# Лабораторная работа №4: работа с файлами, разделение приложения на уровни

## Цели

1. Отработать на практике работу с файлами широко распространённых текстовых форматов CSV и XML.
2. Научиться создавать приложение с простейшей типовой архитектурой разделения на уровни.

## Задание

Написать консольную программу, реализующую типичную финансовую задачу: разнесения оплат по счетам. Исходные данные предоставляются в виде файлов XML и CSV, в результате работы программы должен быть сформирован XML-файл.

### Входные данные

1. Файл платежей формата CSV со следующими полями (поля разделены “;”, предполагается, что в полях символа “;” нет):
   1. Название клиента (строка);
   2. Дата платежа (в формате dd.MM.yyyy);
   3. Номер платежа (строка);
   4. Сумма платежа (число с плавающей точкой, разделитель дробной части «.»).

Первая строка файла содержит заголовок.

1. Файл счетов формата XML со следующей структурой: набор документов (элемент Bills), где каждый элемент (Bill) содержит атрибуты:
   1. Название клиента (строка);
   2. Дата счёта (в формате dd.MM.yyyy);
   3. Номер счёта (строка);
   4. Сумма счёта (число с плавающей точкой, разделитель дробной части «.»).

### Выходные данные

Файл формата XML со списком оплат (Payments), где каждый элемент (Payment) представляет собой запись о платеже со следующими атрибутами:

* 1. Название клиента (строка);
  2. Дата платежа (в формате dd.MM.yyyy);
  3. Номер платежа (строка);
  4. Дата счёта (в формате dd.MM.yyyy);
  5. Номер счёта (строка);

1. Сумма платежа (число с плавающей точкой, разделитель дробной части «.»).

Пример файлов:





### Требования по формированию оплаты

Программа должна ставить в соответствие каждому счёту клиента один или более платёжный документ (относящийся к тому же клиенту) таким образом, чтобы сумма счёта максимально покрывалась платёжными документами. Каждый счёт может быть оплачен более чем одним платежом, каждый платёж может быть использован для оплаты более чем одного счёта. И платежи, и счёта должны быть упорядочены хронологически. Даты платежа и счёта при оплате сравнивать не надо (допускается, что платёж пришёл раньше счёта и наоборот).

Программа должна проверять корректность входных данных (заполнение всех полей и соответствие их значений формату) и выдавать сообщение при несоответствии (достаточно выбросить исключение). В случае любой ошибки в формате исходных данных программа должна быть остановлена.

#### Пример расчёта оплаты

Счета:

* Счёт от 05.05.2016 на сумму 100
* Счёт от 01.05.2016 на сумму 200

Платежи:

* Платёж от 01.08.2016 на сумму 150
* Платёж от 07.08.2016 на сумму 70

Результирующая оплата:

* Счёт от 01.05.2016, Платёж от 01.08.2016, сумма 150
* Счёт от 01.05.2016, Платёж от 07.08.2016, сумма 50
* Счёт от 05.05.2016, Платёж от 07.08.2016, сумма 20

### Требования по структуре программы

В программе должны быть выделены два уровня (в виде соответствующих классов и пространств имён):

* Уровень доступа к данным. Уровень должен выполнять все операции по чтению и записи файлов, но ничего кроме этого.
* Уровень бизнес-логики. Уровень должен реализовывать алгоритм расчёта оплаты, но не заниматься никаким вводом/выводом.

Должны быть реализованы классы (так называемые «модели»), отражающие структуру сущностей программы (счёт, платёж, оплата). Уровень доступа к данным должен читать файлы и возвращать результат в виде списка моделей, и записывать файл, получив на вход список моделей оплаты. Уровень бизнес-логики также должен работать с моделями.

