Replace Type Code with Subclasses

TEAM 18

2252551 徐俊逸

2253744 林觉凯

2153085 马立欣

2154284 杨骏昊

2151422 武芷朵

CONTENTS

01. Problem—Type Code

```
class Employee {
public :
    static const int ENGINEER = 1;
    static const int MANAGER = 2;
    static const int INTERN = 3;

private:
    int type; // type of employee
public:
    explicit Employee(int type) : type(type) {}
    void setrype(int type) {
        this->type = type;
    }

    double calculatePay(double basesalary) const{
        switch (type)
        {
            case ENGINEER :
                return basesalary * 1.2;
            case MANAGER :
                 return basesalary * 1.5;
            case INTERN :
                return basesalary * 0.8;
        default:
            throw invalid_argument("Invalid employee
type");
        }
    }
};
```

Type Code refers to a value or identifier (such as a number, string, or enum) that represents the type or state of an object or entity

Type Code is used to determine behavior of a program, often using ‘if ’ or ‘switch’

Employee type is a type code and the value of it (which could be ENGINEER, MANAGER, or INTERN) affects what the program does in the switch statement

01. Problem

```
class Employee {
public :
    static const int ENGINEER = 1;
    static const int MANAGER = 2;
    static const int INTERN = 3;

private:
    int type; // type of employee
public:
    explicit Employee(int type) : type(type) {}
    void setrype(int type) {
        this->type = type;
    }

    double calculatePay(double basesalary) const{
        switch (type)
        {
            case ENGINEER :
                return basesalary * 1.2;
            case MANAGER :
                 return basesalary * 1.5;
            case INTERN :
                return basesalary * 0.8;
        default:
            throw invalid_argument("Invalid employee
type");
        }
    }
};
```

Poor Maintainability

If values of *employee type* increases, we need to constantly extend the switch statement. It violates the Open/Closed Principle a system should be open for extension but closed for modification.

Difficult to Extend

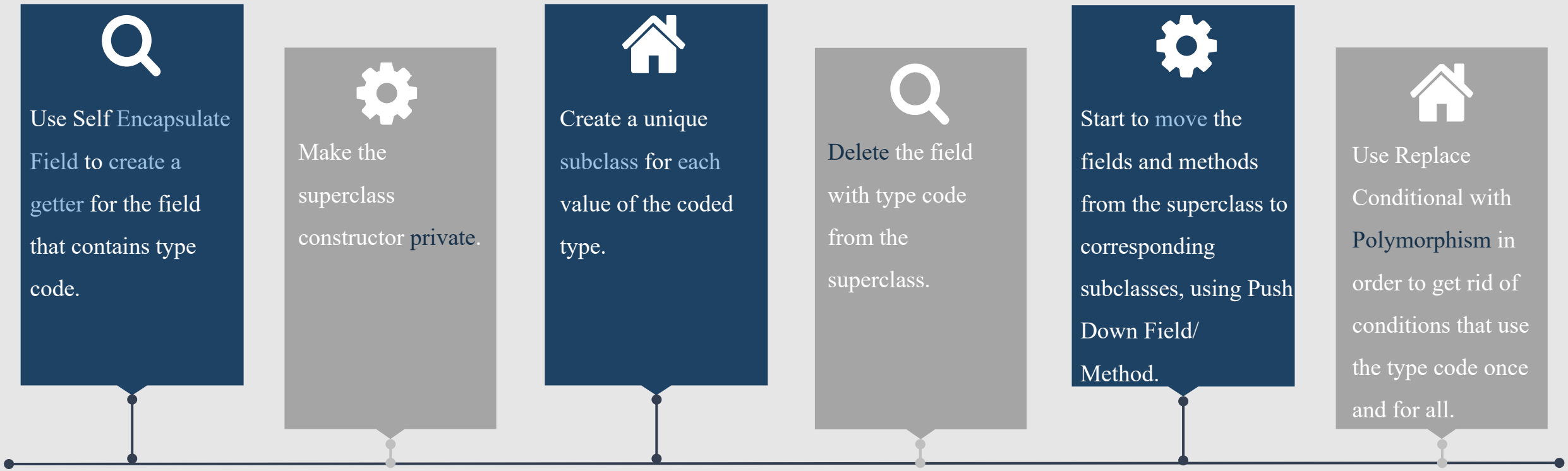
Each time a new state is added, we need to modify the existing code by adding new case statements, which makes the original more prone to errors.

Non-intuitive Type Code

Value of *employee type* is represented by numbers (1, 2, 3)
It is not intuitive and difficult to remember the meaning of the numbers

02. Solutions

Simplify the code by replacing the type code with the subclasses.



