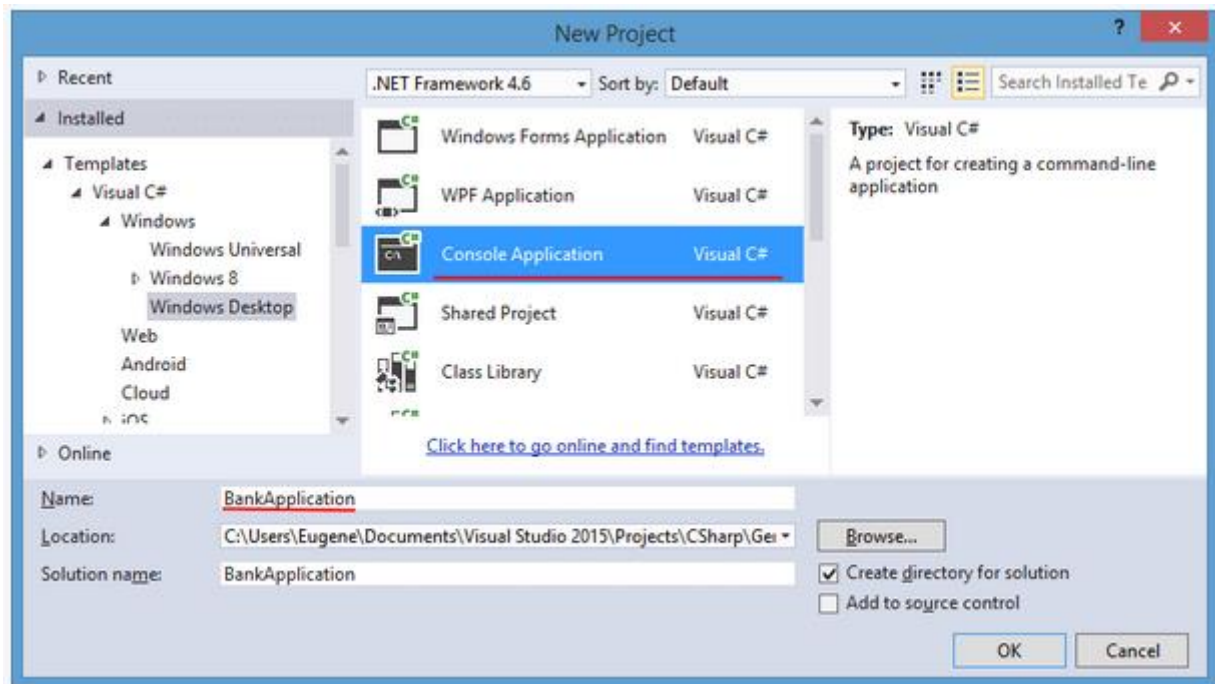


## Создание приложения BankApplication

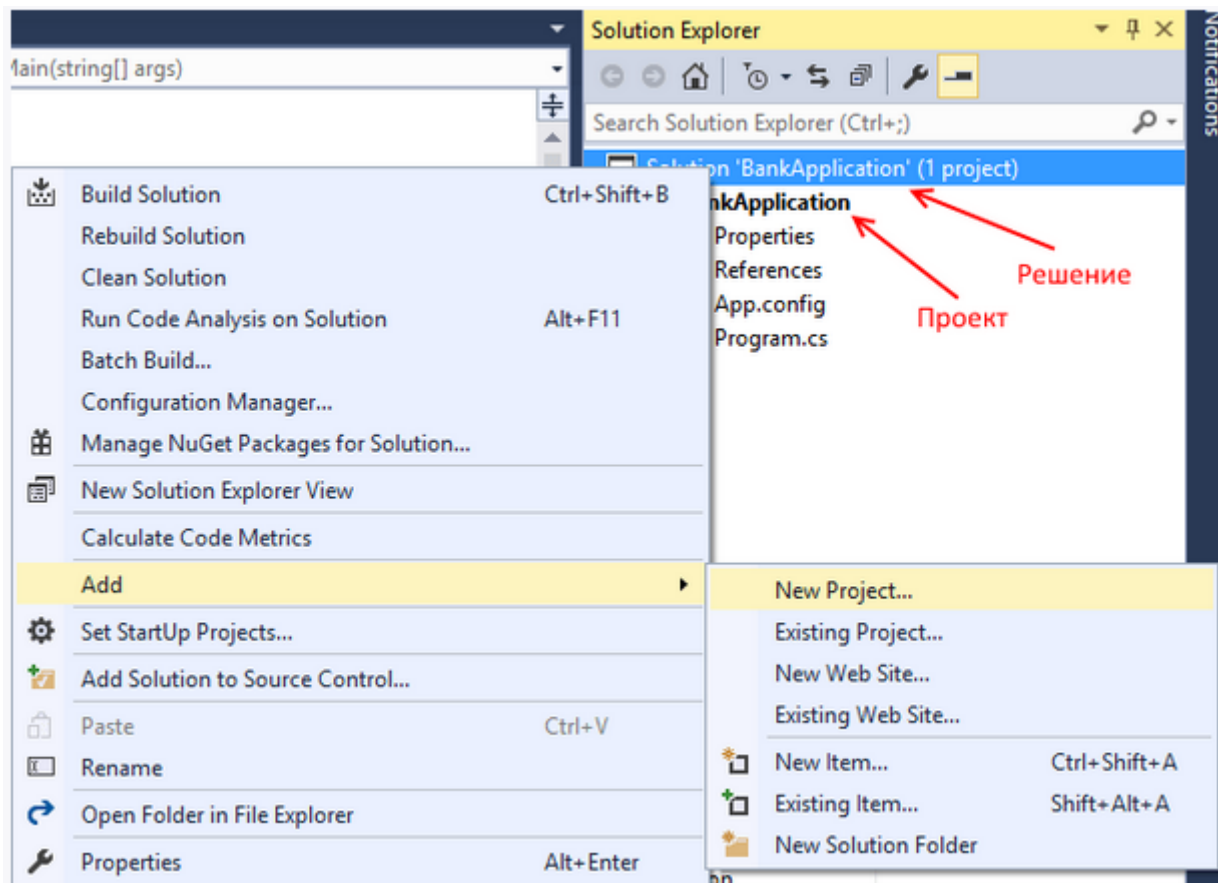
Вначале создадим новый проект по типу Console Application, который назовем BankApplication. Он будет представлять собой прикладное приложение, с которым будет взаимодействовать пользователь:



