Событийно управляемое программирование в .NET

Комментарий к слайду

В данной лекции будут рассмотрены вопросы, относящиеся к истории развития, идеологии, математическому основанию и обзору возможностей событийно управляемого проектирования и реализации программных систем – одного из важнейших аспектов современного объектно-ориентированного программирования.

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 для объектно-ориентированного подхода к программированию аспекта как управление событиями.

Прежде всего, будет представлено сопоставление понятий о событии в математической теории и практике программирования.

Затем будут рассмотрены наиболее продуктивные (с точки зрения целей данного курса) методы моделирования событий.

При этом будут подробно исследованы особенности формализации событий посредством функций и фреймов.

Существенное внимание будет уделено реализации так называемого механизма делегатов в языке программирования C# (который практически применяется для поддержки событий), причем будут рассмотрены примеры использования делегатов как особого вида событий.

Фрагменты программ на языке C# проиллюстрируют практику обработки событий. При этом особая роль будет отведена исследованию механизмов обработки исключительных ситуаций, возникающих в ходе анализа параметров событий.

Лекция завершится обзором литературы для более глубокого исследования материала.

Основные работы в области моделирования событий

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 событий

Комментарий к слайду

Кратко остановимся на наиболее значительных (с точки зрения данного курса) этапах эволюции теории и практики реализации механизмов управления событиями в формальных теориях и языках программирования.

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 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.

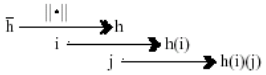
Позднее, в 80-х г.г. , тем же Д.Скоттом (Dana S. Scott), а также М.Фурманом (Michael P. Fourman) был исследован механизм определенных дескрипций для формализации определений. В ходе изложения мы будем неоднократно использовать эту лаконичную и интуитивно прозрачную нотацию.

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

Понятие события в математике

Вообще говоря, под ***событием*** понимается соотнесение над объектом предметной области (или, иначе, индивидом).

Произвольное семейство (действительных) объектов может быть параметризовано (или, иначе, концептуализировано) не только типами, но и событиями.



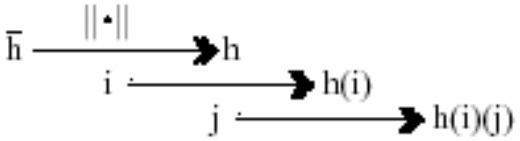
Оценивающее отображение ||•|| переводит индивид $\overline{h}$ языка (программирования) в h. Возможный индивид h∈H переводится событием i ∈ Asg в действительный индивид h(i) ∈ U_i. Следующий шаг, управляемый событием j ∈ Asg, переводит h(i) в состояние h(i)(j).

Комментарий к слайду

Завершив краткий обзор теоретических концепций моделирования событий, рассмотрим подробнее формализацию понятия события с точки зрения последнего из предложенных подходов.

Под событием в математическом смысле далее будем иметь в виду соотнесение над объектом предметной области, который в рамках терминологии курса будем называть индивидом. Неформально говоря, под индивидом понимается такой объект предметной области (или языка программирования), который возможно выделить в этой области (или языке) посредством указания так называемой индивидуализирующей функции. Построение такой функции будем считать зависимым от эксперта в предметной области. Обычно такая функция имеет в качестве области своих значения истинности (а именно, «истина» и «ложь») и является истинной при аппликации к данному объекту и ложной в противном случае.

Ранее нами было рассмотрено понятие типа как совокупности объектов. Заметим, что произвольное семейство (действительных в нашем частном случае) объектов может быть параметризовано (или, иначе, концептуализировано) не только типами, но и событиями.



В соответствии со схемой двухуровневой концептуализации, оценивающее отображение ||·|| переводит индивид $\overline{h}$ языка (в частности, языка программирования) в h. Затем возможный (потенциальный) индивид h из семейства возможных индивидов H переводится событием i из семейства соотнесений Asg в действительный индивид h(i) из семейства действительных индивидов U_i. Аналогично, следующий шаг преобразований, управляемый событием j из семейства Asg, переводит h(i) в состояние h(i)(j).

Понятие события в программировании

Под ***событием*** понимается способ внедрения фрагмента в код с целью изменения поведения программы.

Как только происходит нечто, что интересует программиста или пользователя, активизируется событие и выполняется соответствующий фрагмент кода.

В целом, обработка события подобна вызову процедуры.

