Занятие 6. Модульное тестирование (unit testing)

Тренер: Алексей Дышлевой

Основные вопросы

- Вступ до модульного тестування
- Параметризовані тести
- Data-Driven тести
- Створення модульних тестів за допомогою MS Visual Studio

Модульное тестирование

- Модульное тестирование (unit testing) – изолированная проверка каждого отдельного элемента путем запуска тестов в искусственной среде
- Unit (элемент) – наименьший компонент, который можно скомпилировать
- Драйверы – модули тестов, которые запускают тестируемый элемент
- Заглушки – заменяют недостающие компоненты, которые вызываются элементом. Выполняют следующие действия:
  - Возвращаются к элементу, не выполняя никаких других действий
  - Отображают трассировочные сообщения и иногда предлагают тестировщику продолжить тестирование
  - Возвращают постоянное значение или предлагают тестировщику самому ввести возвращаемое значение
  - Осуществляют простую реализацию недостающей компоненты
  - Имитируют исключительные или аварийные условия

Подходы к unit-тестированию

- White-box. Для конструирования тестов используются внутренняя структура кода и управляющая логика. При этом существует вероятность, что код будет проверятся так, как он был написан, а это не гарантирует корректность логики
- Black-box. Для конструирования тестов используются требования и спецификации ПО
- Недостатки:
- Невозможно найти взаимоуничтожающиеся ошибки
- Некоторые ошибки возникают редко (работа с памятью), их трудно найти и воспроизвести

Стратегия модульного тестирования

- Модульное тестирование является одной из ключевых практик методологии экстремального программирования:
  - Написание тестов помогает войти в рабочий ритм
  - Придает уверенности в работоспособности кода
  - Дает запас прочности при дальнейшей интеграции или изменениях кода
- Модульное тестирование оправдано, если оно:
  - Снижает время на отладку
  - Дает возможность поиска ошибок с меньшими затратами, чем при других подходах
  - Дает возможность дешевого поиска ошибок при изменениях кода в будущем

Цель модульного тестирования

- Получение работоспособного кода с наименьшими затратами
- Его применение оправдано тогда, когда оно дает больший эффект, чем другие методы

Следствие

- Нет смысла писать тесты на весь код. Некоторые ошибки проще найти на более поздних стадиях разработки. Например, нет смысла писать тесты на класс, который используется только одним классом. Эффективней написать тест на вызывающий класс и создать тести, тестирующие все участки кода
- Писать тести для кода потенциально подверженного изменениям более выгодно, чем для кода, изменения которого не предполагаются. Сложная логика меняется чаще, чем простая. В первую очередь имеет смысл писать модульные тесты на сложную логику, а на простую – позже (или тестировать другими методами).
- Для того, чтобы как можно реже менять тесты следует хорошо планировать интерфейсы.

Реальность

- Есть 3 типа проектов:
  - Без покрытия тестами
  - С тестами, которые никто не запускает и не поддерживает
  - С серьезным покрытием. Все тесты проходят

Почему есть проекты 2 типа?

- Бездумное написание тестов **не только не помогает, но вредит проекту**. Если раньше был один некачественный продукт, то написав тесты, не разобравшись в этой теме, вы получите два. И удвоенное время на сопровождение и поддержку.

Еще несколько рекомендаций

- Ваши тесты должны:
  - Быть достоверными
  - Не зависеть от окружения, на котором они выполняются
  - Легко поддерживаться
  - Легко читаться и быть простыми для понимания (даже новый разработчик должен понять **что именно** тестируется)
  - Соблюдать единую конвенцию именования
  - Запускаться регулярно в автоматическом режиме

Планирование тестов

- Код с не оттестированными участками не может быть опубликован
- Тесты должны базироваться на спецификации
- На каждое требование должен быть, как минимум, один тест (ручной или автоматический)
- Любые простые тесты также нужны, несмотря на малую вероятность возникновения ошибки. Цена пропущенной ошибки очень высока
- Наиболее эффективный способ создания тестового набора – совместное использование методов черного и белого ящиков

Стиль написания теста

- Отлично зарекомендовал себя подход AAA *(arrange, act, assert)*
- class CalculatorTests
- {
- public void Sum_2Plus5_7Returned()
- {
- // arrange
- var calc = new Calculator();
- // act
- var res = calc.Sum(2,5);
- // assert
- Assert.AreEqual(7, res);
- }
- }

Принцип тестирования

- **Каждый тест должен проверять только одну вещь.** Если процесс слишком сложен (например, покупка в интернет магазине), разделите его на несколько частей и протестируйте их отдельно.
  Если вы не будете придерживаться этого правила, ваши тесты станут нечитаемыми, и вскоре вам окажется очень сложно их поддерживать.

Тестирование состояния

- Пример:
- Установим датчики, которые будут засекать, когда полив начался и закончился, и сколько воды поступило из системы.

Поддержка тестов

- **Тесты - такой-же код.** Разница только в том, что у тестов другая цель – обеспечить качество вашего приложения. Все принципы, применямые в разработке продакшн-кода могут и должны применяться при написании тестов.
- Есть всего три причины, почему тест перестал проходить:

  Ошибка в продакшн-коде: это баг, его нужно завести в баг-трекере и починить.
- Баг в тесте: видимо, продакшн-код изменился, а тест написан с ошибкой (например, тестирует слишком много или не то, что было нужно). Возможно, что раньше он проходил ошибочно. Разберитесь и почините тест.
- Смена требований. Если требования изменились слишком сильно – тест должен упасть. Это правильно и нормально. Вам нужно разобраться с новыми требованиями и исправить тест. Или удалить, если он больше не актуален.