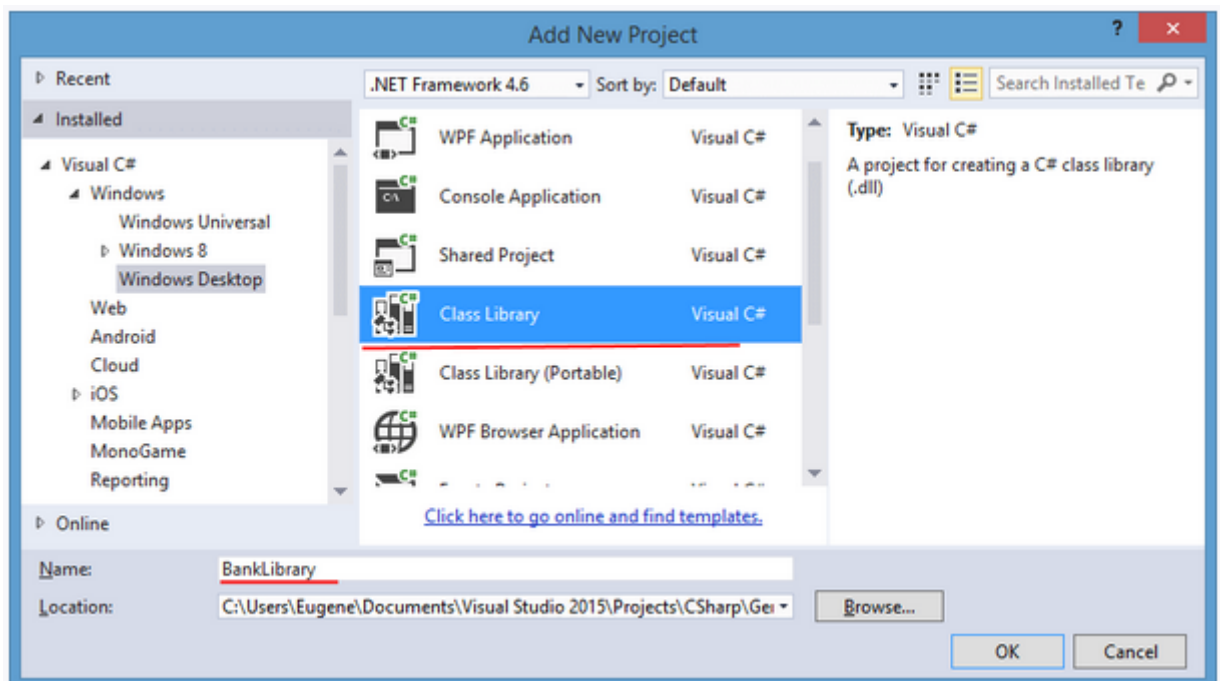
И после этого создается стандартный пустой проект с классом Program и методом Main. Это будет стартовый проект приложения, представляющий интерфейсную часть разрабатываемого программного обеспечения.

## **Создание проекта библиотеки классов**

Для хранения классов и интерфейсов, представляющих бизнес-логику приложения, нередко создаются отдельные проекты, в рамках которых все классы компилируются в файл библиотеки dll, которая затем подключается к главному проекту. Поэтому добавим в решение новый проект. Для этого нажмем правой кнопкой мыши на решение и выберем в контекстном меню Add -> New Project...:



В качестве типа нового проекта выберем шаблон Class Library (Библиотека классов) и назовем новый проект BankLibrary:



После этого в решение добавляется новый проект, который по умолчанию имеет один файл *Class1.cs*. Удалим этот файл. Этот проект будет содержать все классы, которые будут использоваться главным проектом.

Данное приложение будет имитировать работу банка. И, прежде чем начать работу над приложением, выделим сущности, которые будем использовать, а также отношения между

сущностями. В частности, здесь мы можем выделить такие сущности, как банк, банковский счет. Счета могут быть различных видов, например, счета до востребования и депозиты, потому определим несколько сущностей счетов.

Для этого добавим в проект BankLibrary новый интерфейс, который будет описывать функциональность банковского счета. При проектировании интерфейса следует помнить, что он определяет общий функционал, который должен обязательно быть реализован в классах, применяющих данный интерфейс. Причем все члены этого интерфейса являются публичными или общедоступными. То есть если необходимо определить и использовать в классе свойства, методы, события, которые не должны иметь модификатор `public`, то интерфейс для определения подобных свойств и методов не подходит.

Итак, добавим в проект BankLibrary интерфейс `IAccount`, который будет иметь следующее содержание:

```
public interface IAccount
{
    // Положить деньги на счет
    void Put(decimal sum);
    // Взять со счета
    decimal Withdraw(decimal sum);
}
```

Данный интерфейс определяет два метода для того, чтобы положить на счет или вывести средства со счета.

