МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

СЕВЕРО-КАВКАЗСКАЯ ГОСУДАРСТВЕННАЯ АКАДЕМИЯ

СРЕДНЕПРОФЕССИОНАЛЬНЫЙ КОЛЛЕДЖ

Д.Ф. Мамхягов

МДК 05.03 ТЕСТИРОВАНИЕ ИНФОРМАЦИОННЫХ СИСТЕМ

ПРАКТИКУМ
для студентов III курса специальности
09.02.07 Информационные системы и программирование

Черкесск
2024

УДК 004.415.53
ББК 16.33:32.973
М 22

Рассмотрено на заседании ЦК «Информационные технологии»
Протокол № 1 от «01» 09. 2023 г.
Рекомендовано к изданию редакционно-издательским советом СКГА
Протокол № 26 от «29» 09. 2023 г.

Рецензенты: Моисеенко Л.А. – преподаватель СПК «СКГА»

М22 **Мамхягов, Д. Ф.** МДК 05.03. Тестирование информационных систем: практикум для студентов III курса специальности 09.02.07 Информационные системы и программирование / Д.Ф. Мамхягов. – Черкесск: БИЦ СКГА, 2024. –44 с.

Практикум содержит типовые задания по изучению пакета программ MS Office. Выполнение практических заданий позволит развить навыки студентов в области компьютерных информационных технологий.

УДК 004.415.53
ББК 16.33:32.973

© Мамхягов Д.Ф., 2024
© ФГБОУ ВО СКГА, 2024

СОДЕРЖАНИЕ

Введение 4
Практические работа 1. Разработка тестового сценария проекта. 5
Практические работа 2. Разработка тестовых пакетов 7
Практические работа 3. Использование инструментария анализа качества. 9
Практические работа 4. Анализ и обеспечение обработки исключительных ситуаций. 12
Практические работа 5. Функциональное тестирование 18
Практические работа 6. Тестирование безопасности. 23
Практические работа 7. Нагрузочное тестирование, стрессовое тестирование. 25
Практические работа 8. Тестирование интеграции 33
Практические работа 9. Конфигурационное тестирование 34
Практические работа 10. Тестирование установки 36
Список литературы 41

ВВЕДЕНИЕ

Практикум подготовлен в соответствии с Федеральным государственным образовательным стандартом среднего профессионального образования по учебной дисциплине «Тестирование информационных систем». Практикум содержит типовые задания, предполагающие изучение приемов обработки информации, технологий разработки тестовых сценариев. Выполнение практических заданий позволит развить навыки студентов в области компьютерных информационных технологий.

Практикум адресован студентам специальности 09.02.07 Информационные системы и программирование, рекомендуются для использования в среднеспециальных учебных заведениях соответствующего профиля.

ПРАКТИЧЕСКАЯ РАБОТА №1.
Тема: Разработка тестового сценария проекта.

Цель работы: Приобретение навыков создания тестового сценария..

Для разработки тестового сценария проекта вам понадобится четко определить цели тестирования, описать шаги, которые необходимо выполнить, и критерии, по которым будет оцениваться успешность теста. Вот примерный план создания тестового сценария:

1. Определение целей тестирования. Что именно вы хотите проверить? Это может быть функциональность, производительность, безопасность или совместимость.

2. Выбор тестовых данных. Какие данные будут использоваться для тестирования? Убедитесь, что они репрезентативны и соответствуют реальным условиям использования.

3. Описание шагов тестирования. Какие действия необходимо выполнить? Опишите последовательность действий, которые должен выполнить тестировщик.

4. Ожидаемые результаты. Чего вы ожидаете после выполнения тестовых шагов? Это поможет определить, прошел ли тест успешно.

5. Фактические результаты и оценка. Запишите, что произошло в результате тестирования и сравните это с ожидаемыми результатами.

6. Документирование и анализ. Все результаты должны быть задокументированы. Если тест не пройден, проанализируйте причины и определите шаги для устранения проблем.

Для создания детализированного тестового сценария для программы на Visual Studio, вам нужно будет следовать этим шагам:

1. Описание проекта и тестируемой функциональности. Кратко опишите проект и функциональность, которую вы собираетесь тестировать.

2. Тестовое окружение. Укажите конфигурацию тестового окружения, включая версию Visual Studio, операционную систему и любое другое ПО, необходимое для тестирования.

3. Предварительные условия. Перечислите все условия, которые должны быть выполнены перед началом теста (например, запущенные сервисы, настроенные параметры).

4. Тестовые данные. Определите и подготовьте тестовые данные, которые будут использоваться в тесте.

5. Шаги тестирования.

Описание действия.

– Ожидаемый результат: Что должно произойти после выполнения шага.

– Фактический результат: Записывается во время выполнения теста.

– Статус: Успешно / Не успешно.

Продолжите описание шагов до завершения тестового сценария.

6. Постусловия: Опишите состояние системы после выполнения теста.

7. Критерии успеха: Определите, как будет измеряться успех теста.

8. Очистка: Опишите шаги для возврата системы в исходное состояние после тестирования.

9. Документирование результатов: Запишите результаты тестирования и сделайте выводы о качестве программы.

10. Анализ и действия по улучшению: Если тесты выявили проблемы, опишите шаги для их устранения.

