МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Кузбасский государственный технический университет имени Т. Ф. Горбачева»

Кафедра информационных и автоматизированных производственных систем

Составитель
И. С. Сыркин

ТЕСТИРОВАНИЕ ИНФОРМАЦИОННЫХ СИСТЕМ

Методические указания материалы

Рекомендовано цикловой методической комиссией специальности
СПО 09.02.07 Информационные системы и программирование
в качестве электронного издания
для использования в образовательном процессе

Кемерово 2018

Рецензенты

Ванеев О. Н. – кандидат технических наук, доцент кафедры информационных и автоматизированных производственных систем ФГБОУ ВО «Кузбасский государственный технический университет имени Т. Ф. Горбачева»

Чичерин И. В. – кандидат технических наук, доцент, зав. кафедрой информационных и автоматизированных производственных систем ФГБОУ ВО «Кузбасский государственный технический университет имени Т. Ф. Горбачева»

Сыркин Илья Сергеевич

Тестирование информационных систем: методические указания к практическим занятиям и лабораторным работам [Электронный ресурс]: для обучающихся специальности СПО 09.02.07 Информационные системы и программирование очной формы обучения / сост. И. С. Сыркин; КузГТУ. – Электрон. издан. – Кемерово, 2018.

Методические материалы для дисциплины «Тестирование информационных систем» содержат перечень выполняемых практических и лабораторных работ, контрольные вопросы к ним.

Предисловие

Целью освоения дисциплины «Тестирование информационных систем» является приобретение обучающимися знаний в области проведения тестирования ПО в процессе его создания.

Основными задачами изучения дисциплины «Тестирование информационных систем», являются:

1. Изучение способов создания тестов.
2. Проведение тестов безопасности.
3. Изучение нагрузочного тестирования.

Содержание дисциплины в соответствии с учебным планом

В соответствии с учебным планом изучение дисциплины «Технологии разработки программного обеспечения» предусматривает проведение лекционных, практических занятий, лабораторных работ и самостоятельной работы обучающимися очной формы обучения.

Общая трудоемкость дисциплины составляет 142 часа.

Промежуточный контроль – экзамен (6 семестр).

Содержание практических занятий и лабораторных работ

При подготовке к практическим занятиям обучающиеся самостоятельно изучают основную и дополнительную литературу, готовят конспекты по темам, предложенным преподавателем.

На практических занятиях преподаватель осуществляет контроль подготовки качества знаний обучающегося, используя: опрос, обсуждение вопросов по темам изучаемой дисциплины, письменный опрос при текущем контроле и предоставление отчетов по практическим занятиям.

1. Практическое занятие №1. Разработка тестовых сценариев

Целью работы является изучение способов разработки тестов ПО. Результатом практической работы является отчет, в котором должны быть приведены анализ предметной области и требований к системе.

Для выполнения практической работы № 1 студент должен изучить приведенный ниже теоретический материал. Отчет сдается в распечатанном и электронном (файл Word) видах.

1.1. Разработка тестовых сценариев

1. Создайте проект для тестирования

1. Запустите Visual Studio.
2. В меню **Файл** выберите пункт **Создать** > **Проект**.

Откроется диалоговое окно **Новый проект** .

3. В области **Установленные шаблоны**выберите шаблон **Visual C#.**
4. В списке типов приложения выберите пункт **Библиотека классов**.
5. В поле **Имя** введите **Bank** и нажмите **OK**.

Будет создан новый проект Bank. Этот проект отобразится в **обозревателе решений**, а его файл Class1.cs откроется в редакторе кода.

Примечание

Если файл Class1.cs не откроется в редакторе кода, дважды щелкните Class1.cs в **обозревателе решений**, чтобы открыть его.

6. Скопируйте исходный текст из раздела Пример проекта для создания модульных тестов и замените скопированным текстом исходное содержимое файла Class1.cs.
7. Сохранение файла как BankAccount.cs.
8. В меню **Сборка** выберите **Собрать решение**.

Будет создан проект с именем «Bank». Он содержит исходный код, подлежащий тестированию, и средства для его тестирования. Пространство имен BankAccountNS проекта «Bank», содержит открытый класс «BankAccount», методы которого будут тестироваться в приведенных ниже процедурах.

В этой статье проводится тестирование на примере метода Debit. Метод Debit вызывается, когда денежные средства снимаются со счета. Так выглядит определение метода:

```
// Method to be tested.
public void Debit(double amount)
{
   if(amount > m_balance)
   {
      throw new ArgumentOutOfRangeException(«amount»);
   }
   if (amount < 0)
   {
      throw new ArgumentOutOfRangeException(«amount»);
   }
   m_balance += amount;
}
```

2. Создание проекта модульного теста

3. В меню **Файл** выберите **Добавить** > **Создать проект**.

4. В диалоговом окне **Новый проект** разверните узлы **Установленные** и **Visual C#** и выберите **Тест**.

5. В списке шаблонов выберите **Проект модульного теста**.

6. В поле **Имя** введите BankTests, а затем нажмите кнопку **ОК**.

Проект **BankTests** добавляется в решение **Банк**.

7. В проекте **BankTests** добавьте ссылку на проект **Банк**.

В **обозревателе решений** щелкните **Ссылки** в проекте **BankTests**, а затем выберите в контекстном меню **Добавить ссылку**.

8. В диалоговом окне **Диспетчер ссылок** разверните **Решение** и проверьте элемент **Банк**.

9. Создание тестового класса

Создание тестового класса, чтобы проверить класс BankAccount. Можно использовать UnitTest1.cs, созданный в шаблоне проекта, но лучше дать файлу и классу более описательные имена. Можно сделать это за один шаг, переименовав файл в **обозревателе решений**.

10. Переименование файла класса

В **обозревателе решений** выберите файл UnitTest1.cs в проекте BankTests. В контекстном меню выберите команду **Переименовать**, а затем переименуйте файл в BankAccountTests.cs. Выберите **Да** в диалоговом окне, предлагающем переименовать все ссылки на элемент кода UnitTest1 в проекте.

Этот шаг изменяет имя класса на BankAccountTests. Файл BankAccountTests.cs теперь содержит следующий код:

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

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

11. Добавление оператора using в тестируемый проект

Можно также добавить оператор using в класс, чтобы тестируемый проект можно было вызывать без использования полных имен. Вверху файла класса добавьте:

C#Копировать

```
using BankAccountNS;
```

12. Требования к тестовому классу

Минимальные требования к тестовому классу следующие:

- Атрибут [TestClass] является обязательным для платформы модульных тестов Microsoft для управляемого кода в любом классе, содержащем методы модульных тестов, которые необходимо выполнить в обозревателе тестов.
- Каждый метод теста, предназначенный для запуска в обозревателе тестов, должен иметь атрибут [TestMethod].

Можно иметь другие классы в проекте модульного теста, которые не содержат атрибута [TestClass], а также иметь другие методы в тестовых классах, у которых атрибут – [TestMethod]. Можно использовать эти другие классы и методы в методах теста.

13. Создание первого тестового метода

В этой процедуре мы напишем методы модульного теста для проверки поведения метода Debit класса BankAccount. Метод Debit приведен выше в этой статье.

Существует по крайней мере три поведения, которые требуется проверить:

- Метод создает исключение ArgumentOutOfRangeException, если сумма по дебету превышает баланс.
- Метод создает исключение ArgumentOutOfRangeException, если сумма по дебету меньше нуля.
- Если значение дебета допустимо, то метод вычитает сумму дебета из баланса счета.

Совет

Метод по умолчанию TestMethod1 можно удалять, так как он не используется в этом руководстве.

14. Создание метода теста

Первый тест проверяет, снимается ли со счета нужная сумма при допустимом размере кредита (со значением меньшим, чем баланс счета, и большим, чем ноль). Добавьте следующий метод в этот класс BankAccountTests :

```
[TestMethod]
public void Debit_WithValidAmount_UpdatesBalance()
{
   // Arrange
   double beginningBalance = 11.99;
   double debitAmount = 4.55;
   double expected = 7.44;
   BankAccount account = new BankAccount(«Mr. Bryan Walton», beginningBalance);

   // Act
   account.Debit(debitAmount);

   // Assert
   double actual = account.Balance;
   Assert.AreEqual(expected, actual, 0.001, «Account not debited correctly»);
}
```

Метод очень прост: он создает новый объект BankAccount с начальным балансом, а затем снимает допустимое значение. Он использует метод AreEqual, чтобы проверить, что конечный баланс соответствует ожидаемому.

15. Требования к методу теста

Метод теста должен удовлетворять следующим требованиям:

- Он декорируется атрибутом [TestMethod].
- Он возвращает void.
- Он не должен иметь параметров.

16. Сборка и запуск теста

1. В меню **Построение** выберите **Построить решение**.

Если ошибок нет, появится **обозреватель тестов** с элементом **Debit_WithValidAmount_UpdatesBalance** в группе **Незапускавшиеся тесты**.

Совет

Если **обозреватель тестов** не откроется после успешной сборки, выберите в меню пункт **Тест**, щелкните **Windows**, а затем – **Обозреватель тестов**.

2. Выберите **Запустить все**, чтобы выполнить тест. Во время выполнения теста в верхней части окна 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.

3. В данном случае тест пройден не будет. Метод теста будет перемещен в группу **Неудачные тесты.** Выберите этот метод в **обозревателе тестов** для просмотра сведений в нижней части окна.

17. Исправление кода и повторный запуск тестов

18. Анализ результатов теста

Результат теста содержит сообщение, описывающее возникшую ошибку. Для метода AreEqual сообщение отражает ожидаемый результат (параметр **Ожидается<значение>**) и фактически полученный (параметр **Фактическое<значение>**). Ожидалось, что баланс уменьшится, а вместо этого он увеличился на сумму списания.

Модульный тест обнаружил ошибку: сумма списания добавляется на баланс счета, вместо того чтобы вычитаться.

19. Исправление ошибки

Для исправления ошибки замените строку:

```
m_balance += amount;
```

на:

C#Копировать

```
m_balance -= amount;
```

20. Повторный запуск теста

В **обозревателе тестов** выберите **Запустить все**, чтобы запустить тест повторно. Красно-зеленая строка состояния станет зеленой, сигнализируя о том, что тест пройден, а сам тест будет перемещен в группу **Пройденные тесты.**

21. Использование модульных тестов для улучшения кода

В этом разделе рассматривается, как последовательный процесс анализа, разработки модульных тестов и рефакторинга может помочь сделать рабочий код более надежным и эффективным.

22. Анализ проблем

Мы создали тестовый метод для подтверждения того, что допустимая сумма правильно вычитается в методе Debit. Теперь проверим, что метод создает исключение ArgumentOutOfRangeException, если сумма по дебету:

- больше баланса или
- меньше нуля.

23. Создание методов теста

Создадим метод теста для проверки правильного поведения в случае, когда сумма по дебету меньше нуля:

```
[TestMethod]
[ExpectedException(typeof(ArgumentOutOfRangeException))]
public void Debit_WhenAmountIsLessThanZero_ShouldThrowArgumentOutOfRange()
{
   // Arrange
   double beginningBalance = 11.99;
   double debitAmount = -100.00;
   BankAccount account = new BankAccount(«Mr. Bryan Walton», beginningBalance);

   // Act
   account.Debit(debitAmount);

   // Assert is handled by the ExpectedException attribute on the test method.
}
```

Мы используем атрибут ExpectedExceptionAttribute для подтверждения правильности созданного исключения. Данный атрибут приводит к тому, что тест не будет пройден, если не возникнет ис-

ключения ArgumentOutOfRangeException. Если временно изменить тестируемый метод для вызова более общего исключения ApplicationException при значении суммы по дебету меньше нуля, то тест работает правильно – то есть завершается неудачно.

Чтобы проверить случай, когда размер списания превышает баланс, выполните следующие действия:

1. Создать новый метод теста с именем Debit_WhenAmountIsMoreThanBalance_ShouldThrowArgumentOutOfRange.
2. Скопировать тело метода из Debit_WhenAmountIsLessThanZero_ShouldThrowArgumentOutOfRange в новый метод.
3. Присвоить debitAmount значение, превышающее баланс.

24. Запуск тестов

Запуск двух методов теста показывает, что тесты работают правильно.

25. Продолжение анализа

Однако последние два тестовых метода вызывают беспокойство. Нельзя быть уверенным, какое именно условие тестируемого метода создает исключение при запуске любого из тестов. Если каким-либо способом разделить эти два условия, а именно отрицательную сумму по дебету и сумму, большую, чем баланс, то это увеличит достоверность проведения тестов.

Еще раз посмотрев на тестируемый метод и заметив, что оба условных оператора используют конструктор ArgumentOutOfRangeException, который просто получает имя аргумента в качестве параметра:

C#Копировать

throw new ArgumentOutOfRangeException(«amount»);

Так выглядит конструктор, который можно использовать для сообщения более детальной информации: ArgumentOutOfRangeException(String, Object, String) включает имя аргумента, значения аргумента и определяемое пользователем сообщение. Мы можем выполнить рефакторинг тестируемого метода для использования данного конструктора. Более того, можно использовать открытые для общего доступа члены типа для указания ошибок.

26. Рефакторинг тестируемого кода

Сначала определим две константы для сообщений об ошибках в области видимости класса. Добавьте это в тестируемый класс BankAccount:

```
public const string DebitAmountExceedsBalanceMessage = «Debit amount exceeds balance»;
public const string DebitAmountLessThanZeroMessage = «Debit amount is less than zero»;
```

Затем изменим два условных оператора в методе Debit:

```
C#Копировать
  if (amount > m_balance)
  {
    throw new ArgumentOutOfRangeException(«amount», amount, DebitAmountExceedsBalanceMessage);
  }

  if (amount < 0)
  {
    throw new ArgumentOutOfRangeException(«amount», amount, DebitAmountLessThanZeroMessage);
  }
```

27. Рефакторинг тестовых методов

Удалим атрибут ExpectedException метода теста, и вместо этого будем перехватывать исключение и проверять соответствующее ему сообщение. Метод StringAssert.Contains обеспечивает возможность сравнения двух строк.

В этом случае метод Debit_WhenAmountIsMoreThanBalance_ShouldThrowArgumentOutOfRange может выглядеть следующим образом:

```
 [TestMethod]
public void DebitWhenAmountIsMoreThanBalance_ShouldThrowArgumentOutOfRange()
{
  // Arrange
  double beginningBalance = 11.99;
  double debitAmount = 20.0;
  BankAccount account = new BankAccount(«Mr. Bryan Walton», beginningBalance);
```

```
        // Act
        try
        {
            account.Debit(debitAmount);
        }
        catch (ArgumentOutOfRangeException e)
        {
            // Assert
            StringAssert.Contains(e.Message, BankAc-
count.DebitAmountExceedsBalanceMessage);
        }
    }
```

28. Повторное тестирование, переписывание и анализ

Предположим, что в тестируемом методе есть ошибка, и метод Debit даже не создает исключение ArgumentOutOfRangeException, не говоря уже о выводе правильного сообщения с исключением. В этом случае метод теста не сможет обработать этот случай. Если значение debitAmount допустимо (то есть меньше баланса, но больше нуля), то исключение не перехватывается, а утверждение никогда не сработает. Однако метод теста проходит успешно. Это нехорошо, поскольку метод теста должен был завершиться с ошибкой в том случае, если исключение не создается.

Это является ошибкой в методе теста. Для решения этой проблемы добавим утверждение Fail в конце тестового метода для обработки случая, когда исключение не создается.

Однако повторный запуск теста показывает, что тест теперь оказывается непройденным при перехватывании верного исключения. Блок catch перехватывает исключение, но метод продолжает выполняться, и в нем происходит сбой на новом утверждении Fail. Чтобы разрешить эту проблему, добавим оператор return после StringAssert в блоке catch. Повторный запуск теста подтверждает, что проблема устранена. Окончательная версия метода Debit_WhenAmountIsMoreThanBalance_ShouldThrowArgumentOutOfRange выглядит следующим образом:

```
    [TestMethod]
    public void De-
bit_WhenAmountIsMoreThanBalance_ShouldThrowArgumentOutOfRange()
    {
        // Arrange
```

```
        double beginningBalance = 11.99;
        double debitAmount = 20.0;
        BankAccount account = new BankAccount(«Mr. Bryan Walton», begin-
ningBalance);

        // Act
        try
        {
          account.Debit(debitAmount);
        }
        catch (ArgumentOutOfRangeException e)
        {
          // Assert
          StringAssert.Contains(e.Message, BankAc-
count.DebitAmountExceedsBalanceMessage);
          return;
        }

        Assert.Fail(«The expected exception was not thrown.»);
      }
```

Усовершенствования тестового кода привели к созданию более надежных и информативных методов теста. Но что более важно, в результате был также улучшен тестируемый код.

Контрольные вопросы

1. Расскажите порядок создания модульного теста в VS.
2. Что такое рефакторинг кода?
3. Как провести рефакторинг, используя модульные тесты?

2. Практическое занятие №2. Обработка исключительных ситуаций

Целью работы является изучение способов обработки исключительных ситуаций. Результатом практической работы является отчет, в котором должны быть приведены исходные коды программы, демонстрирующей умение обрабатывать исключения.

Для выполнения практической работы № 2 студент должен изучить приведенный ниже теоретический материал. Отчет сдается в распечатанном и электронном (файл Word) видах.

2.1. Основы обработки исключений

Далеко не всегда ошибки случаются по вине того, кто кодирует приложение. Иногда приложение генерирует ошибку из-за действий конечного пользователя, или же ошибка вызвана контекстом среды, в которой выполняется код. В любом случае вы всегда должны ожидать возникновения ошибок в своих приложениях и проводить кодирование в соответствии с этими ожиданиями.

В .NET Framework предусмотрена развитая система обработки ошибок. Механизм обработки ошибок C# позволяет закодировать пользовательскую обработку для каждого типа ошибочных условий, а также отделить код, потенциально порождающий ошибки, от кода, обрабатывающего их.

Что бы ни служило причиной проблем, в конечном итоге приложение начинает работать не так, как ожидается. Прежде чем переходить к рассмотрению структурированной обработки исключений, давайте сначала ознакомимся с тремя наиболее часто применяемыми для описания аномалий терминами:

Программные ошибки (bugs)

Так обычно называются ошибки, которые допускает программист. Например, предположим, что приложение создается с помощью неуправляемого языка C++. Если динамически выделяемая память не освобождается, что чревато утечкой памяти, появляется программная ошибка.

Пользовательские ошибки (user errors)

В отличие от программных ошибок, пользовательские ошибки обычно возникают из-за тех, кто запускает приложение, а не тех, кто его создает. Например, ввод конечным пользователем в текстовом поле неправильно оформленной строки может привести к гене-

рации ошибки подобного рода, если в коде не была предусмотрена возможность обработки некорректного ввода.

Исключения (exceptions)

Исключениями, или исключительными ситуациями, обычно называются аномалии, которые могут возникать во время выполнения и которые трудно, а порой и вообще невозможно, предусмотреть во время программирования приложения. К числу таких возможных исключений относятся попытки подключения к базе данных, которой больше не существует, попытки открытия поврежденного файла или попытки установки связи с машиной, которая в текущий момент находится в автономном режиме. В каждом из этих случаев программист (и конечный пользователь) мало что может сделать с подобными «исключительными» обстоятельствами.

По приведенным выше описаниям должно стать понятно, что структурированная обработка исключений в .NET представляет собой методику, предназначенную для работы с исключениями, которые могут возникать на этапе выполнения. Даже в случае программных и пользовательских ошибок, которые ускользнули от глаз программиста, однако, CLR будет часто автоматически генерировать соответствующее исключение с описанием текущей проблемы. В библиотеках базовых классов .NET определено множество различных исключений, таких как FormatException, IndexOutOfRange Exception, FileNotFoundException, ArgumentOutOfRangeException и т. д.

В терминологии .NET под «исключением» подразумеваются программные ошибки, пользовательские ошибки и ошибки времени выполнения. Прежде чем погружаться в детали, давайте посмотрим, какую роль играет структурированная обработка исключений, и чем она отличается от традиционных методик обработки ошибок.

2.2. Роль обработки исключений в .NET

До появления .NET обработка ошибок в среде операционной системы Windows представляла собой весьма запутанную смесь технологий. Многие программисты включали собственную логику обработки ошибок в контекст интересующего приложения. Например, команда разработчиков могла определять набор числовых констант для представления известных сбойных ситуаций и затем применять эти константы в качестве возвращаемых значений методов.

Помимо приемов, изобретаемых самими разработчиками, в API-интерфейсе Windows определены сотни кодов ошибок с помощью #define и HRESULT, а также множество вариаций простых булевских значений (bool, BOOL, VARIANT BOOL и т. д.). Более того, многие разработчики COM-приложений на языке C++ (а также VB 6) явно или неявно применяют небольшой набор стандартных COM-интерфейсов (наподобие ISupportErrorlnfo. IErrorlnfo или ICreateErrorlnfo) для возврата COM-клиенту понятной информации об ошибках.

Очевидная проблема со всеми этими более старыми методиками – отсутствие симметрии. Каждая из них более-менее вписывается в рамки какой-то одной технологии, одного языка и, пожалуй, даже одного проекта. В .NET поддерживается стандартная методика для генерации и выявления ошибок в исполняющей среде, называемая **структурированной обработкой исключений (SEH – structured exception handling)**.

Прелесть этой методики состоит в том, что она позволяет разработчикам использовать в области обработки ошибок унифицированный подход, который является общим для всех языков, ориентированных на платформу .NET. Благодаря этому, программист на C# может обрабатывать ошибки почти таким же с синтаксической точки зрения образом, как и программист на VB и программист на C++, использующий C++/CLI.

Дополнительное преимущество состоит в том, что синтаксис, который требуется применять для генерации и перехвата исключений за пределами сборок и машин, тоже выглядит идентично. Например, при написании на C# службы Windows Communication Foundation (WCF) генерировать исключение SOAP для удаленного вызывающего кода можно с использованием тех же ключевых слов, которые применяются для генерации исключения внутри методов в одном и том же приложении.

Еще одно преимущество механизма исключений .NET состоит в том, что в отличие от запутанных числовых значений, просто обозначающих текущую проблему, они представляют собой объекты, в которых содержится читабельное описание проблемы, а также детальный снимок стека вызовов на момент, когда изначально возникло исключение. Более того, конечному пользователю можно предоставлять справочную ссылку, которая указывает на опреде-

ленный URL-адрес с описанием деталей ошибки, а также специальные данные, определенные программистом.

2.3. Составляющие процесса обработки исключений в .NET

Программирование со структурированной обработкой исключений подразумевает использование четырех следующих связанных между собой сущностей:

- тип класса, который представляет детали исключения;
- член, способный генерировать (throw) в вызывающем коде экземпляр класса исключения при соответствующих обстоятельствах;
- блок кода на вызывающей стороне, ответственный за обращение к члену, в котором может произойти исключение;
- блок кода на вызывающей стороне, который будет обрабатывать (или перехватывать (catch)) исключение в случае его возникновения.

2.4. Перехват исключений

Принимая во внимание, что .NET Framework включает большое количество предопределенных классов исключений, возникает вопрос: как их использовать в коде для перехвата ошибочных условий? Для того чтобы справиться с возможными ошибочными ситуациями в коде C#, программа обычно делится на блоки трех разных типов:

- Блоки **try** инкапсулируют код, формирующий часть нормальных действий программы, которые потенциально могут столкнуться с серьезными ошибочными ситуациями.
- Блоки **catch** инкапсулируют код, который обрабатывает ошибочные ситуации, происходящие в коде блока try. Это также удобное место для протоколирования ошибок.
- Блоки **finally** инкапсулируют код, очищающий любые ресурсы или выполняющий другие действия, которые обычно нужно выполнить в конце блоков try или catch. Важно понимать, что этот блок выполняется независимо от того, сгенерировано исключение или нет.

1 Try и catch

Основу обработки исключительных ситуаций в C# составляет пара ключевых слов try и catch. Эти ключевые слова действуют совместно и не могут быть использованы порознь. Ниже приведена

общая форма определения блоков try/catch для обработки исключительных ситуаций:

```
try {
// Блок кода, проверяемый на наличие ошибок.
}

catch (ExcepType1 exOb) {
// Обработчик исключения типа ExcepType1.
}
catch (ExcepType2 exOb) {
// Обработчик исключения типа ExcepType2.
}
...
```

где ExcepType – это тип возникающей исключительной ситуации. Когда исключение генерируется оператором try, оно перехватывается составляющим ему пару оператором catch, который затем обрабатывает это исключение. В зависимости от типа исключения выполняется и соответствующий оператор catch. Так, если типы генерируемого исключения и того, что указывается в операторе catch, совпадают, то выполняется именно этот оператор, а все остальные пропускаются. Когда исключение перехватывается, переменная исключения exOb получает свое значение. На самом деле указывать переменную exOb необязательно. Так, ее необязательно указывать, если обработчику исключений не требуется доступ к объекту исключения, что бывает довольно часто. Для обработки исключения достаточно и его типа.

Следует, однако, иметь в виду, что если исключение не генерируется, то блок оператора try завершается как обычно, и все его операторы catch пропускаются. Выполнение программы возобновляется с первого оператора, следующего после завершающего оператора catch. Таким образом, оператор catch выполняется лишь в том случае, если генерируется исключение.

