Dependency Injection, Review

COMP3522 OBJECT ORIENTED PROGRAMMING 2

WEEK 13

Dependency Injection

WHEN YOU WANT DEPENDENCIES BUT YOU DON'T WANT TO HARDCODE THE TYPE OF DEPENDENCY

Dependency Injection

We are going to end with some theory today.

Dependency Injection is a:

- Concept
- Technique
- Framework
- A phrase that you may hear often, especially when working with complex systems.

This sounds scary but once you understand the concept it feels obvious.

In fact. You have all written a fairly complex system that implements Dependency Injection already.

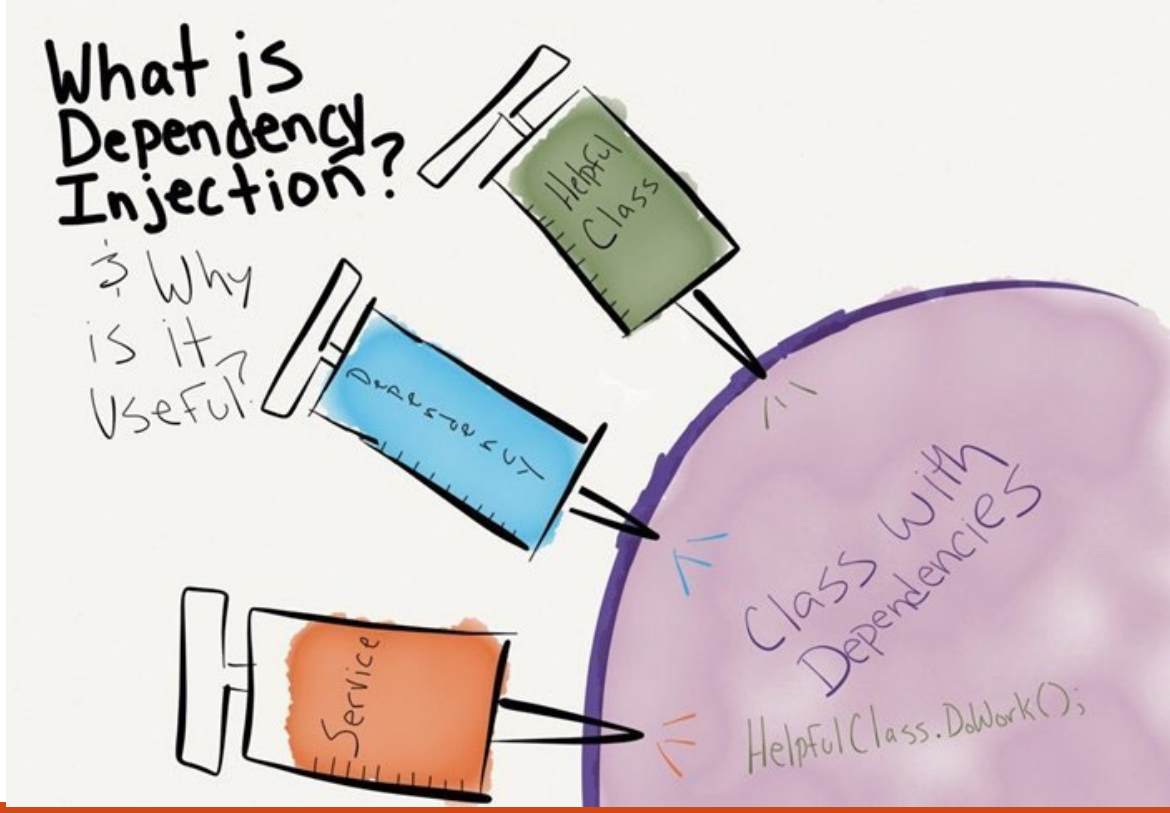
Dependency Injection – Concept

Class A is dependent on an entity B (a class, a system, an external package, etc.)

Our goal through many design patterns is to control and manage these dependencies.

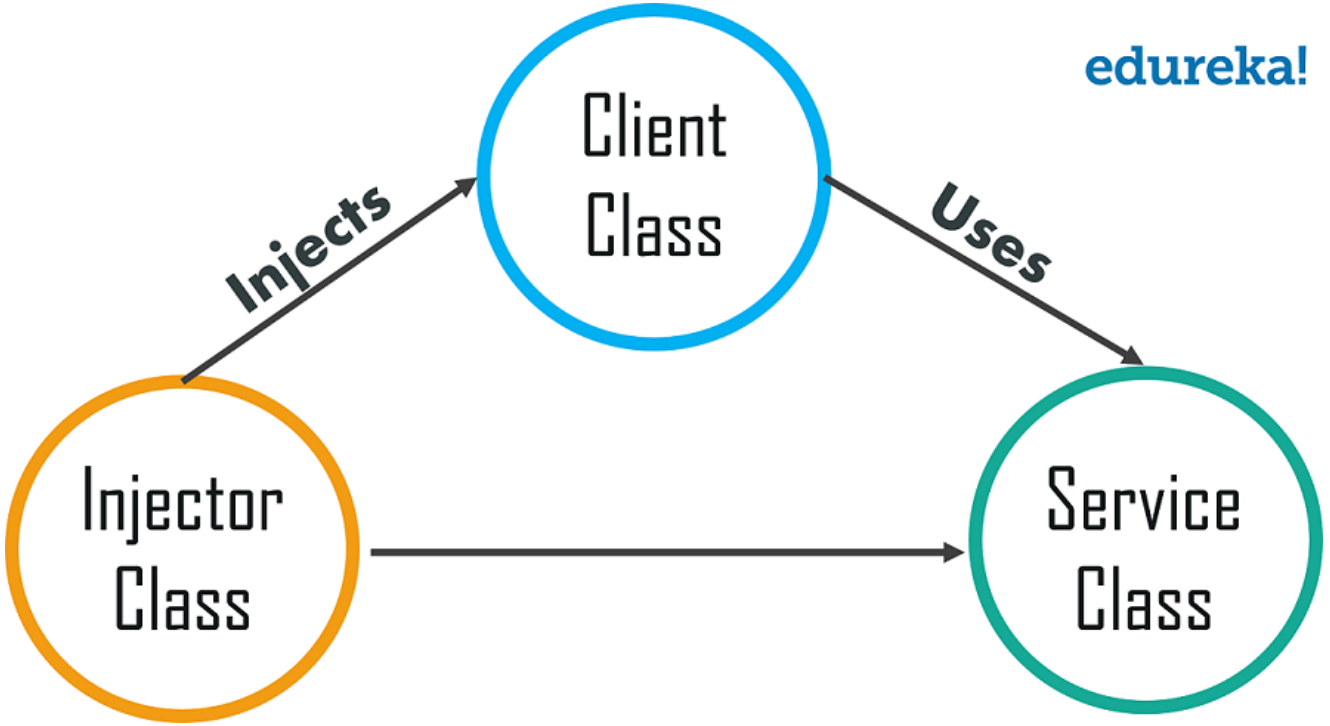
How can we provide a dependency to a dependent class in a way that reduces coupling?

Dependency Injection is a concept and a set of software design principles that enable us to write code that can answer this question.



Dependency Injection

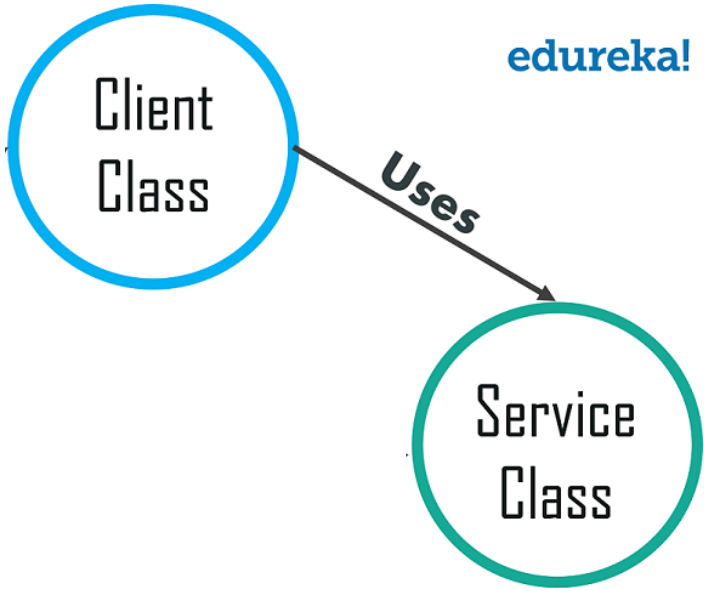
Dependency Injection involves having an external entity (**the Injector**) be responsible for creating (or retrieving) the right dependency for the use case and providing it to the client class.



Dependency Injection involves having an external entity (**the Injector**) be responsible for creating (or retrieving) the right dependency for the use case and providing it to the client class.

To decouple **Class A** from **Class B** and achieve dependency injection:

1. **Class A** should **not** explicitly refer to **Class B**.




```
class ClassA {
    ClassB* class_b;
    ClassA() {
        class_b = new ClassB();
} }
```

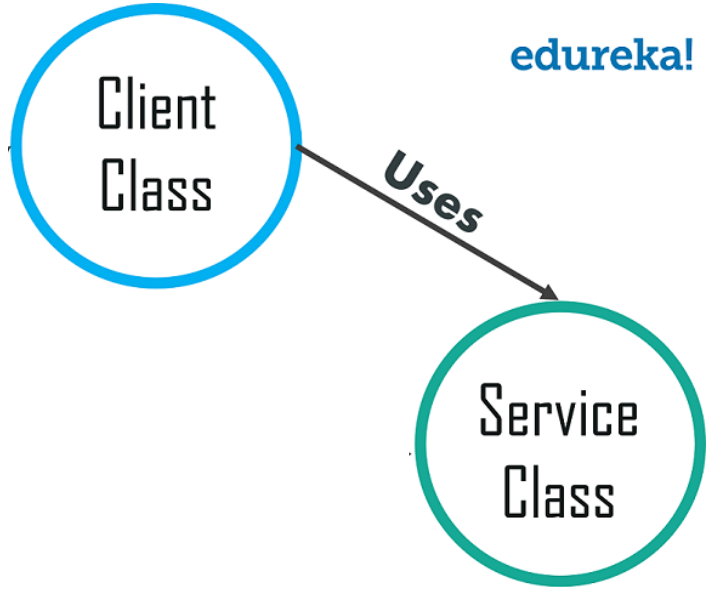
```
class ClassB {}
```

Dependency Injection involves having an external entity (**the Injector**) be responsible for creating (or retrieving) the right dependency for the use case and providing it to the client class.

To decouple **Class A** from **Class B** and achieve dependency injection:

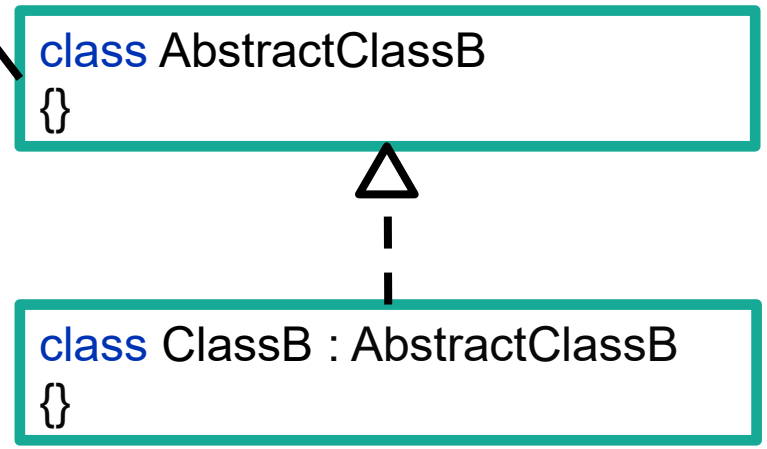
---

1. **Class A** should **not** explicitly refer to **Class B**.
   - Not mention its type (For languages with static typing)
   - **Refer to an abstraction** instead. Usually a common interface of some sort.



A little better...

```
class ClassA {
    AbstractClassB* abstract_class_b;
    ClassA() {
        abstract_class_b = new ClassB();
} }
```

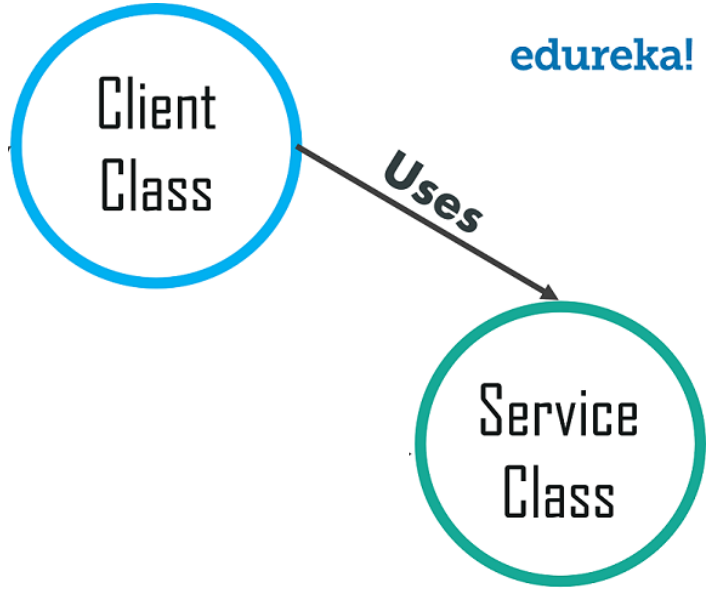


Dependency Injection involves having an external entity (**the Injector**) be responsible for creating (or retrieving) the right dependency for the use case and providing it to the client class.

