Strategy, State, Mediator, Decorator

COMP3522 OBJECT ORIENTED PROGRAMMING 2

WEEK 12

https://refactoring.guru/design-patterns

Categorizing Design Patterns

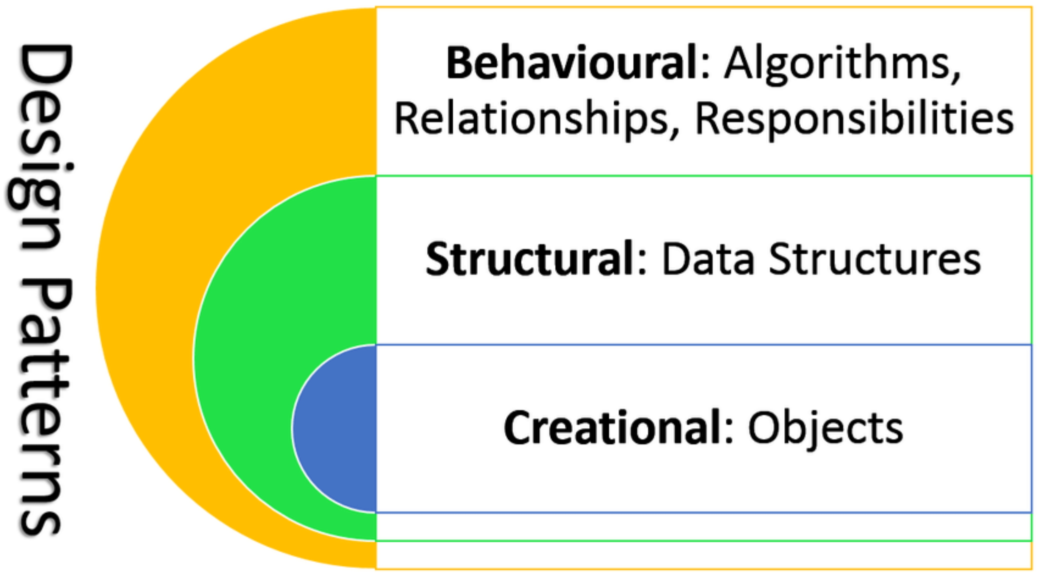
- ❑ **Behavioural (We are looking at these!)**

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

- ❑ **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


Design Patterns
Behavioural: Algorithms, Relationships, Responsibilities
Structural: Data Structures
Creational: Objects

Strategy

WHEN YOU WANT TO BE ABLE TO CHANGE YOUR PLANS

Liskov Substitution Principle

In programming, the Liskov substitution principle states that if ***S*** is a subtype of ***T***, then objects of type ***T*** may be replaced (or substituted) with objects of type ***S***.

Or

Objects in a program should be replaceable with instances of their base types without altering the correctness of that program.

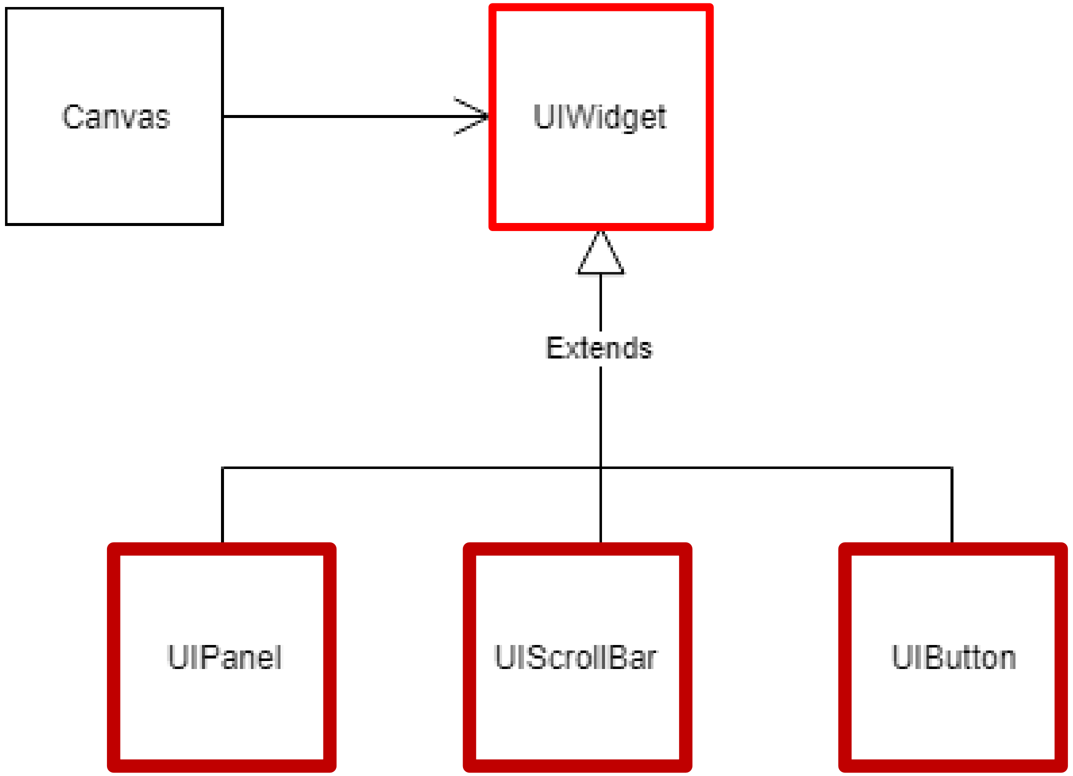
LISKOV SUBSTITUTION PRINCIPLE
If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

Liskov Substitution Principle

```
struct Canvas {
     vector<UIWidget *>ui_widgets;

    Canvas(vector<UIWidget *> widget_list) {
        ui_widgets = widget_list; }

    void draw_screen():
        for (UIWidget *widget : ui_widgets) {
            widget->draw();

}
```

Wherever **UIWidget** is used, we should be able to substitute its subclasses **UIPanel**, **UIScrollBar**, and **UIButton**

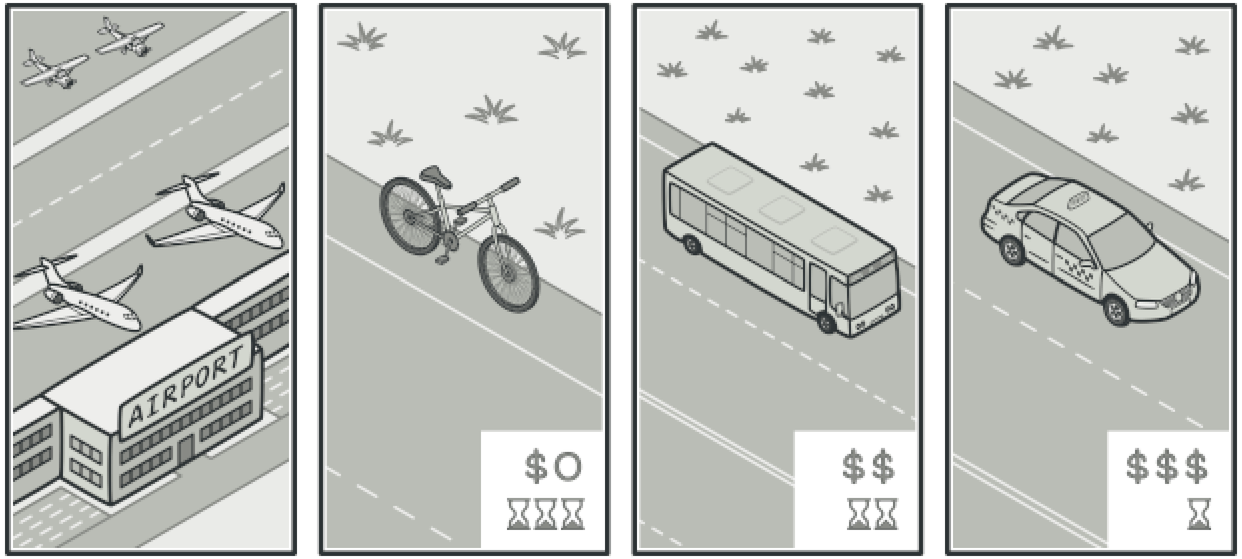
If we can not, then the Liskov substitution principle fails. This means either the classes or the code calling the classes is implemented incorrectly



Strategy

- This is a really simple pattern that embodies Liskov’s Substitution Principle
- In this pattern we create a **hierarchy of behaviours** that inherit from a base class
- We then use **composition** to be able to **swap behaviors** at run time.
- You have done this already at some stage or form
- For example, you want to get to the airport and there are many **strategies** to do so

  -
    ```
    void go_to_airport():
          transport_strategy->go()
    ```
- You can use a **car**, a **bike** or a **bus**. They all have similar behaviours with different implementations



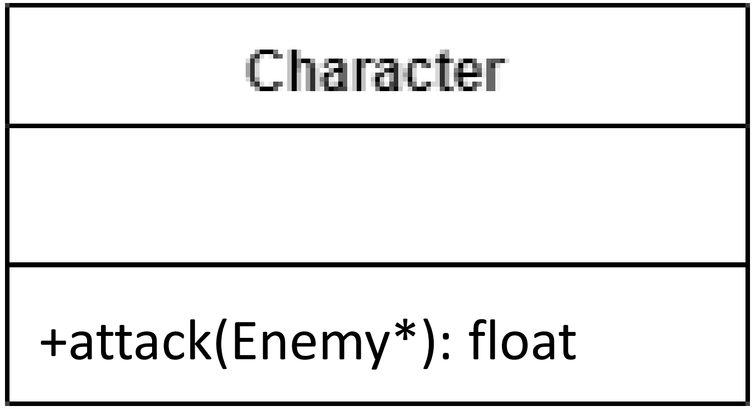
Strategy Scenario

Say you have built a game that lets you equip a character with weapons.

This weapon has a specific attack animation and damage calculations.

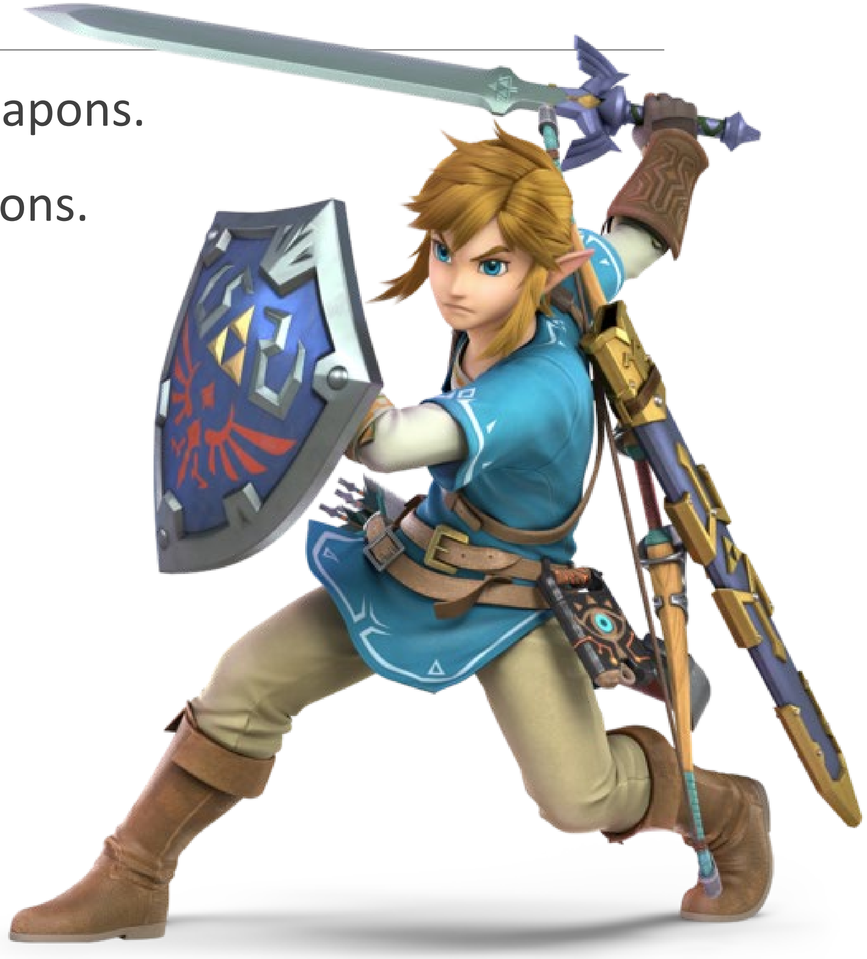
This is handled by the attack method of your character class

You make the best game in the world and it’s an instant success.



Character

+attack(Enemy*): float



Strategy Scenario

Eager to continue pleasing your fans you decide to release an update with 2 new weapons!

You modified the Character class, and added code to the attack method to contain the attack animations and damage calculations for these new weapons.

Over time you **added 5 more weapons**.

```
void attack(Enemy *enemy):
    if weapon == Weapons.SWORD:
        // complicated sword attack logic
    if weapon == Weapons.SPEAR:
        // complicated spear attack logic
    if weapon == Weapons.BOW_AND_ARROW:
        // complicated bow and arrow attack logic
    if weapon == Weapons.CANNON:
        // complicated cannon attack logic
    if weapon == Weapons.MAGIC_WAND:
        // complicated magic wand attack logic
    ...
```

Strategy Scenario

Your Character class is huge. Any change to this causes errors.

Your team of developers keep having merge conflicts with this class. One developer is working on the character movement and you are working on character attack.

Single Responsibility Principle has been violated and so has open closed principle.

```
void attack(Enemy *enemy):
    if weapon == Weapons.SWORD:
        // complicated sword attack logic
    if weapon == Weapons.SPEAR:
        // complicated spear attack logic
    if weapon == Weapons.BOW_AND_ARROW:
        // complicated bow and arrow attack logic
    if weapon == Weapons.CANNON:
        // complicated cannon attack logic
    if weapon == Weapons.MAGIC_WAND:
        // complicated magic wand attack logic
    ...
```

Strategy Scenario – Liskov Substitution Principle to the Rescue!

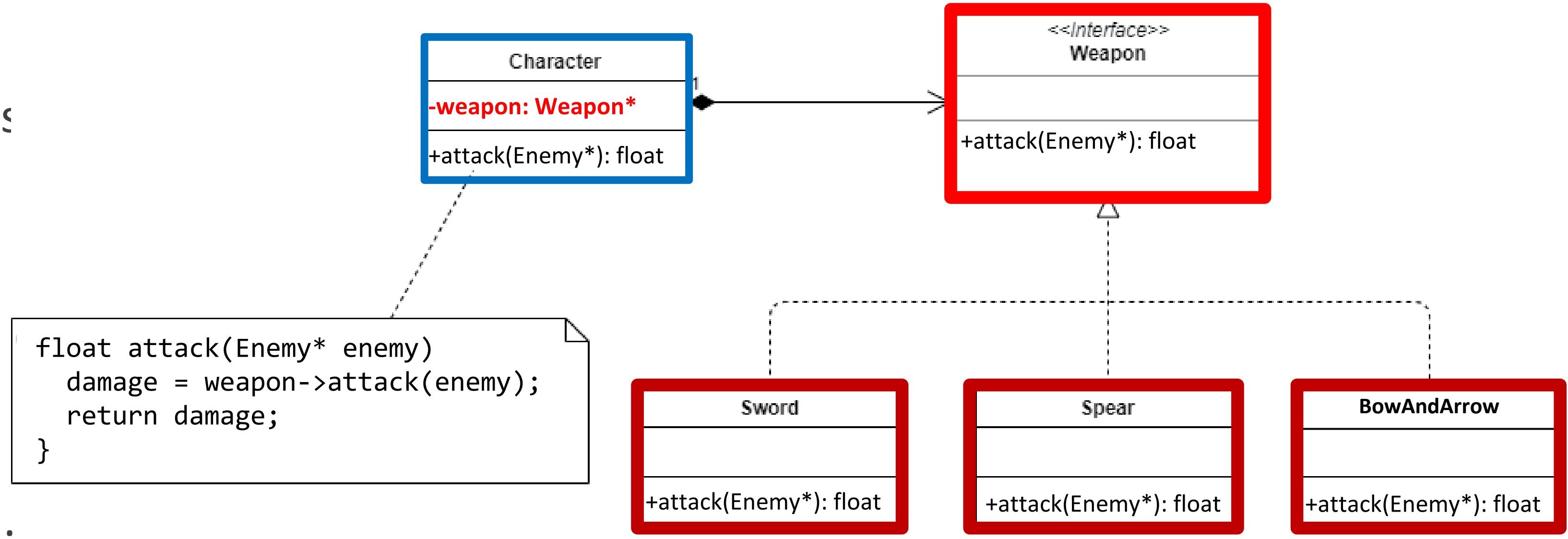
What if we used composition instead.

We can:

1) Separate the attack behaviour into its own class (**Weapon**)
Solves Single Responsibility Principle

2) Inherit from **Weapon** to implement **different kinds of weapons**.
Solves Open Closed Principle

3) Character holds a pointer to **Weapon**.
We can replace this weapon (parent) with any of the different ones (child) during run time.
Liskov Substitution Principle

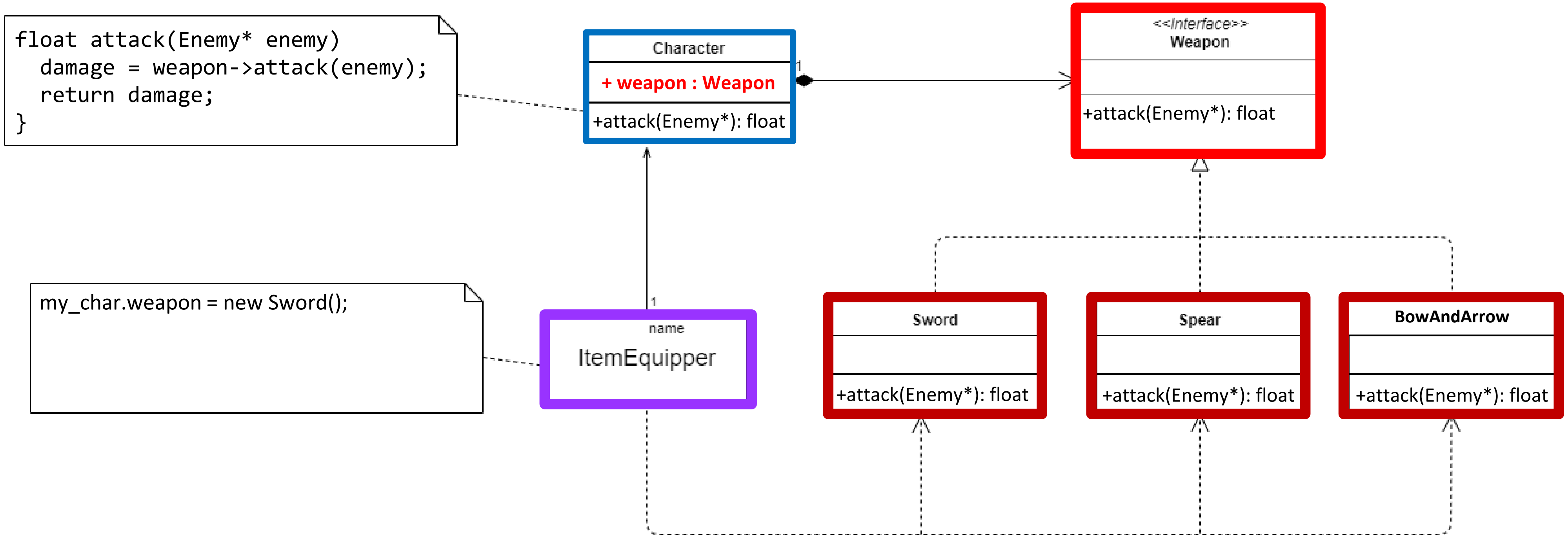


```
float attack(Enemy* enemy)
  damage = weapon->attack(enemy);
  return damage;
}
```

Strategy Scenario – How do we change the weapon?

A separate client class can change the weapon of the character during runtime.

Let's say this class was called ItemEquipper



```
float attack(Enemy* enemy)
  damage = weapon->attack(enemy);
  return damage;
}
```

```
my_char.weapon = new Sword();
```

Character

+ weapon : Weapon

+attack(Enemy*): float

<<Interface>>
Weapon

+attack(Enemy*): float

1

name

ItemEquipper

Sword

+attack(Enemy*): float

Spear

+attack(Enemy*): float

BowAndArrow

+attack(Enemy*): float

Strategy

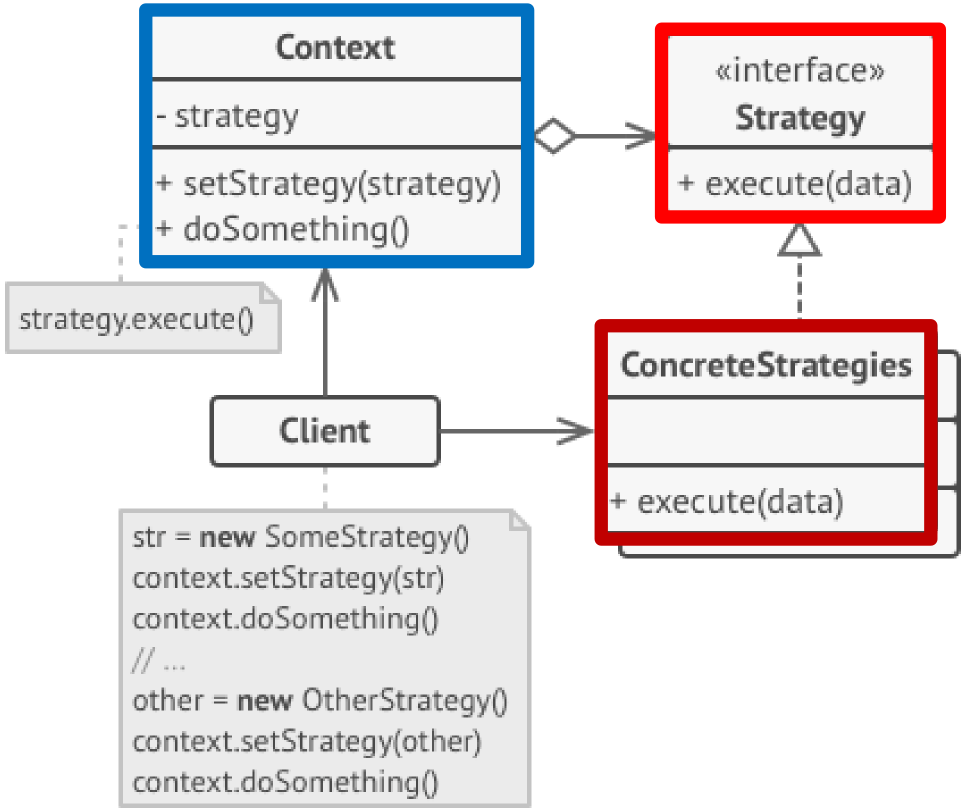
The original class that makes use of the **behaviour/strategy** is known as the **Context**.

In the previous example:

- the **context** was the character.
- the **behavior/strategy** was the weapon

This is a fairly simple pattern.

NOTE: The method(s) in the **Context** and the method(s) in the **Strategies** don't need to have the same interface.