Вот пример тестового сценария для функции логина в приложении:

```
**Тестовый сценарий**: Проверка функции логина.

**Тестовое окружение**: Visual Studio 2019, Windows 10.

**Предварительные условия**: Приложение установлено и запущено.

**Тестовые данные**: Пользовательские учетные данные (логин
и пароль).

**Шаги тестирования**:
1. Введите корректный логин и пароль.
   - **Ожидаемый результат**: Пользователь успешно вошел в
систему.
   - **Фактический результат**: ...
   - **Статус**: ...

2. Введите некорректный логин и пароль.
   - **Ожидаемый результат**: Система отображает сообщение
об ошибке.
   - **Фактический результат**: ...
   - **Статус**: ...

**Постусловия**: Пользователь выходит из системы.

**Критерии успеха**: Все шаги выполнены успешно, и
результаты соответствуют ожидаемым.

**Очистка**: Закрыть приложение.

**Документирование результатов**: Результаты записаны и
проанализированы.

**Анализ и действия по улучшению**: Определены шаги для
исправления обнаруженных проблем.
```

Этот тестовый сценарий может быть адаптирован под различные функции вашего приложения.

Задание: Разработать тестовый сценирий на основе выбранной предметной области.

ПРАКТИЧЕСКАЯ РАБОТА № 2.
Тема: Разработка тестовых пакетов

Цель работы: Изучить основы разработки тестовых пакетов.

Тестовые пакеты — это совокупность тестовых случаев, которые проверяют определенную функциональность или компонент программного обеспечения. Они помогают обеспечить, что ваше ПО работает корректно и соответствует требованиям.

Практическая работа по разработке тестовых пакетов предполагает создание комплекса тестов, которые проверяют различные аспекты программного обеспечения. Шаги, которые помогут вам разработать тестовые пакеты:

1. Определение области тестирования. Определите, какие функции или модули программы будут тестироваться.
2. Разработка тестовых случаев. Создайте список тестовых случаев, которые покрывают все возможные сценарии использования функций или модулей.
3. Подготовка тестовых данных. Соберите или сгенерируйте данные, которые будут использоваться в тестах.
4. Настройка тестового окружения. Убедитесь, что у вас есть все необходимое для выполнения тестов (например, доступ к серверам, настроенные инструменты).
5. Выполнение тестов. Запустите тесты и зафиксируйте результаты.
6. Анализ результатов. Оцените результаты тестов и определите, соответствуют ли они ожиданиям.
7. Документирование. Запишите процесс тестирования и результаты в отчеты.
8. Улучшение и исправление. Если были найдены ошибки, разработайте план их исправления.

Вот пример структуры документа для тестового пакета:

```markdown

| Тестовый пакет | для функции Авторизации |
|---|---|
| Область тестирования | - Функция входа в систему<br>- Восстановление пароля<br>- Регистрация нового пользователя |
| Тестовые случаи | |
| Тестовый случай 1 | Успешный вход в систему |
| Предварительные условия | Пользователь зарегистрирован в системе |
| Тестовые данные | Корректные логин и пароль |
| Шаги | 1. Открыть страницу входа.<br>2. Ввести корректные логин и пароль. |
```

	3. Нажать кнопку "Войти".
Ожидаемый результат	Пользователь перенаправлен на главную страницу
Фактический результат	
Статус	
Тестовый случай 2	Вход с неверным паролем
Предварительные условия	Пользователь зарегистрирован в системе
Тестовые данные	Корректный логин и некорректный пароль
Шаги	1. Открыть страницу входа. 2. Ввести корректный логин и некорректный пароль. 3. Нажать кнопку "Войти".
Ожидаемый результат	Система отображает сообщение об ошибке
Фактический результат	
Статус	
Анализ результатов	
Общие выводы:	Расписать: Обнаруженные проблемы Рекомендации по улучшению
Документирование	Отчеты о тестировании Логи ошибок
Улучшение и исправление	1. План исправления ошибок 2. Меры по предотвращению подобных ошибок в будущем

Вот более подробное описание каждого шага:

1. Определение области тестирования: Начните с определения целей тестирования. Это может быть новая функция, регрессионное тестирование или проверка производительности. Четкое понимание целей поможет вам сфокусироваться на ключевых аспектах.

2. Разработка тестовых случаев: Тестовые случаи должны быть разработаны таким образом, чтобы они покрывали все возможные пути выполнения кода, включая граничные условия и нестандартные ситуации. Каждый тестовый случай должен содержать информацию о предварительных условиях, шагах для выполнения, ожидаемых и фактических результатах, а также статусе теста.

3. Подготовка тестовых данных: Тестовые данные должны быть репрезентативными и включать различные варианты, которые пользователь может встретить в реальной жизни. Это может включать в себя нормальные

данные, граничные значения и некорректные данные для проверки обработки ошибок.

4. Настройка тестового окружения: Тестовое окружение должно максимально соответствовать продуктивному окружению, чтобы результаты тестирования были релевантными. Это включает в себя настройку серверов, баз данных, сетевых конфигураций и других компонентов.

5. Выполнение тестов: Тесты должны выполняться в соответствии с разработанными тестовыми случаями. Важно фиксировать все результаты, включая успешные и неуспешные попытки.

6. Анализ результатов: После выполнения тестов необходимо проанализировать результаты и определить, соответствуют ли они ожидаемому поведению. Любые отклонения должны быть зарегистрированы и исследованы.

7. Документирование: Все аспекты тестирования, включая планы, тестовые случаи, результаты и анализ, должны быть тщательно задокументированы. Это обеспечивает прозрачность и позволяет другим членам команды понять проведенное тестирование.