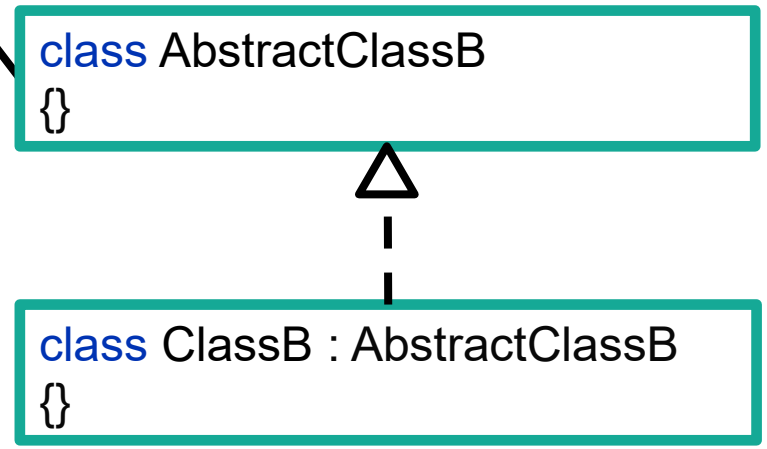
To decouple **Class A** from **Class B** and achieve dependency injection:

---

2. **Class A** should **NOT** be responsible for **constructing or importing/retrieving the dependency**




```
class ClassA {
    AbstractClassB* abstract_class_b;
    ClassA() {
        abstract_class_b = new ClassB();
} }
```

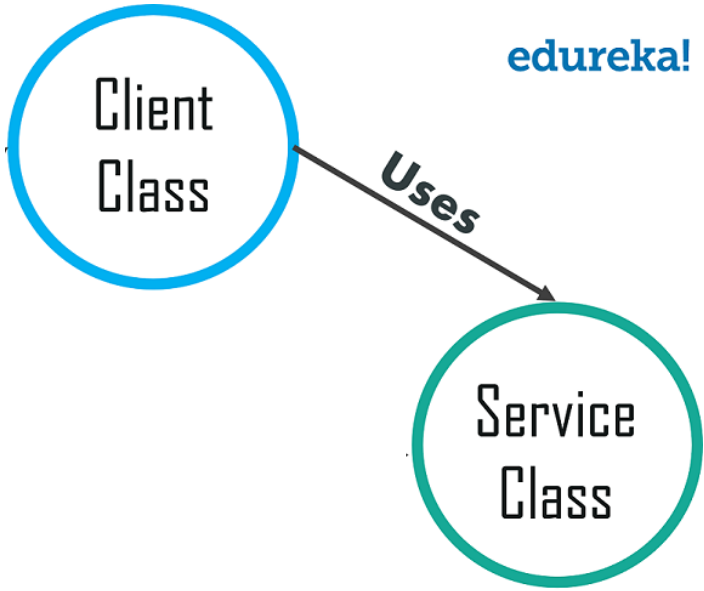


Dependency Injection involves having an external entity (**the Injector**) be responsible for creating (or retrieving) the right dependency for the use case and providing it to the client class.

To decouple **Class A** from **Class B** and achieve dependency injection:

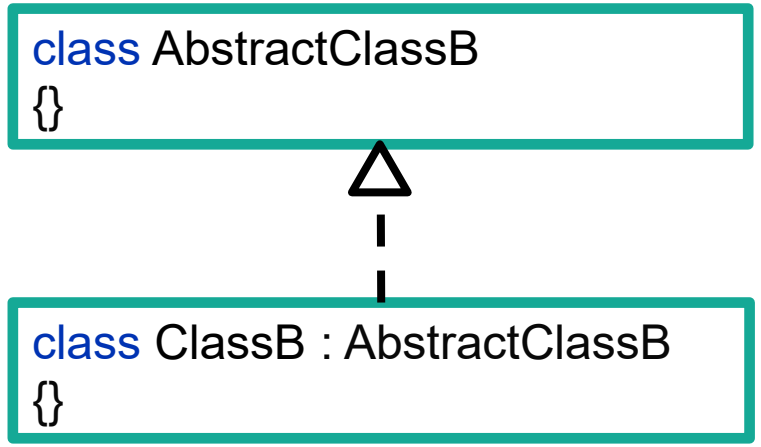
---

2. **Class A** should **NOT** be responsible for **constructing or importing/retrieving the dependency**




```
class ClassA {
    AbstractClassB* abstract_class_b;
    ClassA() {
        abstract_class_b = nullptr;
} }
```

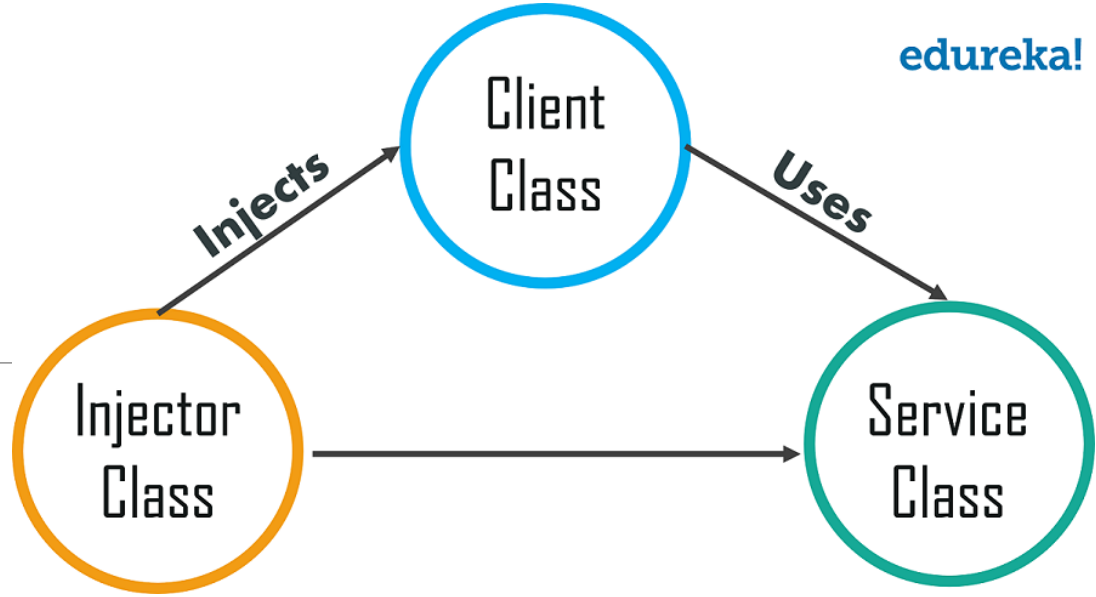
Better… But can't access ClassB…



Dependency Injection involves having an external entity (**the Injector**) be responsible for creating (or retrieving) the right dependency for the use case and providing it to the client class.

To decouple **Class A** from **Class B** and achieve dependency injection:

3. **Class B** should be **constructed (or imported/retrieved) by an external entity**. (**The Injector**)




```
class ClassA {                                   BEST!
    AbstractClassB* abstract_class_b;
    ClassA(AbstractClassB* b) {
        abstract_class_b = b;
} }
```

Uses

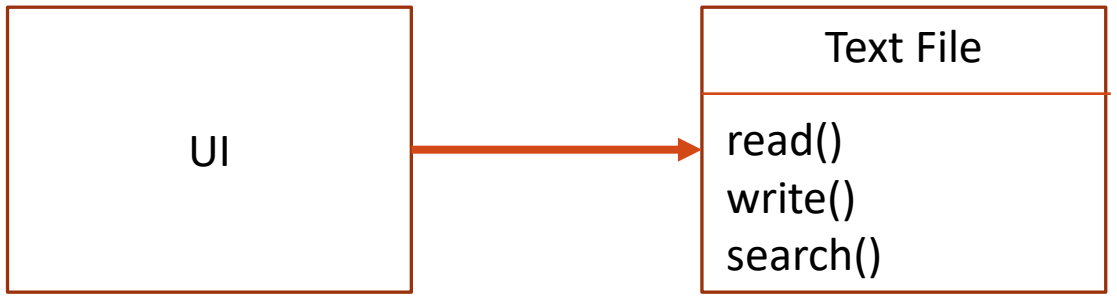
```
class AbstractClassB
{}
```

Injects

```
int main() {
  AbstractClassB* b = new ClassB();
  Class A class_a = ClassA(b); }
```

```
class ClassB : AbstractClassB
{}
```

RECAP: High Level & Low Level Modules



Remember this slide from way back when we were talking about SOLID principles?

Let's go back to this example. Here we can say that the UI is dependent on a Text File.

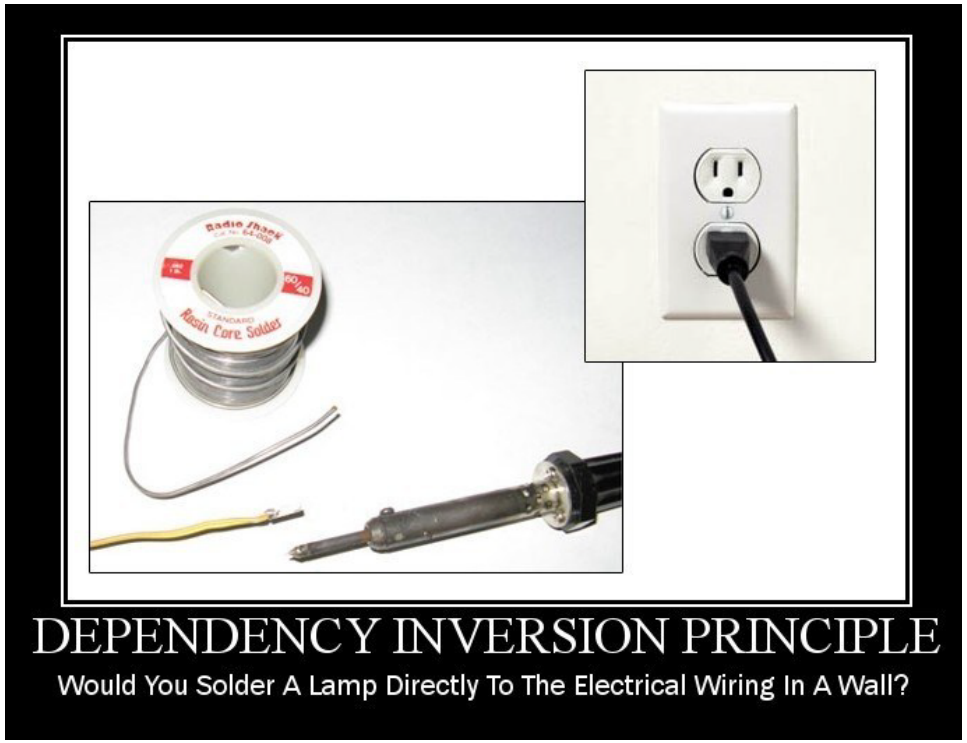
Dependency Inversion Principle Revisited

1. High-level modules should not depend on low-level modules. **Both should depend on abstractions**.
2. Abstractions should not depend on details. **Details should depend on abstractions**.

We call classes that implement details ‘Concrete Classes’. The Text File class is a concrete class.

Abstract classes declare an interface.

Concrete classes actually contain the code that implements the details of the interface.



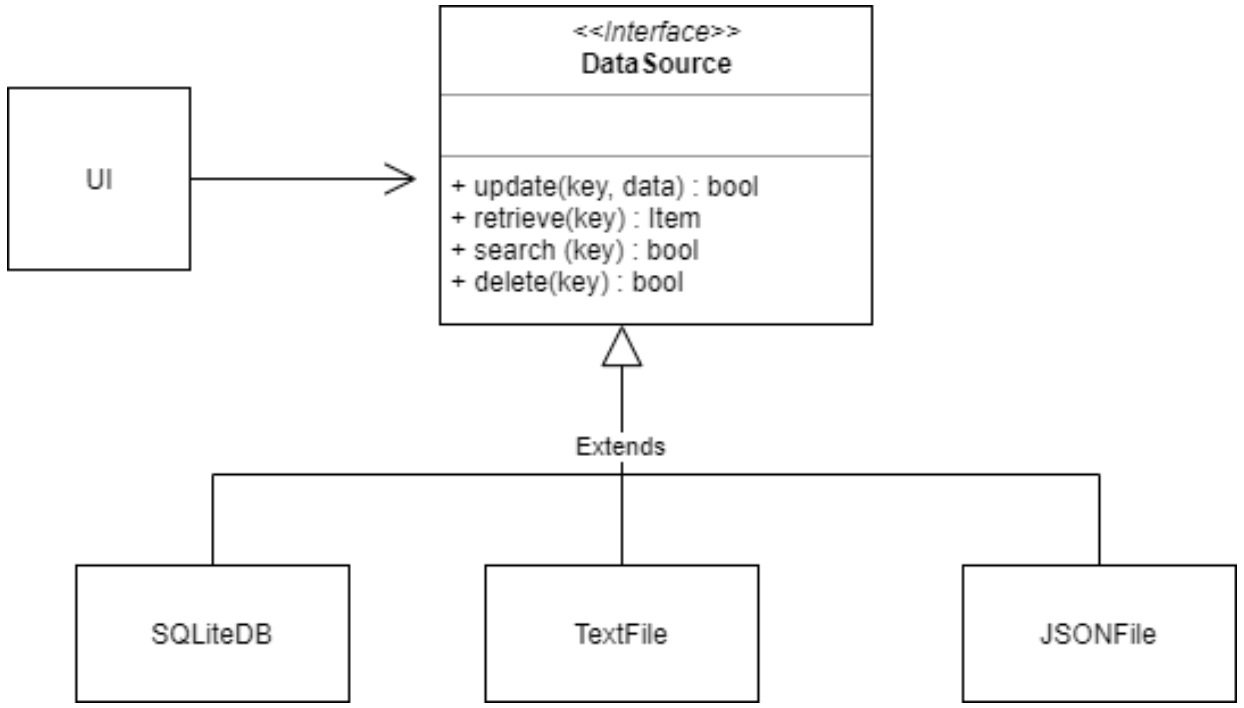
Dependency Inversion Principle Revisited

The **UI Class avoids direct references to the SQLiteDB, TextFile or JsonFile classes**.

It is now dependent on a DataSource Abstraction.

So we met one of the requirements of Dependency Injection. In the UI class we **no longer have a hardcoded reference to any of the 3 low level modules.**

But if the UI class can't instantiate a instance of the low level modules, **then how does it have access to one?**



Constructor Dependency Injection

```
struct UI {
    DataSource *data_source;
    UI(DataSource* data_source) {
        this->data_source = data_source //data source
                                          injected
    }
..
..


int main() {
    DataSource *json_file = new JSONFile("data.json")
    UI ui(json_file); //inject json_file into ui
}
```

We can do this via the **constructor**.

By providing the UI class with a **pointer** to a **concrete DataSource** the **main method** has injected the dependency.

The UI class was not responsible for importing the concrete class. It was done by an **external entity**.