```
str = new SomeStrategy()
context.setStrategy(str)
context.doSomething()
// ...
other = new OtherStrategy()
context.setStrategy(other)
context.doSomething()
```

strategy_sample_code.cpp

Strategy: When and Why do we use it

- When you want to use a different algorithm for different use cases. You can also swap them at run time
- Adheres to the open closed principle. We can add new strategies without modifying the context
- Replace inheritance with composition (similar to bridge pattern)
- Isolate the implementation of a strategy from the code that uses it.

Strategy– Disadvantages

- It’s overkill if you only have a few algorithms that rarely switch.
- Client needs to be aware of the different strategies to be able to select between them.

State pattern

WHEN A CLASS IS AMBITIOUS AND WANTS TO BE MANY THINGS

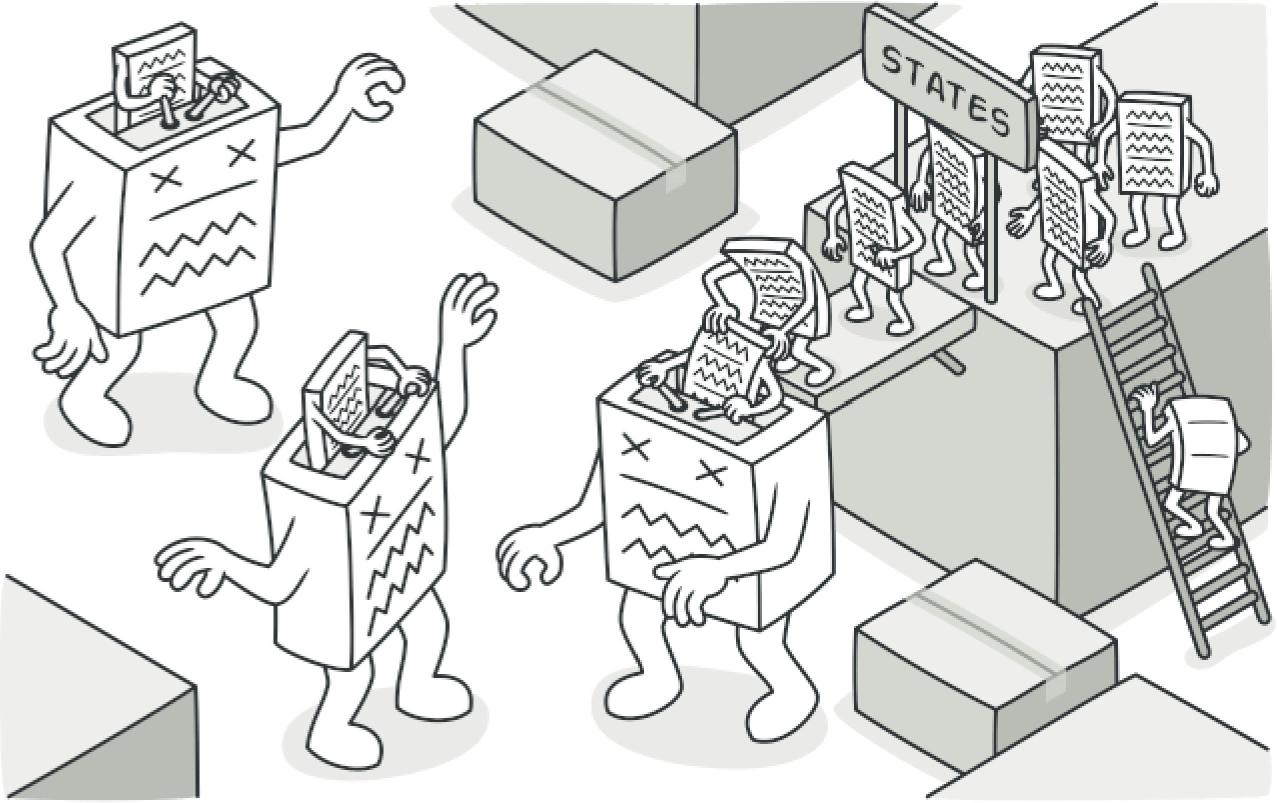
State

- A quick recap: A **state** of an object is defined as the **value and behaviours of the object at any given time** during program execution.

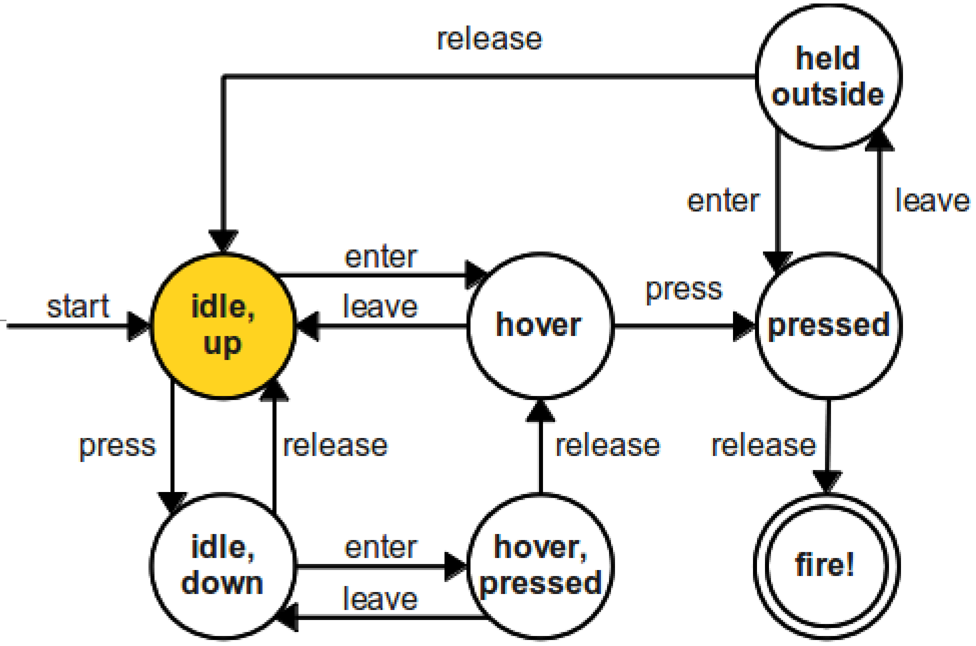
- Depending on its internal state, an object may react differently to different scenarios.

- The **state pattern** enables a **modular way to add states and control transitions**. We **decouple the state from the object**

- A pattern implemented when an object wants to alter its behaviour. It **might behave like a different class**.



A Finite State Machine



A program can be divided into a finite number of states across which its execution is tracked.

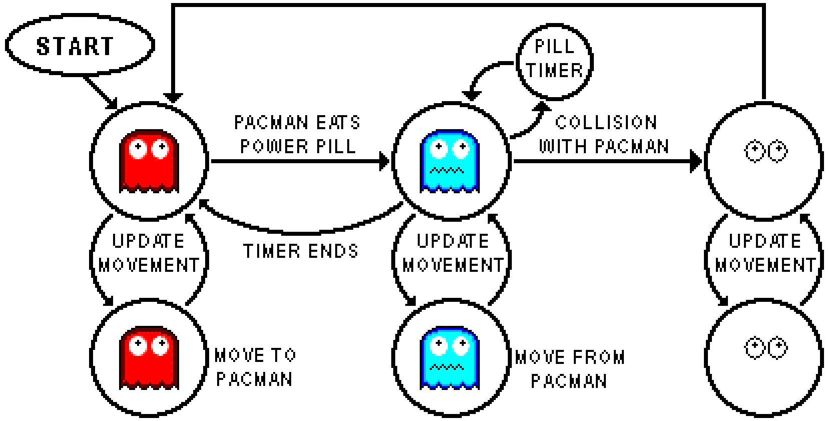
Within a state, the program behaves differently

We can change the state of a program instantly.

Moving from one state to another state is bound by rules. These “Transitions” are also finite and predetermined.

The **circles are our states**

The **arrows between them are transitions** that encapsulate rules for state switching



If you are in the mood for some *‘light’* reading: https://en.wikipedia.org/wiki/Finite-state_machine

Apply this concept to objects

An object can be in multiple states. It’s behaviour is usually determined by its state:

- **Happy State:** It wags its tail, sticks out its tongue and is playful
- **Scared State:** Tail between its legs, tongue inside it’s mouth, whimpering
- **Aggressive State**: Tail is not wagging, growling, tail is raised high, bared teeth

Dog

- **Hover State:** might change color
- **Pressed:** Might change image and how its rendered
- **Idle:** Render default image
- **Disabled:** Button is not rendered

A UI Button:

State – Naïve Implementation

A **ButtonEnum defines the different states a button can take**.

A button stores its **current state as an attribute**

We use a series of **switch statements to change behaviour**.

While this may be acceptable for simple scenarios, it isn’t maintainable in larger systems.

```
enum ButtonEnum { IDLE, HOVER, PRESSED, DISABLED };

class Button {
public:
   ButtonEnum state;
   string imgIdle, imgHover, imgPressed;
   void (*callback)();

   Button(string imgIdle, string imgHover, string imgPressed, void (*callback)()) :
state(ButtonEnum::IDLE), imgIdle(imgIdle), imgHover(imgHover), imgPressed(imgPressed),
callback(callback) {}

   void render() {
      switch (state) {
         case ButtonEnum::IDLE:
            cout << imgIdle << endl;
            break;
         case ButtonEnum::HOVER:
            cout << imgHover << endl;
            break;
         case ButtonEnum::PRESSED:
            cout << imgPressed << endl;
            break;
      } } }; //end Button class

void onButtonPressedCallback() {
   cout << "Button callback" << endl; }

int main() {
   Button button("idle","hover","pressed", onButtonPressedCallback);
   button.render();
   button.state = ButtonEnum::PRESSED;
   button.render();
   return 0;
}
```

State – Naïve Implementation

We want to be able to add and remove states without re-compiling the button class.

We want to be able to implement state-dependent behaviours easily as well.

In other words…

Making any changes to the states will affect the button class.

Button is coupled to it’s state.

We want the button to be decoupled from its state.

```
enum ButtonEnum { IDLE, HOVER, PRESSED, DISABLED };

class Button {
public:
   ButtonEnum state;
   string imgIdle, imgHover, imgPressed;
   void (*callback)();

   Button(string imgIdle, string imgHover, string imgPressed, void (*callback)()) :
state(ButtonEnum::IDLE), imgIdle(imgIdle), imgHover(imgHover), imgPressed(imgPressed),
callback(callback) {}

   void render() {
      switch (state) {
         case ButtonEnum::IDLE:
            cout << imgIdle << endl;
            break;
         case ButtonEnum::HOVER:
            cout << imgHover << endl;
            break;
         case ButtonEnum::PRESSED:
            cout << imgPressed << endl;
            break;
      } } }; //end Button class

void onButtonPressedCallback() {
   cout << "Button callback" << endl; }

int main() {
   Button button("idle","hover","pressed", onButtonPressedCallback);
   button.render();
   button.state = ButtonEnum::PRESSED;
   button.render();
   return 0;
}
```

State Pattern

We separate the **State** out into its own hierarchy.

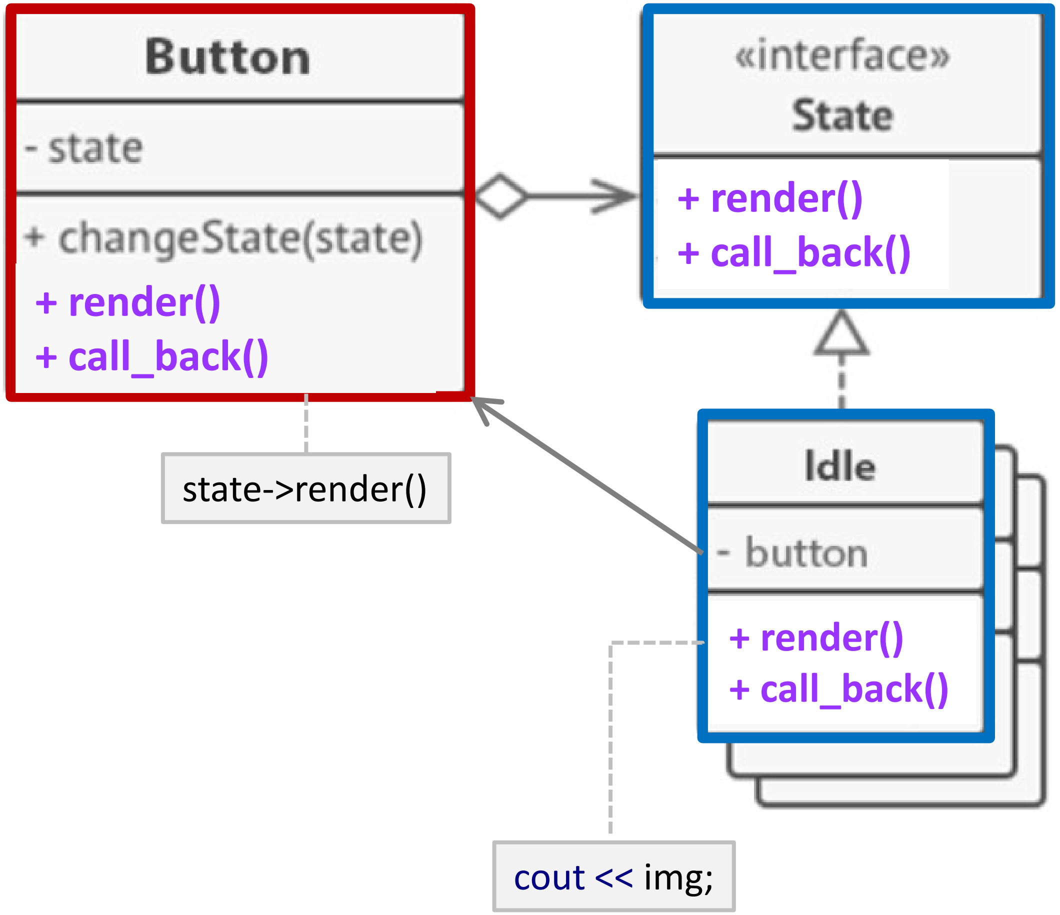
The **Button** should be dependent on a **State** interface, not the concrete states (Idle, Hover, Pressed, Disabled).

Now we can add and remove states easily.

The **State** Interface consists of all the **state-defined behaviours** in the **Button** class. This allows the Button class to communicate with it.

(In a way the Button wraps around its state)

Optionally, the state can keep a reference to the Button.



The generalized pattern

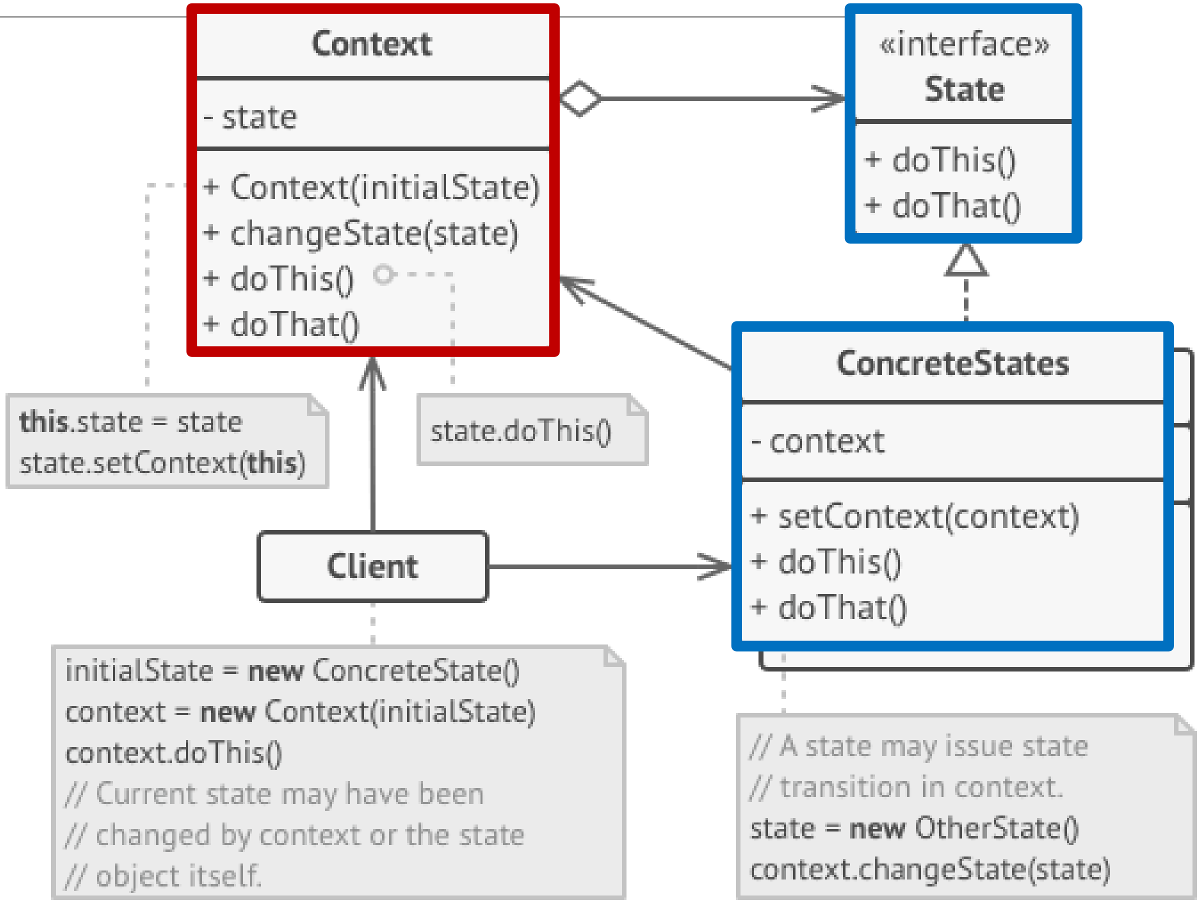
The object that has multiple **States** is called the **Context**

The **Context** has a method to change its state.

The **Context** delegates part or all of its behaviour to the State.

The concrete **state** may or may not have access to the **Context**. If the state is responsible for changing the state of the **Context** then this may be required.

The client instantiates the new State and assigns it to the **Context**.



