Государственный университет – Высшая школа экономики

Факультет Бизнес-Информатики

Кафедра Основ информатики
и прикладного программного обеспечения

Объектно-ориентированный анализ и программирование на языке C# (C_Sharp)

Материалы к **15**-й лекции

Проф. Забудский Е.И.

Москва 2007

Лекция 15

Темы 9 и 10

1. Введение в Объектно-Ориентированное Проектирование (см. лекцию 14)

2. Объектно-ориентированный подход к созданию пользовательского интерфейса (UI)

3. Применение тестирования для создания надежного ПО

4. Реалии индустрии и ОО программирование

Резюме к курсу ООАП

NB	**Задание # 3** (продолжение заданий # # 1 и 2). **Представить в оконном варианте (GUI) программный код, разработанный в соответствии с заданиями ## 1 и 2 /см. в задании пункты 3-а и 3-б на с. 56 в Материалах к 12-й лекции/**
	Срок представления результатов на практическом занятии – 04 (06) апреля 2007 г.

Уважаемые студенты!

Основная цель, которую необходимо достигнуть в результате изучения дисциплины Объектно-ориентированный анализ и программирование – научиться разрабатывать компьютерные модели реальных и концептуальных систем соответствующих направлению Бизнес-информатика.

Необходимым условием усвоения дисциплины является ВАША **самостоятельная** работа

Советую Вам все материалы, подготовленные мной к лекциям и практическим занятиям, распечатать и прорабатывать их! Приведенные C#-программы реализовать в среде MS VS .NET 2005 и разобраться в них.

// **КОММЕНТАРИЙ**. На сайте http://www.firststeps.ru/ “Первые шаги” представлено много интересных обучающих материалов по различным интегрированным средам и языкам программирования, в том числе представлены **C#** и платформа **.NET** (step by step).

Содержание

2. Объектно-ориентированный подход к программированию пользовательского интерфейса

Пользовательский интерфейс обеспечивает взаимодействие между пользователем и системой. Почти каждая современная система имеет ту или иную форму пользовательского интерфейса: **1)** графическую, **2)** командной строки, **3)** телефонную или **4)** речевую. (В некоторых системах сочетаются все четыре типа**!**) В любом случае, необходимо уделить особое внимание разработке и внедрению пользовательского интерфейса. При разработке интерфейса объектно-ориентированное программирование может быть столь же полезно, как и при разработке других аспектов системы.

Сегодня рассмотрим:

- применение ОО программирования к разработке пользовательского интерфейса;
- важность развязки пользовательского интерфейса;
- модели, помогающие развязать пользовательский интерфейс.

2.1. Объектно-ориентированное программирование и пользовательский интерфейс

Процесс разработки и программирования пользовательского интерфейса по существу не отличается от процесса разработки и программирования других частей системы.

Создавая пользовательский интерфейс, иногда приходится изучать некоторые новые функции программного интерфейса приложения (**API**-функции; **A**pplication **P**rogramming **I**nterface). В остальном же к разработке пользовательского интерфейса применимы те же принципы объектно-ориентированного подхода, что и к разработке других частей системы.

Вы узнаете, как приступить к разработке пользовательского интерфейса с точки зрения разработчика. Именно с точки зрения разработчика вы будете проектировать и реализовывать классы, создающие и поддерживающие пользовательский интерфейс.

Лекция **не** охватывает общую идею разработки пользовательского интерфейса**:** проектирование пользовательского интерфейса включает всестороннее изучение способов предоставления пользователю возможностей программы. Общая методология разработки пользовательского интерфейса лежит **вне** пределов программирования; она относится скорее к графическому искусству и психологии. Специальная группа по исследованию взаимодействия между человеком и машиной (**Special Interest Group on Computer-Human Interaction**, **SIGCHI, www.sigchi.org**) Ассоциации по вычислительной технике (**Association for Computing Machinery, ACM**) является отличным источником информации о разработке и использовании пользовательских интерфейсов.

При разработке пользовательского интерфейса *должны* применяться те же принципы объектно-ориентированного подхода, что и при разработке других частей системы! Слишком часто пользовательские интерфейсы просто "навешивают" на систему после ее создания.

Коды пользовательского интерфейса должны быть столь же объектно-ориентированными, что и коды остальных частей программы. При разработке пользовательского интерфейса необходимо правильно использовать: **1)** инкапсуляцию, **2)** наследование и **3)** полиморфизм.

При выполнении объектно-ориентированного анализа (**ООА**) и объектно-ориентированного проектирования (**ООПр**) не забывайте о пользовательском интерфейсе. Если объектно-ориентированный анализ (**ООА**) или объектно-ориентированное проектирование (**ООПр**) выполнены неправильно, то могут быть упущены существенные требования, предъявляемые к создаваемому интерфейсу, и в результате обнаружится, что интерфейс: **1)** недостаточно гибок для желаемого уровня функциональности, или **2)** его не удастся приспособить к будущим изменениям.

2.2. Значение развязки пользовательских интерфейсов

В системе подчас необходимо предусмотреть несколько различных, зачастую несвязанных пользовательских интерфейсов. Так, система снабжения продовольствием предоставляет покупателям возможность размещать заказы: **1)** по телефону, **2)** через **Internet**, **3)** персональные информационные устройства (**P**ersonal **D**igital **A**ssistant**, PDA**) или **4)** непосредственно через настольную прикладную систему, используемую в столе заказов. Каждый из интерфейсов обращается к одной и той же системе, но методы обращения и способы отображения информации различны.

Требования к пользовательскому интерфейсу могут постоянно изменяться. Улучшения системы являются результатом: **1)** добавления новых возможностей и **2)** устранения обнаруженных пользователями недостатков. А чтобы предоставить доступ к новым возможностям и устранить обнаруженные дефекты, пользовательский интерфейс приходится обновлять постоянно. Поэтому **пользовательский интерфейс должен быть спроектирован так, чтобы он был достаточно гибким и легко адаптировался к переменам**.

Чтобы добиться необходимой степени гибкости, **лучше всего разработать систему так, чтобы она была** полностью независимой **от пользовательского интерфейса**. Ведь если система независима от пользовательского интерфейса, **1)** то к ней можно добавить новый вид интерфейса, **2)** либо изменить уже существующий интерфейс**;** **причем изменения в самой системе не потребуются.** Более того, систему, независимую от пользовательского интерфейса, можно тестировать даже тогда, когда разработка интерфейса еще не завершена. Именно независимость системы от пользовательского интерфейса облегчает обнаружение его недостатков и ошибок в самой системе.

Объектно-ориентированное программирование идеально подходит для решения вышеуказанных проблем. Благодаря ему удается разумно распределить функции между частями программы, что позволяет уменьшить влияние изменений на не связанные с этими изменениями части кода. Правильно распределив выполняемые функции, можно в любое время добавить к программе любой интерфейс, не внося изменений в основную программу. Просто **не внедряйте код интерфейса в саму систему**. **Два этих кода** должны быть независимыми.

Рассмотрим пример, демонстрирующий неправильное разделение пользовательского интерфейса. В **листинге 1** показано, как **не нужно** писать пользовательский интерфейс.

Листинг 1. VisualBankAccount.CS

```
1   /* В данной программе обработки банковских счетов в одном классе смешаны:
       1) модель, 2) представление и 3) контроллер. Это неправильное решение, так как
       система и GUI взаимозависимы, и, следовательно, невозможно элегантно вносить
       изменения в GUI и в систему */
2   using System;                                                          Листинг 1
3   using System.Drawing;
4   using System.Windows.Forms;
5
6   namespace WinAppl_OOP21_c304_306_GUI
7   {
8      public class VisualBankAccount : Form          ////////////////////////////////////// начало класса
9      {
10        private double balance;
```

```
11                                                  ////////////////////  Создание GUI ////////////////////
12          Button btnDeposit  = new Button();                  // элементы GUI
13          Button btnWithdraw = new Button();                  // - " - " - " -
14          Label lblBalance   = new Label();                   // - " - " - " -
15          TextBox txtAmount  = new TextBox();                 // - " - " - " -
16
17                                          // 1-й ШАГ - объявление объектов  класса EventHandler
18          EventHandler DepositButtonHandler;
19          EventHandler WithdrawButtonHandler;
20
21          public VisualBankAccount()                  // Создание GUI – конструктор класса
22          {
23              setBalance(0);
24                                  // 2-й ШАГ -  создание объектов (экземпляров) класса EventHandler
25              DepositButtonHandler  = new EventHandler(btnDeposit_Click);     // см. строку 68…
26              WithdrawButtonHandler = new EventHandler(btnWithdraw_Click); // см. строку 74…
27
28                                          // 3-й ШАГ - регистрация объекта класса EventHandler
29              btnDeposit.Click  += DepositButtonHandler;
30              btnWithdraw.Click += WithdrawButtonHandler;
31
32              lblBalance.Location = new Point(35, 15);
33              lblBalance.Size     = new Size(200, 20);
34
35              txtAmount.Location = new Point(35, 50);
36              txtAmount.Size     = new Size(200, 20);
37
38              btnDeposit.Location = new Point(35, 100);
39              btnDeposit.Size     = new Size(100, 20);
40              btnDeposit.Text     = "Deposit";
41
42              btnWithdraw.Location = new Point(135, 100);
43              btnWithdraw.Size     = new Size(100, 20);
44              btnWithdraw.Text     = "Withdraw";
45
46              Controls.Add(btnDeposit);
47              Controls.Add(btnWithdraw);
48              Controls.Add(lblBalance);
49              Controls.Add(txtAmount);
50
```

```
61          Text = "Банковский счет";
62          Size = new Size(285, 185);
63
64          FormBorderStyle = FormBorderStyle.FixedSingle;
65      }                                   ////////////////////  Завершение создания GUI ////////////////////
66
67                                  // 4-й ШАГ - программный код обработчика события Deposit
68      private void btnDeposit_Click(Object sender, System.EventArgs e)
69      {
70          depositFunds(double.Parse(txtAmount.Text));                 // строки 80…
71      }
72
73                                  // 4-й ШАГ - программный код обработчика события Withdraw
74      private void btnWithdraw_Click(Object sender, System.EventArgs e)
75      {
76          withdrawFunds(double.Parse(txtAmount.Text));                // строки 85…
77      }
78
79                              //////// Компьютерная модель банковского счета (система) //////////
80      public void depositFunds(double amount)                     // положить деньги на счет
81      {
82          setBalance(getBalance() + amount);
83      }
84
85      public double withdrawFunds(double amount)                  // снять деньги со счета
86      {
87          if (amount >= getBalance())
88          {
89              amount = getBalance();
90          }
91          setBalance(getBalance() - amount);
92          return amount;
93      }
94
95      protected void setBalance(double newBalance)                // установить остаток-баланс
96      {
97          balance = newBalance;
98          lblBalance.Text = ("Баланс: " + balance);
99      }
100
```

```
101         public double getBalance()                                  // запрос остатка-баланса
102         {
103             return balance;
104         }
105     }                       ////////////////////// конец класса VisualBankAccount ////////////////////////////
106
107     class BalanceGUI                                                ////////////////////////////////////// начало класса
108     {
109         public static void Main(string[] args)
110         {                                                           // Создание экз-ра ba класса VisualBankAccount
111             VisualBankAccount ba = new VisualBankAccount();
112
113     /* посредством вызова метода Run() класса Application, создается выводящий форму
           объект. Обращение к м-ду Run() сообщает C#, что форма должна оставаться видимой,
                                                                    до ее закрытия пользователем*/
114             Application.Run(ba);
115         }
116     }                           ////////////////////// конец класса BalanceGUI ////////////////////////////
117 }
```

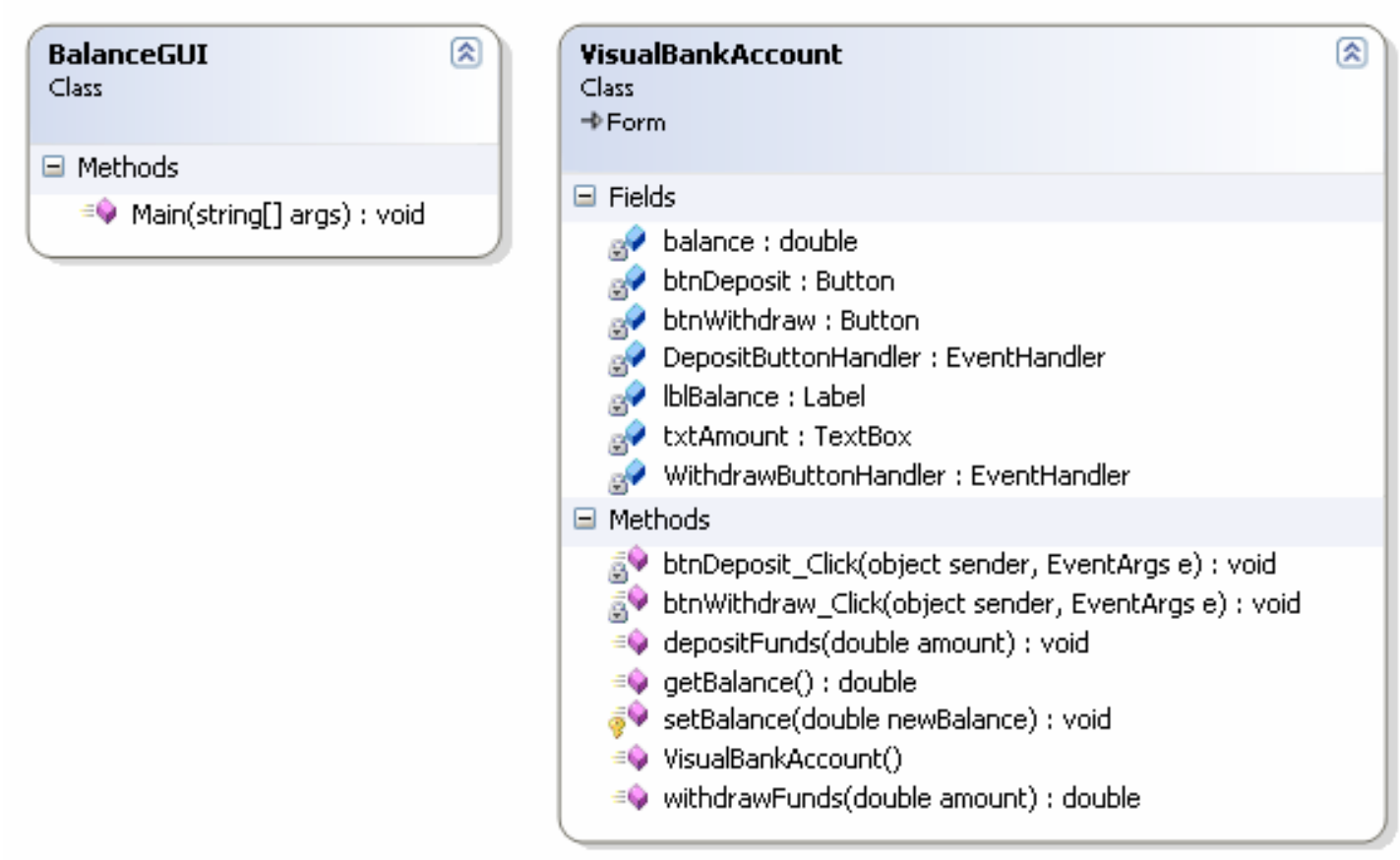


Рис. 1. Диаграмма классов программы на **листинге 1**

Класс **VisualbankAccount** использует для своего визуального представления библиотеку **Form**. Каждый объектно-ориентированный язык (**C++**, **Java**, **C#**) располагает необходимыми библиотеками для создания и представления графических интерфейсов пользователя.

