**4 СЦЕНАРИЙ ВЗАИМОДЕЙСТВИЯ ПОЛЬЗОВАТЕЛЯ С СИСТЕМОЙ**

В данном разделе рассмотрены примеры создания простейших автоматизированных тестов с использованием среды NUnit для модульного тестирования, библиотеки UI Automation для тестирования GUI, а так же фреймворка JMeter для нагрузочного и функционального тестирования веб-приложения.

Данный раздел может быть использован в качестве вспомогательного материала для выполнения практических заданий.

**4.1 Модульное тестирование**

Для создания простейшего unit теста достаточно создать консольное приложение на языке программирования C# и выполнить следующие две операции:

1) подключить библиотеку проекта nunit.framework;

2) подключить соответствующее пространство имён: NUnit.Framework.

Отметим, что класс, содержащий тестовый код, должен иметь директиву public и иметь атрибут TestFixture. Пример класса изображен на рисунке 4.1.

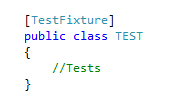


Рисунок 4.1 – Класс, содержащий тестовый код

Предположим, что нам необходимо написать функцию умножения двух целых неотрицательных чисел.

Определим требования для реализации заданной функции умножения:

− так как умножение будет проводиться над двумя неотрицательными числами, то результат должен быть неотрицательный;

− так как умножение будет проводиться над целыми числами, то результат должен представлять целое число;

− результатом заданной функции должно быть умножение.

Объявим функцию умножения двух чисел. Пример объявления представлен на рисунке 4.2.

Определим тестирующую функцию, которая будет непосредственно вызывать функцию Multiplication(int A, int B). Пример такого тест-метода изображен на рисунке 4.3.

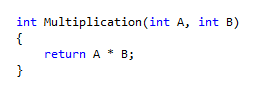


Рисунок 4.2 – Пример объявления функции умножения двух чисел

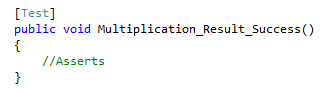


Рисунок 4.3 – Пример тестирующего тест-метода

Как видно из примера на рисунке 4.3 перед объявлением функции определен атрибут Test, который указывает на то, что это метод теста. Тест-метод не должен ничего возвращать и принимать какие-либо параметры. Тест-метод нужен для инициализации необходимых объектов, выполнения теста и проверки результата.

Дополним тест-метод тестовыми данными и утверждениями для проверки корректности результата. Для этого:

− определим тестовые данные, которые буду представлять собой два целочисленных неотрицательных числа;

− определим ожидаемый результат, который будет равен произведению заданных выше чисел;

− проверим, удовлетворяют ли предложенные тестовые данных нашим требованиям.

Пример обновленной функции изображен на рисунке 4.4.

Как показано на рисунке 4.4 тест-метод содержит следующие утверждения:

− IsTrue(bool condition) для проверки выполнения условия на не отрицательность результата;

− IsInstanceOf<type>(object actual) для проверки типа полученного результата;

− AreEqual(int expected, int actual) для проверки полученного результата.

Для запуска тест-метода необходимо скомпилировать текст программы, а затем открыть полученный exe-файл в среде NUnit. После успешного открытия exe-файла NUnit представит все тест-методы тестирующего класса в виде дерева. Пример открытого файла приведен на рисунке 4.5. Как видно дерево включает в себя:

− путь к exe-файлу;

− наименование пространства имен;

− наименование класса, в котором содержится тест метод;

− наименование тест-метода.

