**Лабораторная работа №4**

**“Разработка тестов на основе исполняемой спецификации”**

**Цель работы:** Научиться разрабатывать тесты на основе исполняемой спецификации

**Продолжительность работы** - 4 часа.

**Содержание**

## 1. Разработка, управляемая поведением…..............................................................................1 2. Спецификация на основе примеров………….. ………………………...............................3 3. Пример…...…………..……………........................................................................................5 4. Порядок выполнения лабораторной работы......................................................................10

## Разработка, управляемая поведением

## *Тесты на основе ключевых слов (кeyword-driven)*. Практически всегда автоматизированное тестирование это штука дорогая и сложная, а "экономика должна быть экономной". Автотесты на основе ключеслов (keyword-driven testing) - это как раз та техника, что призвана снизить стоимость за счёт привлечения к написанию собственно тестов технически слабоподготовленных лиц.

## Тестирование на основе действий очень схоже с тестированием, управляемым данными, является в какой-то степени его продолжением и расширением. Основная идея - разделить автоматизацию тестирования на две неравные по времени и стоимости части: написание собственно тестов в виде последовательности ключевых действий и реализацию (создание инфраструктуры).

## Написание тестов подразумевает описание тестовых сценариев в табличном виде, где каждая строка состоит из ключевого слова описывающего некое действие и каких-либо данных. Возможны, естественно, вариации: нет данных, добавлены комментарии и пр.. Такую задачу могут выполнять технически слабые специалисты, знакомые с предметной областью тестируемого приложения. За счёт этого написание собственно тестов становится относительно недорогим.

## Создание инфраструктуры заключается в   1. написании реальных классов, методов, скриптов тестирующих приложение;   2. трансляции ключевых слов из просто текста в вызовы тех методов, запуск нужных скриптов и т.д. .

## Эта часть подхода требует технически грамотных (читаем "дорогих") сотрудников. Зато таких "грамотеев" нужно много меньше, чем писателей тестов.

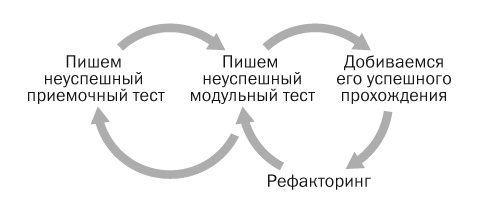
## Таким образом, получаются две условно независимые части: тесты в виде таблиц и код, который собственно "нажимает кнопки" и проверяет результаты.

## *Тесты на основе поведений (behaviour-driven)*. Дальнейшее развитие это техники, которое всё больше набирает популярность это тестирование на основе поведений (*behaviour-driven*). Всё практически то же самое, только тесты не только в виде таблиц, но и в виде структурированного текста. Например: Test Case   Valid Login Step   Given login page is open   When valid username and password are inserted   and credentials are submitted   Then welcome page should be open

## Given, When, Then - фактически зарезервированные слова, определяющие структуру, описывающее то самое поведение.

## Тесты теперь могут писать вообще кто угодно, например аналитики, главное чтоб соблюдали структуру. *Преимущества:* + Можно быстро создавать новые тесты(при наличии инфраструктуры); + "Не-технари" могу писать тесты, что снижает стоимость и теоретически ускоряет процесс; + благодаря популярности этих подходов появилось приличное количество практически готовых к немедленному использованию инструментов. С ними создание инфраструктуры упрощается на порядки. *Недостатки:* - Создание гибких решений может быть трудным; - Те немногие технари, пишущие инфраструктуру, должны быть достаточно сильны в программировании.

## BDD (behavior-driven development) — это вариация разработки, управляемой тестами (test-driven development, TDD), в которой основное внимание уделяется историям пользователей и построению логики и тестов на основе этих историй. Вместо следования одному правилу вы следуете набору операций. Идея в том, что, хотя типичный модульный тест может гарантировать корректную работу одного события в объекте, BDD фокусируется на более широкой концепции — на поведении, которое пользователь ожидает при работе с создаваемой для него системой. BDD часто применяется, чтобы определить критерии приемлемости (acceptance criteria) в процессе обсуждения с клиентами. Например, когда я сижу перед компьютером и заполняю форму New Customer, а затем щелкаю кнопку Save, система должна сохранить информацию о клиенте и вывести мне сообщение о том, что эта информация успешно сохранена.

