ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ

Федеральное государственное образовательное бюджетное учреждение высшего образования
«Санкт-Петербургский государственный университет телекоммуникаций им. проф. М. А. Бонч-Бруевича»

Кафедра Автоматизации предприятий связи

Конспект лекций по дисциплине

Проектирование АСУП

(наименование дисциплины)

основная образовательная программа:

15.03.04 «Автоматизация технологических процессов и производств»
(код и наименование направления подготовки /специальности/)

профиль Алгоритмическое и программное обеспечение киберфизических систем

Санкт-Петербург

2020

- 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.

Пакет, внутри которого будет содержаться откомпилированная программа, состоит из некоторого числа сборок. Каждая сборка содержит код на промежуточном языке и метаданные, описывающие типы данных и методы внутри сборки. Метаданные содержат также: простой хэш, который строится на основе содержимого сборки и может быть использован для проверки ее целостности; информацию о версиях; сведения о том, какие сборки будут вызываться данной сборкой; и, возможно, информацию о том, какие привилегии потребуются для выполнения кода сборки.

Прежний программный продукт состоял бы из исполняемого файла, содержащего точку входа основной программы, и одной или нескольких библиотек или компонентов COM. Продукт для .NET состоит из некоторого числа сборок, одна из которых является исполняемой и содержит точку входа основной программы, а другие представляют собой библиотеки. В нашем случае имеются всего две сборки: исполняемая, содержащая код на C#, и библиотека, содержащая компилированный код VB.NET.

Исполнение

Во время выполнения программы среда исполнения .NET загружает первую сборку, ту, что содержит точку входа основной программы. Среда использует хэш для проверки целостности сборки и метаданные для того, чтобы просмотреть определенные типы и убедиться, что среда сможет выполнить сборку. Правильно разработанные коммерческие приложения должны явно указывать, какие привилегии .NET им могут потребоваться (например, понадобится ли приложению доступ к файловой системе, реестру и т.д.). В этом случае CLR обратится к политике безопасности системы и к учетной записи, под которой выполняется программа, и проверит, может ли она предоставить необходимые привилегии. Если код не запрашивает права явно, они будут предоставлены ему по первому требованию, если, конечно, это возможно.

На этом этапе CLR выполнит еще одно действие для проверки так называемой **безопасности кода по типу памяти** (memory type safety). Код считается безопасным по типу памяти только в том случае, если он обращается к памяти способами, которые может контролировать CLR. Безопасный по типу памяти код гарантированно не будет пытаться прочесть или записать в память, не принадлежащую ему. Это важно, так как .NET имеет механизм (так называемые области приложений), позволяющий нескольким приложениям выполняться в одном процессе. При этом необходимо гарантировать, что никакое приложение не будет пытаться обратиться к памяти другого приложения. Если CLR не будет уверена в том, что код безопасен по типу памяти, то в зависимости от местной политики безопасности она может даже отказать в исполнении кода.

Затем CLR выполняет код. Она создает процесс, в котором будет исполняться код, и отмечает область приложения, в которой размещается главный поток приложения. В некоторых случаях программа может потребовать поместить ее в уже имеющийся процесс запущенного ранее кода, тогда CLR создаст для нее только новую область приложения.

CLR берет первую часть кода, которая требуется для исполнения, и компилирует ее с промежуточного языка на язык ассемблера, после чего выполняет ее из соответствующего потока программы. Каждый раз, когда в процессе исполнения встречается новый метод, не исполнявшийся ранее, он компилируется в исполняемый код. Процесс компиляции происходит только один раз. Как только метод откомпилирован, его адрес заменяется адресом компилированного кода. Таким образом, производительность не ухудшается, так как компилируются только те участки кода, которые действительно используются. Этот процесс носит название компиляции just-in-time. Отметим, что JIT-компилятор может в зависимости от параметров компиляции, указанных в сборке, оптимизировать код в процессе компиляции, например, путем подстановки некоторых методов (inline) вместо их вызовов.

Во время выполнения кода CLR следит за использованием памяти. На основе этих наблюдений она в определенные моменты будет останавливать программу на короткий промежуток времени (обычно миллисекунды) и запускать сборщика мусора, который проверит переменные программы и выяснит, какие из областей памяти активно используются программой, для того, чтобы освободить неиспользуемые участки.