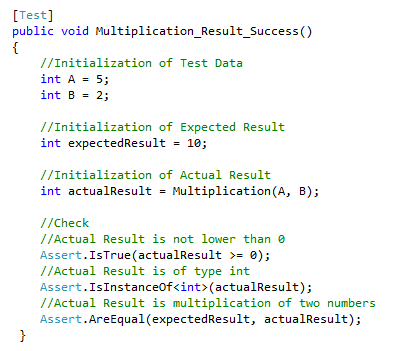


Рисунок 4.4 – Пример тестирующей функции

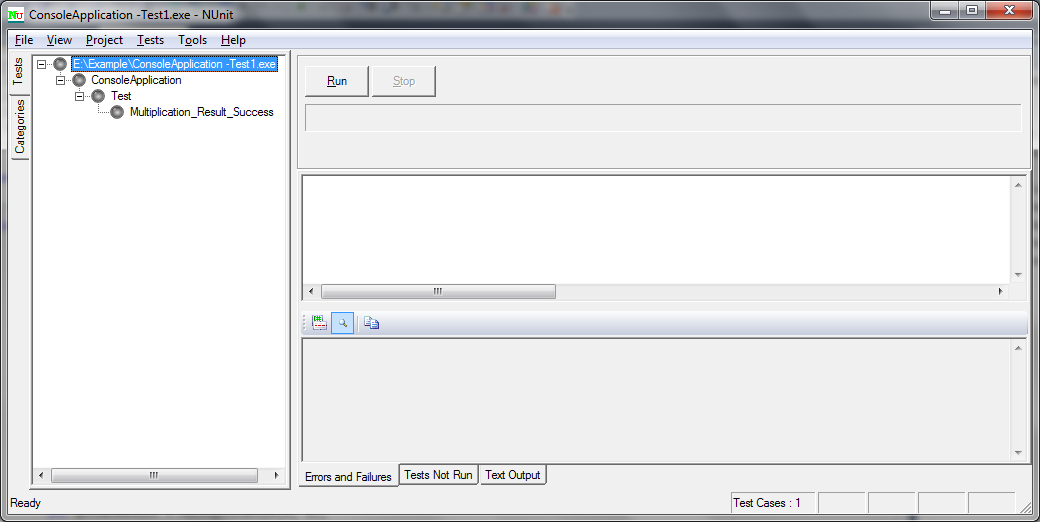


Рисунок 4.5 – Пример открытого exe-файла с тест-методом в среде NUnit

Для запуска тест-метода необходимо нажать на кнопку Run. Все определившиеся в дереве тест методы будут запущены последовательно. После завершения каждого тест-метода он будет помечен соответственно:

− при успешном прохождении тест-метод будет помечен зелёным цветом;

− при провальном прохождении тест-метод будет помечен в красный цвет.

Запустим тест. Пример результата отображен на рисунке 4.6. Как видно, тест успешный.

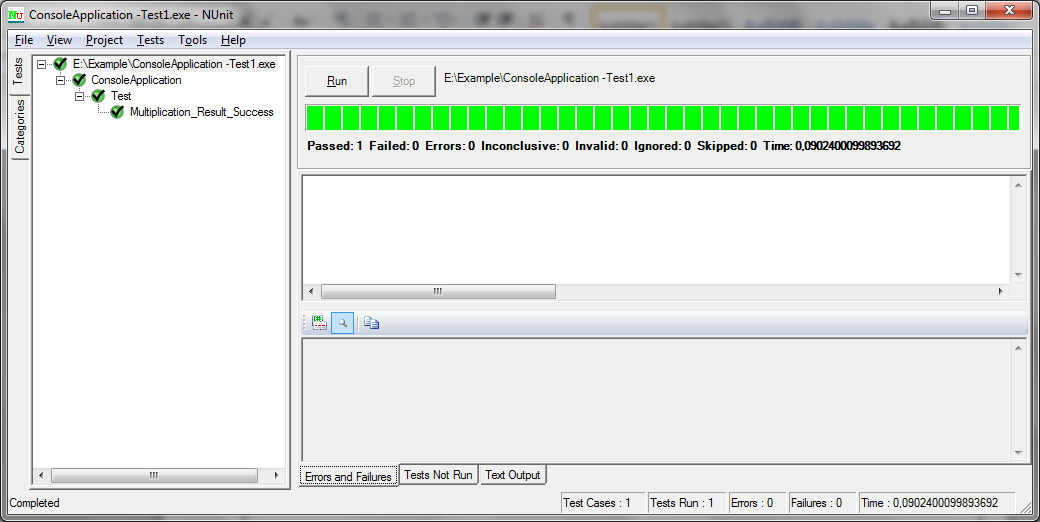


Рисунок 4.6 – Пример успешного выполнения теста

Приведем пример «неуспешного» тест-метода. Для этого модернизируем функцию Multiplication(int A, int B), изменив знак умножения «\*» на знак сложения «+». Пример обновленной функции изображен на рисунке 4.7. Скомпилируем измененный текст программы и запустим тест-метод в среде NUnit.