This is an example of **Constructor Dependency Injection.**

dependencyInjection.cpp

Factory Patterns as Dependency Injection

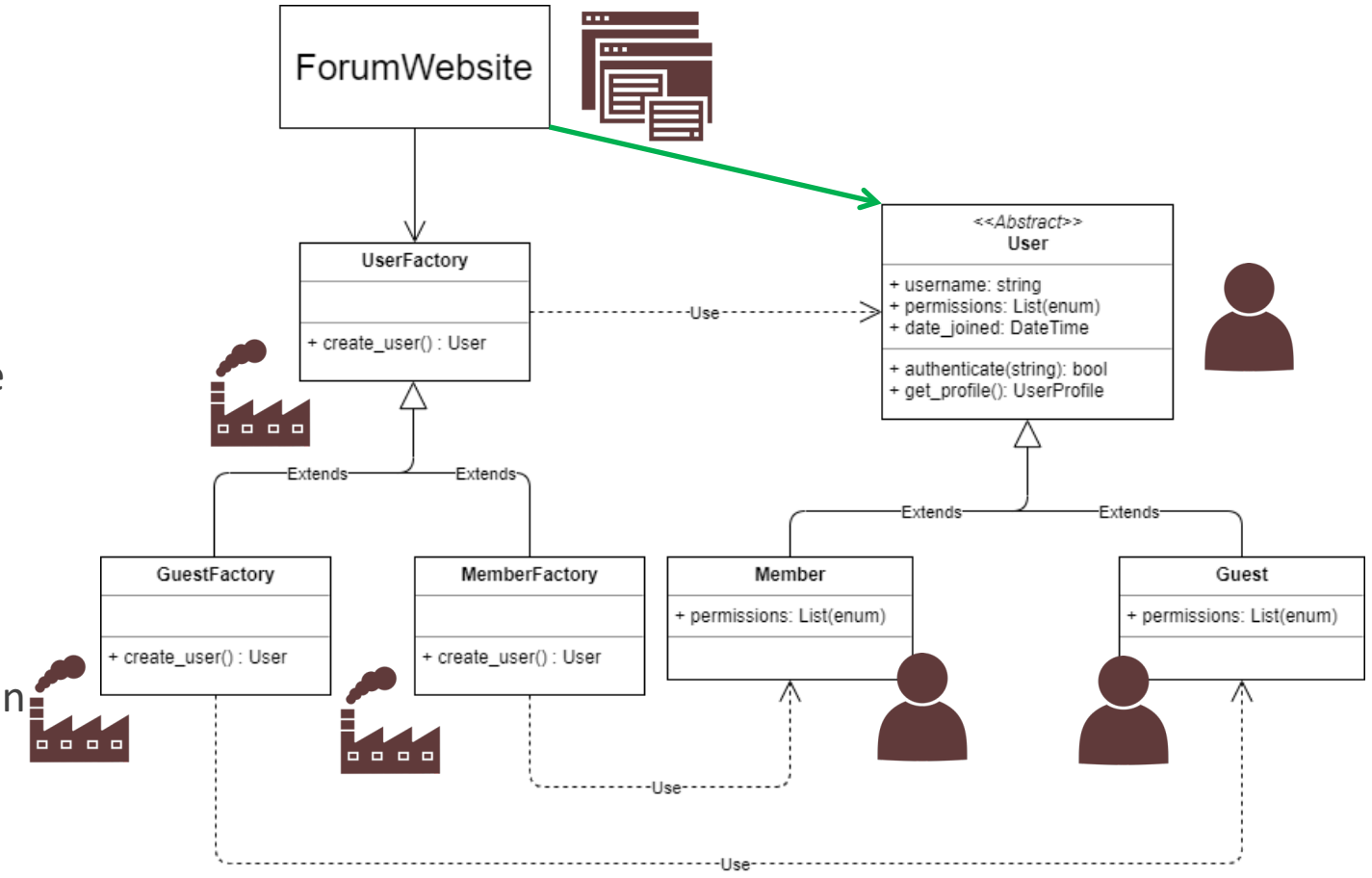
The **Factory Pattern** is a way of implementing **Dependency Injection**.

The client is dependent on a low level module. We implement the Dependency Inversion Principle by making the **client depend on an abstract product Interface** (User).

The client then needs a way of having access to the User, so we create Factory Classes.

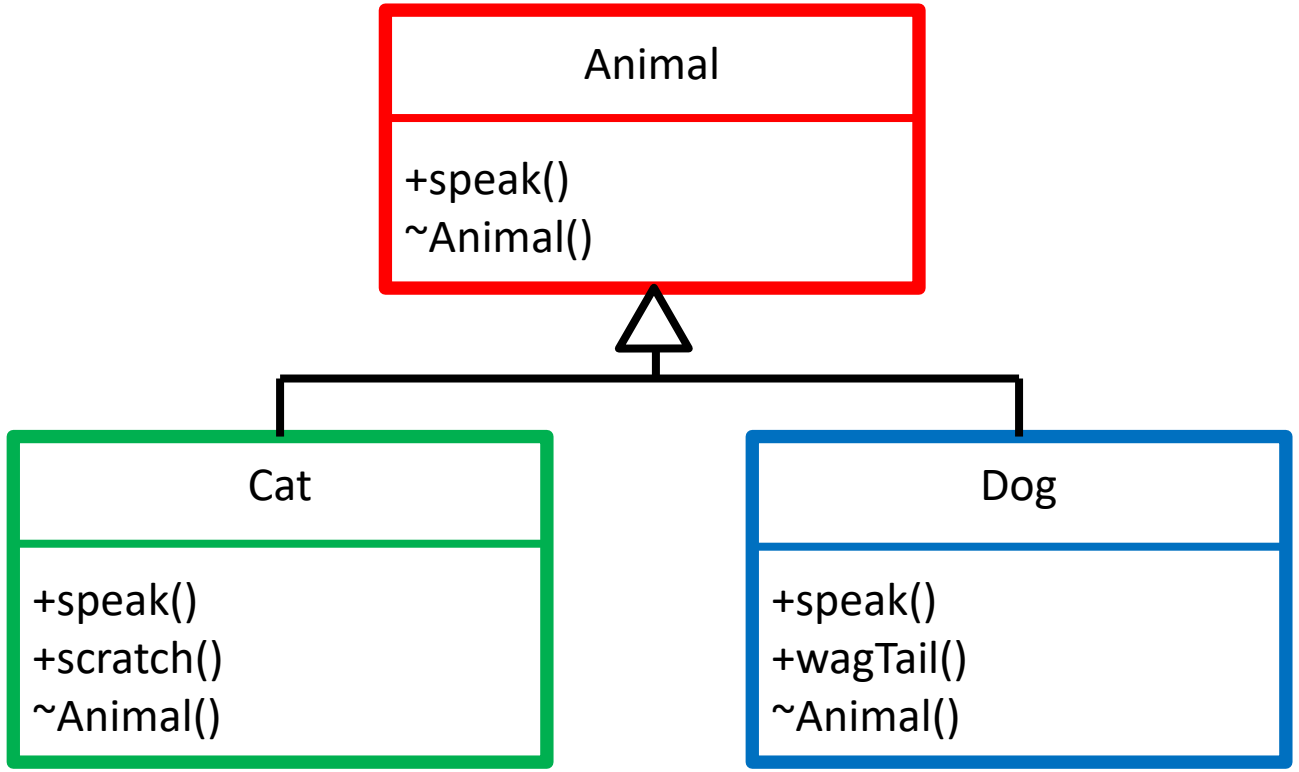
The Factory classes "Inject" the client with the required dependency. **The Factory class is the Injector.**

In this example, the Forum Website is dependent on concrete users. The **UserFactory injects the ForumWebsite with the right kind of user.**



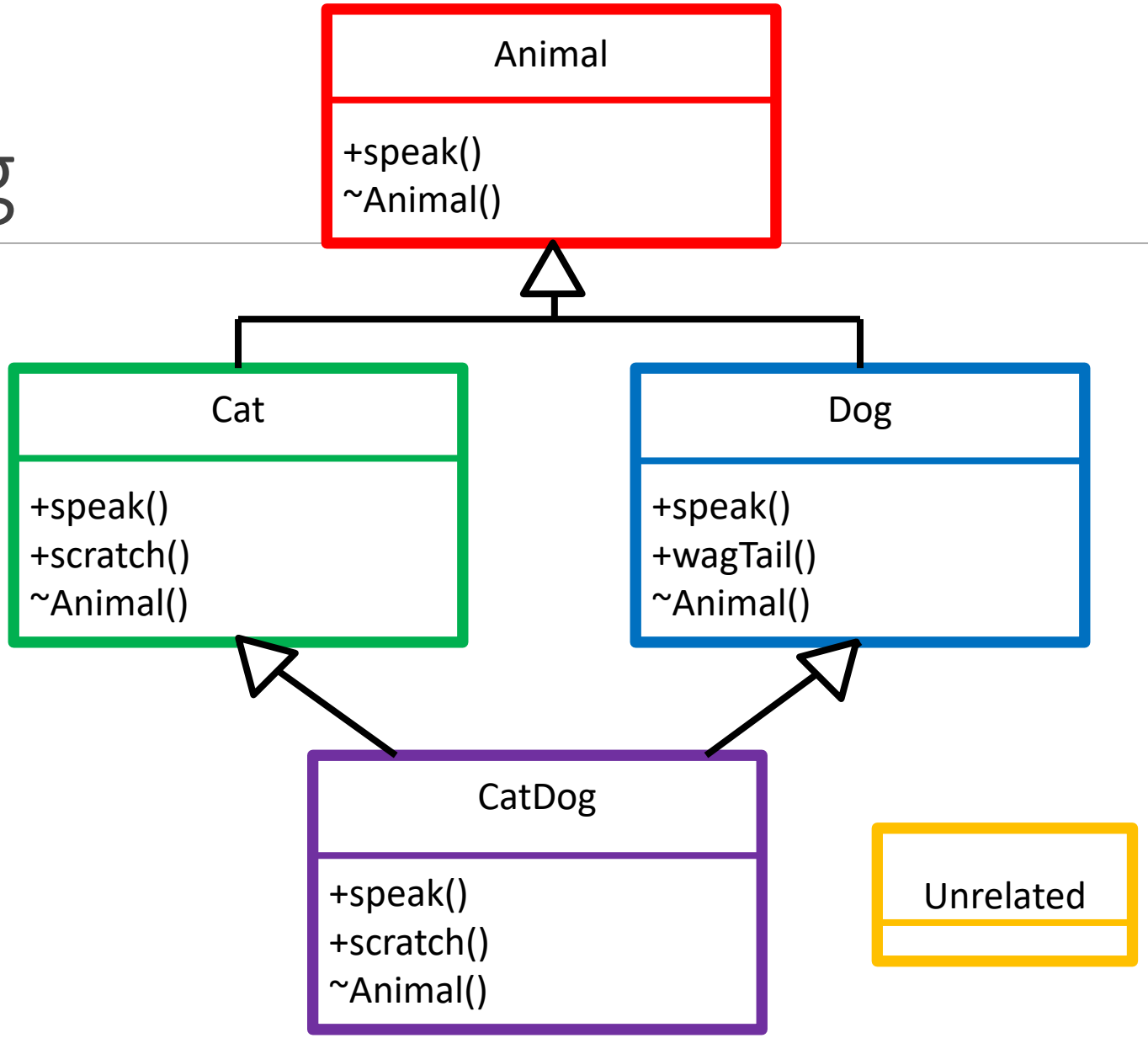
REVIEW CASTING

Casting



casting.cpp

Casting



cross_casting.cpp

DESIGN PATTERNS REVIEW

COMP3522 OBJECT ORIENTED PROGRAMMING 2

WEEK 14

Design Patterns - Advantages

- Don’t re-invent the wheel, use a proven solution instead
- Are abstract and can be applied to different problems
- Communicate ideas and concepts between developers
- Language agnostic. Can be applied to most (if not all) OOP programs.

Design Patterns - Disadvantages

- Can make the system more complex making the system harder to maintain. Patterns are deceptively ‘simple’.
- The system may suffer from pattern overload.
- All patterns have some disadvantages and add constraints to a system. As a result, a developer may need to add a constraint they did not plan for.
- Do not lead to direct code re-use.

What are Design Patterns

What's the best way to learn design patterns?

Which patterns to learn first?

Design pattern complexity

Low	Medium	High
1. Lazy Initialization 2. Façade 3. Singleton 4. Proxy	1. Bridge 2. Observer 3. Strategy 4. Builder 5. Factory	1. Abstract Factory 2. Mediator 3. State 4. Chain of Responsibility 5. Decorator

Categorizing Design Patterns

- **Behavioural**

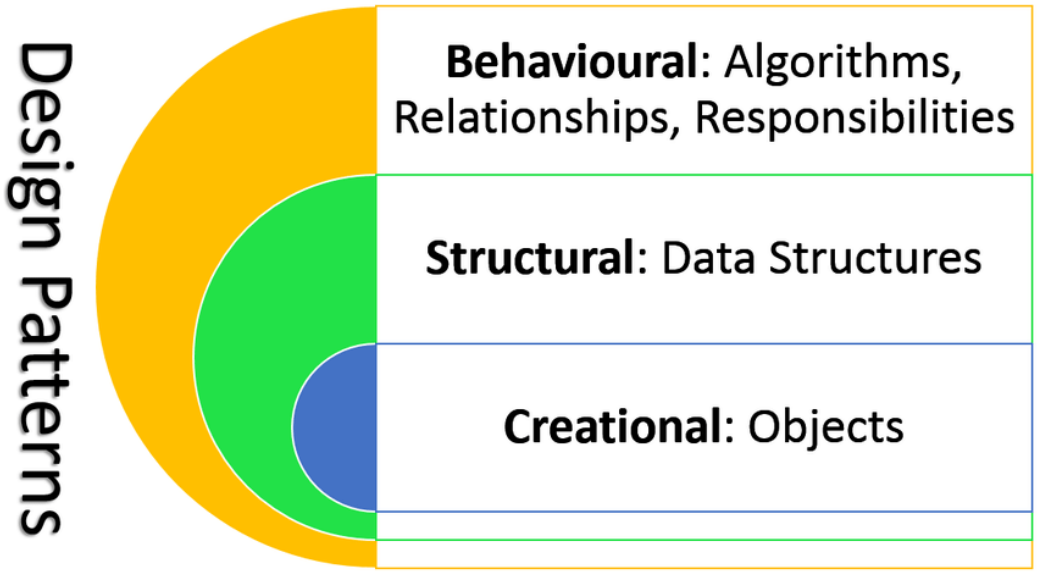
Focused on communication and interaction between objects. How do we get objects talking to each other while minimizing coupling?