CLR также производит загрузку сборок по мере необходимости, в том числе COM-

указывая, что совместимый с CLS код не должен содержать двух имен, отличающихся только регистром символов. Таким образом, код VB.NET может работать с совместимым с CLS кодом.

Этот пример показывает два важных аспекта CLS. Во-первых, конкретные компиляторы не обязаны быть настолько мощными, чтобы поддерживать все особенности .NET,— это должно стимулировать разработку компиляторов с других языков программирования для .NET. Во-вторых, предоставляется гарантия того, что если класс ограничен только CLS-совместимыми функциями, он сможет использовать код, написанный на любом языке. Например, если требуется, чтобы код был CLS-совместимым, нельзя возвращать значения `UInt32`, так как этот тип не является частью CLS. Разумеется, можно возвращать и значения типа `UInt32`, однако в этом случае не гарантируется, что код будет работать со всеми языками.

Подчеркнем, что не совместимый с CLS код допустим. Однако в этом случае не гарантируется, что откомпилированный код будет полностью независим от языка.

Красота этой идеи заключается в том, что ограничение на использование CLS- совместимых особенностей действительно только для тех элементов, которые могут быть видны извне данной сборки: для открытых и защищенных членов классов и открытых классов. Внутри закрытых определений классов можно писать какой угодно не совместимый с CLS код — это не является проблемой, так как код других сборок в любом случае не сможет получить доступ к этой части кода.

CLS практически не затрагивает код на C#, поскольку C# имеет лишь несколько не совместимых с CLS особенностей. Ниже приводится ряд примеров, показывающих, какие ограничения накладывает CLS на C#:

Требование CLS	Действие на код C#
Не разрешаются глобальные методы и переменные	Не действует — C# не позволяет объявлять глобальные методы и переменные.
Запрещается использовать определенные типы данных	Для того чтобы код был CLS-совместимым, не должно быть общих или частных методов типа `sbyte`, `ushort`, `uint` и `ulong`.
Имена должны быть различимы нечувствительными к регистру языками	Для того чтобы код был CLS-совместимым, не следует применять открытые или защищенные методы, чьи имена отличаются только регистром.
Исключения (см. Ниже) должны наследоваться от базового класса Exception	Не действует — C# требует этого автоматически.
Типы-указатели не допустимы	Для того чтобы код был CLS-совместимым, не следует использовать небезопасный код и указатели нигде, кроме закрытых методов.
Переменные списки параметров не допустимы	Не действует — C# поддерживает списки параметров переменной длины, но представляет их в выходном IL-коде в виде массивов фиксированного размера (которые CLS-совместимы).

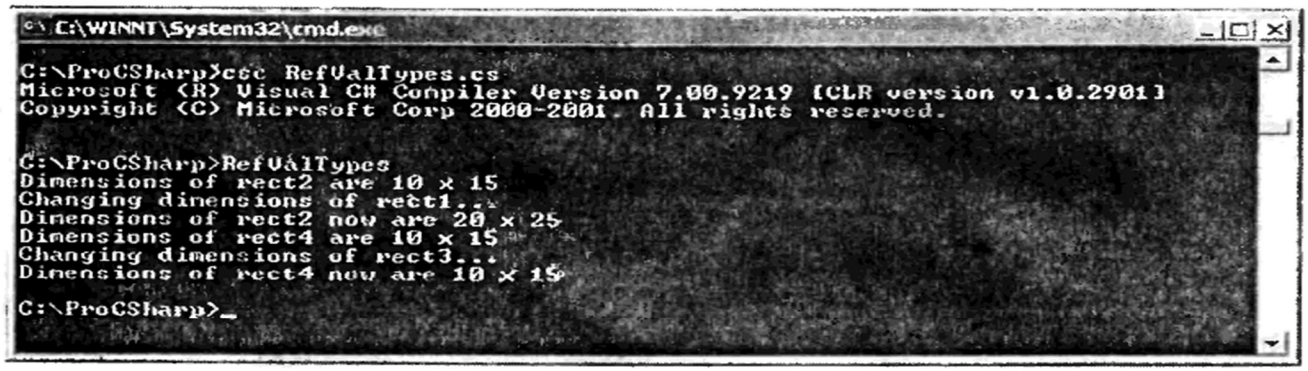
Библиотека базовых классов .NET

Вероятно, одним из самых больших достоинств управляемого кода, помимо упрощения процесса написания кода, является возможность использования библиотеки базовых классов .NET.