Главные элементы графического интерфейса пользователя, такие как кнопки (**Button**), надписи (**Label**) и поля ввода (**TextBox**), создаются из соответствующих стандартных классов (**строки 12…15**).

Всякий раз, когда элемент графического интерфейса пользователя генерирует событие (обычно это происходит по щелчку мышью), оно будет передаваться соответствующему обработчику событий (**строки 67… и 73…**). Следовательно, в данной программе класс **VisualBankAccount** ведет себя как обработчик события. Когда он получает сообщение от одной из кнопок он выполняет нужное действие — **кладет** или **снимает** деньги.

VisualBankAccount реализует все функции класса **BankAccount**, рассмотренного на предыдущих занятиях **(лекция 3, с.38,сл. и лекция 7, с.38,сл.)** и, кроме того, **знает, как представить себя визуально** (**рис. 2**).

Когда пользователь вводит сумму и нажимает кнопку **Deposit** (**Положить**) или **Withdraw** (**Снять**), значение извлекается из поля ввода и вызывается один из методов **depositFunds()** (строки **70…**) или **withdrawFunds()** (строки **76…**).

Поскольку класс **VisualBankAccount** представляет собой панель (**Form**), он может использоваться в любом графическом интерфейсе пользователя, разработанном с помощью языка **C#**. **К сожалению**, пользовательский интерфейс **не отделен** от класса банковских счетов. Такая тесная взаимосвязь: **1)** не позволяет обрабатывать банковские счета с помощью других видов пользовательского интерфейса, а также **2)** не позволяет реализовать другой пользовательский интерфейс без переделки самого класса **VisualBankAccount**. **Фактически для каждого типа пользовательского интерфейса необходима отдельная версия класса**.

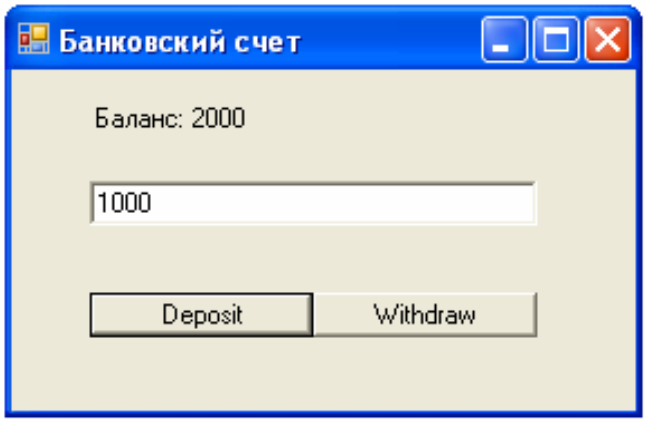


Рис. 2. Представление класса **VisualBankAccount** (результат работы программы на **листинге 1**)

2.3. Развязка пользовательского интерфейса с помощью шаблона модель / вид / контроллер

Шаблон проектирования **модель/вид/контроллер** (**Model View Controller — MVC**) предлагает такой подход к разработке пользовательского интерфейса, при котором **основная система полностью от него независима**.

Модель/вид/контроллер — это всего лишь один из возможных методов разработки объектно-ориентированного пользовательского интерфейса. Существуют и другие приемлемые методы,

но шаблон **модель/вид/контроллер** — метод, прошедший проверку временем и популярный среди разработчиков программного обеспечения. При разработке пользовательского интерфейса, особенно для **Web**, вы, скорее всего, будете иметь дело именно с шаблоном **модель / вид / контроллер** (**MVC**).

Альтернативами шаблону **модель/вид/контроллер** являются **Модель / Документ / Представление** (**Document / View / Model**), предложенная в библиотеке базовых классов **Microsoft** (**Microsoft Foundation Classes, MFC**)[1] и шаблон проектирования **Presentation Abstraction Control (PAC)**. Полное представление об этих альтернативах можно получить из книги Pattern-Oriented Software Architecture A System of Patterns (Архитектура модельно-ориентированного программного обеспечения: система шаблонов), написанной Франком Бушманном (Frank Buschmann) в соавторстве с другими авторами и изданной в издательстве **Wiley** (**ISBN 0-471-95869-7**).

Шаблон **модель/вид/контроллер** (**MVC**) развязывает пользовательский интерфейс, разбивая его проект на следующие части.

- **Модель**, представляющая систему……..**с.10**.
- **Вид**, отображающий модель…… **с.18.**
- **Контроллер** [2], обрабатывающий данные, вводимые пользователем……**с.19**.

Каждая из частей этой триады **MVC** имеет свой уникальный круг обязанностей.

---

[1] **Библиотека базовых классов Microsoft** — библиотека классов, поддерживающая разработку приложений для **Microsoft Windows**; поставляется с **Visual C++** и другими компиляторами.
[2] **Контроллер** — **процессор (программное средство)** для обмена данными с какой-либо подсистемой.

2.3.1. Модель (строки 126…182)

Модель ответственна за:

- доступ к функциям ядра системы;
- доступ к информации о состоянии системы;
- систему уведомления об изменении состояния.

Модель — это слой триады **MVC**, который управляет поведением ядра и состоянием системы. Модель отвечает**: 1)** на запросы **о ее состоянии**, поступающие от вида и контроллера, а также **2)** на запросы **изменения состояния**, поступающие от контроллера.

Система может иметь несколько различных моделей. К примеру, банковская система может быть построена на основе: **1)** модели **счета** и **2)** модели **кассира**. Лучше всего распределить ответственность между несколькими небольшими моделями.

Модель — это всего лишь объект, который представляет систему.

Контроллер — это тот слой триады **MVC**, который интерпретирует данные, вводимые пользователем. В ответ на данные, вводимые пользователем, **контроллер посылает команду модели или виду** для изменения состояния или выполнения какого-либо действия.

Вид — это тот слой триады **MVC**, который отображает графическое или текстовое представление модели. Вид получает всю информацию о состоянии модели от самой модели.

В любом случае **модель** совершенно не подозревает о том, что **вид** или **контроллер** **совершают вызов метода**. Модель знает только, что объект вызывает один из ее методов. Единственной

связью между моделью и пользовательским интерфейсом является **связь через службу уведомления об изменении состояния системы**.

Если вид или контроллер заинтересованы в уведомлении об изменении состояния, они регистрируются в модели. При изменении состояния модели, она просмотрит список зарегистрированных объектов (их часто еще называют слушателями [3], приемниками, приемниками информации (**listener**), обработчиками, наблюдателями, а также блоками наблюдения (**observer**)), и проинформирует каждый объект об изменении состояния. Для создания службы уведомления модель, как правило, задействует шаблон **Observer** (**наблюдатель, блок наблюдения**).

2.3.1.1. Шаблон наблюдателя (Observer)

Шаблон **Observer** предлагает проект механизма **публикации / подписки** среди объектов. Шаблон **Observer** позволяет **объекту** (**наблюдателю**) зарегистрировать свою заинтересованность в другом **объекте** (**наблюдаемом**). Каждый раз, когда наблюдаемый хочет уведомить своих наблюдателей об изменении, он применит к наблюдателю метод **update()** **(строка 10).**

В **листинге 2** определяется интерфейс **Observer**. Все **наблюдатели (view)**, которые хотят зарегистрироваться у наблюдаемого объекта, должны реализовать интерфейс **Observer (см. в листинге 3 строки 80… и 119…)**.

[3] **Слушатель** — это приемник (информации), т.е. процесс. В качестве слушателя может выступать, например, процесс, обеспечивающий контроль правильности выражений, вводимых с клавиатуры.

Листинг 2. Observer.CS

```
8   public interface Observer
9   {                    // Наблюдатель (обозреватель)
10  void update();
11  }
```

Наблюдаемый (**модель**) должен иметь метод, с помощью которого наблюдатели (**view**) могут зарегистрировать (**строки 162…**) свою заинтересованность и отменить (**строки 168…**) ее в модификаторах. В **листинге 3** представлен класс **BankAccountModel / модель – система /** **(строки 126…182)** (и другие классы), в котором: **1)** использован **шаблон наблюдателя** (**Observer**), **2)** а также представлены системные **функции банковского счета** (**строки 131…, 136…, 141…, 146…, 152…**).

Применение шаблона **модель/вид/контроллер** (**MVC**) к классу **VisualBankAccount** (этот класс представлен в **листинге 1**) делает последний более гибким**.** Для создания **модели** извлечены **"системные" (строки 79…104)** функции из **листинге 1,** в котором также содержится код **GUI (строки 12…65)**.

Листинг 3

```
1:  using System;
2:  using System.Collections;
3:  using System.Drawing;
4:  using System.Windows.Forms;
5:
6:  namespace WinAppl_OOP21_c307_316_GUI_MVC
7:  {
8:      public interface Observer                    // Наблюдатель
9:      {
10:         void update();      // будет обновляться отображение баланса: строки 82 и 121
11:     }                                  ////////////////  конец интерфейса Observer
```

```
12:
13:   public class BankAccountView : Form, Observer          // ВИД_1 (View_1) - GUI
14:   {
15:       private BankAccountModel model;
16:       private BankAccountController controller;
17:
18:       Button btnDeposit = new Button();                  // элементы GUI
19:       Button btnWithdraw = new Button();                  // - " - " - " -
20:       Label lblBalance = new Label();                     // - " - " - " -
21:       TextBox txtAmount = new TextBox();                  // - " - " - " -
22:
23:                       // 1-й ШАГ - объявление объектов  класса EventHandler;
24:       EventHandler DepositButtonHandler;
25:       EventHandler WithdrawButtonHandler;
26:
27:       public BankAccountView(BankAccountModel model)
28:       {
29:           this.model = model;
30:           this.model.register(this);                 // строки 162…
31:
32:
33:                   // Контроллер
34:           attachController(makeController());     // строки 98…,  93…
35:
36:             // 2-й ШАГ -  создание объектов (экземпляров) класса EventHandler
37:           DepositButtonHandler = new EventHandler(btnDeposit_Click);
38:           WithdrawButtonHandler = new EventHandler(btnWithdraw_Click);
39:
40:                       // 3-й ШАГ - регистрация объекта класса EventHandler
41:           btnDeposit.Click += DepositButtonHandler;
42:           btnWithdraw.Click += WithdrawButtonHandler;
43:
44:           lblBalance.Location = new Point(35, 15);
45:           lblBalance.Size = new Size(200, 20);
46:
47:           txtAmount.Location = new Point(35, 50);
48:           txtAmount.Size = new Size(200, 20);
49:
50:           btnDeposit.Location = new Point(35, 100);
51:           btnDeposit.Size = new Size(100, 20);
52:           btnDeposit.Text = "Deposit";
53:
```

```
54:             btnWithdraw.Location = new Point(135, 100);
55:             btnWithdraw.Size = new Size(100, 20);
56:             btnWithdraw.Text = "Withdraw";
57:
58:             Controls.Add(btnDeposit);
59:             Controls.Add(btnWithdraw);
60:             Controls.Add(lblBalance);
61:             Controls.Add(txtAmount);
62:
63:             Text = "Банковский счет";
64:             Size = new Size(285, 185);
65:             FormBorderStyle = FormBorderStyle.FixedSingle;
66:         }
67:
68:                     // 4-й ШАГ - программный код обработчика события Deposit
69:         private void btnDeposit_Click(Object sender, System.EventArgs e)
70:         {
71:     controller.depositPerformed();    // сравнить со строкой 70 в листинге 1. См. стр. 195
72:         }
73:
74:                     // 4-й ШАГ - программный код обработчика события Withdraw
75:         private void btnWithdraw_Click(Object sender, System.EventArgs e)
76:         {
77:     controller.withdrawPerformed();  // сравнить со строкой 76 в листинге 1.      См. стр. 201
78:         }
79:
80:     public void update() // Этот метод вызывается моделью, когда она изменяется
81:         {
82:             lblBalance.Text = ("Баланс: " + model.getBalance() + " RUR");
83:         }
84:
85:         // Этот метод обеспечивает доступ к количеству, введенному в поле
86:         public double getAmount()
87:         {
88:             // предполагаем, что пользователь ввел допустимое чмсло
89:             return double.Parse(txtAmount.Text);
90:         }
91:
92:         // Этот метод связывает данный контроллер с видом
93:         public void attachController(BankAccountController controller) // контроллер
94:         {
95:             this.controller = controller;
```

```
96:         }
97:
98:         protected BankAccountController makeController()
99:         {
100:            return new BankAccountController(this, model);
101:        }
102:    }                       ///////////////// конец класса BankAccountView - 1-й вид
103:
104:    public class BankAccountCLV : Observer       // ВИД_2   (View_2) - Консоль
105:    {
106:        /* это второй (альтернативный) вид (view) модели.
107:        Первый ее вид реализован в классе BankAccountView
108:        второй вид не требует контроллера,
109:        так как он не принимает событий от пользователя.
110:        назначение второго вида: печать баланса в командную строку. */
111:        private BankAccountModel model;
112:
113:        public BankAccountCLV(BankAccountModel model)
114:        {
115:            this.model = model;
116:            this.model.register(this);
117:        }
118:
119:        public void update()
120:        {
121:            Console.WriteLine("Текущий баланс: " + model.getBalance() + " RUR");
122:        }
123:
124:    }                   ////////////////////// конец класса BankAccountCLV - 2-й вид
125:
126:    public class BankAccountModel                   // Система - Модель
127:    {
128:        private double balance;
129:        private ArrayList listeners = new ArrayList();
130:
131:        public BankAccountModel(double initDeposit)
132:        {
133:            setBalance(initDeposit);
134:        }
135:
136:        public void depositFunds(double amount)
137:        {
```

```
138:            setBalance(getBalance() + amount);
139:        }
140:
141:        public double getBalance()
142:        {
143:            return balance;
144:        }
145:
146:        protected void setBalance(double newBalance)
147:        {
148:            balance = newBalance;
149:            updateObservers();
150:        }
151:
152:        public double withdrawFunds(double amount)
153:        {
154:            if (amount > getBalance())
155:            {
156:                amount = getBalance();
157:            }
158:            setBalance(getBalance() - amount);
159:            return amount;
160:        }
161:
162:        public void register(Observer o)
163:        {
164:            listeners.Add(o);
165:            o.update();
166:        }
167:
168:        public void deregister(Observer o)
169:        {
170:            listeners.Remove(o);
171:        }
172:
173:        private void updateObservers()
174:        {
175:            IEnumerator i = listeners.GetEnumerator();
176:            while (i.MoveNext())
177:            {
178:                Observer o = (Observer)i.Current;
179:                o.update();
```

```
180:             }
181:         }
182:     }   ////////////////////////// конец класса BankAccountModel - Система
183:
184:     public class BankAccountController // Контроллер - Controller
185:     {
186:         private BankAccountView view;
187:         private BankAccountModel model;
188:
189:      public BankAccountController(BankAccountView view, BankAccountModel
     model)
190:         {
191:             this.view = view;
192:             this.model = model;
193:         }
194:
195:         public void depositPerformed()
196:         {
197:             double amount = view.getAmount();
198:             model.depositFunds(amount);
199:         }
200:
201:         public void withdrawPerformed()
202:         {
203:             double amount = view.getAmount();
204:             model.withdrawFunds(amount);
205:         }
206:     }                  ////////////////// конец класса BankAccountController
207:
208:     public class MVCDriver
209:     {
210:         public static void Main(String[ ] args)
211:         {
212:             Console.Beep(150, 1500);
213:
214:             BankAccountModel model = new BankAccountModel(5000.00);
215:             BankAccountView view = new BankAccountView(model);
216:             BankAccountCLV clv = new BankAccountCLV(model);
217:             Application.Run(view);
218:
219:             Console.Beep(300, 1500);
220:         }
221:     }                        /////////////////  конец класса MVCDriver
222: }
```

