ЛАБОРАТОРНАЯ РАБОТА №4
ОБРАБОТКА ИСКЛЮЧЕНИЙ

Цель работы: приобрести навыки использования механизма исключений для сигнализации об ошибочных ситуациях и их обработки.

Задачи:

3. Научиться использовать систему обработки исключений в C#.
4. Доработать предметную модель из предыдущей лабораторной работы для обработки ошибок.

Результатами работы являются:

- Библиотека классов, моделирующая предметную область согласно варианту задания
- Набор модульных тестов для демонстрации возможностей разработанной библиотеки классов
- Подготовленный отчет

КРАТКАЯ ХАРАКТЕРИСТИКА ОБЪЕКТА ИЗУЧЕНИЯ, ИССЛЕДОВАНИЯ

Операторы try-catch и исключения

Оператор try указывает блок кода, предназначенный для обработки ошибок или очистки. За блоком try должен следовать блок catch, блок finally или оба. Блок catch выполняется, когда возникает ошибка в блоке try. Блок finally выполняется после выполнения блока try (или блока catch, если он предусмотрен), обеспечивая очистку независимо от того, возникала ошибка или нет.

Блок catch имеет доступ к объекту Exception, который содержит информацию об ошибке. Блок catch применяется либо для компенсации последствий ошибки, либо для повторений генерации исключения. Исключение генерируется повторно, если нужно просто зарегистрировать факт возникновения проблемы в журнале или если необходимо сгенерировать исключение нового типа более высокого уровня.

Блок finally добавляет детерминизма к программе: среда CLR стремится выполнять его всегда. Он полезен для проведения задач очистки вроде закрытия сетевых подключений. Оператор try выглядит следующим образом:

```
try
{
 …// Во время выполнения этого блока может
   // возникнуть исключение
}
 catch (ExceptionA ex)
{
... // Обработать исключение типа ExceptionA
}
catch (ExceptionB ex)
{
 ... // Обработать исключение типа ExceptionB
}
```

```
finally
{
. . . / / Код очистки
}
```

Взглянем на код:

```
class Test
{
      static int Calc (int x) => 10 / x;
      static void Main()
      {
            int y= Calc (0);
            Console.WriteLine (y);
      }
}
```

Поскольку x имеет нулевое значение, исполняющая среда генерирует исключение DivideByZeroException и программа завершается. Чтобы предотвратить такое поведение, перехватим исключение следующим образом:

```
class Test
{
   static int Calc (int x) => 10 / x;
   static void Main()
   {
         try
         {
               int y= Calc (0);
               Console.WriteLine (y);
         }
         catch (DivideByZeroException ex)
         {
               Console.WriteLine("x cannot be zero");
               // значение x не может быть равно 0
         }
```

```
            Console.WriteLine ("program completed");
            // программа завершена
    }
}
```

ВЫВОД: x cannot be zero program completed

Этот простой пример предназначен только для иллюстрации обработки исключений. На практике вместо реализации такого сценария лучше явно проверять делитель на равенство нулю перед вызовом Calc. Проверка с целью предотвращения ошибок предпочтительнее реализации блоков try/catch, т.к. обработка исключений является относительно дорогостоящей в плане ресурсов, требуя немало процессорного времени.

Когда возникает исключение, среда CLR выполняет следующую проверку.

Находится ли поток выполнения в текущий момент внутри оператора try, который может перехватит исключение? Если да, то поток выполнения переходит к совместимому блоку catch. Если этот блок catch завершился успешно, поток выполнения перемещается на оператор, следующий после try (сначала выполнив блок finally, если он присутствует). Если нет, то поток выполнения возвращается обратно в вызывающий компонент и проверка повторяется (после выполнения любых блоков finally, внутри которых находится оператор).

Если ни одна из функций в стеке вызовов не взяла на себя ответственность за исключение, то пользователю отображается диалоговое окно с сообщением об ошибке и программа завершается.

Конструкция catch

Конструкция catch указывает тип исключения, подлежащего перехвату. Типом может быть либо класс System.Exception, либо какой-то подкласс System.Exception.

Указание типа System.Exception приводит к перехвату всех возможных ошибок. Это удобно в следующих ситуациях:

- программа потенциально может восстановиться независимо от конкретного типа исключения;

– планируется повторная генерация исключения (возможно, после его регистрации в журнале);
– обработчик ошибок является последним средством перед тем, как программа будет завершена.

Однако более обычной является ситуация, когда перехватываются исключения специфических типов, чтобы не иметь дела с исключениями, для которых обработчик не был предназначен (например, OutOfMemoryException).