Для реакции на изменения состояния счета мы будем использовать событийную модель, то есть обработку различных изменений счета через события. Для этого добавим в проект BankLibrary новый файл *AccountStateHandler.cs*, в котором определим делегат и вспомогательный класс:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace BankLibrary
{
    public delegate void AccountStateHandler(object sender, AccountEventArgs e);

    public class AccountEventArgs
    {
        // Сообщение
        public string Message { get; private set; }
        // Сумма, на которую изменился счет
        public decimal Sum { get; private set; }

        public AccountEventArgs(string _mes, decimal _sum)
        {
            Message = _mes;
            Sum = _sum;
        }
    }
}
```

Делегат `AccountStateHandler` будет использоваться для создания событий. А для обработки событий также определен класс `AccountEventArgs`, который определяет два свойства для чтения: сообщение о событии и сумма, на которую изменился счет.

Теперь определим основной класс приложения Account.

Добавим в проект BankLibrary новый класс Account, который будет иметь следующее определение:

```
public abstract class Account : IAccount
{
    //Событие, возникающее при выводе денег
    protected internal virtual event AccountStateHandler Withdrawed;
    // Событие возникающее при добавлении на счет
    protected internal virtual event AccountStateHandler Added;
    // Событие возникающее при открытии счета
    protected internal virtual event AccountStateHandler Opened;
    // Событие возникающее при закрытии счета
    protected internal virtual event AccountStateHandler Closed;
    // Событие возникающее при начислении процентов
    protected internal virtual event AccountStateHandler Calculated;

    protected int _id; // уникальный id счета
    static int counter = 0; // статический счетчик

    protected decimal _sum; // Переменная для хранения суммы
    protected int _percentage; // Переменная для хранения процента

    protected int _days = 0; // время с момента открытия счета

    public Account(decimal sum, int percentage)
    {
        _sum = sum;
        _percentage = percentage;
        _id=++counter; // увеличиваем счетчик и присваиваем его значение id
    }

    // Текущая сумма на счету
    public decimal CurrentSum
    {
        get { return _sum; }
    }
    // процент счета
    public int Percentage
    {
        get { return _percentage; }
    }
    // id
    public int Id
    {
        get { return _id; }
    }
    // метод, вызываемый после открытия света
    protected internal abstract void OnOpened();
    // метод добавления средств на счет
    public virtual void Put(decimal sum)
    {
        _sum += sum;
        if (Added != null) // вызываем событие добавления денег на счет
            Added(this, new AccountEventArgs("На счет поступило " + sum, sum));
    }
    // метод изъятия денег со счета
    public virtual decimal Withdraw(decimal sum)
    {
        decimal result = 0;
        if (sum <= _sum)
        {
            _sum -= sum;

```

```

        result = sum;
        if (Withdrawed != null)
            Withdrawed(this, new AccountEventArgs("Сумма " + sum + " снята со
счета " + _id, sum));
        }
        else
        {
            if (Withdrawed != null)
                Withdrawed(this, new AccountEventArgs("Недостаточно денег на
счете " + _id, sum));
        }
        return result;
    }
    // закрытие счета
    protected internal virtual void Close()
    {
        if (Closed != null)
            Closed(this, new AccountEventArgs("Счет " + _id + " закрыт. Итоговая
сумма: " + CurrentSum, CurrentSum));
    }
    // увеличиваем кол-во дней
    protected internal void IncrementDays()
    {
        _days++;
    }
    // метод подсчета процентов
    protected internal virtual void Calculate()
    {
        decimal increment = _sum * _percentage / 100;
        _sum = _sum + increment;
        if (Calculated != null)
            Calculated(this, new AccountEventArgs("Начислены проценты в размере:
" + increment, increment));
    }
}

```

Поскольку как такого банковского счета нет, а есть конкретные счета - депозит, до востребования и т.д., то данный класс является абстрактным. В то же время он реализует интерфейс `IAccount`.

Класс определяет ряд событий, которые вызываются при изменении состояния. События, как методы и свойства, также могут наследоваться и переопределяться, поэтому здесь они объявляются с модификатором `virtual`. Для определения событий применяется ранее созданный делегат `AccountStateHandler`.

Каждый счет имеет уникальный идентификатор `id` - некий уникальный номер. Для его получения применяется статический счетчик `counter`.

Почти все методы являются виртуальными, кроме метода `OnOpened`. Данный метод предназначен для генерации события создания счета, чтобы подписчики получили уведомление о том, что счет создан.

Также надо отметить, что большинство членов класса имеют модификатор `protected internal`, то есть они будут видны только внутри проекта `BankLibrary`.

Абстрактный класс `Account` определяет полиморфный интерфейс, который наследуется или переопределяется производными классами. Теперь добавим в проект новый класс, который будет представлять счет до востребования и который будет называться `DemandAccount`:

```

public class DemandAccount : Account
{

```

```

// переопределяем событие
protected internal override event AccountStateHandler Opened;

public DemandAccount(decimal sum, int percentage) : base(sum, percentage)
{
}

protected internal override void OnOpened()
{
    Opened(this, new AccountEventArgs("Открыт новый счет до востребования!
Id счета: " + this.Id, _sum));
}
}

```

В данном классе переопределяется событие `Opened` и метод `OnOpened`. При желании можно переопределить и больше функционала, но ограничимся в данном случае этим.

И также добавим второй класс-наследник `DepositAccount`:

```

public class DepositAccount : Account
{
    protected internal override event AccountStateHandler Added;
    protected internal override event AccountStateHandler Withdrawed;
    protected internal override event AccountStateHandler Opened;
    public DepositAccount(decimal sum, int percentage) : base(sum, percentage)
    {
    }
    protected internal override void OnOpened()
    {
        if (Opened != null)
            Opened(this, new AccountEventArgs("Открыт новый депозитный счет! Id
счета: " + this.Id, _sum));
    }

    public override void Put(decimal sum)
    {
        if (_days % 30 == 0)
            base.Put(sum);
        else if (Added != null)
            Added(this, new AccountEventArgs("На счет можно положить только
после 30-ти дневного периода", 0));
    }

    public override decimal Withdraw(decimal sum)
    {
        if (_days % 30 == 0)
            return base.Withdraw(sum);
        else if (Withdrawed != null)
            Withdrawed(this, new AccountEventArgs("Вывести средства можно только
после 30-ти дневного периода", 0));
        return 0;
    }

    protected internal override void Calculate()
    {
        if (_days % 30 == 0)
            base.Calculate();
    }
}

```

Депозитные счета имеют особенность: они оформляются на продолжительный период, что накладывает некоторые ограничения. Поэтому здесь переопределяются еще три метода. Допустим, что депозитный счет имеет срок в 30 дней, в пределах которого клиент не может

ни добавить на счет, ни вывести часть средств со счета, кроме закрытия всего счета. Поэтому при всех операциях проверяем количество прошедших дней для данного счета:

```
if (_days % 30 == 0)
```

В данном случае сравниваем остаток деления количества дней на 30 дней. Если остаток от деления равен 0, то значит прошел очередной 30-дневный период, по окончании которого происходит начисление процентов, возможен вывод средств или их добавление.

Собственно, это все классы банковских счетов, которые мы будем использовать в программе.

Как правило, банковские счета существуют не сами по себе, а внутри банка, который выступает некоторым контейнером счетов и выполняет функции по управлению ими. Поэтому добавим в проект BankLibrary новый класс Bank:

```
public class Bank<T> where T : Account
{
    T[] accounts;

    public string Name { get; private set; }

    public Bank(string name)
    {
        this.Name = name;
    }
    // метод создания счета
    public void Open(AccountType accountType, decimal sum,
        AccountStateHandler addSumHandler, AccountStateHandler
withdrawSumHandler,
        AccountStateHandler calculationHandler, AccountStateHandler
closeAccountHandler,
        AccountStateHandler openAccountHandler)
    {
        T newAccount = null;

        switch (accountType)
        {
            case AccountType.Ordinary:
                newAccount = new DemandAccount(sum, 1) as T;
                break;
            case AccountType.Deposit:
                newAccount = new DepositAccount(sum, 40) as T;
                break;
        }

        if (newAccount == null)
            throw new Exception("Ошибка создания счета");
        // добавляем новый счет в массив счетов
        if (accounts == null)
            accounts = new T[] { newAccount };
        else
        {
            T[] tempAccounts = new T[accounts.Length + 1];
            for (int i = 0; i < accounts.Length; i++)
                tempAccounts[i] = accounts[i];
            tempAccounts[tempAccounts.Length - 1] = newAccount;
            accounts = tempAccounts;
        }
        // установка обработчиков событий счета
        newAccount.Added += addSumHandler;
        newAccount.Withdrawed += withdrawSumHandler;
        newAccount.Closed += closeAccountHandler;
    }
}
```

```

        newAccount.Opened += openAccountHandler;
        newAccount.Calculated += calculationHandler;

        newAccount.OnOpened();
    }
    //добавление средств на счет
    public void Put(decimal sum, int id)
    {
        T account = FindAccount(id);
        if (account == null)
            throw new Exception("Счет не найден");
        account.Put(sum);
    }
    // вывод средств
    public void Withdraw(decimal sum, int id)
    {
        T account = FindAccount(id);
        if (account == null)
            throw new Exception("Счет не найден");
        account.Withdraw(sum);
    }
    // закрытие счета
    public void Close(int id)
    {
        int index;
        T account = FindAccount(id, out index);
        if (account == null)
            throw new Exception("Счет не найден");

        account.Close();

        if (accounts.Length <= 1)
            accounts = null;
        else
        {
            // уменьшаем массив счетов, удаляя из него закрытый счет
            T[] tempAccounts = new T[accounts.Length - 1];
            for (int i = 0; i < accounts.Length; i++)
            {
                if (i == index)
                    continue;
                tempAccounts[i] = accounts[i];
            }
            accounts = tempAccounts;
        }
    }

    // начисление процентов по счетам
    public void CalculatePercentage()
    {
        if (accounts == null) // если массив не создан, выходим из метода
            return;
        for (int i = 0; i < accounts.Length; i++)
        {
            T account = accounts[i];
            account.IncrementDays();
            account.Calculate();
        }
    }

    // поиск счета по id
    public T FindAccount(int id)
    {
        for (int i = 0; i < accounts.Length; i++)
        {

```



```

        if (accounts[i].Id == id)
            return accounts[i];
    }
    return null;
}
// перегруженная версия поиска счета
public T FindAccount(int id, out int index)
{
    for (int i = 0; i < accounts.Length; i++)
    {
        if (accounts[i].Id == id)
        {
            index = i;
            return accounts[i];
        }
    }
    index = -1;
    return null;
}
}
// тип счета
public enum AccountType
{
    Ordinary,
    Deposit
}

```

Класс банка является обобщенным. При этом параметр `T` имеет ограничение: он обязательно должен представлять класс `Account` или его наследников. Поэтому у любого объекта `T` нам будут доступны методы и свойства класса `Account`.

Все счета в классе хранятся в массиве `accounts`. На момент проектирования класса мы можем не знать, какими именно счетами будет управлять банк. Возможно, это будут любые счета, а может быть только депозитные, то есть объекты `DepositAccount`. Поэтому использование обобщений позволяет добавить больше гибкости.

При создании нового счета в методе `Open` в этот метод передается ряд параметров, в частности, тип счета, который описывается перечислением:

```

public enum AccountType
{
    Ordinary,
    Deposit
}

```

Данное перечисление можно определить после класса `Bank`. В зависимости от типа счета создается объект `DemandAccount` или `DepositAccount` и затем добавляется в массив `accounts`. Поскольку массивы не расширяются автоматически, то фактически мы создаем новый массив с увеличением элементов на единицу и в конец нового массива добавляем новый элемент.

При этом параметризация, то есть создание обобщенных классов имеет ограничение в том, что созданный объект еще необходимо привести к типу `T`:

```
newAccount = new DemandAccount(sum, 1) as T;
```

Такое приведение позволит нам избежать ошибок, например, если мы типизируем класс `Bank` не `Account`, а типом `DepositAccount`, то преобразование

`newAccount = DemandAccount(sum, 1) as T` вернет нам значение `null`. Далее мы можем проверить полученное значение на `null`:

```
if (newAccount == null)
```

Также в метод `Open` передаются обработчики для всех событий класса `Account`, которые устанавливаются после создания объекта `Account`.

В конце вызывается у нового объекта `Account` метод `OnOpened()`, который генерирует событие `Account.Opened`, благодаря чему извне можно получить уведомление о событии.

Для поиска счета в массиве по `id` определяется метод `FindAccount()`. Его перегруженная версия позволяет также получать индекс найденного элемента через выходной параметр `index`:

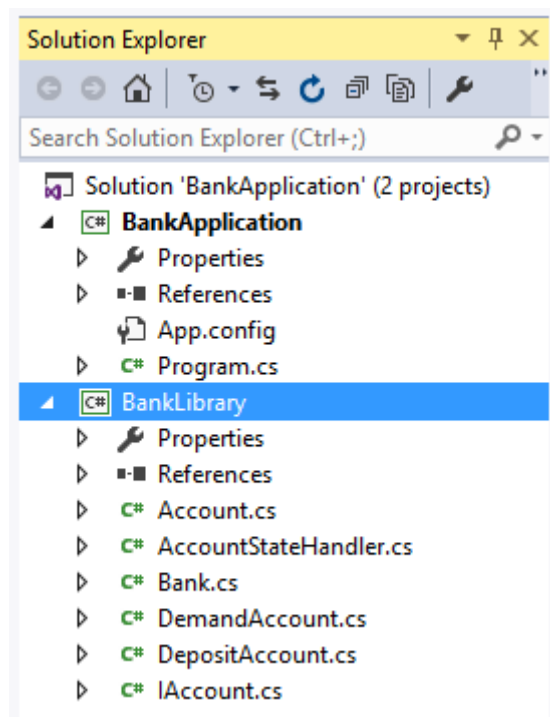
```
public T FindAccount(int id, out int index)
```

В методах `Put`, `Withdraw` и `Close` используются метод `FindAccount()` для получения счета для добавления или вывода средств, а также закрытия. При закрытии счета в методе `Close()` создается новый массив без одного элемента - счета, который надо удалить. Таким образом, происходит удаление счета.

В методе `CalculatePercentage()` проходим по всем элементам массива счетов, увеличиваем у каждого счета счетчик дней и производим начисление процентов.

В общем класс `Bank` является оберткой, через которую из главного проекта мы будем взаимодействовать с объектами `Account`.

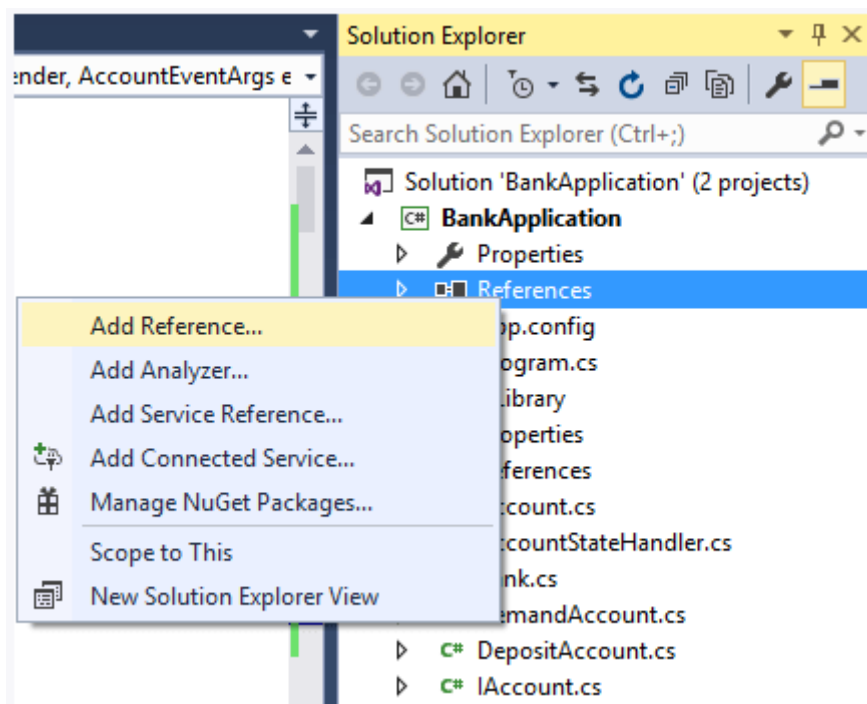
В итоге проект `BankLibrary` должен выглядеть следующим образом:



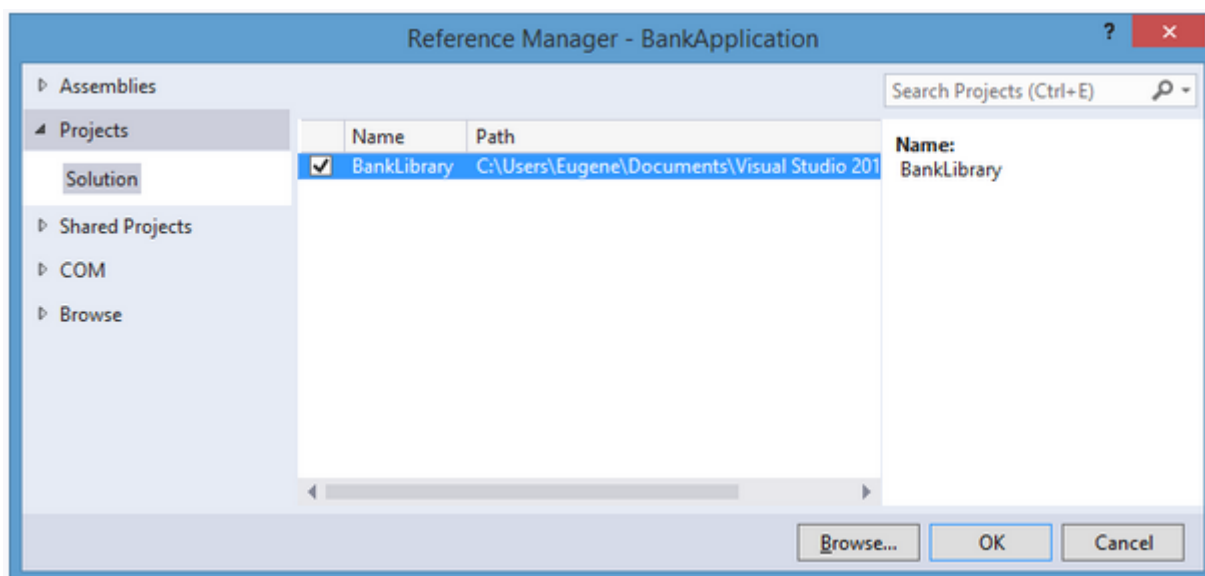
И теперь построим проект. Для этого нажмем на название проекта в окне `Solution Explorer` (Обозреватель решений) правой кнопкой мыши и в появившемся контекстном меню выберем пункт `Build`. После этого в проекте в папке `bin/Debug` будет создан файл библиотеки классов с расширением `dll`.

## Создание проекта интерфейса

Вначале подключим скомпилированную в прошлой теме библиотеку классов. Для этого в главном проекте BankApplication нажмем на пункт References правой кнопкой мыши и в появившемся меню выберем пункт Add Reference...:



Затем в появившемся окне отметим пункт BankLibrary, который будет представлять нашу библиотеку классов, и нажмем на ОК.



После этого в проект будет добавлена ссылка на библиотеку. И если мы раскроем узел Reference, то сможем увидеть ее среди подключенных библиотек.

Теперь изменим файл Program.cs в главном проекте следующим образом:

```
using System;
using BankLibrary;

namespace BankApplication
{
    class Program
```

```

{
    static void Main(string[] args)
    {
        Bank<Account> bank = new Bank<Account>("ЮнитБанк");
        bool alive = true;
        while (alive)
        {
            ConsoleColor color = Console.ForegroundColor;
            Console.ForegroundColor = ConsoleColor.DarkGreen; // выводим
            список команд зеленым цветом
            Console.WriteLine("1. Открыть счет \t 2. Вывести средства \t 3.
Добавить на счет");
            Console.WriteLine("4. Закрыть счет \t 5. Пропустить день \t 6.
Выйти из программы");
            Console.WriteLine("Введите номер пункта:");
            Console.ForegroundColor = color;
            try
            {
                int command = Convert.ToInt32(Console.ReadLine());

                switch (command)
                {
                    case 1:
                        OpenAccount(bank);
                        break;
                    case 2:
                        Withdraw(bank);
                        break;
                    case 3:
                        Put(bank);
                        break;
                    case 4:
                        CloseAccount(bank);
                        break;
                    case 5:
                        break;
                    case 6:
                        alive = false;
                        continue;
                }
                bank.CalculatePercentage();
            }
            catch (Exception ex)
            {
                // выводим сообщение об ошибке красным цветом
                color = Console.ForegroundColor;
                Console.ForegroundColor = ConsoleColor.Red;
                Console.WriteLine(ex.Message);
                Console.ForegroundColor = color;
            }
        }
    }

    private static void OpenAccount(Bank<Account> bank)
    {
        Console.WriteLine("Укажите сумму для создания счета:");

        decimal sum = Convert.ToDecimal(Console.ReadLine());
        Console.WriteLine("Выберите тип счета: 1. До востребования 2.
Депозит");
        AccountType accountType;

        int type = Convert.ToInt32(Console.ReadLine());

        if (type == 2)

```

```

        accountType = AccountType.Deposit;
    else
        accountType = AccountType.Ordinary;

    bank.Open(accountType,
        sum,
        AddSumHandler, // обработчик добавления средств на счет
        WithdrawSumHandler, // обработчик вывода средств
        (o, e) => Console.WriteLine(e.Message), // обработчик начислений
        CloseAccountHandler, // обработчик закрытия счета
        OpenAccountHandler); // обработчик открытия счета
    }

    private static void Withdraw(Bank<Account> bank)
    {
        Console.WriteLine("Укажите сумму для вывода со счета:");

        decimal sum = Convert.ToDecimal(Console.ReadLine());
        Console.WriteLine("Введите id счета:");
        int id = Convert.ToInt32(Console.ReadLine());

        bank.Withdraw(sum, id);
    }

    private static void Put(Bank<Account> bank)
    {
        Console.WriteLine("Укажите сумму, чтобы положить на счет:");
        decimal sum = Convert.ToDecimal(Console.ReadLine());
        Console.WriteLine("Введите Id счета:");
        int id = Convert.ToInt32(Console.ReadLine());
        bank.Put(sum, id);
    }

    private static void CloseAccount(Bank<Account> bank)
    {
        Console.WriteLine("Введите id счета, который надо закрыть:");
        int id = Convert.ToInt32(Console.ReadLine());

        bank.Close(id);
    }
    // обработчики событий класса Account
    // обработчик открытия счета
    private static void OpenAccountHandler(object sender, AccountEventArgs e)
    {
        Console.WriteLine(e.Message);
    }
    // обработчик добавления денег на счет
    private static void AddSumHandler(object sender, AccountEventArgs e)
    {
        Console.WriteLine(e.Message);
    }
    // обработчик вывода средств
    private static void WithdrawSumHandler(object sender, AccountEventArgs e)
    {
        Console.WriteLine(e.Message);
        if (e.Sum > 0)
            Console.WriteLine("Идем тратить деньги");
    }
    // обработчик закрытия счета
    private static void CloseAccountHandler(object sender, AccountEventArgs e)
    {
        Console.WriteLine(e.Message);
    }
}

```

