Содержание

5

Обзор .NET

Почти все возможности исполняющей среды .NET 5 доступны через обширное множество управляемых типов. Типы организованы в иерархические пространства имен и упакованы в набор сборок.

Некоторые типы .NET используются напрямую CLR и являются критически важными для среды управляемого размещения. Такие типы находятся в сборке по имени `System.Private.CoreLib.dll` и включают встроенные типы C#, а также базовые классы коллекций, типы для обработки потоков данных, сериализации, рефлексии, многопоточности и собственной возможности взаимодействия.

Сборка `System.Private.CoreLib.dll` заменяет сборку `mscorlib.dll` из .NET Framework. Во многих местах официальной документации все еще можно встречать ссылки на `mscorlib`.

Уровнем выше находятся дополнительные типы, которые расширяют функциональность уровня CLR, предоставляя такие средства, как XML, JSON, взаимодействие с сетью и LINQ. Они образуют библиотеку базовых классов (Base Class Library — BCL). Выше находятся *прикладные слои*, которые предоставляют API-интерфейсы для разработки определенных видов приложений наподобие веб-приложения или обогащенного клиентского приложения.

Вот что предлагается в настоящей главе:

- краткий обзор библиотеки BCL (которая будет более подробно рассматриваться в оставшихся главах книги);
- высокоуровневый обзор прикладных слоев.

Нововведения версии .NET 5

По сравнению с .NET Core 3 версия .NET 5 предлагает многочисленные улучшения производительности, которые касаются сборщика мусора, компилятора JIT, процессора регулярных выражений, процессора HTTP, LINQ, коллекций и т.д.

.NET 5 также включает поддержку времени выполнения для новых возможностей языка C# 9.

Кроме того, несколько важных усовершенствований было внесено в сериализатор JSON, который раскрывается в дополнительных материалах, доступных для загрузки на веб-сайте издательства. В частности, теперь при сериализации и десериализации объектов он обеспечивает сохранность объектных ссылок.

В составе .NET 5 есть новый 16-битный тип с плавающей точкой по имени `Half` для взаимодействия с процессорами графических плат и новый интерфейс `IReadOnlySet<T>`, который позволяет унифицировать выполнение в отношении коллекций операций над множествами, основанных на чтении.

Взаимодействие с COM теперь поддерживает динамическое связывание (как было в .NET Framework).

API-интерфейс Windows Forms теперь содержит класс `TaskDialog`, предоставляющий доступ к версии метода `MessageBox.Show` с более широкими возможностями, которая реализована операционной системой.

Библиотека BCL в .NET 5 теперь на 80% аннотирована для типов, допускающих `null`.

Что касается развертывания, то .NET 5 включает вариант "подрезки приложений": при создании автономных приложений теперь вы можете выбрать удаление неиспользуемых типов и членов из всех поставляемых сборок (в том числе из исполняющей среды .NET).

Прямая поддержка WinRT из .NET 5 была удалена. Для взаимодействия с API-интерфейсами WinRT вы должны теперь либо применять инструмент *CsWinRT*, который генерирует оболочку C# вокруг библиотеки WinRT, либо ссылаться на пакет NuGet, делающий то же самое. Логическое обоснование удаления поддержки WinRT заключается в том, что главный сценарий использования WinRT — платформа UWP — не запускается под управлением .NET 5, а преемник UWP — WinUI 3 — не будет нуждаться в прямой поддержке WinRT. Причина в том, что WinUI 3 будет поставляться в виде развертываемой исполняющей среды, а не быть частью операционной системы, и потому WinUI 3 откроет доступ к полностью управляемому API-интерфейсу.

.NET Standard

Изобилие публичных библиотек, которые доступны через NuGet, не было бы настолько ценным, если бы они поддерживали только .NET 5. При написании библиотеки вы часто хотите поддерживать различные платформы и версии исполняющей среды. Чтобы достичь такой цели, не создавая отдельные сборки для каждой исполняющей среды, вы должны ориентироваться на наименьший общий знаменатель. Это относительно легко при желании поддерживать только непосредственных предшественников .NET 5: скажем, если вы нацеливаетесь на .NET Core 3.0, то ваша библиотека будет работать под управлением .NET Core 3.0, .NET Core 3.1 и .NET 5+.