8. Улучшение и исправление: На основе результатов тестирования разработайте план улучшений и исправлений. Это может включать в себя исправление ошибок, оптимизацию производительности или улучшение пользовательского интерфейса.

Эти шаги помогут вам систематически подходить к тестированию и обеспечат высокое качество вашего программного обеспечения.

Задание: На основе примера разработать тестовый пакет по выбранной предметной области.

ПРАКТИЧЕСКАЯ РАБОТА № 3

Тема: Использование инструментария анализа качества

Цель работы: получение навыков использования инструментариев анализа качества.

Инструменты анализа качества в Visual Studio помогают разработчикам обнаруживать и исправлять ошибки, а также улучшать качество кода. Вот несколько ключевых инструментов и функций, которые могут быть использованы в Visual Studio для анализа качества информационной системы:

Статический анализ кода: Visual Studio предлагает встроенные средства статического анализа, такие как Code Analysis, которые могут автоматически обнаруживать потенциальные проблемы в коде, такие как утечки памяти или неправильное использование API.

Профилировщик: Инструменты профилирования позволяют разработчикам анализировать производительность приложения, выявлять узкие места и оптимизировать использование ресурсов.

Unit-тестирование: Visual Studio поддерживает создание и выполнение unit-тестов, что является важной частью обеспечения качества кода и предотвращения регрессий в функциональности.

Code Metrics: Этот инструмент позволяет оценить сложность кода и его поддерживаемость, предоставляя метрики, такие как Cyclomatic Complexity и Depth of Inheritance.

Live Unit Testing: Автоматически запускает unit-тесты в фоновом режиме при изменении кода, что помогает быстро обнаруживать ошибки.

IntelliTrace: Позволяет записывать и воспроизводить историю выполнения приложения, что упрощает отладку и анализ проблем.

Эти инструменты могут быть интегрированы в процесс разработки для непрерывного улучшения качества кода и эффективности работы информационной системы.

Пример статического анализа кода в Visual Studio может быть продемонстрирован на простом фрагменте C# кода. Допустим, у нас есть следующий код:

```
public class Calculator
{
   public int Add(int a, int b)
   {
      return a + b;
   }


   public int Divide(int a, int b)

   {

      if (b == 0)

         throw new DivideByZeroException();

      return a / b;
   }
}
```

При использовании статического анализа кода, Visual Studio может выявить потенциальные проблемы, такие как:

- Неправильное использование исключений: В методе Divide генерируется исключение DivideByZeroException, но его можно было бы обработать более грациозно.
- Качество кода: Методы могут быть слишком простыми или очевидными, что указывает на возможность их дальнейшего улучшения или оптимизации.

Visual Studio предложит рекомендации по улучшению кода, такие как использование более сложной логики обработки ошибок или рефакторинг кода для повышения его читаемости и поддерживаемости. Статический анализ также может помочь выявить проблемы с производительностью и безопасностью, предлагая изменения для оптимизации кода.

Code metrics — это важный инструмент для оценки качества кода, который помогает разработчикам понимать сложность, поддерживаемость и тестирование своего кода. В Visual Studio, вы можете использовать встроенные инструменты для анализа метрик кода, которые предоставляют ценную информацию о вашем проекте.

Вот пример того, как могут выглядеть метрики кода для класса Calculator:

```
Тип: Calculator
- Maintainability Index (Индекс поддерживаемости): 85
- Cyclomatic Complexity (Цикломатическая сложность): 2
- Depth of Inheritance (Глубина наследования): 1
- Class Coupling (Связанность классов): 3
- Lines of Code (Строк кода): 12
```

Индекс поддерживаемости – это мера того, насколько легко поддерживать код; более высокие значения лучше. Цикломатическая сложность измеряет сложность управления потоком кода; более низкие значения предпочтительнее. Глубина наследования показывает уровень наследования класса от базовых классов. Связанность классов отражает количество классов, от которых зависит данный класс. Строки кода указывают на общее количество строк кода в классе.

Эти метрики помогают определить потенциальные риски и области для улучшения в коде.

Конечно, давайте рассмотрим пошаговый процесс анализа метрик кода в Visual Studio на примере класса Calculator:

1. Открытие проекта в Visual Studio: Запустите Visual Studio и откройте проект, который содержит класс Calculator.

2. Анализ метрик кода:

– Перейдите в меню Анализ (Analyze).

– Выберите Рассчитать метрики кода (Calculate Code Metrics).

– Для всего решения или выбранного проекта.

3. Просмотр результатов:

– После завершения анализа откроется окно Метрики кода (Code Metrics Results).

– Вы увидите список метрик для каждого класса и метода в вашем проекте.

4. Интерпретация метрик:

– Индекс поддерживаемости (Maintainability Index): Чем выше число (от 0 до 100), тем легче поддерживать код.

– Цикломатическая сложность (Cyclomatic Complexity): Определяет количество путей через код; идеально стремиться к более низким значениям.

– Глубина наследования (Depth of Inheritance): Показывает, сколько уровней наследования имеет класс.

– Связанность классов (Class Coupling): Отражает, сколько других классов используется данным классом.

– Строки кода (Lines of Code): Общее количество строк кода в классе.

5. Принятие решений:

– Используйте полученные данные для определения областей, требующих рефакторинга или улучшения.

– Рассмотрите возможность уменьшения сложности и улучшения читаемости кода.

6. Действия по улучшению кода:

– Примените рекомендации Visual Studio для оптимизации кода.

