Structural Patterns:
Proxy, Facade, Bridge, Decorator

COMP3522 OBJECT ORIENTED PROGRAMMING 2

WEEK 9

Categorizing Design Patterns

❑ **Behavioural**

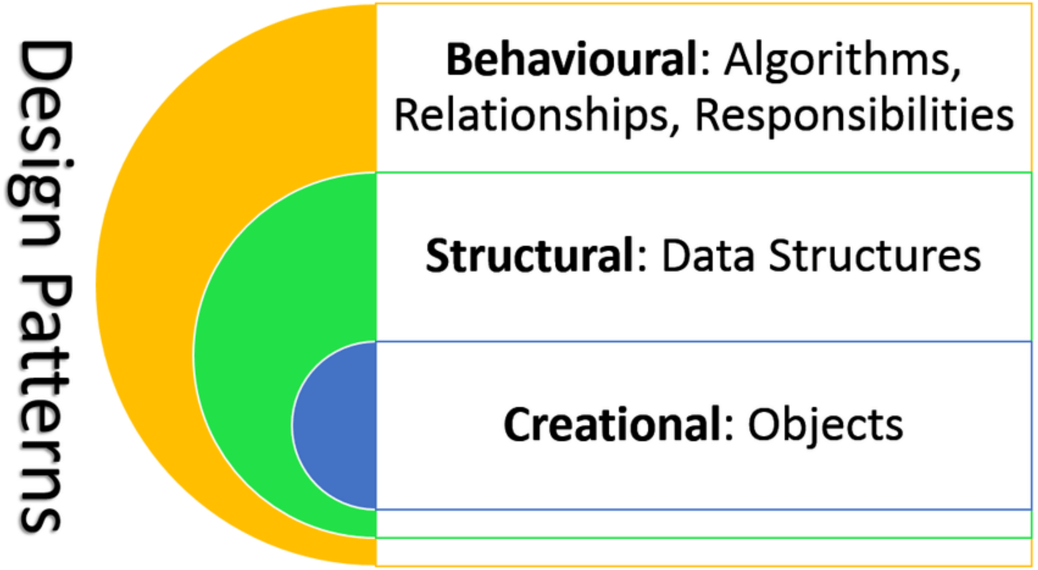
Focused on communication and interaction between objects. How do we get objects talking to each other while minimizing coupling?

❑ **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
Behavioural: Algorithms, Relationships, Responsibilities
Structural: Data Structures
Creational: Objects

Today we will be looking at 4 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.

Decorator

- When you want to dynamically add functionality to an object at runtime

Proxy

What is a wrapper?

A class that **‘wraps’ around another class**.

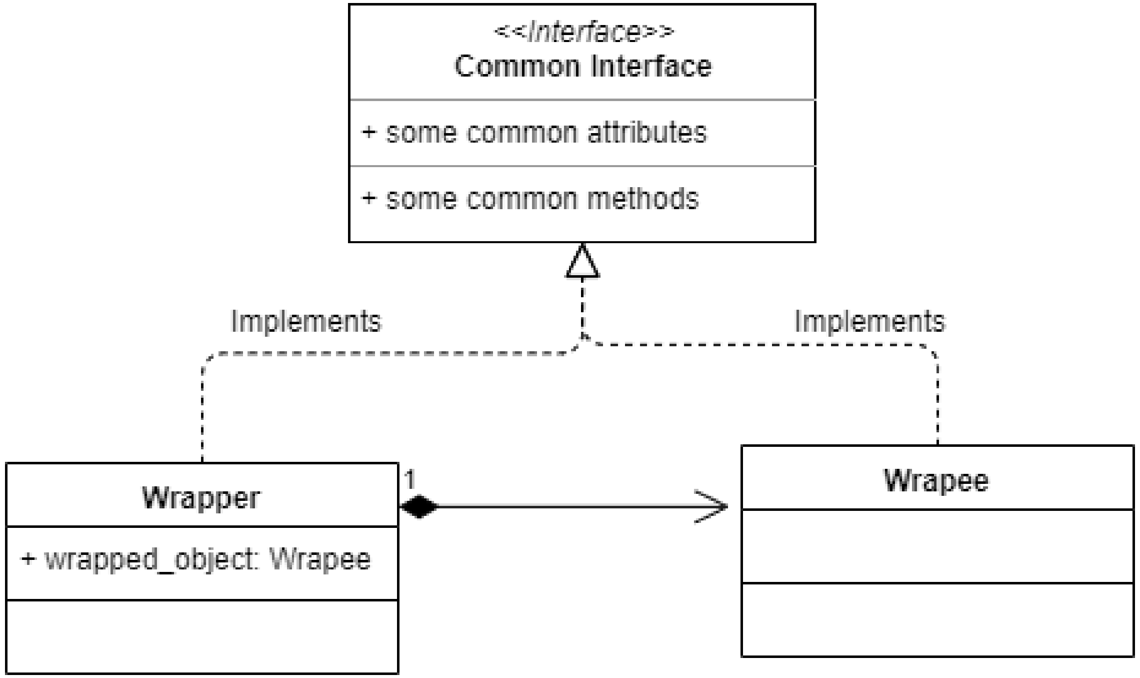
The wrapper has the same interface as the ‘wrapee’.

<u>Remember, this is interface in the OOP sense</u>. The wrapper has the same public method and attributes as the wrapee. We can implement this using an ABC or by leaving it informal (duck typing).

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

What is a wrapper?

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



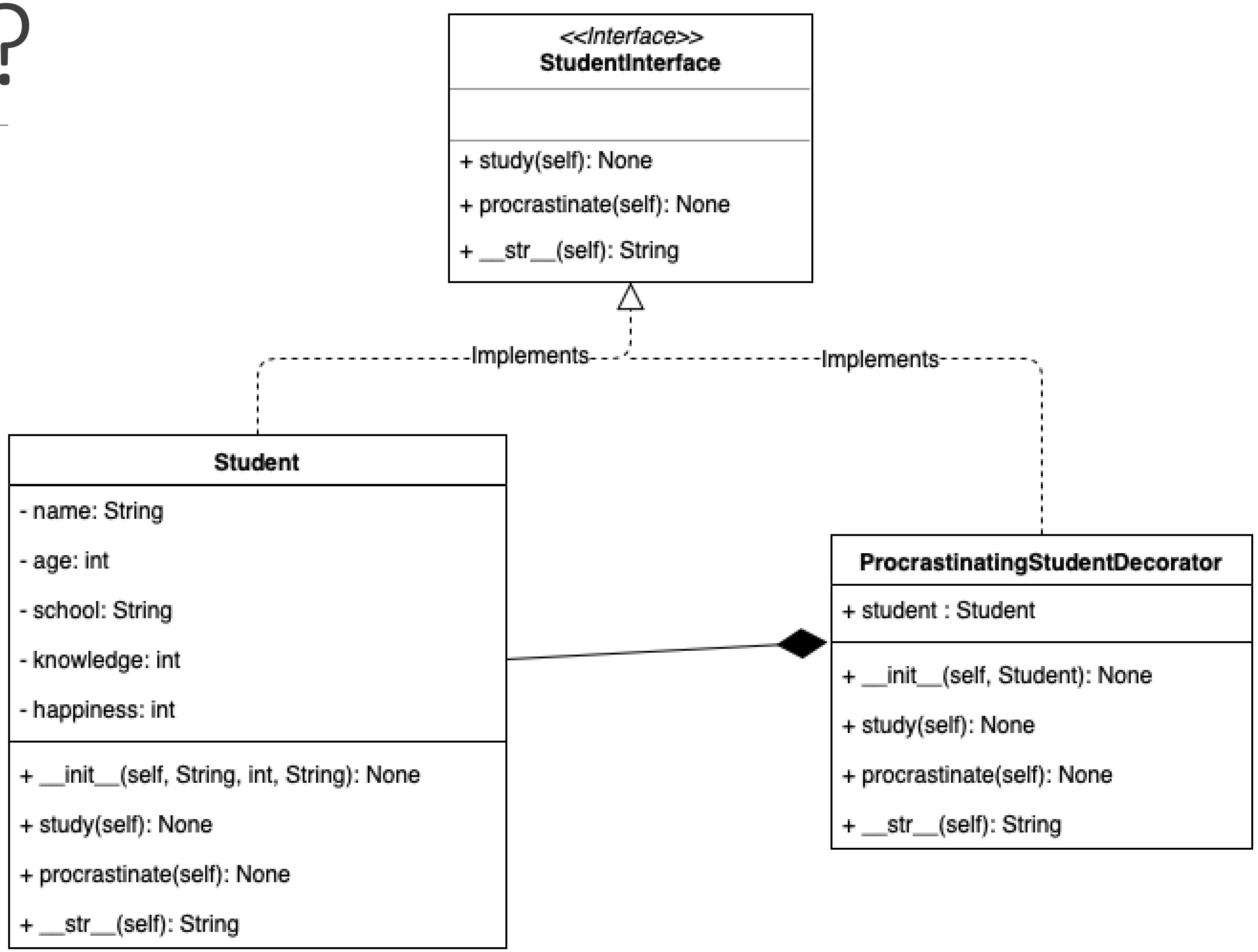
What is a wrapper?

Student is the wrapee

Procrastinating student 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



<<Interface>>
StudentInterface

+ study(self): None
+ procrastinate(self): None
+ __str__(self): String

Student

- name: String
- age: int
- school: String
- knowledge: int
- happiness: int

+ __init__(self, String, int, String): None
+ study(self): None
+ procrastinate(self): None
+ __str__(self): String

ProcrastinatingStudentDecorator

+ student : Student

+ __init__(self, Student): None
+ study(self): None
+ procrastinate(self): None
+ __str__(self): String

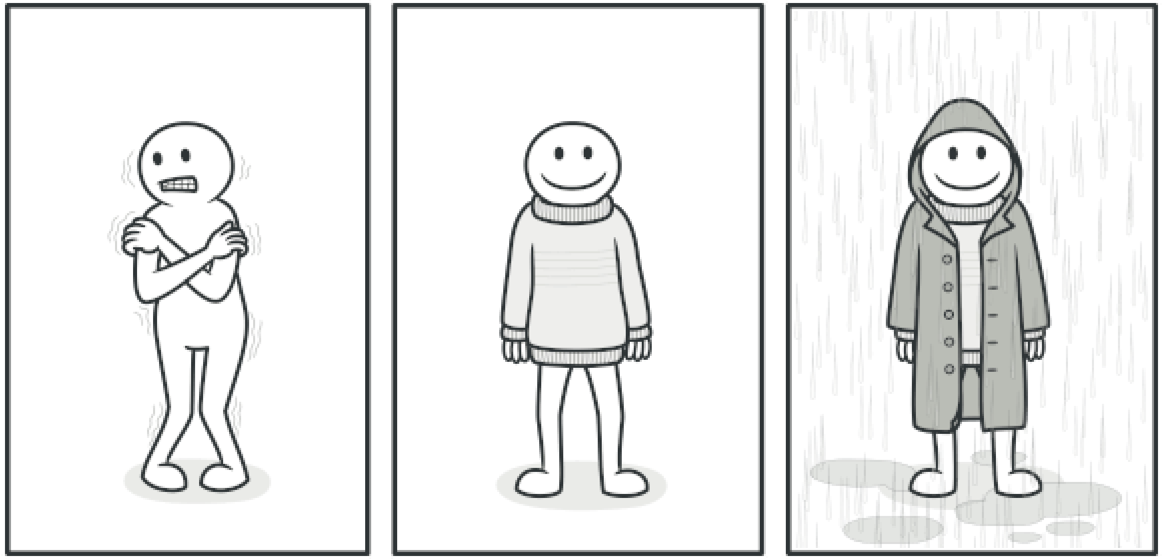
Wrapper.py

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. This let’s us combine the effect of each wrapper.



Proxy

A Proxy is a wrapper.

It controls access to the wrapee.

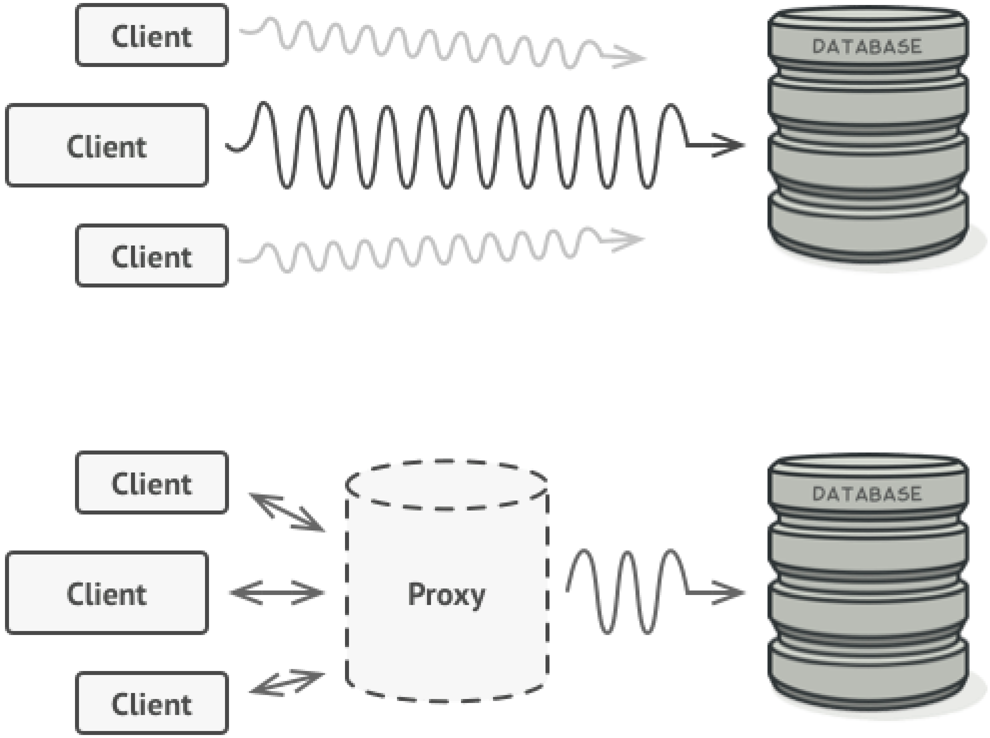
For example:

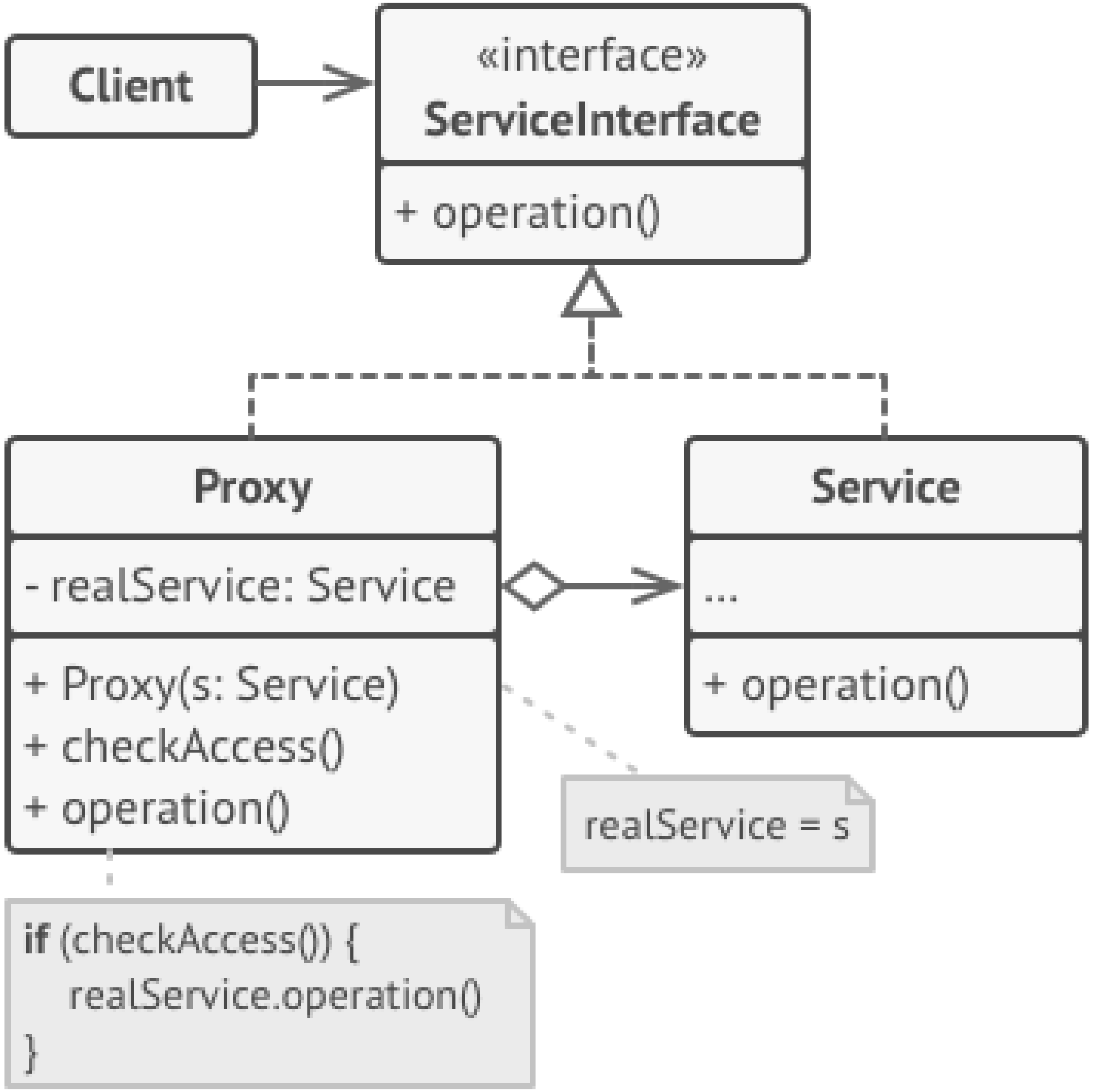
Database queries can be slow.

