Сферы применения C#

В каком-то смысле C# для программирования можно считать тем же, чем является .NET для окружения Windows. Точно так же, как Microsoft добавляла в течение последнего десятилетия все новые и новые возможности в Windows и Windows API, расширялись и языки VB и C++. В результате VB и C++ превратились в довольно мощные языки программирования, но в то же время у них появились проблемы.

Главным достоинством языка Visual Basic является то, что он прост в понимании и позволяет легко выполнять многие программистские задачи, большей частью скрывая от разработчика детали Windows API и структуру компонентов COM. Недостаток заключается в том, что Visual Basic никогда не был полностью объектно-ориентированным. Крупные проекты в этом языке быстро становятся неорганизованными и плохо управляемыми. Из-за того что синтаксис VB унаследован от ранних версий BASIC (который, в свою очередь, был разработан для обучения новичков основам программирования, а не для создания серьезных проектов), он не позволяет писать хорошо структурированные или объектно- ориентированные программы.

Корни C++ лежат в определении языка ANSI C++. Он не является полностью ANSI C++ совместимым, так как Microsoft создала свой компилятор до появления стандарта ANSI C++, но очень близок к нему. К сожалению, это привело к появлению двух проблем. Во-первых, ANSI C++ был разработан более 10 лет назад, отсюда отсутствие поддержки современных концепций (таких, как строки Unicode и создание XML-документации) и наличие архаических синтаксических конструкций, предусмотренных для компиляторов вчерашнего дня (например, отделение определения от описания методов). Во-вторых, Microsoft пыталась приспособить C++ для решения высокопроизводительных задач в Windows, и для этого пришлось добавить в язык большое количество ключевых слов и различных библиотек. В результате язык стал представлять собой некое месиво. Достаточно спросить программиста о том, как много различных определений строки он может придумать: `char*`, `LPTSTR`, `string`, `CString` (в версии MFC), `CString` (в версии WTL), `wchar_t*`, `OLECHAR*` и т.д.

Что же касается .NET, то это абсолютно новая среда, которая должна привнести новые расширения в оба языка. Microsoft решила эту проблему путем добавления еще большего числа специфических ключевых слов к C++ и путем полной перестройки VB в VB.NET, язык, который сохраняет некоторый первоначальный синтаксис VB, но отличается от него настолько, что его смело можно считать новым языком.

Именно поэтому Microsoft решила предложить разработчикам альтернативу — язык, созданный с нуля специально для .NET. В результате появился C#. Официально Microsoft описывает C# как "простой, современный, объектно-ориентированный язык программирования с безопасными типами, производный от C и C++". Многие независимые обозреватели скорее всего изменили бы эту формулировку на "производный от C, C++ и Java". Подобные определения технически корректны, но не показывают легкости и элегантности языка. Синтаксически C# очень похож на C++ и Java, и даже совпадают некоторые ключевые слова. C# также использует блочную структуру со скобками ({}) для отметки блоков кода и точки с запятой для разделения выражений. Первое впечатление от кода на C# таково, что он выглядит как код C++ или Java. Однако если не брать в учет внешнее сходство, C# гораздо проще в изучении, чем C++, и сравним по сложности с Java. Его дизайн более созвучен с современными инструментами разработчика. Он был создан для того, чтобы предоставить программисту простоту использования VB и высокопроизводительный, низкоуровневый доступ к памяти по типу C++ в случае необходимости. К особенностям C# относятся:

- Полная поддержка классов и объектно-ориентированного программирования, включая наследование интерфейсов и реализаций, виртуальных функций и перегрузки операторов.
- Полный и хорошо определенный набор основных типов.
- Встроенная поддержка автоматической генерации XML-документации.
- Автоматическое освобождение динамически распределенной памяти.
- Возможность отметки классов и методов атрибутами, определяемыми пользователем. Это может быть полезно при документировании и способно воздействовать на процесс компиляции (например, можно пометить методы, которые должны компилироваться только в отладочном режиме).
- Полный доступ к библиотеке базовых классов .NET, а также легкий доступ к Windows API (если это действительно необходимо).

- Указатели и прямой доступ к памяти, если они необходимы. Однако язык разработан таким образом, что практически во всех случаях можно обойтись и без этого.
- Поддержка свойств и событий в стиле VB.
- Простое изменение ключей компиляции. Позволяет получать исполняемые файлы или библиотеки компонентов .NET, которые могут быть вызваны другим кодом так же, как элементы управления ActiveX (компоненты COM).
- Возможность использования C# для написания динамических web-страниц ASP.NET.

