# MSUnit

using System;

using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace \_003\_Calculator\_MSUnit

{

// Класс для тестирования калькулятора.

[TestClass] // TestClass - указывает, что класс содержит тестовый код.

public class MSCalculatorTest

{

[TestMethod] // TestMethod - указывает, что метод содержит тестовый код.

public static void MSTestOperations()

{

Calculator c = new Calculator();

Assert.AreEqual(8, c.Add(3, 5)); // 3 + 5 = 8.

Assert.AreEqual(4, c.Sub(6, 2)); // 6 - 2 = 4.

Assert.AreEqual(24, c.Mul(8, 3)); // 8 \* 3 = 24.

Assert.AreEqual(2, c.Div(6, 3)); // 6 / 3 = 2.

}

}

}

# NUunit AsserEqual

namespace Calculator

{

// Класс для тестирования калькулятора.

[TestFixture] // TestFixture - указывает, что класс содержит тестовый код.

class CalculatorTest

{

[Test] // Test - указывает, что метод содержит тестовый код.

public static void TestOperations()

{

Calculator c = new Calculator();

Assert.AreEqual(8, c.Add(3, 5)); // 3 + 5 = 8.

Assert.AreEqual(4, c.Sub(6, 2)); // 6 - 2 = 4.

// Assert.AreEqual(24, c.Mul(8, 3)); // 8 \* 3 = 24.

Assert.AreEqual(2, c.Div(6, 3)); // 6 / 3 = 2.

Assert.AreNotEqual(9, c.Add(3, 5)); // 3 + 5 != 9.

}

}

}

# NUnit Asser Methods

namespace \_004\_Assert

{

class Program

{

static void Main()

{

}

}

[TestFixture]

class MyClassTest

{

[Test]

public void AreSame()

{

string a = "hello";

string b = "world!";

a = b;

// AreSame - проверяет, ссылаются ли переменные на одну и ту же область памяти.

Assert.AreSame(a, b);

}

}

[TestFixture]

class ContainsTest

{

[Test]

public void Contains()

{

ArrayList array = new ArrayList();

array.Add("Alex");

array.Add("Serg");

array.Add("John");

// Contains используется для проверки объектов,

// содержащихся в коллекции или массиве.

Assert.Contains("Alex", array);

}

}

[TestFixture]

class ComparisonsTest

{

private int a, b;

[SetUp]

public void Init()

{

a = 10;

b = 20;

}

[Test]

public void Greater()

{

// Проверяет, является ли один объект больше, чем другой (a > b).

Assert.Greater(a, b);

// a >= b.

Assert.GreaterOrEqual(a, b);

}

[Test]

public void Less()

{

// Проверяет, является ли один объект меньше, чем другой (a < b).

Assert.Less(a, b);

// a <= b.

Assert.LessOrEqual(a, b);

}

}

[TestFixture]

class TypeAssertsTest

{

[Test]

public void InInstanceOf()

{

// Методы для проверки типов объектов.

Assert.IsInstanceOf(typeof(object), "Hello");

Assert.IsNotInstanceOf(typeof(string), 5);

}

}

[TestFixture]

class ConditionTest

{

Calculator calc;

[SetUp]

public void Init()

{

calc = new Calculator();

}

[Test]

public void IsTrue()

{

// 2 + 2 = 4 - True.

Assert.IsTrue(calc.Add(2, 2) == 4);

}

[Test]

public void IsFalse()

{

// 3 \* 6 = 20 - False.

Assert.IsFalse(calc.Mul(3, 6) == 20);

}

[Test]

public void IsNan()

{

double d = double.NaN;

Assert.IsNaN(d); // d is not a number.

Assert.IsNaN(0 / 0f); // 0/0f is not a number.

}

[Test]

public void IsEmpty()

{

Assert.IsEmpty("");

Assert.IsNotEmpty("Hello!");

Assert.IsEmpty(new ArrayList());

Assert.IsNotEmpty(new List<string> { "one", "two" });

}

}

[TestFixture]

class UtilityMethodsTest

{

[Test]

public void Fail()

{

// Метод создает тест, который не будет пройден с сообщением Fail!.

Assert.Fail("Fail!");

}

[Test]

public void Ignore()

{

throw new Exception();

// Этот метод помечает тест, который будет игнорирован при запуске всех тестов

// и не будет выполнен.