Перехватывать исключения нескольких типов можно с помощью множества конструкций catch (опять-таки, данный пример проще реализовать с помощью явной проверки аргументов, а не за счет обработки исключений):

```
class Test
{
      static void Main (string[] args)
      {
      try
      {
            byte b = byte.Parse (args[0]);
            Console.WriteLine (b);
      }
      catch (IndexOutOfRangeException ex)
      {
            // Должен быть предоставлен хотя бы один
         аргумент
            Console.WriteLine ("Please provide at
         least one argurnent");
      }
      catch (FormatException ex)
      {
            // Аргумент должен быть числовым
            Console. WriteLine ( "That' s not a
         number ! ");
      }
```

```
        catch (OverflowException ex)
        {
              // Возникло переполнение
              Console.WriteLine ("You've given те
            more than a byte!");
            }
    }
}
```

Для заданного исключения выполняется только одна конструкция catch. Если вы хотите предусмотреть "страховочную сетку" для перехвата более общих исключений (наподобие System.Exception), то должны размещать более специфические обработчики первыми.

Исключение может быть перехвачено без указания переменной, если доступ к свойствам исключения не нужен:

```
catch (OverflowException) // переменная не указана
{
    …
}
```

Более того, можно опустить и переменную, и тип (это значит, что будут перехватываться все исключения):

```
 catch { ... }
```

Фильтры исключений

Начиная с версии C# 6.0, в конструкции catch можно указывать фильтр исключений, добавляя конструкцию when:

```
catch (WebException ex} when
    (ex.Status == WebExceptionStatus.Timeout)
{
…
}
```

Если в этом примере генерируется исключение WebException, то будет вычислено булевское выражение, находящееся после ключевого слова when. Если результатом является false, то данный блок catch игнорируется и принимаются во внимание любые последующие конструкции catch. Благодаря фильтрам исключений может появиться смысл в повторном перехвате исключения того же самого типа:

```
catch (WebException ex) when
    (ex.Status == WebExceptionStatus.Timeout)
{ ... }
catch (WebException ex) when
    (ex.Status == WebExceptionStatus.SendFailure)
{ ... }
```

Булевское выражение в конструкции when может иметь побочные эффекты, например, вызывать метод, который фиксирует в журнале сведения об исключении в целях диагностики.

Блок finally

Блок finally выполняется всегда - независимо от того, возникало ли исключение, и полностью ли был выполнен блок try. Блоки finally обычно используются для размещения кода очистки.

Блок finally выполняется в любом из следующих случаев:

- после завершения блока catch;
- после того, как поток управления покидает блок try из-за наличия оператора перехода (например, return или goto);
- после завершения блока try.

Единственное, что может воспрепятствовать выполнению блока finally - это бесконечный цикл или неожиданное завершение процесса.

Блок finally содействует повышению детерминизма программы. В приведенном далее примере открываемый файл всегда закрывается независимо от перечисленных обстоятельств:

- блок try завершается нормально;

- происходит преждевременный возврат из-за того, что файл пуст (EndOfStream);
- во время чтения файла возникает исключение IOException.

```
static void ReadFile()
{
      StreamReader reader = null;
      // Из пространства имен System.IO
      try
      {
            reader = File.OpenText ("file.txt");
            if (reader.EndOfStream) return;
            Console.WriteLine
          (reader.ReadToEnd());
      }
      Finally
      {
          if (reader != null) reader.Dispose();
      }
}
```

В этом примере мы закрываем файл с помощью вызова Dispose на StreamReader. Вызов Dispose на объекте внутри блока finally - это стандартное соглашение, соблюдаемое повсеместно в .NET Framework, и оно явно поддерживается в языке C# посредством оператора using.

Оператор using

Многие классы инкапсулируют неуправляемые ресурсы, такие как файловые и графические дескрипторы или подключения к базе данных. Такие классы реализуют интерфейс System.IDisposabe, в котором определен единственный метод без параметров по имени Dispose, предназначенный для очистки этих ресурсов. Оператор using предлагает элегантный синтаксис для вызова Dispose на объекте IDisposable внутри блока finally.

Показанный ниже код:

```
usinq (StreamReader reader =
   File.Opentext ("file.txt"))
{…}
```

в точности эквивалентен следующему коду:

```
{
      StreamReader reader =
         File.OpenText ("file.txt");
      try
      {…}
      finally { if (reader != null)
      ((IDisposaЬe)reader) .Dispose();
      }
}
```

Генерация исключений

Исключения могут генерироваться либо исполняющей средой, либо пользовательским кодом. В следующем примере метод Display генерирует исключение System.ArgumentNullException:

```
class Test
{
      static void Display (string name)
      {
            if (name == null)
            throw new ArgumentNullException
               (nameof (name));
            Console.WriteLine (name);
      }
      static void Main ()
            { try { Display (null); }
            catch (ArgumentNullException ex)
            {
               Console.WriteLine
                     ("Caught the exception");
               // Исключение перехвачено
            }
      }
}
```

Основные свойства класса System.Exception

Ниже описаны наиболее важные свойства класса System. Exception.

