Оглавление

1. Общая характеристика платформы .NET.

В середине 2000 года корпорация Microsoft объявила о работе над новой платформой для создания приложений, которая получила имя *платформа .NET* (*.NET Framework*). Платформа .NET образует каркас, включающий библиотеку классов и технологии разработки Windows-приложений, web-приложений и сервисов, доступа к данным и межпрограммного взаимодействия. Основным инструментом для разработки является интегрированная среда Microsoft Visual Studio.

Базой платформы .NET является *общеязыковая среда исполнения* (*Common Language Runtime*, *CLR*). CLR является «прослойкой» между операционной системой и приложением для .NET Framework. Приложения для платформы .NET состоят из *управляемого кода* (*managed code*). Управляемый код является результатом компиляции исходных текстов. Скомпилированные файлы называются *сборками* (*assembly*) и включают следующие части:

1. *Манифест* (*manifest*) – описание сборки: версия, ограничения безопасности, список необходимых внешних сборок.

2. *Метаданные* – специальное описание всех пользовательских типов данных, размещенных в сборке.

3. *Код на промежуточном языке Microsoft Intermediate Language* (*MSIL* или просто *IL*). Данный код является независимым от операционной системы и типа процессора. В процессе работы приложения он компилируется в машинно-зависимый код специальным компилятором (*Just-in-Time (JIT) compiler*).

Основная задача CLR – это манипулирование сборками: загрузка, JIT-компиляция, создание окружения для выполнения сборок. Важной функцией CLR является размещение памяти при работе приложения и выполнение *автоматической сборки мусора*, то есть фонового освобождения неиспользуемой памяти. Кроме этого, CLR реализует в приложениях для .NET проверку типов, управление политиками безопасности при доступе к коду и некоторые другие функции.

В состав платформы .NET входит обширная библиотека классов *Framework Class Library* (*FCL*). Элементом этой библиотеки является базовый набор классов *Base Class Library* (*BCL*). В BCL входят классы для работы со строками, коллекциями данных, поддержки многопоточности и множество других классов. Частью FCL являются компоненты, поддерживающие

различные технологии обработки данных и организации взаимодействия с пользователем. Это классы для работы с XML, базами данных, создания пользовательских интерфейсов.

В стандартную поставку платформы .NET включено несколько компиляторов. Это компиляторы языков C#, Visual Basic.NET, C++/CLI. Благодаря открытым спецификациям компиляторы для .NET предлагаются различными сторонними производителями. Хотелось бы подчеркнуть, что любой язык для платформы .NET является верхним элементом архитектуры. Имена элементов библиотеки FCL не зависят от языка программирования. Специфичной частью языка остается только синтаксис. Это упрощает межъязыковое взаимодействие, перевод текста программы с одного языка на другой. С другой стороны, в синтаксических элементах любого языка программирования для .NET неизбежно находит свое отражение тесная связь с CLR.

Выделим ещё две части платформы .NET:

- *Система типов данных* (*Common Type System*, *CTS*) – базовые, не зависящие от языка программирования примитивные типы, которыми может манипулировать CLR.
- *Набор правил* для языка программирования (*Common Language Specification*, *CLS*), соблюдение которых обеспечивает создание на разных языках программ, легко взаимодействующих между собой.

В заключение рассмотрим историю версий платформы .NET. Первая официальная версия вышла в феврале 2002 года. В апреле 2003 года была опубликована версия 1.1, содержащая небольшие улучшения. Версия 2.0, вышедшая в ноябре 2005 года, представила обновленную CLR с поддержкой универсальных шаблонов (generics). В синтаксис языков C# и VB.NET также были внесены существенные изменения. Были переработаны и улучшены технологии ASP.NET и ADO.NET. Следующим шагом явился выпуск в ноябре 2006 года версии 3.0, которая содержала набор новых технологий для построения приложений и межпрограммного взаимодействия (Windows Presentation Foundation, Windows Communication Foundation, Workflow Foundation). В ноябре 2007 года вышла версия 3.5, основными особенностями которой являются реализация технологии LINQ и новые версии компиляторов для C# и VB.NET (в августе 2008 года опубликован пакет обновлений для данной версии).

Версия	**Дата выхода**

1.0	1 мая 2002 года
1.1	1 апреля 2003 года
2.0	11 июля 2005 года
3.0	6 ноября 2006 года
3.5	9 ноября 2007 года
4.0	12 апреля 2010 года
4.5	15 августа 2012 года
4.5.1	17 октября 2013 года

2. Классификация типов платформы .NET. и языка C#.

Основой CLR является развитая система типов. Типы языка C# соответствуют определенным типам CLR. Можно сказать, что имя типа из C# - это псевдоним типа из CLR (например, тип int в C# - псевдоним типа System.Int32).

Для дальнейшего изложения систему типов удобно классифицировать. С точки зрения размещения переменных в памяти все типы можно разделить на *ссылочные типы* и *типы значений*. Переменная ссылочного типа, далее называемая *объектом*, содержит ссылку на данные, которые размещены в управляемой динамической памяти. Ссылочные типы – это *класс*, *интерфейс*, *строка*, *массив*, *делегат* и *тип* object. Переменная типа значения содержит непосредственно данные и размещается в стеке. К типам значений относятся *структуры* и *перечисления*. Структуры, в свою очередь, делятся на *числовые типы*, *тип* bool и *пользовательские структуры*.

Числовые типы делятся на *целочисленные типы*, *типы с плавающей запятой* и *тип* decimal. Информация о числовых типах представлена в табл.

Сопоставление типовых псевдонимов C# и типов CLR

Тип C#	Имя типа в CLR	Примечание
sbyte	System.SByte	Знаковые целочисленные типы
short	System.Int16	
int	System.Int32	
long	System.Int64	
byte	System.Byte	Беззнаковые целочисленные типы
ushort	System.UInt16	
uint	System.UInt32	
ulong	System.UInt64	
char	System.Char	

`float`	`System.Single`	Типы с плавающей запятой
`double`	`System.Double`	
`decimal`	`System.Decimal`	Тип данных повышенной точности
`bool`	`System.Boolean`	Тип для хранения логических значений
`T?`	`System.Nullable<T>`	Тип значений `T` с поддержкой `null` (например, `int?`)
`string`	`System.String`	Тип для представления строк
`object`	`System.Object`	Базовый тип
`dynamic`	`System.Object`	Динамический тип с проверкой элементов при выполнении программы

Категория	**Тип C#**	**Размер (бит)**	**Диапазон и точность**
Знаковые целочисленные типы	`sbyte`	8	-128..127
	`short`	16	-32 768..32 767
	`int`	32	-2 147 483 648..2 147 483 647
	`long`	64	-9 223 372 036 854 775 808.. 9 223 372 036 854 775 807
Беззнаковые целочисленные типы	`byte`	8	0..255
	`ushort`	16	0..65535
	`char`	16	Символ в кодировке UTF-16
	`uint`	32	0..4 294 967 295
	`ulong`	64	0..18 446 744 073 709 551 615
Типы с плавающей запятой	`float`	32	От $\pm1.5\times10^{-45}$ до $\pm3.4\times10^{38}$, точность 7 цифр
	`double`	64	От $\pm5.0\times10^{-324}$ до $\pm1.7\times10^{308}$, точность 15 цифр
Тип `decimal`	`decimal`	128	От $\pm1.0\times10^{-28}$ до $\pm7.9\times10^{28}$, точность 28 цифр

Отметим, что типы sbyte, ushort, uint, ulong не соответствуют Common Language Specification. Это означает, что данные типы не следует использовать в интерфейсах межъязыкового взаимодействия. Тип char, хотя формально и относится к целочисленным, представляет символ в 16-битной Unicode-кодировке. Тип decimal удобен для проведения финансовых вычислений.

Тип bool (System.Boolean) служит для представления булевых значений. Переменные данного типа могут принимать значения true или false.

Из ссылочных типов упомянем тип string – это последовательность Unicode-символов. Также существует ссылочный тип object (System.Object), переменной которого можно присвоить любое значение.

Другой подход к классификации типов предполагает деление на *примитивные* и *пользовательские типы*. Примитивные типы встроены в CLR.

Все числовые типы, а также типы bool, string и object принято относить к примитивным типам. Пользовательские типы должны быть описаны программистом при помощи особых синтаксических конструкций. Можно утверждать, что любая программа на языке C# представляет собой набор определенных пользовательских типов. Опишем функциональность, которой обладают пользовательские типы.

1. Класс – тип, поддерживающий всю функциональность объектно-ориентированного программирования, включая наследование и полиморфизм.

2. Структура – тип значения, обеспечивающий инкапсуляцию данных, но не поддерживающий наследование. Синтаксически, структура похожа на класс.

3. Интерфейс – абстрактный тип, реализуемый классами и структурами для обеспечения оговоренной функциональности.

4. Массив – пользовательский тип для представления упорядоченного набора значений некоторых других типов.

5. Перечисление – тип, содержащий в качестве членов именованные целочисленные константы.

6. Делегат – пользовательский тип, инкапсулирующий метод.

Тип char преобразуется в типы sbyte, short, byte явно, а в остальные числовые типы – неявно. Преобразование числового типа в тип char может быть выполнено только в явной форме.

Для более гибкого контроля значений, получаемых при работе с числовыми выражениями, в языке C# предусмотрено использование контролируемого и неконтролируемого контекстов. *Контролируемый контекст* объявляется в форме checked <программный блок>, либо как операция checked(<выражение>). Если при преобразовании типов в контролируемом контексте получается значение, выходящие за пределы целевого типа, то генерируется, либо ошибка компиляции (для константных выражений), либо обрабатываемое исключение (для выражений с переменными).

Неконтролируемый контекст объявляется в форме unchecked <программный блок>, либо как операция unchecked(<выражение>). При использовании неконтролируемого контекста выход за пределы целевого типа ведет к автоматическому «урезанию» результата либо путем отбрасывания бит (целые типы), либо путем округления (вещественные типы).

Массив является ссылочным типом, поэтому перед началом работы любой

массив должен быть создан в памяти. Для этого используется конструктор в форме new <тип>[<количество элементов>].

int[] data; data = new int[10];

Создание массива можно совместить с его объявлением: int[] data = new int[10];

Созданный массив автоматически заполняется значениями по умолчанию для базового типа (ссылочные типы – null, числа – 0, тип bool – false).

Для доступа к элементу массива указывается имя массива и индекс в квад

В языке C# существует способ задания элементов массива при создании. Для этого используется список значений в фигурных скобках. При этом можно не указывать количество элементов, а также полностью опустить указание на тип и ключевое слово new:

```
int[] data_1 = new int[4] { 1, 2, 3, 5 };
int[] data_2 = new int[] { 1, 2, 3, 5 };
int[] data_3 = new[] { 1, 2, 3, 5 };
int[] data_4 = { 1, 2, 3, 5 };
```

При необходимости можно объявить массивы, имеющие несколько размерностей. Для этого в квадратных скобках после имени типа помещают запятые, «разделяющие» размерности

3. Операторы языка C#, их синтаксис.

Методы пользовательских типов состоят из операторов, которые выполняются последовательно. Часто используется *операторный блок* – последовательность операторов, заключённая в фигурные скобки. Иногда возникает необходимость в *пустом операторе* – он записывается как символ **;** (точка с запятой).

1. Операторы объявления

К *операторам объявления* относятся *операторы объявления переменных* и *операторы объявления констант*. Для объявления локальных переменных метода применяется оператор следующего формата:

тип имя-переменной [= начальное-значение];

Здесь *тип* – тип переменной, *имя-переменной* – допустимый идентификатор, необязательное *начальное-значение* – литерал или выражение, соответствующее типу переменной. Локальная переменная может быть объявлена без указания типа, с использованием ключевого слова var.

```
var x = 3;          var y = "Student";
```

```
var z = new Student();
```

Оператор объявления константы имеет следующий синтаксис:

const *тип-константы имя-константы = выражение*;

2. Операторы выражений

Операторы выражений – это выражения, одновременно являющиеся допустимыми операторами:

– операция присваивания (включая инкремент и декремент);
– операция вызова метода или делегата;
– операция создания объекта;
– операция асинхронного ожидания.

```
x = 1 + 2;              // присваивание
x++;                    // инкремент
Console.Write(x);       // вызов метода
new StringBuilder();    // создание объекта
await Task.Delay(1000); // асинхронное ожидание
```

3. Операторы перехода

К *операторам перехода* относятся `break`, `continue`, `goto`, `return`, `throw`. Оператор `break` используется для выхода из операторного блока циклов и оператора `switch`. Оператор `break` выполняет переход на оператор за блоком. Оператор `continue` располагается в теле цикла и применяется для запуска новой итерации цикла. Если циклы вложены, то запускается новая итерация того цикла, в котором непосредственно располагается `continue`.

Оператор `goto` передаёт управление на помеченный оператор. Обычно данный оператор употребляется в форме `goto` ***`метка`***, где ***`метка`*** – это допустимый идентификатор. Метка должна предшествовать помеченному оператору и заканчиваться двоеточием, отдельно описывать метки не требуется:

```
goto label;
. . .
label:
A = 100;
```

Оператор `return` служит для завершения методов. Оператор `throw` генерирует исключительную ситуацию

4. Операторы выбора

Операторы выбора – это операторы `if` и `switch`. Оператор `if` в языке C#

имеет следующий синтаксис:

if (*условие*)
  вложенный-оператор-1
*[*else
  вложенный-оператор-2]

switch (*выражение*)
{
  case *константное-выражение-1*:
    операторы
    оператор-перехода
  case *константное-выражение-2*:
    операторы
    оператор-перехода
  . . .
  *[*default:
    операторы
    оператор-перехода]
}

5. Операторы циклов

К *операторам циклов* относятся операторы `for`, `while`, `do-while`, `foreach`. Для циклов с известным числом итераций используется оператор `for`:

for (*[инициализатор]*; *[условие]*; *[итератор]*) *вложенный-оператор*

while (*условие*) *вложенный-оператор*

do
  вложенный-оператор
while (*условие*);

foreach (*тип идентификатор* in *коллекция*) *вложенный-оператор*

6. Прочие операторы

– Операторы `checked` и `unchecked` позволяют описать блоки *контролируемого* и *неконтролируемого контекстов вычислений*.

– Оператор `try` (в различных формах) применяется для перехвата и обработки *исключительных ситуаций*.

– Оператор `using` используется при *освобождении управляемых ресурсов*.
– Оператор `yield` служит для создания *итераторов*.
– Оператор `lock` применяется для объявления *критической секции*.

4. Принципы объектно-ориентированного программирования (ООП).

Парадигма программирования – это система идей и понятий, определяющих стиль создания компьютерных программ. Примерами парадигм являются императивное программирование, структурное программирование, объектно-ориентированное программирование. В *императивном программировании* процесс вычисления описывается в виде последовательности команд, которые должен выполнить компьютер. *Структурное программирование* основано на представлении программы в виде иерархической структуры связанных блоков (подзадач). Усложнение программного обеспечения привело к широкому распространению *объектно-ориентированного программирования* (ООП). Эта парадигма предлагает рассматривать программу как процесс взаимодействия некоторых вполне самостоятельных единиц, по аналогии с реальным миром называемых объектами. Создание программы заключается в наиболее полном описании соответствующих объектов и кодировании связей между ними. ООП оказалось настолько продуктивной идеей, что большинство современных языков программирования являются либо чисто объектно-ориентированными (Java, C#), либо содержат средства ООП в качестве надстройки (C++, Object Pascal).
ООП основывается на следующих принципах: *абстракция*, *инкапсуляция*, *наследование*, *полиморфизм*.
Абстракция в ООП – это набор наиболее значимых характеристик объекта (способ выделения подобных характеристик называется *абстрагированием*).
Инкапсуляция – это логическое объединение в одном программном типе, называемом *класс*, как данных, так и подпрограмм для их обработки. Данные класса хранятся в *полях класса*, подпрограммы для работы с полями называются *методами класса*.
Следующий тип отношений связан с ситуацией, когда понятие, соответствующее одному классу, уточняется понятием, соответствующим другому классу. Пусть необходим класс для описания служащих – `Employee`. Можно рассуждать так: любой служащий является человеком (`Person`), но служащий – это такой человек, который получает зарплату. Отношение между

классами `Employee` и `Person` называется *наследованием* (*отношение is-a*). Чтобы указать, что один класс является наследником другого, используется следующий синтаксис:

```
class имя-класса-наследника : имя-класса-предка {тело-класса}
```

Наследование от двух и более классов в C# и CLR запрещено.

```
public class Pet
{
    public void Speak() { Console.WriteLine("I'm a pet"); }
}

public class Dog : Pet
{
    public new void Speak() { Console.WriteLine("I'm a dog"); }
}
```

Полиморфи́зм — возможность объектов с одинаковой спецификацией иметь различную реализацию. Язык программирования поддерживает полиморфизм, если классы с одинаковой спецификацией могут иметь различную реализацию — например, реализация класса может быть изменена в процессе наследования. Кратко смысл полиморфизма можно выразить фразой: «Один интерфейс, множество реализаций».

Для организации полиморфного вызова в C# применяется два модификатора: `virtual` указывается для метода базового класса, который мы хотим сделать полиморфным, `override` – для методов производных классов. Эти методы должны совпадать по имени, типу и сигнатуре с перекрываемым методом класса-предка.

```
public class Pet
{
    public virtual void Speak() { Console.WriteLine("I'm a pet");
}
}
public class Dog : Pet
{
    public override void Speak() { Console.WriteLine("I'm a dog");
}
}
Pet pet = new Pet();
Pet dog = new Dog();
```

```
pet.Speak();            // печатает "I'm a pet"
dog.Speak();            // печатает "I'm a dog"
```

5. Описание класса на языке C#. Допустимые элементы класса.

Класс является основным пользовательским типом. Синтаксис объявления класса в C# следующий:

```
модификаторы class имя-класса
{
    [элементы-класса]
}
```

Допустимые элементы класса

1. Поле.

Синтаксис объявления поля класса совпадает с синтаксисом оператора объявления переменной (как правило, идентификаторы полей снабжаются неким оговорённым префиксом). Тип поля всегда должен быть указан явно, использование `var` не допускается. Если для поля не указано начальное значение, то поле принимает значение по умолчанию для соответствующего типа (для числовых типов – `0`, для типа `bool` – `false`, для ссылочных типов – `null`). Для полей можно применять модификатор `readonly`, который запрещает изменение поля после его начальной установки.

```
class Person
{
    readonly int _age = 20;
    string _name = "None";
}
```

Поля с модификатором `readonly` похожи на константы, но имеют следующие отличия: тип поля может быть любым; значение поля можно установить при объявлении или в конструкторе класса; значение поля вычисляется в момент выполнения, а не при компиляции.

2. Константа. Синтаксис объявления константы в классе аналогичен синтаксису, применяемому при объявлении константы в теле метода. Следующие элементы класса будут подробно рассмотрены в дальнейшем.

3. Метод. Методы описывают функциональность класса.

4. Свойство. Свойства класса предоставляют защищённый доступ к полям.

5. Индексатор. Индексатор – это свойство-коллекция, отдельный элемент которого доступен по индексу.

6. Конструктор. Задача конструктора – начальная инициализация объекта (экземплярный конструктор) или класса (статический конструктор).
7. Финализатор. Финализатор автоматически вызывается сборщиком мусора и содержит завершающий код для объекта.
8. Событие. События представляют собой механизм рассылки уведомлений различным объектам.
9. Операция. Язык C# допускает перегрузку некоторых операций для объектов класса.
10. Вложенный пользовательский тип. Описание класса может содержать описание другого пользовательского типа – класса, структуры, перечисления, интерфейса, делегата. Обычно вложенные типы выполняют вспомогательные функции и явно вне основного типа не используются.

Модификаторы доступа для элементов и типов
Для поддержания принципа инкапсуляции элементы класса могут снабжаться специальными *модификаторами доступа*:
– private. Элемент с этим модификатором доступен только в том типе, в котором определён. Например, поле доступно только в содержащем его классе.
– protected. Элемент виден в типе, в котором определён, и в наследниках этого типа (даже если наследники расположены в других сборках). Данный модификатор может применяться только в типах, поддерживающих наследование, то есть в классах.
– internal. Элемент доступен без ограничений, но только в той сборке, где описан.
– protected internal. Элемент виден в содержащей его сборке без ограничений, а вне сборки – только в наследниках типа (т.е. это комбинация protected *или* internal).
– public. Элемент доступен без ограничений как в той сборке, где описан, так и в других сборках, к которым подключается сборка с элементом.

Разделяемые классы
Разделяемые классы (partial classes) – это классы, разбитые на несколько фрагментов, описанных в отдельных файлах с исходным кодом.
Для объявления разделяемого класса используется модификатор partial

6. Наследование в ООП. Полиморфизм.

Следующий тип отношений связан с ситуацией, когда понятие, соответствующее одному классу, уточняется понятием, соответствующим другому классу. Пусть необходим класс для описания служащих – `Employee`. Можно рассуждать так: любой служащий является человеком (`Person`), но служащий – это такой человек, который получает зарплату. Отношение между классами `Employee` и `Person` называется *наследованием* (*отношение is-a*). Наследование является одним из базовых принципов ООП. Наследование предполагает создание новых классов на основе существующих. В нашем случае мы можем не писать класс `Employee` «с нуля», а воспользоваться классом `Person` как основой. При наследовании новый класс называется *классом-потомком* (или *дочерним классом*, *производным классом*), старый – *классом-предком* (или *родительским классом*, *базовым классом*). При помощи наследования можно строить так называемое *дерево классов* (или *иерархию классов*), последовательно уточняя описание класса и переходя от общих понятий к частным.

`class` *имя-класса-наследника* `:` *имя-класса-предка* `{`*тело-класса*`}`

Наследование от двух и более классов в C# и CLR запрещено. Наследник обладает всеми полями, методами и свойствами предка, но элементы предка с модификатором `private` не доступны в наследнике. Конструкторы класса-предка не переносятся в класс-наследник. При наследовании также нельзя расширить область видимости класса.

Для объектов класса-наследника определено неявное преобразование к типу класса-предка. C# содержит две специальные операции, связанные с контролем типов при наследовании. Выражение `x is T` возвращает значение `true`, если типом `x` является или тип `T`, или наследник типа `T`. Выражение `x as T` возвращает объект, приведённый к типу `T`, если это возможно, и `null` в противном случае.

Для обращения к методам класса-предка класс-наследник может использовать ключевое слово `base` в форме `base.`*метод-базового-класса*. Если конструктор наследника должен вызвать конструктор предка, то для этого также используется ключевое слово `base`:

конструктор-наследника`(`*[параметры]*`) : base(`*[параметры_2]*`)`

Модификатор `sealed` определяет *запечатанный класс*, от которого запрещено наследование. Модификатор `abstract` определяет *абстрактный класс*, у которого обязательно должны быть наследники. Объект абстрактного класса создать нельзя, хотя статические элементы такого класса можно вызвать:

```
sealed class SealedClass { }
abstract class AbstractClass { }
public class Pet
{
    public void Speak() { Console.WriteLine("I'm a pet"); }
}

public class Dog : Pet
{
    public new void Speak() { Console.WriteLine("I'm a dog"); }
}
```

Полиморфизм является одним из трёх принципов ООП.

Полиморфи́зм — возможность объектов с одинаковой спецификацией иметь различную реализацию. Язык программирования поддерживает полиморфизм, если классы с одинаковой спецификацией могут иметь различную реализацию — например, реализация класса может быть изменена в процессе наследования. Кратко смысл полиморфизма можно выразить фразой: «Один интерфейс, множество реализаций». Полиморфизм — один из четырёх важнейших механизмов объектно-ориентированного программирования (наряду с абстракцией, инкапсуляцией и наследованием). Полиморфизм позволяет писать более абстрактные программы и повысить коэффициент повторного использования кода. Общие свойства объектов объединяются в систему, которую могут называть по-разному — интерфейс, класс. Таким образом, полиморфизм – особый вид перекрытия методов при наследовании, при котором программный код, работавший с методами родительского класса, пригоден для работы с изменёнными методами дочернего класса.

Полиморфизм включения

Этот полиморфизм называют чистым полиморфизмом. Применяя такую форму полиморфизма, родственные объекты можно использовать обобщенно. С помощью замещения и полиморфизма включения можно написать один

метод для работы со всеми типами объектов Person. Используя полиморфизм включения и замещения можно работать с любым объектом, который проходит тест «is-A». Полиморфизм включения упрощает работу по добавлению к программе новых подтипов, так как не нужно добавлять конкретный метод для каждого нового типа, можно использовать уже существующий, только изменив в нем поведение системы. С помощью полиморфизма можно повторно использовать базовый класс; использовать любого потомка или методы, которые использует базовый класс.

Параметрический полиморфизм

Используя Параметрический полиморфизм можно создавать универсальные базовые типы. В случае параметрического полиморфизма, функция реализуется для всех типов одинаково и таким образом функция реализована для произвольного типа. В параметрическом полиморфизме рассматриваются параметрические методы и типы.

Для организации полиморфного вызова применяется два модификатора: `virtual` указывается для метода базового класса, который мы хотим сделать полиморфным, `override` – для методов производных классов. Эти методы должны совпадать по имени, типу и сигнатуре с перекрываемым методом класса-предка.

```
    public class Pet
    {
        public virtual void Speak() { Console.WriteLine("I'm a pet");
}
    }
    public class Dog : Pet
    {
        public override void Speak() { Console.WriteLine("I'm a dog");
}
    }
    Pet pet = new Pet();
    Pet dog = new Dog();
    pet.Speak();              // печатает "I'm a pet"
    dog.Speak();              // печатает "I'm a dog"
```

При описании метода возможно совместное указание модификаторов `new` и `virtual`. Такой приём создаёт новую полиморфную цепочку замещения.

7. Универсальные шаблоны (generics) в .NET и языке C#.

Универсальные шаблоны (generics) позволяют при разработке пользовательского типа или метода указать в качестве параметра тип, который конкретизируется при использовании. Универсальные шаблоны применимы к классам, структурам, интерфейсам, делегатам и методам.

Универсальные классы и структуры

Поясним необходимость универсальных шаблонов на следующем примере. Пусть разрабатывается класс для представления структуры данных «стек». Чтобы не создавать отдельные версии стека для хранения данных определённых типов, программист выбирает базовый тип object как тип элемента:

```
public class Stack
{
   private object[] _items;
   public void Push(object item) { . . . }
   public object Pop() { . . . }
}
```

Класс Stack можно использовать для разных типов данных:

```
var stack = new Stack();
stack.Push(new Customer());
Customer c = (Customer)stack.Pop();

var stack2 = new Stack();
stack2.Push(3);
int i = (int)stack2.Pop();
```

Минусы: Упаковка и приведение типов.

Опишем класс Stack как универсальный тип. Для этого используется следующий синтаксис: после имени класса в угловых скобках указывается *параметр типа*.

```
public class Stack<T>
{
   private T[] _items;
   public void Push(T item) { . . . }
   public T Pop() { . . . }
}
```

Использовать универсальный тип «как есть» в клиентском коде нельзя, так как он является не типом, а, скорее, «чертежом» типа. Для работы со Stack<T> необходимо использовать *сконструированный тип* (constructed type), указав в угловых скобках аргумент типа. Аргумент-тип может быть любым типом. Можно создать любое количество экземпляров сконструированных типов, и каждый из них может использовать разные аргументы типа.

```
Stack<int> stack = new Stack<int>();
stack.Push(3);
int x = stack.Pop();
```

Обратите внимание: при работе с типом Stack<int> отпала необходимость в выполнении приведения типов при извлечении элементов из стека. Кроме этого, теперь компилятор отслеживает, чтобы в стек помещались только данные типа int. И ещё одна особенность: нет необходимости в упаковке и распаковке типа значения, а это приводит к росту производительности.

Универсальные методы

В некоторых случаях достаточно параметризировать не весь пользовательский тип, а только отдельный метод. *Универсальные методы* (generic methods) объявляются с использованием параметров-типов в угловых скобках после имени метода. Как и при описании универсальных типов, универсальные методы могут содержать ограничения на параметр-тип.

```
void PushMultiple<T>(Stack<T> stack, params T[] values)
{
   foreach (T value in values)
   {
      stack.Push(value);
   }
}
```

Использование универсального метода PushMultiple<T> позволяет работать с любым сконструированным типом на основе Stack<T>.

```
Stack<int> stack = new Stack<int>();
PushMultiple<int>(stack, 1, 2, 3, 4);
```

8. Интерфейсы как элемент ООП.

Согласно общей парадигме ООП, *интерфейс* (interface) – это элемент языка, который служит для специфицирования услуг, предоставляемых классом. Класс может *реализовывать* интерфейс. Реализация интерфейса заключается в

том, что в описании класса данный интерфейс указывается как реализуемый, а в коде класса обязательно определяются все методы, которые имеет интерфейс. Один класс может реализовать несколько интерфейсов одновременно. Возможно объявление переменных как имеющих тип-интерфейс. В такую переменную может быть записан экземпляр любого класса, реализующего интерфейс. Таким образом, с одной стороны, интерфейс – это «договор», который обязуется выполнить класс, реализующий его, с другой стороны, интерфейс – это пользовательский тип, потому что его описание достаточно чётко определяет характеристики объектов, чтобы наравне с классом типизировать переменные.

Использование интерфейсов – один из вариантов обеспечения полиморфизма в объектных языках. Все классы, реализующие один и тот же интерфейс, с точки зрения определяемого ими поведения, ведут себя внешне одинаково. Это позволяет писать обобщённые алгоритмы обработки данных, использующие в качестве типов параметры интерфейсов, и применять их к объектам различных типов, всякий раз получая требуемый результат.

Язык C# следует представленной выше концепции интерфейсов. В C# интерфейсы могут реализовывать не только классы, но и структуры. Поддерживается множественное наследование интерфейсов.

Для объявления интерфейса используется ключевое слово interface. Интерфейс содержит только заголовки методов, свойств и событий. Для свойства указываются только ключевые слова get и (или) set. При объявлении элементов интерфейса не могут использоваться следующие модификаторы: abstract, public, protected, internal, private, virtual, override, static. Считается, что все элементы интерфейса имеют public-уровень доступа:

```
public interface IFlyable
{
    void Fly();                     // метод
    double Speed { get; set; }      // свойство
}
```

Чтобы указать, что тип реализует некий интерфейс, используется синтаксис *имя-типа : имя-интерфейса* при записи заголовка типа. Если класс является производным от некоторого базового класса, то имя базового класса указывается перед именем реализуемого интерфейса.

Элементы интерфейса допускают явную и неявную реализацию. При *неявной реализации* тип должен содержать открытые экземплярные элементы, имена и

сигнатура которых соответствуют элементам интерфейса. При *явной реализации* элемент типа называется по форме *имя-интерфейса.имя-элемента*, а указание любых модификаторов для элемента при этом запрещается.

```
public class Falcon : IFlyable
{
    // неявная реализация интерфейса IFlyable
    public void Fly() { Console.WriteLine("Falcon flies"); }
    public double Speed { get; set; }
}
public class Eagle : IFlyable
{
    // обычный метод
    public void PrintType() { Console.WriteLine("Eagle"); }
    // явная реализация интерфейса IFlyable
    void IFlyable.Fly() { Console.WriteLine("Eagle flies"); }
    double IFlyable.Speed { get; set; }
}
```

Если тип реализует некоторые элементы интерфейса явно, то такие элементы будут недоступны через переменную типа. Допустимо объявить переменную интерфейса, которая может содержать значение любого типа, реализующего интерфейс (для структур будет выполнена операция упаковки). Через переменную интерфейса можно вызывать только элементы интерфейса.

```
Eagle eagle = new Eagle(); // обычное создание объекта
eagle.PrintType();         // у объекта доступен только этот метод
IFlyable x = eagle;        // переменная интерфейса
x.Fly();                   // получили доступ к элементам
интерфейса
x.Speed = 60;
```

Все неявно реализуемые элементы интерфейса по умолчанию помечаются в классе как sealed. А значит, наследование классов не ведёт к прямому наследованию реализаций.

Подобно классам, интерфейсы могут наследоваться от других интерфейсов. При этом наследование интерфейсов может быть множественным. Один класс может реализовывать несколько интерфейсов – имена интерфейсов перечисляются после имени класса через запятую.

9. Делегаты и лямбда-выражения.

Делегат – это пользовательский тип, который инкапсулирует метод. В C# делегат объявляется с использованием ключевого слова `delegate`. При этом указывается имя делегата, тип и сигнатура инкапсулируемого метода:

```
public delegate double Function(double x);
public delegate void Subroutine(int i);
```

Делегат – самостоятельный пользовательский тип, он может быть как вложен в другой пользовательский тип (класс, структуру), так и объявлен отдельно.

Переменные делегата инициализируются конкретными методами при использовании *конструктора делегата* с одним параметром – именем метода (или именем другого делегата). При этом метод должен обладать подходящей сигнатурой:

```
Function F;
Subroutine S;
F = new Function(ClassName.SomeStaticFunction);
S = new Subroutine(obj.SomeInstanceMethod);
```

Упрощённый синтаксис (вызов конструктора делегата генерируется самим компилятором):

```
F = ClassName.SomeStaticFunction;
S = obj.SomeInstanceMethod;
```

Вызов инкапсулированного метода выполняет экземплярный метод делегата `Invoke()`. Вместо использования `Invoke()` можно просто указать аргументы вызываемого метода непосредственно после имени переменной-делегата.

Приведём пример работы с делегатами. Создадим класс, содержащий метод расширения для трансформации массива целых чисел:

```
public static class ArrayHelper
{
    public static int[] Transform(this int[] data, Transformer f)
    {
        var result = new int[data.Length];
        for (int i = 0; i < data.Length; i++)
        {
            // альтернатива: result[i] = f.Invoke(data[i]);
            result[i] = f(data[i]);
        }
        return result;
```

```
    }
}
```

Тип `Transformer` является делегатом, который определяет способ преобразования отдельного числа. Он описан следующим образом:

```
public delegate int Transformer(int x);
```

Создадим класс, который использует `ArrayHelper` и `Transformer`:

```
public class MainClass
{
    public static int TimesTwo(int i) { return i * 2; }

    public int AddFive(int i) { return i + 5; }

    public static void Main()
    {
        int[] a = {1, 2, 3};
        Transformer t = TimesTwo;
        a = a.Transform(t);
        var c = new MainClass();
        a = a.Transform(c.AddFive);
    }
}
```

Делегаты обладают *контравариантностью* относительно типов параметров и *ковариантностью* относительно типа результата. Это значит, что делегат может инкапсулировать метод, у которого входные параметры имеют более общий тип, а тип результата более специфичен, чем описано в делегате (заметим, что вариантность работает только для ссылочных типов).

Делегаты могут быть объявлены как универсальные типы. При этом допустимо использование ограничений на параметр типа, а также указания на ковариантность и контравариантность параметров типа.

```
public delegate TResult Transformer<in T, out TResult>(T x);
```

Ключевой особенностью экземпляров делегатов является то, что они могут инкапсулировать не один метод, а несколько. Подобные объекты называются *групповыми делегатами*. При вызове группового делегата срабатывает вся цепочка инкапсулированных в нем методов.

Групповой делегат объявляется таким же образом, как и обычный. Операция + используется для объединения объектов делегата в один групповой делегат. Операция – удаляет объект из цепочки группового делегата (удаляется последнее вхождение объекта).

Любой пользовательский делегат можно рассматривать как наследник класса `System.MulticastDelegate`, который, в свою очередь, наследуется от `System.Delegate`.

Назначение *анонимных методов* (anonymous methods) заключается в том, чтобы сократить объём кода, который должен писать разработчик при использовании делегатов. При применении анонимных методов формируется безымянный блок кода, который назначается объекту-делегату.

