Компонентное программирование в .NET

Комментарий к слайду

В данной лекции будут рассмотрены вопросы, относящиеся к истории развития, идеологии, математическому основанию и обзору возможностей компонентного проектирования и реализации программных систем – одного из важнейших и наиболее передовых подходов в современном программировании.

Содержание лекции

Комментарий к слайду

Коротко о содержании лекции.

В ходе лекции будут рассмотрены важнейшие научные исследования, относящиеся к эволюции подходов к математическому моделированию такого важнейшего из современных подходов к программированию как компонентно-ориентированного проектирования и реализации программных систем.

Прежде всего, будет представлен краткий обзор современных подходов к программированию.

При этом особое внимание будет уделено обсуждению объектно-ориентированного подхода к программированию и компонентного подхода как усовершенствованного варианта ООП.

Затем будут рассмотрены основные особенности архитектурного решения Microsoft .NET.

При этом в фокусе нашего исследования окажутся такие концепции, как сборка, манифест и пространство имен.

Наконец, предметом нашего внимания станут особенности проектирования и реализации гетерогенных программных систем согласно компонентно-ориентированному подходу.

Лекция завершится обзором литературы для более глубокого исследования материала.

Основные работы в области моделирования компонент

1924 – М.Шейнфинкель (Moses Shönfinkel) разработал простую теорию функций

1934 – А.Черч (Alonso Church) создал ламбда-исчисление и применил его в исследованиях теории множеств

1971 – Д.Скотт (Dana S. Scott) предложил использовать полные и непрерывные решетки для моделирования семантики ламбда-исчисления

80-е г.г.– Д.Скотт (Dana S. Scott) и М.Фурман (Michael P. Fourman) исследовали механизм определенных дескрипций для формализации определений

90-е г.г.– В.Э.Вольфенгаген предложил схему двухуровневой концептуализации для моделирования компонент

Комментарий к слайду

Кратко остановимся на наиболее значительных с точки зрения данного курса этапах эволюции теории и практики реализации механизмов формальных теорий и языков программирования, которые адекватны для исследования компонентно-ориентированного подхода.

Еще в 1924 году М. Шейнфинкель (Moses Shönfinkel) разработал простую (simple) теорию функций, которая фактически являлась исчислением объектов-функций и предвосхитила появление ламбда-исчисления и других теорий, моделирующих поведение объектов и события.

В 1934 г. А. Черч (Alonso Church) предложил исчисление ламбда-конверсий или ламбда-исчисление и применил его для исследования теории множеств. Вклад ученого был настолько фундаментальным, что теория (до сих пор называемая ламбда-исчислением и часто именуемая в литературе ламбда-исчислением Черча) является основополагающей и для рассматриваемых нами вопросов.

В начале 70-х г.г. Д. Скоттом (Dana S. Scott) было предложено использовать для формализации семантики математических теорий (в частности, ламбда-исчисления) так называемые решетки, которые обладают свойствами полноты и непрерывности. На этой основе Д. Скоттом был предложен так называемый денотационный подход к семантике. Такой подход предполагает анализ синтаксически корректных конструкций языка (в том числе объектов или событий) с точки зрения возможности вычисления их значений посредством специализированных функций.

Позднее, в 80-х г.г. тем же Д.Скоттом (Dana S. Scott), а также М.Фурманом (Michael P. Fourman) был исследован механизм определенных дескрипций для формализации определений.

Наконец, в 90-х г.г. В.Э.Вольфенгагеном была предложена так называемая схема двухуровневой концептуализации для моделирования поведения объектов и событий.

Подходы к программированию

Основные подходы к программированию:

- структурный, модульный;
- функциональный;
- логический;
- объектно-ориентированный (ООП);
- смешанный (комбинированный, интегрированный);
- компонентно-ориентированный (.NET);
- чисто объектный

Комментарий к слайду

Остановимся несколько подробнее на языках и подходах к программированию, которые наиболее существенны для формирования компонентно-ориентированного проектирования и реализации программных систем.

Перечислим рассмотренные в ходе курса подходы к программированию:

- ранние неструктурные подходы;
- структурный или модульный подход (задача разбивается на подзадачи, затем на алгоритмы, составляются их структурные схемы и происходит реализация);
- функциональный подход;
- логический подход;
- объектно-ориентированный подход;
- смешанный подход (некоторые подходы возможно комбинировать);
- компонентно-ориентированный (программный проект рассматривается как множество компонент, такой подход принят, в частности, в .NET);
- чисто объектный подход (идеальный с математической точки зрения вариант, который пока не реализован практически).

Заметим, что приведенную классификацию не следует считать единственно верной и абсолютной, поскольку языки программирования постоянно развиваются и совершенствуются, и недавние недостатки устраняются с появлением необходимых инструментальных средств или теоретических обоснований.