****

**Цикл разработки на основе поведений (BDD)**

## Спецификация на основе примеров

*Specification By Example* – это процесс, позволяющий регулярно добиваться выполнения обоих пунктов. В основе процесса лежат agile, tdd, bdd, continuous integration и автоматизация тестирования.  
Ключевые элементы *Specification By Example*:

1. Выделяйте главное *(deriving scope from goals)*
2. Составляйте спецификацию совместно *(specifying collaboratively)*
3. Приводите примеры *(illustrating using examples)*
4. Очищайте спецификацию *(refining the specification)*
5. Внедряйте автоматизацию тестирования без изменения спецификации *(automating validation without changing specification)*
6. Встраивайте выполнение тестов в процесс сборки и развивайте документацию *(validating frequently, evolving a documentation system)*

**Given** — первоначальный контекст (предусловие)   
**When** — событие (что является триггером сценария)  
**Then** — результат, который мы хотим получить

Прежде чем кодировать, составьте спецификацию. Задумайтесь, как часто вы проклинали того, кто составлял требования и приемочные критерии. Возьмите на себя ответственность и составьте спецификацию сами в том виде, в котором вам будет удобно работать.

Вот несколько примеров неудачных пунктов, попавших в спецификацию:

* Все страницы должны отображаться за 0.1 секунды. Слово «все» следует избегать в спецификации в принципе. Да, главная страница должна открываться настолько быстро, насколько возможно. Но вы потратите недели или месяцы, пытаясь загнать в эти рамки сводный отчет, который запускают раз в год. Такие редкие и ресурсоемкие операции могут выполняться долго. Ничего страшного в этом нет.
* User-interface должен выглядеть в стиле OSX. Мода породила тысячи сайтов с заменой родных контролов, а вместе с ними боль тысяч верстальщиков и web-разработчиков. Пусть интерфейс выглядит в стиле OSX на маке и в браузере Сафари. В других системах оставьте родные элементы управления.
* На главной странице веб-сайта должен быть флеш-баннер. Возможно, для этого баннера вообще не нужен флеш, и все можно сделать средствами html/css/javascript, а аналогичный модуль у вас есть уже «из коробки». Избегайте чрезмерного уточнения технологических аспектов в спецификации.

На данный момент существует множество BDD framework’ов, однако “наиболее верным” с точки зрения родоначальников данного термина ([Dan North, David Chelimsky, Aslak Hellesøy](http://www.google.com/search?rlz=1C1GGLS_ruRU348RU344&sourceid=chrome&ie=UTF-8&q=Dan+North,+David+Chelimsky,+Aslak+Helles%C3%B8y)) является такой проект как: [Cucumber](http://cukes.info/) (ранее известный как [RSpec](http://rspec.info/)). Этот проект написан для Ruby. Большинство [RubyOnRails](http://rubyonrails.org/) разработчиков (как основные представители ruby сообщества) пользуются именно этим фреймворком для тестирования своих продуктов.

Известны некоторые способы запуска данного проекта под [.NET с помощью IronRuby](http://wiki.github.com/aslakhellesoy/cucumber/ironruby-and-net), однако должной поддержки со стороны как сред разработки в .NET, так и внимания сообщества такое решение не получило.

Инструменты для .NET в основном копируют формат и принципы, заложенные в Cucumber, их существует несколько:

* [SpecFlow](http://www.specflow.org/) – предмет сегодняшнего разговора (из плюсов – аналогия с Cucumber и интеграция в Visual Studio 2008 и 2010)
* [NBehave](http://code.google.com/p/nbehave/) – работа с данным фреймворком близка к возможностям Cucumber в настоящее время (до этого он так же был в разряде “многословных” и  это, наверное, наиболее распространенный BDD фреймворк среди .NET разработчиков)
* [NSpecify](http://nspecify.sourceforge.net/), [NSpec](http://nspec.tigris.org/) – довольно-таки “многословные” фреймворки для BDD
* [Specter](http://specter.sourceforge.net/) – аналогично, примечателен тем, что написан на [Boo](http://boo.codehaus.org/)

SpecFlow следует специфическому шаблону, который полагается на ключевые слова, помогающие описывать функцию, чье поведение вы определяете. Ключевые слова взяты из языка Gherkin (да, так называют мелкие маринованные огурцы), и все это исходит из утилиты Cucumber (cukes.info). Некоторые из ключевых слов — Given, And, When и Then, и вы можете использовать их при написании сценария.