Надо отметить, что большинство из приведенного выше справедливо и для VB.NET, и для управляемого C++. Однако тот факт, что C# создан с нуля для работы с .NET, означает, что он более полно поддерживает все особенности .NET и предлагает в этом контексте более удобный синтаксис, чем остальные языки. Сам по себе язык C# похож на Java, однако есть некоторые улучшения, и, кроме того, Java не создан для работы в среде .NET.

Подводя черту, можно сказать, что C# является не только мощным языком, который не сложен в изучении, но и, пожалуй, единственным языком на рынке, который был создан на основе современных технологий и инструментов разработки. Изучая опыт предыдущих языков, Microsoft может гарантировать, что C# хорошо проработан и позволяет быстро получать высококачественный код.

Нужно все же отметить несколько ограничений C#. Одной из областей, для которых не предназначен этот язык, являются критичные по времени и высокопроизводительные программы, когда имеет значение, занимает исполнение цикла 1000 или 1050 машинных циклов, и освобождать ресурсы требуется немедленно. C++ останется в этой области наилучшим из языков низкого уровня. В C# отсутствуют некоторые ключевые моменты, необходимые для создания высокопроизводительных приложений, в частности подставляемые функции и деструкторы, выполнение которых гарантируется в определенных точках кода. Однако число приложений, попадающих в эту категорию, невелико.