Давайте рассмотрим пример, в котором будем обрабатывать исключение, возникающее при делении числа на 0:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

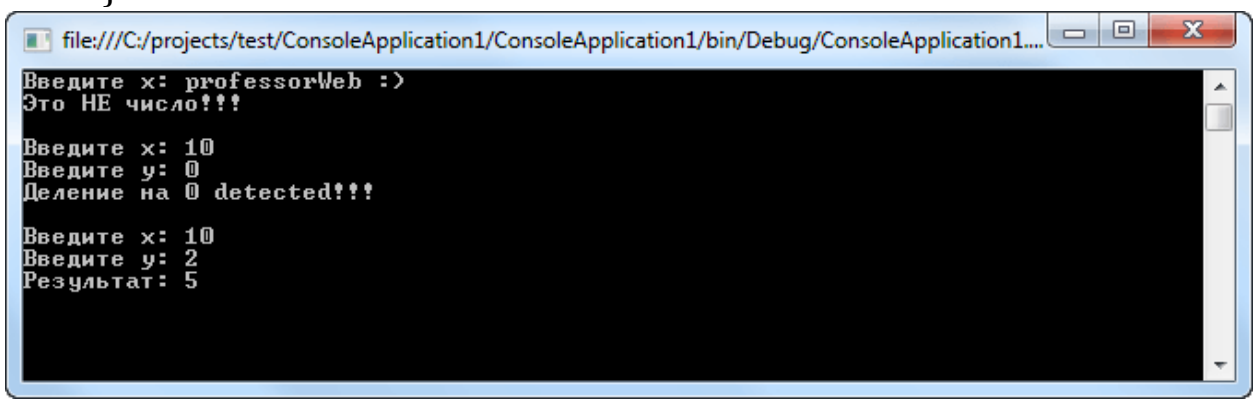
```
namespace ConsoleApplication1
{
   class Program
   {
      static int MyDel(int x, int y)
      {
         return x / y;
      }

      static void Main()
      {
         try
         {
            Console.Write(«Введите x: «);
            int x = int.Parse(Console.ReadLine());
            Console.Write(«Введите y: «);
            int y = int.Parse(Console.ReadLine());

            int result = MyDel(x, y);
            Console.WriteLine(«Результат: « + result);
         }
         // Обрабатываем исключение, возникающее при делении на ноль
         catch (DivideByZeroException)
         {
            Console.WriteLine(«Деление на 0 detected!!!\n»);
            Main();
         }
         // Обрабатываем исключение при неккоректном вводе числа в консоль
         catch (FormatException)
         {
            Console.WriteLine(«Это НЕ число!!!\n»);
            Main();
         }

         Console.ReadLine();
      }
```

```
        }
    }
```

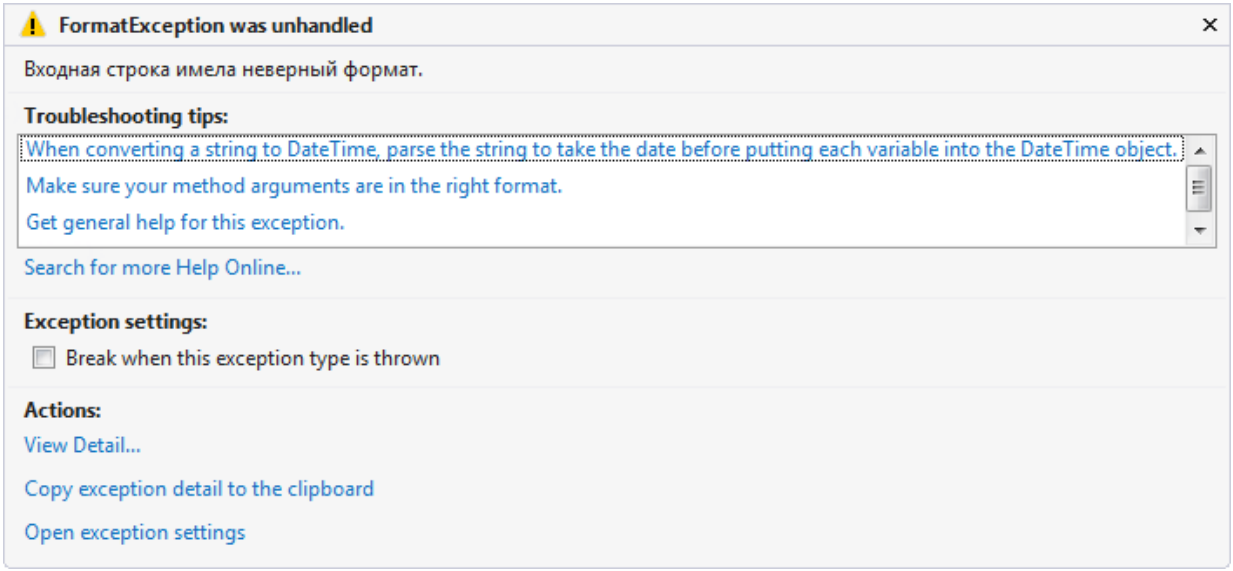


Данный простой пример наглядно иллюстрирует обработку исключительной ситуации при делении на 0 (DivideByZeroException), а также пользовательскую ошибку при вводе не числа (FormatException).

2 Последствия неперехвата исключений

Перехват одного из стандартных исключений, как в приведенном выше примере, дает еще одно преимущество: он исключает аварийное завершение программы. Как только исключение будет сгенерировано, оно должно быть перехвачено каким-то фрагментом кода в определенном месте программы. Вообще говоря, если исключение не перехватывается в программе, то оно будет перехвачено исполняющей системой. Но дело в том, что исполняющая система выдаст сообщение об ошибке и прервет выполнение программы.

Например, если убрать из предыдущего примера исключение FormatException, то при вводе неккоректной строки, IDE-среда VisualStudio выдаст предупреждающее сообщение:



Такие сообщения об ошибках полезны для отладки программы, но, по меньшей мере, нежелательны при ее использовании на практике! Именно поэтому так важно организовать обработку исключительных ситуаций в самой программе.

Контрольные вопросы

1. Что такое исключительная ситуация?
2. Как можно обработать исключительную ситуацию?

3. Практическое занятие №3. «Анализ результатов тестирования»

Целью работы является изучение способов анализа результатов тестирования ПО. Результатом практической работы является отчет, в котором должны быть приведены исходные коды программы, тесты и их результаты.

Для выполнения практической работы № 3 студент должен изучить приведенный ниже теоретический материал. Отчет сдается в распечатанном и электронном (файл Word) видах.

Оценка выполненных тестов (Evaluation of the tests performed)

Метрики покрытия/глубины тестирования (Coverage/ thoroughness measures) Критерии «адекватности» тестирования, в ряде случаев, требуют систематического выполнения тестов для определенных набора элементов программы, задаваемых ее архитектурой или спецификацией. Соответствующие метрики позволяют оценить степень охвата характеристик системы (например, процент различных тестируемых параметров производительности) и глубину их детализации (например, случайное тестирование параметров производительности или с учетом граничных значений и т. п.). Такие метрики помогают прогнозировать вероятностное достижение заданных параметров качества системы.

Введение искусственных дефектов (Fault seeding) «Своими руками?! Никогда! ...» – такова, обычно, первая реакция на идею искусственного внесения дефектов, например, в программный код. На практике, этот подход помогает классифицировать возможные ошибки и следующие за ними сбои, применяя в дальнейшем полученные результаты для моделирования (пусть, часто, и интуитивного) возможных причин реальных сбоев, обнаруженных в процессе тестирования. Безусловно, данная техника должна использоваться с максимальной осторожностью опытными специалистами, хорошо представляющими общую архитектуру тестируемой программной системы и разбирающимися во её внутренних связях.

Оценка мутаций (Mutation score) Получаемое в процессе тестирования мутаций отношение «убитых» к общему числу сгенерированных мутантов помогает измерить эффективность выполняемых тестов. В силу специфики такой техники тестирования, количественные оценки мутаций имеют практическое значение только для определенных типов систем.

Сравнение о относительная эффективность различных техник тестирования (Comparison and relative effectiveness of different techniques).

Различные исследования в области тестирования связаны с попытками сравнения (с точки зрения достигаемого качества продукта) разных подходов к тестированию. Когда мы говорим об «эффективности» тестирования надо чётко договориться, то именно мы подразумеваем под эффективностью, желательно, в количественном выражении. Возможные варианты интерпретации этого понятия – число тестов (данной техники), необходимых для обнаружения первого дефекта; отношение количества всех обнаруженных дефектов к дефектам, найденным с применением заданного подхода и т. п. Только обладая такого рода данными можно говорить о корректности сравнения и оценки эффективности.

Контрольные вопросы

1. Что такое метрики тестирования?
2. Как измеряется покрытие кода?

4. Лабораторная работа №1. Разработка тестового сценария

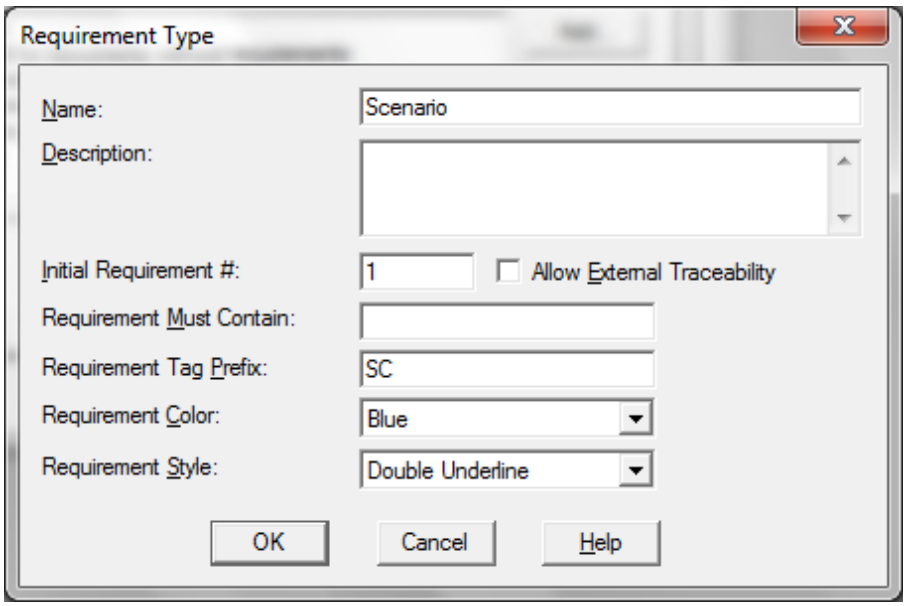
Целью работы является изучение порядка разработки тестового сценария. Результатом работы является отчет, в котором должны быть приведены исходные коды программы, тестовый сценарий.

Для выполнения работы студент должен изучить приведенный ниже теоретический материал. Отчет сдается в распечатанном и электронном (файл Word) видах.

Создание тестовых сценариев (Test Case) в RequisitePro

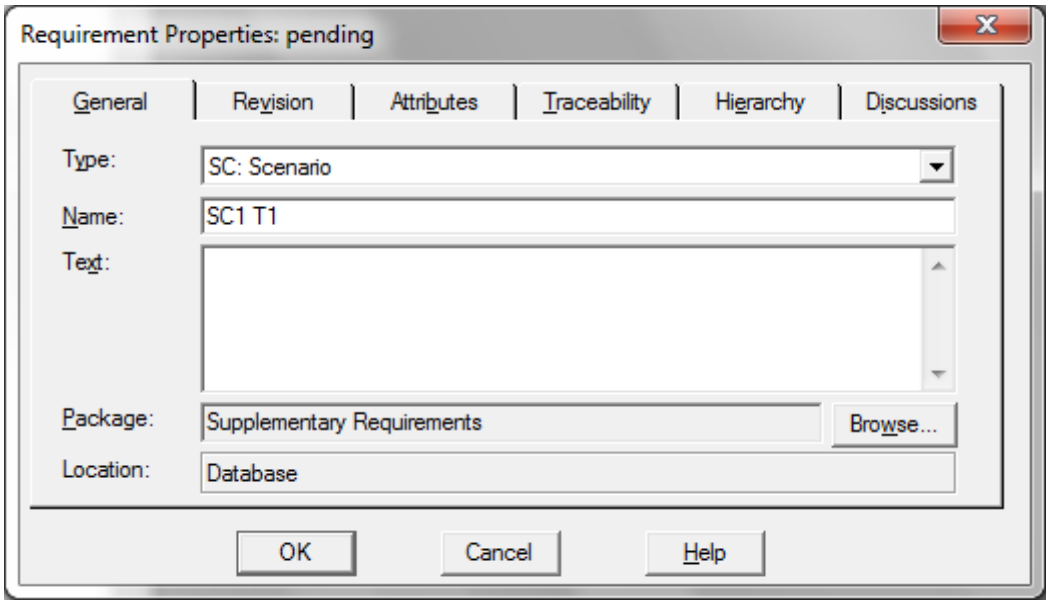
1. Создать новый тип требований:

конт. меню SpaceTravel → Properties → ф. Project Properties | вкл. Requirements Type | кн. Add → ф. Requirement Type | Name ← Scenario, Requirement Tag Prefix ← SC, остальные параметры – по умолчанию, кн. Ok → ф. Project Properties | кн. Ok

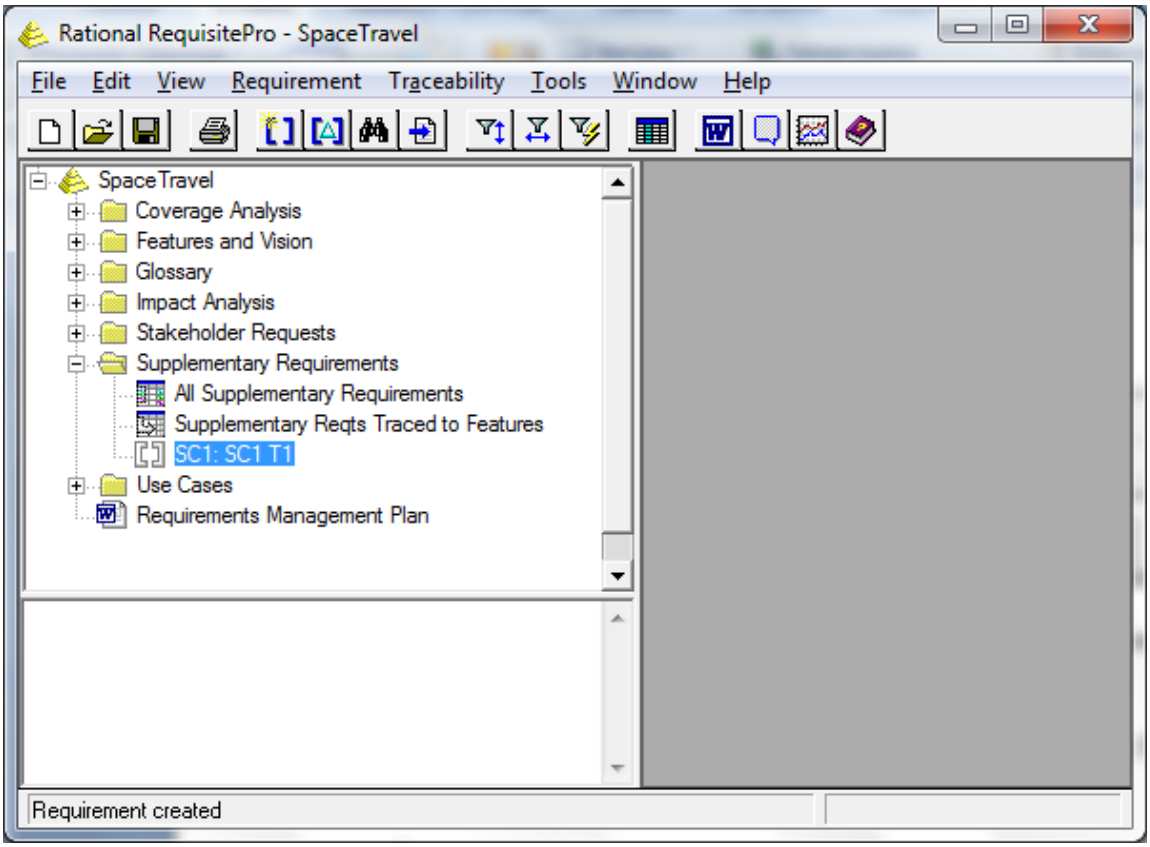


2. Создать новое требование:

SpaceTravel → Supplementary Requirements → конт. меню → New → Requirement… → ф. Requirement Properties: pending | Type ← SC: Scenario, Name ← SC1 T1 (номер сценария использования + номер тестового сценария), кн. Ok



Проверить наличие нового требования в окне проекта.

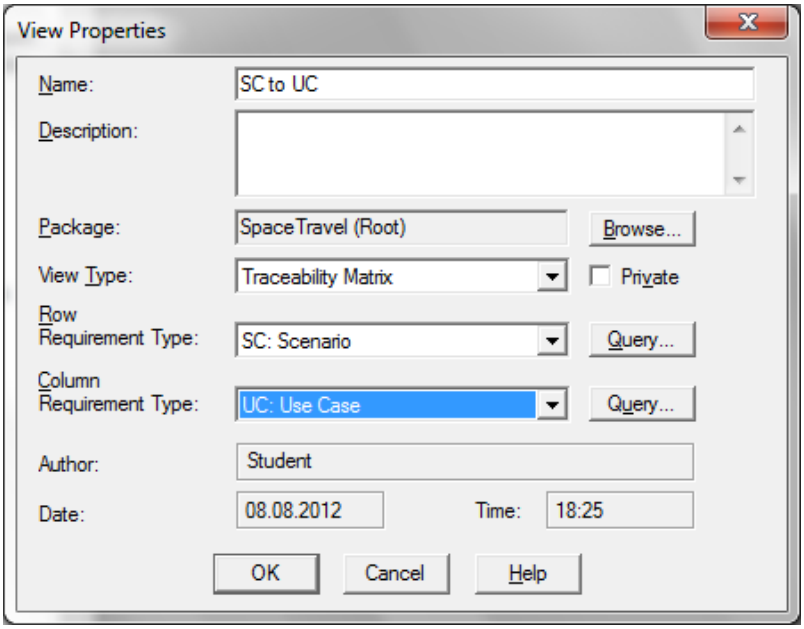


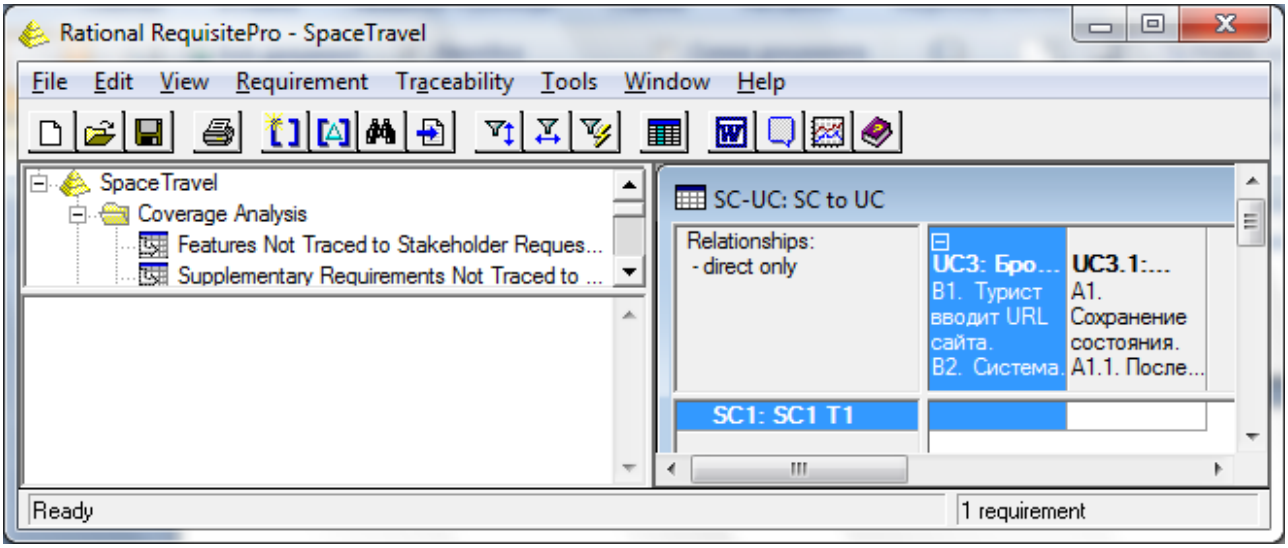
Аналогичным образом определить требования типа Scenario для остальных сценариев использования и тестовых сценариев.

3. Создать матрицу трассировки для связи требований Scenario с требованиями других типов:

3.1. Создать матрицу трассировки

ф. Rational RequisitePro – SpaceTravel → конт. меню → New → View… → ф. View Properties | Name ← SC to UC, View Type ← Traceability Matrix, Row Requirement Type ← SC: Scenario, Column Requirement Type ← UC: Use Case, кн. Ok → матрица трассировки на экране

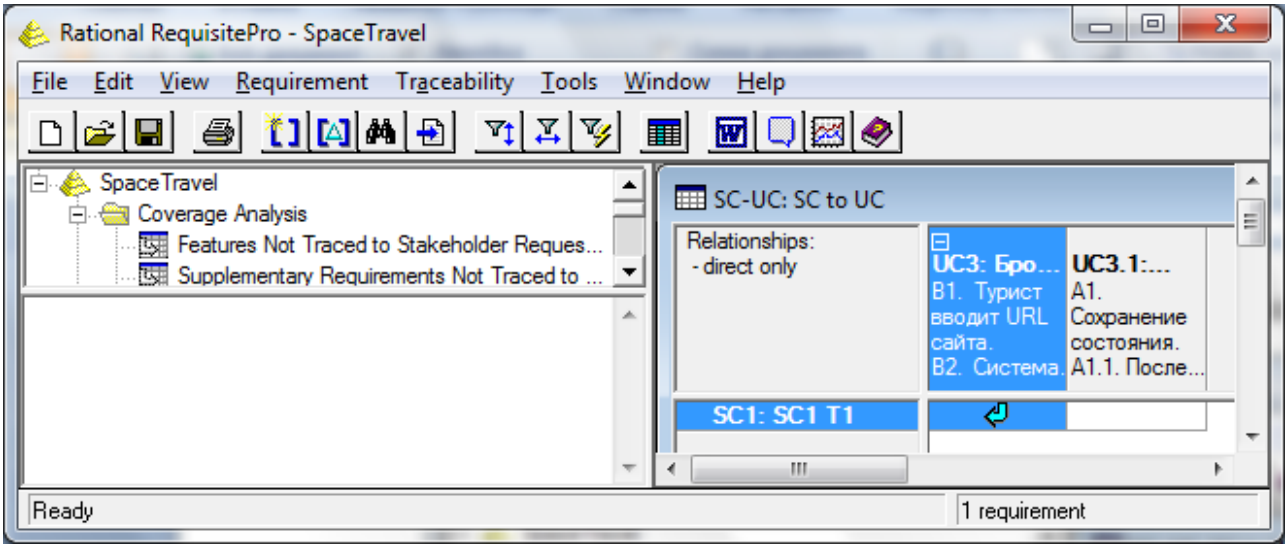




3.2. Установить связи между требованиями типа SC и UC

Сценарий использования (Use Case) и основанный на нем тестовый сценарий (Scenario) → установить курсор на пересечении строки, соответствующей тестовому сценарию, и столбца, соответ-

ствующего сценарию использования → контекстное меню → Trace From.

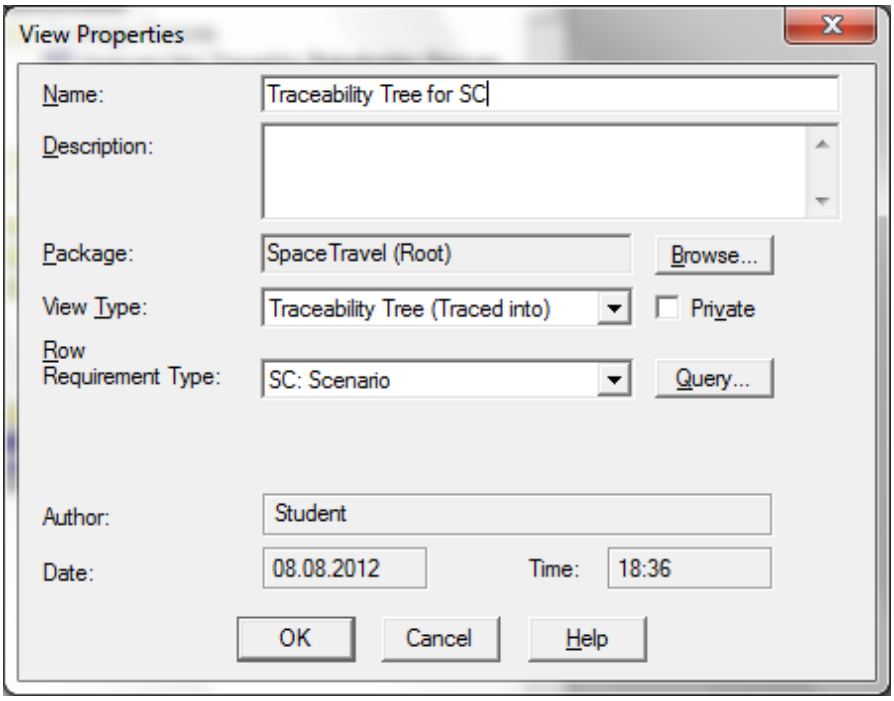


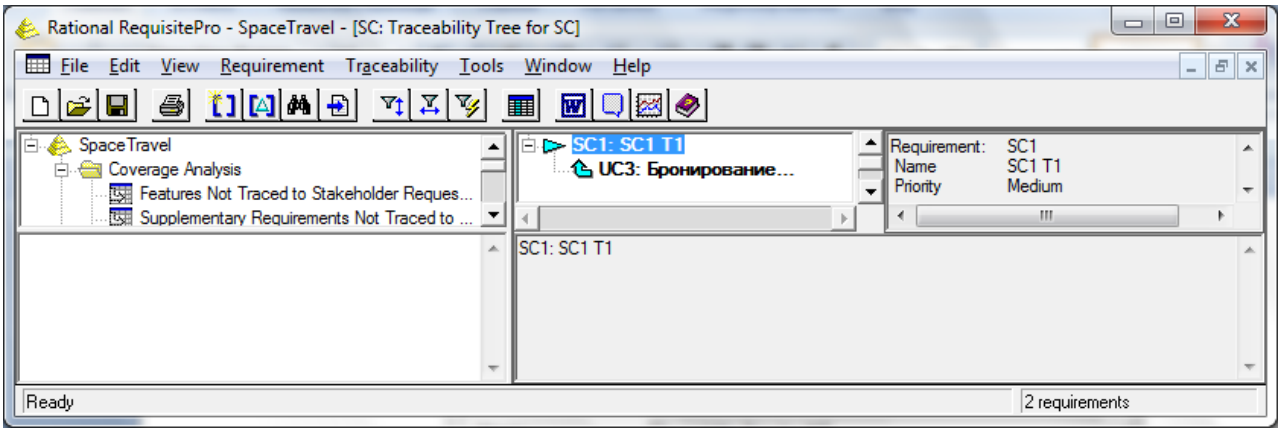
3.3. Аналогичным образом установить связи для всех тестовых сценариев (один сценарий использования может привести к нескольким тестовым сценариям)

3.4. Закрыть матрицу трассировки

4. Создать дерево трассировки

ф. Rational RequisitePro – SpaceTravel → конт. меню → New → View… → ф. View Properties | Name ← Traceability Tree for SC, View Type ← Traceability Tree (Traced into), Row Requirement Type ← SC: Scenario, кн. Ok → результат на экране





Контрольные вопросы

1. Приведите порядок составления тестового сценария.
2. Что такое матрица трассировки?
3. Как можно задать матрицу трассировки?