Лямбда-выражения и *лямбда-операторы* – это альтернативный синтаксис записи анонимных методов. Начнём с рассмотрения лямбда-операторов. Пусть имеется анонимный метод:

```
Func<int, bool> f = delegate(int x)
                    {
                        int y = x - 100;
                        return y > 0;
                    };
```

При использовании лямбда-операторов список параметров отделяется от тела оператора символами `=>`, а ключевое слово `delegate` не указывается:

```
Func<int, bool> f = (int x) => { int y = x - 100; return y > 0; };
```

Более того, так как тип параметра лямбда-оператора уже фактически указан слева при объявлении `f`, то его можно не указывать справа. В случае одного параметра можно также опустить обрамляющие его скобки[1]:

```
Func<int, bool> f = x => { int y = x - 100; return y > 0; };
```

Когда лямбда-оператор содержит в своём теле единственный оператор `return`, он может быть записан в компактной форме *лямбда-выражения*:

```
Func<int, bool> f_2 = x => x > 0;
```

Анонимные методы и лямбда-операторы способны захватывать внешний контекст вычисления. Если при описании тела анонимного метода применялась внешняя переменная, вызов метода будет использовать *текущее* значение переменной. Захват внешнего контекста иначе называют *замыканием* (closure).

```
// код, который написал пользователь
public class MainClass
{
    public static void Main()
```

[1] Если параметров несколько, скобки нужно указывать. Когда лямбда-оператор не имеет входных параметров, указываются пустые скобки.

```
    {
        int external = 0;
        System.Func<int, bool> f = x => x > external;
        external = 10;
        var result = f(1);
    }
}
```

10. Событийное программирование. Описание и использование событий.

Событийно-ориентированное программирование (СОП) – парадигма программирования, в которой выполнение программы определяется *событиями* – действиями пользователя (клавиатура, мышь), сообщениями других программ и потоков, событиями операционной системы. СОП, как правило, применяется при построении пользовательских интерфейсов и при программировании игр, в которых осуществляется управление множеством объектов.

Работу с событиями можно условно разделить на три этапа:

– *объявление* события (publishing);

– *регистрация* получателя события (subscribing);

– *генерация* события (raising).

В языке C# событие можно объявить в пределах класса или структуры. Для этого используется ключевое слово `event` и делегат, описывающий метод обработки события. Обычно этот делегат имеет тип возвращаемого значения `void`[2].

`модификаторы event тип-делегата имя-события;`

Встретив подобное объявление события, компилятор добавляет в класс или структуру `private`-поле с именем *`имя-события`* и типом *`тип-делегата`*. Кроме этого, для обслуживания события компилятор создаёт два метода `add_`*`Name`*`()` и `remove_`*`Name`*`()`, где *`Name`* – имя события. Эти методы содержат код, добавляющий и удаляющий обработчик события в цепочку группового делегата, связанного с событием. Если программиста по каким-либо причинам не устраивает подход, используемый компилятором, он может реализовать собственную логику добавления и удаления обработчика события[3]. Для этого

[2] Интерфейсы также могут содержать события, объявленные указанным способом.

[3] Это справедливо для классов и структур, но не для интерфейсов.

при объявлении события указывается блок, содержащий секции `add` и `remove`:

```
модификаторы event тип-делегата имя-события
{
    add { операторы }
    remove { операторы }
};
```

В блоке добавления и удаления обработчиков обычно размещается код, добавляющий или удаляющий метод в цепочку группового делегата. Поле-делегат в этом случае также должно быть явно объявлено в типе.

Для генерации события в требуемом месте кода помещается вызов в формате *имя-события*(*фактические-аргументы*). Предварительно обычно проверяют, назначен ли обработчик события. Генерация события может происходить только в том же типе, в котором событие объявлено[4].

Рассмотрим этап регистрации получателя события. Чтобы отреагировать на событие, его надо ассоциировать с *обработчиком события*. Обработчиком может быть метод, приводимый к типу события (делегату). В качестве обработчика может выступать анонимный метод или лямбда-оператор. Назначение и удаление обработчиков события выполняется при помощи операторов `+=` и `-=`.

Платформа .NET предлагает средства стандартизации работы с событиями. В частности, для типов событий зарезервированы следующие делегаты:

```
public delegate void EventHandler(object sender, EventArgs e);
public delegate void EventHandler<T>(object sender, T e)
                                        where T : EventArgs;
```

Как видим, данные делегаты предполагают, что первым параметром будет выступать объект, в котором событие было сгенерировано. Второй параметр используется для передачи информации события. Это либо класс `EventArgs`, либо наследник этого класса с необходимыми полями.

Сама генерация события обычно выносится в отдельный виртуальный метод класса. В этом методе непосредственно перед генерацией события проверяется, установлен ли обработчик события. Перед проверкой можно создать копию события (это актуально в многопоточных приложениях).

Приведём пример класса, содержащего объявление и генерацию события. Данный

[4] Поведение, аналогичное событиям, можно получить, используя открытие поля делегатов. Ключевое слово `event` заставляет компилятор проверять, что описание и генерация события происходят в одном типе, и запрещает для события все операции, кроме `+=` и `-=`.

класс будет включать метод с целым параметром, устанавливающий значение поля класса. Если значение параметра отрицательно, генерируется событие, определённое в классе:

```
public delegate void Handler(int val);

public class ExampleClass
{
    private int _field;

    public int Field
    {
        get { return _field; }
        set
        {
            _field = value;
            if (value < 0)
            {
                // проверяем, есть ли обработчик
                if (NegativeValueSet != null)
                {
                    NegativeValueSet(value);
                }
            }
        }
    }

    public event Handler NegativeValueSet;
}


public class MainClass
{
    public static void Reaction(int i)
    {
        Console.WriteLine("Negative value = {0}", i);
    }

    public static void Main()
    {
        var c = new ExampleClass();
```

```
            c.Field = -10;   // нет обработчиков, нет реакции на
событие

            // назначаем обработчик
            c.NegativeValueSet += Reaction;
            c.Field = -20;    // вывод: "Negative value = -20"

            // назначаем ещё один обработчик в виде лямбда-выражения
            c.NegativeValueSet += i => Console.WriteLine(i);
            c.Field = -30;    // вывод: "Negative value = -30" и "-30"

            // удаляем первый обработчик
            c.NegativeValueSet -= Reaction;
        }
    }
```

11. Синтаксис генерации и обработки исключительных ситуаций на языке C#

Для генерации исключительной ситуации используется оператор `throw`:

throw *[объект-класса-исключительной-ситуации]*;

Объект, указанный после `throw`, должен быть объектом класса исключительной ситуации. В C# классами исключительных ситуаций являются класс `System.Exception` и все его наследники. Класс `Exception` – это базовый класс для представления исключительных ситуаций. Разработчик может создать собственный класс для представления информации об исключительной ситуации. Единственным условием для этого класса в C# является прямое или косвенное наследование от класса `Exception`. Рассмотрим пример с генерацией исключительной ситуации:

```
using System;
public class ExampleClass
{
    public int Field
    {
        get { return _field; }
        set
        {
```

```
                throw new ArgumentOutOfRangeException();
            }
        }
    }
    public class MainClass
    {
        public static void Main()
        {
            var a = new ExampleClass();
            a.Field = -3;    // ИС генерируется, но не обрабатывается!
        }
    }
```

Для перехвата исключительных ситуаций служит операторный блок `try – catch – finally`. Синтаксис блока следующий:

```
try
{
    [операторы,-способные-вызвать-исключительную-ситуацию]
}
[один-или-несколько-блоков-catch]
[finally
{
    операторы-из-секции-завершения
}]
```

Операторы из части `finally` (если она присутствует) выполняются всегда, вне зависимости от того, произошла исключительная ситуация или нет. Если один из операторов, расположенных в блоке `try`, вызвал исключительную ситуацию, управление немедленно передаётся на блоки `catch`. Синтаксис отдельного блока `catch` следующий:

```
catch [(тип-ИС [идентификатор-объекта-ИС])]
{
    операторы-обработки-исключительной-ситуации
}
```

Здесь *идентификатор-объекта-ИС* – это некая временная переменная, которая может использоваться для извлечения информации из объекта исключительной ситуации. Отдельно описывать эту переменную не надо. Если указать в блоке `catch` оператор `throw` без аргумента, это приведёт к тому, что обрабатываемая исключительная ситуация будет сгенерирована повторно.

Добавим блок перехвата ошибки в предыдущий пример:

```
var a = new ExampleClass();
try
{
    a.Field = -3;
}
catch (ArgumentOutOfRangeException ex)
{
    Console.WriteLine(ex.Message);
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}
finally
{
    Console.WriteLine("This line is printed always (finally)");
}
```

Если используется несколько блоков `catch`, то обработка исключительных ситуаций должна вестись по принципу «от частного – к общему», так как после выполнения одного блока `catch` управление передаётся на часть `finally` (при отсутствии `finally` – на оператор после `try - catch`). Компилятор C# не позволяет разместить блоки `catch` так, чтобы предыдущий блок перехватывал исключительные ситуации, предназначенные последующим блокам.

12. Жизненный цикл объектов. Алгоритмы «сборки мусора».

Все типы платформы .NET делятся на ссылочные типы и типы значений. Переменные типов значений создаются в стеке, их время жизни ограничено тем блоком кода, в котором они объявляются. Например, если переменная типа значения объявлена в некотором методе, то после выхода из метода память в стеке, занимаемая переменной, автоматически освободится. Переменные ссылочного типа (объекты) располагаются в динамической памяти – *управляемой куче* (managed heap). Размещение объектов в управляемой куче происходит последовательно. Для этого CLR поддерживает указатель на свободное место в куче, перемещая его на соответствующее количество байтов после выделения памяти очередному объекту.

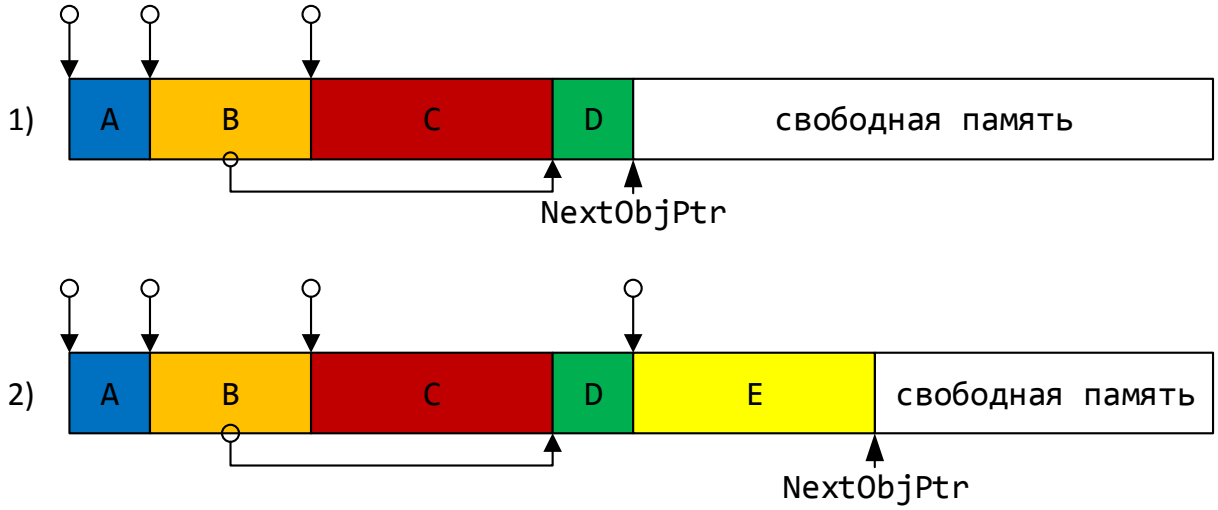


Рис. 1. Управляемая куча до (1) и после (2) выделения памяти под объект E.

Алгоритм сборки мусора

Если новый объект требует для размещения больше памяти, чем имеющийся свободный объем, CLR запускает процесс, называемый *сборка мусора*[5] (garbage collection). На первом этапе сборки мусора строится *граф используемых объектов*. Отправными точками в построении графа являются *корневые объекты*. Это объекты следующих категорий:

– локальная переменная или аргумент выполняемого метода (а также всех методов в стеке вызова);

– статическое поле;

– объект в *очереди завершения* (этот термин будет разъяснён позже).

При помощи графа используемых объектов выясняется реально занимаемая этими объектами память. Затем происходит *дефрагментация кучи* – используемые объекты перераспределяются так, чтобы занимаемая ими память составляла единый блок в начале кучи. После этого сборка мусора завершается, и новый объект размещается в управляемой куче.

[5] Компилятор C# генерирует код, принудительно запускающий сборку мусора при окончании работы программы.

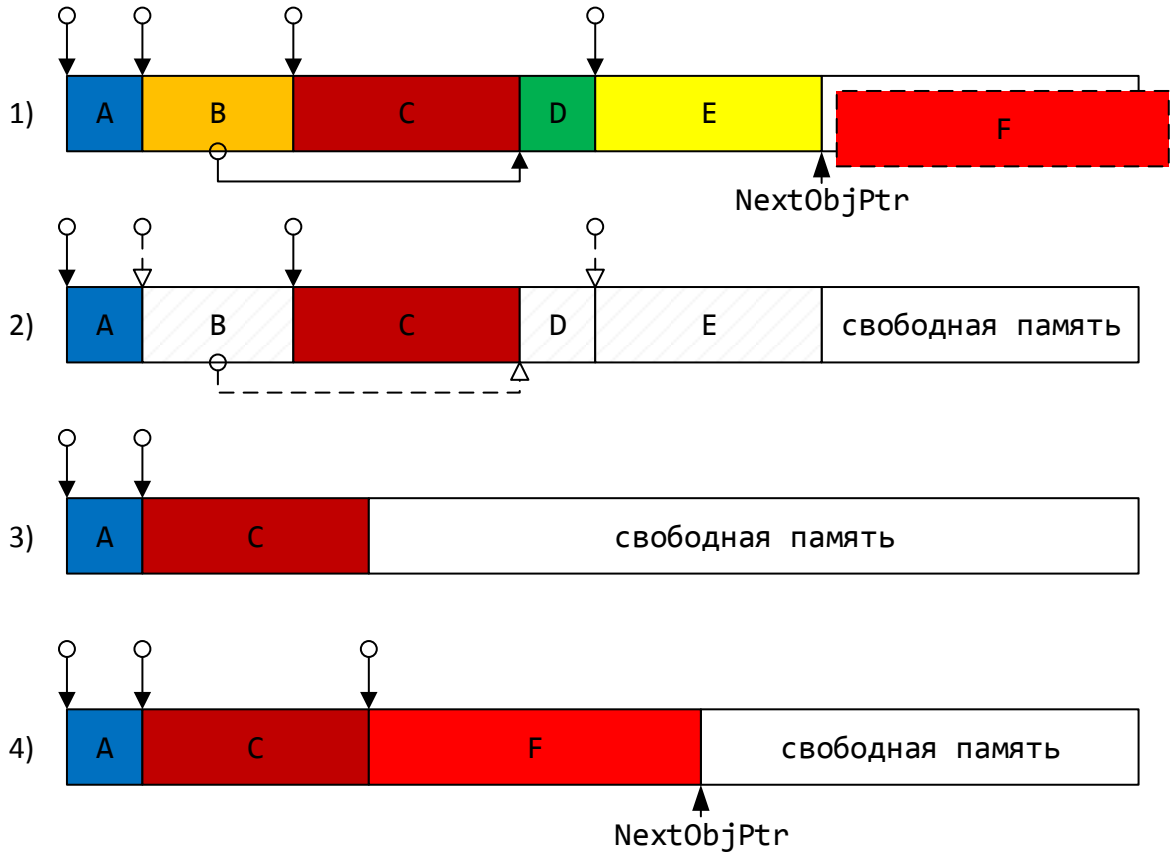


Рис. 2. Фазы сборки мусора: 1) инициирование, 2) выявление используемых объектов, 3) дефрагментация, 4) размещение объекта.

При размещении и удалении объектов CLR использует ряд оптимизаций. Во-первых, объекты размером 85000 и более байтов размещаются в отдельной управляемой *куче больших объектов* (Large Object Heap). При сборке мусора данная куча не дефрагментируется, так как копирование больших блоков памяти снижает производительность. Во-вторых, в управляемой куче для малых объектов выделяется три поколения – Gen0, Gen1 и Gen2. Вначале все объекты относятся к поколению Gen0. После первой сборки мусора «пережившие» её объекты переходят в поколение Gen1. В дальнейшем сборка мусора будет работать с объектами поколения Gen1, только если освобождение памяти в Gen0 дало неудовлетворительный результат. Если в Gen1 произошла сборка мусора, то «пережившие» её объекты переходят в поколение Gen2. Отметим, что куча больших объектов всегда рассматривается как куча объектов поколения Gen2.

Сборщик мусора представлен статическим классом `System.GC`, который обладает несколькими полезными методами (приведён неполный список):

1. `Collect()` – вызывает принудительную сборку мусора в программе.
2. `GetGeneration()` – возвращает номер поколения для указанного объекта;
3. `SuppressFinalize()` – подавляет вызов финализатора для объекта;
4. `WaitForPendingFinalizers()` – приостанавливает текущий поток выполнения до тех пор, пока не будут выполнены все финализаторы освобождаемых объектов.

13. Стандартные типы платформы .NET для представления коллекций.

Платформа .NET включает большой набор типов для предоставления стандартных коллекций - списков, множеств, словарей. Эти типы можно разделить на несколько категорий: базовые интерфейсы и вспомогательные классы, классы для коллекций-списков и коллекций-словарей, набор классов для построения собственных коллекций. Используется несколько пространств имен:

- System.Collections – содержит типы, в которых элемент коллекции представлен как object (*слаботипизированные коллекции*).
- System.Collections.Specialized – специальные коллекции.
- System.Collections.Generic – универсальные классы и интерфейсы коллекций.
- System.Collections.ObjectModel – базовые и вспомогательные типы, которые могут применяться для построения пользовательских коллекций.

Перечисляемый тип (enumerable type) – это тип, который имеет экземплярный метод GetEnumerator(), возвращающий перечислитель. *Перечислитель (enumerator)* – объект, обладающий свойством Current, представляющим текущий элемент набора, и методом MoveNext() для перемещения к следующему элементу. Оператор foreach получает перечислитель, вызвав у объекта метод GetEnumerator(), а затем использует MoveNext() и Current для итерации по набору.

Опишем набор интерфейсов, реализуемых практически всеми типами коллекций.

Интерфейсы IEnumerable и IEnumerator описаны в пространстве имён System.Collections:

```
public interface IEnumerable
{
    IEnumerator GetEnumerator();
}

public interface IEnumerator
{
    object Current { get; }
    bool MoveNext();
    void Reset();
}
```

Свойство IEnumerator.Current имеет тип object, то на каждой итерации выполняется приведение этого свойства к типу переменной цикла . Это может повлечь ошибки времени выполнения. Избежать ошибок помогает реализация перечисляемого типа при помощи универсальных интерфейсов IEnumerable<T> и IEnumerator<T>:

Универсальные интерфейсы IEnumerable<T> и IEnumerator<T> наследуются от обычных версий.

```
public interface IEnumerable<out T> : IEnumerable
{
    IEnumerator<T> GetEnumerator();
}
public interface IEnumerator<out T> : IDisposable, IEnumerator
{
    T Current { get; }
}
```

Пример

```
List<string> words = new List<string> {"melon", "avocado"};
foreach (string word in words)
{
     Console.WriteLine(word);
}
```

Слаботипизированные словари реализуют интерфейс IDictionaryEnumerator для перебора своих пар «ключ-значение».

Интерфейсы ICollection, IList, IDictionary и их универсальные аналоги предоставляют возможность определения размера коллекции, доступа к элементу по индексу, поиска элемента и модификации коллекции.

ICollection - интерфейс для коллекций, запоминающих число хранимых

элементов. Он имеет соответствующее свойство, а также методы для копирования коллекции в массив и свойства для синхронизации коллекции при многопоточном использовании.

Универсальный интерфейс ICollection<T> также поддерживает свойство для количества элементов. Кроме этого, он предоставляет методы для добавления и удаления элементов.

Интерфейс IList описывает набор данных, которые проецируются на массив. Дополнительно к функциональности, унаследованной от IEnumerable и ICollection, интерфейс IList позволяет обращаться к элементу по индексу, добавлять, удалять и искать элементы.

Возможности интерфейса IList<T> тождественны возможностям IList.

Интерфейсы IDictionary и IDictionary<TKey, TValue> определяют протокол взаимодействия для всех коллекций-словарей.

Интерфейсы IDictionary и IDictionary<TKey, TValue> используют вспомогательные структуры DictionaryEntry и KeyValuePair<TKey, TValue>, у которых определены свойства Key и Value.

Для работы с коллекциями-множествами предназначен интерфейс `ISet<T>`.

Для взаимодействия с коллекциями, предназначенными только для чтения, в платформе .NET версии 4.5 введён интерфейс `IReadOnlyList<T>`.

Рис. 1 демонстрирует связи между интерфейсами коллекций.

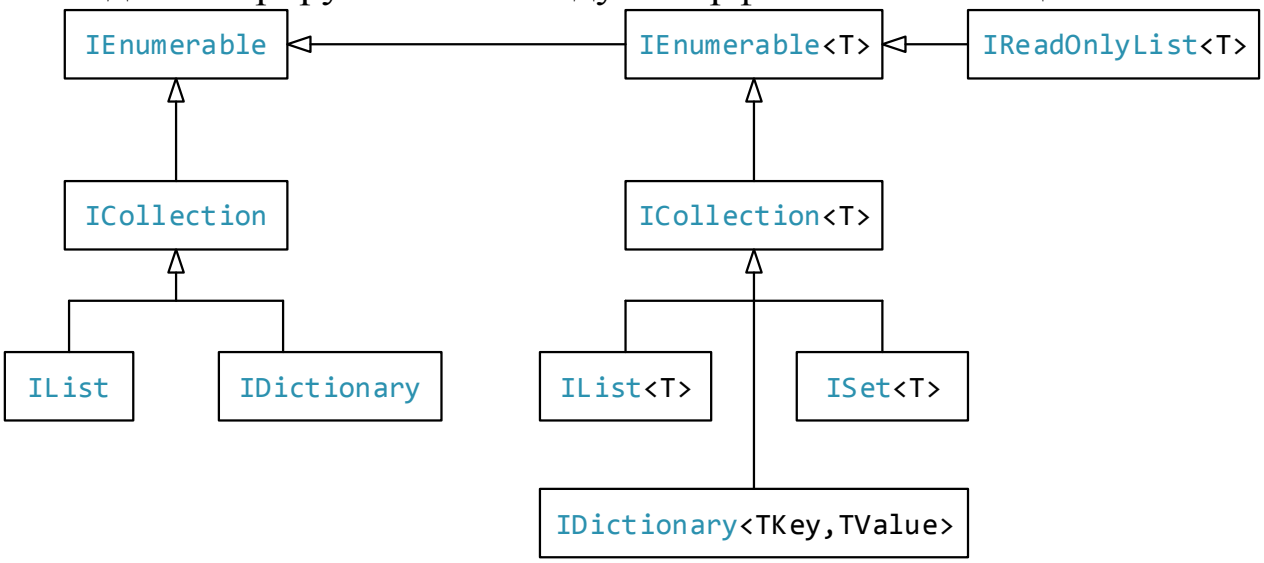


Рис. 1. Стандартные интерфейсы коллекций.

Массивы и класс System.Array

Класс `System.Array` является базовым классом для любого массива.

Любой массив реализует интерфейсы `IStructuralEquatable`, `IStructuralComparable`, `IList` и `IList<T>`, причём первые три интерфейса реализованы явно (методы `Add()` и `Remove()` генерируют исключение в случае коллекции фиксированной длины, которой является массив).

```
    Array a = Array.CreateInstance(typeof (string), 2);    // тип, длина
    a.SetValue("hi", 0);                    // a[0] = "hi";
    a.SetValue("there", 1);                 // a[1] = "there";
    string s = (string) a.GetValue(0);      // s = a[0];
```

Типы для работы с коллекциями-списками

Класс `List<T>` из пространства имён `System.Collections.Generic` – основной класс для представления наборов, которые допускают динамическое добавление элементов[6]. Для хранения данных набора используется внутренний массив. Класс `List<T>` реализует интерфейсы `IList<T>`, `IList`, `IReadOnlyList<T>`.

```
    List<string> words = new List<string> {"melon", "avocado"};
    words.AddRange(new[] {"banana", "plum"});
    words.Insert(0, "lemon");
    words.InsertRange(0, new[] {"peach", "apple"});
    words.Remove("melon");
    words.RemoveAt(3);
    words.RemoveAll(s => s.StartsWith("a"));
    List<string> subset = words.GetRange(1, 2);
    string[] wordsArray = words.ToArray();
```

В пространстве имён `System.Collections` имеется слаботипизированный аналог класса `List<T>` – класс `ArrayList`

Класс `LinkedList<T>` служит для представления двусвязного списка. Такой список позволяет осуществлять вставку и удаление элемента без сдвига остальных элементов. Однако доступ к элементу по индексу требует прохода по списку. `LinkedList<T>` реализует интерфейсы `ICollection` и `ICollection<T>`. Каждый элемент двусвязного списка представлен объектом `LinkedListNode<T>`.

Классы `Queue<T>` и `Stack<T>` реализуют структуры данных «очередь» и «стек» на основе массива. В пространстве имён `System.Collections`

имеются слаботипизированные аналоги классов `Queue<T>` и `Stack<T>` – классы `Queue` и `Stack`.

Типы для работы с коллекциями-множествами

Класс `HashSet<T>` описывает множество, в котором вхождение элемента проверяется на основе хэш-кода.

```
var setOne = new HashSet<char>("the quick brown fox");
var setTwo = new HashSet<char>("jumps over the lazy dog");
setOne.IntersectWith(setTwo);                // результат = the uro
Console.WriteLine(setOne.Contains('t'));     // True
Console.WriteLine(setOne.Contains('j'));     // False
setTwo.RemoveWhere(c => c < 'k');
```

Класс `SortedSet<T>` – это множество, поддерживающее набор элементов в отсортированном порядке.

```
var setOne = new SortedSet<char>("the quick brown fox");
Console.WriteLine("{0} {1}", setOne.Min, setOne.Max);
IEnumerable<char> reversed = setOne.Reverse();
var setTwo = setOne.GetViewBetween('a', 'p');
Console.WriteLine("{0} {1}", setTwo.Min, setTwo.Max);
```

Оба класса реализуют интерфейс `ISet<T>`.

Типы для работы с коллекциями-словарями

Универсальный класс `Dictionary<TKey, TValue>` – классический словарь с возможностью указать тип для ключа и тип для значения. Класс `Dictionary<TKey, TValue>` реализует обе версии интерфейса `IDictionary` (обычную и универсальную). В пространстве имён `System.Collections` имеется слаботипизированный аналог класса `Dictionary<TKey,TValue>` – класс `Hashtable.`

```
// конструируем словарь и помещаем в него один элемент
// обратите внимание на синтаксис инициализации словаря
var d = new Dictionary<string, int> {{"One", 1}};
// помещаем элементы, используя индексатор и метод Add()
d["Two"] = 22;
d.Add("Three", 3);
// обновляем существующий элемент
d["Two"] = 2;
```

Класс `OrderedDictionary` – слаботипизированный класс, запоминающий порядок добавления элементов в словарь. В некотором смысле, данный класс является комбинацией классов `Hashtable` и

`ArrayList`. Для доступа к элементам в `OrderedDictionary` можно применять как ключ, так и целочисленный индекс.

Класс `ListDictionary` использует для хранения элементов словаря не хэш-таблицу, а односвязный список. Это делает данный класс неэффективным при работе с большими наборами данных. `ListDictionary` рекомендуется использовать, если количество хранимых элементов не превышает десяти. Класс `HybridDictionary` – это форма словаря, использующая список для хранения при малом количестве элементов, и переключающаяся на применение хэш-функции, когда количество элементов превышает определённый порог. Оба класса `ListDictionary` и `HybridDictionary` являются слаботипизированными и находятся в пространстве имён `System.Collections.Specialized`.

Платформа .NET предоставляет три класса-словаря, организованных так, что их элементы всегда отсортированы по ключу: `SortedDictionary<TKey, TValue>`, `SortedList` и `SortedList<TKey, TValue>`.

Типы для создания пользовательских коллекций

Универсальный класс `Collection<T>` является настраиваемой оболочкой над классом `List<T>`. В дополнение к реализации интерфейсов `IList<T>` и `IList`, класс `Collection<T>` определяет четыре виртуальных метода `ClearItems()`, `InsertItem()`, `RemoveItem()`, `SetItem()` и свойство для чтения `Items`, имеющее тип `IList<T>`. Переопределяя виртуальные методы, можно модифицировать нормальное поведение класса `List<T>` при изменении набора.

Класс `ReadOnlyCollection<T>` – это наследник `Collection<T>`, предоставляющий доступ для чтения элементов, но не для модификации коллекции. Конструктор класса принимает в качестве аргумента объект, реализующий `IList<T>`. Класс не содержит открытых методов добавления или удаления элемента, но можно получить доступ к элементу по индексу и изменить его.

```
public class Track
{
    public string Title { get; set; }
    public uint Length { get; set; }
    public Album Album { get; internal set; }
}
```

```
public class Album : Collection<Track>
{
    protected override void InsertItem(int index, Track item)
    {
        base.InsertItem(index, item);
        item.Album = this;
    }

    protected override void SetItem(int index, Track item)
    {
        base.SetItem(index, item);
        item.Album = this;
    }

    protected override void RemoveItem(int index)
    {
        this[index].Album = null;
        base.RemoveItem(index);
    }

    protected override void ClearItems()
    {
foreach (Track track in this)
        {
            track.Album = null;
        }
        base.ClearItems();
    }
}
```

Класс `ObservableCollection<T>` – это коллекция, позволяющая отслеживать модификации своего набора данных. Этот класс наследуется от `Collection<T>` и реализует интерфейс `INotifyCollectionChanged.`

Абстрактный класс `KeyedCollection<TKey, TItem>` является наследником `Collection<T>`. Этот класс добавляет возможность обращения к элементу по ключу (как в словарях). При использовании `KeyedCollection<TKey, TItem>` требуется переопределить метод `GetKeyForItem()` для вычисления ключа элемента.

14. Технология LINQ to Objects.

Платформа .NET версии 3.5 представила новую технологию работы с коллекциями – *Language Integrated Query* (LINQ). По типу обрабатываемой информации LINQ делится на LINQ to Objects – библиотеки для обработки коллекций объектов в памяти, LINQ to SQL – библиотеки для работы с базами данных, LINQ to XML предназначена для обработки XML-информации. Технически, LINQ to Objects – это набор классов, содержащих типичные методы обработки коллекций: поиск данных, сортировка, фильтрация. Ядром LINQ to Objects является статический класс `Enumerable`, размещённый в пространстве имён System.Linq . Этот класс содержит набор методов расширения интерфейса `IEnumerable`<T>, которые в дальнейшем будут называться *операторами LINQ*.

Группы LINQ операторов:

1. Оператор условия Where (отложенные вычисления).
2. Операторы проекций (отложенные вычисления).
3. Операторы упорядочивания (отложенные вычисления).
4. Оператор группировки GroupBy (отложенные вычисления).
5. Операторы соединения (отложенные вычисления).
6. Операторы работы с множествами (отложенные вычисления).
7. Операторы агрегирования.
8. Операторы генерирования (отложенные вычисления).
9. Операторы кванторов и сравнения.
10. Операторы разбиения (отложенные вычисления).
11. Операторы элемента.
12. Операторы преобразования.

```
public class Student
{
    public string Name { get; set; }
    public int Age { get; set; }
    public IEnumerable<int> Marks { get; set; }
}

var gr = new List<Student>{
new Student {Name = "Smirnov", Age = 18, Marks = new[] {10, 8, 9}},
new Student {Name = "Ivanova", Age = 20, Marks = new[] {5, 6, 9}},
```

```
    new Student {Name = "Kuznetsov", Age = 18, Marks = new[] {7, 7,
4}},
    new Student {Name = "Sokolov", Age = 20, Marks = new[] {7, 8, 8}},
    new Student {Name = "Lebedeva", Age = 20, Marks = new[] {9, 9, 9}}
    };
```

1. Оператор условия Where().

Оператор производит фильтрацию коллекции, основываясь на аргументе-предикате. Сигнатуры оператора:

```
var list = new List<int> {1, 3, -1, -4, 7};
var r1 = list.Where(x => x < 0);
var r2 = gr.Where(student => student.Age > 19);
var r3 = gr.Where((student, pos) => student.Age > 19 && pos < 3);
```

2. Операторы проекций.

Операторы проекций применяются для выборки информации, при этом они могут изменять тип элементов итоговой коллекции. Основным оператором проекции является **Select**():

Оператор **SelectMany**() может применяться, если результатом проекции является набор данных. В этом случае оператор соединяет все элементы набора в одну коллекцию.

```
var r1 = gr.Select(student => student.Name);
var r2 = gr.Select(student => new {student.Name, student.Age});
var r3 = gr.SelectMany(student => student.Marks);
```

Коллекция r1 будет содержать имена студентов. Коллекция r2 состоит из объектов анонимного типа с полями Name и Age. Коллекция r3 – это все оценки студентов (15 элементов типа `int`).

3. Операторы упорядочивания.

Операторы **OrderBy**() и **OrderByDescending**() выполняют сортировку коллекции по возрастанию или убыванию соответственно.

Если после выполнения сортировки по одному ключу требуется дополнительная сортировка по другому ключу, нужно воспользоваться операторами **ThenBy**() и **ThenByDescending**(). Имеется также оператор **Reverse**(), обращающий коллекцию.

```
var r1 = Enumerable.Reverse(gr);
```

```
var r2 = gr.OrderBy(student => student.Age);
var r3 = gr.OrderByDescending(student => student.Age)
           .ThenBy(student => student.Name);
```

4. Оператор группировки GroupBy().

Оператор группировки **GroupBy**() разбивает коллекцию на группы элементов с одинаковым значением некоторого ключа. Оператор **GroupBy**() имеет перегруженные версии, позволяющие указать селектор ключа, преобразователь элементов в группе, объект, реализующий IEqualityComparer<T> для сравнения ключей.

Сгруппируем студентов по возрасту и для каждой группы возрастов выведем ключ и элементы:

```
var r1 = gr.GroupBy(student => student.Age);
foreach (IGrouping<int, Student> group in r1)
{
    Console.WriteLine(group.Key);
    foreach (Student student in group)
    {
        Console.WriteLine(student.Name);
    }
}
```

5. Операторы соединения.

Операторы соединения применяются, когда требуется соединить две коллекции, элементы которых имеют общие атрибуты. Основным оператором соединения является оператор **Join**().

Оператор **GroupJoin**() порождает набор, группируя элементы внутренней коллекции при нахождении соответствия с элементом внешней коллекции: Если же соответствие не найдено, в результат включается пустая группа.

Оператор **Zip**() порождает набор на основе двух исходных коллекций, выполняя заданное генерирование элементов. Длина результирующей коллекции равна длине меньшей из двух исходных коллекций.

6. Операторы работы с множествами.

В LINQ to Objects имеется набор операторов для работы с множествами.

Оператор **Distinct**() удаляет из коллекции повторяющиеся элементы.

Операторы **Union**(), **Intersect**() и **Except**() представляют объединение, пересечение и разность двух множеств.

7. Операторы агрегирования.

К операторам агрегирования относятся операторы, результатом работы которых является скалярное значение. Следующие операторы возвращают количество элементов коллекции. При этом может быть использована перегруженная версия, принимающая в качестве второго аргумента предикат фильтрации.

