

A **Guest** has the following **permissions**:

- Read Posts
- Like
- Share Posts
- Flag Posts

A **Member** can do everything a **Guest** can, but they can also **write** posts.

**Step 1:** Let's create a UML Class diagram depicting the **User hierarchy**.

# 让我们一起来构建一个工厂模式。

假设我们需要为一个托管论坛留言板的网站实现一个工厂模式。当用户登录、注册成为会员或以访客身份浏览论坛时，该网站会创建一个新的用户对象。

我们的系统可以处理两种类型的用户。一种是**访客**和一种是**会员**。

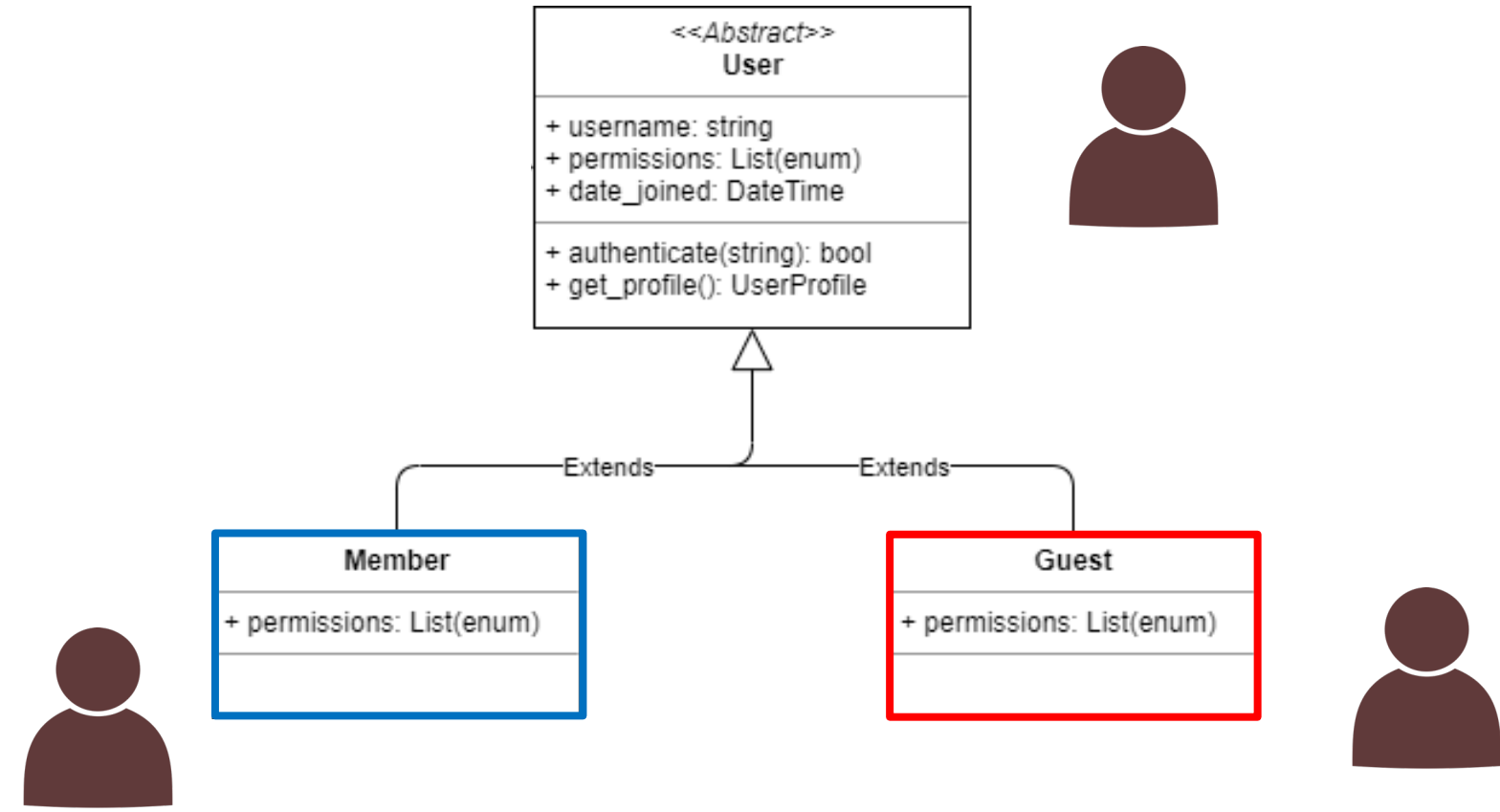
一个**访客**具有以下**权限**:

- 阅读帖子
- 点赞
- 分享帖子
- 标记帖子

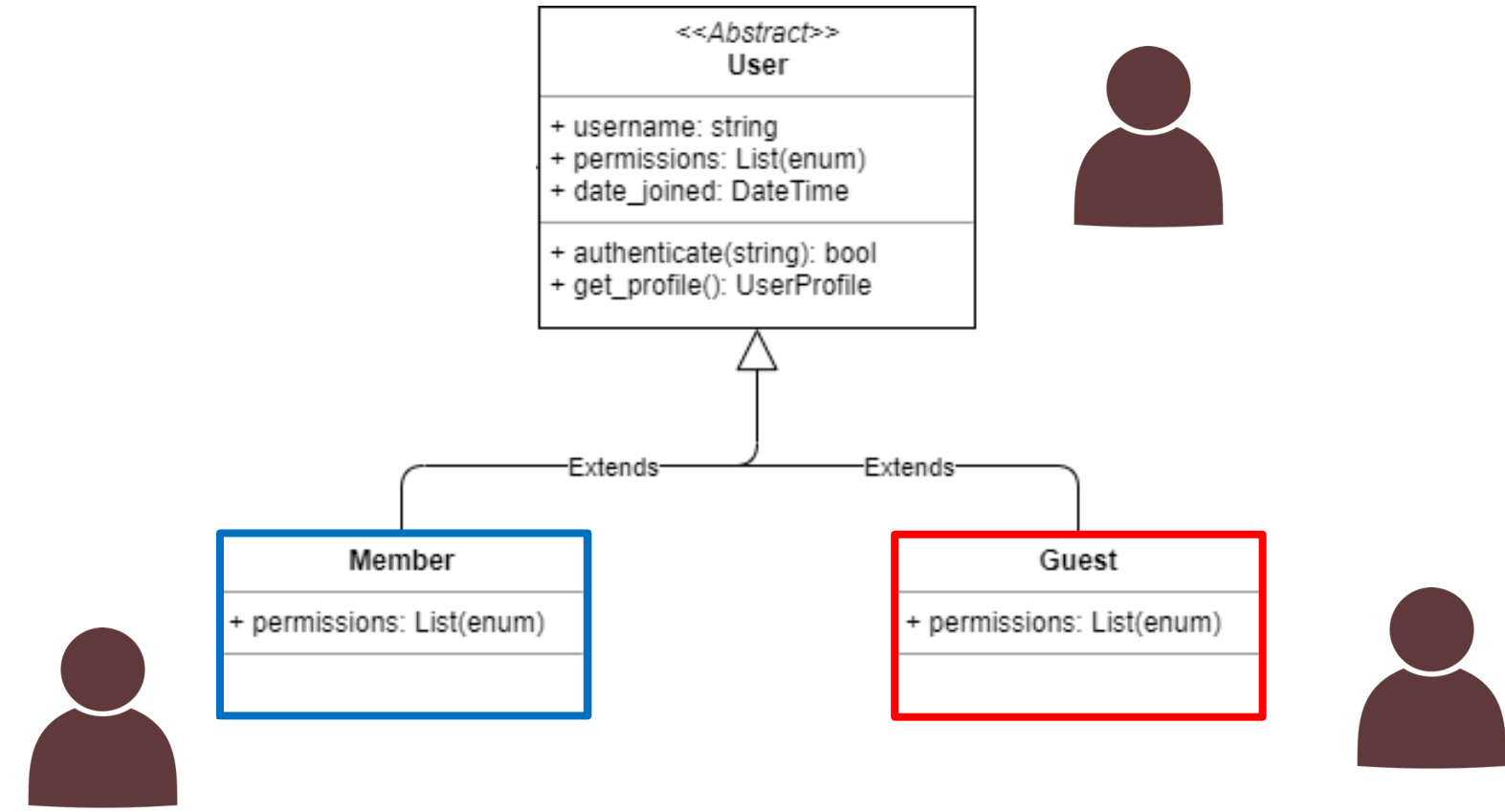
A **成员** 可以做 **访客** 能做的所有事情，但他们还可以 **撰写** 帖子。