5. Лабораторная работа №2. Использование инструментария анализа качества

Целью работы является изучение инструментов анализа качества ПО. Результатом работы является отчет, в котором должны быть приведены исходные коды программы, и результаты анализа качества разработанной программы.

Для выполнения работы студент должен изучить приведенный ниже теоретический материал. Отчет сдается в распечатанном и электронном (файл Word) видах.

В основе модели качества SonarQube лежит реализация методологии SQALE (Software Quality Assessment based on Lifecycle Expectations) с определенными дополнениями. Как известно, методология SQALE фокусируется в основном на сложности поддержки кода (maintainability) и не учитывает риски проекта. Например, если сегодня в проекте обнаружилась критическая проблема безопасности, строгое следование методологии SQALE обязывает вас устранить все уже существующие проблемы с надежностью (reliability), возможностью изменений (changeability), тестируемостью (testability) и т. д., и только затем вернуться к новой критической проблеме. На самом деле, если потенциальные проблемы существуют в коде давно и не проявляют себя в виде пользовательских баг-репортов, гораздо важнее сфокусироваться на исправлении новых багов.

С учетом этого, разработчики SonarQube модифицировали модель качества, основанную на SQALE, чтобы акцентировать внимание на следующих важных моментах:

- Модель качества должна быть максимально простой в использовании.
- Баги и уязвимости не должны теряться среди проблем поддерживаемости (maintainability).
- Наличие серьезных багов и уязвимостей в проекте должно приводить к тому, что требования Quality Gate не выполнены.
- Проблемы поддерживаемости кода тоже важны, и их нельзя игнорировать.
- Вычисление стоимости устранения проблем (использование модели анализа SQALE) важно и должно выполняться.

Стандартный Quality Gate SonarQube использует следующие значения метрик для определения того, что код успешно прошел проверки:

- 0 новых багов.
- 0 новых уязвимостей.
- Коэффициент технического долга на новом коде <= 5%.
- Покрытие нового кода не ниже 80%.

Команда Sonar определила семь смертных грехов разработчиков, увеличивающих технический долг:

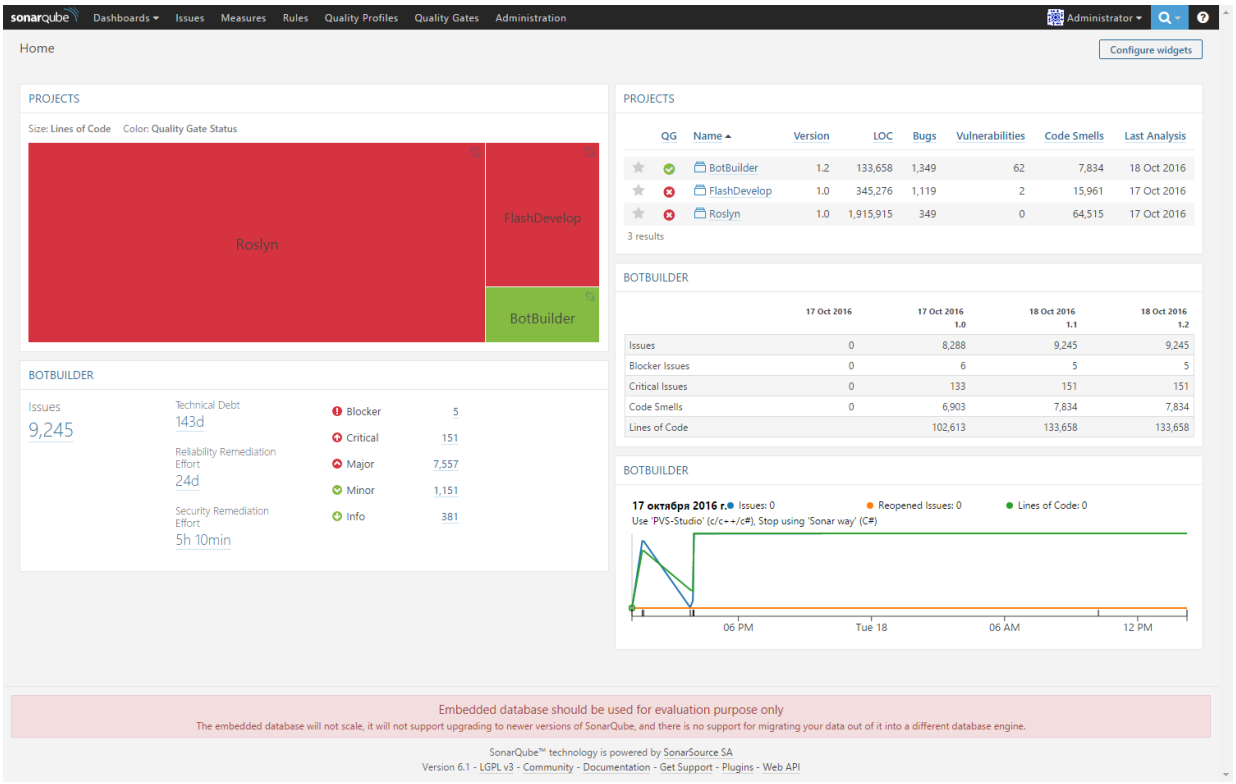
- Баги и потенциальные багги.
- Нарушение стандартов кодирования.
- Дублирование кода.
- Недостаточное покрытие модульными тестами.
- Плохое распределение сложности.
- Спагетти-дизайн.
- Недостаточно или слишком много комментариев.

Платформа SonarQube предназначена для того, чтобы помогать бороться с этими семью грехами.

Рассмотрим более подробно основные возможности SonarQube.

- Главная страница

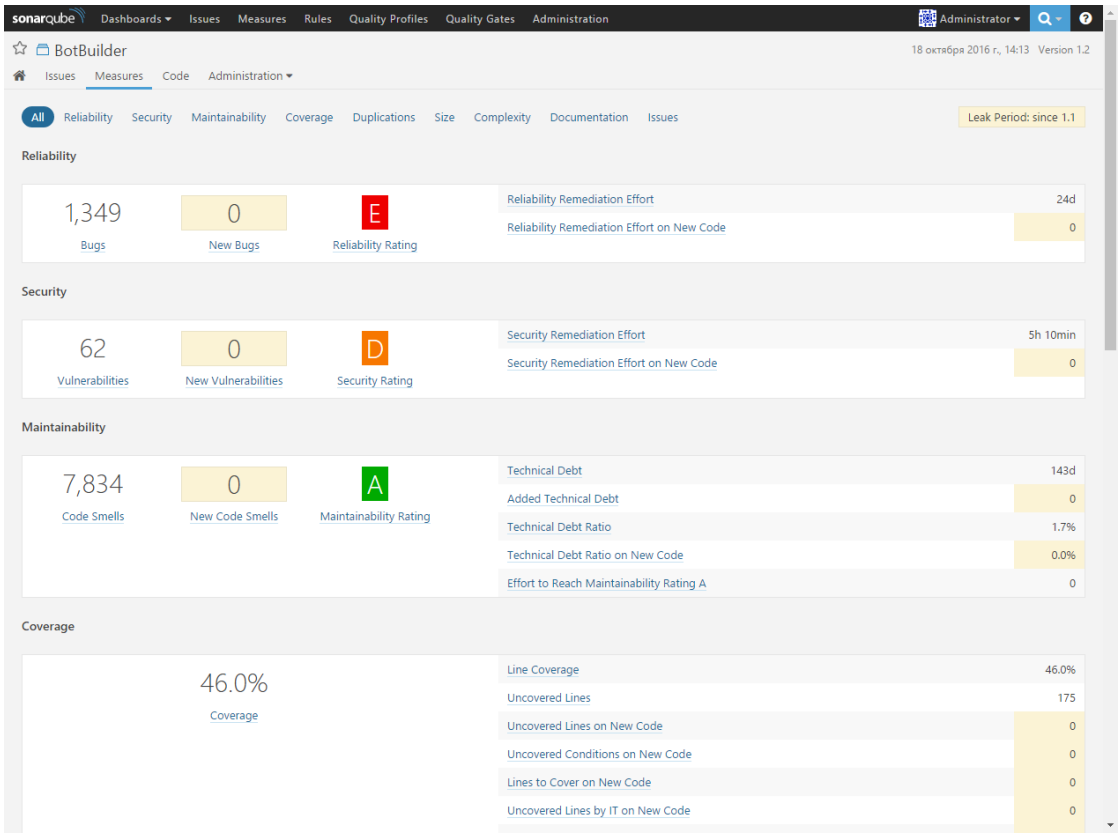
На главной странице SonarQube вы видите список проектов, добавленных в систему, с краткой статистикой по каждому проекту: версия сборки, количество строк кода, количество багов, уязвимостей и признаков «кода с душком», дата последнего анализа:



Содержимое главной страницы можно настроить под ваши цели с помощью большого набора встроенных виджетов, позволяющих визуализировать состояние кода проектов в SonarQube.

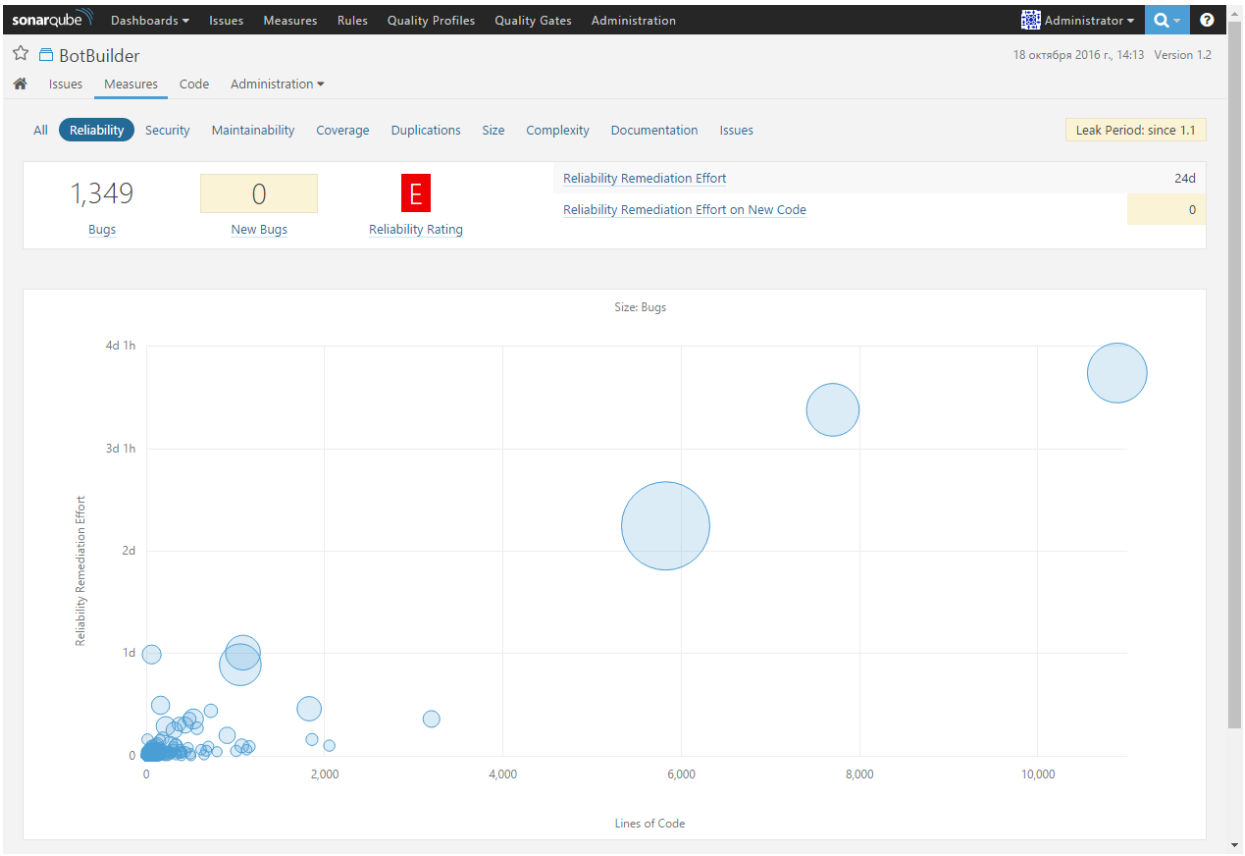
Метрики проекта

Для получения более детальной информации о состоянии проекта перейдем на страницу метрик проекта:

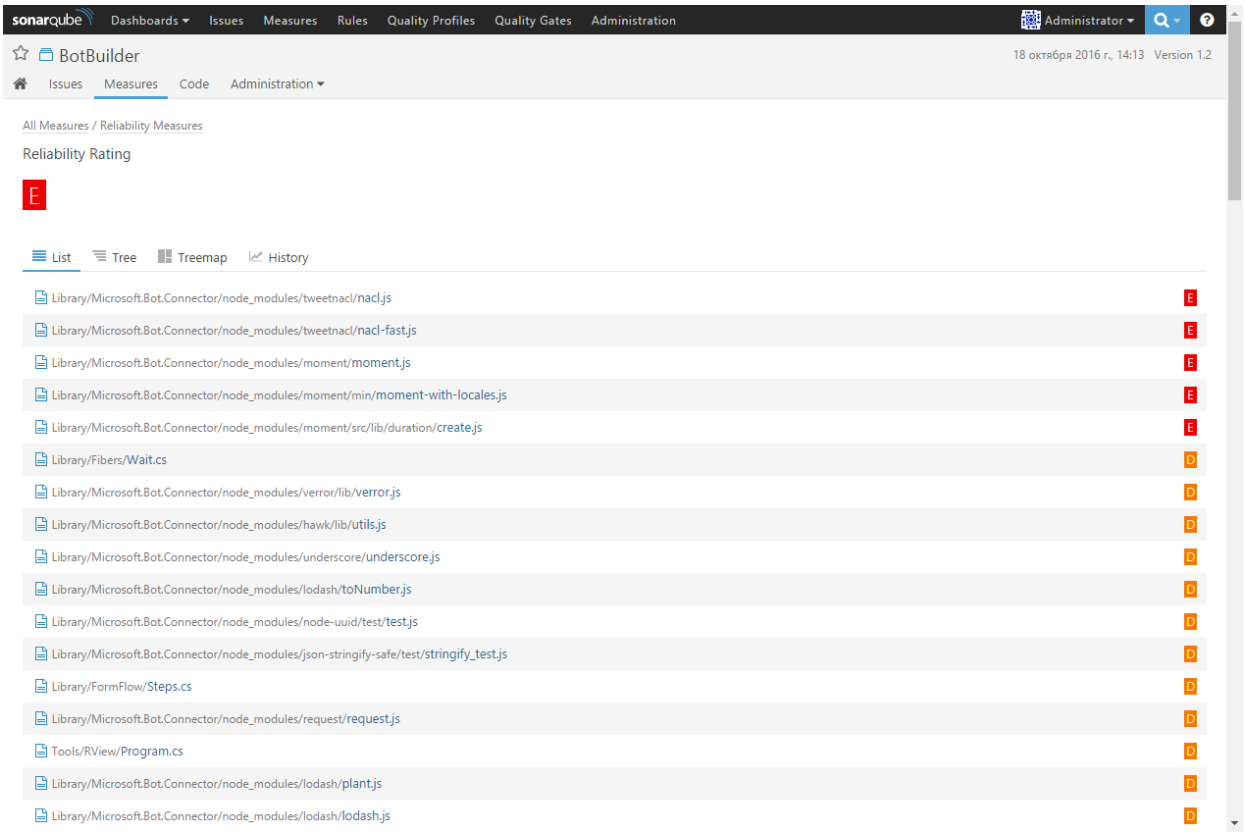


Здесь представлена информация о следующих метриках кода: Reliability (Надежность), Security (Безопасность), Maintainability (Поддерживаемость), Coverage (Покрытие тестами), Duplications (Дублирование), Size (Размер кодовой базы), Complexity (Цикломатическая сложность), Documentation (Документирование кода) и Issues (Ошибки).

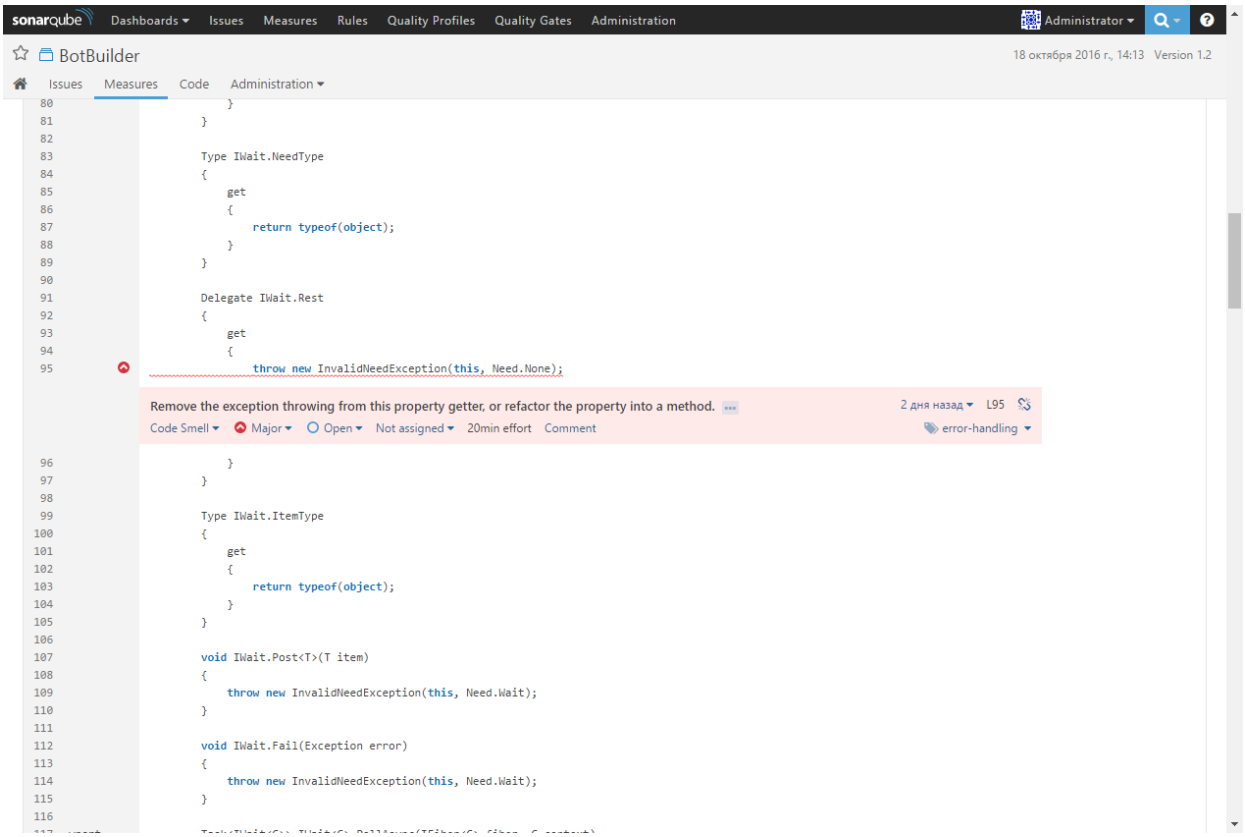
Перейдя к метрике Reliability, мы получаем информацию об общем количестве обнаруженных багов и новые баги, обнаруженные во время последнего анализа, рейтинг надежности кода по шкале от A до E, где E – наихудший рейтинг, свидетельствующий о том, что был найден по крайней мере один blocker баг, а также время, необходимое на устранение всех найденных ошибок:



Платформа SonarQube позволяет анализировать метрики кода сверху вниз, от уровня проекта в целом до отдельных модулей и файлов. Так, например, если вы кликните на рейтинг надежности (Reliability Rating), вы увидите список файлов проекта, отсортированных по возрастанию рейтинга надежности. Это позволит сфокусироваться на наиболее проблемных участках кода:

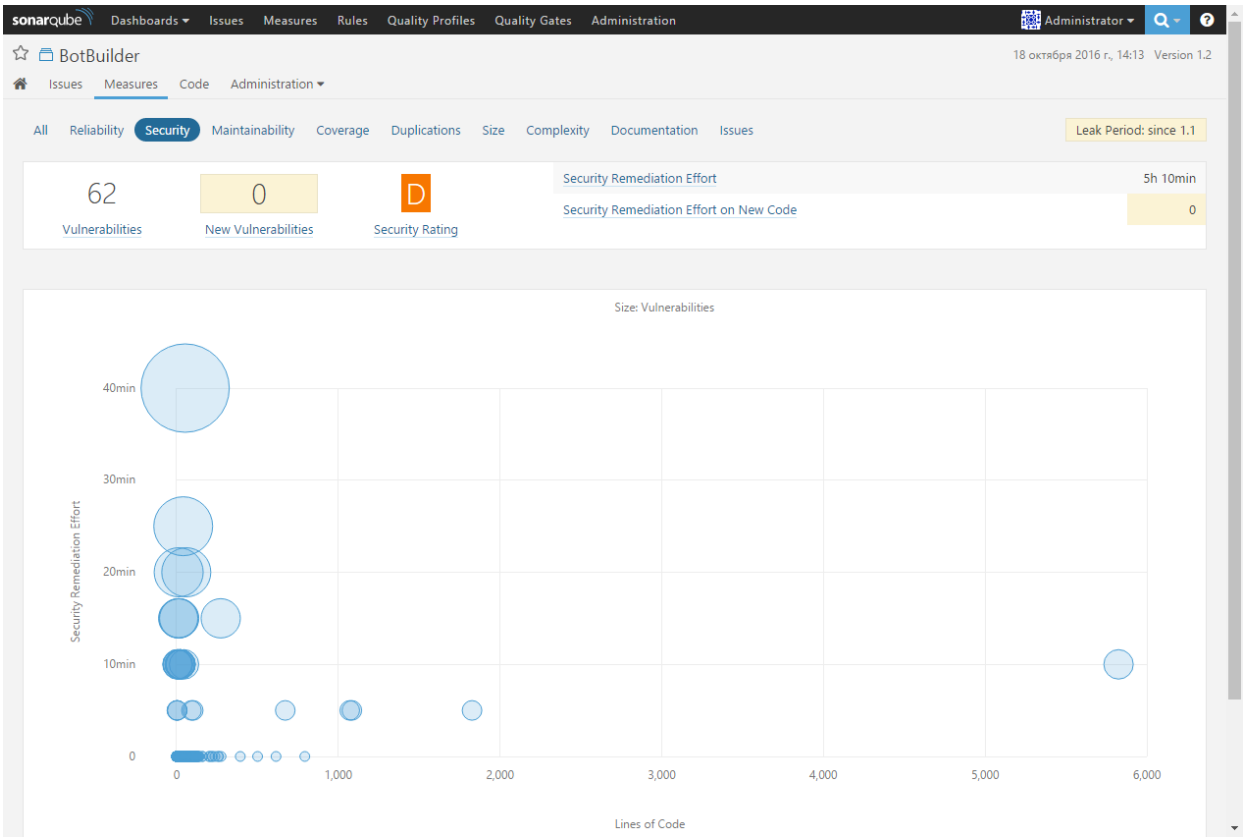


Затем вы можете перейти к файлу с исходным кодом и к конкретным участкам кода, в которых обнаружены ошибки:

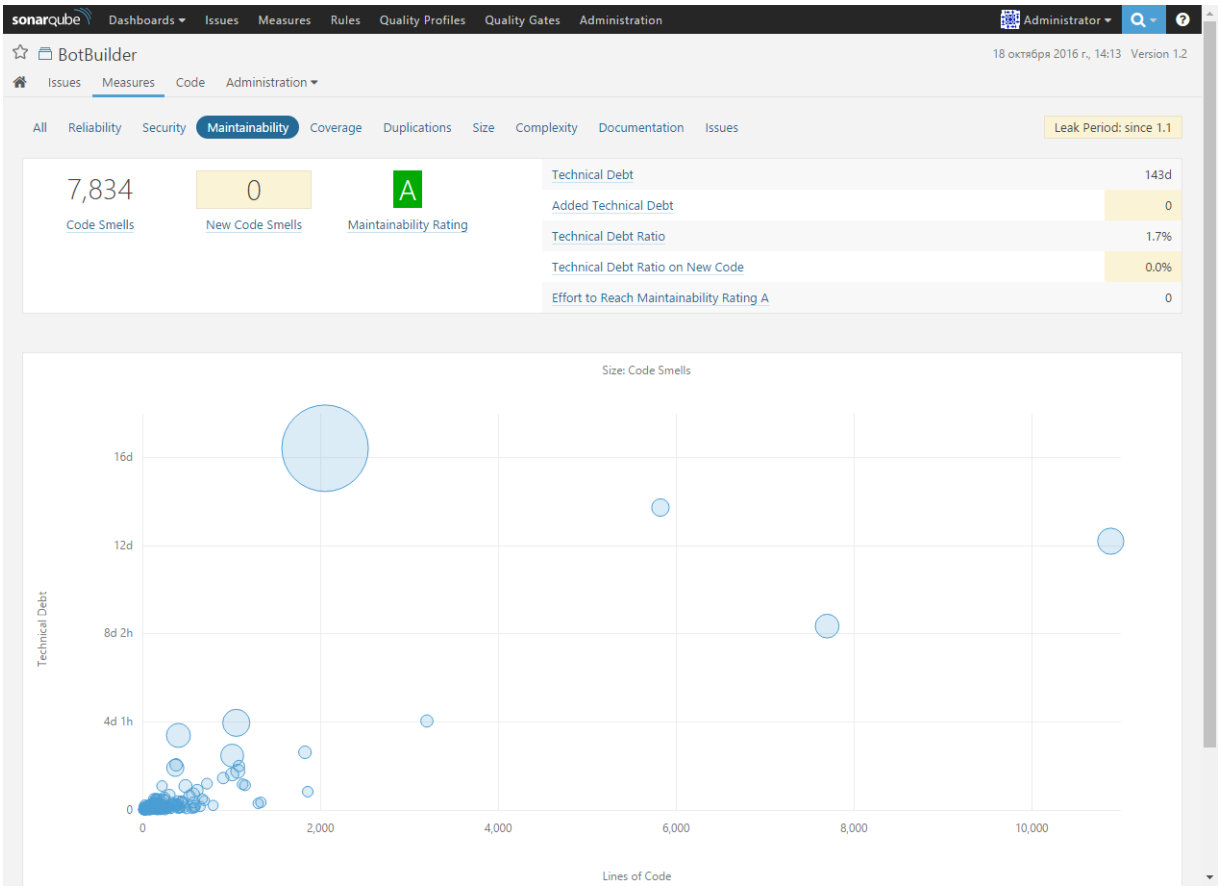


Такая навигация сверху вниз доступна и для других метрик.