```
}
```

В начале файла подключается библиотека:

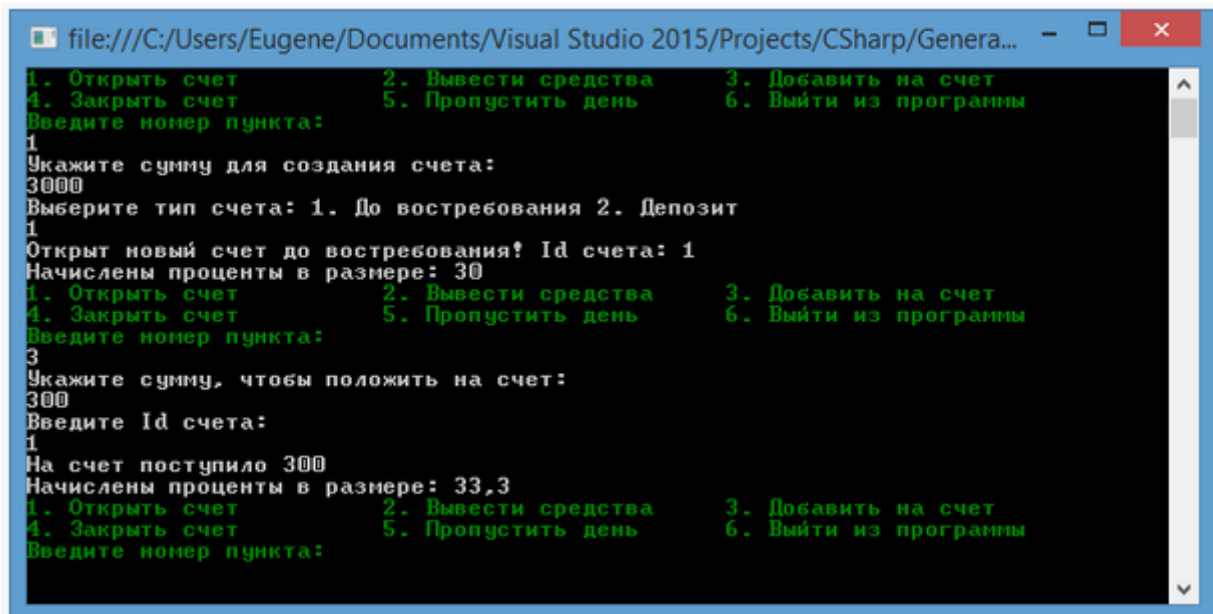
```
using BankLibrary;
```

В методе Main создается объект Bank, который типизирован классом Account и через который мы будем взаимодействовать с объектами Account.

В цикле while выводится список команд, который должен выбрать пользователь. После выбора одной из них в конструкции switch выполняется соответствующая команда. Каждая команда представляет получения ввода от пользователя, его преобразование с помощью класса Convert и передача аргументов методам объекта Bank.

Каждая итерация цикла while соответствует одному дню, поэтому в конце цикла вызывается метод `bank.CalculatePercentage()`, который увеличивает у объектов Account счетчик дней и производит начисление процентов.

В итоге получится следующая программа, имитирующая работу банка и взаимодействие с пользователем:

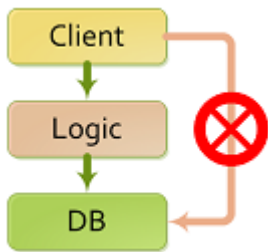