Любой интерфейс пользователя построен на основе обработки событий (`onClick`,`onMouseMove`,`onMouseOver` и т.д.).

События, взаимодействующие с сетью, операционной системой, сторонними приложениями и т.д. могут также активизироваться по времени.

Комментарий к слайду

На первый взгляд, определение понятия события в программировании существенно отличается от одноименной математической концепции.

Под событием в языке программирования обычно понимается способ внедрения того или фрагмента в программный код с целью изменения поведения программы.

Как только происходит изменение среды вычислений из числа представляющих интерес для разработчика или пользователя программного обеспечения, активизируется событие и выполняется соответствующий фрагмент кода.

В целом, с точки зрения практического программирования, обработка события подобна вызову процедуры, причем в качестве параметров выступают те или иные характеристики среды вычислений.

Любой интерфейс пользователя (или, в математической терминологии, среда вычислений) построен на основе обработки событий (`onClick`,`onMouseMove`,`onMouseOver` и т.д.). События, которые осуществляют взаимодействие с каналами локальных сетей, операционной системой, сторонними приложениями и т.д. могут также активизироваться по времени.

В соответствии со схемой двухуровневой концептуализации, первый уровень может означать, например, инициацию пользователем события «щелчок левой кнопкой мыши», а второй – изменение состояния объекта меню при выборе соответствующего пункта. Как видим, сначала возможный индивид становится действительным (т.е. происходит активация события), а затем осуществляется означивание объекта программы (изменяется текущая позиция меню). Фрагментом кода программы в таком случае является метод, изменяющий текущую позицию меню, который активируется, исходя из значения первого соотнесения (т.е. конкретизации события). Таким образом, на основе механизма событий осуществляется управление программой.

Использование делегатов для обработки событий в языке C# (1)

Под ***делегатом*** понимается метод-тип.

Описание типа-делегата

```
delegate void Notifier (string sender);
                    // обычное описание метода
                    // с ключевым словом delegate
```

Описание переменной-делегата

```
Notifier greetings;
```


Комментарий к слайду

После изложения понятийного аппарата концепции событийно-управляемого программирования перейдем к вопросу реализации данного механизма применительно к языку объектно-ориентированного программирования C#.

В целях реализации механизма событий в языке программирования C# предусмотрен так называемый механизм делегатов.

Заметим, что механизм делегатов в языке C# является существенным усовершенствованием ранних подходов сходной функциональности в языках программирования C и C++, известных под названием функции обратного вызова (callback function), основанных на использовании малоинформативных и потенциально небезопасных указателей в оперативной памяти.

Преимущество делегатов языка C# состоит в большей управляемости и безопасности кода (в частности, в C# существует возможность контроля соответствия типов параметров делегатов и логики вызова).

В качестве интуитивного определения понятия делегата отметим, что под делегатом понимается метод, который ведет себя как тип.

В качестве иллюстрации приведем описание типа- и переменной-делегатов на языке C#:

```
delegate void Notifier (string sender);
// обычное описание метода с ключевым словом delegate


Notifier greetings;
```

```
// описание переменной-делегата
```


Использование делегатов для обработки событий в языке C# (2)

Присваивание метода переменной-делегату:

```
void SayHello(string sender) {
  Console.WriteLine("Hello from " + sender);
}
greetings = new Notifier(SayHello);
```

Вызов переменной-делегата:

```
greetings("John");  // активирует SayHello("John") =>
"Hello from John"
```


Комментарий к слайду

Как видно из приведенного фрагмента программы, формальное описание типов- и переменных-делегатов не отличается от традиционного описания методов и переменных.

Фундаментальное отличие переменной-делегата от ранее рассмотренных объектов языка C# состоит в том, что в качестве значения переменной-делегату можно присвоить метод. Проиллюстрируем это утверждение следующим фрагментом программы на языке C#:

```
void SayHello(string sender) {
          Console.WriteLine("Hello from " + sender);
}
     greetings = new Notifier(SayHello);
```

Как видно из приведенного примера, на основе ранее описанного типа-делегата `Notifier` в соотнесении с вновь описанным методом `SayHello` осуществляется присваивание значения переменной `greetings`. Фактически данный пример изображает первый шаг концептуализации.

