ТЕХНОЛОГИЧЕСКОЕ ОБРАЗОВАНИЕ

Степанов Олег Георгиевич

ОПИСАНИЕ ПРОСТЕЙШЕГО ИНДИВИДУАЛЬНОГО ПРОЦЕССА РАЗРАБОТКИ НА ПРИМЕРЕ ПЛАТФОРМЫ .NET

ВВЕДЕНИЕ

В программах, предназначенных для подготовки специалистов в области разработки программного обеспечения (ПО), важное место занимают курсы, посвященные алгоритмам и структурам данных, различным платформам и методологиям разработки промышленного ПО. Тем не менее, к сожалению, достаточно мало внимания уделяется индивидуальной разработке ПО, хотя умение грамотно организовать его разработку является важнейшим качеством современного разработчика. Обычно эти навыки приобретаются во время производственной практики или в начале трудовой деятельности, но получение знаний в течение этого периода имеет ряд недостатков. Во-первых, время, потраченное на обучение, уменьшает время, предназначенное для разработки ПО, а во-вторых, в таком режиме опыт приобретается через наблюдение за коллегами, что приводит к несколько одностороннему представлению об организации процесса.

Хорошей альтернативой могло бы стать обучение основам индивидуальной организации процесса во время обучения в ВУЗе. Помимо увеличения уровня подготовки выпущенных специалистов, это позволило бы повысить качество учебных работ, выполняемых в рамках различных курсов.

В данной статье приводится пример процесса разработки, который не требует большого опыта в разработке ПО и может быть внедрен в качестве требования при выполнении учебных заданий широкого спектра сложности.

Описанный процесс может быть использован при разработке приложений на платформе .NET. Данная платформа была выбрана не случайно: усилия Microsoft по продвижению ее в образовательных учреждениях достаточно серьезны, и возможность использования .NET в учебных курсах подтверждена как российскими [1], так и зарубежными [2] специалистами.

В статье предлагается набор инструментов, проверенный как автором, так и многими другими разработчиками во всем мире, а также описывается интеграция их в единый процесс разработки.

ОБЗОР НАБОРА ИНСТРУМЕНТОВ

В первую очередь, опишем набор инструментов, интеграция которых будет рассмотрена в данной статье. Практически все они решают задачи, являющиеся частью любого серьезного процесса разработки для одного разработчика.

Во-первых, необходим .NET Framework SDK. Здесь мы не будем рассматривать альтернативные реализации CLI, например, от

группы Mono [3], а воспользуемся стандартным .NET Framework SDK от Microsoft.

Во-вторых, необходимо средство для редактирования кода. Вариантов множество: обычный текстовый редактор, возможно, с поддержкой подсветки кода; Visual Studio .NET; SharpDevelop [4]. В принципе, для наших целей хватит простейшего варианта.

Это базовый набор, который необходим для создания любого приложения. Но он не является достаточным – даже при использовании интегрированного пакета Visual Studio .NET множество важных задач требуют использования дополнительных инструментов:

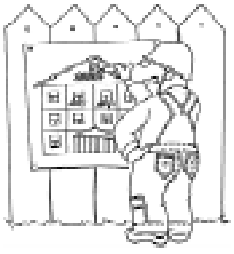
• *Проверка кода:* в разработке очень важно следовать единым правилам написания кода – это могут быть некоторые личные принципы, корпоративные правила кодирования или общие стандарты, принятые для данного языка или технологии. Для разработки на языке C# существуют так называемые Code Design Guidelines [5], разработанные Microsoft, которых рекомендуется придерживаться при написании кода.