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.

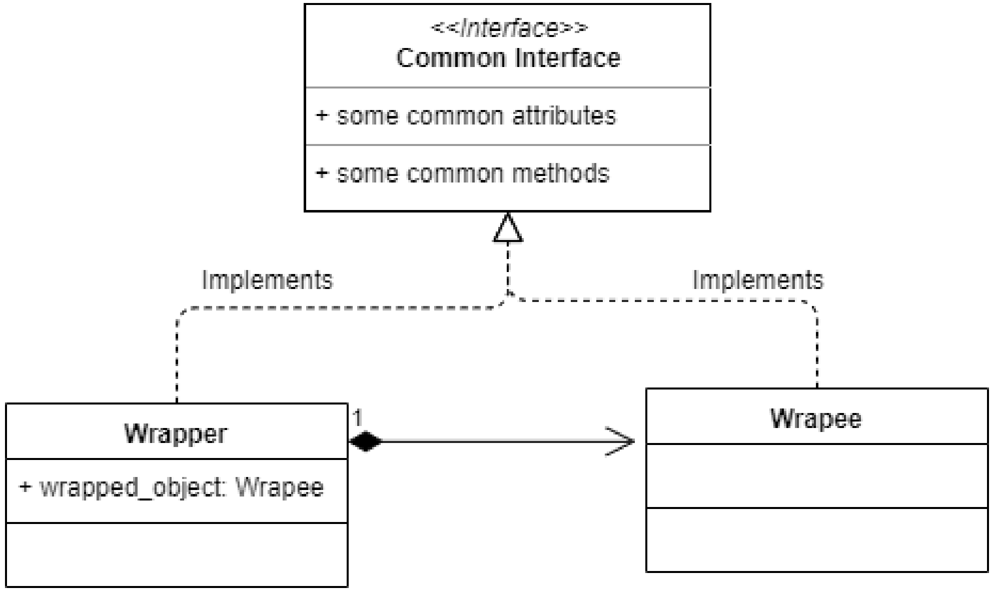
It can handle authorization, result caching and other optimizations. The client doesn’t even need to know.

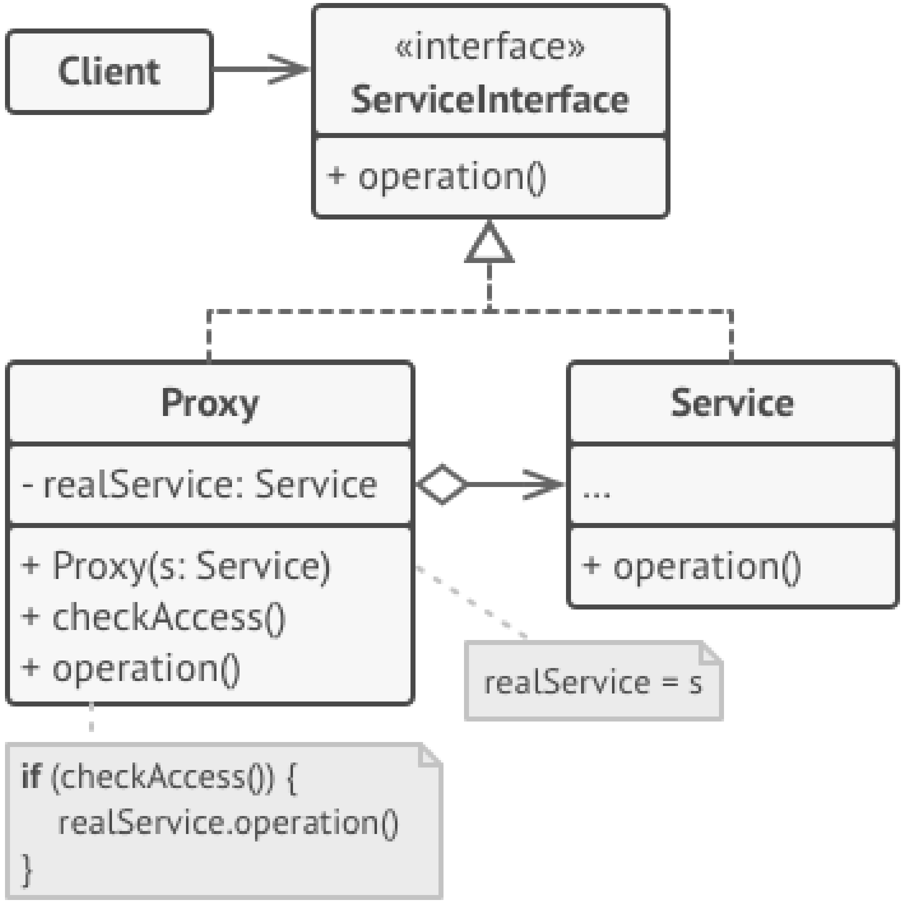




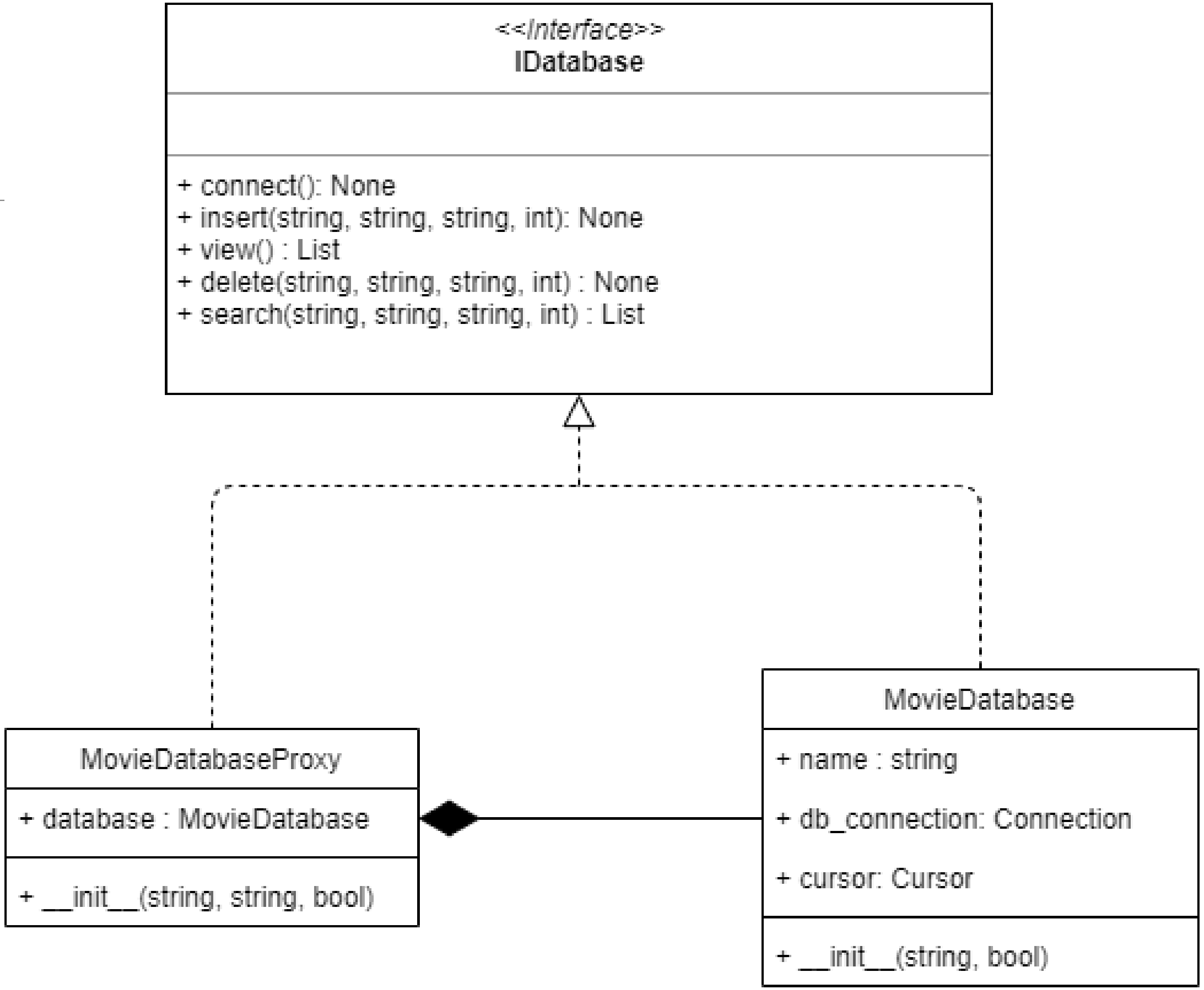
The Proxy Pattern

This is the exact same structure as a wrapper.





Let’s build a proxy



<<Interface>>
IDatabase

+ connect(): None
+ insert(string, string, string, int): None
+ view() : List
+ delete(string, string, string, int) : None
+ search(string, string, string, int) : List

MovieDatabaseProxy

+ database : MovieDatabase

+ __init__(string, string, bool)

MovieDatabase

+ name : string

+ db_connection: Connection

+ cursor: Cursor

+ __init__(string, bool)

simple_proxy.py
movie_database_proxy_example.py

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.
- Response from the service might be delayed if the proxy is carrying out extensive work before or after invoking the service.

Facade

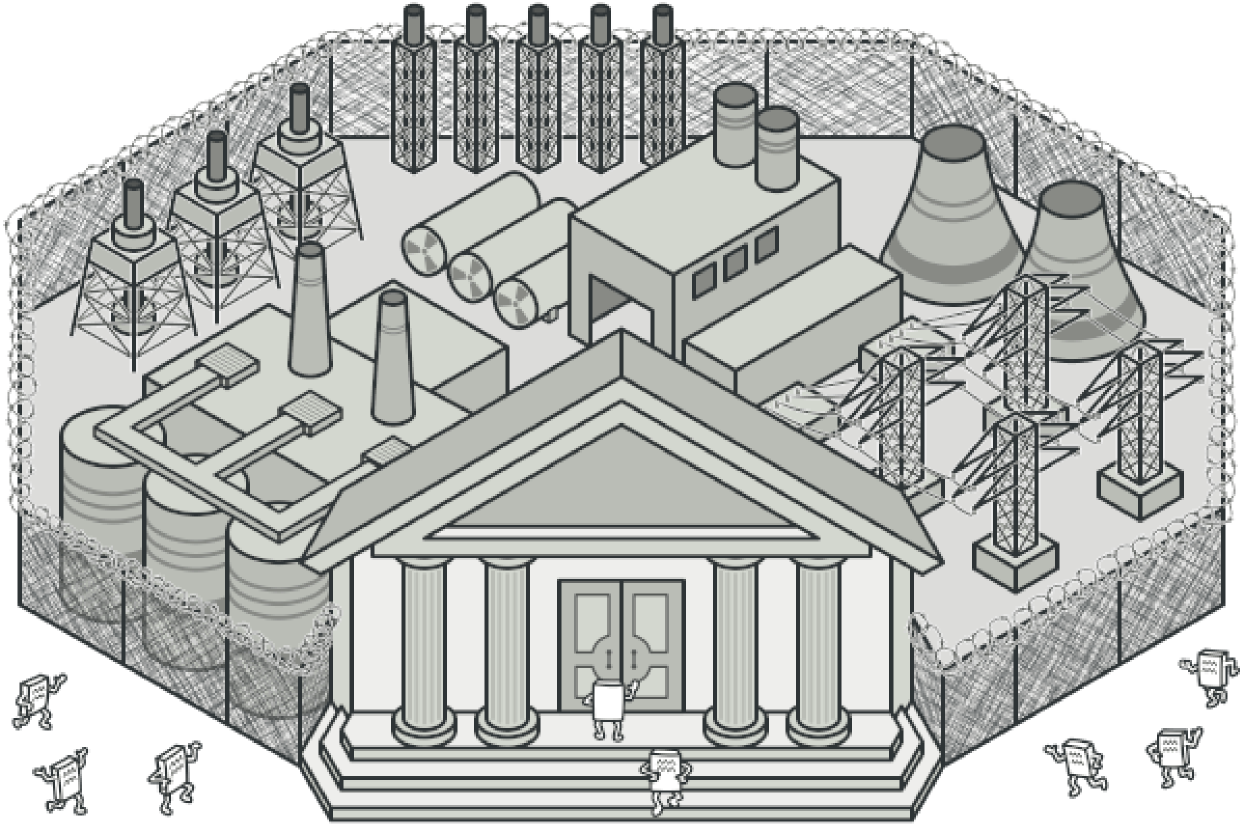
WHEN YOU WANT TO HIDE THE MESS RIGHT BEFORE YOUR FRIENDS COME OVER FOR BOARD GAMES…

Facade

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

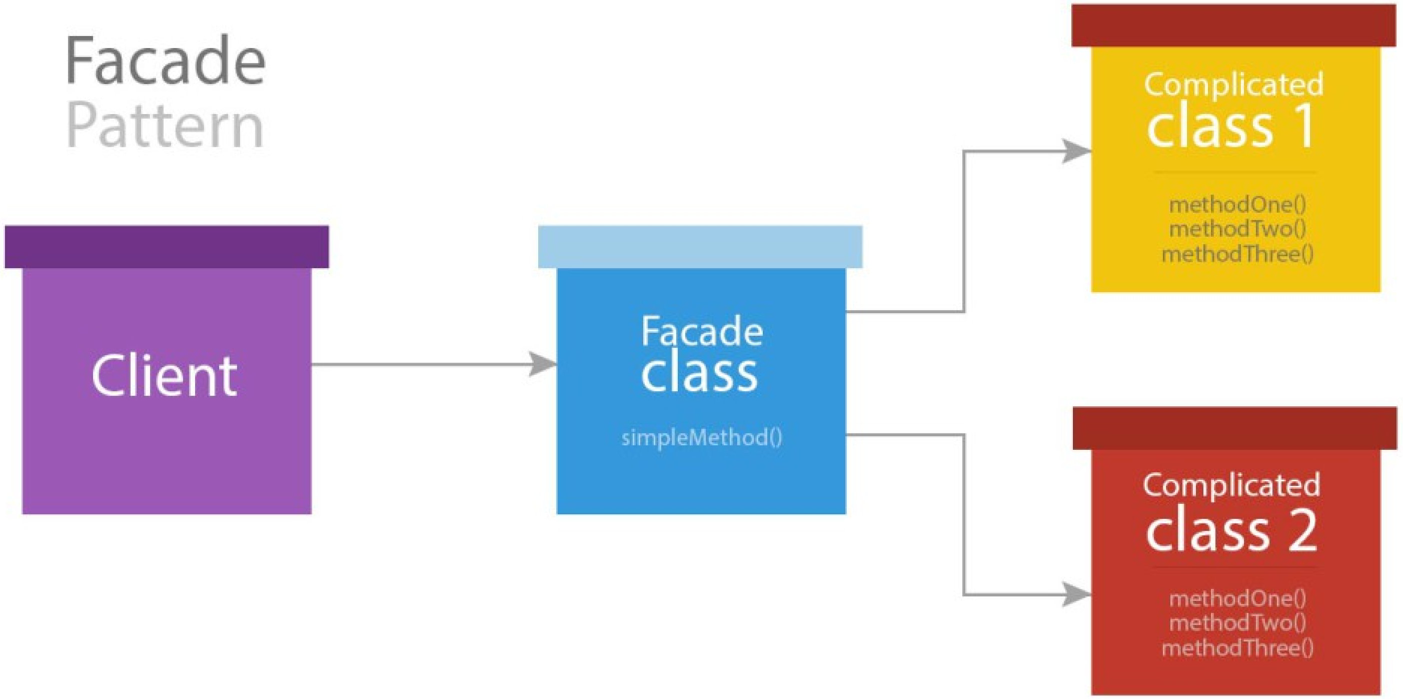
Used when we need to provide a simple interface to a complex system