- StackTrace – строка, представляющая все методы, которые были вызваны, начиная с источника исключения и заканчивая блоком catch.
- Message – строка с описанием ошибки.
- InnerException – внутреннее исключение (если есть), которое привело к генерации внешнего исключения. Это свойство само может иметь другое свойство InnerException.

Общие типы исключений

Перечисленные ниже типы исключений широко используются в CLR и .NET Framework. Их можно генерировать самостоятельно или применять в качестве базовых классов для порождения специальных типов исключений.

- *System.ArgumentException.* Генерируется, когда функция вызывается с некорректным аргументом. Как правило, это указывает на наличие ошибки в программе.
- *System.ArgumentNullException.* Подкласс ArgumentException, который генерируется, когда аргумент функции (неожиданно) равен null.
- *System.ArgwnentOutOfRangeException.* Подкласс ArgumentException, который генерируется, когда (обычно числовой) аргумент имеет слишком большое или слишком малое значение. Например, это исключение возникает при передаче отрицательного числа в функцию, принимающую только положительные значения.
- *System.InvalidOperationException.* Генерируется, когда состояние объекта оказывается неподходящим для успешного выполнения метода, независимо от любых заданных значений аргументов. В качестве примеров можно назвать чтение файла, который не был открыт, или получение следующего элемента из перечислителя, когда лежащий в основе список был изменен на середине выполнения итерации.

- *System.NotSupportedException.* Генерируется для указания на то, что конкретная функциональность не поддерживается. Хорошим примером может служить вызов метода Add на коллекции, для которой IsReadOnly возвращает true.
- *System.NotimplementedException.* Генерируется для указания на то, что функция пока еще не реализована.
- *System.ObjectDisposedException.* Генерируется, когда объект, на котором вызывается функция, был освобожден.

Еще одним часто встречающимся типом исключения является NullReferenceException. Среда CLR генерирует это исключение, когда вы пытаетесь получить до· ступ к члену объекта, значение которого равно null (что указывает на ошибку в коде). Исключение NullReferenceException можно генерировать напрямую (в тестовых целях) следующим образом:

```
throw null;
```

ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РБОТУ

Доработать предметную модель согласно выданному варианту задания.

ТРЕБОВАНИЯ К РЕАЛИЗАЦИИ

При выполнении лабораторной работы необходимо использовать стандарт кодирования языка C#. Для разработанных классов создать набор модульных тестов, проверяющих корректность кода, а также демонстрирующих полученный API.

ВАРИАНТЫ ЗАДАНИЙ

Вариант № 1. «Служба доставки»

Заказ может находиться в одном из следующих состояний: «формируется», «в обработке», «доставляется», «доставлен» (рис. 4.1).

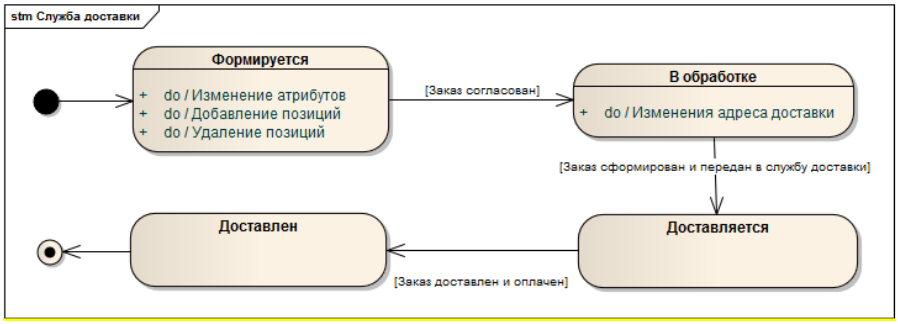

Рис. 4.1. Диаграмма состояний заказа
(приложение «Служба доставки»)

В состоянии «формируется» заказ оказывается сразу после создания, в это время в него могут добавляться/удаляться позиции, меняться атрибуты «Срочная доставка», «Адрес доставки».

После окончательного согласования с клиентом заказа, он переводится в состояние «в обработке». В это время заказ

комплектуется и готовится к доставке. У заказа, находящегося «в обработке», может изменяться только атрибут «Адрес доставки».

После того, как заказ сформирован и передан в службу доставки, он переводится в состояние «доставляется». Начиная с этого момента атрибуты и состав заказа редактироваться не могут.

После передачи клиенту и оплаты, заказ переходит в состояние «доставлен».

В рамках данной лабораторной работы необходимо внести следующие изменения в модель, разработанную в ходе выполнения лабораторных работ №№ 2-3:

- добавить в сущность «Заказ» атрибуты, содержащие текущий статус заказа, дату/время подтверждения заказа клиентом, дату/время передачи в службу доставки и дату/время доставки заказа клиенту;
- реализовать операции перехода заказа по состояниям.