– Улучшите обработку ошибок, уменьшите связанность классов и упростите логику, если это необходимо.

Это базовые шаги для работы с метриками кода в Visual Studio.

ПРАКТИЧЕСКАЯ РАБОТА № 4.

Тема: Анализ и обеспечение обработки исключительных ситуаций.

Цель работы: 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.

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 генерации 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 раз- личных исключений, таких как FormatException, IndexOutOfRange Exception, FileNotFound Exception, ArgumentOut OfRangeException 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.

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

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.

Перехват исключений.

Принимая во внимание, что .NET Framework включает больое 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 столк- нуться 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 вводечисла в
консоль
catch (FormatException)
{
Console.WriteLine(«Это НЕ число!!!\n»);Main();
}
Console.ReadLine();
}

}
```

```
file:///C:/projects/test/ConsoleApplication1/ConsoleApplication1/bin/Debug/ConsoleApplication1....
Введите x: professorWeb :)
Это НЕ число!!!

Введите x: 10
Введите y: 0
Деление на 0 detected!!!

Введите x: 10
Введите y: 2
Результат: 5
```

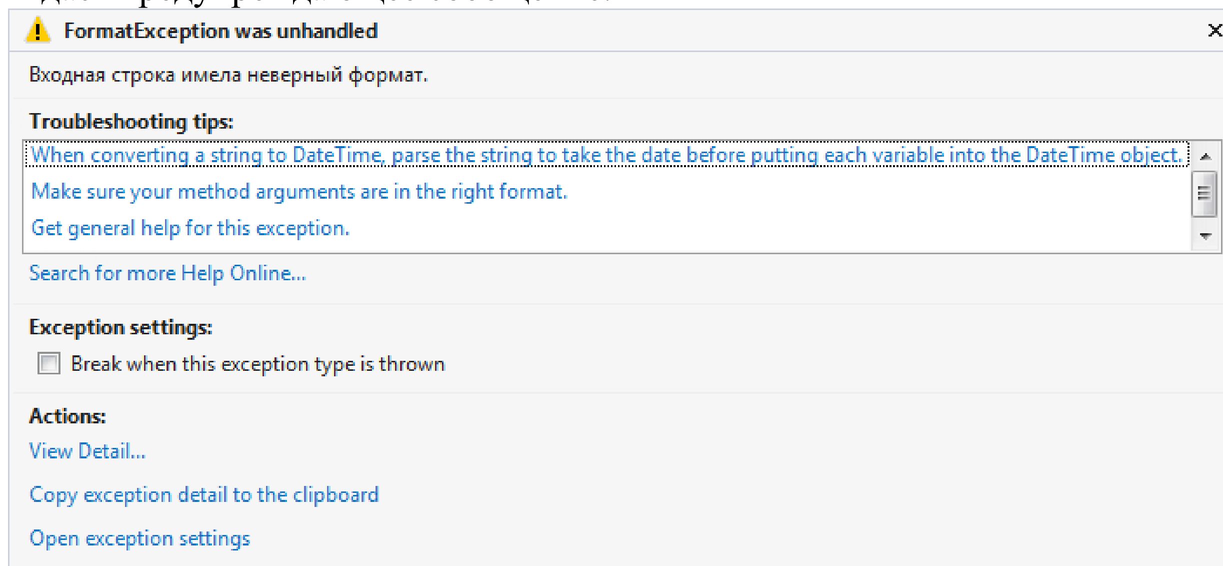
Данный простой пример наглядно иллюстрирует обработку исключительной ситуации при делении на 0

(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?

ПРАКТИЧЕСКАЯ РАБОТА № 5.
Тема: Функциональное тестирование.

Цель работы: Изучение информационной технологии создания функционального тестированиея.

Закодированные тесты пользовательского интерфейса (CUITs) управляют вашим приложением через его пользовательский интерфейс. Эти тесты включают функциональное тестирование элементов управления пользовательского интерфейса. Они позволяют вам убедиться, что все приложение, включая его пользовательский интерфейс, функционирует правильно. Закодированные тесты пользовательского интерфейса полезны там, где в пользовательском интерфейсе есть проверка или другая логика, например, на веб-странице. Они также часто используются для автоматизации существующего ручного тестирования.

Создать тест с кодированием пользовательского интерфейса в Visual Studio несложно. Вы просто выполняете тест вручную, пока конструктор тестов с кодированием пользовательского интерфейса работает в фоновом режиме. Вы также можете указать, какие значения должны отображаться в определенных полях. Конструктор закодированных тестов пользовательского интерфейса записывает ваши действия и генерирует на их основе код. После создания теста вы можете отредактировать его в специализированном редакторе, который позволяет изменять последовательность действий.

