Министерство науки и высшего образования РФ

Федеральное государственное бюджетное образовательное учреждение высшего образования

"Кубанский государственный технологический университет"

(ФГБОУ ВО "КубГТУ")

Институт КСиИБ

Кафедра ИСП

**ОТЧЁТ**

по лабораторной работе №3

в рамках освоения дисциплины

**«Рефакторинг и работа с унаследованным кодом»**

**Тема: «Базовые техники рефакторинга. Обмен частей между классами»**

Обучающегося группы 21-КБ-ПР1

Направление подготовки 09.03.04 Программная инженерия

Студент Кузнецов Вадим Алексеевич

Преподаватель Степанова Елизавета Владимировна

Краснодар  
2024

**Цель работы:**

Изучить базовые техники рефакторинга позволяющие различным образом манипулировать классами программы.

**Задание:**

1) Преобразовать методы и классы программы в соответствии с предложенными в пункте 2.2 изменениями, применив описанные в теории методы рефакторинга.

2) Проверить корректность работы нового метода и работоспособность формирования счета.

**1) Переносим (копируем) методы GetBonus в Goods делая их public.**

public class Goods

{

public const int REGULAR = 0;

public const int SALE = 1;

public const int SPECIAL\_OFFER = 2;

private String \_title;

private int \_priceCode;

public Goods(String title, int priceCode)

{

\_title = title;

\_priceCode = priceCode;

}

public int getPriceCode()

{

return \_priceCode;

}

public void setPriceCode(int arg)

{

\_priceCode = arg;

}

public String getTitle()

{

return \_title;

}

//---Метод для получения бонусов

public int GetBonus(Item each)

{

switch (each.getGoods().getPriceCode())

{

//Обычный товав

case Goods.REGULAR:

return (int)(each.getQuantity() \* each.getPrice() \* 0.05);

//Товар распродажи

case Goods.SALE:

return (int)(each.getQuantity() \* each.getPrice() \* 0.01);

}

return 0;

}

}

**2) Заменяем в statement GetBonus(each) на each.GetGoods().GetBonus(each).**

//------Определить сумму для каждой строки

//---Получаем скидку

double discount = GetDiscount(each);

//---Получаем бонусы

int bonus = each.getGoods().GetBonus(each);

//---Учитываем скидку и бонусы

double itemSum = GetSum(each);

double sumWithDiscount = itemSum - discount;

double usedBonus = GetUsedBonus(each, sumWithDiscount);

double thisAmount = sumWithDiscount - usedBonus;

**3) Создаем proxy-метод (сокрытие делегирования) GetBonus() в классе Item для избавления от GetGoods().**

//---Метод для сокрытия делегирования GetBonus()

public int GetBonus(Item each)

{

return \_Goods.GetBonus(each);

}

**4) Так как обрабатываемый объект класса Item: each и передаваемый параметр в GetBonus(each) суть одно и тоже, то можно удалить each из списка параметров, а внутри метода заменить его на this.**

//---Метод для сокрытия делегирования GetBonus()

public int GetBonus()

{

return \_Goods.GetBonus(this);

}

**5) Далее можно избавиться от this заменой его 2-мя параметрами:**

//---Метод для сокрытия делегирования GetBonus()

public int GetBonus()

{

return \_Goods.GetBonus(\_quantity, \_price);

}

**Аналогичные операции проделываются и с GetDiscount**

//---Метод для сокрытия делегирования GetDiscount()

public double GetDiscount()

{

return \_Goods.GetDiscount(\_quantity, \_price);

}

**До:**

public String statement()

{

double totalAmount = 0;

int totalBonus = 0;

List<Item>.Enumerator items = \_items.GetEnumerator();

//---Добавляем оглавления

string header = GetHeader(); //заменить result

string itemString="";

while (items.MoveNext())

{

Item each = items.Current;

//------Определить сумму для каждой строки

//---Получаем скидку c учётом бонусов

double discount = GetDiscount(each);

//---Получаем бонусы

int bonus = GetBonus(each);

//---Учитываем скидку и бонусы

double itemSum = GetSum(each);

double sumWithDiscount = itemSum - discount;

double usedBonus = GetUsedBonus(each, sumWithDiscount);

double thisAmount = sumWithDiscount - usedBonus;

//---Вызов метода собирающего все товары

itemString += GetItemString(each, discount+usedBonus, thisAmount, bonus);

//---Добавление суммы и Накопление бонусов

totalAmount += thisAmount;

totalBonus += bonus;

}

//---Добавляем нижний колонтитул

string footer = GetFooter(totalAmount, totalBonus);

//Запомнить бонус клиента

\_customer.receiveBonus(totalBonus);

return header + itemString + footer;

}

public class Goods

{

public const int REGULAR = 0;

public const int SALE = 1;

public const int SPECIAL\_OFFER = 2;

private String \_title;

private int \_priceCode;

public Goods(String title, int priceCode)

{

\_title = title;

\_priceCode = priceCode;

}

public int getPriceCode()

{

return \_priceCode;

}

public void setPriceCode(int arg)

{

\_priceCode = arg;