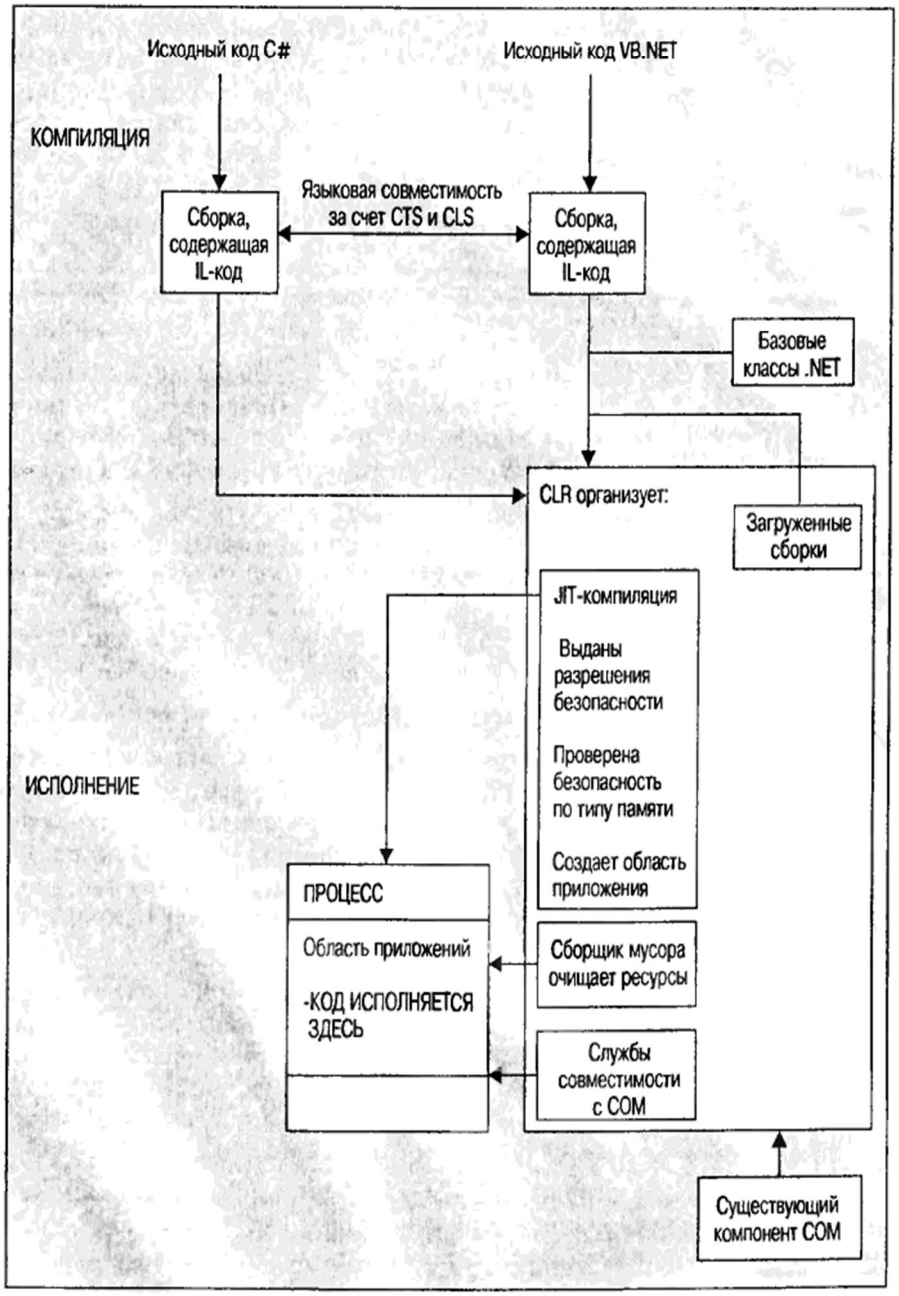
Новая среда разработки

Платформа .NET предлагает новую среду разработки — Visual Studio.NET. До сих пор каждый язык имел свою собственную среду разработки в рамках Visual Studio 6. Программисты C++ применяли свою среду (ее нередко ошибочно называют просто Visual Studio 6), разработчики на VB использовали так называемую VB6 IDE, а соответствующим эквивалентом для J++ являлась среда Visual J++. И все эти среды, реализуя многие функции друг друга, выглядели совершенно разными. Самыми обделенными оказались разработчики ASP-страниц, они обычно применяли Visual Interdev, среду, которая предлагала лишь немногие возможности по генерации и отладке кода из тех, что требовались пользователям компилируемых языков (однажды автор слышал, как Visual Interdev назвали Визуальным блокнотом). Каждая из этих сред (за исключением Interdev) обладала своими преимуществами, но все же это были совершенно разные среды, с отличающимися интерфейсами пользователя.

Каждая среда разработки имеет свои слабые и сильные стороны, которые в большой степени отражают достоинства и недостатки соответствующего языка. Например, Visual Studio (C++) обладает мощными возможностями отладки, не доступными в других средах, в то время как среда разработки Visual Basic хороша в случае визуального программирования, когда с помощью одного щелчка мыши можно поместить в форму огромное число элементов управления ActiveX, а соответствующий VB-код среда сгенерирует автоматически. Так как целью .NET является полная совместимость языков (включая возможность перехода от кода на одном языке к коду на другом языке в отладчике), то наличие отдельных сред разработки совершенно недопустимо.

Платформа .NET обладает новой средой разработки, Visual Studio.NET, которая может одинаково работать с кодом C++, C#, VB.NET и ASP.NET. Visual Studio.NET сочетает в себе все лучшие свойства соответствующих сред Visual Studio 6. Когда вы начнете писать в Visual Studio.NET, вам придется привыкать к другому расположению окон и к другой системе меню, однако положительной стороной является то, что это более мощная среда разработки по сравнению со всеми предыдущими.

некоторые из базовых классов .NET (практически невозможно создать приложение, которое будет делать что- нибудь полезное, но при этом не будет использовать базовые классы .NET):



На диаграмме прямоугольники показывают основные компоненты, связанные с компиляцией и исполнением программы, а стрелки – исполняемые задачи. В верхней части рисунка представлен процесс раздельной компиляции каждого проекта в сборку. Две сборки способны взаимодействовать друг с другом благодаря свойствам совместимости языков .NET. Нижняя часть диаграммы демонстрирует процесс JIT-компиляции из IL в сборках в машинный код, который выполняется в области приложения внутри процесса. Показаны некоторые действия, которые выполняет код внутри CLR для достижения этой цели.

Компиляция

Перед запуском программа должна быть откомпилирована. Однако, в отличие от преж-

(в противоположность ссылкам) разрешены только в некоторых фрагментах кода в C#, но никак не в VB. Использование указателей в коде немедленно нарушит проверку безопасности типов памяти, выполняемую CLR. И хотя требование безопасности типов ухудшает производительность, все же в большинстве случаев выгоды, получаемые от использования служб, предоставляемых .NET (например, области приложений), и основывающиеся как раз на безопасности типов, будут перекрывать это ухудшение.

