Structural Patterns: Proxy, Facade, Bridge

COMP3522 OBJECT ORIENTED PROGRAMMING 2

WEEK 10

https://refactoring.guru/design-patterns

Categorizing Design Patterns

- **Behavioural**

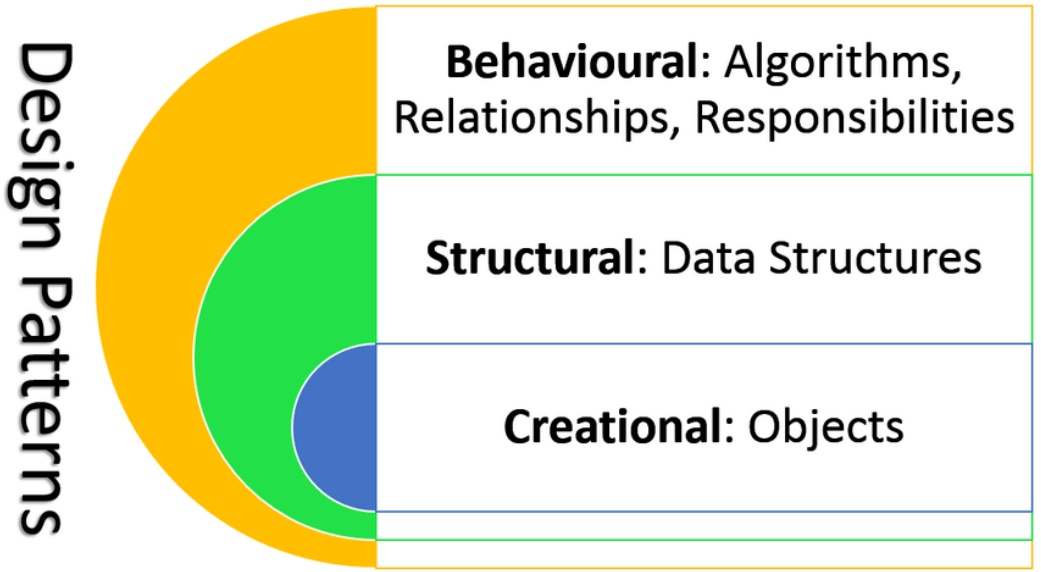
Focused on communication and interaction between objects. How do we get objects talking to each other while minimizing coupling?

- **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



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.

Wrapper

What is a wrapper?

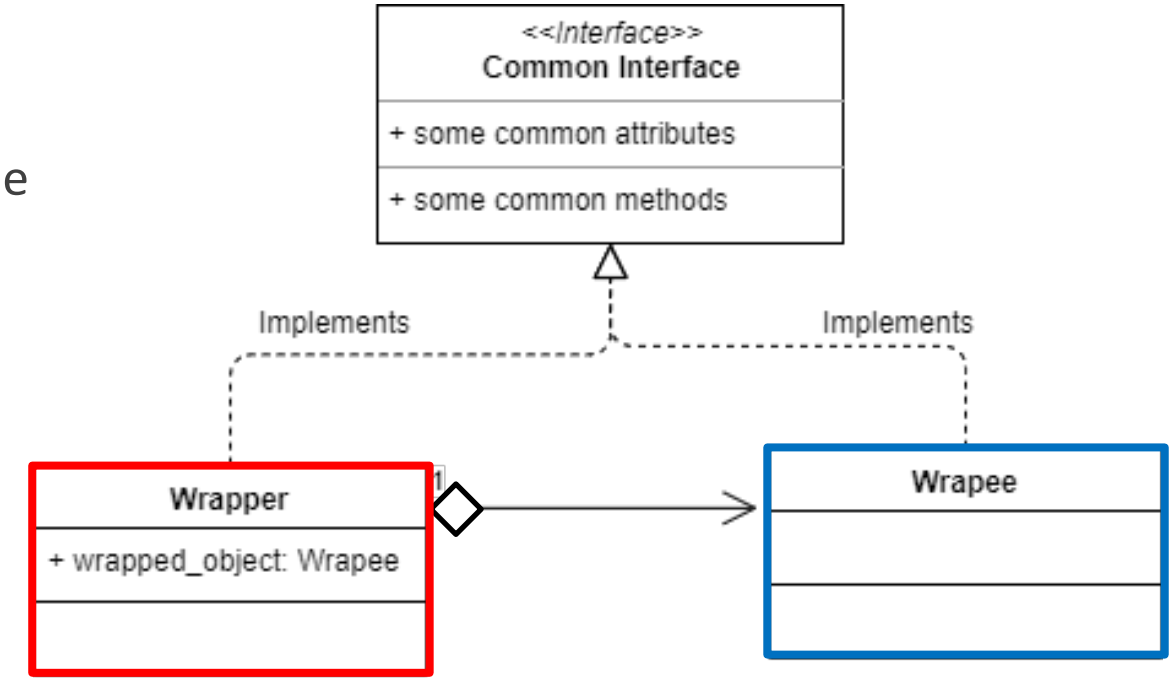
A class that **'wraps' around another class**.

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



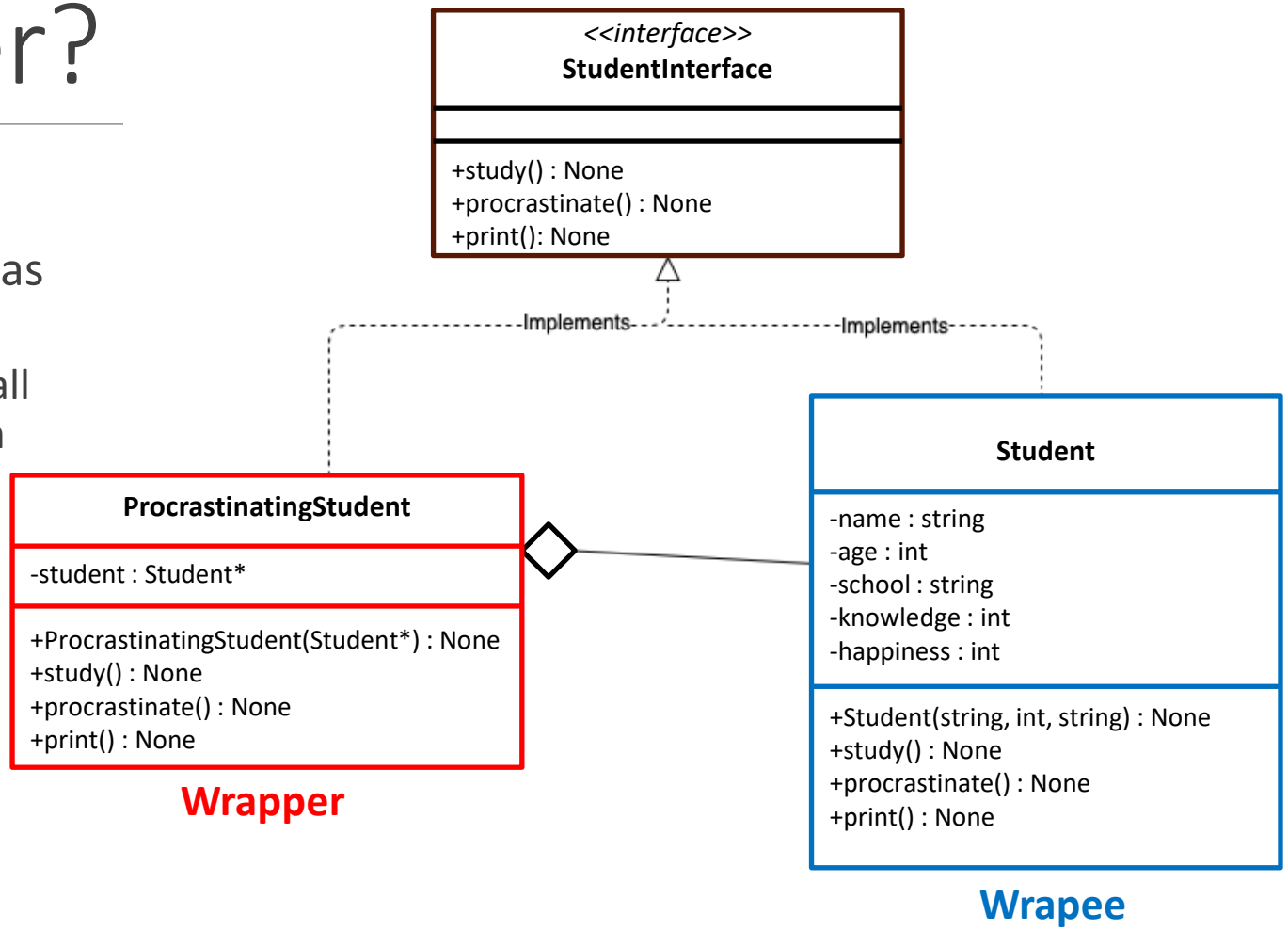
What is a wrapper?

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**



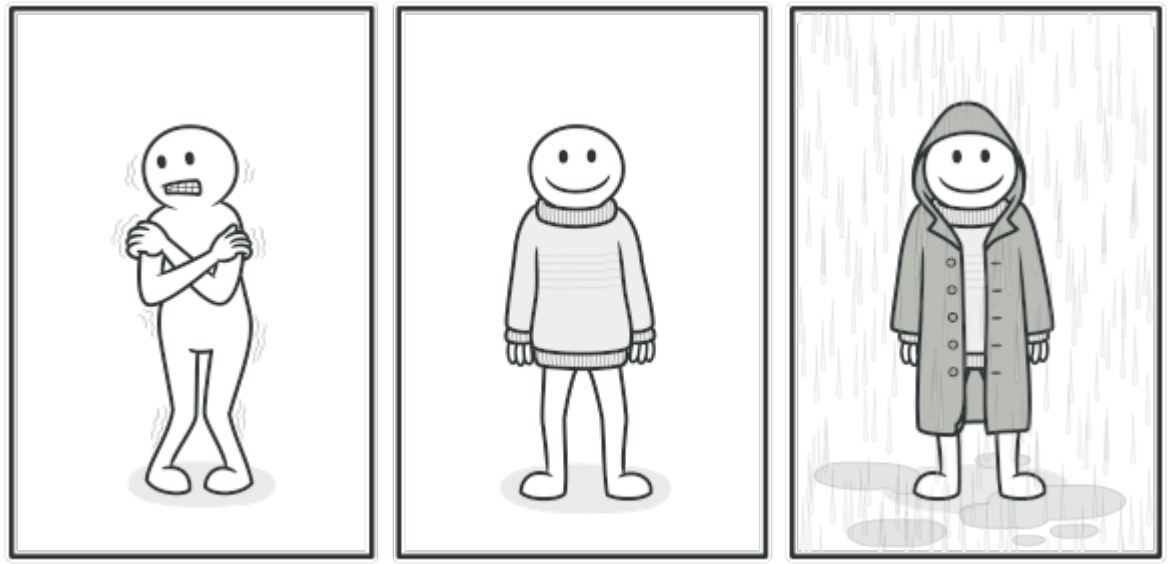
Wrapper.cpp

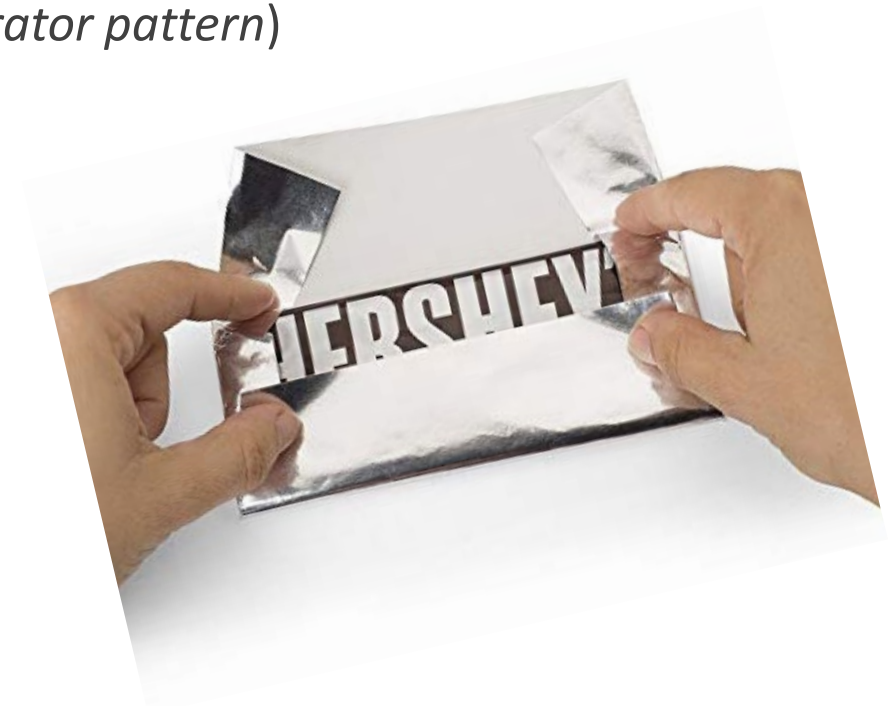
Wearing many layers (Multiple Wrappers)

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*)





Proxy

Proxy

A Proxy is a wrapper.