```
this.state = state
state.setContext(this)
```

```
state.doThis()
```

```
initialState = new ConcreteState()
context = new Context(initialState)
context.doThis()
// Current state may have been
// changed by context or the state
// object itself.
```

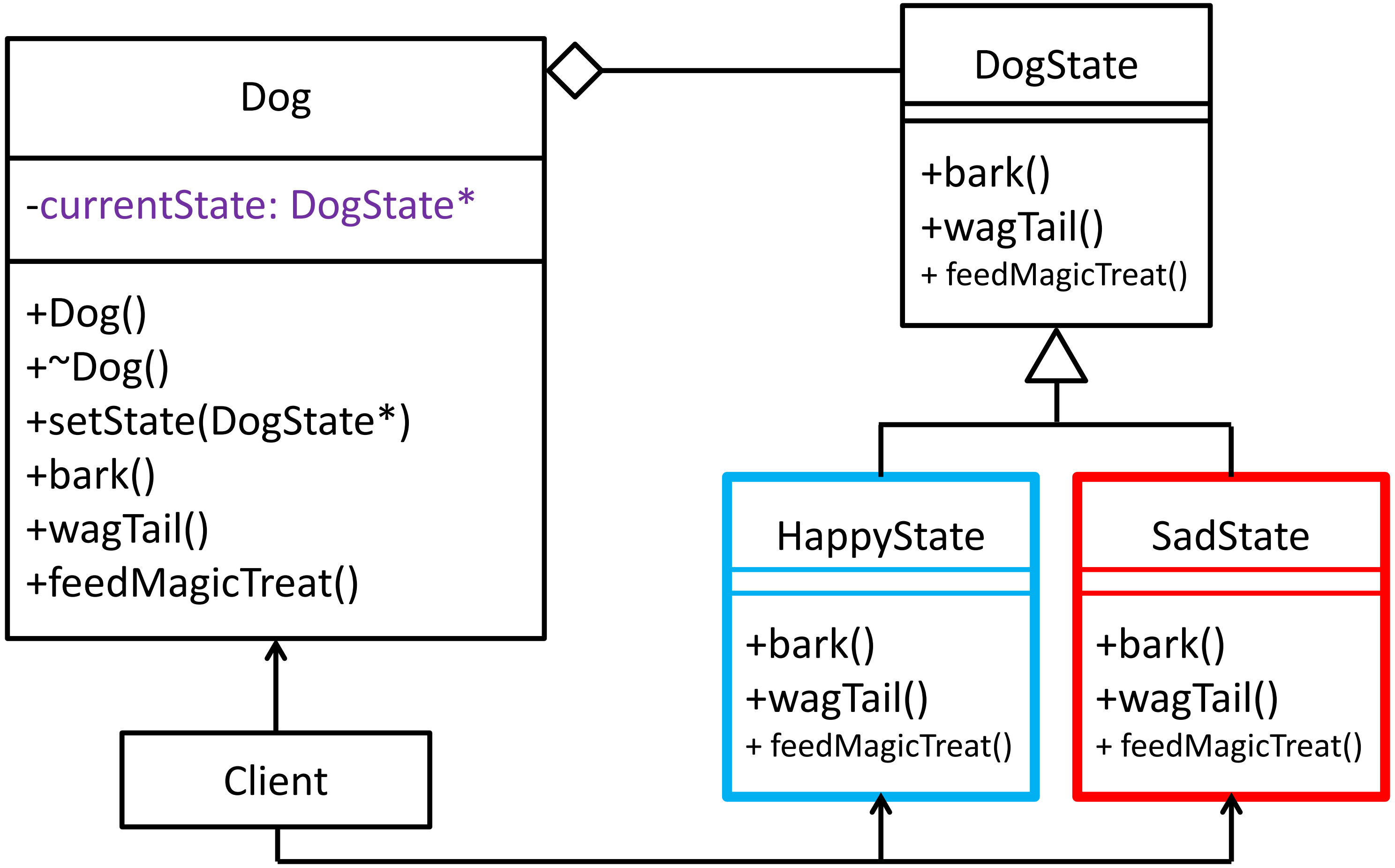
```
// A state may issue state
// transition in context.
state = new OtherState()
context.changeState(state)
```

Behavioural – State

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

Happy – happily barking, wagging tail

Sad – whimpering, tail between legs



dogState.cpp

Behavioural – State

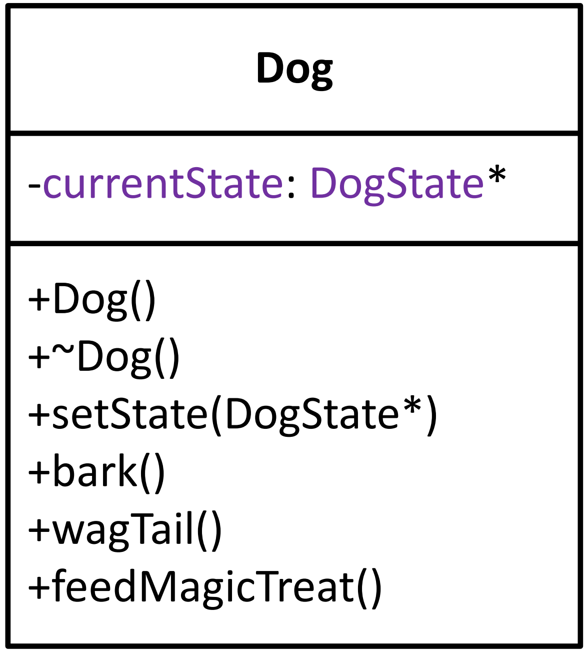
We'll demonstrate the State pattern using our Dog example. The goal over the next few slides is:

1. Create a **Dog**
2. Set the **Dog** to the HappyState
3. Make it perform some actions in the HappyState
4. Set the **Dog** to the SadState
5. Make it perform some actions in the SadState

dogState.cpp

Behavioural – State

Client instantiates **Dog** object



Dog

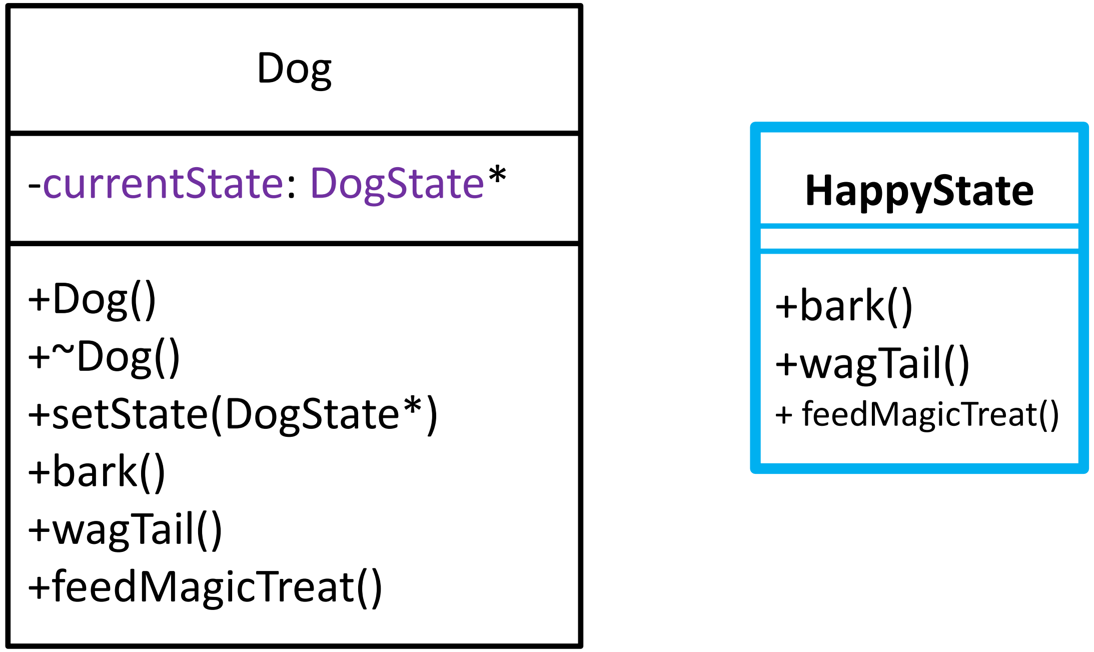
-currentState: DogState*

+Dog()
+~Dog()
+setState(DogState*)
+bark()
+wagTail()
+feedMagicTreat()

dogState.cpp

Behavioural – State

Client instantiates **HappyState** object



Dog

-currentState: DogState*

+Dog()
+~Dog()
+setState(DogState*)
+bark()
+wagTail()
+feedMagicTreat()

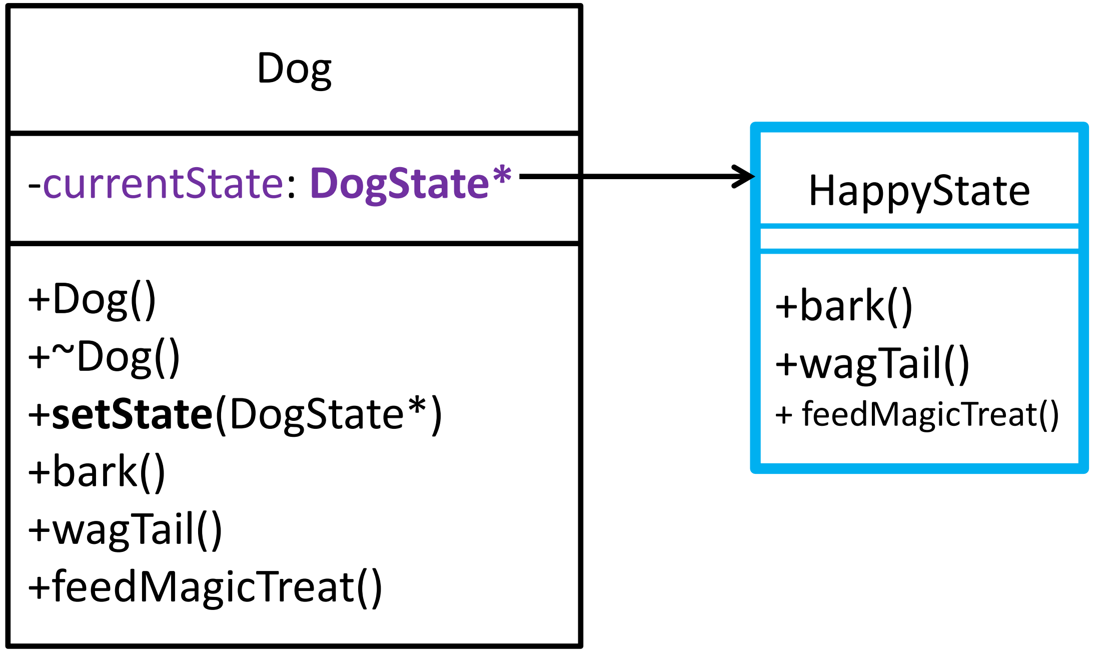
HappyState

+bark()
+wagTail()
+ feedMagicTreat()

dogState.cpp

Behavioural – State

Client sets **Dog** object's state to **HappyState** object
- Code in **setState** makes currentState pointer point to HappyState object



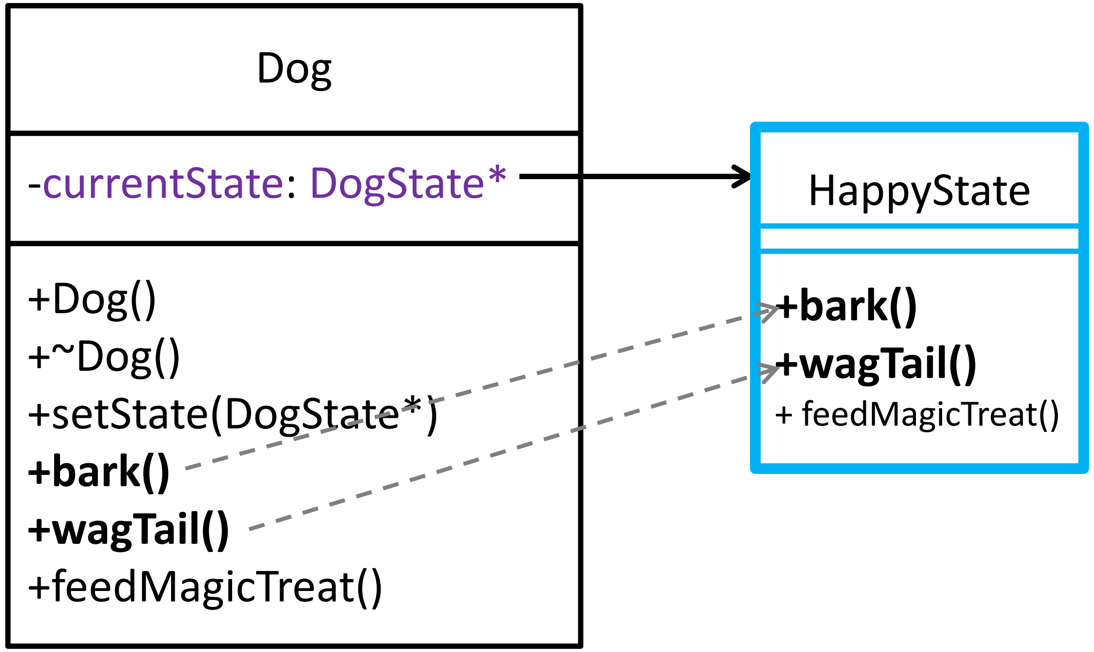
dogState.cpp

Behavioural – State

Dog is now set to HappyState :) Let's make it perform some actions

Client calls Dog's **bark()**, **wagtail()** functions

- Code in these functions uses currentState* pointer to call **HappyState**'s **bark()**, **wagtail()** functions
- Output is "Happy bark" and "Happy tail wag"



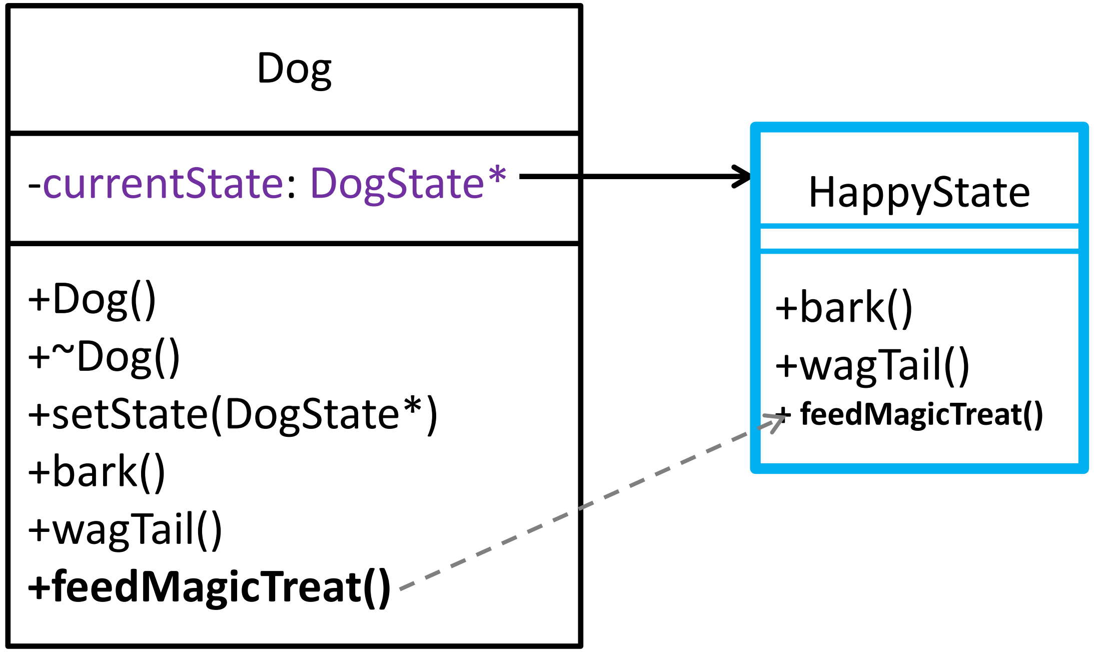
dogState.cpp

Behavioural – State

Now we want to change Dog's **HappyState** to **SadState** using the **feedMagicTreat()** function

Client calls Dog's **feedMagicTreat()**

- This calls HappyState's **feedMagicTreat()** function

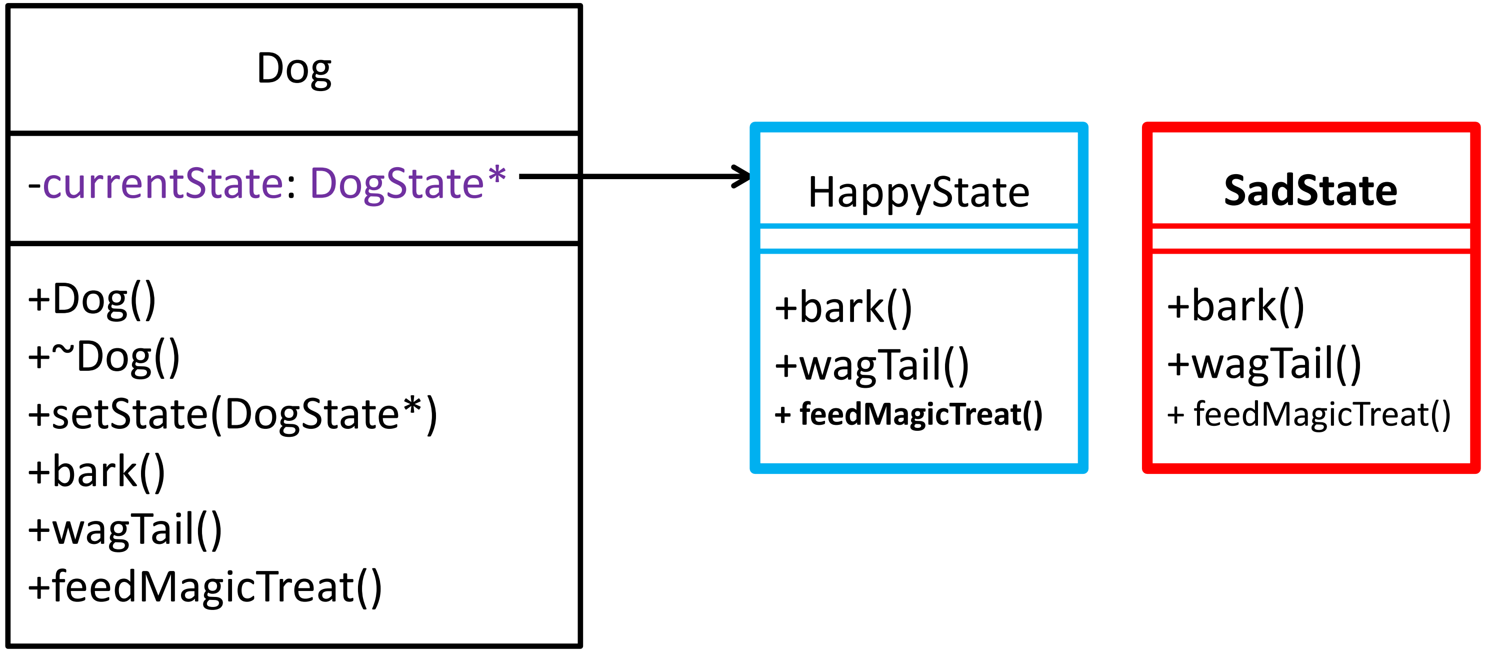


dogState.cpp

Behavioural – State

Code in HappyState's **feedMagicTreat()** function:

- Instantiates **SadState**

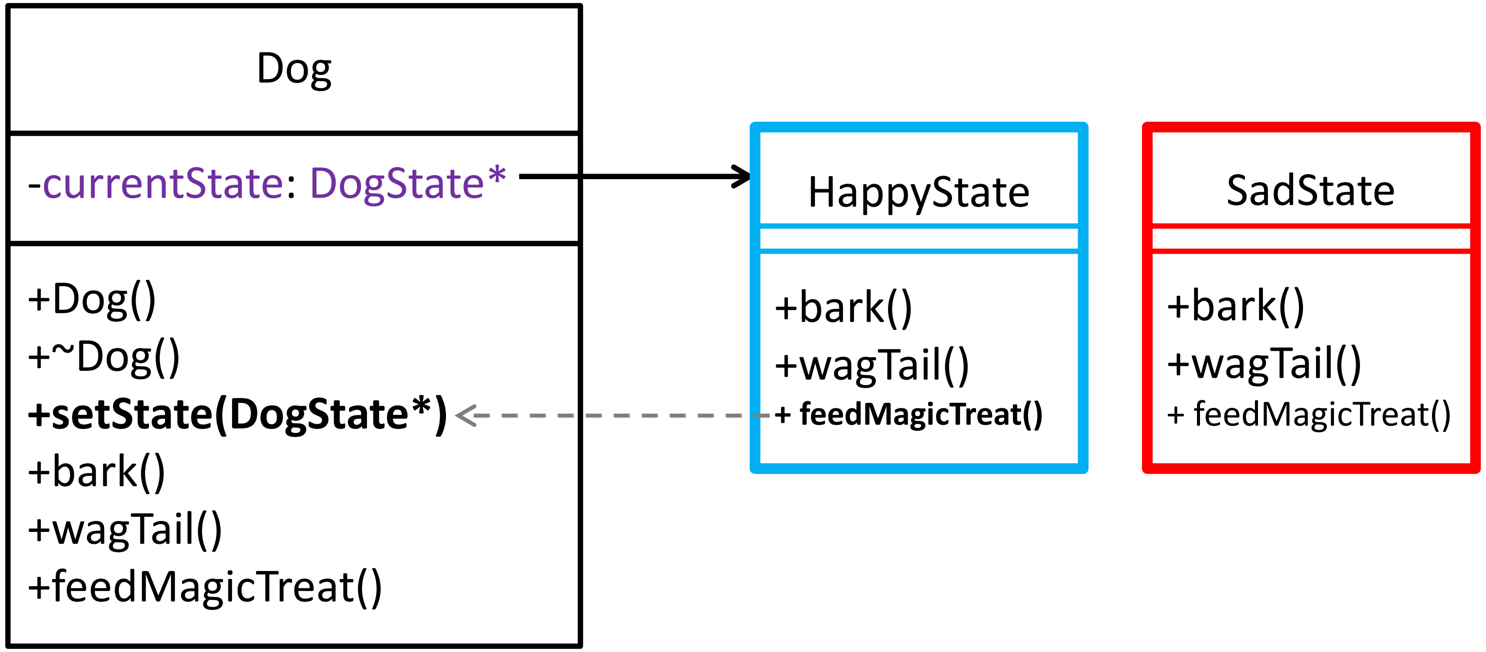


dogState.cpp

Behavioural – State

Code in HappyState's **feedMagicTreat()** function:

- Instantiates **SadState**
- Calls Dog's **setStateFunction**

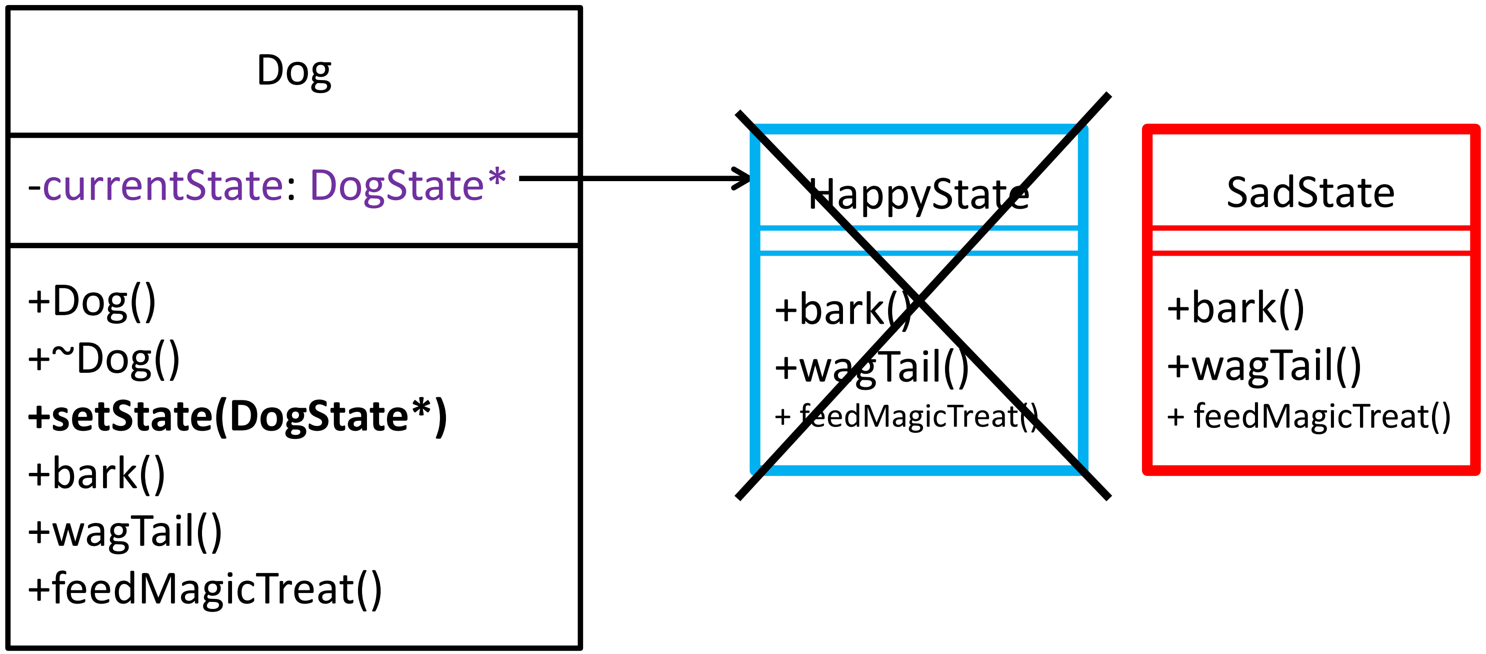


dogState.cpp

Behavioural – State

Code in Dog's **setStateFunction**:

- Deletes object currentState point at (**HappyState**)

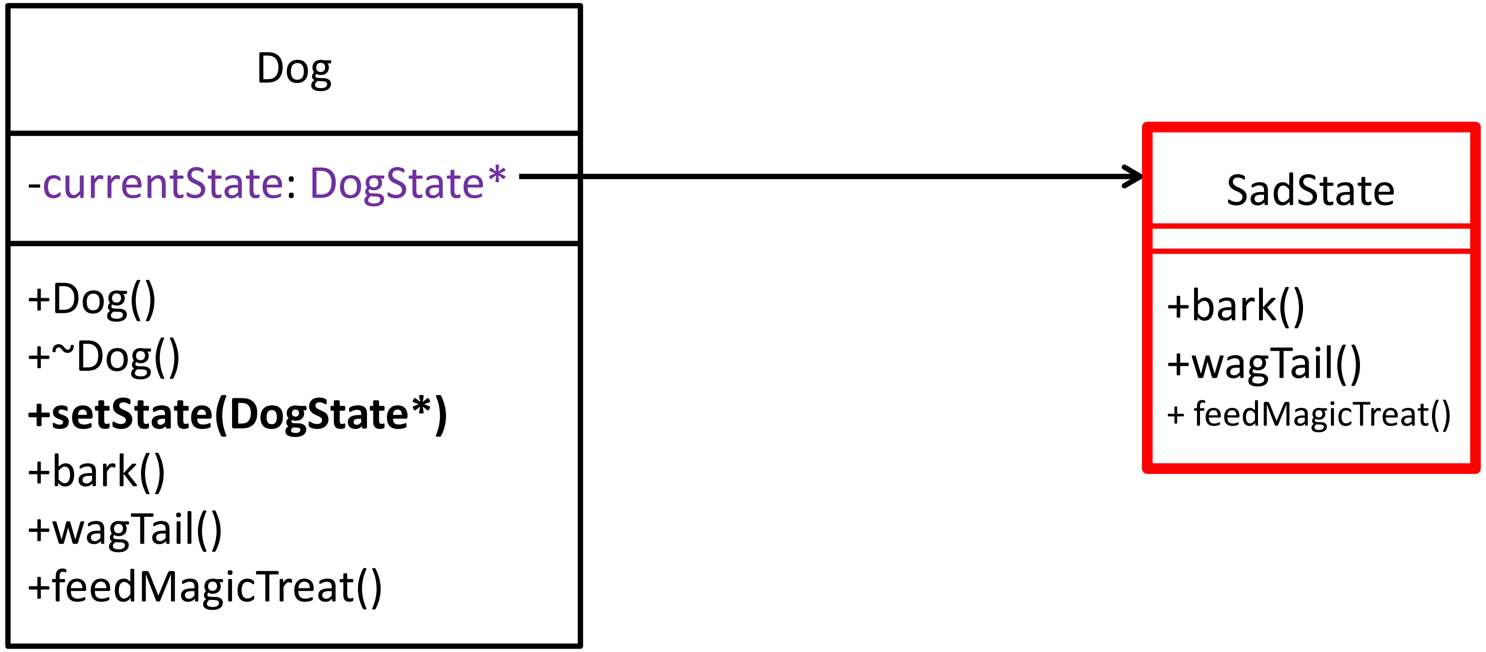


dogState.cpp

Behavioural – State

Code in Dog's **setStateFunction**:

- Deletes object currentState point at (**HappyState**)
- Sets currentState to point to **SadState**



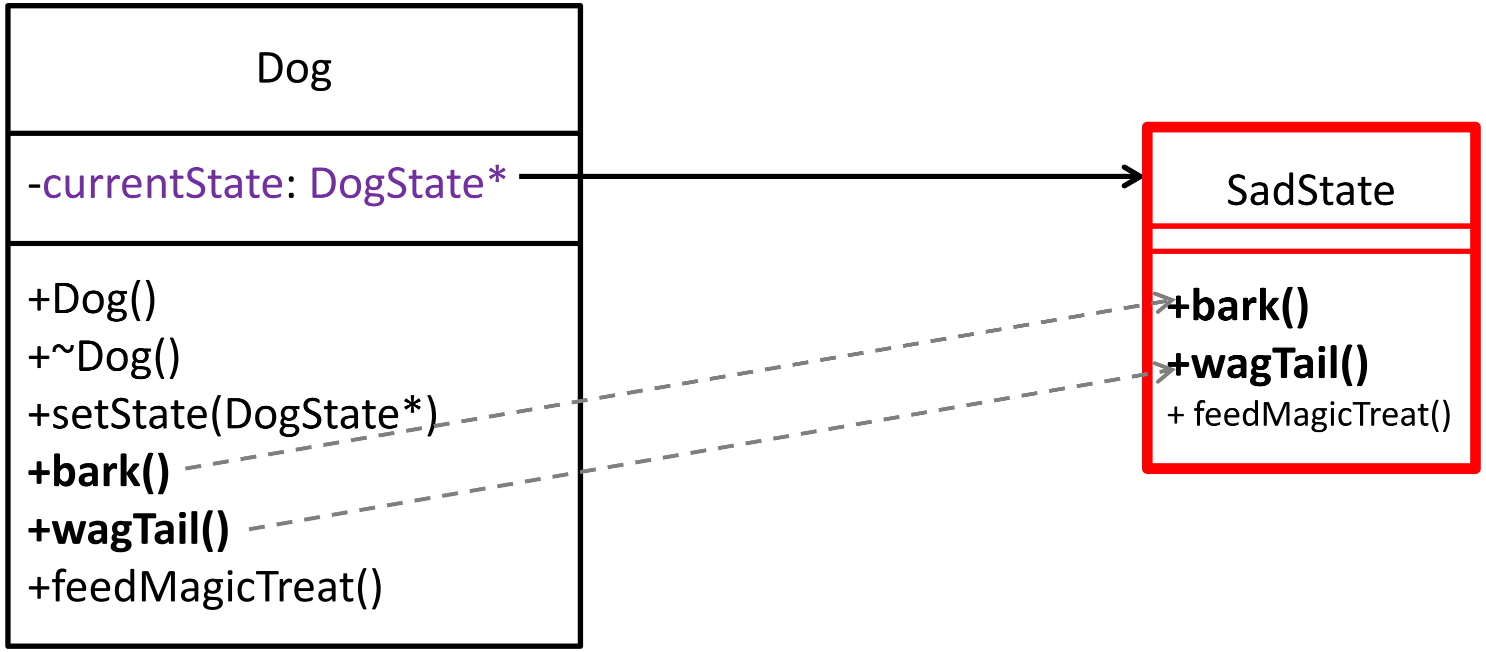
dogState.cpp

Behavioural – State

Dog is now set to **HappyState :(** Let’s make it perform some actions

Client calls Dog’s **bark()**, **wagtail()** functions

- Code in these functions uses currentState pointer to call **SadState**’s **bark()**, **wagtail()** functions
- Output is “Sad bark” and “Sad tail wag”



dogState.cpp

State: Why and When do we use it

- Use this pattern when an object’s behaviour is **heavily dependent on its states** and the number of states can be large (or you may want to add more in the future)
