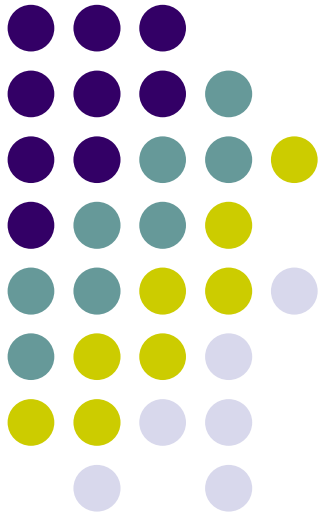
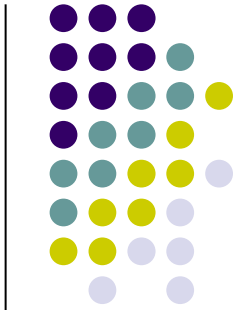
Модульное тестирование с помощью библиотеки NUnit



Содержание



- Мифы о тестировании
- Модульное тестирование с помощью NUnit
- Рекомендации к написанию тестов
- Полезная информация о C#

Введение

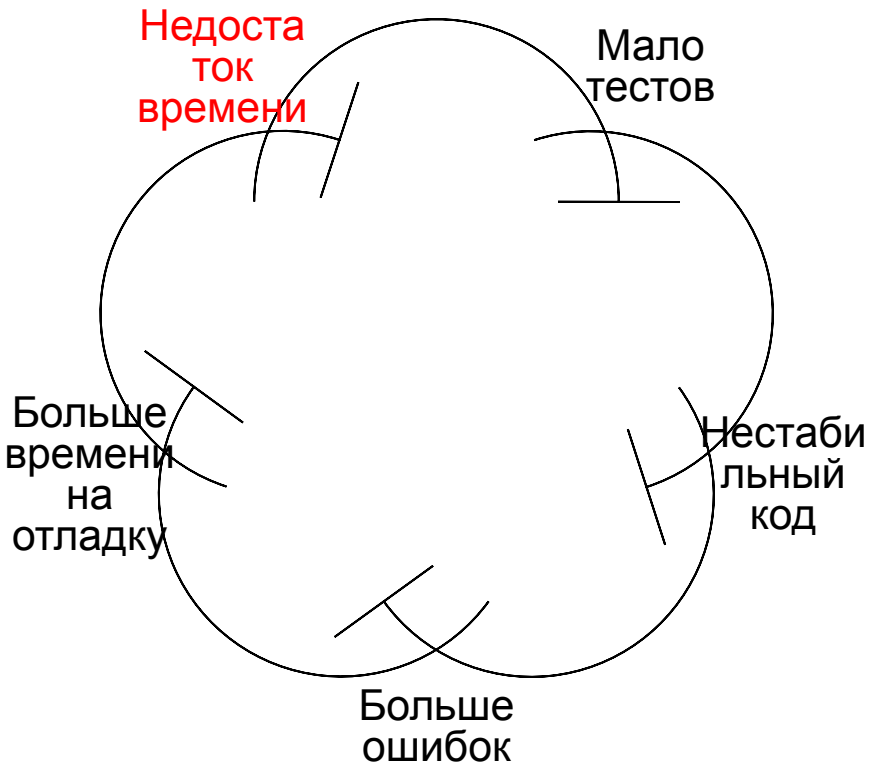
- Что такое модульное тестирование?
  - Тестирование отдельных функций системы.
  - Как правило выполняется разработчиком модуля.
  - Может быть легко автоматизировано.
  - Закладывает основу для регрессионного тестирования приложения.

Мифы о тестировании

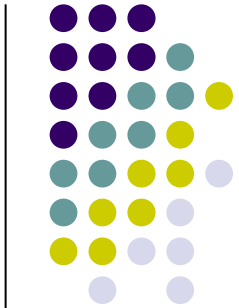
- «У меня нет времени на тесты».
- «Тестирование – скучное и не творческое занятие».
- «Мой код и так отлично работает».
- «Это работа для отдела тестирования. У них получится лучше».

Миф №1
«У меня нет времени на тесты»

Написание тестов стабилизирует код и позволяет существенно сократить время отладки.



Пример программы



Пусть есть класс, реализующий математические функции.