It controls access to the wrapee.

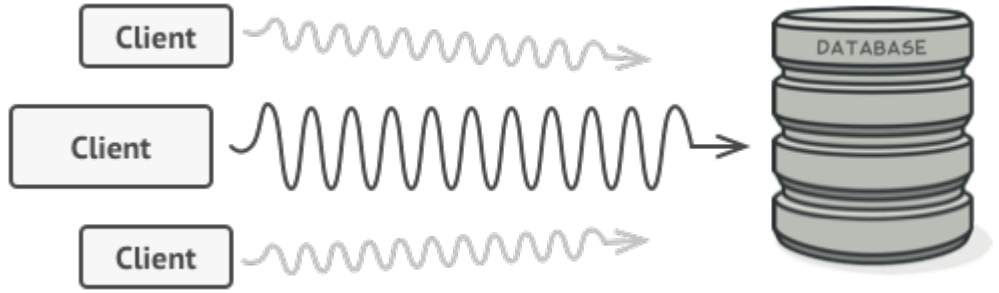
For example:

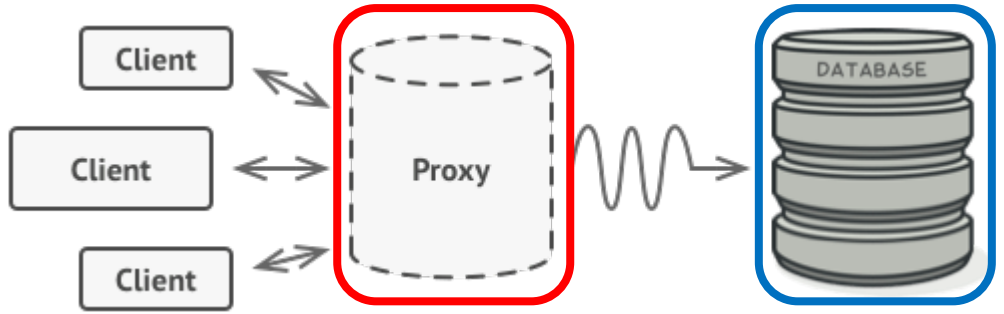
Database queries can be slow.

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.

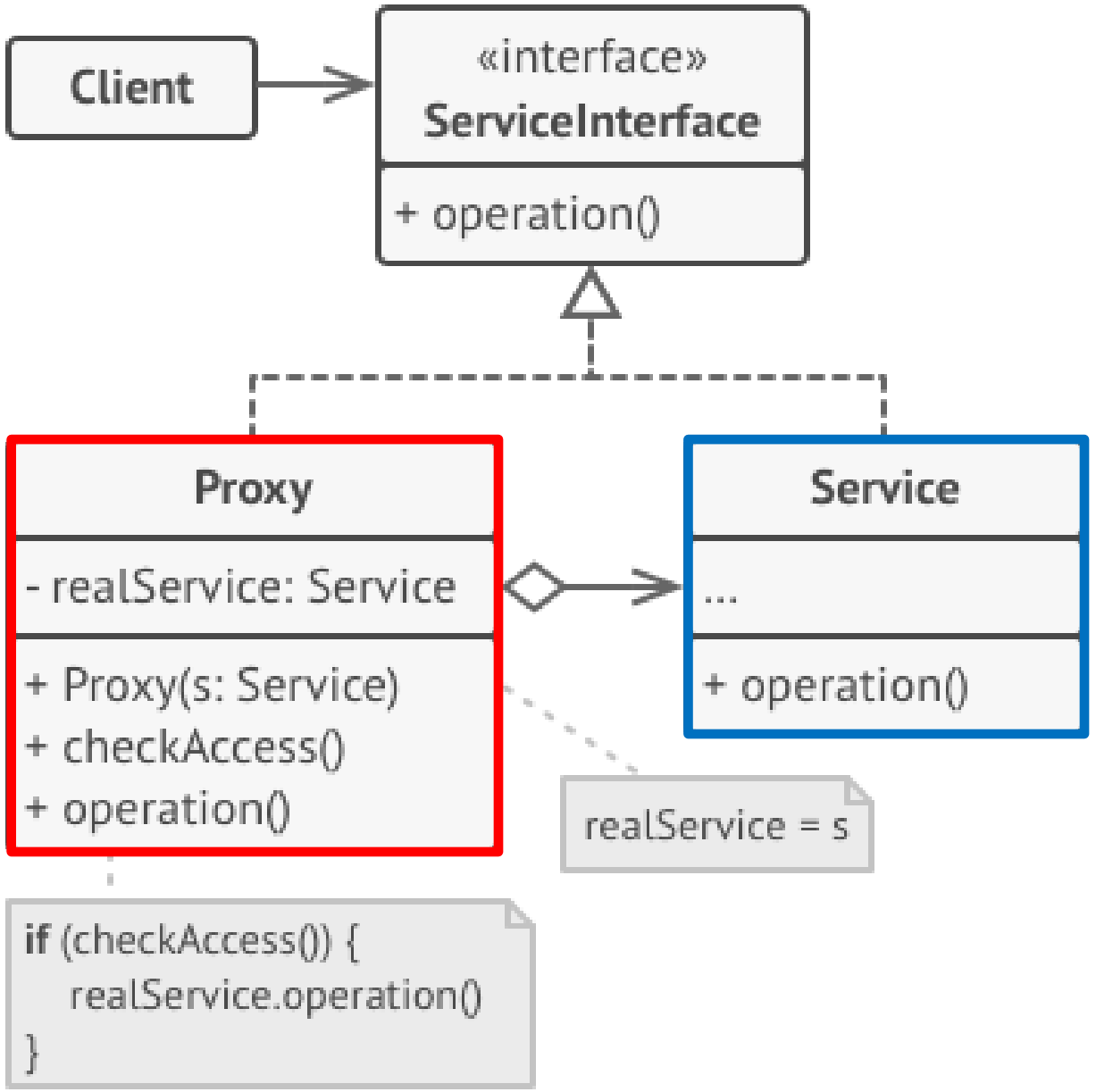
A **proxy** in this case, is an object that has the **same interface** as the **database** and has access to it.

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

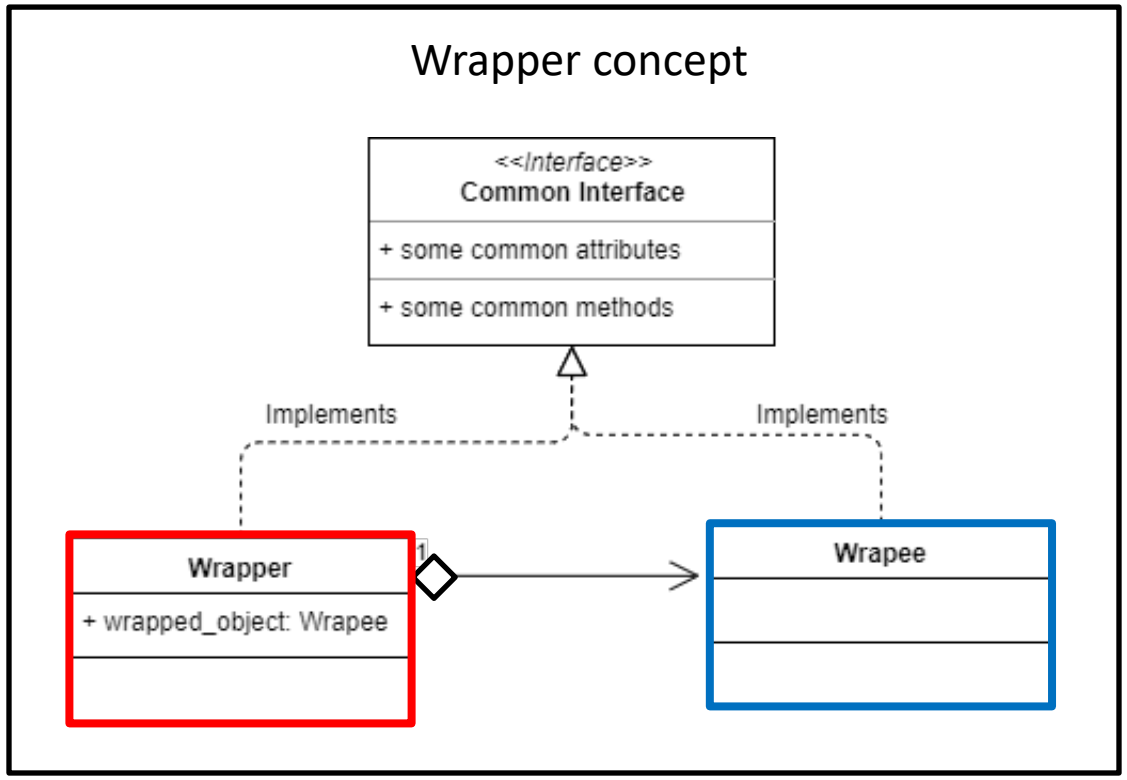


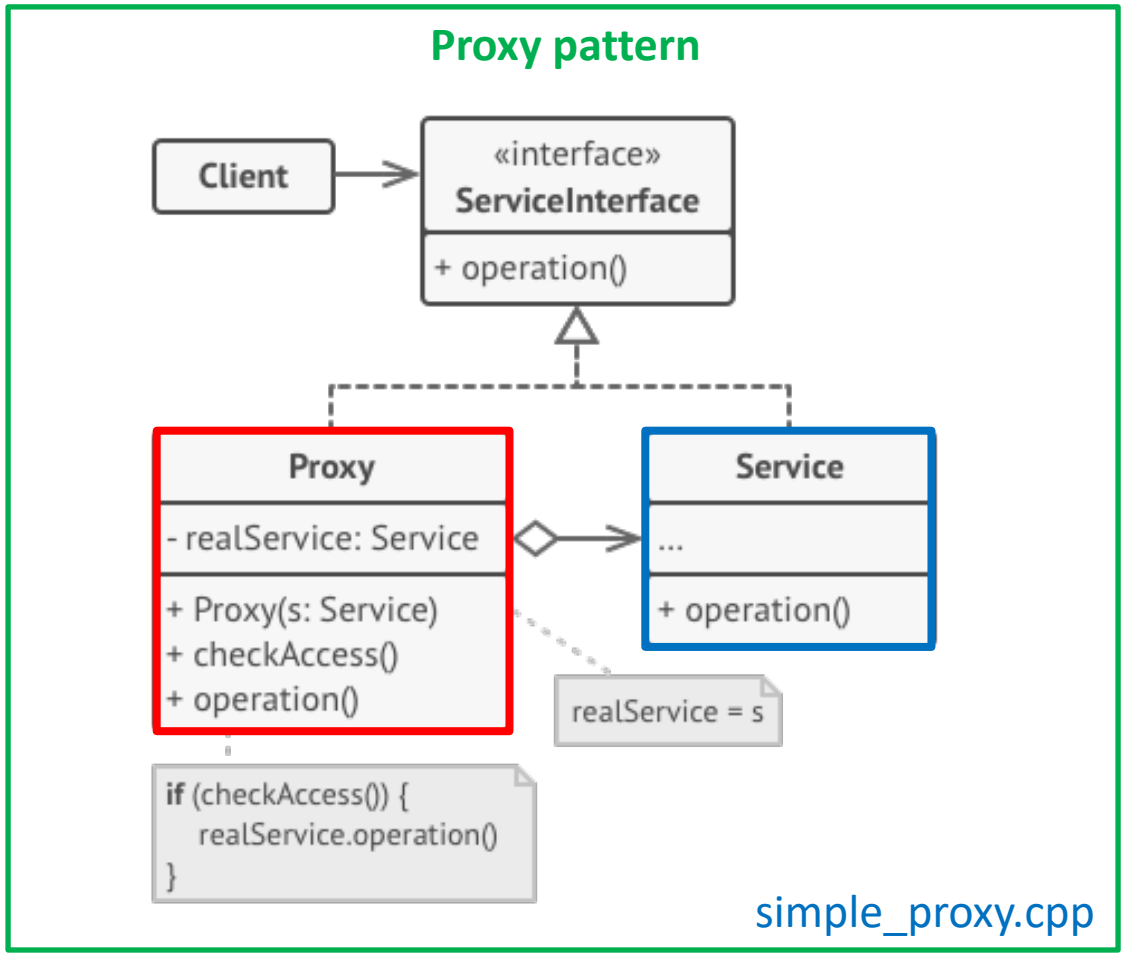


The Proxy Pattern



This is the exact same structure as a wrapper.