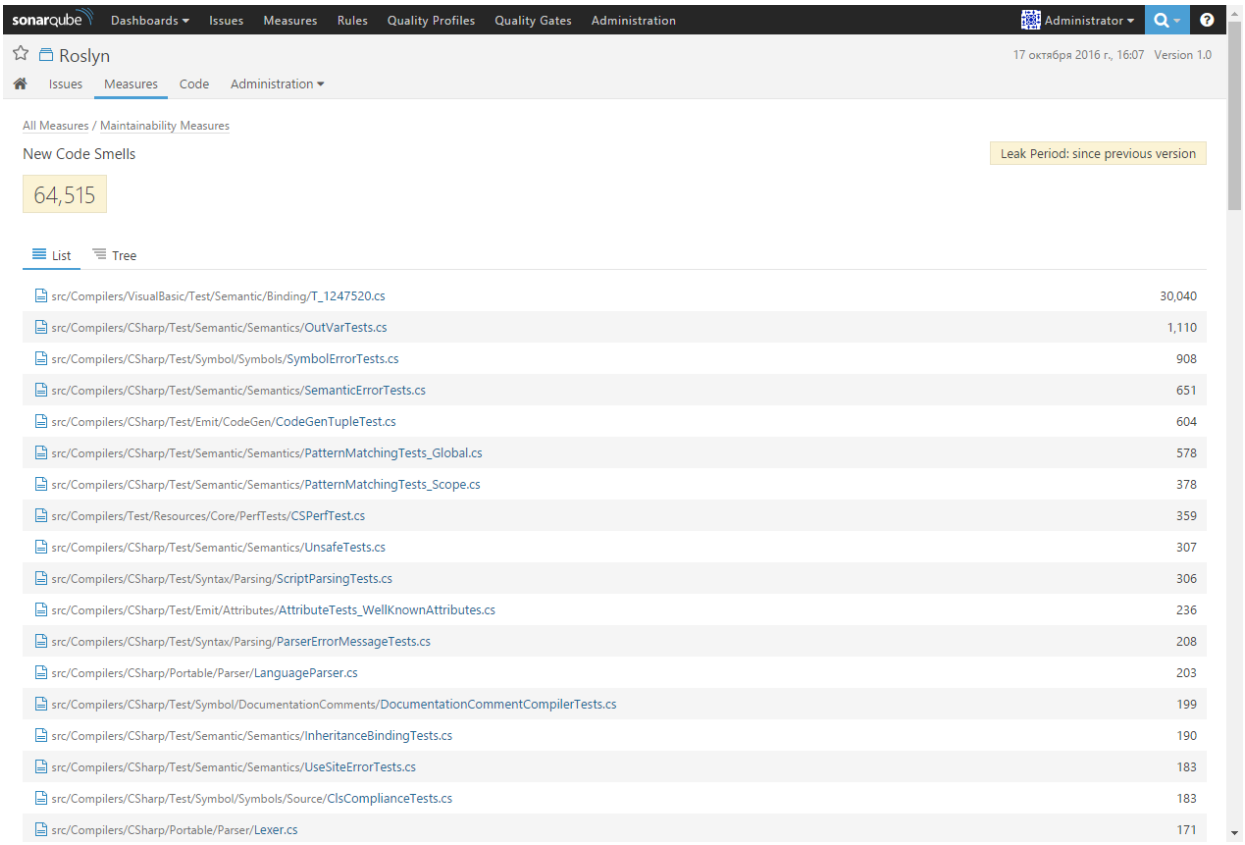
На странице метрики Security доступна информация об общем количестве уязвимостей, новых уязвимостях, рейтинге безопасности (также по шкале от A до E), и времени, которое потребуется на устранение уязвимостей:



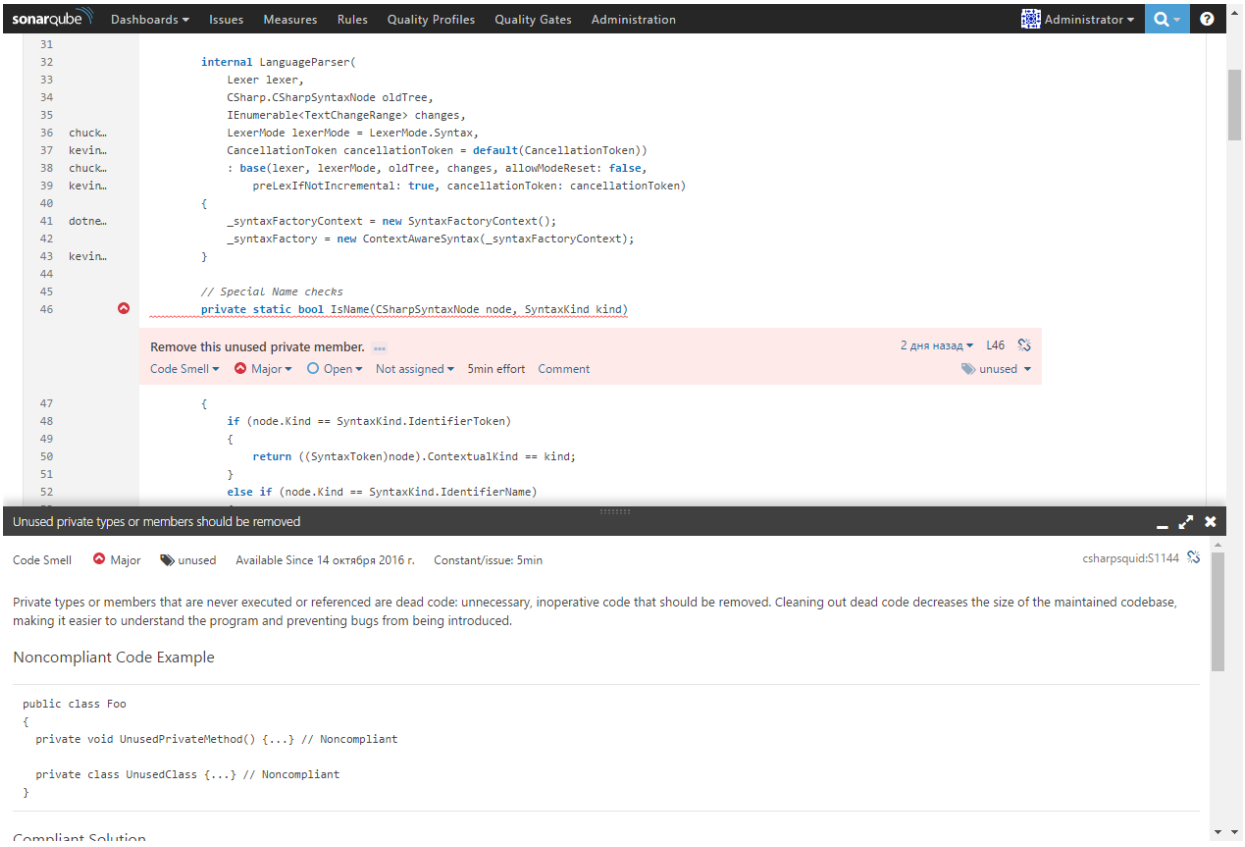
Страница Maintainability содержит информацию о техническом долге в проекте:



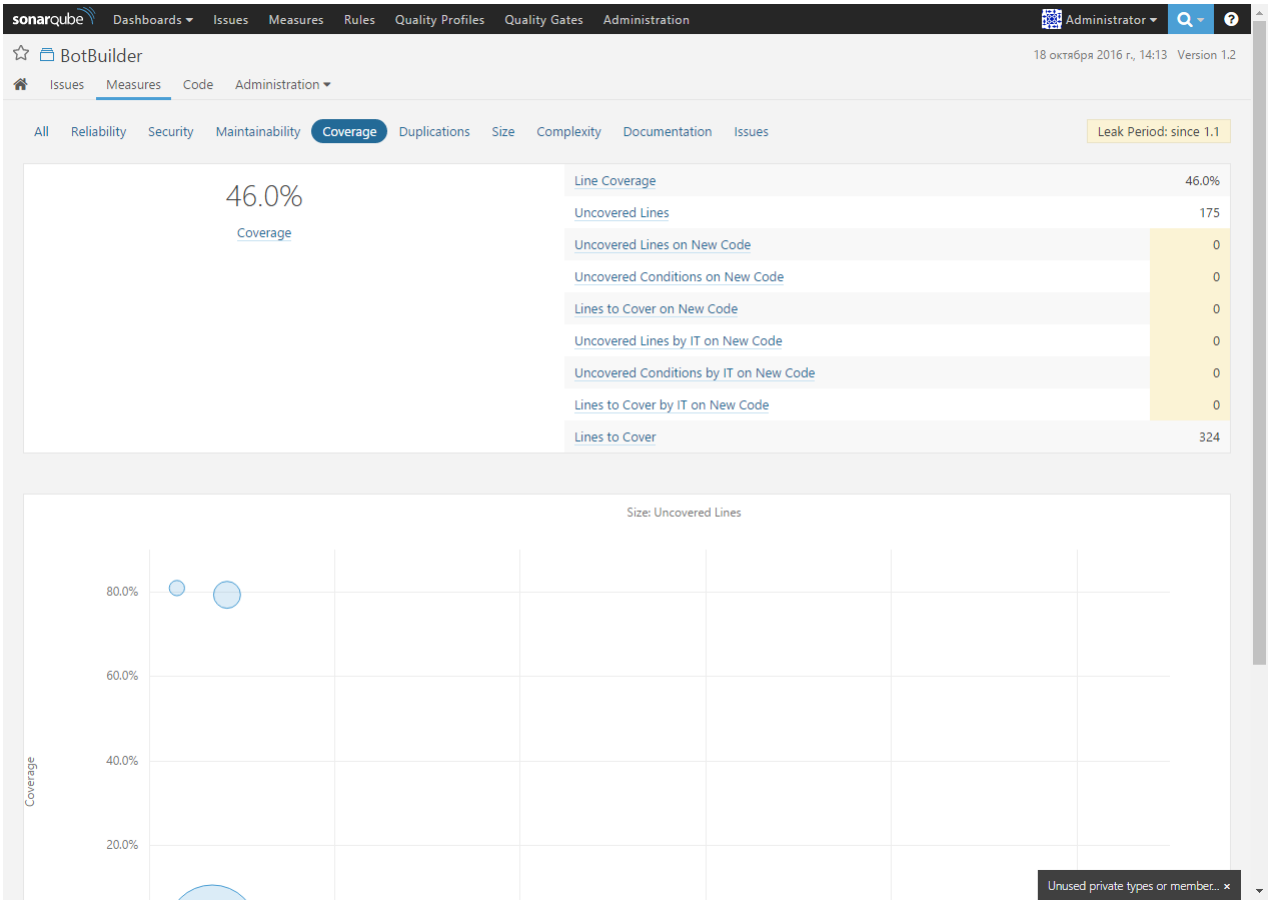
Благодаря навигации «сверху вниз» вы можете перейти к списку файлов, отсортированных по количеству обнаружений кода с душком:



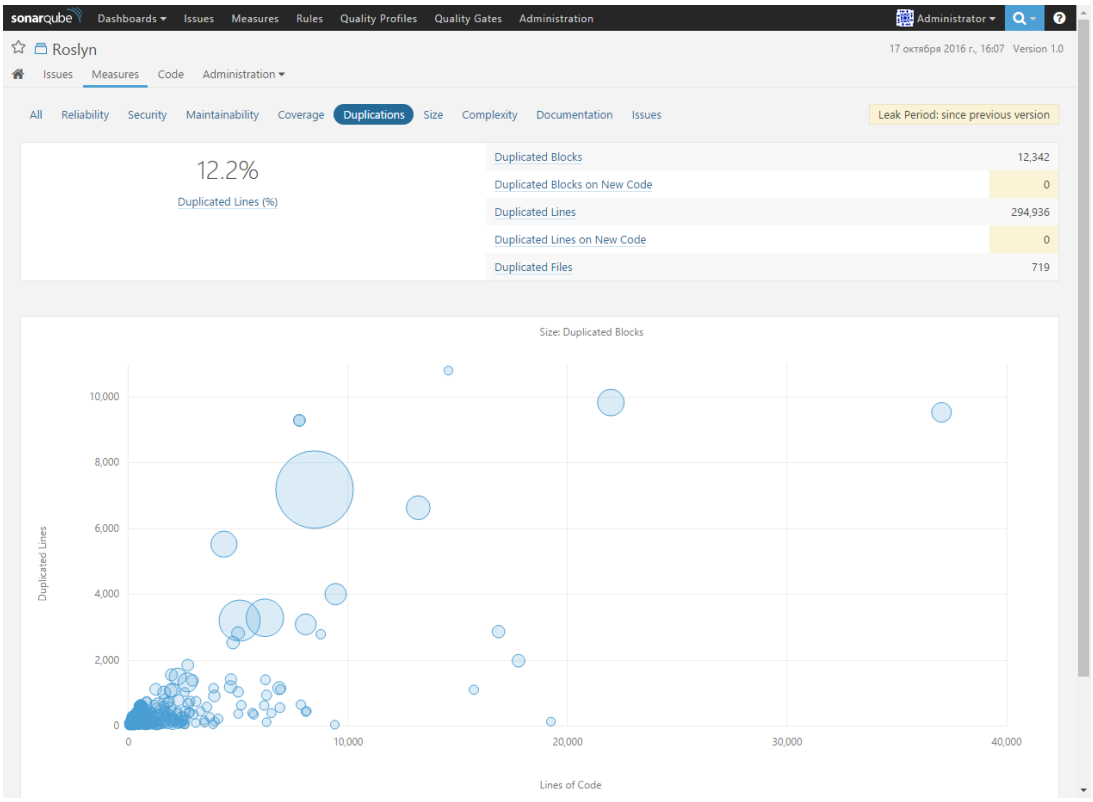
и затем непосредственно к коду, который требует внимания:



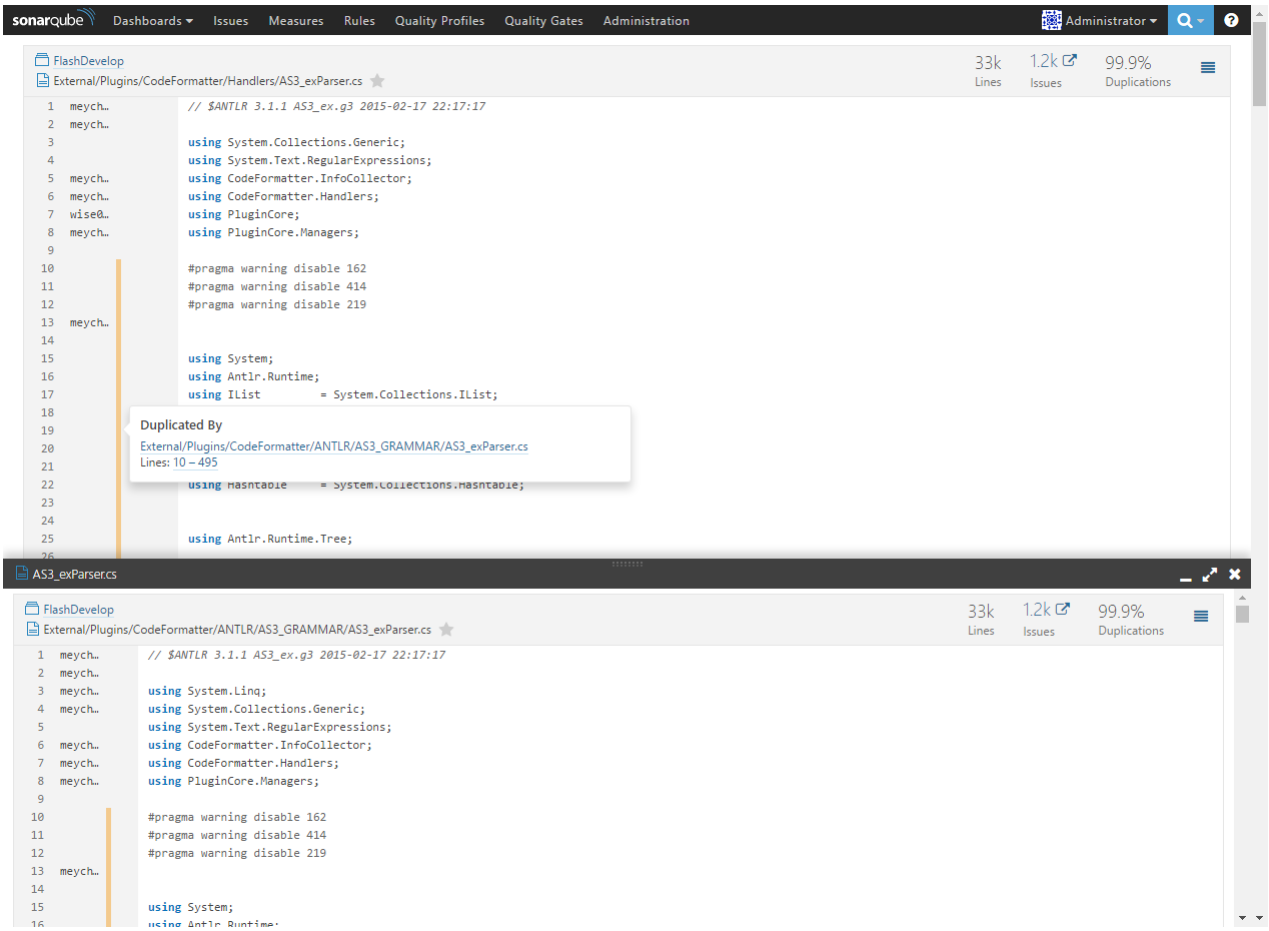
На странице Coverage представлена информация о покрытии кода тестами:



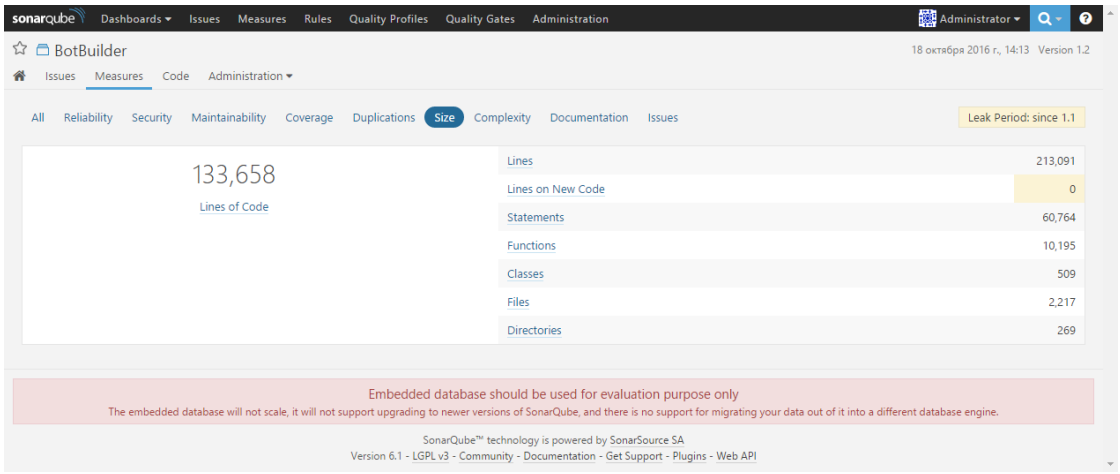
Страница Duplications содержит информацию о дублировании кода в проекте:



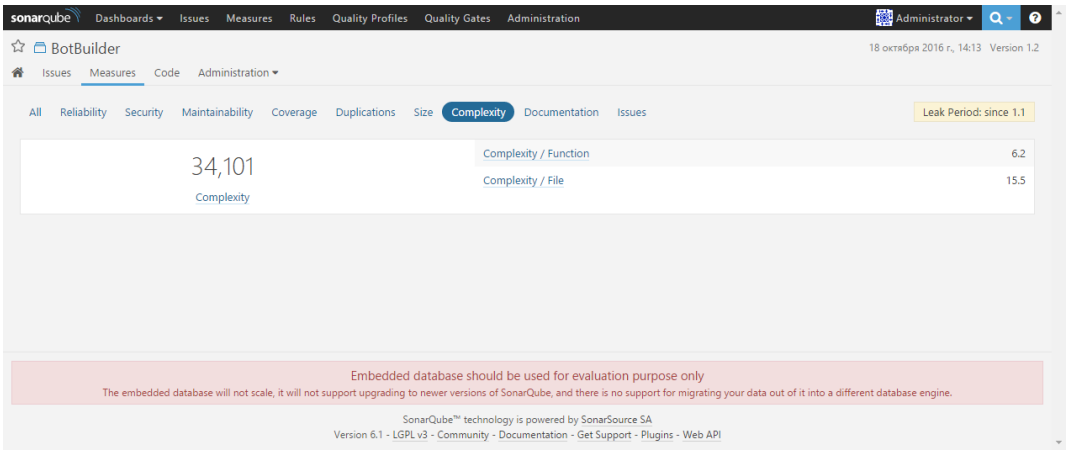
С помощью этой метрики вы легко можете обнаружить повторяющиеся строки, блоки кода и даже целые файлы:



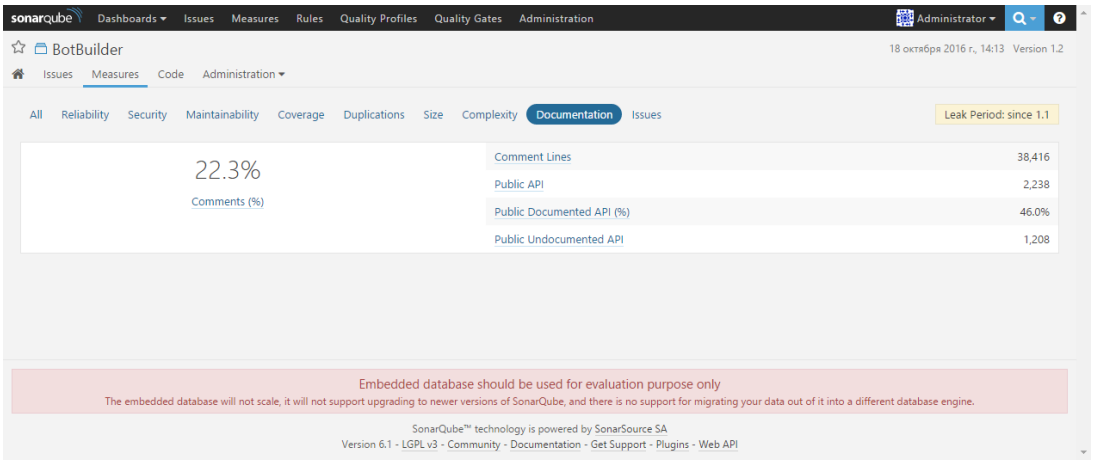
Страница Size содержит информацию о размере проекта: количество строк кода, выражений, функций, классов, файлов и директорий:



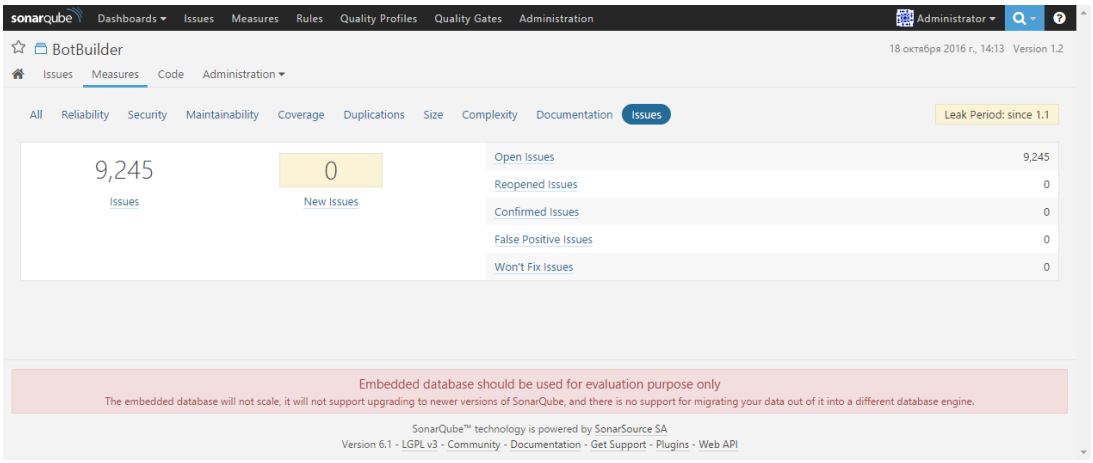
На странице Complexity представлена информация о суммарной цикломатической сложности проекта, а также о средней сложности функций и файлов:



Страница Documentation предоставляет информацию о комментариях в коде: отношение строк с комментариями к общему количеству строк в проекте, количество строк с комментариями, количество публичных API и уровень документирования публичных API:

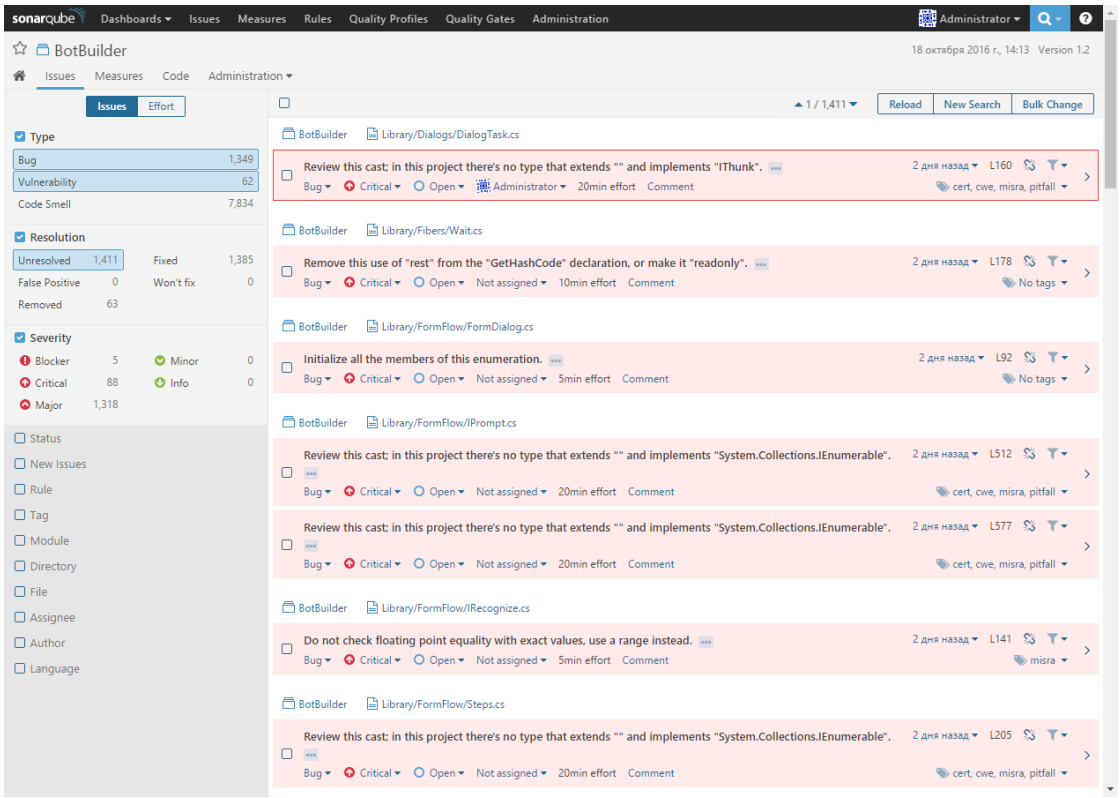


Последняя вкладка в разделе метрик проекта – Issues – содержит общее количество найденных проблем в коде (сумма количества багов, уязвимостей и code smells), а также распределение проблем по состоянию: открытые, переоткрытые, подтвержденные, ложные срабатывания и won't fix:



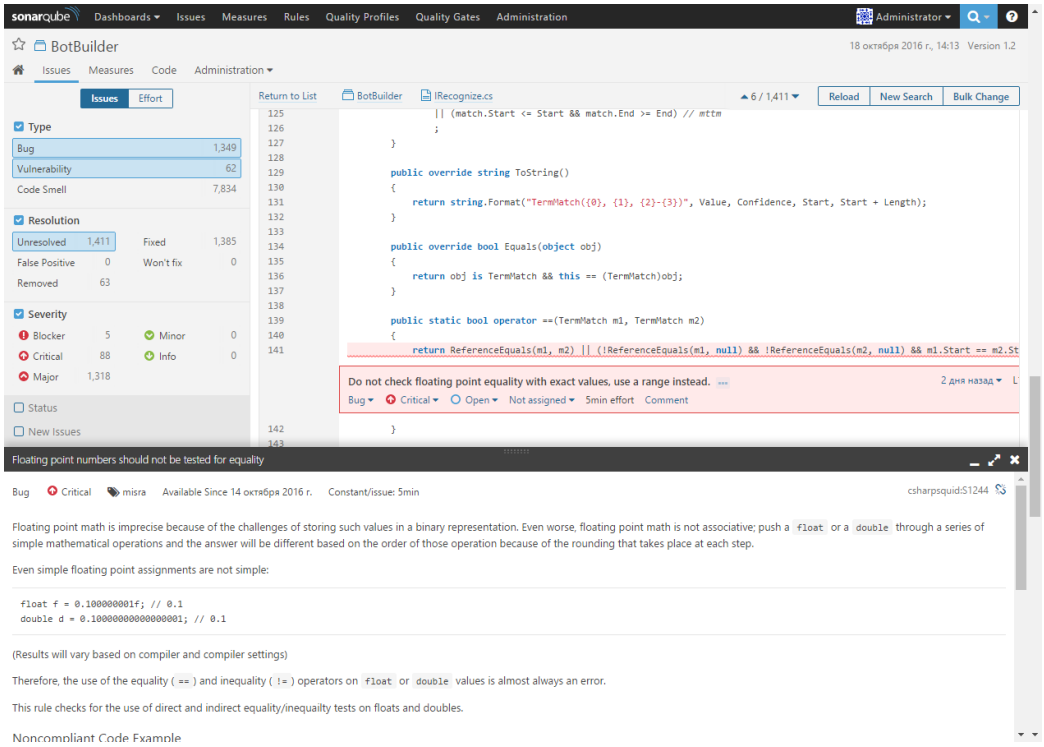
- Навигация по ошибкам и коду

После анализа метрик кода посмотрим, как SonarQube позволяет работать с найденными проблемами в коде. Для этого перейдем в раздел Issues:

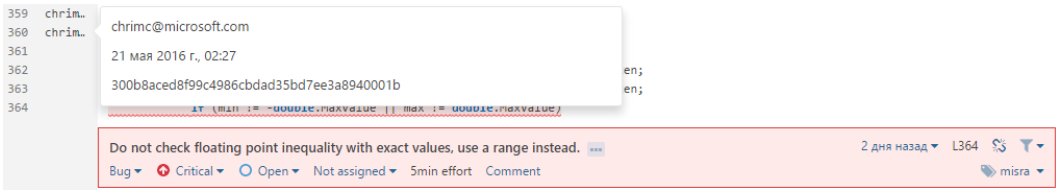


Здесь представлены все найденные проблемы в коде с широкими возможностями фильтрации, что позволяет сфокусироваться на наиболее важных проблемах. Следует отметить, что SonarQube позволяет сохранять настройки фильтров, чтобы повторно их использовать.

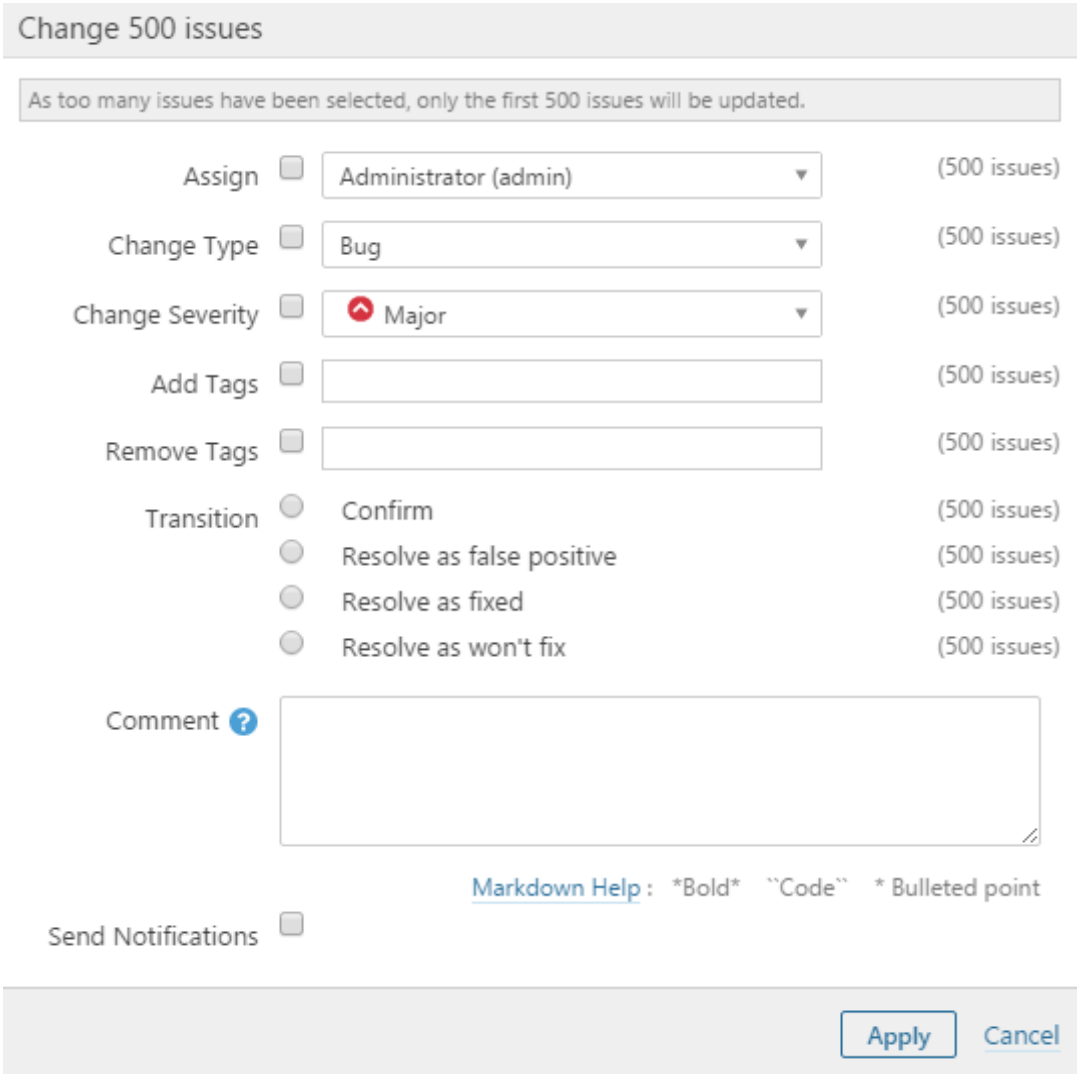
По двойному клику на сообщении об ошибке вы можете перейти к коду, в котором была найдена проблема. Также доступно детальное описание ошибки и рекомендации, как ее исправить:



Обратите также внимание, что, благодаря интеграции с системами контроля версий, видно, кто и когда внес изменения в код, вызвавшие срабатывание анализатора:



Интеграция с системами контроля версий позволяет также автоматически назначать баги в SonarQube на тех разработчиков, которые их допустили. Также вы можете назначать баги на разработчиков вручную, изменять их тип (bug, vulnerability или code smell), важность, теги, добавлять комментарии. Для большего удобства использования доступна функция массового изменения багов:



- Rules, Quality Profiles и Quality Gates

Диагностические правила (Rules), профили качества (Quality Profiles) и границы качества (Quality Gates) – ключевые понятния платформы SonarQube. Каждый плагин для SonarQube, осуществляющий статический анализ кода, содержит репозиторий с описанием диагностических правил, которые этот плагин выполняет. Нарушения этих правил используются для определения технического долга в коде и вычисления времени на устранение проблем. Для удобства использования правила объединяются в профили качества (Quality Profiles). По умолчанию, SonarQube создает дефолтный профиль качества для каждого поддерживаемого языка, но вы можете создавать свои профили качества с тем набором диагностических правил, которые вам могут быть полезны. Например, для анализа критически важных проектов, требования к качеству кода которых самые строгие, можно определить профиль качества, содер-

жащий все доступные диагностики, а для менее критичных проектов можно определить менее строгий профиль качества, содержащий только серьезные ошибки, что позволит не отвлекаться на незначительные code smells.

Quality Gate – это индикатор соответствия (или несоответствия) кода проекта заданным пороговым значениям метрик. По умолчанию, все проекты, добавленные в SonarQube, используют стандартный quality gate, в котором определены следующие метрики и их пороговые значения:

- Новые баги = 0
- Новые уязвимости = 0
- Коэффициент технического долга на новом коде <= 5%
- Покрытие нового кода >= 80%

Контрольные вопросы

1. Какие действия необходимо выполнить, чтобы определить метрики качества кода?
2. Каким образом определяется покрытие кода?

6. Лабораторная работа №3. Функциональное тестирование

Целью работы является изучение функционального тестирования ПО. Результатом работы является отчет, в котором должны быть приведены исходные коды программы, результаты функционального тестирования ПО.

Для выполнения работы студент должен изучить приведенный ниже теоретический материал. Отчет сдается в распечатанном и электронном (файл Word) видах.

6.1. Определение

Функциональное тестирование – это проверка ПО на правильность функционирования в идеальных условиях, в отличие от нефункционального, где либо условия неидеальны (нагрузочное тестирование), либо тестируется не правильность функционирования, а что-то иное (тестирование удобства пользования или структура программы).

Есть много приложений, для которых производительность и удобство пользования некритичны. Во всяком случае, часто требования к ПО содержат только функциональную часть. И практически не бывает требований к ПО без функциональной части. Отсюда делаем вывод, что функциональное тестирование проводить нужно для любого ПО.

Как правило, функциональное и нефункциональное тестирование ПО можно проводить параллельно, поэтому обычно это делается разными людьми или командами. В большинстве источников указывается, что функциональное тестирование – это синоним black-box тестирования (при котором программа рассматривается как черный ящик).

6.2. Black-box, white-box, gray-box тестирование.

По степени доступа к коду различают два типа тестирования: тестирование «черного ящика» (black-box), или функциональное тестирование – без доступа к коду, и «белого ящика» (white-box), или структурное тестирование – тестирование кода.

В случае «черного ящика» программа рассматривается как конечный автомат, с входными и выходными данными, набором внутренних состояний и переходов между ними. Тестируется правильность поведения программы при различных входных данных и внутреннем состоянии. Правильность определяется исходя главным

образом из спецификации, а также любыми другими способами, кроме изучения кода (см. лекцию 1).

В случае «белого ящика» тестировщик пишет тесткейсы, основываясь исключительно на коде программы (тесты на правильность кода).

Расширение black-box тестирования, в котором также применяется изучение кода, называется тестированием «серого ящика» (gray-box). В этом случае правильность поведения определяется любым удобным способом, в том числе изучением кода, что позволяет писать более эффективные black-box тесты (то есть тесты на правильность поведения).

Терминология приведена по книге A Practitioner's guide to Software Test Design (Lee Copeland).

6.3. Методы отбора тестов для black-box тестирования

Любая программа может рассматриваться как конечный автомат, с входными и выходными данными, набором внутренних состояний и переходов между ними.

Чтобы провести полное тестирование программы, нужно проверить правильность ее поведения при всех возможных комбинациях входных данных и во всех возможных внутренних состояниях. Это невозможно из-за огромного числа комбинаций даже в простейших случаях. Поэтому на практике отбираются только наиболее важные тесты, такой отбор можно производить несколькими методами.

Тестирование сценариев использования – юз-кейсов (use-cases)

Чтобы удостовериться в правильности перехода программы между различными внутренними состояниями, в идеале следует протестировать все возможные переходы между каждым из состояний (не только одношаговые, но и многошаговые, то есть все пути в графе состояний).

Чтобы уменьшить число тестов, можно проверить только те переходы, которые имеют смысл для пользователя. Use-case – это логически завершенная последовательность действий. Например, открытие файла в Notepad – это use-case, а выбор пункта меню «Открыть файл» в Notepad – это не use-case, а лишь первый шаг юзкейса «открытие файла».

Тестирование сценариев является самым необходимым видом тестирования. Программа должна выполнять операции, для которых

она предназначена. Если пользователь может выбрать пункт меню, но файл не открывается – это очень серьезный баг.

Здесь проверяется правильность перехода программы между внутренними состояниями при выполнении определенных операций (т. е. при определенных входных данных).

Тестирование классов эквивалентности.

Чтобы удостовериться в правильности поведения программы при различных входных данных, в идеале следует протестировать все возможные значения для каждого элемента этих данных (а также все возможные сочетания входных параметров).

Например, пусть мы тестируем программу для отдела кадров, в ней есть поле «Возраст соискателя».

Пример взят из книги A Practitioner's guide to Sofware Test Design (Lee Copeland).

Требования по возрасту у нас будут такие:

0–13 лет – не нанимать;

14–17 лет – можно нанимать на неполный день;

18–54 года – можно нанимать на полный день;

55–99 лет – не нанимать.

Чтобы проверить все возможные разрешенные данные (только разрешенные!) нам нужно протестировать ввод чисел от 0 до 99. (Возможен ведь еще ввод отрицательных чисел и нечисловых данных.) Так ли необходимо тестировать все числа от 0 до 99? В случае, если программа анализирует каждое число по отдельности, вот таким образом, то видимо, да:

```
...
if (age == 13) hireStatus=«NO»;
if (age == 14) hireStatus=«PART»;
if (age == 15) hireStatus=«PART»;
if (age == 16) hireStatus=«PART»;
if (age == 17) hireStatus=«PART»;
if (age == 18) hireStatus=«FULL»;
...
```

Но к счастью, программы обычно пишут по-другому:

```
if (age >= 0 && age <=13)
        hireStatus=«NO»;
if (age >= 14 && age <=17)
        hireStatus=«PART»;
```

```
if (age >= 18 && age <=54)
        hireStatus=«FULL»;
if (age >= 55 && age <=99)
        hireStatus=«NO»;
```

Становится очевидным, что можно протестировать одно из чисел каждого диапазона. Например: 5, 15, 20, 60. А также граничные значения (первое и последнее значения из каждого диапазона): 0, 13, 14, 17, 18, 54, 55, 99.

Чтобы уменьшить количество тестируемых значений, производится:

а) разбиение множества всех значений входной переменной на подмножества (классы эквивалентности), а затем;

б) тестирование одного любого значения из каждого класса.

Все значения из каждого подмножества должны быть эквивалентны для наших тестов. То есть, если тест проходит успешно для одного значения из класса эквивалентности, он должен проходить успешно для всех остальных. И наоборот, если тест не проходит для одного значения, он не должен проходить для всех остальных.

В данном случае имеем 12 классов эквивалентности (каждое из 8 граничных значений по сути является отдельным классом).

Чтобы проверить правильность работы программы на всех разрешенных данных, нужно провести 12 тестов.

Запрещенные данные тестируются аналогично – можно выделить классы эквивалентности «дробное число от 0 до 99», «отрицательное число», «число больше 99», «набор букв», «пустая строка» и т. д.

Таким образом, метод классов эквивалентности можно разделить на три этапа:

1. Тестирование разрешенных значений.

2. Тестирование граничных значений.

3. Тестирование запрещенных значений.

Часто в литературе второй и третий этапы называют отдельными методами, но сути это не меняет.

Попарное тестирование.

Метод классов эквивалентности применяется для тестирования каждого входного параметра по отдельности.

Пусть наша программа принимает на вход десяток параметров. Баги, возникающие при определенном сочетании всех десяти пара-

метров, довольно редки. Вообще, взаимное влияние параметров, о котором пользователь не знает – это баг интерфейса (интерфейс интуитивно не понятен).

Чаще всего будут встречаться ситуации, в которых один параметр влияет на один из оставшихся, т. е. самыми частыми будут баги, возникающие при определенном сочетании двух каких-то параметров.

Таким образом, можно упростить себе задачу и протестировать все возможные значения для каждой из пар параметров. Такой подход называется попарным тестированием (pairwise testing).

Вот пример. Пусть имеется 3 двоичных входных параметра (3 чекбокса). Количество всех возможных комбинаций – 2 в степени 3 = 8 , значит, нужно произвести 8 тестов. Давайте попробуем сэкономить, тестируя чекбоксы попарно.

Выпишем все комбинации для первого и второго чекбоксов:

1-й	2-й
0	0
0	1
1	0
1	1

Добавим третий столбец так, чтобы во втором и третьем столбце получились все 4 двоичные комбинации. Это можно сделать разными способами, мы сделаем так (на первый столбец можно не обращать внимания):

1-й	2-й	3-й
0	0	0
0	1	0
1	0	1
1	1	1

Итак, с помощью четырех наборов входных данных (четырех тестов) мы протестируем две пары параметров: первый со вторым и второй с третьим. Осталось протестировать пару «первый с третьим».

Выпишем отдельно 1 и 3 столбцы:

1-й	3-й
0	0
0	0
1	1
1	1

Как видно, мы имеем здесь две из четырех возможных комбинаций. Комбинации «01» и «10» здесь отсутствуют, а комбинации «00» и «11» присутствуют два раза. Ну что же, добавим еще 2 строки (еще два теста)

1-й	3-й
0	0
0	0
1	1
1	1
0	1
1	0

Вернем второй столбец на его законное место:

1-й	2-й	3-й
0	0	0
0	1	0
1	0	1
1	1	1
0		1
1		0

Выходит, что последние два теста можно проходить при любых значениях второго параметра. Можно дописать для определенности нули в эти пустые места:

1-й	2-й	3-й
0	0	0
0	1	0
1	0	1
1	1	1
0	0	1
1	0	0

Получаем 6 тестов вместо 8 при полном переборе.

Можно ли сэкономить еще? Оказывается, можно.

Вернемся к 1 шагу:

1-й	2-й
0	0
0	1
1	0
1	1

Давайте допишем третий столбец другим способом, поменяв порядок комбинаций:

1-й	2-й	3-й
0	0	1
0	1	0
1	0	0
1	1	1

Все комбинации для 1 и 2, а также для 2 и 3 параметра здесь есть. Отлично.

Посмотрим теперь на комбинации 1 и 3 параметра

1-й	3-й
0	1
0	0
1	0
1	1

Ого! Что мы видим? Изменив порядок значений в третьем столбце, мы одним махом убили двух зайцев: скомбинировали и 2-й с 3-м, и 1-й с 3-м параметры.

Итого имеем всего 4 строки, то есть 4 теста, эквивалентные первоначальным шести:

1-й	2-й	3-й
0	0	1
0	1	0
1	0	0
1	1	1

Полный перебор всех комбинаций в третьем столбце гарантированно даст минимальное количество тестов. Однако, судя по тому, что алгоритмы такой минимизации разрабатываются до сих пор, полный перебор неприемлем из-за большого времени исполнения. Существуют программы, дающие приемлемый результат в приемлемое время, например, программа PICT от Microsoft.

Ортогональный массив OA(N,k,s,t) – это двумерный массив из N рядов (итераций) и k колонок (факторов) из набора S (т. е. факторы могут принимать любое из s значений), обладающий свойством: выбрав любые t колонок (0<=t<=k) мы получим в рядах все комбинации сочетаний из s по t (Количество повторений одинаковых комбинаций обозначают через λ. Чаще всего рассматривают масси-

вы, где λ = 1, т. е. каждая комбинация встречается только один раз). Параметр t называют мощностью ортогонального массива.

В попарном тестировании применяется ортогональный массив мощности 2 – это двумерный массив такой, что любые 2 колонки этого массива содержат все возможные комбинации (пары) значений, хранящихся в массиве.

Использование информации о программе при gray-box тестировании.

Главным минусом black-box тестирования является то, что тестировщик не знает, какую часть ПО он тестирует. Некоторые существующие пути в программе (о которых нет информации ни в требованиях, ни в документации), могут никогда не быть проверены. Уменьшить количество таких путей можно путем анализа внутреннего устройства программы.

Информация о базе данных

Если программа использует для своей работы какую-либо базу данных, мы можем проанализировать типы полей, в которые записываются переменные программы. А потом проанализировать ограничения, которые накладывает база данных.

Например, если вводимая фамилия пользователя записывается в поле типа «строка» длиной 128 символов, мы должны:

1) попробовать найти фамилию длиннее, чем 128 символов – здесь будет довольно серьезный баг, если такие фамилии существуют – человек с такой фамилией не сможет воспользоваться нашей системой;

2) вне зависимости от того, существуют или нет такие фамилии, попробовать ввести строку длиннее 128 символов – программа не должна ломаться (должно показываться внятное сообщение об ошибке).

Информация о других внешних системах

Если программа интегрируется с другими внешними системами, помимо базы данных, можно также проанализировать ограничения таких систем. Например, если мы тестируем почтовый IMAP-клиент, следует убедиться, что он корректно обрабатывает длинные пути к папкам на сервере (чаще всего, ограничение на длину пути составляет 255 символов)

Информация о коде программы

Если мы имеем доступ к коду программы, мы можем

а) увидеть специальные случаи, которые не попали в документ с требованиями и которые необходимо протестировать или, напротив;

б) увидеть, что какие-то вещи тестировать не имеет смысла.

6.4. Методы отбора тестов для White-Box тестирования

White-box тестирование является не функциональным, а структурным – тестируется код программы. От тестировщика требуются, таким образом, навыки программиста. Главным минусом такого тестирования является то, что тестировщик никогда не сможет обнаружить, что какая-то функциональность не реализована, поскольку нельзя протестировать код, который не написан.

Различают два метода отбора тестов:

1. Тестирование путей исполнения (control-flow testing).
2. Тестирование работы с данными (data-flow testing).

Подробно эти методы описаны в книге A Practitioner's guide to Sofware Test Design (Lee Copeland). Описание их выходит за рамки данной лекции, поскольку они не являются методами функционального тестирования.

Контрольные вопросы

1. Что такое Black Box?
2. Что такое White Box?
3. Приведите порядок отбора тестов для различных случаев.

7. Лабораторная работа №4. Тестирование безопасности

Целью работы является изучение тестирования безопасности ПО. Результатом работы является отчет, в котором должны быть приведены исходные коды программы, результаты тестирования безопасности ПО.

Для выполнения работы студент должен изучить приведенный ниже теоретический материал. Отчет сдается в распечатанном и электронном (файл Word) видах.

Тестирование безопасности – процесс достижения совершенства в открытом космосе при условии, что с Вашим скафандром что-то не так. Но если вернуться на Землю, то тестирование безопасности веб-приложения – это попытка найти все те места, в которых могли допустить ошибку разработчики или просто не предусмотреть / забыть (подчеркните ваш случай).

Веб-приложение и веб-сервер неразрывно связаны. Тестирование одного без другого не даст полной картины бедствия. Тестируя защиту веб-приложения, мы ищем уязвимые места для атаки на пользователей. Тестируя защиту веб-сервера, мы ищем уязвимые места для атаки на сервер, его инфраструктуру.