Рассмотрим тест, переработанный с обычного вида тестов в MSTest на тест в SpecFlow

**Исходный тест**

[**TestMethod**]

public void **CreatePluralName\_SucceedsOnSamples**()

{

// setup

**var** target = new **NameCreator**();

**var** pluralSamples = new **Dictionary**<string, string>

{

{ "ballista", "ballistae" },

{ "class", "classes"},

{ "box", "boxes" },

{ "byte", "bytes" },

{ "bolt", "bolts" },

{ "fish", "fishes" },

{ "guy", "guys" },

{ "ply", "plies" }

};

foreach (**var** sample **in** pluralSamples)

{

// act

**var** result = target.**CreatePluralName**(sample.**Key**);

// verify

**Assert**.**AreEqual**(sample.**Value**, result);

}

}

**Тест в SpecFlow**

**Feature**: PluralNameCreation

In order to assign names to Collection type of Navigation Properties

I want to convert a singular name to a plural name

**@PluralName**

**Scenario** **Outline**: Create a plural name

**Given** I have a 'Name' defined as '<name>'

**When** I convert 'Name' to plural 'Result'

**Then** 'Result' should be equal to '<result>'

**Examples**:

| name | result |

| ballista | ballistae |

| class | classes |

| box | boxes |

| byte | bytes |

| bolt | bolts |

| fish | fishes |

| guy | guys |

| ply | plies |

## В этом самом классическом подходе «входные» данные для теста специально создаются в самом тесте.

## Пример

Рассмотрим пример разработки для калькулятора.

@calculator

Feature: Sum

As a math idiot

I want to be told the sum of two numbers

So that I can avoid silly mistakes

@positive @sprint1

Scenario: Add two numbers

Given I have entered 50 into the calculator

And I have entered 70 into the calculator

When I press add

Then the result should be 120 on the screen

Стоит обратить внимание на @атрибуты. У них есть несколько важных свойств. Первое, — если вы используете NUnit, SpecFlow добавить атрибут *[NUnit.Framework.CategoryAttribute(«calculator»)]*. Это очень удобно для составления тест-планов. Деление по категориями поддерживают R#, родной NUnit Runner и Team City.

Давайте автоматизируем этот сценарий. Интерфейс калькулятора имеет следующий вид:

public interface ICalculator

{

decimal Sum(params decimal[] values);

decimal Minus(decimal a, decimal b);

decimal Sin(decimal a);

decimal Multiply(params decimal[] values);

decimal Divide(decimal a, decimal b);

}

Добавим сервисный контекст тестирования:

public class CalculationContext

{

private readonly List<decimal> \_values = new List<decimal>();

public ICalculator Calculator { get; private set; }

public decimal Result { get; set; }

public Exception Exception { get; set; }

public List<decimal> Values

{

get { return \_values; }

}

public CalculationContext()

{

Calculator = new Calculator();

}

}

Для автоматизации шагов SpecFlow использует специальные атрибуты:

[Binding]

public class Sum : CalcStepsBase

{

public CalculationContext Context {get;set;}

public Sum(CalculationContext context)

{

Context = CalculationContext();

}

[Given("I have entered (.\*) into the calculator")]

public void Enter(int digit)

{

Context.Values.Add(digit);

}

[When("I press (.\*)")]

public void Press(string action)

{

switch (action.ToLower())

{

case "add":

case "plus":

Context.Result = Context.Calculator.Sum(Context.Values.ToArray());

break;

default: throw new InconclusiveException(string.Format("Action \"{0}\" is not implemented", action));

}

}

[Then("the result should be (.\*) on the screen")]

public void Result(decimal expected)

{

Assert.AreEqual(expected, Context.Result);

}

}

В этом подходе есть несколько преимуществ:

1. Каждый шаг нужно автоматизировать лишь однажды
2. Вы избегаете проблемы со сложными цепочками наследования, код выглядит гораздо понятнее
3. Атрибуты используют регулярные выражения, так один атрибут может «поймать» несколько шагов. Атрибут When, в данном случае, отработает для фразы «add» и «plus»