**Пример корректной структуры классов**

Модели:

class Bill { … }

class PayDoc { … }

class Payment { … }

Уровень доступа к данным:

class PaymentRepository {

public IEnumerable<Bill> GetBills ( ) { … }

public IEnumerable<PayDoc > GetPayDocs ( ) { … }

public void SetPayDocs ( IEnumerable<Payment payments> ) { … }

}

Уровень бизнес-логики:

class PaymentLogic {

public void CalculatePayments ( ) { … }

}

Выполнение перечисленных выше требований оценивается как «хорошо». Для получения оценки «отлично» программа должна занимать менее 200 строк (суммарно по всем файлам). Считаются только строки с кодом, но не с комментариями или пустые. В каждой строке должно быть не более одного оператора, запрещены длинные строки кода (более 100 символов). Вычислять размер кода можно встроенным средством Visual Studio.

## Демонстрация работы

1. Запустить программу с тестовыми файлами;
2. Продемонстрировать файл, получившийся в результате;
3. Изменить данные в обоих тестовых файлах;
4. Повторно запустить программу;
5. Продемонстрировать результирующий файл;
6. Внести ошибку в один из файлов (например, изменив формат даты или указан неправильную дату);
7. Запустить программу;
8. Программа должна остановиться с выдачей ошибки.

## Вспомогательные материалы

### Форматы файлов

#### CSV

CSV представляет собой текстовый файл, содержащий таблицу – одна строка в файле представляет собой одну строку таблицы. Поля отделены друг от друга символом-разделителем (обычно это “;”, “,” или табуляция), строки отделены друг от друга перевода строки. Символ-разделитель, встречающийся в поле, каким-либо образом экранируется, чтобы не смешиваться с символом-разделителем полей. Первая строка таблицы может содержать заголовок, а может и не содержать . Для чтения CSV существуют специальные библиотеки, но в простых случаях можно ограничиться самостоятельным чтением и разбором файла. Пример файла:

Имя;Фамилия;День рождения

Иван;Иванов;01.05.1995

Петров;Сергей;01.05.2000

#### XML

XML представляет собой текстовый файл, содержащий дерево узлов. Типы узлов: элемент, атрибут, текст и некоторые другие. Формат строго формализован, полностью описан в спецификации. На текущий момент является одним из самых распространённых текстовых форматов общего назначения. Для чтения и записи есть множество библиотек и классов, в том числе входящих в состав .NET. Пример файла:

<?xml version="1.0" encoding="utf-8"?>

<Persons>

<Person Name="Петр" LastName="Петров" Birthday="01.05.1995">

<Hobby>

<Cars Priority="1"/>

<Girls Priority="2"/>

<Math Priority="3"/>

</Hobby>

</Person>

</Persons>

### Разделение на уровни

Разделение программы на уровни, реализующие чётко определённый функционал, призвано снизить сложность, упростить изменение и повысить качество программного кода. Это достигается за счёт изоляции уровней друг от друга – каждый уровень делает только свою часть работы, и никакую другую. При необходимости, функционал любого уровня может быть изменён, без изменения других уровней. Типовой пример разделения программы на уровни:



### Полезные классы и методы .NET

1. Построчное чтение текстового файла: System.IO.StreamReader
2. Чтение и запись XML, классический способ: System.Xml.XmlDocument. Пример:

var doc = new XmlDocument ( );

doc.Load ( @"C:\Счета.xml" );

foreach ( XmlElement e in doc.SelectNodes ( "Bills/Bill" ) )

Console.WriteLine ( e.Attributes["Client"].Value );

XmlDocument doc = new XmlDocument ( );

XmlElement root = doc.CreateElement ( "Payments" );

XmlElement child = doc.CreateElement ( "Payment" );

XmlAttribute attr = doc.CreateAttribute ( "Client" );

attr.Value = "Василий";

child.Attributes.Append ( attr );

root.AppendChild ( child );

doc.AppendChild ( root );

doc.Save ( @"c:\Payments.xml" );

1. Чтение и запись XML, другой вариант: System.Xml.Linq.XDocument. Пример:

var doc = XDocument.Load ( @"C:\Счета.xml" );

foreach ( XElement e in doc.Root.Nodes ( ) )

Console.WriteLine ( e.Attribute ( "Client" ).Value );

XDocument doc = new XDocument (

new XElement ( "Payments",

new XElement ( "Payment",

new XAttribute ( "Client", "Василий" ) ) ) );

doc.Save ( @"c:\Payments.xml" );