Интуитивные определения основных понятий

Объектом называется математическое представление сущности реального мира (или предметной области), которое используется для моделирования.

Классом называется весьма общая сущность, которая может быть определена как совокупность элементов.

Свойством (или атрибутом) называется пропозициональная функция, определенная на произвольном типе (данных).

Методом (или функцией) называется операция, определенная над объектами некоторого класса.

Комментарий к слайду

Сформулируем определения таких основополагающих для объектно-ориентированного подхода к программированию понятий, как объект, класс, свойство и метод.

Под объектом будем понимать математическое представление сущности реального мира (или предметной области), которое используется для моделирования.

Классом будем называть весьма общую сущность, которая может быть определена как совокупность элементов (нужно заметить, что класс при объектно-ориентированном подходе к программированию – это, как правило, первичное, неопределяемое понятие, до некоторой степени аналогичное теоретико-математическому понятию множества, или, точнее, домена).

Под свойством (или атрибутом) будем понимать пропозициональную функцию, определенную на произвольном типе (данных).

Методом (или функцией) назовем операцию, которая определена над объектами того или иного класса.

Принципы объектно-ориентированного программирования:

1. Абстракция данных

2. Наследование конкретных атрибутов объектов и функций оперирования объектами на основе иерархии

3. Инкапсуляция (свойства и методы «спрятаны» внутри объекта)

4. Полиморфизм (функции с возможностью обработки данных переменного типа)

Комментарий к слайду

Напомним, что важнейшим преимуществом объектно-ориентированного программирования является повышение производительности труда программистов при проектировании и реализации программного обеспечения. Другое преимущество ООП перед ранними подходами к программированию состоит в возрастании процента повторного использования уже разработанного программного кода.

При этом, в отличие от предыдущих подходов к программированию, объектно-ориентированный подход требует глубокого понимания основных концепций, на которых он зиждется. К числу основополагающих понятий ООП обычно относят абстракцию данных, наследование, инкапсуляцию и полиморфизм.

Преимуществом предлагаемого курса является то обстоятельство, что уже изученные в первой его части курса разделы computer science (такие как, например, ламбда-исчисление и комбинаторная логика) позволяют сформировать глубокое и точное понимание фундаментальных понятий объектно-ориентированного программирования. В частности, понятие абстракции – основной операции ламбда-исчисления – для нас является уже хорошо знакомым.

Напомним качественные основы фундаментальных принципов ООП. Наследование конкретных атрибутов объектов и функций оперирования объектами основано на иерархии. Инкапсуляция означает «сокрытие» свойств и методов внутри объекта. Полиморфизм, как и в функциональном программировании, понимается как наличие функций с возможностью обработки данных переменного типа.

Абстракция и методы ее моделирования

Вообще говоря, под ***абстракцией*** понимается выражение языка программирования, отличное от идентификатора.

Значение функции или переменной может быть присвоено абстракции и является значением последней.

Поведение абстракции заключается в приложении функции к аргументу.

Абстракция адекватно моделируется ламбда-исчислением (а именно, посредством операции абстракции).

Комментарий к слайду

Рассмотрим более подробно такой фундаментальный принцип объектно-ориентированного подхода к программированию, как абстракция.

В разделах математики, исследующих моделирование процесса создания программ, под абстракцией принято понимать произвольное выражение языка программирования, которое является отличным от идентификатора.

Важнейшей операцией, которая была исследована нами в первой части курса, является операция вычисления значения выражения или команды, т.е. операция означивания (в частности, функция вычисления значения явно используется при построении семантики языка программирования). В этой связи важно установить, что является значением абстракции. Будем считать, что значение функции или переменной может быть присвоено абстракции и является значением последней.

В объектно-ориентированном программировании каждый объект является принципиально динамической сущностью, т.е. изменяется в зависимости от времени (а также от воздействия внешних по отношению к нему факторов). Иначе говоря, объект обладает тем или иным образом поведения. В отношении абстракции как объекта, поведение заключается в приложении функции к аргументу.

Как мы уже отмечали, концепция абстракции в объектно-ориентированном программировании адекватно моделируется посредством ламбда-исчисления. Точнее говоря, операция абстракции в полной мере является моделью одноименного понятия ООП.

Наследование и методы его моделирования

Вообще говоря, под ***наследованием*** понимается свойство производного объекта сохранять поведение (атрибуты и операции) базового (родительского).

В языках программирования понятие наследование означает применимость (некоторых) свойств или методов базового класса для классов, производных от него (а также для их конкретизаций).