Базовые классы .NET представляют собой большую коллекцию классов управляемого кода. Они были созданы Microsoft и позволяют выполнять практически любые задачи, которые ранее были доступны благодаря Windows API. Эти классы следуют той же объектной модели, основанной на одиночном наследовании, что используется промежуточным языком. Это означает, что можно как создать экземпляр класса, определенного в библиотеке базовых классов .NET, так и определить класс, производный от данного.

```
    {
        RefTypeRectangle rect1 = new RefTypeRectangle();
        rect1.Width = 10;
        rect1.Height = 15;
        RefTypeRectangle rect2 = rect1;
        Console.WriteLine("Dimensions of rect2 are " + rect2.Width +
                          " x " + rect2.Height);
        Console.WriteLine("Changing dimensions of rect1...");
        rect1.Width = 20;
        rect1.Height = 25;
        Console.WriteLine("Dimensions of rect2 now are " + rect2.Width +
                          " x " + rect2.Height);
        ValTypeRectangle rect3 = new ValTypeRectangle();
        // Теперь установим высоту и ширину здесь
        rect3.Width = 10;
        rect3.Height = 15;
        ValTypeRectangle rect4 = rect3;
        Console.WriteLine("Dimensions of rect4 are " + rect4.Width +
                          " x " + rect4.Height);
        Console.WriteLine("Changing dimensions of rect3...");
        rect3.Width = 20;
        rect3.Height = 25;
        Console.WriteLine("Dimensions of rect4 now are " + rect4.Width +
                          " x " + rect4.Height);
        return 0;
    }
}
```

Результат выполнения этого кода таков:



Заметьте, как изменение полей в первоначальном классе вызывает аналогичное изменение полей у копии. Это происходит потому, что классы являются типами по ссылке — две переменные, `rect1` и `rect2`, ссылаются на одно и то же место в памяти, где хранится объект. Следовательно, изменение полей одной из переменных вызовет такое же изменение полей другой переменной. Поведение двух структур, однако, заметно отличается. Структуры являются типами по значению, поэтому когда мы объявляем вторую структуру и инициализируем ее значениями из первой структуры, это действие физически создает структуру в памяти. Изменение полей первой структуры не влияет на вторую.

Такое поведение показывает, зачем нужны две различные группы типов. При копировании примитивов, таких как `int`, или передаче их внутрь методов мы, скорее всего, пожелаем скопировать значение, а не ссылку. Более важно то, что нам не нужны потери времени на создание в куче объекта каждый раз, когда требуется целое число. Однако при работе с большими объектами эффективнее копировать ссылку на объект, а не весь объект целиком. Например, если структура передается как параметр метода, все данные структуры должны быть скопированы и размещены в куче. По этой причине необходимо осторожно обращаться со структурами и использовать их только как коллекции простых типов данных. Однако применение структур может повысить производительность, поскольку они уменьшают потери времени на создание экземпляров объектов.

Преобразование типов

Часто требуется преобразовывать данные из одного типа в другой. Рассмотрим код:

```
byte value1 = 10;
byte value2 = 23;
byte total;
total = value1 + value2;  // Когда складываются два значения byte, они
                           // неявно преобразуются в int, поэтому попытка
                           // записать результат обратно в byte
                           // вызовет ошибку
Console.WriteLine(total);
```

При компиляции этих строк будет выдано сообщение об ошибке Cannot implicitly convert type 'int' to 'byte' (Невозможно неявно преобразовать 'int' в 'byte'). Проблема здесь состоит в том, что при сложении двух значений byte результат будет иметь тип int, а не byte. Это происходит потому, что byte может содержать восемь битов данных (до 255), и при сложении двух байтов можно получить величину, которая не поместится в один байт. Если необходимо все же сохранить результат в переменной byte, нужно явно преобразовать результат в byte. Это может быть сделано двумя способами: неявно — компилятор выполнит преобразование за нас, не спрашивая подтверждения, или явно — компилятору необходимо специально указать на то, чтобы он преобразовал данные к другому типу.

Неявные преобразования

Преобразования типов могут быть выполнены автоматически, но только в том случае, если преобразуемое значение не изменяется. Следовательно, значение short или int может быть присвоено переменной long без каких-либо проблем. Переменную без знака можно присваивать переменной со знаком, пока значение переменной без знака попадает в допустимый диапазон для переменной со знаком. Кроме того, могут быть выполнены определенные преобразования целых типов в типы с плавающей точкой.