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.

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.

Facade

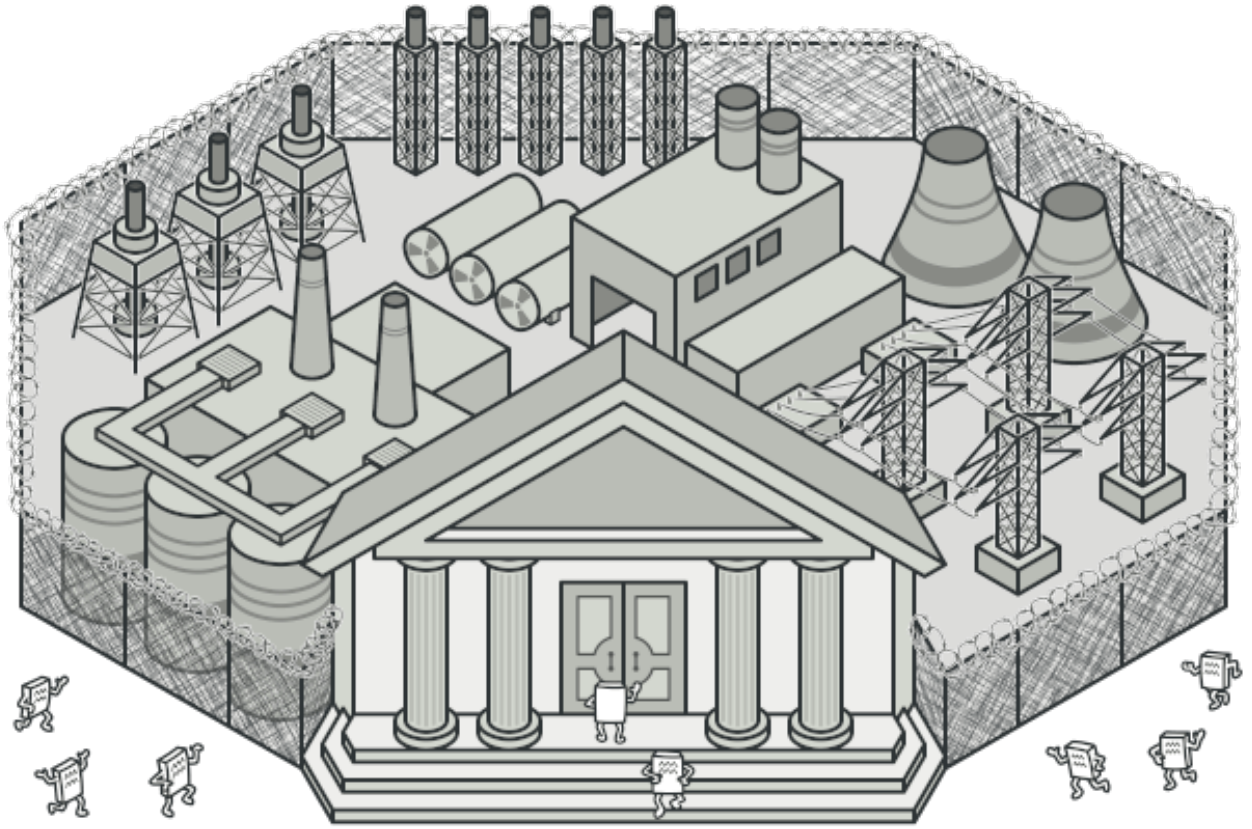
WHEN YOU WANT TO HIDE THE MESS RIGHT BEFORE YOUR FRIENDS COME OVER FOR BOARD GAMES…

Facade

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

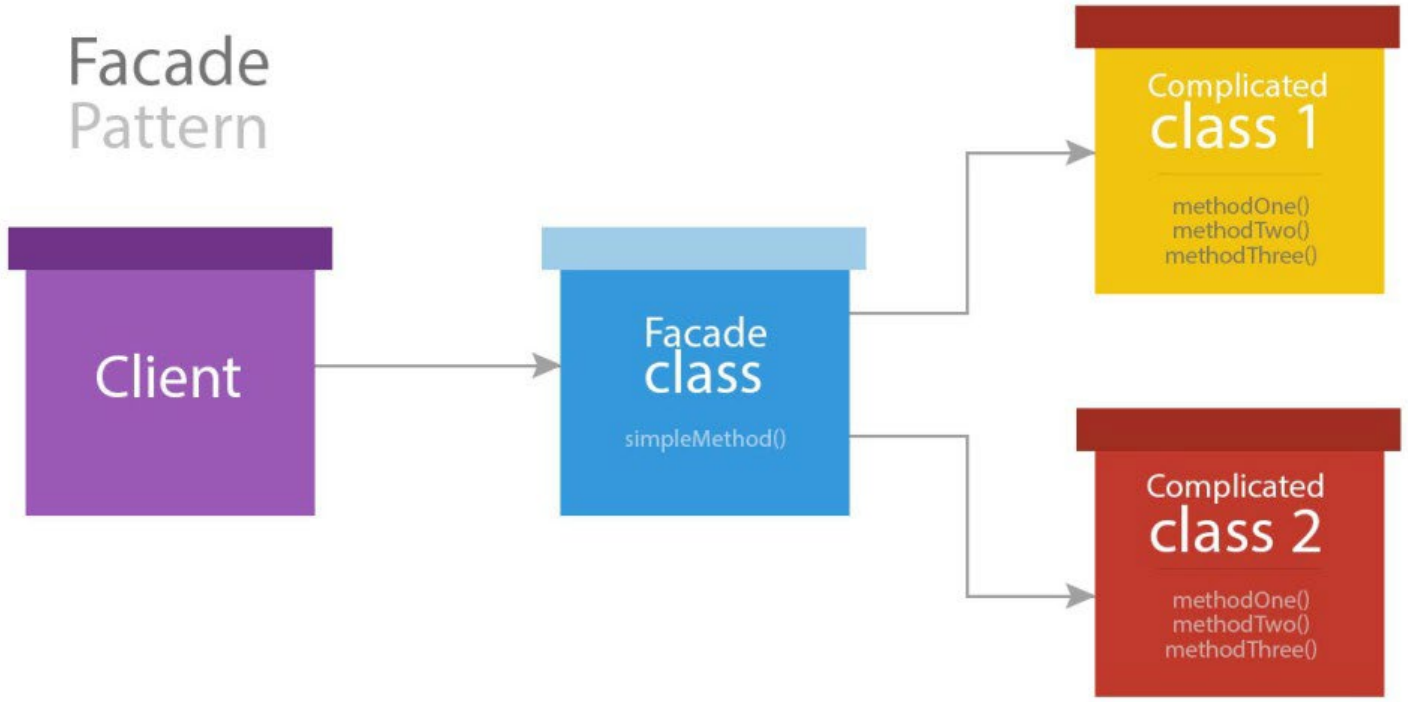
Used when we need to provide a **simple interface to a complex system**

Abstracts away the details of the complex system or API

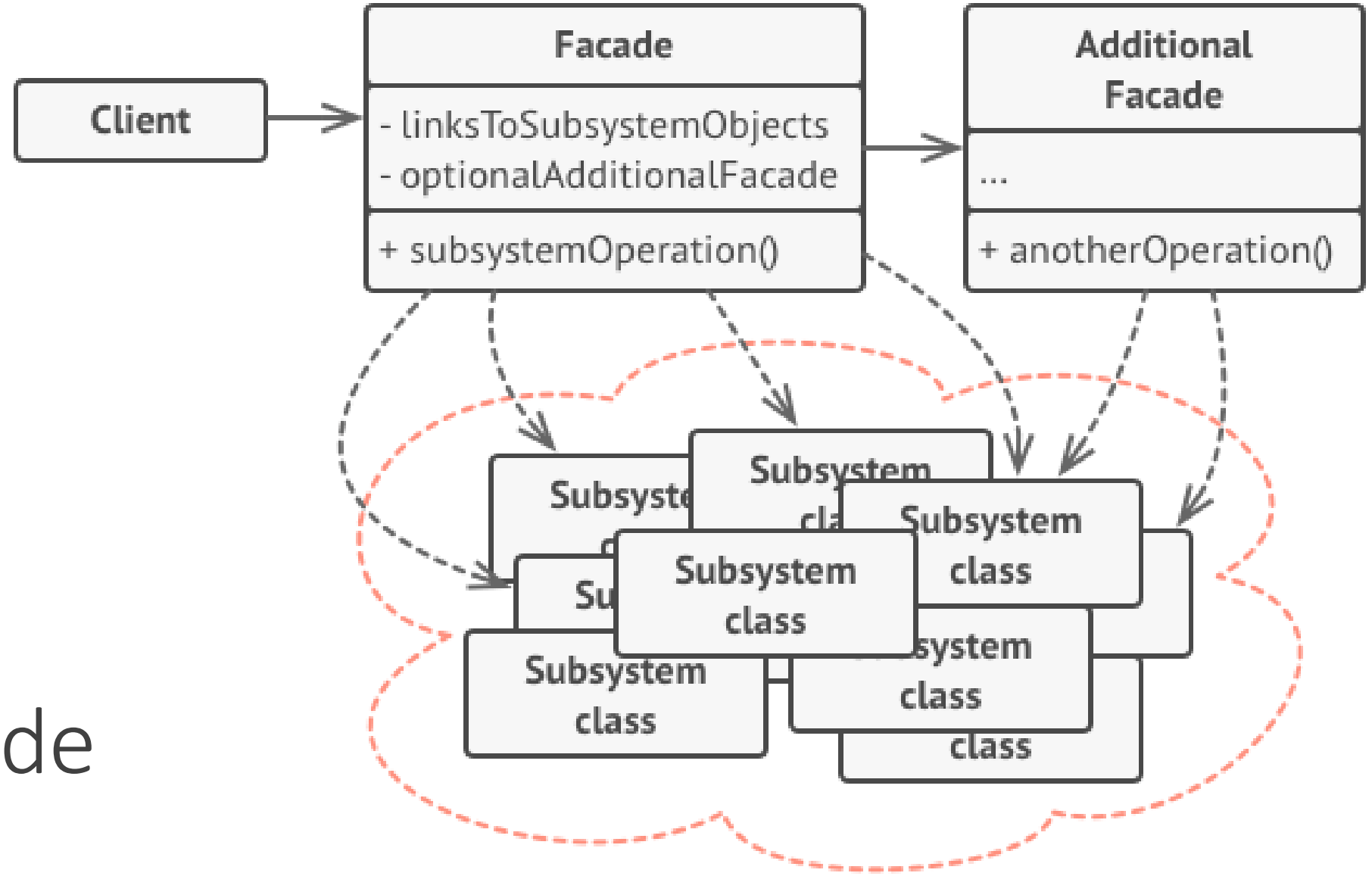


Facade

- 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**.