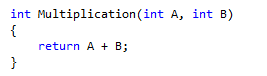


Рисунок 4.7 – Пример модернизированной функции Multiplication(int A, int B)

Результат выполнения отображён на рисунке 4.8. Как видно из примера тест не пройден. Проваленные тесты содержат информацию об ожидаемом и действительном результатах. В данном случае ожидаемым результатом было число 10, а действительным – число 7.

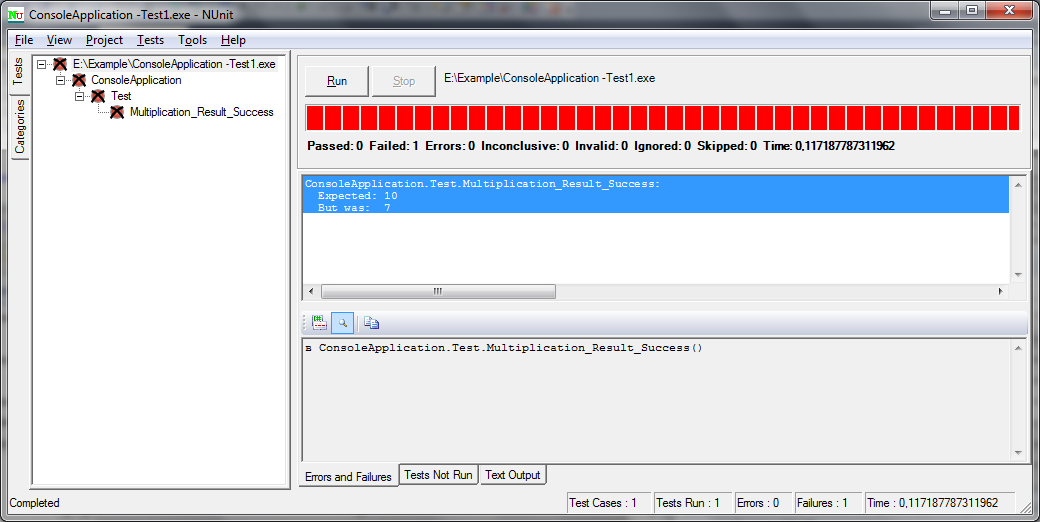


Рисунок 4.8 – Пример проваленного теста

**4.2 Автоматизированное тестирование GUI**

За пример возьмём приложения, написанное на C#, которое будет выводить квадрат или куб введенного числа по нажатию на кнопку Ok, в зависимости от выбранного значения «Square» или «Cube». Пример приложения изображен на рисунке 4.9.

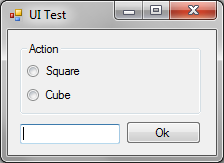


Рисунок 4.9 – Приложение для подсчёта квадрата или куба введенного числа

Текст программы, отвечающий за расчёт квадрата или куба введенного числа и вывод результат в текстовое поле, приведен на рисунке 4.10.

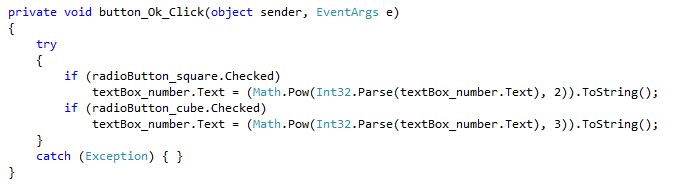


Рисунок 4.10 – Текст программы, отвечающий за расчёт квадрата или куба введенного числа и вывод результат в textBox\_number

Тестирование будем проводить из консольного приложение. Для этого откроем среду разработки Visual Studio и создадим новое Cosole Application. В данном примере будет использоваться язык С#, однако возможно использовать и другой язык совместимый с .NET.

Первое, что необходимо сделать, в созданном Console Application это добавить ссылки проекта на библиотеки UIAutomationClient.dll и UIAutomationTypes.dll. Эти библиотеки являются частью .NET Framework 3.0 и обычно находятся в каталоге %PROGRAMFILES%\Reference Assemblies\Microsoft\Framework\v3.0.