Тестирование поведения

- Пример:
- Запускаем цикл (12 часов). И через 12 часов проверяем, хорошо ли политы растения, достаточно ли воды, каково состояние почвы и т.д.

TDD

- TDD (test-driven development) – методика разработки, позволяющая оптимизировать использование модульных тестов. Речь идет об оптимальном, а не максимальном применении.
- Задача, которую преследует TDD – достижение баланса между усилиями и результатом
- Основа TDD – цикл «red/ green/ refactor»:
- В первой фазе программист пишет тест
- Во второй – код, необходимый для того, чтоб тест работал
- В третьей, при необходимости, производится рефакторинг
- В соответствии с принципом «Test First», следует писать только такой код, который абсолютно необходим, чтобы тесты выполнялись успешно

Пример – создание теста

- Создание тестового проекта

Пример – класс-тест

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
namespace UnitTestProject1
{
[TestClass]
public class UnitTest1
{
[TestMethod, ExpectedException(typeof(ArgumentException))]
public void TestMethod1()
{
TDDExample.FactorialClass.Factorial(-1);
}
}
}
```

Пример – тестируемый класс

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Numerics;
namespace TDDExample
{
public class FactorialClass
{
public static BigInteger Factorial(int number)
{
if (number < 0) throw new ArgumentException("Argument is Less than Zero.");
if (number == 0) return 1;
else return Factorial(number - 1) * number;
}
}
}
```

Пример – запуск теста

Пример – тест с другими данными

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
namespace UnitTestProject1
{
[TestClass]
public class UnitTest1 {
[TestMethod, ExpectedException(typeof(ArgumentException))]
public void TestMethod1()
{
TDDExample.FactorialClass.Factorial(10);
}
}
}
```

Класс Assert

- Проверяет условия, использующие утверждения «истина/ ложь» в процессе выполнения модульных тестов
- Microsoft.VisualStudio.TestTools.UnitTesting
- Методы:
- AreEqual, AreEqual<T> / AreNotEqual, AreNotEqual<T> + 17 перегрузок – проверяет 2 указанных объекта на равенство/ неравенство. Утверждение не выполняется, если объекты равны
- AreSame / AreNotSame + 2 перегрузки – проверяет, ссылаются ли две указанные объектные переменные на один и тот же объект/ разные объекты
- Equals – определяет равенство объектов

Методы Assert

- Fail – отменяет выполнение утверждения без проверки каких-либо условий
- Inconclusive – указывает, что утверждение не может быть проверено
- IsFalse / IsTrue – проверяет, имеет ли указанное условие значение false/true
- IsInstanceOfType / IsNotInstanceOfType – проверяет, является / не является ли объект экземпляром заданного типа
- IsNull / IsNotNull – проверяет, имеет/ не имеет ли указанный объект значение null
- ReplaceNullChars – заменяет в строке символы null ('\0') на "\\0"

Пример – тестируемый класс

```
public class FactorialClass
{
public static BigInteger Factorial(int number)
{
if (number < 0) throw new
ArgumentException("Argument is Less than Zero.");
if (number == 0) return 1;
else return Factorial(number - 1) * number;
}
public static double CircleSquare(double radius)
{
return 3.14 * radius * radius;
}
public static double CircleLength(double radius)
{
return 2.0 * 3.14 * radius;
}
}
```

Пример – класс-тест

```
[TestClass]
public class UnitTest1 {
[TestMethod, ExpectedException(typeof(ArgumentException))]
public void TestFactorial() {
TDDExample.FactorialClass.Factorial(-100);
}
[TestMethod]
public void TestCircleSquare() {
double testRadius = 10;
double expectedResult = Math.PI * Math.Pow(testRadius, 2);
double actualResult = TDDExample.FactorialClass.CircleSquare(testRadius);
double tolerance=0.01;
Assert.AreEqual(expectedResult, actualResult, tolerance);
}
[TestMethod]
public void TestCircleLength() {
double testRadius = 10;
double expectedResult = 2.0 * Math.PI * testRadius;
double actualResult = TDDExample.FactorialClass.CircleLength(testRadius);
double tolerance = 0.01;
Assert.AreEqual(expectedResult, actualResult, tolerance);
}
}
```

Параметризованные тесты

- В MSTest нет возможности создавать параметризированные тесты.
- Пример на Nunit
- [TestCase("(-Inf,+Inf)", Result = true)]
- [TestCase("[0,1.5)", Result = true)]
- [TestCase("(0,0)", Result = false)]
- [TestCase("[0,-1.12)", Result = false)]
- [TestCase("(Inf,-Inf]", Result = false)]
- [TestCase(null, Result = false, ExpectedException = typeof(ArgumentNullException))]
- public bool

```
Test_Interval_Validity(string range)
 {
    try
    {
      Interval.Parse(range);
      return true;
    }
    catch (FormatException)
    {
      return false;
    }
}
```

Data-Driven тесты

- Используется некоторый источник данных (практический любой файл) для проверки теста на данных из источника
- Используется специальный атрибут (пример таблицы Excel)
- [DataSource(dataDriver, connectionStr, "Shit1$", DataAccessMethod.Sequential)]

- В классе следует прописать:
- const string dataDriver = "System.Data.OleDb"; const string connectionStr = "Provider=Microsoft.ACE.OLEDB.12.0;Data Source=C:\\matrix.xlsx;Extended Properties=\"Excel 12.0 Xml;HDR=YES\";";

Задача

- Создать тесты для программы из домашнего задания 3

Задача

- Разработать приложение-калькулятор, используя подход TDD

Домашнее задание

- Написать модульные тесты для предыдущего домашнего задания (4.1, 4.2, 5.1, 5.2)