Специализированный конструктор тестов с кодированием пользовательского интерфейса и редактор упрощают создание и редактирование тестов с кодированием пользовательского интерфейса, даже если ваши основные навыки сосредоточены на тестировании, а не на программировании. Но если вы разработчик и хотите расширить тест более продвинутым способом, код структурирован таким образом, чтобы его было легко скопировать и адаптировать. Например, вы можете записать тест для покупки чего-либо на веб-сайте, а затем отредактировать сгенерированный код, чтобы добавить цикл, в котором покупается много товаров.

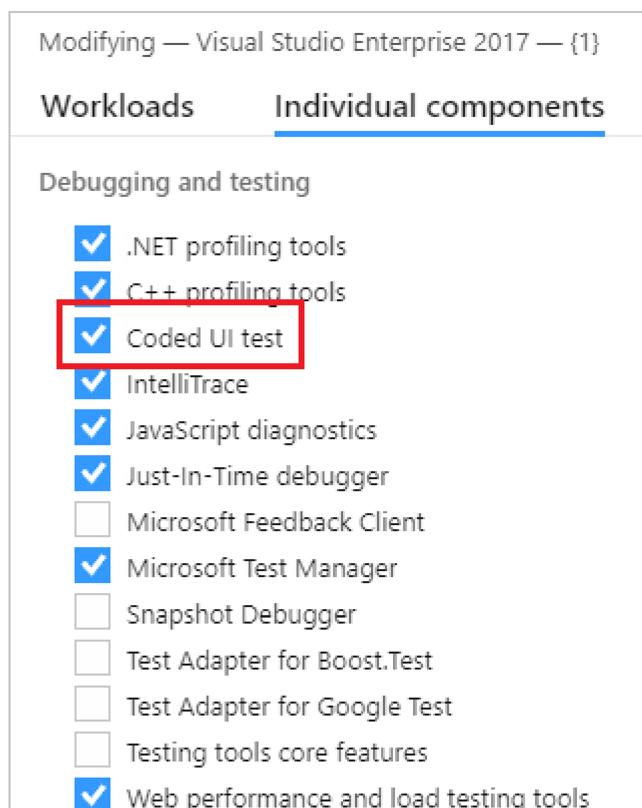
Закодированный тест пользовательского интерфейса для автоматизированного функционального тестирования на основе пользовательского интерфейса устарел. Visual Studio 2019 - последняя версия, в которой будет полностью доступен закодированный тест пользовательского интерфейса. Рекомендуемо использовать Playwright для тестирования веб-приложений и Appium с WinAppDriver для тестирования приложений для настольных компьютеров и UWP. Рассмотрите Xamarin.UITest для тестирования приложений iOS и Android с использованием тестовой платформы NUnit. Чтобы уменьшить воздействие на пользователей, некоторая минимальная поддержка по-прежнему будет доступна в Visual Studio 2022 Preview 4 или более поздней версии.

Установите компонент Coded UI test.

Чтобы получить доступ к инструментам и шаблонам тестирования с кодированием пользовательского интерфейса, установите компонент Visual Studio Coded UI test.

1. Запустите установщик Visual Studio, выбрав Инструменты > Получить инструменты и функции.

2. В установщике Visual Studio выберите вкладку Отдельные компоненты, а затем прокрутите вниз до раздела Отладка и тестирование. Выберите компонент Coded UI test.

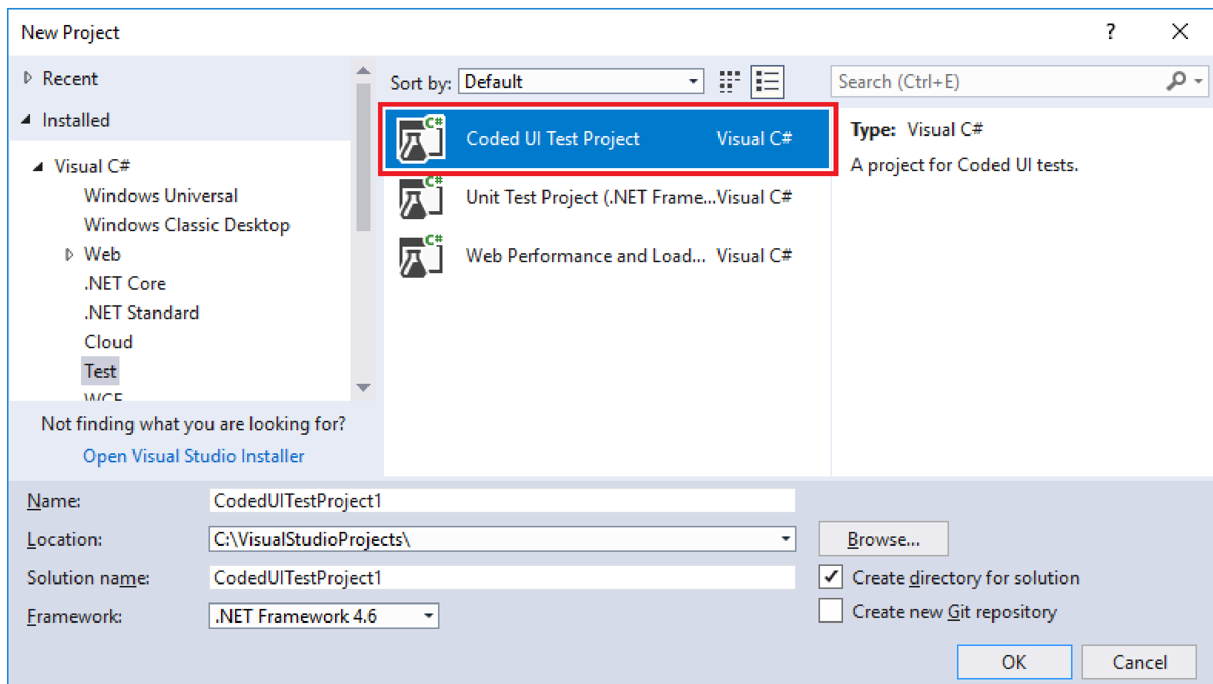


3. Выберите Изменить.

Создайте закодированный тест пользовательского интерфейса

1. Создайте проект тестирования закодированного пользовательского интерфейса.

Закодированные тесты пользовательского интерфейса должны содержаться в проекте кодированного тестирования пользовательского интерфейса. Если у вас еще нет проекта кодированного тестирования пользовательского интерфейса, создайте его. Выберите Файл > Создать > Проект. Найдите и выберите шаблон проекта Coded UI Test Project.



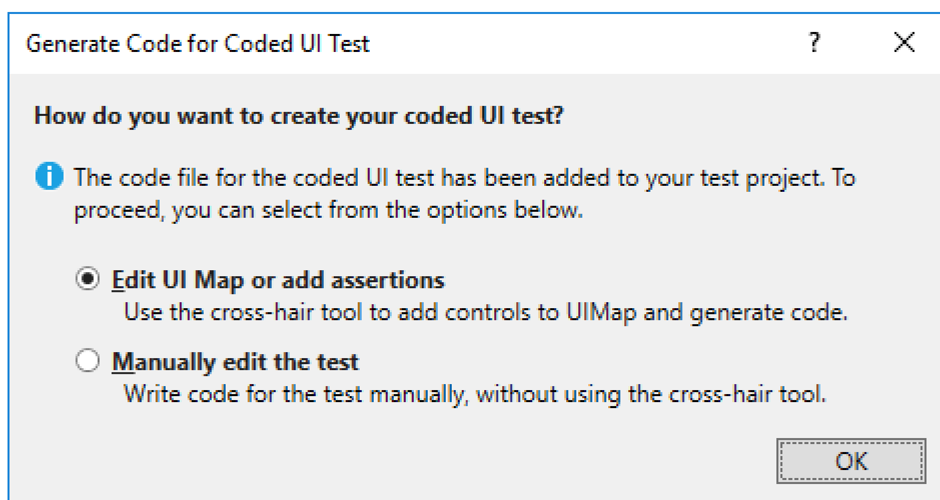
Примечание

Если вы не видите шаблон проекта Coded UI Test Project, вам необходимо установить компонент Coded UI test.

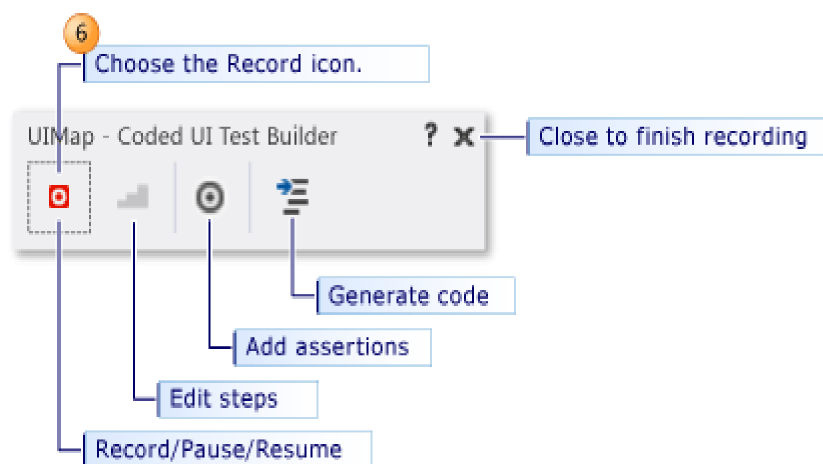
Добавьте файл закодированного теста пользовательского интерфейса.

Если вы только что создали проект с кодированным пользовательским интерфейсом, первый файл CUIT добавляется автоматически. Чтобы добавить другой тестовый файл, откройте контекстное меню проекта Coded UI test в Обозревателе решений, а затем выберите Добавить > Закодированный тест пользовательского интерфейса.

В диалоговом окне "Сгенерировать код для закодированного теста пользовательского интерфейса" выберите "Записать действия" > "Отредактировать карту пользовательского интерфейса" или "Добавить утверждения".