Проиллюстрируем далее порядок вызова переменной-делегата следующим фрагментом программы на языке C#:

```
greetings("John");
// активирует SayHello("John") => "Hello from John"
```

Как видно из приведенного примера, означивание переменной-делегата `greetings` активирует соотнесенный с ней на предыдущем шаге метод `SayHello` в соотнесении

"John" с генерацией состояния "Hello from John". Фактически данный пример изображает второй шаг концептуализации.

Современные языки программирования и .NET: II семестр
Лекция 11: Событийно управляемое программирование в .NET

Управление событиями с помощью переменных-делегатов языка C# (1)

Переменной-делегату может быть присвоен любой из назначенных методов :

```
void SayGoodBye(string sender) {
  Console.WriteLine("Good bye from " + sender);
}

greetings = new Notifier(SayGoodBye);

greetings("John");        //SayGoodBye("John") =>
"Good bye from John"
```


Комментарий к слайду

Отметим далее, что в предыдущих фрагментах программ рассматривался лишь один вариант соотнесения, характеризующего второй шаг концептуализации.

Для более наглядной иллюстрации работы механизма делегатов в языке программирования C#, приведем пример фрагмента программы, характеризующего еще одно из соотнесений семейства для второго шага концептуализации:

```
void SayGoodBye(string sender) {
          Console.WriteLine("Good bye from " + sender);
}

greetings = new Notifier(SayGoodBye);

greetings("John"); //SayGoodBye("John") =>  "Good bye from John"
```

Как видно из приведенного примера, переменная-делегат greetings типа Notifier может принимать в качестве значения любой из методов SayHello и SayGoodBye.

В данном случае, в отличие от предыдущего примера, значением, вычисляемым в ходе выполнения метода SayGoodBye, соотнесенного с переменной-делегатом greetings, является конкретизация "Good bye from John", полученная на основе означивания SayGoodBye("John").

Таким образом, становится ясным, каким образом посредством активизации (в зависимости от значения переменной-делегата) того или иного метода, становится

возможным управлять поведением программы в зависимости от событий, происходящих в среде вычислений.

Управление событиями с помощью переменных-делегатов языка C# (2)

Перечислим следующие особенности переменных-делегатов:

1. Переменная-делегат может иметь пустое значение null (метод не назначен).

2. Пустая переменная-делегат не может быть вызвана (возникает исключительная ситуация).

3. Переменные-делегаты являются объектами первого рода (first class object): их можно хранить в структурах данных, передавать как параметры и т.д.

Комментарий к слайду

Для более подробного рассмотрения механизмов расширенного управления программой на языке C# посредством событий на основе делегатов, необходимо предварительно исследовать основные особенности переменных данного типа.

Прежде всего, необходимо отметить то обстоятельство, что переменные-делегаты могут иметь пустое значение null, что соответствует отсутствию назначенного им метода. Здесь проявляется аналогия с теорией вычислений Д.Скотта в том отношении, что любой домен непременно имеет неопределенное значение. Эта параллель наводит на предположение, что теория вычислений Д.Скотта способна адекватно формализовать событийно управляемое программирование. Оказывается, что это предположение весьма важно, а его исследование – весьма продуктивно.

Кроме того, пустые переменные-делегаты в языке C# не подлежат вызову. При попытке осуществить обращение к такой переменной в среде программирования возникает исключительная ситуация. И снова здесь прослеживается аналогия с теорией вычислений Д.Скотта.

Еще одной особенностью переменных-делегатов является их принадлежность к классу объектов первого рода (first class object). Согласно правилам языка программирования C#, делегаты возможно хранить в структурах данных, передавать как параметры и т.д. Таким образом, делегаты как модели событий во многом сходны с функциями в математическом понимании этого слова. Следовательно, для формализации механизмов, основанных на событиях, вполне пригодны уже хорошо знакомые нам формальные системы ламбда-исчисления и комбинаторной логики.

Делегаты как объекты языка C# (1)

Проиллюстрируем использование переменных-делегатов на следующих примерах.

Создадим переменную-делегат:

```
new DelegateType (obj.Method);
```

Переменная-делегат хранит как сам метод, так и его приемник (receiver), при этом сама не имеет параметров:

```
new Notifier(myObj.SayHello);
```

Объект `obj` сможет быть описан как `this`, а также опущен:

```
new Notifier(SayHello);
```


Комментарий к слайду

Обсудив основные возможности использования описания и использования механизма делегатов в языке программирования C#, рассмотрим свойства делегатов как объектов языка более подробно. При этом будем приводить необходимые примеры в виде кода на языке программирования C#.

