Модульное тестирование

Модульное тестирование

Или **юнит-тестирование** *(unit testing)* — процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы.

Идея состоит в том, чтобы писать тесты для каждой нетривиальной функции или метода. Это позволяет достаточно быстро проверить, не привело ли очередное изменение кода к регрессии, то есть к появлению ошибок в уже протестированных местах программы, а также облегчает обнаружение и устранение таких ошибок.

Модульное тестирование. Цель

Цель модульного тестирования состоит в выявлении локализованных в модуле ошибок в реализации алгоритмов, а также в определении степени готовности системы к переходу на следующий уровень разработки и тестирования.

Модульное тестирование чаще всего проводится по принципу "белого ящика ".

Модульное тестирование. Подходы

Существует два основных подхода к тестированию классов: просмотр *(review)* программного кода и тестовые прогоны.

Просмотр исходного кода ПО производится с целью обнаружения ошибок и дефектов, возможно, до того, как это ПО заработает. Просмотр кода предназначен для выявления таких ошибок, как неспособность выполнять то или иное требование спецификации или ее неправильное понимание, а также алгоритмических ошибок в реализации.

Просмотр программного кода

Пошаговое выполнение является одним из методом автономного тестирования и отладки небольших программ.

При пошаговом выполнении программы код выполняется строчка за строчкой.

В среде Microsoft Visual Studio возможны следующие команды пошагового выполнения:

- **Step Into -** если выполняемая строчка кода содержит вызов функции, процедуры или метода, то происходит вызов, и программа останавливается на первой строчке вызываемой функции, процедуры или метода.
- **Step Over -** если выполняемая строчка кода содержит вызов функции, процедуры или метода, то происходит вызов и выполнение всей функции и программа останавливается на первой строчке после вызываемой функции.
- **Step Out** - предназначена для выхода из функции в вызывающую функцию. Эта команда продолжит выполнение функции и остановит выполнение на первой строчке после вызываемой функции.

Просмотр программного кода

Контрольная точка (*breakpoint*) - точка программы, которая при ее достижении посылает отладчику сигнал. По этому сигналу либо временно приостанавливается выполнение отлаживаемой программы, либо запускается программа «агент», фиксирующая состояние заранее определенных переменных или областей в данный момент.

Когда выполнение в контрольной точке приостанавливается, отлаживаемая программа переходит в режим останова (*break mode*). Вход в режим останова не прерывает и не заканчивает выполнение программы и позволяет анализировать состояние отдельных переменных или структур данных. Возврат из режима *break mode* в режим выполнения может произойти в любой момент по желанию пользователя.

Стратегия «белого ящика»

Тестирование белого ящика предполагает, что проверяющий располагает исходным кодом модуля.

Текст программы обычно состоит из последовательности выполняемых операторов (присваивания, печати, вызова функций и пр.), условных операторов, операторов выбора и циклов.

Стратегия «белого ящика»

При использовании стратегии «белого ящика» тестовые наборы формируют путем анализа маршрутов, предусмотренных алгоритмом.

Под ***маршрутами*** при этом понимают последовательности операторов программы, которые выполняются при конкретном варианте исходных данных. Тестирование, проводимое по составленным таким образом тестам, называют ***структурным*** или *тестированием по маршрутам*.

Стратегия «белого ящика»

Исходный текст программы представляется в виде потокового графа управления.

Структурные критерии тестирования базируются на основных элементах управляющего графа программы - операторах, ветвях и путях.

Стратегия «белого ящика»

В основе структурного тестирования лежит концепция максимально полного тестирования всех маршрутов программы. Если алгоритм программы включает ветвление, то при одном наборе исходных данных может быть выполнена последовательность операторов, реализующая действия, которые предусматривает одна ветвь, а при втором – другая. Соответственно, для программы будут существовать маршруты, различающиеся выбранным при ветвлении вариантом.

Считают, что программа проверена полностью, если с помощью тестов удается осуществить выполнение программы по всем возможным маршрутам передач управления.

Однако, даже в программе среднего уровня сложности число неповторяющихся маршрутов может быть очень велико, и, следовательно, полное или исчерпывающее тестирование маршрутов, как правило, невозможно.