```
file:///C:/Users/Eugene/Documents/Visual Studio 2015/Projects/CSharp/Genera...
1. Открыть счет          2. Вывести средства      3. Добавить на счет
4. Закрыть счет          5. Пропустить день      6. Выйти из программы
Введите номер пункта:
1
Укажите сумму для создания счета:
3000
Выберите тип счета: 1. До востребования 2. Депозит
1
Открыт новый счет до востребования! Id счета: 1
Начислены проценты в размере: 30
1. Открыть счет          2. Вывести средства      3. Добавить на счет
4. Закрыть счет          5. Пропустить день      6. Выйти из программы
Введите номер пункта:
3
Укажите сумму, чтобы положить на счет:
300
Введите Id счета:
1
На счет поступило 300
Начислены проценты в размере: 33,3
1. Открыть счет          2. Вывести средства      3. Добавить на счет
4. Закрыть счет          5. Пропустить день      6. Выйти из программы
Введите номер пункта:
```

## Проверка приложения на соответствие архитектуре слоев

Далее необходимо построить архитектуру приложения и проверить написанный код на соответствие архитектуре. Рассмотрим решение данной задачи.

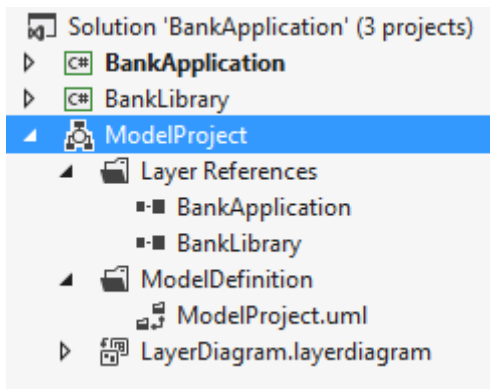
В профессиональной разработке ПО используется архитектурный шаблон слоев\уровней программной системы, позволяющий эффективно прятать реализацию и абстрагировать компоненты разного уровня, делая их слабосвязанными и легко заменяемыми. Слои нижнего уровня могут изменяться без особого риска нарушить работу верхних уровней приложения, облегчен рефакторинг. Единственное очевидное условие, которое вы должны соблюдать – это придерживаться принятой архитектуры. Но иногда бывает, что программист соблазняется вызвать пару методов «через голову». Например, из слоя интерфейса обратиться напрямую в слой базы данных.



Вылавливать такие случаи несоответствия кода архитектуре слоев на большой системе может быть очень затруднительно. К счастью в Visual Studio есть инструменты, которые могут значительно облегчить эту задачу.

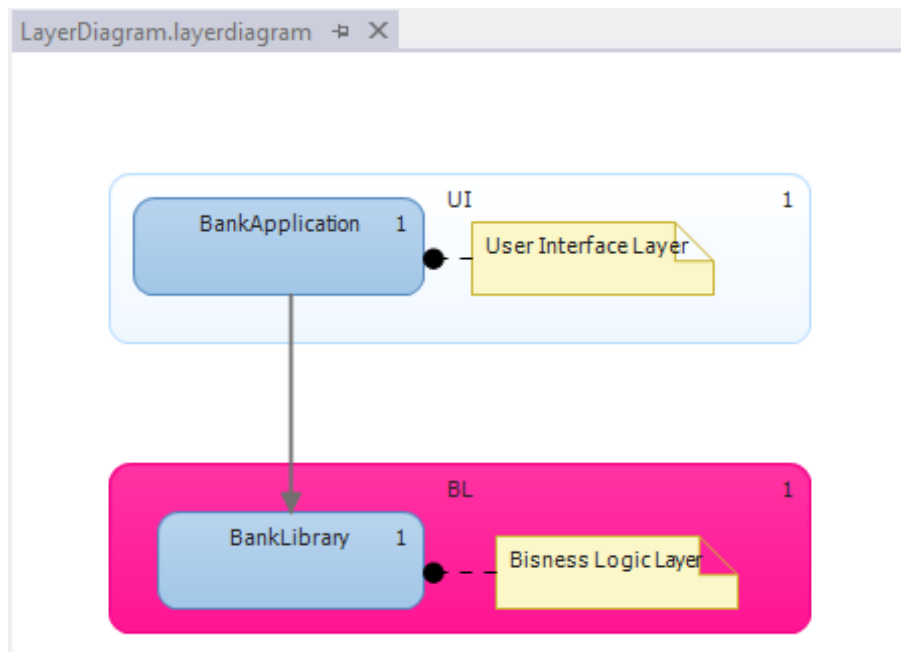
В Visual Studio есть компонент, который называется Layered Diagrams – диаграммы слоев. Полезное в нем то, что впоследствии можно сопоставить сборки проекта конкретным слоям, и в автоматическом режиме проверить, нет ли среди кода приложения вызовов которые противоречат принятой архитектуре.

Внутри команды всем очевиден порядок вызовов, и что не следует из интерфейса обращаться напрямую к слою БД. Когда в проекте лишь три-четыре сборки – это, возможно, не будет проблемой. Но когда их количество значительно увеличивается, отслеживать вручную, кто кого и из какого слоя может вызывать, станет затруднительно. Поэтому задокументируем нашу схему, добавив в Solution проект с типом Modelling (File/Add New Project/Modelling Projects/Modelling Project):

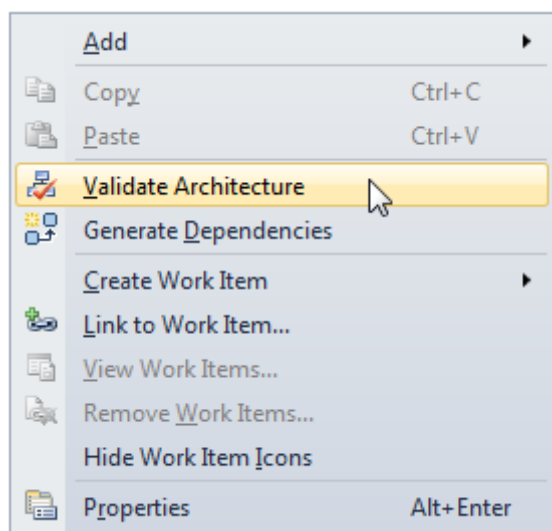


Далее добавляем диаграмму слоев (Add New Item/Layer Diagram) и рисуем нашу архитектуру слоев. Затем создаем сопоставление слоев сборкам. После того, как созданы основные компоненты системы в виде сборок и нарисована диаграмма слоев, необходимо сопоставить их между собой, просто перетащив из Solution Explorer узлы проектов на прямоугольники слоев.

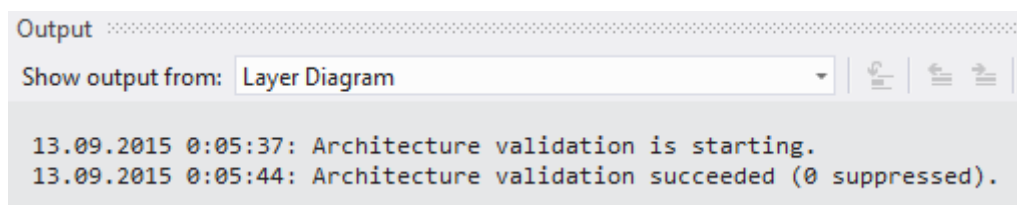
При этом в правом верхнем углу будет отображаться количество сборок, сопоставленных с этим слоем, а в Modelling проекте появятся ссылки на компоненты решения



Теперь все готово для автоматической проверки архитектуры, достаточно вызвать контекстное меню на диаграмме слоев:

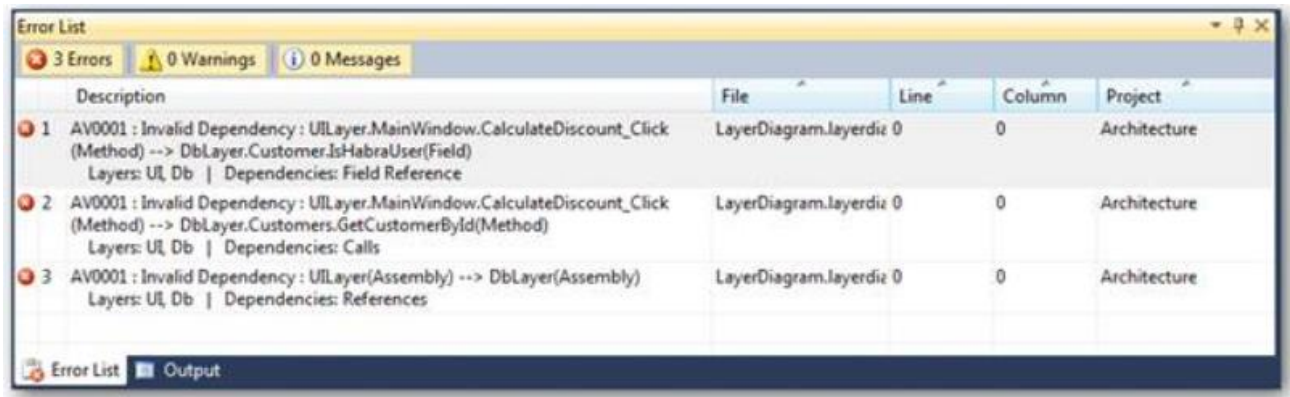


После анализа, который работает через Reflection, получим список несоответствий нашего кода архитектуре. В нашем случае результат проверки говорит о том, что код нашего решения полностью соответствует принятой архитектуре слоев:



В случае несоответствия мы получили бы отчет об ошибках:





## Ресурсы:

1. Объектно-ориентированное программирование. Практика  
<http://metanit.com/sharp/tutorial/3.29.php>
2. Как проверить приложение на соответствие архитектуре слоев  
<http://habrahabr.ru/company/microsoft/blog/129073/>