Операторы: **Sum**() - находит сумму по предикату, **Average**() - вычисляет среднее значение по предикату, **Min**() - ищет минимальный элемент по предикату, **Max**() - ищет максимальный соответственно, ...

8. Операторы генерирования.

Эта группа операторов позволяет создать набор данных. Первый оператор группы – оператор **Range**(). Он просто выдаёт указанное количество подряд идущих целых чисел, начиная с заданного значения.

Продемонстрируем использование **Range**() в задаче поиска простых чисел, которую решим при помощи LINQ:

Следующий оператор генерирования – оператор **Repeat**(). Он создаёт коллекцию, в которой указанный элемент повторяется требуемое число раз. Для ссылочных типов дублируются ссылки, а не содержимое.

Последним генерирующим оператором является оператор **Empty**(), который порождает пустое перечисление определённого типа.

9. Операторы кванторов и сравнения.

Операторы кванторов похожи на соответствующие операторы в математической логике.

Оператор **Any**() проверяет наличие хотя бы одного элемента в коллекции, удовлетворяющего указанному предикату. Вторая версия оператора **Any**() просто проверяет коллекцию на непустоту. Оператор **All**() возвращает `true`, если все элементы коллекции удовлетворяют предикату. И, наконец, оператор **Contains**() проверяет, входит ли заданное значение в коллекцию.

Оператор **SequenceEqual**() сравнивает две коллекции и возвращает `true`, если обе коллекции имеют одну длину, и их соответствующие элементы равны:

10. Операторы разбиения.

Операторы разбиения выделяют некую часть исходной коллекции (например, первые десять элементов).

Комбинация операторов **Take**() и **Skip**() часто применяется, чтобы организовать постраничный просмотр информации (например, просмотр

записей из большой таблицы):

```
var bigTable = Enumerable.Range(1, 1000);
int pageSize = 20, pageNumber = 6;
var page = bigTable.Skip((pageNumber - 1)*pageSize).Take(pageSize);
```

11. Операторы элемента.

Эта группа операторов предназначена для выделения из коллекции единственного элемента, удовлетворяющего определённым условиям.
Оператор **First**() выделяет первый элемент (или первый элемент, удовлетворяющий определённому предикату).
Если в коллекции нет элементов, или не нашлось элементов, удовлетворяющих предикату, оператор **First**() выбрасывает исключение `InvalidOperationException`. Если требуется, чтобы исключение не выбрасывалось, а возвращалось предопределённое значение для типа, следует использовать оператор **FirstOrDefault**().
Аналогично работают операторы **Last**() и **LastOrDefault**() для выделения последнего элемента.
Операторы **Single**() и **SingleOrDefault**() рассчитаны на то, что коллекция (или набор элементов, удовлетворяющих предикату) будет состоять из одного элемента, который данные операторы и возвращают. Если в коллекции нет элементов, или их оказалось больше одного, оператор **Single**() выбрасывает исключение `InvalidOperationException`. Оператор **SingleOrDefault**() выбрасывает такое исключение, если элементов оказалось больше одного.
Пара операторов **ElementAt**() и **ElementAtOrDefault**() пытаются вернуть элемент на указанной целочисленной позиции.
Оператор **DefaultIfEmpty**() проверяет коллекцию на пустоту. Если в коллекции нет элементов, то возвращается или значение по умолчанию для типа, или указанное значение. Если коллекция непустая, то она и возвращается.

12. Операторы преобразования.

Назначение данных операторов – преобразования универсальных коллекций, реализующих `IEnumerable`<T>, в конкретные типы.
Оператор **AsEnumerable**() возвращает свой аргумент, приведённый к типу `IEnumerable`<T>.
Операторы **ToArray**() и **ToList**() выполняют преобразование коллекции в

массив или список.
Существует несколько версий оператора **ToDictionary**(), порождающего из коллекции словарь.
Оператор **ToLookup**() преобразовывает коллекцию в объект класса `Lookup`<K, V> из пространства имён System.Linq. Класс `Lookup`<K, V> представляет словарь, в котором ключу сопоставлен не единственный элемент, а набор элементов.
Оператор **OfType**<T>() итерируется по коллекции и генерирует список, содержащий только элементы заданного типа T.
Оператор **Cast**<T>() итерируется по слаботипизированной коллекции и пытается выполнить приведение каждого элемента к указанному типу.

Язык C# версии 3.0 вводит новые ключевые слова и особые синтаксические расширения для записи некоторых операторов LINQ в виде выражений запросов. Составленное выражение запросов должно подчиняться следующим правилам
1. Выражение должно начинаться с конструкции `from`, которая указывает на обрабатываемую коллекцию.
2. Затем выражение может содержать ноль или более конструкций `from`, `let` или `where`. Конструкция `let` представляет переменную и присваивает ей значение. Конструкция `where` фильтрует элементы коллекции.
3. Затем выражение может включать ноль или более конструкций `orderby`, с полями сортировки и необязательным указанием на направление упорядочивания. Направление может быть `ascending` или `descending`.
4. Затем следует конструкция `select` или `group`.
5. Наконец, в оставшейся части выражения может следовать необязательная конструкция продолжения. Такой конструкцией является `into`.

15. Работа с файлами и потоками данных на платформе .NET.

Платформа .NET включает несколько классов для работы с объектами файловой системы – дисками, каталогами, файлами. Они доступны в пространстве имён `System.IO`.

Класс `DriveInfo` инкапсулирует информацию о диске. Он имеет статический метод `GetDrives()` для получения массива объектов `DriveInfo`, соответствующих дискам операционной системы.

Классы `Directory`, `File`, `DirectoryInfo` и `FileInfo` предназначены для работы с каталогами и файлами. Первые два класса выполняют операции при помощи статических методов, вторые два – при помощи экземплярных методов.

Рассмотрим работу с классами `DirectoryInfo` и `FileInfo`. Данные классы являются наследниками абстрактного класса `FileSystemInfo`. Этот класс содержит следующие основные элементы, перечисленные в таблице:

Имя элемента	Описание
`Attributes`	Свойство позволяет получить или установить атрибуты объекта файловой системы (тип – перечисление `FileAttributes`)
`CreationTime`	Время создания объекта файловой системы
`Exists`	Свойство для чтения, проверка существования объекта файловой системы
`Extension`	Свойство для чтения, расширение файла
`FullName`	Свойство для чтения, полное имя объекта файловой системы
`LastAccessTime`, `LastAccessTimeUtc`	Время последнего доступа к объекту файловой системы (локальное или всемирное координированное)
`LastWriteTime`, `LastWriteTimeUtc`	Времени последней записи для объекта файловой системы (локальное или всемирное координированное)
`Name`	Свойство для чтения; имя файла или каталога
`Delete()`	Метод удаляет объект файловой системы
`Refresh()`	Метод обновляет информацию об объекте файловой системы

Конструктор класса `DirectoryInfo` принимает в качестве параметра строку с именем того каталога, с которым будет производиться работа. Для указания текущего каталога используется строка `"."`. При попытке работать с данными несуществующего каталога генерируется исключение.

Класс `DirectoryInfo` обладает двумя наборам методов для получения дочерних подкаталогов, файлов, или объектов `FileSystemInfo`. Методы вида `Get`*`Элементы`*`()` выполняются немедленно и возвращают массив. Методы вида `Enumerate`*`Элементы`*`()` используют отложенное выполнение и возвращают перечислитель:

```
// получаем файлы, удовлетворяющие маске, из всех подкаталогов
FileInfo[] f = dir.GetFiles("*.txt", SearchOption.AllDirectories);
// получаем файлы, используя отложенное выполнение
foreach (var fileInfo in dir.EnumerateFiles())
  Console.WriteLine(fileInfo.Name);
```

Класс `FileInfo` описывает файл на диске и позволяет производить операции с этим файлом. Наиболее важные элементы класса представлены в таблице.

Имя элемента	Описание
`AppendText()`	Создает объект `StreamWriter` для добавления текста к файлу
`CopyTo()`	Копирует существующий файл в новый файл
`Create()`	Создает файл и возвращает объект `FileStream` для работы
`CreateText()`	Создает объект `StreamWriter` для записи текста в новый файл
`Decrypt()`	Дешифрует файл, зашифрованный методом `Encrypt()`
`Directory`	Свойство для чтения, каталог файла
`DirectoryName`	Свойство для чтения, полный путь к файлу
`Encrypt()`	Шифрует файл с учётом системных данных текущего пользователя
`IsReadOnly`	Булево свойство - указывает, является ли файл файлом только для чтения
`Length`	Свойство для чтения, размер файла в байтах
`MoveTo()`	Перемещает файл (возможно, с переименованием)
`Open()`	Открывает файл с указанными правами доступа
`OpenRead()`	Создает объект `FileStream`, доступный только для чтения
`OpenText()`	Создает объект `StreamReader` для чтения информации из текстового файла
`OpenWrite()`	Создает объект `FileStream`, доступный для чтения и записи

Как правило, код, работающий с данными файла, вначале вызывает метод `Open()`. Рассмотрим перегруженную версию метода `Open()`, которая принимает три параметра. Первый параметр определяет режим запроса на открытие файла. Для него используются значения из перечисления `FileMode`:

- `Append` – открывает файл, если он существует, и ищет конец файла. Если файл не существует, то он создается. Этот режим может использоваться только с доступом `FileAccess.Write`;
- `Create` – указывает на создание нового файла. Если файл существует, он будет перезаписан;
- `CreateNew` – указывает на создание нового файла. Если файл существует, генерирует исключение `IOException`;
- `Open` – операционная система должна открыть существующий файл;
- `OpenOrCreate` – операционная система должна открыть существующий файл или создать новый, если файл не существует;
- `Truncate` – система должна открыть существующий файл и обрезать его до нулевой длины.

Второй параметр метода `Open()` определяет тип доступа к данным файла. Для него используются элементы перечисления `FileAccess`:

- `Read` – файл будет открыт только для чтения;
- `ReadWrite` – файл будет открыт и для чтения, и для записи;
- `Write` – файл открывается только для записи, то есть добавления данных.

Третий параметр задаёт возможность совместной работы с открытым файлом и представлен значениями перечисления `FileShare`:

- `None` – совместное использование запрещено, на любой запрос на открытие файла будет возвращено сообщение об ошибке;
- `Read` – файл могут открыть и другие пользователи, но только для чтения;
- `ReadWrite` – другие пользователи могут открыть файл и для чтения, и для записи;
- `Write` – файл может быть открыт другими пользователями для записи.

Платформа .NET содержит развитый набор типов для поддержки операций ввода/вывода информации. Типы для поддержки ввода/вывода можно разбить на две категории: типы для представления потоков данных и адаптеры потоков. Поток данных – это абстрактное представление данных в виде последовательности байт. Поток либо ассоциируется с неким физическим хранилищем (файлами на диске, памятью, сетью), либо

декорирует (обрамляет) другой поток, преобразуя данные тем или иным образом. Адаптеры потоков служат оболочкой потока, преобразуя информацию определённого формата в набор байт. Резюмируя вышесказанное, представим классы для работы с потоками в виде следующих категорий:

1. Абстрактный класс `System.IO.Stream` - базовый класс для других классов, представляющих потоки.
2. Классы для работы с потоками, связанными с хранилищами.
   - `FileStream` – класс для работы с файлами, как с потоками (пространство имён `System.IO`).
   - `MemoryStream` – класс для представления потока в памяти (пространство имён `System.IO`).
   - `NetworkStream` – работа с сокетами, как с потокам (пространство имён `System.Net.Sockets`).
   - `PipeStream` - абстрактный класс из пространства имён `System.IO.Pipes`, базовый для классов-потоков, которые позволяют передавать данные между процессами системы.
3. Декораторы потоков.
   - `DeflateStream` и `GZipStream` – классы (пространство имён `System.IO.Compression`) для потоков со сжатием данных.
   - `CryptoStream` – поток зашифрованных данных (пространство имён `System.Security.Cryptography`).
   - `BufferedStream` – поток с поддержкой буферизации данных (пространство имён `System.IO`).
4. Адаптеры потоков.
   - `BinaryReader` и `BinaryWriter` – классы для ввода/вывода примитивных типов в двоичном формате.
   - `StreamReader` и `StreamWriter` – классы для ввода/вывода информации в строковом представлении.
   - `XmlReader` и `XmlWriter` – абстрактные классы для ввода/вывода XML.

Элементы абстрактного класса `Stream`

Категория	**Элементы**
Чтение данных	`bool CanRead { get; }`

	`IAsyncResult BeginRead(byte[] buffer, int offset, int count, AsyncCallback callback, object state)`
	`int EndRead(IAsyncResult asyncResult)`
	`int Read(byte[] buffer, int offset, int count)`
	`int ReadByte()`
Запись данных	`bool CanWrite { get; }`
	`IAsyncResult BeginWrite(byte[] buffer, int offset, int count, AsyncCallback callback, object state)`
	`int EndWrite(IAsyncResult asyncResult)`
	`void Write(byte[] buffer, int offset, int count)`
	`void WriteByte(byte value)`
	`void CopyTo(Stream destination)`
Перемещение	`bool CanSeek { get; }`
	`long Position { get; set; }`
	`void SetLength(long value)`
	`long Length { get; }`
	`long Seek(long offset, SeekOrigin origin)`
Закрытие потока	`void Close()`
	`void Dispose()`
	`void Flush()`
Таймауты	`bool CanTimeout { get; }`
	`int ReadTimeout { get; set; }`
	`int WriteTimeout { get; set; }`
Другие элементы	`static readonly Stream Null`
	`static Stream Synchronized(Stream stream)`

```
    // записываем 100.000 байт в файл
    File.WriteAllBytes("myFile.bin", new byte[100000]);
    using (FileStream fs = File.OpenRead("myFile.bin"))// читаем, используя буфер
    {
        using (BufferedStream bs = new BufferedStream(fs, 20000))
        {
            bs.ReadByte(); Console.WriteLine(fs.Position); // 20000
        }
    }
```

16. Использование XML на платформе .NET.

Расширяемый язык разметки **(eXtensible Markup Language, XML)** – это способ описания структурированных данных. *Структурированными данными*

называются такие данные, которые обладают заданным набором семантических атрибутов и допускают иерархическое описание. XML-данные содержатся в *документе*, в роли которого может выступать файл, поток или другое хранилище информации, способное поддерживать текстовый формат.

LINQ TO XML

Технология LINQ to XML предоставляет программный интерфейс для работы с XML-документами, описываемыми в виде дерева объектов. Этот программный интерфейс обеспечивает создание, редактирование и трансформацию XML, при этом возможно применение LINQ-подобного синтаксиса.

LINQ to XML содержит набор классов, сосредоточенных в пространстве имён `System.Xml.Linq` (рис. 6):

1. Абстрактные классы `XObject`, `XNode` и `XContainer` служат основой для иерархии классов, соответствующих различным объектам XML.
2. Классы `XElement` и `XDocument` представляют XML-элемент и XML-документ соответственно.
3. Класс `XAttribute` служит для описания XML-атрибута.
4. Класс `XText` представляет текст в XML-элементе.
5. Класс `XName` представляет имя атрибута или элемента.
6. Классы `XDeclaration`, `XDocumentType`, `XProcessingInstruction`, `XComment` описывают соответствующие части XML-документа.
7. Статический класс `Extensions` содержит методы расширения для выполнения запросов к XML-данным.

Центральную роль в работе с XML-данными играют классы `XElement`, `XDocument` и `XAttribute`. Для создания отдельного XML-элемента обычно используется один из конструкторов класса `XElement`:

```
public XElement(XElement other);
public XElement(XName name);
public XElement(XStreamingElement other);
public XElement(XName name, object content);
public XElement(XName name, params object[] content);
```

Ниже приведены различные варианты вызова конструктора `XElement`:

```
var e1 = new XElement("name", "Earth");
// <name>Earth</name>
var e2 = new XElement("planet", e1);
// <planet>
```

```
//   <name>Earth</name>
// </planet>
var e3 = new XElement("period", new XAttribute("units", "days"));
// <period units="days" />
var e4 = new XElement("comment", new XComment("the comment"));
// <comment>
//   <!--the comment-->
// </comment>
var e5 = new XElement("list", new List<object> {"text",
                                  new XElement("name", "Mars")});
// <list>
//   text<name>Mars</name>
// </list>
var e6 = new XElement("moon", null);
// <moon />
var e7 = new XElement("date", DateTime.Now);
// <date>2014-01-19T11:04:54.625+02:00</date>
```

Пятая версия конструктора `XElement` подобна четвёртой, но позволяет предоставить множество объектов для содержимого:

```
var p = new XElement("planets",
              new XElement("planet", new XElement("name",
"Mercury")),
              new XElement("planet", new XElement("name", "Venus")),
              new XElement("planet", new XElement("name", "Earth"),
                  new XElement("moon",
                      new XElement("name", "Moon"),
                      new XElement("period", 27.321582,
                                  new XAttribute("units",
"days")))));
Console.WriteLine(p);    // выводим первые три планеты в виде XML
```

Использование конструктора `XAttribute` для создания XML-атрибута очевидно из приведённых выше примеров. Конструкторы класса `XDocument` позволяют указать декларацию XML-документа и набор объектов содержимого. В этот набор могут входить комментарии, инструкции по обработке и не более одного XML-элемента:

```
var d = new XDeclaration("1.0", "utf-8", "yes");
// используем элемент p из предыдущего примера
var doc = new XDocument(d, new XComment("первые три планеты"), p);
Console.WriteLine(doc.Declaration);    // печатаем декларацию
```

```
Console.WriteLine(doc);                    // печатаем документ
```

Кроме применения конструкторов, объекты XML можно создать из строкового представления при помощи статических методов `XElement.Parse()` и `XDocument.Parse()`:

```
var planet = XElement.Parse("<name>Earth</name>");
```

Для сохранения элемента или XML-документа используется метод `Save()`, имеющийся у `XElement` и `XDocument`. Данный метод перегружен и позволяет выполнить запись в текстовый файл или с применением адаптеров `TextWriter` и `XmlWriter`. Кроме этого, можно указать опции сохранения (например, отключить автоматическое формирование отступов элементов).

```
doc.Save("planets.xml", SaveOptions.None);
```

Загрузка элемента или XML-документа выполняется статическими методами `XElement.Load()` или `XDocument.Load()`. Метод `Load()` перегружен и позволяет выполнить загрузку из файла, произвольного URI, а также с применением адаптеров `TextReader` и `XmlReader`. Можно задать опции загрузки (например, связать с элементами XML номер строки в исходном тексте).

```
var d1 = XDocument.Load("planets.xml", LoadOptions.SetLineInfo);
var d2 = XElement.Load("http://habrahabr.ru/rss/");
```

Рассмотрим набор методов и свойств, используемых в LINQ to XML при выборке данных. Класс `XObject` имеет свойство `NodeType` для типа XML-узла и свойство `Parent`, указывающее, какому элементу принадлежит узел.

Методы классов `XNode` и `XContainer` позволяют получить у элемента наборы предков и дочерних узлов (элементов). При этом возможно указание фильтра – имени элемента. Большинство методов возвращают коллекции, реализующие `IEnumerable`[7].

```
// методы выборки у XNode
public IEnumerable<XElement> Ancestors();              // + XName name
public IEnumerable<XElement> ElementsAfterSelf();      // + XName name
public IEnumerable<XElement> ElementsBeforeSelf();     // + XName name
public bool IsAfter(XNode node);
public bool IsBefore(XNode node);
```

[7] Запись `+ XName name` означает наличие перегруженной версии метода с параметром `name` типа `XName`.

```
    public IEnumerable<XNode> NodesAfterSelf();
    public IEnumerable<XNode> NodesBeforeSelf();

    // методы выборки у XContainer
    public IEnumerable<XNode> DescendantNodes();
    public IEnumerable<XElement> Descendants();          // + XName
name
    public XElement Element(XName name);
    public IEnumerable<XElement> Elements();             // + XName
name
```

Класс `XDocument` позволяет получить корневой элемент при помощи свойства `Root`. В классе `XElement` есть методы для запроса элементов-предков и элементов-потомков, а также методы для запроса атрибутов.

```
    // методы выборки у XElement
    public IEnumerable<XElement> AncestorsAndSelf();     // + XName
name
    public XAttribute Attribute(XName name);
    public IEnumerable<XAttribute> Attributes();         // + XName
name
    public IEnumerable<XNode> DescendantNodesAndSelf();
    public IEnumerable<XElement> DescendantsAndSelf();   // + XName
name
```

Статический класс `Extensions` определяет несколько методов расширения, работающих с коллекциями элементов или узлов XML:

```
    IEnumerable<XElement> Ancestors<T>(...) where T: XNode;
    IEnumerable<XElement> AncestorsAndSelf(...);
    IEnumerable<XAttribute> Attributes(...);
    IEnumerable<XNode> DescendantNodes<T>(...) where T: XContainer;
    IEnumerable<XNode> DescendantNodesAndSelf(...);
    IEnumerable<XElement> Descendants<T>(...) where T: XContainer;
    IEnumerable<XElement> DescendantsAndSelf(...);
    IEnumerable<XElement> Elements<T>(...) where T: XContainer;
    IEnumerable<T> InDocumentOrder<T>(...) where T: XNode;
    IEnumerable<XNode> Nodes<T>(...) where T: XContainer;
    void Remove(this IEnumerable<XAttribute> source);
    void Remove<T>(this IEnumerable<T> source) where T: XNode;
```

Продемонстрируем примеры запросов, используя файл `planets.xml` с описанием четырёх планет. Загрузим файл и выведем имена планет:

```
    var xml = XDocument.Load("planets.xml");
```

```
    var query = xml.Root.Elements("planet")
                        .Select(planet =>
planet.Element("name").Value);
    foreach (string name in query)
        Console.WriteLine(name);
```

Для описания пространства имён XML в LINQ to XML используется класс `XNamespace`. У этого класса нет открытого конструктора, но определено неявное приведение строки к `XNamespace`:

```
XNamespace ns = "http://astronomy.com/planet";
```

Чтобы указать на принадлежность имени к определённому пространству имён, следует использовать перегруженную версию оператора `+`, объединяющую объект `XNamespace` и строку в результирующий объект `XName`:

```
XElement jupiter = new XElement(ns + "name", "Jupiter");
// <name xmlns="http://astronomy.com/planet">Jupiter</name>
```

В дополнение к LINQ to XML, платформа .NET содержит несколько программных интерфейсов для работы с XML. Для этого обычно используются классы из пространств имён вида `System.Xml.*` (сборка `System.Xml.dll`).

Классы `XmlReader` и `XmlWriter` – это основа механизма последовательного чтения, обработки и записи XML-документов. Такой подход выгодно использовать, когда документ слишком велик, чтобы читать его в память целиком, или содержит ошибки в структуре.

Для чтения XML-документов применяется класс `XmlReader` и его наследники `XmlTextReader` (чтение на основе текстового потока), `XmlNodeReader` (разбор XML из объектов `XmlNode`) и `XmlValidatingReader` (чтение с проверкой схемы XML-документа). Класс `XmlReader` содержит статический метод `Create()`, создающий объект для чтения на основе `Stream`, `TextReader` или строки URI (частный случай URI – имя файла):

```
XmlReader reader = XmlReader.Create("planets.xml");
```

Метод `Create()` принимает в качестве дополнительного аргумента объект класса `XmlReaderSettings`, который задаёт различные опции чтения данных:

Следующий пример демонстрирует разбор XML-файла и печать разобранных конструкций:

```
using (var r = XmlReader.Create("planets.xml"))
{
    while (r.Read())
    {
        Console.Write("{0}\t", r.Depth);
        switch (r.NodeType)
        {
            case XmlNodeType.Element:
            case XmlNodeType.EndElement:
                Console.WriteLine(r.Name);
                break;
            case XmlNodeType.Text:
            case XmlNodeType.Comment:
            case XmlNodeType.XmlDeclaration:
                Console.WriteLine(r.Value);
                break;
        }
    }
}
```

Для чтения атрибутов текущего элемента можно использовать индексатор `XmlReader`, указав имя или позицию атрибута (если атрибута не существует, индексатор вернёт значение `null`):

```
// переписанный фрагмент оператора switch из предыдущего примера
switch (r.NodeType)
{
    case XmlNodeType.Element:
        Console.WriteLine(r.Name);
        string attribute = r["units"];
        Console.WriteLine(attribute ?? string.Empty);
        break;
    . . .
}
```

Класс `XmlWriter` – это абстрактный класс для создания XML-данных. Подчеркнём, что XML-данные всегда могут быть сформированы в виде простой строки и затем записаны в любой поток. Однако такой подход не лишён недостатков: возрастает вероятность неправильного формирования структуры XML из-за элементарных ошибок. Класс `XmlWriter` и его наследники (например, `XmlTextWriter`) предоставляют более «помехоустойчивый» способ генерации XML-документа.

Приведём пример работы с классом XmlWriter.

```
// опция для формирования отступов в документе
var settings = new XmlWriterSettings {Indent = true };
using (var writer = XmlWriter.Create("customers.xml", settings))
{
    // начинаем с XML-декларации
    writer.WriteStartDocument();

    // открывающий тег с двумя атрибутами
    writer.WriteStartElement("customer");
    writer.WriteAttributeString("id", "1");
    writer.WriteAttributeString("status", "archived");

    // вложенный элемент со строковым содержимым
    writer.WriteElementString("name", "Alex");

    // так пишутся элементы с нестроковым содержимым
    // для этого используется метод WriteValue()
    writer.WriteStartElement("birthdate");
    writer.WriteValue(new DateTime(1975, 8, 4));
    writer.WriteEndElement();

    // закрывающие теги (принцип стека)
    writer.WriteEndElement();
    writer.WriteEndDocument();
}
```

Этот пример формирует следующий XML-документ:

```
<?xml version="1.0" encoding="utf-8"?>
<customer id="1" status="archived">
  <name>Alex</name>
  <birthdate>1975-08-04T00:00:00</birthdate>
</customer>
```

Классы XmlNode, XmlAttribute, XmlElement, XmlDocument служат для представления XML-документа в виде дерева объектов. Программный интерфейс, основанный на использовании данных классов, являлся предшественником LINQ to XML. В связи с этим ограничимся только простым примером, демонстрирующим работу с указанными классами:

```
public static void OutputNode(XmlNode node)
{
```

```
        Console.WriteLine("Type= {0} \t Name= {1} \t Value= {2}",
                          node.NodeType, node.Name, node.Value);
        if (node.Attributes != null)
        {
            foreach (XmlAttribute attr in node.Attributes)
            {
                Console.WriteLine("Type={0} \t Name={1} \t Value={2}",
                                  attr.NodeType, attr.Name,
attr.Value);
            }
        }
        // если есть дочерние элементы, рекурсивно обрабатываем их
        if (node.HasChildNodes)
        {
            foreach (XmlNode child in node.ChildNodes)
            {
                OutputNode(child);
            }
        }
    }

    // пример использования метода OutputNode()
    var doc = new XmlDocument();
    doc.Load("planets.xml");
    OutputNode(doc.DocumentElement);
```

17. Состав и взаимодействие сборок на платформе .NET.

В платформе .NET *сборка* (*assembly*) – это единица развёртывания и контроля версий. Сборка состоит из одного или нескольких *программных модулей* и, возможно, *данных ресурсов*. Эти компоненты могут размещаться в отдельных файлах, либо содержаться в одном файле. В любом случае, сборка содержит в некотором из своих файлов *манифест*, описывающий состав сборки. Будем называть сборку *однофайловой*, если она состоит из одного файла. В противном случае сборку будем называть *многофайловой*. Тот файл, который содержит манифест сборки, будем называть *главным файлом сборки*.

Рис. 3. Однофайловая и многофайловая сборки.

Простые приложения обычно представлены однофайловыми сборками. При разработке сложных приложений переход к многофайловым сборкам даёт следующие преимущества:

1. Ресурсы (текстовые строки, изображения и т. д.) можно хранить вне приложения, что позволяет при необходимости изменять ресурсы без перекомпиляции приложения.
2. Если исполняемый код приложения разделён на несколько модулей, то модули загружаются в память только по мере надобности.
3. Скомпилированный модуль может использоваться в нескольких сборках.

Платформа .NET разделяет сборки на *локальные* (или *сборки со слабыми именами*) и *глобальные* (или *сборки с сильными именами*). Если `UL.dll` рассматривается как локальная сборка, то при выполнении приложения она должна находиться в том же каталоге, что и `main.exe`. Локальные сборки обеспечивают простоту развёртывания приложения (все его компоненты сосредоточены в одном месте) и изолированность компонентов. Имя локальной сборки – *слабое имя* – это имя файла сборки без расширения.

Хотя использование локальных сборок имеет свои преимущества, иногда необходимо сделать сборку общедоступной. До появления платформы .NET доминировал подход, при котором код общих библиотек помещался в системный каталог простым копированием фалов при установке. Такой подход привел к проблеме, известной как «*ад DLL*» (*DLL Hell*). Инсталлируемое приложение могло заменить общую библиотеку новой версией, при этом другие приложения, ориентированные на старую версию библиотеки, переставали работать. Для устранения «ада DLL» в платформе .NET используется специальное защищенное *хранилище сборок* (*Global Assembly Cache*, *GAC*).

Сборки, помещаемые в GAC, должны удовлетворять определённым условиям. Во-первых, такие глобальные сборки должны иметь цифровую подпись. Это исключает подмену сборок злоумышленниками. Во-вторых, для глобальных сборок отслеживаются версии и языковые культуры. Допустимой является ситуация, когда в GAC находятся разные версии одной и той же сборки, используемые разными приложениями.

Сборка, помещенная в GAC, получает *сильное имя*. Как раз использование сильного имени является тем признаком, по которому среда исполнения понимает, что речь идет не о локальной сборке, а о сборке из GAC. Сильное имя включает: имя главного файла сборки (без расширения), версию сборки, указание о региональной принадлежности сборки и маркер открытого ключа сборки:

```
 Name, Version=1.2.0.0, Culture=neutral,
PublicKeyToken=1234567812345678
```

18. Метаданные и информация о типах. Технология «отражения».

При создании сборки в неё помещаются метаданные, которые являются описанием всех типов в сборке и их элементов. Программист может работать с метаданными, используя специальный механизм, называемый *отражением* (reflection). Главные элементы, которые необходимы для использования возможностей отражения – это класс `System.Type` и типы из пространств имён `System.Reflection` и `System.Reflection.Emit`.

Класс `Type` служит для хранения информации о типе. Существует несколько способов получить объект этого класса:

1. Вызвать у объекта метод `GetType()`. Данный метод определён на уровне `System.Object`, а значит, присутствует у любого объекта:

```
Foo foo = new Foo();          // Foo – это некий класс
Type t = foo.GetType();
```

2. Использовать статический метод `Type.GetType()`, которому передаётся имя типа в виде строки. Это имя должно быть полным, то есть включать пространство имён и, при необходимости, имя сборки, содержащей тип:

```
Type t1 = Type.GetType("System.Int32");
Type t2 = Type.GetType("SomeNamespace.Foo, MyAssembly");
```

3. Использовать операцию C# `typeof`, аргументом которой является тип:

```
Type t = typeof (Foo);
```

Операцию `typeof` можно применять к массивам и универсальным шаблонам. Причём в последнем случае допускается использовать как сконструированный тип, так и исходный тип-шаблон (обратите внимание на синтаксис записи универсального шаблона).

```
Type t1 = typeof (int[]);
Type t2 = typeof (char[,]);
Type t3 = typeof (List<int>);    // сконструированный тип
Type t4 = typeof (List<>);       // универсальный тип
```

Свойства класса Type позволяют узнать имя типа, имя базового типа, является ли тип универсальным, в какой сборке он размещается и другую информацию. Кроме этого, Type имеет специальные методы, возвращающие данные о полях типа, свойствах, событиях, методах и их параметрах.

Рассмотрим пример получения информации о типе. Будем анализировать примитивный тип `System.Int32`:

```
Type t = typeof (Int32);
Console.WriteLine("Full name = " + t.FullName);
Console.WriteLine("Base type is = " + t.BaseType);
Console.WriteLine("Is sealed = " + t.IsSealed);
Console.WriteLine("Is class = " + t.IsClass);
foreach (Type iType in t.GetInterfaces())
{
    Console.WriteLine(iType.Name);
}
foreach (FieldInfo fi in t.GetFields())
{
    Console.WriteLine("Field = " + fi.Name);
}
foreach (PropertyInfo pi in t.GetProperties())
{
    Console.WriteLine("Property = " + pi.Name);
}
foreach (MethodInfo mi in t.GetMethods())
{
    Console.WriteLine("Method Name = " + mi.Name);
    Console.WriteLine("Method Return Type = " + mi.ReturnType);
    foreach (ParameterInfo pr in mi.GetParameters())
    {
        Console.WriteLine("Parameter Name = " + pr.Name);
        Console.WriteLine("Type = " + pr.ParameterType);
```

```
        }
    }
```

Пространство имён `System.Reflection` содержит типы для получения информации и манипулирования сборкой и модулем сборки. При помощи класса Assembly можно получить информацию о сборке, при помощи класса Module – о модуле. Основные элементы этих классов перечислены в табл. 17 и табл. 18.

```
Assembly assembly = Assembly.GetExecutingAssembly();
Console.WriteLine(assembly.FullName);
foreach (Module module in assembly.GetModules())
{
    Console.WriteLine(module.FullyQualifiedName);
    foreach (Type type in module.GetTypes())
    {
        Console.WriteLine(type.FullName);
    }
}
```

19. Многопоточное программирование. Синхронизации потоков выполнения (на примере платформы .NET).

Основные классы, предназначенные для поддержки многопоточности, сосредоточены в пространстве имен `System.Threading`. На платформе .NET каждый *поток выполнения* (*thread*) представлен объектом класса `Thread`. Для организации собственного потока необходимо создать объект этого класса. Класс `Thread` имеет четыре перегруженных версии конструктора:

```
public Thread(ThreadStart start);
public Thread(ThreadStart start, int maxStackSize);
public Thread(ParameterizedThreadStart start);
public Thread(ParameterizedThreadStart start, int maxStackSize);
```

В качестве первого параметра конструктору передаётся делегат, инкапсулирующий метод, выполняемый в потоке. Доступно два типа делегатов: второй позволяет при запуске метода передать ему параметр-объект:

```
public delegate void ThreadStart();
public delegate void ParameterizedThreadStart(object obj);
```

Дополнительный параметр конструктора может использоваться для указания максимального размера стека, выделяемого потоку (по умолчанию

каждый поток резервирует около 1 мегабайта).

Для запуска потока требуется вызвать метод `Start()` (перегруженная версия метода получает объект, передаваемый методу потока как параметр).

```
var th = new Thread(DoSomeWork);
th.Start();
```

Рассмотрим основные свойства класса `Thread`:

- Статическое свойство `CurrentThread` возвращает объект, представляющий текущий поток.
- Свойство `Name` служит для назначения потоку имени.
- Целочисленное свойство для чтения `ManagedThreadId` возвращает уникальный числовой идентификатор управляемого потока.
- Свойство для чтения `ThreadState`, значением которого являются элементы одноимённого перечисления, позволяет получить текущее состояние потока.
- Булево свойство для чтения `IsAlive` позволяет определить, выполняется ли поток.
- Свойство `Priority` управляет *приоритетом* выполнения потока относительно текущего процесса. Значением этого свойства являются элементы перечисления `ThreadPriority`: `Lowest`, `BelowNormal`, `Normal`, `AboveNormal`, `Highest`.
- Булево свойство `IsBackground` позволяет сделать поток фоновым. Среда исполнения .NET разделяет все потоки на *фоновые* и *основные*. Процесс не может завершиться, пока не завершены все его основные потоки. В то же время, завершение процесса автоматически останавливает все фоновые потоки, при этом не гарантируется выполнение их блоков `finally`.
- Свойства `CurrentCulture` и `CurrentUICulture` имеют тип `CultureInfo` и задают текущую языковую культуру.