Чтобы упростить такую проверку, выпущены средства автоматического анализа кода на предмет соответствия стандартам. Большинство из них позволяют расширять набор правил, что помогает отслеживать часто возникающие ошибки. В этой категории также присутствует несколько игроков, вот некоторые из них: FxCop от Microsoft и Code Analysis Tool из пакета DevPartner от компании CompuWare, известной многим разработчикам инструментами BoundsChecker и SoftICE. Здесь мы рассмотрим работу с Microsoft FxCop ввиду двух очевидных преимуществ: малого размера (дистрибутив занимает около 4.5 мегабайт вместе с базой правил, которая, кстати, может быть расширена пользователем) и бесплатности;

• *Блочное тестирование (unit testing):* это широкая тема, которой посвящен не один труд. Некоторые методологии (например, Extreme Programming [6]) включают unit testing как одну из основных практик, а многие компании – в корпоративные правила разработки кода. Инструментов существует очень много, но здесь мы рассмотрим удобное бесплатное средство под названием NUnit;

• *Сборка кода:* При работе с хоть сколько-нибудь сложными проектами (больше одного исходного файла) проблема сборки кода в исполняемые модули приобретает особое значение. Важно собрать код в правильном порядке, причем, что особенно важно в больших проектах, нужно избежать повторной сборки кода, для которого уже есть собранный модуль, если код не изменился с момента последней сборки (иначе сборка может занять слишком много времени, что особенно неприятно при использовании continuous integration [7]), правильно настроить среду выполнения, скопировать все нужные файлы и т. п. На уровне пакетных скриптов такую проблему решить непросто, но, к счастью, существуют специальные средства, обеспечивающие корректную сборку кода. Такие средства включены практически в каждую современную интегрированную среду разработки, но в большинстве случаев они достаточно бедны, так что приходится использовать специальные инструменты. Здесь мы опишем работу со средством NAnt [8] – бесплатным инструментов для сборки .NET–проектов. Отметим, что Microsoft работает над собственной системой построения кода MSBuild, которая в ближайшее время, вероятно, станет стандартным инструментом, но в данной статье эта система не описывается, так как она официально еще недоступна широкому кругу разработчиков.

Этот набор инструментов автор использует практически постоянно при разработке на .NET. Инструменты хорошо взаимодействуют между собой (например, в состав NAnt входит команда, запускающая тестирование продукта с помощью NUnit) и прекрасно показали себя в работе.



ПРОЕКТИРОВАНИЕ ПРИЛОЖЕНИЯ

Теперь мы имеем все необходимые инструменты под рукой, можно начинать рабо-

тать над приложением. В рамках статьи мы создадим простейшее приложение, которое будет вычислять среднее арифметическое значение для набора чисел типа float, передаваемых в виде массива.

Мы не собираемся освещать здесь вопросы проектирования приложений. Раздел выделен для того, чтобы показать правильную последовательность шагов и не писать тесты до создания ОО-модели.

Для нашего приложения достаточно одного класса с одним public-методом, который и будет вычислять среднее арифметическое. Сразу напишем заглушку для этого класса, которая определит его интерфейс. Написание заглушек является очень полезной практикой, особенно в проектах с несколькими разработчиками, так как позволяет согласовать интерфейсы прямо в коде – вы не обязаны уже иметь реализацию, чтобы разработчик, использующий компонент, мог писать код, использующий его. В нашем случае заглушка будет выглядеть максимально просто (см. листинг 1).

Из этого кода можно вынести сразу несколько хороших практических рекомендаций:

• Используйте осмысленные имена для пространств имен. Хорошим тоном считается иметь иерархическую организацию, начинающуюся с названия вашей компании (практику использования перевернутых доменных имен – org.ksoap.serialization – лучше оставить в Java). Хорошим примером можно считать Microsoft.Office.Excel. Важно, чтобы по названию пространства имен можно было бы сразу определить назначение кода – в нашем примере имя пространства имен можно просто разложить на части, и сразу будет видно, что:

– код написан мной (Nocturne);

– код написан для статей (Articles);

– статья посвящена процессу разработки (DevelopmentProcess);

– код является примером к статье (Demo).

В Visual Studio можно задать корневое пространство имен для всего проекта (по умолчанию оно соответствует названию

Листинг 1.

```
using System;

namespace Nocturne.Articles.DevelopmentProcess.Demo
{
   /// <summary>
   /// Calculates average value of array of floats.
   /// </summary>
   public class AverageCalculator
   {
      public AverageCalculator()
      {
      }

      /// <summary>
      /// Calculates average value of array of floats.
      /// </summary>
      /// <param name="values">Array of values to calculate average for</param>
      /// <returns>Average value for the given array</returns>
      public float CalculateAverage( float[] values )
      {
         throw new NotImplementedException("The method hasn't been
                                                    implemented yet");
      }
   }
}
```

проекта). Это делается в окне свойств проекта (рисунок 1).