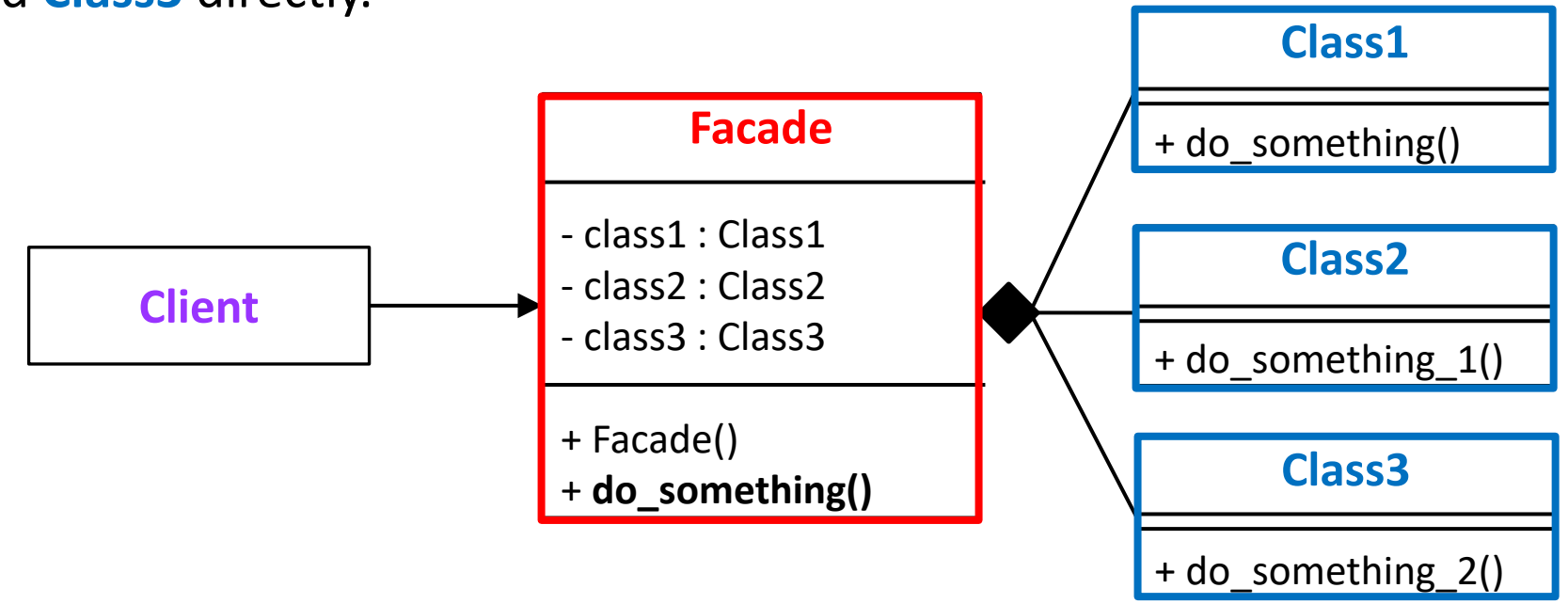
The Facade Pattern



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 Façade's **do_something** function
- Client now only needs to call Façade's **do_something** function instead of calling Class1, Class2, and Class3 directly.

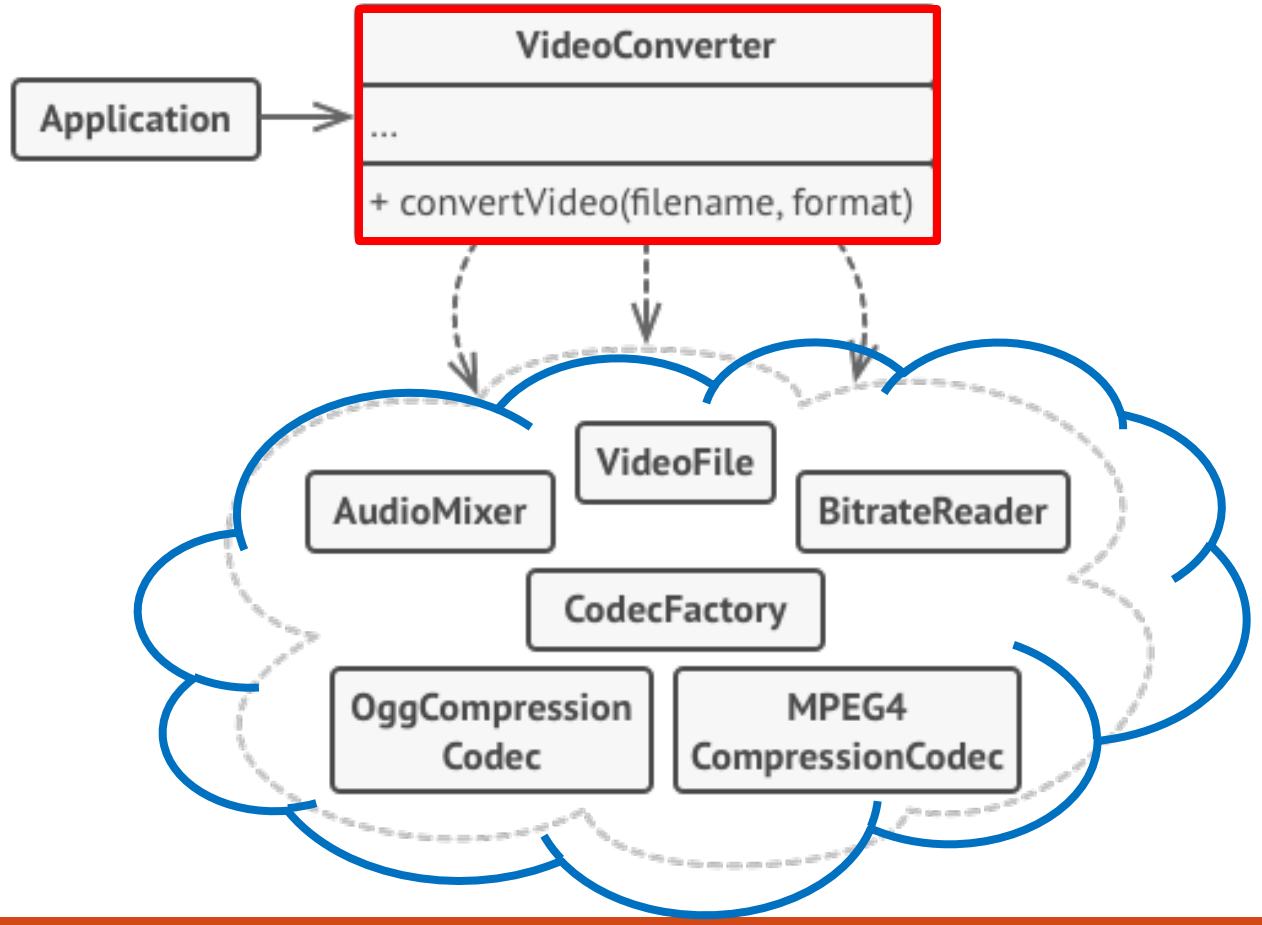


simple_facade.cpp

Facade Example

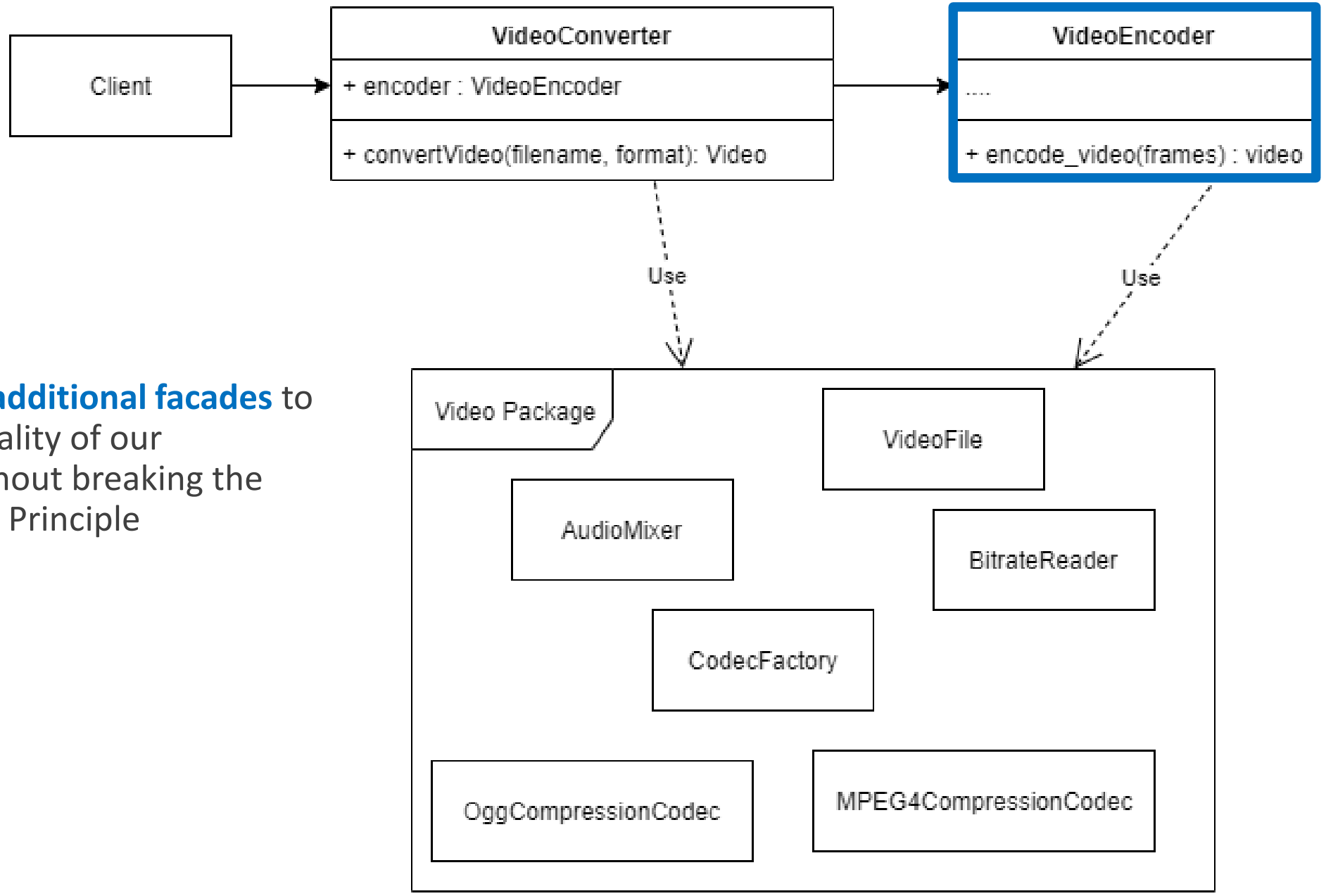
Say you imported a video conversion package that offered different ways to convert a video file with a number of customizations.

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

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



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**.

Facade - Disadvantages

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

Bridge

WHEN COMPOSITION SAVES THE DAY

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.