Кроме свойств, класс `Thread` содержит методы для управления потоком. Статический метод `Sleep()` приостанавливает выполнение текущего потока (можно указать количество миллисекунд или значение `TimeSpan`). Статический метод `Yield()` указывает на необходимость передать управление следующему ожидающему потоку ОС. Метод `Join()` позволяет дождаться завершения работы того потока, у которого вызывается. Модификация данного метода блокирует выполнение текущего потока на

указанное количество времени.

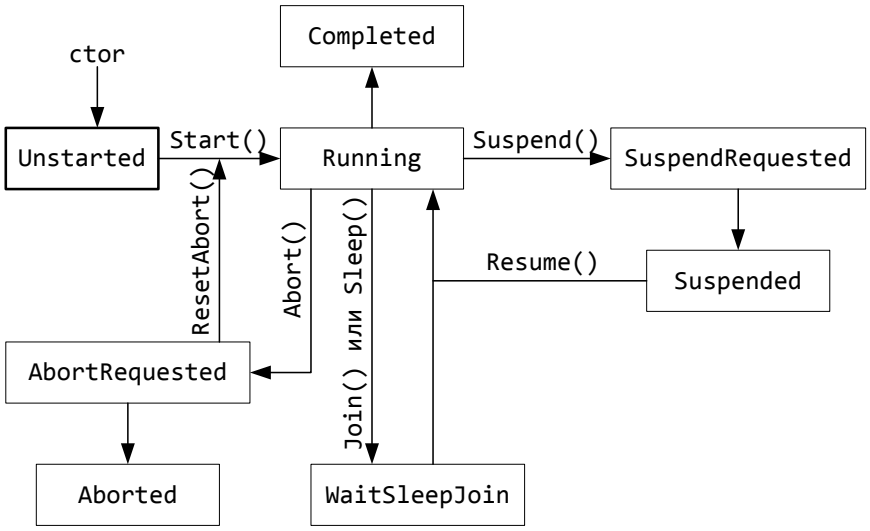
```
var th = new Thread(DoSomeWork);
th.Start();    // создали и запустили поток
th.Join();     // ждем, пока поток отработает
var th_2 = new Thread(DoSomeWork);
th_2.Start();
// будем ждать 1 секунду. Если за это время поток th_2
// завершиться, то значение res будет равно true
bool res = th_2.Join(1000);
```

Для завершения работы выбранного потока используется метод `Abort()`. Данный метод генерирует специальное исключение `ThreadAbortException`. Особенность исключения состоит в том, что его невозможно подавить при помощи `catch`-блока. Исключение может быть отслежено тем потоком, который кто-то собирается уничтожить, а при помощи статического метода `ResetAbort()` запрос на уничтожение потока можно отклонить.

```
public class MainClass
{
    public static void ThreadProc()
    {
        while (true)
        {
            try
            {
                Console.WriteLine("..."); Thread.Sleep(1000);
            }
            catch (ThreadAbortException e)
            {
                // отлавливаем попытку уничтожения и отменяем её
                Thread.ResetAbort();
            }
        }
    }
    public static void Main()
    {
        var th = new Thread(ThreadProc);
        th.Start(); Thread.Sleep(3000);
        th.Abort(); // пытаемся прервать работу потока th
        th.Join(); // ждём завершения потока, но не дождёмся
    }
```

```
}
```

Диаграмма состояний потока с указанием значения свойства `ThreadState`:



Для выполнения в отдельном потоке повторяющегося метода можно применить класс `Timer` из пространства `System.Threading`.

Создание отдельного потока – это довольно «затратная» операция с точки зрения расхода времени и памяти. Платформа .NET поддерживает специальный механизм, называемый *пул потоков*. Пул потоков используют многие классы и технологии платформы .NET - асинхронные делегаты, таймеры, ASP.NET.

Пул состоит из двух основных элементов: *очереди методов* и *рабочих потоков*. Характеристикой пула является его *ёмкость* – максимальное число рабочих потоков. При работе с пулом метод сначала помещается в очередь. Если у пула есть свободные рабочие потоки, метод извлекается из очереди и направляется свободному потоку для выполнения. Если свободных потоков нет, но ёмкость пула не достигнута, для обслуживания метода формируется новый рабочий поток. Однако этот поток создается с задержкой в полсекунды. Если за это время освободится какой-либо из рабочих потоков, то он будет назначен на выполнение метода, а новый рабочий поток создан не будет. Важным нюансом является то, что несколько первых рабочих потоков в пуле создаётся без полусекундной задержки.

Для работы с пулом используется статический класс `ThreadPool`. Метод `SetMaxThreads()` позволяет изменить ёмкость пула, которая по умолчанию равна 1023. Метод `SetMinThreads()` устанавливает количество рабочих потоков, создаваемых без задержки. По умолчанию их число равно количеству

процессорных ядер. Для помещения метода в очередь пула служит метод `QueueUserWorkItem()`.

Синхронизация потоков

При использовании многопоточности естественным образом возникает вопрос об управлении совместным доступом к данным и синхронизации потоков.

Если метод запускается в нескольких потоках, только локальные переменные метода будут уникальными для потока. Поля объектов по умолчанию разделяются между всеми потоками. В пространстве имён `System` определён атрибут `[ThreadStatic]`, применяемый к статическим полям. Если поле помечено таким атрибутом, то каждый поток будет содержать свой экземпляр поля. Для `[ThreadStatic]`-полей не рекомендуется делать инициализацию при объявлении, так как код инициализации выполнится только в одном потоке.

Для неразделяемых статических полей класса можно использовать тип `ThreadLocal<T>`. Перегруженный конструктор `ThreadLocal<T>` принимает функцию инициализации поля. Значение поля хранится в свойстве `Value`.

Синхронизация потоков – это координирование действий потоков для получения предсказуемого результата. Средства синхронизации потоков можно разделить на четыре категории:

- простые методы приостановки выполнения потока (`Suspend()`, `Resume()`, `Sleep()`, `Yield()`, `Join()`);
- блокирующие конструкции;
- конструкции подачи сигналов;
- незадерживающие средства синхронизации.

Первые три категории используют для синхронизации приостановку потока. Приостановленный поток практически не потребляет времени процессора, однако его выполнение легко возобновляется системой при наступлении условия «пробуждения».

Блокирующие конструкции обеспечивают исключительный доступ к ресурсу (например, к полю или фрагменту кода), гарантируя, что в каждый момент времени с ресурсом работает только один поток. Блокировка позволяет потокам работать с общими данными, не мешая друг другу. Для организации блокировок платформа .NET предоставляет такие классы, как `Monitor`, `Mutex`, `Semaphor`, `SemaphorSlim`, а язык C# - специальный

оператор `lock`, который является всего лишь скрытым способом работы со статическим классом `Monitor`. Объявление критической секции эквивалентно следующему коду:

```
Monitor.Enter(buffer);
try
{
    // операторы критической секции
}
finally
{
    Monitor.Exit(buffer);
}
```

Метод `Monitor.Enter()` определяет вход в критическую секцию, метод `Monitor.Exit()` – выход из секции. Аргументами методов является объект-идентификатор критической секции.

Кроме `Enter()` и `Exit()`, класс `Monitor` обладает ещё несколькими полезными методами. Например, метод `Wait()` применяется внутри критической секции и снимает с неё блокировку (при этом можно задать период времени, на который снимается блокировка). При вызове `Wait()` текущий поток останавливается, пока не будет вызван (из другого потока) метод `Monitor.Pulse()`.

Иногда ресурс нужно блокировать так, чтобы читать его могли несколько потоков, а записывать - только один. Для этих целей предназначен класс `ReaderWriterLockSlim`. Его экземплярные методы `EnterReadLock()` и `ExitReadLock()` задают секцию чтения ресурса, а методы `EnterWriteLock()` и `ExitWriteLock()` - секцию записи ресурса.

Потребность в *синхронизации на основе подачи сигналов* возникает, когда один поток ждёт прихода уведомления от другого потока. Для осуществления данной синхронизации используется базовый класс `EventWaitHandle`, его наследники `AutoResetEvent` и `ManualResetEvent`, а также класс `ManualResetEventSlim`. Имея доступ к объекту `EventWaitHandle`, поток может вызвать его метод `WaitOne()`, чтобы остановиться и ждать сигнала. Для отправки сигнала применяется вызов метода `Set()`. Если используются `ManualResetEvent` и `ManualResetEventSlim`, все ожидающие потоки освобождаются и продолжают выполнение. При использовании `AutoResetEvent` ожидающие потоки освобождаются и запускаются

последовательно, на манер очереди.

Чтобы организовать *незадерживающую синхронизацию*, используется статический класс `System.Threading.Interlocked`. Класс `Interlocked` имеет методы для инкремента, декремента и сложения аргументов типа `int` или `long`, а также методы присваивания значений числовым и ссылочным переменным. Каждый метод выполняется как атомарная операция.

```
    int x = 10, y = 20;
    Interlocked.Add(ref x, y);                      // x = x + y
    Interlocked.Increment(ref x);                   // x++
    Interlocked.Exchange(ref x, y);                 // x = y
    Interlocked.CompareExchange(ref x, 50, y);      // if (x == y) x =
50
```

20. Класс Task и выполнение асинхронных операций с его помощью

Задача (task) – это инструмент для организации параллельной работы. Однако, являясь абстракцией более высокого уровня, она призвана устранить некоторые ограничения потоков. В частности, у потоков отсутствует механизм *продолжений*, когда после завершения метода, работающего в потоке, в этом же потоке автоматически запускается другой заданный метод. Затруднено получение значения функции, выполняющейся в отдельном потоке. Наконец, необдуманное создание множества потоков ведёт к повышенному расходу памяти и замедлению работы приложения. Концепция задач была представлена в четвёртой версии платформы .NET.

Для описания задач используются объекты классов `Task` и `Task<TResult>`, размещённых в пространстве имён `System.Threading.Tasks`.

Простейший способ создания и запуска задачи – вызов статического метода `Task.Run()`. Этот метод принимает в качестве аргумента объект типа `Action`.

```
    var task = Task.Run(() => Console.WriteLine("Hello"));
```

Класс `TaskFactory` содержит набор методов, соответствующих некоторым сценариям использования задач – `StartNew()`, `FromAsync()`, `ContinueWhenAll()`, `ContinueWhenAny()`. Экземпляр `TaskFactory` доступен через статическое свойство `Task.Factory`. Вызов `Task.Run()` – это сокращение для `Task.Factory.StartNew()`.

```
    // аналог предыдущего примера
    Action action = () => Console.WriteLine("Hello");
    var task = Task.Factory.StartNew(action);
```

Для создания задачи можно использовать один из перегруженных конструкторов класса Task. При этом указывается аргумент типа Action – метод, выполняемый в задаче.

```
    Action<object> method = x => { Console.WriteLine(x.ToString());
};
    var task2 = new Task(method, 25);
```

Перегруженные конструкторы класса `Task` принимают опциональные аргументы типа `CancellationToken` и `TaskCreationOptions`. Перечисление `TaskCreationOptions` задаёт вид задачи (например, `LongRunning` – долгая задача). Структура `CancellationToken` применяется для прерывания задачи (это так называемый *токен отмены*).

```
    var task1 = new Task(action, TaskCreationOptions.LongRunning);
```

Метод `Start()` запускает задачу, вернее, помещает её в очередь запуска *планировщика задач*. По умолчанию применяется планировщик на основе пула потоков, но существует возможность использовать пользовательский планировщик. Метод `RunSynchronously()` выполняет задачу синхронно (в терминах используемого планировщика задач).

Методы класса `Task` `Wait()`, `WaitAll()` и `WaitAny()` останавливают основной поток до завершения задачи (или задач). Перегруженные версии методов позволяют указать период ожидания завершения и токен отмены.

```
    task1.Wait(1000);
    Task.WaitAll(task1, task2);
```

Класс `Task<TResult>` наследуется от `Task` и описывает задачу, возвращающую значение типа `TResult`. Дополнительно к элементам базового класса, `Task<TResult>` объявляет свойство `Result` для хранения вычисленного значения. Конструкторы класса `Task<TResult>` принимают аргументы типа `Func<TResult>` и `Func<object, TResult>` (опционально – аргументы типа `CancellationToken` и `TaskCreationOptions`).

Обработка исключений и отмена выполнения задач

Если код внутри задачи генерирует необработанное исключение (задача *отказывает*), это исключение автоматически повторно сгенерируется при вызове метода `Wait()` или при доступе к свойству `Result` класса `Task<TResult>`. CLR помещает исключение в оболочку `System.AggregateException`. При

параллельном возникновении нескольких исключений все они собираются в единое исключение `AggregateException` и доступны в свойстве-коллекции `InnerExceptions`.

```
var task = Task.Run(() => { throw new Exception(); });
try
{
    task.Wait();
}
catch (AggregateException ex)
{
    var message = ex.InnerException.Message;
    Console.WriteLine(message);
}
```

При работе с задачами применяется особый подход для отмены их выполнения. Структура типа `CancellationToken` (токен отмены) – это своеобразный маркер того, что задачу можно отменить. Токен отмены принимает в качестве аргумента, в частности, конструктор задачи. Класс `CancellationTokenSource` содержит свойство `Token` для получения токенов отмены и метод `Cancel()` для отмены выполнения всех задач, использующих общий токен.

```
var tokenSource = new CancellationTokenSource();
// используем один токен в двух задачах
new Task(SomeMethod, tokenSource.Token).Start();
new Task(OtherMethod, tokenSource.Token).Start();
// в нужный момент отменяем обе задачи
tokenSource.Cancel();
```

Продолжения

Продолжение сообщает задаче, что после её завершения она должна продолжить делать что-то другое. Первый способ организации продолжения заключается в использовании экземплярного метода задачи `ContinueWith()`.

```
var task1 = Task.Run(() => Console.Write("Task.."));
var task2 = task1.ContinueWith(t => Console.Write("continuation"));
```

Выполнение продолжения можно запланировать на основе завершения множества предшествующих задач при помощи статических методов `Task.WhenAll()` и `Task.WhenAny()`.

Второй способ организации продолжения заключается в использовании

объекта ожидания. *Объект ожидания* – это любой объект, имеющий методы `OnCompleted()` и `GetResult()` и булево свойство `IsCompleted`. Вызов метода `GetAwaiter()` на задаче возвращает объект ожидания. Метод `OnCompleted()` принимает в качестве аргумента делегат, содержащий код продолжения. Метод `GetResult()` возвращает результат работы предшественника (или void).

```
    // задача подсчёта простых чисел (используется LINQ)
    Task<int> prime = Task.Run(() =>
    Enumerable.Range(2, 3000000 - 2).Count(n =>
        Enumerable.Range(2, (int) Math.Sqrt(n) - 1).All(i => n%i >
0)));
    // получаем объект продолжения
    var awaiter = prime.GetAwaiter();
    // указываем, что делать после окончания предшественника
    awaiter.OnCompleted(() =>
            {
                // получаем результат вычислений предшественника
                int result = awaiter.GetResult();
                Console.WriteLine(result);
            });
```

Асинхронные функции в C#

Для поддержки асинхронного программирования в версии C# 5.0 появилась операция await. Синтаксис этой операции следующий:

```
    var результат = await выражение;
    оператор(ы);
```

Компилятор разворачивает приведённую выше конструкцию в такой функциональный эквивалент:

```
    var awaiter = выражение.GetAwaiter();
    awaiter.OnCompleted(()=>
            {
                var результат = awaiter.GetResult();
                оператор(ы);
    }
```

Операция await применима и к выражению, не возвращающему значения:

```
    await выражение;
    оператор(ы);
```

Выражение, на котором применяется await, обычно является задачей. Тем не менее, компилятор удовлетворит любой объект ожидания (определение

данного понятия приведено выше). Операция await может применяться только внутри метода (или лямбда-выражения) со специальным модификатором async. Метод должен возвращать void либо тип Task или Task<TResult>. Методы с модификатором async называются *асинхронными функциями*. Встретив выражение await, процесс выполнения (обычно) производит возврат в вызывающий код – почти как оператор yield return в итераторе. Но перед возвратом исполняющая среда присоединяет к ожидающей задаче признак продолжения, гарантирующий, что когда задача завершается, управление перейдёт обратно в метод и продолжится с места, в котором оно его оставило.

```
Func<Task> unnamed = async () =>
            {
                // Task.Delay() - асинхронный аналог Thread.Sleep()
                await Task.Delay(1000);
                Console.Write("Done");
            };
```

21. Структурные шаблоны проектирования.

Шаблоны (или *паттерны*) *проектирования* (design patterns) – это многократно используемые решения распространённых проблем, возникающих при разработке программного обеспечения.

Структурные шаблоны – показывают, как объекты и классы объединяются для образования сложных структур.

Компоновщик (Composite)

Его задача – агрегировать объекты, обладающие общим интерфейсом, и обеспечить выполнение операций этого интерфейса над всеми агрегируемыми объектами.

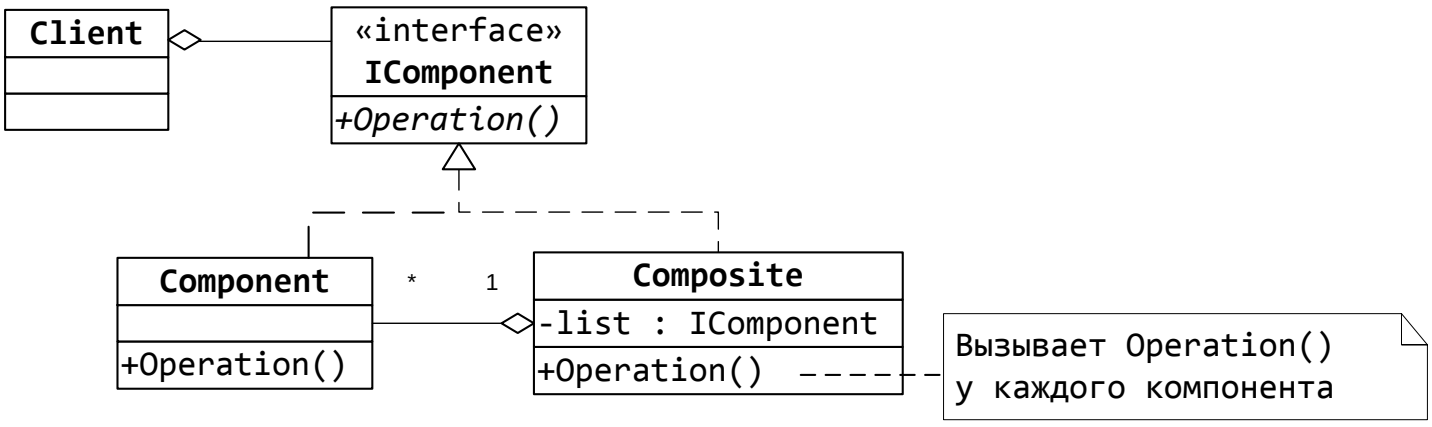


Рис. 6. Дизайн шаблона Компоновщик.

Приспособленец (Flyweight)

Шаблон Приспособленец предлагает эффективный способ разделить общую информацию, находящуюся в небольших объектах. Суть шаблона в том, чтобы разделить состояние некоторого объекта на состояния трех типов. *Внутреннее состояние* (intrinsic state) принадлежит самому объекту. Тип Flyweight реализует интерфейс IFlyweight, который определяет операции, в которых заинтересована остальная часть системы. Клиент владеет *общим* (неразделяемым) *состоянием* (unshared state), а также коллекцией приспособленцев, которых производит класс-фабрика (FlyweightFactory). Наконец, *внешнее состояние* (extrinsic state) не появляется в системе как таковое. Если оно понадобится, то будет вычислено уже во время выполнения программы для каждого внутреннего состояния.

Адаптер (Adapter)

Шаблон Адаптер предназначен для обеспечения совместной работы классов, которые изначально не были предназначены для совместного использования. Такие ситуации часто возникают, когда идет работа с существующими библиотеками кода. Нередко их интерфейс не отвечает требованиям клиента, но изменить библиотеку возможности нет. Возникает задача адаптации библиотеки для клиента.

Дизайн шаблона Адаптер показан на рис. 8.

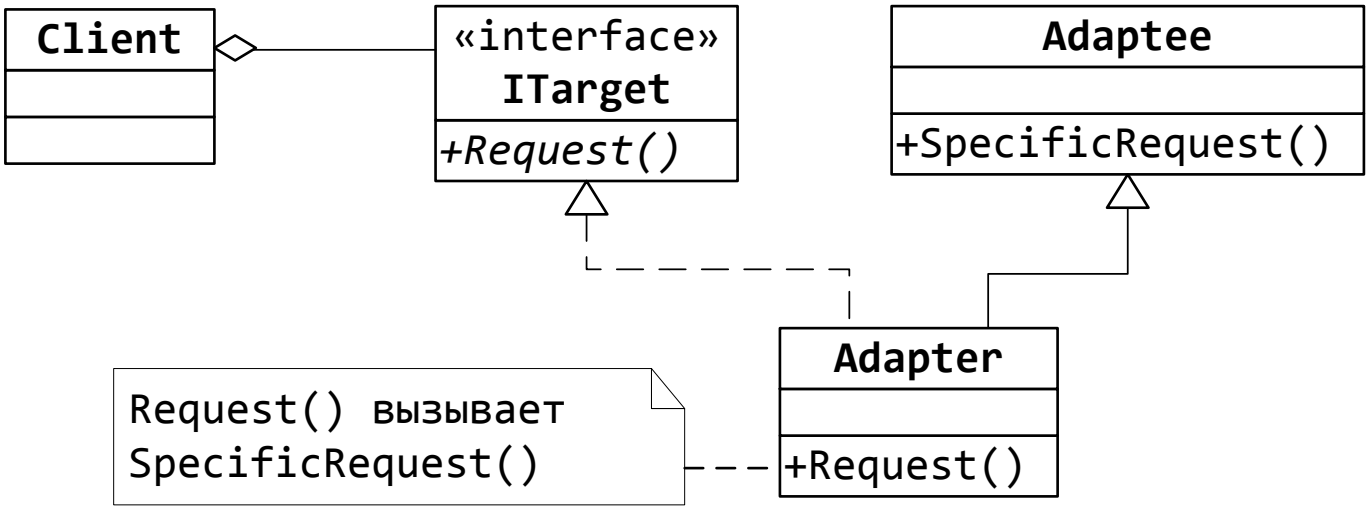


Рис. 8. Дизайн шаблона Адаптер.

Фасад (Façade)

Назначение шаблона Фасад состоит в предоставлении различных высокоуровневых представлений подсистем, детали реализации которых

скрыты от клиента. В общем случае набор операций, желаемый для клиента, может формироваться в виде набора из разных частей подсистемы. Сокрытие деталей – это ключевая концепция программирования.

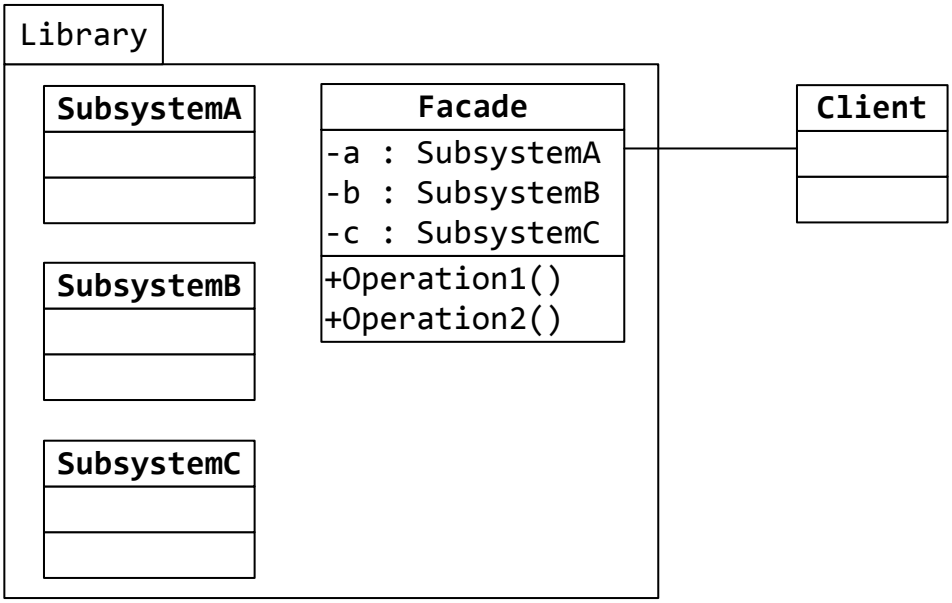


Рис. 9. Дизайн шаблона проектирования Фасад.

Декоратор (Decorator)

Шаблон Декоратор даёт способ для динамического добавления к объекту нового состояния и поведения. При этом исходный объект не знает о том, что он «декорируется». Ключевым моментом реализации шаблона является то, что класс-декоратор одновременно наследуется от декорируемого класса и агрегирует объект этого класса.

UML-диаграмма, показывающая типичный дизайн шаблона Декоратор, представлена на рис. 3. На диаграмме и декоратор, и декорируемый объект реализуют общий интерфейс. Возможен вариант, когда вместо интерфейса используется общий класс-предок. Кроме этого, декоратор агрегирует декорируемый объект, добавляя к нему новое поведение.

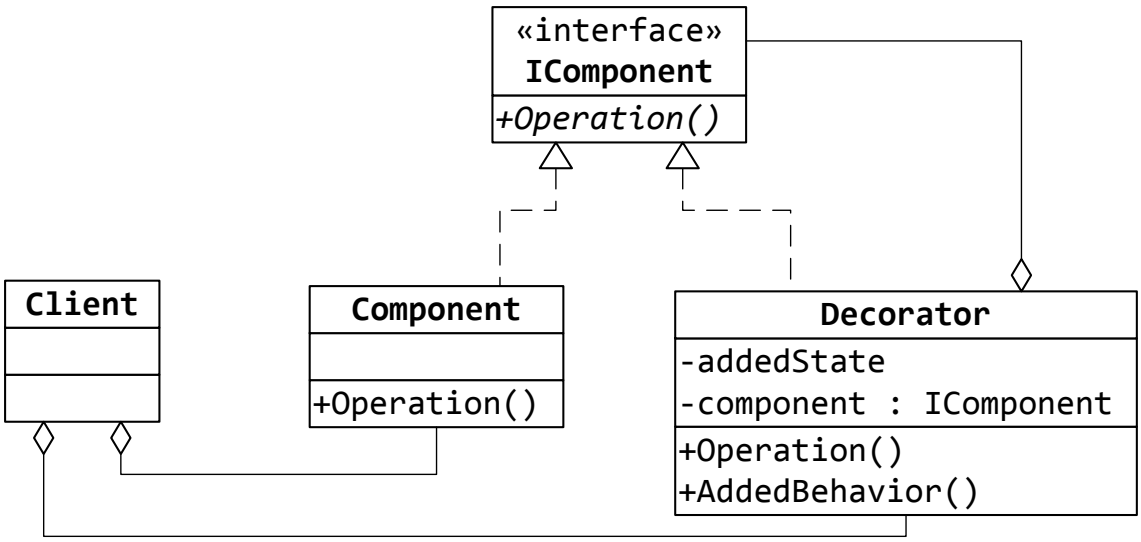


Рис. 3. UML-диаграмма шаблона Декоратор.

При практической реализации шаблона декорируемый объект обычно передается как параметр конструктора декоратора и сохраняется во внутреннем поле. Методы декоратора могут использовать сохраненный объект при своей работе.

Обратите внимание на то, что несколько декораторов независимо могут применяться к одному и тому же объекту, а также на то, что декоратор может декорировать объект, который уже декорирован.

Заместитель (Proxy)

Шаблон проектирования Заместитель применяется для контроля создания и доступа к объекту. Заместитель – это обычно небольшой открытый объект, «за спиной» которого находится сложный объект, выполняющий реальную работу и создающийся только при наступлении определённых условий.

Иллюстрацией шаблона Заместитель может служить работа с сайтом, который требует необязательной авторизации. Если пользователь не авторизовался, ему доступен сравнительно небольшой функционал. После авторизации (это условие создания сложного объекта) активируются дополнительные функции.

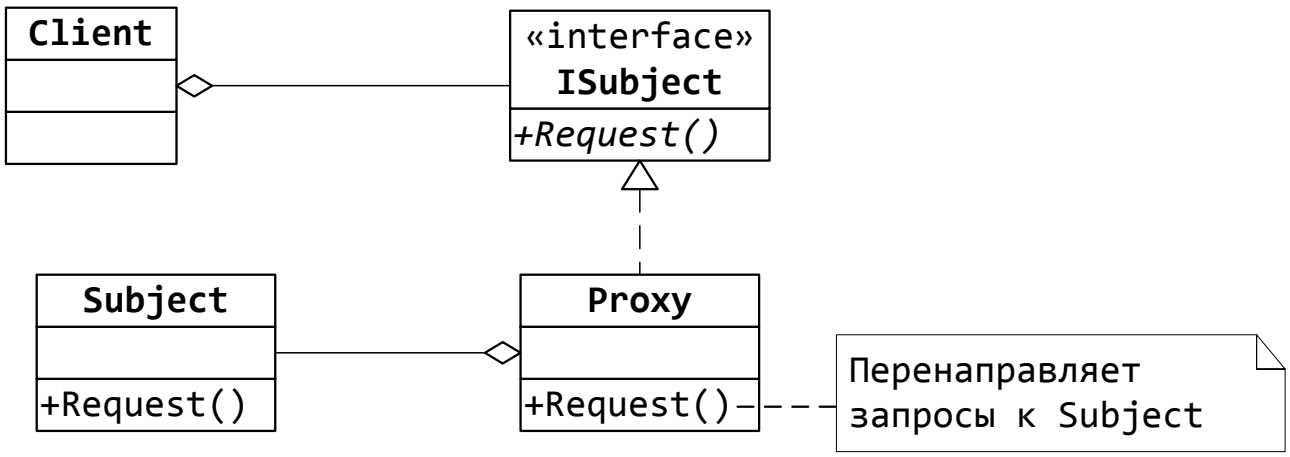


Рис. 4. UML-диаграмма шаблона Заместитель.

Мост (Bridge)

Шаблон Мост используется в тех случаях, когда имеется отдельная иерархия абстрактных классов и интерфейсов и соответствующая иерархия реализаций. Мост соединяет абстракции и реализации в виде независимых динамических классов.

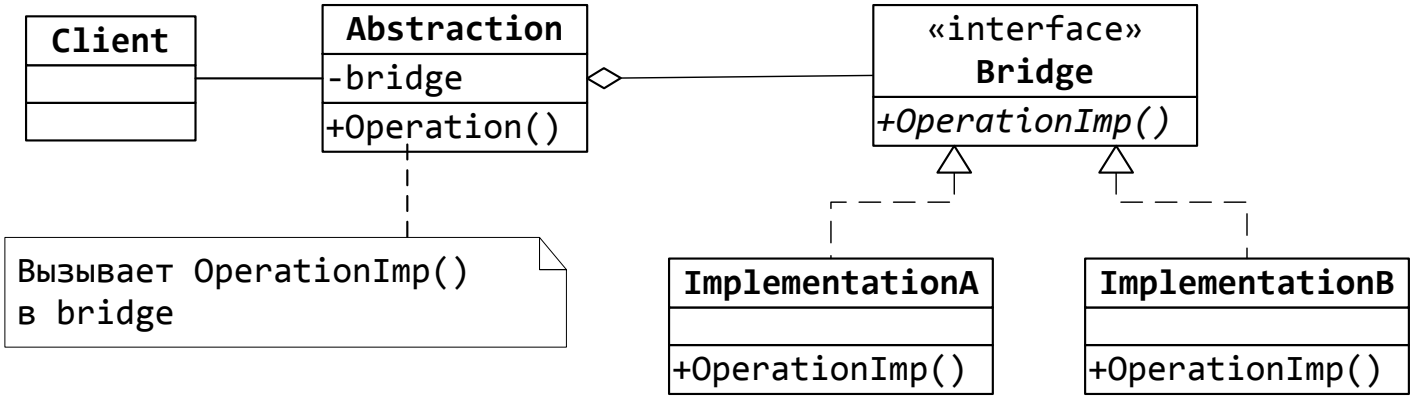


Рис. 5. UML-диаграмма шаблона Мост.

22. Порождающие шаблоны проектирования.

Шаблоны (или *паттерны*) *проектирования* (design patterns) – это многократно используемые решения распространённых проблем, возникающих при разработке программного обеспечения.

Порождающие шаблоны – управляют и контролируют процесс создания и жизненный цикл объектов.

Фабричный метод (Factory method)

Фабричный метод занимается созданием объектов. Каждый такой объект принадлежит некому классу, однако все классы либо имеют общего предка,

либо реализуют общий интерфейс. Фабричный метод сам решает, какой конкретный класс нужно использовать для создания очередного объекта. Решение принимается либо на основе информации, предоставленной клиентом, либо на основе внутреннего состояния метода.

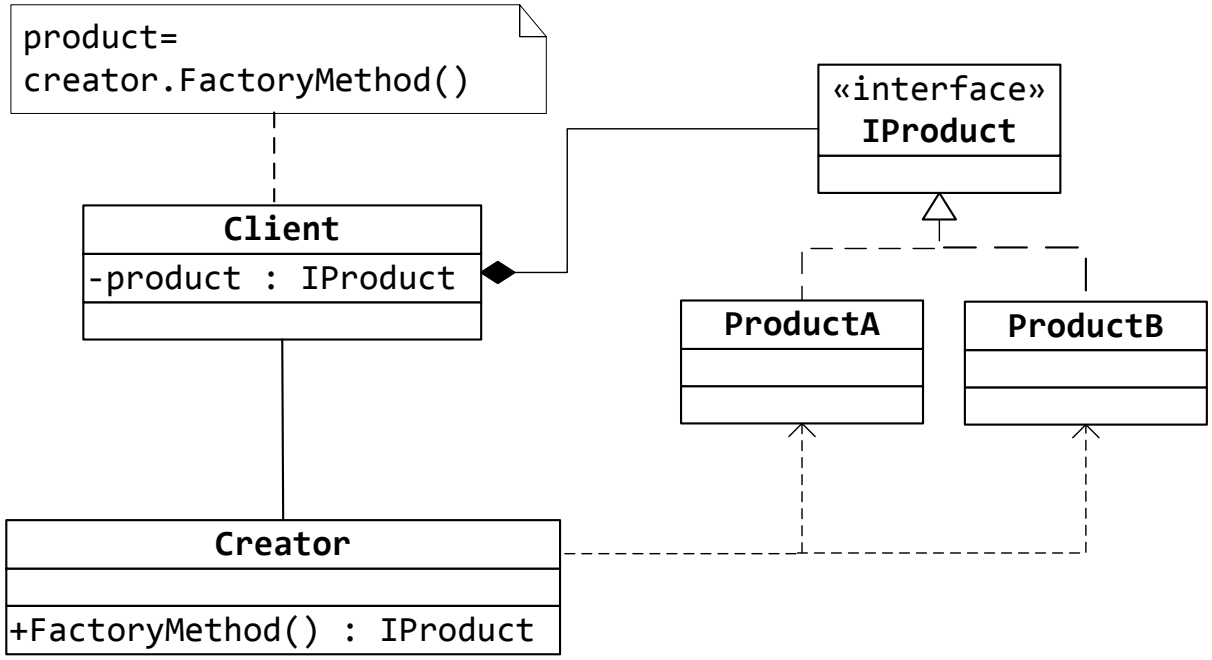


Рис. 11. UML-диаграмма шаблона Фабричный метод.