• Используйте XML-комментарии. Во-первых, это стандартный способ рассказать людям о своем коде, а во-вторых, существуют специальные программы (например, NDoc), которые позволяют по XML-комментариям строить документацию к коду. Совсем необязательно писать комментарии к каждому предложению в коде, особенно если вы следуете хорошей практике не писать длинных методов – в этом случае общий комментарий к методу позволяет разобраться в коде без особых проблем.

• Для заглушек можно использовать специально предназначенное для этого исключение. Но иногда, особенно когда проверяется работа смежных систем, проще возвращать какое-либо постоянное значение.

Теперь приступим к применению наших инструментов. И начнем с написания тестов.

РАЗРАБОТКА СИСТЕМЫ ТЕСТОВ

Имея модель приложения и спецификацию, можно приступать к созданию тестов. Тестировать рекомендуется каждый метод строго согласно спецификации. Пусть наша мини-спецификация будет такой:

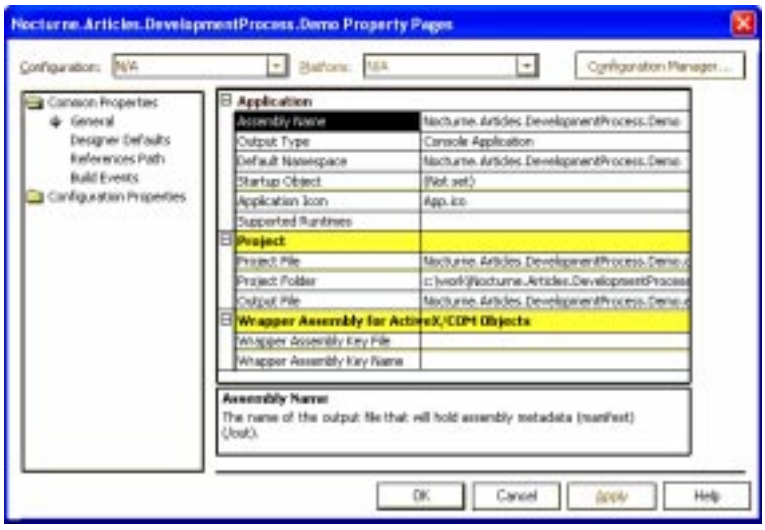


Рисунок 1.

> «Метод CalculateAverage класса AverageCalculator должен вычислять среднее арифметическое для массива чисел типа float. Если передан пустой массив, то он должен порождать исключение типа ArgumentException, а если вместо массива передана пустая ссылка (null), то исключение должно порождаться типа ArgumentNullException».

Построим теперь систему блочных тестов, соответствующих этой спецификации, с помощью NUnit 2.0.

Удобство .NET и NUnit в том, что тесты можно встраивать прямо в сам код. Это делается с помощью специальных тестовых классов, методы которых и производят тестирование; единственное, что нужно еще сделать, – это добавить ссылку на сборку nunit.framework.dll, чтобы получить доступ к NUnit API. Классы специально помечаются, затем система NUnit загружает вашу сборку, ищет тестовые классы с помощью Reflection и выполняет их методы. С помощью специального API тестовые методы уведомляют NUnit об успехе или провале теста, а он уже сводит их в удобное дерево тестов и отображает вам.

Например, тестовый класс для приведенной выше спецификации будет выглядеть следующим образом (см. листинг 2).

Сразу отметим ряд полезных практических рекомендаций, проиллюстрировав их на этом примере:

• Удобно все тесты выделять в отдельное пространство имен Tests и располагать файлы тестов в соответствующей поддиректории (кстати, например, Visual Studio автоматически задаст пространство имен для файлов в этой папке, если, естественно, было прописано корневое пространство имен для этого проекта, как было показано выше).