Нужно все это в первую очередь людям: простым смертным, решившим заказать пиццу, а не отправить данные своей карты непонятно куда; простым владельцам пиццерий, не беспокоящимся, что кто-то смог бесплатно научиться заказывать пиццу через их приложение; простым разработчикам, которым не придется править код в 3 часа утра.

Как правило, небольшие организации, имеющие свой сайт и небольшой сервер с битриксом, думают, что они слишком малы для того, чтобы стать мишенью для атаки. И в этом они заблуждаются. Безусловно, вероятность таргетированной атаки на них ниже, чем на финансовых гигантов, но все-таки не равна нулю. В нынешнее время нейронных сетей и повальной автоматизации никто не будет выяснять, большой ли денежный оборот у фирмы. Главное – это количество уникальных посетителей, ведь суммарно в их карманах может оказаться больше «фишек», чем компания получает за год.

Существуют компании, которые уже взломали, и компании, которые еще не взломали. На практике это вопрос времени. Правильные компании, которые тратят на безопасность значительную часть прибыли, – это нормальный порядок вещей для всего цивили-

зованного мира. В нашей стране, к сожалению, пока к этому не пришли, считая, что «хата наша с краю»студент-старшекурсник вполне способен настроить одну «циску» и запретить доступ к внутренним ресурсам извне (мол, «денег нет, но вы держитесь»). Ведь мы не привыкли тратить деньги на свою безопасность: бюджет отделов по информационной безопасности (ИБ) обычных компаний только сокращается. С другой стороны, многие адекватные компании (читай, иностранные) формируют собственный штат по информационной безопасности, другие многие нанимают специалистов на аутсорс, немногие запускают программы баг-баунти, где любой желающий может принять участие в поиске уязвимостей. Безусловно, все они хотят сохранить свои доходы и свою репутацию.

На что и на кого ориентировано тестирование безопасности? Кто получит выгоду от этого в первую очередь?

Естественно, в первую очередь от безопасного серфа сайта выигрывает пользователь. Если нет нужды беспокоиться о том, что Ваши персональные данные могут утечь в сеть, то и доверять ресурсу Вы будете больше. А если счастлив пользователь, то счастлив и владелец веб-ресурса (и тем более он счастлив, чем меньше у него рисков потерять финансы).

ИЗНУТРИ

Исследование безопасности веб-ресурса – сложная и кропотливая работа, требующая внимательности, фантазии и творческого подхода. Исследователю безопасности необходимо глубокое понимание технической изнанки работы веб-приложения и веб-сервера. Каждый новый проект дает пищу для фантазии, каждый новый инструмент – просторы для творчества. Да и вообще, тестирование безопасности больше похоже на исследовательскую работу – это постоянный поиск и анализ.

Каждый новый проект требует применения новых инструментов, изучения новых технологий, поглощения множества книг и статей, которые достаточно трудно найти. Большая часть информации находится в англоязычном пространстве, что добавляет определенные трудности в освоении материала людям, языком английским не владеющим.

Если говорить о процессе тестирования безопасности, то в целом он не сильно отличается от тестирования обычного: поиск, ло-

кализация, воспроизведение, заведение, отчет. Приоритеты, безусловно, зависят от заказчика, от целей тестирования.

К сожалению, некоторые курсы по тестированию безопасности / анализу защищенности / аудиту безопасности в интернете внушают, что достаточно пройтись по веб-приложению каким-нибудь сканером безопасности – и все готово! Отчет есть, уязвимости – вот они! Что же еще вам нужно? Но не стоит быть таким наивным. Большинство уязвимостей ищется и находится именно вручную, при внимательном изучении. Они могут быть совершенно несложными, но автоматические сканеры пока еще не способны их обнаружить.

Классификацией векторов атак и уязвимостей занимается сообщество OWASP (Open Web Application Security Project) – международная некоммерческая организация, сосредоточенная на анализе и улучшении безопасности программного обеспечения.

OWASP (https://www.owasp.org/) составил список из 10-и самых опасных уязвимостей, которым могут быть подвержены интернет-ресурсы. Сообщество обновляет и пересматривает этот список раз в три года, поэтому он содержит актуальную информацию. Последнее обновление было сделано 2017:

- внедрение кода;
- некорректная аутентификация и управление сессией;
- утечка чувствительных данных;
- внедрение внешних XML– сущностей (XXE);
- нарушение контроля доступа;
- небезопасная конфигурация;
- межсайтовый скриптинг;
- небезопасная десериализация;
- использование компонентов с известными уязвимостями;
- отсутствие журналирования и мониторинга.

Организация OWASP дополнительно к своему списку из 10 самых опасных уязвимостей разработала методические рекомендации (практикум) по тестированию безопасности веб-приложений. В них подробно, шаг за шагом, описано, как и что необходимо тестировать, на что обратить внимание в первую очередь, а на что – во вторую. Методика носит рекомендательный характер и, конечно, никого ни к чему не обязывает (инженер, проводящий тестирование, некоторые моменты может перенести, а другие – вообще опус-

тить), но, тем не менее, позволяет сделать максимальное покрытие для веб-приложения.

Итак, весь процесс тестирования состоит из двух этапов:

1. Пассивный, во время которого тестировщик пытается понять логику приложения и «играет» с ним. Могут использоваться инструменты для сбора информации. Например, с помощью HTTP прокси можно изучить все HTTP-запросы и ответы. В конце данного этапа тестировщик должен понимать все точки входа приложения (HTTP-заголовки, параметры, куки и пр.).

2. Активный. Во время данного этапа тестировщик проводит тесты в соответствии с методологией. Все тесты разбиты на одиннадцать подразделов:

- сбор информации;
- тестирование конфигурации;
- тестирование политики пользовательской безопасности;
- тестирование аутентификации;
- тестирование авторизации;
- тестирование управления сессией;
- тестирование обработки пользовательского ввода;
- обработка ошибок;
- криптография;
- тестирование бизнес-логики;
- тестирование уязвимостей на стороне пользователя.

Какие инструменты используются для анализа безопасности? Оценив объемы бедствия, следует ~~сбеж~~рассмотреть существующие инструменты. Конечно же, главные Ваши аргументы против несправедливости в сетевых именах – это глаза и мозг. На самом же деле, добрые люди разработали огромный инструментарий – начиная от специализированных скриптов, «заточенных» для какой-то одной конкретной цели, и заканчивая целыми комбайнами – готовыми выжать максимальные выводы из минимума вводных. К сожалению, часто такой результат оказывается лже-срабатыванием. Как правило, инженер по тестированию при выборе инструментов основывается на приоритетах: что важнее – время или область покрытия? Современные разработчики довели автоматизацию до небывалых высот, поэтому можно смело следовать принципу Парето: 80% работы скармливать автоматизированным анализаторам, а все

оставшееся «проходить руками». Но, честно говоря, результаты автоматизированных средств все равно придется изучать и проверять.

Приведем небольшой список категорий инструментов:

- сканеры веб-уязвимостей;
- инструменты для эксплуатации уязвимостей;
- инструменты криминалистики;
- сканеры портов;
- инструменты мониторинга трафика;
- отладчики;
- руткит детекторы;
- инструменты шифрования;
- инструменты для брутфорса.

Из-за разгильдяйства разработчиков. Из-за невнимательности. Из-за халтуры. Из-за лени. Из-за-за…

Но чаще всего уязвимости возникают из-за неопытности. Не все разработчики представляют себе, как злоумышленник будет атаковать их продукт. Некоторые считают, что достаточно экранировать кавычку (‘)в пользовательском вводе или спастись «magic_quotes» – и можно будет избежать SQL-инъекции. Отсюда и получается, что превентивные меры мы принимаем, а что делать дальше – не знаем. И WAF’ы нас не спасут.

Контрольные вопросы

1. Перечислите основные атаки на код.
2. Что такое внедрение кода?
3. Какие основные причины появления уязвимостей в коде?

8. Лабораторная работа №5. «Нагрузочное тестирование, стрессовое тестирование»

Целью работы является изучение нагрузочного тестирования ПО. Результатом работы является отчет, в котором должны быть приведены исходные коды программы, результаты нагрузочного тестирования ПО.

Для выполнения работы студент должен изучить приведенный ниже теоретический материал. Отчет сдается в распечатанном и электронном (файл Word) видах.

Подготовка проекта и запись тестов

В качестве основного инструмента для нагрузочного тестирования мы будем использовать MS Visual Studio Enterprise 2017 (в других редакциях студии поддержка данного типа проектов может отсутствовать) и тип проекта **Web Performance and Load Test Project**.

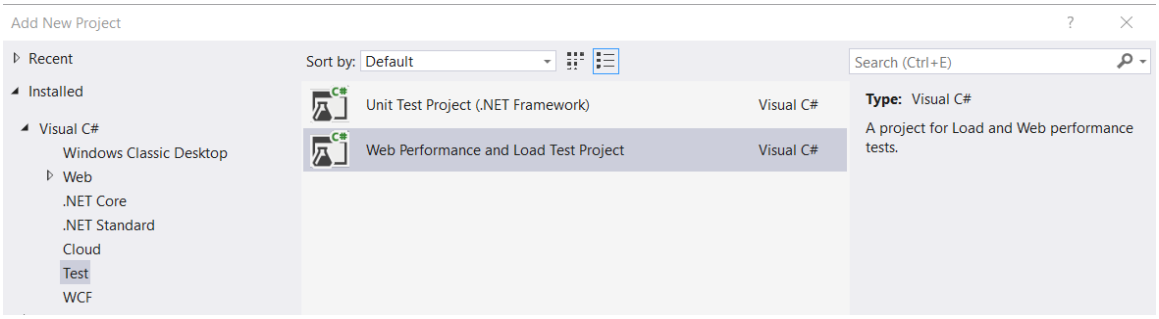


Рис. 1. Создание нового проекта

После создания проекта нам необходимо будет создать тесты для каждого из ранее определенных пользовательский действий. Ограничимся созданием теста для одного пользовательского действия в качестве примера, поскольку остальные действия создаются по аналогии.

Для тестов мы будем использовать стандартный тип теста Web Performance Test, встроенный в Visual Studio.

Нашим первым тестом, который мы создадим, будет тест, открывающий детали продукта в интернет-магазине.

Для создания теста выберем из списка предложенных Studio тип теста «**Web Performance Test**», зададим имя «**Storefront-ProductDetail**».

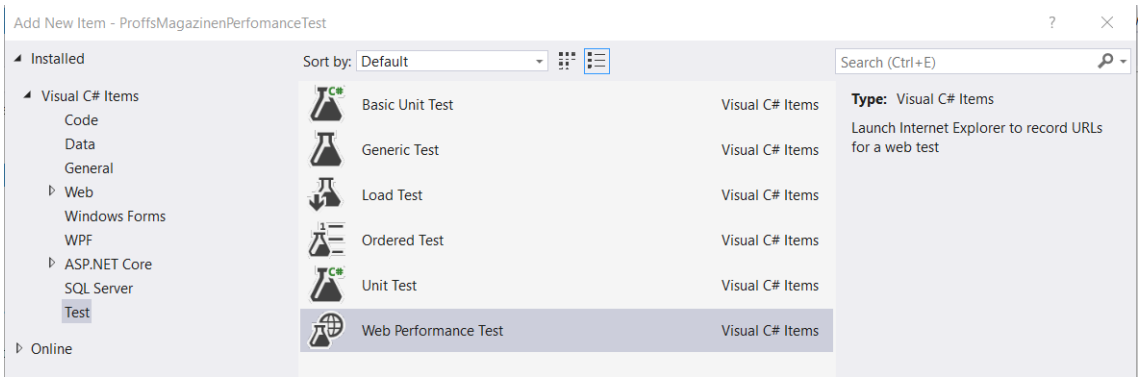


Рис. 2. Выбор типа теста в Visual Studio

Для данного типа теста Visual Studio сразу же попытается открыть браузер, где можно будет в интерактивном режиме прокликать необходимые действия непосредственно на сайте, но мы этого делать не будем, но сразу закроем браузер и остановим запись. В итоге мы получим пустой тест **Storefront-ProductDetail.webtest**.

Далее нам нужно добавить источник данных для данного теста, для того чтобы можно было использовать различные параметры запроса в рамках одного теста, для этого в V**S Studio Web Performance Test** предусмотрена такая возможность.

В качестве источника данных для нашего теста мы будем использовать таблицу в базе данных, где хранятся записи о продуктах. После этого у нас появится возможность использовать данные из подключенного источника в запросе, который должен открывать детали продукта на тестируемом приложении. Достигается это путем вставки конструкции «{{Имя источника данных.Название таблицы.Название колонки}}».

В итоге после всех манипуляций наш первый тест примет вот такой вид.

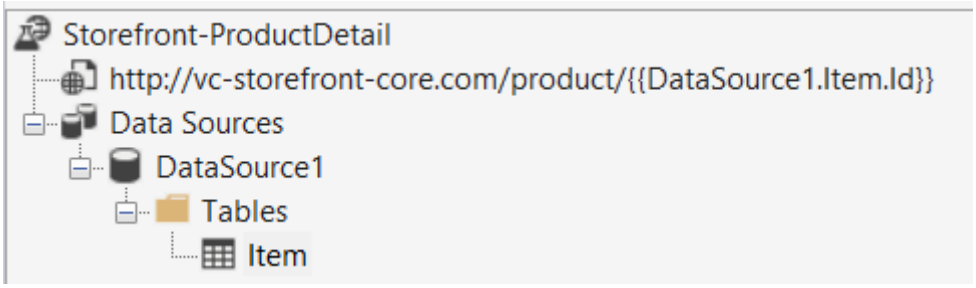


Рис. 3. Содержимое теста

Настало время для первого запуска, попытаемся запустить наш тест и убедиться, что он работает корректно.

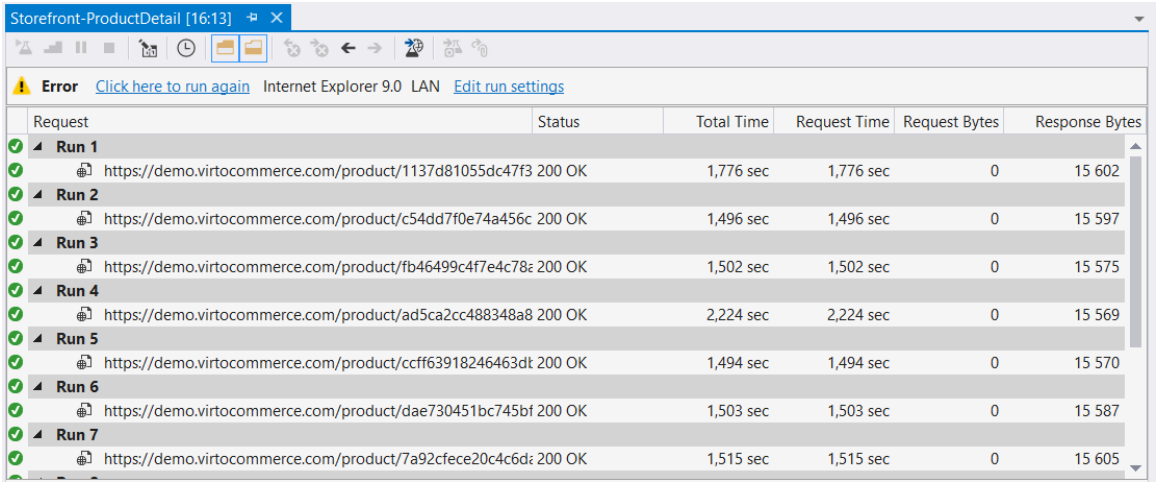


Рис. 4. Результат работы единичного теста

По аналогии создадим тесты для все остальных наших сценариев.

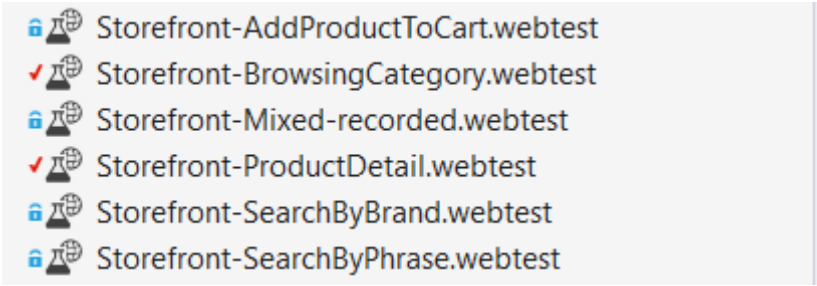


Рис. 5. Результирующий набор тестов

После этого у нас практически все готово к созданию комбинированного теста, который будет эмулировать реальное поведение пользователя на сайте.

Для этого добавляем в наш проект новый **LoadTest.**

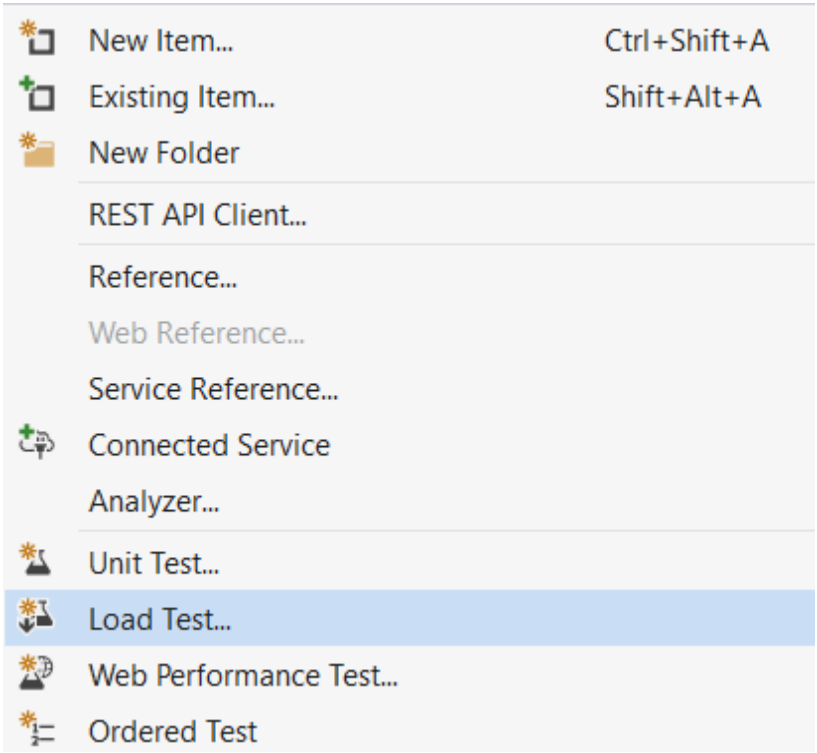


Рис. 6. Создание нового load-теста

В появившемся мастере выбираем тип **On-premises Load test**.

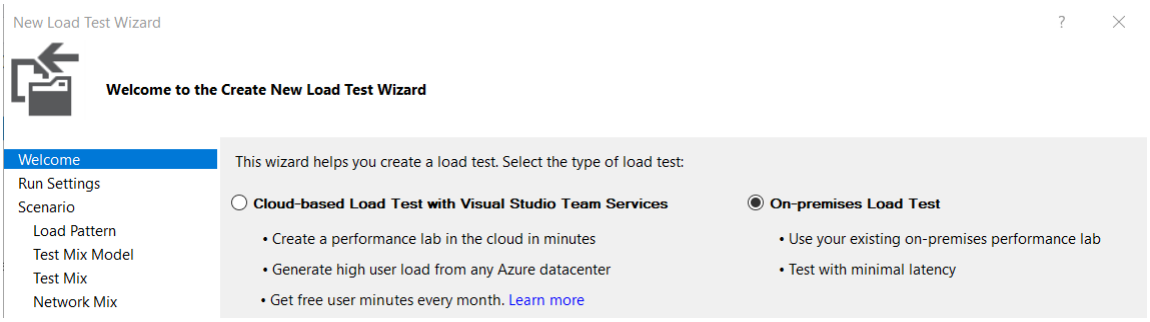


Рис. 7. Выбор типа теста

Этот пункт требует определённого разъяснения, ведь вы справедливо спросите: «Причем тут on-premise?» Тема статьи о тестировании с помощью Teams Services и MS Azure, но тут есть нюанс, так как мы для тестов используем источники данных в виде таблиц или других внешних сервисов, то с этим могут возникнуть определенные сложности, когда мы попытаемся запустить данный тест в облаке.

После тщетных попыток заставить работать такие тесты в облаке мы отказались от этой затеи и решили использовать для тестирования так называемые «записанные» тесты, которые получаются

путем записи запросов, генерируемых тестами работающих локально и имеющих подключение к источникам данных.

Для записи тестов мы используем Fiddler, у которого есть возможность экспорта запросов в формат **Visual Studio Web Tests**. Немного далее мы расскажем более подробно про процедуру записи такого теста.

На последующих шагах выбираем продолжительность тестирования, количество пользователей и, самое главное – указываем, из каких тестов будет состоять наш **MixedLoadTest** и в каких пропорциях они будут использоваться.

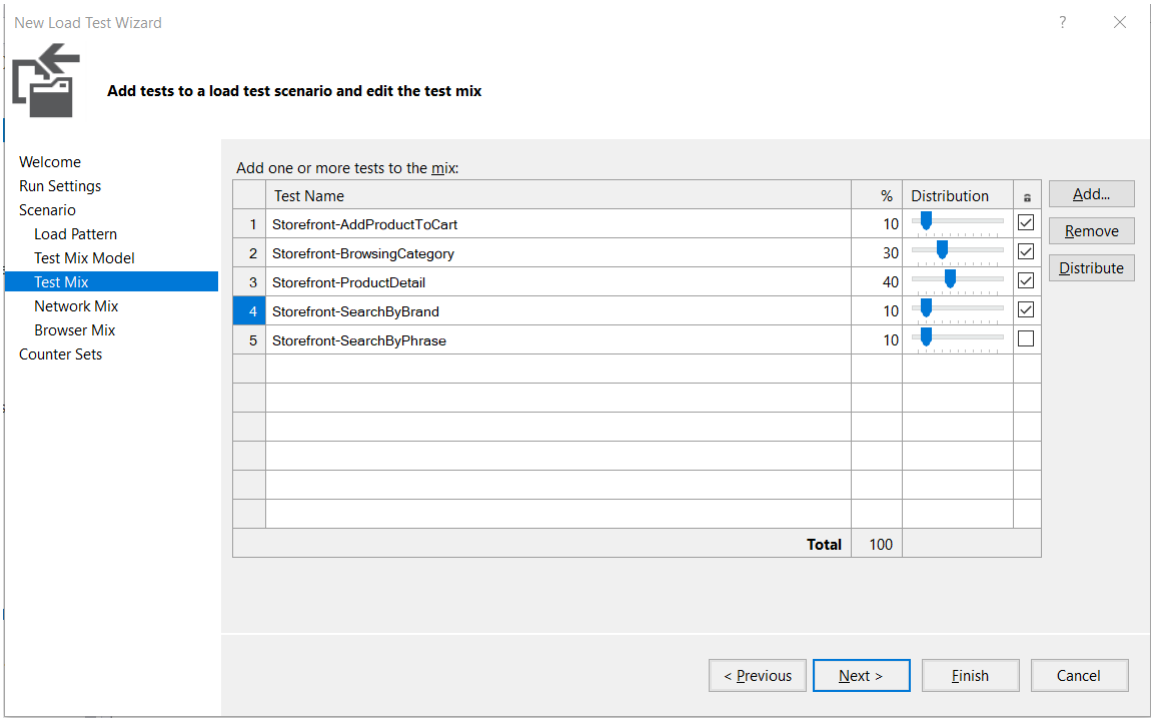


Рис. 8. Составляющие теста

В результате после всех действий мы получим комбинированный **MixedLoadTest**, настроенный для запуска для локально-развернутого приложения.

Далее нам необходимо запустить данный тест и попытаться записать с помощью **Fiddler** все запросы которые будут генерироваться в результате работы теста, а также получить «запись», которую мы сможем запустить уже непосредственно в облаке.

Предварительно запускаем **Fiddler** и наш **MixedLoadTest** тест.

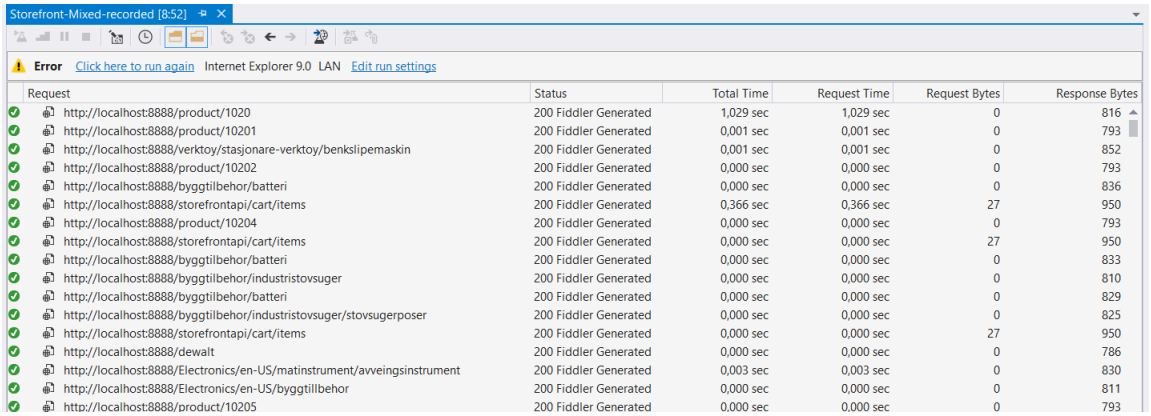


Рис. 9. Результат работы теста

После отработки всех данных получим вот такую картинку в Fiddler.