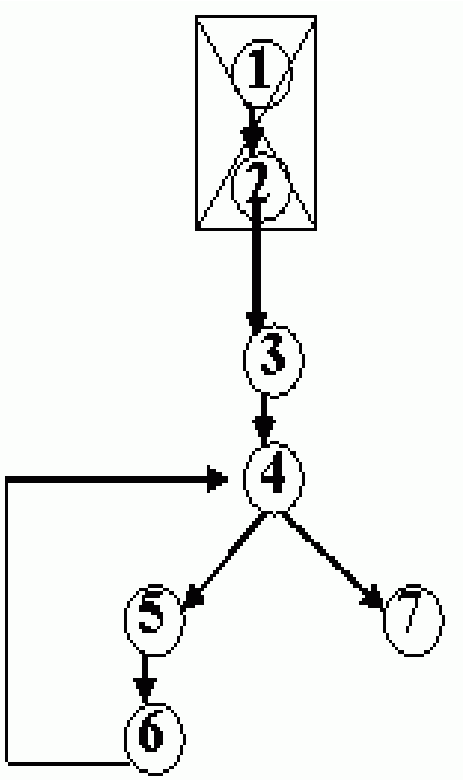
Последовательность составления тестов

1. На основе алгоритма (текста) программы формируется управляющий (потоковый) граф (или граф-схема). **Узлы** (вершины) потокового графа соответствуют линейным участкам программы, включают один или несколько операторов программы. **Дуги** (ориентированные ребра) потокового графа отображают поток управления в программе (передачи управления между операторами).

Пример

Управляющий граф программы

```
/* Функция вычисляет
   неотрицательную   степень n числа
   x */
1  double Power(double x, int n){
2  double z=1; int i;
3  for ( i = 1;
4        i <= n;
5        i++ )
6         { z = z*x; }       /* Возврат в п.4 */
7   return z;
   }
```



примеры путей: (3,4,7), (3,4,5,6,4,5,6), (3,4,5,6)

примеры ветвей: (3,4) (4,5,6,4) (4,7)

Последовательность составления тестов

2. Выбирается критерий тестирования. Формирование тестовых наборов для тестирования маршрутов может осуществляться по нескольким критериям:

- ✓ покрытие маршрутов,
- ✓ покрытие операторов,
- ✓ покрытие решений (переходов),
- ✓ покрытие условий,
- ✓ покрытие решений/условий,
- ✓ комбинаторное покрытие условий,
- ✓ покрытие потоков данных,
- ✓ покрытие циклов.

3. Подготавливаются тестовые варианты, инициирующие выполнение каждого пути. Каждый тестовый вариант формируется в следующем виде:

Исходные данные (ИД):

Ожидаемые результаты (ОЖ.РЕЗ.):

Основные проблемы тестирования

Тестирование программы на всех входных значениях невозможно.

Невозможно тестирование и на всех путях.

Следовательно, надо отбирать конечный набор тестов, позволяющий проверить программу **на основе интуитивных представлений.**

Требование к тестам - программа на любом из них должна останавливаться, т.е. не зацикливаться. В теории алгоритмов доказано, что не существует общего метода для решения этого вопроса, а также вопроса, достигнет ли программа на данном тесте заранее фиксированного оператора.

Задача о выборе конечного набора тестов (X,Y) для проверки программы в общем случае неразрешима.

Структурные критерии

- **Тестирование команд** (критерий C0) - набор тестов в совокупности должен обеспечить прохождение каждой команды не менее одного раза. Это необходимое, но недостаточное условие для приемлемого тестирования.
- **Тестирование ветвей** (критерий C1) - набор тестов в совокупности должен обеспечить прохождение каждой ветви не менее одного раза.
- **Тестирование путей** (критерий C2) - набор тестов в совокупности должен обеспечить прохождение каждого пути не менее 1 раза. Если программа содержит цикл (в особенности с неявно заданным числом итераций), то число итераций ограничивается константой. Этот критерий удовлетворяет критерию покрытия команд, но является более «сильным».

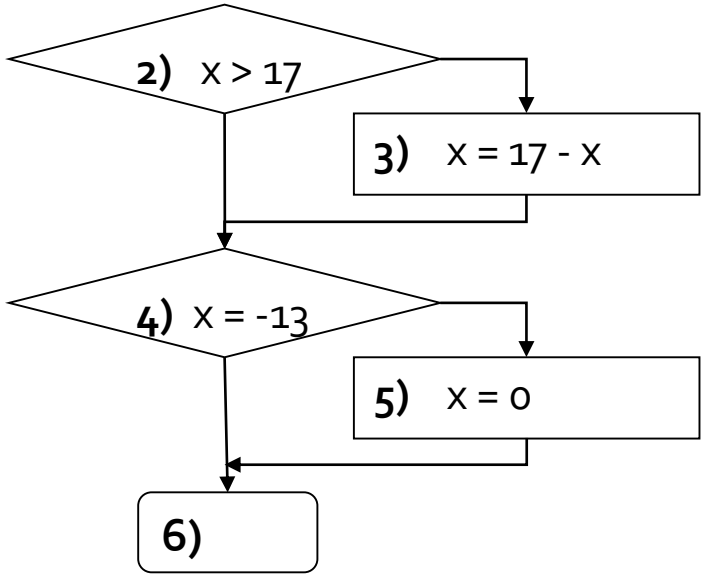
Структурные критерии

- **Тестирование условий** - покрытие всех булевских условий в программе. В этом случае формируют некоторое количество тестов, достаточное для того, чтобы все возможные результаты каждого условия в решении были выполнены, по крайней мере, один раз. Однако, как и в случае покрытия решений, этот критерий не всегда приводит к выполнению каждого оператора, по крайней мере, один раз. К критерию требуется дополнение, заключающееся в том, что каждой точке входа управление должно быть передано, по крайней мере, один раз.

