Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»**

**(ФГБОУ ВО «КубГТУ»)**

**Институт компьютерных систем и информационной безопасности**

**Кафедра информационных систем и программирования**

**ОТЧЁТ**

**Дисциплина: Тестирование и отладка программного обеспечения**

**Тема: «Базовые техники рефакторинга.**

**Межклассовые рефакторинги»**

**Лабораторная: 3**

Работу выполнил: Басма Нассиф Хуссейн

Направление подготовки: 09.03.04 Программная инженерия

Преподаватель: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Е. В. Степанова

**Цель работы:** изучить базовые техники рефакторинга позволяющие различным образом манипулировать классами программы.

**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;

}

}

**Задание**

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

**Порядок выполнение**

1. Создать подклассы (со спецификатором public для доступа к ним из тестов)

public class SaleGoods {}

public class SpecialGoods {}

public class RegularGoods {}

1. выделяем из ветвей switch методы GetBonus и GetDiscount, и переносим их в подклассы с модификатором override.

public class RegularGoods : Goods

{

public RegularGoods(string title, int priceCode) : base(title, priceCode)

{

}

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

{

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

}

// Метод для получения скидки

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

{

if (\_quantity > 2)

return (\_quantity \* \_price) \* 0.03; // 3%

return 0;

}

}

public class SaleGoods : Goods

{

public SaleGoods(string title, int priceCode) : base(title, priceCode)

{

}

// Метод для получения бонуса

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

{

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

}

// Метод для получения скидки

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

{

if (\_quantity > 3)

return (\_quantity \* \_price) \* 0.01; // 0.1%

return 0;

}

}

public class SpecialGoods : Goods

{

public SpecialGoods(string title, int priceCode) : base(title, priceCode)

{

}

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

{

return 0;

}

// Метод для получения скидки

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

{

if (\_quantity > 10)

return (\_quantity \* \_price) \* 0.005; // 0.5%

return 0;

}

}

1. В базовом классе Goods оставляем методы с возвращением значений по умолчанию (нулями) и добавляем им модификатор virtual.

// Метод для получения бонуса

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

{

return 0;

}

// Метод для получения скидки

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

{

return 0;

}

1. В базовом классе меняем уровень доступа для полей с private на protected для получения доступа к ним в производных классах.

protected String \_title;

protected int \_priceCode;

1. После этого программа должна компилироваться, но тесты не проходить (так как все время возвращаются нули вместо известных скидок)

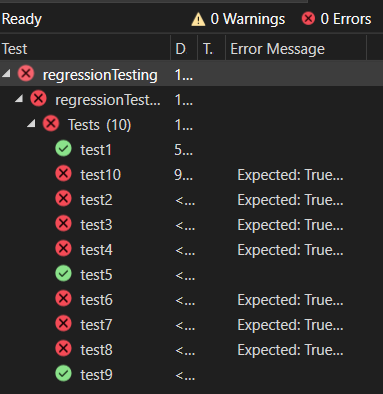


Рисунок 1 – Не порождённые тесты

1. Заменяем места создания экземпляров Goods(name, type) на [type]Goods(name, type)

[Test]

public void test1()

{

var result = bill.statement();

var expectedFooter = "Сумма счета составляет 0\nВы заработали 0 бонусных балов";

Assert.IsTrue(result.Contains(expectedFooter));

}

[Test]

public void test2()

{

bill.addGoods(new Item(new RegularGoods("Товар 1", Goods.REGULAR), 1, 100));

var result = bill.statement();

Assert.IsTrue(result.Contains("Сумма счета составляет 100\n"));

Assert.IsTrue(result.Contains("Вы заработали 5 бонусных балов"));

}

[Test]

public void test3()

{

bill.addGoods(new Item(new SpecialGoods("Товар 2", Goods.SPECIAL\_OFFER), 11, 50));

var result = bill.statement();

Assert.IsTrue(result.Contains("Сумма счета составляет 537,25"));

Assert.IsTrue(result.Contains("Вы заработали 0 бонусных балов"));

}

[Test]

public void test4()

{

bill.addGoods(new Item(new SaleGoods("Товар со скидкой", Goods.SALE), 11, 200));

var result = bill.statement();

Assert.IsTrue(result.Contains("Сумма счета составляет 2178"));

Assert.IsTrue(result.Contains("Вы заработали 22 бонусных балов"));

}

[Test]

public void test5()

{

bill.addGoods(new Item(new SpecialGoods("Товар 2", Goods.SPECIAL\_OFFER), 5, 50));

var result = bill.statement();

Assert.IsTrue(result.Contains("Сумма счета составляет 240"));

Assert.IsTrue(result.Contains("Вы заработали 0 бонусных балов"));

}

[Test]

public void test6()

{

bill.addGoods(new Item(new SaleGoods("Товар со скидкой", Goods.SALE), 2, 200));

var result = bill.statement();

Assert.IsTrue(result.Contains("Сумма счета составляет 400"));

Assert.IsTrue(result.Contains("Вы заработали 4 бонусных балов"));

}

[Test]

public void test7()

{

bill.addGoods(new Item(new RegularGoods("Товар 1", Goods.REGULAR), 3, 100));

var result = bill.statement();

Assert.IsTrue(result.Contains("Сумма счета составляет 291\n"));

Assert.IsTrue(result.Contains("Вы заработали 15 бонусных балов"));

}

[Test]

public void test8()

{

bill.addGoods(new Item(new RegularGoods("Товар 1", Goods.REGULAR), 2, 100));

var result = bill.statement();

Assert.IsTrue(result.Contains("Сумма счета составляет 200\n"));