Наследование моделируется (иерархическим) отношением частичного порядка и адекватно формализуется посредством:

1) фреймовой нотации Руссопулоса (N.D. Roussopulos);
2) диаграмм Хассе (Hasse)

Комментарий к слайду

Под наследованием будем понимать свойство производного объекта сохранять поведение родительского объекта. Под поведением будем иметь в виду для математического объекта его атрибуты и операции над ним, а для языкового объекта ООП – поля и методы.

Таким образом, применительно к языку программирования концепция наследования означает, что свойства и методы базового класса равно применимы к его производным объектам. Заметим, что дочерний объект не обязательно наследует все без исключения атрибуты и операции родительского, а лишь некоторые из них. Такой подход характерен как для классов объектов в целом, так и для отдельных их конкретизаций, или, иначе, экземпляров.

Теоретическая концепция наследования удовлетворительно моделируется посредством отношения (или, точнее, иерархии) частичного порядка. Существует целый ряд формализаций наследования, но наиболее адекватными и концептуально ясными являются графические модели. Среди них следует выделить уже упомянутые ранее подходы: фреймовую нотацию Н.Руссопулоса и диаграммы Хассе.

Понятие инкапсуляции в программировании

Вообще говоря, под ***инкапсуляцией*** понимается доступность объекта исключительно посредством его свойств и методов.

Таким образом, свойствами объекта (явно описанными или производными) возможно оперировать исключительно посредством его методов.

Свойства инкапсуляции:

- совместное хранение данных и функций;
- сокрытие внутренней информации от пользователя;
- изоляция пользователя от особенностей реализации

Комментарий к слайду

В неформальной постановке вопроса будем понимать под инкапсуляцией доступность объекта исключительно посредством его свойств и методов.

Другими словами, концепция инкапсуляции призвана обеспечивать безопасность проектирования и реализации программного обеспечения на основе локализации манипулирования объектом в областях его полей и методов.

Иначе говоря, свойствами объекта возможно оперировать исключительно посредством его методов. Это замечание касается как свойств, явно определенных в описании объекта, так и свойств, унаследованных данным объектом от другого (других).

Практическая важность концепции инкапсуляции для современных языков объектно-ориентированного программирования (в том числе и для языка C#) определяется следующими фундаментальными свойствами.

Прежде всего, реализация концепции инкапсуляции обеспечивает совместное хранение данных (или, иначе, полей) и функций (или, иначе, методов) внутри объекта.

Как следствие, механизм инкапсуляции приводит к сокрытию информации о внутреннем «устройстве» объекта данных (или, в терминах языков ООП, свойств и методов объекта) от пользователя того или иного объектно-ориентированного приложения.

Таким образом, пользователь, получающий программное обеспечение как сервис, оказывается изолированным от особенностей среды реализации.

Понятие полиморфизма в программировании

Вообще говоря, под ***полиморфизмом*** понимается возможность оперировать объектами, не обладая точным знанием их типов.

Рассмотрим пример полиморфной функции:

```
void Poly(object o) {
  Console.WriteLine(o.ToString());
}
```

а также вариантов ее использования:

```
Poly(25);
Poly("John Smith");
Poly(3.141592536m);
Poly(new Point(12,45));
```


Комментарий к слайду

Под полиморфизмом будем иметь в виду возможность оперирования объектами без однозначной идентификации их типов.

Проиллюстрируем сходные черты и особенности реализации концепции полиморфизма при функциональном и объектно-ориентированном подходе к программированию следующим примером фрагмента программы на языке C#:

```
void Poly(object o) { Console.WriteLine(o.ToString());}
```

Как видно, приведенный пример представляет собой описание полиморфной функции `Poly`, которая выводит на устройство вывода (например, на экран) произвольный объект `o`, преобразованный к строковому формату (`o.ToString()`).

Рассмотрим ряд примеров применения функции `Poly`:

```
Poly(25);
Poly("John Smith");
Poly(3.141592536m);
Poly(new Point(12,45));
```

Заметим, что независимо от типа аргумента (в первом случае это целое число `25`, во втором – символьная строка `"John Smith"`, в третьем – вещественное число π=`3.141592536`, в четвертом – объект типа `Point`, т.е. точка на плоскости с координатами `(12,45)`), обработка происходит единообразно и, как и в случае с языком функционального программирования SML, функция генерирует корректный результат.

Компонентно-ориентированное программирование как расширение ООП

Под ***компонентом*** понимается независимый модуль для повторного использования и разворачивания.

Свойства компонента:

1) более крупная единица, чем объект (объект - это конструкция уровня языка программирования);
2) содержит множественные классы;
3) не зависит от языка программирования (в большинстве случаев).

Вообще говоря, автор и пользователь компонента географически распределены и используют разные языки.

Комментарий к слайду

Обсудив концепции ООП, которые являются вполне применимыми и для компонентно-ориентированного программирования, рассмотрим основные особенности компонентного подхода.