Критерии покрытия решений и условий не заменяют друг друга, поэтому на практике используется **комбинированный критерий покрытия условий/решений**, совмещающий требования по покрытию и решений, и условий. Тесты должны составляться так, чтобы, по крайней мере, один раз выполнились все возможные результаты каждого условия и все результаты каждого решения, и каждому оператору управление передавалось, по крайней мере, один раз.

Пример

```
1  public void Method (ref int x) {
2  if (x>17)
3        x = 17-x;
4  if (x==-13)
5        x = 0;
6  }
```



критерий команд (C0):
(вх, вых) = {(30, 0)} - все операторы трассы 1-2-3-4-5-6

критерий ветвей (C1):
(вх, вых) = {(30, 0), (17, 17)}

критерий путей (C2):
(вх, вых) = {(30,0), (17,17), (-13,0), (21,-4)}

Условия операторов if

	(30,0)	(17,17)	(-13,0)	(21,-4)
2 if (x>17)	>	<=	<=	>
4 if (x==-13)	=	!=	=	!=

Недостаток структурных критериев

Критерий ветвей C2 проверяет программу более тщательно, чем критерии - C1, однако даже если он удовлетворен, нет оснований утверждать, что программа реализована в соответствии со спецификацией.

Например, если спецификация задает условие, что |x|<100, то невыполнимость можно подтвердить на тесте (-177,-177).

Структурные критерии не проверяют соответствие спецификации, если оно не отражено в структуре программы. Поэтому при успешном тестировании программы по критерию C2 можно не заметить ошибку, связанную с невыполнением некоторых условий спецификации требований.

Модульное тестирование. Тестовый прогон

Модульное тестирование (*при тестовом прогоне*) обычно подразумевает создание вокруг каждого модуля определенной среды.

Модульное тестирование

Модульное тестирование, или юнит-**тестирование** (*unit testing*) — процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы. Идея состоит в том, чтобы писать тесты для каждой нетривиальной функции или метода.

Модульное тестирование проводится вызывая код, который необходимо проверить и при поддержке сред разработки, таких как фреймворки (*frameworks* - каркасы) для модульного тестирования или инструменты для отладки

Модульное тестирование

- Тестируется минимально возможный для тестирования компонент, например, отдельный класс или функция.
- Процесс проверки корректностей модулей(классов) программы.
- Заключается в изолированной проверке каждого отдельного элемента путем запуска тестов в искусственной среде.

Цель

- Достоверность в соответствии требованиям каждого отдельного модуля системы перед тем, как будет произведена его интеграция в состав системы
- Получение работоспособного кода с наименьшими затратами
- Определение степени готовности системы к переходу на следующий уровень разработки и тестирования
- Разработка тестов, проверяющих работу каждой нетривиальной функции или метода модуля

Задачи модульного тестирования

- Поиск и документирование несоответствий требованиям
- Поддержка разработки и рефакторинга низкоуровневой архитектуры системы и межмодульного взаимодействия
- Поддержка рефакторинга модулей
- Поддержка устранения дефектов и отладки

Рефакторинг — это контролируемый процесс улучшения кода, без написания новой функциональности.
Рефакторинг — процесс полного или частичного преобразования внутренней структуры программы при сохранении её внешнего поведения.
Цель рефакторинга — сделать код программы легче для понимания; без этого рефакторинг нельзя считать успешным.

Особенности модульных тестов

- Всегда должны проходить на 100%.
- Отделены от кода приложения.
- Независимы друг от друга, просты.
- Часто пишутся программистами.
- Могут писаться **ДО** кода приложения.

