SOLID Principles

Equuleus Technologies

Why SOLID Principles?

The traits of well designed software are as follows

Maintainability - The ease with which a software system or component can be modified to correct faults, improve performance, or other attributes, or adapt to a changed environment

Testability - Testable system lets you effectively test individual part of the system in isolation

Flexibility - Flexibility of a system allows it to be adapted to work in different ways than previously envisioned

Extensibility - Extensibility is the ease in which new features can be added

Using SOLID design principles we can build systems that are less coupled and that allow for systems that satisfy the above criteria

What is SOLID?

SOLID is acronym for

SRP - Single Responsibility Principle
- a class should have only a single responsibility

OCP - Open/Closed Principle
- software entities … should be open for extension, but closed for modification

LSP - Liskov Substitution Principle
- objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program

ISP - Interface Segregation Principle
- many client-specific interfaces are better than one general-purpose interface

DIP - Dependency Inversion Principle
- Depend upon Abstractions. Do not depend upon concretions

Equuleus Technologies

The Initial Product

An Employee class with the following fields

- EmployeeId
- Name
- DateOfJoining
- EmployeeType
- Salary

Two operations

- Work()
- IsEmployeeEligibleForPromotion(); // eligible if the employee worked for over a year

New Scope

Scope

- Lets allow to calculate the TakeHomeSalary()
- We should have a method CalculateTaxBracket() that should calculate the tax bracket of the Employee. This way we can add rules for calculating the tax rate without polluting the TakeHomeSalary()

Single Responsibility Principle is about actors and high level architecture

This principle advocates with the separation of concerns, a class should have only one responsibility and if it has more than one responsibility it should be broken into separate classes.

Consider an Employee class

The logic for Calculating the TakeHomeSalary uses method CalculateTaxBracket that is in the Employee class.

Issue:

Clearly we have a violation of the SRP, the CalculateTaxBracket responsibility should not belong to the Employee class. This is something that belongs to the TaxRateController.

Any changes in the tax rate logic will require us to change the Employee class

Solution

Breakdown classes so that each class has a specific purpose and all functionality is cohesive

Code – Single Responsibility Principle

Before

```
public decimal TakeHomeSalary()
    {
       return this.Salary * (decimal) (1 - this.CalculateTaxBracket()/100);
    }
```

After

Create TaxRateController class that is responsible for calculating the tax rate, now Employee is no longer responsible for calculating tax rates.

```
public readonly TaxRateController TaxRateController = new TaxRateController();

public decimal TakeHomeSalary()
    {
       return this.Salary * (decimal) (1 - this.TaxRateController.CalculateTaxBracket()/100);
    }
```

Equuleus Technologies

New Scope

Scope

- Actually we can get to Dependency Injection without an added scope, we can achieve this by
  - Injecting Dependencies
  - Using Interfaces instead of Implementations

Equuleus Technologies

handling over control from the function itself to the caller function.

High level modules should not depend upon low-level modules. Both should depend upon abstractions.

Since we are using a single example and building upon the concept. We will discuss Dependency Injection sooner as it is a rudimentary step that deals with defining dependencies.

Issue

The TaxRateController is instantiated in the Employee class, there is a hard dependency on the lower layers and we are not using abstractions

Solution

Refactor the constructor to take ITaxRateController object as a parameter.

Code - Dependency Injection Principle

Before

```
public Employee(string employeeId, string name, System.DateTime dateOfJoining, decimal salary)
```

After

Create a constructor that accepts a ITaxRateController object allowing the caller to initialize the dependency

```
public Employee(string employeeId, string name, System.DateTime dateOfJoining, decimal salary,
        ITaxRateController taxRateController)
```

Equuleus Technologies

New Scope

Scope

- We realize that some Employees are Union employees, so we should reflect Union deducted from the TakeHomeSalary() calculation
- We have a policy that if the Employee has been working for over a year we deduct $800 else we deduct $1000

Open/Close Principle is about class design and feature extensions

The classes should be open for extension, but closed for modification. It means that trying to extend functionality we should not need to change existing code

Issue

- Trying to introduce union dues requires us to change the Employee class.
- We are forced to use if/else or switch statements to modify existing code to extend functionality

Solution

- Use Strategy pattern can be used to decouple the logic (Demo - recommended)
- Factory/Template pattern can also can be used to decouple the logic

Code - Open/Close Principle

Before

DeductUnionDues in Employee class is responsible for calculating Union dues, we are required to change the employee class whenever the rules of calculating union dues change.

After

Using inheritance over switch statements the DeductUnionDues functionality will not need changes in Employee class. Adding another EmployeeType will cause a change in Employee class hence the class is open for extension and closed for modification

Introduced UnionEmployee/NonUnionEmployee subclasses, the UnionEmployee class houses the logic. If we need to add another classification of Employee (say ContractEmployee with different union deductions) none of the class need changing.

New Scope

Scope

- We have included automation as part of our staff, they will have ID & Name and we should be able to treat them as non union Employee .
- We can create the employee as non union Employee while overriding the behavior of IsEmployeeEligibleForPromotion()

subtyping and inheritance

The Liskov Substitution Principle says that the object of a derived class should be able to replace an object of the base class without bringing any errors in the system or modifying the behavior of the base class

Issue

Consider the scenario where we are adding a new Employee classification (RobotEmployee). Considering some attributes that apply and some don't the question is does RobotEmployee change the Employee class behavior.

Solution

RobotEmployee is not a proper subclass of Employee as it changes the behavior of IsEmployeeEligibleForPromotion

Change hierarchy to extract only information that is applicable to Work and make that a base class that both RobotEmployee and other implement from

Consider Composition of the Worker object in the Employee class if we don't want to inherit that class

Code - Liskov Substitution Principle

Before

We introduced a class called RobotEmployee, we had to override the method IsEmployeeEligibleForPromotion as the condition does not apply.

The issue here is that this changes the base class behavior and RobotEmployee may not be a good candidate for inheriting from Employee

Tests targeting the base classes wont be applicable for RobotEmployee hence breaking behavior

After

We will need to change the inheritance hierarchy. We will need to create an Worker class that defines the common functionality and RobotEmployee and Employee both Inherit from.

New Scope

Scope

- We have decided to change to waive Union dues for employees who have been employed for 5 yrs or more.

Equuleus Technologies

Interface Segregation Principle is about business logic to clients communication

No client should be forced to depend on methods it does not use and thus easier to refactor, change, and redeploy

Issue

The client consuming the Employee classes depend on fat interfaces which contain employee related methods along with union specific methods. Clients that don't need union related functionality still need to be redeployed if there are changes in union related functionality

Solution

The interface should be composed of smaller interfaces this way client wont need to be redeployed when unrelated functionality changes.

Code - Interface Segregation Principle

Before

We had one IEmployee interface which had both Employee along Union related methods. This caused coupling and hence clients that did not implement Union related functionality needed redeployment

After

Define an IUnionDeduct interface and move the union related methods from the IEmployee interface. Also compose an IUnionEmployee employee that aggregates the IUnionDeduct & IEmployee interfaces. This way for client not consuming IUnionEmployee will not need redeployment.

Resources

Source for the above exercise can be downloaded from here

SOLID.zip