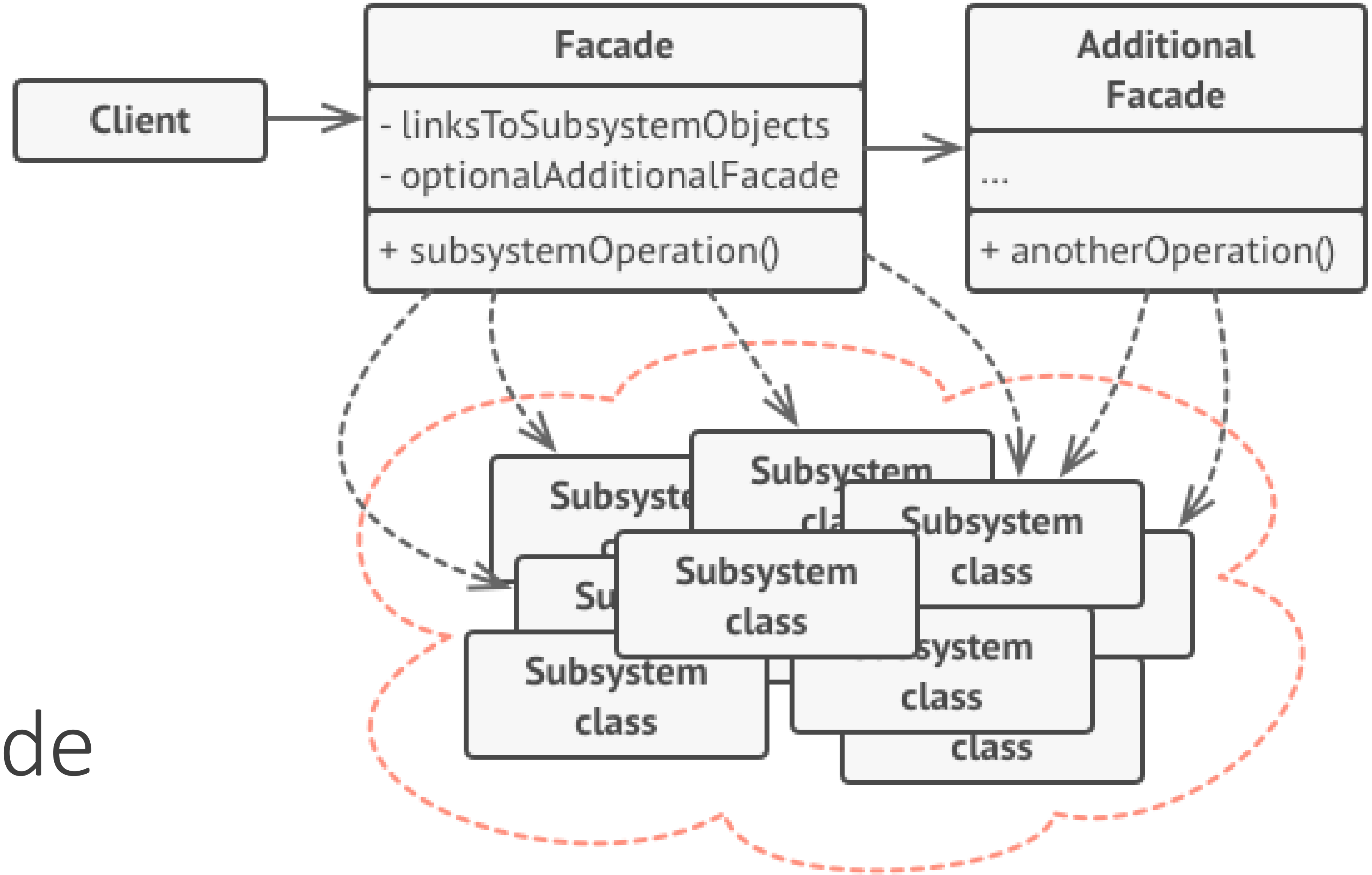
Abstracts away the details of the complex system or API



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 packages to only use a subset of those features and to solve a specific need.
- Façade encapsulate these complex system and hides their complex behaviour.
- The rest of our code only needs to be aware of the simple interface provided by the Façade.





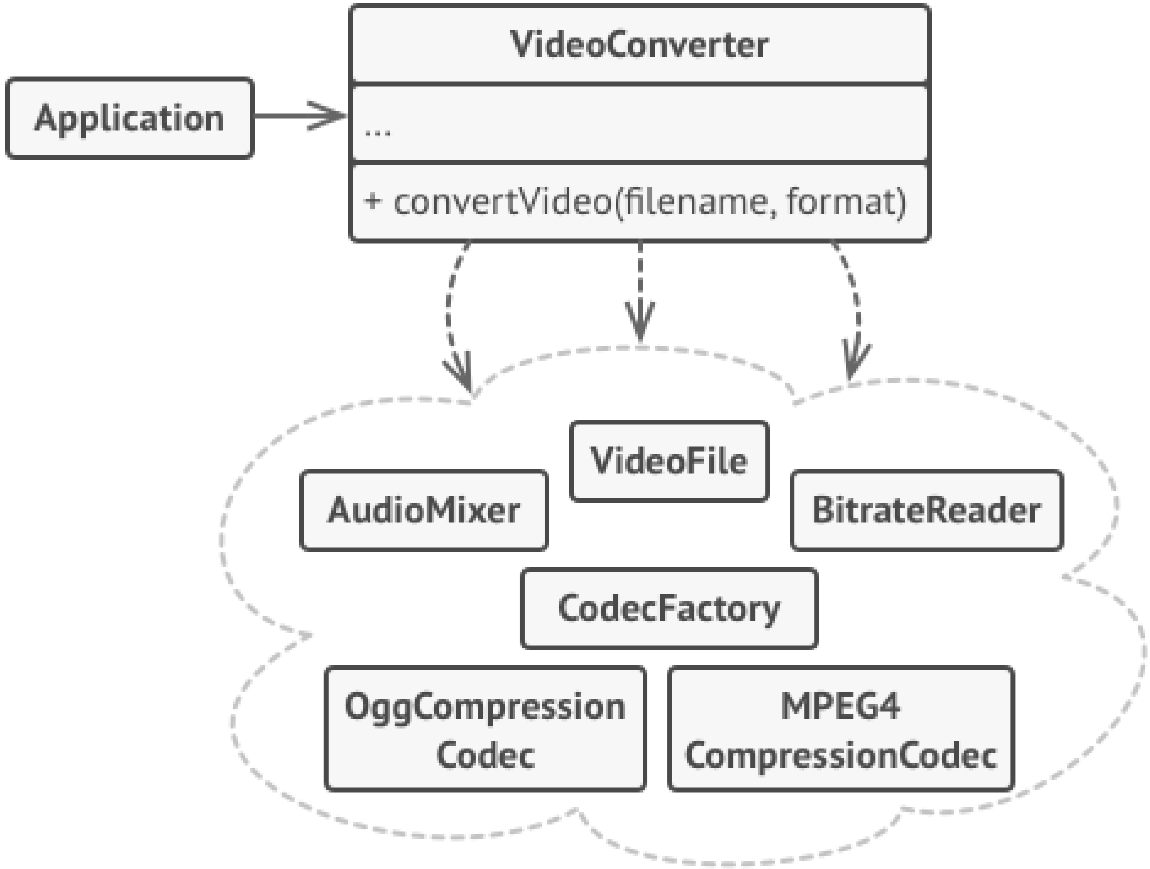
The Facade Pattern

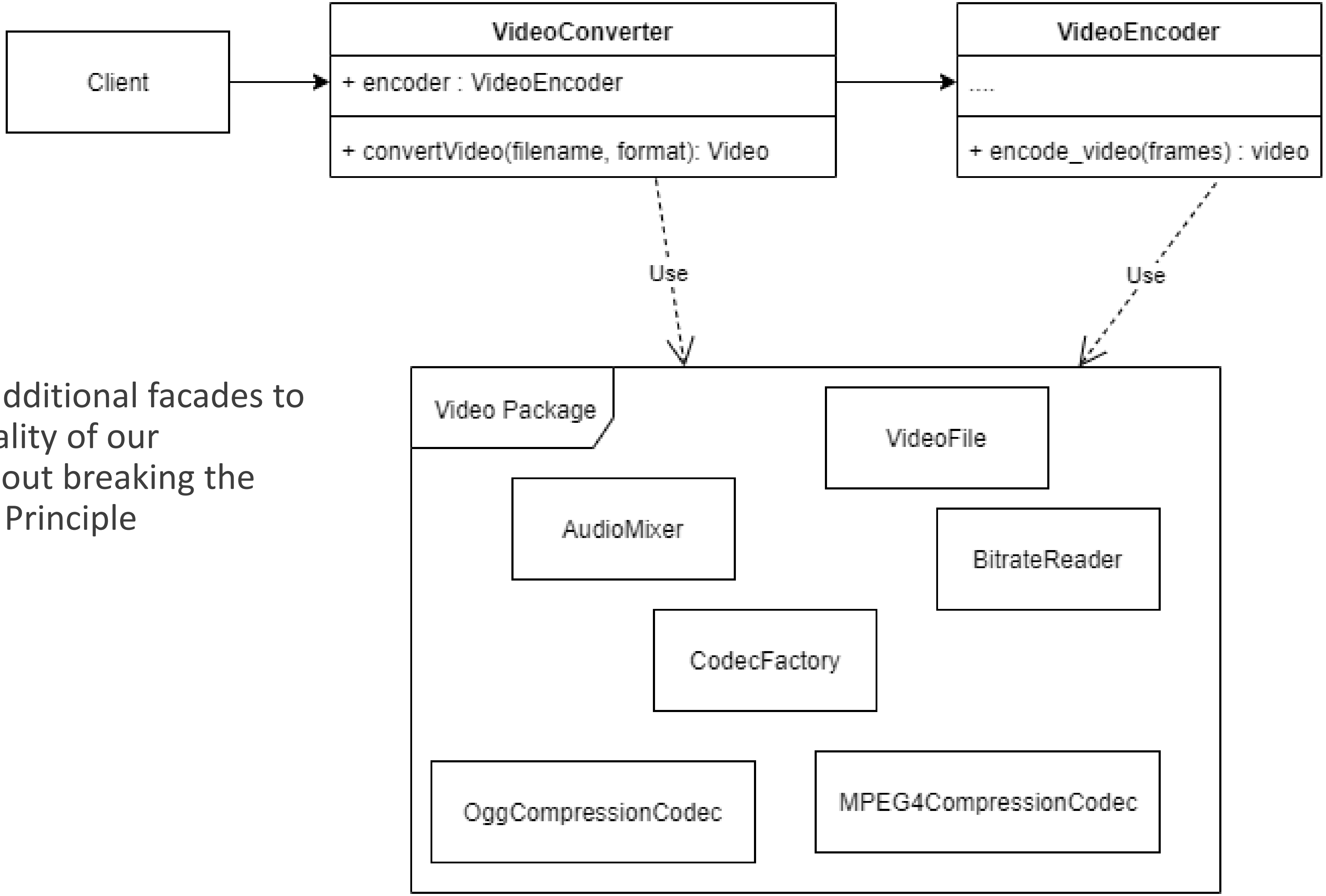
Facade Example

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.

simple_facade_py
video_package_facade_example.py





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.

Let’s look at a problem.

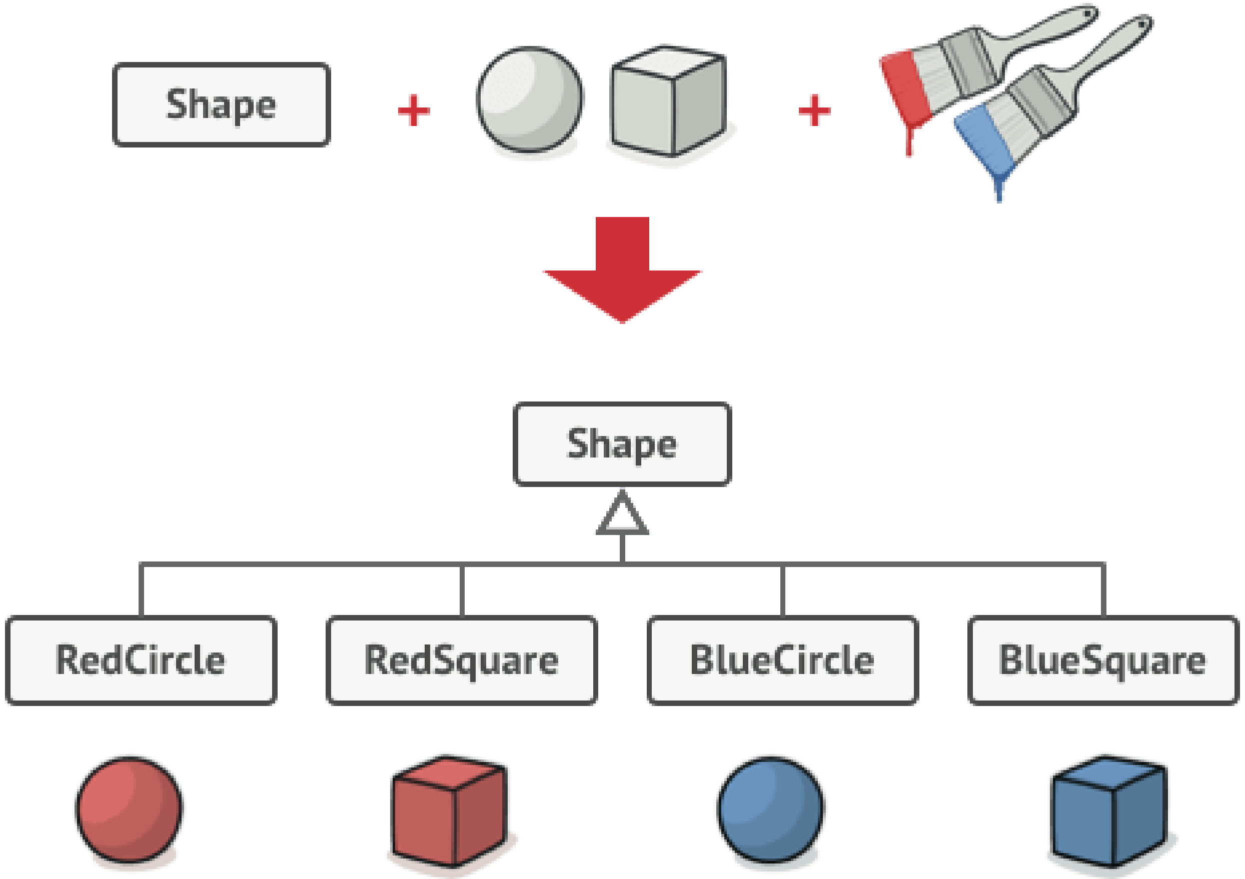
Bridge – A simple example

Say we have a Shape class which can further be specialized into Circle and Square.

We want to also change the colors of these shapes to be either Red or Blue as well. So now we have 4 classes.

How many classes will we have if we add an extra shape? Say a Triangle.