Появится конструктор тестов закодированного пользовательского интерфейса.



2. Запишите последовательность действий.

Чтобы начать запись, выберите значок Записи. Выполните действия, которые вы хотите протестировать в своем приложении, включая запуск приложения, если это требуется. Например, при тестировании веб-приложения можно запустить браузер, перейти на веб-сайт и выполнить вход в приложение.

Чтобы приостановить запись, например, если вам нужно разобраться с входящей почтой, выберите Приостановить.

Предупреждение.

Все действия, выполняемые на рабочем столе, будут записываться. Приостановите запись, если вы выполняете действия, которые могут привести к включению в запись конфиденциальных данных.

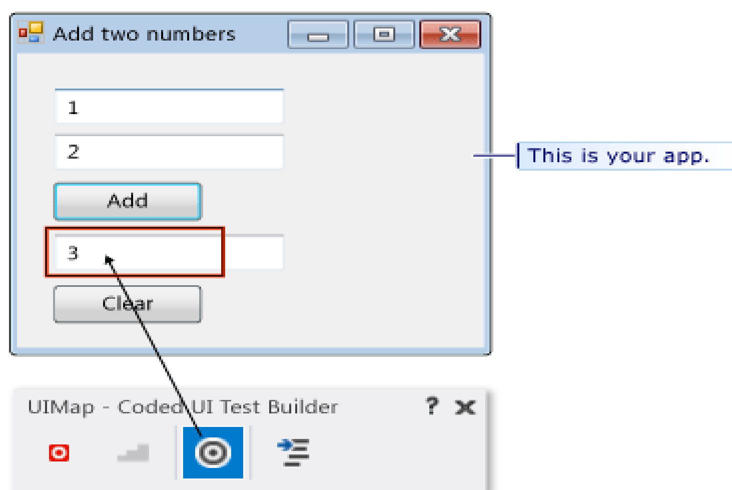
Чтобы удалить действия, которые вы записали по ошибке, выберите Редактировать шаги.

Чтобы сгенерировать код, который будет повторять ваши действия, выберите значок Сгенерировать код и введите название и описание вашего метода кодированного тестирования пользовательского интерфейса.

3. Проверьте значения в полях пользовательского интерфейса, таких как текстовые поля.

Выберите Добавить утверждения в конструкторе тестов с кодированным пользовательским интерфейсом, а затем выберите элемент управления пользовательского интерфейса в запущенном приложении. В появившемся списке свойств выберите свойство, например, Текст в текстовом поле. В контекстном меню выберите Добавить утверждение. В диалоговом окне выберите оператор сравнения, значение сравнения и сообщение об ошибке.

Закройте окно утверждения и выберите Сгенерировать код.



Чередуйте действия по записи и проверке значений. Генерируйте код в конце каждой последовательности действий или проверок. При желании вы сможете вставить новые действия и проверки позже.

4. Просмотрите сгенерированный тестовый код.

Чтобы просмотреть сгенерированный код, закройте окно конструктора тестов пользовательского интерфейса. В коде вы можете увидеть названия, которые вы дали каждому шагу. Код находится в созданном вами файле CUIT:

C#Копировать

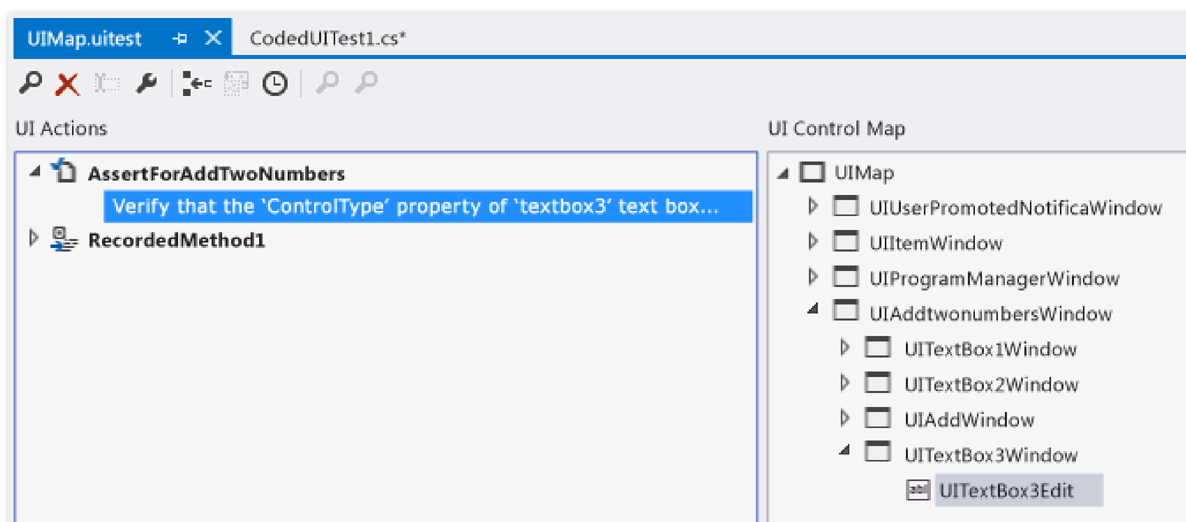
```
[CodedUITest]
public class CodedUITest1
{ ...
 [TestMethod]
 public void CodedUITestMethod1()
 {
    this.UIMap.AddTwoNumbers();
    this.UIMap.VerifyResultValue();
    // To generate more code for this test, select
    // "Generate Code" from the shortcut menu.
 }
}
```

5. Добавьте дополнительные действия и утверждения.

Наведите курсор на соответствующую точку в методе тестирования, а затем в контекстном меню выберите Сгенерировать код для закодированного теста пользовательского интерфейса. В этот момент будет вставлен новый код.

6. Отредактируйте подробные сведения о тестовых действиях и утверждениях.

Откройте UIMap.uitest. Этот файл открывается в редакторе тестов с кодированием пользовательского интерфейса, где вы можете редактировать любую последовательность действий, которые вы записали, а также редактировать свои утверждения.