**步骤 1：** 让我们创建一个 UML 类图来描述 **用户层级结构**。

## Step 1: Define a product hierarchy

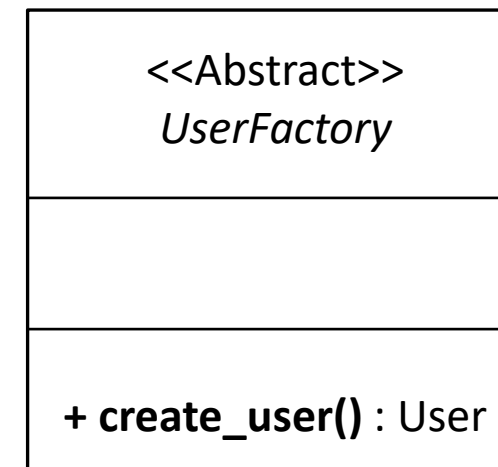


## 步骤1：定义产品层级



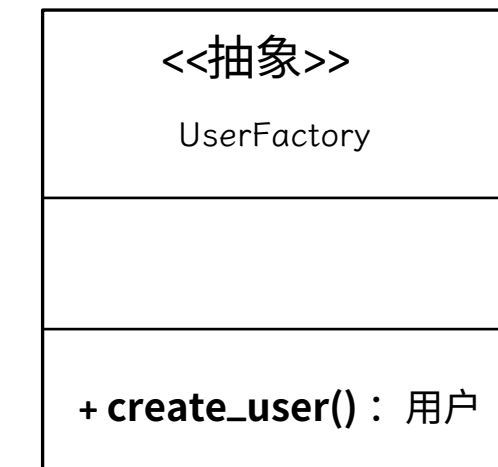
# Step 2: Define a factory class

- Let's create a UserFactory class.
- Add it to the UML diagram
- Specify the **creation method**.



# 步骤2：定义一个工厂类

- 让我们创建一个 UserFactory 类。
- 将其添加到 UML 图中
- 指定 **创建方法**。



# Step 2: Define a factory class

- In the Factory Pattern, a Factory class doesn't need to solely be responsible for creating objects. It can be a larger class with a **creation method**.
- **The base factory doesn't need to be abstract**. It can create a default product. In our case, let's make it abstract though.
- For example a UserFactory class might also keep track of how many user accounts are currently active.

<<Abstract>> UserFactory
+ create_user() : User

# 步骤2： 定义一个工厂类

- 在工厂模式中，工厂类不必仅仅负责创建对象。它可以是一个更大的类，其中包含一个**创建方法**。
- **基础工厂不一定是抽象的**。它可以创建默认产品。但在本例中，我们将其设为抽象类。
- 例如，UserFactory 类可能还需要跟踪当前有多少用户账户处于激活状态。

<<抽象>> UserFactory
+ create_user() : 用户

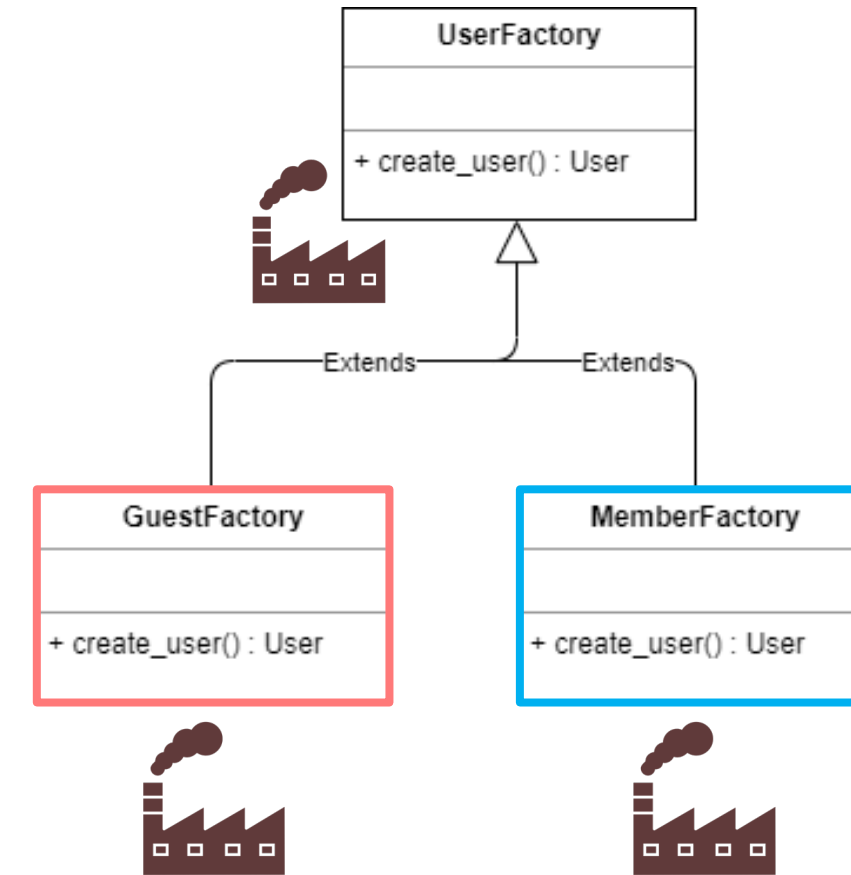
## Step 3: Create a Factory hierarchy and override the creation method.