Во-первых, следует отметить то обстоятельство, что компонентно-ориентированный подход к проектированию и реализации программных систем и комплексов является в некотором смысле развитием объектно-ориентированного, и прежде всего практически более пригоден для разработки крупных и распределенных систем (например, корпоративных приложений).

Прежде всего, сформулируем основополагающее для рассматриваемого подхода определение компонента. Под компонентом будем далее иметь в виду независимый модуль программного кода, предназначенный для повторного использования и разворачивания. Как видно из определения, применение компонентного программирования призвано обеспечить более простую, быструю и прямолинейную процедуру первоначальной инсталляции прикладного программного обеспечения, а также увеличить процент повторного использования кода, т.е. усилить основные преимущества ООП.

Говоря о свойствах компонентов, следует прежде всего отметить, что это существенно более крупные единицы, чем объекты (в том смысле, что объект является конструкцией уровня языка программирования). Другими отличиями компонентов от традиционных объектов является возможность содержать множественные классы и (в большинстве случаев) независимость от языка программирования.

Заметим, что, автор и пользователь компонента, вообще говоря, территориально распределены и используют разные языки. Вполне возможно, что они не только пишут программы, но и общаются на разных языках.

Модели, поддерживающие компонентное программирование

1. ***Component Object Model*** (COM) – изначальный стандарт Microsoft для компонент.
   - Определяет протокол для конкретизации и использования компонент внутри процесса, между процессами или между компьютерами.
   - Основа для ActiveX, OLE и многих других технологий.
   - Может создаваться в Visual Basic, C++, .NET, и др.
2. ***Java Beans*** – стандарт Sun Microsystems для компонентов (не является независимым от языка)
3. ***CORBA*** (громоздкий IDL-интерфейс, сложность отображения одного языка реализации в другой)

Комментарий к слайду

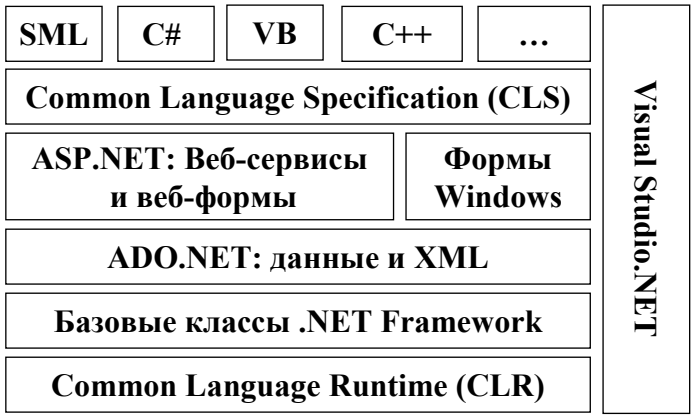
Получив представление о компонентах и их отличиях от традиционных объектов ООП, рассмотрим существующие подходы к моделированию вариаций компонентного подхода в современной практике проектирования и реализации программных комплексов и систем.

Прежде всего, поскольку основной вычислительной средой для исследования в данном курсе является Microsoft .NET, обсудим особенности модели для компонентно-ориентированной разработки программного обеспечения, предложенной корпорацией Microsoft. Эта модель называется компонентной объектной моделью (или Component Object Model, COM) и является изначальным стандартом, принятым для компонентной разработки приложений в корпорации Microsoft.

Компонентная модель COM определяет протокол для конкретизации (т.е. создания экземпляров) и использования компонент (по аналогии с классами и объектами) как внутри одного и того же процесса, так и между различными процессами или компьютерами, предназначенными для выполнения того или иного программного проекта, основанного на компонентной технологии. Модель COM является достаточно универсальной и используется в качестве фундамента для таких технологий проектирования и реализации программного обеспечения, как ActiveX, OLE и целого ряда других технологий. Приложения для COM-модели могут создаваться средствами таких языков и сред разработки как Visual Basic, C++, .NET и др.

Другим известным стандартом для компонентной модели является стандарт компании Sun Microsystems, известный как JavaBeans, который не обладает свойством независимости от языка программирования. Еще одним широко используемыми стандартом компонентного программирования является архитектура объектных запросов CORBA (несмотря на поддержку многоязычной разработки приложений, существуют сложности отображения одного языка реализации в другой, для этой цели применяется достаточно громоздкий интерфейс, основанный на специальном языке описания IDL).

Архитектурная схема .NET Framework и Visual Studio.NET



Комментарий к слайду

Кратко напомним основные аспекты архитектурного решения Microsoft .NET Framework, отметив прежде всего то обстоятельство, что важную роль играет среда разработки Microsoft Visual Studio.NET, а первостепенное значение отводится среде выполнения программ – Common Language Runtime (CLR).Среда выполнения программ CLR реализует управление памятью, типами данных, межъязыковым взаимодействием, разворачиванием (deployment) приложений.