Класс **BankAccountModel** похож на исходный класс **BankAccount**, представленный в предыдущих занятиях **(лекция 3, с.38,сл. и лекция 7, с.38,сл.)**; однако модель *использует* еще и **шаблон наблюдателя** (**Observer**) (**строки 8…11**) **для добавления поддержки регистрации** (**строки 162…**) **и обновления** (**строки 173…**) **объектов, которые заинтересованы в уведомлениях об изменении состояния.**

Данный класс содержит всю системную логику. Функция **withdrawFunds()** (**строки 152…160**) проверяет снимаемую со счета сумму, чтобы убедиться, что она не больше баланса. Принципиально важно реализовать подобные правила предметной области (домена) **внутри модели (NB)**. Если эти правила проникнут в вид или контроллер, каждый вид или контроллер будет вынужден поддерживать эти правила. **Это было бы смешением обязанностей, что является типичной ошибкой**. **Смешение** обязанностей **затрудняет изменение правила**, так как в результате его приходится изменять **повсюду**.

Рис. 3. Диаграмма классов программы на **листинге 3**

Отношения заменяемости также делают помещение правила в **вид** (**view**) опасной практикой. В силу отношений заменяемости вид будет работать с **любым *подклассом***; а ведь если разместить правила снятия со счета (**строки 152…**) в виде, то вид не будет правильно работать для класса **OverdraftAccount** **(лекция 3, с.38,сл. и лекция 7, с.38,сл.)** при превышении остатка. Это произойдет потому, что объекты класса **OverdraftAccount** **позволяют** снимать суммы, **большие** текущего остатка.

2.3.2. Вид (строки 13…102) (строки 104…244)

Вид отвечает за:

- отображение модели для пользователя;
- регистрацию **в** модели для уведомления об изменениях состояния;
- получение от модели информации о ее состоянии.

Вид — это тот уровень триады **MVC**, который предоставляет информацию пользователю. Вид получает отображаемую информацию от модели, используя общедоступный интерфейс модели. **Он также регистрируется в модели с тем, чтобы получать информацию об изменении состояния и обновлять себя в соответствии с ней**.

Одна модель может иметь **несколько** различных **видов**.

Поскольку модель уже создана, самое время реализовать **вид** банковского счета. В **листинге 3** содержится вид класса **BankAccountModel** (то есть класс **BankAccountView /** (**строки 13…102**)**/**).

Конструктор класса **BankAccountView** (**строки 27…66**) принимает ссылки на **BankAccountModel** (**строки 27, 29, 30**). После создания объект класса **BankAccountView (строка 215): 1)** регистрируется **в** модели (**строки 30 и 162…**), **2)** создает контроллер (**строки 34 и 98…**), и **3)** прикрепляет себя к нему (**строки 34 и 93…**), а затем **4)** создает свой пользовательский интерфейс (**строки 37…65**). **Вид** использует **модель** для получения всей информации, необходимой для отображения. При изменении баланса **модель** вызовет метод вида **update()** (**обновить**). **При вызове метода update() вид обновит свое отображение баланса (строки 80…83).**

Как правило, **вид** создает свой собственный **контроллер**, как, например, класс **BankAccountView** поступает в методе **makeController() (строки 98…100)**. Внутри метода **attachController (строки 93…96) вид** регистрирует **контроллер**.

Обычно **вид имеет только один контроллер**.

Заметьте, что **attachController()** — метод общедоступный (**public**). Используя этот метод, можно переключать контроллеры без необходимости указывать подкласс **вида (вид может быть базовым классом**). Этот метод позволяет создавать различные контроллеры и передавать их виду. Контроллер, с которым вы познакомитесь в следующем разделе, интерпретирует события пользователя так же, как это делал класс **VisualBankAccount** (**листинг 1**; с небольшой модификацией).

В отличие от вида, у которого может быть только один контроллер, **модель может иметь несколько различных видов**. В **листинге 3** показан **второй вид** для банковского счета – это класса **BankAccountCLV** (**строки 104…124**).

В классе **BankAccountCLV** также реализован метод интерфейса **Observer,** который печатает баланс в командную строку (**строка 119…122**). Несмотря на то, что это простое поведение, класс **BankAccountCLV** – альтернативный вид класса **BankAccountModel**. Этот вид не требует контроллера, т.к. он не принимает событий от пользователя. Таким образом, наличие контроллера не является необходимым. Вид не всегда отображается на экране.

Возьмем, к примеру, текстовый редактор. Модель текстового редактора будет отслеживать введенный текст, форматирование, сноски и пр. Один вид будет отображать текст в главном редакторе, другой может преобразовывать информацию модели в формат **PDF, HTML** или **Postscript**, а затем выводить это представление в файл. Вид, выводящий информацию в файл, не отображает ее на экране; вместо этого данный вид записывает представление в файл. Другие программы могут открыть файл, чтобы прочитать и отобразить данные, хранящиеся в нем.

2.3.3. Контроллер (строки 184…206)

Контроллер отвечает за:

- перехват событий пользователя, поступающих из вида;
- перехват события и вызов соответствующих методов модели или вида;
- регистрацию в модели для уведомления об изменении состояния (в случае заинтересованности).

Контроллер выступает как **связующее звено между видом и моделью**. Контроллер **1)** перехватывает события, поступающие от вида, и **2)** превращает их в запросы к модели или виду.

Вид имеет только один контроллер, а у контроллера есть только один вид. Некоторые виды позволяют устанавливать их контроллер напрямую.

Каждый вид имеет один контроллер, который **обслуживает все взаимодействия пользователя с системой (NB)**. Если контроллер зависим от информации о состоянии, он должен зарегистрировать себя в модели для уведомления об изменениях состояния.

Когда **модель** и **вид** уже созданы, остается только построить **контроллер** для класса **BankAccountView**. В **листинге 3** представлен **контроллер** этого **вида** – это класс **BankAccountController (строки 184…206)**.

После создания **контроллер** принимает: ***1)*** ссылки как на **вид** (**строки 197, 203**), ***2)*** так и на **модель** (**строки 198, 204**). **Контроллер использует вид** для извлечения суммы (количества денег) из поля ввода **(строки 197, 203, 86…90)**. **Контроллер использует модель** при действительном снятии денег (**строки 204, 152…160**) и помещении их на счет (**строки 198, 136…139**).

Сам по себе **BankAccountController** довольно прост. Контроллер получает события от **вида посредством обработчиков событий (строки 68…78)**. Событий всего два – это клики мышкой на кнопках **Deposit** (**положить**), **Withdraw** (**снять**). **Контроллер интерпретирует события по мере их получения (строки 195…205).** Он обращается к **модели** – выполняет нужный вызов.

В отличие от класса **VisualBankAccount (листинг 1)**, **контроллеру** нужно вызвать только метод **depositFunds()** (**строка 198**) или **withdrawFunds()** (**строка 204**). О соблюдении правил домена позаботилась модель.

2.4. Сборка модели, вида и контроллера (строки 208…221)

В **листинге 3** представлена небольшая главная (**Main**) функция (**строки 210…221**), соединяющая **модель** и два **вида**. Функция **Main** ничего не делает с контроллером, т.к. об этом заботится вид.

Функция **Main** вначале создает экземпляр модели (**строка 214**). Когда **Main** располагает моделью, она может создавать различные виды (**строки 215 и 216**). На **рис. 4** изображены выходные данные.

При выполнении класса **MVCDriver** на экране отобразятся два различных представления одной и той же модели. Использование шаблона **MVC** позволяет **создавать сколько угодно представлений** базовых моделей.

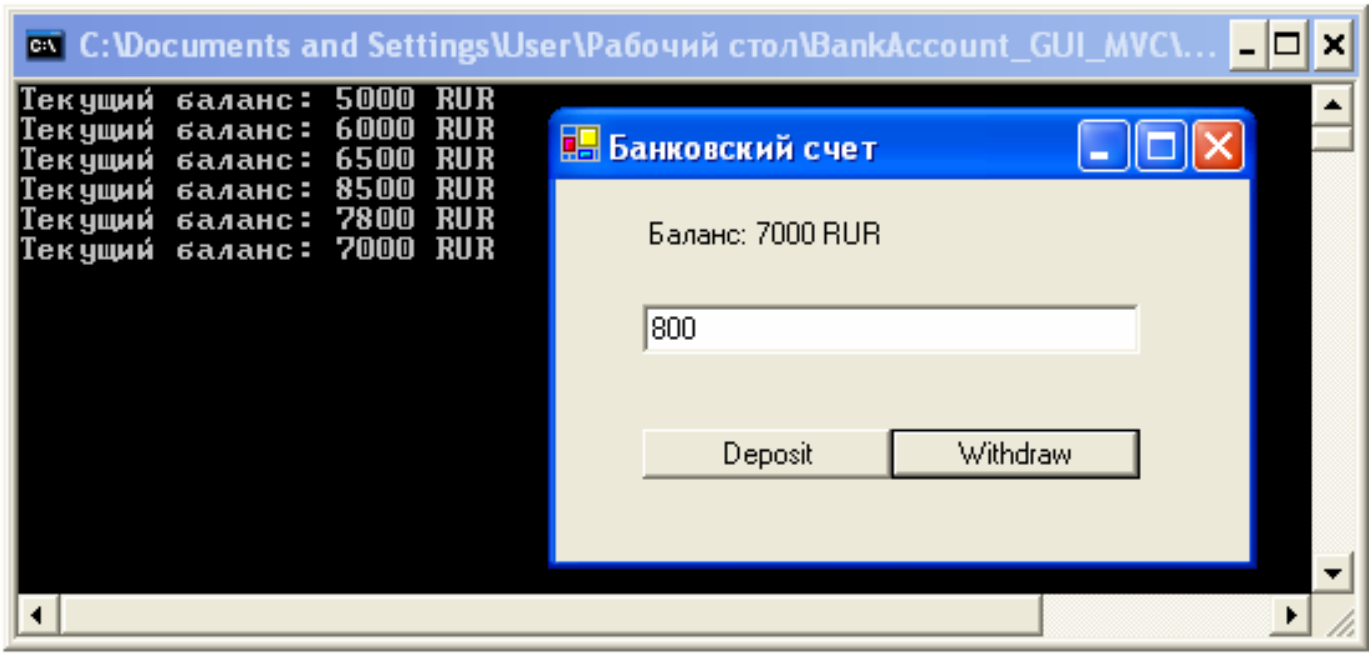


Рис. 4. Модель банковского счета с **двумя** представлениями (**видами**)

2.5. Проблемы, которые могут возникнуть при использовании шаблона модель / вид / контроллер

Как и любой шаблон, **модель/вид/контроллер** имеет свои недостатки и, соответственно, критиков.

К недостаткам можно отнести следующее.

- Акцент на обработке данных.
- Сильная связь между **видом/контроллером и моделью**.
- Высокая вероятность неэффективной работы.

То, как проявятся указанные недостатки, зависит: **1)** от поставленной задачи и **2)** предъявляемых ею требований.