The title says it all.

We want to create our concrete factories in this step.

Each concrete factory will create a new object of its corresponding user type and return it.

- **GuestFactory** creates **Guests**
- **MemberFactory** creates **Members**



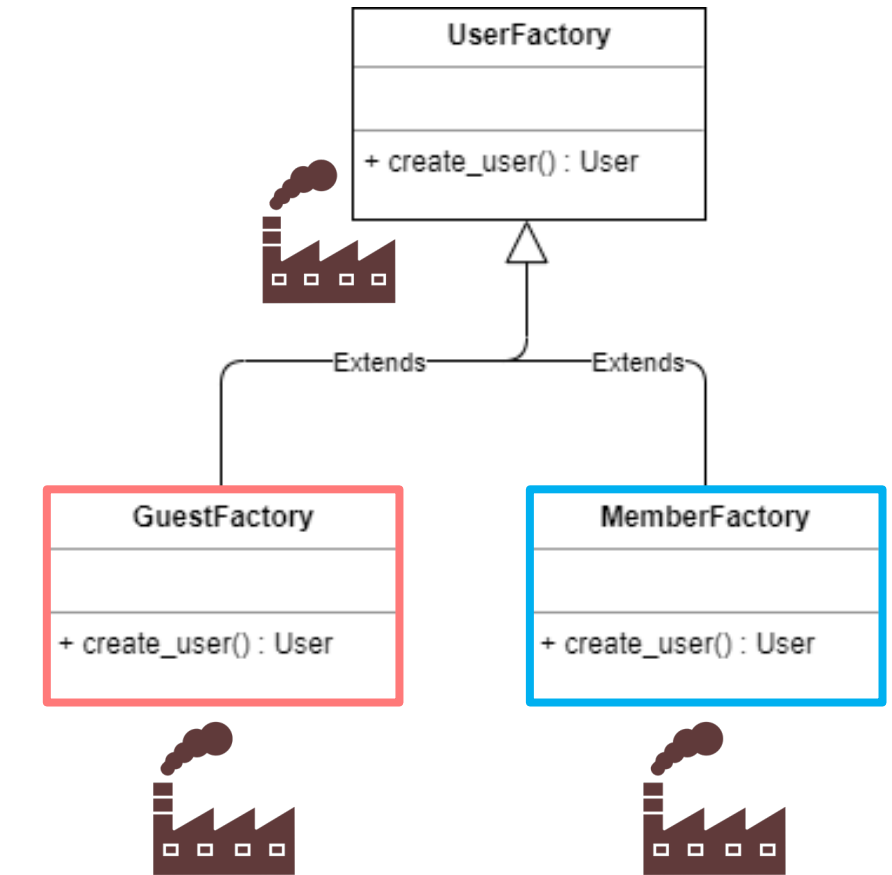