- Avoid massive if statement blocks to handle object behaviours based on state conditions.
- You can create **hierarchies of state** classes if there are a lot of similar **states that share code**.
- Implements the **Single Responsibility Principle**. Each state is self-contained.
- Implements the **Open/Closed Principle**. We can add and remove states without modifying the context and multiple if-else statements.

State – Disadvantages

- Can become **hard to maintain if the object rarely changes state** or only has a few states. Sometimes a simpler solution involving conditional statements works better.
- State Transitions can be complex, especially if **each state knows about its neighbouring states** (that it transitions from/into). This can make the **states highly coupled** with each other.
- There are a lot more classes and code to maintain.

How is State different from Strategy?

They solve different problems

Strategy

- Choose a strategy at run time
- Strategies **don’t know** about each other
- Context and Strategy **interfaces don’t have to be the same**
- Strategy usually does not have a reference to context

State

- Change between states often based on some rules prescribed by a finite state machine.
- States **can know** about each other
- Context and state **interfaces match**
- Can have a reference to context

Mediator

THE MIDDLE PERSON THAT TAKES CARE OF EVERYTHING SO YOU DON’T HAVE TO.

Mediator – The Problem.

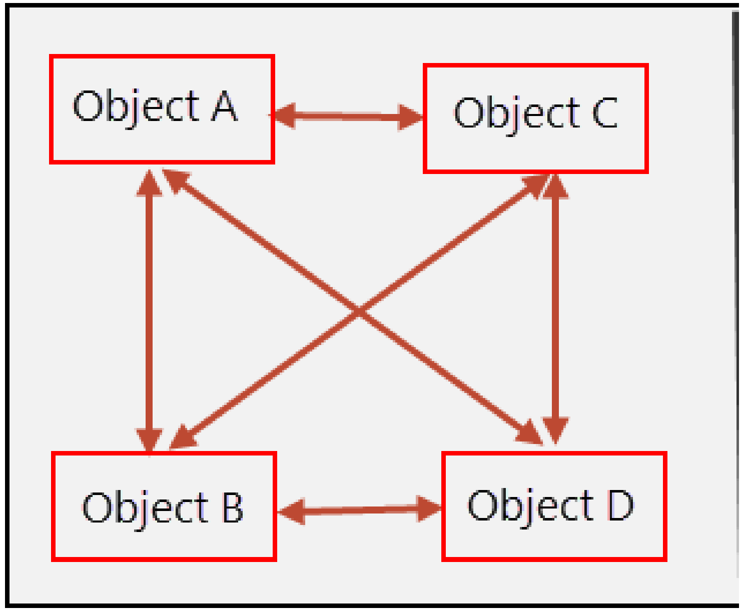
Often our program can be broken down into modular subsystems.

These subsystems are often made up of certain objects (let’s call them **components**) that talk and are dependent on each other.

This **communication** between the components facilitates the intended behaviour of the system.

But what happens when this flow of communication becomes **unregulated** and difficult to follow/debug?

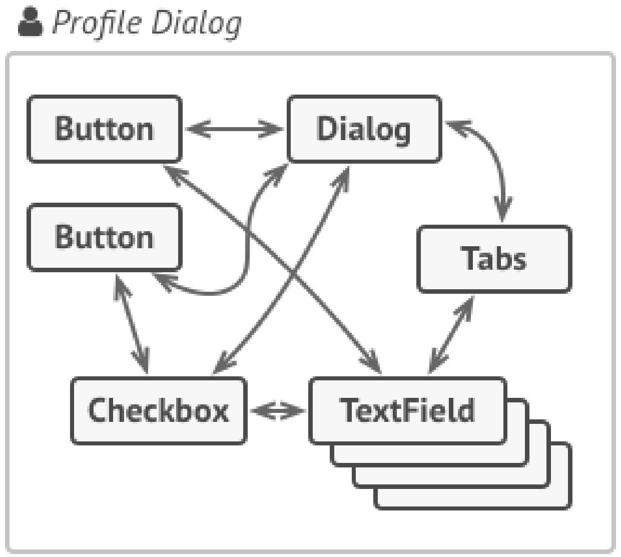
The system becomes **highly coupled** and difficult to maintain.



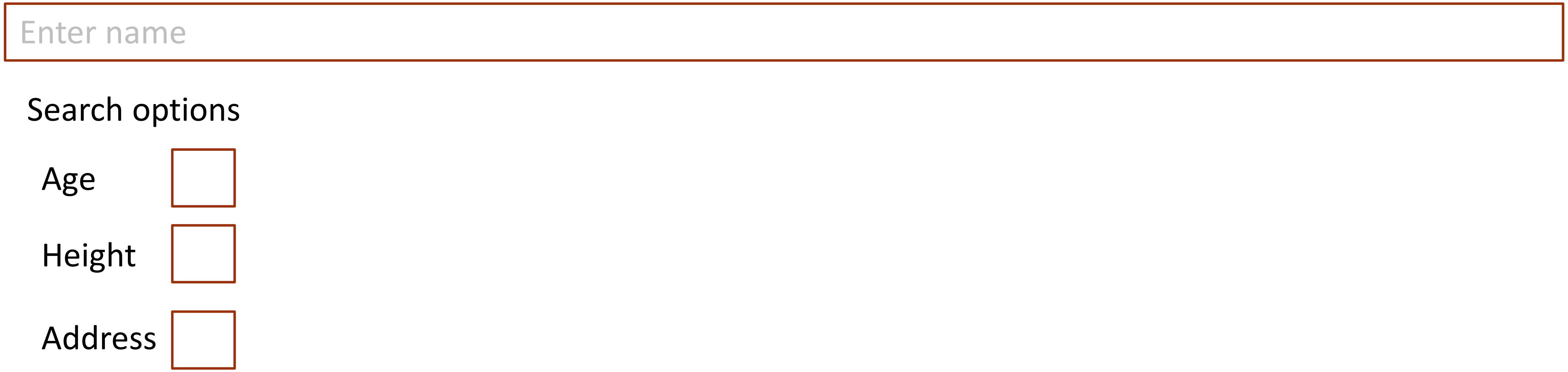
Mediator – An Example

We often see this in UI systems.

Within a certain **context** (a dialog box, or a screen for example) there are a number of different components that interact with each other.



Mediator – An Example
Person finder



Enter name

Search options

Age □

Height □

Address □

This is a search window to find people, with a number of filters represented as checkboxes. Selecting a checkbox may require that an additional text field be displayed.

Mediator – An Example
Person finder

Jeff

Search options

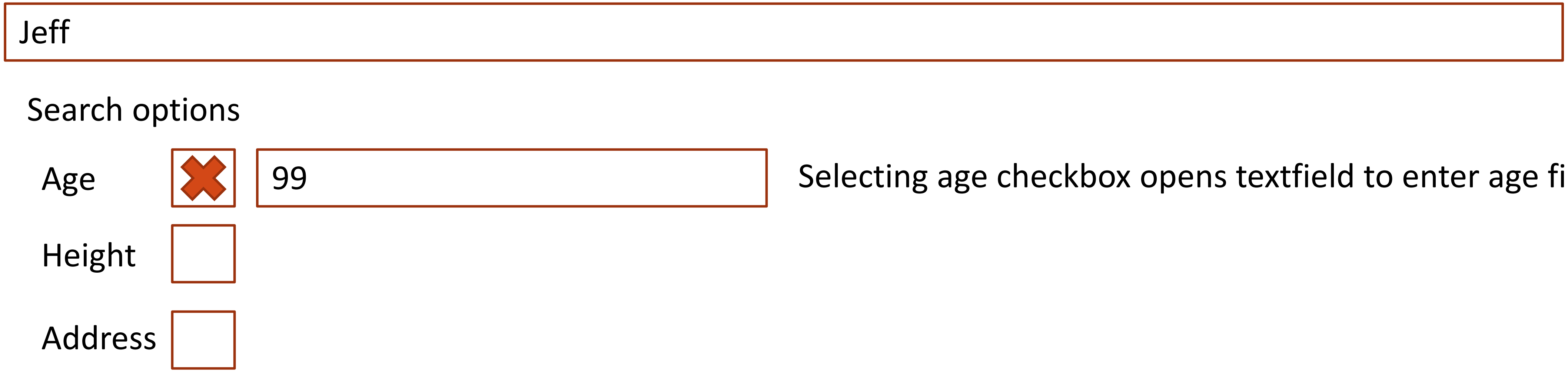
Age □

Height □

Address □

Entered **name** into search, but want to filter by age and height

Mediator – An Example
Person finder



Jeff

Search options

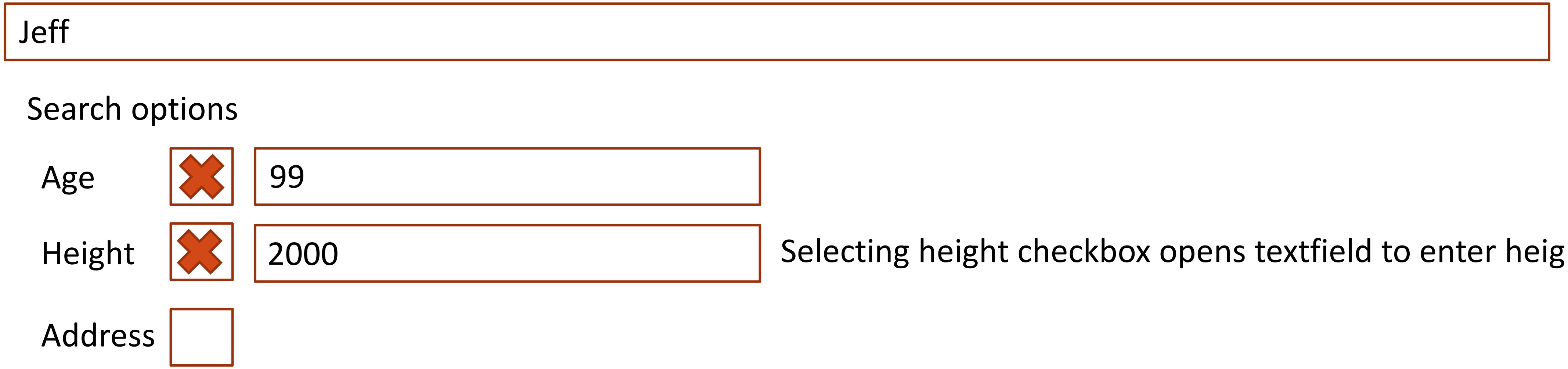
Age ☒ 99

Height □

Address □

Selecting age checkbox opens textfield to enter age filter

Mediator – An Example
Person finder



Jeff

Search options

Age ☒ 99

Height ☒ 2000

Address □

Selecting height checkbox opens textfield to enter height filter

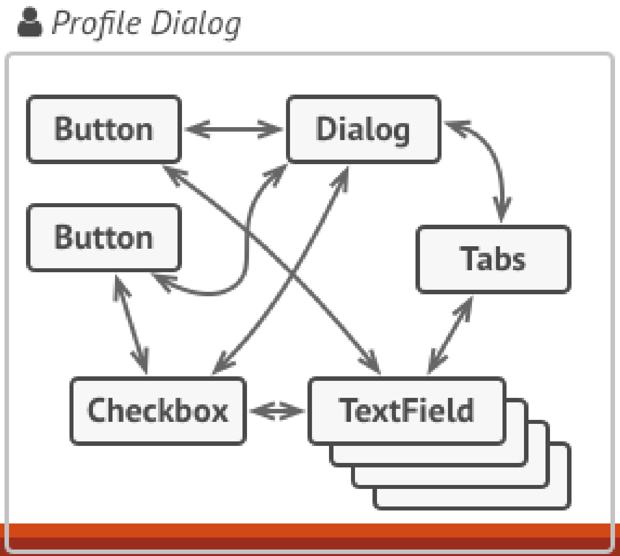
Mediator – An Example

In the previous slides we saw a search window with a number of filters represented as checkboxes. Selecting a **checkbox** may require that an **additional text field** be displayed.

Below is bad code, it makes it difficult to re-use the **Checkbox** in any other screen. Breaks so many SOLID Principles.

Inheriting from the Checkbox class to satisfy this one screen would also be extremely unnecessary and not maintainable.