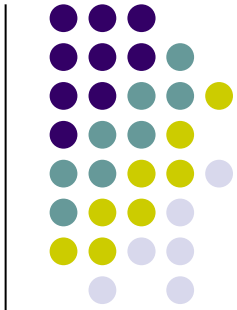
```
public class Calculator {
  public static int Sum(int x, int y){
    return x+y; //возвращает результат сложения двух чисел
  }
}
```

Вариант модульного тестирования №1

Некоторые проверки можно поместить в сам класс.

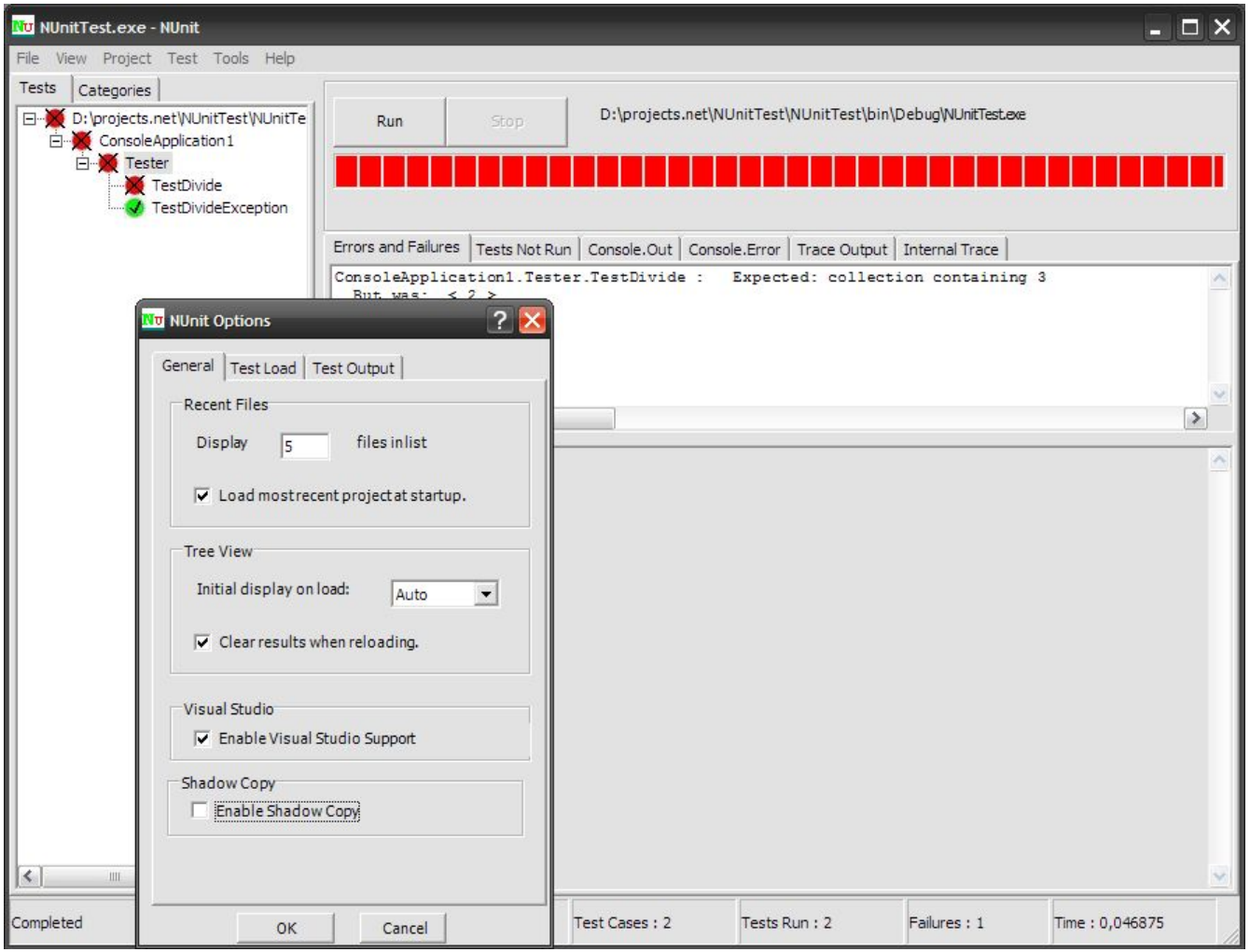
```
public class Calculator {
  public static void main(String[] args) {
    if (Sum(1, 3) == 4) { //проверяем, что при сложении 1 и
3             //нам возвращается 4
      Console.WriteLine("Test1 passed.");
    } else {Console.WriteLine("Test1 failed.");
  }
}
```