## 步骤3：创建一个工厂层次结构并重写创建方法。

标题说明了一切。

我们希望在此步骤中创建具体的工厂。

每个具体工厂将创建其自身的对象。  
对应用户类型并返回它。

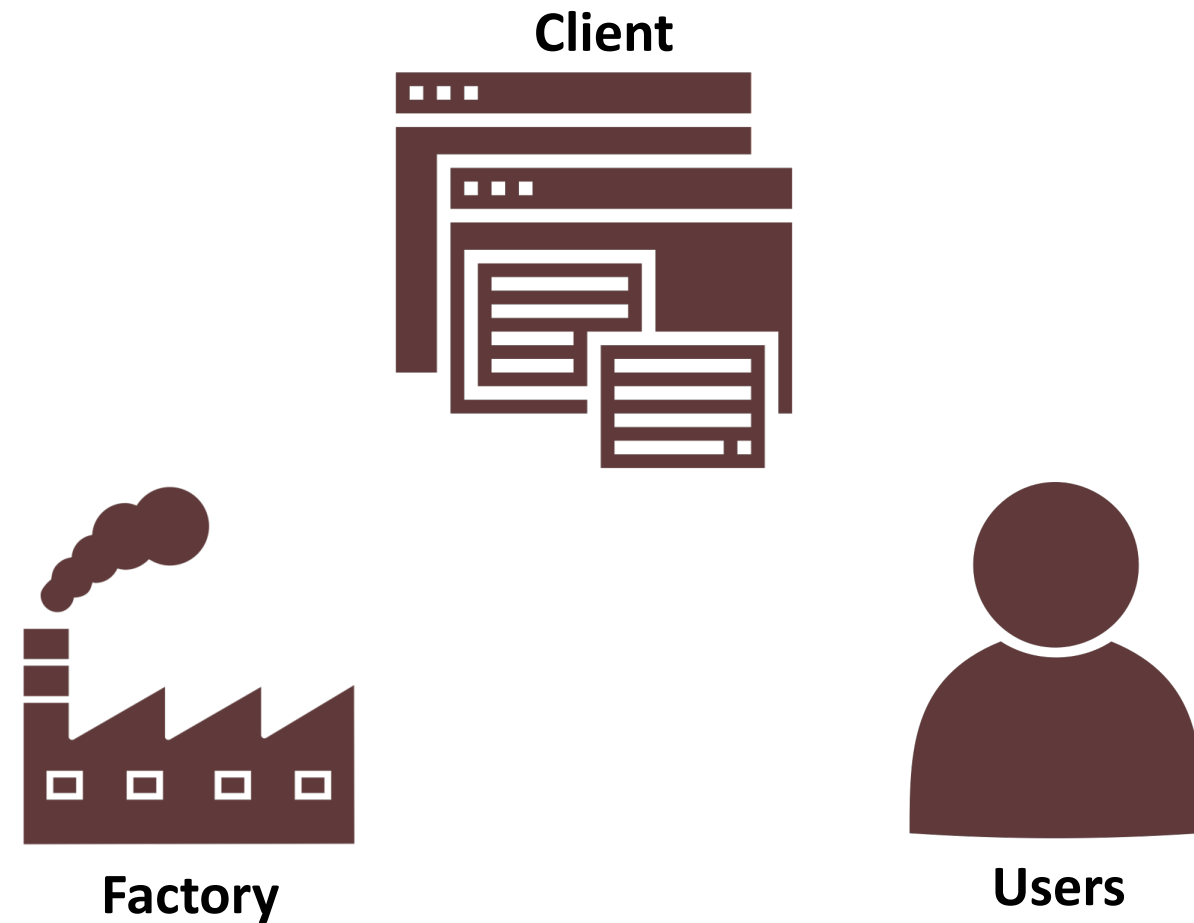
- **GuestFactory** 创建 **Guests**
- **MemberFactory** 创建 **Members**



## Step 4: Integrate it into our Forum Website

Bring it all together.