}

public String getTitle()

{

return \_title;

}

}

public class Item

{

private Goods \_Goods;

private int \_quantity;

private double \_price;

public Item(Goods Goods, int quantity, double price)

{

\_Goods = Goods;

\_quantity = quantity;

\_price = price;

}

public int getQuantity()

{

return \_quantity;

}

public double getPrice()

{

return \_price;

}

public Goods getGoods()

{

return \_Goods;

}

}

**После:**

public String statement()

{

double totalAmount = 0;

int totalBonus = 0;

List<Item>.Enumerator items = \_items.GetEnumerator();

//---Добавляем оглавления

string header = GetHeader(); //заменить result

string itemString="";

while (items.MoveNext())

{

Item each = items.Current;

//------Определить сумму для каждой строки

//---Получаем скидку

double discount = each.GetDiscount();

//---Получаем бонусы

int bonus = each.GetBonus();

//---Учитываем скидку и бонусы

double itemSum = GetSum(each);

double sumWithDiscount = itemSum - discount;

double usedBonus = GetUsedBonus(each, sumWithDiscount);

double thisAmount = sumWithDiscount - usedBonus;

//---Вызов метода собирающего все товары

itemString += GetItemString(each, discount+usedBonus, thisAmount, bonus);

//---Добавление суммы и Накопление бонусов

totalAmount += thisAmount;

totalBonus += bonus;

}

//---Добавляем нижний колонтитул

string footer = GetFooter(totalAmount, totalBonus);

//Запомнить бонус клиента

\_customer.receiveBonus(totalBonus);

return header + itemString + footer;

}

public class Goods

{

public const int REGULAR = 0;

public const int SALE = 1;

public const int SPECIAL\_OFFER = 2;

private String \_title;

private int \_priceCode;

public Goods(String title, int priceCode)

{

\_title = title;

\_priceCode = priceCode;

}

public int getPriceCode()

{

return \_priceCode;

}

public void setPriceCode(int arg)

{

\_priceCode = arg;

}

public String getTitle()

{

return \_title;

}

//---Метод для получения бонусов

public int GetBonus(int \_quantity, double \_price)

{

switch (\_priceCode)

{

//Обычный товав

case Goods.REGULAR:

return (int)(\_quantity \* \_price \* 0.05);

//Товар распродажи

case Goods.SALE:

return (int)(\_quantity \* \_price \* 0.01);

}

return 0;

}

//---Метод для получения скидки

public double GetDiscount(int \_quantity, double \_price)

{

double discount = 0;

switch (\_priceCode)

{

//Обычный товав

case Goods.REGULAR:

if (\_quantity > 2)

discount = (\_quantity \* \_price) \* 0.03; // 3%

break;

//Специальное предложение

case Goods.SPECIAL\_OFFER:

if (\_quantity > 10)

discount = (\_quantity \* \_price) \* 0.005; // 0.5%

break;

//Товар распродажи

case Goods.SALE:

if (\_quantity > 3)

discount = (\_quantity \* \_price) \* 0.01; // 0.1%

break;

}

return discount;

}

}

public class Item

{

private Goods \_Goods;

private int \_quantity;

private double \_price;

public Item(Goods Goods, int quantity, double price)

{

\_Goods = Goods;

\_quantity = quantity;

\_price = price;

}

public int getQuantity()

{

return \_quantity;

}

public double getPrice()

{

return \_price;

}

public Goods getGoods()

{

return \_Goods;

}

//---Метод для сокрытия делегирования GetBonus()

public int GetBonus()

{

return \_Goods.GetBonus(\_quantity, \_price);

}

//---Метод для сокрытия делегирования GetDiscount()

public double GetDiscount()

{

return \_Goods.GetDiscount(\_quantity, \_price);

}

}

**Контрольные вопросы**

**1) В чем суть рефакторинга «Выделение класса»?**

Предполагает разделение ответственностей между классами, когда один класс становится слишком сложным и перегруженным функциями и данными. Создается новый класс, в который переносятся соответствующие поля и методы, чтобы каждый класс представлял собой ясно очерченную абстракцию и выполнял отчетливо определенный набор обязанностей.

**2) В чем суть рефакторинга «Встраивание класса»?**

Применяется, когда класс выполняет недостаточно функций и его содержимое можно перенести в другой класс. Этот процесс включает перемещение всех функций и полей из одного класса в другой и последующее удаление исходного класса. Этот метод рефакторинга используется для упрощения архитектуры программы, уменьшения количества классов и концентрации связанных функций в одном месте, повышая тем самым читаемость и облегчая поддержку кода.

**3) В чем суть рефакторинга «Замена магического числа символической константой»?**

Заключается в замене числовых литералов в коде, имеющих специфическое значение, на именованные константы. Это улучшает читаемость и поддерживаемость кода, облегчая понимание значения чисел и упрощая их изменение в будущем.

**4) В чем суть рефакторинга «Замена кода типа подклассами»?**

Используется, когда класс содержит поле типа, которое влияет на его поведение через условные операторы. Вместо использования числовых или строковых кодов для определения поведения, создаются подклассы для каждого значения кода типа, и полиморфизм используется для соответствующего изменения поведения. Это улучшает структуру кода, делает его более понятным и упрощает добавление новых типов.