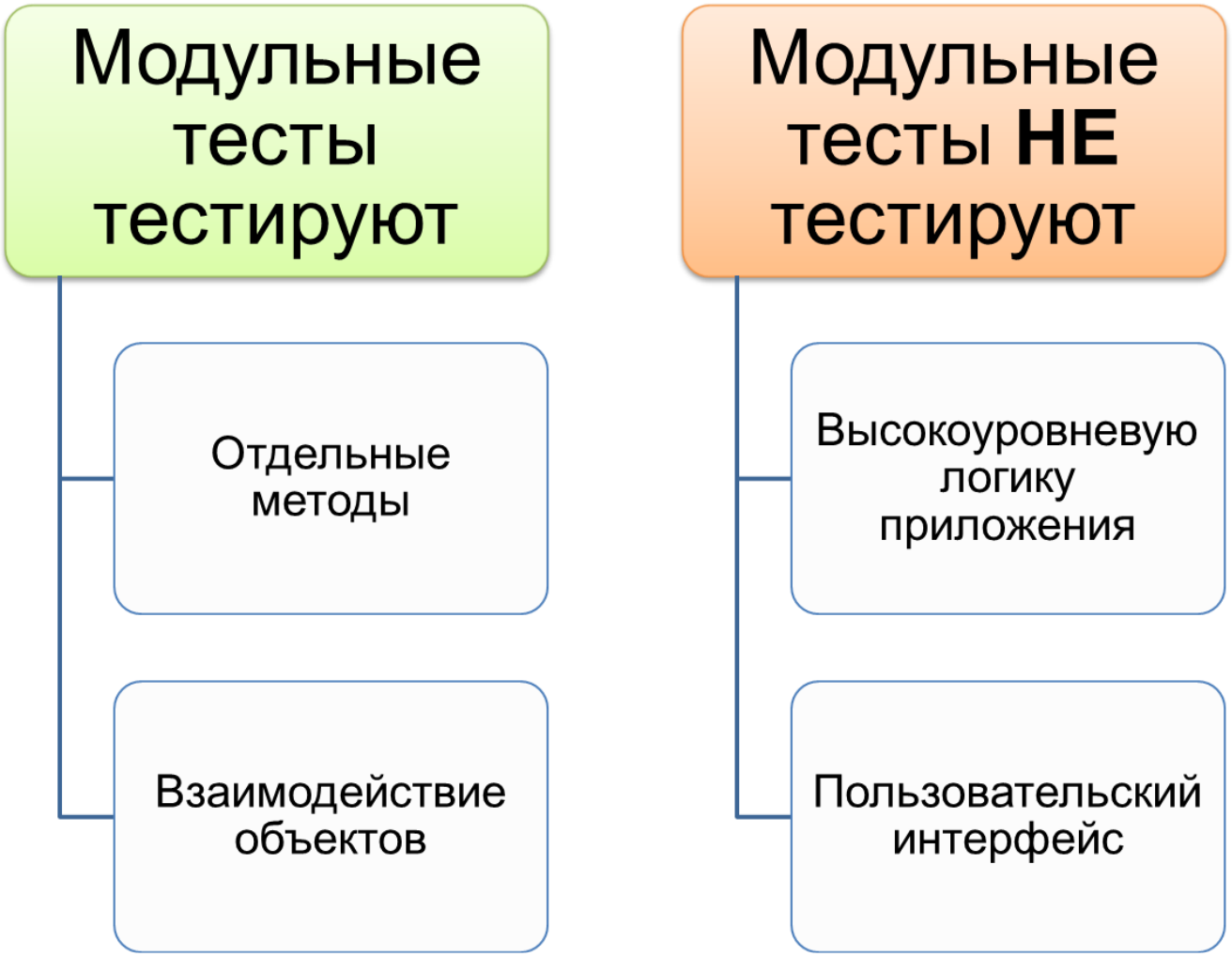
Преимущества модульных тестов

- Повышают качество архитектуры приложения.
- Стимулируют написание простых методов.
- Поощряют изменения в коде.
- Упрощают интеграцию кода.
- Помогают документированию кода.
- Минимизируют зависимости в системе.
- Создаются на основе бесплатных фреймворков.
- Созданы для многих языков программирования.

Недостатки

- Увеличение срока разработки
- При изменении требований, ранее созданные тесты становятся ненужными

Что проверяют модульные тесты



Что проверяют модульным тестированием

- Метод.
- Метод, который ничего не возвращает.
- Класс.
- Взаимодействие классов.
- Геттеры и сеттеры.
- Конструкторы.
- Исключения.
- Внешние зависимости.

Модульное тестирование и TDD

- Один из наиболее эффективных подходов к компонентному (модульному) тестированию - это подготовка автоматизированных тестов до начала основного кодирования (разработки) программного обеспечения.

Это называется **разработка от тестирования** (*test-driven development*) или подход тестирования вначале (*test first approach*). При этом подходе создаются и интегрируются небольшие куски кода, напротив которых запускаются тесты, написанные до начала кодирования. Разработка ведется до тех пор пока все тесты не будут успешно пройдены.

Модульное тестирование и TDD

Разработка через тестирование *(test-driven development - TDD)* **--**

техника программирования, при которой модульные тесты для программы или её фрагмента пишутся до самой программы и, по существу, управляют её разработкой.

Разработка через тестирование предполагает…

- Проектируя код, постоянно запускаем его и получаем представление о том, как он работает, это помогает принимать правильные решения.
- Самостоятельно пишем свои собственные тесты, так как не можем ждать, пока кто-нибудь напишет их.

Разработка через тестирование

- Среда разработки должна быстро реагировать на небольшие модификации кода.
- Дизайн программы должен базироваться на множестве сильно связанных компонент, которые слабо сцеплены друг с другом, благодаря чему тестирование упрощается.

Порядок действий

- Написать тест.
- Заставить компилироваться.
- Запустить тест и убедиться, что он не работает.
- Заставить тест работать.
- Рефакторинг.
- Проверить, что тест работает.