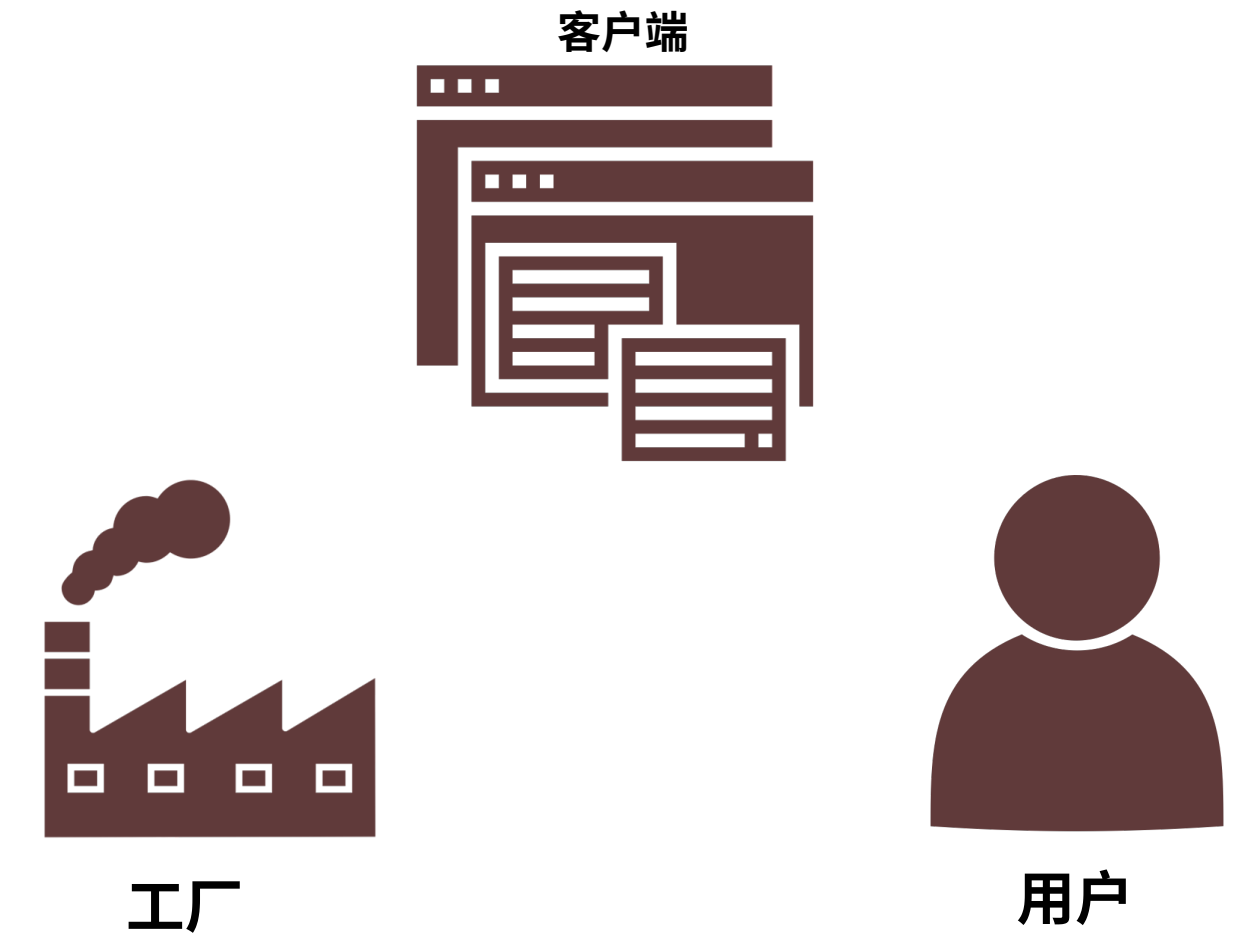
Add a client class, and mark the associations between Client, Factories and Users.