2.5.1. Акцент на обработке данных

В рейтинге верности принципам объектно-ориентированного программирования шаблон **модель/вид/контроллер** не занимает верхних позиций — **виной тому акцент на обработке данных**. Вместо того чтобы давать команду объекту что-либо делать с его данными, *вид запрашивает данные у модели* (**строки 82 и 121**) *, а затем отображает их на экране.*

Этот недостаток можно сгладить, отображая на экране только те данные, которые удаляются из модели. Не выполняйте дополнительной обработки данных. Если обнаружилось, что вы выполняете дополнительную обработку данных после их извлечения (**строки 89 и 121**) или перед тем, как применить к модели метод, вполне вероятно, что это следует перепоручить модели. Необходимую и излишнюю обработку данных разделяет очень тонкая черта.

Если в каждом виде повторяется один и тот же код, подумайте над возможностью перенесения логики в модель.

Избегая использовать шаблон **MVC** единственно ради верности принципам объектно-ориентированного программирования, тем самым вы рискуете бросить вызов некоторым реалиям

программирования. Возьмем, к примеру, **Web-сайт**. Некоторые компании проводят четкое разделение между представлением (**видом**) и логикой бизнеса (**моделью**). Последовательное применение этого принципа имеет под собой веские основания**:** программисты будут писать программу, а те, кто отвечает за содержание сайта — содержание. Изъятие содержания из программного кода означает, что теперь содержание могут создавать непрограммисты. Совмещение двух этих уровней означало бы, что-либо человек, пишущий содержание, должен быть программистом, либо программисту пришлось бы внедрять содержание в код. Но если бы в код было внедрено содержание, то модифицировать сайт было бы намного труднее.

Как показывает действительность, требования **не высечены в граните**. Это отлично иллюстрирует пример с **Web-сайтом**. **Web-сайт** должен предоставлять **HTML**-документ, отображаемый браузером пользователя. Но ведь есть же "карманные" компьютеры, предназначенные для выполнения некоторых специальных функций (**PDA. P**ersonal **D**igital **A**ssistant), мобильные телефоны и другие устройства отображения! Ни одно из этих устройств не может непосредственно отображать **HTML**-документ. Заглянем в будущее — вполне возможно, что через полгода появится новый тип дисплеев. Способность отвечать возникающим требованиям принуждает заботиться о гибкости проекта. Если ваша **система** достаточно **статична**, а требования к ней заранее четко определены, можно использовать альтернативу **MVC**, например, **PAC (Presentation Abstraction Control)**. Если же вы не можете похвастаться четкими требованиями к программе, следует использовать шаблон **модель/вид/контроллер (MVC)**.

2.5.2. Сильная связь

И **вид**, и **контроллер** сильно привязаны к общедоступному интерфейсу **модели (public-**методы **класса BankAccountModel, листинг 3)**. Изменения в модели влекут за собой изменения, как вида, так и контроллера. Используя шаблон **MVC**, вы принимаете на веру предположение, что **модель стабильна**, а **вид может изменяться**. Если это не так, необходимо: **1)** либо прибегнуть **к другому шаблону**, **2)** либо примириться с тем, что придется **вносить изменения 1) и в вид 2) и контроллер**.

Вид и контроллер также тесно связаны друг с другом. Контроллер почти всегда используется исключительно с определенным видом. Тщательным планированием можно добиться возможности повторного использования кода. Даже если это не удалось, шаблон **MVC** все равно обеспечивает хорошее разделение обязанностей между объектами. Объектно-ориентированное программирование предназначено не только для повторного использования кода.

2.5.3. Вероятность неэффективной работы

При разработке и внедрении пользовательского интерфейса, основанного на шаблоне **модель/вид/контроллер**, следует быть очень осторожным, чтобы уменьшить вероятность неэффективной работы. Она может закрасться в любую из составных частей триады **MVC**.

Модель не должна передавать наблюдателям незатребованные уведомления об изменениях состояния.

При разработке контроллера и вида, возможно, вам придется применить кэширование данных, — если извлечение их из модели происходит слишком медленно. После уведомления об изменении состояния, извлекайте только измененное состояние. Шаблон наблюдателя (**Observer**) может быть дополнен так, чтобы модель передавала идентификатор методу **update().** Вид может использовать этот идентификатор для того, чтобы определить, нужно ли ему обновлять себя.

Резюме

Пользовательский интерфейс — важная составляющая любой системы. Для некоторых пользователей интерфейс — это единственная часть системы, с которой они взаимодействуют; собственно, для них интерфейс и будет самой системой. К анализу, проектированию и реализации пользовательского интерфейса следует подходить так же, как к анализу, проектированию и реализации любой из других частей программы. **Пользовательский интерфейс никогда не должен писаться "вдогонку" и навешиваться на систему в последнюю минуту.**

Хотя существует множество подходов к разработке пользовательского интерфейса, использование шаблона **модель/вид/контроллер** дает преимущество гибкости, отделяя пользовательский интерфейс от основной системы. Однако, как и любое другое решение, связанное с разработкой пользовательского интерфейса, применение шаблона **MVC** требует тщательного взвешивания всех "за" и "против". Шаблон **MVC** не в силах оградить вас от реальностей системы.

Контрольные вопросы

1. В чем **отличие**: **1)** анализа, **2)** проектирования и **3)** реализации пользовательского интерфейса от соответствующих шагов разработки других частей программы?
2. Почему нужно отделять пользовательский интерфейс от основной программы?
3. Назовите составные **MVC** триады.
4. Назовите две возможные альтернативы шаблону **MVC**.
5. Опишите обязанности модели.
6. Опишите обязанности вида.
7. Опишите обязанности контроллера.
8. Сколько моделей может быть в системе? Сколько видов может быть в системе? Сколько контроллеров может быть в системе?
9. Что может быть причиной снижения эффективности при использовании шаблона **MVC**?
10. Какие допущения предполагает шаблон **MVC**?
11. Какова история создания шаблона **MVC**? (Этот вопрос предполагает использование **Web**-ресурсов.)

Ответы на вопросы

1. **Анализ**, **проектирование** и **реализация** пользовательского интерфейса (**UI**) ничем не отличается от остальных этапов. На всех этапах разработки ему следует уделять не меньше внимания, чем другим частям системы. Разработка пользовательского интерфейса, как и любой другой части системы, должна быть проведена на высоком уровне.
2. **Пользовательский интерфейс следует отделять от основной программы таким образом, чтобы система и пользовательский интерфейс не были связаны**. Ведь если пользовательский интерфейс переплетается с функциональными возможностями ядра, то изменить его весьма затруднительно.

   Кроме того, если пользовательский интерфейс переплетается с системой, то в системе трудно использовать другие пользовательские интерфейсы или даже другие его типы.
3. Тремя компонентами являются модель, вид и контроллер.
4. Шаблоны проектирования **Presentation Abstraction Control (PAC)** и **Модель / Документ / Представление** (**Document / View / Model**) — две альтернативы шаблона **модель / вид / контроллер** (**Model / View / Controller** — **MVC**).
5. "Модель" — один из слоев (уровней) триады **MVC**, который управляет поведением ядра и из-

менением состояния системы. "Контроллер" использует модель для настройки поведения системы. "Вид" использует модель для получения информации о состоянии системы с целью ее отображения.

"**Модель**" также имеет механизм уведомления об изменениях. "**Вид**" и "**Контроллер**" могут использовать этот механизм для отслеживания изменений состояния модели.

6. "**Вид**" – компонент триады **MVC**, ответственный за отображения модели пользователю.
7. Функции "**Контроллера**" — интерпретация событий, генерируемых пользователем. Контроллер в соответствии с этими событиями определяет изменения в поведении "Модели" или в "Виде".
8. Система может иметь много моделей. Модель может иметь много различных представлений (видов). Представление (вид) может иметь один контроллер, а контроллер может управлять только одним представлением (видом).
9. И модель, и представление (вид), и контроллер могут быть реализованы неэффективно. **Модель должна избегать ненужных уведомлений об изменении состояния**. Представления (виды) и контроллеры должны при первой же возможности кэшировать данные.
10. Шаблон **MVC** предполагает **стабильную** модель и **изменяющееся** представление (вид).
11. По адресу **http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html** находится написанный доктором Стивом Бурбеком (Steve Burbeck, Ph.D.) превосходный обзор истории и мотивации создания шаблона **MVC** "**Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC)**" ("**Программирование Приложений в Smalltalk-80 (TM): Как использовать модель/вид/контроллер (MVC)**").

    При знакомстве с историей выясняется, что шаблон **MVC** первоначально создавался как часть языка **Smalltalk**. В настоящее время шаблон **MVC** используется почти во всех языках программирования. Отсюда следует важный вывод: **шаблоны проектирования — это не средство конкретного языка – это шаблоны, работающие в любом языке с необходимыми средствами**.

    MVC представляет проект, который выходит за рамки языка реализации.

3. Применение тестирования для создания надежного программного обеспечения

Используя объектно-ориентированный подход, стараются писать более **понятные**, **надежные**, **повторно используемые**, **удобные в сопровождении**, **расширяемые** программы, притом такие, чтобы было **удобно осуществлять периодический выпуск (издание) новых версий**. Но для достижения желаемой цели необходимо понимать следующее. **Создание хорошей объектно-ориентированной программы** — не дело рук случая, а **результат тщательных: 1)** анализа и **2)** проектирования, а также **3)** аккуратной реализации. Ни в коем случае нельзя терять из виду основные принципы объектно-ориентированного подхода. Только тогда программирование в объектно-ориентированном стиле начнет приносить ожидаемые плоды.

Но даже **осторожный анализ** и **аккуратная реализация** это еще не панацея от всех бед. Ведь никто не застрахован от собственных ошибок или ошибок других людей, И ошибки случаются! Поэтому, чтобы создать надежную программу, **необходимо проверять (тестировать) программное обеспечение.**

Вы узнаете:

- когда тестирование переходит в итеративный (повторяющийся) процесс;
- различные **типы** тестирования;
- как проверить собственные классы;
- как провести испытания (тестирование) незаконченного программного обеспечения;
- что необходимо сделать для написания более надежного кода;
- как сделать тестирование более эффективным.

3.1. Испытание объектно-ориентированного программного обеспечения

Объектно-ориентированный подход не предотвращает проникновения ошибок в программное обеспечение. Ведь их совершают даже лучшие программисты. Ошибки – это дефекты программ, происходящие вследствие: **1) опечаток**, **2) логических ошибок** или просто "глупых" **3) ошибок при кодировании**. И хотя самым общим источником ошибок является **реализация** объекта (т.е. сам объект), форма их проявления может быть самая разнообразная.

Например, ошибки могут проявляться тогда, когда один объект неправильно использует другой. Ошибки происходят даже в результате: **1)** фундаментальных недостатков анализа или **2)** непосредственно проектирования. Вполне естественно, что объектно-ориентированная система полна взаимодействующими объектами. Эти **(3)** взаимодействия могут быть источником практически всех видов ошибок.

К счастью, избежать дефектов в программах возможно. Использование тестирования делает возможной проверку всех этапов**: 1) анализа**, **2) проектирования** и **3) реализации**.

N B	Подобно объектно-ориентированному программированию, испытание не является магическим решением проблемы. Чрезвычайно сложно полностью проверить программное обеспечение более-менее приличного размера. Время, необходимое для тестирования всего кода, а также большое количество связей в нетривиальной программе сильно осложняют тестирование, и делают невозможным всеобщее испытание программного обеспечения. Поэтому даже проверенный код может содержать скрытые ошибки. Наилучшее, что можно сделать в этом случае, — потратить на тестирование столько ресурсов, чтобы удостовериться в необходимом качестве программы и, в тоже время, уложиться в сроки и бюджет, выделенные на выполнение задания. Реально количество времени, потраченное на тестирование, будет зависеть от рамок проекта и собственного уровня удобства программиста.

3.2. Роль и место тестирования в итеративной технологии разработки программного обеспечения

Рис. 5 иллюстрирует цикл разработки, ранее представленный в **лекции 12, раздел 3.** "Введение в объектно-ориентированный анализ (ООА)", **с.33,сл.** Испытание программного обеспечения – последний шаг в итеративной цепочке.

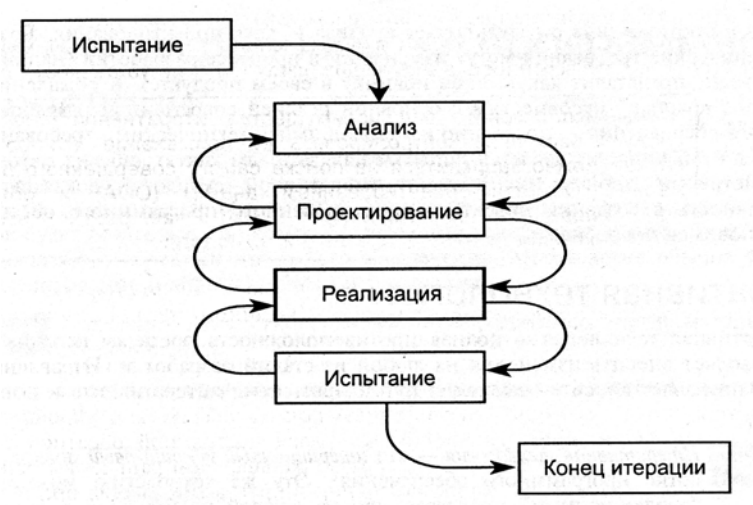


Рис. 5. Итерация

Тестирование в конце каждого цикла разработки очень важный этап, во время которого проверяется, не нарушена ли существующая функциональность программы внесенными на этой итерации изменениями. На стадии испытания также проверяется правильность работы добавленных на данной итерации новых функций. По этой причине испытания, выполненные перед окончанием цикла разработки, часто называются**: функциональным тестированием**, **тестированием на функциональном уровне**, **приемочными испытаниями**, **приемо-сдаточными испытаниями**, **испытаниями при приемке, приемкой или тестированием приемлемости**.

Примечание	**Тестирование в конце каждой итерации** — **очень важный этап создания ПО**. Для того, чтобы закончить очередную итерацию, система должна пройти тест. Однако испытания нужно проводить также и во время других этапов итеративного цикла разработки. Рассмотрим использование тестирования на стадии реализации и изучим этапы тестирования в процессе разработки. Тестирование – **1)** это и конечная цель, а также и **2)** то, что необходимо выполнять на протяжении всего цикла разработки. Ведь может оказаться невозможным провести испытания программы, которая не проверяется вплоть до окончания реализации. Поэтому необходимо **соединять процесс разработки и тестирования**, сделав испытание неотъемлемой частью процесса разработки программного обеспечения.