03. Steps and Examples - Step1

Original code:

```
class Employee {
private:
    int typeCode;
public:
    Employee(inttypeCode) : typeCode(typeCode) {}
    int getTypeCode() const {
        return typeCode;
    }
    void printEmployeeInfo() const {
        if (typeCode ==1) {
            cout <<"This is a Manager"<< endl;
        } elseif (typeCode ==2) {
            cout <<"This is a Technical Employee"<< endl;
        } elseif (typeCode ==3) {
            cout <<"This is a Sales Employee"<< endl;
        }
    }
};
int main() {
    Employee emp1(1);
    Employee emp2(2);
    Employee emp3(3);
    emp1.printEmployeeInfo();
    emp2.printEmployeeInfo();
    emp3.printEmployeeInfo();
    return0;
}
```

TypeCode-to represent the employee type

Operation-to print info

- Manager
- Technical staff
- Salesperson

1. Self Encapsulate Field

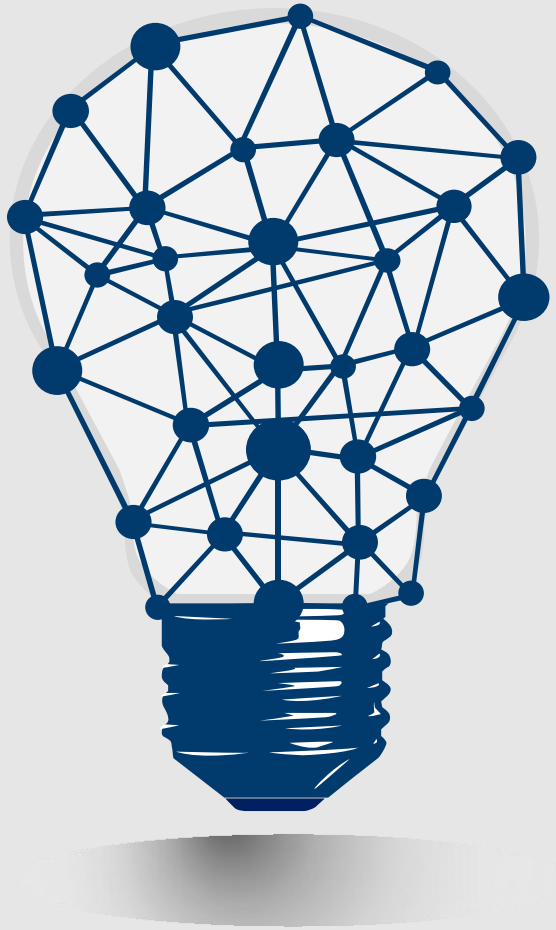
- to encapsulate the typeCode field and provide a getter method that ensures that access to the field is made by method rather than direct access.

```
class Employee {
private:
    int typeCode;
public:
    Employee(inttypeCode) : typeCode(typeCode) {}
    int getTypeCode() const {
        return typeCode;
    }
    virtualvoid printEmployeeInfo() const {
        if (getTypeCode() ==1) {
            cout <<"This is a Manager"<< endl;
        } elseif (getTypeCode() ==2) {
            cout <<"This is a Technical Employee"<< endl;
        } elseif (getTypeCode() ==3) {
            cout <<"This is a Sales Employee"<< endl;
        }
    }
};
```

private

getTypeCode method

03. Steps and Examples - Step2



2. Private constructor with factory method

- Make sure that an instance of each Employee subclass can only be created under a specific condition or manner.
- Focusing the instantiated logic in the factory method makes it easier to extend new subclasses in the future.

```
class Employee {
private:
    int typeCode;
    Employee(int typeCode) : typeCode(typeCode) {}
public:
    static Employee*createEmployee(int typeCode) {
        if (typeCode ==1) {
            return new Manager();
        } else if (typeCode ==2) {
            return new TechnicalEmployee();
        } else if (typeCode ==3) {
            return new SalesEmployee();
        }
        return nullptr;
    }
    int getTypeCode() const {
        return typeCode;
    }
    virtual void printEmployeeInfo() const=0;
};
```