Существенным преимуществом конструктивного решения .NET является компонентно-ориентированный подход к проектированию и реализации программного обеспечения. Суть подхода состоит в принципиальной возможности создания независимых составляющих программного обеспечения с унифицированной интерфейсной частью для многократного повторного и распределенного использования. При этом продуктивность решения обусловлена многоязычностью интегрируемых программных проектов (концепция .NET потенциально поддерживает произвольный язык программирования, в числе наиболее известных языков – C#, Visual Basic, C++ и др.)

В ходе компиляции программа на .NET-совместимом языке программирования трансформируется в соответствии с заранее заданной обобщенной спецификацией языка Common Type System (CTS). Система типов CTS полностью описывает все типы данных, поддерживаемые средой выполнения, определяет их взаимосвязи и хранит их отображения в систему типов .NET.

Под Common Language Specification (или CLS) понимается набор правил, определяющих подмножество обобщенных типов данных, в отношении которых гарантируется, что они безопасны при использовании во всех языках .NET.

Интерфейсы реализуются посредством форм Windows и ASP.NET для веб-приложений.

Сборки Common Language Runtime в .NET

Вообще говоря, под ***сборкой*** понимается логическая единица разворачивания приложения, содержащая манифест, метаданные, MSIL и необходимые ресурсы.

Под ***манифестом*** понимается совокупность метаданных о компонентах сборки (версия, типы, зависимости и т.д.).

Метаданные типов исчерпывающе описывают все типы, определенные в сборке: свойства, методы, аргументы, возвращаемые значения, атрибуты, базовые классы и т.д.

Под ***Microsoft Intermediate Language*** (или MSIL, IL) понимается семейство языков, которые компилируются в IL (или управляемый код).

Замечание. IL всегда компилируется в native-код до выполнения.

Комментарий к слайду

Обсудим средства поддержки жизненного цикла программного обеспечения в рамках компонентной идеологии подхода .NET.

Для установки на компьютеры пользователей ранее созданного прикладного программного обеспечения создаются инсталляционные комплекты в форме так называемых сборок.

Сборкой называется логическая единица, содержащая множество модулей, необходимых для осуществления инсталляции программного обеспечения. Сборка характеризуется уникальностью, которая обеспечивается идентификатором версии сборки и цифровой подписью автора. Сборка является самодостаточной единицей для установки программного обеспечения и не требует никаких дополнений. Возможно как индивидуальное, так и коллективное (сетевое) использование сборки на основе компонентной технологии. Сборка обеспечивает простой и удобный механизм инсталляции и экономит средства на разворачивание программного обеспечения, сводя к минимуму затраты времени и труда на установку.

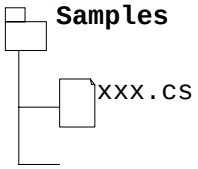
Описание сборки содержится в так называемом манифесте, где хранятся метаданные о компонентах сборки, идентификация автора и версии, сведения о типах и зависимостях, а также режим и политика использования. Метаданные типов манифеста исчерпывающе описывают все типы, определенные в сборке, а именно, свойства, методы, аргументы, возвращаемые значения, атрибуты, базовые классы и т.д.

Семейство языков, которые компилируются в так называемый промежуточный язык (Intermediate Language или IL, т.е. в так называемый управляемый код) объединяется в Microsoft Intermediate Language или MSIL.

Заметим, что промежуточный язык IL всегда компилируется в естественный (native) код до выполнения программы.

Пространства имен в языке C# (1)

Файл может содержать множество пространств имен:




```
xxx.cs
namespace A {...}
namespace B {...}
namespace C {...}
```

Пространства имен и классы не однозначно соответствуют каталогам и файлам:

```
xxx.cs
namespace A {
    class C {...}
}
```


Комментарий к слайду

Для более эффективного манипулирования системой типизации компонент создаваемого программного обеспечения в рамках модели COM, концепция .NET предусматривает механизм пространств имен (`namespace`).

Пространством имен будем называть механизм среды Microsoft .NET, предназначенный для идентификации типов объектов языков программирования и среды реализации.

Описания пространств имен по аналогии с описаниями типов данных размещаются в файлах. Перечислим основные свойства, которыми характеризуются пространства имен в среде Microsoft .NET. Прежде всего, пространства имен могут как объединять различные сборки, так и быть вложенными друг в друга. Кроме того, файлы с описаниями могут содержать множественные пространства имен. Важно отметить, что между пространствами имен и файлами не существует однозначного соответствия. Наконец, полное имя типа должно содержать все необходимые пространства имен.