Если обнаружена ошибка, необходимо вернуться назад и исправить ее. Обычно возвращаются к этапу реализации и пробуют исправить возникшую проблему прямо в коде. Иногда все, что нужно сделать, это просто исправить дефект, повторно все испытать и продолжить разработку дальше. Однако **ошибки иногда могут происходить 1) в результате недостатков проектирования** или даже **2)** из-за поверхностного изучения или неправильного истолкования требований. Для устранения таких недостатков необходимо **вернуться назад** на этап **проектирования** и **анализа**, и только потом можно будет исправить ошибку в коде программы.

N B	После исправления не достаточно просто проверить программу на предмет отсутствия найденной ошибки. **Необходимо произвести полное тестирование**. Ведь избавившись от одной, в программу можно легко внести другую, а то и несколько новых ошибок.

Неправильное понимание требований еще на стадии анализа означает, что **система**, скорее всего, **не будет функционировать так, как того ожидает заказчик**. Система должна работать так, как ожидается; иными словами, она должна иметь предсказуемое поведение. Поэтому необходимо **: 1)** не только проверить код на отсутствие ошибок реализации, **2)** но и убедиться, что поведение системы не отличается от ожидаемого.

NB	Тестирование заключается в необходимости написания и выполнения контрольных примеров, каждый из которых проверяет какую либо специфическую часть системы.

Контрольный пример — основное элементарное звено процесса тестирования. Во время испытания выполняется ряд **контрольных примеров**, чтобы полностью проверить правильность работы системы. Каждый контрольный пример содержит совокупность тестовых данных (т.е. наборов входных и ожидаемых выходных данных, **см. строки 92…97 в листинге 4 на с. 31**).

Есть два метода тестирования. Тест**: 1)** может выполняться по определенному пути в системе (**метод "прозрачного ящика"**) или **2)** проверять некоторое определенное поведение (**метод "черного ящика"**).

В результате прогона контрольного примера проверяется определенная часть функций программы, чтобы посмотреть, ведет ли себя система должным образом. Если она ведет себя, как ожидается, то считается, что система выдержала испытание данным контрольным примером, в противном случае говорят, что система не выдержала испытание данным контрольным примером. Если при выполнении контрольного примера были получены неправильные выходные данные, то это указывает на наличие ошибок в системе. Всегда необходимо добиваться 100% правильного выполнения всех контрольных примеров. Даже если при прогоне сотен контрольных примеров будут получены правильные выходные данные, не нужно поддаваться соблазну игнорировать неправильные выходные данные, полученные при выполнении одного, пусть "самого незначительного", контрольного примера. **Система должна пройти каждое контрольное испытание**, в противном случае продолжать дальнейшую работу нельзя!

Есть два метода построения контрольных примеров: **1)** метод "**прозрачного ящика**" и **2)** метод "**черного ящика**". Эффективная стратегия испытаний состоит в **соединении** этих двух методов.

При **тестировании методом черного ящика** проверяется **правильность поведения** системы. Задавая определенные входные данные, при ***тестировании методом черного ящика*** проверяется, соответствует ли видимый вывод или наблюдаемое поведение тому, которое описано в спецификациях класса или приложения.

При **тестировании методом прозрачного ящика** испытания **основаны исключительно на реализации методов**. Этим видом тестирования желательно охватить все 100% кода.

В основу **испытания отдельных классов** методом черного ящика кладутся **функциональные требования к классу**. Основу **испытаний всей системы** методом черного ящика **составляют отдельные случаи использования программы (прецеденты)**. Как бы то ни было, в ходе **испытаний методом черного ящика** проверяется, совпадает ли **поведение объекта** или **программы** с ожидаемым **(см. строки 92…97 и строки 99…106 в листинге 4 на с. 31)**. Например, если с помощью некоторого метода предполагается суммировать два числа, то при тестировании ему передаются два слагаемых и затем проверяется, действительно ли получился правильный результат, т.е. является ли результат суммой переданных методу двух чисел. Если предполагается, что система позволяет добавлять и удалять отдельные предметы из тележки для магазинов самообслуживания, то испытание методом "черного ящика" будет состоять из попыток добавить предметы в тележку или удалить их из нее.

С другой стороны, **тестирование методом прозрачного ящика основано на реализации методов**. **Цель такого тестирования** — гарантировать **выполнение каждой ветви** программы. По некоторым оценкам **считается, что испытания методом "черного ящика" охватывают только треть или половину кода**. В случае использования метода **прозрачного** ящика, испытания проектируются так, **чтобы осуществить проверку каждого ветвления (оператора перехода) кода в надежде исключить любые скрытые ошибки**.

Совет	Исключая самые простые программы, тестирование методом прозрачного ящика едва ли может охватить даже разумное количество комбинаций ветвления программы. Есть две возможности повысить эффективность тестирования**: 1)** писать программы так, чтобы они имели **минимальное количество ветвлений** («**долой**» **goto**); **2)** определить наиболее критические ветви и обязательно их проверить.

Например, в методе, который делит два числа, есть ветвь обработки ошибки, переход на которую программа выполняет в случае попытки деления на **0**. Следовательно, должен существовать контрольный пример, в ходе выполнения которого гарантированно возникает состояние ошибки. Если такая ситуация не определена в документации интерфейса, нельзя узнать об этой ветви, не посмотрев непосредственно на код. Поэтому **тестирование методом прозрачного ящика должно быть основано непосредственно на самом коде**.

В любом случае, методы черного и прозрачного ящика определяют создание контрольных примеров. Причем каждый из них играет важную роль во всех формах испытаний.

3.3. Формы тестирования

Всего существует **четыре основных формы тестирования**: **1)** от низкоуровневого тестирования **отдельных** классов и объектов, **2)** до высокоуровневого испытания **всей** системы. Выполнение каждой формы тестирования помогает гарантировать общее качество программного обеспечения.

3.3.1. Автономное тестирование, блочное тестирование, тестирование элементов программы или тестирование модуля

Блочное тестирование, или тестирование элементов (программы) — наиболее низкоуровневая часть процесса испытания. **Во время блочного тестирования за один раз проверяется только какая-нибудь одна функция** **(см. строки 95 и 61…69 в листинге 4 на с. 31)**.

Автономное тестирование, блочное тестирование, тестирование элементов (программы), или тестирование модуля — наиболее низкоуровневая часть тестирования. В ходе тестирования элементов программы **тестирующий модуль посылает объекту сообщение**, а затем проверяет, что он принимает от объекта ожидаемый результат. Поэтому при тестировании элементов программы **проверяется только одна функция за один раз** **(см. строки 95 и 61…69 в лист. 4 на с. 31)**.

Используя объектно-ориентированный подход, в процессе тестирования элементов программы проверяется отдельный класс объектов. Модуль тестирования проверяет объект следующим образом**: 1)** сначала **посылает объекту сообщение (см. строку 95…)**, а затем **2) проверяет, возвратил ли объект ожидаемый результат** **(… и строку 101 в листинге 4 на с. 31)**. Для тестирования элементов программы может использоваться любой из двух методов: **черного ящика** или **прозрачного ящика**. И на самом деле, **чтобы убедиться в правильности работы объектов, обычно используются оба**. Для **каждого** разрабатываемого **класса** необходимо предусмотреть соответствующий блочный тест. И, вероятно, необходимо написать контрольные примеры прежде, чем приступать к проектированию класса.

Сосредоточимся на тестировании **элементов программы**, как наиболее важном для написания надежного объектно-ориентированного программного обеспечения. Фактически тестирование элементов программы приходится выполнять на протяжении всего процесса разработки.

3.3.2. Тестирование сопряжений, или проверка взаимодействия и функционирования компонентов системы

Объектно-ориентированная система состоит из взаимодействующих объектов. В то время как с помощью тестирования элементов программы исследуется каждый объект в отдельности, в ходе проверки **взаимодействия** и **функционирования** компонентов системы проверяется, взаимодействуют ли должным образом объекты, из которых состоит программа. То, что работает в отдельности, может стать причиной аварийного завершения в результате взаимодействия с другими объектами! Общий источник ошибок такого рода — **1)** разногласие в форматах ввода/вывода, конфликты ресурсов, а также **2)** неправильная последовательность вызовов методов.

С помощью **тестирования сопряжений**, или **проверки взаимодействия и функционирования компонентов системы** проверяется **правильность совместной работы нескольких объектов**.

В **основу проверки взаимодействия и функционирования компонентов системы**, как и **в основу тестирования элементов программы**, может быть положена **любая их двух концепций** тестирования**: 1)** метод прозрачного ящика или **2)** метод черного ящика. Необходимо **выполнить процедуру проверки** взаимодействия и функционирования компонентов системы **для каждого важного взаимодействия в системе.**

3.3.3. Комплексное тестирование, или испытания системы

С помощью испытаний системы проверяется работоспособность системы в целом. Выполняя испытания системы, необходимо также проверить программу на работоспособность в нестандартных и не описанных случаях использования. Так можно проверить, что система остается управляемой и легко восстанавливается при непредвиденных ситуациях.

Совет	Во время испытаний системы проводятся следующие испытания. **Испытания с помощью случайных воздействий (1).** Состоят из попыток выполнения операций в случайном порядке; **Испытания с пустой базой данных (2).** Гарантируют, что система сможет правильно завершить работу в случае, когда главная причина сбоя связана с базой данных. **Видоизмененный случай использования (3).** Тест заменяет допустимый случай использования недопустимым. В ходе испытаний проверяется, что система может должным образом восстанавливаться при некорректном использовании.

В ходе **комплексного тестирования**, или **испытаний системы** проверяется работоспособность системы в целом. При испытаниях системы проверяется, что система правильно работает**: 1)** в предусмотренных вариантах ее использования, а также **2)** не теряет управления в необычных и непредусмотренных случаях.

Испытания системы также включают в себя **нагрузочные испытания**, т.е. **1)** испытания в жестком физическом режиме (например, при повышенной температуре) и **2)** испытания в рабочих условиях. Эти испытания гарантируют, что система удовлетворяет требованиям к функционированию, т.е. всем требованиям к рабочим характеристикам. Если возможно, лучше выполнить эти испытания в среде, которая настолько близка к рабочей среде программы, насколько это возможно.

Испытания системы — важный аспект приемочных испытаний. В ходе испытаний системы сразу проверяются все взятые в целом функциональные элементы, поэтому одно испытание может коснуться многих различных объектов и подсистем. Чтобы завершить этот этап, программа должна успешно пройти все указанные испытания.

3.3.4. Регрессивное тестирование

Результаты тестирования действительны лишь до тех пор, пока в протестированной части что-

нибудь не изменится. Когда какой-нибудь аспект системы изменяется, эту, и все зависимые от нее части, необходимо протестировать снова. **Регрессивное тестирование** — это процесс **повторения : 1)** тестирования элементов программы, **2)** проверки взаимодействия и функционирования компонентов системы и **3) испытания системы после сделанных в программе изменений**[1].

В ходе регрессивного (возвратного) тестирования проверяются те части системы, которые были испытаны ранее, но впоследствии оказались изменены. Всякий раз, когда изменяется какая-либо часть системы, ее, а также все зависимые от нее части необходимо испытать повторно.

Очень важно провести повторные испытания даже после небольшого изменения. Небольшое изменение может привнести в код дефект, потенциально могущий нарушить работоспособность всей системы. К счастью, регрессивное тестирование просто и представляет собой повторное тестирование элементов программы, проверки взаимодействия и функционирования компонентов системы, а также испытания системы.

3.4. Руководство по написанию надежного кода

Для обеспечения высокого качества программного обеспечения важен каждый вид испытаний. Но сегодня мы сосредоточимся на том, что лучше всего делать ежедневно, чтобы гарантировать соответствующее качество создаваемых программ. Чтобы создать надежный код, необходимо**: 1)** тестировать элементы программы, а также **2)** учиться находить различия между: **а) состояниями ошибки** и **б) самими ошибками** и **в) писать удобную для использования документацию (см. раздел 3.6, с. 38…40)**.

3.4.1. Объединение разработки и тестирования

На **рис. 5** не отображено одно очень важное обстоятельство: тестирование должно быть непрерывным процессом, продолжающимся на протяжении всей разработки. Испытание не должно быть чем-то избегаемым, откладываемым до конца, выполняемым кем-то другим или пропускаемым полностью. Ведь иногда невозможно начать тестирование внезапно, сразу по окончании написания программы. Вместо этого необходимо научиться **начинать тестирование еще во время разработки**. Чтобы протестировать разработанные части программы, **нужно написать автономные тесты элементов программы для каждого создаваемого вами класса**.

3.4.2. Пример тестирования элементов программы

Рассмотрим тестирование элементов программы на примере класса **SavingsAccount**, впервые представленного в **лекции 3 "Наследование: напишем программу", с.38,сл.**. В **листинге 4** представлен класс **SavingsAccountTest**.