Ситуация становится более запутанной, когда вы хотите поддерживать также .NET Framework и Xamarin. Дело в том, что каждая такая исполняющая среда имеет CLR и BCL с перекрывающимися функциональными средствами — ни одна исполняющая среда не является чистым подмножеством остальных.

.NET Standard решает проблему путем определения искусственных подмножеств, которые обеспечивают работу под управлением целого диапазона унаследованных исполняющих сред. Ориентируясь на .NET Standard, вы можете легко писать библиотеки с широким охватом.

.NET Standard — не исполняющая среда, а просто спецификация, описывающая минимальный базовый уровень функциональности (типы и члены), который гарантирует совместимость с определенным набором исполняющих сред. Концепция похожа на интерфейсы C#: стандарт .NET Standard подобен интерфейсу, который конкретные типы (исполняющие среды) могут реализовывать.

.NET Standard 2.0

Наиболее полезной версией является *.NET Standard 2.0*. Библиотека, которая ориентирована на .NET Standard 2.0, а не на специфическую исполняющую среду, будет работать без каких-либо изменений под управлением большинства современных и унаследованных исполняющих сред, все еще используемых в настоящее время, в том числе:

- .NET Core 2.0+
- UWP 10.0.16299+
- Mono 5.4+ (CLR/BCL из более старых версий Xamarin)
- .NET Framework 4.6.1+

Для нацеливания на .NET Standard 2.0 добавьте в свой файл `.csproj` следующие строки:

```
<PropertyGroup>
  <TargetFramework>netstandard2.0</TargetFramework>
<PropertyGroup>
```

Большинство API-интерфейсов, описанных в книге, поддерживаются стандартом .NET Standard 2.0.

.NET Standard 2.1

.NET Standard 2.1 является надмножеством стандарта .NET Standard 2.0, которое поддерживает (только) следующие платформы:

- .NET Core 3+
- Mono 6.4+

.NET Standard 2.1 не поддерживается ни одной из версий .NET Framework (и на момент написания главы даже UWP), что делает его менее полезным, чем .NET Standard 2.0.

В частности, в .NET Standard 2.1 (но не в .NET Standard 2.0) доступны следующие API-интерфейсы:

- `Span<T>` (глава 23)
- `Reflection.Emit` (глава 18)
- `ValueTask<T>` (глава 14)

Более старые стандарты .NET Standard

Существуют также более старые стандарты .NET Standard, самыми заметными из которых считаются 1.1, 1.2, 1.3 и 1.6. Стандарт с более высоким номером всегда является строгим надмножеством стандарта с меньшим номером. Например, при написании библиотеки, которая нацелена на .NET Standard 1.6, вы будете поддерживать не только последние версии основных исполняющих сред, но также .NET Core 1.0. А если вы нацеливаете библиотеку на .NET Standard 1.3, то будет поддерживаться все, что уже упоминалось, плюс .NET Framework 4.6.0 (табл. 5.1).

Таблица 5.1. Старые стандарты .NET Standard

Если вы нацеливаете библиотеку на…	То будут также поддерживаться…
.NET Standard 1.6	.NET Core 1.0
.NET Standard 1.3	Указанное выше плюс .NET Framework 4.6.0
.NET Standard 1.2	Указанное выше плюс .NET Framework 4.5.1, Windows Phone 8.1, WinRT для Windows 8.1
.NET Standard 1.1	Указанное выше плюс .NET Framework 4.5.0, Windows Phone 8.0, WinRT для Windows 8.0

В стандартах 1.x отсутствуют тысячи API-интерфейсов, которые находятся в стандарте 2.0, в том числе большинство того, что мы описываем в этой книге. В результате нацеливание на какой-то стандарт 1.x становится гораздо более затруднительным, особенно в случае, когда нужна интеграция с существующим кодом или библиотеками.

Вы можете думать о стандарте .NET Standard как о наименьшем общем знаменателе. В случае стандарта .NET Standard 2.0 исполняющие среды, которые ему следуют, имеют похожую библиотеку BCL, поэтому наименьший общий знаменатель является крупным и полезным. Тем не менее, если вас также интересует совместимость с инфраструктурой .NET Core 1.0 (с ее значительно усеченной библиотекой BCL), тогда наименьший общий знаменатель — .NET Standard 1.x — значительно сужается и становится менее полезным.