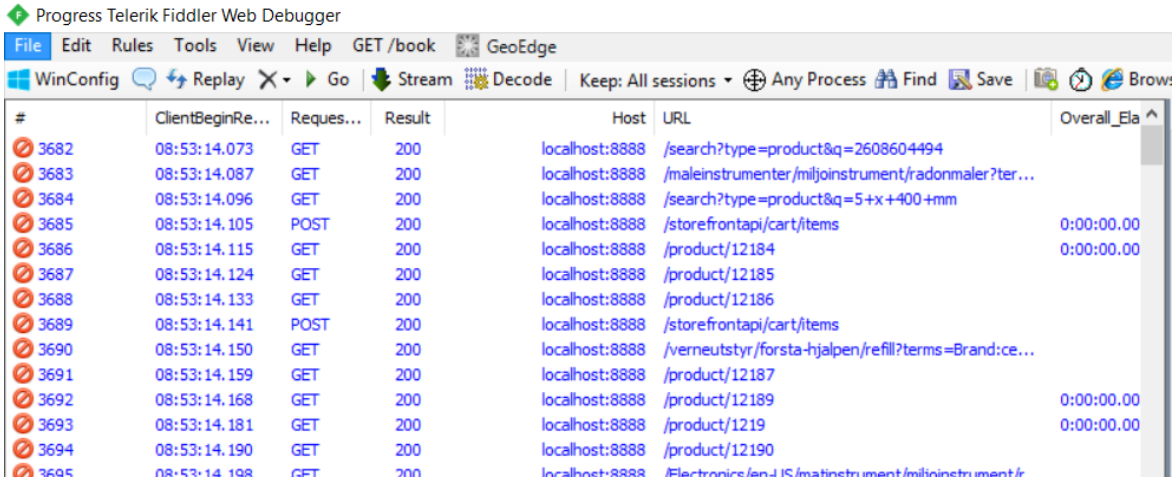


Рис. 10. Сессия теста в Fiddler

Далее в Fiddler сохраняем все сессии в формате **Visual Studio Web Tests**, File -> Export sessions -> All sessions -> Visual Studio Web Tests и добавляем полученный файл в проект. Напомню, что данное действие необходимо для того, чтобы мы смогли получить тест без привязки к внешним источникам данных, так как с запуском такого рода тестов могут возникнуть проблемы в облачной среде.

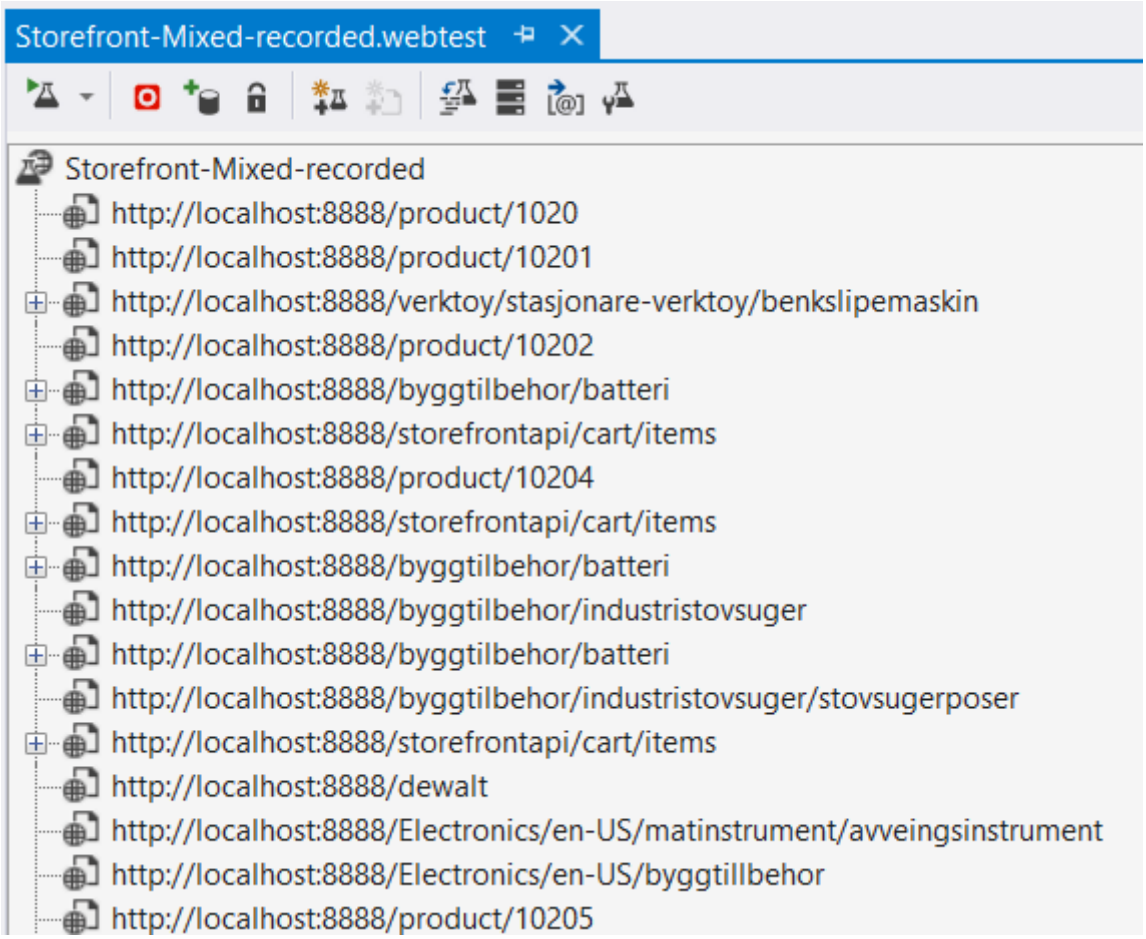


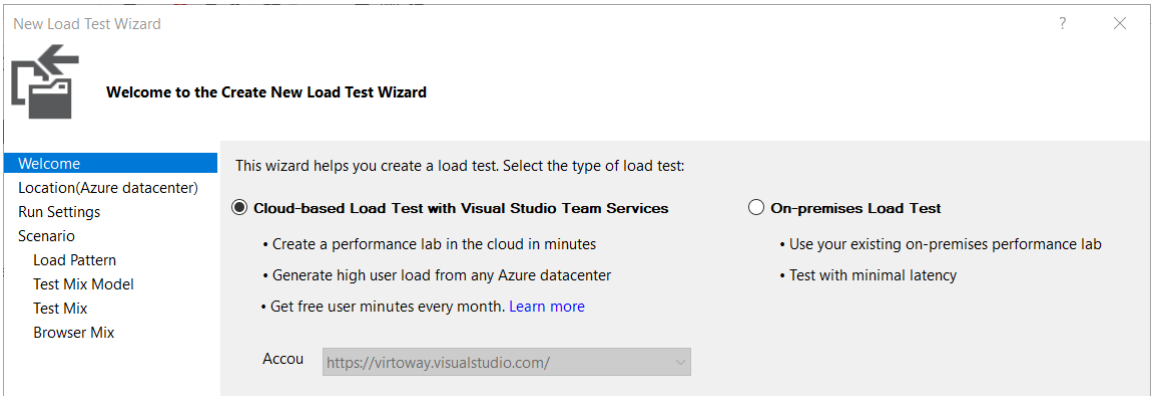
Рис. 11. Детали «записанного» теста

Теперь практически все готово для запуска нашего теста в облаке, последним шагом по подготовке теста нужно в любом текстовом редакторе открыть «записанный» **MixedLoadTest** и заменить в нем localhost:8888 (адрес прокси, Fiddler-a) на адрес нашего магазина в облаке.

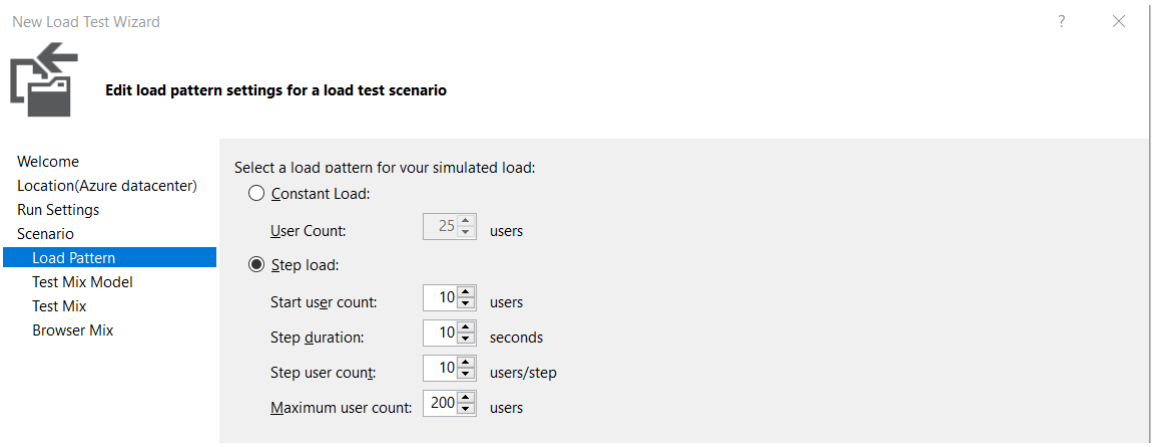
Запуск теста в облаке

Для запуска тестов в облаке нам потребуется действующая учетная запись в **Visual Studio Team Services**.

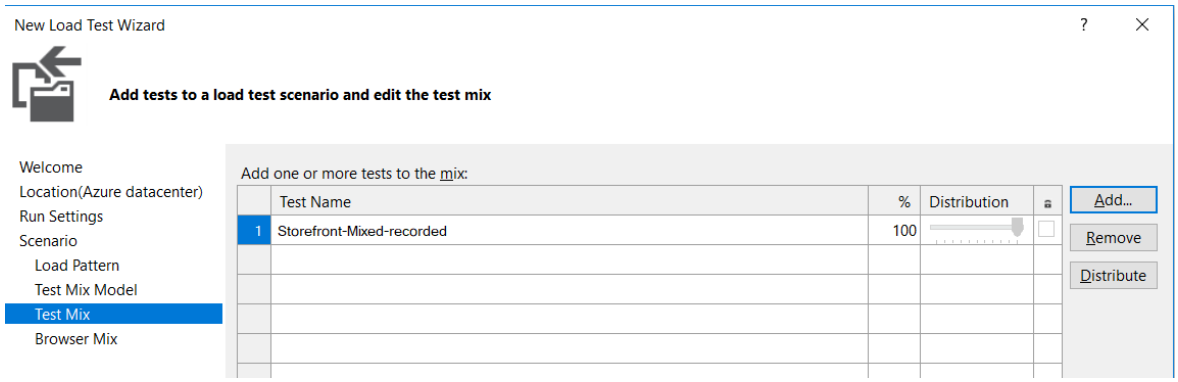
Создаем новый LoadTest, только на этот раз выбираем **Cloud-based Load Test with Visual Studio Team Services**.



На следующих шагах выбираем дата-центр, с которого будет генерироваться трафик на тестируемый ресурс, а также максимальное количество агентов (пользователей) для константного паттерна, либо, если мы хотим использовать постепенное увеличение нагрузки, то необходимо задать соответствующие параметры.



На шаге выбора тестов, выбираем единственный тест, который мы записали ранее с помощью **Fiddler**, он и будет эмулировать «реальную» нагрузку на тестируемый ресурс.



После завершения создания запускаем тест, в процессе выполнения которого студия будет показывать некоторые ключевые метрики, такие как производительность и полоса пропускания, а также строить графики в реальном времени.

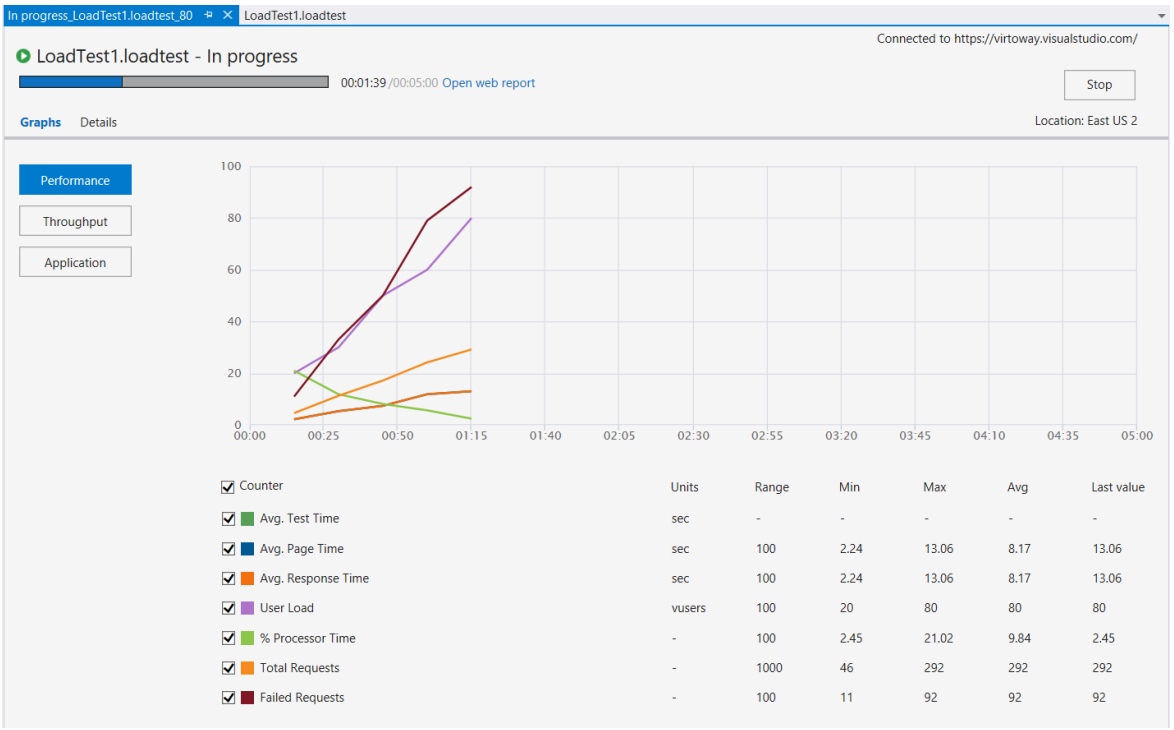


Рис. 12. Процесс работы теста в облаке

После завершения теста также можно посмотреть сохраненный веб отчет в VSTS:

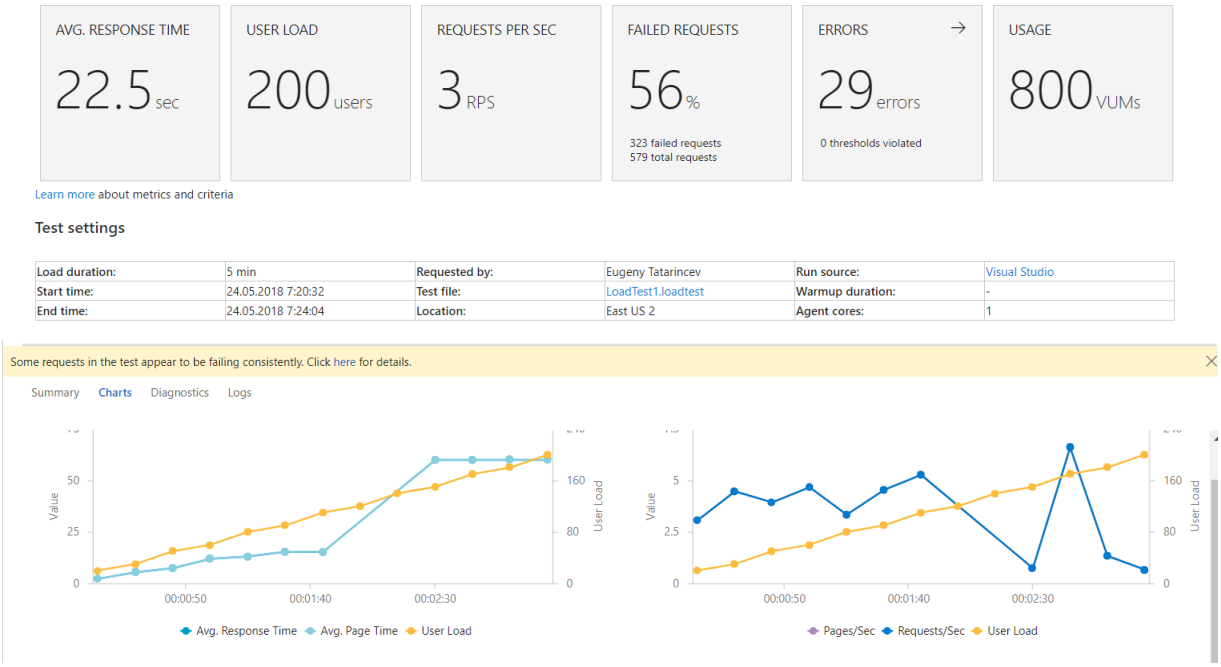


Рис. 13. Web отчет на Visual Studio Team Services портале

Анализ результатов

Самый важный момент – это обработка и анализ полученных результатов теста. Для рассматриваемой задачи требовалось оценить производительность приложения, работающего на различных конфигурация Azure Web Apps B2 и B3 тарифах.

Для этого мы запускали «записанный» тест повторно для приложения на разных конфигурациях и фиксировали полученные результаты в Excel документе.

В итоге получился вот такой отчет:

Рис. 14. Результирующий отчет тестирования

Проанализировав полученные данные, удалось выяснить предельную нагрузку которое может выдержать наше приложение –

она составляет около 60 одновременных пользователей или 9 запросов/сек. при среднем времени отдачи страницы 2.5 сек. На графике видно, что проблемы с производительностью начинаются резко после определённого порогового значения количества запросов.

Контрольные вопросы

1. Как создать тест для нагрузочного тестирования?
2. Как анализировать данные нагрузочного тестирования?
3. Как запустить тест в облаке Azure?

9. Лабораторная работа №6. «Интеграционное тестирование»

Целью работы является изучение интеграционного тестирования ПО. Результатом работы является отчет, в котором должны быть приведены исходные коды программы, результаты инеграционного тестирования ПО.

Для выполнения работы студент должен изучить приведенный ниже теоретический материал. Отчет сдается в распечатанном и электронном (файл Word) видах.

9.1. Задачи и цели интеграционного тестирования

Результатом тестирования и верификации отдельных модулей, составляющих программную систему, должно быть заключение о том, что эти модули являются внутренне непротиворечивыми и соответствуют требованиям. Однако отдельные модули редко функционируют сами по себе, поэтому следующая задача после тестирования отдельных модулей – тестирование корректности взаимодействия нескольких модулей, объединенных в единое целое. Такое тестирование называют интеграционным. Его цель – удостовериться в корректности совместной работы компонент системы.

Интеграционное тестирование называют еще тестированием архитектуры системы. С одной стороны, это название обусловлено тем, что интеграционные тесты включают в себя проверки всех возможных видов взаимодействий между программными модулями и элементами, которые определяются в архитектуре системы – таким образом, интеграционные тесты проверяют полноту взаимодействий в тестируемой реализации системы. С другой стороны, результаты выполнения интеграционных тестов – один из основных источников информации для процесса улучшения и уточнения архитектуры системы, межмодульных и межкомпонентных интерфейсов. То есть, с этой точки зрения, интеграционные тесты проверяют корректность взаимодействия компонент системы.

Примером проверки корректности взаимодействия могут служить два модуля, один из которых накапливает сообщения протокола о принятых файлах, а второй выводит этот протокол на экран. В функциональных требованиях к системе записано, что сообщения должны выводиться в обратном хронологическом порядке. Однако, модуль хранения сообщений сохраняет их в прямом порядке, а модуль вывода использует стек для вывода в обратном порядке. Модульные тесты, затрагивающие каждый модуль по отдельности,

не дадут здесь никакого эффекта – вполне реальна обратная ситуация, при которой сообщения хранятся в обратном порядке, а выводятся с использованием очереди. Обнаружить потенциальную проблему можно только проверив взаимодействие модулей при помощи интеграционных тестов. Ключевым моментом здесь является то, что в обратном хронологическом порядке сообщения выводит система в целом, т. е., проверив модуль вывода и обнаружив, что он выводит сообщения в прямом порядке, мы не сможем гарантировать, что мы обнаружили дефект.

В результате проведения интеграционного тестирования и устранения всех выявленных дефектов получается согласованная и целостная архитектура программной системы, т. е. можно считать, что интеграционное тестирование – это тестирование архитектуры и низкоуровневых функциональных требований.

Интеграционное тестирование, как правило, представляет собой итеративный процесс, при котором проверяется функциональной все более и более увеличивающейся в размерах совокупности модулей.

9.2. Организация интеграционного тестирования

Структурная классификация методов интеграционного тестирования

Как правило, интеграционное тестирование проводится уже по завершении модульного тестирования для всех интегрируемых модулей. Однако это далеко не всегда так. Существует несколько методов проведения интеграционного тестирования:

- восходящее тестирование;
- монолитное тестирование;
- нисходящее тестирование.

Все эти методики основываются на знаниях об архитектуре системы, которая часто изображается в виде структурных диаграмм или диаграмм вызовов функций. Каждый узел на такой диаграмме представляет собой программный модуль, а стрелки между ними представляют собой зависимость по вызовам между модулями. Основное различие методик интеграционного тестирования заключается в направлении движения по этим диаграммам и в широте охвата за одну итерацию.

Восходящее тестирование. При использовании этого метода подразумевается, что сначала тестируются все программные моду-

ли, входящие в состав системы и только затем они объединяются для интеграционного тестирования. При таком подходе значительно упрощается локализация ошибок: если модули протестированы по отдельности, то ошибка при их совместной работе есть проблема их интерфейса. При таком подходе область поиска проблем у тестировщика достаточно узка, и поэтому гораздо выше вероятность правильно идентифицировать дефект.

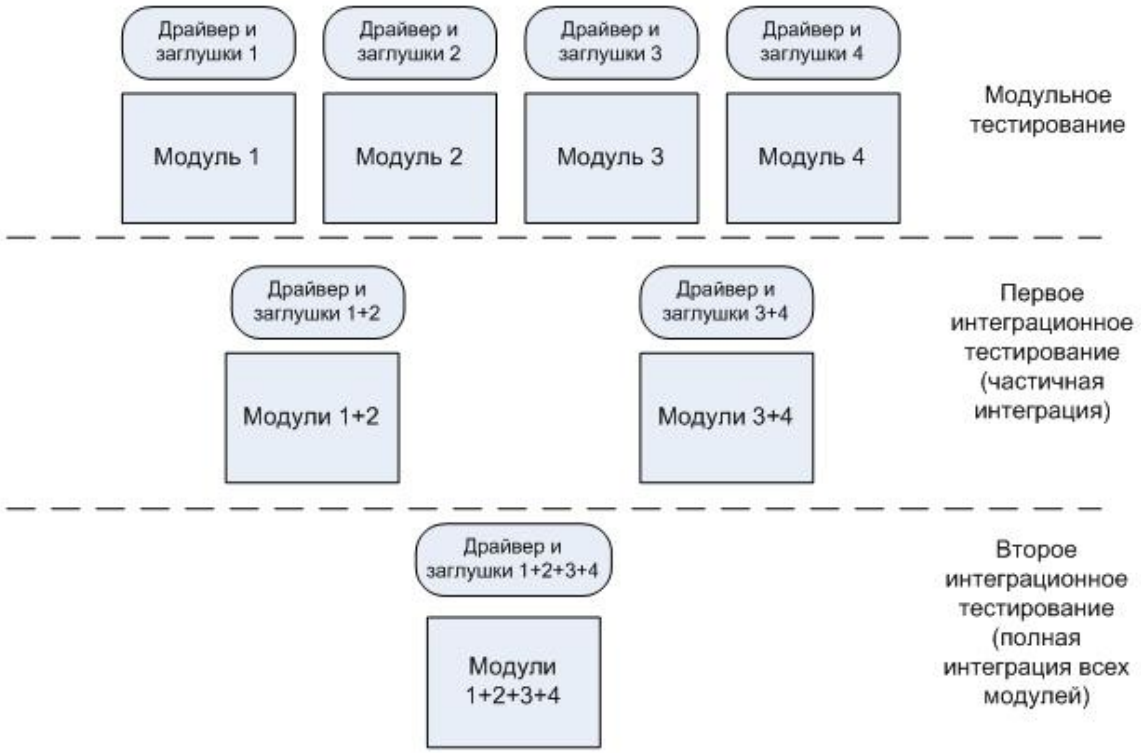


Рис. 1. Разработка драйверов и заглушек
при восходящем интеграционном тестировании

Однако, у восходящего метода тестирования есть существенный недостаток – необходимость в разработке драйвера и заглушек для модульного тестирования перед проведением интеграционного тестирования и необходимость в разработке драйвера и заглушек при интеграционном тестировании части модулей системы (рис. 1)

С одной стороны драйверы и заглушки – мощный инструмент тестирования, с другой – их разработка требует значительных ресурсов, особенно при изменении состава интегрируемых модулей, иначе говоря, может потребоваться один набор драйверов для модульного тестирования каждого модуля, отдельный драйвер и заглушки для тестирования интеграции двух модулей из набора, от-

дельный – для тестирования интеграции трех модулей и т. п. В первую очередь это связано с тем, что при интеграции модулей отпадает необходимость в некоторых заглушках, а также требуется изменение драйвера, которое поддерживает новые тесты, затрагивающие несколько модулей.

Монолитное тестирование предполагает, что отдельные компоненты системы серьезного тестирования не проходили. Основное преимущество данного метода – отсутствие необходимости в разработке тестового окружения, драйверов и заглушек. После разработки всех модулей выполняется их интеграция, затем система проверяется вся в целом. Этот подход не следует путать с системным тестированием, которому посвящена следующая лекция. Несмотря на то, что при монолитном тестировании проверятся работа всей системы в целом, основная задача этого тестирования – определить проблемы взаимодействия отдельных модулей системы. Задачей же системного тестирования является оценка качественных и количественных характеристик системы с точки зрения их приемлемости для конечного пользователя.

Монолитное тестирование имеет ряд серьезных недостатков.

- Очень трудно выявить источник ошибки (идентифицировать ошибочный фрагмент кода). В большинстве модулей следует предполагать наличие ошибки. Проблема сводится к определению того, какая из ошибок во всех вовлечённых модулях привела к полученному результату. При этом возможно наложение эффектов ошибок. Кроме того, ошибка в одном модуле может блокировать тестирование другого.
- Трудно организовать исправление ошибок. В результате тестирования тестировщиком фиксируется найденная проблема. Дефект в системе, вызвавший эту проблему, будет устранять разработчик. Поскольку, как правило, тестируемые модули написаны разными людьми, возникает проблема – кто из них является ответственным за поиск устранение дефекта? При такой «коллективной безответственности» скорость устранения дефектов может резко упасть.
- Процесс тестирования плохо автоматизируется. Преимущество (нет дополнительного программного обеспечения, сопровождающего процесс тестирования) оборачивается недостатком. Каждое внесённое изменение требует повторения всех тестов.