Точно так же разработчикам на VB не придется думать о типах переменных, поскольку VB автоматически выполнит все необходимые преобразования. Если в каком-то фрагменте кода вместо типа `String` будет передан тип `Integer`, VB преобразует `Integer` в `String`. Философия IL и .NET состоит в том, что удобство неявных преобразований перевешивается связанными с этим проблемами безопасности типов и, в частности, потенциальными трудноуловимыми ошибками времени выполнения, возникающими из-за неверных типов данных. Поэтому при написании кода, предназначенного для .NET, эти преобразования придется целенаправленно выполнять вручную.

Существует несколько серьезных причин, по которым важен строгий контроль типов,— на самом деле некоторые части .NET не будут без него работать.

- В первую очередь среда исполнения общего языка полагается на способность проверки кода для определения того, какие операции необходимо выполнить перед непосредственным запуском кода. Это важно как с точки зрения предоставления привилегий доступа, так и с точки зрения проверки того, что код не может повредить другой код, который выполняется в другой области приложений, но в том же адресном пространстве. Промежуточный язык был разработан для облегчения проверок, и совершенно очевидно, что если бы присутствовали какие-то сомнения насчет типов данных в коде, такие проверки не были бы возможны.
- Для того чтобы выяснить, какую память надо освободить, сборщик мусора должен уметь определять тип данных, содержащийся в каждой ячейке памяти (иначе он не сможет узнать, какой объем памяти занимает переменная). Опять же, любые неоднозначности в типах данных вызовут проблемы, и среда исполнения .NET начнет работать неправильно.
- Совместимость языков, одна из основ платформы .NET, базируется на хорошо определенном и цельном наборе типов данных.

Система типов, используемая промежуточным языком, известна как общая система типов (CTS) (см. ниже).

Свойства IL: подведение итогов

К основным характеристикам промежуточного языка относятся:

- Объектно-ориентированный подход с одиночным наследованием классов
- Интерфейсы
- Типы по ссылке и значению
- Обработка ошибок с помощью исключений
- Система строгого контроля типов

Любой язык, который рассчитан на платформу .NET, должен поддерживать эти концепции. Что касается существующих языков, то это не является проблемой для C++, однако означает, что определение Visual Basic должно быть улучшено для поддержки этих требований. Новая версия Visual Basic, VB.NET, существенно отличается от предыдущей версии (VB6). C# изначально разрабатывался с учетом всех требований промежуточного языка, поэтому он полностью поддерживает указанные концепции.

Совместимость языков

Работа с .NET означает, что код компилируется в промежуточный язык и что необходимо программировать с использованием традиционного объектно- ориентированного подхода. Однако этого недостаточно для обеспечения совместимости языков. В конце концов, C++ и Java используют одинаковые объектно-ориентированные парадигмы, однако не считаются совместимыми.

В этом разделе рассматриваются межъязыковые возможности, предоставляемые .NET. В значительной степени они основываются на использовании общей системы типов и общей спецификации языка.

- Visual Studio.NET — интегрированная среда разработки, с помощью которой можно писать, компилировать и отлаживать код на всех языках .NET, включая C#, VB.NET, управляемый C++, страницы ASP.NET и неуправляемый код C++. Visual Studio.NET обсуждается в главе 8.
- Компиляторы командной строки для C#, VB.NET и C++.
- ILDASM — утилита с оконным интерфейсом, которую можно использовать для просмотра содержимого сборки, включая манифест и метаданные. ILDASM описывается в главе 10.

Сборщик мусора

Сборщик мусора — это ответ .NET на проблемы управления памятью, в частности, на вопрос, связанный с перераспределением памяти, требующейся приложениям. До сих пор в Windows использовались две методики перераспределения памяти, динамически запрашиваемой процессами у системы: методика перераспределения памяти самим приложением и применение в объектах счетчиков ссылок. В дополнение Java использует сборщик мусора, аналогичный работающему в .NET.