- **Structural**

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

- **Creational**

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



Categorizing Design Patterns

- **Behavioural**

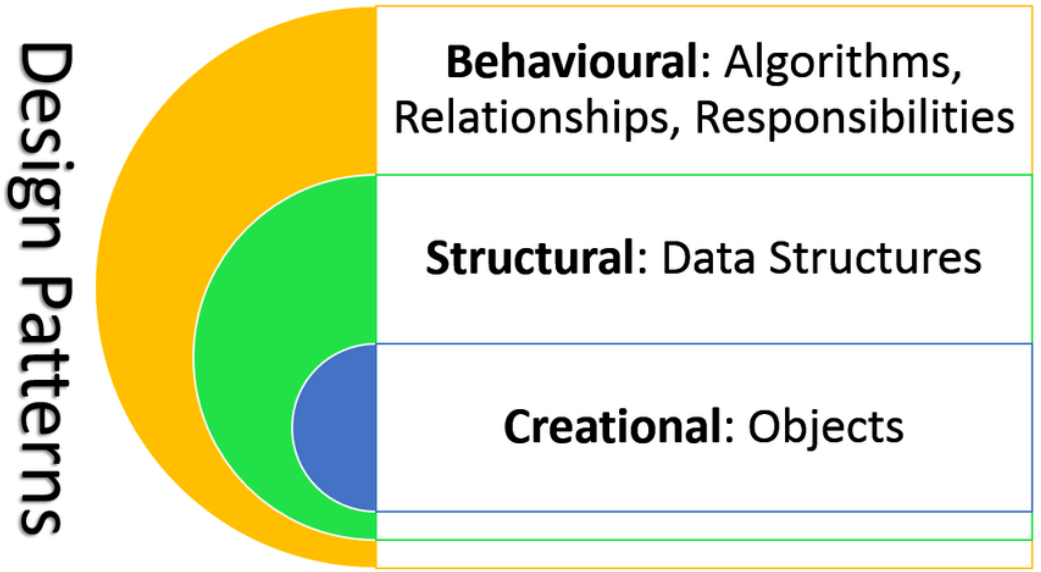
  Observer, Chain of Responsibility, Strategy, State, Mediator

- **Structural**

  Façade, Proxy, Bridge, Decorator

- **Creational**

  Singleton, Factory, Abstract Factory, Builder, Lazy Initialization

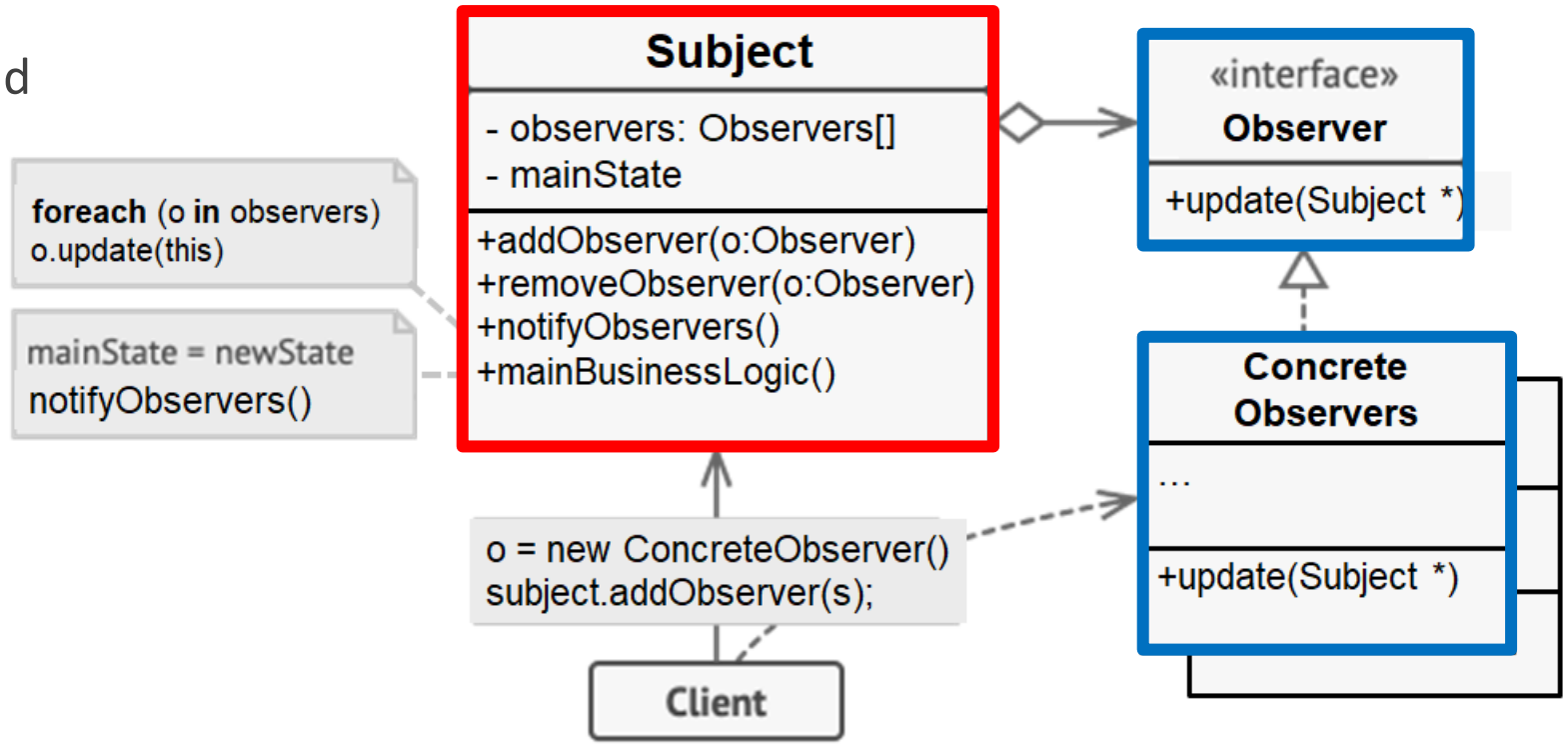


Behavioural - Observer

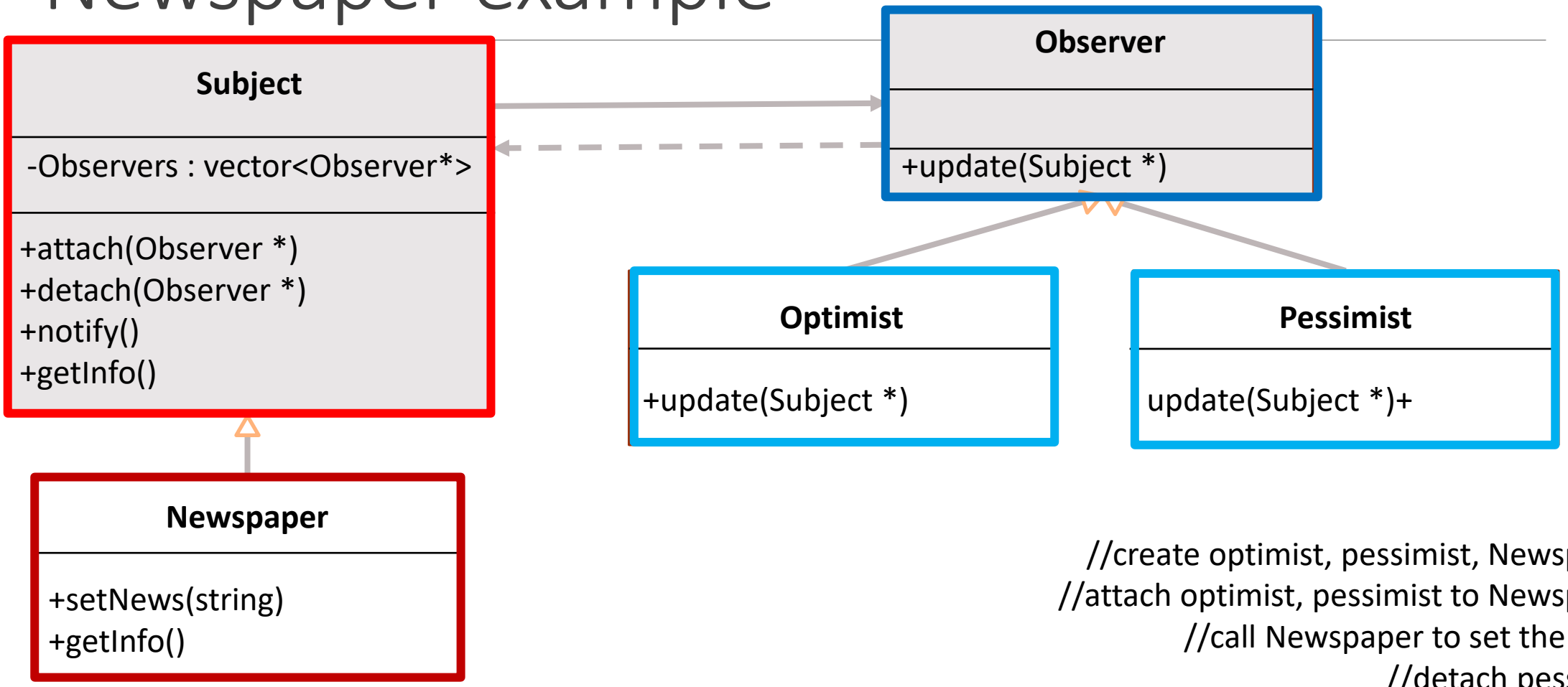
An object, called the **Subject**, maintains a list of its dependents, called **Observer**s, and notifies them automatically of any state changes

When you need an object (**Subject**) to notify other objects (**Observers**) but limit dependencies. Subject doesn't know the concrete types of observers

Create a system to model an auctioneer informing bidders of the current bid and accepting new bids



Newspaper example



Main

```
//create optimist, pessimist, Newspaper
//attach optimist, pessimist to Newspaper
//call Newspaper to set the news
//detach pessimist
//call Newspaper to set the news
```

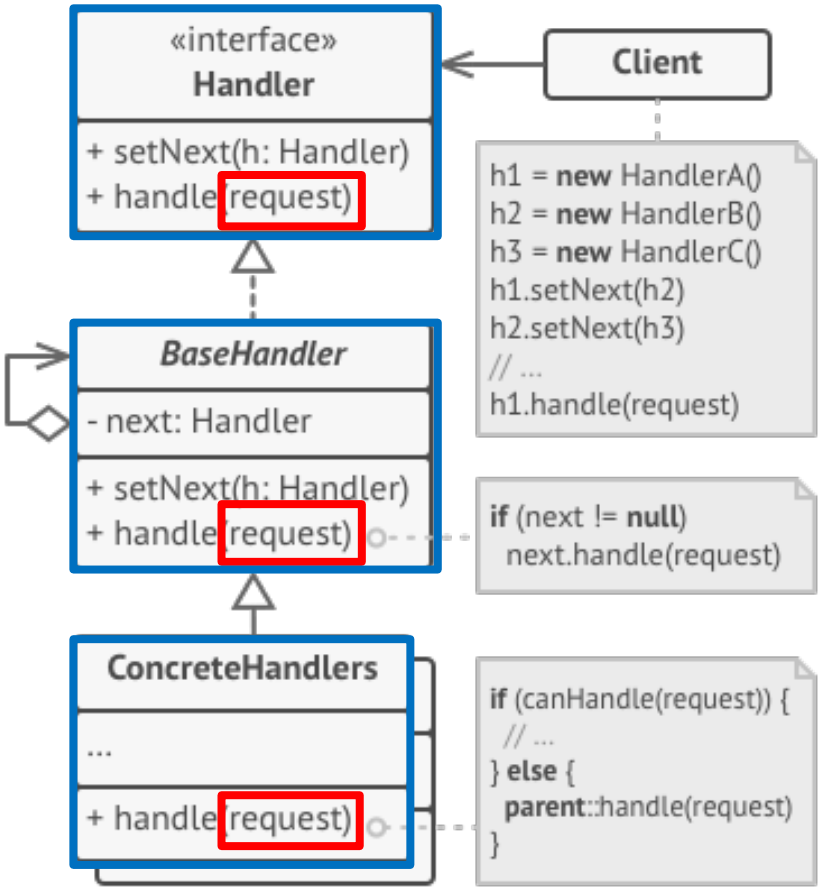
ObserverNewspaper.cpp

Behavioural – Chain of responsibility

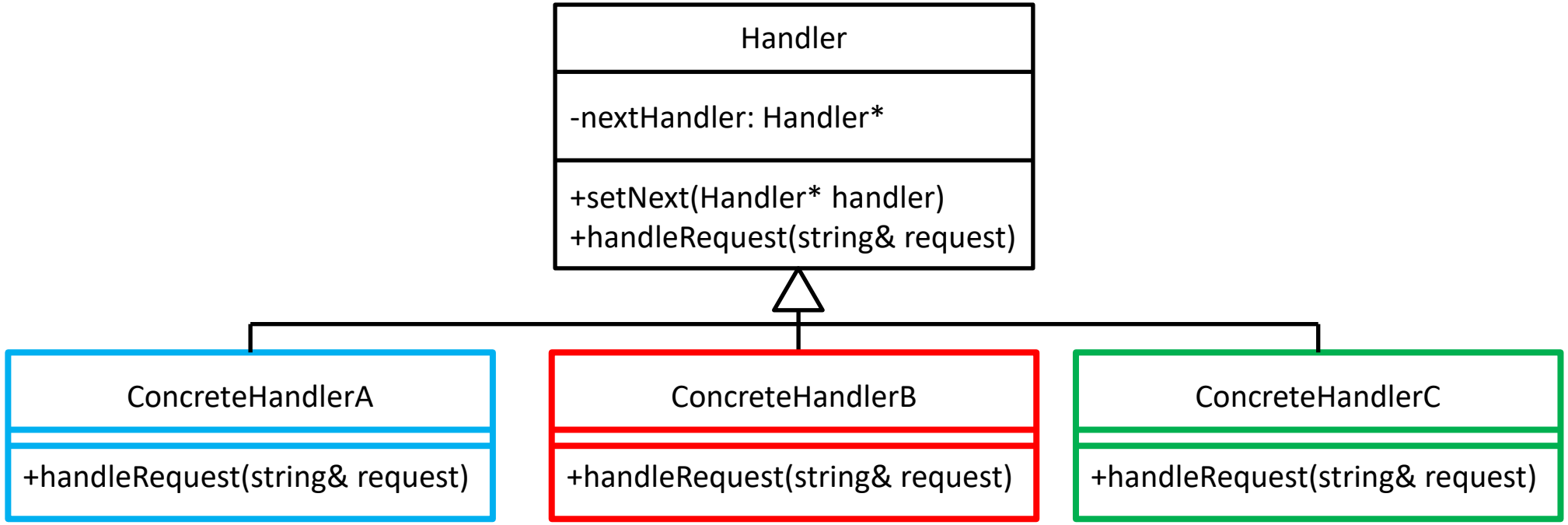
Use this when: a **request** needs to go through a **series of processing steps**. Decouples the sequencing of these steps and makes them re-usable

Example: A **document** needs to go through **several stages of validation**. Need to validate:

- **courses have correct name and year**
- **check fees paid**
- **student information correct**



Behavioural – Chain of responsibility



chain_of_responsibility.cpp

Behavioural – Strategy

Use this when: an **object** needs to switch **behaviours** at runtime

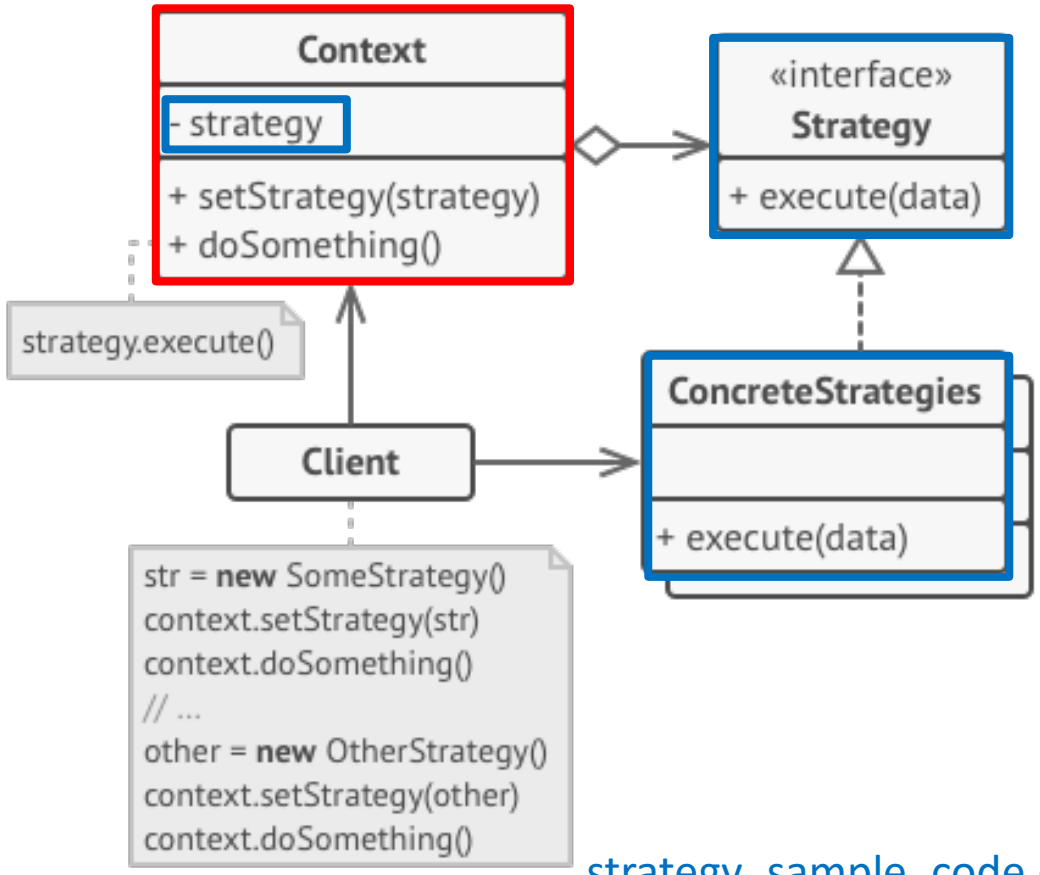
Example:

A game **character** wants to use different **weapons** when attacking

The character and weapon classes contain attack methods.