Совместимость .NET Framework и .NET 5

Поскольку инфраструктура .NET Framework существует настолько долго, нередко можно встретить библиотеки, которые доступны *только* для .NET Framework (без каких-либо эквивалентов для .NET Standard, .NET Core или .NET 5).

Чтобы смягчить такую ситуацию, проектам .NET 5 и .NET Core разрешено ссылаться на сборки .NET Framework при соблюдении описанных ниже условий.

- Если сборка .NET Framework обратится к API-интерфейсу, не поддерживаемому в .NET Core, тогда сгенерируется исключение.
- Распознавание нетривиальных зависимостей может быть безуспешным (и часто таковым оказывается).

Скорее всего, на практике это будет работать в простых случаях, таких как сборка, которая представляет собой оболочку для неуправляемой DLL-библиотеки.

Версии исполняющих сред и языка C#

Компилятор C# автоматически выбирает версию языка в зависимости от исполняющей среды, на которую ориентируется проект:

- для .NET 5 он выбирает C# 9;
- для .NET Core 3.x, Xamarin и .NET Standard 2.1 он выбирает C# 8;
- для .NET Core 2.x, .NET Framework, .NET Standard 2.0 и предшествующих версий он выбирает C# 7.3.

Причина в том, что недавние версии языка C# полагаются на типы, которые доступны только в последних версиях исполняющих сред.

Ссылочные сборки

Когда ваш проект нацелен на .NET Standard, он неявно ссылается на сборку по имени `netstandard.dll`, которая содержит все разрешенные типы и члены для выбранной версии .NET Standard. Она называется *ссылочной сборкой*, потому что существует только в интересах компилятора и не содержит скомпилированный код. Во время выполнения "настоящие" сборки идентифицируются через атрибуты перенаправления сборок (выбор сборок зависит от того, под управлением какой исполняющей среды и платформы в итоге будет производиться запуск).

Интересно отметить, что похожие вещи происходят при нацеливании на .NET 5. Ваш проект неявно ссылается на набор ссылочных сборок, типы которых отражают содержимое сборок времени выполнения для выбранной версии .NET. Это помогает управлять версиями и межплатформенной совместимостью и также позволяет нацеливаться на версию .NET, которая отличается от версии, установленной на вашей машине.

Среда CLR и библиотека BCL

Системные типы

Наиболее фундаментальные типы находятся прямо в пространстве имен `System`. В их состав входят встроенные типы C#, базовый класс `Exception`, базовые классы `Enum`, `Array` и `Delegate`, а также типы `Nullable`, `Type`, `DateTime`, `TimeSpan` и `Guid`. Кроме того, пространство имен `System` вклю-

чает типы для выполнения математических функций (Math), генерации случайных чисел (Random) и преобразования между различными типами (Convert и BitConverter).

Фундаментальные типы описаны в главе 6 вместе с интерфейсами, которые определяют стандартные протоколы, используемые повсеместно в .NET для решения таких задач, как форматирование (IFormattable) и сравнение порядка (IComparable).

В пространстве имен System также определен интерфейс IDisposable и класс GC для взаимодействия со сборщиком мусора; мы рассмотрим их в главе 12.

Обработка текста

Пространство имен System.Text содержит класс StringBuilder (редактируемый или *изменяемый* родственник string) и типы для работы с кодировками текста, такими как UTF-8 (Encoding и его подтипы). Мы раскроем их в главе 6.

Пространство имен System.Text.RegularExpressions содержит типы, которые выполняют расширенные операции поиска и замены на основе образца; такие типы будут описаны в главе 25.

Коллекции

В .NET предлагаются разнообразные классы для управления коллекциями элементов. Они включают структуры, основанные на списках и словарях, и работают в сочетании с набором стандартных интерфейсов, которые унифицируют их общие характеристики. Все типы коллекций определены в следующих пространствах имен, описанных в главе 7:

```
System.Collections                  // Необобщенные коллекции
System.Collections.Generic          // Обобщенные коллекции
System.Collections.Specialized      // Строго типизированные коллекции
System.Collections.ObjectModel      // Базовые типы для создания
                                    // собственных коллекций
System.Collections.Concurrent       // Коллекции, безопасные в
                                    // отношении потоков (глава 22)
```