Прежде всего, рассмотрим порядок описания языкового объекта-делегата. Проиллюстрируем обобщение создания переменной-делегата в виде формы Бэкуса-Наура (БНФ):

```
<описание_переменной-делегата>::=
     new <тип_делегата> (<объект>.<метод>);
```

При этом переменная-делегат в ходе соотнесения инициализируется конкретным методом явно указанного объекта в соответствии с типом делегата. Необходимо также отметить, что в структуре переменной-делегата хранится не только сам метод, но и его выход или приемник (receiver). Тем не менее, переменная-делегат не имеет собственных параметров:

```
new Notifier(myObj.SayHello);
```

В данном фрагменте программного кода на языке C# приведен конкретный пример описания переменной-делегата как конкретизации типа-делегата `Notifier` в соотнесении с уже известным нам методом `SayHello` определенного пользователем объекта `myObj`. Заметим, что присутствующий в описании объект `obj` сможет быть определен явно посредством описателя `this`, а может быть и опущен, как, например, в следующем фрагменте программы на языке C#:

```
new Notifier(SayHello);
```

Делегаты как объекты языка C# (2)

В случае статического метода вместо объекта указывается имя класса:

```
new Notifier(MyClass.StaticSayHello);
```

Метод **не** описывается как абстрактный, но может быть описан посредством `virtual`, `override` или `new`.

Описание метода должно соответствовать описанию типа делегата в следующих отношениях:

- одинаковое количество параметров;
- одинаковые типы параметров (включая тип возвращаемого значения);
- одинаковые виды параметров (`ref, out, value`).

Комментарий к слайду

Обсудив основные возможности описания и использования механизма делегатов с динамическими методами, рассмотрим особенности статического случая.

При этом обобщенный формат описания переменной-делегата для языка программирования C# в виде формы Бэкуса-Наура (БНФ) примет вид:

```
<описание_переменной-делегата>::=
     new <тип_делегата> (<класс>.<метод>)
```

Таким образом, описание переменной-делегата со статическим методом в форме кода на языке программирования C# может иметь, например, следующий вид:

```
new Notifier(MyClass.StaticSayHello);
```

Существует еще ряд особенностей, характеризующих статический случай использования переменных-делегатов в языке программирования C#. Перечислим наиболее существенные из них.

Прежде всего, следует отметить, что метод в составе делегата не может быть описан как абстрактный (`abstract`), но может быть определен с использованием описателей как виртуальный (`virtual`), замещенный (`override`) или как экземпляр (`new`).

Кроме того, описание метода должно соответствовать описанию типа делегата, т.е. оба описания должны иметь одинаковое количество параметров и типы параметров (включая тип возвращаемого значения), причем виды параметров (в частности, с передачей по ссылке (`ref`), по значению (`value`), а также возвращаемых (`out`)) должны совпадать.

Использование делегатов множественного вызова для управлениями событиями в языке C#

Переменная-делегат может содержать множественные значения в одно и то же время:

```
Notifier greetings;
greetings = new Notifier(SayHello);
greetings += new Notifier(SayGoodbye);

greetings("John");  // "Hello from John"
        // "Good bye from John"

greetings -= new Notifier(SayHello);

greetings("John");  // "Good bye from John"
```

Заметим, что если множественный делегат является функцией, то возвращаются как значение, так и параметр последнего вызова.

Комментарий к слайду

Еще одним важным свойством переменных-делегатов является их динамическая природа. Теоретически это означает, что в зависимости от (временного) соотнесения переменная-делегат пробегает по домену значений конкретизаций связанного с ней метода.

Практика программирования на языке C# показывает, что переменная-делегат может содержать множественные значения в одно и то же время. Проиллюстрируем это утверждение следующим фрагментом программы на языке C#:

```
Notifier greetings;
greetings = new Notifier(SayHello);
greetings += new Notifier(SayGoodbye);
greetings("John");
          // "Hello from John"
          // "Good bye from John"
greetings -= new Notifier(SayHello);
greetings("John");
          // "Good bye from John"
```

Как видно из приведенного примера, фрагмент программы на языке C# содержит описание уже известной нам переменной-делегата `greetings` типа-делегата `Notifier`. Переменной-делегату `greetings` в качестве значения последовательно инкрементно присваиваются конкретизации-методы `SayHello` и `SayGoodbye`. При этом отладочный вывод значения переменной `greetings` с конкретизацией `"John"` демонстрирует семейство значений `"Hello from John"` и `"Good bye from John"`. По декрементном присваивании переменной-делегату `greetings` конкретизации-метода `SayHello`, отладочный вывод значения переменной демонстрирует значение `"Good bye from John"`.