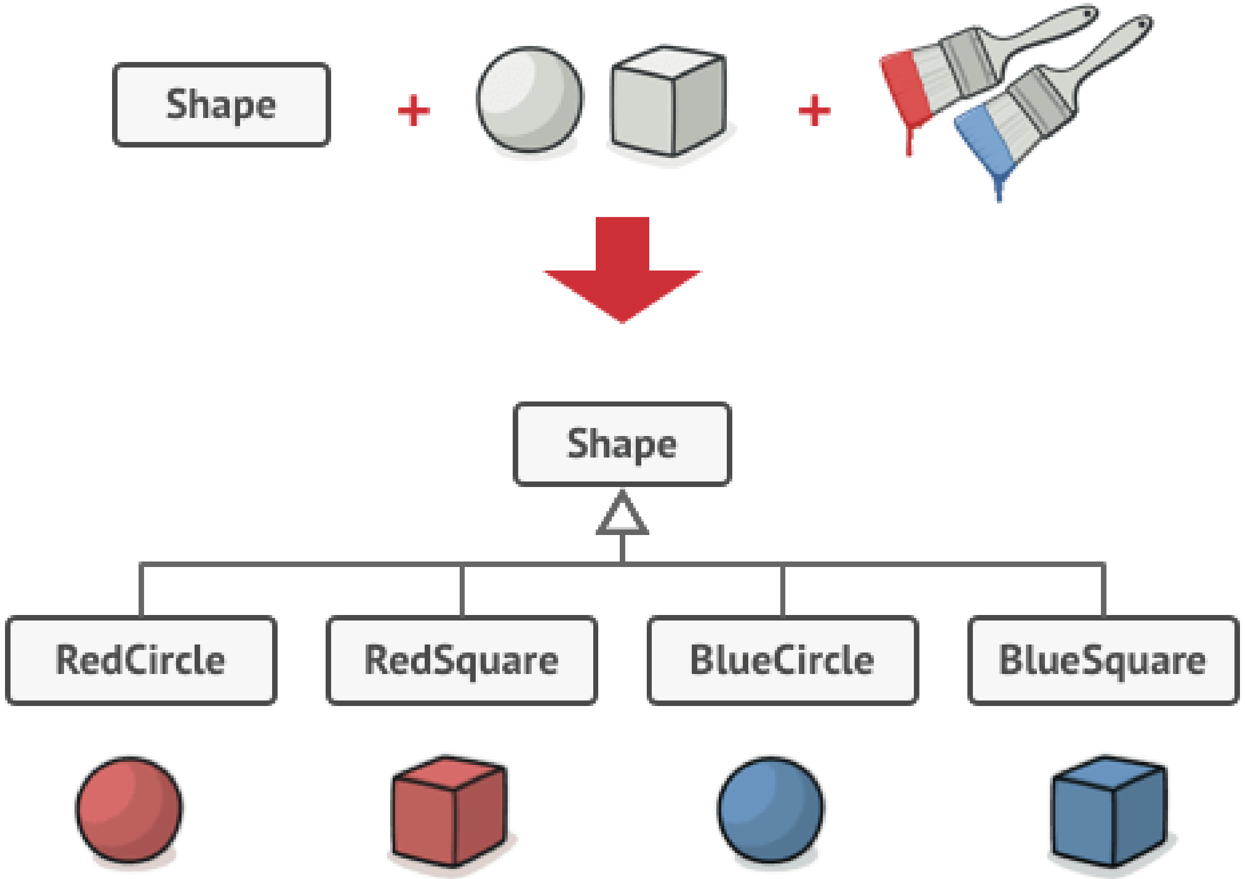
Right, now what if we added a third color, say Green. How many classes would we have?



Bridge – A simple example

Adding a new shape or color will cause the hierarchy to grow exponentially.

This is going to be way too many classes to maintain.



Bridge – A simple example

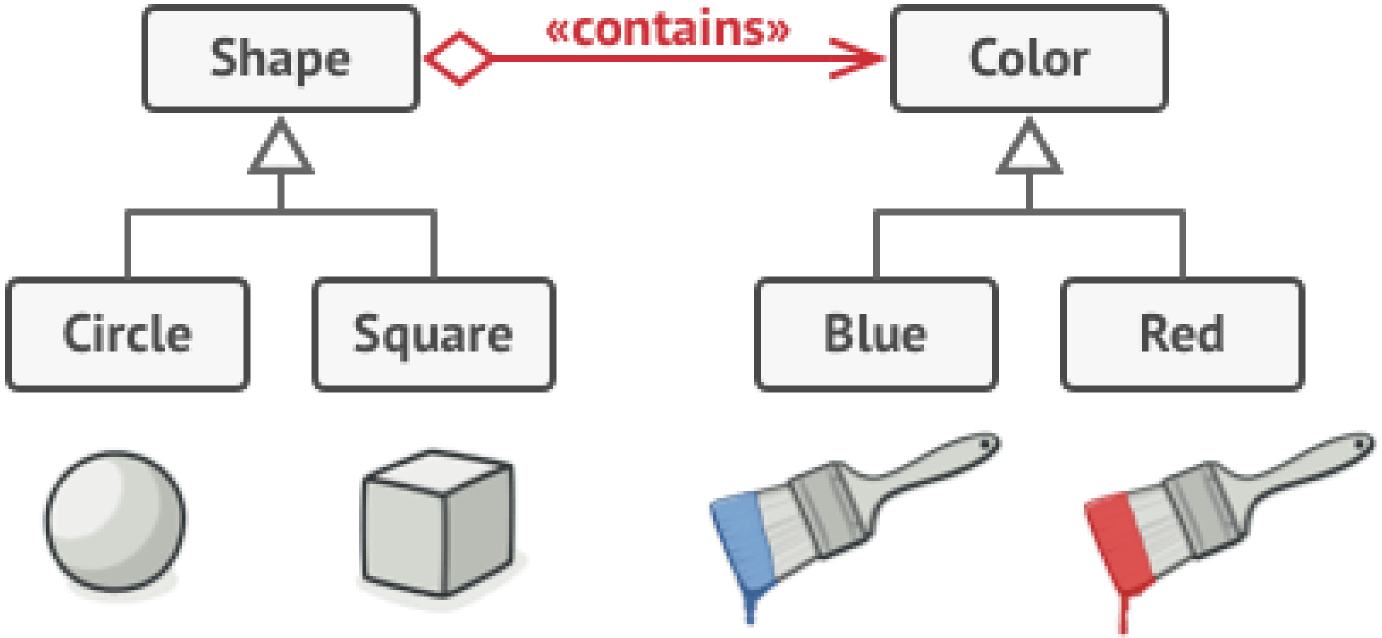
Composition and Liskov substitution principle (and arguably Dependency Inversion Principle) to the rescue!

We separate Color and Shape into separate hierarchy and make Color an attribute of shape. Shape is now composed of color.

If we need to add a new color we just add a class to the color hierarchy.

If we want to add a new shape, we add a class to the Shape hierarchy.

This is much more maintainable.



Bridge – Abstraction and Implementation

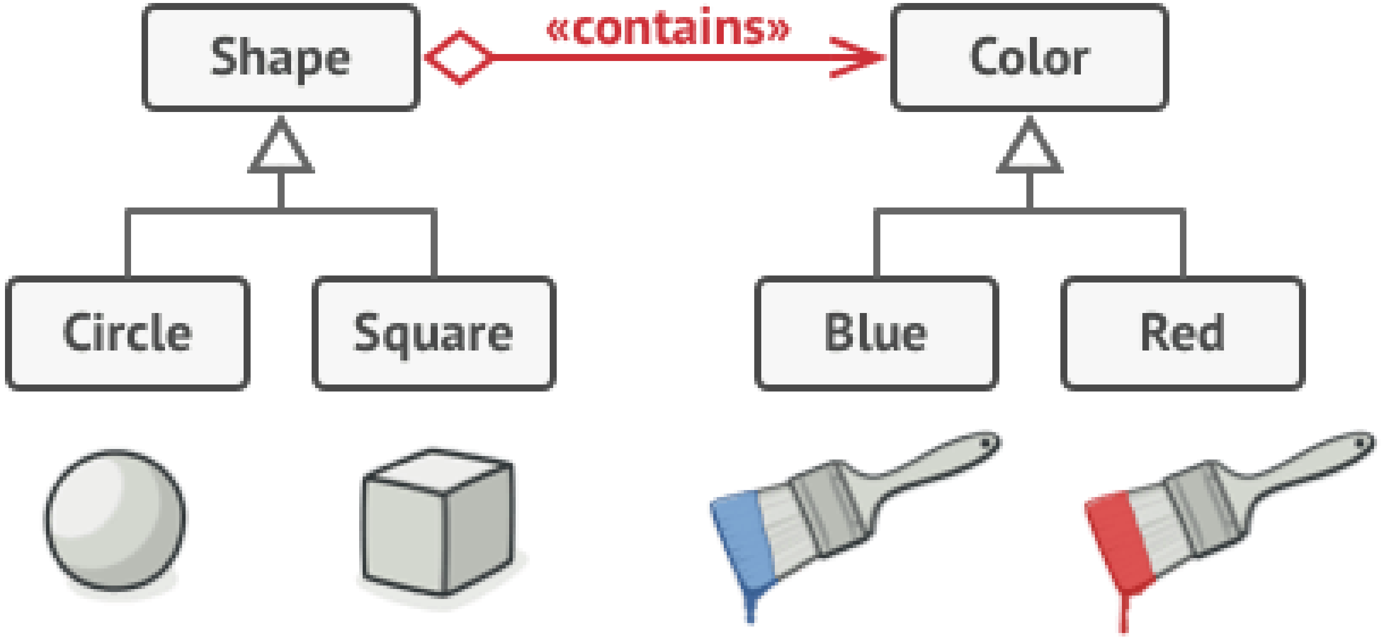
Let’s get philosophical…

Shape in our system has 2 dimensions to it. It has the visual form, which is the shape itself. This is arguably a high level entity.

We say that this visual form is ***“implemented”*** by it’s color. The shape doesn’t do anything, it is just a high level control layer that controls the form a color can take. We say that the Shape is an **“Abstraction”**.

These are relative terms in this case.

simple_bridge_example.py



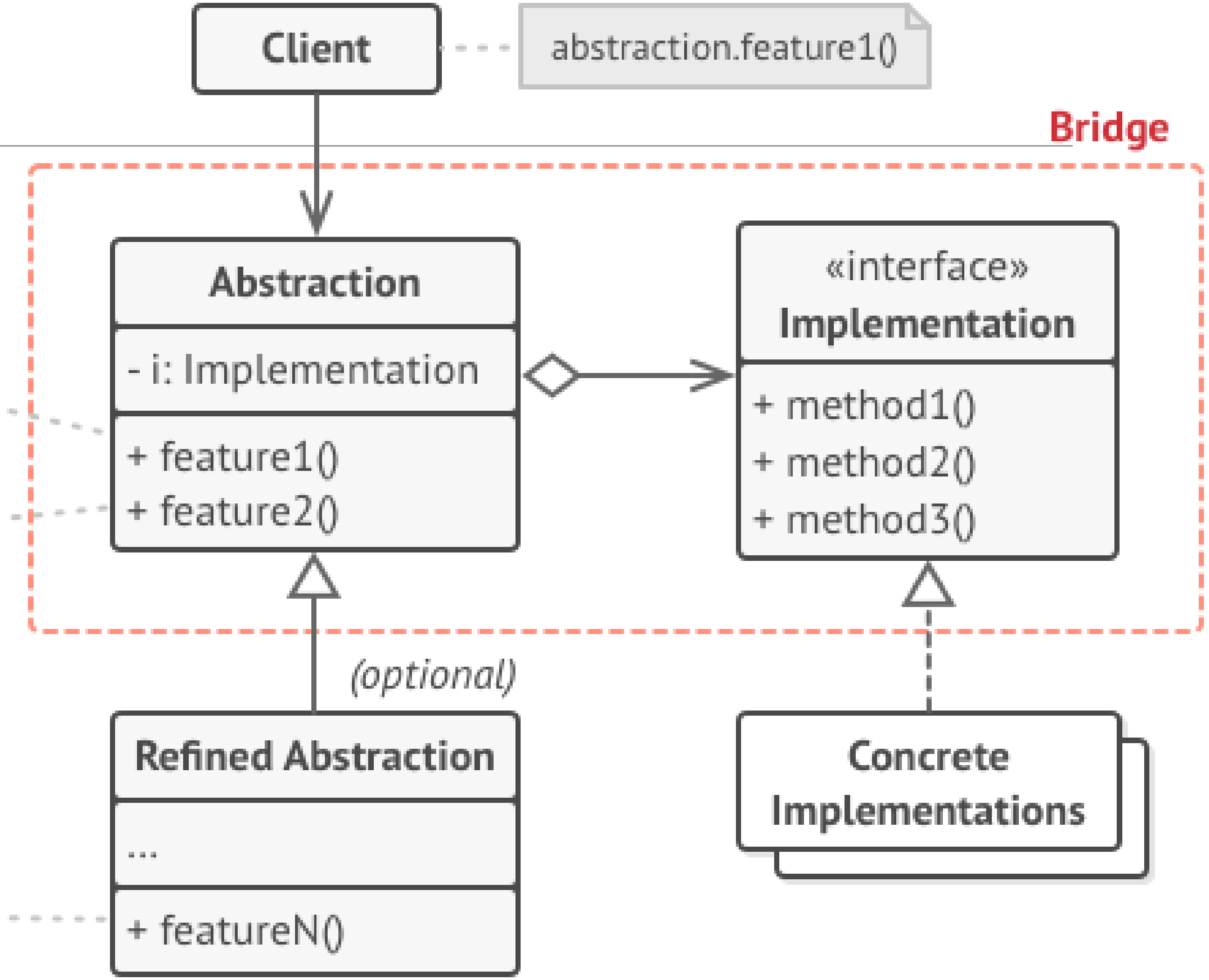
Bridge

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 Shape just controls the form a color takes. It doesn't do anything else.

Let's look at another example.



Bridge – A different example

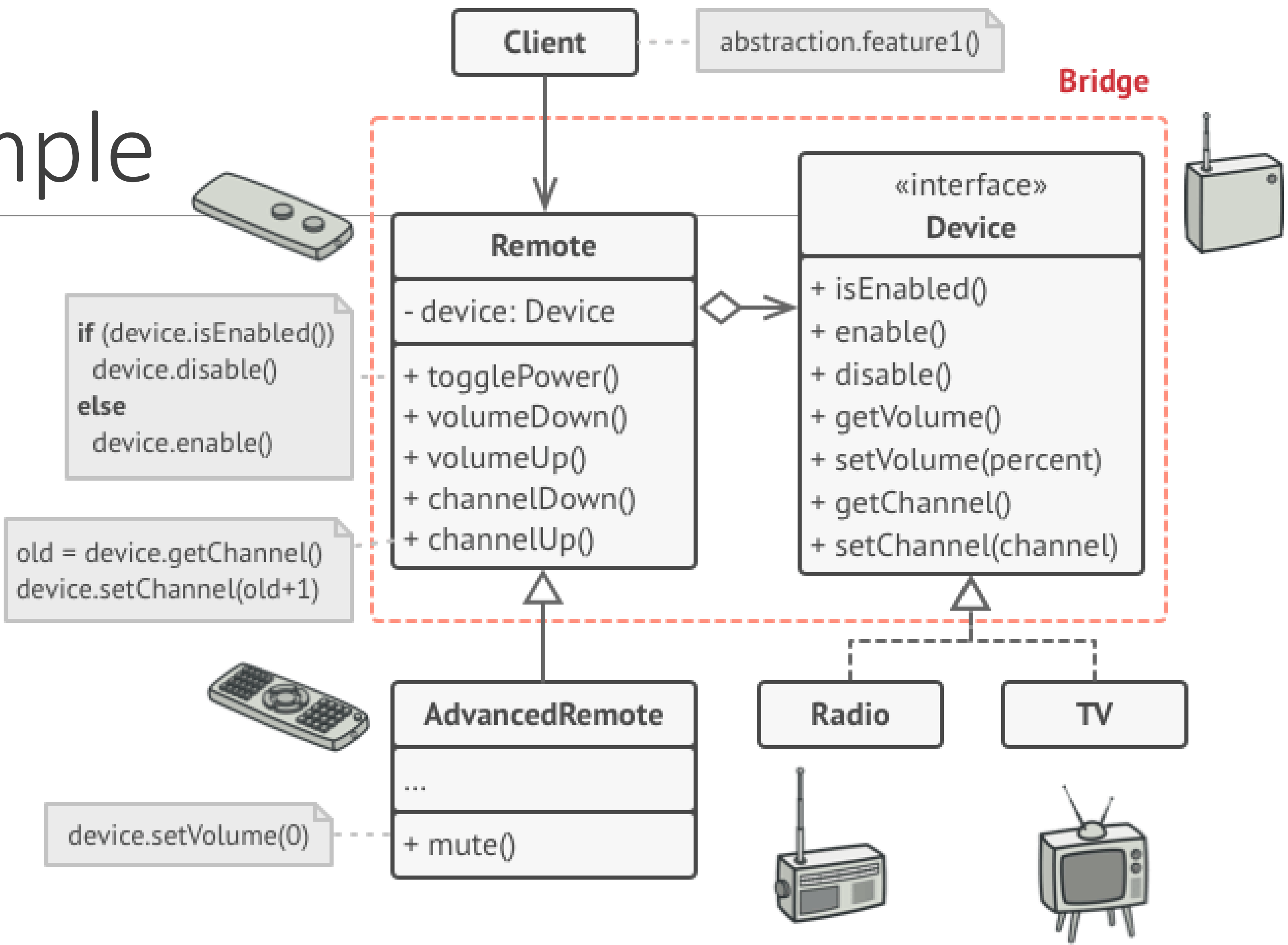
A Remote is an abstraction of a device.