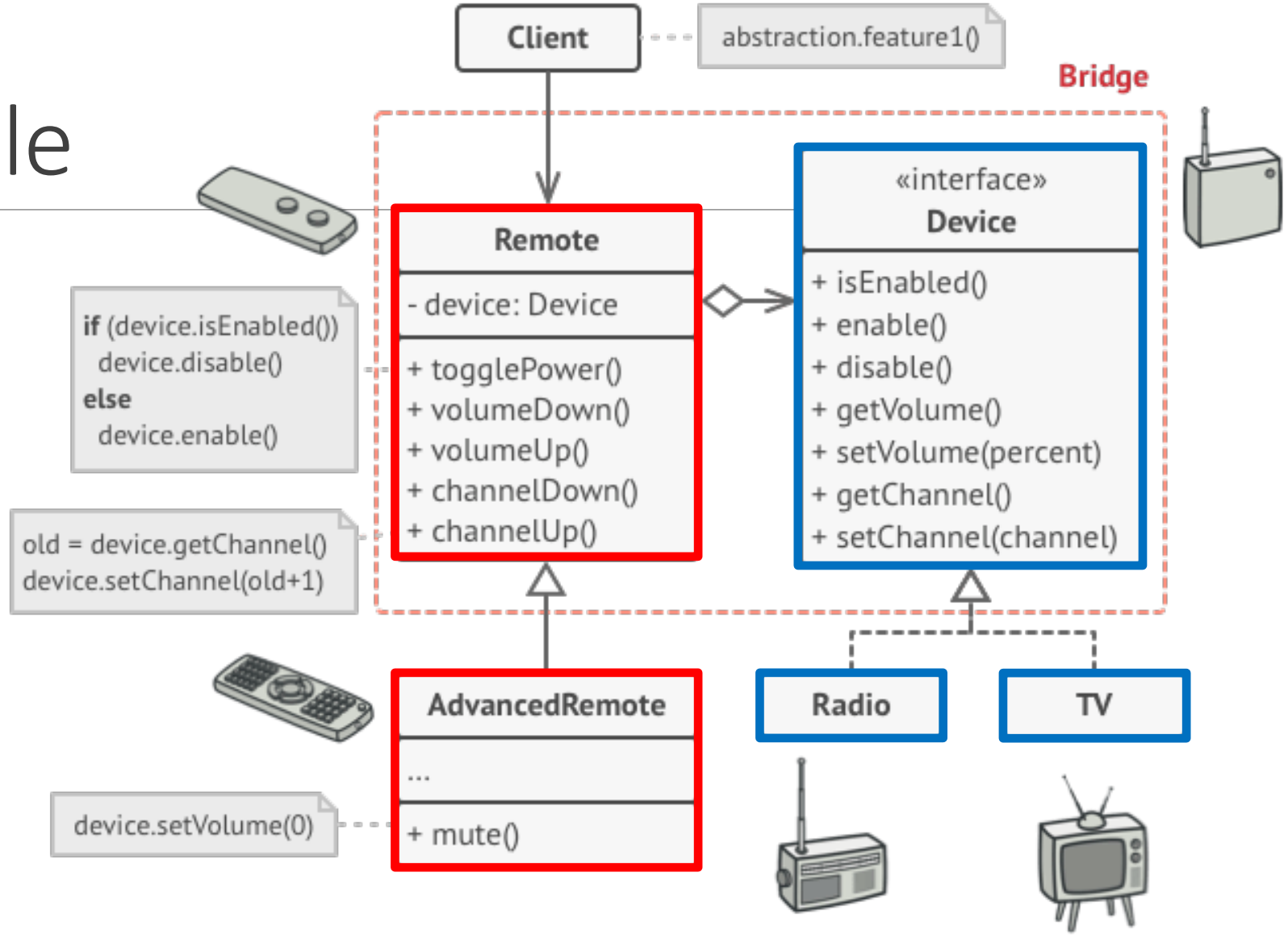
Bridge example

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.



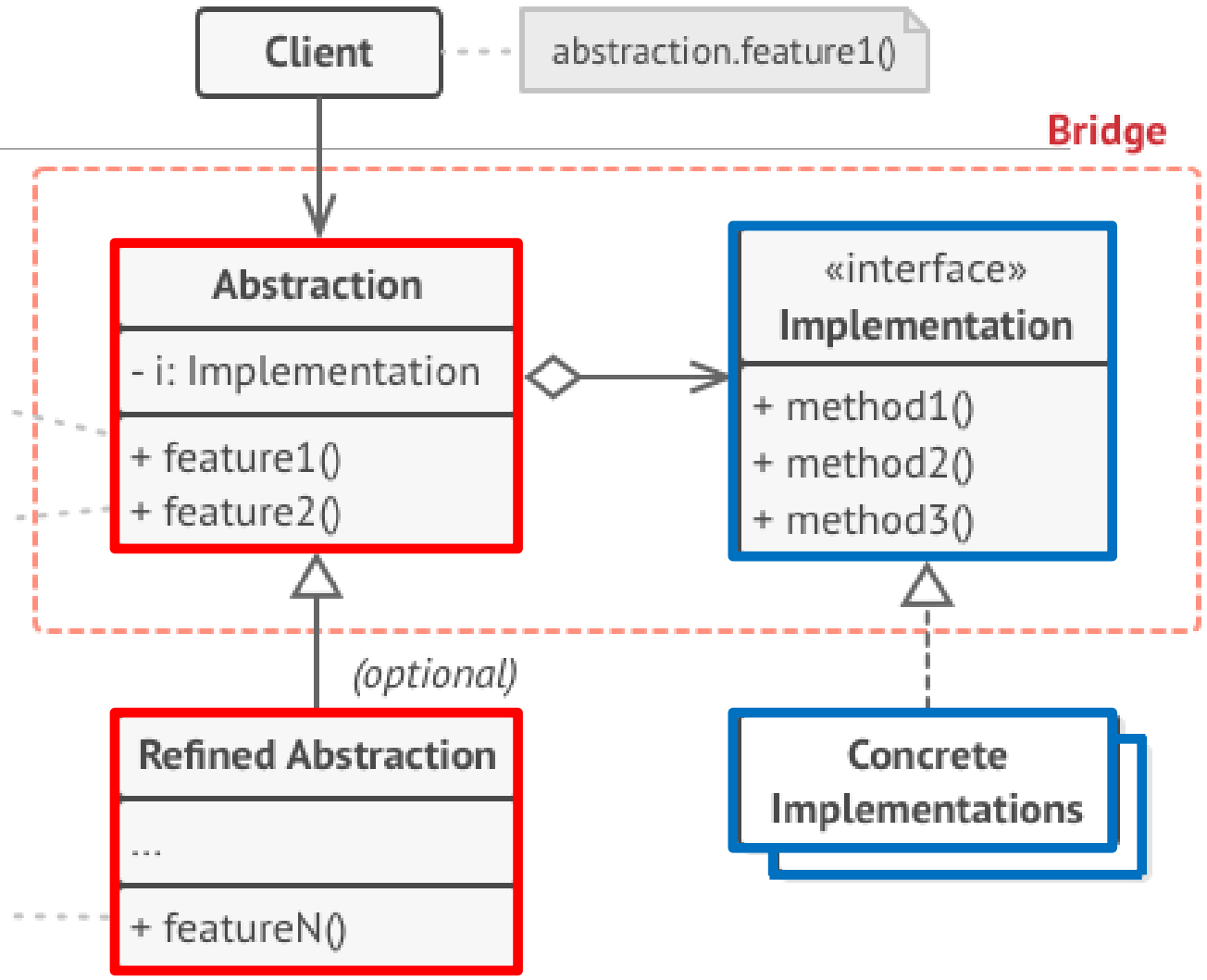
device_remote_bridge_example.cpp

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.



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.

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.

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".

Factory pattern

COMP3522 OBJECT ORIENTED PROGRAMMING 2

WEEK 10

Categorizing Design Patterns

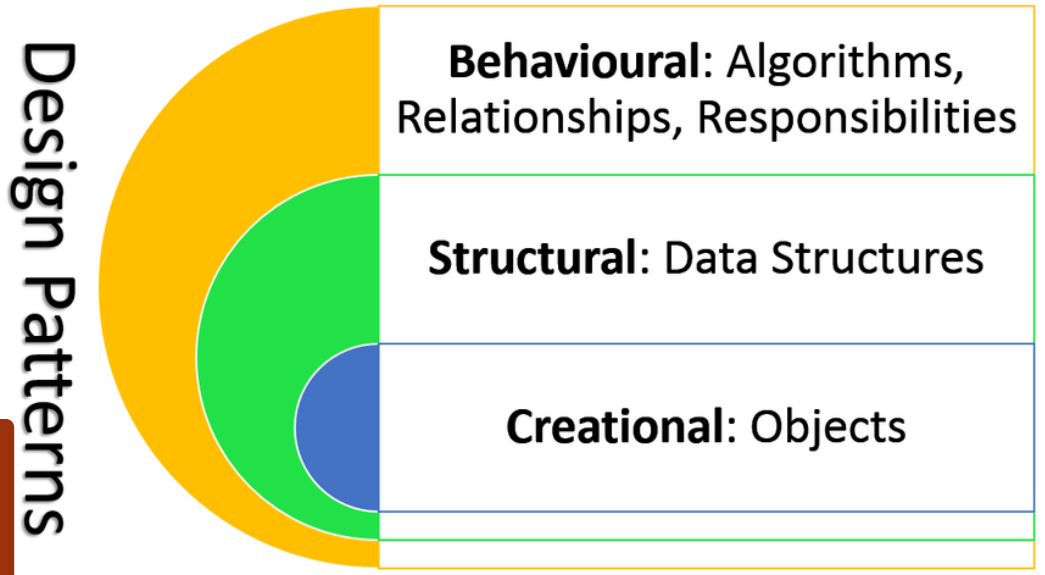
- **Behavioural**

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

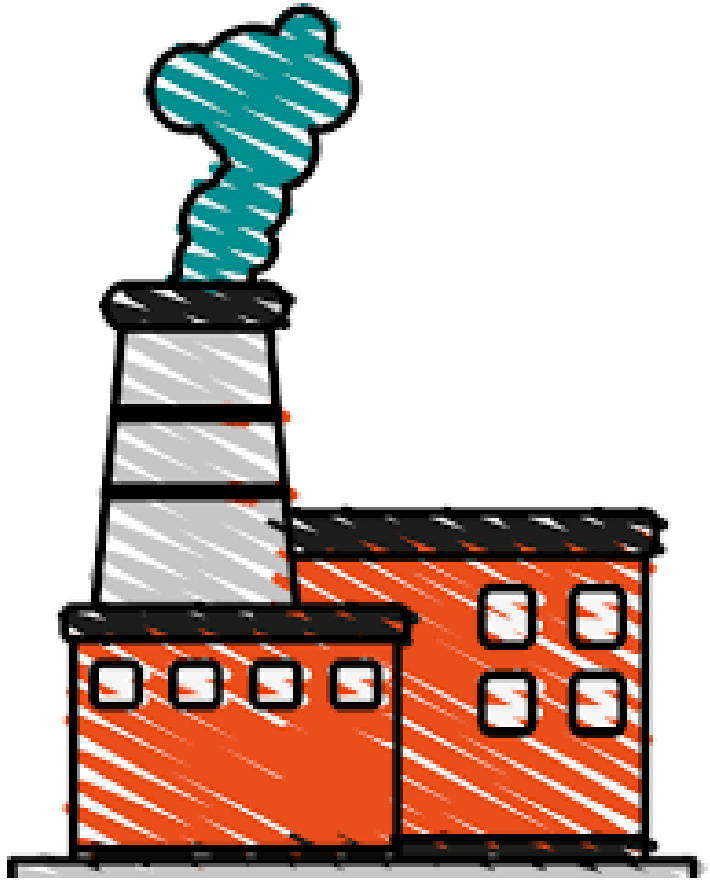
- **Structural** How do classes and objects combine to form structures in our programs? Focus on architecting to allow for maximum flexibility and maintainability.