Private constructor

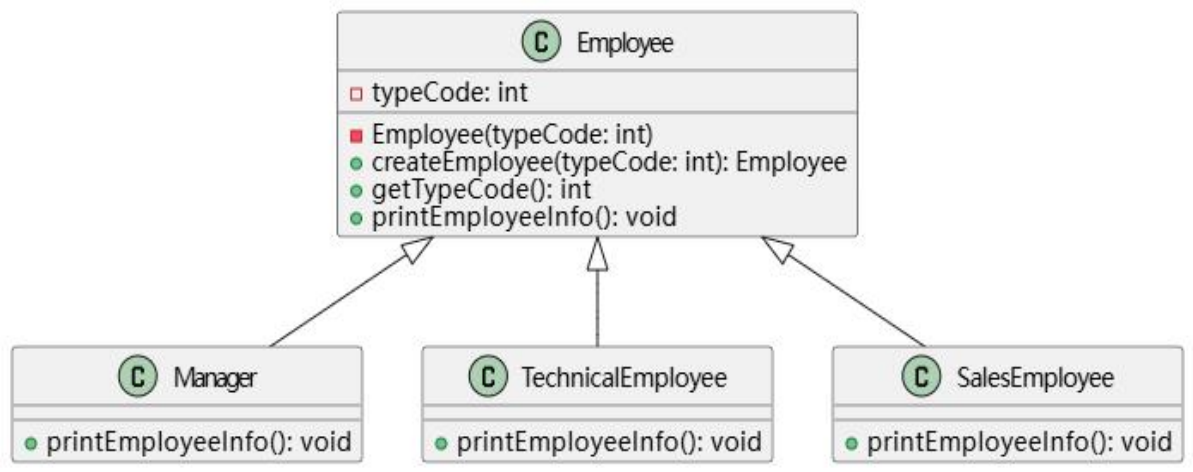
Static factory method

Getter Method

03. Steps and Examples - Step3

3. Create unique subclasses

- Each subclass implements the printEmployeeInfo method according to its own employee type.
- If new employee types are added in the future, the method improves flexibility and scalability.



Manager

```
class Manager : public Employee {
public:
    Manager() : Employee(1) {}
    void printEmployeeInfo() const override {
        cout<<"This is a Manager"<<endl;
    }
};
```

Technical Employee

```
class TechnicalEmployee : public Employee {
public:
    TechnicalEmployee() : Employee(2) {}
    void printEmployeeInfo() const override {
        cout<<"This is a TechnicalEmployee"<<endl;
    }
};
```

Sales Employee

```
class SalesEmployee : public Employee {
public:
    SalesEmployee() : Employee(3) {}
    void printEmployeeInfo() const override {
        cout <<"This is a Sales Employee"<< endl;
    }
};
```

03. Steps and Examples - Step4

4.Delete code field

- Remove the typeCode field from the Employee superclass.
- Make the printEmployeeInfo method a pure virtual method.

- This makes the Employee class an abstract class, which cannot be directly instantiated and means that each subclass must implement its own printEmployeeInfo method.

Original code:

remove the type code field

Rewrite as a purely virtual function

```
class Employee{
    private:
        int typeCode;
    public:
        ...// Other functions
        void printEmployeeInfo() const{
            if(typecode ==1){
                cout<<"This is a Manager"<< endl;
            }
            else if(typecode ==2){
                cout<<"This is a Technical Employee"<<endl;
            }
            else if(typecode ==3){
                cout<<"This is a Sales Employee"<<endl;
            }
        }
}
```

Refactored code:

```
class Employee{
    public:
        ...// Other functions
        // A pure virtual method.
        virtual void printEmployeeInfo()
const=0;
}
```

03. Steps and Examples - Step5、6

5

Move the fields and methods to the subclasses

- Move the fields and methods related to `typeCode` from the superclass to the subclasses.
- Ensure that each subclass is fully independent, handling its own logic without relying on the implementation of the parent class.

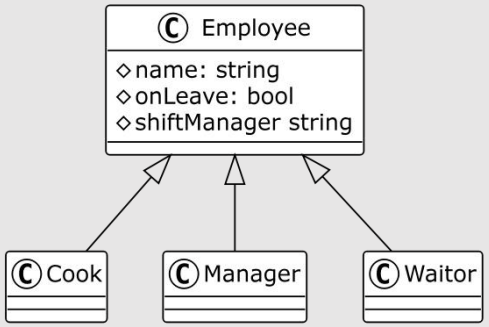
```
int main() {
    Employee* emp1=Employee::createEmployee(1); // manager
    Employee* emp2=Employee::createEmployee(2); //
technical staff
    Employee* emp3=Employee::createEmployee(3); // salesman

    emp1->printEmployeeInfo();
    emp2->printEmployeeInfo();
    emp3->printEmployeeInfo();

    delete emp1;
    delete emp2;
    delete emp3;
    return 0;
}
```