Цикл разработки

Красный — напишите небольшой тест, который не работает, а возможно, даже не компилируется.

Зеленый — заставьте тест работать как можно быстрее, при этом не думайте о правильности дизайна и чистоте кода. Напишите ровно столько кода, чтобы тест сработал.

Рефакторинг — удалите из написанного кода любое дублирование.

Красный — зеленый — рефакторинг

Тест -

Это процедура, которая позволяет подтвердить или опровергнуть работоспособность кода

- Автоматические тесты можно выполнить даже если не хватает времени на тестирование.
- Тесты должны выполняться быстро.
- Тест позволяет избежать страха и сомнений при изменении кода.

Изолированный тест *(Isolated Test)*

- Выполнение одного теста не должно влиять на другие тесты.
- Порядок выполнения тестов не должен иметь значения.
- Приложение должно быть собрано из множества небольших, взаимодействующих друг с другом объектов.

Вначале тест *(Test First)*

- Тесты должны писаться до кода. Это позволяет контролировать количество работы и следить за дизайном кода.

Принцип написания теста

Метод теста модуля должен следовать согласованному формату, чтобы упростить его понимание другими разработчиками. В целом, лучшим считается следующий формат.

1. **Arrange** (подготовка).
2. **Act** (выполнение).
3. **Assert** (проверка).

Arrange: подготовка среды, в которой выполняется код

Act: тестирование кода (обычно представляет одну строку кода)

Assert: убеждаемся, что результат теста именно тот, что и ожидали

Принцип написания теста

1. **Arrange:** Может понадобиться написание какого-то подготовительного кода для создания экземпляра тестируемого класса, а также любых зависимостей, которые могут быть у него.
2. **Act:** После завершения подготовительной части к собственно тесту можно выполнить интересующий метод или свойство. Можно выполнять метод или свойство с параметрами (если это применимо), подготовленными в разделе ***Arrange***.
3. **Assert:** Когда выполнили интересующий метод или свойство, тест должен проверить, что этот метод или свойство сделали именно то, что от него ожидалось. В фазе ***Assert*** вызываются методы проверки для сравнения полученных результатов с ожидаемыми. Если полученные результаты соответствуют ожидаемым, модульный тест пройден. В ином случае тест провален.

Принцип написания теста

Следуя этому формату, тесты обычно выглядят примерно так:

```
[Test]
public void SomeTestMethod()
{
    // Arrange
    // *Вставка кода для подготовки теста
    // Act
    // *Вставка кода для вызова тестируемого метода или
свойства
    // Assert
    // *Вставка кода для проверки завершения теста
    // ожидаемым образом
}
```

Список инструментов

- Java: JUnit;
- C++: CppUnit, Boost Test;
- Delphi: DUnit;
- PHP: PHPUnit;
- C#: NUnit.

Полный список:

http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks

NNit

NUnit — открытая среда юнит-тестирования приложений на C#.

NUnit обычно используется для разработке на платформе Mono, в то время как Microsoft Unit Testing Framework обычно используется в .NET Framework.

Между этими фреймворками существуют ***отличия***, которые заключаются в основном в ***названии атрибутов*** .

Первое что нужно отметить, это отличие между подключаемыми пространствами имен:

NUnit	Microsoft Unit Testing Framework
using NUnit.Framework;	**using** Microsoft.VisualStudio.TestTools.UnitTesting;

NUnit vs Microsoft Unit Testing Framework

NUnit	Microsoft Unit Testing Framework	Описание
TestFixture	**TestClass**	Указывает на тестирующий класс
TestFixtureSetUp	**ClassInitialize**	Указывает на метод, который нужно вызвать перед запуском всех тестов
TestFixtureTearDown	**ClassCleanup**	Указывает на метод, который нужно вызвать после выполнения всех тестов
SetUp	**TestInitialize**	Указывает на метод, который нужно вызвать перед запуском каждого теста
TearDown	**TestCleanup**	Указывает на метод, который нужно вызвать после выполнения каждого теста
Test	**TestMethod**	Указывает на сам метод теста.

NUnit. Пример

```
[TestFixture]
public class TestGame {
    private Game game;

    [SetUp]
    public void SetUp() {
        game = new Game();
    }

    [Test]
    public void TestTwoThrowsNoMark() {
        game.Add(5);
        game.Add(4);
        Assert.AreEqual(9, game.GetScore());
    }
}
```

NUnit. Утверждения

Традиционная модель проверок, корни которой лежат, пожалуй, в самом первом публичном unit testing framework, **JUnit**, подразумевает, что есть некоторый класс (***Assert***), имеющий набор статических методов для проверки различных условий.

Первые версии **NUnit** использовали традиционную модель проверок, и она до сих пор сохранена для обеспечения обратной совместимости.

Сейчас основная модель проверок основана на использовании отдельных объектов-constraints (реализующих интерфейс IConstraint), каждый из которых реализует ту или иную проверку.