Листинг 4. SavingsAccountTest.cs

1	`using System;` `// Листинг 4`
2	
3	`namespace ConsAppl_OOP21_c331_test`
4	`{`
5	`public class BankAccount` `// Класс BankAccount - базовый класс.`
6	`{ // В общем коде предусмотрено сохранение и манипулирование счетом-балансом.`

[1] Вот еще одно определение регрессивного тестирования (regression testing): регрессивное тестирование — **это тестирование с возвратом от более сложных тестов к простым**. Регрессивными испытаниями (regression tests) называются *и***спытания, выявляющие способность новой системы к выполнению всех требуемых функций заменяемой системы**. Кроме того, употребляется также термин возвратное тестирование (regression test)

```
7
8       private double balance;
9
10      public BankAccount(double initDeposit)                    // конструктор
11      {
12          setBalance(initDeposit);
13      }
14
15      public virtual double withdrawFunds(double amount)        // снять деньги со счета.
16      {
17          if (amount >= balance)
18          {
19              amount = balance;
20          }
21          setBalance(getBalance() - amount);
22          return amount;
23      }
24
25      public void depositFunds(double amount)                   // положить деньги на счет
26      {    // базовый класс не применяет никакой стратегии, не проверяет правильность ввода
27          setBalance(getBalance() + amount);
28      }
29
30      public double getBalance()                                // запрос остатка-баланса
31      {
32          return balance;
33      }
34
35      protected void setBalance(double newBalance)              // установить остаток-баланс
36      {
37          balance = newBalance;
38      }
39  }//////////////////////////////////////////////Конец класса BankAccount - базовый////////////////////////////////
40
41  public class SavingsAccount : BankAccount                     // производный класс
42  {                                   /* Класс SavingsAccount НАСЛЕДУЕТ класс BankAccount
                                        и действия в соответствии с описанием класса SavingsAccount:
                                                balance = balance + (balance * interestRate */
43
44      private double interestRate;
45                                      // конструктор создает новый счет SavingsAccount
46      public SavingsAccount(double initBalance, double interestRate): base(initBalance)
47      {
```

```
48          setInterestRate(interestRate);
49      }
50
61      public void addInterest()                   // вычислить и прибавить процент к счету
62      {           // этот метод проверяется контрольным примером: см. строки 92…97
63          double balance = getBalance();
64          double rate = getInterestRate();
65          double interest = balance * rate;
66
67          double new_balance = balance + interest;
68
69          setBalance(new_balance);
70      }
71
72      public void setInterestRate(double interestRate)      // установить процентную ставку
73      {
74          this.interestRate = interestRate;
75      }
76
77  public double getInterestRate()  // сделать запрос процентной ставки (нормы процента)
78      {
79          return interestRate;
80      }
81  }           //////////////////////конец класса  SavingsAccount//////////////////////////
82
83    public class SavingsAccountTest                          // Класс SavingsAccountTest
84    {
85      public static void Main(String[] args)
86      {
87          SavingsAccountTest sat = new SavingsAccountTest();
88          sat.test_applyingInterest();
89          Console.ReadLine();
90      }
91
92      public void test_applyingInterest()       //  это контрольный пример: строки 92…97
93      {                                   // 0.05 – правильно установленная процентная ставка
94          SavingsAccount act = new SavingsAccount(10000.00, 0.05);
95          act.addInterest();                      // этот метод проверяется: см. строки 61…70
96          print_getBalanceResult(act.getBalance(), 10500.00, 1); // см. строки 30…33
97      }
98
99      private void print_getBalanceResult(double actual, double expected, int test)
100     {
```

```
101   if (actual == expected)  // сравнение фактического и ожидаемого (10500) результатов
102         {                          // тест прошел
103       Console.WriteLine("Прошел тест №" + test + " - процент установлен правильно");
104       Console.WriteLine("Фактическое значение счета: " + actual);
105       Console.WriteLine("Ожидаемое (правильное) значение счета: " + expected);
106         }
107         else
108         {                          // тест потерпел неудачу
109   Console.WriteLine("Не прошел тест №" + test + " - процент установлен неправильно");
110    Console.WriteLine("Фактическое значение счета: " + actual);
111   Console.WriteLine("Ожидаемое (правильное) значение счета: " + expected);
112         }
113      }
114   }
115 }
```

Результат работы программы:

1-й запуск

Прошел тест №1 - процент установлен правильно (0.05)
Фактическое значение счета: 10500
Ожидаемое (правильное) значение счета: 10500

2-й запуск

Не прошел тест №1 - процент установлен неправильно (0.04)
Фактическое значение счета: 10400
Ожидаемое (правильное) значение счета: 10500

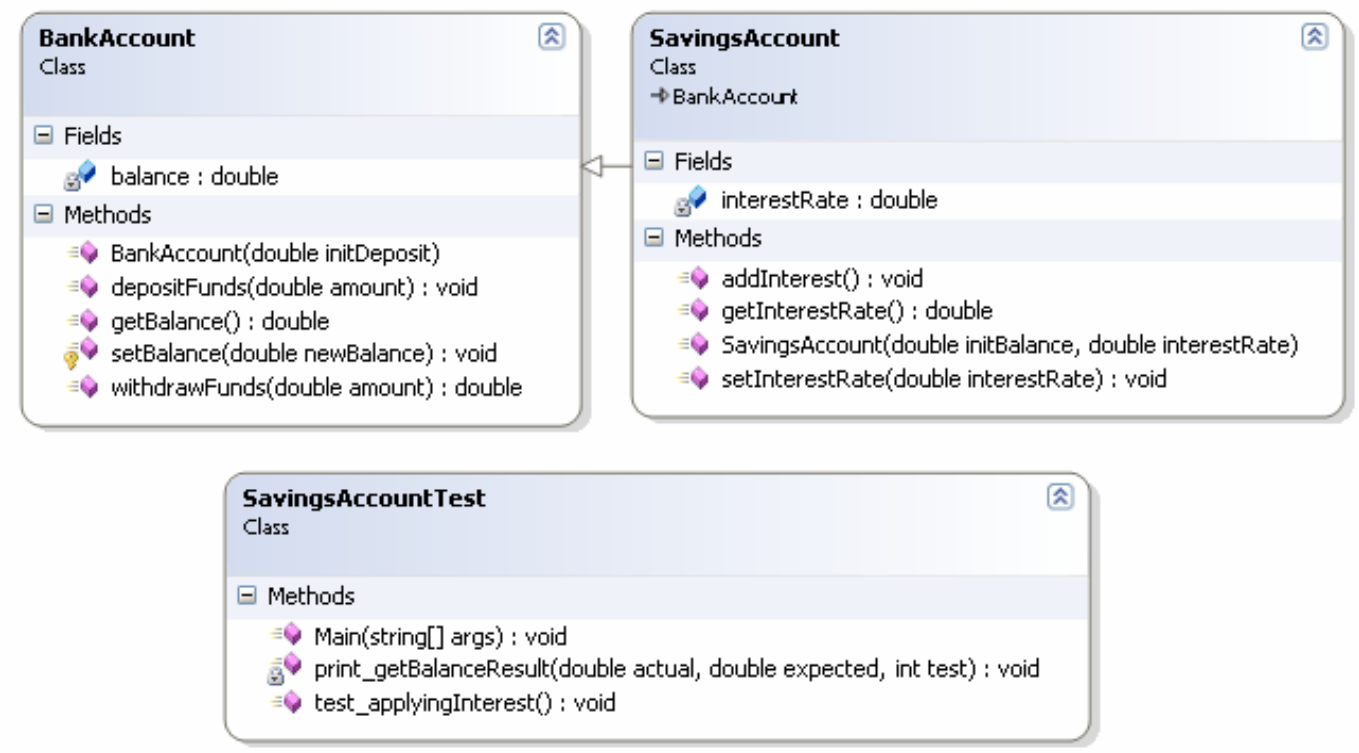


Рис. 6. Диаграмма классов программы на **листинге 4**

Класс **SavingsAccountTest (строки 83…114)** тестирует один из элементов программы — класс **SavingsAccount (строки 41…81)**. Модуль тестирования может состоять из совокупности контрольных примеров. В нашем случае, класс **SavingsAccoutTest** имеет только один контрольный пример: **test_applyingInterest (строки 92…97)**. Контрольный пример **test_applyingInterest** позволяет убедиться в том, что метод **addInterest() (см. строки 95 и 61…70)** класса **SavingsAccount**, приведенный в качестве примера, правильно учитывает процент в балансе. В процессе тестирования элементов программы **может выполняться большое количество контрольных примеров**, но каждый из них должен проверять **только одно из свойств объекта**.

Класс **SavingsAccountTest** является **модулем тестирования** элементов программы **(см. раздел 3.3.1, с.27,сл.)**, потому что он проверяет объект — наиболее низкоуровневый блок, или **модуль** в объектно-ориентированном мире. При тестировании элемента программы **за один раз должен исследоваться только один объект**. Это означает, что **каждый объект должен быть пригодным для испытания как самостоятельный элемент**. Если это не так, то объект, возможно, нельзя протестировать вообще. А если протестировать и удастся, то может случиться так, что, хотя и неумышленно, вам придется одновременно тестировать довольно много объектов.

При написании автономных тестов необходимо в максимально возможной степени избегать ручной проверки правильности вывода. Как можно видеть в **test_applyingInterest**, контрольный пример **автоматически** выполняет все необходимые проверки правильности. Ручная проверка правильности часто требует много времени и служит источником ошибок. При прогоне контрольных примеров программист желает получить точные результаты как можно быстрее, причем с минимальной затратой усилий. В противном случае он может начать пренебрегать тестированием.

3.4.3. Почему необходимо писать автономные тесты

Автономные тесты[2] помогают обнаружить ошибки. Если что-нибудь не так в написанном вами классе, вы узнаете об этом сразу же, так как автономный тест сообщит об ошибке. **Автономные тесты** — первая линия защиты от возникновения ошибок в программе. **Перехватить ошибки на уровне отдельного класса намного проще**, чем пытаться выследить ошибку в ходе проверки взаимодействия и функционирования компонентов системы и испытаний системы.

---

[2] **Под автономными тестами** (**unit test**) в данном случае подразумевается любая система (например, **программа или модуль**), **которая используется при прогоне контрольных примеров.**

[3] Если, конечно, удастся написать все необходимые автономные тесты и выполнить их все. Именно потому и существует столь обширная (и столь сложная!) теория (нет, скорее даже философия!) тестирования, что для сколько-нибудь сложной программы не удается выполнить ни одно из указанных выше условий! Например, для проверки правильности программы сложения двух вещественных чисел необходимо на вход программы подать всевозможные пары вещественных чисел! Кроме того, написание программ, проверяющих выполнение спецификаций в разрабатываемых программах, ничуть не более простая задача, чем написание самих программ. Поэтому в программах, проверяющих выполнение спецификаций в разрабатываемых программах, ошибок может быть больше, чем в тестируемых. А потому для них также необходимо написать тестирующие их программы, для которых, в свою очередь... Ну, в общем, вы поняли... Правда, иногда можно считать, что в качестве программы, тестирующей программу, тестирующую программу **A**, можно взять саму программу **A**! Но в этом случае придется протестировать их обе одновременно**...** Конечно, если бы оказалось, что для любой программы можно автоматически построить проверяющую ее программу, то задача тестирования была бы разрешена раз и навсегда, причем для всех программ! Но построение программы, проверяющей какое-либо нетривиальное свойство (например, свойство завершать свое выполнение после конечного числа шагов) на множестве программ — алгоритмически неразрешимая задача. Так что рассчитывать на полностью автоматический критерий остановки в тестировании не приходится.

Автономные тесты также автоматически позволяют узнать, когда проектирование класса закончено: класс готов, когда без ошибок выполняются все контрольные примеры![3] Для разработчика чрезвычайно полезно иметь такой четкий критерий остановки. В противном случае, могут возникнуть трудности при попытке определить, "готов" ли класс. Знание того, что разработка класса завершена и можно переходить к написанию следующего, уберегает от соблазна добавить в класс больше, чем необходимо, функциональных возможностей.

Написание автономных тестов помогает при разработке самого класса, особенно в том случае, если контрольные примеры создаются до реализации самого класса; однако, чтобы написать контрольный пример, необходимо вообразить, что класс уже существует. Поступая так, вы получаете большую свободу экспериментирования с интерфейсом ваших классов. Как только заканчивается написание автономного теста, можно реализовывать класс, а затем все компилировать.

Тестирование элементов программы может помочь разложить код на отдельные блоки. Изменения делать проще в том случае, когда есть автономный тест, поскольку существует постоянная обратная связь, благодаря которой вы можете практически мгновенно испытать проделанные в коде изменения. При этом нет необходимости сильно волноваться по поводу возникновения ошибок; ведь можно просто повторно прогнать тесты, чтобы увидеть, не испортилось ли что-нибудь. **Автономный тест позволяет избавиться от надоедливого вопроса, который так часто задает себе программист: "не нарушил ли я что-нибудь?".** А избавившись от этого вопроса, можно делать изменения с большей уверенностью.

Наконец, вы не обязаны зацикливаться на испытаниях системы. Вы можете заняться другими проектами, а **испытания написанных вами классов можете поручить другим членам вашей команды**. Ведь при наличии автономного теста проводить испытания объектов может не только разработчик классов, но и кто-то другой.

3.4.4. Написание автономных тестов

SavingsAccountTest и **test_applyingInterest() (строки 83… и 92… в листинге 4 на с.31)** — довольно простые образцы автономных тестов и контрольных примеров. Однако написание автономного теста с самого начала для каждого класса может потребовать много времени. Вообразите систему, для которой необходимо написать сотни контрольных примеров. Если писать автономные тесты для каждого класса с нуля, придется выполнить много лишней работы. Поэтому необходимо либо создавать, либо **повторно использовать каркас тестирования**.

Каркас — повторно используемая концептуальная модель. **Каркас содержит все классы**, общие для всей предметной области, и служит основой для конкретного приложения в этой области.

Классы, из которых состоит каркас, определяют общий проект приложения. Чтобы создать приложение, разработчик просто расширяет существующие и затем создает свои собственные классы, ориентированные на решение конкретной проблемы.

Вернемся к проблеме тестирования. Каркас тестирования определяет скелет (программы), который можно многократно использовать для написания и выполнения автономных тестов. Каркас тестирования **позволяет быстро и удобно писать автономные тесты**, устраняя необходимость вручную выполнять избыточную работу, во время которой столь часто делаются ошибки. Необходимо помнить, что все, что вы программируете, может содержать ошибки, даже код, выполняющий тестирование.

Наличие хорошо проверенного каркаса тестирования помогает избежать многих ошибок тестирования. Если же каркаса тестирования нет, то для предотвращения ошибок тестирования понадобятся дополнительные затраты на испытания.

Законченный скелет тестирования содержит: **1) базовые** классы **для написания автономных тестов**, **2) встроенную поддержку для автоматизирования испытаний**, а также **3)** утилиты, помогающие интерпретировать и выводить сообщения. Разработан свободно распространяемsq каркас тестирования **JUnit**, выпущенный под "**IBM Public License**" ("**Публичная лицензия IBM**") для испытания **Java**-классов. **JUnit** можно загрузить с **Web**-узла по адресу: http://www.junit.org/ .