Change the character's attack behavior by swapping the weapon objects (axe, sword, hammer)

Character's attack method forwards calls to the weapon's attack method



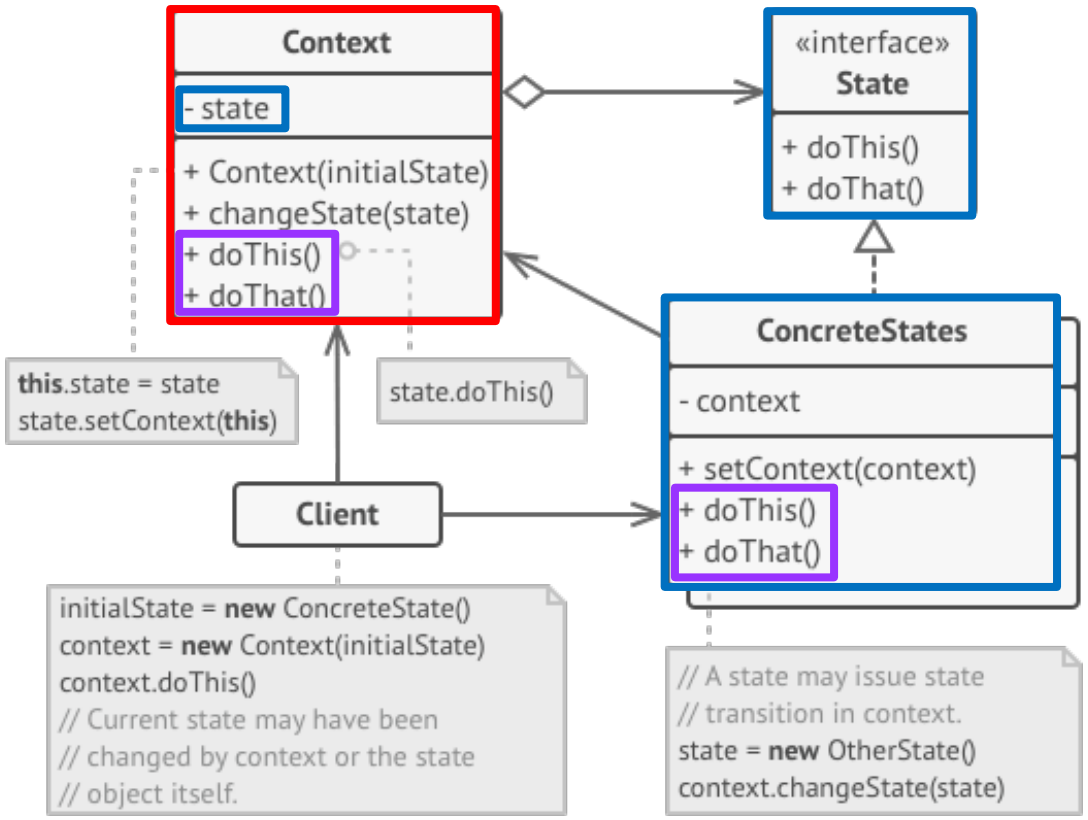
strategy_sample_code.cpp

Behavioural – State

When the same **object** can take on **different behaviours** depending on its **state**.

Having different specialized behaviours within the same class. The **context** changes their **state** by swapping out their internal state object. This causes all the **behaviours** in the context to use the new state object's behaviors

Use this when: you need almost all the behaviors of an object to change depending on its internal state

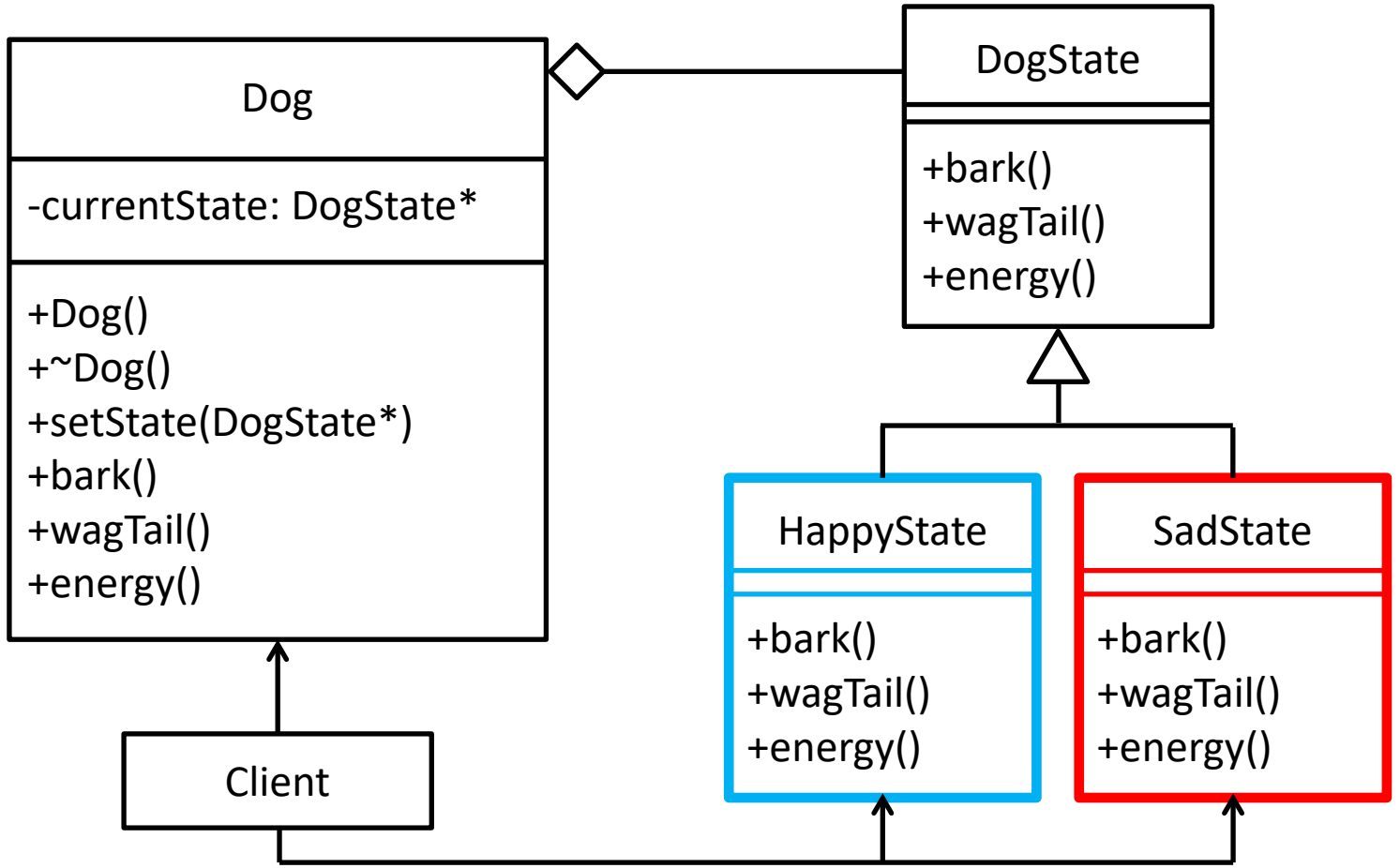


Behavioural – State

Create a system to simulate a dog behaving differently depending on its mood.

Happy – happily barking, wagging tail, high energy

Sad – whimpering, tail between legs, low energy



dogState.cpp

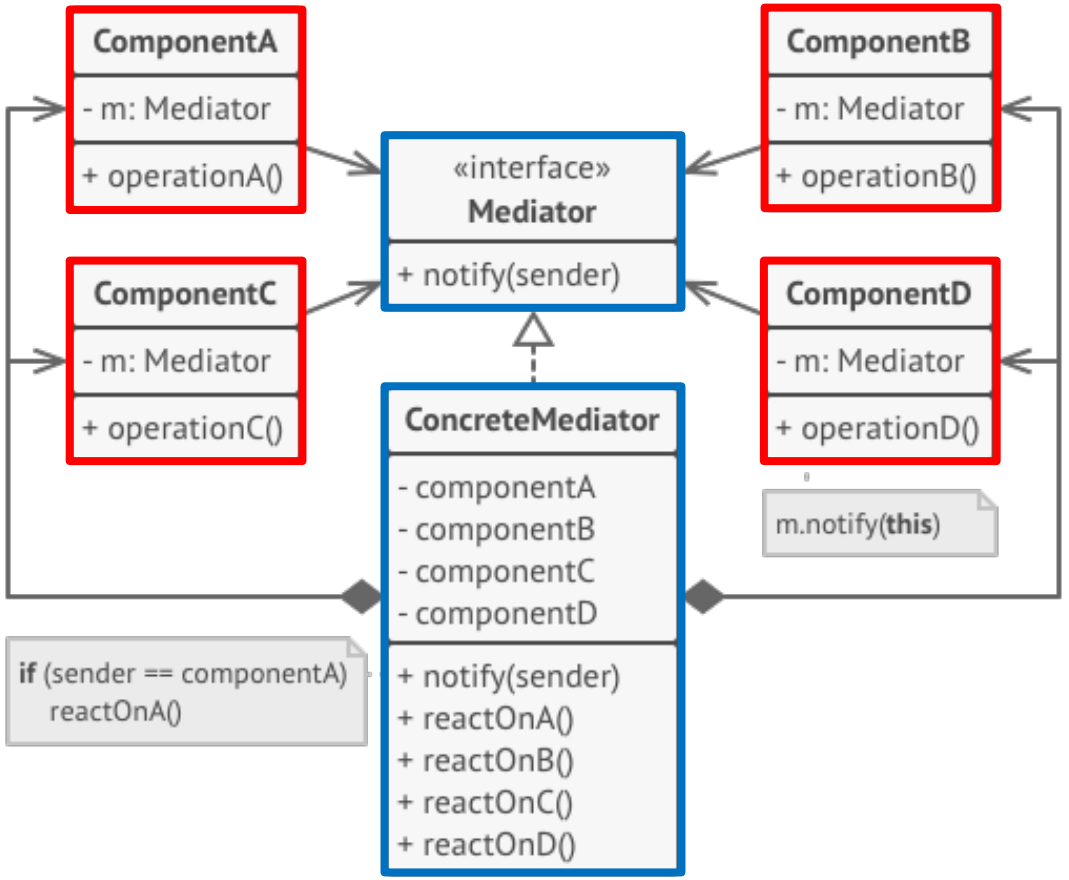
Behavioural – Mediator

Defines an **object** that encapsulates how a set of **other objects** interact

Components are unaware of other components, they only communicate with the mediator

When a **component** needs to send a message, they blindly notify **mediator**. Mediator object retrieves the notification and contains logic to call other components' behaviors

Use this when: you need to decouple **components** communicating with each other, and instead rely on a **central location** to handle communication



Behavioural – Mediator

Example: Model a traffic light system

A **TrafficLight** can change its color to red, yellow, and green

A **Car** will:

- stop on a red light

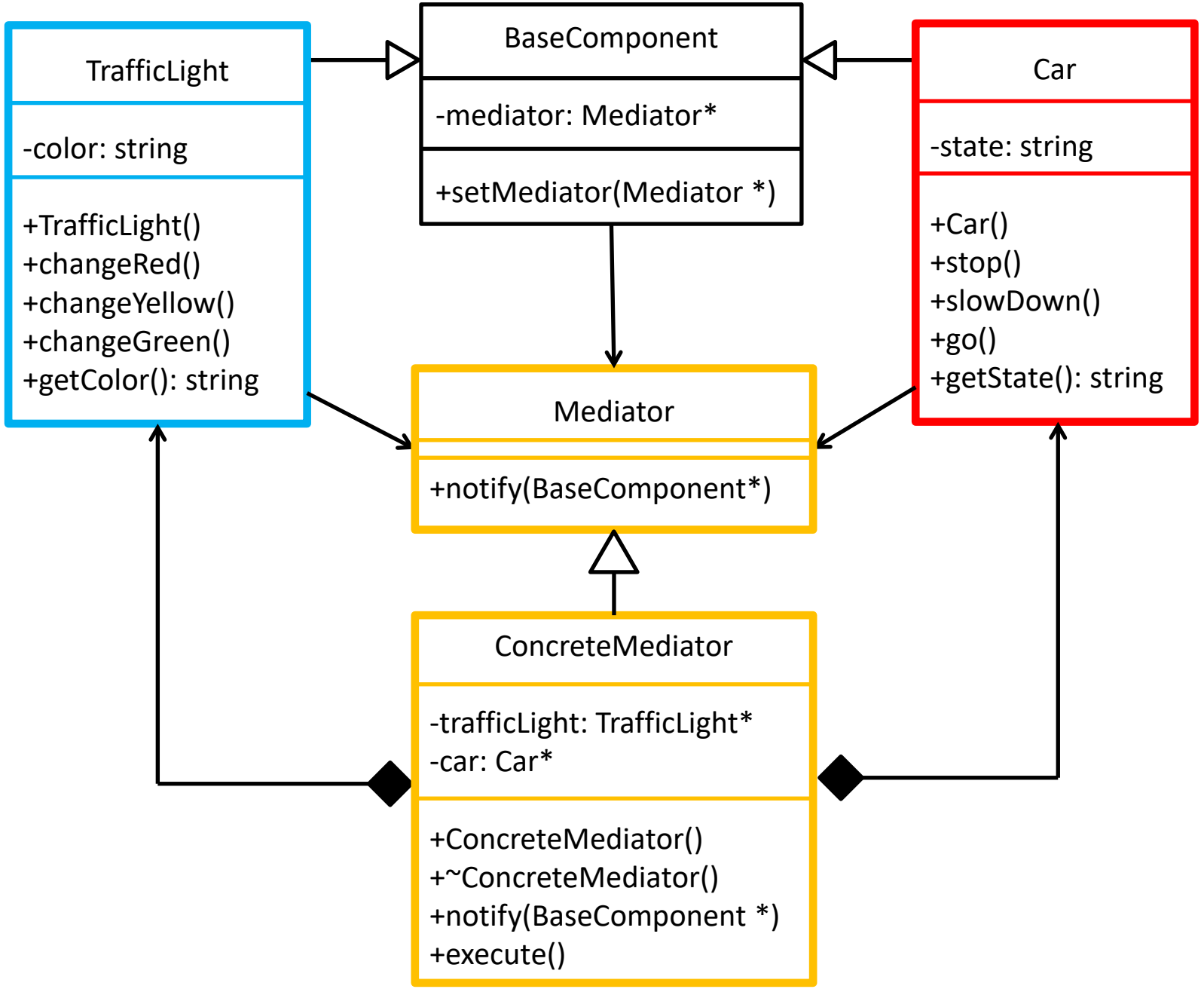
- slow down on a yellow light

- go on a green light

We need all these separate components to communicate with each other, but without coupling them