Методика применения программного кода для перераспределения памяти используется низкоуровневыми высокопроизводительными языками, например C++. Эта методика эффективна и имеет то преимущество, что ресурсы не используются сверх положенного срока. Большим недостатком, однако, является значительное число ошибок. Код, который запрашивает память, должен в конце информировать систему о том, что память ему больше не требуется. В C++ для этого предусмотрено ключевое слово `delete`, кроме того, существуют различные функции API, предназначенные для той же цели. Программисты должны быть очень осторожны и внимательно следить за тем, чтобы освобождалась вся используемая память. Об этом нередко забывают, что приводит к утечкам памяти. Современные среды разработчика предоставляют инструменты для обнаружения утечек памяти, но эти ошибки все же очень сложно выявить, так как проявляются они лишь тогда, когда происходит утечка большого количества памяти и Windows в определенный момент просто отказывается выделить память процессу из-за ее отсутствия. К этому моменту работа всего компьютера может сильно замедлиться из-за потребления большого объема памяти.

Использование счетчиков ссылок применяется COM-объектами. Идея заключается в том, что каждый COM-объект поддерживает информацию о том, как много клиентов в данный момент используют ссылки на него. Когда число ссылок становится равным нулю, компонент уничтожает себя и освобождает память и ресурсы. Проблема здесь заключается в том, что COM-объект рассчитывает на корректное поведение клиентов, которые должны сообщать ему об окончании своей работы с объектом (что осуществляется путем вызова метода `IUnknown.Release()`). Стоит только одному из клиентов не сделать этого, и объект останется в памяти. В некоторых случаях это является потенциально еще более серьезной проблемой, чем простая утечка памяти в C++, так как COM-объект может существовать в своем собственном процессе, а это означает, что он никогда не будет удален системой (в случае утечек памяти система, по крайней мере, может освободить всю память по завершении процесса).

Какое же решение применяется в .NET?

Среда исполнения .NET полагается на так называемый **сборщик мусора**, который представляет собой программу, чьей целью является освобождение памяти. Смысл заключается в том, что вся динамически запрашиваемая память распределяется в куче (что справедливо для всех языков). По мере того как .NET выясняет, что для данного процесса куча полностью заполняется и, следовательно, требует очистки, она вызывает сборщик мусора. Сборщик мусора просматривает переменные кода, находящиеся в данный момент в области видимости, исследуя ссылки на объекты, хранящиеся в куче, для определения того, какие из них доступны в коде, или иными словами, какие объекты содержат ссылки на себя. Все объекты, на которые нет ссылок, считаются более недоступными из кода и должны быть уничтожены.

Посмотрим, как это работает на практике, воспользовавшись фрагментом кода на C#:

```
{
   TextBox UserInputArea;
   UserInputArea = new TextBox();
   TextBox txtBoxCopy = UserInputArea;

   // Допустим, что сборщик мусора вызван здесь
   // Обработка данных
```

можно осуществить преобразование из uint в long, так как диапазон uint по сути дела является верхней частью диапазона long (от нуля и выше). Также можно преобразовывать целые числа в числа с плавающей точкой. Здесь правила немного отличаются. Можно осуществлять преобразования между типами одного размера, например int/uint в float и long/ulong в double, и допустимо также преобразование long/ulong в float. В процессе этого можно потерять 4 байта данных, однако это всего лишь означает, что величина float будет иметь меньшую точность, чем в случае double; значение величины вообще не затрагивается. Преобразование float в double следует тем же правилам, что для целых со знаком.

Явные преобразования

Существует большое число преобразований, которые не могут быть выполнены неявно. При попытке сделать это компилятор выдаст ошибку. Нельзя выполнить неявно преобразования:

- int в short — возможна потеря данных
- int в uint — возможна потеря данных
- uint в int — возможна потеря данных
- float в int — будет потеряно все, что идет после десятичной точки
- любой числовой тип в char — будут потеряны данные
- decimal в любой числовой тип — десятичный тип внутренне устроен по-другому, нежели форматы для целых чисел и чисел с плавающей точкой

Однако эти преобразования могут быть выполнены явно с использованием приведения. Когда один тип приводится к другому, компилятор вынужден выполнять преобразование. Типичный синтаксис приведения:

```
long val = 30000;
int I = (int) val;      // Допустимое преобразование.
                        // Максимум int равен 2147483647.
```

Тип, к которому необходимо привести значение, указывается в круглых скобках перед этим значением. Для программистов, хорошо знакомых с C, такой синтаксис является привычным. Что касается специальных ключевых слов для приведения в C++, таких как static_cast, то в C# их нет, и вам придется использовать более ранний синтаксис C.