- **Creational (We are looking at these!)**

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



First let's clarify the word Factory…



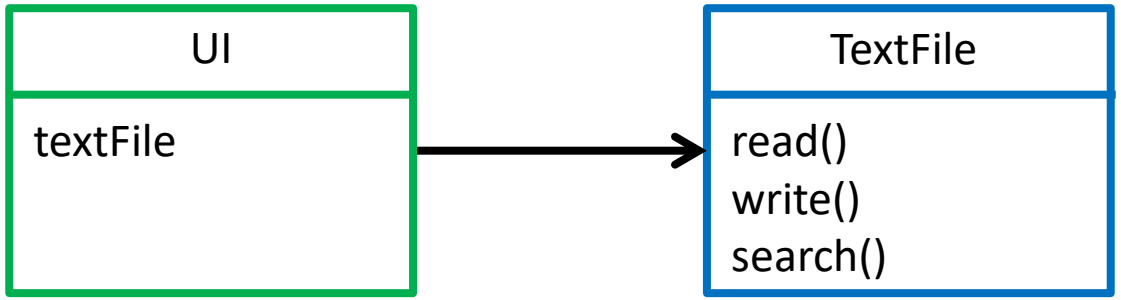
- Something that is responsible for making things
- In this case objects.
- The word **factory** is used a lot, don't mix it up with the **Factory Pattern** or the **Abstract Factory Pattern**. They are also responsible for making objects but are different.
- A Factory could refer to:
  - The pattern(s)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

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;
}
```

Remember the dependency inversion principle?



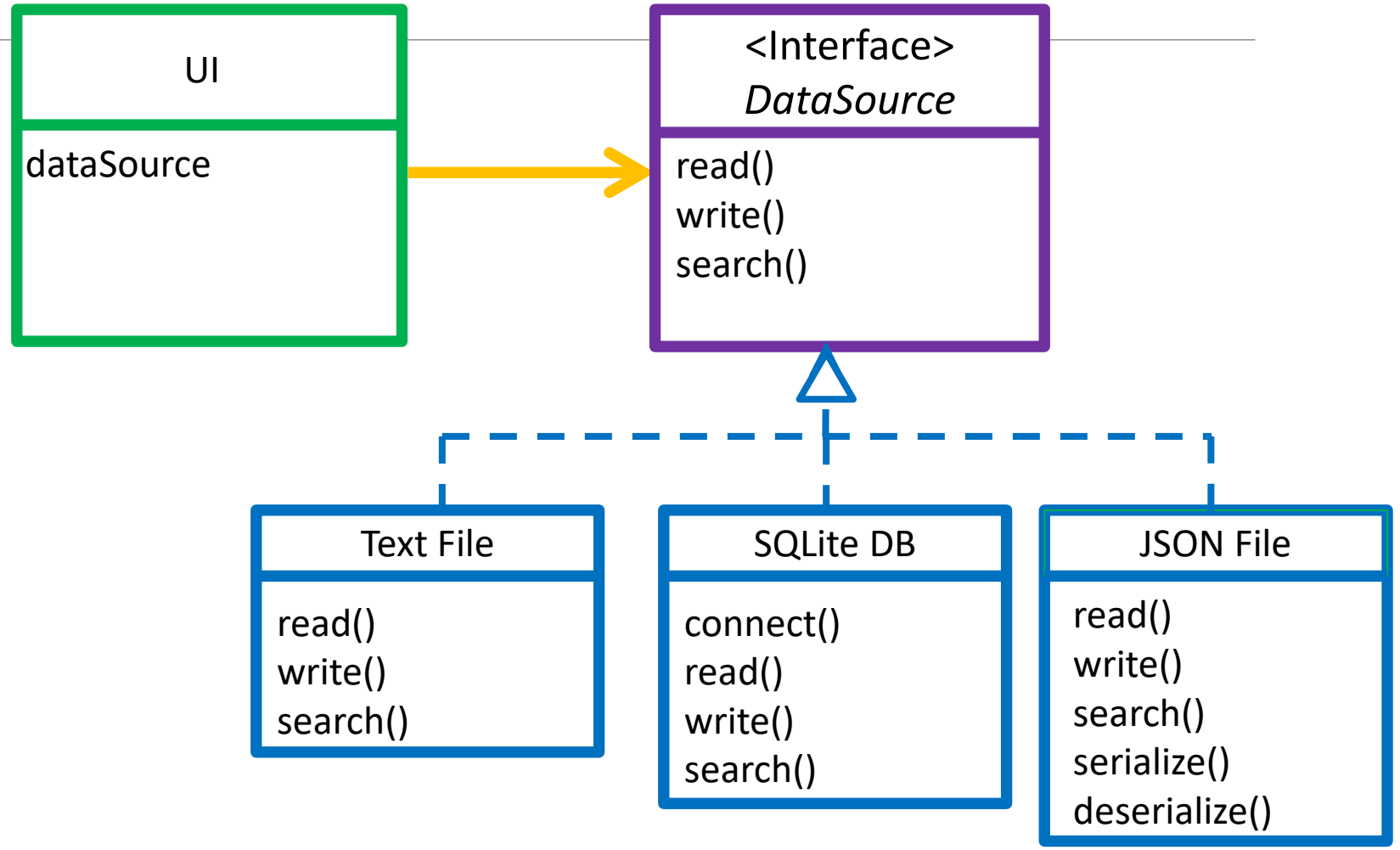
Say we were developing an app with a UI that was populated with some data from a text file.

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!

Remember the dependency inversion principle?

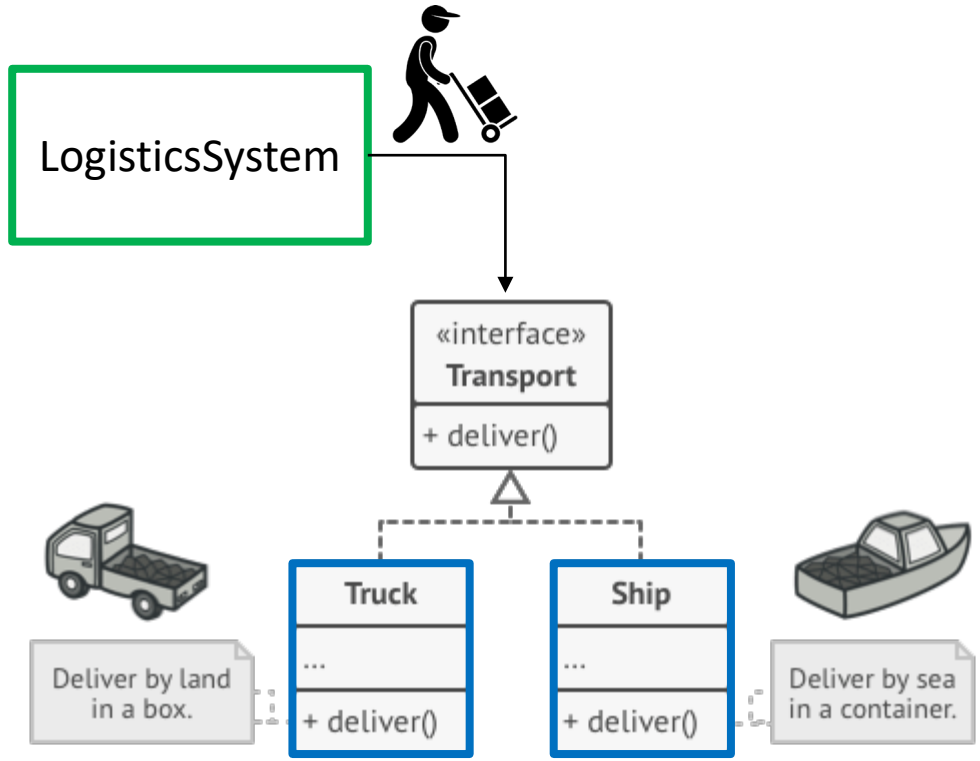
We can decouple our system to instead depend on a **data source** which is an **abstraction** that **hides** different **data sources**.

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