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

The Bridge Pattern let’s us manage two sets of classes that define an entity in which one dimension of it (Remote, or Shape) is coupled with another dimension (Device or Color).

Let’s implement this example.

device_remote_bridge_example.py



Client

abstraction.feature1()

Bridge

Remote

- device: Device

+ togglePower()
+ volumeDown()
+ volumeUp()
+ channelDown()
+ channelUp()

```
if (device.isEnabled())
  device.disable()
else
  device.enable()
```

```
old = device.getChannel()
device.setChannel(old+1)
```

«interface»
Device

+ isEnabled()
+ enable()
+ disable()
+ getVolume()
+ setVolume(percent)
+ getChannel()
+ setChannel(channel)

AdvancedRemote

...

+ mute()

```
device.setVolume(0)
```

Radio

TV

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.
- Create platform independent systems. Another way of saying this is that we can hide platform details.

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.

Decorator Pattern

The problem

How do I support multiple different ‘optional’ behaviours without inflating my inheritance hierarchy and still maintaining the Open/Closed Principle?

An Example:

Doughnut shop menu

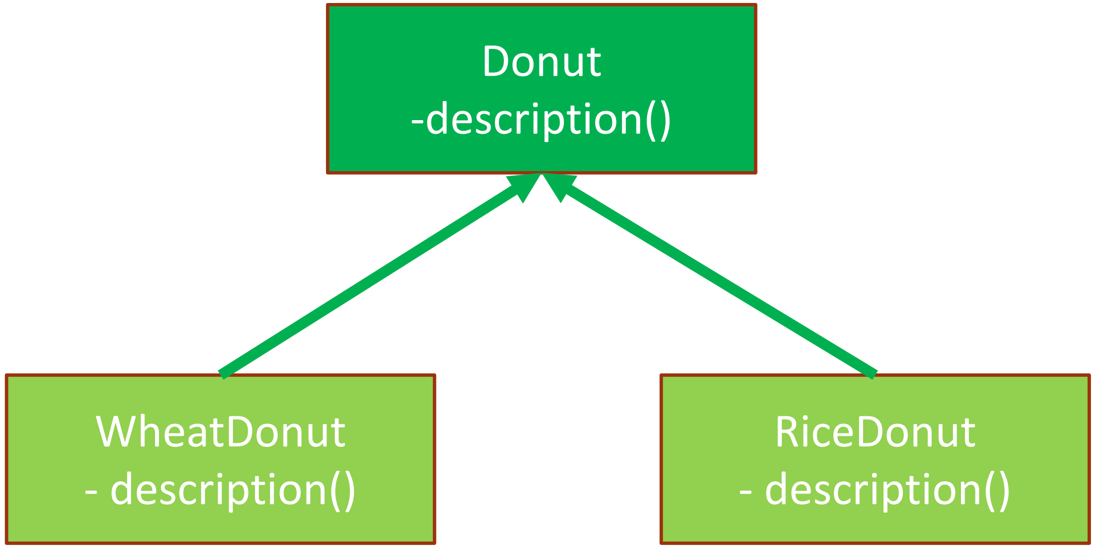
A different class for each doughnut, they all share the same base class and interface but come in an infinite combinations and varieties.

You sell two different types of donuts

- Wheat donut
- Rice donut

Introduction - Problem

- You might model it to look something like this

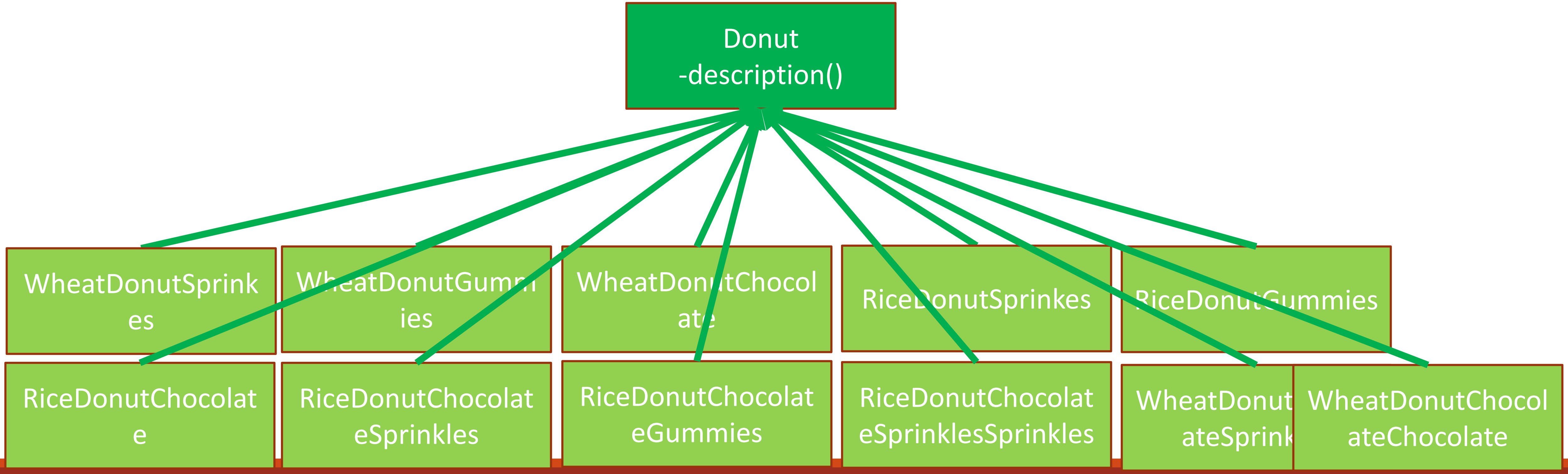


Introduction - Problem

- But then I ask you to have multiple toppings
  - Sprinkles
  - Gummies
  - Chocolate

Introduction - Problem

- You might model it to look something like this (Please don't…)
- Imagine if I asked for combinations (Rice Donut Sprinkles Chocolate)



Donut
-description()

WheatDonutSprinkes

WheatDonutGummies

WheatDonutChocolate

RiceDonutSprinkes

RiceDonutGummies

RiceDonutChocolate

RiceDonutChocolateSprinkles

RiceDonutChocolateGummies

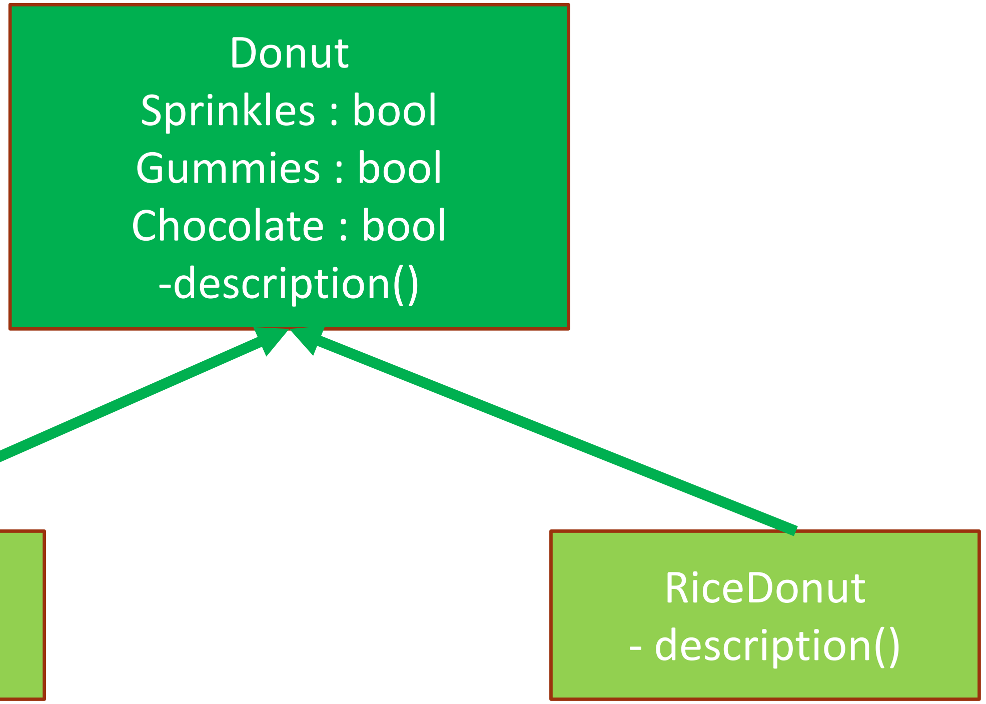
RiceDonutChocolateSprinklesSprinkles

WheatDonut ateSprink

WheatDonutChocolateChocolate

Introduction - Problem

- We can move the toppings to the Donut instead



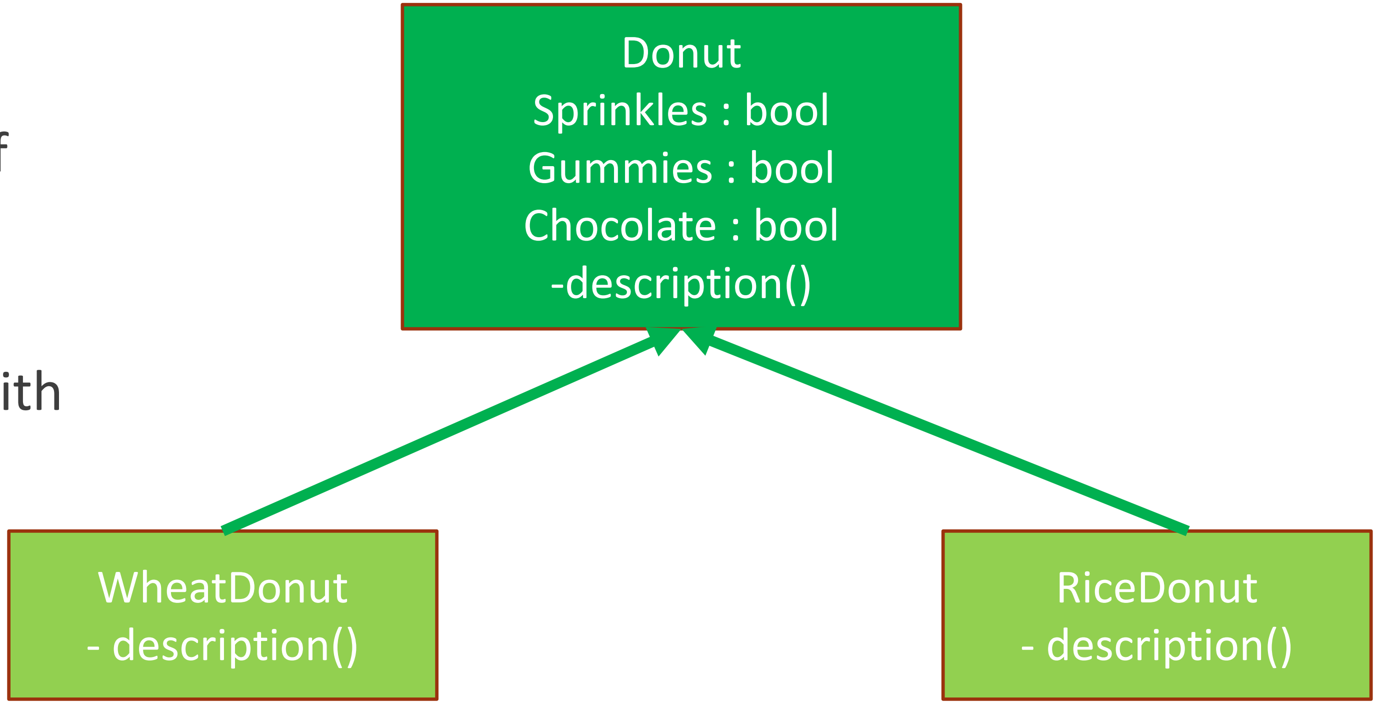
Donut
Sprinkles : bool
Gummies : bool
Chocolate : bool
-description()

WheatDonut
- description()

RiceDonut
- description()

Introduction - Problem

- Issues
  - Violates Open-Closed Principle
  - Will need to modify parent Donut if
    - New toppings added
    - Have double toppings
  - Some toppings might not go well with some Donut, but still inherited



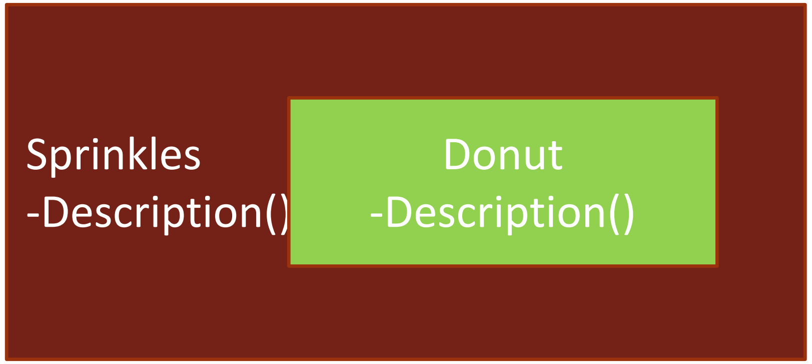
Introduction - Decorator

- Let’s try something different
- Let’s start with a donut object

Donut
-Description()

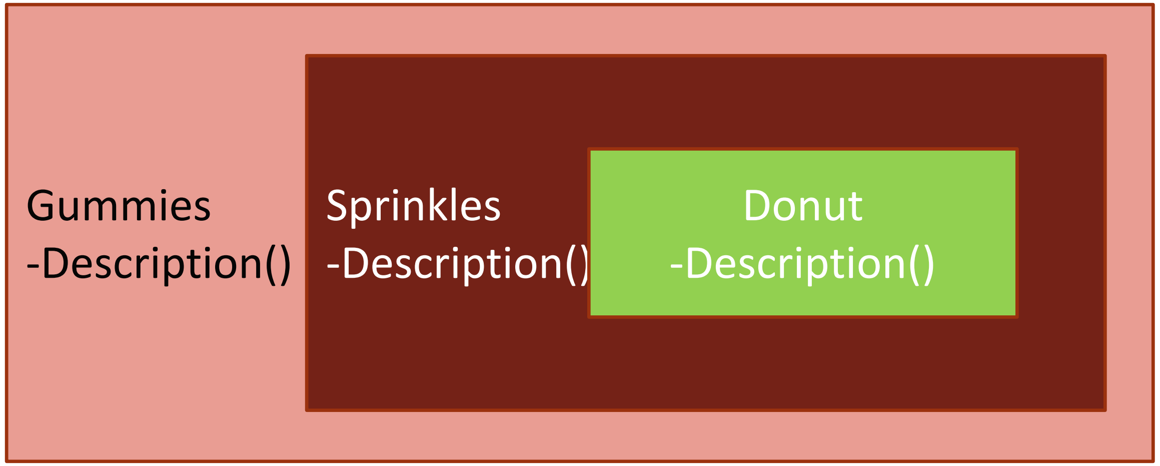
Introduction - Decorator

- We’ll “decorate” our donut object with toppings
  - Wrap our donut object with Sprinkles topping



Introduction - Decorator

- We’ll “decorate” our Donut object with toppings
  - Wrap our donut object with Sprinkles topping
  - Wrap that with another topping, Gummies



Defining Decoration

We ‘wrap’ our base object using decorative pieces.

In the donut analogy, it could be gummies, sprinkles, or even the filling (think of this as decorating the inside).

These decorative pieces

- Re-use the code of the Base Object also known as the Base Component
- Extend the functionality while maintaining the same interface by adding specialized behaviours or overriding attributes.

> “**Decorator** is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.”