Далее приведён код, использующий шаблон Фабричный метод и соответствующий описанному выше гипотетическому примеру.

```
public interface IProduct
{
    string ShipFrom();
}

public class ProductFromAfrica : IProduct
{
    public String ShipFrom()
    {
        return "from South Africa";
    }
}

public class ProductFromSpain : IProduct
{
    public String ShipFrom()
    {
```

```
        return "from Spain";
    }
}

public class DefaultProduct : IProduct
{
    public String ShipFrom()
    {
        return "not available";
    }
}

public class Creator
{
    public IProduct FactoryMethod(int month)
    {
        if (month >= 4 && month <= 11)
        {
            return new ProductFromAfrica();
        }
        if (month == 12 || month == 2 || month == 1)
        {
            return new ProductFromSpain();
        }
        return new DefaultProduct();
    }
}

public class FactoryMethodExample
{
    private static void Main()
    {
        var c = new Creator();
        IProduct product;
        for (var i = 1; i <= 12; i++)
        {
            product = c.FactoryMethod(i);
            Console.WriteLine("Avocados " + product.ShipFrom());
        }
    }
```

```
}
```

Одиночка (Singleton)

Шаблон Одиночка гарантирует создание единственного экземпляра объекта некоторого класса.

Необходимость в единичных объектах часто возникает при программировании физических устройств. Если в системе установлен единственный принтер, то нужно быть уверенным, что в программе имеется единственный объект, описывающий этот принтер и предоставляющий доступ к его настройкам. Иначе возможна ситуация рассогласования – различные (возможно, противоречивые) настройки будут выполняться в разных частях программы над одним физическим объектом.

Шаблон Одиночка добавляет функциональность путём модификации существующего класса. Модификация требует следующих шагов.

- Конструктор класса делается закрытым (private).
- Добавляется закрытое статическое поле только для чтения, которое инстанциируется, используя закрытый экземплярный конструктор класса.
- Для доступа к закрытому статическому полю добавляется открытое статическое свойство.

Код, который соответствует описанным шагам реализации шаблона, представлен ниже.

```
public sealed class Singleton
{
    // Закрытый конструктор
    private Singleton() { }

    // Закрытое поле, хранит экземпляр класса
    private static readonly Singleton uniqueInstance =
                                                    new
Singleton();

    // Открытое свойство для доступа к экземпляру
    public static Singleton Instance
    {
        get { return uniqueInstance; }
    }
```

```
}
```

Возможен вариант, когда инициализация внутреннего экземпляра проводится при первом обращении к открытому свойству:

```
// Один из вариантов реализации шаблона Singleton
public sealed class Singleton
{
    private Singleton() { }

    private static Singleton uniqueInstance;

    public static Singleton Instance
    {
        get
        {
            if (uniqueInstance == null)
            {
                uniqueInstance = new Singleton();
            }
            return uniqueInstance;
        }
    }
}
```

Однако такой подход не является потокобезопасным, и использовать его не рекомендуется.

Абстрактная фабрика (Abstract factory)

Шаблон Абстрактная фабрика предназначен для создания объектов, принадлежащих одному набору классов и используемых совместно. Абстрактная фабрика переопределяется в конкретных классах-фабриках, создающих объекты одного набора. Этот шаблон изолирует имена классов и их определения от клиента: единственный способ для клиента получить необходимый объект – это воспользоваться одной из реализаций абстрактной фабрики.

Дизайн шаблона Абстрактная фабрика включает большое число участников, однако он достаточно прост (рис. 13). Клиент хранит ссылку на конкретную фабрику, реализующую интерфейс IFactory. Диаграмма показывает, что один из вариантов установки связи клиента и фабрики – это передача объекта-фабрики в конструкторе клиента. Клиент работает с

определенным набором объектов, но использует для этого только реализуемые классами объектов интерфейсы (IProductA и IProductB). Для получения нужного объекта клиент вызывает один из методов фабрики.

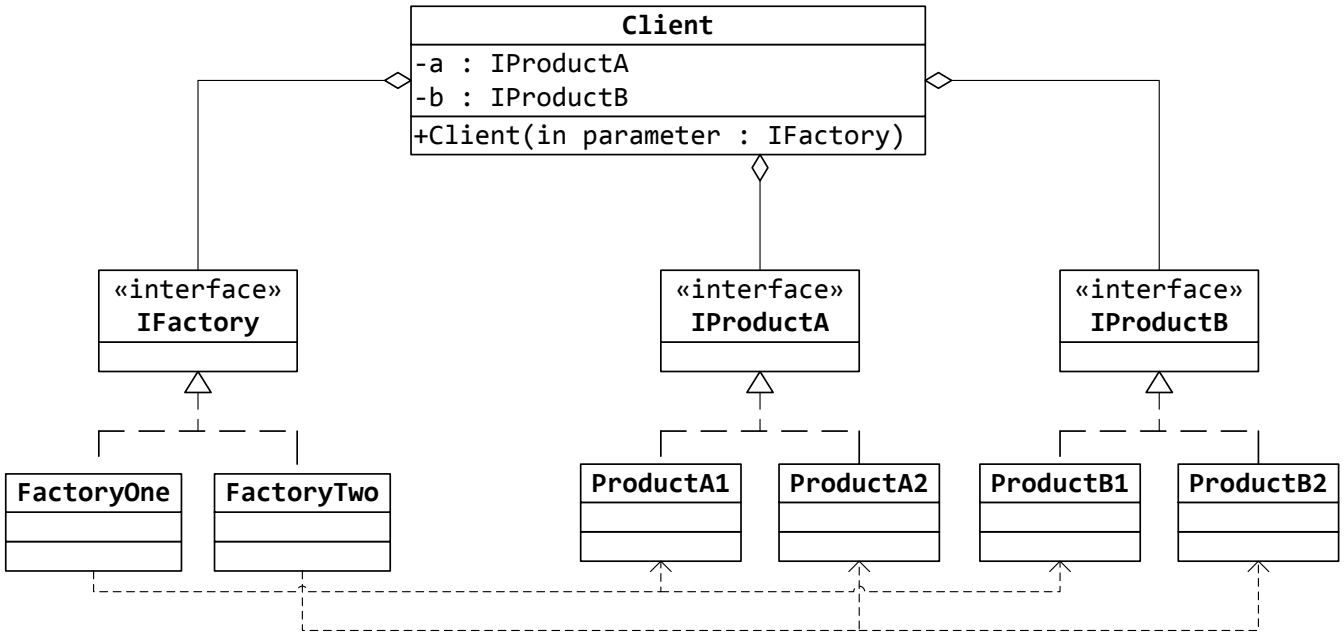


Рис. 13. Шаблон абстрактная фабрика.

Рассмотрим пример кода с реализацией шаблона абстрактная фабрика. Отметим одну важную отличительную особенность представленной реализации шаблона. Вместо набора классов-фабрик и классов для объектов клиента используются универсальные шаблоны (generics). Конкретный класс-фабрика получается конструированием из общего универсального шаблона при помощи параметра TBrand.

```
    // Общий интерфейс фабрики
    public interface IFactory
    {
        IBag CreateBag();
        IShoes CreateShoes();
    }

    // Это generic-класс будет конструироваться в конкретные фабрики
    public class Factory<TBrand> : IFactory where TBrand : IBrand,
new()
    {
        public IBag CreateBag()
        {
```

```
        return new Bag<TBrand>();
    }

    public IShoes CreateShoes()
    {
        return new Shoes<TBrand>();
    }
}

// Два интрефейса, с которыми работает клиент
public interface IBag
{
    string Material { get; }
}

public interface IShoes
{
    int Price { get; }
}

// Generic-классы для конкретных классов-продуктов
public class Bag<TBrand> : IBag where TBrand : IBrand, new()
{
    private readonly TBrand myBrand;

    public Bag()
    {
        myBrand = new TBrand();
    }

    public string Material
    {
        get { return myBrand.Material; }
    }
}

public class Shoes<TBrand> : IShoes where TBrand : IBrand, new()
{
    private readonly TBrand myBrand;
```

```
    public Shoes()
    {
        myBrand = new TBrand();
    }

    public int Price
    {
        get { return myBrand.Price; }
    }
}

// Интерфейс для описания брэндов и конкретные брэнды
public interface IBrand
{
    int Price { get; }
    string Material { get; }
}

public class Gucci : IBrand
{
    public int Price
    {
        get { return 1000; }
    }

    public string Material
    {
        get { return "Crocodile skin"; }
    }
}

public class Poochy : IBrand
{
    public int Price
    {
        get { return new Gucci().Price / 3; }
    }

    public string Material
    {
```

```
            get { return "Plastic"; }
        }
    }

    public class Client<TBrand> where TBrand : IBrand, new()
    {
        public void ClientMain()
        {
            IFactory factory = new Factory<TBrand>();
            var bag = factory.CreateBag();
            var shoes = factory.CreateShoes();
            Console.WriteLine("A Bag is made from " + bag.Material);
            Console.WriteLine("Shoes' cost is" + shoes.Price);
        }
    }

    public static class AbstractFactoryExample
    {
        private static void Main()
        {
            new Client<Poochy>().ClientMain();
            new Client<Gucci>().ClientMain();
        }
    }
```

Строитель (Builder)

Шаблон Строитель позволяет клиенту создавать сложный объект, задавая для него только тип и содержимое. Одинаковый процесс создания способен порождать различные объекты.

Рассмотрим дизайн шаблона Строитель. В его основе лежит компонент Director (*режиссер*), который вызывает объекты-строители. Количество объектов-строителей произвольно, однако все они реализуют интерфейс IBuilder. Строители поставляют элементы для создания финального объекта Product. Как только объект Product будет создан, компонент Director предоставляет его клиенту.

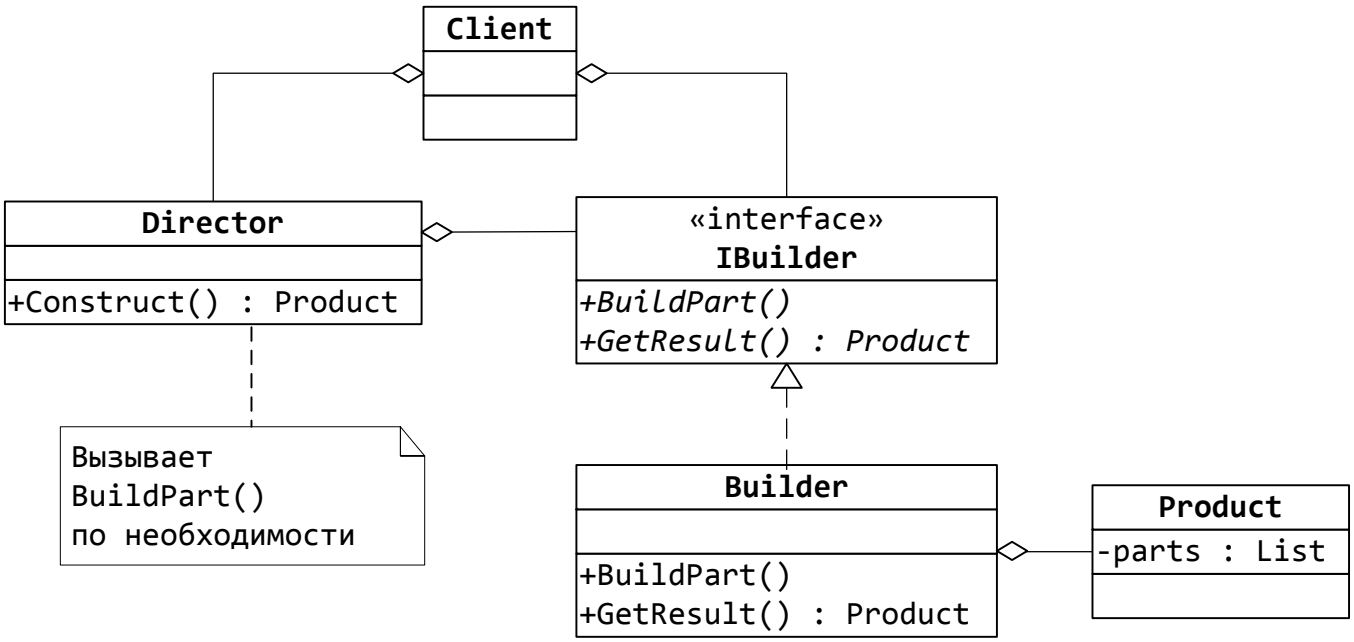


Рис. 14. Дизайн шаблона Строитель.

Прототип (Prototype)

Шаблон Прототип позволяет создавать специальные объекты, ничего не знаю об их классе или деталях создания. Механизм можно описать так: клиенту, инициирующему создание объектов, предоставляются объекты-прототипы. Затем целевые объекты создаются путем клонирования этих объектов-прототипов.

Объект обычно создаётся путем вызова конструктора. Если же используется шаблон Прототип, то клиент знает только об интерфейсе IPrototype, реализующем операции клонирования объекта. Реальный класс объекта клиенту не известен.

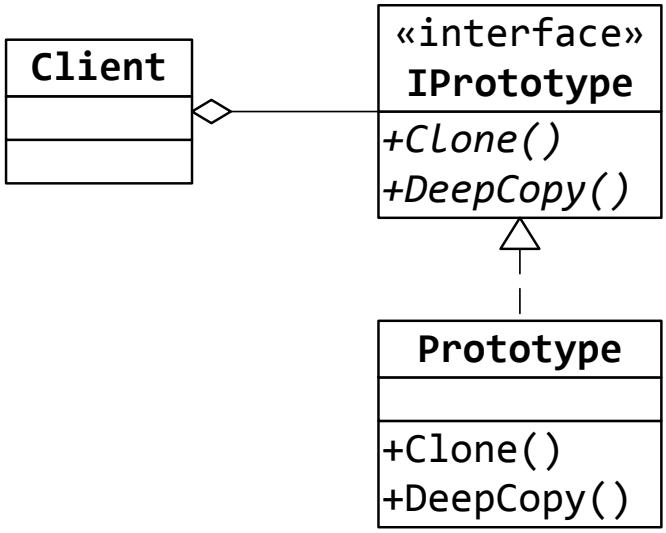


Рис. 10. Дизайн шаблона Прототип.

При реализации шаблона Прототип для платформы .NET следует учитывать, что пользовательские объекты могут быть ссылочными типами, а, значит, требовать дополнительных усилий по получению полной копии.

23. Шаблоны поведения

Шаблоны (или *паттерны*) *проектирования* (design patterns) – это многократно используемые решения распространённых проблем, возникающих при разработке программного обеспечения.

Шаблоны поведения – используются для организации, управления и объединения различных вариантов поведения объектов.

Стратегия (Strategy)

При помощи шаблона Стратегия из клиента выделяется алгоритм, который затем инкапсулируется в типах, реализующих общий интерфейс. Это позволяет клиенту выбирать нужный алгоритм путем инстанциирования объектов необходимых типов. Кроме этого, шаблон допускает изменение набора доступных алгоритмов со временем.

Предположим, что требуется разработать программу, которая показывает календарь. Одно из требований к программе – она должна отображать праздники, отмечаемые различными нациями и религиозными группами. Это требование можно выполнить, помещая логику генерирования для каждого набора праздников в отдельный класс. Основная программа будет выбирать необходимый класс из набора, исходя, например, из действий пользователя (или конфигурационных настроек).

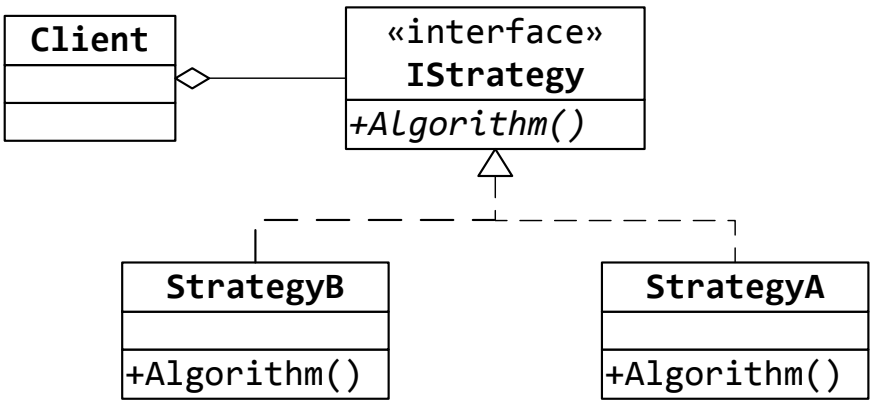


Рис. 15. UML-диаграмма шаблона Стратегия.

Ниже приведён пример кода с реализацией шаблона Стратегия.

```
public interface IHolidaySet
{
    List<string> GetHolidays();
}

public class USHolidaySet : IHolidaySet
{
    public List<string> GetHolidays()
    {
        return new List<string> { "01.01.09", "25.05.09",
                                  "04.07.09", "07.09.09",
                                  "26.11.09", "25.12.09" };
    }
}

public class RussianHolidaySet : IHolidaySet
{
    public List<string> GetHolidays()
    {
        return new List<string> { "01.01.09", "07.01.09",
                                  "23.02.09", "08.03.09",
                                  "19.04.09", "12.06.09",
                                  "04.11.09" };
    }
}

public class Client
{
```

```
        private readonly IHolidaySet holidaySetStrategy;

        public Client(IHolidaySet strategy)
        {
            holidaySetStrategy = strategy;
        }

        public bool CheckForHoliday(string date)
        {
            return holidaySetStrategy.GetHolidays().Contains(date);
        }
    }

    public static class StrategyExample
    {
        private static void Main()
        {
            var client = new Client(new USHolidaySet());
            var result = client.CheckForHoliday("01.01.09");
        }
}
```

Наблюдатель (Observer)

Шаблон Наблюдатель определяет отношение между объектами таким образом, что когда один из объектов меняет своё состояние, все другие объекты получают об этом уведомление.

Иллюстрацией применения шаблона Наблюдатель служит знакомая по языку C# система событий. Один объект публикует событие, остальные объекты могут подписаться на событие и получать уведомление о его наступлении. Собственно, основное назначение шаблона Наблюдатель – это реализация системы работы с событиями.

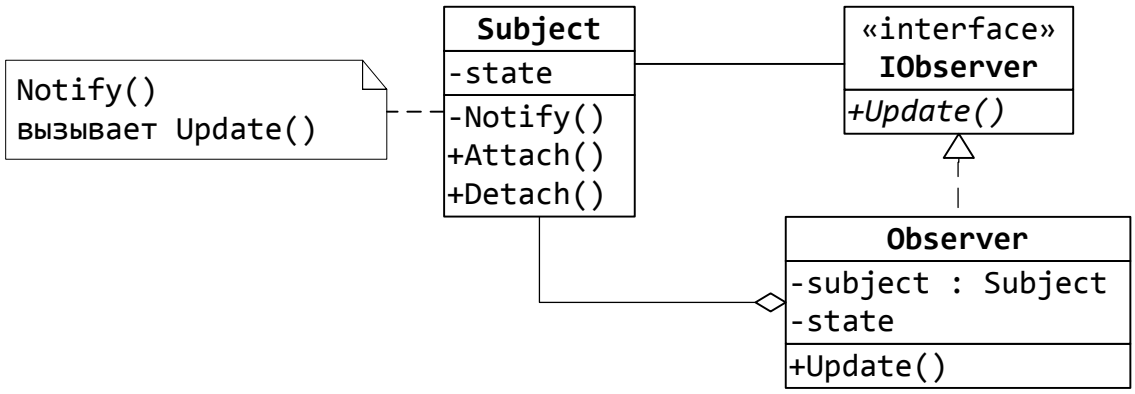


Рис. 22. Дизайн шаблона Наблюдатель.

Далее приводится пример кода, демонстрирующего реализацию шаблона Наблюдатель. Заметим, что в примере намеренно не используются события.

```
    // аналог Subject
    public abstract class Stock
    {
        private double price;
        private readonly IList<IInvestor> investors =
                                                    new
List<IInvestor>();

        public double Price
        {
            get { return price; }
            set
            {
                if (price != value)
                {
                    price = value;
                    Notify();
                }
            }
        }

        public string Symbol { get; private set; }

        protected Stock(string symbol, double price)
        {
            Symbol = symbol;
            this.price = price;
```

```
    }

    public void Attach(IInvestor investor)
    {
        investors.Add(investor);
    }

    public void Detach(IInvestor investor)
    {
        investors.Remove(investor);
    }

    public void Notify()
    {
        foreach (var investor in investors)
        {
            investor.Update(this);
        }
    }
}

public class IBM : Stock
{
    public IBM(double price) : base("IBM", price) { }
}

// интерфейс, аналогичный IObserver
public interface IInvestor
{
    void Update(Stock stock);
}

// конкретный обозреватель
public class Investor : IInvestor
{
    public string Name { get; private set; }
    public Stock Stock { get; set; }

    public Investor(string name)
    {
```

```
            Name = name;
        }

        public void Update(Stock stock)
        {
            Console.WriteLine("Notified {0} of {1}'s change to {2:C}",
                                        Name,              stock.Symbol,
stock.Price);
        }
    }

    public class ObserverExample
    {
        private static void Main()
        {
            var ibm = new IBM(120.00);
            ibm.Attach(new Investor("Sorros"));
            ibm.Attach(new Investor("Berkshire"));
            ibm.Price = 120.10;
            ibm.Price = 121.00;
            ibm.Price = 120.50;
            ibm.Price = 120.75;
            Console.ReadKey();
        }
```

Шаблонный метод (Template method)

Предположим, что программная логика некоего алгоритма представлена в виде набора вызовов методов. При использовании Шаблонного метода создается абстрактный класс, который реализует только часть методов программной логики, оставляя детали реализации остальных методов своим потомкам. Благодаря этому общая структура алгоритма остаётся неизменной, в то время как некоторые конкретные шаги могут изменяться.

```
    public abstract class Sorter
    {
        private readonly int[] data;

        protected Sorter(params int[] source)
        {
            data = new int[source.Length];
```

```
            Array.Copy(source, data, source.Length);
        }

        protected abstract bool Compare(int x, int y);

        public void Sort()
        {
            for (var i = 0; i < data.Length - 1; i++)
            {
                for (var j = i + 1; j < data.Length; j++)
                {
                    if (Compare(data[j], data[i]))
                    {
                        var temp = data[i];
                        data[i] = data[j];
                        data[j] = temp;
                    }
                }
            }
        }

        public int[] GetData()
        {
            var result = new int[data.Length];
            Array.Copy(data, result, data.Length);
            return result;
        }
    }

    public class GreaterFirstSorter : Sorter
    {
        public GreaterFirstSorter(params int[] source) : base(source){
}

        protected override bool Compare(int x, int y)
        {
            return x > y;
        }
    }
```

```
    public class TemplateMethodExample
    {
        private static void Main()
        {
            var sorter = new GreaterFirstSorter(1, 3, -10, 0);
            sorter.Sort();
            var result = sorter.GetData();
        }
}
```

Состояние (State)

Шаблон Состояние можно рассматривать как динамическую версию шаблона Стратегия. При использовании шаблона Состояние поведение объекта меняется в зависимости от текущего контекста.

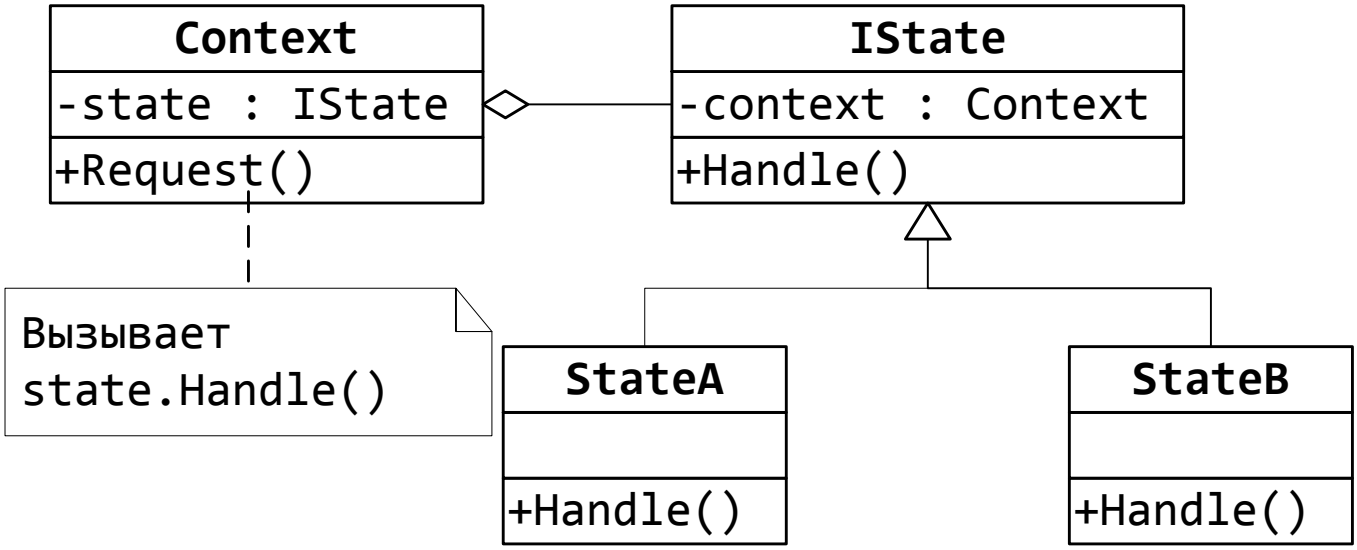


Рис. 17. Дизайн шаблона состояние.

При практической реализации шаблона Состояние отдельные объекты-состояния могут быть оформлены с применением шаблона Одиночка и получать объект-контекст в качестве одно из параметров своих методов.

Цепочка обязанностей (Chain of responsibility)

Шаблон Цепочка обязанностей работает со списком объектов, называемых *обработчиками*. Каждый из обработчиков имеет естественные ограничения на множество запросов, которые он в состоянии поддержать. Если текущий обработчик не может поддержать запрос, он передает его следующему обработчику в списке. Так продолжается, пока запрос не будет обработан, или пока список не закончится.

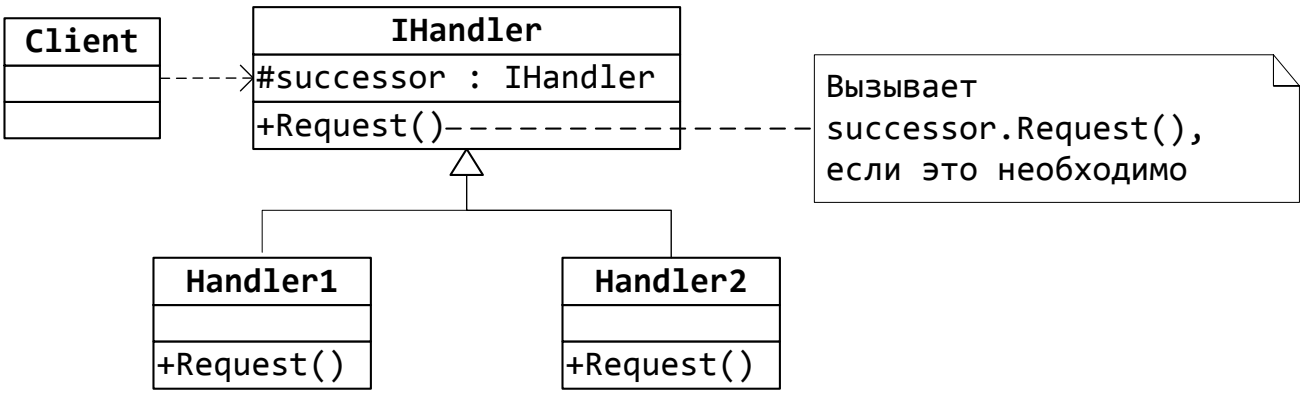


Рис. 18. UML-диаграмма шаблона Цепочка обязанностей.

Команда (Command)

Шаблон Команда инкапсулирует команды в объекте таким образом, что можно управлять их выбором, ставить в очередь, отменять и выполнять иные манипуляции.

Дизайн шаблона Команда показан на рис. 19.

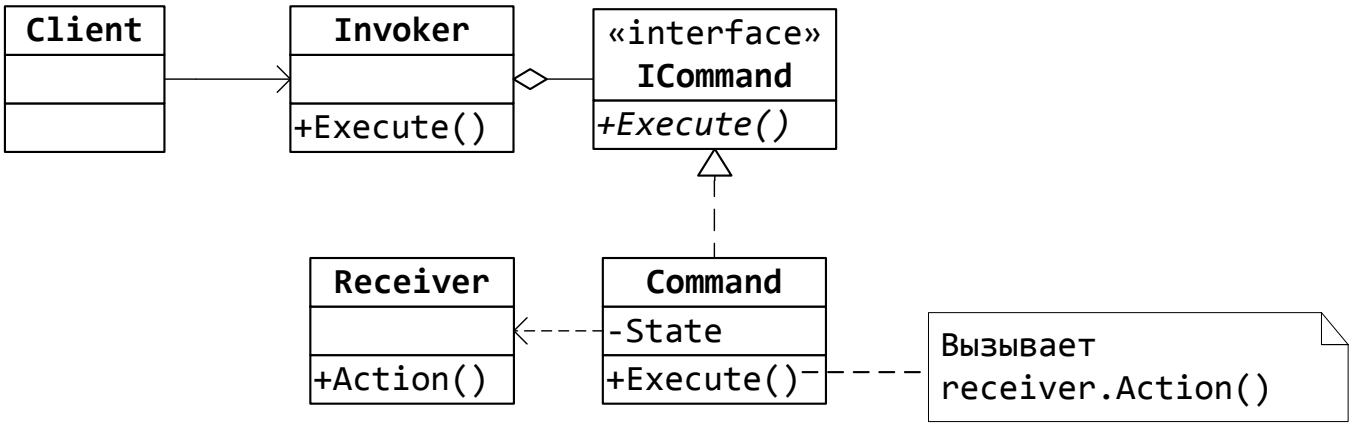


Рис. 19. Дизайн шаблона Команда.

Итератор (Iterator)

Назначение шаблона Итератор – предоставить последовательный доступ к элементам коллекции без информации о её внутреннем устройстве. Дополнительно шаблон может реализовывать функционал по фильтрации элементов коллекции.

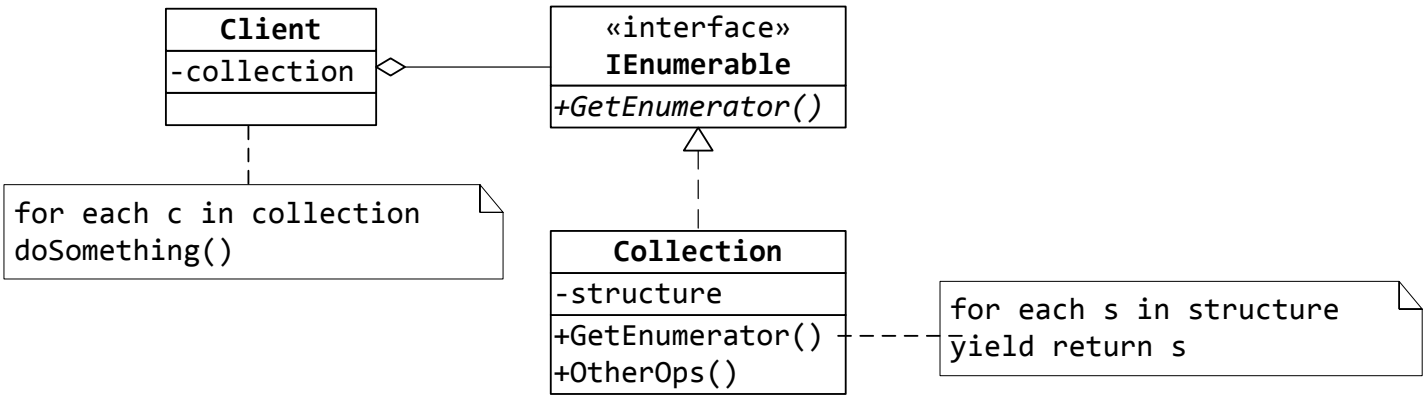


Рис. 19. Шаблон Итератор.

При практической реализации шаблона Итератор на C#, безусловно, используются такие возможности, как перечислители, интерфейсы IEnumerable и IEnumerator, оператор yield.

Посредник (Mediator)

Шаблон Посредник служит для обеспечения коммуникации между объектами. Этот шаблон также инкапсулирует протокол, которому должна удовлетворять процедура коммуникации.

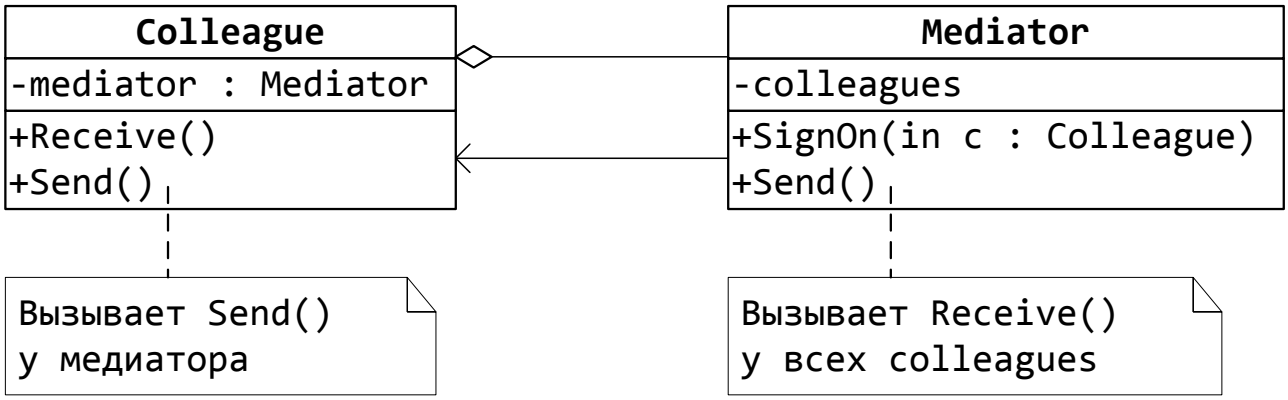


Рис. 21. Дизайн шаблона Посредник.

Дизайн шаблона посредник предполагает наличие двух выделенных классов, использующих сообщения для взаимного обмена информацией: Colleague (коллега) и Mediator (посредник). Объект Colleague регистрирует объект Mediator и сохраняет его в своём внутреннем поле. При регистрации посреднику предоставляется функция обратного вызова Receive(). В свою очередь, посредник поддерживает список зарегистрировавших его объектов. Как только один из объектов Colleague вызывает свой метод Send(), посредник вызывает у остальных зарегистрированных объектов метод Receive(). При практической реализации шаблона Посредник в .NET Framework список

зарегистрированных объектов можно поддерживать в виде словаря, а можно использовать для рассылки сообщений возможности групповых делегатов.

Посетитель (Visitor)

Шаблон Посетитель служит для выполнения операций над всеми объектами, объединёнными в некоторую структуру. При этом выполняемые операции не обязательно должны являться частью объектов структуры.

Основная идея шаблона Посетитель состоит в том, что каждый элемент объектной структуры содержит метод Accept(), который принимает на вход в качестве аргумента специальный объект, Посетитель, реализующий заранее известный интерфейс. Этот интерфейс содержит по одному методу Visit() для каждого типа узла. Метод Accept() в каждом узле должен вызывать методы Visit() для осуществления навигации по структуре.