NUnit. Утверждения.

Традиционная модель

- Класс Assert, содержит набор методов, позволяющих выполнять различные проверки данных.
- Если данные неверны, метод оповещает среду выполнения NUnit, что тест не пройден.

NUnit. Утверждения.

Традиционная модель

- ***Assert.AreEqual*** – проверяет равенство входных параметров;
- ***Assert.AreNotEqual*** – проверяет то, что входные параметры неравны;
- ***Assert.AreSame*** – проверка на то, что входные параметры ссылаются на один и тот же объект;
- ***Assert.AreNotSame*** – входные параметры не ссылаются на один и тот же объект;
- ***Assert.Contains*** – метод получает на входе объект и коллекцию и проверяет, что данный объект содержится в это коллекции;
- ***Assert.IsNu***ll – входной параметр – null;
- ***Assert.IsEmpty*** – входной параметр – пустая коллекция.
- ***Assert.Fail*** – прерывает выполнение теста и среде NUnit, что тест не пройден. Эта функция может быть использована, когда нужно организовать более сложные типы проверок.

NUnit. Утверждения

- С версии NUnit 2.4 введены constraint-based утверждения.

Новый тип утверждений базируется на статической функции

Assert.That() – Ожидаем.Что()

Синтаксис для всех Constraint проверок всегда одинаков:

Assert.That(<Проверяемый объект>, <Способ проверки>)

NUnit. Пример

```
[Test]
public void ConstraintBasedTest1()
{
   string actualValue = "Привет";
   Assert.That(actualValue,Is.EqualTo("Привет"));

   //или (одинаковые проверки)

   Assert.That(actualValue,
      new EqualConstraint("Привет"));
}
```

Ранее можно было напрямую создавать новые экземпляры класса с помощь ***new***. Но это просто не удобочитаемо для человека. Поэтому разработчики *NUnit* создали класс ***Is***, который выполняет создание нового экземпляра *Constraint*, с помощью своих статических методов.

Создание и запуск модульных тестов

1. Создать проект для тестирования

- ❑ Запустите *Visual Studio*.
- ❑ На начальном экране выберите *Создать проект*.
- ❑ Найдите и выберите шаблон проекта **Консольное приложение (.NET Core) на C#** и нажмите кнопку *"Далее"*.
- ❑ Присвойте проекту ***имя*** и нажмите кнопку *"Создать"*.

Создание и запуск модульных тестов

2. Создать проект модульного теста

- В *обозревателе решений* щелкните решение правой кнопкой мыши и выберите пункты ***Добавить > Создать проект***.
- Найдите и выберите шаблон проекта

  Тестовый проект MSTest (.NET Core) на C#

  и нажмите кнопку *"Далее"*.
- Назовите проект ***имя_тест***.
- Нажмите кнопку *"Создать"*.

  Проект ***имя_тест*** добавляется в решение ***имя***.
- В проекте ***имя_тест*** добавьте ссылку на проект ***имя***.

В итоге :

(пример модульного теста)

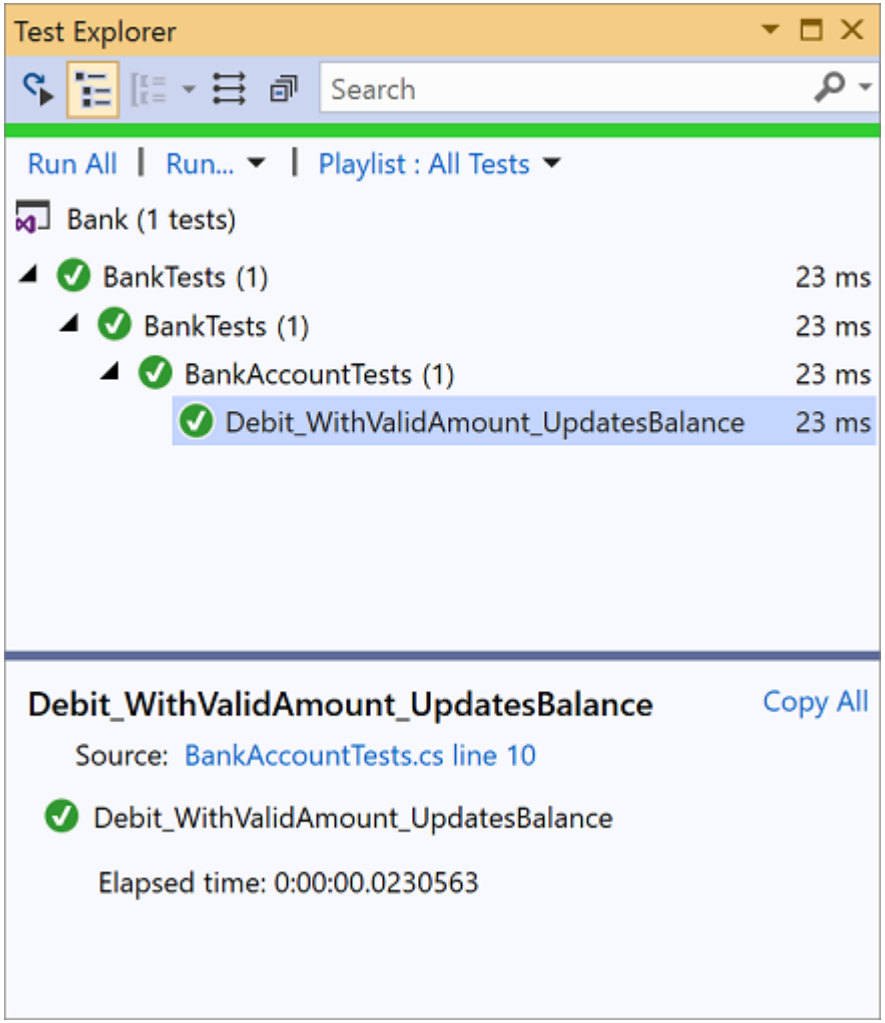
```
using BankAccountNS;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace BankTests
{
        [TestClass]
        public class BankAccountTests
        {
                [TestMethod]
                public void TestMethod1()
                {
                }
        }
 }
```