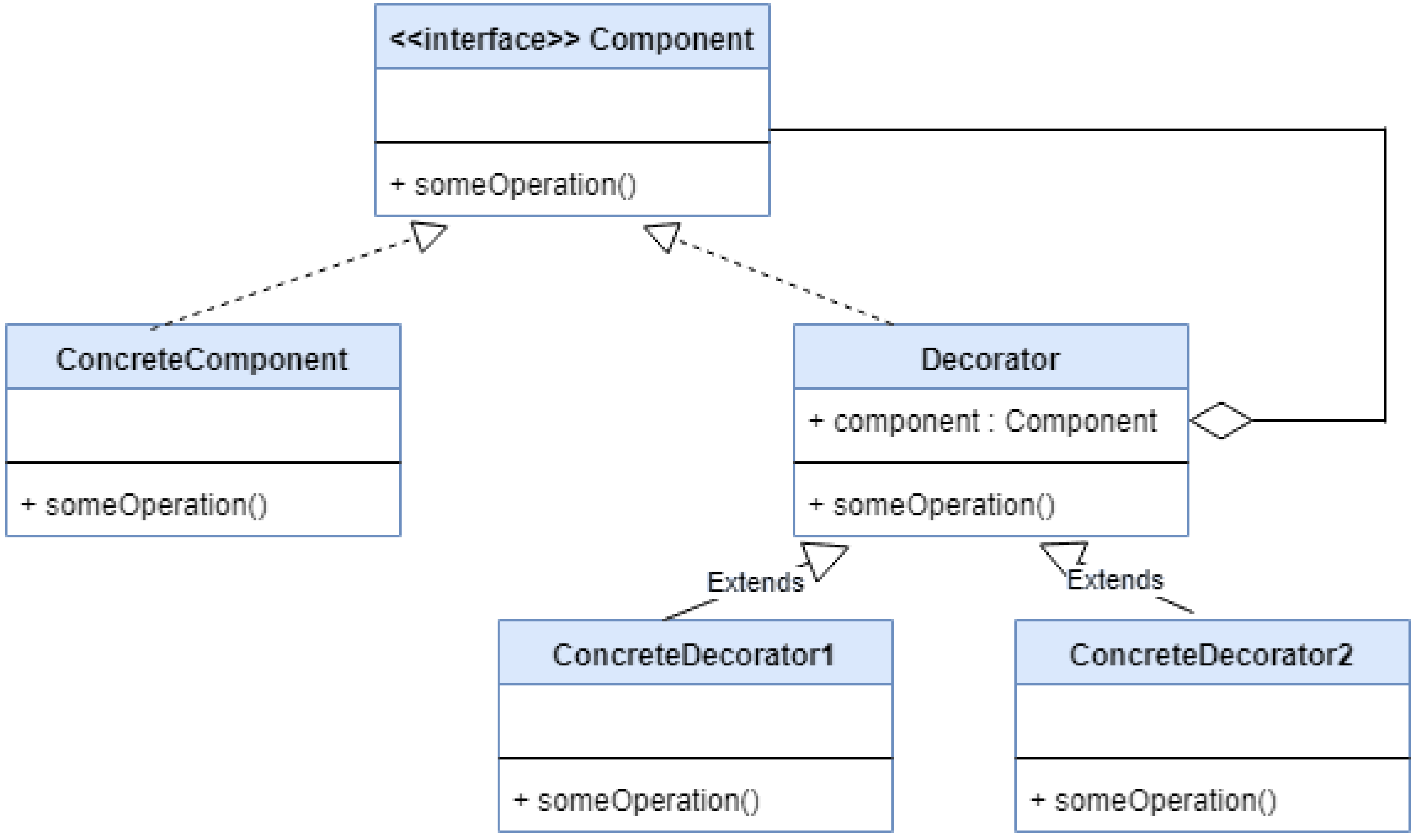
Decorators

Decorators are just wrappers with one added feature.

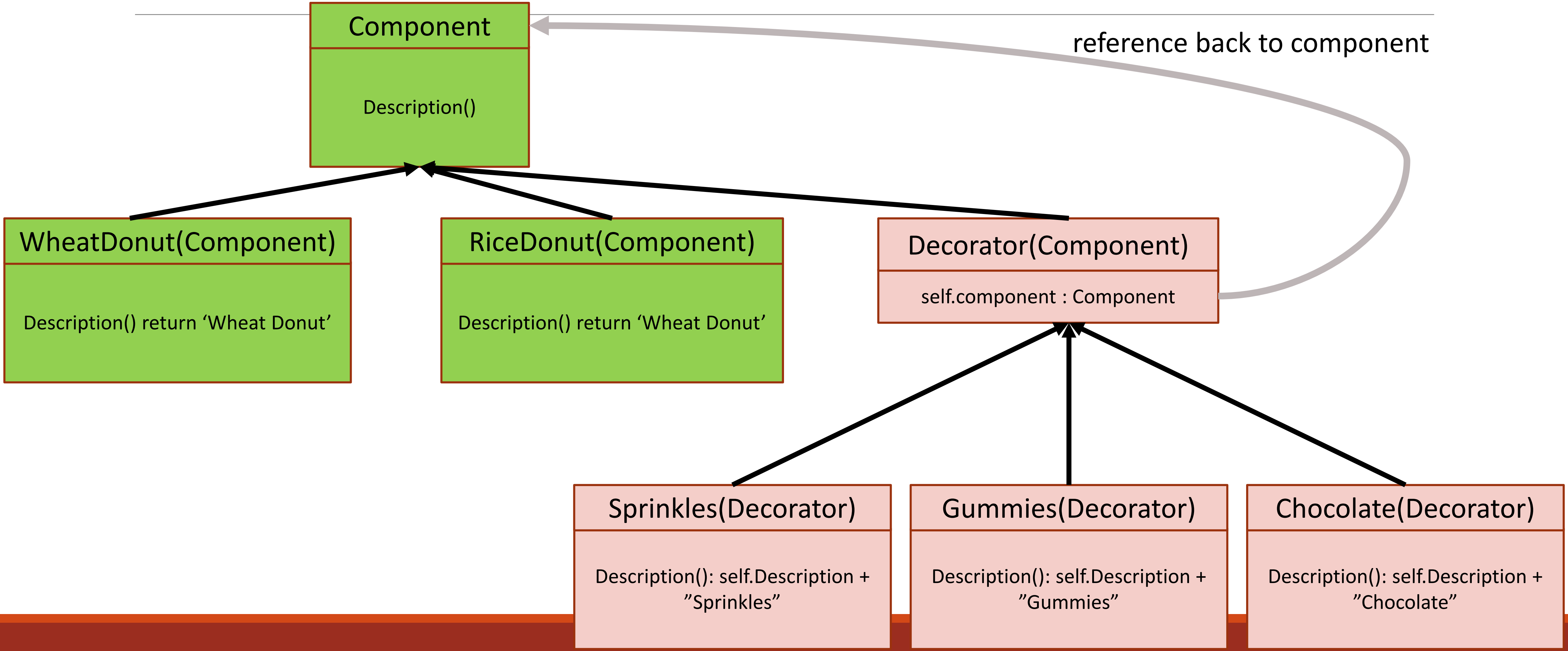
All decorators inherit from a Base Decorator Class. This allows us to define multiple different wrappers that can wrap around the concrete component and around other wrappers as well.

Take a look at the UML Diagram.

The **Decorator** is a **base class** that **wraps around** anything that implements a **Component interface**, like ConcreteComponent for example.



Donuts…



Component

Description()

WheatDonut(Component)

Description() return ‘Wheat Donut’

RiceDonut(Component)

Description() return ‘Wheat Donut’

Decorator(Component)

self.component : Component

reference back to component

Sprinkles(Decorator)

Description(): self.Description + ”Sprinkles”

Gummies(Decorator)

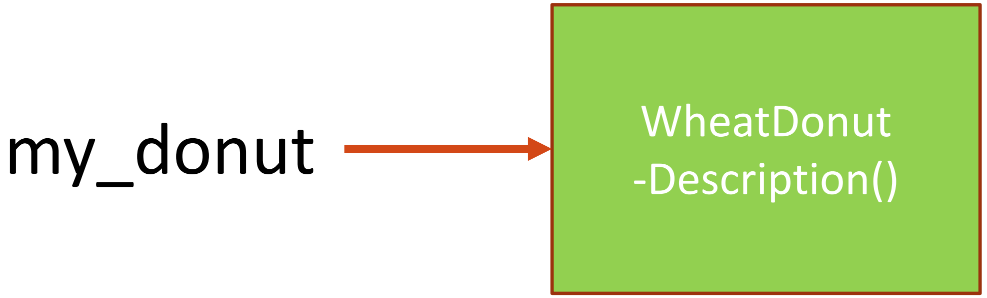
Description(): self.Description + ”Gummies”

Chocolate(Decorator)

Description(): self.Description + ”Chocolate”

Introduction - Decorator

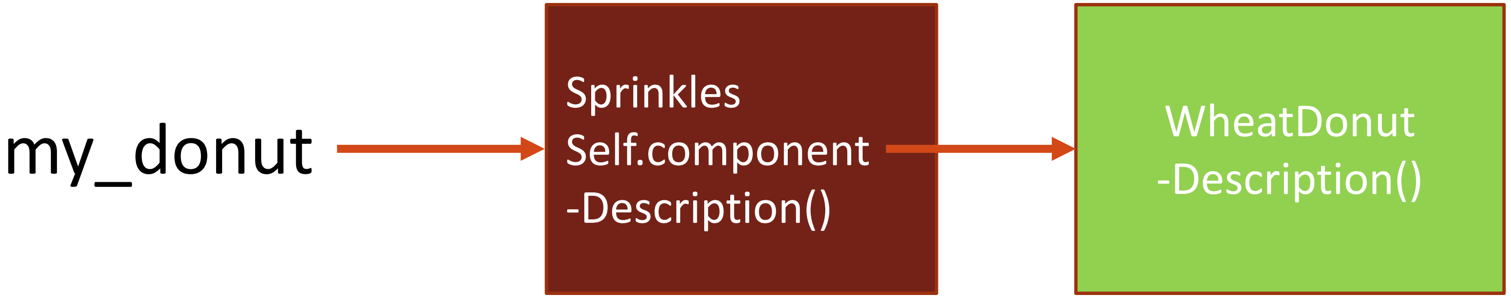
- my_donut = WheatDonut();



- Create WheatDonut component

Introduction - Decorator

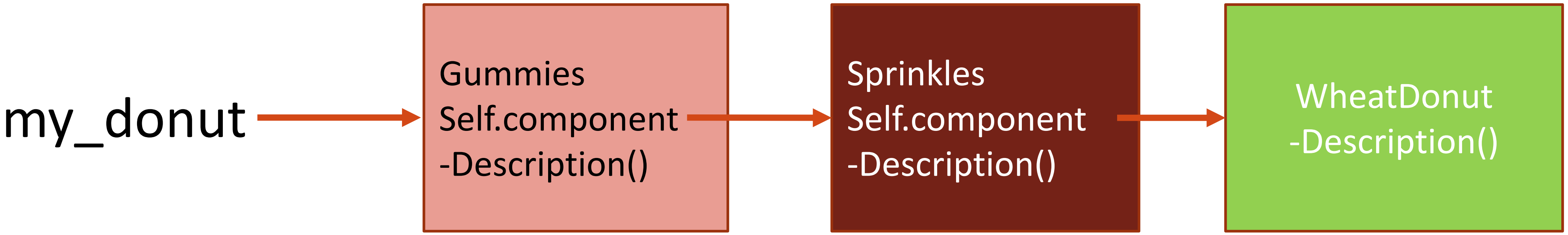
- my_donut = WheatDonut();
- my_donut = Sprinkles(my_donut);



- Create Sprinkles object, set its internal variable to point to my_donut
- Sprinkle’s grandparent base class is Component, so the reference it returns is saved into my_donut

Introduction - Decorator

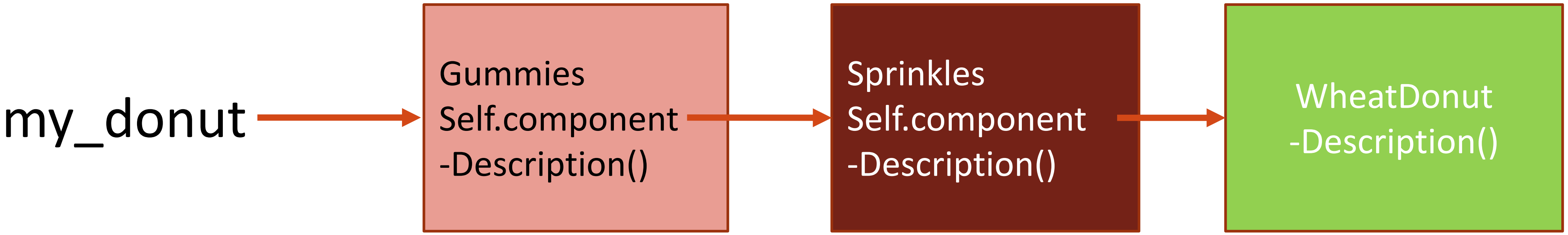
- my_donut = WheatDonut();
- my_donut = Sprinkles(my_donut);
- my_donut = Gummies(my_donut);



- Create Gummies object, set its internal variable to point to my_donut
- Gummies' grandparent base class is Component, so the reference it returns is saved into my_donut

Introduction - Decorator

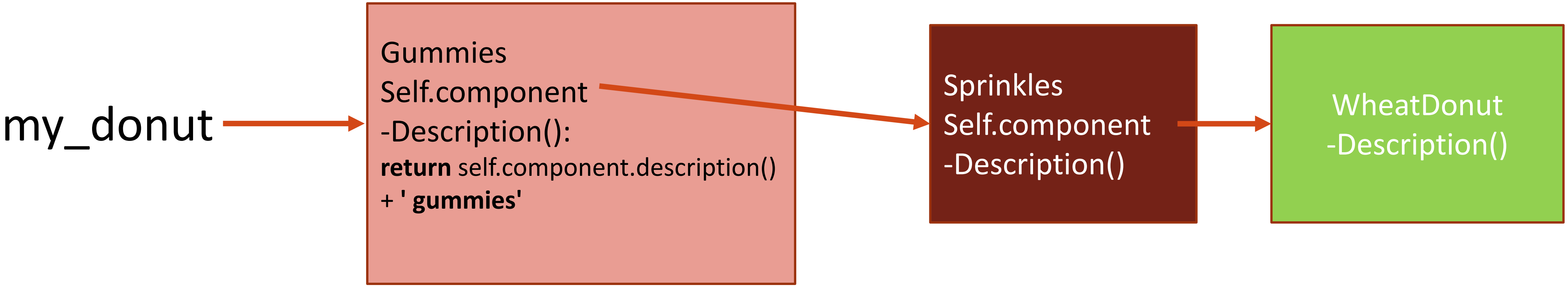
- my_donut = WheatDonut();
- my_donut = Sprinkles(my_donut);
- my_donut = Gummies(my_donut);
- print(my_donut.description())



- Notice how starting from **my_donut**, it looks like a **linked list**
- Get total description of the donut by calling my_donut->Description();

Introduction - Decorator

- my_donut = WheatDonut();
- my_donut = Sprinkles(my_donut);
- my_donut = Gummies(my_donut);
- print(my_donut.description())



my_donut

Gummies
Self.component
-Description():
return self.component.description()
+ **' gummies'**

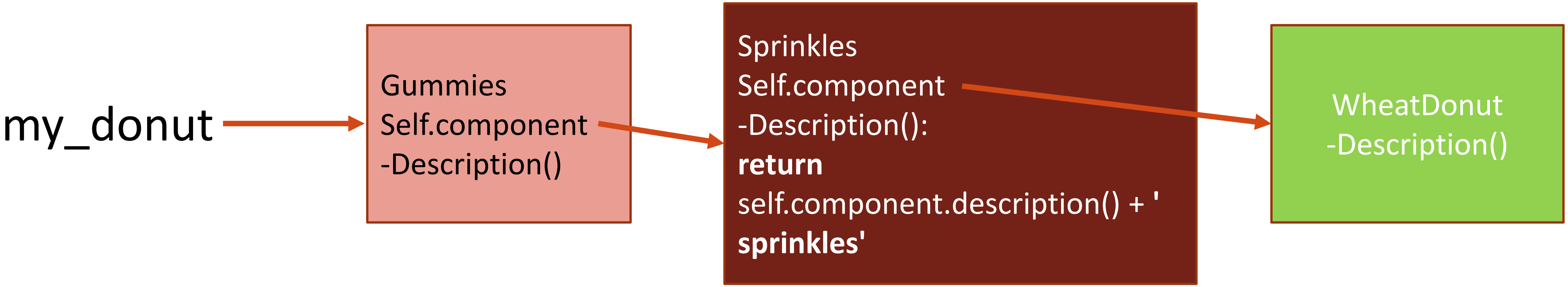
Sprinkles
Self.component
-Description()