Have the **Mediator** handle passing messages between all the components

trafficMediator.cpp



Categorizing Design Patterns

- **Behavioural**

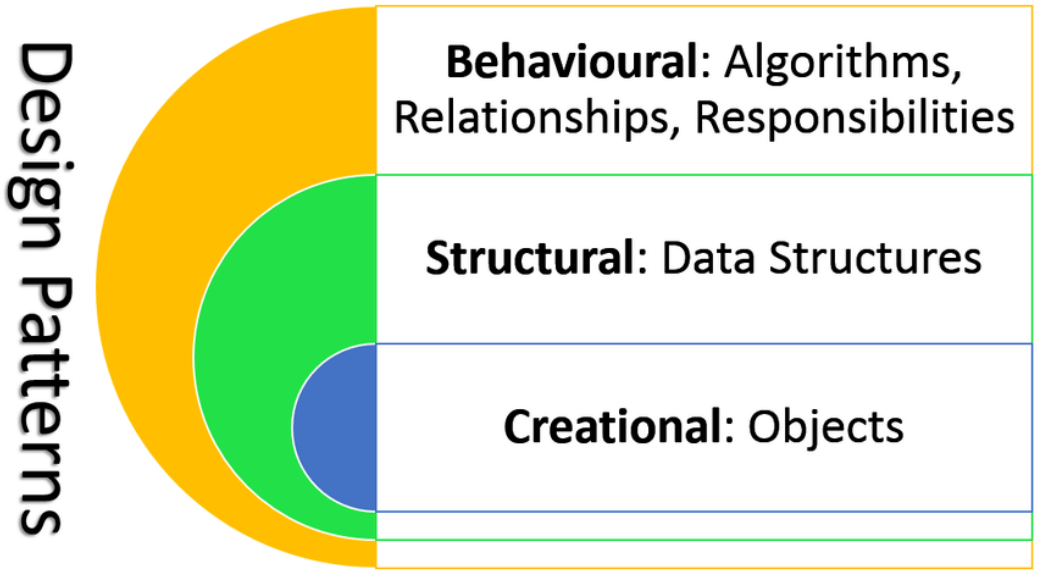
Observer, Chain of Responsibility, State, Strategy, Mediator

- **Structural**

Façade, Proxy, Bridge, Decorator

- **Creational**

Singleton, Factory, Abstract Factory, Builder, Lazy Initialization



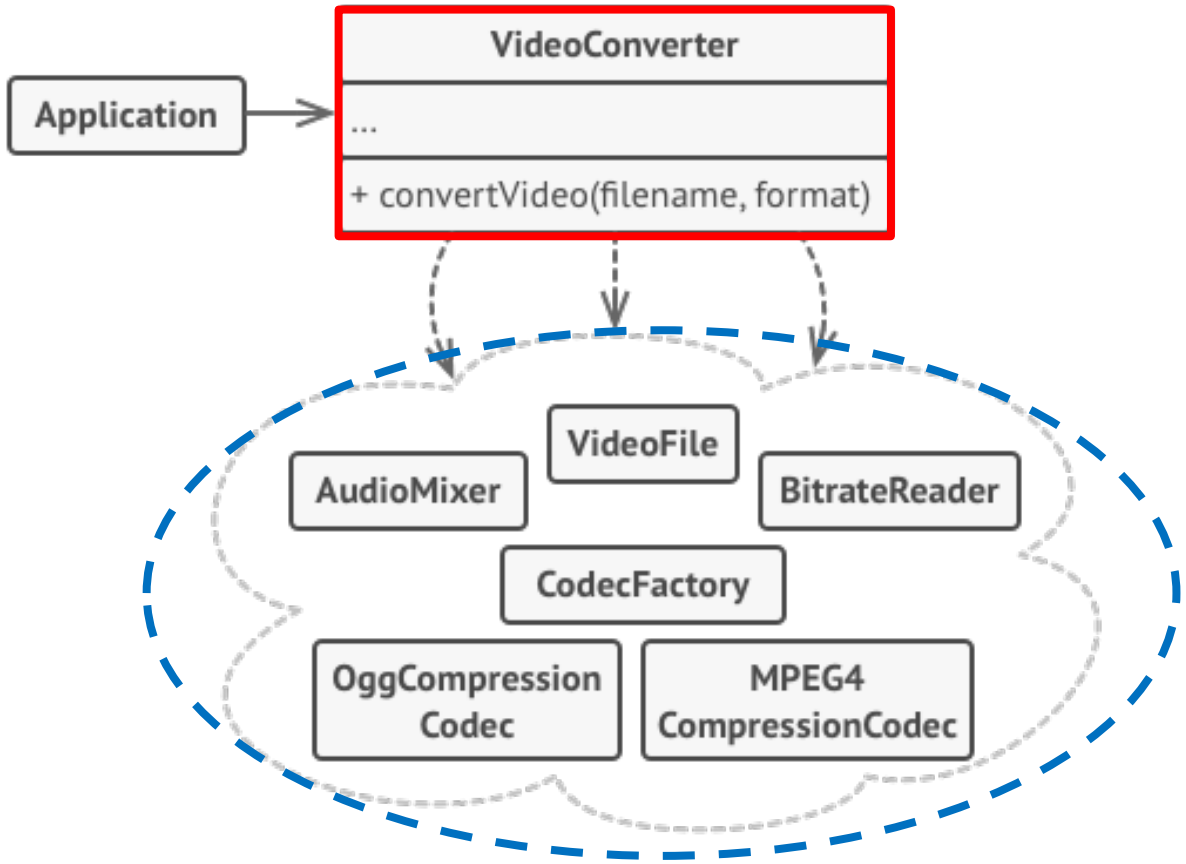
Structural – Façade

A **simplified interface** encapsulates a **complex system or API**. Client only interacts with the façade instead of complex systems

Use this when: you want to **hide the complexity of a system** from the client but allow them to use it in a **simplified form**.

Example: **Video converter**. Only want to expose the convert video function to the client. But behind the scenes, there are **many systems** the facade must call to perform the action

simple_facade.cpp
video_package_facade_example.cpp



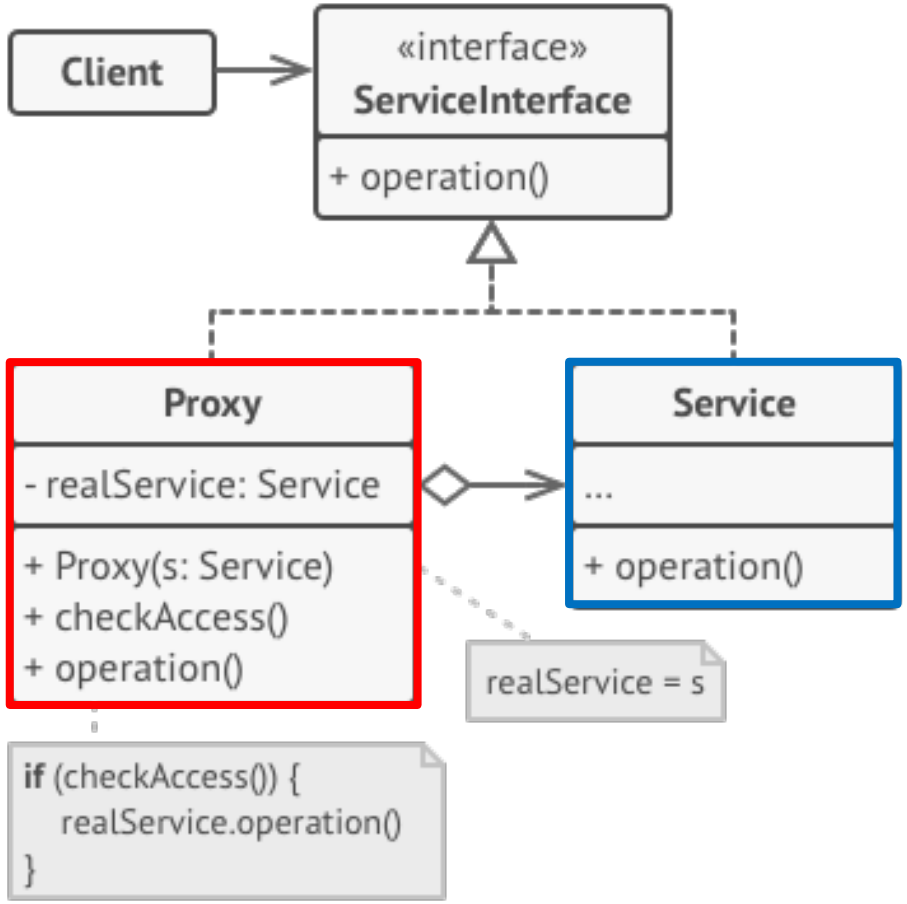
Structural – Proxy

A class functioning as an interface to something else. A **proxy is a wrapper** or agent object that is being called by the client to access the **real serving object** behind the scenes

Use this when: you have a **real service or object that you wish to hide the client** from directly interacting with.

Example: Youtube **database** is a real service. Prevent all users in the world to access one database. Create a **proxy object** that intercepts requests and forwards to Youtube database. Repeat requests are cached

simple_proxy.cpp



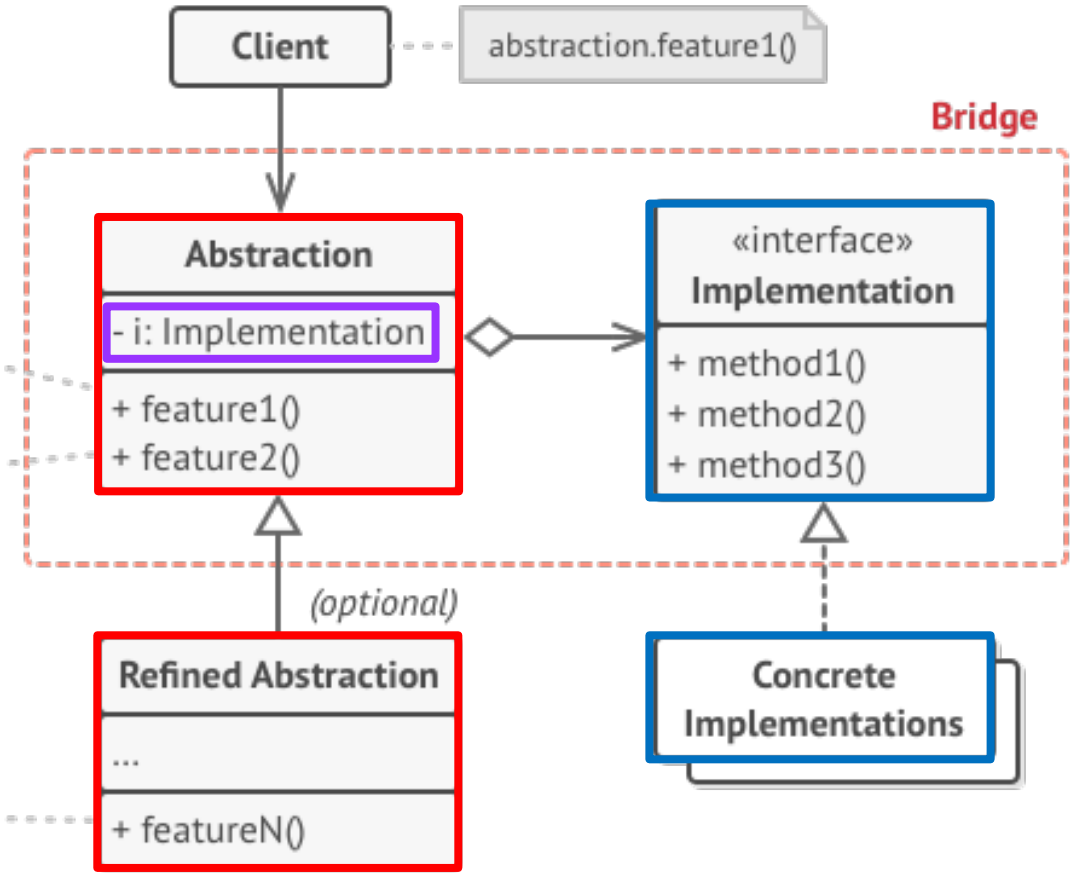
Structural – Bridge

The bridge pattern allows the **Abstraction** and the **Implementation** to be developed independently.