• Принято создавать отдельный класс тестов для

Листинг 2.

```
using System;

using NUnit.Framework;

namespace Nocturne.Articles.DevelopmentProcess.Demo.Tests
{
   /// <summary>
   /// Test class for the <see
cref="T:Nocturne.Articles.DevelopmentProcess.Demo.AverageCalculator"/> class.
   /// </summary>
   [TestFixture]
   public class AverageCalculatorTest
   {
      [Test]
      public void TestCalculateAverage()
      {
         AverageCalculator calculator = new AverageCalculator();

         Assertion.AssertEquals("CalculateAverage should return average value",
          2.0F, calculator.CalculateAverage(new float[] {1.0F, 2.0F, 3.0F}));

         try
         {
            calculator.CalculateAverage(new float[] {});
            Assertion.Fail("CalculateAverage should throw ArgumentException
                                                    for empty array");
         }
         catch (ArgumentException) {}

         try
         {
            calculator.CalculateAverage(null);
            Assertion.Fail("CalculateAverage should throw ArgumentNullException
                                          for null reference argument");
         }
         catch (ArgumentNullException) {}
      }
```

каждого класса системы, при этом название формируется из названия тестируемого класса с суффиксом Test. Можно, конечно, и называть их так же, как и тестируемые классы, ибо тесты находятся в отдельном пространстве имен, но в этом случае возникает неоднозначность с сокращенными именами классов и приходится писать полное имя тестируемого класса (с пространством имен). Отметим, что тестовые классы выделяются атрибутом TestFixture.

- Тестирование внутри класса можно разбивать по-разному. Можно, как показано в примере выше, писать отдельный тестовый метод для каждого метода. В этом случае название метода тестового класса формируется из названия тестируемого метода с префиксом Test. Можно также писать отдельный метод для каждого сценария тестирования. В нашем случае мы в одном методе проводим три сценария, как и описано в спецификации: нормальные данные, пустой массив и нулевая ссылка, в отдельный метод.

В этом пункте отметим еще важный момент: методы нужно помечать атрибутом Test.

Сначала проведем smoke-test и убедимся, что наш код компилируется без ошибок. Для этого запускаем NUnit GUI (ко-

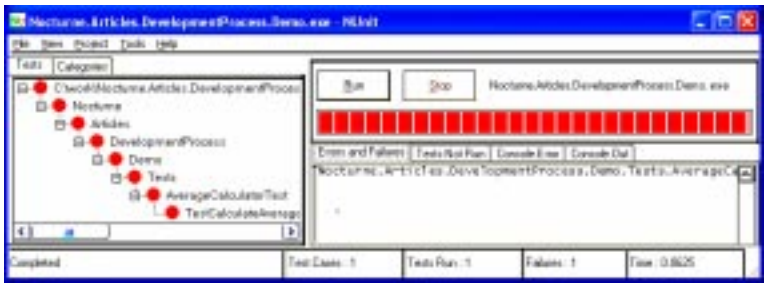


Рисунок 2.

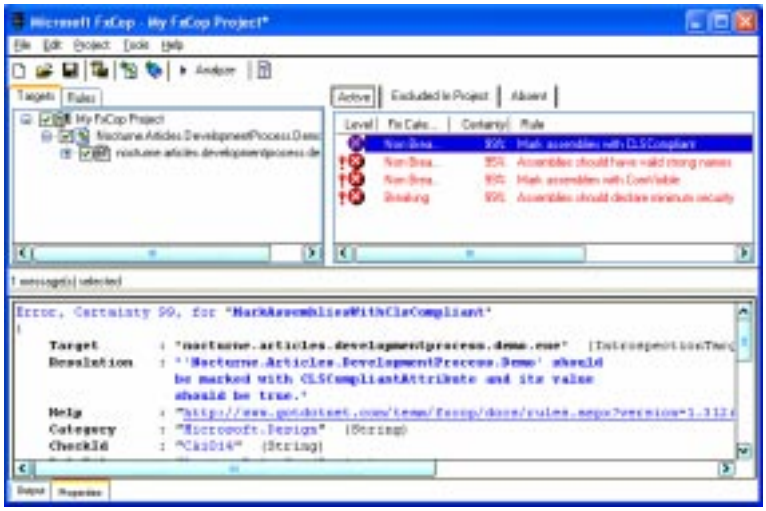


Рисунок 3.