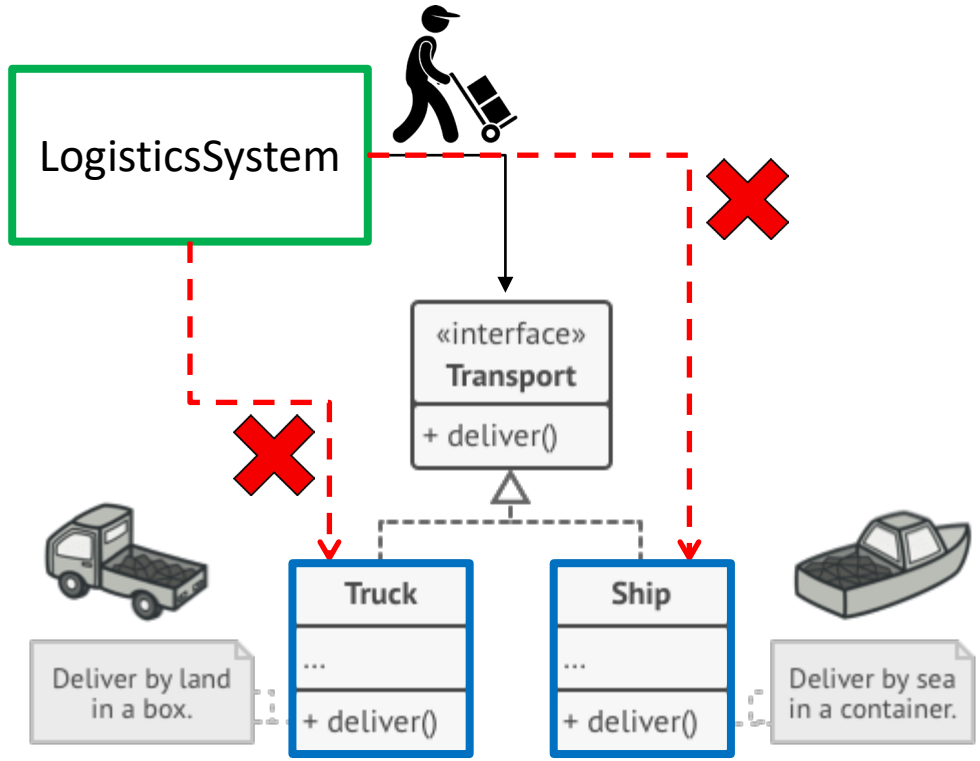
Factory Pattern

- 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**.



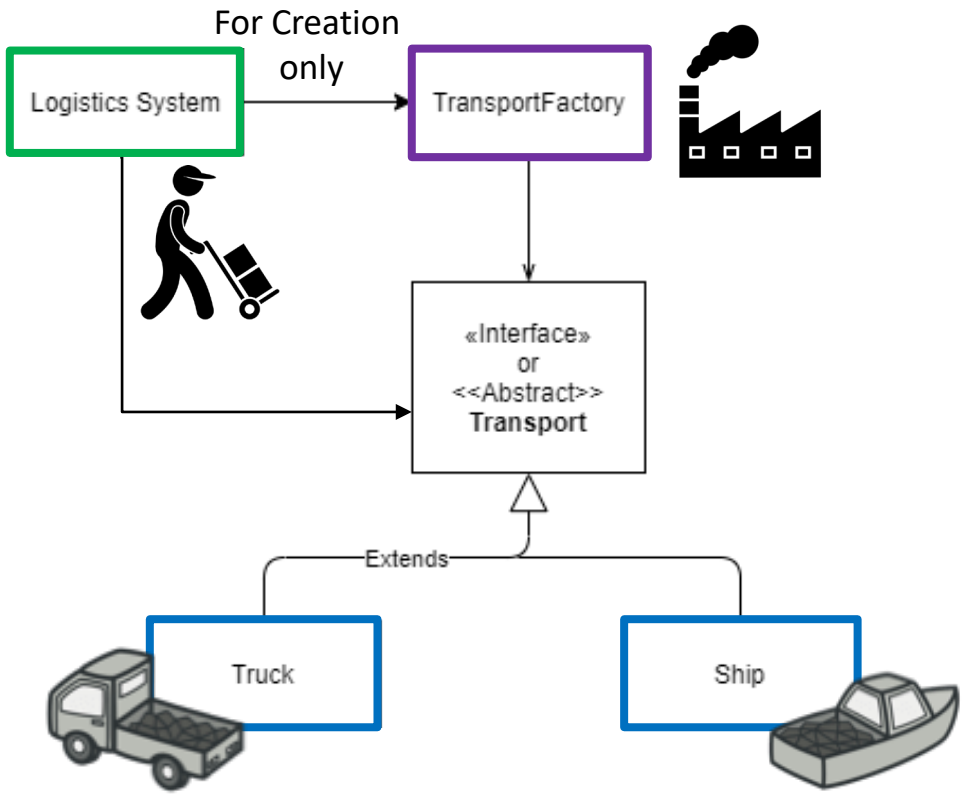
Factory Pattern

- 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**



Factory Pattern

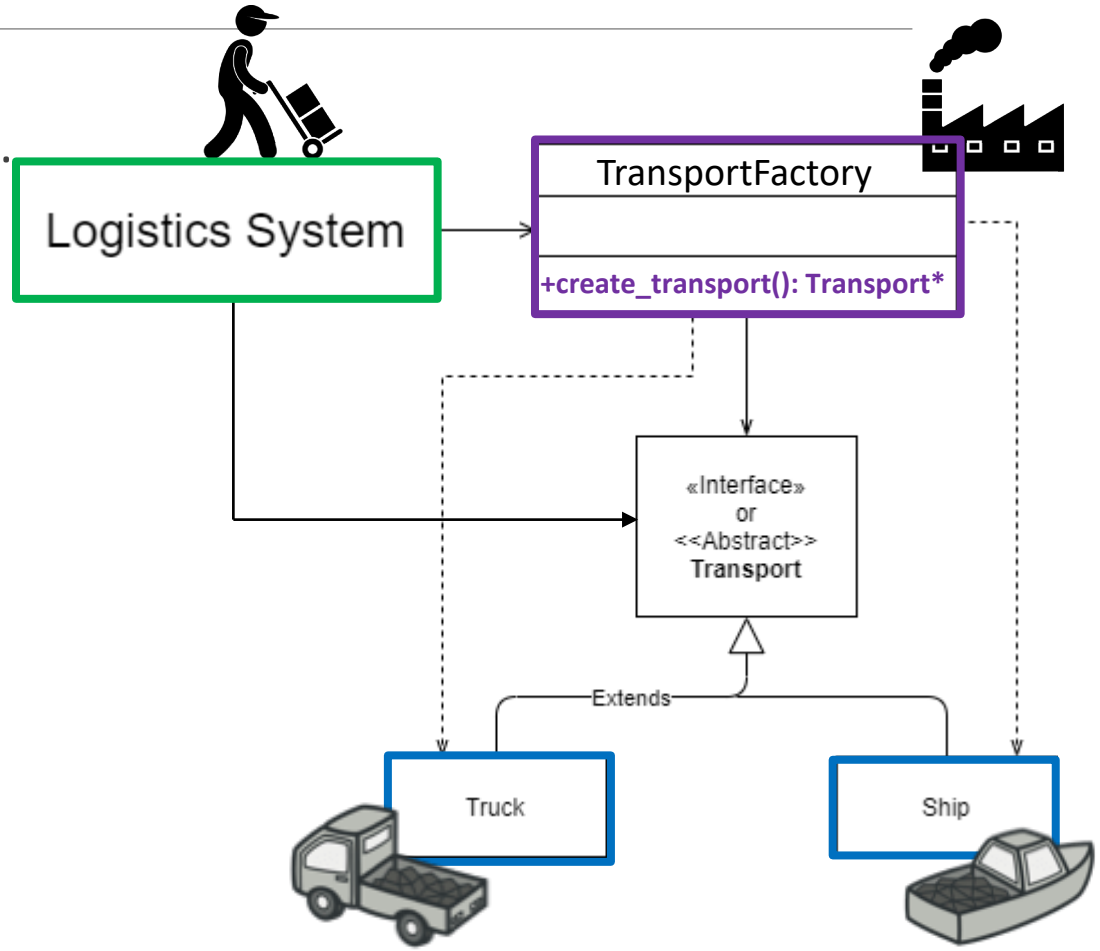
- So when it comes to creating these objects we need another entity to handle the responsibility of creating a specific type of **Transport**.
- We need a **factory** which will create a concrete **Truck/Ship** and return it to the **Logistics System**.
- As far as the **Logistics System** is concerned.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



Factory Pattern

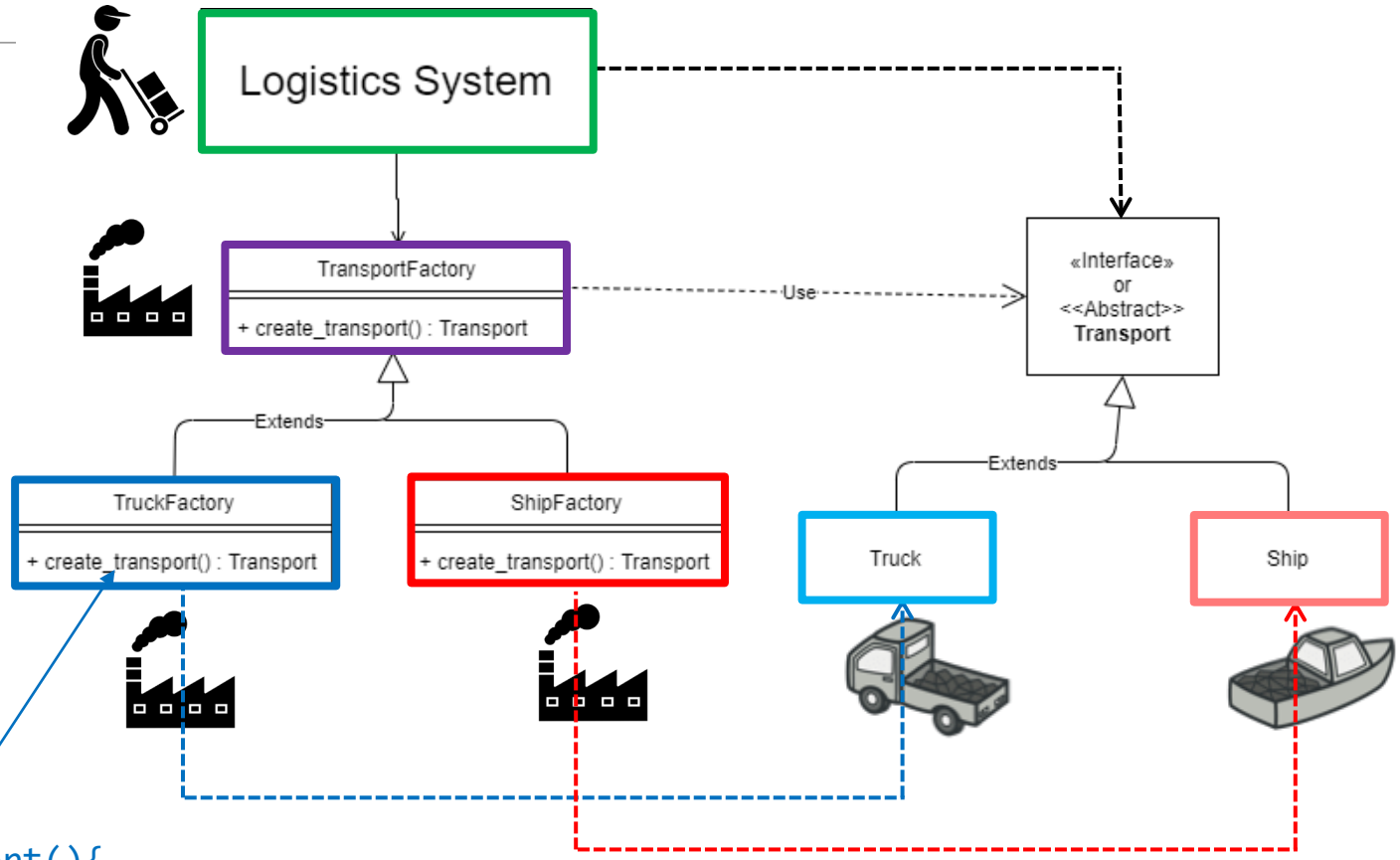
- This still isn't enough. All we have done is move the call to our **Truck()**, **Ship()** constructor to the **TransportFactory**. This isn't that useful.
- Say our **TransportFactory** has a create_transport() method. Now this method will have a massive **if-else** or switch statement block where 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();
}
```