Эта операция может быть опасной. Вы должны точно знать, что делаете. Даже простое преобразование из long в int способно вызвать проблему, если значение переменной long больше, чем максимально возможное значение int. Например:

```
long val = 3000000000;
int I = (int)val;       // Неправильное преобразование.
                        // Максимальное значение int равно 2147483647.
```

В данном случае не выдается сообщение об ошибке, но и ожидаемый результат не получается. Если мы выполним этот код и выведем на экран величину, которая содержится в I, то получим:

```
-1294967296
```

Никогда не следует полагаться на то, что преобразование типа даст ожидаемый результат. В C# есть оператор checked, который можно использовать для проверки того, что операция не вызывает переполнение стека. С помощью этого оператора можно выяснить, безопасно ли приведение типа, и заставить среду исполнения сгенерировать исключение переполнения, если это не так:

```
long val = 3000000000;
int i = checked((int)val);
```

Помня о том, что любое приведение типов потенциально небезопасно, в программе необходимо предусмотреть код, который будет обрабатывать возможные ошибки приведения типов. Ниже мы рассмотрим обработку исключений с помощью конструкции try ... catch, которая является полезным и необходимым инструментом программиста.

С помощью приведения типов можно выполнять самые разные преобразования, например:

```
double price = 25.30;
int approximatePrice = (int) (price + 0.5);
```

интеллектуально реагировать на конкретные типы ошибок. Например, для некоторых ошибок процедура может выполнить какие-то восстановительные работы и продолжить исполнение. В случае других, более серьезных ошибок процедура может произвести запись в журнал и прекратить дальнейшее исполнение кода.

Каждый метод в программе на C# необходимо снабжать обработчиком ошибок.

Более подробно структурированная обработка ошибок рассматривается в главе 6.

Структура программы

К настоящему моменту мы познакомились с основными "строительными блоками", входящими в состав языка C#: с типами данных и с операторами управления ходом выполнения программы. Но как соединить эти кирпичики вместе для того, чтобы получить завершенную программу? Ключевой момент здесь заключается в работе с классами.

Классы

Классы играют важнейшую роль в программе на C#, и следующие две главы полностью посвящены объектно-ориентированному программированию в C#. Для начала же необходимо получить представление об основах работы с классами в C#, так как совершенно невозможно написать программу на C# без их использования. Классы по существу являются шаблонами, из которых можно создавать объекты. Каждый объект содержит данные и имеет методы для работы с этими данными и для доступа к ним. Класс определяет, какие данные может содержать каждый объект этого класса, но не содержит самих данных. Например, класс, представляющий потребителя, может определять такие поля, как `CustomerID`, `FirstName`, `LastName` и `Address`, которые будут использоваться для хранения информации о конкретном клиенте. Позже можно создать экземпляр объекта этого класса для представления конкретного клиента и заполнить поля этого объекта.

Члены класса

Данные и функции внутри класса называют **членами** класса. Официальная терминология Microsoft делает различие между данными класса и функциями класса. Помимо этих членов, классы могут содержать вложенные типы (например, другие классы).

Данные класса

Данные класса — это те члены, которые содержат данные для класса: поля, константы и события.

- Поля представляют собой любые переменные, связанные с классом. Если определить переменную на уровне класса, то на самом деле это будет поле класса. Если поля объявлены как `public`, они доступны за пределами класса. Например, можно определить класс `PhoneCustomer` с полями `CustomerID`, `FirstName` и `LastName`:

```
class PhoneCustomer
{
   public int CustomerID;
   public string FirstName;
   public string LastName;
}
```

После того как будет создан экземпляр объекта `PhoneCustomer`, к этим полям можно осуществлять доступ с помощью синтаксиса *`объект.имя_поля`*:

```
PhoneCustomer Customer1 = new PhoneCustomer();
Customer1.CustomerID = 1000;
Customer1.FirstName = "Иван";
Customer1.LastName = "Петров";
Console.WriteLine(Customer1.FirstName + " " + Customer1.LastName);
```

- Константы могут быть ассоциированы с классами точно так же, как и переменные. Константы, объявленные как `public`, будут доступны вне класса.
- События являются членами класса, которые позволяют объекту уведомлять вызывающего о том, что произошли некоторые программные изменения, например, изменилось поле или свойство класса либо имела место некоторая форма взаимодействия с пользователем. Клиент может содержать код, известный как обработчик ошибок, который реагирует на событие (см. главу 6).