Интерпретатор (Interpreter)

Шаблон Интерпретатор задаёт способ вычисления выражений, записанных на неком специальном языке. Основная идея шаблона – определение класса для каждого символа языка. Вследствие этого выражение на языке представляется в виде композитной объектной структуры.

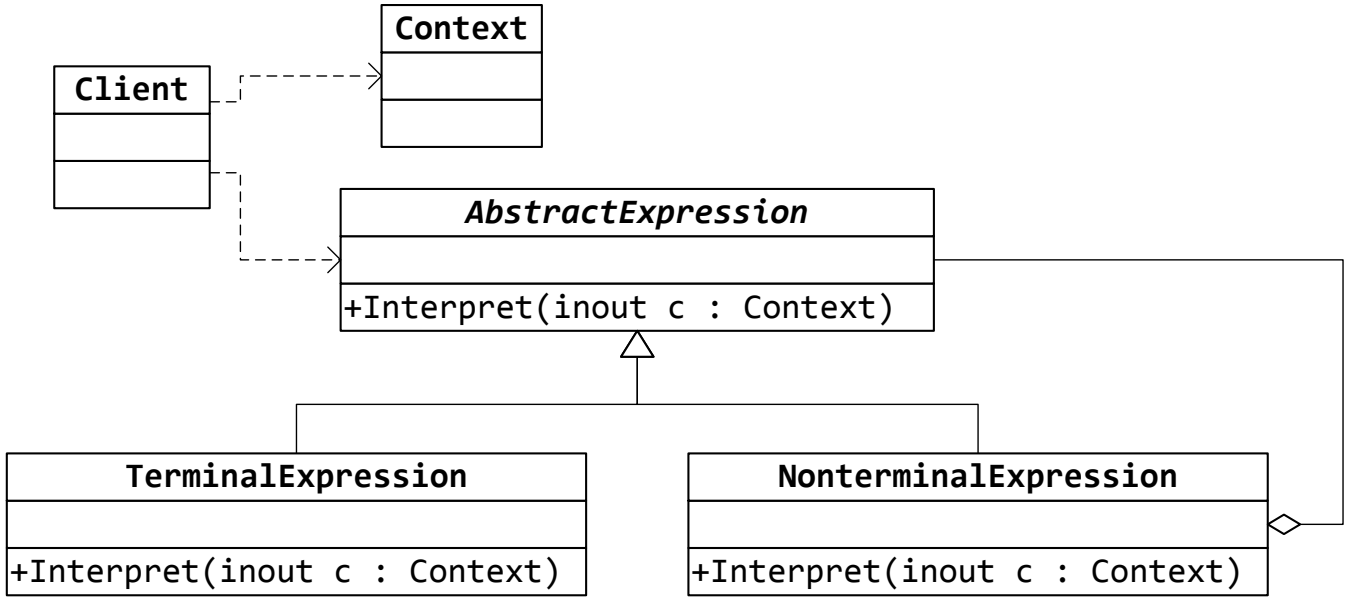


Рис. 23. Структура шаблона Интерпретатор.

Хранитель (Memento)

Этот шаблон используется для захвата и сохранения внутреннего состояния объекта с возможностью дальнейшего восстановления состояния. Важной особенностью шаблона Хранитель является то, что он позволяет

сохранить внутреннее состояние объекта без нарушения правил инкапсуляции.

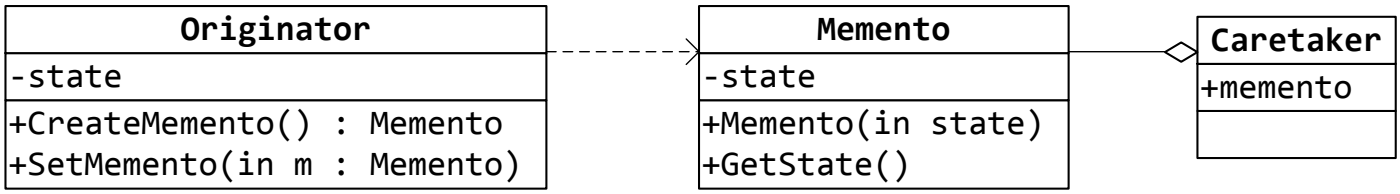


Рис. 23. UML-диаграмма шаблона Хранитель.

На рис. 23 показана UML-диаграмма шаблона хранитель. Ниже описаны элементы диаграммы.

- **Составитель**. Этот тот класс, состояние которого предполагается сохранять. Составитель включает метод CreateMemento(), который используется для генерации объекта-хранителя, содержащего состояние составителя. Также составитель содержит метод SetMemento(), восстанавливающий состояние по хранителю.
- **Хранитель**. Объект-хранитель содержит информацию о состоянии Составителя. То, что содержит Хранитель, контролируется Составителем. Хранитель защищает сохраняемую информацию, предоставляя усеченный внешний интерфейс, не позволяющий эту информацию редактировать.
- **Смотритель**. Класс Смотритель используется для хранения объектов-хранителей для дальнейшего использования. Смотритель используется только для хранения – он не просматривает и не изменяет внутренне состояние Хранителей. Смотритель может содержать как единичный объект-хранитель, так и коллекцию хранителей.

24. Архитектура ПО. Примеры типовых архитектур.

Архитектура программного обеспечения – это представление, которое дает информацию о компонентах ПО, обязанностях отдельных компонент и правилах организации взаимосвязей между компонентами.

Набор принципов, используемых в архитектуре, формирует *архитектурный стиль*. Применение архитектурного стиля сродни употреблению шаблона проектирования, но не на уровне компонента (модуля или класса), а на уровне всей создаваемой системы ПО.

Основные архитектурные стили

Архитектурный стиль	**Описание**
Клиент-сервер	Разделение системы на два приложения – клиент и сервер. При работе клиент посылает запросы на обслуживание серверу
Архитектура, основанная на использовании компонентов	Разделение системы на компоненты, которые могут быть повторно использованы и не зависят друг от друга. Каждый компонент снабжается известным интерфейсом для коммуникаций
Многоуровневая архитектура	Разделение функций приложения на группы (уровни), которые организованы в виде стекового набора.
Шина сообщений	Система, которая может посылать и передавать информационные сообщения в определенном формате по общему коммуникационному каналу. Благодаря этому организуется взаимодействие систем без указаний конкретных получателей сообщений
N-звеньевая/3-звеньевая архитектура	Разделение функций подобно архитектуре, основанной на уровнях. Однако группировка происходит не только на логическом, но и на физическом уровне – отдельным группам соответствует отдельный компьютер (сервер, кластер)
Объектно-ориентированная архитектура	Представление системы в виде набора взаимодействующих объектов
Выделенное представление	Выделение в системе отдельных групп функций для взаимодействия с пользователями и обработки данных
Архитектура, ориентированная на сервисы	Каждый компонент системы представлен в виде независимого сервиса, предоставляющего свои функции по стандартному протоколу

«Клиент-сервер»

Архитектурный стиль «клиент-сервер» (client/server) описывает отношение между двумя компьютерными программами, в котором одна программа – *клиент* – выполняет запросы к другой программе – *серверу*. На модели «клиент-сервер» созданы приложения для работы с базами данных, электронной почтой и для доступа к веб-ресурсам.

Архитектура, основанная на использовании компонентов

Компонентный стиль в архитектуре ПО сосредоточен на выделении отдельных компонент и организации взаимодействия между ними. Этот стиль решает задачи структурирования приложений.

Многоуровневая архитектура

Многоуровневая архитектура сосредоточена на иерархическом распределении отдельных частей системы при помощи эффективного разделения отношений. Каждая часть соотносится с определенным *уровнем* (layer), для каждого уровня заданы выполняемые им функции, уровни выстроены в стековую структуру (то есть находятся один поверх другого). Например, типичная архитектура для веб-приложений включает уровень представления (компоненты пользовательского интерфейса), уровень бизнес-логики (обработка данных) и уровень доступа к данным.

Принципы многоуровневой архитектуры:

- Проектирование четко устанавливает разграничение функций между уровнями.
- Нижние уровни не зависимы от верхних уровней. Это позволяет использовать компоненты одного уровня в разных приложениях.
- Верхние уровни вызывают функции нижних уровней, но при этом взаимодействуют только соседние уровни иерархии.

Использование многоуровневой архитектуры обеспечивает следующие преимущества:

- **Изоляция**. Разработка и обновление ПО могут быть изолированы рамками одного уровня.
- **Производительность**. Распределение уровней на отдельные физические компьютеры повышает производительность и отказоустойчивость.
- **Тестируемость**. Уровни допускаю независимое тестирование.

Многоуровневая архитектура активно применяется при создании бизнес-

приложений и сайтов, особенно приложений масштаба предприятия. При этом используется следующий набор уровней (рис. 24).

1. *Уровень представления* (presentation layer) – ответственен за взаимодействие с пользователем, ввод и вывод информации.
2. *Бизнес уровень* или *уровень бизнес-логики* (business logic layer) – обрабатывает информацию, реализуя конкретные бизнес-правила.
3. *Уровень доступа к данным* (data access layer) – обеспечивает загрузку и сохранение информации, используя источник данных (файл, база данных) или внешний сервис.

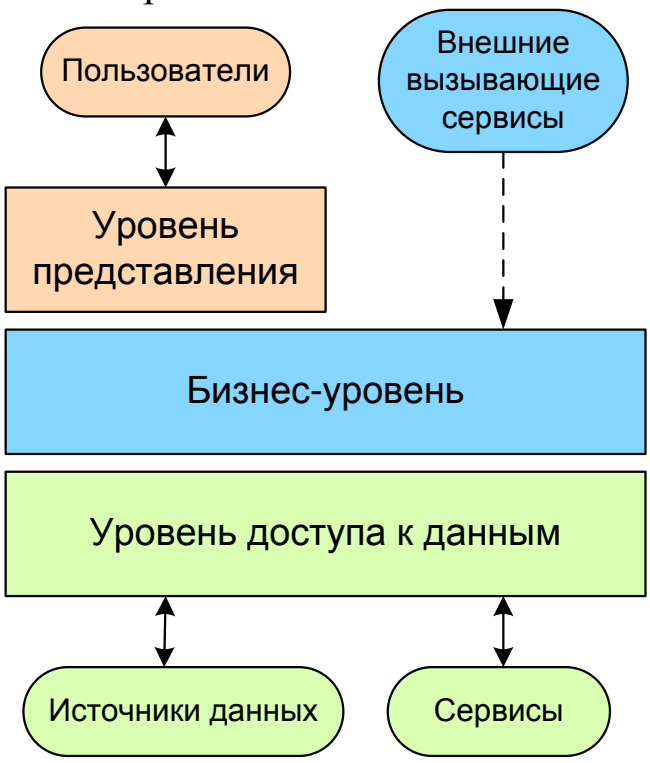


Рис. 24. Уровни бизнес-приложения.

Для каждого уровня дополнительно можно выделить типичный набор компонент (рис. 25). Заметим, что не все из перечисленных компонент (и даже уровней) должны присутствовать в каждом бизнес-приложении.

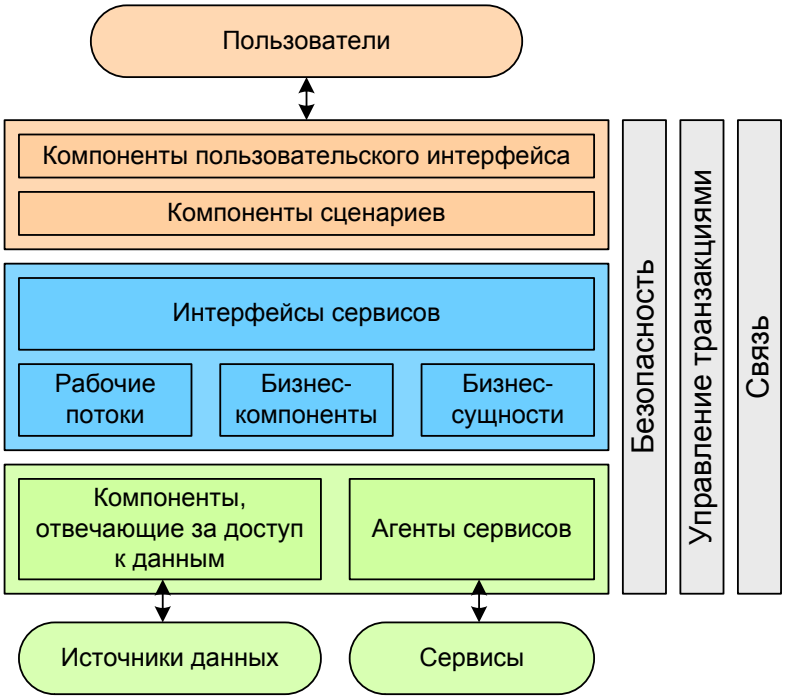


Рис. 25. Компоненты отдельных уровней.

Шина сообщений

Архитектура, основанная на шине сообщений, подразумевает наличие общего коммуникационного канала, используя который компоненты обмениваются информацией. Компонент помещает сообщение в коммуникационный канал, после этого сообщение рассылается всем заинтересованным компонентам. Данная архитектура направлена на решение коммуникационных задач.

Выделенное представление

Выделенное представление – это стиль обработки запросов или действий пользователя, а также манипулирования элементами интерфейса и данными. Стиль подразумевает отделение элементов интерфейса от логики приложения.

В качестве примера использования выделенного представления рассмотрим шаблон *модель-представление-контроллер* (Model-View-Controller, MVC). Этот шаблон имеет три основных компонента (рис. 26):

1. Модель представляет данные, с которыми работает приложение. Модель также реализовывает логику обработки данных согласно заданным бизнес-правилам и обеспечивает чтение и сохранение данных во внешних источниках.
2. Представление обеспечивает способ отображение данных модели.

3. Контроллер обрабатывает внешние запросы и координирует изменение модели и актуальность представления.

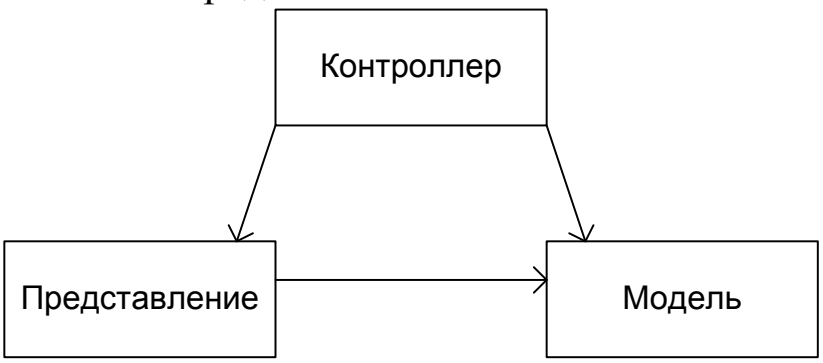


Рис. 26. Модель-представление-контроллер.

Важно отметить, что как представление, так и контроллер зависят от модели. Однако модель не зависит ни от представления, ни от контроллера. Это одно из ключевых достоинств подобного разделения. Оно позволяет строить модель независимо от визуального представления, а также создавать несколько различных представлений и контроллеров для одной модели.

N-звеньевая архитектура

Этот архитектурный стиль развертывания приложений подразумевает разделение компонентов на функциональные группы, подобно тому, как это происходит в многоуровневой архитектуре. Группа (реже – несколько групп) формируют *звено* – часть приложения, которая физически обособлена, то есть выполняется в отдельном процессе или на отдельном физическом компьютере.

Объектно-ориентированная архитектура

При использовании объектно-ориентированной архитектуры система воспринимается как набор взаимодействующих объектов. Каждый такой объект содержит данные и необходимое поведение, а коммуникация между объектами происходит через открытые интерфейсы и путем посылки и приема сообщений.

Архитектура, ориентированная на сервисы

Архитектура, ориентированная на сервисы (Service-Oriented Architecture, SOA), предоставляет требуемые функции в виде набора сервисов. Сервисы используют стандартные протоколы для вызова своих функций, публикации в сети и обнаружения. Отдельный сервис должен рассматриваться как независимое приложение, не как компонент или объект. Основной задачей при использовании SOA является определение интерфейса сервиса и схемы передаваемых при вызове сервиса данных.

25. Технология Windows Presentation Foundation (WPF) – общее описание.

Windows Presentation Foundation (WPF) – это технология для построения клиентских приложений Windows, являющаяся частью платформы .NET. WPF разработана как альтернатива технологии Windows Forms. Ниже перечислены основные особенности технологии WPF.

1. Собственные методы построения и рендеринга элементов. В Windows Forms классы для элементов управления делегируют функции отображения системным библиотекам, таким как user32.dll. В WPF любой элемент управления полностью строится (рисуется) самой WPF. Для аппаратного ускорения рендеринга применяется технология DirectX (рис. 1).

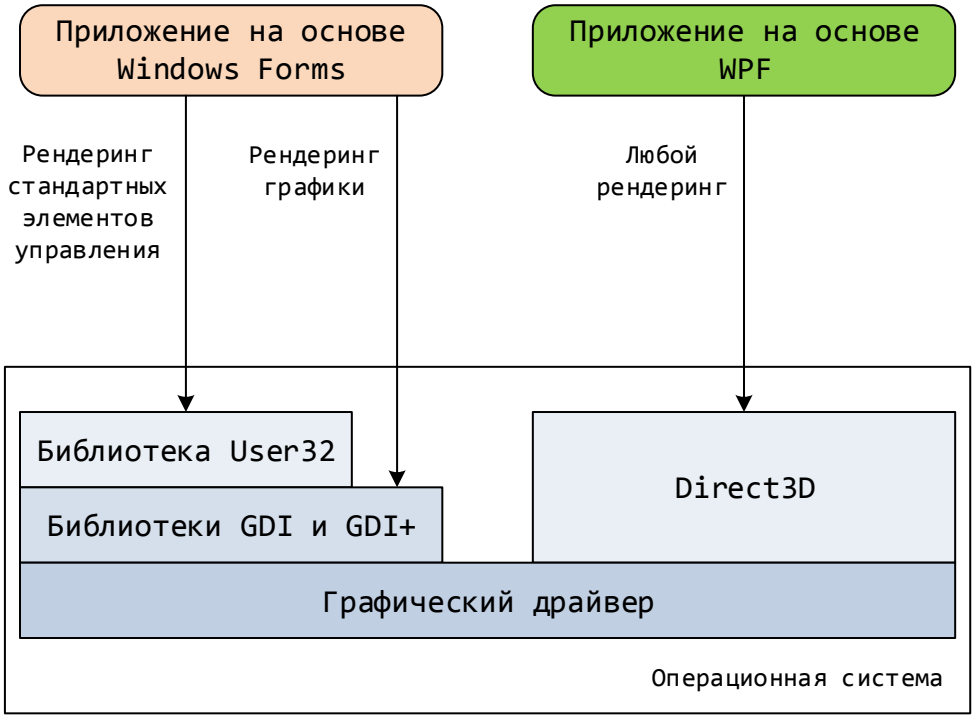


Рис. 1. Рендеринг в приложениях на основе Windows Forms и WPF.

2. Независимость от разрешения устройства вывода. Для указания размеров в WPF используется собственная единица измерения, равная 1/96 дюйма. Кроме этого, технология WPF ориентирована на использование не пиксельных, а векторных примитивов.

3. Декларативный пользовательский интерфейс. В WPF визуальное содержимое окна можно полностью описать на языке XAML. Это язык разметки, основанный на XML. Так как описание пользовательского

интерфейса отделено от кода, дизайнеры могут использовать профессиональные инструменты (например, Microsoft Expression Blend), чтобы редактировать файлы XAML, улучшая внешний вид приложения. Применение XAML является предпочтительным, но не обязательным – приложения WPF можно создавать, используя только код.

4. Веб-подобная модель компоновки. WPF поддерживает гибкий визуальный поток, размещающий элементы управления на основе их содержимого. В результате получается пользовательский интерфейс, который может быть адаптирован для отображения динамичного содержимого.

5. Стили и шаблоны. Стили стандартизируют форматирование и позволяют повторно использовать его по всему приложению. Шаблоны дают возможность изменить способ отображения любых элементов управления, даже таких основополагающих, как кнопки или поля ввода.

6. Анимация. В WPF анимация – неотъемлемая часть программного каркаса. Анимация определяется декларативными дескрипторами, и WPF запускает её в действие автоматически.

7. Приложения на основе страниц. В WPF можно строить приложения с кнопками навигации, которые позволяют перемещаться по коллекции страниц. Кроме этого, специальный тип приложения WPF – XBAP – может быть запущен внутри браузера.

Простейшее приложение WPF

Построим простейшее однооконное приложение WPF. Для этого создадим файл `Program.cs` и поместим в него следующий код:

```
using System;
using System.Windows;

public class Program
{
    [STAThread]
    public static void Main()
    {
        var myWindow = new Window();
        myWindow.Title = "WPF Program";
        myWindow.Content = "Hello, world";
        var myApp = new Application();
        myApp.Run(myWindow);
    }
```

```
    }
```

Проанализируем этот код. Пространство имён `System.Windows` содержит классы `Window` и `Application`, описывающее окно и приложение соответственно. Точка входа помечена атрибутом `[STAThread]`. Это обязательное условие для любого приложения WPF, оно связано с моделью многопоточности WPF. В методе `Main()` создаётся и настраивается объект окна, затем создаётся объект приложения. Вызов метода `Run()` приводит к отображению окна и запуску цикла обработки событий (окно ждёт действий пользователя). Чтобы скомпилировать приложение, необходимо указать ссылки на стандартные сборки `PresentationCore.dll`, `PresentationFramework.dll`, `System.Xaml.dll` и `WindowsBase.dll`.

Отметим, что приложение допускает другую организацию. Вместо настройки объекта класса `Window` можно создать наследник этого класса и выполнить настройку в конструкторе наследника или в специальном методе:

```
    // наследник класса Window, описывающий пользовательское окно
    public class MainWindow : Window
    {
        public MainWindow()
        {
            Title = "WPF Program";
            Content = "Hello, world";
        }
    }
```

В Visual Studio приложениям WPF соответствует отдельный шаблон проекта. Этот шаблон ориентирован на использование XAML, поэтому в случае однооконного приложения будет создан следующий набор файлов:

– файл `MainWindow.xaml.cs` на языке C# и `MainWindow.xaml` на языке XAML описывают класс `MainWindow`, являющийся наследником класса `Window`;

– файлы `App.xaml.cs` и `App.xaml` описывают класс `App`, наследник класса `Application`.

Ниже приведён файл `MainWindow.xaml` для простейшего окна:

```
    <Window x:Class="WpfApplication.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      Title="WPF Program" Height="250" Width="400">
```

```
    <!-- содержимое окна -->
    Hello, world
</Window>
```

Visual Studio выполняет компиляцию проекта, созданного по шаблону WPF, в два этапа. Вначале для каждого файла XAML генерируется два файла, сохраняемых в подкаталогах `obj\Debug` или `obj\Release` (в зависимости от цели компиляции):

1. файл с расширением `*.baml` (BAML-файл) – двоичное представление XAML-файла, внедряемое в сборку в виде ресурса;
2. файл с расширением `*.g.cs` – разделяемый класс, который соответствует XAML-описанию. Этот класс содержит поля для всех именованных элементов XAML и реализацию метода `InitializeComponent()`, загружающего BAML-данные из ресурсов сборки. Кроме этого, класс содержит метод, подключающий все обработчики событий.

На втором этапе сгенерированные файлы компилируются вместе с исходными файлами C# в единую сборку (рис. 2).

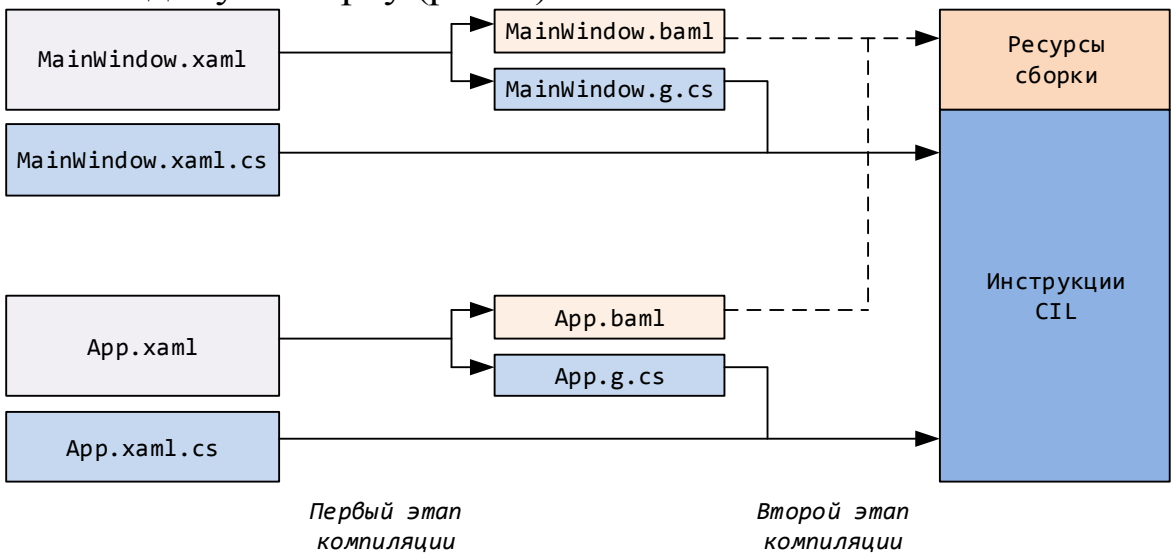


Рис. 2. Компиляция приложения WPF в Visual Studio.

26. Язык XAML

Расширяемый язык разметки приложений (eXtensible Application Markup Language, XAML[8]) – это язык для представления дерева объектов .NET, основанный на XML. Данные XAML превращаются в дерево объектов при помощи *анализатора XAML* (XAML parser). Основное назначение XAML –

[8] Произносится как ['zæməl].

описание пользовательских интерфейсов в приложениях WPF. Однако XAML используется и в других технологиях, в частности, в Silverlight.
Рассмотрим основные правила XAML. Документ XAML записан в формате XML. Это означает, что имена элементов XAML чувствительны к регистру, нужна правильная вложенность элементов, а некоторые символы требуют особого обозначения (например, `&` – это символ `&`). Кроме этого, XAML по умолчанию игнорирует лишние пробельные символы (однако это поведение изменяется установкой у элемента атрибута `xml:space="preserve"`).
Объектные элементы XAML описывают объект некоторого типа платформы .NET и задают значения открытых свойств и полей объекта. Имя элемента указывает на тип объекта. Ниже приведено описание XAML для объекта класса `Button` (кнопка), а также эквивалентный код на языке C#:

```
<!-- определение объекта в XAML -->
<Button Width="100">
  I am a Button
</Button>

// определение объекта в коде
Button b = new Button();
b.Width = 100;
b.Content = "I am a Button";
```

Типы .NET обычно вложены в пространства имён. В XAML пространству имён .NET ставится в соответствие пространство имён XML. Для этого используется следующий синтаксис:

```
xmlns:префикс="clr-namespace:пространство-имён"
```

При необходимости указывается сборка, содержащая пространство имён:

```
xmlns:префикс="clr-namespace:пространство-имён;assembly=имя-сборки"
```

Для нужд WPF зарезервировано два пространства имён XML:
1. `http://schemas.microsoft.com/winfx/2006/xaml/presentation` – обычно является пространством имён по умолчанию (указывается без префикса) и соответствует набору пространств имён .NET с типами WPF (эти пространства имён имеют вид `System.Windows.*`).
2. `http://schemas.microsoft.com/winfx/2006/xaml` – отвечает пространству имён `System.Windows.Markup`, а также позволяет выделить директивы (указания) для анализатора XAML. Пространству имён анализатора XAML по традиции ставят в соответствие префикс `x`.

```
    <!-- у корневого элемента Window заданы три пространства имён -->
    <Window
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      xmlns:sys="clr-namespace:System;assembly=mscorlib">
```

Для установки значений свойств объекта в XAML можно использовать атрибуты XML, элементы свойств и содержимое элемента. При использовании *атрибутов* указывается имя свойства и значение свойства в виде строки:

```
    <!-- задаём у кнопки красный фон -->
    <Button Background="Red" />
```

Анализатор XAML применяет для преобразования строки в значение свойства специальные *конвертеры типов* (конвертеры не используются для строк, чисел и элементов перечислений). Приведённый выше фрагмент XAML эквивалентен следующему коду на C#:

```
    // TypeConverter и TypeDescriptor определены в
System.ComponentModel
    var b = new Button();
    TypeConverter convert = TypeDescriptor.GetConverter(typeof
(Brush));
    b.Background = (Brush) convert.ConvertFromInvariantString("Red");
```

Элемент свойства вложен в объектный элемент и имеет вид `<имя-типа.имя-свойства>`. Содержимое элемента свойства рассматривается как значение свойства. Обычно элементы свойств используются для значений, являющихся объектами.

```
    <Button>
      <Button.Width>100</Button.Width>
      <Button.Background>Red</Button.Background>
    </Button>
```

Тип, соответствующий объектному элементу, может быть помечен атрибутом `[ContentProperty]` с указанием имени *свойства содержимого*. В этом случае анализатор XAML рассматривает содержимое объектного элемента (за исключением элементов свойств) как значение для свойства содержимого. Например, в классе `ContentControl` (он является базовым для класса `Button`) свойством содержимого является `Content`:

```
    [System.Windows.Markup.ContentProperty("Content")]
    public class ContentControl
    {
```

```
        public object Content { get; set; }
        // другие элементы класса ContentControl не показаны
    }
```

Это означает, что следующие два фрагмента XAML эквиваленты:

```
<Button Content="Click me!" />
<Button>Click me!</Button>
```

Если тип реализует интерфейсы `IList` или `IDictionary`, при описании объекта этого типа в XAML дочерние элементы автоматически добавляются в соответствующую коллекцию. Например, свойство `Items` класса `ListBox` имеет тип `ItemCollection`, а этот класс реализует интерфейс `IList`:

```
<ListBox>
  <ListBox.Items>
    <ListBoxItem Content="Item 1" />
    <ListBoxItem Content="Item 2" />
  </ListBox.Items>
</ListBox>
```

Кроме этого, `Items` – это свойство содержимого для `ListBox`, а значит, приведённое XAML-описание можно упростить:

```
<ListBox>
  <ListBoxItem Content="Item 1" />
  <ListBoxItem Content="Item 2" />
</ListBox>
```

Во всех предыдущих примерах использовалось конкретное указание значения свойства. Механизм *расширений разметки* (markup extensions) позволяет вычислять значение свойства при преобразовании XAML в дерево объектов. Технически, любое расширение разметки – это класс, унаследованный от `System.Windows.Markup.MarkupExtension` и перекрывающий функцию `ProvideValue()`. Встретив расширение разметки, анализатор XAML генерирует код, который создаёт объект расширения разметки и вызывает `ProvideValue()` для получения значения. Приведём пример расширения разметки:

```
using System;
using System.Windows.Markup;

namespace MarkupExtensions
{
    public class ShowTimeExtension : MarkupExtension
    {
```

```
            public string Header { get; set; }

            public ShowTimeExtension() { }

            public override object ProvideValue(IServiceProvider sp)
            {
                return string.Format("{0}: {1}", Header,
DateTime.Now);
            }
        }
    }
```

Если расширение разметки используется в XAML как значение атрибута, то оно записывается в фигурных скобках (если имя расширения имеет суффикс *`Extension`*, этот суффикс можно не указывать). В фигурных скобках также перечисляются через запятую аргументы конструктора расширения и пары для настройки свойств расширения в виде *`свойство=значение`*.

```
    <Window xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:local="clr-namespace:MarkupExtensions">
      <StackPanel>
        <Button Content="{local:ShowTime}" />
        <Button Content="{local:ShowTime Header=Time}" />
      </StackPanel>
    </Window>
```

Расширения разметки могут применяться как значения элементов свойств:

```
    <Button>
      <Button.Content>
        <local:ShowTime Header="Time" />
      </Button.Content>
    </Button>
```

В табл. 1 представлены стандартные расширения разметки, доступные после подключения пространства имён `System.Windows.Markup`.

Таблица 1

Расширения разметки из `System.Windows.Markup`

Имя	Описание, пример использования
`x:Array`	Представляет массив. Дочерние элементы становятся элементами массива `<x:Array Type="{x:Type Button}">`

	<Button /> <Button /> </x:Array>
x:Null	Представляет значение null Style="{x:Null}"
x:Reference	Используется для ссылки на ранее объявленный элемент <TextBox Name="customer" /> <Label Target="{x:Reference customer}" />
x:Static	Представляет статическое свойство, поле или константу Height="{x:Static SystemParameters.IconHeight}"
x:Type	Аналог применения оператора typeof из языка C#

Рассмотрим некоторые директивы анализатора XAML, применяемые в WPF. Анализатор генерирует код, выполняющий по документу XAML создание и настройку объектов. Действия с объектами (в частности, обработчики событий) обычно описываются в отдельном классе. Чтобы связать этот класс с документом XAML используется директива-атрибут `x:Class`. Этот атрибут применяется только к корневому элементу и содержит имя класса, являющегося наследником класса корневого элемента:

```
<!-- у корневого элемента Window задан атрибут x:Class -->
<Window x:Class="WpfApplication.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
```

Чтобы сослаться на объект в коде, этот объект должен иметь имя. Для указания имени объекта используется директива-атрибут `x:Name`:

```
<Button x:Name="btn" Content="Click me!" />
```

Заметим, что многие элементы управления WPF имеют свойство `Name`. Анализатор XAML использует соглашение, по которому задание свойства `Name` эквивалентно указанию директивы-атрибута `x:Name`.

Существует возможность встроить фрагмент кода в XAML-файл. Для этого используется директива-элемент `x:Code`. Такой элемент должен быть непосредственно вложен в корневой элемент, у которого имеется атрибут `x:Class`.

```
<Window x:Class="WpfApplication.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

      <Button x:Name="btn" Click="btn_click" Content="Click me!" />
```

```
    <x:Code>
      <![CDATA[
        void btn_click(object sender, RoutedEventArgs e)
        {
            btn.Content = "Inline Code Works!";
        }
      ]]>
    </x:Code>
</Window>
```

Директива-атрибут `x:Key` применяется при описании дочерних элементов объекта-словаря, и позволяет указать ключ словаря для элемента:

```
<Window.Resources>
  <SolidColorBrush x:Key="borderBrush" Color="Red" />
  <SolidColorBrush x:Key="textBrush" Color="Black" />
</Window.Resources>
```

27. Основные элементы управления WPF.