После добавления ссылок проекта, пространство имен System.Windows.Automation становится видным тестовой программе, и появляется возможность добавить оператор использования в пространство имен. Так же понадобятся такие пространства имен как System.Diagnostics (для использования класса Process) и System.Threading (для использования метода Thread.Sleep).

Добавим в приложение блок try/catch для обработки любых неустранимых ошибок.

Пример основы тестирующего средства изображен на рисунке 4.11.

Создание теста включает в себя:

1) вызов тестируемого приложения;

2) получение ссылки на форму;

3) получение ссылок на все пользовательские элементы управления;

4) ввод тестовых данных;

5) проверка результата;

6) завершение работы с тестируемой формой.

Чтобы вызвать тестируемую форму из консольного приложения можно использовать метод Process.Start и передать путь к форме (файлу формата exe) в качестве параметра.

При автоматизации тестирования пользовательского интерфейса, необходимо думать о каждой визуальной сущности (элементах управления, окнах и так далее) на рабочем компьютере как о части иерархического дерева с окном рабочего стола в качестве корневого элемента. Это общий принцип для большинства способов автоматизации тестирования пользовательского, включая использующие библиотеку UIAutomation. Пример такой структуры изображен на рисунке 2.2. Поэтому, чтобы получить ссылку на тестируемое приложение, нужно узнать ссылку на окно рабочего стола. А затем по ссылке на окно рабочего стола – ссылку на окно тестируемого приложения. Большинство объектов при автоматизации тестов с использованием библиотеки UIAutomation принадлежат к типу AutomationElement. Таким образом, можно получить ссылку на визуальный элемент верхнего уровня («Рабочий стол») используя RootElement – статичное свойство AutomationElement.

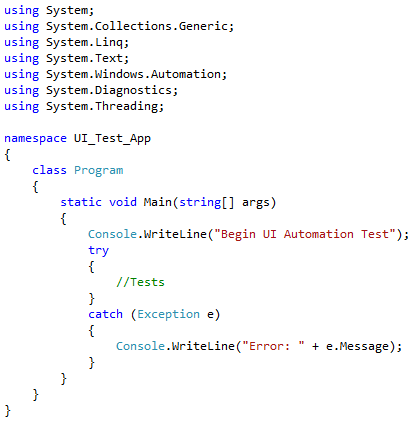


Рисунок 4.11 – Основа тестирующего приложения

Далее необходимо получить ссылку на тестируемое приложение. Для этого после открытия тестируемой формы можно вызвать метод FindFirst объекта ссылки на окно рабочего стола.

Пример получения ссылки на форму приведен на рисунке 4.12.

Как видно из примера метод FindFirst ищет первую доступную ссылку на AutomationElement, соответствующую указанному условию. Первый переданный аргумент TreeScope.Children указывает, что просматривать необходимо только дочерние элементы управления контекста (в данном случае aeDesktop). Второй аргумент описывает условие, которое можно интерпретировать как «Элемент со свойством Name, равным UI Test».

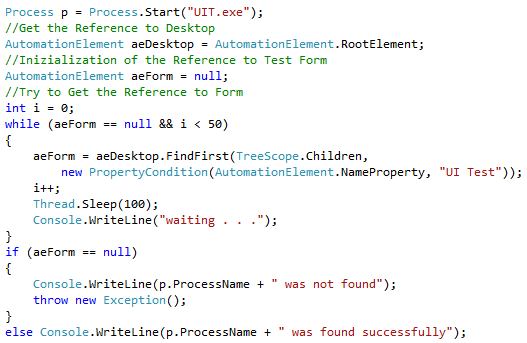


Рисунок 4.12 – Получение ссылки на тестируемое приложение

Метод получения ссылок на пользовательские элементы управления идентичен методу получения ссылки на форму, описанному в пункте 2. Пример получения ссылок на пользовательские элементы управления, такие как текстовое поле textBox\_number, кнопку button\_Ok и кнопок radioButton\_square и radioButton\_cube представлены на рисунках 4.13, 4.14 и 4.15 соответственно.