The client code only accesses the Abstraction part without being concerned about the Implementation part

Use this when: you want to **break down a large class into smaller components**. These components are connected to the main abstraction class via a **bridge** (reference to implementation)

Example: Creating a TV **remote** class that controls a **Device**



Structural – Bridge

Example: Creating a TV remote class that controls a Device

Remote is the "**abstraction**" it can control any kind of device

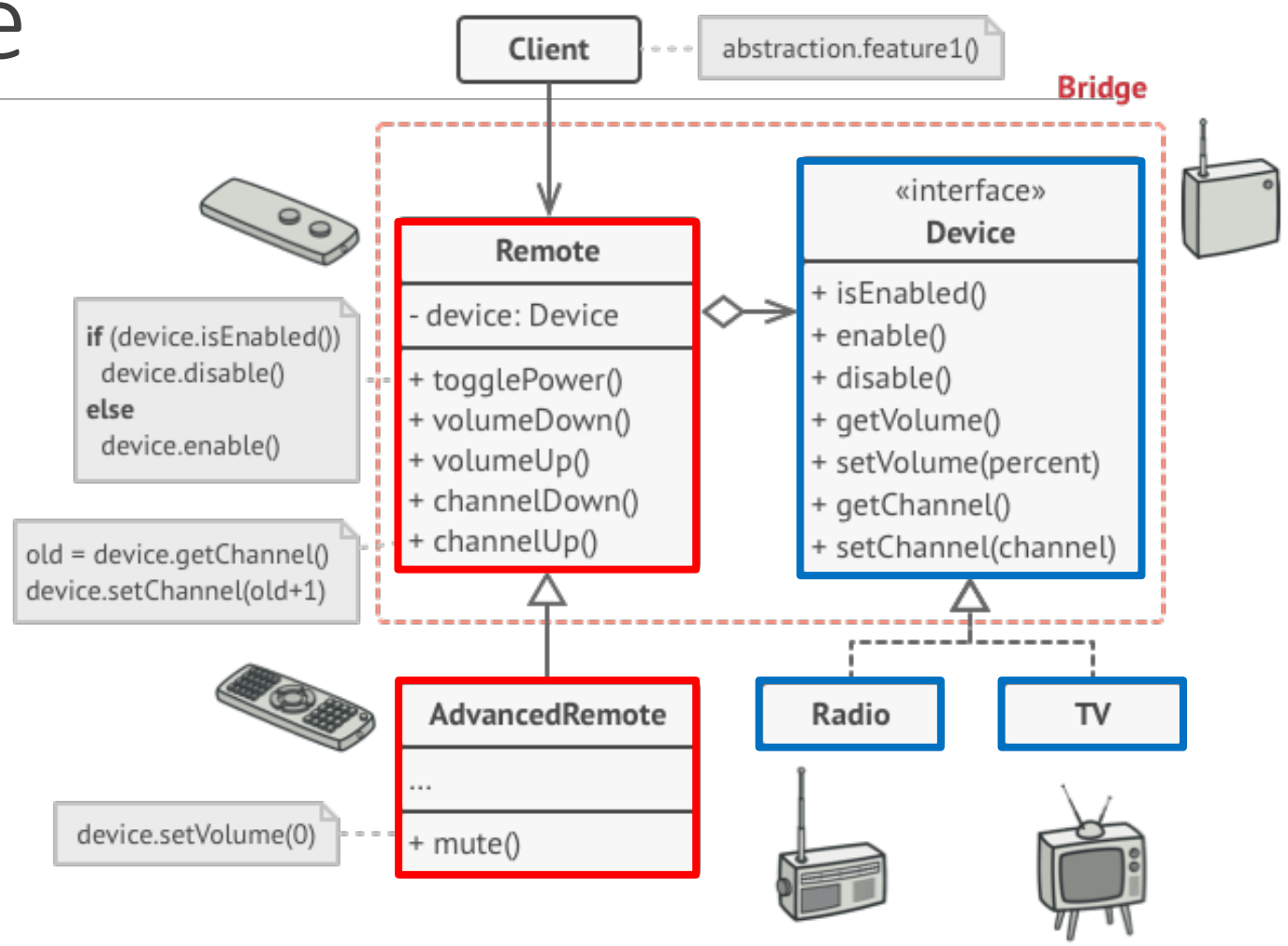
The **device** is the "**implementation**"

This way a remote can control a radio or tv

- Functions in remote simply call functions in device

The advanced remote has the extra functionality of muting any device

device_remote_bridge_example.cpp

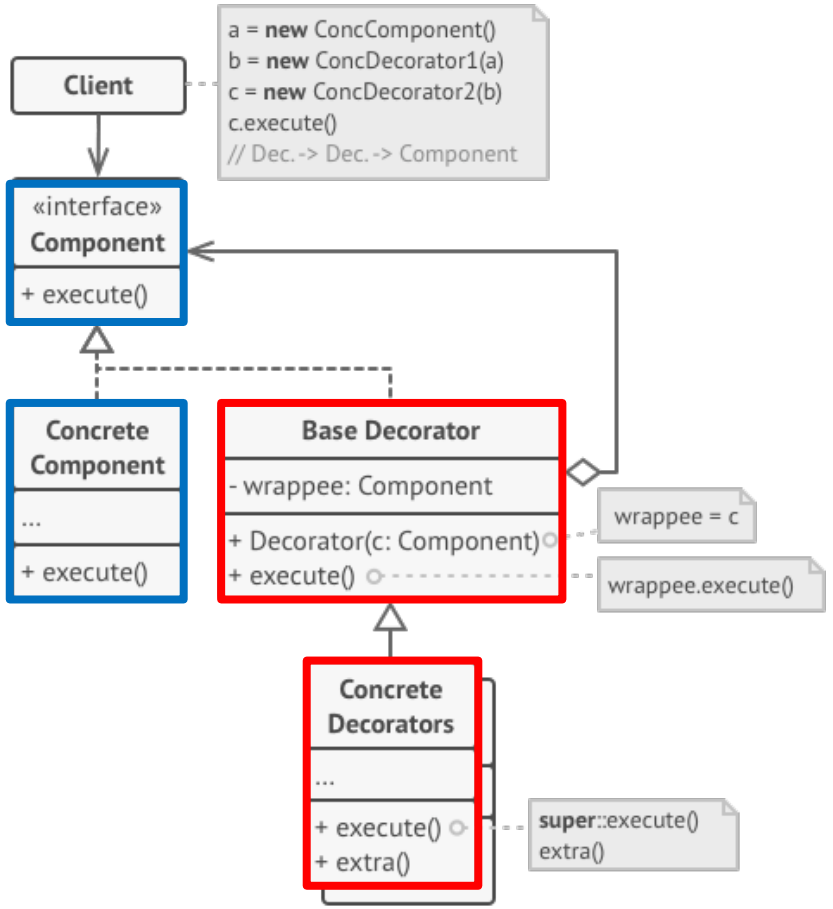


Structural – Decorator

Allows **behavior** to be added to an individual **object**, dynamically, without affecting the behavior of other objects from the same class

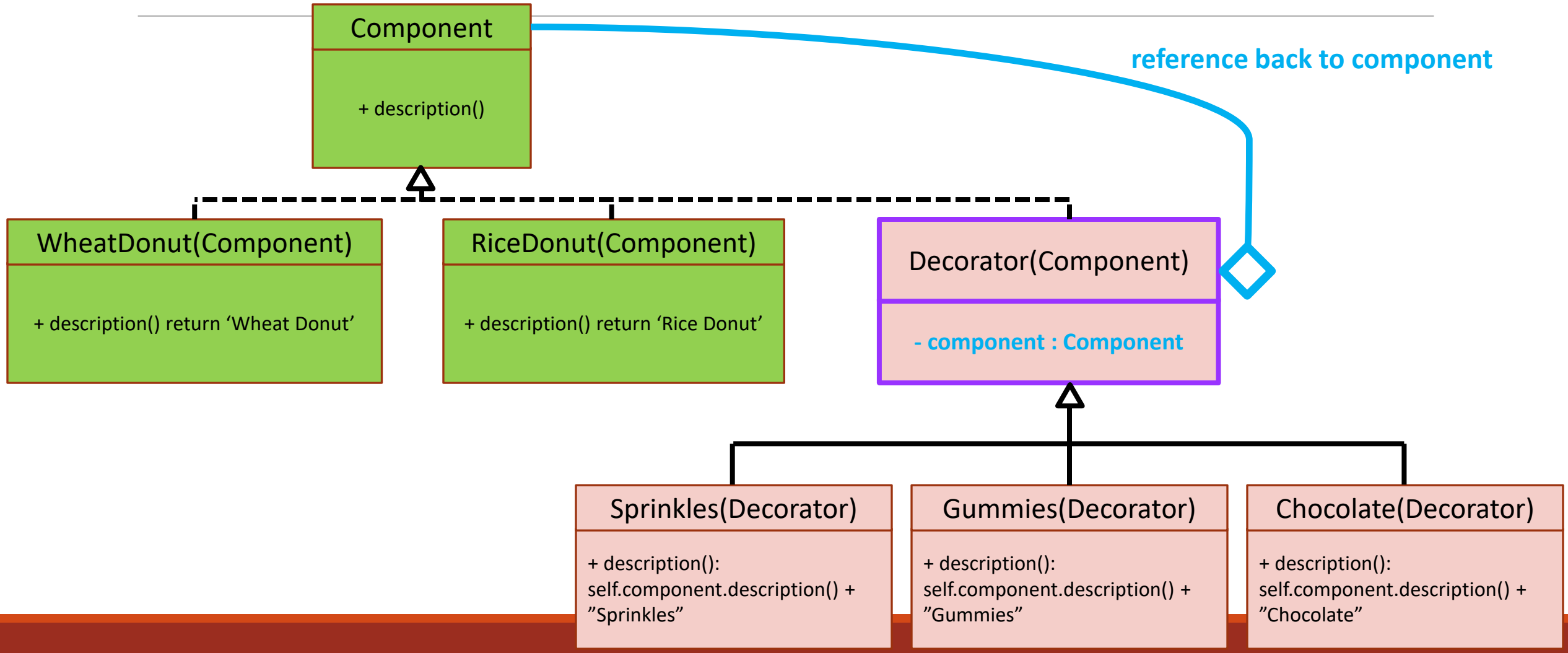
Use this when: you want to **dynamically add behavior to an object** without going back to re-write the object

Example: **Donut object** where we can add any number and type of **toppings** to it



decorator.cpp

Decorator: Donut example



Categorizing Design Patterns

- **Behavioural**

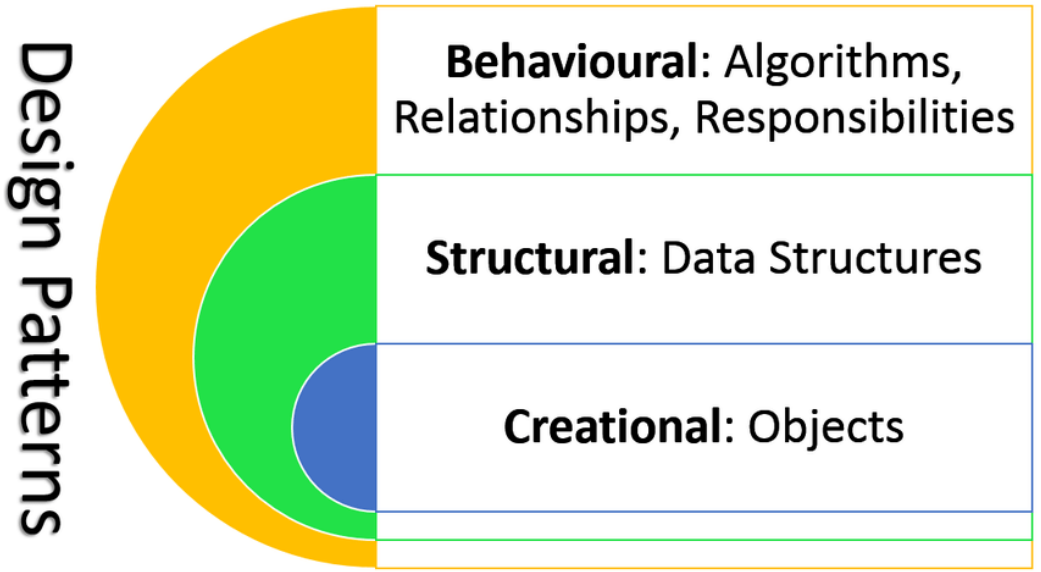
Observer, Chain of Responsibility, State, Strategy, Mediator

- **Structural**

Façade, Proxy, Bridge, Decorator

- **Creational**

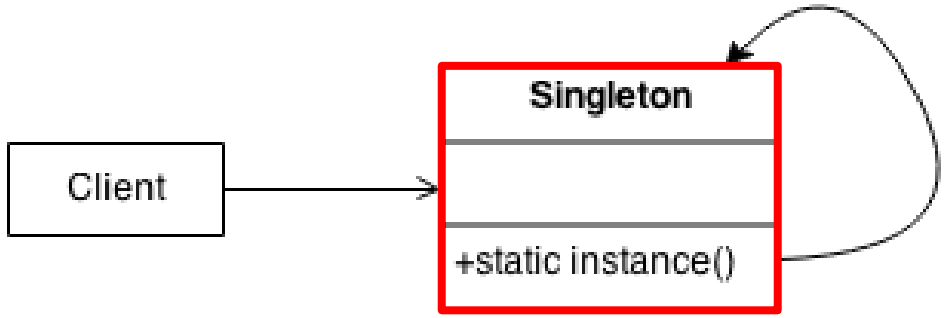
Singleton, Factory, Abstract Factory, Builder, Lazy Initialization



Creational - Singleton

Ensure a class has only one instance, and provide a global point of access to it.

Use this when: an application needs **one and only one instance of an object that is globally accessible**



Example: Save system in a game. The **save system** can be accessed by any screen, controller, object in the game.

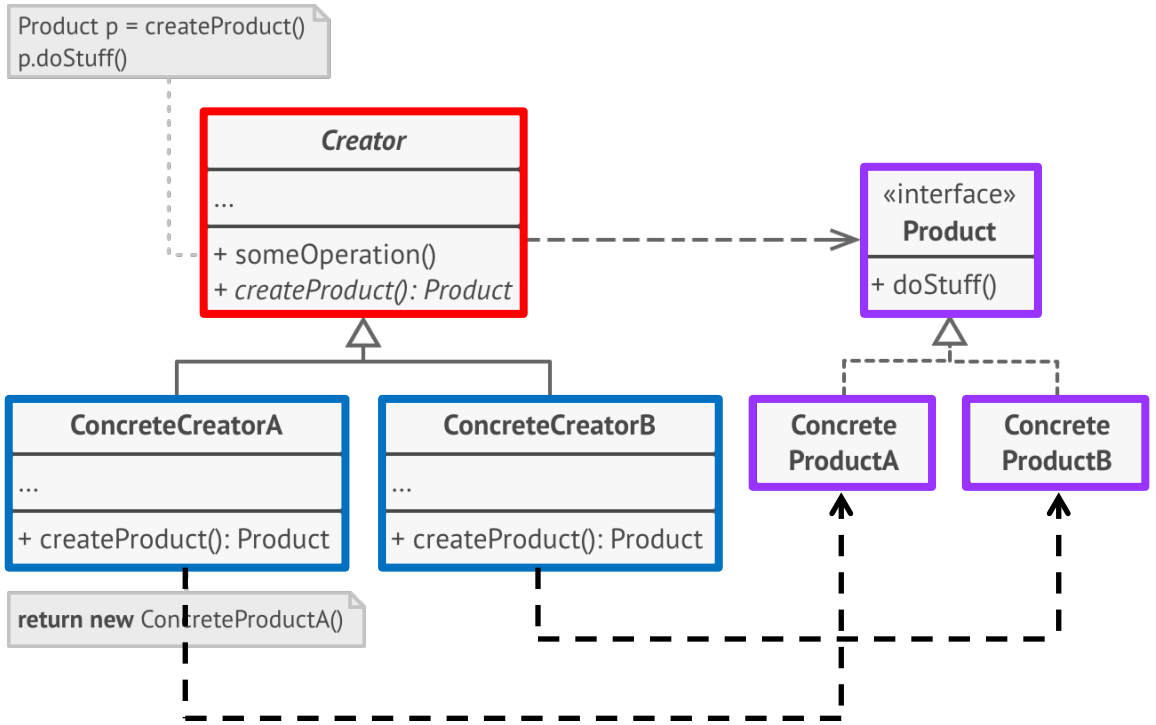
singleton.cpp

Creational - Factory

When you want to separate **creation code** from **client code**.