Приведем пример файла `xxx.cs` с описанием множественных пространств имен на языке C#:

```
xxx.cs
namespace A {...}
namespace B {...}
namespace C {...}
```

Приведем пример файла `xxx.cs` с описанием пространств имен на языке C# с неоднозначным (с учетом предыдущего примера) соответствием файлов и пространств имен:

```
xxx.cs
namespace A {
     class C {...}
}
```

Пространства имен в языке C# (2)

Пространства имен могут импортироваться:

```
using System;
```

Одни пространства имен могут импортироваться в другие:

```
using A;
namespace B {
    using C;
    ...
}
```

Разрешаются псевдонимы:

```
using F = System.Windows.Forms;
...
F.Button b;
```

для явного указания полных и сокращенных имен.

Комментарий к слайду

Кроме свойств, перечисленных выше, механизм пространств имен в среде вычислений .NET обладает еще целым рядом важных особенностей.

Так, допускается импорт пространств имен с использованием зарезервированного слова `using` языка программирования C# (похожий подход реализован в модулях языка Modula-2). Проиллюстрируем эту особенность фрагментом программы на языке C#:

```
using System;
```

Кроме того, существует возможность импорта одних пространств имен в другие. Проиллюстрируем это свойство фрагментом программы на языке C#:

```
using A;
namespace B {
     using C;
     ...
}
```

Отметим также, что для явного указания полных и сокращенных имен разрешаются псевдонимы (alias). Проиллюстрируем это свойство следующим примером программы на языке C#:

```
using F = System.Windows.Forms;
...
F.Button b;
```

Как видно из приведенного фрагмента программы, для удобства именования стандартного пространства имен .NET под именем `System.Windows.Forms` применяется псевдоним `F`.

Сборки в языке программирования C# (1)

Под ***сборкой*** понимается самодостаточная исполняемая единица с информацией для развертывания и уникальным номером версии, содержащая типы и другие ресурсы.

Сборка является атомарной единицей для развертывания.

Каждый тип сборки имеет уникальный номер версии.

Сборка может содержать несколько пространств имен; пространство имен может занимать несколько сборок.

Сборка может состоять из нескольких файлов, объединенных в составе манифеста (аналога оглавления).

Комментарий к слайду

Рассмотрим особенности использования механизма сборок, важнейшей концепции компонентного программирования, применительно к языку C#.

Напомним, что под сборкой понимается самодостаточная исполняемая единица, содержащая все необходимые данные для инсталляции, развертывания и повторного использования.

Сборка является минимальной единицей для развертывания приложений, т.е. представляет собой своеобразный атом компонентного программирования.

Каждый тип сборки характеризуется уникальным идентификатором – номером версии сборки. Таким образом, каждый программный проект формируется в виде сборки, которая является самодостаточным компонентом для разворачивания, тиражирования и повторного использования. Сборка идентифицируется цифровой подписью автора и уникальным номером версии.

Между сборками и пространствами имен существует следующее соотношение. Сборка может содержать несколько пространств имен. В то же время, пространство имен может занимать несколько сборок.

Сборка может иметь в своем составе как один, так и из несколько файлов, которые объединяются в составе так называемого манифеста или описания сборки, который на привычном нам естественном языке аналогичен оглавлению книги. Манифест содержит метаданные о компонентах сборки, идентификацию автора и версии, сведения о типах и зависимостях, а также режим и политику использования сборки. Метаданные типов манифеста в полной мере описывают все типы, которые описаны в сборке.

Сборки в языке программирования C# (2)

Сборка в языке программирования C# является аналогом компонента в среде программирования .NET.

При компиляции создается либо ***сборка*** (исполняемый EXE-файл и файл DLL-библиотеки с манифестом) либо ***модуль*** (файл .NETMODULE без манифеста).

Прочие модули и ресурсы могут быть добавлены посредством компоновщика сборок.

Комментарий к слайду

В ходе обсуждения реализации механизма сборок в языке программирования C# следует отметить еще несколько важных характеристик.

Прежде всего, сборка как элемент языка программирования C# является аналогом компонента в среде проектирования и реализации программного обеспечения Microsoft .NET.

В результате компиляции программного кода на языке C# в среде вычислений .NET (например, в .NET Framework SDK) создается либо сборка, либо так называемый модуль.

При этом сборка существует в форме исполняемого файла (с расширением EXE), а также файла динамически присоединяемой библиотеки (с расширением DLL). Естественно, в состав сборки входит манифест. Модуль представляет собой файл с расширением .NETMODULE и, в отличие от сборки, не содержит в своем составе манифеста.