Рисунок 4.13 – Получение ссылки текстового поля textBox\_number

Как показано на рисунках 4.13 и 4.14 вместо NameProperty используется ControlTypeProperty со значением ControlType.Edit и ControlType.Button. Используемое условие ищет первый встречающийся элемент управления заданного типа (ControlType.Edit для текстового поля и ControlType.Button для кнопки).



Рисунок 4.14 – Получение ссылки на кнопку button\_Ok

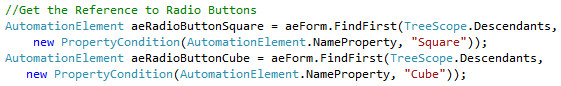


Рисунок 4.15 – Получение ссылок на кнопки radioButton\_square и radioButton\_cube

Заметим, что первый параметр, передаваемый методы FindFirst, - TreeScope.Descendants, а не TreeScope.Children, как в предыдущих примерах. Это объясняется тем, что кнопки radioButton\_square и radioButton\_cube являются наследующими по отношению к форме и дочерними по отношению к контейнеру groupBox\_Action. То есть, поиск надо вести среди потомков элемента, включая дочерние элементы, а не только по дочерним элементам.

После получения ссылок на все пользовательские элементы управления можно перейти непосредственно к тестированию – вводу тестовых данных. Однако изменения состояний элемента управления (ввод данных, выбор/снятие чекбокса, выбор значения из списка и др.) происходит с помощью использования, так называемых, паттернов – шаблона или образца. Объекты шаблонов можно представить себе как абстрактный способ предоставления функций элемента управления, независимых от его типа, или облика. Другими словами, конкретные экземпляры AutomationPattern, такие как ValuePattern, можно использовать для предоставления функций конкретного элемента управления.

Иначе говоря, ControlType элемента управления раскрывает, к какому роду элементов управления принадлежит данный элемент, а его Pattern раскрывает, что он может сделать. Важно отметить, что элемент управления может поддерживать несколько шаблонов.

На рисунках 4.16, 4.17 и 4.18 приведены примеры ввода тестовых данных.

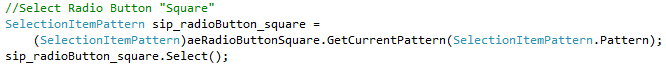


Рисунок 4.16 – Выбор элемента управления radioButton\_square

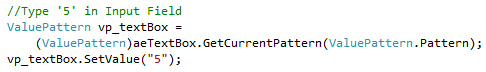


Рисунок 4.17 – Ввод числа в поле ввода textBox\_number

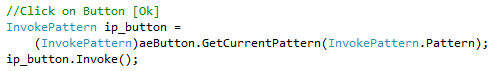


Рисунок 4.18 – Нажатие кнопки button\_Ok

На этапе проверки результатов необходимо выполнить сравнение полученного результата с эталонным значением. Результат работы программы получим в поле textBox\_number и выполним проверку. Пример проверки изображен на рисунке 4.19.

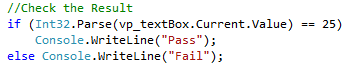


Рисунок 4.19 – Проверка результата

Как видно из примера на рисунке 4.19 в консоль будет выведен текст «Pass», если полученный результат будет равен ожидаемому значению. В противном случае отобразится текст «Fail».

На последнем этапе можно закончить тест и закрыть тестируемое приложение. Для этого воспользуемся классом WindowPattern. Пример реализации закрытия формы изображен на рисунке 4.20.

На рисунке 4.21 представим результат успешного и проваленного выполнения написанного теста.

Итак, написание UI тестов с использованием библиотеки UI Automation включает в себя 3 основные концепции:



Рисунок 4.20 – Закрытие формы

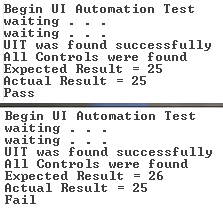


Рисунок 4.21 – Результаты выполнения теста

− использование класса PropertyCondition для указания конкретного элемента управления пользовательского интерфейса;

− использование класса ControlType для определения типа элемента управления пользовательского интерфейса;

− использование класса AutomationPattern для определения функций элемента управления пользовательского интерфейса.