Запросы

Язык LINQ позволяет выполнять безопасные в отношении типов запросы к локальным и удаленным коллекциям (например, к таблицам SQL Server) и описан в главах 8–10. Крупное преимущество языка LINQ заключается в том, что он предоставляет согласованный API-интерфейс запросов для разнообразных предметных областей. Основные типы находятся в перечисленных ниже пространствах имен:

```
System.Linq                         // LINQ to Objects и PLINQ
System.Linq.Expressions             // Для ручного построения выражений
System.Xml.Linq                     // LINQ to XML
```

XML и JSON

XML и JSON поддерживаются в .NET повсеместно. Внимание в главе 10 сосредоточено целиком на LINQ to XML — легковесной модели документных объектов (Document Object Model — DOM) для XML, которую можно конструировать и опрашивать с помощью LINQ. В главе 11 описаны высокопроизводительные низкоуровневые классы для чтения/записи XML, схем и таблиц стилей XML, а также типы для работы с JSON:

```
System.Xml              // XmlReader, XmlWriter и старая модель W3C DOM
System.Xml.Linq         // Объектная модель документа LINQ to XML
System.Xml.Schema       // Поддержка для XSD
System.Xml.Serialization// Декларативная сериализация XML для типов .NET
System.Xml.XPath        // Язык запросов XPath
System.Xml.Xsl          // Поддержка таблиц стилей

System.Text.Json       // Средство чтения/записи JSON и DOM
```

Сериализатор JSON раскрывается в дополнительных материалах, доступных для загрузки на веб-сайте издательства.

Диагностика

В главе 13 мы рассмотрим регистрацию в журнале и утверждения, а также покажем, как взаимодействовать с другими процессами, выполнять запись в журнал событий Windows и проводить мониторинг производительности. Соответствующие типы определены в пространстве имен `System.Diagnostics` и его подпространствах.

Параллелизм и асинхронность

Многим современным приложениям в каждый момент времени приходится иметь дело с несколькими действиями. Начиная с версии C# 5.0, решение стало проще за счет асинхронных функций и таких высокоуровневых конструкций, как задачи и комбинаторы задач. Все это подробно объясняется в главе 14, которая начинается с рассмотрения основ многопоточности. Типы для работы с потоками и асинхронными операциями находятся в пространствах имен `System.Threading` и `System.Threading.Tasks`.

Потоки данных и ввод-вывод

В .NET предоставляется потоковая модель для низкоуровневого ввода-вывода. Потоки данных обычно применяются для чтения и записи напрямую в файлы и сетевые подключения и могут соединяться в цепочки либо помещаться внутрь декорированных потоков с целью добавления функциональности сжатия или шифрования. В главе 15 описана потоковая архитектура, а также специфическая поддержка для работы с файлами и каталогами, сжатием, изолированным хранилищем, каналами и файлами, отображенными в память. Тип `Stream` и типы ввода-вывода определены в пространстве имен `System.IO` и его подпространствах, а типы WinRT для файлового ввода-вывода — в пространстве имен `Windows.Storage` и его подпространствах.

Работа с сетями

С помощью типов из пространства имен `System.Net` можно напрямую работать со стандартными сетевыми протоколами, такими как HTTP, FTP, TCP/IP и SMTP. В главе 16 будет показано, как взаимодействовать с использованием каждого из упомянутых протоколов, начиная с простых задач вроде загрузки вебстраницы и заканчивая применением TCP/IP для извлечения сообщений электронной почты POP3. Ниже перечислены пространства имен, которые будут рассмотрены:

```
System.Net
System.Net.Http          // HttpClient
System.Net.Mail          // Для отправки электронной почты через SMTP
System.Net.Sockets       // TCP, UDP и IP
```

Сборки, рефлексия и атрибуты

Сборки, в которые компилируются программы на C#, состоят из исполняемых инструкций (представленных на языке IL) и метаданных, которые описывают типы, члены и атрибуты программы. С помощью рефлексии можно просматривать метаданные во время выполнения и предпринимать действия вроде динамического вызова методов. Посредством пространства имен `Reflection.Emit` можно конструировать новый код на лету.