Use this when: you want a **client or service to only depend on a base class for object creation and not each concrete class.** In other words, the client depends on a base factory for object creation, instead of each individual concrete factory

Example: Forum that needs access to different types of users (Guest, member). **Factory** can produce both types and the forum uses them without knowing the **concrete type**



forum_user_factory.cpp

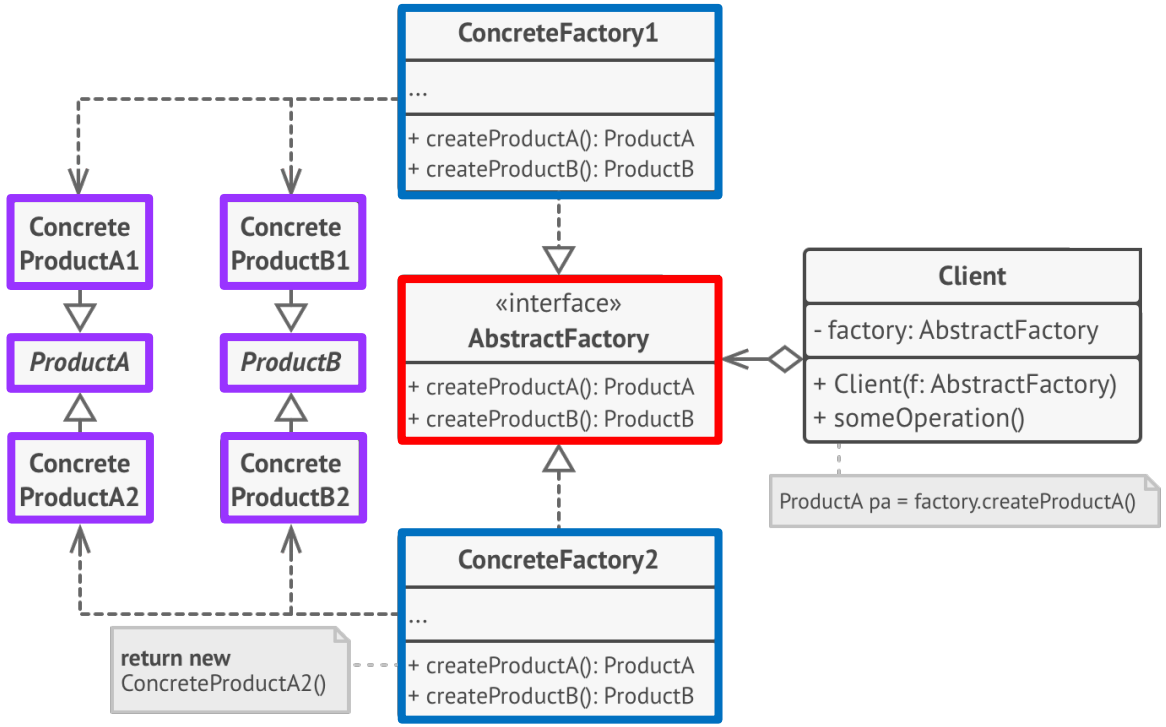
Creational – Abstract Factory

A more evolved form of the Factory Pattern

Allows for the creation of a group of related objects .

Use this when: you want to create **different variants of families of objects.** Client also relies on **base factory** for creation instead of each **individual concrete factory**

Example: Client wants to order different themes of **furniture**. Client gets passed the **appropriate Factory,** which creates a set of **themed furniture**.



game_abstract_factory.cpp

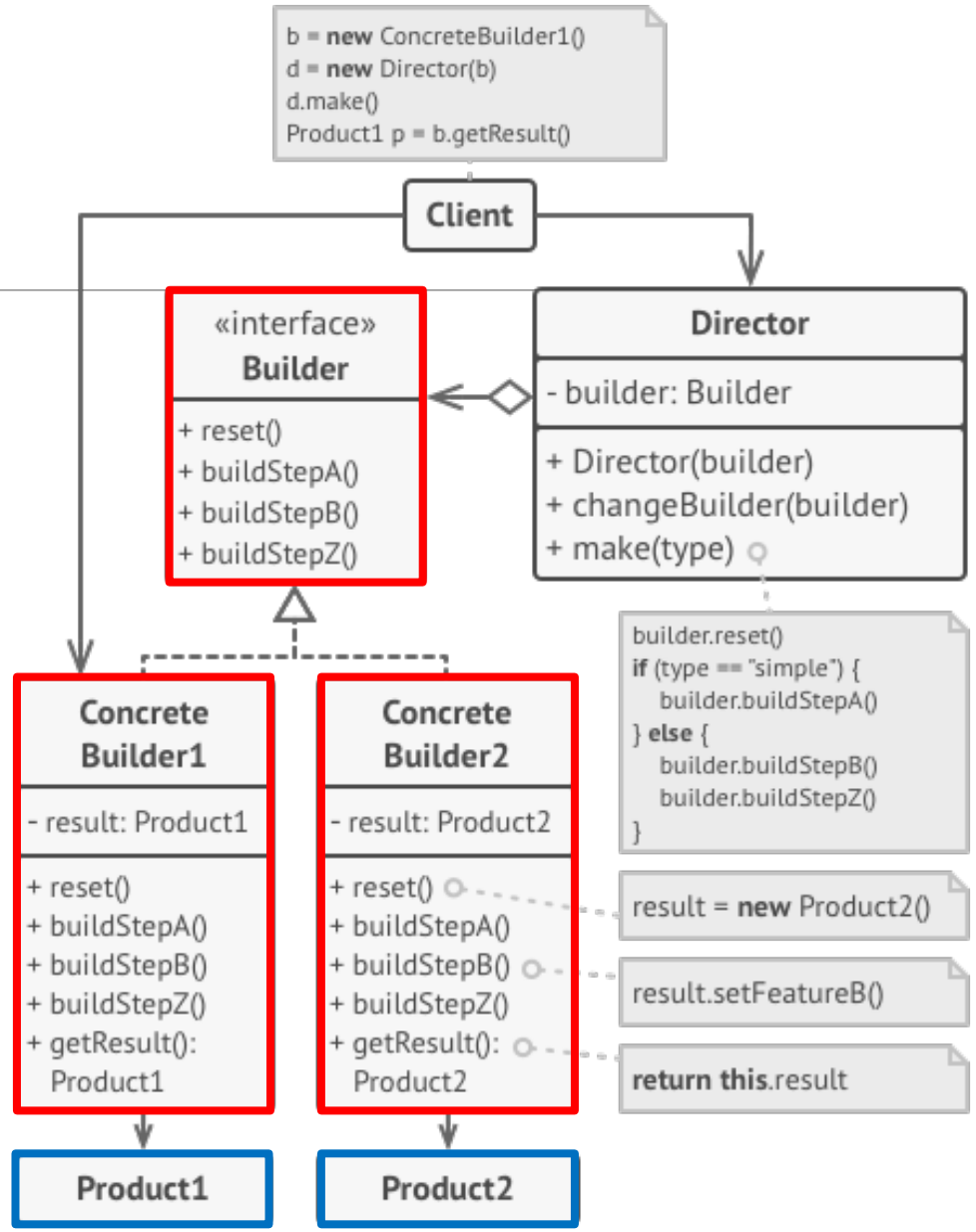
Creational – Builder

Designed to provide a flexible solution to various object creation problems in object-oriented programming.

Use this when: you want to **separate the construction** of a complex object from its **representation**. The code typically written in the **constructor** of an object is **moved into a separate class**

Example: Want to create different types of pizza, but the client doesn't want to manually specify all ingredients. They want to order from a menu. The menu tells the **builder** to build a very specific type of **pizza**, and returns that to the client

builder_sample_code.cpp



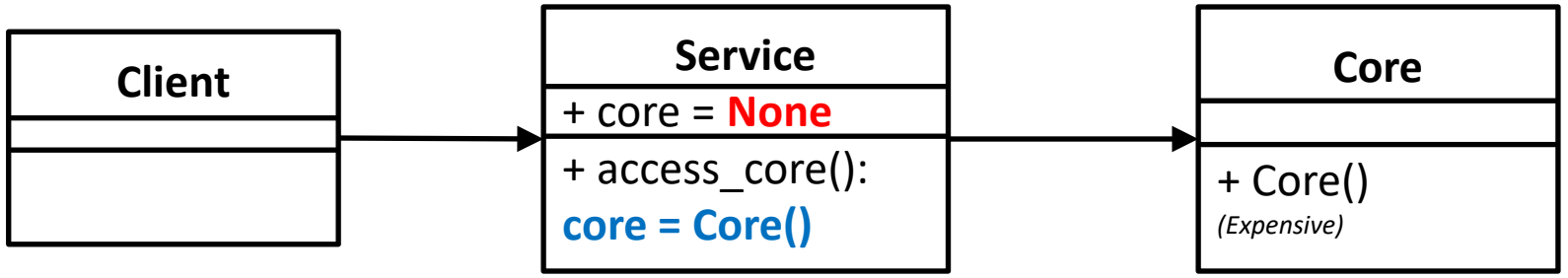
Creational – Lazy initialization

Initialization of an object occurs only when it is actually needed and not before to preserve simplicity of usage and improve performance.

Use this when: you want to **delay using a resource until the last possible moment**. Saving time and resources if the resource is never used

Example: Screen in a program that isn't instantiated until it's actually needed.

Lazy initialization often used in the Singleton pattern



lazy_initialization_sample_code.cpp

What I've used in practice

"What is the most commonly used design pattern that you see at work or wish students would utilize more.."

1. Singleton – My favorite for managers/controllers in games. Whenever I need **one central object to control a large overarching task** in the system. SoundManager, Save/Load manager, ScreenManager
2. Observer – Great for having an **object communicate with many objects** when something important happens
3. Lazy initialization – Used as part of singleton and whenever I want to **save performance** and **not initializing everything at once**
4. Creation method – Have a static class who's **only job is to create specific objects**
5. Bridge – Probably used it unknowingly when **creating classes through composition**

Design patterns scenario activity

Scenario

You are writing code to model different kinds of offices. Offices can have multiple conference rooms, bathrooms, cubicles, and executive offices

There may even be duplex offices (offices with a second storey). You need tocome up with a way to instantiate all these different kinds of Offices while still allowing for customizability

What design pattern would be most appropriate to address this issue?

There's a **lot of design patterns**, it's **hard to choose**. Let's break it down into several steps

Design patterns scenario activity

Scenario

*You are writing code to **model different kinds of offices**. Offices can have **multiple conference rooms**, **bathrooms**, **cubicles**, and **executive offices***

*There may even be **duplex offices** (offices with a second storey). You need to come up with a way to **instantiate** all these different kinds of Offices while still allowing for **customizability***

Step 1 – Understand the problem

Appears to be a problem related with **creating** something with **multiple optional components** in a **customizable** way

Design patterns scenario activity

Scenario

You are writing code to model different kinds of offices. Offices can have multiple conference rooms, bathrooms, cubicles, and executive offices

There may even be duplex offices (offices with a second storey). You need tocome up with a way to instantiate all these different kinds of Offices while still allowing for customizability

Step 2 – What broad category of design patterns does this fall under?

1. Behavioral – how objects in system communicate with low coupling (Communication)
2. **Structural – how objects/classes combine to create new structures (Structure/Form)**
3. **Creational – how to instantiate objects or groups of objects in new ways (Instantiation)**

Design patterns scenario activity

Scenario

You are writing code to model different kinds of offices. Offices can have multiple conference rooms, bathrooms, cubicles, and executive offices

There may even be duplex offices (offices with a second storey). You need tocome up with a way to instantiate all these different kinds of Offices while still allowing for customizability

Step 3 – Examine each pattern to see which most closely matches our scenario

- **Structural**

Decorator, Façade, Proxy, Bridge

- **Creational**

Singleton, Factory, Abstract Factory, Builder, Lazy Initialization

Design patterns scenario activity

Scenario

You are writing code to model different kinds of offices. Offices can have multiple conference rooms, bathrooms, cubicles, and executive offices

There may even be duplex offices (offices with a second storey). You need tocome up with a way to instantiate all these different kinds of Offices while still allowing for customizability

Step 3 – Examine each pattern to see which most closely matches our scenario

- **Structural**

Decorator, Façade, Proxy, Bridge

All of these appear to be wrappers, which isn't exactly what we want.

Decorator is close because it adds **behaviors** to a central object. But we don't want to add **behaviors**. We want to add optional **properties** to an existing object.

Design patterns scenario activity

Scenario

You are writing code to model different kinds of offices. Offices can have multiple conference rooms, bathrooms, cubicles, and executive offices

There may even be duplex offices (offices with a second storey). You need to come up with a way to instantiate all these different kinds of Offices while still allowing for customizability

Step 3 – Examine each pattern to see which most closely matches our scenario

- Creational

Pattern	Description
Singleton	instantiate global object
Factory	instantiate different types of objects
Abstract Factory	instantiate families of objects
Builder	instantiate customizable objects
Lazy Initialization	instantiate object when needed

Design patterns scenario activity

Scenario

You are writing code to model different kinds of offices. Offices can have multiple conference rooms, bathrooms, cubicles, and executive offices

There may even be duplex offices (offices with a second storey). You need to come up with a way to instantiate all these different kinds of Offices while still allowing for customizability

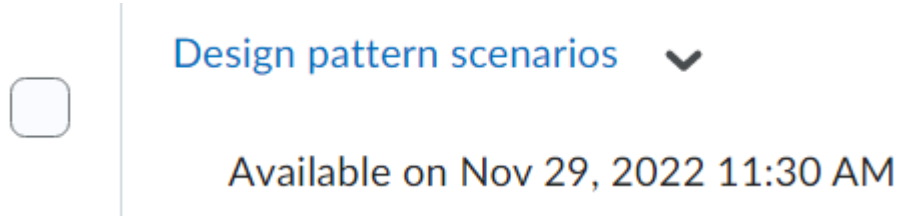
Step 3 – Examine each pattern to see which most closely matches our scenario

- **Creational**

Pattern	Description
Singleton	instantiate global object
Factory	instantiate different types of objects
Abstract Factory	instantiate families of objects
Builder	**instantiate customizable objects**
Lazy Initialization	instantiate object when needed

Design patterns scenario activity

Learning Hub > Activities > Surveys



1. Purpose is to choose the design pattern that will best solve the scenario that's mentioned
2. Work in pairs to fill out the survey
3. We'll discuss afterwards
4. Design patterns cheat sheet