Данный тест может быть модифицирован. Например, можно не задавать жестко в тексте теста тестовые данные и ожидаемый результат, а передавать эту информацию тестовой программе через командную строку. Или, одним из вариантов, является создание базы данных или XML файла и, перебирая данные каждого тестового случая, анализировать их.

**4.3 Тестирование в JMeter**

Рассмотрим создания простого тест плана для тестирования веб-приложения. Будет создано пять пользователей, которые посылают запросы на две страницы веб сайта. Каждый пользователь выполнить свой тест два раза. Поэтому, всего будет сгенерировано 20 HTTP запросам.

Для создания тест плана будут использованы следующие элементы:

− Thread Group;

− HTTP Request;

− HTTP Request Defaults;

− Graph Results.

Первый шаг, который нужно сделать для создания тест плана, - это добавить элемент Thread Group. В этом элементе будет задаваться количество пользователей, и имитироваться, как часто пользователи должны отправлять запросы и сколько запрос должен отправить каждый пользователь.

Для добавления Tread Group необходимо выполнить следующую последовательность действий: Test Plan > Add > Thread Group. Элемент Thread Group будет добавлен в под Test Plan.

Выбрав добавлен элемент Thread Group в правой части среды (далее фреймворк) отобразятся свойства элемента. Изменим их согласно нашему условию. Элемент Thread Group должен иметь вид, как показано на рисунке 4.22.

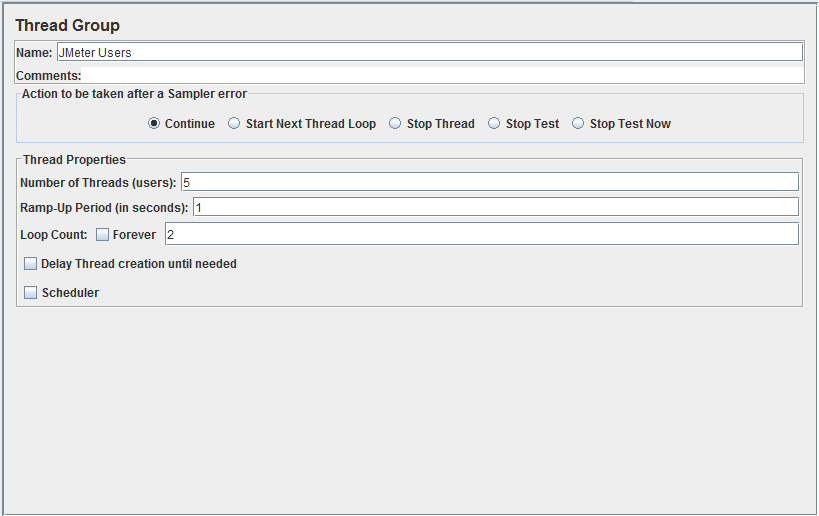


Рисунок 4.22 – Элемент Thread Group

Создадим два HTTP-запроса. Первый для страницы http://nunit.org, а второй для страницы http://nunit.org/index.php?p=download. JMeter будет посылать запросы в порядке их появления в дереве. Итак, для добавления HTTP-запроса необходимо добавить элемент HTTP Request (JMeter Users > Add > Sampler > HTTP Request). Примеры запросов изображены на рисунках 4.23 и 4.24 для первой и второй страниц. Как видно из приведенных рисунков, в качестве имени сервера указали nunit.org.

Добавим в тест-план элемент слушателя View Results Tree (Add > Listeners > View Results Tree), который будет отвечать за предоставление всех результатов HTTP запросов.

После запуска теста, элемент View Results Tree будет отображать все выполнившиеся запросы. Пример такого результата изображен на рисунке 4.25. Как видно из рисунка все HTTP запросы были выполнены успешно.