Кроме того, необходимо для всех операций модели реализовать проверку предусловий:

- корректности переданных параметров;
- допустимости выполнения операции в текущем состоянии объекта.

Например, в качестве заказчика для заказа или товара для позиции заказа не может быть задано значение «null», в качестве количества товара в позиции заказа или стоимости товара не могут указываться отрицательные величины, заказ в состоянии «доставляется» нельзя перевести в состояние «формируется», у заказа в состоянии «формируется» нельзя удалить позицию и т. п.

Приложение не должно аварийно завершаться в случае ввода пользователем некорректных значений даты, чисел и т. п., а должно адекватно реагировать на ошибку. Например, выводить понятное сообщение и предлагать повторить ввод. (Указание: Ввод некорректных данных пользователем – это ситуация нормальная, а значит приводить к генерации исключений она не должна. Поэтому при реализации этого требования не следует ждать исключения и обрабатывать его в операторе try/catch, а заранее проверять корректность вводимых значений и на основе этой проверки действовать. В данной ситуации можно использовать методы TryParse)

Вариант № 2. «Управление задачами»

Задача может находиться в одном из следующих состояний: «утверждается», «реализуется», «проверяется», «закрыта» (рис. 4.2).

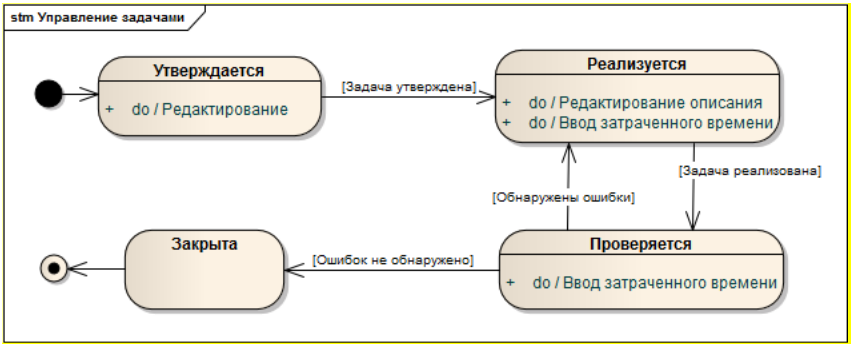

Рис. 4.2. Диаграмма состояний задачи
(приложение «Управление задачами»)

После создания и до полного согласования с заказчиком задача находится в состоянии «утверждается». У задачи в этом состоянии могут редактироваться все атрибуты, кроме номера, даты завершения работ (CloseDate) и потраченного времени.

После утверждения задача переходит в состояние «реализуется»; в это время разрешается редактировать описание задачи и затраченное время (HoursSpent).

После реализации отдел тестирования выполняет проверку задачи (состояние «проверяется»), при этом может редактироваться только затраченное время (HoursSpent).

Если ошибок в реализации не обнаружено, то задача закрывается (состояние «закрыта»), при закрытии может быть указана дата завершения работ (CloseDate). Если были обнаружены ошибки в реализации, то задача возвращается в состояние «реализуется».

Задача в состоянии «закрыта» редактироваться не может.

В рамках данной лабораторной работы необходимо внести следующие изменения в модель, разработанную в ходе выполнения лабораторных работ №№ 2-3:

– добавить в сущность «Задача» атрибуты, содержащие текущее состояние задачи, дату/время утверждения задачи и дату/время завершения реализации;
– реализовать операции перехода задачи по состояниям.

Кроме того, необходимо для всех операций модели реализовать проверку предусловий:

– корректности переданных параметров;
– допустимости выполнения операции в текущем состоянии объекта.

Например, в качестве заказчика для проекта или должности сотрудника не может быть задано значение «null», в качестве затраченного на выполнение задачи времени или почасовой ставки сотрудника не могут указываться отрицательные величины, разряд сотрудника может быть только целым числом от 1 до 5, задачу в состоянии «закрыта» нельзя перевести в состояние «реализуется», у задачи в состоянии «реализуется» нельзя редактировать атрибут «Отдельно оплачивается заказчиком» и т. п.

Приложение не должно аварийно завершаться в случае ввода пользователем некорректных значений даты, чисел и т. п., а должно адекватно реагировать на ошибку. Например, выводить понятное сообщение и предлагать повторить ввод. (Указание: Ввод некорректных данных пользователем – это ситуация нормальная, а значит приводить к генерации исключений она не должна. Поэтому при реализации этого требования не следует ждать исключения и обрабатывать его в операторе try/catch, а заранее проверять корректность вводимых значений и на основе этой проверки действовать. В данной ситуации можно использовать методы TryParse)

Общий порядок работы с приложением остается, как и был реализован в лабораторных работах № 2 и № 3.