В главе 17 мы опишем строение сборок и объясним, как их динамически загружать и изолировать. В главе 18 мы раскроем рефлексию и атрибуты — покажем, как инспектировать метаданные, динамически вызывать функции, записывать специальные атрибуты, выпускать новые типы и производить разбор низкоуровневого кода IL. Типы для применения рефлексии и работы со сборками находятся в следующих пространствах имен:

```
System
System.Reflection
System.Reflection.Emit
```

Динамическое программирование

В главе 19 мы рассмотрим несколько паттернов для динамического программирования и работы со средой DLR. Мы покажем, как реализовать паттерн “Посетитель” (Visitor), создавать специальные динамические объекты и взаимодействовать с IronPython. Типы, предназначенные для динамического программирования, находятся в пространстве имен `System.Dynamic`.

Криптография

.NET обеспечивает всестороннюю поддержку для популярных протоколов хеширования и шифрования. В главе 20 мы раскроем хеширование, симметричное шифрование и шифрование с открытым ключом, а также API-интерфейс Windows Data Protection. Типы для этого определены в следующих пространствах имен:

```
System.Security
System.Security.Cryptography
```

Расширенная многопоточность

Асинхронные функции в C# значительно облегчают параллельное программирование, поскольку снижают потребность во взаимодействии с низкоуровневыми технологиями. Тем не менее, все еще возникают ситуации, когда нужны сигнальные конструкции, локальное хранилище потока, блокировки чтения/записи и т.д. Данные вопросы подробно обсуждаются в главе 21. Типы, связанные с многопоточностью, находятся в пространстве имен `System.Threading`.

Параллельное программирование

В главе 22 мы рассмотрим библиотеки и типы для работы с многоядерными процессорами, включая API-интерфейсы для реализации параллелизма задач, императивного параллелизма данных и функционального параллелизма (PLINQ).

Span<T> и Memory<T>

Для содействия микро-оптимизации в "горячих" точках в плане производительности среда CLR предлагает несколько типов, которые помогают программировать так, чтобы снизить нагрузку на диспетчер памяти. Двумя ключевыми типами подобного рода являются `Span<T>` и `Memory<T>`, которые будут описаны в главе 23.

Возможность взаимодействия с собственным кодом и COM

Вы можете взаимодействовать с собственным кодом и с кодом COM. Возможность взаимодействия с собственным кодом позволяет вызывать функции из неуправляемых DLL-библиотек, регистрировать обратные вызовы, отображать структуры данных и работать с собственными типами данных. Возможность взаимодействия с COM позволяет обращаться к типам COM (на машинах Windows) и открывать для COM доступ к типам .NET. Типы, поддерживающие такую функциональность, определены в пространстве имен System.Runtime.InteropServices и рассматриваются в главе 24.

Регулярные выражения

В главе 25 мы покажем, как можно использовать регулярные выражения для сопоставления с символьными шаблонами в строках.

Сериализация

.NET предлагает несколько систем для сохранения и восстановления объектов в двоичном или текстовом представлении. Такие системы могут применяться для передачи данных, а также сохранения и восстановления объектов из файлов. В дополнительных материалах, доступных для загрузки на веб-сайте издательства, раскрываются все четыре механизма сериализации: двоичный сериализатор, (обновленный) сериализатор JSON, сериализатор XML и сериализатор на основе контрактов данных.

Компилятор Roslyn

Сам компилятор C# написан на языке C# — проект называется Roslyn, а библиотеки доступны в виде пакетов NuGet. С помощью этих библиотек вы можете эксплуатировать функциональность компилятора многими способами помимо компиляции исходного кода в сборку, скажем, писать инструменты для анализа и рефакторинга кода. Компилятор Roslyn рассматривается в дополнительных материалах, доступных для загрузки на веб-сайте издательства.

Прикладные слои

Приложения, основанные на пользовательском интерфейсе, можно разделить на две категории: *тонкий клиент*, равнозначный веб-сайту, и *обогащенный клиент*, представляющий собой программу, которую конечный пользователь должен загрузить и установить на компьютере или на мобильном устройстве.

Для разработки приложений тонких клиентов на языке C# предусмотрена инфраструктура ASP.NET Core, которая запускается в среде Windows, Linux и macOS. Кроме того, инфраструктура ASP.NET Core позволяет реализовывать API-интерфейсы для веб-сети.