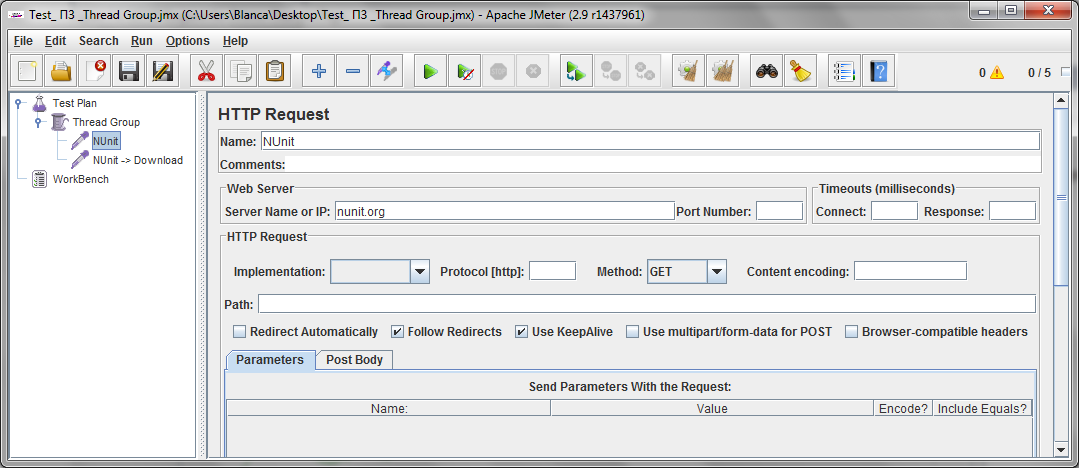


Рисунок 4.23 – HTTP запрос к странице http://nunit.org

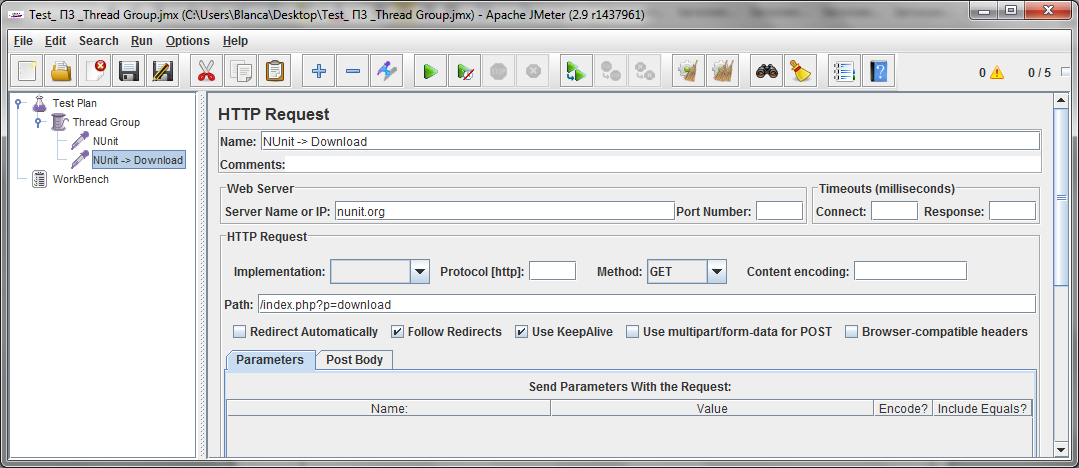


Рисунок 4.24 – HTTP запрос к странице http://nunit.org/index.php?p=download

Теперь добавим элемент слушателя Graph Results (Add > Listeners > View Results Tree), который так же будет отвечать за предоставление результата запросов, но в графическом представлении. Увеличим количество циклов повторения до 25 раз и запустим тест-план. Результат такого тест-плана рассмотрим на рисунке 4.26, представленным элементом Graph Results.

Все значения предоставляются в миллисекундах. Graph Result предоставляет следующую информации:

− data – время отклика каждой отдельной единицы данных т.е. каждого проверенного адреса;

− average – усредненное время отклика, объективный график изменения нагрузки;

− median – значение медианы (используется в статистике);

− deviation – погрешность, стандартное отклонение;

− throughput – пропускная способность выполняемых запросов.

В данном пример достаточно значений Average и Throughput. Они отобразят нагрузку на веб-сервер и пропускную способность запросов. По графику выше видно, что время отклика равно 1567 мс, и дальше оно не растет. А это значит, что сервер нормально выдерживает нагрузку в 5 виртуальных пользователей.

Тест-план может быть модифицирован дополнительными элементами.