6

Replace Conditional with Polymorphism

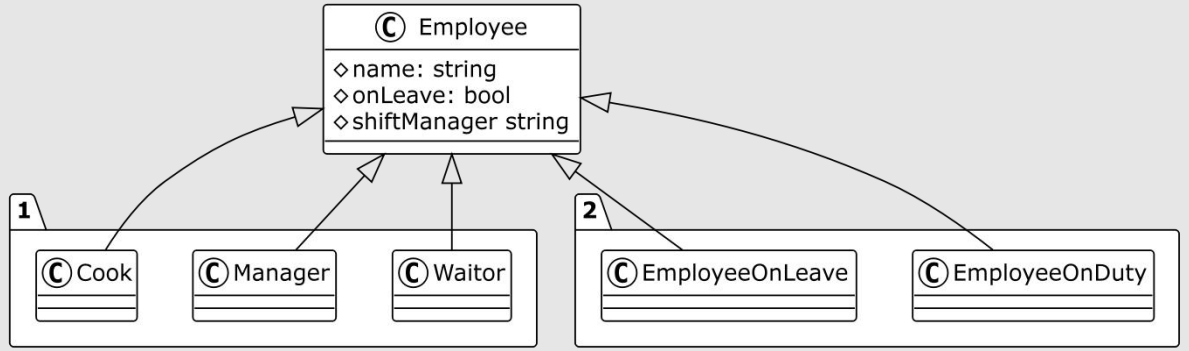
- Replace the code that originally required conditional statements with polymorphism.

04. Limitations - Dual Hierarchy



Before

We have a very simple while reasonable class hierarchy, as pictured.



After

Now, if we are trying to apply the technique **Replace Type Code with Subclasses** on class ***Employee*** to remove the type code onLeave, we will definitely run into trouble because the most direct way we have just discussed will result in what the picture looks like.

- **"If you already have an existing class hierarchy, you cannot create a dual hierarchy through inheritance."** In this case, simply create another subclass to replace the type code -> dual hierarchy, generally undesirable.
- As you can see, the **dual hierarchy** kicks in, which is clearly not what we want.

04. Limitations - The values of type code can change

```
class Payment {
public:
    static const int CREDIT_CARD =1;
    static const int PAYPAL =2;
    static const int BANK_TRANSFER =3;
private:
    int paymentType;
public:
    explicit Payment(int type) :
          paymentType(type) {}

    void setPaymentType(int type) {
        paymentType = type;
    }
    double calculateFee(double amount) const {
        switch (paymentType) {
            case CREDIT_CARD:
                return amount *0.02;
            case PAYPAL:
                return amount *0.03;
            case BANK_TRANSFER:
                return5.0;
            default:
                throw std::
          invalid_argument("Invalid payment type");
        }
    }
};
```

Replacing typecode with subclasses doesn't work when it changes after the object is created.
Because the object's class cannot be dynamically replaced.

In this example, each of the three type codes represents a different payment method(*credit card*, *paypal*, and *bank transfer*). Each payment method has its own fee calculation logic.

To address the issue of role changes while retaining the object entity, we use the Strategy pattern.

04. Limitations - The values of type code can change

The steps are as follows:

1 Define the strategy interface

```
class PaymentStrategy {
public:
    virtual ~PaymentStrategy() =default;
    virtual double calculateFee(double amount) const=0;
};
```

2 Implement specific strategy classes

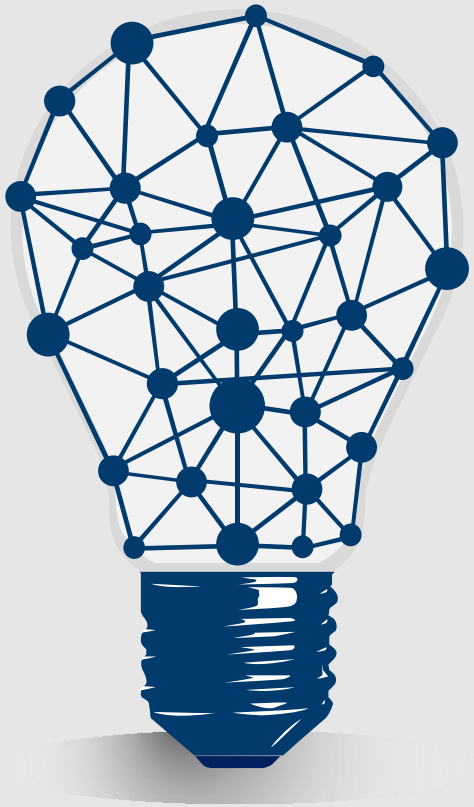
```
class PaymentStrategy {
public:
    virtual ~PaymentStrategy() =default;
    virtual double calculateFee(double amount) const=0;
};
class CreditCardPayment : public PaymentStrategy {
public:
    double calculateFee(double amount) const override {
        return amount *0.02;
    }
};
class PayPalPayment : public PaymentStrategy {
public:
    double calculateFee(double amount) const override {
        return amount *0.03;
    }
};
class BankTransferPayment : public PaymentStrategy {
public:
    double calculateFee(double amount) const override {
        return 5.0;
    }
};
```