Загружаемый из **сети (WWW)** каркас тестирования **JUnit** содержит исходный текст. **JUnit** превосходно спроектирован, так что очень полезно изучить код и сопровождающую документацию. В частности, эта документация представляет собой превосходный пример документирования шаблонов проектирования, использованных для написания **JUnit**.

3.5. JUnit (http://www.junit.org/ - загрузите **JUnit** и прочитайте **Cookstour**, который находится в каталоге **doc**)

JUnit имеет**: 1)** классы для создания автономных тестов, **2)** классы для проверки правильности выходных данных и **3)** предоставляет **(а)** графический интерфейс пользователя **(GUI)** или **(б)** среду командной строки для выполнения контрольных примеров. **junit.framework.TestCase** — **базовый** класс для определения автономных тестов. **Для написания автономных тестов нужно просто создать производный от TestCase класс, который (1) перегружает несколько методов и (2) предоставляет собственные методы для выполнения контрольных примеров**.

Создавая контрольные примеры для **JUnit**, **необходимо начинать название всех методов контрольных примеров с test** (**NB**). JUnit имеет механизм, автоматически загружающий и выполняющий любой метод, имя которого начинается с **test**.

В листинге 5 представлена **JUnit**-версия **SavingsAccountTest**.

Листинг 5. SavingsAccountTest.Java

```
1  import junit.framework.TestCase;                                    Листинг 5
2  import junit.framework.Assert;
3
4  public class SavingsAccountTest extends TestCase
5  {
6                  // Общий класс SavingsAccountTest расширяет базовый класс TestCase
7      public void test_applyingInterest()          // это контрольный пример
8      {
9      SavingsAccount act = new SavingsAccount(  10000.00, 0.05  );
10     act.addInterest();              // это проверяемый метод
11     Assert.assertTrue( "Процентная ставка  установлена неправильно",
                                                    act.getBalance() == 10500.00 );
12     }
13
14     public SavingsAccountTest(String name)       // см. строку 1… в листинге 6
15     {
16             // (Строковое название)
17             super(name); // название
18     }
19 }
```

JUnit-версия класса **SavingsAccountTest** значительно проще, чем оригинал (**листинг 4, строки 83…**). Используя **JUnit**, нет необходимости дублировать логические проверки или код, выводящий

сообщения. Можно просто использовать класс **Assert (см. строки 2 и 11)**, поставляемый в составе **JUnit**. Класс **Assert** предоставляет методы, принимающие булево выражение в качестве аргумента. Если булево выражение равно **false**, регистрируется ошибка. В нашем примере тест передает **Assert** булево выражение, возвращаемое как результат сравнения **act.getBalanse() == 10500.00**. Если в результате вычисления сравнения получается **false**, **JUnit** зарегистрирует ошибку.

Но как же запустить эти тесты?

В **JUnit** предусмотрено несколько вариантов прогона контрольных примеров. Эти варианты делятся на две категории**: 1)** статические и **2)** динамические. Выбрав использование статического метода, необходимо переопределить метод **runTest()**, чтобы вызвать тот контрольный пример, который необходимо запустить.

В **Java** наиболее удобным является **использование анонимных классов**, если нет желания создавать отдельный класс для каждого из тестов, которые необходимо запустить. Пример анонимного объявления приведен в листинге 6.

Листинг 6. Анонимный SavingsAccountTest

```
1  SavingsAccountTest test = new SavingsAccountTest(  " Тест  Процентной ставки  " ) // см. стр. 14 в лист.5
2   {
3   public void runTest()
4   {
5  test_applyingInterest();          // см. строку 7 в листинге 5
6  }
7  };
8  test.run();
```

Анонимные классы удобны потому, что они позволяют переопределить метод при создании объекта. И все это без необходимости создавать именованный класс в отдельном файле. В данном примере в главном методе (**Main**) создается **SavingsAccountTest** **(строки 4… в листинге 5)**, но метод **runTest()** переопределяется так, чтобы прогнать контрольный пример **test_applyingInterest()**.

Анонимный класс — класс, который не имеет названия. **Анонимные классы не обладают именем**, потому что они определяются (доопределяются) непосредственно при создании экземпляра, т.е. объекта. Они не объявляются в отдельном файле или в качестве внутренних классов.

Анонимные классы — превосходный выбор для однократно используемых классов (если класс маленький). **Анонимные классы избавляют от необходимости создавать отдельные именованные классы.**

Несмотря на то, что анонимные классы предоставляют определенные удобства, они имеют и **недостатки. Необходимо создавать отдельный анонимный класс для каждого вызываемого метода контрольного примера.** Если контрольных примеров много, использование анонимных классов может повлечь за собой написание большого количества избыточного кода.

Чтобы преодолеть этот недостаток, в **JUnit** предусмотрен также динамический механизм, **который ищет и выполняет любой метод, имя которого начинается с test**. Воспользуемся этим динамическим механизмом.

Для выполнения нескольких контрольных примеров в **JUnit** предусмотрен механизм, называемый **тестовым (испытательным) набором** или **тестовым комплектом**. **JUnit** имеет механизм статического определения набора тестов, которые прогоняются в комплекте; тем не менее, **динамический механизм все равно найдет все методы тестирования**. Будем полагаться на этот ав-

томатический механизм поиска и выполнения тестов.

В **JUnit** предусмотрено также еще несколько других удобных возможностей. Рассмотрим модифицированную версию **SavingsAccountTest** в листинге 7, который также испытывает метод **withdrawFunds()**.

Листинг 7. SavingsAccountTest.Java

```
1  import junit.framework.TestCase;
2  import  junit.framework.Assert;
3
4  public class SavingsAccountTest extends TestCase
5  {                                  // Общий класс SavingsAccountTest расширяет TestCase
6  private SavingsAccount act;        // act – это переменная типа проверяемого класса
7
8        public void test_applyingInterest()     // это контрольный пример  №1
9       {
10       act.addInterest();                        // это проверяемый метод
11      Assert.assertTrue( " Процентная ставка  установлена неправильно ",
                                                  act.getBalance() ==  10500.00 );
12      }
13
14     public void test_withdrawFunds()       // это контрольный пример  №2
15     {
16     act.withdrawFunds(500.00);             // это проверяемый метод
17     Assert.assertTrue( " Забирается неправильная сумма ", act.getBalance() == 9500.00 );
18     }
19
20      protected void setup() // это переопределяющий метод -
21     {                              // переопр-ся м-д базового класса TestCase
22     act = new SavingsAccount( 10000.00, 0.05 );          // установка
23     }
24
25    public SavingsAccountTest(String name)       // см. строку 1… в листинге 6
27    {                       // Строковое название
28     super( name ); // название
29     }
30 }
```

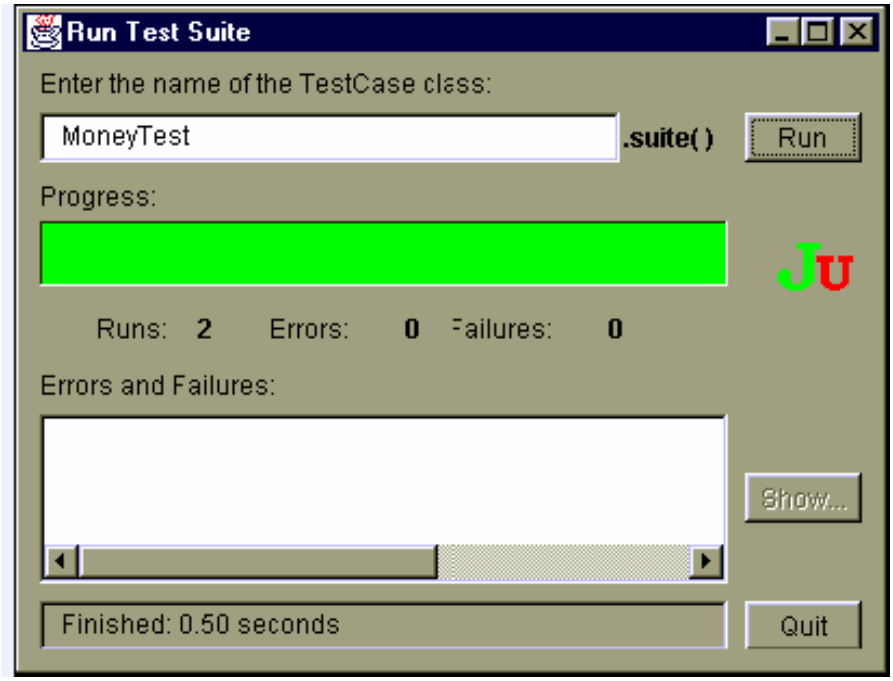
Обратите внимание, что в этой версии тест содержит два новых метода: метод **test_withdrawFunds()** (**строки 14…18**) и **setup()** (**строки 20…23**). При этом **setup()** переопределяет метод в **базовом** классе **TestCase**. Каждый раз, когда **JUnit** вызывает тестовый метод, сначала вызывается **setup()**, чтобы установить "**испытательную арматуру**".

Примечание	
	Испытательная арматура определяет множество объектов, которые будут испытываться с помощью данного теста. Именно на создание испытательной арматуры иногда приходится потратить большинство времени, необходимого для написания контрольных примеров. Испытательная арматура подготавливает множество объектов, которые будут испытываться с помощью данного контрольного примера. Арматуры удобны также потому, что одну и ту же арматуру можно совместно использовать с разнообразными наборами контрольных примеров, избегая, таким образом, необходимости дублирования кода.

Вызывая **setup()**, **JUnit** гарантирует, что объекты арматуры будут в известном состоянии перед выполнением каждого теста. **JUnit** также предоставляет соответствующий метод **clearDown()** для очистки арматуры после прогона тестов.

Для выполнения контрольных примеров в **JUnit** предусмотрена утилита прогона тестов, которая осуществляет тестирование и собирает результаты испытаний. В **JUnit** имеется две версии этой утилиты: **1)** графическая и **2)** основанная на командной строке.

Для выполнения **SavingsAccounTest** в среде графического интерфейса необходимо набрать **java junit.swingui.TestRunner**



На **рис. 7** показано основное окно утилиты **JUnit.**

С помощью интерфейса пользователя выполняется поиск класса **SavingsAccountTest.** Загрузив его, выполнять тестирование можно просто нажимая кнопку **Run** (**пуск**).

JUnit — великолепный инструмент, позволяющий получать четкую, мгновенную обратную связь с контрольными примерами. Так что, если после внесения изменений в программу вас беспокоит вопрос "**Может быть, я нарушил что-нибудь?**", вы можете быстро узнать ответ, просто запустив тестирование повторно.

3.6. Создание эффективной документации

Есть еще один способ, с помощью которого можно **улучшить качество работы: задокументировать ее.** Имеется много форм документирования, каждая с собственным уровнем эффективности.

3.6.1. Исходный текст программы как документация

Исходный текст и даже автономные тесты являются одним из возможных типов документации. Если сопровождение написанной вами программы придется вести другим людям, важно, чтобы код был читабельным и хорошо отформатированным. Иначе никто не сможет осмыслить код и извлечь из него пользу.

Исходный текст — наиболее важная форма документации, потому что это — единственный вид документирования, который необходимо поддерживать обязательно.

3.6.2. Соглашения по программированию

Первый шаг, который поможет превратить код в хорошую документацию, **состоит в выборе соглашения по программированию и в том, чтобы придерживаться этого соглашения**. Соглашения по программированию могут охватывать все**:** от выравнивания фигурных скобок до именования переменных. Определенное соглашение не является чем-то важным. Важно то, чтобы команда, работающая над проектом, и, предпочтительно, вся компания, **выбрали** то или иное **соглашение** и **придерживались его**. Тогда любой программист сможет усовершенствовать любую часть кода, по крайней мере, не отвлекаясь на форматирование.

Листинг 8 представляет один из возможных вариантов объявления **C#-классов**.

Листинг 8. Образец оформления класса

```
public class <ClassName> : <BaseClassName> , <LIST OF INTERFACES>
{
// Общие переменные (public)
// Защищенные переменные (protected)
// Частные переменные (private)
// КОНСТАНТЫ
// Общие методы (public)
// Защищенные методы (protected)
// Частные методы (private)
}
```

Имена классов и названия методов должны всегда начинаться с **Прописной** Буквы. **Имена переменных** должны всегда начинаться со **строчной** буквы.

В любом составном названии, т.е., состоящем из нескольких слов, каждое вложенное слово начинается с заглавных букв. Например: **someMethod()** и **HappyObject.**

Константы всегда набираются **ПРОПИСНЫМИ** (ЗАГЛАВНЫМИ) БУКВАМИ.

Вот один из способов объявления методов и вложенных операторов **if/else**:

```
public void Method()
{
        if (условный оператор)
        { ….
        }
   else
  {….
   }
}
```

3.6.3. КОНСТАНТЫ

Константы могут также служить формой документирования. Константы желательно использовать везде, где используете жестко запрограммированное значение. Удачно названные константы могут облегчить понимание назначения кода.

3.6.4. Комментарии

Подобно удачно использованной константе, ничто так не помогает сделать код более понятным, как хорошо расположенный комментарий. Однако необходимо найти равновесие в использовании комментариев. **Если комментировать слишком много, комментарии потеряют свое значение**.

Вот пример бесполезного использования комментариев — довольно обычной ошибки:

```
public void id(int id)
        {
                this.id = id; // устанавливает id
        }
```

Подобного типа комментарии сообщают о том, что делает код. **А если необходимо разъяснять код, вероятно, он слишком запутанный. Комментарии сообщают, для чего предназначен код.** Они должны описывать непонятные интуитивно выражения или части программы.

NB	Обратите внимание, что комментарий под сигнатурой метода не заменит удачно подобранного имени метода.

3.6.5. Названия (имена)

Названия переменных, методов и имена классов должны быть содержательными, причем необходимо последовательно придерживаться выбранного правописания. Например, всегда печатать с прописной буквы второе слово составного названия типа **testCase**. В качестве альтернативы, можно разбивать слова, используя знак подчеркивания (_), как в **test_case**. Наиболее важна **не**противоречивость. Необходимо сделать правило присваивания имен частью соглашения по программированию и затем **неукоснительно следовать** ему.