Заметим, что если множественный делегат является функцией, то возвращаются как значение, так и параметр последнего вызова.

События как особый вид переменных-делегатов (1)

Проиллюстрируем управление событиями следующим расширенным примером использования делегатов:

```
class Model {
  public event Notifier notifyViews;
  public void Change() { ... notifyViews("Model"); }
}

class View1 {
  public View1(Model m) { m.notifyViews += new
Notifier(this.Update1); }
  void Update1(string sender) {
Console.WriteLine(sender + " was changed"); }
}
```


Комментарий к слайду

Обсудив наиболее существенные для данного курса аспекты типов- и переменных-делегатов, перейдем к рассмотрению порядка использования механизма делегатов для создания событийно управляемых программ.

Проиллюстрируем подход к управлению событиями посредством механизма делегатов следующим фрагментом программного кода на языке программирования C# (приводим начало фрагмента):

```
class Model {
     public event Notifier notifyViews;
     public void Change() {
          ... notifyViews("Model"); }
     }
class View1 {
     public View1(Model m) {
          m.notifyViews += new Notifier(this.Update1);
     }
void Update1(string sender){
     Console.WriteLine(sender + "was changed"); }
}
```

Как видно из приведенного примера, фрагмент программы на языке C# содержит описание класса `Model` с полем-событием `notifyViews` (описатель `event`) и методом `Change`, оповещающем через делегат о смене соотнесения. Кроме того, в данном фрагменте содержатся описания класса `View1` с одноименными методом для просмотра

состояния делегата посредством метода `Update1`, содержащего вывод на стандартное устройство отладочной информации о смене приложения-«отправителя» сообщения.

События как особый вид переменных-делегатов (2)

```
  class View2 {
  public View2(Model m) { m.notifyViews += new
Notifier(this.Update2); }
  void Update2(string sender) {
Console.WriteLine(sender + " was changed"); }
}

class Test {
  static void Main() {
        Model m = new Model(); new View1(m); new
View2(m);
        m.Change();
  }
}
```

Заметим, что события используются вместо обычных переменных-делегатов для улучшения абстракции, поскольку событие может активировать только тот класс, в котором оно описано.

Комментарий к слайду

Завершим иллюстрацию использования делегатов для событийно управляемого программирования :

```
class View2 {
     public View2(Model m){
          m.notifyViews += new Notifier(this.Update2);
     }
     void Update2(string sender){
          Console.WriteLine(sender + " was changed");
     }
}

class Test {
     static void Main() {
          Model m = new Model();
          new View1(m);
          new View2(m);
          m.Change();
     }
}
```

Как видно из приведенного фрагмента программы, пример завершается описанием класса `View2`, аналогичного классу `View1`, а также класса `Test`, который инициализирует классы `View1` и `View2` и запускает метод `Change`, инициирующий смену соотнесений (и состояний) переменных-делегатов.

Заметим, что события используются вместо обычных переменных-делегатов для увеличения уровня абстракции программной компоненты, поскольку событие может активировать только тот класс, в котором оно описано.

Обработка исключений в языке C#: оператор try (1)

```
FileStream s = null;
try {
  s = new FileStream(curName, FileMode.Open);
  ...
} catch (FileNotFoundException e) {
  Console.WriteLine("file {0} not found", e.FileName);
} catch (IOException) {
  Console.WriteLine("some IO exception occurred");
} catch {
  Console.WriteLine("some unknown error occurred");
} finally {
  if (s != null) s.Close();
}
```


Комментарий к слайду

Процесс обработки событий, как и процесс проектирования и реализации программного обеспечения в целом, не может быть свободным от ошибок. Ранее в ходе изложения уже упоминалось о таком важном свойстве доменов как наличие неопределенного элемента. Кроме того, при обсуждении семантики рассматривалась модель генерации ошибочной ситуации при невозможности связывания переменной со значением. Подобные ситуации часто возникают и при обработке событий. Рассмотрим особенности реализации оператора `try` языка C# для обнаружения и обработки ошибок на следующем примере:

```
FileStream s = null;
     try {s = new FileStream(curName, FileMode.Open);
          ...
     }
     catch (FileNotFoundException e){
          Console.WriteLine("file {0} not found", e.FileName);
     }
     catch (IOException){
          Console.WriteLine("some IO exception occurred");
     }
     catch {
          Console.WriteLine("some unknown error occurred");
     }
     finally {if (s != null) s.Close(); }
```