```
class Checkbox {
        ..
        ..
if (selected)
        age_text_field->enabled = true
        ..
        ..
}
```

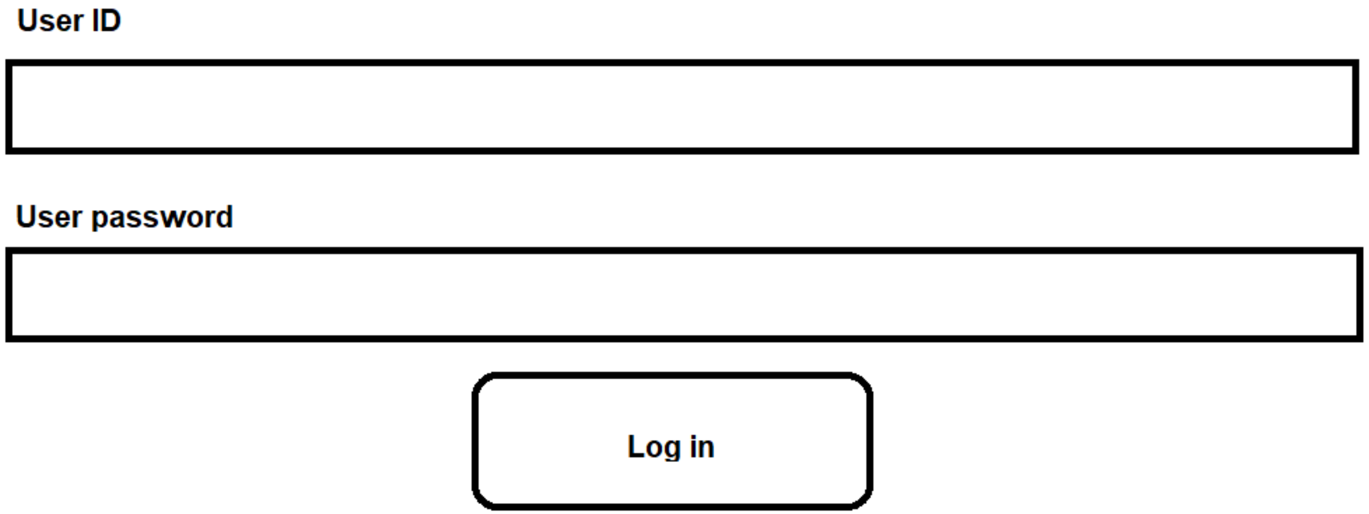


Mediator – An Example

Let’s examine how we may implement the following log in screen.

Notice there are many components:

1. TextLabels
2. Inputfields
    1. User ID field
    2. User password field
3. Screens
4. LogInButton



mediator_problem.cpp

Mediator – The solution

The Mediator pattern **reduces the dependencies** between a large number of components of a system.

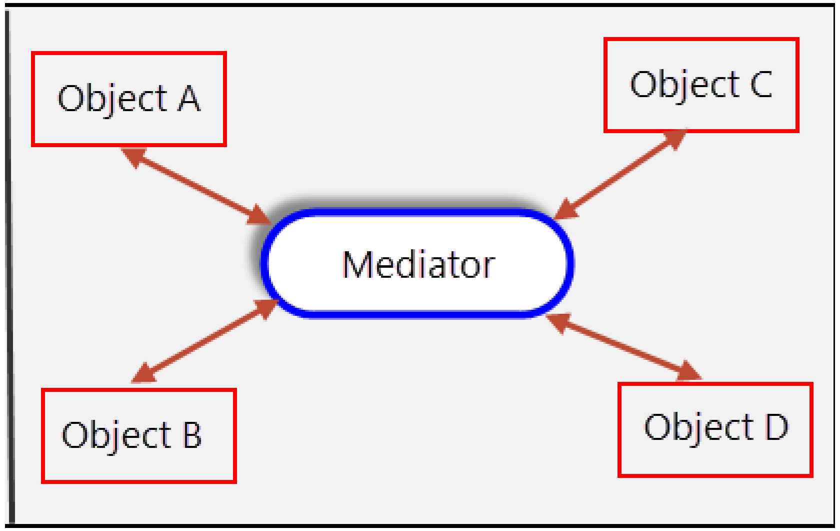
It **restricts communications** between the different components by introducing a middleperson that '***Mediates'*** the communication between them.

This is very similar to the concept of **Controller classes** that you often come across in code.

The Mediator class/entity can be thought of as a formalized structured controller.

The components of a system are now unaware of each other and **only depend on a Mediator**.

This relationship is **bidirectional**.



https://www.123rf.com/photo_59304312_traffic-road-traffic-controller-control-transportation-on-street-cop-pointsman-vector-illustration.html

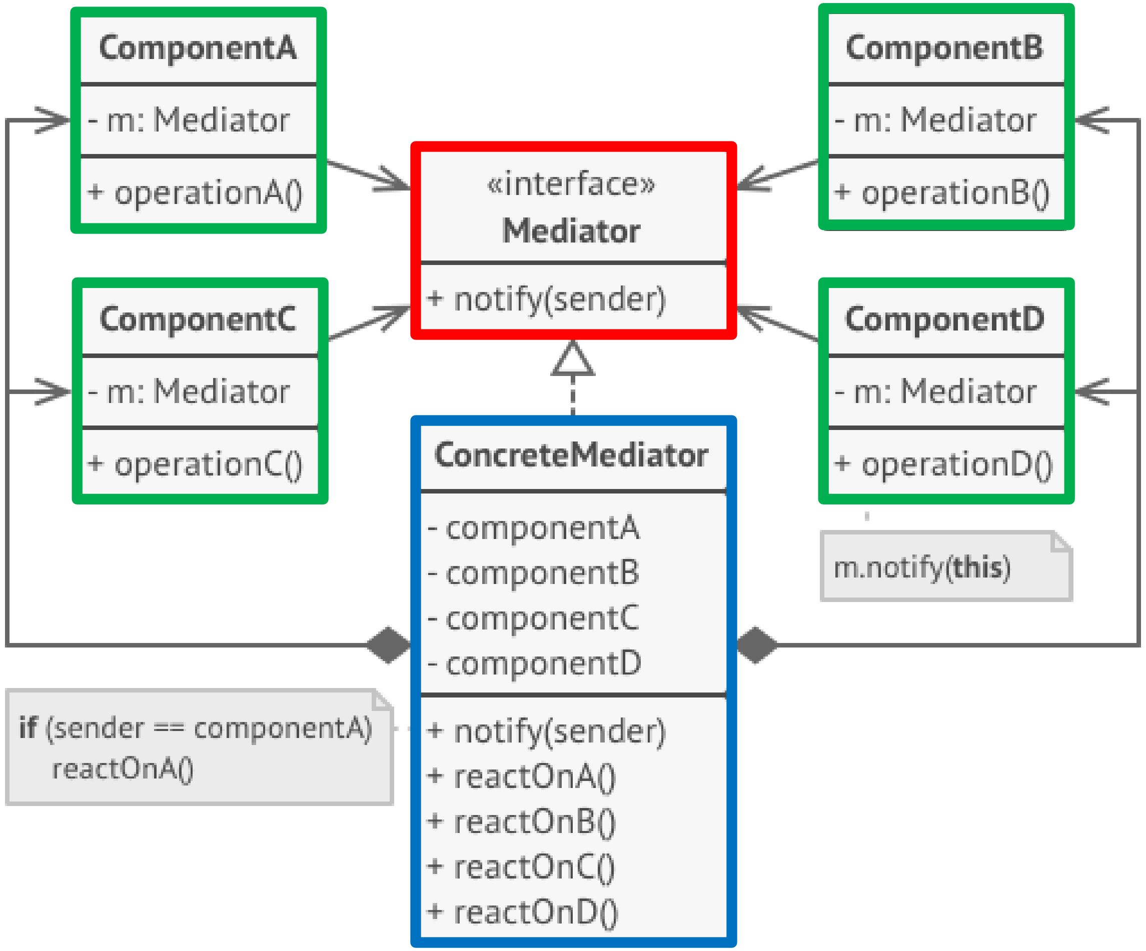
Mediator UML

All Mediators in a system can implement a **common interface**.

Each context or subsystem would have its own **concrete mediator**

By having a common mediator, we could **re-use the components across contexts**.

For example, a button can be used with a LogInMediator and a RegisterUserMediator.



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

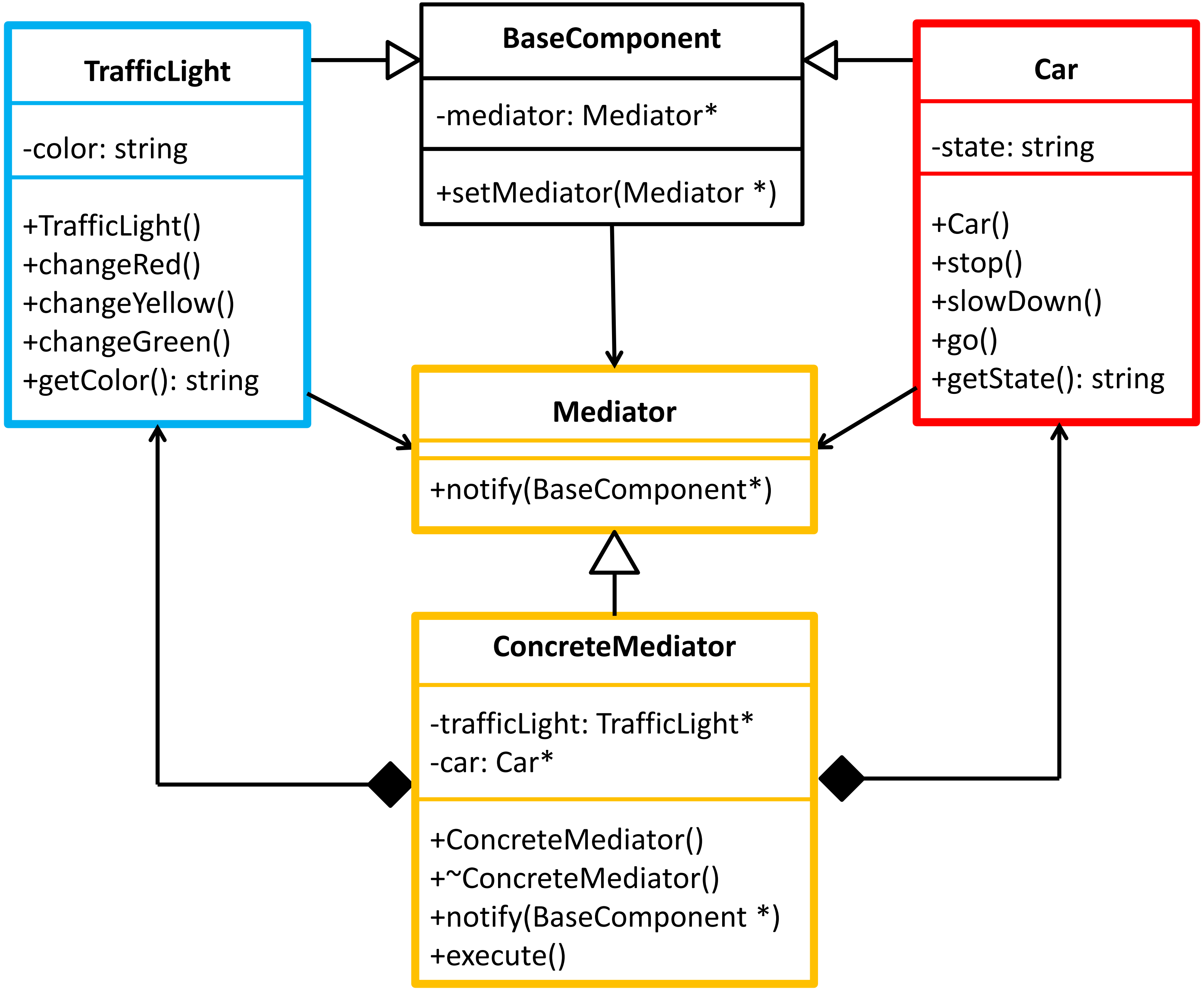
- slow down on a yellow light

- 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



Some important tidbits.

Mediators don’t necessarily have to only be for UI systems

They can be used in other systems where we need objects or subsystems to talk to each other while being decoupled and unaware of each other.

Similar to the Observer Pattern, Mediators can cause **different parts of a system to react to a component**.

That being said, it is different from an observer pattern.

- **Mediators** facilitate **bidirectional communication**.
- An **observer** is more of a **broadcast system** that gets triggered when the state of the observed object changes.

Mediator: Why and When do we use it

- A Mediator should be introduced when the **communication** patterns between different objects **becomes chaotic and highly coupled**.