Наблюдения

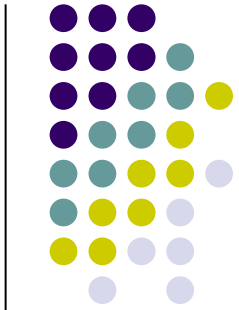


- Тесты неудобно хранить в самой программе.
- Выход - внешняя библиотека, подключенная к проекту
- Часто используемые библиотеки модульного тестирования: NUnit, CUnit, JUnit, …

Библиотека Nunit

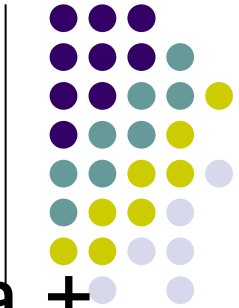


Пример тестового класса



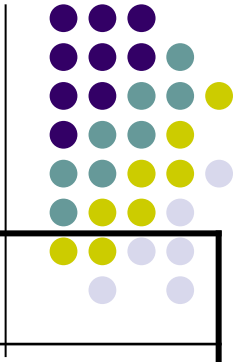
```
using System;
using NUnit.Framework;

[TestFixture]
public class LargestTest
{
  [Test]
  public void LargestOfNumber()
  {
  Assert.Greater(2, 1);
  }
}
```

Тестовый проект



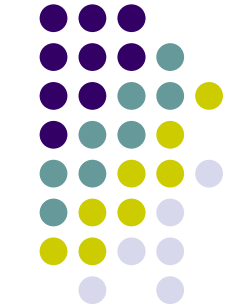
- Как правило, имя_тестируемого_проекта + "Test“ (н-р CalculatorTest.dll)
- Тот же солюшен, что и тестируемый проект
- Разметка атрибутами NUnit
  - TestFixture
  - TestFixtureSetUp
  - SetUp
  - Test
  - TearDown
  - TestFixtureTearDown

Атрибуты NUnit



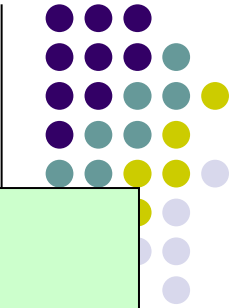
Атрибут	Значение
Test	тестовый метод
TestFixture	атрибут класса, содержащего тесты
TestFixtureSetUp	функция, выполняющаяся один раз при создании класса помеченного TestFixture
SetUp	функция, выполняющаяся перед каждым тестом
TearDown	функция, выполняющаяся после исполнения каждого теста
TestFixtureTearDown	функция, выполняющаяся один раз при удалении класса помеченного TestFixture
Ignore	временное отключение теста
ExpectedException	ожидаемый тип исключения для проверки "некорректных" данных

Проверка условий (Assert)



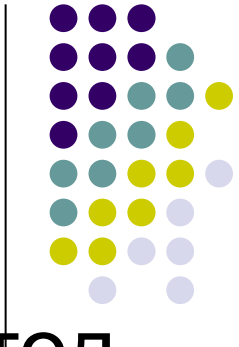
- Класс Assert
  - **Assert.AreEqual** – эквивалентны ли 2 параметра метода (пожалуй, самый популярный ассёрт)
  - **IsTrue (IsFalse)** – является ли выражение истинным (ложным)
  - **IsNull (IsNotNull)** – является ли параметр null
  - **Contains** – для коллекций
  - и некоторые другие, которые редко имеет смысл использовать

Использование NUnit



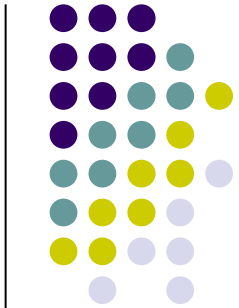
```
//подключение библиотеки
using NUnit.Framework;
//Тест должен быть помечен атрибутом [TestFixture]
[TestFixture]
public class CalculatorTest {
//тестирующие методы помечены атрибутом [Test] и начинаются с
префикса “Test”
[Test]
 public void TestSum() {
    int x = 3;
         int y = 5;
         int expResult = 8;
         int result = Calculator.Sum(x, y);
         //проверка условия на совпадение
         Assert.AreEqual(expResult, result);
    //Assert.AreEquals(Calculator.Sum(3, 5));
}
```