3.6.6. Методы и заголовки классов

При записи класса или метода необходимо обязательно предусмотреть заголовок. **Заголовок метода включает** описание, список параметров, описание возвращаемых значений, а также условия возникновения исключительных ситуаций и описание побочных эффектов. Заголовок может даже включать некоторые предусловия. **Заголовок класса обычно включает** описание, номер версии, список авторов и историю исправлений.

Когда программируете в **C#**, воспользуйтесь **XML**-документацией (**E**xtensible **M**arkup **L**anguage).

Резюме

Познакомились с тестированием и узнали, что может сделать разработчик, чтобы гарантировать качество своей работы. Существует четыре всеохватывающих формы тестирования:

1. тестирование элементов программы (блочное тестирование);
2. проверка взаимодействия и функционирования компонентов системы;
3. испытания системы (системное тестирование);
4. регрессивное тестирование.

В ежедневной работе тестирование элементов программы — первая линия защиты против ошибок. Тестирование элементов программы заставляет рассмотреть проект с позиций испытания, и предоставляет механизм, упрощающий переразложение на части.

Вы также увидели важность правильной обработки состояний ошибок и поддержания документации. Все это способствует повышению качества кода.

Вопросы и ответы

Почему разработчики ненавидят тестирование?

Кажется, среди программистов имеется культура "избегания тестирования". Я чувствую, что это проблема культуры программирования. В большинстве компаний **группа поддержки качества (QA-group)** отделена от разработчиков. Они приходят, испытывают систему и пишут отчеты об ошибках. Такой подход заставляет программистов обороняться, особенно, если менеджер проекта возлагает ответственность за недоделки на разработчиков. В этом случае тестирование представляется каким-то наказанием и источником дополнительной работы.

Взгляд на тестирование как на дополнительную работу — также часть проблемы. Многие проектные группы откладывают испытания до конца разработки. Таким образом, это становится чем-то, что нужно выполнять после окончания "настоящей" работы. К сожалению, чем больше затягивать с испытанием, тем тяжелее его будет выполнить потом. А когда вы приступите к испытанию, вероятно, сразу столкнетесь с большим количеством ошибок, так как испытаний до этого момента не проводилось. И это, в свою очередь, отбрасывает разработчиков далеко назад, в "фазу наказания за все недоделки, допущенные на самых начальных этапах разработки". Поэтому разработчикам приходится трудиться в тяжелом режиме работы, особенно если учитывать вероятную близость окончания отведенного срока. А чем ближе к концу отведенное время, тем больше давление со стороны менеджера проекта.

Желание избегать тестирования — проблема, подкармливающая сама себя: чем больше вы хотите избежать испытаний, тем неприятнее они будут.

Почему испытания программного продукта часто выполняются отдельной группой поддержки качества?

Независимые испытания помогают гарантировать, что испытатели подсознательно не избегут областей, где могут существовать проблемы — недостаток, к которому иногда склонны разработчики.

На протяжении сегодняшней лекции мы рассматривали JUnit. Почему был выбран JUnit?

JUnit — свободно распространяемый и хорошо работающий инструмент тестирования. **JUnit** хорошо разработан и сделан достаточно небольшим, так чтобы разработчик мог легко управлять разработкой проекта. В **JUnit** также отсутствуют украшения, бантики и другие ненужные свойства, которые затрудняют использование многих других продуктов.

Однако, **JUnit** — каркас, **предназначенный исключительно для тестирования элементов программы**. Поэтому **для проверки взаимодействия и функционирования компонентов системы и испытаний системы необходимо найти другие инструментальные средства**.

Тестирование компонентов похоже на бремя. У меня нет времени на тестирование элементов программы. Что я должен делать?

Тестирование компонентов похоже на бремя только в первый раз, во время написания тестов для элементов программы. Справедливости ради необходимо отметить, что иногда написание тестов становится достаточно ресурсоемким. Написание тестов для элементов программы окупается только через некоторое время. Однократно написанный тест можно использовать многократно. Каждый раз, изменив реализацию класса, можно просто повторно прогнать тест. Автономные тесты упрощают процесс изменения кода.

Что касается отсутствия времени, то это — слабый аргумент. Вообразите, сколько времени понадобится для того, чтобы найти, проследить и исправить ошибки, если испытания будут откладываться до самого конца, когда обычно разработчик находится под еще большим давлением.

Как узнать, выполнено ли достаточное количество тестов?

Необходимо произвести проверку на минимальном уровне, выполнив тестирование тестов для элементов программы для каждого класса, проверку взаимодействия и функционирования компонентов системы для всех главных взаимодействий в системе и испытание системы для каждого варианта использования. Проводить или нет дополнительные испытания, зависит от конечного срока завершения проекта, а также от

уровня собственного удобства.

Контрольные вопросы и упражнения

1. Как ошибки могут проникнуть в программное обеспечение? (Или, возможно, вопрос должен звучать так: "Почему в процессе разработки программного обеспечения возникают ошибки?".)
2. Что такое контрольный пример?
3. Назовите два метода, которые можно положить в основу испытаний.
4. Дайте определение тестирования методом "прозрачного ящика" и испытания методом "черного ящика".
5. Назовите четыре формы тестирования.
6. Определите понятие **тестирования элементов программы**.
7. В чем смысл **: 1)** проверки взаимодействия и функционирования компонентов системы и **2)** испытания системы?
8. Почему нельзя откладывать испытания до окончания проекта?
9. Почему необходимо избегать ручной или визуальной проверки правильности при тестировании? В чем состоит альтернатива такой проверки?
10. Что такое каркас?
11. Что такое фиктивный объект?
12. Почему необходимо использовать фиктивные объекты?
13. Каково различие между ошибкой и состоянием ошибки?
14. Как гарантировать качество программы во время написания кода?

Упражнения

1. Из **Internet** (http://www.junit.org/) загрузите **JUnit** и прочитайте **Cookstour**, который находится в каталоге **doc**.

Ответы на вопросы и упражнения

1. Ошибки могут возникать в результате опечаток, логических ошибок или "глупых" ошибок, сделанных при кодировании. Они также могут появляться вследствие некорректного взаимодействия объектов или ошибок анализа или проектирования.
2. Контрольный пример — стандартный блок испытания. Каждая форма тестирования состоит из контрольных примеров, и каждый из них проверяет определенный аспект системы.
3. В основу построения контрольных примеров можно положить метод "**черного** ящика" либо метод "**прозрачного** ящика".
4. Метод "прозрачного ящика" основан на структуре проверяемого кода. Испытания методом "прозрачного ящика" предназначаются для того, чтобы охватить 100 % проверяемого кода.

   В основу испытаний методом "черного ящика" кладутся технические требования. Испытания методом "черного ящика" проводятся для того, чтобы удостовериться, что система ведет себя так, как ожидается.
5. Основные формы тестирования:
   - *a)* блочное тестирование (автономное тестирование, тестирование элементов программы, или тестирование модуля);
   - *b)* тестирование сопряжений (проверка взаимодействия и функционирования компонентов системы, комплексные испытания, компоновочные испытания, совместные испытания);
   - *c)* испытание системы (проверка системы, системные испытания, системный тест) и регрессивное (возвратное) тестирование.

6. Блочное тестирование (автономное тестирование, тестирование элементов программы, или тестирование модуля) — испытательный механизм самого нижнего уровня. При проведении автономного тестирования объекту посылается сообщение и проверяется получение ожидаемого результата. При автономном тестировании за один раз проверяется только одна функция.

7. При комплексных испытаниях (тестировании сопряжений, проверке взаимодействия и функционирования компонентов системы, компоновочных испытаниях, совместных испытаниях) проверяется правильность взаимодействия объектов.
   При испытаниях системы (проверка системы, системные испытания, системный тест) следует убедиться в том, что система ведет себя в полном соответствии с тем, как предписано в случаях использования (прецедентах), а также что система может адекватно функционировать в непредвиденных случаях использования.

8. Не следует откладывать испытания на конец работы. Испытания, по мере их выполнения, помогают сразу же обнаружить ошибки. Если же отложить испытания до окончания работы, ошибок накопится много, а выявить и устранить их будет значительно сложнее.

   Испытания, проводимые в процессе разработки, помогают изменять код и могут существенно улучшить проект.

9. При ручной или визуальной проверке правильности легко допустить ошибки. Следует стремиться избегать такой проверки вообще. Вместо такой проверки следует всецело положиться на автоматический механизм подтверждения правильности результатов тестирования элементов программы.

10. Каркас определяет многократно используемую концептуальную модель. В качестве базы для своего специфического приложения можно использовать классы этой модели.

11. Фиктивный (пробный, ложный, имитирующий) объект — это упрощенная замена реального объекта, которая помогает протестировать объекты.

12. Фиктивные (пробные, ложные, имитирующие) объекты позволяют провести автономные испытания отдельных классов. Они также предоставляют такие возможности проверки, реализовать которые в иных условиях было бы трудно или даже невозможно.

13. Ошибка является результатом недоработки или дефекта в системе. Состояние ошибки (аварийная ситуация), с другой стороны, не является дефектом, а, скорее, возникшим в процессе функционирования условием (состоянием), которое должно быть предусмотрено при разработке системы, причем в случае возникновения такого условия (состояния) система должна правильно функционировать при этом условии и должным образом обработать его.

14. При написании кода можно гарантировать его качество с помощью блочного тестирования (тестирования элементов программы), корректной обработки исключений и соответствующей документации.

Ответы к упражнениям

1. В документе **Cookstour** описан проект и концепция **JUnit**.

4. Реалии индустрúи и объектно-ориентированное программирование (ООП)

В курсе рассмотрено объектно-ориентированное программирование (ООП) с азов. Предполагалось, что вам нет нужды использовать в своей последующей работе системы, созданные не на основе объектно-ориентированного подхода, что у вас нет наработок, основанных на использовании процедурного программирования, что вы начинаете с чистого листа. Иными словами предполагалось, что в ваших проектах не используются библиотеки процедур, а все, что вы используете, разработано с применением объектно-ориентированного подхода.

Но на самом деле во многих случаях вам придется иметь дело с компонентами, весьма далекими от объектно-ориентированного подхода. Рассмотрим, например, **реляционные базы данных**. **Обычно они не являются объектно-ориентированными**. Объектно-ориентированные базы данных – большая редкость и вне узких отраслей промышленности все еще используются редко.

Столкнувшись с подобной необходимостью, лучше всего смириться и создать для таких компонентов объектно-ориентированную оболочку (упаковщик). Например, при работе с реляционными базами данных лучше всего создать слой файловых объектов. Вместо того чтобы напрямую обращаться из программы к базе данных, и создавать нужные вам объекты с помощью запросов на языке **SQL** (**S**tructured **Q**uery **L**anguage — язык структурированных запросов – международный стандартный язык для определения и доступа к реляционным базам данных), сделайте так, чтобы ваши объекты обращались к посреднику — слою файловых объектов, а тот уже взаимодействовал с базой данных.

На самом деле совершенно нереально перечислить все случаи, в которых вам придется сталкиваться с системами, не являющимися объектно-ориентированными.

Пройдет немало времени, прежде чем все программные системы будут переделаны в системы с объектно-ориентированной архитектурой (если это вообще когда-нибудь произойдет). Вы должны быть предупреждены о подобных сложностях и готовы к элегантному их преодолению.

Резюме к курсу ООАП

Вот и все**!** Всего за **15*** (**!!!???**) лекций были изложены основы объектно-ориентированного программирования. Дальше все зависит от вас самих. К этому моменту вы обладаете некоторыми знаниями для того, чтобы начать применять объектно-ориентированное программирование (ООП) в вашей работе. Удачи!

Изучение объектно-ориентированного программирования (ООП) не должно ограничиться этим курсом. Напишите список тем, которые вы хотите изучить глубже. Расставьте их по важности, ищите в **Internet** материалы [**9…13**] по этим темам (пользуйтесь моими материалами [**4**]) и учитесь, учитесь, учитесь!

Настоящее знание ООАП – это интересная, престижная и высокооплачиваемая работа **!!!**

---

* Отметим, что в Swiss Federal Institute of Technology Zurich (это бывший Цюрихский политехникум, где учились Альберт Эйнштейн и Джон фон Нейман и работал Никлаус Вирт, автор языков программирования Pascal, Modula, Oberon) в рамках курса по объектно-ориентированному программированию на первом году обучения еженедельно проводятся две лекции и три практических занятия.

Литература к курсу

Базовый учебник

1. Мейер Б. Объектно-ориентированное конструирование программных систем. М.: Русская Редакция, 2005.

Основная

2. Буч Г., Якобсон А., Рамбо Дж. **UML**. С.-Петербург: Питер, 2006.

3. Гамма Э. Приемы объектно-ориентированного проектирования. Паттерны проектирования. С.-Петербург: Питер, 2006.

4. Забудский Е.И. Учебно-методические материалы по дисциплине «Объектно-ориентированный анализ и программирование». М.: Кафедра ОИиППО ГУ-ВШЭ, 2005, 2006, 2007.

Internet-ресурс – http://new.hse.ru/C7/C17/zabudskiy-e-i/default.aspx .

5. Кватрани Т. Визуальное моделирование с помощью Rational Rose 2002 и UML. М.: Вильямс, 2003.

6. Лафоре Р. Объектно-ориентированное программирование в C++. С.-Петербург: Питер, 2006.

7. Троелсен Э. C# и платформа .NET. С.-Петербург: Питер, 2006.

8. Синтес А. Освой самостоятельно объектно-ориентированное программирование за 21 день. Москва; С.-Петербург; Киев: Вильямс, 2002.

Дополнительная – **Internet-ресурсы**

9. Новые книги раздела **C#** – http://books.dore.ru/bs/f6sid16.html

10. C# и **.NET** по шагам – http://www.firststeps.ru .

11. UML – язык графического моделирования – http://www.uml.org/ .

12. JUnit – каркас тестирования для испытания *Java*-классов – http://www.junit.org .

13. Пакет объектного моделирования **Rational Rose** – http://www-306.ibm.com/software/rational/

Дополнительная – книги

14. Мэтт Вайсфельд. Объектно-ориентированный подход: Java, .NET, C++. М.: КУДИЦ-ОБРАЗ, 2005.

15. Дж. Кьоу, М. Джеанини. Объектно-ориентированное программирование. С.-Петербург: Питер, 2005.