## 步骤 4： 将其集成到我们的论坛网站中

将所有内容整合在一起。

添加一个客户端类，并标记 Client、Factories 和 Users 之间的关联关系。



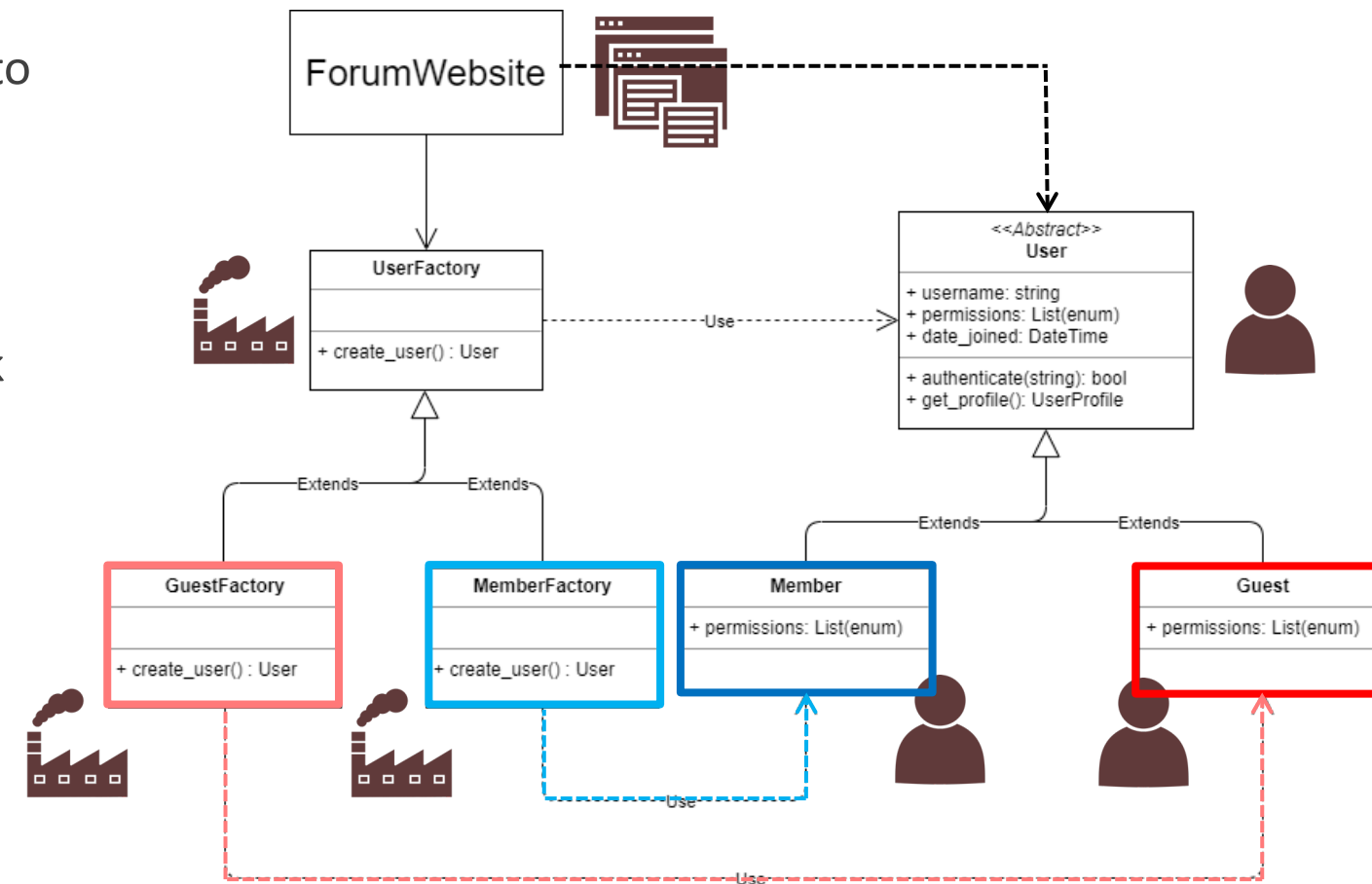
## Step 4: Integrate it into our Forum Website

We can have a ForumController that prompts the user for the type of user to create

And that's it! We have a working Factory Pattern.

It's pretty straight forward if you break it down right?

[forum\\_user\\_factory.cpp](#)