торый удобно встроить в среду разработки, чтобы он вызвался для текущего проекта по горячей клавише), открываем в нем сборку и запускаем тесты. Результат соответствует ожиданиям (рисунок 2).

Как я уже упоминал, NUnit представляет все тесты в виде удобного дерева, так что сразу можно увидеть, какие тесты выполнены успешно, а какие – нет. NUnit позволяет также организовывать тесты в так называемые test suites, но в нашем случае в этом нет необходимости, ибо у нас есть один тест и он не выполнился, причем в окне статуса сразу можно увидеть причину: *The method hasn't been implemented yet.*

Теперь можно приступать к реализации, но сначала проверим соответствие нашего кода стандартам. Для этого запустим программу FxCop, создадим новый проект, добавим в него нашу сборку и проанализируем ее (рисунок 3).

Как видно, в нашем коде не все так гладко. На самом деле, приведенные здесь ошибки достаточно важны, если вы собираетесь распространять вашу сборку за пределы собственного компьютера, но в нашем случае мы не будем описывать, что нужно сделать, тем более что каждая ошибка подробно документирована.

Убедившись, что действительно серьезных несоответствий нет, можно приступить к написанию кода метода CalculateAverage. (см. листинг 3).

Листинг 3.

```
/// <summary>
/// Calculates average value of array of floats.
/// </summary>
/// <param name="values">Array of values to calculate average for</param>
/// <returns>Average value for the given array</returns>
public float CalculateAverage( float[] values )
{
   float result = 0;

   foreach (float value in values)
      result += value;

   result /= values.Length;

   return result;
}
```

Теперь скомпилируем сборку и выполним тесты (рисунок 4).

Как видно, тесты не прошли. Дело в том, что наша реализация соответствует только первому пункту спецификации, а проверка уже второго ведет к ошибке. Теперь реализуем все три пункта, введя проверки входных данных – в этом случае реализация будет такой, как в листинге 4.

Проведя тестирование, можно увидеть следующий результат – тесты пройдены успешно (рисунок 5).

Рисунок 4.

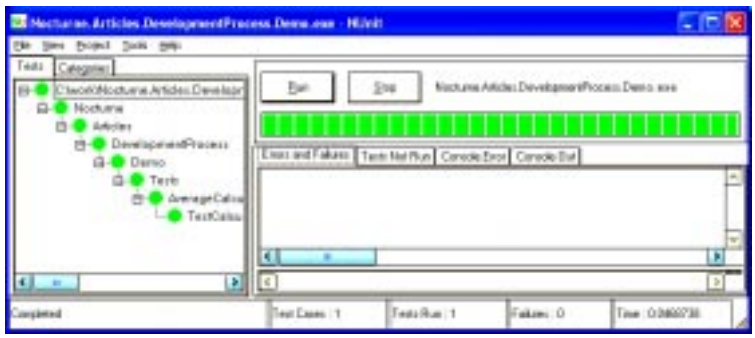
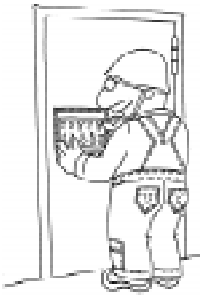


Рисунок 5.

ПОСТРОЕНИЕ КОДА



Как уже упоминалось выше, в хоть сколько-нибудь сложных проектах проблема быстрого построения кода встает достаточно остро, особенно когда требуется построение кода в реальном времени. Для такой операции удобно использовать скрипты. Можно писать скрипты на языках сценариев общего назначения – Windows Command Shell, Jscript/VBScript + Windows Scripting Host и т.п., но чаще всего они не приспособлены для решения наших задач, и приходится множество функций дописывать своими руками.

Листинг 4.

```
/// <summary>
  /// Calculates average value of array of floats.
  /// </summary>
  /// <param name="values">Array of values to calculate average for</param>
  /// <returns>Average value for the given array</returns>
  public float CalculateAverage( float[] values )
  {
    if (values == null)
      throw new ArgumentNullException("values", "Array should not be null");
    if (values.Length == 0)
      throw new ArgumentException("values", "Array should not be empty");

    float result = 0;

    foreach (float value in values)
      result += value;

    result /= values.Length;

    return result;
  }
```

Тут на помощь приходят специальные системы построения. В данной статье мы рассмотрим, как уже упоминалось, систему NAnt, которая является результатом переноса системы построения Ant для Java.