Assert.Ignore("Ignore");

}

}

}

# NUnit Is

namespace \_010\_AssertSyntaxTests

{

// AssertionHelper - класс для использования унаследованного синтаксиса

[TestFixture]

public class AreEqualSyntax: AssertionHelper

{

[Test]

public void ClassicAreEqual()

{

// Classic syntax workarounds

Assert.AreEqual(typeof(string), "Hello".GetType());

Assert.AreEqual("System.String", "Hello".GetType().FullName);

Assert.AreNotEqual(typeof(int), "Hello".GetType());

Assert.AreNotEqual("System.Int32", "Hello".GetType().FullName);

}

[Test]

public void HelperAreEqual()

{

// Helper syntax

Assert.That("Hello", Is.TypeOf(typeof(string)));

Assert.That("Hello", Is.Not.TypeOf(typeof(int)));

}

[Test]

public void InheritedAreEqual()

{

// Inherited syntax

Expect("Hello", TypeOf(typeof(string)));

Expect("Hello", Not.TypeOf(typeof(int)));

}

}

}

# NUnit Contains

namespace \_010\_AssertSyntaxTests

{

[TestFixture]

class ContainsSyntax : AssertionHelper

{

[Test]

public void ClassicContains()

{

int[] iarray = new int[] { 1, 2, 3 };

string[] sarray = new string[] { "a", "b", "c" };

// Classic syntax

Assert.Contains(3, iarray);

Assert.Contains("b", sarray);

}

[Test]

public void HelperContains()

{

int[] iarray = new int[] { 1, 2, 3 };

string[] sarray = new string[] { "a", "b", "c" };

// Helper syntax

Assert.That(iarray, Has.Member(3));

Assert.That(sarray, Has.No.Member("x"));

}

[Test]

public void InheritedContains()

{

int[] iarray = new int[] { 1, 2, 3 };

string[] sarray = new string[] { "a", "b", "c" };

// Inherited syntax

Expect(iarray, Contains(3));

Expect(sarray, Contains("b"));

Expect(sarray, Not.Contains("x"));

}

}

}

# DI Container

using System;

using System.Collections.Generic;

using System.IO;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace \_005\_DI\_Container

{

class FileDataObject : IDataAccessObject

{

public List<string> GetFiles()

{

string path = Directory.GetCurrentDirectory();

List<string> list = new List<string>();

DirectoryInfo d = new DirectoryInfo(path);

FileInfo[] files = d.GetFiles();

foreach (var file in files)

{

list.Add(file.Name);

}

return list;

}

}

}

using Microsoft.Practices.Unity;

namespace \_005\_DI\_Container

{

class FileManager

{

public FileManager()

{

}

// Свойство, через которое будет внедрена зависимость.

[Dependency]

public IDataAccessObject DataAccessObject { get; set; }

public bool FindLogFile(string fileName)

{

if (DataAccessObject == null)

{

throw new ArgumentNullException("dataAccessObject", "Parameter dataAcessObject cannot be null");

}

List<string> files = DataAccessObject.GetFiles();

foreach (var file in files)

{

if (file.Contains(fileName))

{

return true;

}

}

return false;

}

}

}

amespace \_005\_DI\_Container

{

[TestFixture]

class FileManagerTest

{

[Test]

public void FindLogFileTest5()

{

// Создание DI контейнера.

IUnityContainer container = new UnityContainer();

// Регистрирование типов интерфейсов и классов, которые их реализуют для инстансирования зависимостей.

container.RegisterType<IDataAccessObject, StubFileDataObject>();

//container.RegisterType<IDataAccessObject, FileDataObject>();

// Создание объекта типа FileManager и внедрение всех зависимостей

//в соответствии с зарегистрированными в контейнере типами.

FileManager manager = container.Resolve<FileManager>();

Assert.IsTrue(manager.FindLogFile("file2.log"));

}

}

}

namespace \_005\_DI\_Container

{

public interface IDataAccessObject

{

List<string> GetFiles();

}

}

namespace \_005\_DI\_Container

{

class StubFileDataObject : IDataAccessObject

{

public List<string> GetFiles()

{

List<string> list = new List<string>();

list.Add("file1.txt");

list.Add("file2.log");

list.Add("file3.exe");

return list;

}

}

}

namespace \_005\_DI\_Container

{

class Program

{

static void Main()

{

string fileName = "file2.log";

// Создания DI контейнера.