Вариант № 3. «Учебная программа»

Индивидуальная учебная программа студента может находиться в одном из следующих состояний: «черновик», «составлена», «утверждена», «в архиве» (рис. 4.3).

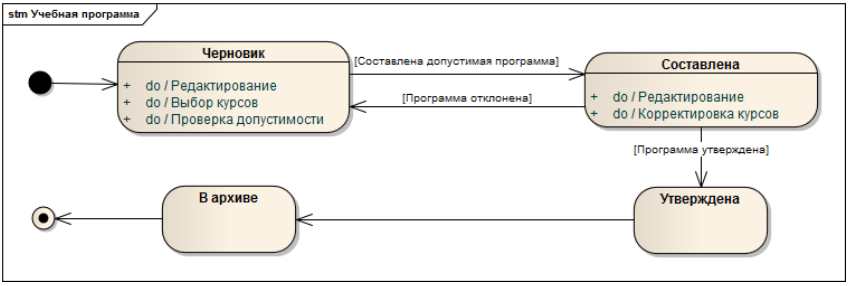

Рис. 4.3. Диаграмма состояний учебной программы (приложение «Учебная программа»)

После создания программа находится в состоянии «черновик». Студент осуществляет выбор курсов, формирование и проверку программы.

Когда программа готова, студент переводит ее в состояние «составлена» (в это состояние не может быть переведена недопустимая программа). Ответственный преподаватель осуществляет проверку, при необходимости корректирует программу, и либо утверждает ее, переводя в состояние «утверждена», либо отклоняет, переводя в состояние «черновик».

После завершения обучения студента программа переводится в состояние «в архиве».

В рамках данной лабораторной работы необходимо внести следующие изменения в модель, разработанную в ходе выполнения лабораторных работ №№ 2-3:

- добавить в сущность «Учебная программа» атрибуты, содержащие текущее состояние программы, дату/время составления и дату/время утверждения;
- реализовать операции перехода программы по состояниям.

Кроме того, необходимо для всех операций модели реализовать проверку предусловий:

- корректности переданных параметров;
- допустимости выполнения операции в текущем состоянии объекта.

Например, в качестве студента или получаемой степени в программе не может быть задано значение «null», в качестве часов лекций и практики не могут указываться отрицательные величины, программу в состоянии «в архиве» нельзя перевести в состояние «черновик», у программы в состоянии «утверждена» нельзя добавлять и удалять курсы и т. п.

Приложение не должно аварийно завершаться в случае ввода пользователем некорректных значений даты, чисел и т. п., а должно адекватно реагировать на ошибку. Например, выводить понятное сообщение и предлагать повторить ввод. (Указание: Ввод некорректных данных пользователем – это ситуация нормальная, а значит приводить к генерации исключений она не должна. Поэтому при реализации этого требования не следует ждать исключения и обрабатывать его в операторе try/catch, а заранее проверять корректность вводимых значений и на основе этой проверки действовать. В данной ситуации можно использовать методы TryParse)

Варианты № 4. «Начисление зарплаты»

Табель рабочего времени может находиться в одном из следующих состояний: «заполняется», «заполнен», «закрыт» (рис. 4.4).

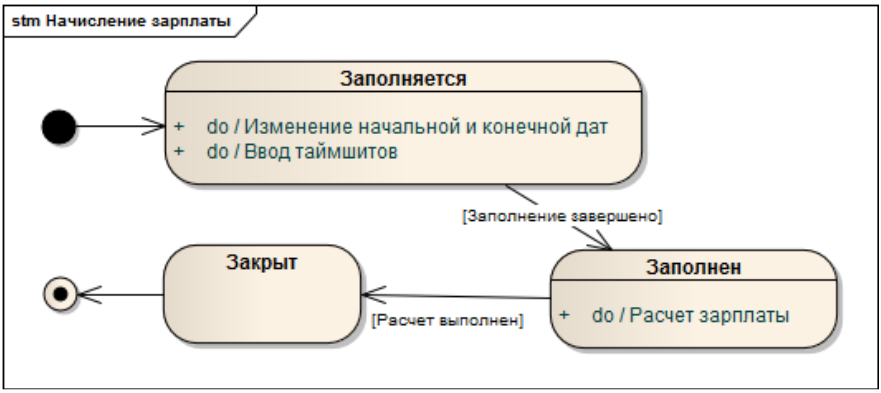

Рис. 4.4. Диаграмма состояний табеля рабочего времени (приложение «Начисление зарплаты»)

После создания табель находится в состоянии «заполняется». В этом состоянии в табель вносятся записи о рабочем времени (таймшиты), могут редактироваться атрибуты «Начало заполнения» и «Конец заполнения».