WheatDonut
-Description()

- Calls will chain into toppings until they reach WheatDonut

Introduction - Decorator

- my_donut = WheatDonut();
- my_donut = Sprinkles(my_donut);
- my_donut = Gummies(my_donut);
- print(my_donut.description())



my_donut

Gummies
Self.component
-Description()

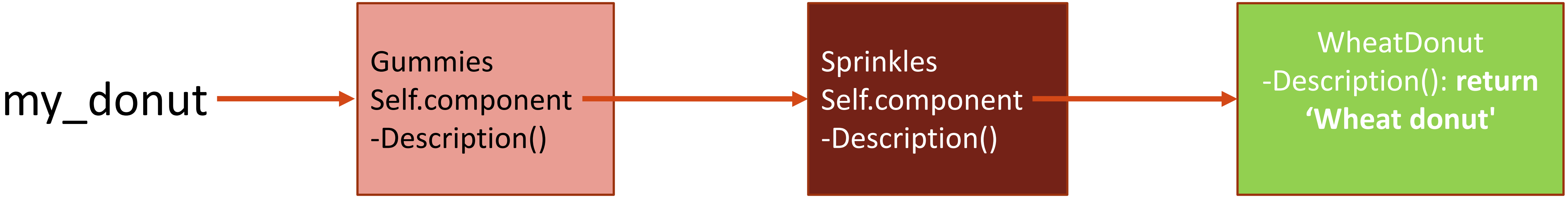
Sprinkles
Self.component
-Description():
return
self.component.description() + '
sprinkles'

WheatDonut
-Description()

- Calls will chain into toppings until they reach WheatDonut

Introduction - Decorator

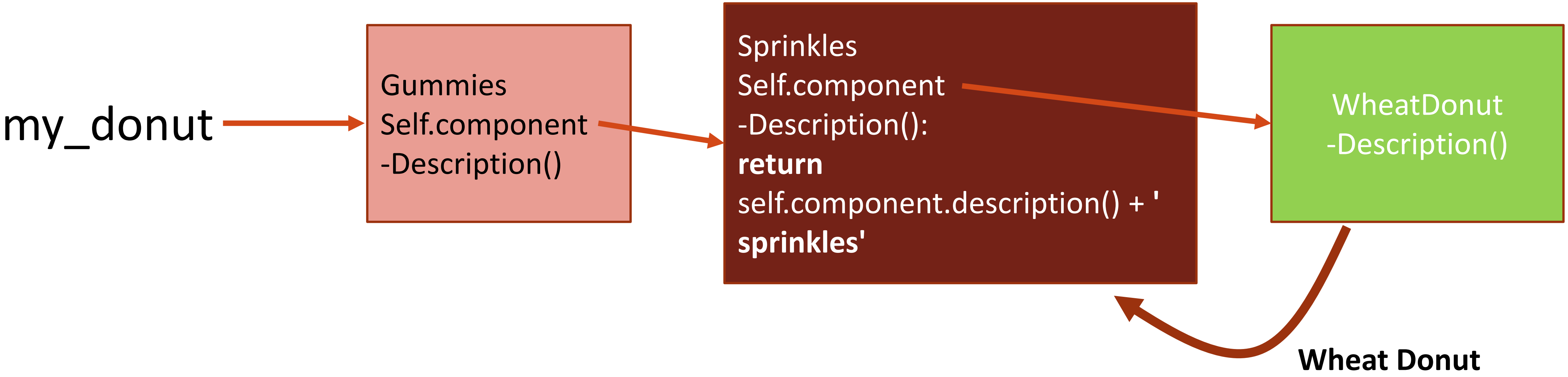
- my_donut = WheatDonut();
- my_donut = Sprinkles(my_donut);
- my_donut = Gummies(my_donut);
- print(my_donut.description())



- Descriptions added up as we return from Description() function calls

Introduction - Decorator

- my_donut = WheatDonut();
- my_donut = Sprinkles(my_donut);
- my_donut = Gummies(my_donut);
- print(my_donut.description())



my_donut

Gummies
Self.component
-Description()

Sprinkles
Self.component
-Description():
return self.component.description() + ' **sprinkles'**

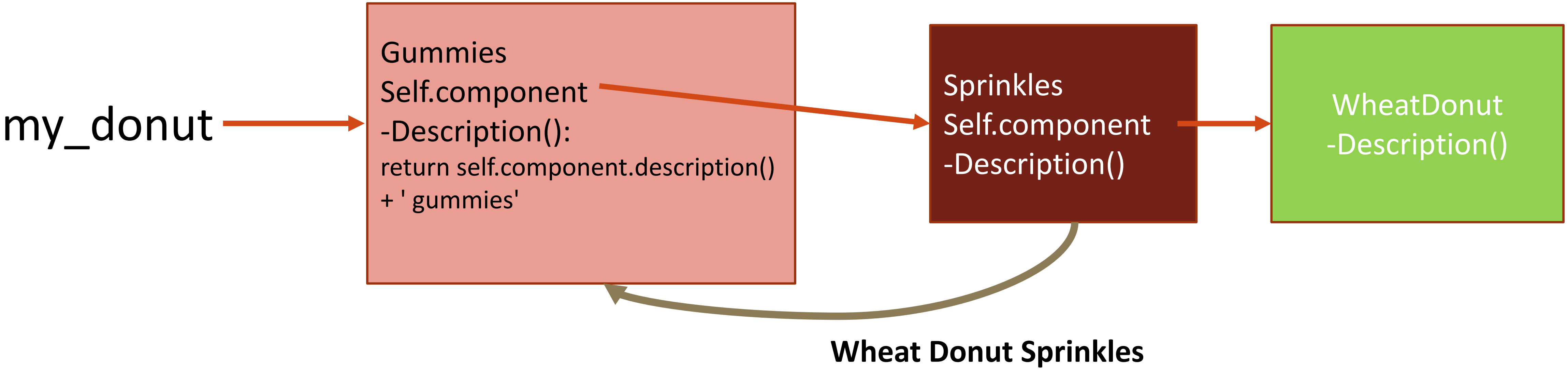
WheatDonut
-Description()

Wheat Donut

- Calls will chain into toppings until they reach WheatDonut

Introduction - Decorator

- my_donut = WheatDonut();
- my_donut = Sprinkles(my_donut);
- my_donut = Gummies(my_donut);
- print(my_donut.description())



my_donut

Gummies
Self.component
-Description():
return self.component.description()
+ ' gummies'

Sprinkles
Self.component
-Description()

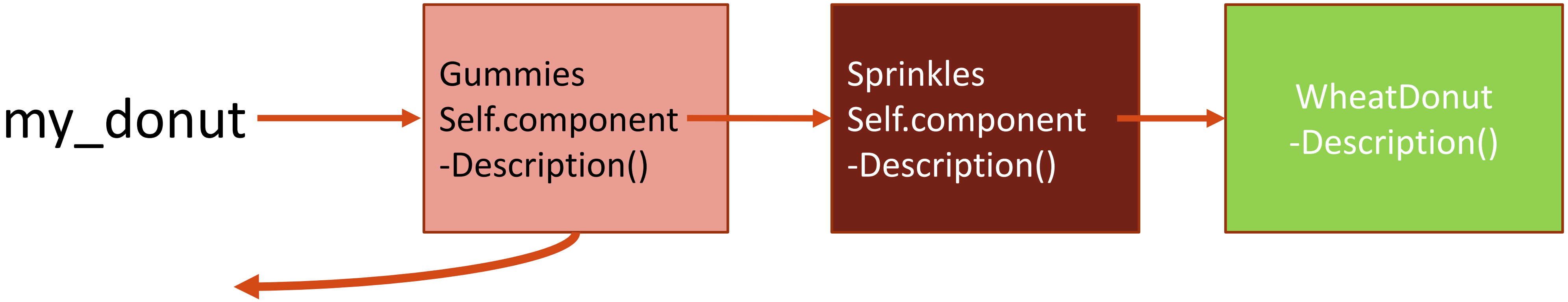
WheatDonut
-Description()

Wheat Donut Sprinkles

- Calls will chain into toppings until they reach WheatDonut

Introduction - Decorator

- my_donut = WheatDonut();
- my_donut = Sprinkles(my_donut);
- my_donut = Gummies(my_donut);
- print(my_donut.description())



my_donut

Gummies
Self.component
-Description()

Sprinkles
Self.component
-Description()

WheatDonut
-Description()

Wheat donut sprinkles gummies

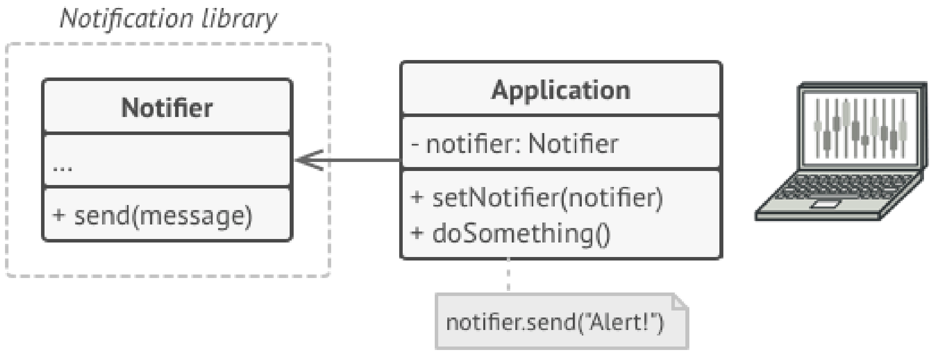
- Calls will chain into toppings until they reach WheatDonut

simple_decorator.py

Let’s take a look at a real world example..

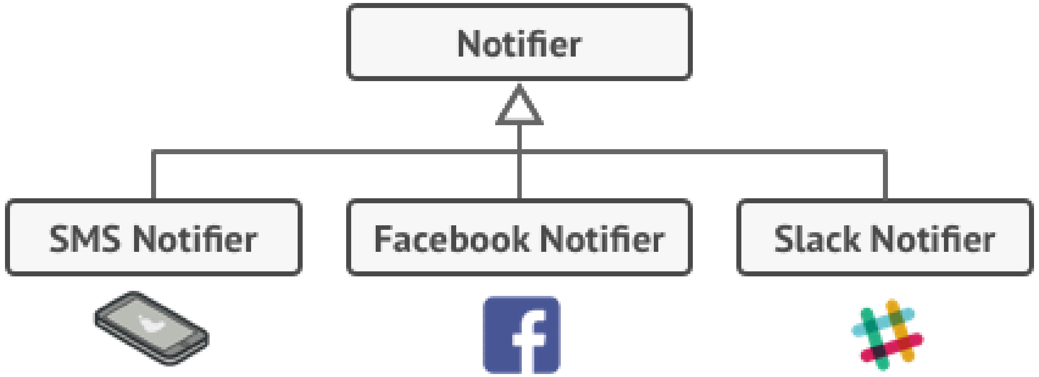
Say we have a notification system.

When something happens we want to send a notification, perhaps via email.



But then over time, we realize that we may want to send notifications via other channels as well. So we do what any sensible programmer would do.

We inherit! Notifier is now a base class and we can have different notifications. Easy right?



Let's take a look at a real world example..

Wait a second, what if we want to send a notification alongside multiple channels at once?

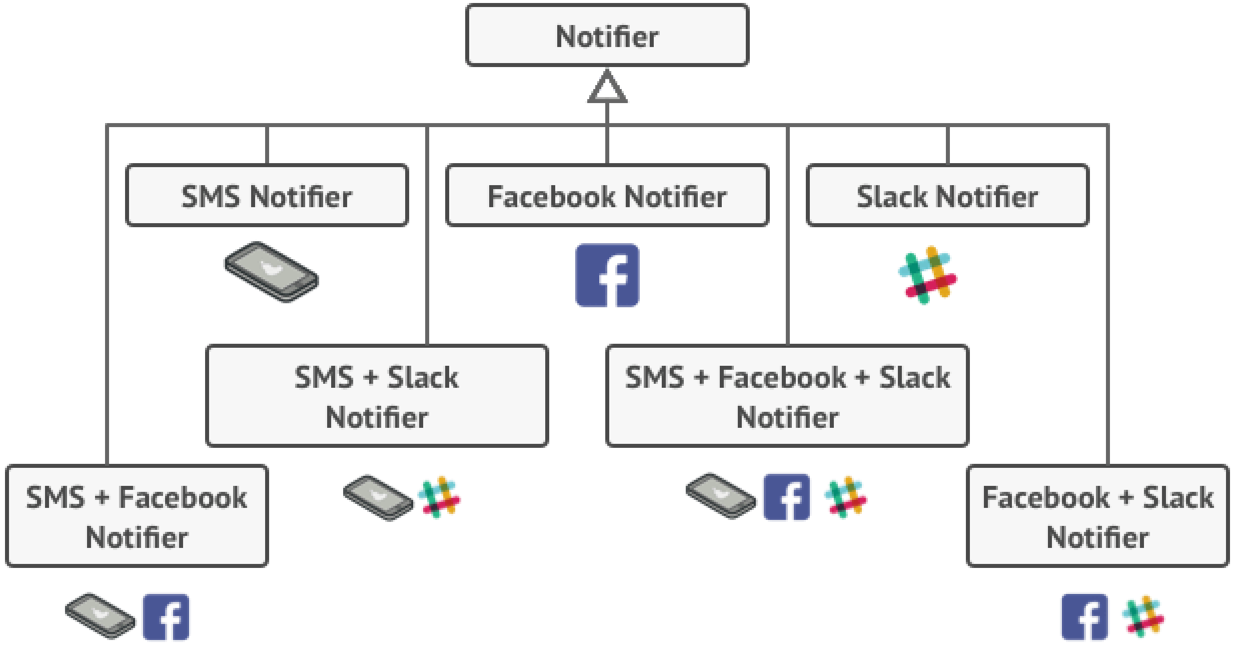
Different users have different devices or apps.

Person A might receive facebook and sms notifications, while **Person B** might receive email and slack notifications.

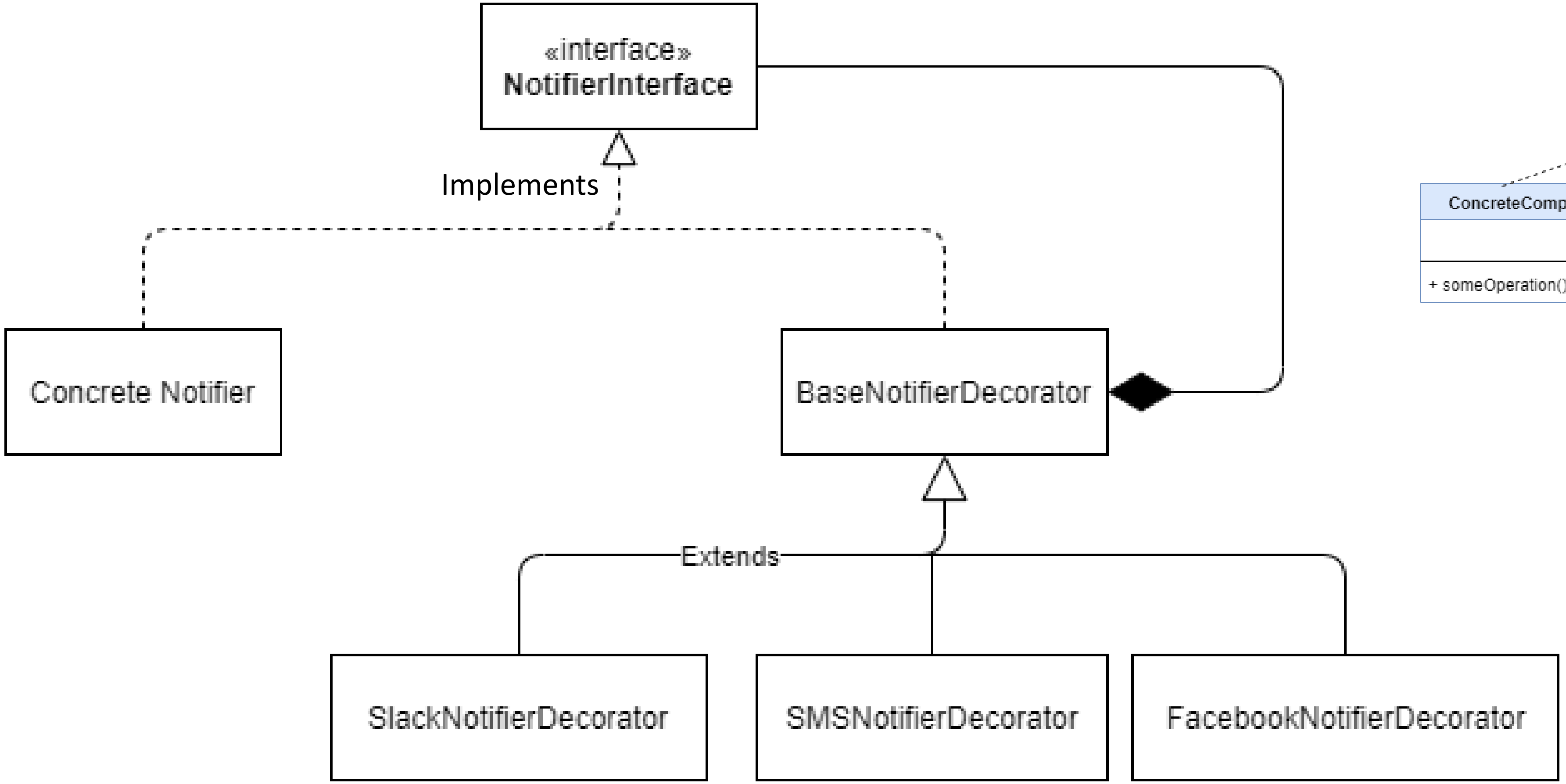
Person C may receive all!