Заметим, что интеграция в программный проект других модулей и ресурсов (в частности, типов и метаданных) может быть осуществлена посредством системного программного обеспечения, известного под названием компоновщика сборок.

Преимущества компонентного программирования

1. Снижение стоимости программного обеспечения
2. Повторное использование кода
3. Унификация обработки объектов различной природы
4. Менее человекозависимый процесс создания программного обеспечения
5. Строгое математическое основание (ламбда-исчисление)
6. Концепция универсальна и одинаково применима для функционального программирования и ООП

Комментарий к слайду

Подводя итоги рассмотрения основных аспектов концепции компонентно-ориентированного подхода к программированию и особенностей реализации этой концепции применительно к языку программирования C#, кратко отметим достоинства подхода.

Прежде всего, важным практическим следствием реализации концепции компонентного подхода для экономики программирования является снижение стоимости проектирования и реализации программного обеспечения.

Еще одним существенным достоинством компонентного программирования является возможность усовершенствования стратегии повторного использования кода. Код с более высоким уровнем абстракции не требует существенной модификации при адаптации к изменившимся условиям задачи или новым типам данных.

Кроме того, к преимуществам концепции компонентного программирования следует отнести унификацию обработки объектов различной природы. В самом деле, абстрактные классы и методы позволяют единообразно оперировать гетерогенными данными, причем для адаптации к новым классам и типам данных не требуется реализации дополнительного программного кода.
Важно также отметить, что идеология компонентного программирования основана на строгом математическом фундаменте (в частности, в виде формальной системы ламбда-исчисления), что обеспечивает интуитивную прозрачность исходного текста для математически мыслящего программиста, а также верифицируемость программного кода.

Наконец, концепция компонентного программирования является достаточно универсальной и в равной степени применима для различных подходов к программированию, включая функциональный и объектно-ориентированный.

Вопросы, рассмотренные в рамках курса

1) объектно-ориентированный подход к программированию;
2) основные понятия ООП (объекты, классы, методы, абстракция, инкапсуляция, наследование, полиморфизм);
3) математические основы ООП (ламбда-исчисление, фреймы, решетки, коцептуализация);
4) синтаксис и семантика языков ООП (на примере C#);
5) теория типов и система типизации в .NET;
6) событийно-ориентированное программирование;
7) применение концепции .NET для реализации ООП;
8) компонентный подход к программированию

Комментарий к слайду

Завершая вторую часть учебного курса, посвященного исследованию современных языков программирования (на примере языка программирования C#) и поддерживающих их сред вычислений (на примере инструментально-технологической платформы .NET), кратко резюмируем содержание рассмотренных вопросов и проблем.

Прежде всего, нами был рассмотрен объектно-ориентированный подход к проектированию и реализации программного обеспечения в сопоставлении с другими подходами.

Затем было дано представление о важнейших концепциях, которые составляют теоретическое и практическое основание объектно-ориентированного подхода к программированию. В частности, были рассмотрены концепции объекта, класса, метода, абстракции, инкапсуляции, наследования, полиморфизма, а также подходы к их формализации на основе исчисления ламбда-конверсий, комбинаторной логики, теории решеток и концептуализации.

Далее, посредством перечисленных теорий были формализованы такие важнейшие аспекты объектно-ориентированных языков программирования, как синтаксис и семантика.

Кроме того, было исследовано понятие типа, изучены основы теории типов и типизации в языках программирования, реализованных в среде вычислений .NET.

После этого, мы перешли к рассмотрению вопросов, связанных с событийно-ориентированным программированием.

Наконец, было исследована трансформация ООП в приложении к среде .NET, которая привела к появлению компонентного подхода, основанного на объектной модели COM.

Вопросы для дальнейшего исследования (в рамках двухсеместрового расширенного курса):

1) компонентная разработка интегрированных гетерогенных программных систем на профессиональном уровне (на основе SML и C#);
2) разработка событийно-управляемых приложений;
3) строгий сравнительный анализ функционального и объектно-ориентированного подходов к программированию (на основе computer science);
4) семантика событийно-управляемого (а, возможно, и компонентно-ориентированного) программирования

Комментарий к слайду

Вполне естественно, что, исходя из обширного спектра и существенной глубины рассматриваемой проблематики, ряд важнейших аспектов реализации современных языков программирования (в рамках изложенного курса) был лишь обозначен или изложен весьма конспективно.

В связи с этим, в ходе дальнейшего исследования планируется систематическое изучение следующих вопросов:

1) компонентная разработка интегрированных гетерогенных программных систем на профессиональном уровне (на примере языков программирования SML и C#);

2) разработка событийно-управляемых приложений;

3) математически строгий сравнительный анализ функционального и объектно-ориентированного подходов к программированию на основе изученных теоретических разделов computer science;

4) формализация семантики событийно-управляемого (и, в случае достаточных ресурсов, компонентно-ориентированного) программирования.