- Use this pattern when you want a component to be **unaware of other components** but **still be able to communicate** with them.
- Implements the Single Responsibility principle. The **responsibility to facilitate communication between objects is extracted into one class**.
- Implements the Open/Closed Principle. We can introduce **new mediators for new contexts without changing the components**.

Mediator – Disadvantages

- The Mediator itself becomes a complex object and an **epicentre for coupling**.

Check out God Objects, this is what is known as an *Anti-Pattern.*

a **God object** is an object that *knows too much* or *does too much*

https://en.wikipedia.org/wiki/God_object

Categorizing Design Patterns

- ❑ **Behavioural**

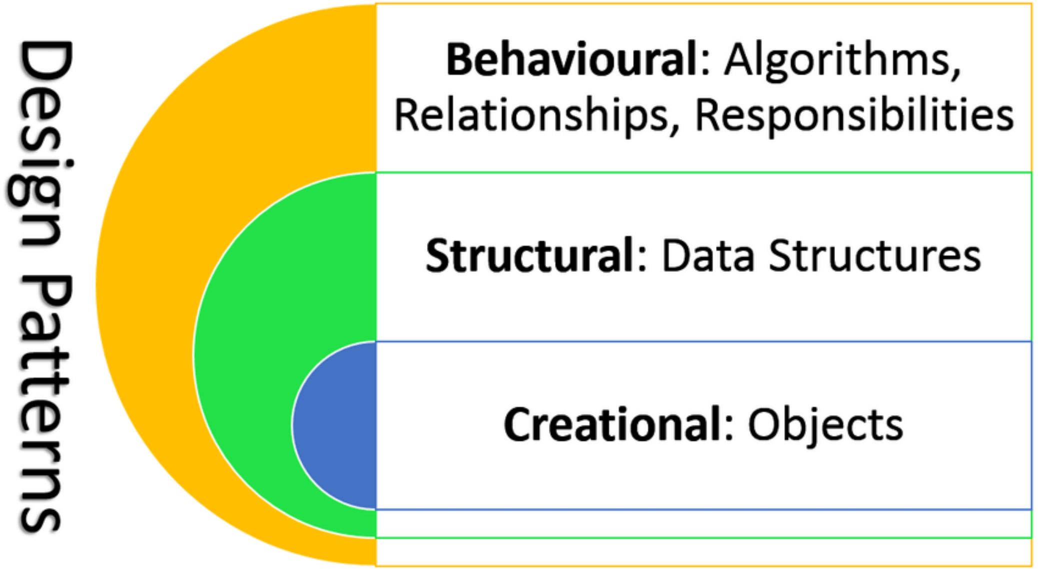
Focused on communication and interaction between objects. How do we get objects talking to each other while minimizing coupling?

- ❑ **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

Wrapper

Previous 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.

What is a wrapper?

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

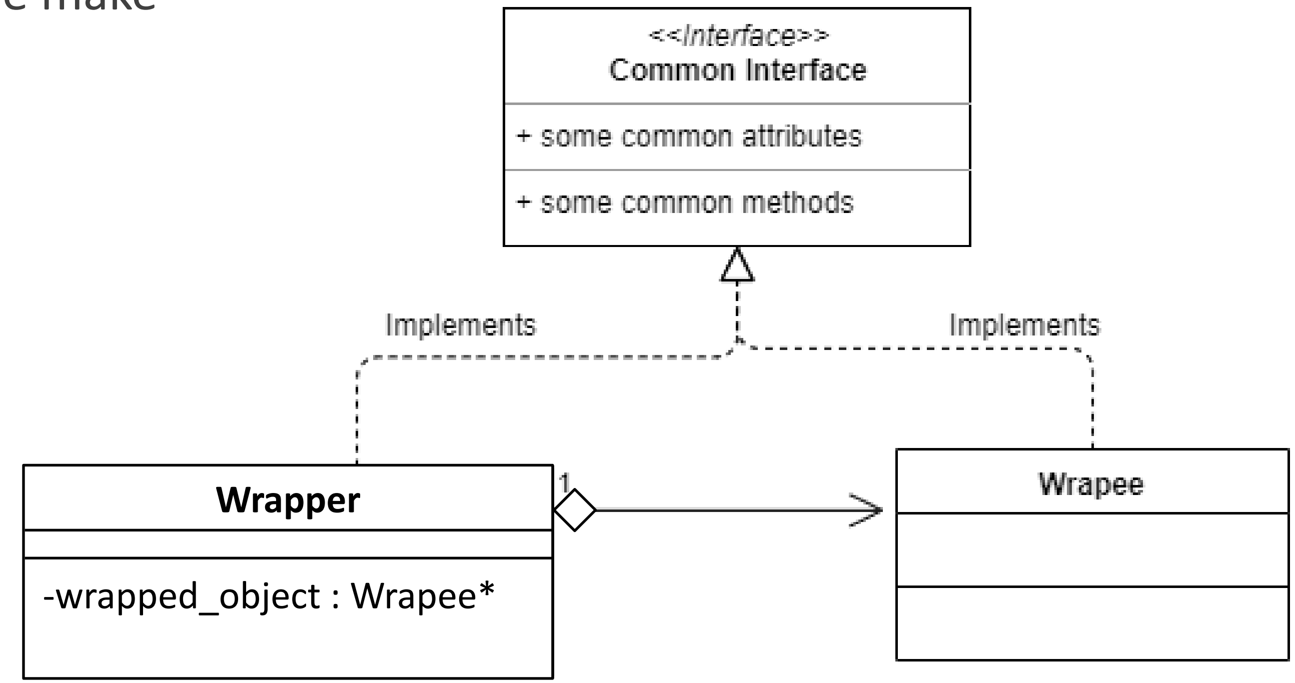
The wrapper has the same interface as the 'wrapee'.

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.

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?

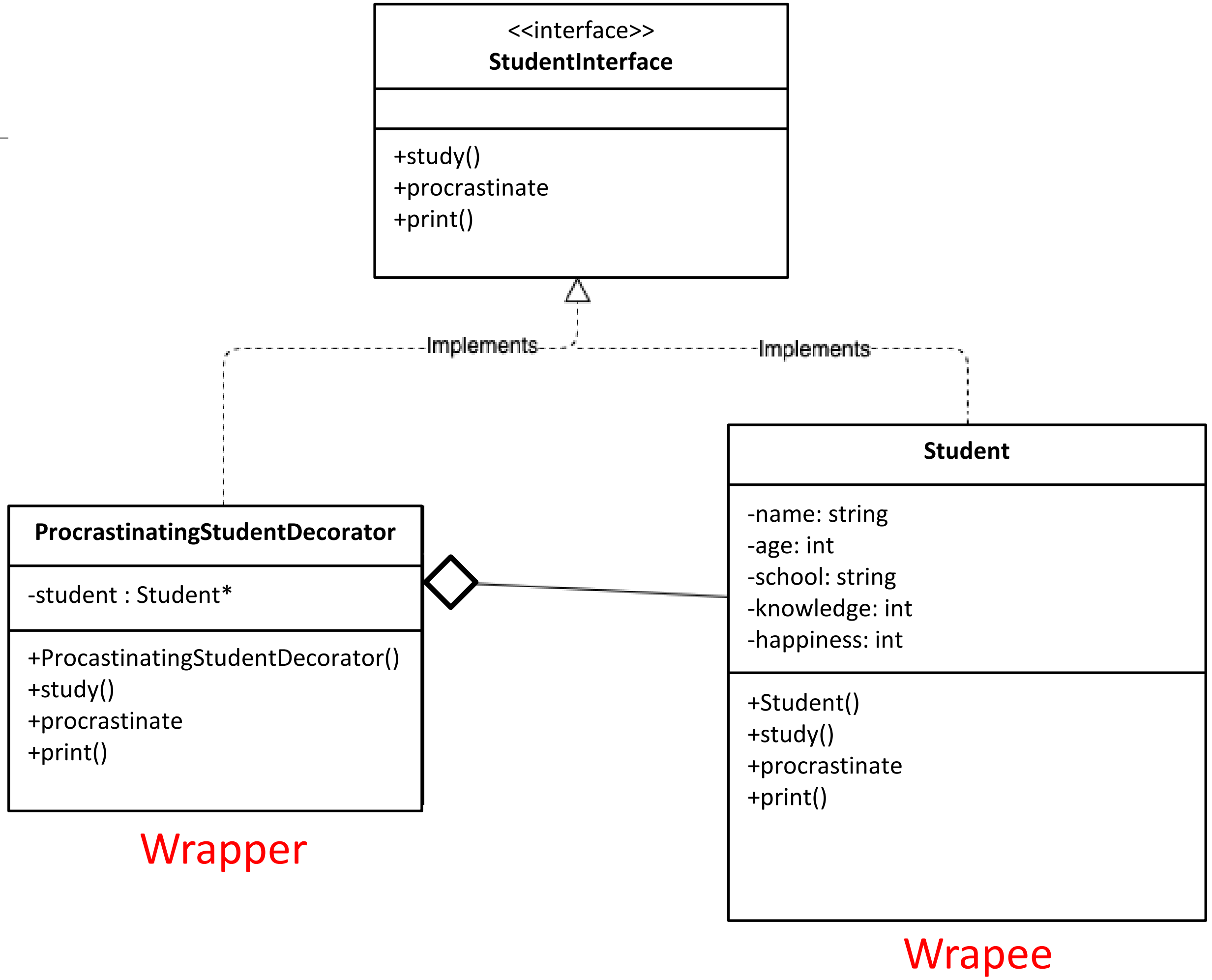
Student is the wrapee

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

Wrapper.cpp

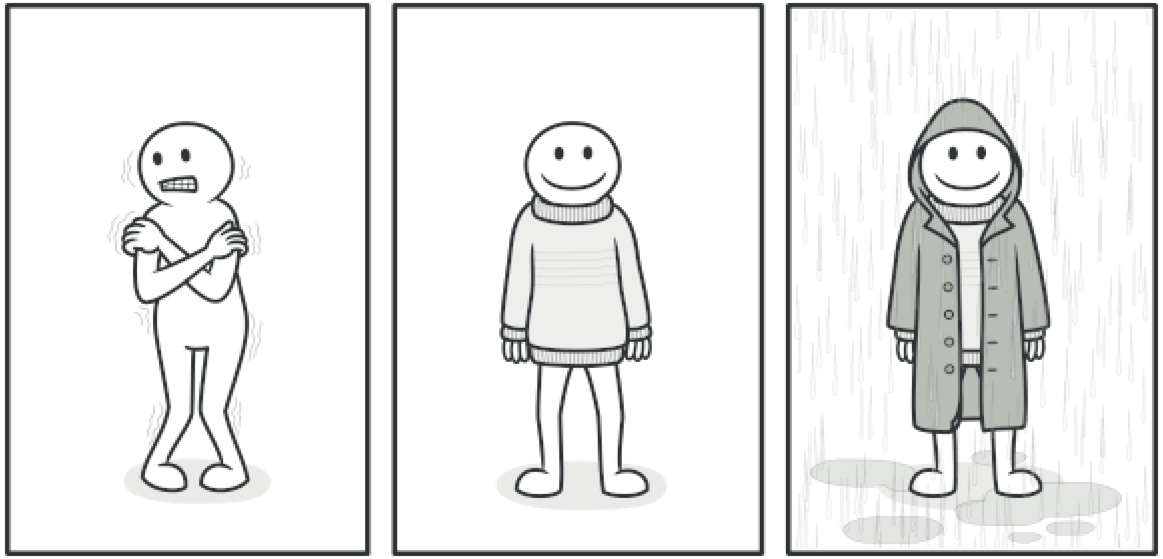


Wearing many layers (Multiple Wrappers)

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. For instance in the **Decorator pattern**



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:

Donut shop menu

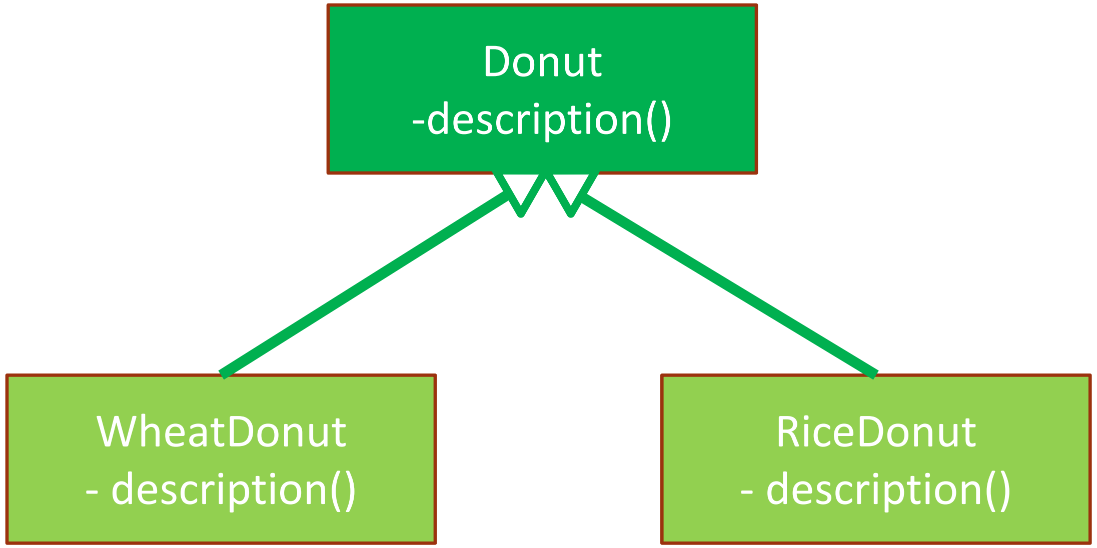
A different class for each donut, they all share the same base class and interface but come in infinite combinations and varieties.

You sell two different types of donuts

- Wheat donut
- Rice donut

Introduction - Problem

- You might model it to look something like this

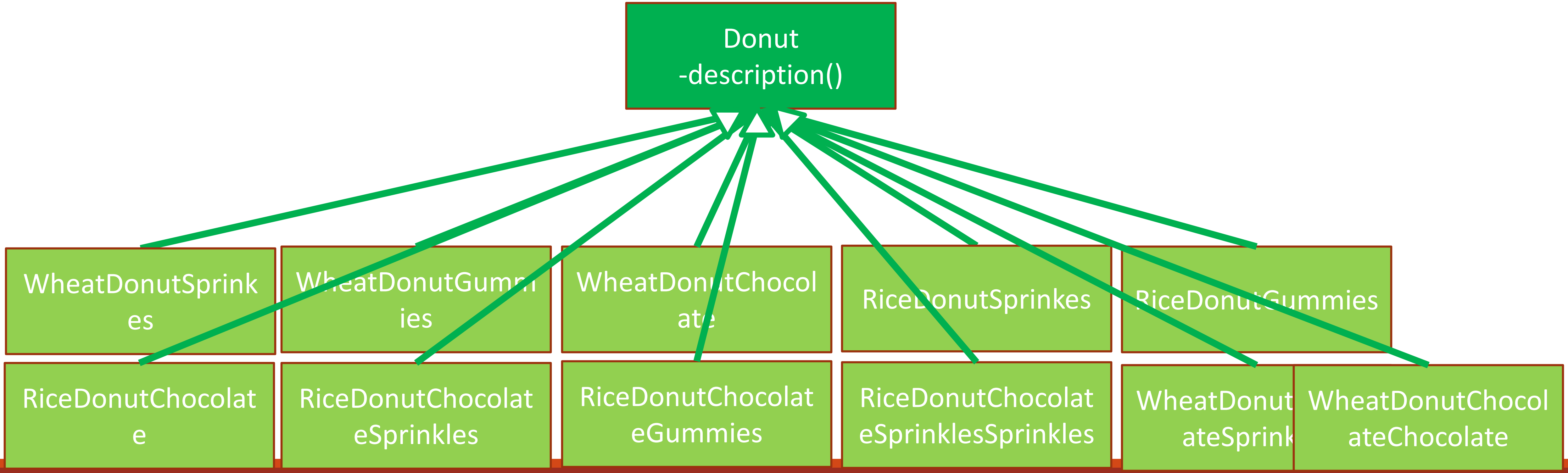


Introduction - Problem

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

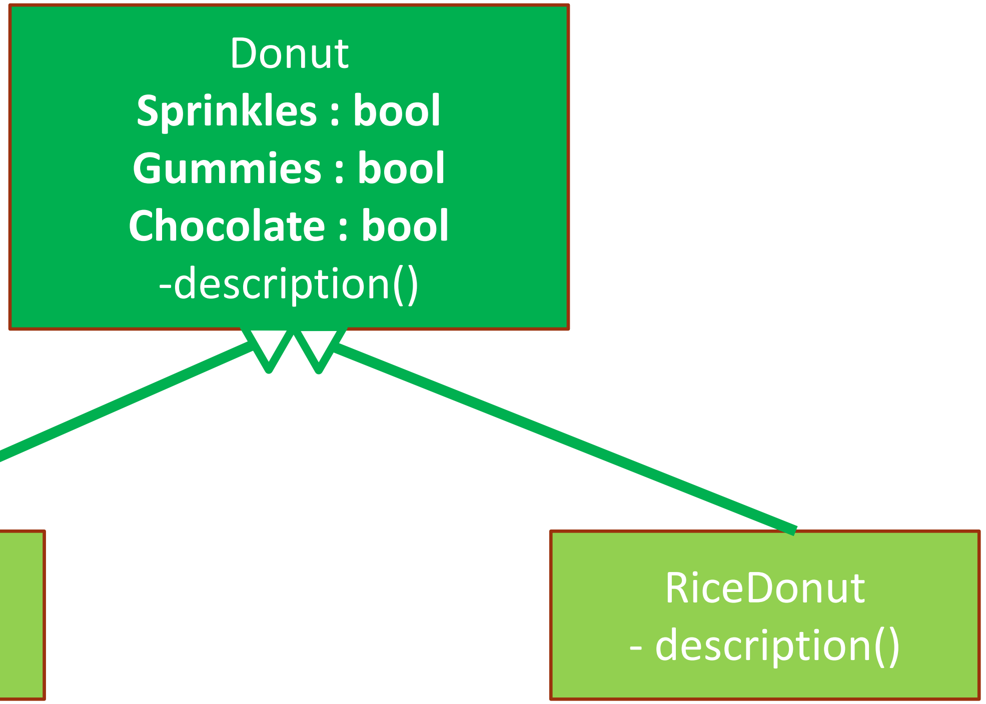
Introduction - Problem

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



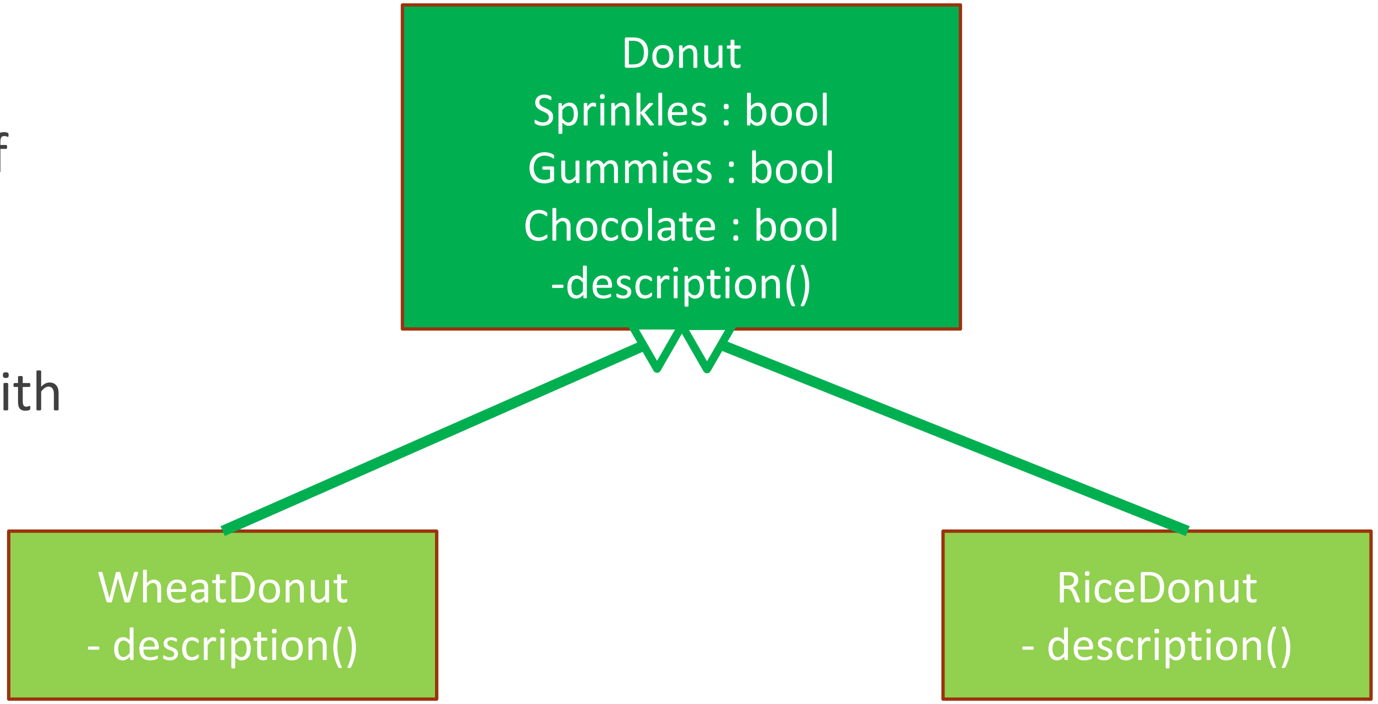
Introduction - Problem

- We can move the **toppings** to the Donut instead



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

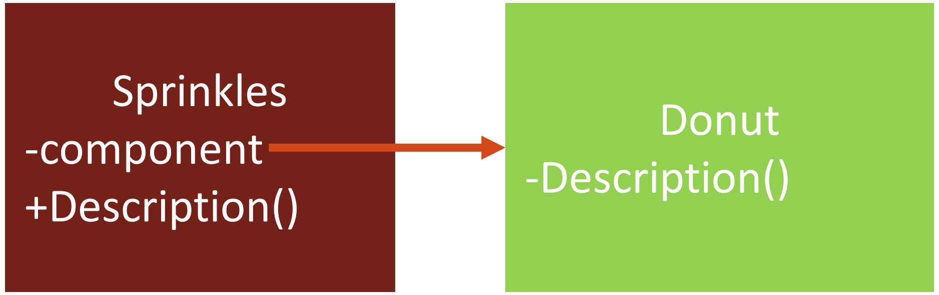