Сборка и запуск теста

- Выберите **Построить решение.**
- Откройте **Обозреватель тестов**, выбрав
  Тест > Windows > Обозреватель тестов
  в верхней строке меню.
- Выберите **Запустить все** , чтобы выполнить тест.
- Во время выполнения теста в верхней части окна **Обозреватель тестов** отображается анимированная строка состояния. По завершении тестового запуска строка состояния становится зеленой, если все методы теста успешно пройдены, или красной, если какие-либо из тестов не пройдены (В данном случае тест пройден не будет).
- Выберите этот метод в обозревателе тестов для просмотра сведений в нижней части окна.

Сборка и запуск теста



NUnit. Использование сторонней платформы тестирования

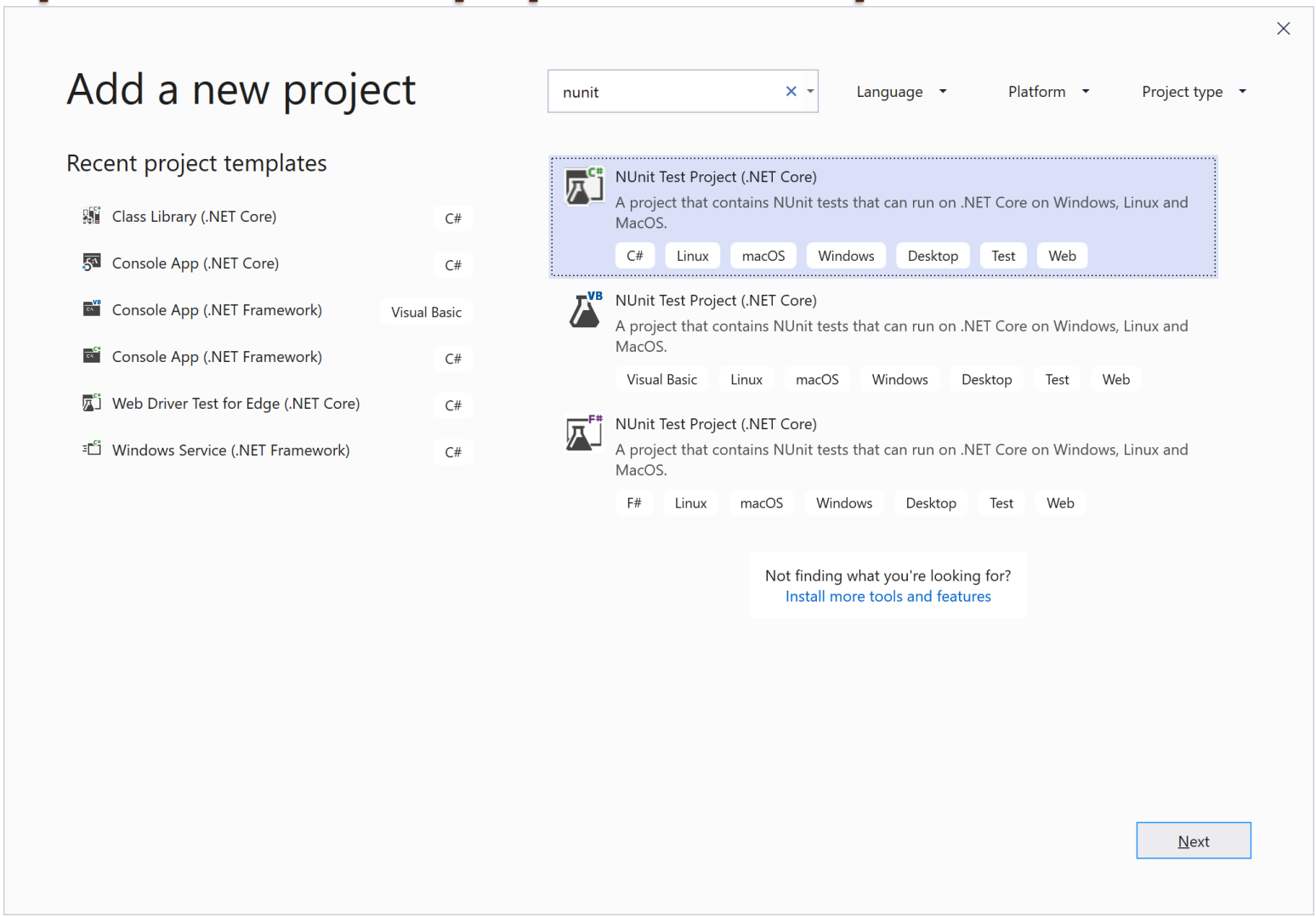
Модульные тесты можно выполнять в *Visual Studio* с помощью сторонних платформ тестирования, таких как *Boost*, *Google* и *NUnit*. Используйте **диспетчер пакетов NuGet**, чтобы установить пакет **NuGet** для выбранной платформы. Или, для платформ тестирования *NUnit* и *xUnit*, *Visual Studio* включает предварительно настроенные шаблоны проекта тестирования, которые включают необходимые пакеты **NuGet**.