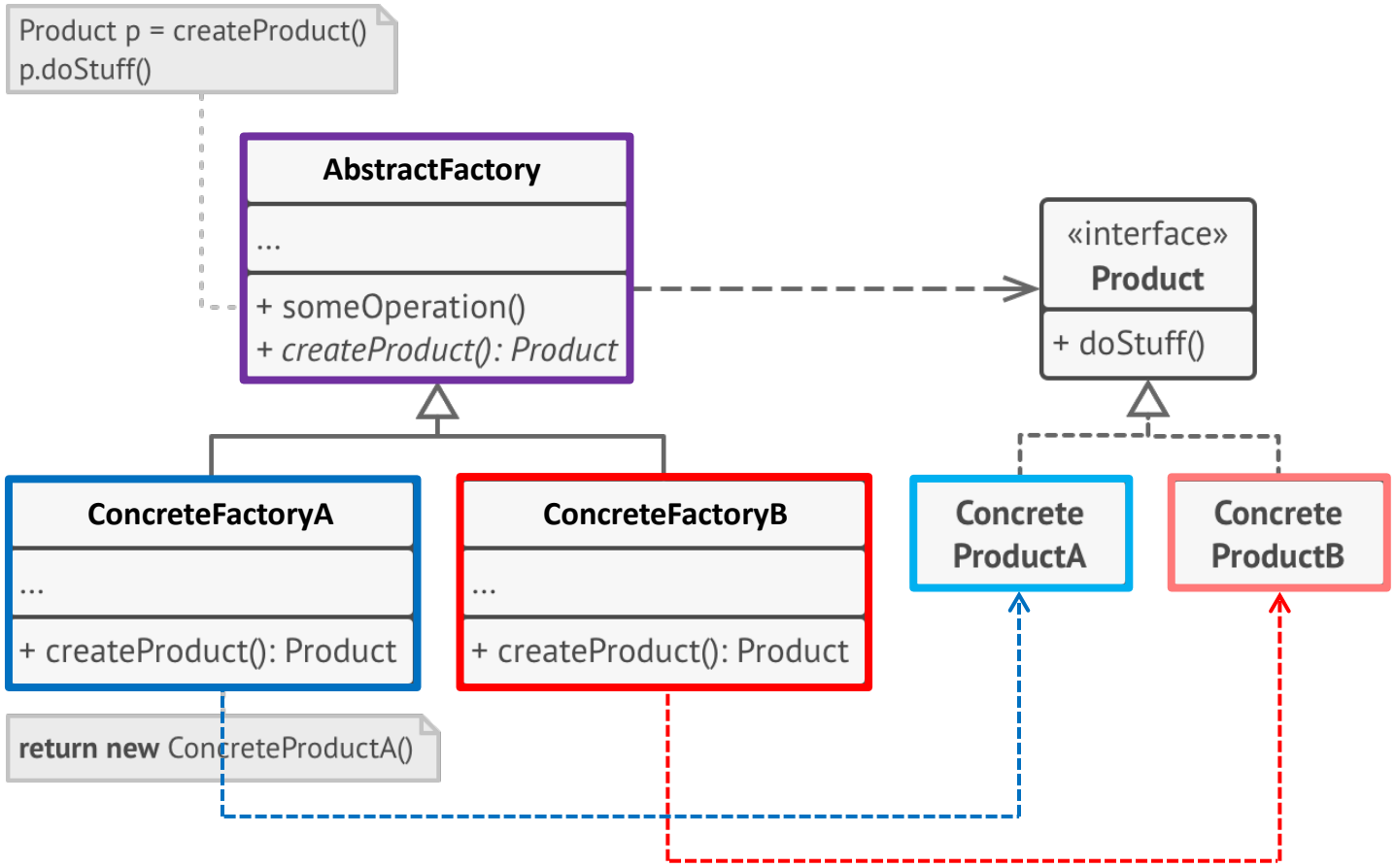
Factory Pattern

- 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();
}
```

Factory Pattern



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**.

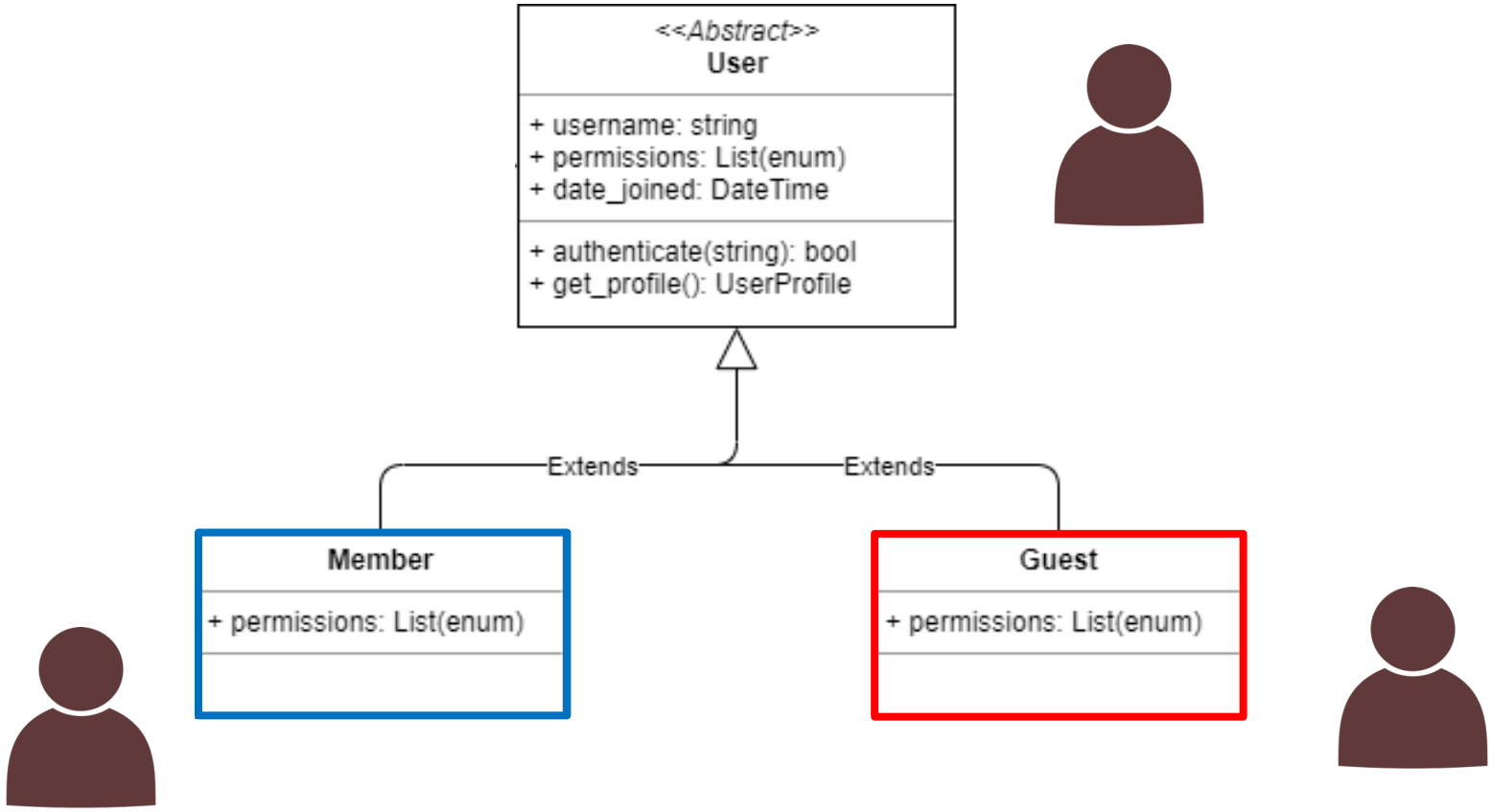
A **Guest** has the following **permissions**:

- Read Posts
- Like
- Share Posts
- Flag Posts

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

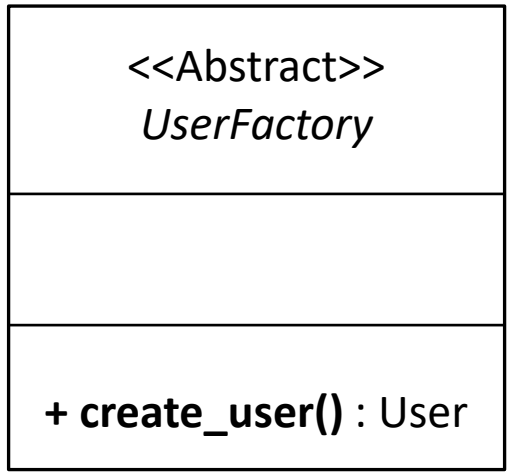
Step 1: Let's create a UML Class diagram depicting **the User hierarchy.**

Step 1: Define a product hierarchy



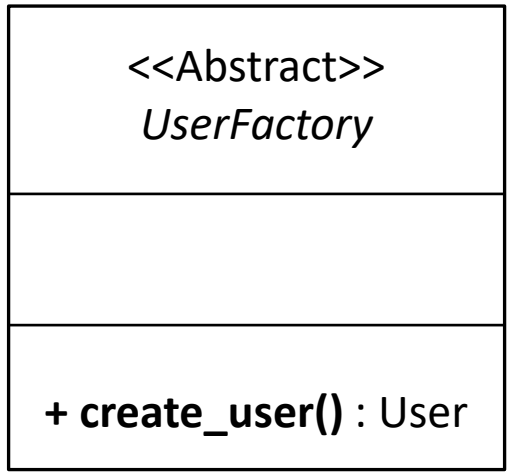
Step 2: Define a factory class

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



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.



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

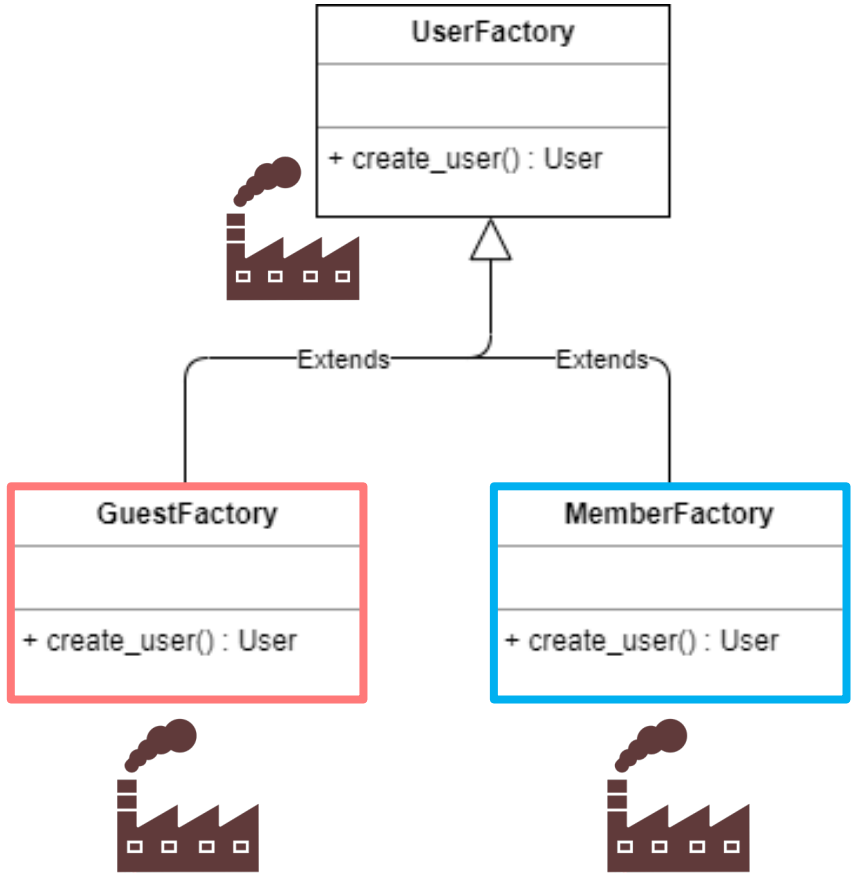
The title says it all.

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**

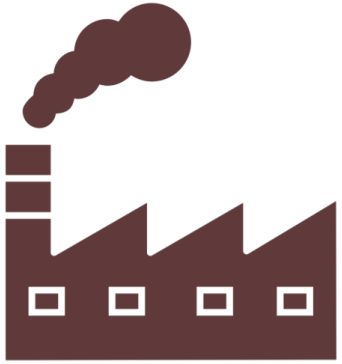
- **MemberFactory** creates **Members**

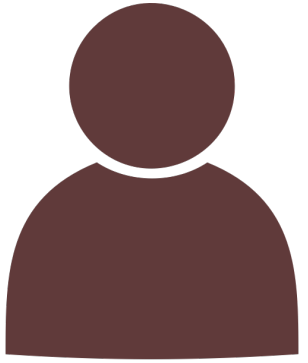


Step 4:
Integrate it into our Forum Website

Bring it all together.

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





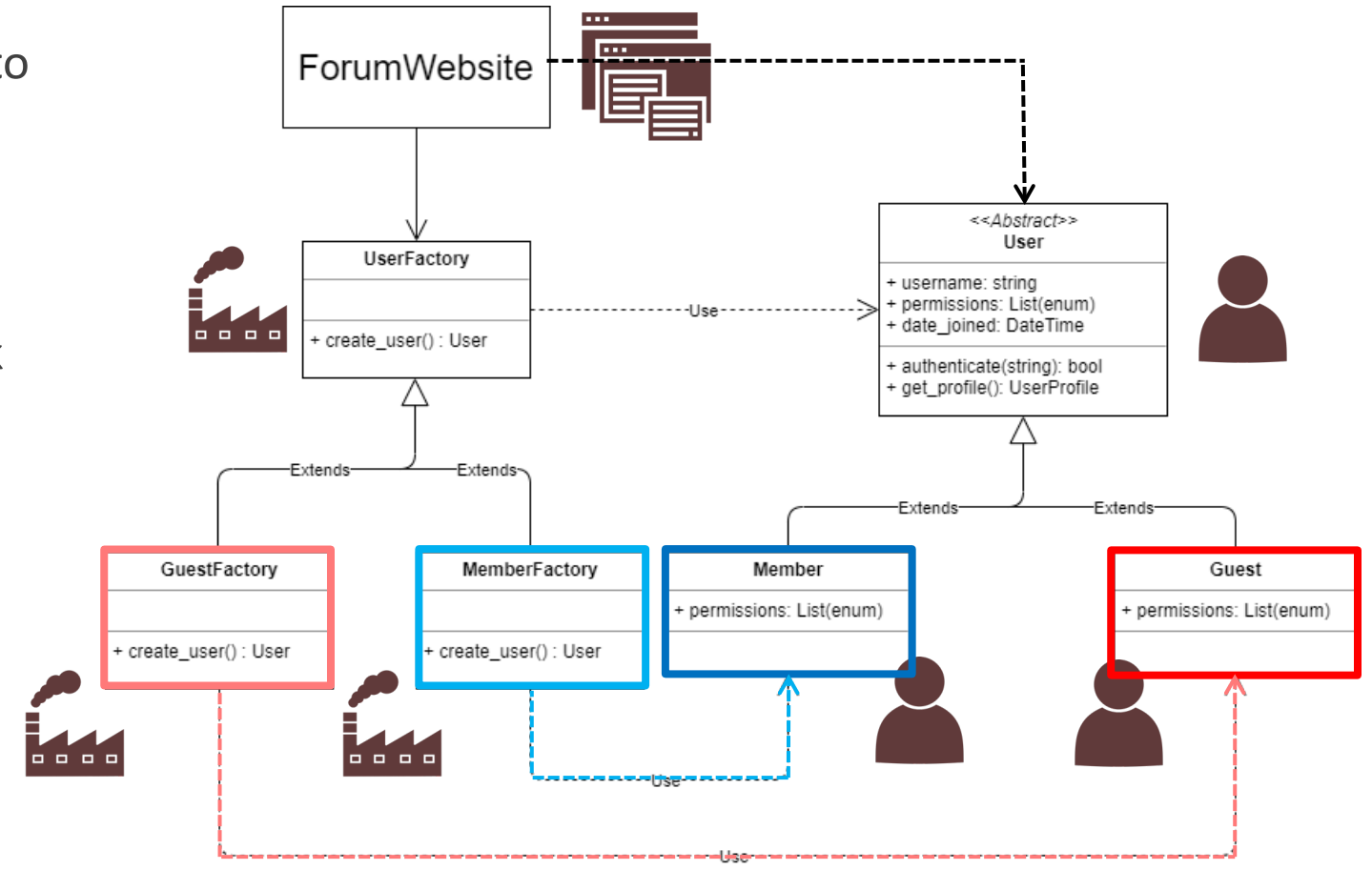
Step 4:
Integrate it into our Forum Website

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



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.

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.