- Let’s try something different
- Let’s start with a **donut** object

Donut

-Description()

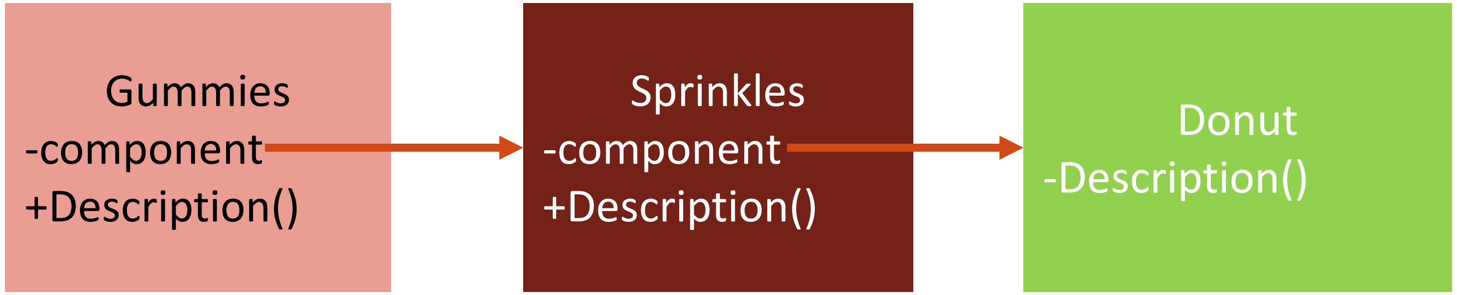
Introduction - Decorator

- 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**.”

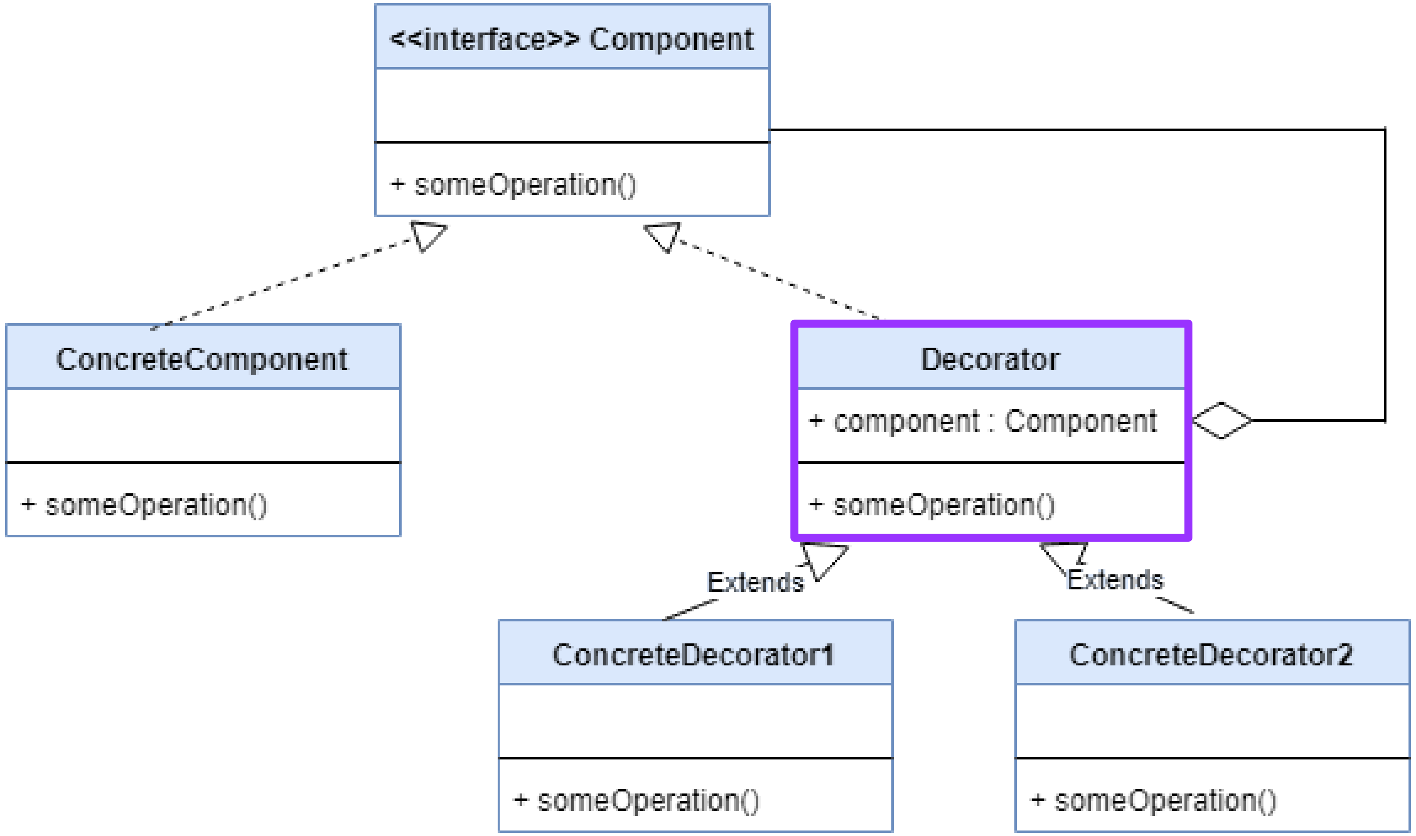
Decorators

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.



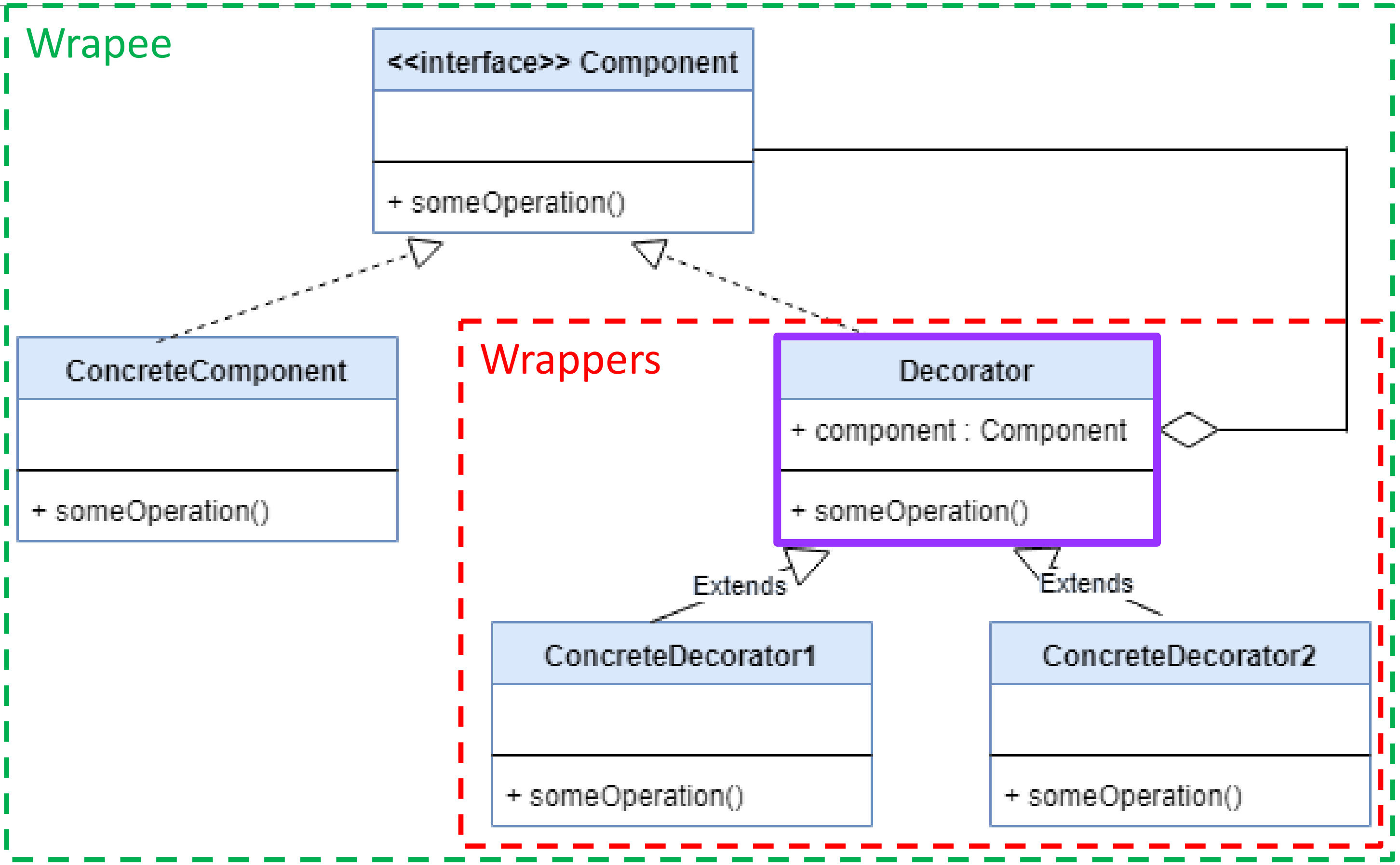
Decorators

Decorators are just wrappers with one added feature.

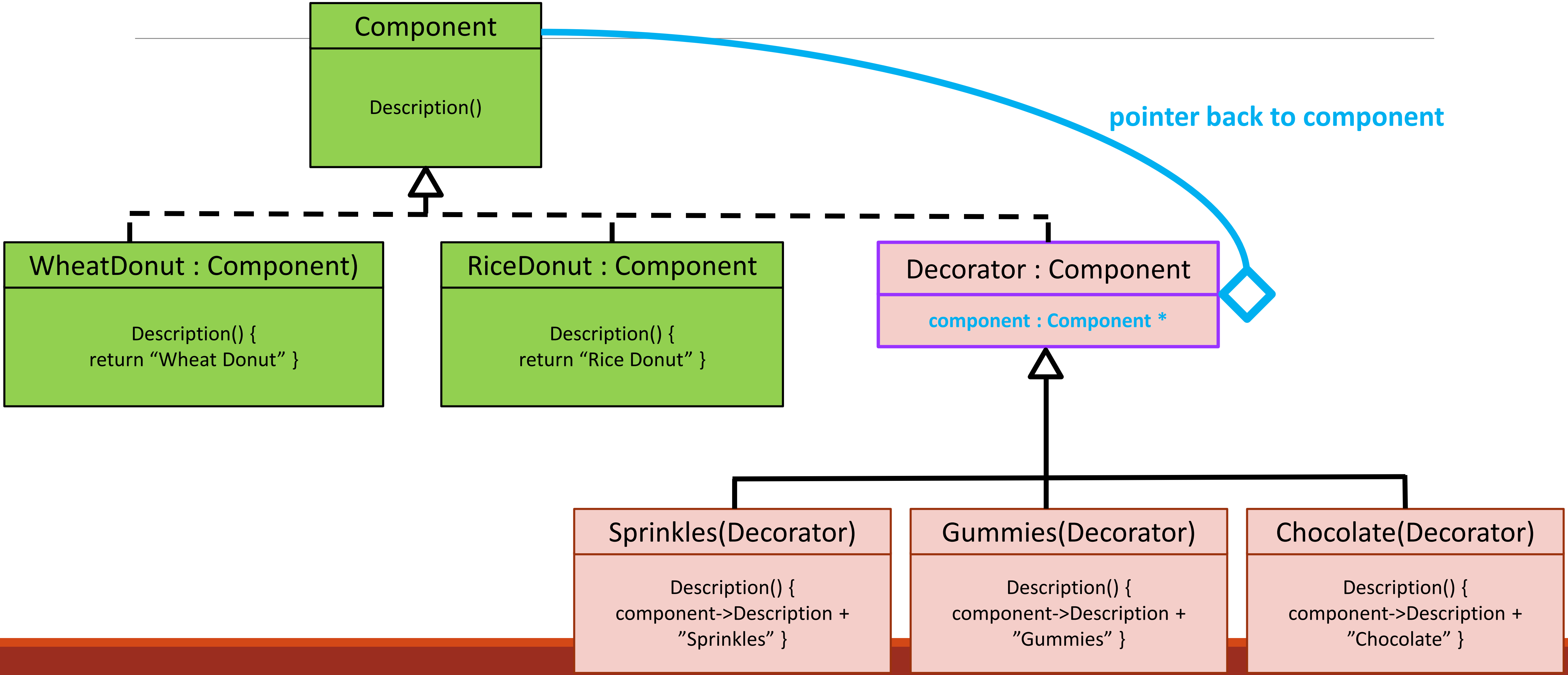
All decorators inherit from a **Base Decorator Class**. This allows us to define multiple different wrappers that can wrap around the concrete component and around other wrappers as well.

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()

pointer back to component

WheatDonut : Component)

Description() {
return “Wheat Donut” }

RiceDonut : Component

Description() {
return “Rice Donut” }

Decorator : Component

component : Component *

Sprinkles(Decorator)

Description() {
component->Description +
”Sprinkles” }

Gummies(Decorator)

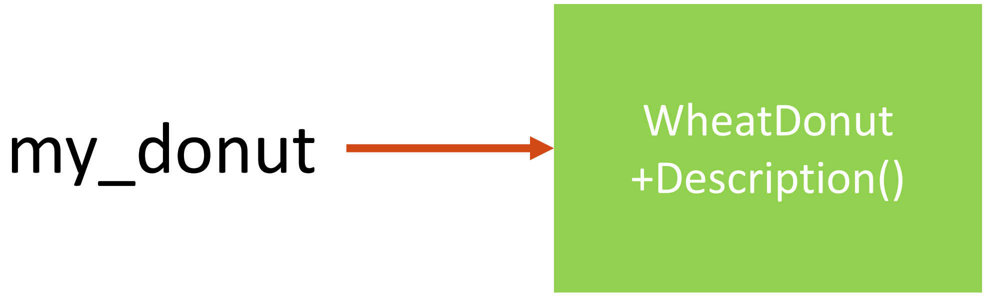
Description() {
component->Description +
”Gummies” }

Chocolate(Decorator)

Description() {
component->Description +
”Chocolate” }

Introduction - Decorator

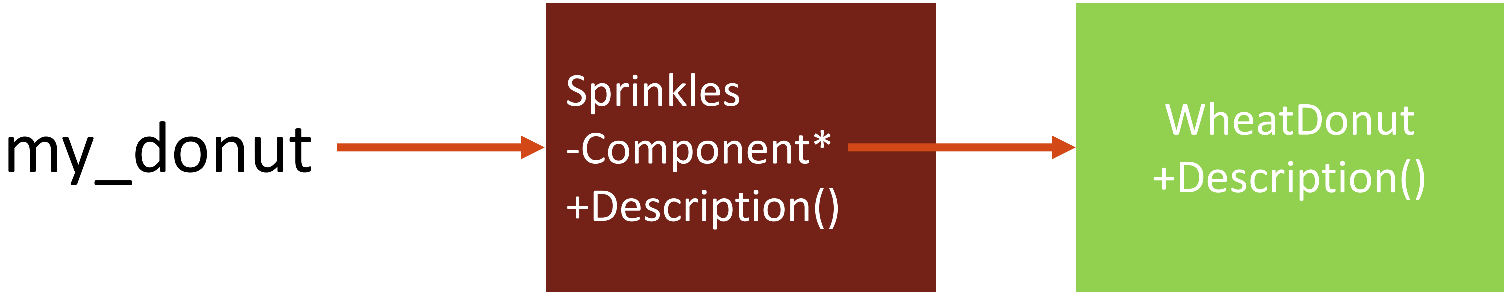
- Component *my_donut = new **WheatDonut**();



- Create **WheatDonut** component

Introduction - Decorator

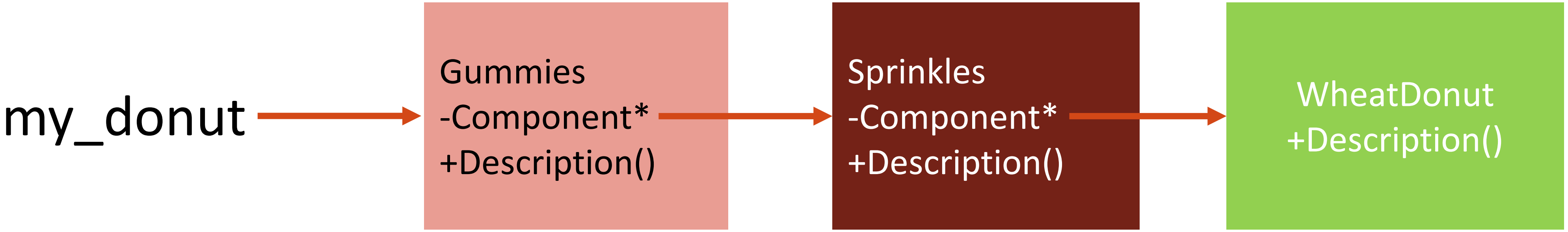
- Component *my_donut = new **WheatDonut**();
- my_donut = new **Sprinkles**(my_donut);



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

Introduction - Decorator

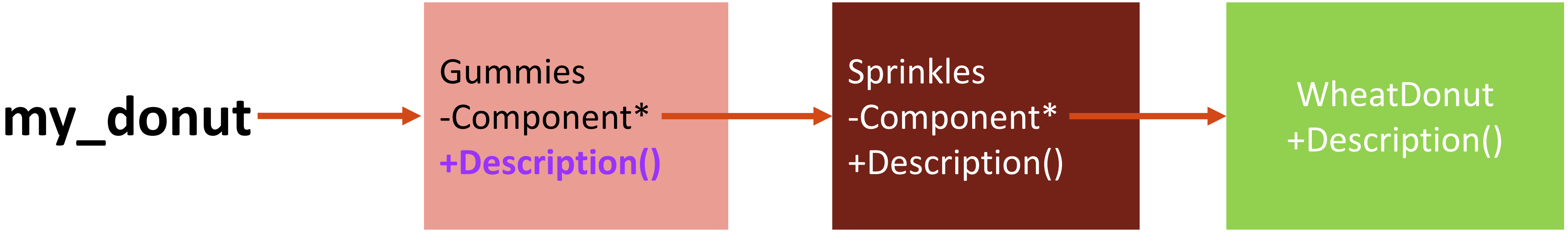
- Component *my_donut = new **WheatDonut**();
- my_donut = new **Sprinkles**(my_donut);
- my_donut = new **Gummies**(my_donut);



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

Introduction - Decorator

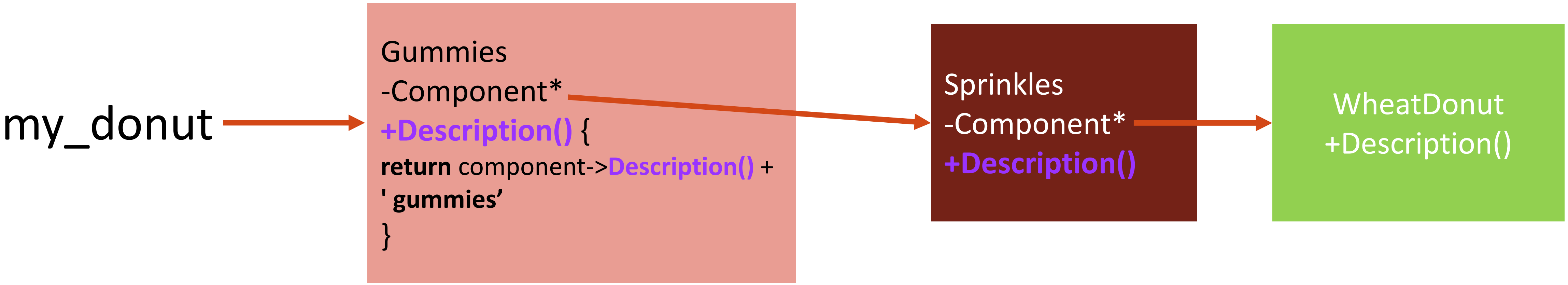
- Component *my_donut = new **WheatDonut**();
- my_donut = new **Sprinkles**(my_donut);
- my_donut = new **Gummies**(my_donut);
- cout << **my_donut**->**Description**();



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

Introduction - Decorator

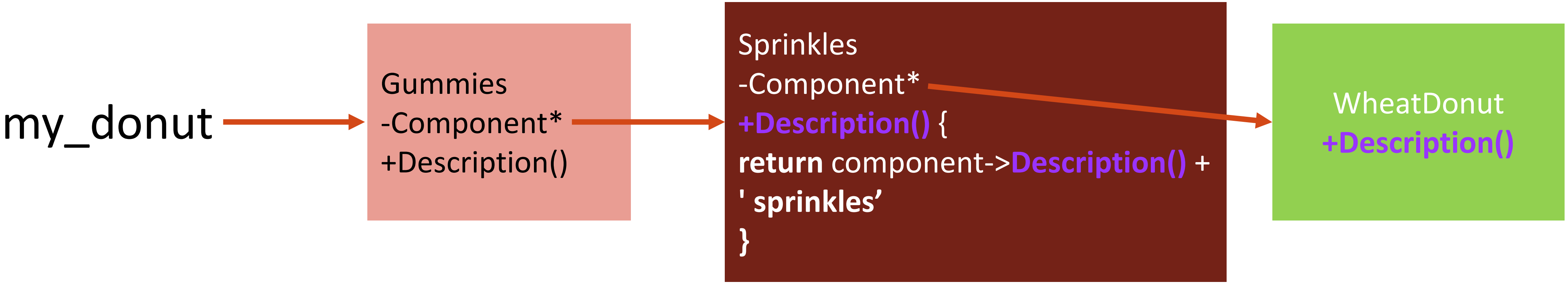
- Component *my_donut = new **WheatDonut**();
- my_donut = new **Sprinkles**(my_donut);
- my_donut = new **Gummies**(my_donut);
- cout << my_donut->**Description();**



- Calls to Description function will chain into toppings until they reach **WheatDonut**
- Gummies’ Description function calls Sprinkles’ Description function