Как видно из примера, оператор `try` анализирует выход функции чтения из потокового файла. Для обработки вариантов при разборе возможных ошибочных ситуаций

используется оператор `catch` , в данном примере генерирующий диагностические сообщения. В случае ложности всех `catch`-альтернатив выполняется блок операторов, следующий за словом `finally` (происходит штатное закрытие файла).

Обработка исключений в языке C#: оператор try (2)

Перечислим основные свойства оператора `try`:

1) условия обнаружения (`catch`) проверяются последовательно;

2) в итоге, одно из условий всегда удовлетворяется (если список условий не пуст);

3) имя параметра `exception` в условии обнаружения можно опустить;

4) тип `exception` должен быть выводим из `System.Exception;`

5) в случае отсутствия параметра `exception` , подразумевается `System.Exception`

Комментарий к слайду

По результатам анализа приведенного фрагмента программы на языке C# рассмотрим основные характеристики обработки исключительных ситуаций с учетом особенностей среды вычислений Microsoft .NET.

Прежде всего, необходимо отметить то обстоятельство, что условия обнаружения `catch` в составе оператора `try` проверяются последовательно сверху вниз.

Таким образом, при полном разборе случаев разработчиком программного обеспечения, одно из условий обнаружения `catch` всегда удовлетворяется (естественно, в том случае, если список условий не пуст).

Кроме того, синтаксис языка программирования C# разрешает опустить имя параметра `exception` в условии обнаружения `catch`.

Далее, заметим, что тип конкретизации исключительной ситуации `exception` должен быть выводим из базового класса системы типизации Microsoft .NET под названием `System.Exception`.

Наконец, важным свойством оператора является тот факт, что при отсутствии явного указания параметра `exception` подразумевается обработка событий в соответствии с умолчаниями, принятыми для полей и методов базового класса `System.Exception` системы типизации Microsoft .NET.

Описание класса System.Exception

Свойства:

`e.Message` сообщение об ошибке в виде строки; установка нового `Exception(msg)`;

`e.StackTrace` просмотр стека вызова методов как строки;

`e.Source` приложение или объект, которые вызвали исключительную ситуацию;

`e.TargetSite` метод объекта, который вызвал исключительную ситуацию;

`...`

Методы:

`e.ToString()` возвращает имя исключительной ситуации;

`...`

Комментарий к слайду

Для иллюстрации особенностей механизма обработки исключительных ситуаций в среде вычислений Microsoft .NET приведем фрагмент содержательного описания системного базового класса `System.Exception` системы типизации .NET, оформив его для наглядности в форме следующей таблицы:

Таблица . Некоторые свойства и методы класса `System.Exception`.

	Имя свойства или метода	Функции свойства или метода
Свойства:		
	`e.Message`	сообщение об ошибке в виде строки;
		установка нового сообщение об ошибке `Exception(msg);`
	`e.StackTrace`	просмотр стека вызова методов как строки;
	`e.Source`	приложение или объект, которые вызвали исключительную ситуацию;
	`e.TargetSite`	метод объекта, который вызвал исключительную ситуацию;
	`...`	
Методы:		
	`e.ToString()`	возвращает имя исключительной ситуации;
	`...`	

Как видно из приведенной таблицы, класс `System.Exception` представляет собой описание многочисленных объектов и методов, которые контролируют стандартный ход обработки исключительных ситуаций, возникающих в программных реализациях (в том числе и разработанных на языке C#). При этом каждая исключительная ситуация характеризуется уникальным диагностическим сообщением.

Активация исключительной ситуации в языке C#

1. Посредством недопустимой операции (неявно):
   1) деление на нуль;
   2) выход за границы массива;
   3) обращение к пустому (`null`) указателю;
   . . .
2. Посредством оператора `throw` (явно):

```
throw new FunnyException(10);

class FunnyException : ApplicationException {
  public int errorCode;
  public FunnyException(int x) { errorCode = x; }
}
```


Комментарий к слайду

При исследовании механизмов обработки исключительных ситуаций в той или иной системе, среде или языке программирования, неизбежно возникает вопрос об источниках таких ситуаций.