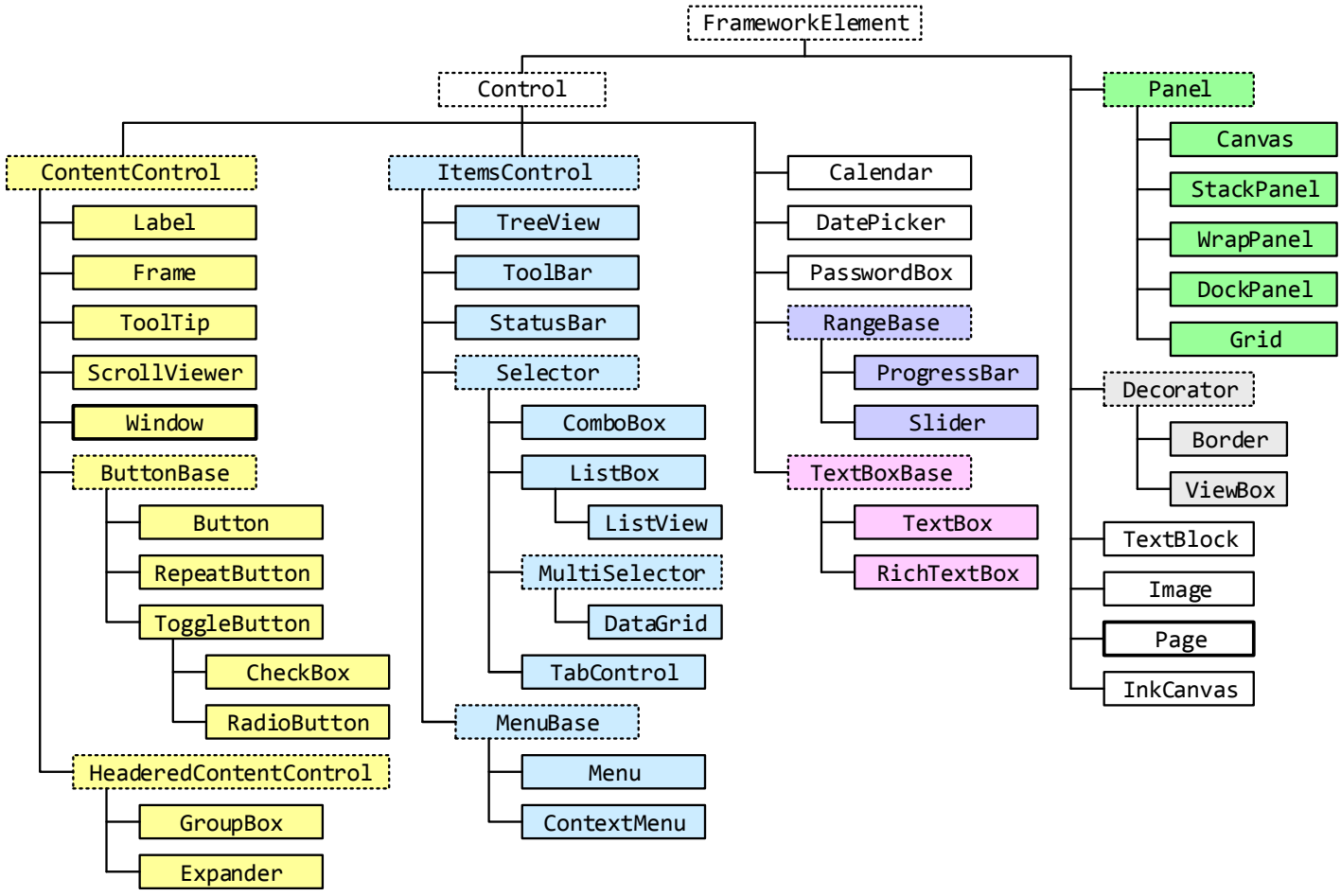


Рис. 12. Стандартные элементы WPF.

Формально, к элементам управления относятся классы, унаследованные от класса `Control`. Рассмотрим некоторые собственные свойства этого класса, разбив их на группы:

1. Внешний вид. Свойство `Template` задаёт шаблон, полностью определяющий внешний вид элемента управления.

2. Позиционирование содержимого. За позиционирование ответственны свойства `Padding`, `HorizontalContentAlignment` и `VerticalContentAlignment`.

3. Цвета и окантовка.

`Foreground` – кисть для отображения содержимого;
`Background` – кисть для фона элемента;
`BorderBrush` и `BorderThickness` – кисть и ширина окантовки.

4. Шрифт содержимого.

`FontFamily` – имя семейства шрифтов (например, `"Arial"`);
`FontSize` – размер шрифта в единицах WPF;
`FontStyle` – наклон текста. Нужный набор можно получить из статических свойств класса `FontStyles`: `Normal`, `Italic` или `Oblique`;
`FontWeight` – вес (т.е. толщина) текста. Предварительно заданный набор можно получить, используя статические свойства класса `FontWeight`;
`FontStretch` – величина, на которую растягивается или сжимается текст.

5. Табуляция. Целочисленное свойство `TabIndex` и булево `IsTabStop`.

7.1. Элементы управления содержимым

Элементы управления содержимым – это элементы, допускающие размещение единственного дочернего элемента, представляющего их содержимое. Элементы управления содержимым наследуются от класса `ContentControl`. У этого класса имеется объектное свойство `Content` и булево свойство `HasContent`. Если в `Content` помещается объект-наследник `UIElement`, для отображения вызывается метод `OnRender()` этого объекта. Для объектов других типов делается попытка применить шаблон данных. Если шаблон данных не задан в свойстве `ContentTemplate` или в ресурсах приложения, у дочернего элемента вызывается метод `ToString()`, а результат обрамляется в элемент `TextBlock`.

Элементы управления содержимым можно разбить на три подгруппы: *кнопки*, *простые контейнеры*, *контейнеры с заголовком*.

Все кнопки наследуются от абстрактного класса `ButtonBase`.

Элемент управления Button определяет обычную кнопку, которая может быть кнопкой по умолчанию или кнопкой отмены.
Элемент RepeatButton – кнопка, непрерывно генерирующая событие Click, если на ней задержать нажатие.
ToggleButton – кнопка с «залипанием». Будучи нажатой, она остаётся в таком состоянии, пока по ней не произведут повторное нажатие.
Флажок-переключатель формально относится к кнопкам и представлен классом CheckBox. Этот класс является прямым наследником ToggleButton с переписанным шаблоном внешнего вида.
Элемент управления RadioButton также унаследован от ToggleButton. Особенность RadioButton – поддержка группового поведения, при котором установка одного переключателя сбрасывает остальные. По умолчанию к одной группе относятся переключатели, имеющие общий родительский контейнер. Можно задать строковое свойство GroupName, чтобы определить собственный способ выделения группы переключателей.

```
<StackPanel>
  <RadioButton Margin="2" GroupName="Gender" Content="Male"
               IsChecked="True" />
  <RadioButton Margin="2" GroupName="Gender" Content="Female" />

  <RadioButton Margin="2" GroupName="Status" Content="Single"
               IsChecked="True" />
  <RadioButton Margin="2" GroupName="Status" Content="Married" />
  <RadioButton Margin="2" GroupName="Status" Content="Divorced" />
  <RadioButton Margin="2" GroupName="Status" Content="Widower" />

  <CheckBox Margin="2" Content="Has children" />

  <Button Margin="8" Click="btn_Click" Content="Save Profile" />
</StackPanel>
```

Перейдём к разбору простых контейнеров, которые (согласно своему названию) предназначены для простого обрамления дочернего содержимого.
Один из простых контейнеров – элемент Label. Обычно он используется для представления текста, но в WPF его содержимое может быть любым.

```
<Label Target="{x:Reference userName}" Content="_User:" />
<TextBox x:Name="userName" />
```

Элемент управления ToolTip отображает своё содержимое в виде

всплывающей подсказки. `ToolTip` может содержать любые элементы управления, но с ними нельзя взаимодействовать.

```
    <CheckBox Margin="10" Content="Simple CheckBox"
              ToolTipService.VerticalOffset="15">
      <CheckBox.ToolTip>
        <StackPanel>
          <Label FontWeight="Bold" Background="Blue"
Foreground="White"
                 Content="The CheckBox" />
          <TextBlock TextWrapping="Wrap" Width="200">
            CheckBox is a familiar control.
            In WPF it is a ToggleButton styled differently.
          </TextBlock>
        </StackPanel>
      </CheckBox.ToolTip>
    </CheckBox>
```

Элемент управления `Frame` предназначен для изолирования своего содержимого от остальной части пользовательского интерфейса. Его свойство `Source` указывает на отображаемую HTML-страницу или XAML-страницу:

```
    <Frame Source="http://www.pinvoke.net" />
```

Последняя группа элементов управления содержимым – контейнеры с заголовком. Эти элементы наследуются от класса `HeaderedContentControl`, который добавляет объектное свойство `Header` к классу `ContentControl`. Простейший контейнер с заголовком представлен элементом `GroupBox`. Следующий пример демонстрирует использование этого элемента управления для группировки нескольких флажков:

```
    <GroupBox Margin="5" Header="Grammar">
      <StackPanel>
        <CheckBox Content="Check grammar as you type" />
        <CheckBox Content="Hide grammatical errors" />
        <CheckBox Content="Check grammar with spelling" />
      </StackPanel>
    </GroupBox>
```

Элемент управления `Expander` напоминает `GroupBox`, но содержит в заголовке особую кнопку, которая позволяет спрятать и показать содержимое контейнера.

```
    <Expander Header="Grammar" ExpandDirection="Right">
      <StackPanel Margin="10">
```

```
      <CheckBox Content="Check grammar as you type" />
      <CheckBox Content="Hide grammatical errors" />
      <CheckBox Content="Check grammar with spelling" />
    </StackPanel>
  </Expander>
```

7.2. Списковые элементы управления

Списковые элементы управления (далее для краткости – *списки*) могут хранить и представлять коллекцию элементов. Для большинства списков доступна гибкая настройка внешнего вида, а также функции фильтрации, группировки и сортировки данных, однако в данном параграфе рассматриваются только простейшие аспекты работы со списками.

Все списки наследуются от класса `ItemsControl`, который определяет несколько полезных свойств:

`Items` – коллекция типа `ItemCollection`, сохраняющая элементы списка (свойство доступно только для чтения).

`ItemsSource` – свойство позволяет связать список с набором данных. Свойство доступно для установки во время выполнения приложения.

`HasItems` – булево свойство только для чтения. Показывает, имеются ли элементы в списке.

`DisplayMemberPath` – строковое свойство, определяющее отображаемое значение для элемента списка. `DisplayMemberPath` поддерживает синтаксис, который называется *путь к свойству*.

Списковые элементы управления можно разделить на три подгруппы: *селекторы*, *меню*, *списки без категории*. Селекторы допускают индексацию и выбор элементов списка. К селекторам относятся элементы управления `ListBox`, `ComboBox`, `TabControl`, `ListView` и `DataGrid`. Все селекторы наследуются от класса `Selector`, который предоставляет следующие свойства и события:

`SelectedItem` – первый выбранный элемент списка.

`SelectedIndex` – индекс выбранного элемента (-1, если ничего не выбрано).

`SelectedValuePath` – путь к свойству для значения выбранного элемента.

`SelectedValue` – значение выбранного элемента. Если не установлено свойство `SelectedValuePath`, это значение совпадает с `SelectedItem`.

`SelectionChanged` – событие, генерируемое при изменении `SelectedItem`.

В классе `Selector` определены два присоединённых булевых свойства,

используемых с элементами списка, – `IsSelected` и `IsSelectionActive`. Второе свойство является свойством только для чтения и позволяет узнать, установлен ли на выделенном элементе фокус. Кроме этого, класс `Selector` имеет два присоединённых события – `Selected` и `Unselected`.

Элемент управления `ListBox` отображает список и допускает множественный выбор элементов.

```
<ListBox SelectionMode="Multiple" SelectedIndex="2">
  <TextBlock Margin="5" Text="Text Block" />
  <Button Margin="5" Content="Button" />
  <CheckBox Margin="5" Content="Check Box" />
  <RadioButton Margin="5" Content="Radio Button" />
  <Label Margin="5" Content="Label" />
</ListBox>
```

Класс `ComboBox` позволяет выбрать один элемент из списка, отображая текущий выбор и раскрывая список элементов по требованию.

Элемент управления `TabControl` – это простой набор страниц с закладками. По умолчанию закладки расположены вверху.

```
<TabControl SelectedIndex="1" TabStripPlacement="Left">
  <TabItem Header="Tab 1">Content for Tab 1</TabItem>
  <TabItem Header="Tab 2">Content for Tab 2</TabItem>
  <TabItem Header="Tab 3">Content for Tab 3</TabItem>
</TabControl>
```

Класс `ListView` унаследован от `ListBox`. Этот список позволяет настроить свой внешний вид при помощи свойства.

```
<ListView xmlns:sys="clr-namespace:System;assembly=mscorlib">
  <ListView.View>
    <GridView>
      <GridViewColumn Header="Day" Width="40"
                      DisplayMemberBinding="{Binding Day}"/>
      <GridViewColumn Header="Month" Width="60"
                      DisplayMemberBinding="{Binding Month}"/>
      <GridViewColumn Header="Year" Width="60"
                      DisplayMemberBinding="{Binding Year}"/>
      <GridViewColumn Header="Day of Week"
                      DisplayMemberBinding="{Binding DayOfWeek}"/>
    </GridView>
  </ListView.View>
  <sys:DateTime>1/1/2012</sys:DateTime>
```

```
      <sys:DateTime>1/7/2012</sys:DateTime>
      <sys:DateTime>3/8/2012</sys:DateTime>
      <sys:DateTime>4/24/2012</sys:DateTime>
      <sys:DateTime>5/1/2012</sys:DateTime>
      <sys:DateTime>5/9/2012</sys:DateTime>
    </ListView>
```

Таблица `DataGrid` позволяет отображать и редактировать данные. Внешний вид `DataGrid` определяют колонки, которые хранятся в коллекции `Columns`.

```
    <DataGrid x:Name="dataGrid" AutoGenerateColumns="False">
      <DataGrid.Columns>
        <DataGridTextColumn Header="Name" Binding="{Binding Name}"/>
        <DataGridHyperlinkColumn Header="Website"
                                 Binding="{Binding Site}" />
        <DataGridCheckBoxColumn Header="Has kids?"
                                Binding="{Binding HasKids}" />
      </DataGrid.Columns>
    </DataGrid>

    public class Person
    {
        public string Name { get; set; }
        public Uri Site { get; set; }
        public bool HasKids { get; set; }
    }

    // код в конструкторе MainWindow - заполняем DataGrid
    var uri = new Uri("http://www.eden.com");
    var adam = new Person {Name = "Adam", Site = uri, HasKids = true};
    var eve = new Person {Name = "Eve", Site = uri, HasKids = true};
    dataGrid.ItemsSource = new List<Person> {adam, eve};
```

Перейдём к рассмотрению элементов управления из категории меню. Класс `Menu` может содержать коллекцию любых объектов, но ожидается, что будут использованы объекты `MenuItem` и `Separator`. `Separator` – простой элемент, представляющий разделитель пунктов меню. `MenuItem` – это контейнер с заголовком (наследник `HeaderedItemsControl`).

```
    <DockPanel LastChildFill="False">
      <Menu DockPanel.Dock="Top">
        <MenuItem Header="_File">
```

```
        <MenuItem Header="_New..." />
        <MenuItem Header="_Open..." />
        <Separator />
        <MenuItem Header="Sen_d To">
          <MenuItem Header="Mail Recipient" />
          <MenuItem Header="My Documents" />
        </MenuItem>
      </MenuItem>
      <MenuItem Header="_Edit" />
      <MenuItem Header="_View" />
    </Menu>
  </DockPanel>
```

Класс `Ribbon` (пространство имён `System.Windows.Controls.Ribbon` и одноимённая сборка) позволяет создать *ленту* – элемент интерфейса с меню, панелями инструментов и вкладками.

Класс `ContextMenu` является контейнером объектов `MenuItem` и служит для представления контекстного меню некоторого элемента.

Рассмотрим списковые элементы без категории. Элемент управления `TreeView` отображает иерархию данных в виде дерева с узлами, которые могут быть свёрнуты и раскрыты. Каждый элемент в `TreeView` является объектом `TreeViewItem`.

```
  <TreeView>
    <TreeViewItem Header="ItemsControl">
      <TreeViewItem Header="Selector">
        <TreeViewItem Header="ListBox" />
        <TreeViewItem Header="ComboBox" />
        <TreeViewItem Header="TabControl" />
      </TreeViewItem>
      <TreeViewItem Header="HeaderedItemsControl" >
        <TreeViewItem Header="MenuItem" />
        <TreeViewItem Header="TreeViewItem" />
      </TreeViewItem>
    </TreeViewItem>
  </TreeView>
```

Элемент управления `ToolBar` представляет панель инструментов. Он просто группирует кнопки, разделители (`Separator`) и другие дочерние элементы на одной панели. Хотя `ToolBar` можно разместить в произвольном месте,

обычно один или несколько таких объектов размещают внутри элемента ToolBarTray, что позволяет перетаскивать и переупорядочить панели инструментов. Элемент управления StatusBar группирует элементы подобно ToolBar. Обычно элемент StatusBar размещается в нижней части окна.

7.3. Прочие элементы управления

Рассмотрим оставшиеся стандартные элементы управления, разбив их на следующие подгруппы: *текстовые элементы*, *элементы для представления диапазона*, *элементы для работы с датами*.

К текстовым элементам относятся TextBlock, TextBox, RichTextBox, PasswordBox. TextBlock формально не является элементом управления, так как унаследован непосредственно от FrameworkElement. Он предназначен для отображения небольшой порции текста. Для текста доступны настройки шрифта, выравнивания, переноса, а сам текст может включать теги XAML XPS.

```
<StackPanel>
  <!-- текст в разметке -->
  <TextBlock x:Name="tb1" Text="Simple TextBlock" />
  <TextBlock x:Name="tb2" TextWrapping="Wrap" FontSize="20">
    <Bold>TextBlock</Bold> is designed to be
    <Italic>lightweight.</Italic>
  </TextBlock>
</StackPanel>

// текст в коде
tb1.Text = "Simple TextBlock";

tb2.Inlines.Clear();
tb2.Inlines.Add(new Bold(new Run("TextBlock")));
tb2.Inlines.Add(new Run(" is designed to be "));
tb2.Inlines.Add(new Italic(new Run("lightweight.")));
```

Рис. 23. Демонстрация возможностей TextBlock.

Элемент управления TextBox служит для отображения и ввода текста, заданного в строковом свойстве Text.

```
<TextBox SpellCheck.IsEnabled="True" Text="Erorr" />
```

Элемент управления RichTextBox – это «продвинутая» версия TextBox. Многие свойства у этих элементов общие, так как они унаследованы от одного

базового класса `TextBoxBase`. Содержимое `RichTextBox` сохраняется в свойстве `Document` типа `FlowDocument`, который создан для поддержки XPS.
Элемент управления `PasswordBox` предназначен для ввода паролей.
Элементы для представления диапазона `ProgressBar` и `Slider` хранят и отображают в некой форме числовое значение, попадающее в заданный диапазон.
`ProgressBar` обычно используют для визуализации процесса выполнения длительной операции.
Элемент управления `Slider` – это слайдер (ползунок) с возможностью ручной установки значения из диапазона.

```
<StackPanel>
  <ProgressBar Value="80" Height="20" Margin="10"/>
  <Slider Maximum="30" Value="25"
          TickPlacement="BottomRight" TickFrequency="2"
          IsSelectionRangeEnabled="True"
          SelectionStart="10" SelectionEnd="20"/>
  <Slider Height="100" Maximum="30" HorizontalAlignment="Center"
          Orientation="Vertical" />
</StackPanel>
```

Элементами для работы с датами являются `Calendar` и `DatePicker`.
`Calendar` отображает небольшой календарь с возможностью клавиатурной навигации и выбора
Элемент `DatePicker` позволяет задать дату, набирая её с клавиатуры или применяя выпадающий элемент `Calendar`.

```
<StackPanel Orientation="Horizontal">
  <Calendar DisplayMode="Month" DisplayDate="1/1/2012"
            DisplayDateEnd="12/31/2012">
    <Calendar.BlackoutDates>
      <CalendarDateRange Start="1/1/2012" End="1/10/2012" />
    </Calendar.BlackoutDates>
  </Calendar>

  <DatePicker Margin="20,0" VerticalAlignment="Top"
              SelectedDateFormat="Long" SelectedDate="1/1/2012"
              DisplayDateStart="1/1/12" DisplayDateEnd="12/31/12"
              FirstDayOfWeek="Monday" />
</StackPanel>
```

28. Компоновка в WPF

Контейнер компоновки – это класс, реализующий определённую логику компоновки дочерних элементов. Технология WPF предлагает ряд стандартных контейнеров компоновки:

1. `Canvas`. Позволяет элементам позиционироваться по фиксированным координатам.
2. `StackPanel`. Размещает элементы в горизонтальный или вертикальный стек. Контейнер обычно используется в небольших секциях сложного окна.
3. `WrapPanel`. Размещает элементы в сериях строк с переносом. Например, в горизонтальной ориентации `WrapPanel` располагает элементы в строке слева направо, затем переходит к следующей строке.
4. `DockPanel`. Выравнивает элементы по краю контейнера.
5. `Grid`. Встраивает элементы в строки и колонки невидимой таблицы.

Все контейнеры компоновки являются панелями, которые унаследованы от абстрактного класса `System.Windows.Controls.Panel`. Этот класс содержит несколько полезных свойств:

`Background` – кисть, используемая для рисования фона панели. Кисть нужно задать, если панель должна принимать события мыши (как вариант, это может быть прозрачная кисть).

`Children` – коллекция элементов, находящихся в панели. Это первый уровень вложенности – другими словами, это элементы, которые сами могут содержать дочерние элементы.

`IsItemsHost` – булево значение. Устанавливается в `true`, если панель используется для показа элементов в шаблоне спискового элемента управления.

`ZIndex` – присоединённое свойство класса `Panel` для задания высоты визуального слоя элемента. Элементы с большим значением `ZIndex` выводятся поверх элементов с меньшим значением.

`Canvas` – контейнер компоновки, реализующий позиционирование элементов путём указания фиксированных координат. Для задания позиции элемента следует использовать присоединённые свойства `Canvas.Left`, `Canvas.Right`, `Canvas.Top`, `Canvas.Bottom`. Эти свойства определяют расстояние от соответствующей стороны `Canvas` до ближайшей грани

элемента[9]. Использование контейнера Canvas демонстрируется в следующем примере:

```
    <Canvas>
      <Button Background="Red" Content="Left=0, Top=0" />
      <Button Canvas.Left="20" Canvas.Top="20"
Background="Orange"
              Content="Left=20, Top=20" />
      <Button Canvas.Right="20" Canvas.Bottom="20"
Background="Yellow"
              Content="Right=20, Bottom=20" />
      <Button Canvas.Right="0" Canvas.Bottom="0"
Background="Lime"
              Content="Right=0, Bottom=0" />
      <Button Canvas.Right="0" Canvas.Top="0"
Background="Aqua"
              Content="Right=0, Top=0" />
      <Button Canvas.Left="0" Canvas.Bottom="0"
Background="Magenta"
              Content="Left=0, Bottom=0" />
    </Canvas>
```

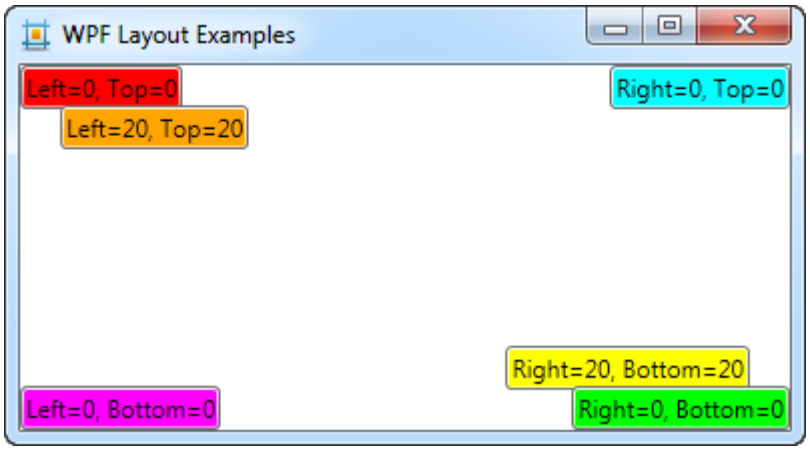


Рис. 6. Кнопки в контейнере Canvas.

StackPanel – популярный контейнер компоновки, который размещает дочерние элементы последовательно, по мере их объявления в контейнере. Свойство `Orientation` управляет направлением размещения дочерних

[9] При одновременной установке Canvas.Left имеет преимущество перед Canvas.Right, а Canvas.Top – перед Canvas.Bottom.

элементов и принимает значения из одноимённого перечисления: `Vertical` (по умолчанию) или `Horizontal`.

```
<StackPanel Orientation="Horizontal">
  <Button Background="Red" Content="One" />
  <Button Background="Orange" Content="Two" />
  <Button Background="Yellow" Content="Three" />
  <Button Background="Lime" Content="Four" />
  <Button Background="Aqua" Content="Five" />
  <Button Background="Magenta" Content="Six" />
</StackPanel>
```

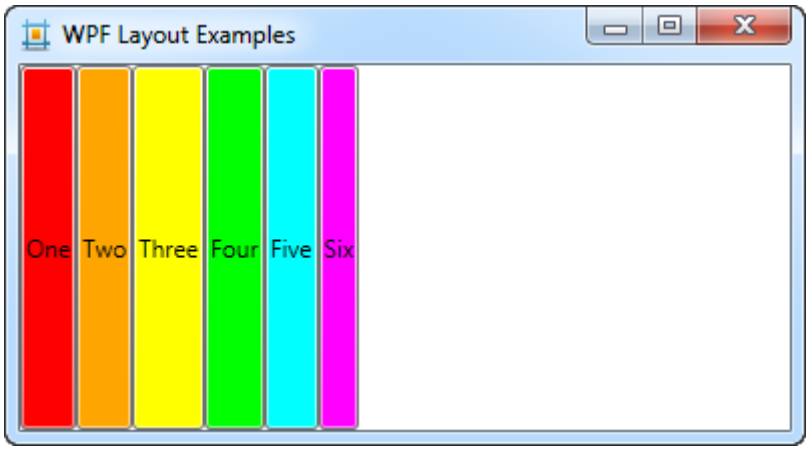


Рис. 7. `StackPanel` с горизонтальной ориентацией.

`WrapPanel` – это контейнер компоновки, который во многом аналогичен `StackPanel`. Однако `WrapPanel` использует автоматический перенос элементов, для которых не хватает вертикального (горизонтального) пространства, в новый столбец (строку). `WrapPanel` поддерживает несколько свойств настройки:

`Orientation` – свойство аналогично одноименному свойству у `StackPanel`, но по умолчанию использует значение `Horizontal`.

`ItemHeight` – единая мера высоты для всех дочерних элементов. В рамках заданной единой высоты каждый дочерний элемент располагается в соответствие со своим свойством `VerticalAlignment` или усекается.

`ItemWidth` – единая мера ширины для всех дочерних элементов. В рамках заданной единой ширины каждый дочерний элемент располагается в соответствие со своим свойством `HorizontalAlignment` или усекается.

По умолчанию, свойства `ItemHeight` и `ItemWidth` не заданы (имеют значение `double.NaN`). В этой ситуации ширина столбца (высота строки)

определяется по самому широкому (самому высокому) дочернему элементу.

```
    <WrapPanel ItemHeight="80">
      <Button Width="60" Height="40" Background="Red"
Content="One" />
      <Button Width="60" Height="20" Background="Orange"
              Content="Two" />
      <Button Width="60" Height="40" Background="Yellow"
              Content="Three" />
      <Button Width="60" Height="20" VerticalAlignment="Top"
              Background="Lime" Content="Four" />
      <Button Width="60" Height="40" Background="Aqua"
Content="Five" />
      <Button Width="60" Height="20" Background="Magenta"
              Content="Six" />
    </WrapPanel>
```

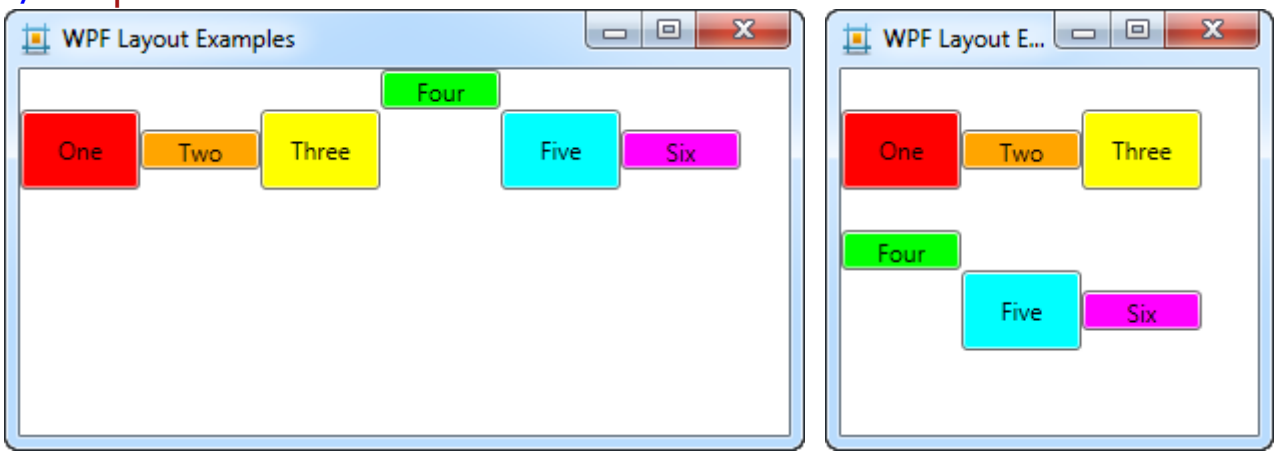


Рис. 8. Элементы на `WrapPanel` для разной ширины окна.

`DockPanel` (*док*) реализует *примыкание* (docking) дочерних элементов к одной из своих сторон. Примыкание настраивается при помощи присоединённого свойства `DockPanel.Dock`, принимающего значения `Left`, `Top`, `Right` и `Bottom` (перечисление `System.Windows.Controls.Dock`). Примыкающий элемент растягивается на всё свободное пространство дока по вертикали или горизонтали (в зависимости от стороны примыкания), если у элемента явно не задан размер. Порядок дочерних элементов в доке имеет значение. Если в `DockPanel` свойство `LastChildFill` установлено в `true` (это значение по умолчанию), последний дочерний элемент занимает всё свободное пространство дока.

```
    <DockPanel>
      <Button DockPanel.Dock="Top" Background="Red" Content="1
(Top)" />
      <Button DockPanel.Dock="Left" Background="Orange"
              Content="2 (Left)" />
      <Button DockPanel.Dock="Right" Background="Yellow"
              Content="3 (Right)" />
      <Button DockPanel.Dock="Bottom" Background="Lime"
              Content="4 (Bottom)" />
      <Button Background="Aqua" Content="5 (Fill)" />
    </DockPanel>
```

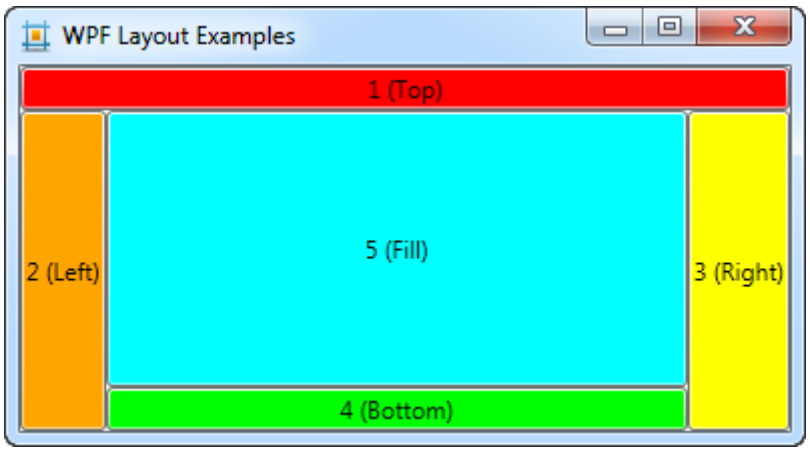


Рис. 9. Док с набором кнопок.

`Grid` – один из наиболее гибких и широко используемых контейнеров компоновки. Он организует пространство как таблицу с настраиваемыми столбцами и строками. Дочерние элементы размещаются в указанных ячейках таблицы.

```
    <Grid>
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="100" />
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="*" />
      </Grid.ColumnDefinitions>
      <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="2*" />
      </Grid.RowDefinitions>
```

```
        <Button Grid.Column="0" Grid.Row="0"
                Background="Red" Content="One" />
        <Button Grid.Column="1" Grid.Row="0" Width="60"
                Background="Orange" Content="Two" />
        <Button Grid.Column="2" Grid.Row="0"
                Background="Yellow" Content="Three" />
        <Button Grid.Column="0" Grid.Row="1"
                Background="Lime" Content="Four" />
        <Button Grid.Column="1" Grid.Row="1"
Grid.ColumnSpan="2"
                Background="Aqua" Content="Five" />
    </Grid>
```

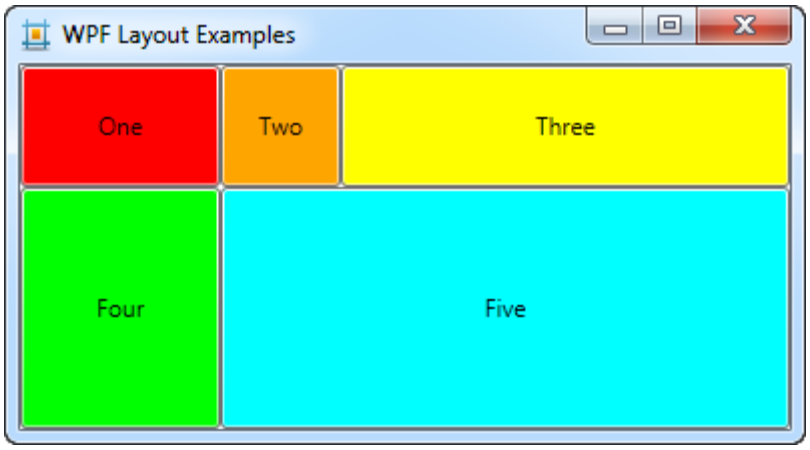


Рис. 10. Демонстрация контейнера `Grid`.

При настройке `Grid` необходимо задать набор столбцов и строк с помощью коллекций `ColumnDefinitions` и `RowDefinitions`. Для столбцов может быть указана ширина, для строк – высота. При этом допустимо использовать абсолютное значение, подбор по содержимому или *пропорциональный размер*. В последнем случае применяется символ * и необязательный коэффициент пропорциональности. В примере, приведённом выше, высота второй строки равна удвоенной высоте первой строки, независимо от высоты окна.

Дочерние элементы связываются с ячейками `Grid` при помощи присоединённых свойств `Grid.Column` и `Grid.Row`. Если несколько элементов расположены в одной ячейке, они наслаиваются друг на друга. Один элемент может занять несколько ячеек, определяя значения для присоединённых свойств `Grid.ColumnSpan` и `Grid.RowSpan`.