После завершения заполнения табель переходит в состояние «заполнен». В этом состоянии осуществляется проверка введенных данных и расчет заработной платы.

После расчета заработной платы табель переводится в состояние «закрыт». В этом состоянии с табелем не могут выполняться никакие операции.

В рамках данной лабораторной работы необходимо внести следующие изменения в модель, разработанную в ходе выполнения лабораторных работ №№ 2-3:

- добавить в сущность «Табель рабочего времени» атрибуты, содержащие текущее состояние табеля, дату/время фактического завершения и дату/время закрытия;
- добавить в сущность «Табель рабочего времени» атрибут «Величина заработной платы» (SalaryAmount);
- реализовать операции перехода табеля по состояниям.

Кроме того, необходимо для всех операций модели реализовать проверку предусловий:

- корректности переданных параметров;
- допустимости выполнения операции в текущем состоянии объекта.

Например, в качестве сотрудника для табеля или должности для сотрудника не может быть задано значение «null», в качестве часовой ставки должности и отработанного времени не могут указываться отрицательные величины, табель в состоянии «закрыт» нельзя перевести в состояние «заполняется», в табель в состоянии «заполнен» нельзя добавлять и удалять записи о рабочем времени и т. п.

Приложение не должно аварийно завершаться в случае ввода пользователем некорректных значений даты, чисел и т. п., а должно адекватно реагировать на ошибку. Например, выводить понятное сообщение и предлагать повторить ввод. (Указание: Ввод

некорректных данных пользователем – это ситуация нормальная, а значит приводить к генерации исключений она не должна. Поэтому при реализации этого требования не следует ждать исключения и обрабатывать его в операторе try/catch, а заранее проверять корректность вводимых значений и на основе этой проверки действовать. В данной ситуации можно использовать методы TryParse)

Вариант № 5. «Качество работы»

С точки зрения приложения, каждое изделие может находиться в одном из следующих состояний: «изготавливается», «дорабатывается», «выпущена» и «утилизация» (рис. 4.5).

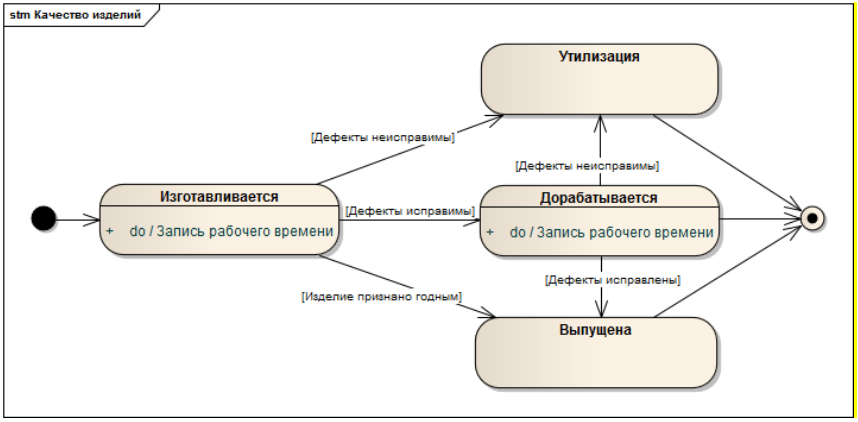

Рис. 4.5. Диаграмма состояний изделия
(приложение «Качество работы»)

Сразу после создания сущность «Изделие» находится в состоянии «изготавливается».

После завершения изготовления изделие либо выпускается (осуществляется переход в состояние «выпущена» и фиксируется дата/время выпуска), либо признается бракованным.

Бракованное изделие либо утилизируется (осуществляется переход в состояние «утилизация» и указывается значение атрибута «Брак»), если дефекты неисправимы, либо направляется на доработку (осуществляется переход в состояние «дорабатывается» и указывается значение атрибута «Брак»).

Изделие в состоянии «дорабатывается» обрабатывается приложением по тем же правилам, что и изделие в состоянии «изготавливается» (т. е. может быть либо выпущено, либо отправлено на дальнейшую доработку, либо отправлено на утилизацию).

Для изделия в состояниях «изготавливается» и «дорабатывается» могут вносится записи в журнал рабочего времени.

В рамках данной лабораторной работы необходимо внести следующие изменения в модель, разработанную в ходе выполнения лабораторных работ №№ 2-3:

- добавить в сущность «Изделие» атрибут, содержащий текущее состояние изделия;
- реализовать операции перехода изделия по состояниям.

Кроме того, необходимо для всех операций модели реализовать проверку предусловий:

- корректности переданных параметров;
- допустимости выполнения операции в текущем состоянии объекта.