Дальнейшие исследования, согласно концепции изложения курса, будут проводиться синтетически по направлениям теоретического обоснования программирования на основе изученных формальных систем computer science и современной практики проектирования и реализации программного обеспечения на основе универсальной и прогрессивной программно-инструментальной платформы Microsoft .NET.

Библиография (1)

1. Lowy J. Programming .NET Components. O'Reilly, 2003, 480 p.p.
2. Lowy J. COM and .NET Component Services. O'Reilly, 2001, 384 p.p.
3. Thai T.L., Lam H. .NET Framework Essentials, 2nd ed. O'Reilly, 2002, 376 p.p.
4. Schönfinkel M. 'Über die Bausteine der matematischen Logik, Math. Annalen 92, pp. 305-316, 1924. Translation printed as 'On the building blocks of mathematical logic', in van Heijenoort, J. (ed.), From Frege to Gödel, Harvard University Press, 1967

Комментарий к слайду

К сожалению, в рамках времени, отведенных на одну лекцию, можно лишь в общих чертах охарактеризовать специфику такого многоаспектного явления, как компонентно-ориентированный подход к проектированию и реализации программного обеспечения. Для более детального ознакомления с особенностями, достижениями и проблемами в теории моделирования данного подхода и практики реализации связанных с ним механизмов рекомендуется следующий список литературы:

1.Lowy J. Programming .NET Components. O'Reilly, 2003, 480 p.p.
2.Lowy J. COM and .NET Component Services. O'Reilly, 2001, 384 p.p.
3.Thai T.L., Lam H. .NET Framework Essentials, 2nd ed. O'Reilly, 2002, 376 p.p.
4.Schönfinkel M. 'Über die Bausteine der matematischen Logik, Math. Annalen 92, pp. 305-316, 1924. Translation printed as 'On the building blocks of mathematical logic', in van Heijenoort, J. (ed.), From Frege to Gödel, Harvard University Press, 1967

Кратко остановимся на источниках.

Работа [1] посвящена проектированию и реализации компонент для среды программирования .NET.

В работе [2] рассматриваются вопросы проектирования сервисов для .NET в рамках компонентной идеологии.

В работе [3] описываются основные особенности архитектурного решения .NET Framework.

Работа [4] является фундаментальным исследованием в области синтаксиса и семантики ламбда-исчисления, основной формализации языков и подходов к программированию, не исключая и компонентный.

Библиография (2)

5. Church A. The calculi of lambda-conversion.- Princeton, 1941, ed. 2, 1951
6. Scott D.S. The lattice of flow diagrams.- Lecture Notes in Mathematics, 188, Symposium on Mathematics of Algorithmic Languages.- Springer-Verlag, 1971, p.p. 311-372
7. Wolfengagen V.E. Event-driven objects. In: Proc. CSIT'1999, Moscow, Russia, 1999, Vol.1, p.p. 88-96
8. Wolfengagen V.E. Fuctional notation for indexed concepts. In: Proc. WFLP'2000, Benicassim, Spain, Sept. 2000. http://www.dsic.upv.es/~wflp2000/
9. Scott D.S. Domains for denotational semantics. ICALP 1982, 577-613

Комментарий к слайду

Продолжим обсуждение работ, посвященных исследованию основных элементов событийно управляемого программирования.

5.Church A. The calculi of lambda-conversion.- Princeton, 1941, ed. 2, 1951
6.Scott D.S. The lattice of flow diagrams.- Lecture Notes in Mathematics, 188, Symposium on Mathematics of Algorithmic Languages.- Springer-Verlag, 1971, p.p. 311-372
7.Wolfengagen V.E. Event-driven objects. In: Proc. CSIT'1999, Moscow, Russia, 1999, Vol.1, p.p. 88-96
8.Wolfengagen V.E. Fuctional notation for indexed concepts. In: Proc. WFLP'2000, Benicassim, Spain, Sept. 2000. http://www.dsic.upv.es/~wflp2000/
9.Scott D.S. Domains for denotational semantics. ICALP 1982, 577-613

Работа [5] является исчерпывающим описанием синтаксиса и семантики ламбда-исчисления, основной формализации языков программирования, в том числе языков компонентного программирования.

В работе [6] представлен вариант подхода к семантике динамики и статики объектов в форме теории решеток, по которым «протекает» информация.

В работах [7,8] исследуется событийная ориентированность объектных теорий с учетом динамики и статики объектов, а также их взаимовлияния.

В работе [9] рассматриваются вопросы, связанные с развитием денотационной семантики, имеющей широкое распространение в теории моделирования объектов, и, в частности, с ее представлением посредством доменов.