Было:

class Employee {

private int type;

public const int ENGINEER = 1;

public const int SALESMAN = 2;

public const int MANAGER = 3;

public Employee(int type) {

this.type = type;

}

}

Стало:

abstract class Employee {

public static Employee Create(int type) {

switch (type) {

case ENGINEER: return new Engineer();

case SALESMAN: return new Salesman();

case MANAGER: return new Manager();

default: throw new ArgumentOutOfRangeException("Недопустимое значение кода типа");

}

}

public abstract int GetEmployeeType();

}

class Engineer : Employee {

public override int GetEmployeeType() {

return Employee.ENGINEER;

}

}

class Salesman : Employee {

public override int GetEmployeeType() {

return Employee.SALESMAN;

}

}

class Manager : Employee {

public override int GetEmployeeType() {

return Employee.MANAGER;

}

}

**5) В чем суть рефакторинга «Замена кода типа состоянием/ стратегией»?**

Используется, когда в классе присутствует код типа, влияющий на его поведение, но невозможно применить подклассы из-за изменяемости типа или других ограничений. Вместо использования кода типа создается иерархия классов, представляющих различные состояния или стратегии, и объект класса делегирует поведение этим объектам. Это улучшает гибкость и расширяемость программы, позволяя добавлять новые типы поведения без изменения существующего кода.

Было:

class Employee {

private int type;

public const int ENGINEER = 1;

public const int SALESMAN = 2;

public const int MANAGER = 3;

public Employee(int type) {

this.type = type;

}

public int PayAmount() {

switch (type) {

case ENGINEER: return monthlySalary;

case SALESMAN: return monthlySalary + commission;

case MANAGER: return monthlySalary + bonus;

default: throw new Exception("Incorrect Employee");

}

}

}

Стало:

abstract class EmployeeType {

public abstract int PayAmount(Employee emp);

}

class Engineer : EmployeeType {

public override int PayAmount(Employee emp) {

return emp.MonthlySalary;

}

}

class Salesman : EmployeeType {

public override int PayAmount(Employee emp) {

return emp.MonthlySalary + emp.Commission;

}

}

class Manager : EmployeeType {

public override int PayAmount(Employee emp) {

return emp.MonthlySalary + emp.Bonus;

}

}

class Employee {

public int MonthlySalary { get; set; }

public int Commission { get; set; }

public int Bonus { get; set; }

private EmployeeType type;

public Employee(EmployeeType type) {

this.type = type;

}

public int PayAmount() {

return type.PayAmount(this);

}

}

**6) В чем суть рефакторинга «Замена подкласса полями»?**

Применяется, когда подклассы отличаются только константными значениями. Вместо использования подклассов, данные значения переносятся в поля родительского класса, а подклассы удаляются. Это упрощает структуру кода, уменьшая его сложность и количество классов.

Было:

abstract class Person {

public abstract bool IsMale();

public abstract char GetCode();

}

class Male : Person {

public override bool IsMale() {

return true;

}

public override char GetCode() {

return 'M';

}

}

class Female : Person {

public override bool IsMale() {

return false;

}

public override char GetCode() {

return 'F';

}

}

Стало:

class Person {

private readonly bool isMale;

private readonly char code;

private Person(bool isMale, char code) {

this.isMale = isMale;

this.code = code;

}

public bool IsMale() {

return isMale;

}

public char GetCode() {

return code;

}

public static Person CreateMale() {

return new Person(true, 'M');

}

public static Person CreateFemale() {

return new Person(false, 'F');

}

}

**7) В чем суть рефакторинга «Замена условного оператора полиморфизмом»?**

Применяется, когда поведение объекта зависит от его типа, выраженного через условный оператор. Вместо использования условных операторов, поведение делегируется подклассам с помощью переопределения методов. Это упрощает код, сокращая его зависимость от условных операторов и улучшая его расширяемость.

Было:

class Employee {

private int type;

public const int ENGINEER = 0;

public const int SALESMAN = 1;

public const int MANAGER = 2;

public Employee(int type) {

this.type = type;

}

public int PayAmount() {

switch (type) {

case ENGINEER:

return MonthlySalary;

case SALESMAN:

return MonthlySalary + Commission;

case MANAGER:

return MonthlySalary + Bonus;

default:

throw new Exception("Неправильный тип сотрудника");

}

}

}

Стало:

abstract class Employee {

public abstract int PayAmount();

}

class Engineer : Employee {

public override int PayAmount() {

return MonthlySalary;

}

}

class Salesman : Employee {

public override int PayAmount() {

return MonthlySalary + Commission;

}

}

class Manager : Employee {

public override int PayAmount() {

return MonthlySalary + Bonus;

}

}

**Заключение**

В результате выполнения лабораторной работы были изучены базовые техники рефакторинга позволяющие различным образом манипулировать методами между классами программы.

Изучить базовые техники рефакторинга позволяющие различным образом манипулировать классами программы.