IUnityContainer container = new UnityContainer();

// Регистрирование типов интерфейсов и классов, которые их реализуют для инстансирования зависимостей.

container.RegisterType<IDataAccessObject, FileDataObject>();

//container.RegisterType<IDataAccessObject, StubFileDataObject>();

// Создание объекта типа FileManager и внедрение всех зависимостей

//в соответствии с зарегистрированными в контейнере типами.

FileManager manager = container.Resolve<FileManager>();

if (manager.FindLogFile(fileName))

{

Console.WriteLine("File {0} is found!", fileName);

}

else

{

Console.WriteLine("File {0} is not found!", fileName);

}

// или так

FileManager manager2 = new FileManager();

container.BuildUp<FileManager>(manager2);

if (manager2.FindLogFile("file2.log"))

{

Console.WriteLine("File {0} is found!", fileName);

}

else

{

Console.WriteLine("File {0} is not found!", fileName);

}

Console.ReadKey();

}

}

}

# Ninject

using Ninject.Modules;

namespace \_006\_Ninject

{

class ConfigFileObjectData : NinjectModule

{

public override void Load()

{

//this.Bind<IDataAccessObject>().To<FileDataObject>();

this.Bind<IDataAccessObject>().To<StubFileDataObject>();

this.Bind<FileManager>().ToSelf();

}

}

}

namespace \_006\_Ninject

{

class FileDataObject : IDataAccessObject

{

public List<string> GetFiles()

{

string path = Directory.GetCurrentDirectory();

List<string> list = new List<string>();

DirectoryInfo d = new DirectoryInfo(path);

FileInfo[] files = d.GetFiles();

foreach (var file in files)

{

list.Add(file.Name);

}

return list;

}

}

}

namespace \_006\_Ninject

{

class FileManager

{

IDataAccessObject dataAccessObject;

public FileManager()

{

}

public FileManager(IDataAccessObject dataAccessObject)

{

this.dataAccessObject = dataAccessObject;

}

public bool FindLogFile(string fileName)

{

List<string> files = dataAccessObject.GetFiles();

foreach (var file in files)

{

if (file.Contains(fileName))

{

return true;

}

}

return false;

}

}

}

using Ninject;

using NUnit.Framework;

namespace \_006\_Ninject

{

[TestFixture]

class FileManagerTest

{

[Test]

public void FindLogFileTest6()

{

// Ninject Initialization

IKernel ninjectKernel = new StandardKernel(new ConfigFileObjectData());

FileManager manager = ninjectKernel.Get<FileManager>();

Assert.IsTrue(manager.FindLogFile("file2.log"));

}

}

}

namespace \_006\_Ninject

{

interface IDataAccessObject

{

List<string> GetFiles();

}

}

namespace \_006\_Ninject

{

class StubFileDataObject : IDataAccessObject

{

public List<string> GetFiles()

{

List<string> list = new List<string>();

list.Add("file1.txt");

list.Add("file2.log");

list.Add("file3.exe");

return list;

}

}

}

namespace \_006\_Ninject

{

class Program

{

static void Main()

{

string fileName = "file2.log";

FileManager mgr = new FileManager(new FileDataObject());

if (mgr.FindLogFile(fileName))

{

Console.WriteLine("File {0} is found!", fileName);

}

else

{

Console.WriteLine("File {0} is not found!", fileName);

}

Console.ReadKey();