Introduction - Decorator

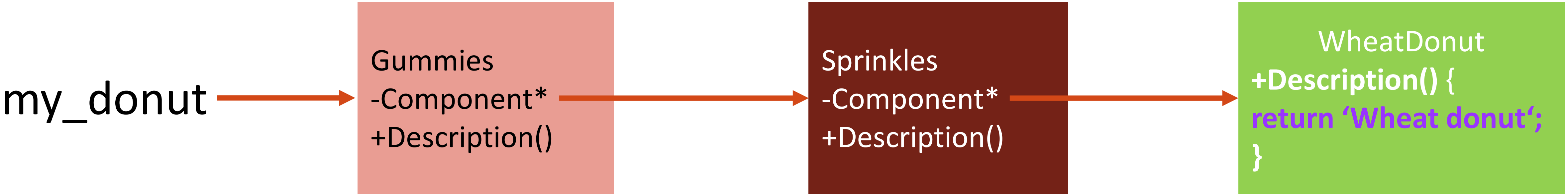
- Component *my_donut = new **WheatDonut**();
- my_donut = new **Sprinkles**(my_donut);
- my_donut = new **Gummies**(my_donut);
- cout << my_donut->Description();



- Sprinkles’ Description function calls WheatDonut’s Description function

Introduction - Decorator

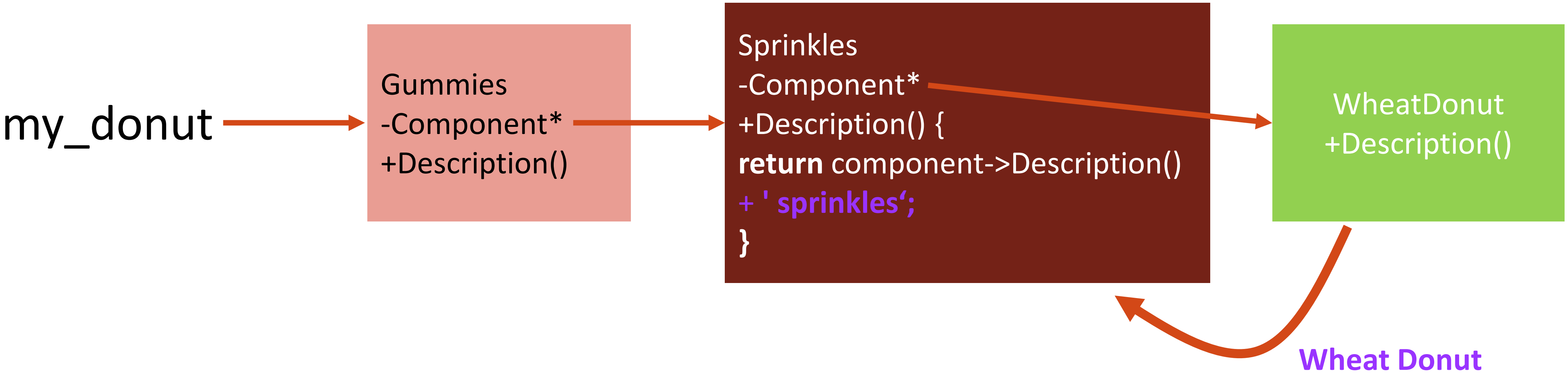
- Component *my_donut = new WheatDonut();
- my_donut = new **Sprinkles**(my_donut);
- my_donut = new **Gummies**(my_donut);
- cout << my_donut->Description();



- String is appended as we return from **WheatDonut**’s **Description()** function calls
- **WheatDonut**’s **Description()** function returns “**Wheat donut**”

Introduction - Decorator

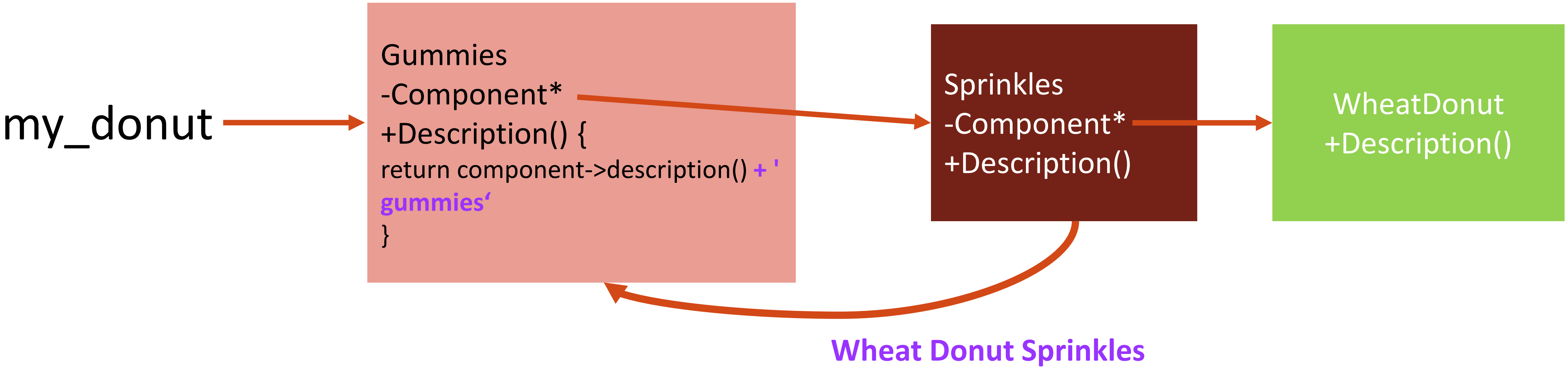
- Component *my_donut = new **WheatDonut**();
- my_donut = new **Sprinkles**(my_donut);
- my_donut = new **Gummies**(my_donut);
- cout << my_donut->Description();



- “**sprinkles**” appended to returned “**Wheat donut**”. This string also returned

Introduction - Decorator

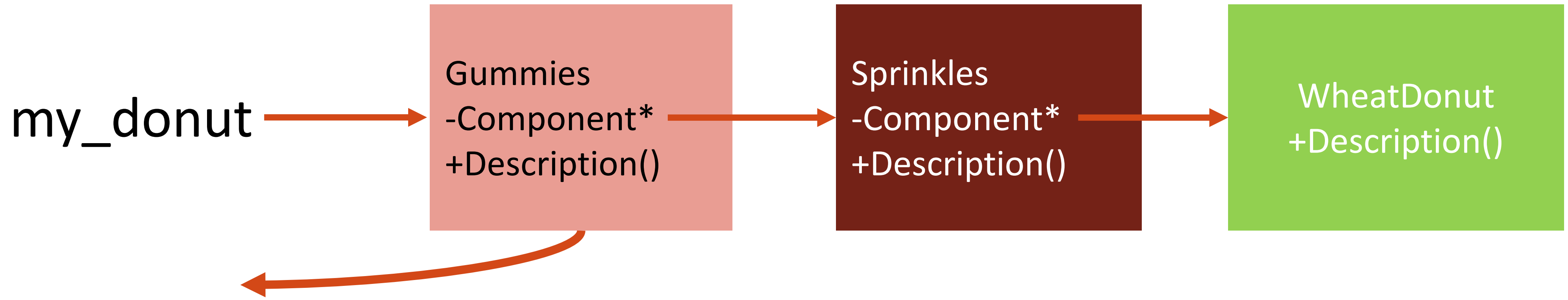
- Component *my_donut = new **WheatDonut**();
- my_donut = new **Sprinkles**(my_donut);
- my_donut = new **Gummies**(my_donut);
- cout << my_donut->Description();



- "**gummies**" appended to returned "**Wheat donut sprinkles**". This string also returned

Introduction - Decorator

- Component *my_donut = new WheatDonut();
- my_donut = new Sprinkles(my_donut);
- my_donut = new Gummies(my_donut);
- cout << my_donut->Description();



Wheat donut sprinkles gummies

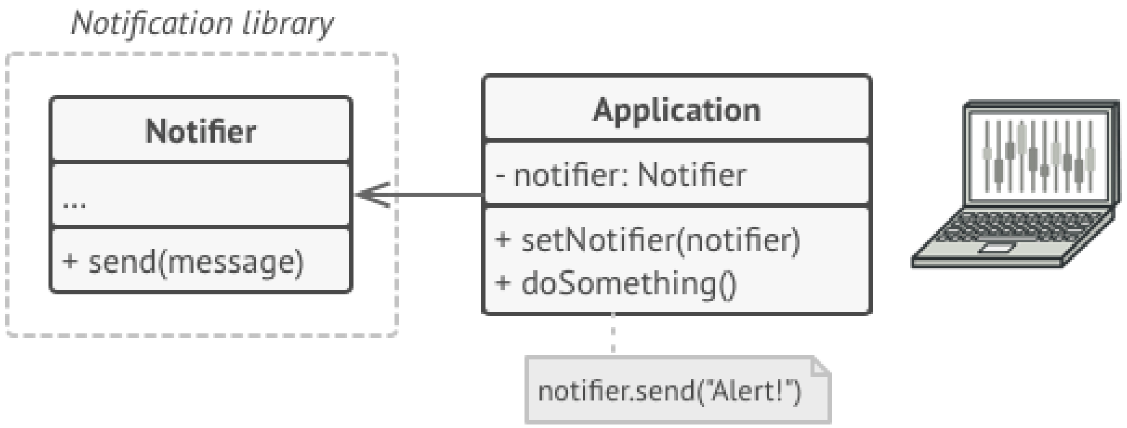
- Final “**Wheat donut sprinkles gummies**” returned to print out

Decorator.cpp

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

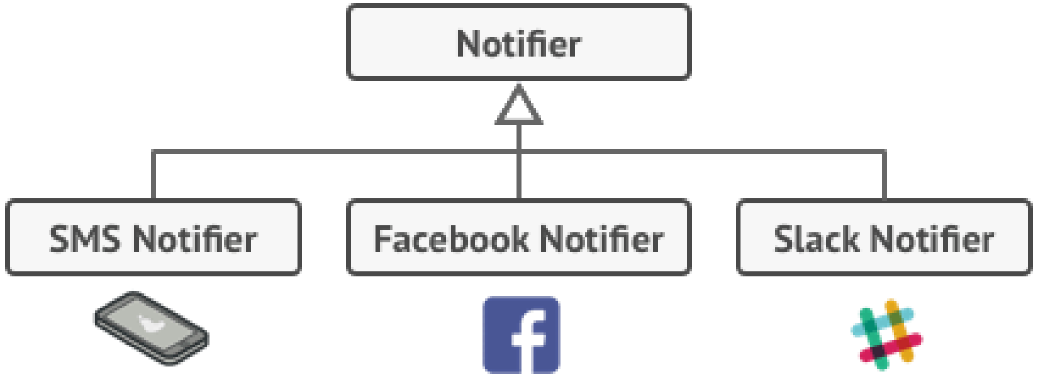
Say we have a notification system.

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?

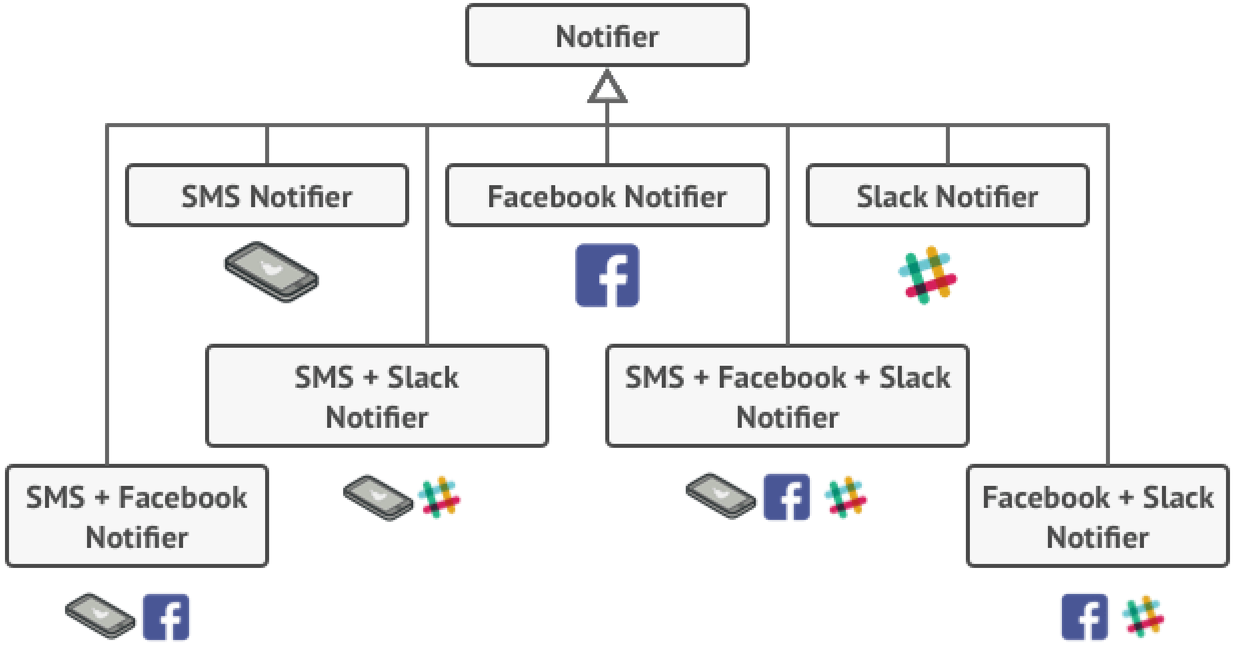
Different users have different devices or apps.

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

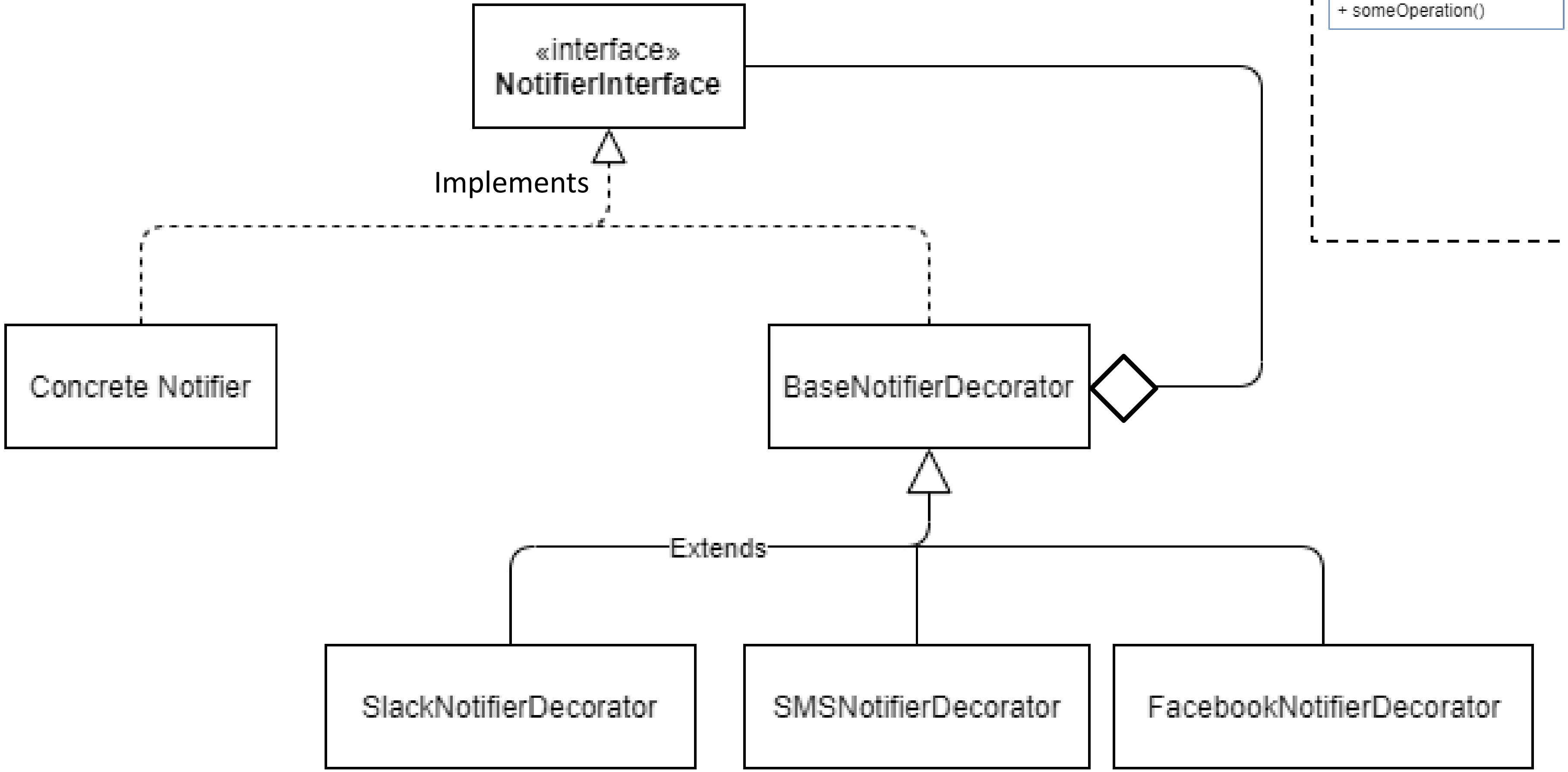
Person C may receive all!

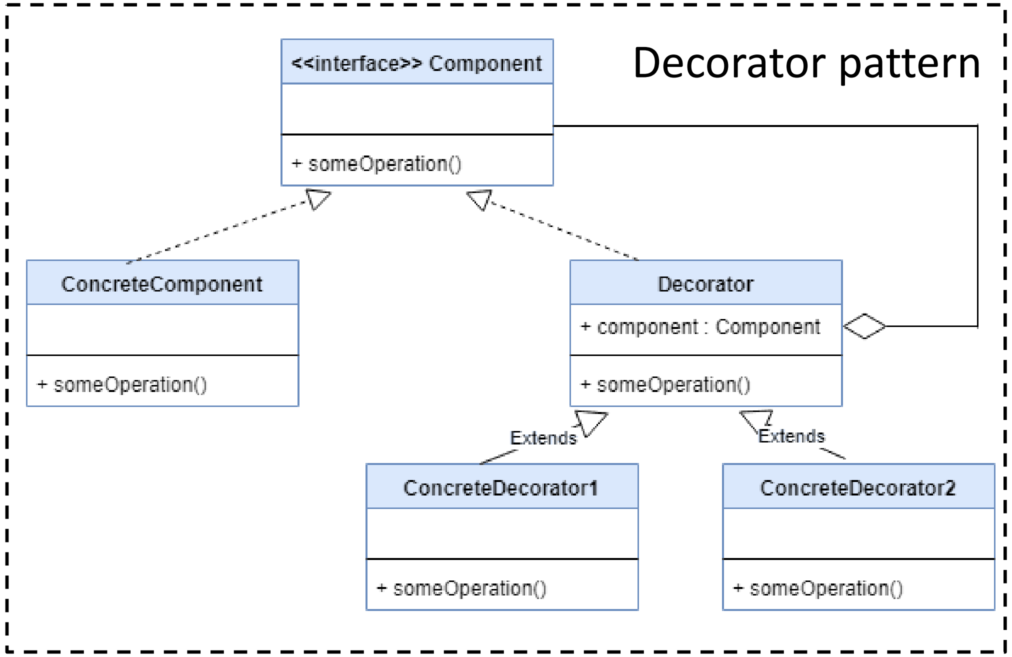
This inheritance hierarchy is becoming convoluted and difficult to maintain. There must be a simpler solution!

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



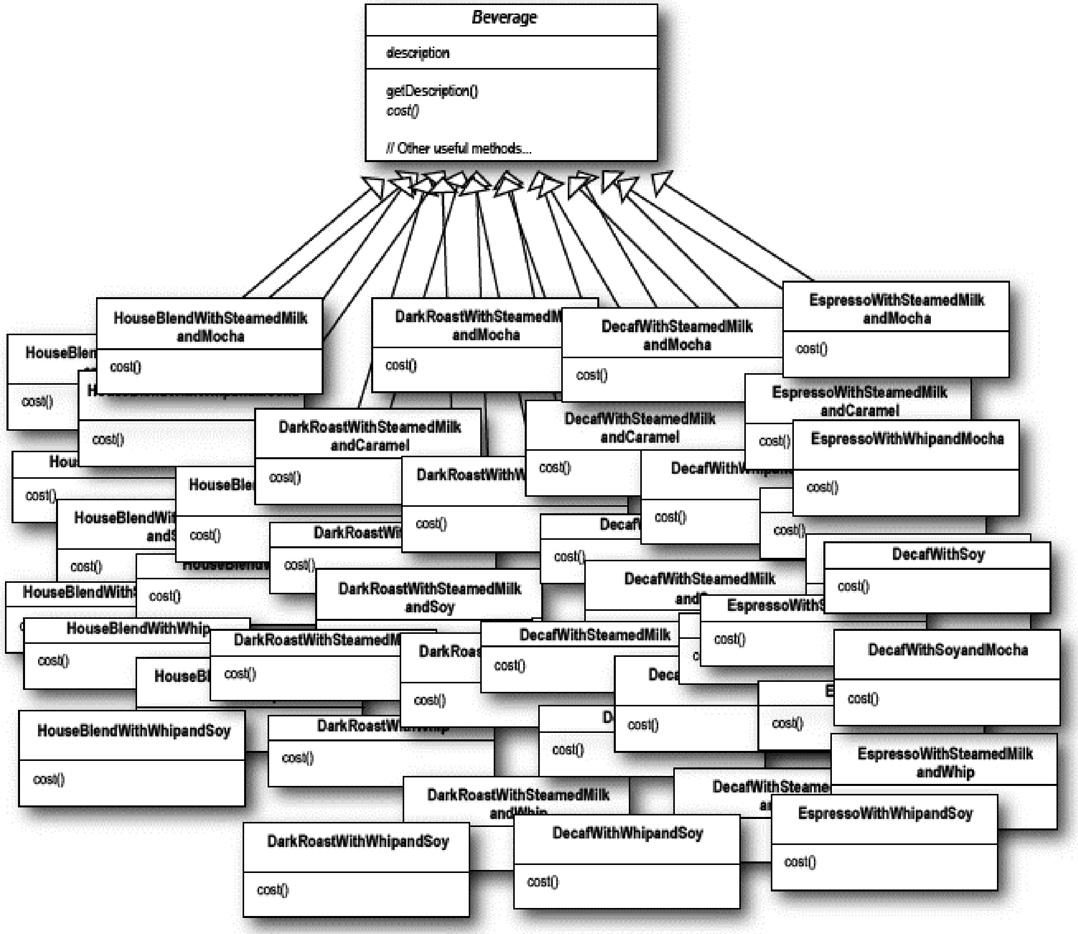
Notifier Decorator





Decorator Advantages

We get to avoid this:



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