The decorator pattern

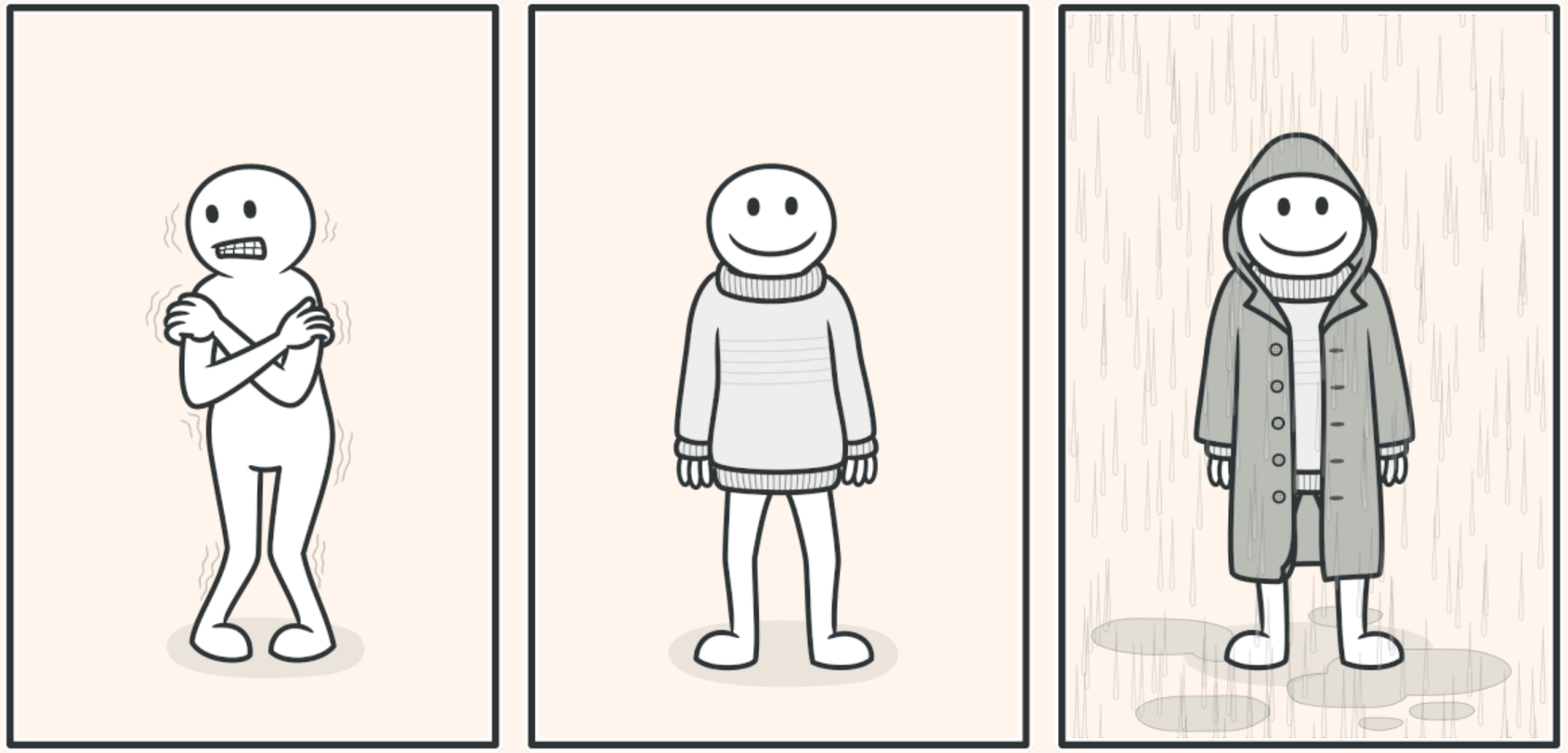
START

Team:

Phan Huỳnh Anh Thư - 17095
Võ Công Minh - 10421040

The decorator pattern

START



Example:

Combined effect of wearing multi-layer clothes

Definition:

The Decorator Pattern is a design pattern that allows you to add new functionality to an object by wrapping it with one or more decorator objects. It is a structural pattern because it is concerned with the structure of the code, rather than the behavior of the objects.

Why use the Decorator Pattern?
&
How does the Decorator Pattern work?

Why: The Decorator Pattern is useful in situations where you have an object that you want to extend with additional functionality, but you don't want to modify the original object. It can also be useful when you have a large number of possible combinations of functionality that you want to be able to apply to an object.

How: The Decorator Pattern works by creating a new class that "wraps" the original object. This new class is called the decorator, and it adds new functionality to the original object by implementing the same interface as the original object and forwarding calls to the original object as needed.

Implemetation

```cpp
#include <iostream>

class Coffee {
public:
    virtual double getCost() = 0;
    virtual std::string getDescription() = 0;
};

class Espresso : public Coffee {
public:
    double getCost() {
        return 2.0;
    }
    std::string getDescription() {
        return "Espresso";
    }
};

class CoffeeDecorator : public Coffee {
public:
    CoffeeDecorator(Coffee* coffee) {
        this->coffee = coffee;
    }
    double getCost() {
        return coffee->getCost();
    }
    std::string getDescription() {
        return coffee->getDescription();
    }
private:
    Coffee* coffee;
};
```

```cpp
class Cream : public CoffeeDecorator {
public:
    Cream(Coffee* coffee) : CoffeeDecorator(coffee) {
    }
    double getCost() {
        return CoffeeDecorator::getCost() + 0.5;
    }
    std::string getDescription() {
        return CoffeeDecorator::getDescription() + ", Cream";
    }
};

int main() {
    Coffee* espresso = new Espresso();
    std::cout << espresso->getDescription() << " costs $" << espresso->getCost() << std::endl;

    Coffee* espressoWithCream = new Cream(espresso);
    std::cout << espressoWithCream->getDescription() << " costs $" << espressoWithCream->getCost() << std::endl;

    delete espresso;
    delete espressoWithCream;
    return 0;
}
```

Avantages

1. Add new function without modifying the original object
2. Flexible and modular combine different sets
3. Add or remove responsibilities from an object at runtime
4. Follow Single Responsibility Principle

Disavantages

1. Difficult to remove a specific wrapper
2. Complexity
3. Harder to debug
4. Ugly initial configuration code of layers

THANK YOU!