Причины ошибки тестов



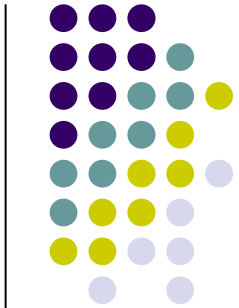
- Неправильно работает тестируемый метод.
- Методы, вызываемые из тестируемого, генерируют исключение по каким-то причинам.
- Некорректно написан тест.

Рекомендации к написанию тестов



- Название тестового метода.
- Размер теста.
- Ожидаемый результат.
- Тестовые данные.

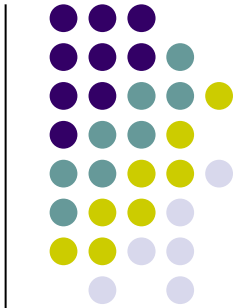
Название тестового метода



- Имя теста должно описывать:
  - Тестируемую функциональность.
  - Возможно, условия тестирования.

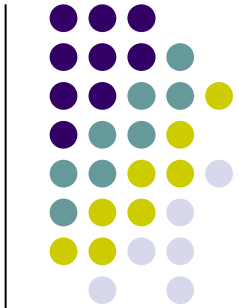
Непонятно	Понятно
Test1	TestAddUser
TestAddUserThrowException	TestAddUserWithoutPassword

Размер теста



- Тестовый метод должен быть коротким.
  - Дополнительные проверки -> вспомогательные методы.
- Количество проверок (assert) должно быть минимальным.
  - Иначе сложно будет найти причину ошибки.
- Каждый тест - одна единица бизнес-логики:
  - Простой метод.
  - Один из исходов конструкции if..else.
  - Один из случаев (case) блока switch.
  - Исключение, обрабатываемое блоком try…catch
  - Исключение, генерируемое (throw) в методе.

Ожидаемый результат



- Ожидаемый результат должен быть константой.
- Не следует в тесте повторять тестируемую логику, подсчитывая результат.

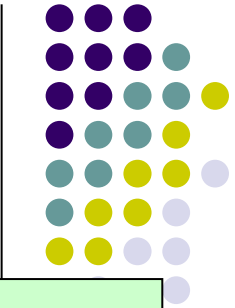
```
public void TestBalance1() {
    Account account = new Account();
    account.Deposit(10);
    account.Withdraw(5);
    account.Deposit(6);
     int expectedBalance = 11; //правильно
    Assert.AreEqual(expectedBalance, account.Balance);
}

public void TestBalance2() {
    Account account = new Account();
    account.Deposit(10);
    account.Withdraw(5);
    account.Deposit(6);
    int expectedBalance = 10 - 5 + 6; //неправильно
    Assert.AreEqual(expectedBalance, account.Balance);
}
```

Тестовые данные

- Тестовые данные и ожидаемый результат должны быть рядом.
- Если объект user удобнее создавать вне тестового метода, следует использовать именованные константы.

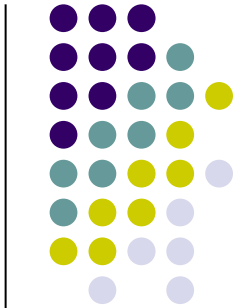
```
public void TestIsPasswordValid() {
   Assert.IsTrue(user.IsPasswordValid("abcdef"));
//понять,правильно ли написан тест, можно лишь отыскав
   где создается объект user
  Assert.IsFalse(user.IsPasswordValid("123456"));
}
```

Тестовые данные (cont.)



```
public void TestIsPasswordValid() {
    User user = new User("Name", "abcdef");
 Assert.IsTrue(user.IsPasswordValid("abcdef"));
    //здесь все понятно
 Assert.IsFalse(user.IsPasswordValid("123456"));
}
```

```
public void TestIsPasswordValid() {
   Assert.IsTrue(user.IsPasswordValid(validPassword));
 //здесь тоже
   Assert.IsFalse(user.IsPasswordValid(wrongPassword));
}
```



Вопросы?