}

}

}

# Mock

* Модульные тесты можно условно разделить на две группы:
  1. **Тесты состояния** (state-based tests) — тесты, проверяющие что вызываемый метод объекта отработал корректно, проверяя состояние тестируемого объекта после вызова метода. State based tests базируются на Stub-объектах. При State-Based тестах как правило используется класс Assert.
  2. **Тесты взаимодействия** (interaction tests) — это тесты, в которых тестируемый объект производит манипуляции с другими объектами. Применяются, когда требуется удостовериться, что тестируемый объект корректно взаимодействует с другими объектами. Interaction tests базируются на Mock-объектах. При Interaction тестах как правило пользуются Isolation Frameworks.
* **Mock-объект** – это управляемая замена существующих зависимостей в системе. Mock-объекты заменяют реальные объекты системы и позволяют проверить вызовы своих членов тестируемым классом. Mock-объекты отличаются от Stub объектов тем, что они могут быть причиной не прохождения теста.
* Как правило, если один тест использует больше одного Mock объекта он одновременно тестирует несколько вещей. Поэтому в одном тесте рекомендуется использовать не больше одного Mock объекта и несколько Stub объектов.
* Проблемы созданных вручную Stub и Mock объектов:
  1. Сложно создать Fake объект для интерфейса, имеющего много членов.
  2. Создание Fake объектов порой занимает много времени.
  3. Приходится писать много кода для сохранения состояния часто вызываемого Mock-объекта.
  4. Для проверки всех параметров вызываемых методов приходится делать множество проверок.
  5. Проблема с использованием созданных Mock и Stub объектов в других тестах
* **Isolation Framework** – это инструмент автоматизирующий процесс создания заглушек для классов, интерфейсов методов, используемых в тестируемом методе.
* **Rhino Mocks** — одна из популярных платформ для динамической генерации mock объектов, используемых в юнит тестировании приложений .NET Framework.
* **Fake** – объекты, имеющие работающие реализации, но в таком виде, который делает их неподходящими для production-кода. Fake объект может быть представлен как обобщенное понятие управляемой замены существующих объектом. Может являться как Stub объектом, так и Mock объектом.
* **Dynamic Fake Object** – Stub или Mock объект, создаваемый во время выполнения, избавляющий программиста от ручной реализации интерфейса заменяемого объекта. Rhino Mocks Isolation Framework позволяет создавать Dynamic Fake Objects.
* **Rhino** Mocks позволяет создавать и использовать Mock объекты двумя способами:
  1. Используя модель Record-and-Play
  2. Используя модель Arrange-Act-Assert

**Модель Record-and-Play** базируется на создании некоего сценария выполнения теста с ожидаемыми входными и выходными данными. Если Выполняемый тест придерживается созданного сценария он успешно завершается.

namespace \_002\_Mocks

{

class FileManager

{

ILogService logService;

IMailService mailService;

public FileManager(ILogService logService, IMailService mailService)

{

this.logService = logService;

this.mailService = mailService;

}

public void Analize(string fileName)

{

try

{

if (fileName.Length < 8)

logService.LogError("Filename too short: " + fileName);

if (Path.GetExtension(fileName) != ".txt")

logService.LogError("FileExtension error: " + fileName);

}

catch (Exception ex)

{

mailService.SendMail("some@mail.mail", ex.Message);

}

}

}

}

using NUnit.Framework;

namespace \_002\_Mocks

{

[TestFixture]

class FileManagerTest

{

[Test]

public static void FileManagerTest2()

{

var logService = new StubLogService();

var mailService = new MockMailService();

FileManager mgr = new FileManager(logService, mailService);

string fileName = "SomeFile.log";

mgr.Analize(fileName);

Assert.AreEqual("FileExtension error: " + fileName, mailService.message);

}

}

}

namespace \_002\_Mocks

{

public interface ILogService

{

void LogError(string error);

}

}

namespace \_002\_Mocks

{

public interface IMailService

{

void SendMail(string destination, string message);

}

}

namespace \_002\_Mocks

{

class MockMailService : IMailService

{

public string message;

public string destination;

public void SendMail(string destination, string message)

{

this.message = message;

this.destination = destination;

}

}

}

namespace \_002\_Mocks

{

class StubLogService : ILogService

{

public void LogError(string error)

{

throw new Exception(error);

}

}

}

# RhinoMock

namespace \_004\_RhinoMocks

{

class FileManager

{

ILogService service;

public FileManager(ILogService service)

{

this.service = service;

}

public void Analize(string fileName)

{

if (fileName.Length < 8)

service.LogError("Filename too short:" + fileName);

if (Path.GetExtension(fileName) != ".txt")

service.LogError("FileExtension error: " + fileName);

}

}

}

namespace \_004\_RhinoMocks

{

[TestFixture]

class FileManagerTest

{

[Test]

public static void FileManagerTest4()

{

string fileName = "TestFileName.exe";

MockRepository rhinoEngine = new MockRepository();

ILogService logService = rhinoEngine.DynamicMock<ILogService>();

using (rhinoEngine.Record())

{

logService.LogError("FileExtension error: " + fileName);

}

FileManager fileManager = new FileManager(logService);

fileManager.Analize(fileName);

rhinoEngine.Verify(logService);

}

}

}

namespace \_004\_RhinoMocks

{

public interface ILogService

{

void LogError(string error);

}

}