NUnit. Использование сторонней платформы тестирования

Чтобы создать модульные тесты, использующие **NUnit**:

- Откройте решение, содержащее код, который нужно протестировать.
- В **обозревателе решений** щелкните решение правой кнопкой мыши и выберите *Добавить > Создать проект*.
- Выберите шаблон проекта **Тестовый проект NUnit**.

NUnit. Использование сторонней платформы тестирования

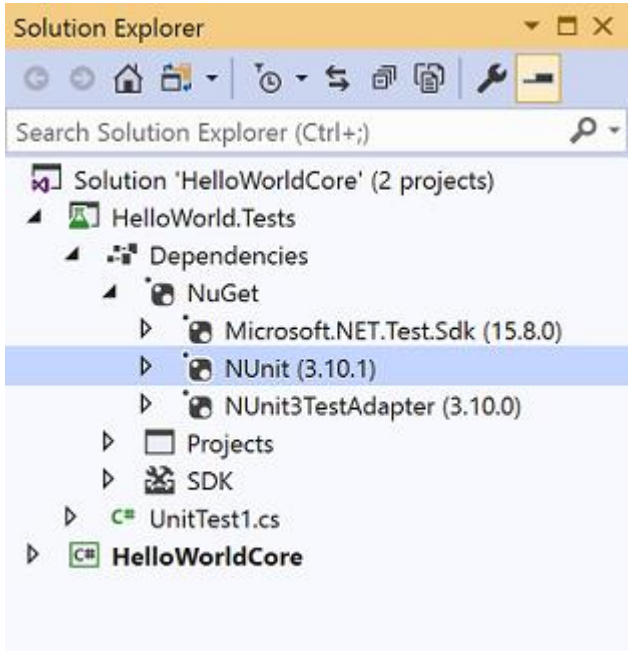


- Щелкните **Далее**, назовите проект и нажмите кнопку **Создать.**

NUnit. Использование сторонней платформы тестирования

- Шаблон проекта содержит ссылки на пакеты **NuGet** для *NUnit* и *NUnit3TestAdapter*.

- Добавьте ссылку из проекта тестирования на проект, содержащий код, который необходимо протестировать.
  В обозревателе решений щелкните проект правой кнопкой мыши и выберите **Добавить > Ссылка**.



В итоге :

(пример модульного теста через NUnit)

```
using BankAccountNS;
using Nunit.Framework;
Using System.IO;
using System;

namespace NUnitTestProject1
{
    public class Tests
    {
        [SetUp]
        public void Setup()
        {
        }
        [Test]
        public void Test1()
        {
            Assert.Pass();
        }
    }
}
```

Модульное тестирование. Полезный материал

https://habr.com/ru/post/169381/

https://docs.microsoft.com/ru-ru/dotnet/core/testing/unit-testing-best-practices

https://docs.microsoft.com/ru-ru/visualstudio/test/getting-started-with-unit-testing?view=vs-2019

http://schiciuc.blogspot.com/2011/02/constraint-based-nunit-framework.html

http://iqhouse.blogspot.com/2011/07/nunit-constraint.html

Контрольные вопросы:

1. Что представляет собой unit-тестирование? Цель.
2. Задачи модульного тестирования. Недостатки и преимущества.
3. Описать два подхода модульного тестирования.
4. Что представляет собой технология TDD?
5. Что представляет собой шаблон теста?
6. Какие основные атрибуты используются при описании модульного теста при использовании фреймворка *Microsoft Unit Testing Framework*
7. Описать методы класса **Assert**