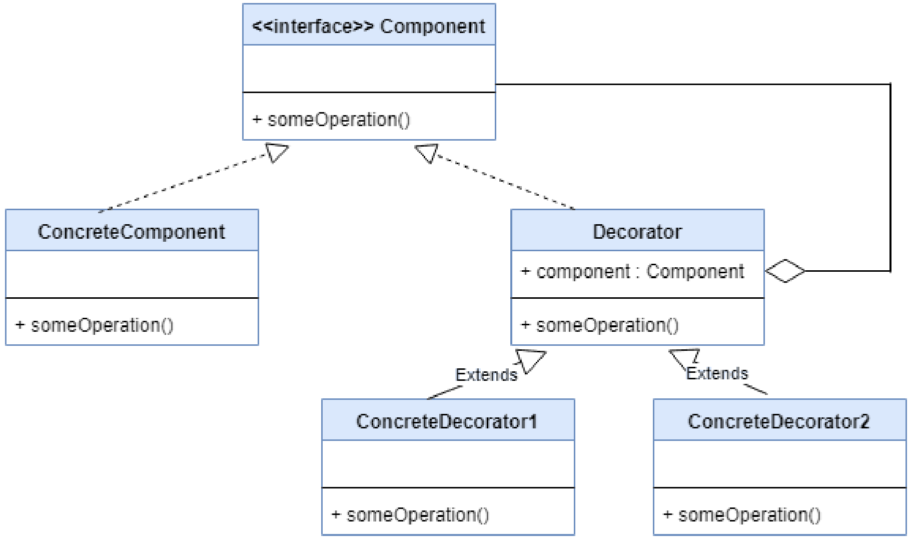
This inheritance hierarchy is becoming convoluted and difficult to maintain. There must be a simpler solution!

What if we defined each channel as a decorator that could wrap around a Notifier?



Notifier Decorator





Let’s implement the Decorator Pattern together.

Say we have a FileDataSource.

We can Read and Write to this Data Source.

Say we wanted to encrypt the data before writing and decrypt it after reading

Say we also wanted to compress the data using some compression algorithm before writing and decompress it before reading.

Now what if we wanted to encrypt and compress the data?

Let’s avoid a multiple inheritance scenario (CompressedEncryptedFileDataSource – Yikes) .

Let’s implement wrappers that extend these behaviours and see it in action!

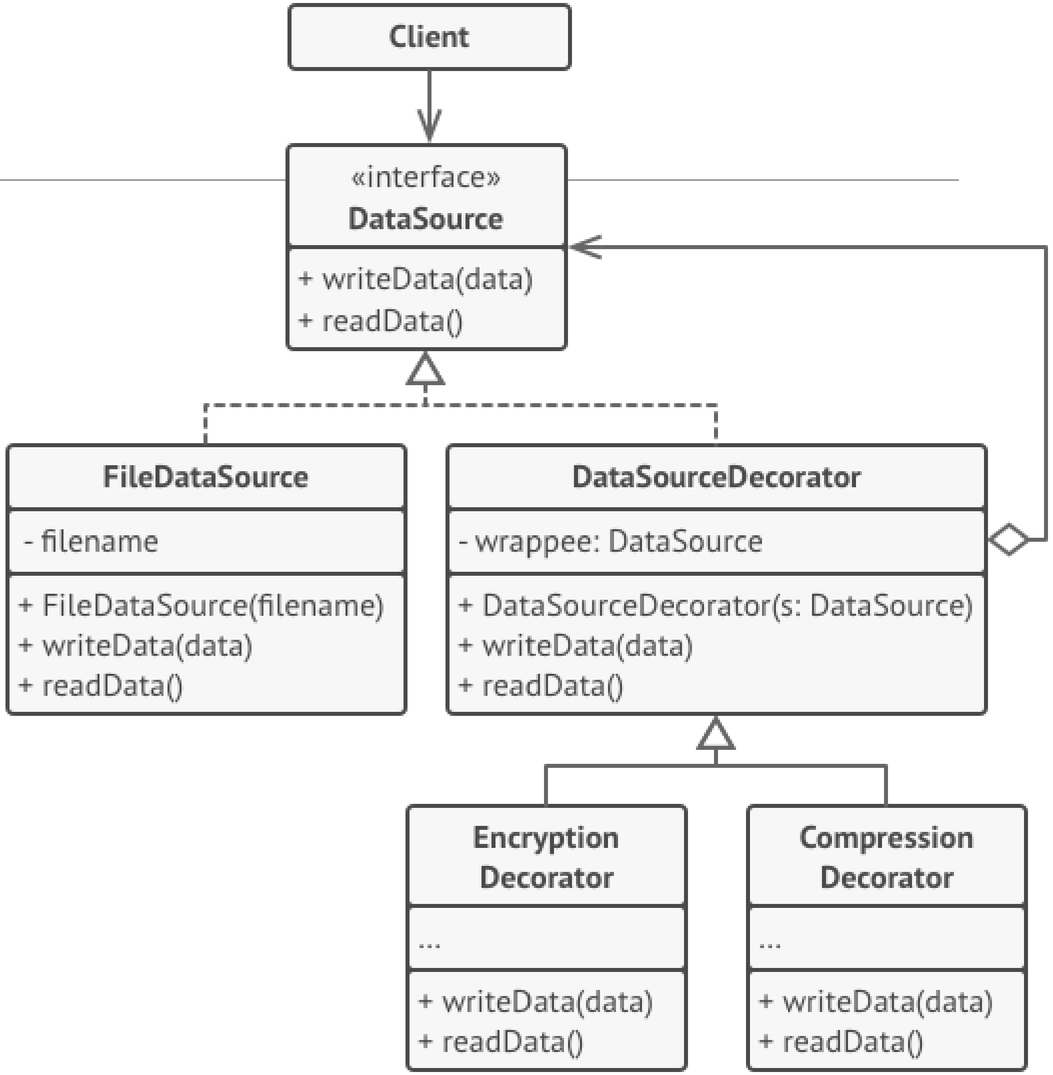
Data Source

Here is the UML Diagram for the DataSource implementation.

Remember these are language agnostic. They don't depend on any specific programming language

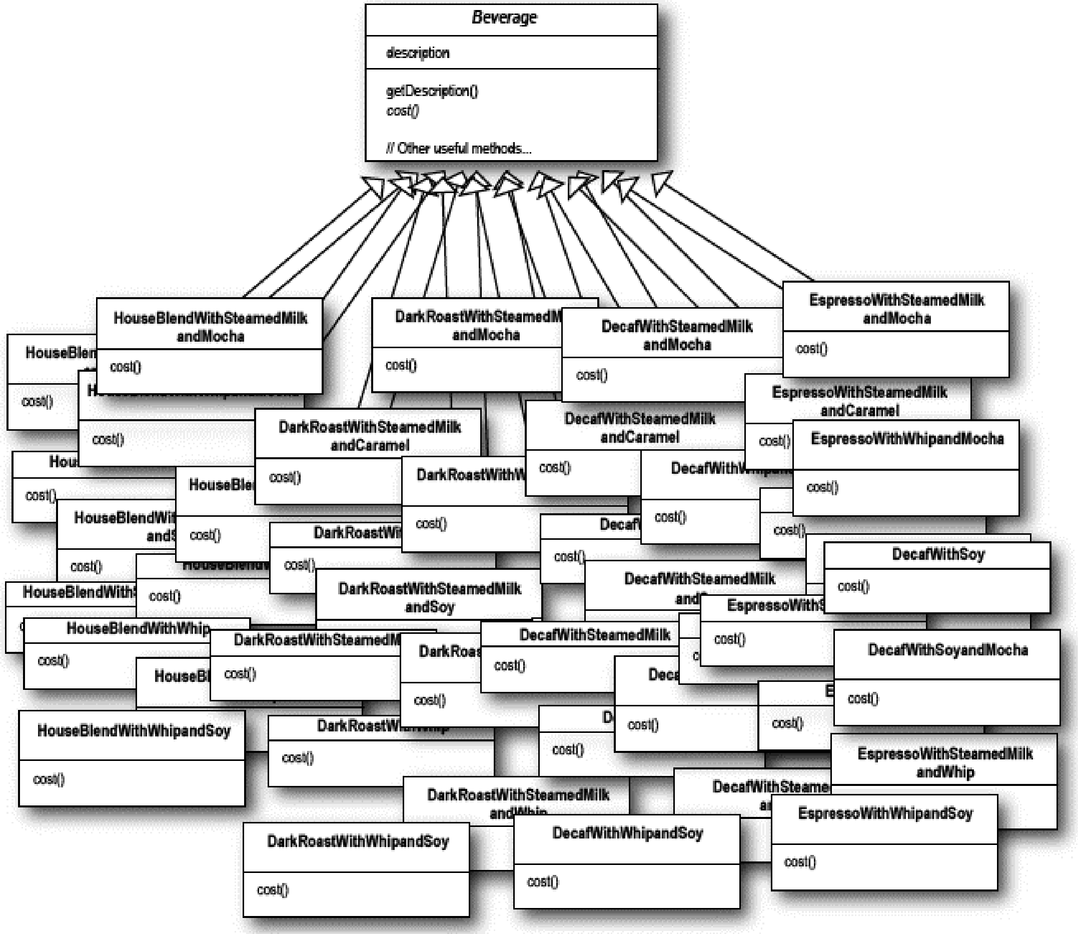
In Python we have the flexibility to not create a separate formal DataSource Interface and just leave it to Duck Typing.

decorator_example.py



Decorator Advantages

We get to avoid this:


Beverage
description
getDescription()
cost()
// Other useful methods...
HouseBlendWithSteamedMilk andMocha
DarkRoastWithSteamedMilk andMocha
DecafWithSteamedMilk andMocha
EspressoWithSteamedMilk andMocha
EspressoWithSteamedMilk andCaramel
EspressoWithWhipandMocha
DarkRoastWithSteamedMilk andCaramel
DecafWithSteamedMilk andCaramel
DarkRoastWithSteamedMilk andSoy
DecafWithSoy
DecafWithSoyandMocha
HouseBlendWithWhip
HouseBlendWithWhipandSoy
DarkRoastWithWhip
DecafWithSteamedMilk
EspressoWithSteamedMilk andWhip
DarkRoastWithSteamedMilk andWhip
EspressoWithWhipandSoy
DarkRoastWithWhipandSoy
DecafWithWhipandSoy
cost()

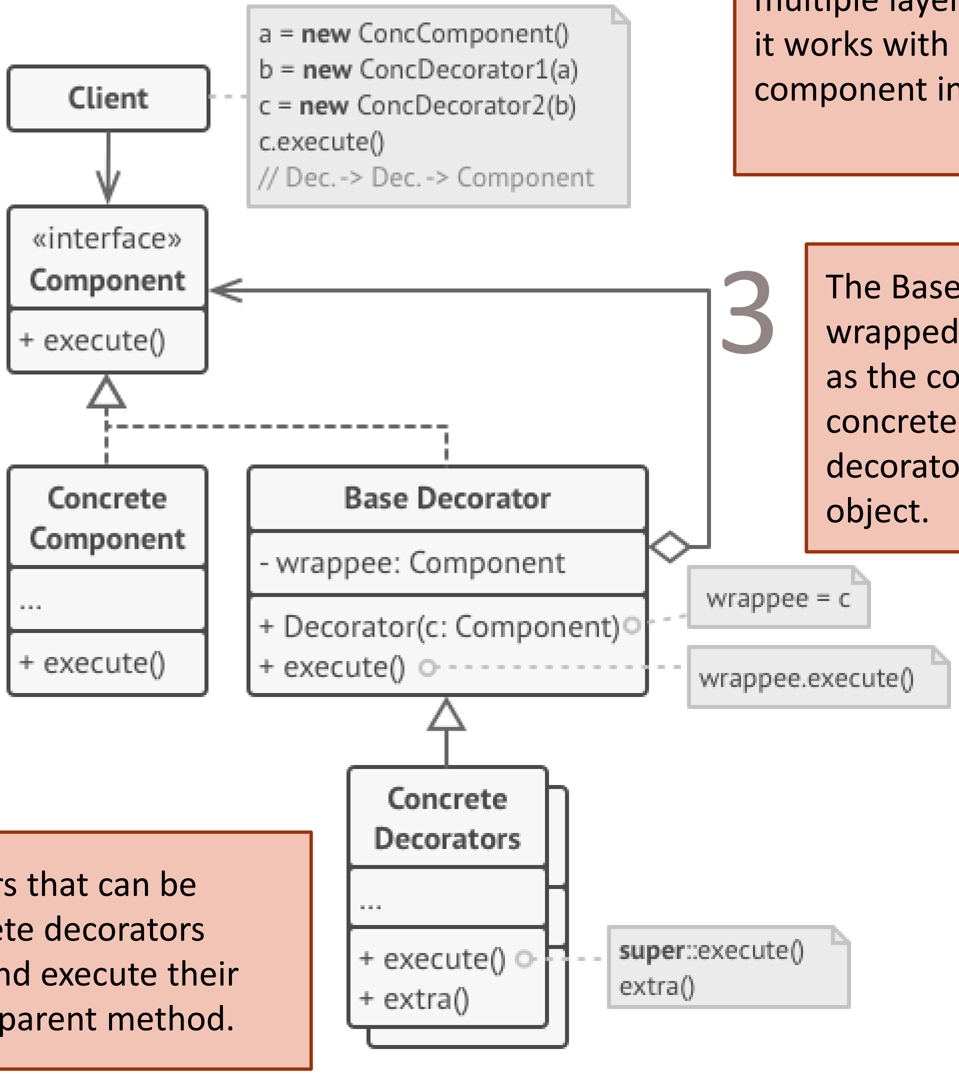
Decorator Advantages

- Adhere to the Single Responsibility Principle. Each decorator is responsible for one thing.
- Adhere to the Open Close Principle. Our Concrete Component is closed to modifications but open to extension (Indirectly via decorators).
- Allow for multiple combinations and behaviours without the need for multiple inheritance. We avoid directly sub-classing from the concrete component itself.
- Add or remove responsibilities at run-time.

Decorator Disadvantages

- Can be difficult to remove or access the concrete object or a specific wrapper in the stack of wrappers.
- Subclasses need to maintain the same interface (probably a good thing, but restrictive).
- Can be difficult to trace and debug if too many decorators are layered one upon the other.
- Hard to avoid scenarios where the behavior changes based on the order in the decorators stack.

Recap



```
a = new ConcComponent()
b = new ConcDecorator1(a)
c = new ConcDecorator2(b)
c.execute()
// Dec. -> Dec. -> Component
```

1 The **Component** declares the common interface for both wrappers and wrapped objects.

2 Concrete Component is a class of objects being wrapped. It defines the basic behavior, which can be altered by decorators.

3 The Base Decorator class has a field for referencing a wrapped object. The field's type should be declared as the component interface so it can contain both concrete components and decorators. The base decorator delegates all operations to the wrapped object.

4 Concrete Decorators define extra behaviors that can be added to components dynamically. Concrete decorators override methods of the base decorator and execute their behavior either before or after calling the parent method.

5 The Client can wrap components in multiple layers of decorators, as long as it works with all objects via the component interface.

SOURCE: https://refactoring.guru/design-patterns/decorator

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”.

That’s it for this week!

Quiz next Friday!

Everything we did this week and week 6.