3 Modify the main class

```
class Payment {
private:
    std::unique_ptr<PaymentStrategy> paymentStrategy;
public:
    Payment(std::unique_ptr<PaymentStrategy> strategy) :
paymentStrategy(std::move(strategy)) {}
    void setPaymentStrategy(std::unique_ptr<PaymentStrategy> strategy)
{
        paymentStrategy =std::move(strategy);
    }
    double calculateFee(double amount) const {
        return paymentStrategy->calculateFee(amount);
    }
};
```

4 Usage example

```
int main() {
    Payment payment(std::make_unique<CreditCardPayment>());
    double amount =1000.0;
    std::cout <<"Credit Card Fee:
"<<payment.calculateFee(amount) <<std::endl;

payment.setPaymentStrategy(std::make_unique<PayPalPayment>
());
    std::cout <<"PayPal Fee:
"<<payment.calculateFee(amount) <<std::endl;

payment.setPaymentStrategy(std::make_unique<BankTransferPay
ment>());
    std::cout <<"Bank Transfer Fee:
"<<payment.calculateFee(amount) <<std::endl;
    return 0;
}
```



The output is as follows:

```
Credit Card Fee: 20
PayPal Fee: 30
Bank Transfer Fee: 5
```

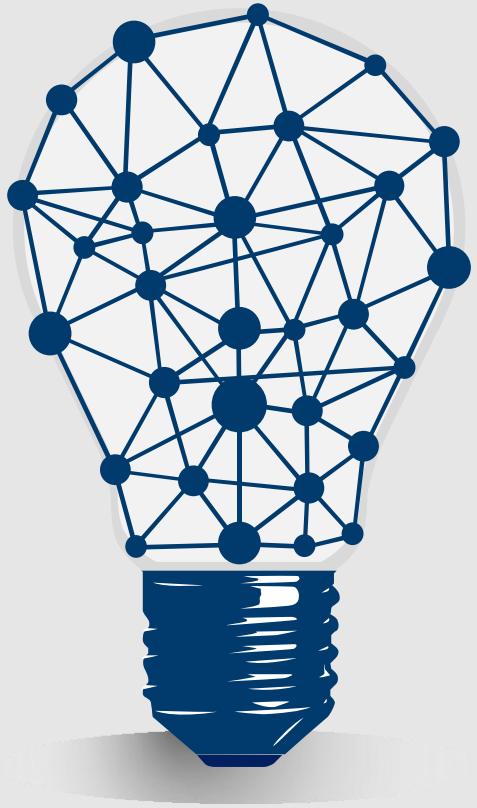
05. Pros & cons—Benefits

Eliminate conditional statements

- Different behaviors for different types are handled by their respective subclasses
- no need for if-else or switch-case statements to check types

Concentrate responsibilities

- Each subclass is responsible for the behavior related to its specific type
- aligned with the Single Responsibility Principle



Enhance scalability

- only need to create a new subclass without modifying existing code when a new type is added
- follow the Open/Closed Principle

Improve the readability and maintainability

- avoid centralized conditional logic by separating type-related behaviors into specific subclasses
- Each subclass has clear and independent behavior

05. Pros & cons—Drawbacks

01

Increase the number of classes

A rapid increase in the number of classes and add complexity to class management and maintenance

02

Excessive inheritance

Require frequent modifications to subclasses, or introduce multiple levels of inheritance

03

High refactoring cost

Especially in legacy systems where type codes are widely used and tightly coupled with other modules

References

	Author	Doucument Title	Source
[1]	Rockit.Zone	Replacing Type Code with Subclasses	https://rockit.zone/post/switch-case/replacing-type-code-with-subclasses/.
[2]	Fanatixan	Replacing Type Code with Class	https://dev.to/fanatixan/replacing-type-code-with-class-1c5i.
[3]	V. Musco et al.	Refactoring Opportunities for Replacing Type Code with Subclass and State.	https://www.researchgate.net/publication/328510210_Identifying_refactoring_opportunities_for_replacing_type_code_with_subclass_and_state.
[4]	Refactoring.Guru	Replace Type Code with Subclasses	https://refactoring.guru/replace-type-code-with-subclasses

Thanks for listening!

TEAM 18

2252551 徐俊逸

2253744 林觉凯

2153085 马立欣

2154284 杨骏昊

2151422 武芷朵