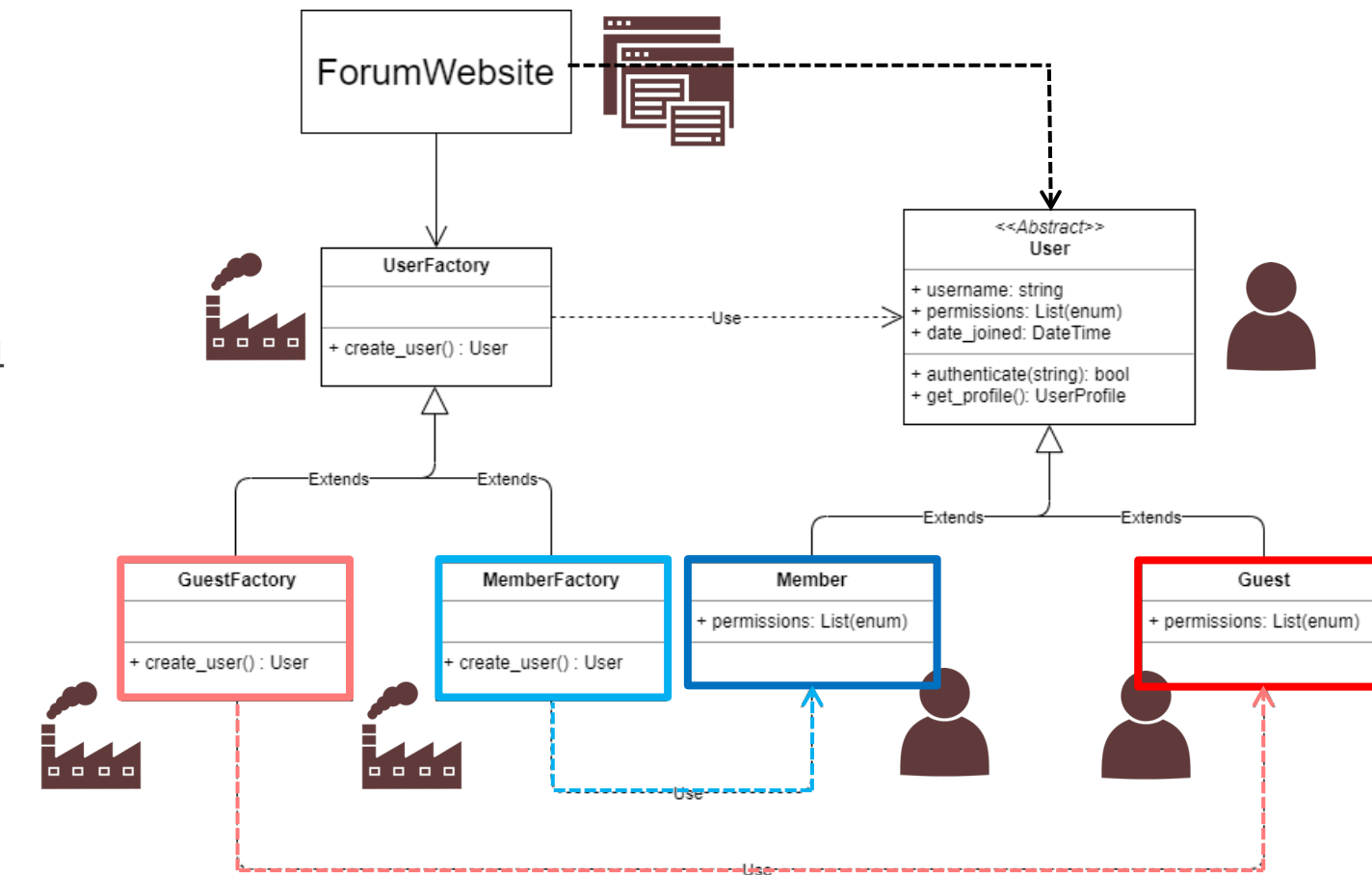
## 步骤 4： 将其集成到我们的论坛网站中

我们可以创建一个 ForumController，提示用户输入要创建的用户类型。

就这样！我们已经实现了一个可用的工厂模式。

如果你正确地将其分解，其实相当简单明了，对吧？

[论坛\\_用户\\_factory.cpp](#)



# Factory Pattern:

## Why and When do we use it

- Adheres to the Single Responsibility, Open Closed, Liskov Substitution and Dependency Inversion Principle.
- When we want to provide a **separate interface for object creation** where each subclass can alter the type of object created.
- Use the Factory Pattern when the **exact type and dependencies of the objects you need are unknown** or susceptible to change. If we want to add a new product, just create a new Factory and override the creation method in it.
- Use this pattern when you want to **separate and encapsulate the creation process**. This is handy if we want to cache or pick objects from a pool, etc.



# 工厂模式：我们为何以及何时使用它

- 遵循单一职责、开闭原则、里氏替换和依赖倒置原则。
- 当我们希望为对象创建提供一个**独立的接口**，并且每个子类可以改变所创建对象的类型时。
- 当所需对象的具体类型和依赖关系未知或可能发生变化时，使用工厂模式。如果我们要添加一个新产品，只需创建一个新的工厂并在其中重写创建方法即可。
- 当您想要 **分离并封装创建过程** 时，请使用此模式。如果我们希望从对象池中缓存或选择对象等，这将非常方便。



# Factory Pattern – Disadvantages

- **There are a lot of classes** in play. This can make the code complex and hard to debug.
- Sometimes the **classes can get artificial**. You may decide to create a whole new subclass for a very minor change in the object creation process.



# 工厂模式——缺点

- **涉及的类很多**，这可能会使代码变得复杂且难以调试。
- 有时 **类可能会变得不自然**。你可能仅仅因为对象创建过程中一个非常微小的改动，就决定创建一个全新的子类。