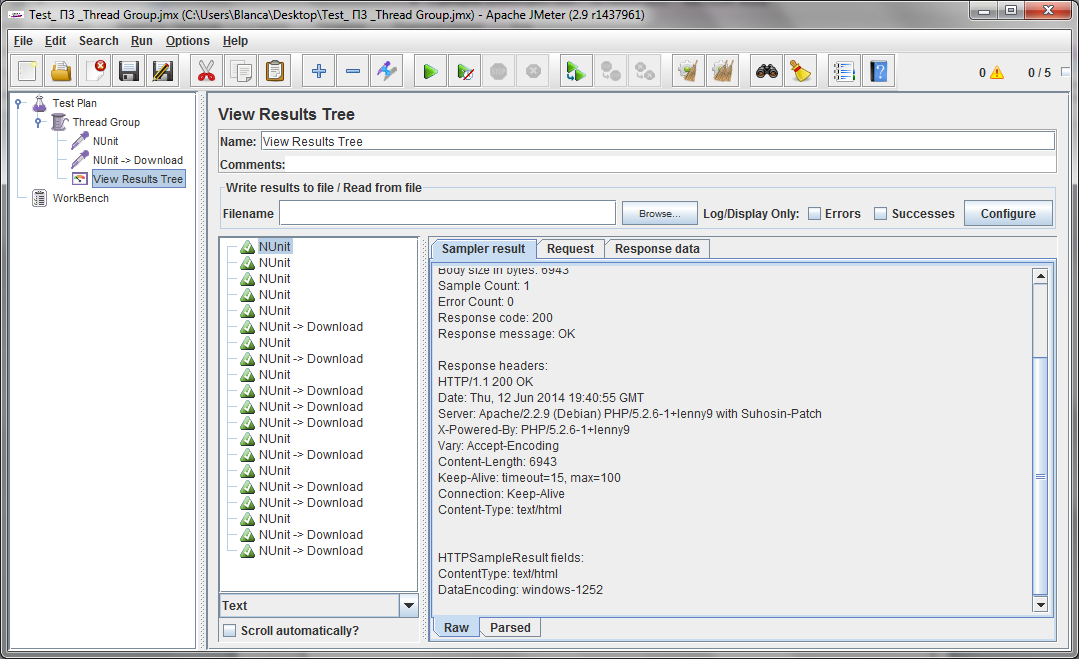


Рисунок 4.25 – Результат использования элемента View Result Tree

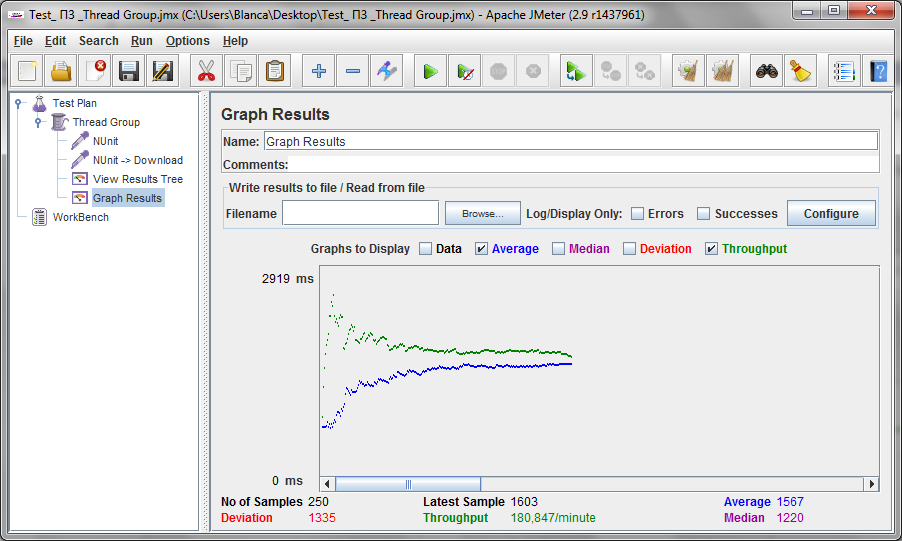


Рисунок 4.26 – Результат использования элемента Graph Results