Исходя из вида изученного оператора `try` языка программирования C#, а также из стандартных свойств и методов обработки исключений класса `System.Exception` среды вычислений .NET, можно сделать предположение о том, что существуют явные и неявные источники возникновения исключительных ситуаций.

К неявным источникам исключений будем относить недопустимые операции, возникающие во время выполнения программы (будем считать, что компилятор работает корректно). В частности, недопустимыми операциями будем считать хорошо знакомые нам ситуации деления на нуль, выхода за границы массива, обращения к пустому (`null`) указателю и т.д.

Однако, для реализации событийно управляемых программ в языке программирования C# существует механизм явной генерации исключений, который реализуется посредством оператора `throw` (приводим соответствующий фрагмент программы):

```
throw new FunnyException(10);
class FunnyException : ApplicationException{
     public int errorCode;
     public FunnyException(int x){
           errorCode = x;
     }
```

```
}
```

Как видно из приведенного фрагмента программы, оператор `throw` реализует явный вызов экземпляра класса `FunnyException` с генерацией кода ошибки.

Фрагмент иерархии исключений языка C# (1)

Exception
SystemException
  ArithmeticException
    DivideByZeroException
    OverflowException
    ...
  NullReferenceException
  IndexOutOfRangeException
  InvalidCastException
  ...

Комментарий к слайду

Заметим, что в силу высокой сложности и гетерогенной природы среды вычислений Microsoft .NET, семейство исключительных ситуаций весьма многообразно.

Для удобства систематизации, хранения, поиска и анализа исключений, а также в силу особенностей строения системы типизации Microsoft .NET, классификация исключительных ситуаций в ней организована по иерархическому принципу.

Приведем для сведения небольшой фрагмент сложной структуры иерархии стандартных исключительных ситуаций среды вычислений Microsoft .NET:

```
Exception
     SystemException
          ArithmeticException
               DivideByZeroException
               OverflowException
               ...
          NullReferenceException
          IndexOutOfRangeException
          InvalidCastException
          ...
```

Заметим, что существует еще более общий уровень иерархии исключений, чем `SystemException`, а именно, уровень `Exception`.

В данной иерархии исключительных ситуаций перечислены наиболее типичные неявные исключения: деление на нуль и переполнение как варианты арифметического исключения.

Другим подклассом исключений являются обращение к пустой ссылке, выход за границы массива, а также неверный вызов.

Фрагмент иерархии исключений языка C# (2)

```
ApplicationException
  ... //специализированные исключения
  ...
IOException
  FileNotFoundException
  DirectoryNotFoundException
  ...
WebException
...
```


Комментарий к слайду

Продолжим анализ иерархии исключительных ситуаций среды вычислений .NET.

```
ApplicationException
          ... //специализированные исключения
          ...
     IOException
          FileNotFoundException
          DirectoryNotFoundException
          ...
     WebException
     ...
```

Отдельный подкласс исключений составляют исключения, встречающиеся в приложениях.

Еще одним важным подклассом исключительных ситуаций являются ошибки ввода-вывода, связанные, в частности, с невозможностью найти файл или каталог.

Наконец, еще одним подклассом исключительных ситуаций, существенным для среды вычислений Microsoft .NET, ориентированной на распределенные сетевые вычисления и веб-сервисы, является подкласс веб-исключений.

Алгоритм поиска условия обнаружения:

Цепочка вызовов просматривается в обратном направлении до тех пор, пока не находится метод с соответствующим условием обнаружения. Если таковой не найден, программа аварийно завершается с дампом стека.

Исключительные ситуации не должны с обязательностью обнаруживаться в языке C#.
Не существует различия между:
1) помеченными исключениями, которые необходимо обнаружить, и
2) непомеченными исключениями, которые нет необходимости обнаружить.

Замечание. Хотя такой подход представляется удобным, он приводит к созданию менее устойчивых приложений.

Комментарий к слайду

Рассмотрим особенности реализации механизма реакции на исключительные ситуации в языке программирования C# и среде Microsoft .NET.

Поиск условия обнаружения исключительной ситуации в языке программирования C# реализован на основе следующего обобщенного алгоритма.