Контейнер `Grid` позволяет изменять размеры столбцов и строк при помощи

перетаскивания, если используется элемент управления GridSplitter. Вот несколько правил работы с этим элементом:
1. GridSplitter должен быть помещён в ячейку Grid. Лучший подход заключается в резервировании специального столбца или строки для GridSplitter со значениями Height или Width, равными Auto.
2. GridSplitter всегда изменяет размер всей строки или столбца, а не отдельной ячейки. Чтобы сделать внешний вид GridSplitter соответствующим такому поведению, растяните его по всей строке или столбцу, используя присоединённые свойства Grid.ColumnSpan или Grid.RowSpan.
3. Изначально GridSplitter настолько мал, что его не видно. Дайте ему минимальный размер. Например, в случае вертикальной разделяющей полосы установите VerticalAlignment в Stretch, а Width – в фиксированный размер.
4. Выравнивание GridSplitter определяет, будет ли разделительная полоса горизонтальной (используемой для изменения размеров строк) или вертикальной (для изменения размеров столбцов). В случае горизонтальной разделительной полосы установите VerticalAlignment в Center (что принято по умолчанию). В случае вертикальной разделительной полосы установите HorizontalAlignment в Center.
Ниже приведён фрагмент разметки, использующей горизонтальный GridSplitter:

```
    <Grid>
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="100" />
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="*" />
      </Grid.ColumnDefinitions>
      <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="2*" />
      </Grid.RowDefinitions>
      <GridSplitter Grid.Column="0" Grid.Row="1"
Grid.ColumnSpan="3"
```

```
                    HorizontalAlignment="Stretch" Height="3"
                    VerticalAlignment="Center" />
    </Grid>
```

Классы `RowDefinition` и `ColumnDefinition` обладают строковым свойством `SharedSizeGroup`, при помощи которого строки или столбцы объединяются в *группу, разделяющую размер*. Это означает, что при изменении размера одного элемента группы (строки или столбца), другие элементы автоматически получают такой же размер. Разделение размеров может быть выполнено как в рамках одного контейнера `Grid`, так и между несколькими `Grid`. В последнем случае необходимо установить свойство `Grid.IsSharedSizeScope` в значение `true` для внешнего контейнера компоновки:

```
    <Grid IsSharedSizeScope="True">
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"
SharedSizeGroup="myGroup" />
        <ColumnDefinition />
        <ColumnDefinition SharedSizeGroup="myGroup" />
      </Grid.ColumnDefinitions>

      <!-- определение элементов Grid -->
    </Grid>
```

29. Использование стилей и шаблонов в WPF

Стиль – это коллекция значений свойств, которые могут быть применены к элементу. В WPF стили играют ту же роль, которую CSS играет в HTML-разметке. Подобно CSS, стили WPF позволяют определять общий набор характеристик форматирования и применять их по всему приложению для обеспечения согласованности. Стили могут работать автоматически, предназначаться для элементов конкретного типа и каскадироваться через дерево элементов.

```
    <StackPanel Orientation="Horizontal" Margin="30">
      <StackPanel.Resources>
        <Style x:Key="buttonStyle">
```

```
        <Setter Property="Button.FontSize" Value="22" />
        <Setter Property="Button.Background" Value="Purple"
/>
        <Setter Property="Button.Foreground" Value="White"
/>
        <Setter Property="Button.Height" Value="80" />
        <Setter Property="Button.Width" Value="80" />
        <Setter Property="Button.RenderTransformOrigin"
                Value=".5,.5" />
        <Setter Property="Button.RenderTransform">
          <Setter.Value>
            <RotateTransform Angle="10" />
          </Setter.Value>
        </Setter>
      </Style>
    </StackPanel.Resources>

    <Button Style="{StaticResource buttonStyle}">1</Button>
    <Button Style="{StaticResource buttonStyle}">2</Button>
    <Button Style="{StaticResource buttonStyle}">3</Button>
  </StackPanel>
```

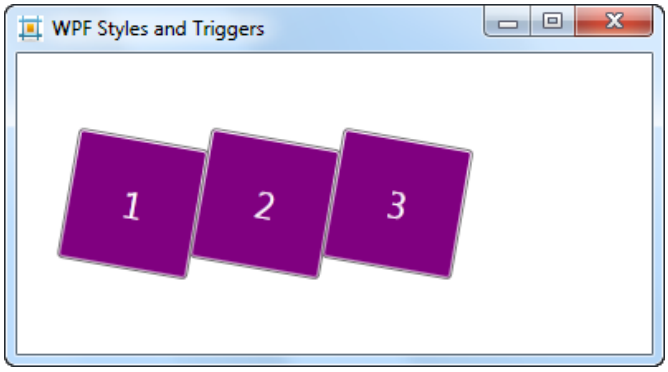


Рис. 39. Кнопки, к которым применён стиль.

Любой стиль в WPF – это объект класса `System.Windows.Style`. Основным свойством стиля является коллекция `Setters` (свойство содержимого), в которой каждый элемент задаёт значение для некоторого свойства зависимостей (стили работают только со свойствами зависимостей). Чтобы задать значение для свойства зависимостей, нужно указать имя этого свойства. В нашем примере имя включает префикс – класс элемента. Если требуется

применить один стиль к визуальным элементам разных типов, для префикса используется имя общего типа-предка:

```
    <StackPanel Orientation="Horizontal" Margin="30">
      <StackPanel.Resources>
        <Style x:Key="controlStyle">
          <Setter Property="Control.FontSize" Value="22" />
          <Setter Property="Control.Background" Value="Purple"
/>
          <Setter Property="Control.Foreground" Value="White"
/>
          <Setter Property="Control.Height" Value="80" />
          <Setter Property="Control.Width" Value="80" />
        </Style>
      </StackPanel.Resources>
      <Button Style="{StaticResource controlStyle}"
Content="1" />
      <TextBox Style="{StaticResource controlStyle}"
Text="Hello" />
      <Expander Style="{StaticResource controlStyle}"
Content="3" />
    </StackPanel>
```

Свойство стиля `TargetType` позволяет указать конкретный тип, к которому применяется стиль. В этом случае префикс в установщиках свойств использовать необязательно. Определяя стиль, можно построить его на основе стиля-предка (свойство стиля `BasedOn`). Стили также могут содержать локальные логические ресурсы (коллекция `Resources`).Стиль позволяет задать обработчики событий. Для этого применяется объект `EventSetter`:

```
<!-- можно записать TargetType="{x:Type Button}" -->
    <Style x:Key="baseStyleWH" TargetType="Button"
           BasedOn="{StaticResource baseStyle}">
      <Style.Resources>
        <sys:Double x:Key="size">80</sys:Double>
      </Style.Resources>
      <Setter Property="Height" Value="{StaticResource size}"
/>
```

```
      <Setter Property="Width" Value="{StaticResource size}"
/>
      <EventSetter Event="MouseEnter"
     Handler="button_MouseEnter" />
    </Style>
```

12. Шаблоны

Шаблоны – фундаментальная концепция технологии WPF. Шаблоны обеспечивают настраиваемое представление (внешний вид) для элементов управления и произвольных объектов, отображаемых как содержимое элементов.

12.1. Шаблоны элементов управления

Большинство элементов управления имеют *внешний вид* и *поведение*. Рассмотрим кнопку: её внешним видом является область для нажатия, а поведением – событие `Click`, которое вызывается в ответ на нажатие кнопки. WPF эффективно разделяет внешний вид и поведение, благодаря концепции *шаблона элемента управления*. Шаблон элемента управления полностью определяет визуальную структуру элемента. Шаблон переопределяем – в большинстве случаев это обеспечивает достаточную гибкость и освобождает от необходимости написания пользовательских элементов управления.
Шаблон элемента управления – это экземпляр класса `System.Windows.ControlTemplate`. Основным свойством шаблона является свойство содержимого `VisualTree`, которое содержит визуальный элемент, определяющий внешний вид шаблона. В элементах управления ссылка на шаблон устанавливается через свойство `Template`.
Самый большой недостаток шаблона заключается в том, что он не отображает содержимое кнопки (свойство `Content`). Исправим это. У шаблона может быть установлено свойство `TargetType`. Оно содержит тип элемента управления, являющегося целью шаблона. Если это свойство установлено, при описании `VisualTree` для ссылки на содержимое элемента управления можно использовать объект `ContentPresenter` (для элементов управления содержимым) или объект `ItemsPresenter` (для списков). (см.)

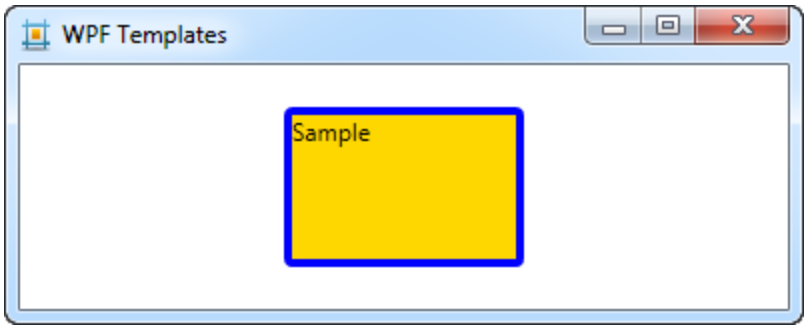


Рис. 44. Шаблон, отображающий контент.

Вторая версия шаблона не учитывает отступ, заданный на кнопке при помощи свойства `Padding`. Чтобы исправить это, используем привязку данных. В шаблонах допустим особый вид привязки – `TemplateBinding`. Эта привязка извлекает информацию из свойства элемента управления, являющегося целью шаблона. (см.)

```
    <Button Content="Sample" Width="120" Height="80"
Padding="15">
      <Button.Template>
        <ControlTemplate TargetType="Button">
          <Border BorderBrush="Blue" BorderThickness="4"
                  CornerRadius="2" Background="Gold">
            <ContentPresenter Margin="{TemplateBinding
Padding}" />
          </Border>
          <ControlTemplate.Triggers>
            <Trigger Property="IsMouseOver" Value="True">
              <Setter TargetName="brd" Property="Background"
                                       Value="Yellow" />
            </Trigger>
          </ControlTemplate.Triggers>
        </ControlTemplate>
      </Button.Template>
    </Button>
```

В шаблонах элементов управления часто используются триггеры. Например, для кнопки при помощи триггеров можно реализовать изменение внешнего вида при нажатии или при перемещении указателя мыши. (см.)

Обратите внимание – элемент триггера `Setter` содержит установку свойства `TargetName`. Как ясно из контекста, `TargetName` используется, чтобы

обратиться к именованному дочернему элементу визуального представления. Этот дочерний элемент должен быть описан до триггера.
Однако во многих случаях шаблоны выглядят сложнее. Более сложные шаблоны могут включать элементы управления, которые инициируют встроенные команды или обработчики событий, вложенные элементы управления, которые могут иметь свои собственные шаблоны, или использовать именованные элементы (имена у которых обычно начинаются с префикса *PART_)*.

12.2. Шаблоны данных

Шаблон данных – механизм для настройки отображения объектов заданного типа. Любой шаблон данных – это объект `System.Windows.DataTemplate`. Основное свойство шаблона данных – `VisualTree`. Оно содержит визуальный элемент, определяющий внешний вид шаблона. При формировании `VisualTree` обычно используется привязка данных для извлечения информации из объекта, для которого применяется шаблон. Сам шаблон данных, как правило, размещают в ресурсах окна или приложения.
Рассмотрим пример использования шаблонов данных. Пусть имеется объект класса `Person`, описанный в ресурсах окна и являющийся содержимым окна:

```
    // класс Person объявлен в пространстве имён WpfTemplates
    public class Person
    {
        public string Name { get; set; }
        public double Age { get; set; }
    }

    <Window x:Class="WpfTemplates.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentat
ion"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      xmlns:local="clr-namespace:WpfTemplates"
      Width="400" Height="160" Title="WPF Templates">
      <Window.Resources>
        <local:Person x:Key="smith" Name="Mr. Smith"
Age="27.3" />
```

```
    </Window.Resources>
    <Window.Content>
      <StaticResourceExtension ResourceKey="smith" />
    </Window.Content>
  </Window>
```

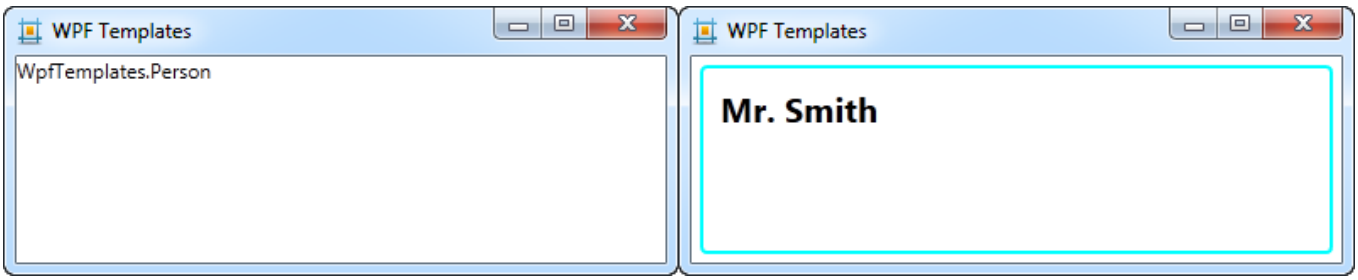


Рис. 45. Показ объекта `Person` без шаблона данных и при помощи шаблона.

Определим для `Person` шаблон данных. У класса `DataTemplate` имеется свойство `DataType`, устанавливающее тип, к которому будет применяться шаблон. Если задано это свойство, шаблон будет использоваться в любой ситуации, где нужно отобразить объект.

```
  <!-- остальное описание окна не изменилось -->
  <Window.Resources>
    <local:Person x:Key="smith" Name="Mr. Smith" Age="27.3"
/>
    <DataTemplate DataType="{x:Type local:Person}">
      <Border Name="bord" BorderBrush="Aqua"
BorderThickness="2"
              CornerRadius="3" Padding="10" Margin="5">
        <TextBlock FontSize="20" FontWeight="Bold"
                   Text="{Binding Name}" />
      </Border>
    </DataTemplate>
  </Window.Resources>
```

30. Основные концепции реляционных баз данных и языка SQL.

Реляционная СУБД представляет собой совокупность именованных двумерных таблиц данных, логически связанных (находящихся в отношении) между собой. Таблицы состоят из строк и именованных столбцов, строки представляют собой экземпляры информационного объекта, столбцы –

атрибуты объекта.

№	Наименование	Ед. изм.	Цена	Кол-во
1	Кирпич	штука	255	10000
2	Краска	литр	580	670
3	Шифер	лист	130	500
…	…	…	…	…
10001	Гвоздь	штука	20	8000
10002	Кабель	метр	100	200

В рассмотренном ранее примере таблица (назовем ее “Склад”) состоит из информационных объектов-строк, отдельная строка содержит сведения об отдельном товаре. Каждый товар характеризуется некими параметрами-атрибутами (“Наименование”, “Цена” и т.д.). Строки иногда называют записями, а столбцы – полями записи. Таким образом, в реляционной модели все данные представлены для пользователя в виде таблиц значений данных, и все операции над базой сводятся к манипулированию таблицами.

Связи между отдельными таблицами в реляционной модели в явном виде могут не описываться. Они устанавливаются пользователем при написании запроса на выборку данных и представляют собой условия равенства значений соответствующих полей.

Реляционная база данных – это совокупность отношений, содержащих всю информацию, которая должна храниться в БД. Однако пользователи могут воспринимать такую базу данных как совокупность таблиц.

1. Каждая таблица состоит из однотипных строк и имеет уникальное имя.

2. Строки имеют фиксированное число полей (столбцов) и значений (множественные поля и повторяющиеся группы недопустимы). Иначе говоря, в каждой позиции таблицы на пересечении строки и столбца всегда имеется в точности одно значение или ничего.

3. Строки таблицы обязательно отличаются друг от друга хотя бы единственным значением, что позволяет однозначно идентифицировать любую строку такой таблицы.

4. Столбцам таблицы однозначно присваиваются имена, и в каждом из них размещаются однородные значения данных (даты, фамилии, целые числа или денежные суммы).

В реляционной модели при логическом связывании таблиц применяется следующая терминология:

Первичный ключ (или главный ключ, primary key, PK). Представляет

собой столбец или совокупность столбцов, значения которых однозначно идентифицируют строки.

Вторичный (или внешний ключ, foreign key, FK). Столбец или совокупность столбцов, которые в данной таблице не являются первичными ключами, но являются первичными ключами в другой таблице.

Основным критерием качества разработанной модели данных является ее соответствие так называемым нормальным формам (НФ). Основная цель нормализации – устранение избыточности данных. Коддом были определены три нормальные формы, которые включают одна другую. Другими словами, если модель данных соответствует 2НФ, то она одновременно соответствует и 1НФ. Соответствие 3НФ подразумевает соответствие 1НФ и 2НФ.

Первая нормальная форма гласит: информация в каждом поле таблицы является неделимой и не может быть разбита на подгруппы. Неправильно:

…	Иванов, 15 отдел, начальник	…

Правильно:

Фамилия	Должность	№ отдела
Иванов	Начальник	15

Вторая нормальная форма гласит: таблица соответствует 1НФ и в таблице нет неключевых атрибутов, зависящих от части сложного (состоящего из нескольких столбцов) первичного ключа.

Отделы

№ отдела	Наименование
15	Производственный отдел
10	Отдел продаж

Структура

№ отдела	Должность	Количество сотрудников
15	Начальник	1
15	Инженер	5
10	Начальник	1
10	Менеджер	10

Третья нормальная форма гласит: таблица соответствует первым двум НФ и все неключевые атрибуты зависят только от первичного ключа и не зависят друг от друга.

Сотрудники

Табельный №	Фамилия	Оклад	№ отдела
1	Иванов	500	15
2	Петров	400	15
3	Иванов	600	10

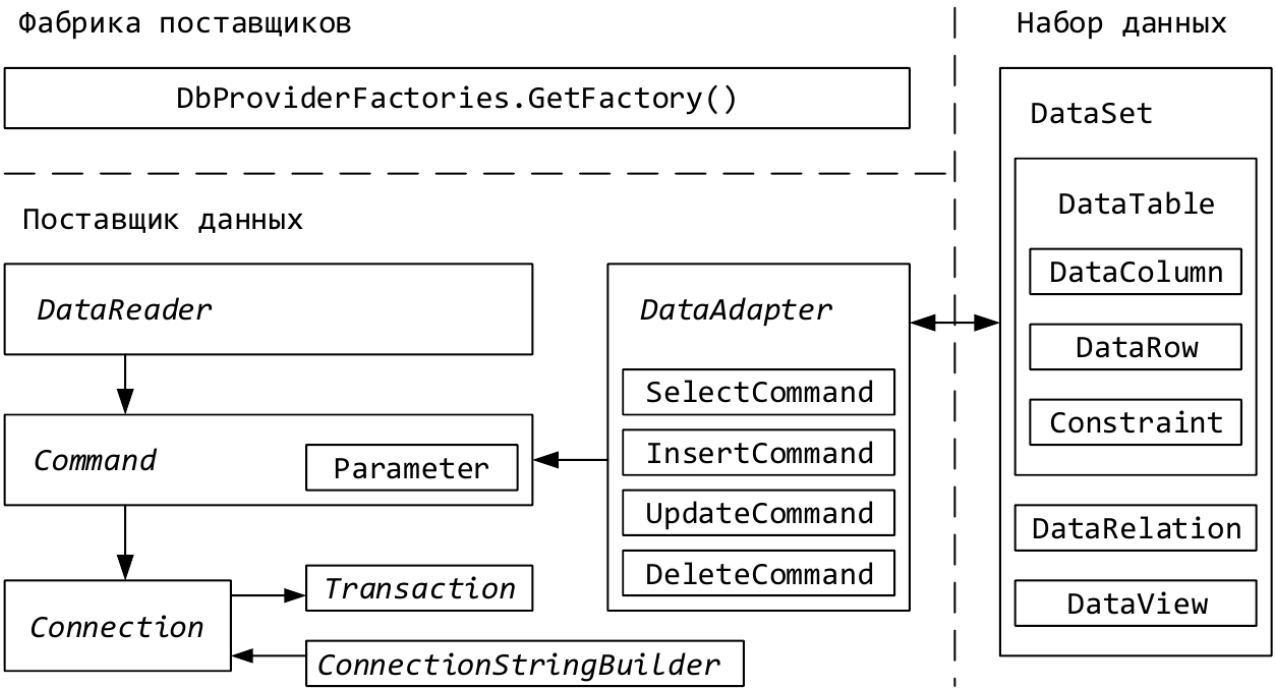
Отделы

№ отдела	Наименование
15	Производственный отдел
10	Отдел продаж

Взаимодействие приложений и пользователей с реляционными СУБД осуществляется посредством специального языка структурированных запросов (Structured Query Language, сокращенно – SQL). SQL был разработан еще в начале 70-х годов XX века и представляет собой непроцедурный язык, состоящий из набора стандартных команд на английском языке. Термин “непроцедурный” означает, что изначально в языке отсутствуют алгоритмические конструкции (переменные, переходы по условию, циклы и т.д.) и возможность компоновать логически связанные команды в единые программные блоки (процедуры и функции).

31. Технология ADO.NET – общая архитектура.

Технология ADO.NET – это часть Microsoft .NET Framework, обеспечивающая основные возможности по работе с реляционными базами данных и источниками данных. Рассмотрим общую архитектуру ADO.NET. Главными элементами ADO.NET являются: поставщик данных, абстрактная фабрика поставщиков и рассоединённый набор данных.



Поставщик данных (data provider) – это совокупность типов для непосредственного взаимодействия с конкретным источником данных. Любой поставщик содержит четыре основных класса: `Connection`, `Command`, `DataReader` и `DataAdapter`. Назначение класса `Connection` – установка и поддержка соединения с источником данных. Класс `Command` служит для выполнения команд и запросов. Можно выполнить команды, не возвращающие данные (например, создать в базе таблицу), и запросы для получения скалярного значения или набора данных. В последнем случае для чтения данных используется объект класса `DataReader` – однонаправленный курсор в режиме «только для чтения». Класс `DataAdapter` служит своеобразным «мостом» между поставщиком данных и рассоединённым набором данных. Этот класс содержит четыре команды для выборки, обновления, вставки и удаления данных.

В состав поставщика данных входят некоторые вспомогательные элементы. Класс `ConnectionStringBuilder` используется для построения строк соединения. Класс `Transaction` служит для описания транзакции данных. Класс `Parameter` описывает параметры отдельной команды, а класс `CommandBuilder` облегчает построение самих команд. Также поставщик имеет классы `Exception` и `Error` для описания исключительных ситуаций и ошибок, возникающих при взаимодействии с источником данных.

Последним элементом архитектуры ADO.NET является рассоединённый

набор данных (data set). Набор данных содержит информационный фрагмент источника данных. Для описания набора данных используются классы из пространства имён `System.Data`. Эти классы универсальны и не зависят от поставщика данных. Главным компонентом набора является класс `DataSet`, агрегирующий объекты остальных классов. Класс `DataTable` служит для описания таблиц. Элементами класса `DataTable` являются коллекции объектов `DataColumn` (колонки таблицы), `DataRow` (строки таблицы) и `Constraint` (ограничения на значения элементов таблицы). Класс `DataRelation` описывает связи между таблицами.

32. Технология ADO.NET – соединение с базой и выполнение команд.

В ADO.NET любое действие с источником данных можно выполнить при наличии соединения с этим источником. Каждый поставщик данных содержит класс, ответственный за описание соединений. Например, поставщик данных для SQL Server использует класс `SqlConnection`.

Класс соединения реализует интерфейс `System.Data.IDbConnection`. и наследуется от класса `System.Data.Common.DbConnection`. Элементы интерфейса `IDbConnection` перечислены в таблице:

Имя элемента	Описание
`ConnectionString`	Строка, описывающая параметры подключения к базе данных
`ConnectionTimeout`	Время ожидания открытия подключения (в секундах) перед тем, как возникнет исключение «Невозможно подключиться к базе». По умолчанию – 15 секунд, ноль соответствует бесконечному ожиданию. Значение устанавливается строкой подключения
`Database`	Имя базы данных, к которой подключаемся. Значение устанавливается строкой подключения, но может быть изменено вызовом метода `ChangeDatabase()`. Не все поставщики поддерживают это свойство
`State`	Состояние соединения. Принимает значение из перечисления `ConnectionState`. В настоящий момент поддерживаются `ConnectionState.Open` и `ConnectionState.Closed`

`BeginTransaction()`	Метод начинает транзакцию в базе данных
`ChangeDatabase()`	Устанавливает новую базу данных для использования. Является аналогом команды `USE` в SQL Server. Поставщики для СУБД Oracle не поддерживают этот метод
`CreateCommand()`	Возвращает объект, реализующий интерфейс `IDbCommand` (команду), но специфичный для конкретного поставщика данных
`Open()` и `Close()`	Попытка соединиться и разъединиться с источником данных

Строка подключения служит для указания параметров подключения к базе данных. В строке подключения через точку с запятой перечислены пары вида «имя параметра =значение». В таблице перечислены некоторые возможные параметры подключения. Не все из них являются обязательными для указания, некоторые специфичны для определенных поставщиков данных. Через наклонную черту указаны возможные альтернативные названия параметров.

Имя параметра	Описание параметра
`Data Source / Server/ Address / Addr / Network Address`	Имя или сетевой адрес сервера. Для локальных серверов используется значение `localhost`
`Initial Catalog / Database`	Имя базы данных
`Integrated Security / Trusted_Connection`	Если установлено в `true` или `SSPI`, поставщик данных пытается подключиться к серверу, используя имя и пароль пользователя в системе Windows
`User ID`	Идентификатор пользователя базы данных
`Password / Pwd`	Пароль пользователя базы данных

Задать строку подключения можно, либо указав её как параметр конструктора класса соединения, либо при помощи свойства `ConnectionString`. Естественно, строка подключения задается до вызова у соединения метода `Open()`. Параметры строки необходимо помнить и задавать без ошибок, так как эти ошибки будут обнаружены только в момент

подключения к базе, но не при компиляции. Для устранения этого недостатка в поставщиках данных имеется специальный класс, унаследованный от `DbConnectionStringBuilder`, который предназначен для построения правильных строк подключения. Отдельные параметры строки задаются как типизированные свойства этого класса.

Для увеличения производительности приложений поставщики данных могут поддерживать *пул соединений* (*connection pool*). Сущность пула заключается в следующем. При вызове метода `Close()` соединение с базой не разрывается, а помещается в буфер. Если приложение захочет открыть соединение, аналогичное существующему в буфере, то система возвращает открытое подключение из пула.

Управляемые поставщики обычно содержат специальные классы, описывающие исключения. Для поставщика MS SQL Server это класс `SqlException`. В классе `SqlException` доступно свойство `Errors` - набор объектов типа `SqlError`. В этих объектах содержится дополнительная информация об определенном исключении, полученная от базы данных.

Работа с базой данных основана на выполнении запросов. Для выполнения запросов любых типов поставщик данных предоставляет класс, который реализует интерфейс `IDbCommand` и наследуются от класса `DbCommand`. В данном параграфе рассматривается класс `SqlCommand`.

Для создания команды существуют два основных способа. Первый способ – использование конструктора. Класс `SqlCommand` содержит несколько перегруженных конструкторов: конструктор без параметров; конструктор, которому передается как параметр текст команды; конструктор, принимающий как параметры текст команды и объект-соединение.

Второй способ – это создание команды на основе объекта-соединения. В этом случае команда автоматически связывается с соединением.

Связь команды с соединением должна быть установлена перед выполнением команды. В общем случае для этого используется свойство `Connection`.

Свойство команды `CommandText` содержит текст команды, а свойство `CommandType` определяет, как следует понимать этот текст. Это свойство может принимать следующие значения:

- `CommandType.Text`. Текст команды – это SQL-инструкции. Обычно такие команды передаются в базу без предварительной обработки (за

исключением случаев передачи параметров). Этот тип команды устанавливается по умолчанию.

- `CommandType.StoredProcedure`. Текст команды – это имя хранимой процедуры, которая находится в базе данных.
- `CommandType.TableDirect`. Команда предназначена для извлечения из БД полной таблицы. Имя таблицы указывается в `CommandText`. Данный тип команд поддерживает только поставщик OLE DB.

За подготовкой команды следует её выполнение. Естественно, до выполнения команда должна быть связана с соединением, и это соединение должно быть открыто. В ADO.NET существует несколько способов выполнения команд, которые отличаются лишь информацией, возвращаемой из БД. Ниже перечислены методы выполнения команд, поддерживаемые всеми поставщиками.

- `ExecuteNonQuery()`. Этот метод применяется для запросов, не возвращающих данные. Метод возвращает суммарное число строк, добавленных, измененных или удаленных в результате выполнения команды.
- `ExecuteScalar()`. Метод выполняет команду и возвращает первый столбец первой строки первого результирующего набора данных. Метод может быть полезен при выполнении запросов или хранимых процедур, возвращающих единственных результат.
- `ExecuteReader()`. Этот метод выполняет команду и возвращает объект `DbDataReader`. Тип возвращаемого ридера соответствует поставщику. Метод `ExecuteReader()` используется, когда требуется получить набор данных из базы (например, как результат выполнения команды `SELECT`).

В качестве примера использования метода `ExecuteScalar()` приведём код, при помощи которого подсчитывается число записей в таблице `Songs`:

```
// пример создания и настройки команды
var con_str = new SqlConnectionStringBuilder
              {
                  DataSource = "(local)",
                  InitialCatalog = "MusicCatalog",
                  IntegratedSecurity = true
              };
var con = new SqlConnection(con_str.ConnectionString);
```

```
    var cmd = new SqlCommand
              {
                  Connection = con,
                  CommandText = "SELECT COUNT(*) FROM Songs",
                  CommandTimeout = 3,
                  CommandType = CommandType.Text
              };
    con.Open();
    // приведение типов необходимо, так как метод ExecuteScalar()
    // возвращает значение типа object
    var count = (int)cmd.ExecuteScalar();
    con.Close();
```

Если в предыдущем примере изменить текст команды на `"SELECT * FROM Songs"`, то в результате выполнения вместо ожидаемого набора данных получим значение первого элемента в первой колонке таблицы `Songs`.

Поставщик для MS SQL Server поддерживает метод выполнения команды, возвращающий объект класса `XmlReader`. Это метод `ExecuteXmlReader()`. Метод поддерживается для SQL Server 2000 и более поздних версий и требует, чтобы возвращаемые данные были в формате XML.

В платформе .NET версии 2.0 у класса `SqlCommand` был реализован ряд новых методов, позволяющих выполнять асинхронные операции с базой. Для методов `ExecuteNonQuery()`, `ExecuteReader()` и `ExecuteXmlReader()` появились парные асинхронные методы (например, `BeginExecuteNonQuery()` и `EndExecuteNonQuery()`). Использование асинхронных команд может быть полезно в тех случаях, когда необходимо выполнение нескольких длительных операций или хранимых процедур. Для применения асинхронных команд необходимо указать в строке соединения параметр «`async=true`».

33. Технология ADO.NET – рассоединённый набор данных.

ADO.NET предоставляет возможность работы с рассоединенным набором данных. Такой набор реализуется объектом класса `DataSet`. Это реляционная структура, которая хранится в памяти. `DataSet` содержит набор таблиц (объектов класса `DataTable`) и связей между таблицами (объекты класса

DataRelation). В свою очередь, отдельная таблица содержит набор столбцов (объекты класса DataColumn), строк (объекты класса DataRow) и ограничений (объекты классов, унаследованных от Constraint). Столбцы и ограничения описывают структуру отдельной таблицы, а строки хранят данные таблицы.

Технически, отдельные компоненты DataSet хранятся в специализированных коллекциях. Например, DataSet содержит коллекции Tables и Relations. Таблица имеет коллекции Columns (для колонок), Rows (для строк), Constraints (для ограничений), ParentRelations и ChildRelations (для связей таблицы). *Любая* подобная коллекция обладает сходных набором свойств и методов. Коллекции имеют перегруженные индексаторы для обращения к элементу по номеру и по имени, методы добавления, поиска и удаления элементов. Методы добавления перегружены и обеспечивают как добавление существующего объекта, так и автоматическое создание соответствующего объекта перед помещением в коллекцию.

Для набора данных DataSet введём понятие схемы данных. Под *схемой* будем понимать совокупность следующих элементов:

- Имена таблиц;
- Тип и имя отдельных столбцов таблицы;
- Ограничения на столбцы таблицы: уникальность, отсутствие пустых значений, первичные и внешние ключи;
- Связи между таблицами;
- События набора данных и таблицы, которые происходят при работе со строками (аналоги *триггеров* в базах данных).

Схема данных может быть определена различными способами:

- Вручную, путем создания и настройки свойств столбцов, таблиц, связей;
- Автоматически, при загрузке данных в набор из базы;
- Загрузкой схемы, которая была создана и сохранена ранее в XSD-файле.

Правильно созданная схема обеспечивает контроль целостности данных в приложении перед их загрузкой в базу. К сожалению, при загрузке данных из базы в пустой набор генерируется только часть схемы данных (например, в схеме будут отсутствовать связи между таблицами). Рекомендуется подход, при котором в пустом наборе программно создается полная схема, и только затем в этот набор производится считывание данных.

Каждый поставщик данных содержит класс, описывающий *адаптер*

данных. В частности, поставщик для MS SQL Server имеет класс `SqlDataAdapter`. Адаптер данных является своеобразным мостом между базой данных и `DataSet`. Он позволяет записывать данные из базы в набор и производит обратную операцию. В принципе, подобные действия вполне осуществимы при помощи команд и ридеров. Использование адаптера данных – более унифицированный подход.

Основными свойствами адаптера являются `SelectCommand`, `InsertCommand`, `DeleteCommand` и `UpdateCommand`. Это объекты класса `Command` для выборки данных и обновления базы. При помощи метода адаптера `Fill()` происходит запись данных из базы в `DataSet` или таблицу, метод `Update()` выполняет перенос данных в базу.

В начале работы с адаптером его нужно создать и инициализировать свойства-команды. Адаптер содержит несколько перегруженных конструкторов. Пример кода:

```
// 1. Обычный конструктор без параметров.
// Необходимо заполнить команды вручную
var da_1 = new SqlDataAdapter();
// 2. В качестве параметра конструктора – объект-команда
var cmd = new SqlCommand("SELECT * FROM Songs");
var da_2 = new SqlDataAdapter(cmd);
// 3. Параметры: текст запроса для выборки и объект-
соединение
var con = new SqlConnection("Server=(local);. . .");
var da_3 = new SqlDataAdapter("SELECT * FROM Songs", con);
// 4. Параметры – строка запроса и строка соединения
var s = "SELECT * FROM Songs";
var c = "Server=(local);. . .";
var da_4 = new SqlDataAdapter(s, c);
```

Любой адаптер должен иметь ссылку на соединение с базой данных. Адаптер использует то соединение, которое задано в его объектах-командах.

Итак, адаптер создан. Теперь можно использовать его метод `Fill()` для заполнения некоторого набора данных:

```
var da = new SqlDataAdapter();
DataSet ds = new DataSet();
// Строго говоря, метод Fill() - функция, возвращающая
```

```
// число строк (записей), добавленных в DataSet
da.Fill(ds);
```

Заметим, что вызов метода `Fill()` не нарушает состояние соединения с БД. Если соединение было открыто до вызова `Fill()`, то оно останется открытым и после вызова. Если соединение было не установлено, метод `Fill()` откроет соединение, произведет выборку данных и закроет соединение. Так же ведут себя и все остальные методы адаптера, работающие с базой.

Поведение адаптера при заполнении `DataSet` зависит от настроек адаптера и от наличия схемы в объекте `DataSet`. Пусть при помощи адаптера заполняется пустой `DataSet`. В этом случае адаптер создаст в `DataSet` *минимальную схему*, используя имена и тип столбцов из базы и стандартные имена для таблиц. Команда выборки данных может быть настроена на получение нескольких таблиц.

Адаптер имеет свойство-коллекцию `TableMappings`, которое позволяет сопоставить имена таблиц базы и таблиц `DataSet`. Любой элемент коллекции `TableMappings` содержит свойство `ColumnMappings`, которое осуществляет отображение имен столбцов.

Предположим, что заполняемый набор данных уже обладает некой схемой. Адаптер содержит свойство `MissingSchemaAction`, значениями которого являются элементы одноименного перечисления. По умолчанию значение свойства – `Add`. Это означает добавление в схему новых столбцов, если они в ней не описаны. Возможными значениями являются также `Ignore` (игнорирование столбцов, не известных схеме) и `Error` (если столбцы не описаны в схеме, генерируется исключение).

Адаптер данных имеет три события:

- `FillError` – событие наступает, если при заполнении `DataSet` адаптер столкнулся с какой-либо ошибкой;
- `RowUpdating` – событие наступает перед передачей измененной строки в базу данных;

`RowUpdated` – событие наступает после передачи измененной записи в базу данных.