Для разработки приложений обогащенных клиентов на выбор доступно несколько API-интерфейсов:

- слой Windows Desktop, который включает API-интерфейсы WPF и Windows Forms и функционирует на настольных компьютерах Windows 7/8/10;
- платформа UWP, запускаемая на настольных компьютерах и устройствах Windows 10;
- платформа Xamarin, функционирующая на мобильных устройствах iOS и Android (в следующем крупном выпуске запланирована поддержка настольных компьютеров macOS).

Вдобавок существуют сторонние библиотеки вроде Avalonia, которые предлагают межплатформенную поддержку пользовательских интерфейсов.

ASP.NET Core

ASP.NET Core — это легковесный модульный преемник ASP.NET, поддерживающий популярный паттерн *MVC* (Model-View-Controller — модель-представление-контроллер). ASP.NET Core подходит для создания веб-сайтов, API-интерфейсов на основе REST и микрослужб. Кроме того, ASP.NET Core может работать в сочетании с двумя популярными фреймворками для одностраничных приложений: React и Angular.

ASP.NET Core функционирует под управлением Windows, Linux и macOS и может самостоятельно размещаться в специальном процессе. В отличие от своего предшественника из .NET Framework (ASP.NET) архитектура ASP.NET Core не зависит от `System.Web` и от исторического багажа веб-форм.

Как и любая архитектура тонкого клиента, ASP.NET Core обладает следующими общими преимуществами по сравнению с обогащенными клиентами:

- отсутствует развертывание на клиентской стороне;
- клиент может запускаться на любой платформе, которая поддерживает веб-браузер;
- легко развертывать обновления.

Windows Desktop

Прикладной слой Windows Desktop предлагает на выбор два API-интерфейса для реализации пользовательских интерфейсов в приложениях обогащенных клиентов: WPF и Windows Forms. Оба API-интерфейса работают в среде Windows Desktop/Server 7–10.

WPF

Инфраструктура WPF появилась в 2006 году и с тех пор неоднократно расширялась. В отличие от своей предшественницы, Windows Forms, она явно визуализирует элементы управления с использованием DirectX, обеспечивая перечисленные ниже преимущества.

- WPF поддерживает развитую графику, включая произвольные трансформации, трехмерную визуализацию, мультимедиа-возможности и подлинную прозрачность. Оформление поддерживается через стили и шаблоны.
- Основная единица измерения не базируется на пикселях, поэтому приложения корректно отображаются при любой настройке DPI (dots per inch — точек на дюйм).
- Она располагает обширной и гибкой поддержкой динамической компоновки, означающей возможность локализации приложения без опасности того, что элементы будут перекрывать друг друга.
- Применение DirectX делает визуализацию быстрой и способной извлекать преимущества от аппаратного ускорения графики.
- Она предлагает надежную привязку к данным.
- Пользовательские интерфейсы могут быть описаны декларативно в XAML-файлах, которые можно сопровождать независимо от файлов отделенного кода, что помогает отделить внешний вид от функциональности.

Инфраструктура WPF требует некоторого времени на изучение из-за своего размера и сложности. Типы, предназначенные для написания WPF-приложений, находятся в пространстве имен `System.Windows` и во всех его подпространствах за исключением `System.Windows.Forms`.

Windows Forms

Windows Forms — это API-интерфейс обогащенного клиента, который поставлялся с первой версией .NET Framework в 2000 году. По сравнению с WPF она является относительно простой технологией, которая предлагает большинство возможностей, необходимых во время разработки типового Windows-приложения. Она также играла важную роль в сопровождении унаследованных приложений.

Тем не менее, в сравнении с WPF инфраструктура Windows Forms обладает рядом недостатков, большинство из которых объясняется тем, что она является оболочкой для GDI+ и библиотеки элементов управления Win32.

- Несмотря на предоставление механизмов осведомленности о настройках DPI, все же слишком легко писать приложения, которые некорректно отображаются на клиентах с настройками DPI, отличающимися от таких настроек у разработчика.
- Для рисования нестандартных элементов управления используется API-интерфейс GDI+, который вопреки достаточно высокой гибкости медленно визуализирует крупные области (и без двойной буферизации может вызывать мерцание).
- Элементы управления лишены подлинной прозрачности.
- Большинство элементов управления не поддерживают компоновку. Например, поместить элемент управления изображением внутрь заголовка элемента управления вкладкой не удастся. Настройка списковых представлений, полей с раскрывающимися списками и элементов управления с вкладками, которая была бы тривиальной в WPF, требует много времени и сил.
- Трудно корректно и надежно реализовать динамическую компоновку.