Осуществляется просмотр цепочки вызовов методов в обратном направлении до тех пор, пока не найден метод с соответствующим условием обнаружения `catch`. В случае, если такой метод не удается обнаружить, программа завершается аварийно и выдается дамп стека (оперативной памяти).

Подчеркнем, что алгоритм поиска исключительных ситуаций не обязательно должен результативно завершиться даже для такого современного и позволяющего создавать относительно безопасный код языка программирования как C#.

Заметим, что в языке C# не существует различия между помеченными исключениями, которые необходимо обнаружить, и непомеченными исключениями, в обнаружении которых нет необходимости.

Несмотря на то, что такой подход в первом приближении представляется удобным, он заведомо приводит к созданию менее устойчивого и безопасного прикладного (и тем более системного) программного кода.

Преимущества событийно-ориентированного программирования

1. Возможность моделирования произвольных реальных объектов
2. Потенциальная легкость настройки интерфейса
3. Программирование, основанное на сценариях (скрипты изменяют состояние программы)
4. Высокий процент повторного использования программного кода
5. Гибкость реинжиниринга программного обеспечения
6. Строгое математическое основание (концептуализация)

Комментарий к слайду

Подводя итоги рассмотрения основных аспектов теории и практики событийно управляемого программирования на языке C#, можно сделать следующие выводы.

Во-первых, как и в объектно-ориентированном программировании, управление событиями (а, другими словами, поведением объектов программ) открывает возможность моделирования реальных объектов сколь угодно сложной структуры и природы.

Во-вторых, событийно ориентированное программирование обеспечивает потенциальную легкость настройки интерфейса пользователя (и даже его адаптации в процессе работы программы в соответствии с требованиями пользователя).

Кроме того, процедуры обработки событий представляют собой методы в терминологии ООП. Таким образом, основным предметом программирования являются сценарии, причем методы делегатов, реализованные в форме скриптов, изменяют состояние программы.

Затем, важным преимуществом программирования, ориентированного на события или скрипты, является высокий процент повторного использования программного кода. Заметим, что значительное количество событий уже реализовано в среде вычислений .NET, и нами был рассмотрен фрагмент их иерархии.

Далее, необходимо отметить такой позитивный аспект событийного программирования как гибкость реинжиниринга (т.е. разработки в обратном направлении) программного обеспечения. По сути речь идет о концептуализации, которая является теоретической основой наших рассуждений.

Именно концептуализация является тем строгим математическим основанием, которое позволяет не только моделировать управление событиями, но и обеспечить верифицируемость создаваемого программного обеспечения.

Библиография (1)

1. 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
2. Church A. The calculi of lambda-conversion.- Princeton, 1941, ed. 2, 1951
3. Barendregt H.P. The lambda calculus (revised edition), Studies in Logic, 103, North Holland, Amsterdam, 1984
4. 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

Комментарий к слайду

К сожалению, в рамках времени, отведенных на одну лекцию, можно лишь в общих чертах охарактеризовать специфику такого многоаспектного явления, как событийно-управляемый 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.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
2.Church A. The calculi of lambda-conversion.- Princeton, 1941, ed. 2, 1951
3.Barendregt H.P. The lambda calculus (revised edition), Studies in Logic, 103, North Holland, Amsterdam, 1984
4.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

Кратко остановимся на источниках. Работы [1-3] являются исчерпывающим описанием синтаксиса и семантики ламбда-исчисления, основной формализации языков программирования, в том числе языков ООП. В работе [4] представлен вариант подхода к семантике динамики и статики объектов в форме теории решеток, по которым «протекает» информация.

Библиография (2)

5. Scott D.S. Identity and existence in intuitionistic logic.- In: Application of Sheaves.- Berlin: Springer, 1979, p.p. 600-696
6. Fourman M. The logic of topoi. In: Handbook of Mathematical Logic, J.Barwise et al., eds. North-Holland, 1977
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.Scott D.S. Identity and existence in intuitionistic logic.- In: Application of Sheaves.- Berlin: Springer, 1979, p.p. 600-696
6.Fourman M. The logic of topoi. In: Handbook of Mathematical Logic, J.Barwise et al., eds. North-Holland, 1977
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] представляет собой систематизацию важнейших понятий – тождества и существования – в логиках высших порядков и затрагивает вопросы принципиальной формализуемости реального мира.

В работе [6] обсуждаются способы построения математически и логически корректных определений понятий предметной области, т.е. строится своего рода объектная модель.

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.