Предыдущий код нельзя было выполнить потому, что при преобразовании int в byte мы потенциально могли потерять 3 байта данных. Компилятор не позволит сделать этого, если только мы специально не укажем ему, что это именно то, чего мы добиваемся. Если мы сохраним результат в переменной long, а не byte, проблема исчезнет:

```
byte value1 = 10;
byte value2 = 23;
long total;
total = value1 + value2;
Console.WriteLine(total);
```

Тип long имеет больше битов данных, чем int, поэтому не существует риска потери данных. В этих обстоятельствах компилятор выполнит преобразование за нас, т.е. неявно.

В таблице приводятся неявные преобразования типов, допустимые в C#:

Из	В
sbyte	short, int, long, float, double, decimal
byte	short, ushort, int, uint, long, ulong, float, double, decimal
short	int, long, float, double, decimal
ushort	int, uint, long, ulong, float, double, decimal
int	long, float, double, decimal
uint	long, ulong, float, double, decimal
long, ulong	float, double, decimal
float	double
char	ushort, int, uint, long, ulong, float, double, decimal

Заметим, что выполнять неявные преобразования можно только по отношению к типам, которые преобразуются из менее крупных целых типов в более крупные или из целых без знака в целые со знаком такого же размера. Это означает, что недопустимо преобразование из uint в int или наоборот, так как эти типы охватывают разные диапазоны значений, и в процессе преобразования данные могут быть потеряны. Однако

Оператор using

Применение оператора `using` (в отличие от директивы `using`) гарантирует, что объекты, интенсивно использующие ресурсы, будут освобождены сразу по завершении работы с ними. Синтаксис этого оператора:

```
using (объект)
{
   // Код, использующий объект
}
```

Здесь *объект* — это экземпляр класса, который реализует интерфейс `IDisposable`. Все классы, реализующие `IDisposable`, *обязаны* реализовать метод `Dispose`, который служит для освобождения ресурсов, используемых объектом. Метод `Dispose` вызывается сразу по завершении блока `using`. Оператор `using` и интерфейс `IDisposable` подробно рассматриваются в главе 5.

Обработка исключений

Структурированная обработка исключений является относительно недавним нововведением в C++, и C# тоже реализует ее. С точки зрения синтаксиса, структурированная обработка исключений состоит из трех различных блоков кода. Для того чтобы реализовать в методе структурированную обработку исключений, необходимо добавить в метод каждый из этих трех блоков.

- **Блок try** инкапсулирует код, который программа пытается выполнить. Если в процессе исполнения этого кода возникает ошибка, или исключительное состояние, то говорят, что происходит исключение.
- **Блок catch** следует за блоком `try`. Он инкапсулирует код, который предпринимает действия по обработке ошибки, сгенерированной в блоке `try`.
- **Блок finally** располагается в самом конце процедуры обработки ошибок. Этот код выполняется всегда, независимо от того, успешно ли закончилось выполнение функции или было сгенерировано исключение. Из блока `finally` нельзя выйти (например, используя `goto`). Если оператор перехода приводит к передаче управления за пределы блока `try`, соответствующий блок `finally` все равно будет выполнен:

```
public void SomeRoutine()
{
   try
   {
      // Здесь начинается выполнение.
      // Этот код может сгенерировать или не сгенерировать исключение.
   }
   catch
   {
      // Если в блоке try сгенерировано исключение,
      // вызывается код, расположенный в этом блоке.
   }
   finally
   {
      // Код в данном блоке исполняется по завершении выполнения процедуры
      // независимо от того, было ли сгенерировано исключение.
   }
}
```

Ошибки генерируются автоматически, если во время исполнения происходят неверные действия, такие как деление на нуль, выход за границы диапазона или переполнение целого типа. Более того, код может явно вызвать исключение, если он определит, что возникло исключительное состояние. (Например, он может вызвать исключение, если не удается открыть важный файл или введенное значение аргумента является недопустимым.)

После обнаружения исключения блок `catch`, который обрабатывает эту ошибку, получает объект, содержащий информацию о ней. В частности, блок `catch` получает ссылку на объект, который является производным от класса `System.Exception`. Для большинства ошибок времени исполнения существует подкласс класса `Exception`. Вы можете реализовать свои подклассы для данного класса.

Для повышения надежности можно реализовать в методе несколько блоков `catch` для обработки каждого типа возможных ошибок. Эта мера позволяет коду более