Для использования этой системы нужно всего лишь создать простой скрипт для построения, основанный на XML, а затем исполнить его простой командой. Кстати, рекомендуем включать дистрибутив nant в состав выпускной версии, ибо в базовой конфигурации он занимает совсем мало места. Пример простейшего сценария построения, находится в файле template.build.

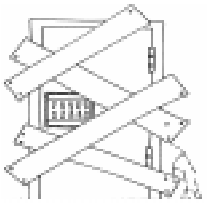
Этот пример поможет нам разобраться в структуре файла построения: корневым тегом является тег project, описывающий проект. В нем могут быть описаны свойства проекта (тег property) и так называемые targets – «цели» проекта, каждая из которых задает определенный сценарий сборки. Принято поддерживать, по крайней мере, три цели – для выпускных версий (release), для отладочных версий (debug) и для очистки промежуточных файлов (clean). Так и сделано в примере выше. Отладочная цель просто компилирует исходный код в отладочном режиме, выпускная – помимо этого прогоняет все тесты и собирает документацию на основе XML-комментариев.

Также удобно написать некий механизм для автоматической нумерации построений. Лично я написал простой скрипт на JScript (см. листинг 5).

Листинг 5.

```
// Increases build number in version string of the project

function doAction()
{
   try
   {
        objArguments = WScript.Arguments;

   if (objArguments.length < 1)
   {
        WScript.Echo("[!] Missing arguments");
        return;
   }

        objBuild = new ActiveXObject("Msxml2.DOMDocument.4.0");
        objBuild.async = false;
        objBuild.load(WScript.Arguments(0));
        objProjectBuild = objBuild.selectSingleNode("/project/
                                  property[@name = 'project.build']");

        build = new Number(objProjectBuild.getAttribute("value"));
        build = build + 1;

        objProjectBuild.setAttribute("value", build);

        objBuild.save(WScript.Arguments(0));
     }
     catch(e)
     {
        WScript.Echo("[!]Exception: " + e.description);
     }
}

doAction();
```

Этот простой скрипт открывает build-файл, переданный ему в качестве параметра, находит в нем значение свойства project.build и увеличивает его на единицу. Если вызов этого скрипта прописать в командном файле, используемом для построения версии, то увеличение номера построения будет происходить автоматически.



ЗАКЛЮЧЕНИЕ

Подведем итоги. Мы рассмотрели, как можно использовать вместе различные утилиты для разработчика, чтобы связать их в единую линию. Отметим, что интеграция рассматривалась для одного разработчика. В случае продукта список задач, а соответственно, и инструментов значительно расширяется. Например, при использовании continuous integration удобно настроить CruiseControl.NET [10]. Но эти темы выходят за рамки данной статьи – возможно, я рассмотрю их в одной из следующих.

Литература

1. *A.A. Terekhov*, *D. Boulychev*, A. *Moscal, N. Voyakovskaya* «Teaching Compiler Development Using .NET Platform», In Proceedings of the 2nd International Workshop on .NET Technologies, Plzen, Czech Republic, May 31-June 2, 2004. P. 33–39.

2. *H. Paul Haiduk.* Object-oriented classic data structures for CS2 in C#, Journal of Computing Sciences in Colleges.

3. The Mono Project, (http://www.go-mono.com).

4. #Develop by IC#Code, (http://www.icsharpcode.net).

5. .NET Framework Design Guidelines (http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpgenref/html/cpconnetframeworkdesignguidelines.asp).

6. eXtreme Programming Explained 1st edition, Kent Beck, Addison-Wesley Pub Co.

7. Continuous Integration by Martin Fowler (http://www.martinfowler.com/articles/continuousIntegration.html).

8. The NAnt Project (http://nant.sourceforge.net).

9. NUnitAsp (http://nunitasp.sourceforge.net/).

10. CruiseControl.NET (http://www.continuousintegration.net/).

Степанов Олег Георгиевич,
студент 5 курса факультета
Информационных Технологий и
Программирования СПбГУ ИТМО.