Assert.IsTrue(result.Contains("Вы заработали 10 бонусных балов"));

}

[Test]

public void test9()

{

bill.addGoods(new Item(new RegularGoods("Товар 1", Goods.REGULAR), 0, 100));

var result = bill.statement();

Assert.IsTrue(result.Contains("Сумма счета составляет 0"));

Assert.IsTrue(result.Contains("Вы заработали 0 бонусных балов"));

}

[Test]

public void test10()

{

bill.addGoods(new Item(new RegularGoods("Товар 1", Goods.REGULAR), 6, 100));

bill.addGoods(new Item(new SaleGoods("Товар со скидкой", Goods.SALE), 11, 200));

bill.addGoods(new Item(new SpecialGoods("Товар 2", Goods.SPECIAL\_OFFER), 15, 50));

var result = bill.statement();

Assert.IsTrue(result.Contains("Сумма счета составляет 3496,25"));

Assert.IsTrue(result.Contains("Вы заработали 52 бонусных балов"));

}

1. После этого тесты должны проходить.

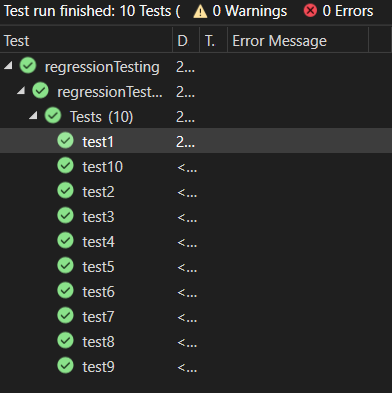


Рисунок 2 – Исправленные тесты

1. Удаляем параметр type из конструкторов, а затем (при помощи автозамены) из мест создания объектов.

public Goods(String title)

{

\_title = title;

}

public SpecialGoods(string title) : base(title) {}

public SaleGoods(string title) : base(title) {}

public RegularGoods(string title) : base(title) {}

1. Удаляем метод GetTypeCode
2. В методе statement заменяем его на проверку типа

//---Метод получения использованных бонусов

public double GetUsedBonus(Item each, double sumWithDiscount)

{

double usedBonus = 0;

if (each.getGoods().GetType() == typeof(RegularGoods))

{

if (each.getQuantity() > 5)

usedBonus = \_customer.useBonus((int)(sumWithDiscount));

} else if (each.getGoods().GetType() == typeof(SpecialGoods))

{

if (each.getQuantity() > 1)

usedBonus = \_customer.useBonus((int)(sumWithDiscount));

}

return usedBonus;

}

1. Удаляем поля REGULAR /SALE/SPECIAL

public class Goods

{

protected String \_title;

protected int \_priceCode;

public Goods(String title)

{

\_title = title;

}

public void setPriceCode(int arg)

{

\_priceCode = arg;

}

public String getTitle()

{

return \_title;

}

// Метод для получения бонуса

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

{

return 0;

}

// Метод для получения скидки

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

{

return 0;

}

}

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

**До:**

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 бонуса

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

{

switch (\_priceCode)

{

case Goods.REGULAR:

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

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)

{

case Goods.REGULAR:

if (\_quantity > 2)

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

break;

case Goods.SPECIAL\_OFFER:

if (\_quantity > 10)

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

break;

case Goods.SALE:

if (\_quantity > 3)

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

break;

}

return discount;

}

}

public double GetUsedBonus(Item each, double sumWithDiscount)

{

double usedBonus = 0;

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

{

//Обычный товав

case Goods.REGULAR:

if (each.getQuantity() > 5)

usedBonus = \_customer.useBonus((int)(sumWithDiscount));

break;

//Специальное предложение

case Goods.SPECIAL\_OFFER:

if (each.getQuantity() > 1)

usedBonus = \_customer.useBonus((int)(sumWithDiscount));

break;

}

return usedBonus;

}

**После:**

public class Goods

{

protected String \_title;

protected int \_priceCode;

public Goods(String title)

{

\_title = title;

}

public void setPriceCode(int arg)

{

\_priceCode = arg;

}

public String getTitle()

{

return \_title;

}

// Метод для получения бонуса

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

{

return 0;

}

// Метод для получения скидки

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

{

return 0;

}

}

public class RegularGoods : Goods

{

public RegularGoods(string title) : base(title)

{

}

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

{

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

}

// Метод для получения скидки

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

{

if (\_quantity > 2)

return (\_quantity \* \_price) \* 0.03; // 3%

return 0;

}

}

public class SaleGoods : Goods

{

public SaleGoods(string title) : base(title)

{

}

// Метод для получения бонуса

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

{

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

}

// Метод для получения скидки

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

{

if (\_quantity > 3)

return (\_quantity \* \_price) \* 0.01; // 0.1%

return 0;

}

}

public class SpecialGoods : Goods

{

public SpecialGoods(string title) : base(title)

{

}

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

{

return 0;

}

// Метод для получения скидки

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

{

if (\_quantity > 10)

return (\_quantity \* \_price) \* 0.005; // 0.5%

return 0;

}

}

// BILL

public double GetUsedBonus(Item each, double sumWithDiscount)

{

double usedBonus = 0;

if (each.getGoods().GetType() == typeof(RegularGoods))

{

if (each.getQuantity() > 5)

usedBonus = \_customer.useBonus((int)(sumWithDiscount));

} else if (each.getGoods().GetType() == typeof(SpecialGoods))

{

if (each.getQuantity() > 1)

usedBonus = \_customer.useBonus((int)(sumWithDiscount));

}

return usedBonus;

}

**Заключение:**

Были изучены базовые техники рефакторинга позволяющие различным образом манипулировать классами программы.