Например, в качестве номенклатуры изделия не может быть задано значение «null», в качестве затраченного времени в журнале не может указываться отрицательная величина, изделие в состоянии «утилизация» нельзя перевести в состояние «изготавливается», для изделия в состоянии «выпущено» нельзя добавлять и удалять записи о затраченном рабочем времени и т. п.

Приложение не должно аварийно завершаться в случае ввода пользователем некорректных значений даты, чисел и т. п., а должно адекватно реагировать на ошибку. Например, выводить понятное сообщение и предлагать повторить ввод. (Указание: Ввод некорректных данных пользователем – это ситуация нормальная, а значит приводить к генерации исключений она не должна. Поэтому при реализации этого требования не следует ждать исключения и обрабатывать его в операторе try/catch, а заранее проверять корректность вводимых значений и на основе этой проверки действовать. В данной ситуации можно использовать методы TryParse)

Вариант № 6. «Лаборатория»

С точки зрения приложения, каждый исследуемый образец может находиться в одном из следующих состояний: «прием», «исследование», «обработка результатов» и «архив» (рис. 4.6).

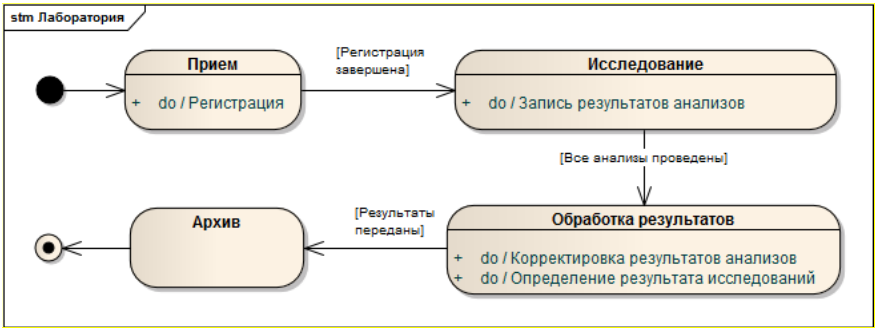

Рис. 6. Диаграмма состояний образца (приложение «Лаборатория»)

Сразу после создания сущность «Образец» находится в состоянии «прием». В этом состоянии осуществляется его регистрация и заполнение атрибутов «описание» и «контактная информация».

После завершения регистрации в лаборатории, образец отправляется на проведение анализов (осуществляется переход в состояние «исследование» и фиксируется дата/время). С этого момента изменения атрибутов сущности «Образец» недопустимы.

По мере проведения анализов в системе фиксируются их результаты (т. е. добавляются объекты «Результат анализа (Test)»).

После завершения анализов фиксируется время завершения исследования образца, и он переходит в состояние «обработка результатов». В этом состоянии определяется результат исследования, в ходе чего могут корректироваться результаты проведенных анализов (но уже не могут добавляться результаты новых анализов).

После определения результатов и их передачи контактному лицу образец переходит в состояние «архив», фиксируется время архивации и ФИО сотрудника лаборатории, утвердившего результат исследования. В состоянии «архив» не может редактироваться никакая информация, связанная с образцом и его анализами.

В рамках данной лабораторной работы необходимо внести следующие изменения в модель, разработанную в ходе выполнения лабораторных работ №№ 2-3:

- добавить в сущность «Образец» атрибуты: текущее состояние образца, дату/время начала исследований, дату/время завершения исследований, ФИО сотрудника, утвердившего результат исследования;
- реализовать операции перехода изделия по состояниям.

Кроме того, необходимо для всех операций модели реализовать проверку предусловий:

- корректности переданных параметров;
- допустимости выполнения операции в текущем состоянии объекта.

Например, в качестве типового анализа (TestDescription) для анализа (Test) не может быть задано значение «null», в диапазоне значений (Range) минимум не может быть больше максимума, образец в состоянии «архив» нельзя перевести в состояние «исследование», для образца в состоянии «прием» нельзя добавлять и удалять записи о результатах анализов и т. п.

Приложение не должно аварийно завершаться в случае ввода пользователем некорректных значений даты, чисел и т. п., а должно адекватно реагировать на ошибку. Например, выводить понятное сообщение и предлагать повторить ввод. (Указание: Ввод некорректных данных пользователем – это ситуация нормальная, а значит приводить к генерации исключений она не должна. Поэтому при реализации этого требования не следует ждать исключения и обрабатывать его в операторе try/catch, а заранее проверять корректность вводимых значений и на основе этой проверки действовать. В данной ситуации можно использовать методы TryParse)

Общий порядок работы с приложением остается, как и был реализован в лабораторных работах № 2 и № 3.

Вариант № 7. «Учебные сертификаты»

Сущность AttendedCourse может находиться в одном из следующих состояний: «обучение», «аттестация», «отчислен», «архив» (рис. 4.7).