Последний пункт является веской причиной отдавать предпочтение WPF перед Windows Forms, даже если разрабатывается бизнес-приложение, которому необходим только пользовательский интерфейс, а не учет "поведенческих особенностей пользователей". Элементы компоновки в WPF, подобные `Grid`, упрощают организацию меток и текстовых полей таким образом, что они будут всегда выровненными — даже при смене языка локализации — без запутанной логики и какого-либо мерцания. Кроме того, не придется приводить все к наименьшему общему знаменателю в смысле экранного разрешения — элементы компоновки WPF изначально проектировались с поддержкой изменения размеров.

В качестве положительного момента следует отметить, что инфраструктура Windows Forms относительно проста в изучении и все еще поддерживается в немалом количестве сторонних элементов управления.

Типы Windows Forms находятся в пространствах имен `System.Windows.Forms` (сборка `System.Windows.Forms.dll`) и `System.Drawing` (сборка `System.Drawing.dll`). Последнее пространство имен также содержит типы GDI+ для рисования специальных элементов управления.

UWP и WinUI 3

Универсальная платформа Windows (Universal Windows Platform — UWP) представляет собой API-интерфейс обогащенного клиента, который предназначен для разработки сенсорных приложений, ориентированных на настольные компьютеры и устройства Windows 10. Слово "универсальная" относится к ее способности функционировать на различных устройствах Windows 10, в том числе Xbox, Surface Hub и HoloLens. Однако платформа UWP не совместима с ранними версиями Windows, включая Windows 7 и Windows 8/8.1.

API-интерфейс UWP использует XAML и кое в чем похож на WPF. Ниже перечислены его ключевые отличия.

- Приложения UWP поставляются главным образом через Магазин Microsoft.
- Приложения UWP работают в песочнице, чтобы уменьшить угрозу со стороны вредоносного программного обеспечения, а это означает, что они не могут выполнять такие задачи, как чтение или запись произвольных файлов, и их нельзя запускать с повышенными административными правами.
- Платформа UWP опирается на типы WinRT, которые являются частью операционной системы (Windows), а не управляемой исполняющей среды. В результате при написании приложений вы обязаны объявлять диапазон версий Windows 10 (скажем, от Windows 10 сборки 17763 до Windows 10 сборки 18362). Таким образом, вам необходимо либо ориентировать приложения на старый API-интерфейс, либо требовать от пользователей установки самого последнего обновления Windows.

Для решения последнего вопроса в Microsoft предлагают библиотеку WinUI 3, которая переносит API-интерфейсы WinRT из операционной системы в исполняющую среду (обеспечивая доступ к полностью управляемому интерфейсу). Библиотека WinUI 3 также поможет преодолеть разрыв между инфраструктурами Windows Desktop и UWP: вместо выбора одной из них появляется возможность смешивания и подгонки компонентов из обеих инфраструктур.

К API-интерфейсу UWP относятся пространства имен `Windows.UI` и `Windows.UI.Xaml`.

Xamarin и Xamarin Forms

Xamarin позволяет писать мобильные приложения на языке C#, которые ориентированы на операционную систему iOS или Android. Xamarin.iOS и Xamarin.Android функционируют под управлением .NET CLR/BCL на базе Mono (производный продукт исполняющей среды с открытым кодом Mono).

В дополнение к API-интерфейсам, специфичным для iOS и Android, имеется продукт под названием *Xamarin Forms*, который действует как верхний слой, позволяя одному проекту пользовательского интерфейса генерировать приложения и для iOS, и для Android. В настоящее время компания Microsoft работает над новым развитием Xamarin Forms, называемым MAUI (Multi-platform App UI — пользовательский интерфейс для многоплатформенных приложений), который даст возможность создавать также межплатформенные настольные приложения (для Windows и macOS). Новый продукт будет поставляться в версии Mono, согласованной с выходом .NET 6.

За дополнительной информацией обращайтесь на веб-сайты `https://www.xamarin.com` и `https://github.com/dotnet/maui`.