#### Альтернативная запись

@positive

Scenario: Paste numbers

Given I have entered two numbers

| a | b |

| 1 | 2 |

When I press add

Then the result should be 3 on the screen

[Given("I have entered two numbers")]

public void Paste(Table values)

{

var calcRow = values.CreateInstance<CalcTable>();

Context.Values.Add(calcRow.A);

Context.Values.Add(calcRow.B);

}

public class CalcTable

{

public decimal A { get; set; }

public decimal B { get; set; }

}

## Такой вариант записи может быть удобен, когда нужно заполнить большой объект. Например, данные аккаунта пользователя.

## Конечно, одного теста недостаточно, писать так десятки сценариев для разных чисел удовольствие сомнительное. На помощь приходит *Scenario Outline:*

**@calculator**

**Feature**: Calculations

As a math idiot

I want to be told the calculation result of two numbers

So that I can avoid silly mistakes

**@positive** **@b12** **@tc34**

**Scenario** **Outline**: Add two numbers

**Given** I have entered <firstValue> into the calculator

**And** I have entered <secondValue> into the calculator

**When** I press <action>

**Then** the <result> should be on the screen

**Examples**:

| firstValue | secondValue | action | result |

| 1 | 2 | plus | 3 |

| 2 | 3 | minus | -1 |

| 2 | 2 | multiply | 4 |

SpecFlow подставит значения из таблицы в *плейсхолдер (заглушку)*. Уже неплохо, но нужно еще дописать автоматизацию:

[When("I press (.\*)")]

public void Press(string action)

{

switch (action.ToLower())

{

case "add":

case "plus":

Context.Result = Context.Calculator.Sum(Context.Values.ToArray())*;*

break*;*

case "minus":

Context.Result = Context.Calculator.Minus(Context.Values[0], Context.Values[1])*;*

break*;*

case "multiply":

Context.Result = Context.Calculator.Multiply(Context.Values.ToArray())*;*

break*;*

case "sin":

Context.Result = Context.Calculator.Sin(Context.Values[0])*;*

break*;*

default: throw new InconclusiveException(string.Format("Action \"{0}\" is not implemented", action))*;*

}

}

[Then("the result should be (.\*) on the screen")]

[Then("the (.\*) should be on the screen")]

public void Result(decimal expected)

{

Assert.AreEqual(expected, Context.Result)*;*

}

Мы изменили для лучшей читаемости вторую строку. Поэтому на метод Result нужно повесить второй атрибут. На выходе вы получите 3 теста с отчетами вида:

Given I have entered 1 into the calculator

-> done: Sum.Enter(1) (0,0s)

And I have entered 2 into the calculator

-> done: Sum.Enter(2) (0,0s)

When I press plus

-> done: Sum.Press("plus") (0,0s)

Then the result should be 3 on the screen

-> done: Sum.Result(3) (0,0s)

**Порядок выполнения лабораторной работы**

1. Связать части калькулятора (Логика, Прослойка, Графический интерфейс воедино).
2. Разработать тесты на основе исполняемой спецификации для проверки работоспособности всей программы в целом.

Инструкции по работе с тестами:

<http://www.specflow.org/getting-started/>

<https://habrahabr.ru/post/182160/>

<http://toolsqa.com/specflow/set-up-specflow/>

Синтаксис gherkin:

http://docs.behat.org/en/v2.5/guides/1.gherkin.html

**Ссылки:**

1. <http://ojjjk.blogspot.ru/2013/03/c-keyword-driven-behaviour-driven.html>
2. <https://butaji.wordpress.com/2010/02/04/%D0%BF%D0%BE%D0%BB%D0%BD%D0%BE%D1%86%D0%B5%D0%BD%D0%BD%D1%8B%D0%B9-behavior-driven-development-%D0%BD%D0%B0-net/>
3. <http://www.oszone.net/22523/SpecFlow>
4. <https://habrahabr.ru/post/182032/>
5. <https://habrahabr.ru/post/166747/>
6. <https://habrahabr.ru/post/268561/>
7. <https://msdn.microsoft.com/magazine/gg490346>