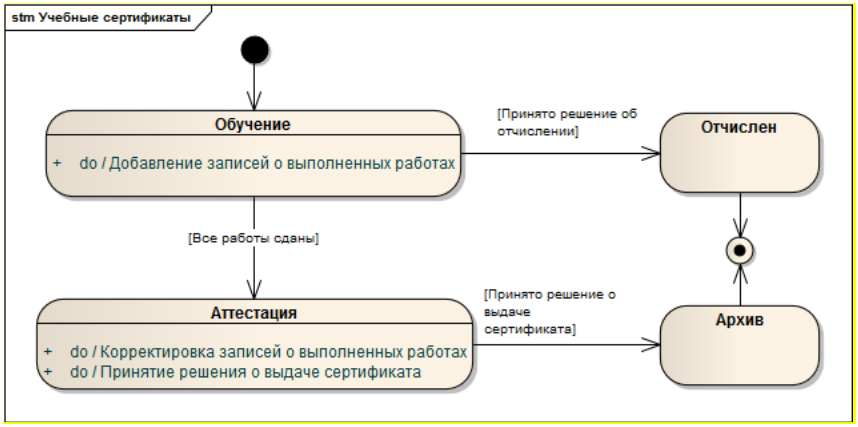

Рис. 4.7. Диаграмма состояний сущности AttendedCourse (приложение «Учебные сертификаты»)

Сразу после создания сущность AttendedCourse находится в состоянии «обучение». В ходе обучения студента заполняется информация о результатах сдачи им требуемых работ.

Периодически преподаватель осуществляет контроль сдачи работ. Если оказывается, что студент на момент контроля не сдал ни одной работы из запланированных, преподаватель отчисляет студента. Сущность AttendedCourse переходит в состояние «отчислен», в ней фиксируется дата/время отчисления.

После сдачи студентом всех работ (включая итоговый экзамен) сущность AttendedCourse переходит в состояние «аттестация». В этом состоянии преподаватель может корректировать результаты студента (но не может добавлять новые записи). Преподаватель принимает решение о выдаче или не выдаче сертификата, фиксирует принятое решение, а также дату/время. Сущность переходит в состояние «архив».

В состоянии «архив» информация редактироваться не может.

В рамках данной лабораторной работы необходимо внести следующие изменения в модель, разработанную в ходе выполнения лабораторных работ №№ 2-3:

- добавить в сущность AttendedCourse атрибуты, содержащие: текущее состояние, дату/время отчисления, дату/время принятия решения о выдаче сертификата, само решение о выдаче сертификата;
- реализовать операции перехода AttendedCourse по состояниям.

Кроме того, необходимо для всех операций модели реализовать проверку предусловий:

- корректности переданных параметров;
- допустимости выполнения операции в текущем состоянии объекта.

Например, в качестве требований к работе не может быть задано значение «null», в качестве количества баллов по работе не может указываться отрицательная величина, AttendedCourse в состоянии «отчислен» нельзя перевести в состояние «аттестация», для AttendedCourse в состоянии «архив» нельзя добавлять и удалять записи о выполняемых работах и т. п.

Приложение не должно аварийно завершаться в случае ввода пользователем некорректных значений даты, чисел и т. п., а должно адекватно реагировать на ошибку. Например, выводить понятное сообщение и предлагать повторить ввод. (Указание: Ввод некорректных данных пользователем – это ситуация нормальная, а значит приводить к генерации исключений она не должна. Поэтому при реализации этого требования не следует ждать исключения и обрабатывать его в операторе try/catch, а заранее проверять корректность вводимых значений и на основе этой проверки действовать. В данной ситуации можно использовать методы TryParse)

КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ

1. Опишите механизм обработки ошибок посредством исключений.

2. Приведите синтаксис генерации исключения. Назовите правила, по которым выбирается класс генерируемого исключения.

3. Приведите синтаксис конструкции try/catch/finally. Каково назначение ее блоков?

4. Назовите и охарактеризуйте основные стратегии обработки исключений в блоке catch.

5. Сформулируйте основные правила обработки исключений.

6. Как и для чего организуются цепочки исключений?

7. Опишите иерархию исключений библиотеки .NET Framework BCL.

ФОРМА ОТЧЕТА ПО ЛАБОРАТОРНОЙ РАБОТЕ

На выполнение лабораторной работы отводится 8 часов: 7 часов на выполнение работы, 1 час на подготовку и защиту отчета по лабораторной работе.

Номер варианта студенту выдается преподавателем.

Отчет на защиту предоставляется в печатном виде.

Структура отчета (на отдельном листе(-ах)): титульный лист, формулировка задания (вариант), UML-диаграмма классов модели, этапы выполнения работы (со скриншотами), результаты выполнения работы. выводы.