Нисходящее тестирование предполагает, что процесс интеграционного тестирования движется следом за разработкой. Сначала тестируют только самый верхний управляющий уровень системы, без модулей более низкого уровня. Затем постепенно с более высокоуровневыми модулями интегрируются более низкоуровневые. В результате применения такого метода отпадает необходимость в драйверах (роль драйвера выполняет более высокоуровневый модуль системы), однако сохраняется нужда в заглушках (рис. 2).

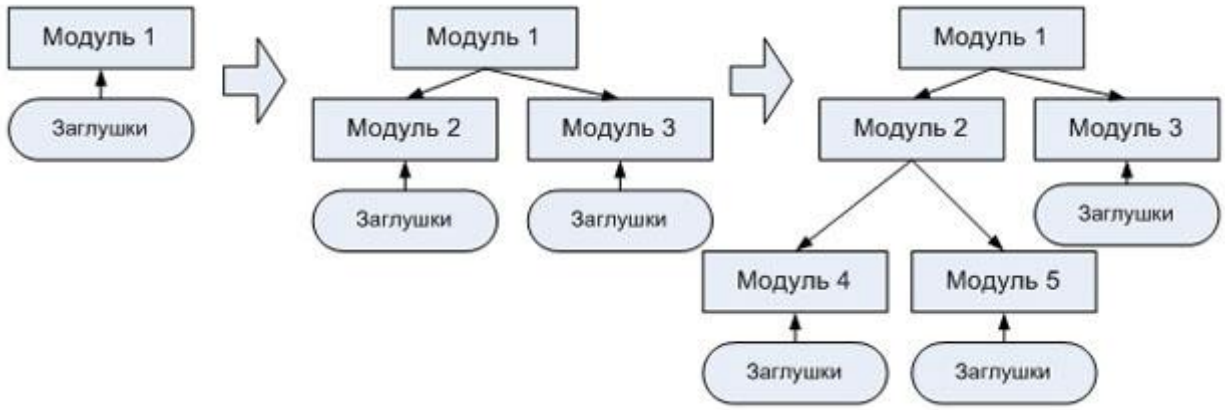


Рис. 2. Постепенная интеграция модулей
при нисходящем методе тестирования

У разных специалистов в области тестирования разные мнения по поводу того, какой из методов более удобен при реальном тестировании программных систем. Йордан доказывает, что нисходящее тестирование наиболее приемлемо в реальных ситуациях, а Майерс полагает, что каждый из подходов имеет свои достоинства и недостатки, но в целом восходящий метод лучше.

В литературе часто упоминается метод интеграционного тестирования объектно-ориентированных программных систем, который основан на выделении кластеров классов, имеющих вместе некоторую замкнутую и законченную функциональность. По своей сути такой подход не является новым типом интеграционного тестирования, просто меняется минимальный элемент, получаемый в результате интеграции. При интеграции модулей на процедурных языках программирования можно интегрировать любое количество модулей при условии разработки заглушек. При интеграции классов в кластеры существует достаточно нестрогое ограничение на закон-

ченность функциональности кластера. Однако, даже в случае объектно-ориентированных систем возможно интегрировать любое количество классов при помощи классов-заглушек.

Вне зависимости от применяемого метода интеграционного тестирования, необходимо учитывать степень покрытия интеграционными тестами функциональности системы.

Временная классификация методов интеграционного тестирования

На практике чаще всего в различных частях проекта применяются все рассмотренные в предыдущем разделе методы в совокупности. Каждый модуль тестируют по мере готовности отдельно, а потом включают в уже готовую композицию. Для одних частей тестирование получается нисходящим, для других – восходящим. В связи с этим представляется полезным рассмотреть еще один тип классификации типов интеграционного тестирования – классификацию по времени интеграции.

В рамках этой классификации выделяют:

- тестирование с поздней интеграцией;
- тестирование с постоянной интеграцией;
- тестирование с регулярной или послойной интеграцией.

Тестирование с поздней интеграцией – практически полный аналог монолитного тестирования. Интеграционное тестирование при такой схеме откладывается на как можно более поздние сроки проекта. Этот подход оправдан в том случае, если система является конгломератом слабо связанных между собой модулей, которые взаимодействуют по какому-либо стандартному интерфейсу, определенному вне проекта (например, в случае, если система состоит из отдельных Web-сервисов). Схематично тестирование с поздней интеграцией может быть изображено в виде цепочки R-C-V-R-C-V-R-C-V-I-R-C-V-R-C-V-I, где R – разработка требований на отдельный модуль, C – разработка программного кода, V – тестирование модуля, I – интеграционное тестирование всего, что было сделано раньше.

Тестирование с постоянной интеграцией подразумевает, что, как только разрабатывается новый модуль системы, он сразу же интегрируется со всей остальной системой. При этом тесты для этого модуля проверяют как сугубо его внутреннюю функциональность, так и его взаимодействие с остальными модулями системы.

Таким образом, этот подход совмещает в себе модульное тестирование и интеграционное. Разработки заглушек при таком подходе не требуется, но может потребоваться разработка драйверов. В настоящее время именно этот подход называют unit testing, несмотря на то, что в отличие от классического модульного тестирования здесь не проверяется функциональность изолированного модуля. Локализация ошибок межмодульных интерфейсов при таком подходе несколько затруднена, но все же значительно ниже, чем при монолитном тестировании. Большая часть таких ошибок выявляется достаточно рано именно за счет частоты интеграции и за счет того, что за одну итерацию тестирования проверяется сравнительно небольшое число межмодульных интерфейсов.

Схематично тестирование с постоянной интеграцией может быть изображено в виде цепочки R-C-I-R-C-I-R-C-I, в которой фаза тестирования модуля намеренно опущена и заменена на тестирование интеграции.

При **тестировании с регулярной или послойной интеграцией** интеграционному тестированию подлежат сильно связанные между собой группы модулей (слои), которые затем также интегрируются между собой. Такой вид интеграционного тестирования называют также иерархическим интеграционным тестированием, поскольку укрупнение интегрированных частей системы, как правило, происходит по иерархическому принципу. Однако, в отличие от нисходящего или восходящего тестирования, направление прохода по иерархии в этом подходе не задано.

В табл. 1 приведены основные характеристики рассмотренных в данной лекции видов интеграционного тестирования. Время интеграции характеризует момент, когда проводится первое интеграционное тестирование и все последующие. Частота интеграции – насколько часто при разработке выполняется интеграция. Необходимость в драйверах и заглушках определена в последних двух строках таблицы.

Таблица 1

Основные характеристики различных видов интеграционного тестирования

Свойство	Вид интеграции					
	Восходящее	**Нисходящее**	**Монолитное**	**Поздняя интеграция**	**Постоянная интеграция**	**Регулярная интеграция**
Время интеграции	Поздно (после тестирования модулей)	Рано (параллельно с разработкой)	Поздно (после разработки всех модулей)	Поздно (после разработки всех модулей)	Рано (параллельно с разработкой)	Рано (параллельно с разработкой)
Частота интеграции	Редко	Часто	Редко	Редко	Часто	Часто
Нужны ли драйверы	Да	Нет	Нет	Нет	Да	Да
Нужны ли заглушки	Да	Да	Нет	Нет	Нет	Да

Планирование интеграционного тестирования

Процесс организации и планирования интеграционного тестирования во многом схож с процессом организации модульного тестирования, рассмотренном в предыдущей лекции. Однако интеграционное тестирование имеет ряд организационных особенностей, перечисленных ниже.

На этапе планирования разрабатывается концепция и стратегия интеграции – документ, где описан общий подход к определению последовательности, в которой должны интегрироваться модули. Как правило, концепция основывается на одном из видов интеграции, рассмотренных выше (например, на нисходящей), но учитывает особенности конкретной системы (например, вначале должны интегрироваться компоненты работы с базой данных, затем пользовательского интерфейса; затем интерфейсные компоненты и компоненты работы с БД интегрируются вместе).

Составляется интеграционный тест-план, например, кластерного типа, в котором для каждого кластера из интегрированных модулей определяется следующее:

- кластеры, от которых зависит данный кластер;

- кластеры, которые должны быть протестированы до тестирования данного кластера;
- описание функциональности тестируемого кластера;
- список модулей в кластере;
- описание тестовых примеров для проверки кластера;

Планирование интеграционного тестирования должно быть синхронизировано с общим планом проекта, причем выделяемые для интеграционного тестирования кластеры и сроки их тестирования должны учитывать приоритеты важности частей системы. Зачастую рассмотрение приоритетов связано с тем, что системы разрабатываются в несколько этапов, на каждом из которых в эксплуатацию вводится только часть новой системы. Интеграционное тестирование в данном случае должно укладываться в общий план-график проекта и учитывать затраты ресурсов на тестирование интеграции с уже работающими частями системы.

Контрольные вопросы

1. Что такое интеграционное тестирование?
2. Какие виды интеграционного тестирования существуют?
3. Что такое «заглушка»?

10. Лабораторная работа №7. «Конфигурационное тестирование»

Целью работы является изучение конфигурационного тестирования ПО. Результатом работы является отчет, в котором должны быть приведены исходные коды программы, конфигурационного тестирования ПО.

Для выполнения работы студент должен изучить приведенный ниже теоретический материал. Отчет сдается в распечатанном и электронном (файл Word) видах.

Конфигурационное тестирование (Configuration Testing) – специальный вид тестирования, направленный на проверку работы программного обеспечения при различных конфигурациях системы (заявленных платформах, поддерживаемых драйверах, при различных конфигурациях компьютеров и т. д.).

В зависимости от типа проекта конфигурационное тестирование может иметь разные цели:

1. Проект по профилированию работы системы

Цель Тестирования: определить оптимальную конфигурацию оборудования, обеспечивающую требуемые характеристики производительности и времени реакции тестируемой системы.

2. Проект по миграции системы с одной платформы на другую

Цель Тестирования: Проверить объект тестирования на совместимость с объявленным в спецификации оборудованием, операционными системами и программными продуктами третьих фирм.

Примечание: В **ISTQB Syllabus** вообще не говорится о таком виде тестирования как конфигурационное. Согласно глоссарию, данный вид тестирования рассматривается там как тестирование портируемости: configuration testing: See **portability testing**. **portability testing**: The process of testing to determine the portability of a software product.

1. Уровни проведения тестирования

Для клиент-серверных приложений конфигурационное тестирование можно условно разделить на два уровня (для некоторых типов приложений может быть актуален только один):

1. Серверный.

2. Клиентский.

На первом (серверном) уровне, тестируется взаимодействие выпускаемого программного обеспечения с окружением, в которое оно будет установлено:

1. Аппаратные средства (тип и количество процессоров, объем памяти, характеристики сети / сетевых адаптеров и т. д.).

2. Программные средства (ОС, драйвера и библиотеки, стороннее ПО, влияющее на работу приложения и т. д.).

Основной упор здесь делается на тестирование с целью определения оптимальной конфигурации оборудования, удовлетворяющего требуемым характеристикам качества (эффективность, портативность, удобство сопровождения, надежность).

На следующем (клиентском) уровне, программное обеспечение тестируется с позиции его конечного пользователя и конфигурации его рабочей станции. На этом этапе будут протестированы следующие характеристики: удобство использования, функциональность. Для этого необходимо будет провести ряд тестов с различными конфигурациями рабочих станций:

1. Тип, версия и битность операционной системы (подобный вид тестирования называется **кросс-платформенное тестирование**).

2. Тип и версия Web барузера, в случае если тестируется Web приложение (подобный вид тестирования называется **кросс-браузерное тестирование**).

3. Тип и модель видео адаптера (при тестировании игр это очень важно).

4. Работа приложения при различных разрешениях экрана.

5. Версии драйверов, библиотек и т. д. (для JAVA приложений версия JAVA машины очень важна, тоже можно сказать и для .NET приложений касательно версии .NET библиотеки) и т. д.

3. Порядок проведения тестирования

Перед началом проведения конфигурационного тестирования рекомендуется:

- создавать матрицу покрытия (**матрица покрытия** – это таблица, в которую заносят все возможные конфигурации);
- проводить приоритезацию конфигураций (на практике, скорее всего, все желаемые конфигурации проверить не получится);

- шаг за шагом, в соответствии с расставленными приоритетами, проверяют каждую конфигурацию.

Уже на начальном этапе становится очевидно, что чем больше требований к работе приложения при различных конфигурациях рабочих станций, тем больше тестов нам необходимо будет провести. В связи с этим, рекомендуем, по возможности, автоматизировать этот процесс, так как именно при конфигурационном тестировании автоматизация реально помогает сэкономить время и ресурсы. Конечно же автоматизированное тестирование не является панацеей, но в данном случае оно окажется очень эффективным помощником.

Контрольные вопросы

1. Что такое конфигурационное тестирование?
2. Какие основные цели преследует конфигурационное тестирование?
3. Что такое матрица покрытия?

11. Лабораторная работа №8. «Тестирование установки»

Целью работы является изучение тестирования установки ПО. Результатом работы является отчет, в котором должны быть приведены исходные коды программы, результаты тестирования установки ПО.

Для выполнения работы студент должен изучить приведенный ниже теоретический материал. Отчет сдается в распечатанном и электронном (файл Word) видах.

11.1. Тестирование Установки или Installation Testing

Тестирование установки направленно на проверку успешной инсталляции и настройки, а также обновления или удаления программного обеспечения.

В настоящий момент наиболее распространена установка ПО при помощи **инсталляторов** (специальных программ, **которые сами по себе** так же **требуют надлежащего тестирования**).

В реальных условиях инсталляторов может не быть. В этом случае придется самостоятельно выполнять установку программного обеспечения, используя документацию в виде инструкций или readme файлов, шаг за шагом описывающих все необходимые действия и проверки.

В распределенных системах, где приложение разворачивается на уже работающем окружении, простого набора инструкций может быть мало. Для этого, зачастую, пишется план установки (**Deployment Plan**), включающий не только шаги по инсталляции приложения, но и шаги отката (**roll-back**) к предыдущей версии, в случае неудачи. Сам по себе **план установки также должен пройти процедуру тестирования** для избежания проблем при выдаче в реальную эксплуатацию. Особенно это актуально, если установка выполняется на системы, где каждая минута простоя – это потеря репутации и большого количества средств, например: банки, финансовые компании или даже баннерные сети. Поэтому тестирование установки можно назвать одной из важнейших задач по обеспечению качества программного обеспечения.

Именно такой комплексный подход с написанием планов, пошаговой проверкой установки и отката инсталляции, полноправно можно назвать **тестированием установки** или **Installation Testing.**

11.2. Особенности тестирования инсталляторов

Инсталлятор – это «обычная» программа, основные функции которой – Установка (Инсталляция), Обновление и Удаление (Деинсталляция) программного обеспечения.

Всем известна народная мудрость: **»Встречают по одежке, а провожают по уму»**. Инсталляционное приложение и есть та самая одежка, по которой создается первое впечатление о Вашем продукте. Именно поэтому тестирование установки – это одна из важнейших задач.

Являясь обычной программой, инсталлятор обладает рядом особенностей, среди которых стоит отметить следующие:

- Глубокое взаимодействие с операционной системой и зависимость от неё (файловая система, реестр, сервисы и библиотеки).
- Совместимость как родных, так и сторонних библиотек, компонент или драйверов, с разными платформами.
- Удобство использования: интуитивно понятный интерфейс, навигация, сообщения и подсказки.
- Дизайн и стиль инсталляционного приложения.
- Совместимость пользовательских настроек и документов в разных версиях приложения.
- И многое другое.

Если эти особенности не зарядили Вас на серьезное отношение к тестированию инсталляционных программ, то хочу привести небольшой **список рисков, который покажет всю значимость корректной работы инсталляторов:**

- Риск Потери Пользовательских Данных.
- Риск Вывода Операционной Системы Из Строя.
- Риск Неработоспособности Приложения.
- Риск Не Корректной Работы Приложения.

В тоже время, как и на любую программу, **на инсталлятор накладываются** некоторые **функциональные требования**. Объединив их со списком особенностей, мы получим более полную картину, показывающую объем предстоящих работ по тестированию

В большинстве случаев инсталлятор представляет собой приложение в виде мастера (Wizard), которое может обладать специфическими требованиями. С современным изобилием персональных компьютеров, серверов и операционных систем, возникла потребность в установки одного и того же программного обеспечения на

разные платформы. Для этого инсталляторы должны понимать что и куда они устанавливают в зависимости от окружения.

Распишем подробнее, «**Что?**» необходимо проверить, для оценки правильности работы инсталлятора:

- **Установка (Инсталляция)**

1. Наличие достаточных для установки приложения ресурсов таких как: оперативная память, дисковое пространство и т.д.
2. Корректность списка файлов в инсталляционном пакете:
   a. при выборе различных типов установки, либо установочных параметров список файлов и пути к ним также могут отличаться;
   b. отсутствие лишних файлов (проектные файлы, не включенные в инсталляционный пакет, не должны попасть на диск пользователя).
3. Регистрация приложения в ОС.
4. Регистрация расширений для работы с файлами:
   a. для новых расширений;
   b. для уже существующих расширений.
5. Права доступа пользователя, который ставит приложение:
   a. права на работу с системным реестром;
   b. права на доступ к файлам и папкам, например %Windir%\system32.
6. Корректность работы мастера установки (Installation Wizard.)
7. Инсталляция нескольких приложений за одни заход.
8. Установка одного и того же приложения в разные рабочие директории одной рабочей станции.

- **Обновление**

1. Правильность списка файлов, а так же отсутствие лишних файлов:
   a. проверка списка файлов при разных параметрах установки;
   b. отсутствие лишних файлов.
2. Обратная совместимость создаваемых данных
   a. сохранность и корректная работа созданных до обновления данных;
   b. возможность корректной работы старых версий приложения с данными, созданными в новых версиях.
3. Обновление при запущенном приложении.
4. Прерывание обновления.

- **Удаление (Деинсталляция)**

1. Корректное удаление приложения:

a. удаление из системного реестра установленных в процессе инсталляции библиотек и служебных записей;

b. удаление физических файлов приложения;

c. удаление/восстановление предыдущих файловых ассоциаций;

d. сохранность файлов созданных за время работы с приложением.

2. Удаление при запущенном приложении.

3. Удаление с ограниченным доступом к папке приложения.

4. Удаление пользователем без соответствующих прав.

«**Как тестировать Инсталляции?**»

- **Установка (Инсталляция)**

1. Ресурсы необходимые для установки программного обеспечения должны анализироваться именно в начале инсталляционного процесса. В случае их нехватки пользователь должен получать соответствующее предупреждение.

2. Получение списка файлов должно проводиться До, а проверка самих файлов – После установки!

a. самое правильное – это попытаться получить список файлов от сборщика инсталляционного пакета. Фраза: «Возьмите и составьте список после установки ПО, там все верно» – это провокация и на нее лучше не поддаваться;

b. если программа содержит файлы подписанные сертификатом от MS, то не исключено, что могут попадаться временные файлы, которые уже не нужны для работы приложения (*.tmp и т. д.). Обратите внимание, что один и тот же инсталляционный пакет может устанавливать под разные ОС разные наборы файлов. Поэтому тестировать надо делать под каждую ОС отдельно.

3. Практически для любой ОС важна регистрация рабочих библиотек и служебных записей. Например, в ОС Windows вам необходимо будет проверить системный реестр на предмет корректной записи новых данных и регистрации новых/нового приложения.

4. После установки приложения необходимо проверить правильность регистрации расширений для рабочих файлов установленной программы в ОС. Сделайте двойной щелчок по документу, в результате он должен открыться вашим приложением. Если же вы

получите в результате диалог выбора программы для открытия файлов данного типа или же он будет открыт другим приложением, то налицо будет ошибка регистрации расширения в ОС.

5. По-хорошему инсталляционная программа должна при старте проверять учетную запись пользователя и сразу корректно сообщать о каких-либо проблемах с правами. Но не исключен вариант, что на какие-нибудь служебные папки будут установлены дополнительные ограничения. Можно попытаться самим смоделировать ситуации когда какая-нибудь из папок закрыта для записи, например «\Program Files», «\Windows», «%Windir%\system32», а также проверить как будет себя вести приложение при невозможности записать какие-нибудь из файлов по нужному пути. Не исключен вариант, что без некоторых файлов работоспособность всего приложения не будет нарушена. В этом случае достаточно указать о проблеме с копированием файла(ов) в лог и НЕ выводить сообщение об ошибке.

6. Необходимо провести полное **Тестирование мастера установки (Installation Wizard)** приложения.

7. Достаточно часто встречаются ситуации, когда приложение помимо себя самого ставит еще какой-нибудь продукт. Это может быть, как отдельное стороннее приложение, так и демонстрация своего же продукта. В процессе установки обычно это будет не еще один набор шагов мастера установки, а просто небольшое сообщение, что мол ставится такой-то продукт. При тестировании необходимо будет обратить особое внимание на то, что мы имеем последовательную установку нескольких продуктов. Если первый из них требует перезагрузку после своей установки, то второй может инсталлироваться некорректно, особенно если и там и там ставятся драйвера вглубь системы. Интерес представляют ошибки, возникающие именно на стыке установки двух приложений.

8. Встречаются приложения, которые можно установить в разные рабочие директории одной и той же рабочей станции, и при этом они будут работать независимо друг от друга, не создавая никаких конфликтных ситуаций. Но это не всегда так. Могут возникать конфликты с доступам к общим ресурсам на диске, в памяти и/или в системе. Все это должно быть протестировано тщательнейшим образом.

- **Обновление**

1. По аналогии с **пунктом 2** раздела «Установка (Инсталляция)».

2. Проверка обратной совместимости включает следующие шаги:

a. после установки обновления, все ранее созданные приложением объекты, такие как документы, формы, сохранения (если это игра) должны открываться и работать без ошибок. Такое поведение называется обратная совместимость (backward compatibility). Пользовательские настройки должны остаться прежними, конечно если обновления не затрагивали именно их изменение;

b. созданные в новой версии однотипные документы должны корректно открываться в старых версиях, конечно если целью обновления не стояло изменение формата и структуры файлов. Если же был внедрен новый формат, то в новой версии должна быть реализована возможность сохранения документа в старом формате.

3. В случае если обновляемое приложение запущено, пользователь должен получить предупреждение о том, что обновление невозможно, при работающем приложении.

4. Заметим, что процесс обновления может быть остановлен в любой момент, при этом исходное приложение должно остаться не изменённым и в рабочем состоянии. Допустим, у вас установлено приложение версии 1, вы пытаетесь обновить его до версии 2, но в процессе установки передумали и прервали установку. В этом случае инсталлятор должен вернуть все уже сделанные изменения, почистить использованные для обновления временные файлы и завершить свою работу. При этом приложение версии 1 остается в рабочем состоянии.

- **Удаление (Деинсталляция)**

1. Хорошей практикой считается удаление созданного в процессе инсталляции (регистрационные записи в системном реестре, библиотеки в системных каталогах %Windir%\system32, файлы и т. д.). Условно процесс тестирования деинсталляции можно разбить на несколько частей:

a. проверка, что из системного реестра удалены, установленные в процессе инсталляции, служебные записи и ссылки на библиотеки;

b. проверка физического удаления файлов приложения;

c. проверка того, что после удаления приложения, зарегистрированные во время установки файловые расширения удалены, а ранее существующие (зарегистрированные до инсталляции) – восстановлены;

d. проверка сохранности данных созданных за время работы с приложением. Вполне вероятно, что лежат они где-то в глубине каталога самой программы. Это могут быть служебные скрипты, сохранения от игр или прочие созданные пользователем данные, удаление которых нанесет урон пользователю. Только предположите, что при удалении MS Office, все Ваши документы будут удалены вместе с ним. Поэтому подобные данные нельзя удалять без подтверждения пользователя.

2. В случае, если удаляемое приложение запущено, пользователь должен получить предупреждение о том, что удаление невозможно, пока приложение работает.

3. Стоит проверить поведение инсталлятора, если каталог программы закрыт для удаления по правам доступа. В данном случае процесс удаления не должен производиться, и пользователь должен получить соответствующее сообщение.

4. Если пользователь не авторизован для удаления приложения, то он должен получить соответствующее сообщение и процесс удаления должен быть прерван.

Тестирование мастера установки или Installation Wizard Testing

Умные люди писали: «Визарды – это зло». С этим можно соглашаться или нет, но тестировать их все равно приходится. Предлагается следующий план тестирования инсталляционного визарда:

1. Определить все пути от начала до конца, и затем расставить приоритеты для каждого из них. Это поможет нам избежать излишних затрат и усилий при прохождении низкоприоритеных путей.

2. Забудьте про GUI. Постарайтесь описать тест-кейсы без привязки к интерфейсным элементам. К примеру, GUI контролы checkbox/radiobutton или меню из двух пунктов это просто выбор между true и false, важно то, на что он влияет в конечном счете.

3. Если по результатам прохождения визарда получается какой либо проперти файл (файл, описывающий свойства в виде списка: **свойство=значение**), который потом передается дальше в процедуру экспорта. В этом случае можно разделить проверки на два

этапа – первый, создавать (генерировать) такие проперти файлы и проверять, что экспорт работает правильно. Второй – проверять, что через GUI получаются правильные проперти файлы.

4. Не забудьте заняться таким рутинным видом тестирования визардов, как ходить туда-обратно по страницам:

a. ничего не меняя, все ответы должны сохраняться;

b. меняя что-либо на предыдущей странице, на следующей должно произойти адекватное изменение либо сброс ответов.

5. Убедитесь, что визард адекватно реагирует на неправильные ответы и не дает ходить дальше.

6. Кнопка Cancel (Close) должна работать всегда и на всех страницах визарда.

7. Создайте для каждого из возможных путей мастера установки шаблонный результат (в идеале, сделайте их несколько – для разных входных данных). Затем, по возможности, автоматизированно или вручную сравнивайте полученный результат с шаблонным.

8. Выделите те опции, которые не влияют ни на какие другие и на которые другие не оказывают влияния. Работу этих опций можно будет тестировать изолированно от других.

Контрольные вопросы

1. Что такое тестирование инсталляции?
2. Какие этапы содержит тестирование инсталляции?
3. Как тестировать установщик?

СОДЕРЖАНИЕ