7. Запустите тест.

Воспользуйтесь проводником тестов или откройте контекстное меню в методе тестирования, а затем выберите Запустить тесты.

Задание: На основе приведенного примера провести функциональное тестирование.

ПРАКТИЧЕСКАЯ РАБОТА № 6.
Тема: Тестирование безопасности

Цель работы: Приобрести и закрепить практические навыки по тестированию безопасности.

Тестирование безопасности — это ключевой аспект разработки программного обеспечения, который помогает обнаруживать и устранять уязвимости в коде, чтобы предотвратить потенциальные атаки. Visual Studio предлагает различные инструменты и функции для улучшения безопасности кода.

Вот основные шаги для тестирования безопасности в Visual Studio:

1. **Использование статического анализа кода**:

- Включите статический анализ кода, который может обнаруживать уязвимости на ранних этапах разработки.
- Используйте встроенные правила анализа или настройте свои собственные для проверки специфических требований безопасности.

2. **Интеграция с инструментами безопасности**:

- Интегрируйте Visual Studio с внешними инструментами безопасности, такими как **Microsoft Security Code Analysis** или **SonarQube**, для расширенного анализа.

3. **Регулярное обновление зависимостей**:

– Используйте **NuGet Package Manager** для управления зависимостями и убедитесь, что все библиотеки и фреймворки обновлены до последних безопасных версий.

4. **Проверка на уязвимости**:

– Проведите ручное тестирование кода на наличие уязвимостей, используя **различные техники и инструменты**, такие как **fuzz testing** или **penetration testing**.

5. **Обучение и осведомленность**:

– Обучите команду разработчиков лучшим практикам безопасности и постоянно повышайте уровень знаний в области безопасности кода.

6. **Код-ревью**:

– Регулярно проводите код-ревью с участием других разработчиков для выявления потенциальных проблем безопасности.

7. **Использование атрибутов безопасности**:

– Применяйте атрибуты безопасности в коде, такие как [PrincipalPermission], для управления доступом и разграничения прав.

8. **Соблюдение стандартов безопасности**:

– Следуйте стандартам и рекомендациям безопасности, таким как **OWASP Top 10**, для написания более безопасного кода.

Эти шаги помогут вам улучшить безопасность вашего кода в Visual Studio.

Пример пошагового процесса тестирования безопасности на основе простого веб-приложения:

1. **Планирование**:

– Определите цели тестирования и уязвимости, которые необходимо проверить.

– Создайте план тестирования, включающий различные типы тестов (например, статический анализ, динамический анализ, пенетрационное тестирование).

2. **Статический анализ кода**:

– Используйте инструменты статического анализа, такие как **Microsoft Security Code Analysis**, для автоматического обнаружения уязвимостей.

– Пример команды для запуска анализа в Visual Studio:

```
– MSBuild.exe /t:Rebuild /p:Configuration=Release /p:RunCodeAnalysis=true
```

3. **Динамический анализ**:

– Запустите инструменты динамического анализа, такие как **OWASP ZAP** или **Burp Suite**, для тестирования веб-приложения в реальном времени.

– Проведите сканирование на наличие уязвимостей, таких как XSS, SQL инъекции, CSRF и другие.

4. **Пенетрационное тестирование**:

– Выполните ручное пенетрационное тестирование, чтобы проверить, как атакующий может эксплуатировать уязвимости.

– Используйте тестовые сценарии для имитации атак и проверки защиты приложения.

5. **Ревизия кода**:

– Проведите ручную проверку кода на предмет уязвимостей, которые могли быть пропущены автоматическими инструментами.

– Обратите внимание на обработку ошибок, проверку входных данных и безопасность аутентификации.

6. **Исправление уязвимостей**:

– Анализируйте результаты тестов и определите, какие уязвимости необходимо устранить.

– Разработайте и внедрите исправления для обнаруженных уязвимостей.

7. **Повторное тестирование**:

– После внесения изменений проведите повторное тестирование, чтобы убедиться, что уязвимости устранены.

– Убедитесь, что исправления не внесли новых уязвимостей в систему.

8. **Документирование**:

– Запишите все найденные уязвимости, проведенные тесты и исправления.

– Создайте отчет о тестировании безопасности для дальнейшего анализа и улучшения процессов.

Этот пример демонстрирует базовый процесс тестирования безопасности. Важно регулярно проводить тестирование и обновлять практики безопасности в соответствии с новыми угрозами и тенденциями в области безопасности.

ПРАКТИЧЕСКАЯ РАБОТА № 7.

Тема: Нагрузочное тестирование, стрессовое тестирование

.

Цель работы: 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 иэлектронном (файл Word) видах.

Подготовка проекта и запись тестов

В качестве основного инструмента для нагрузочного тестиро- вания 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**.

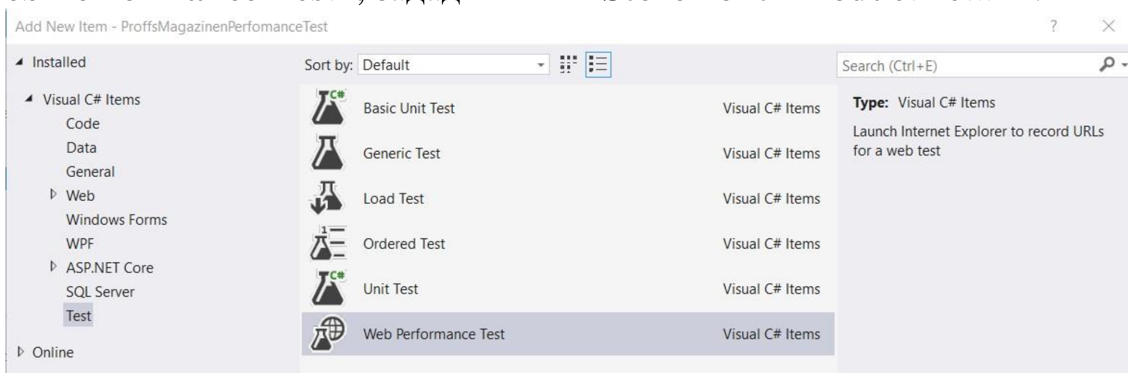
После создания проекта нам необходимо будет создать тесты для каждого из ранее определенных пользовательский действий. Ограничимся созданием теста для одного пользовательского действия в качестве примера, поскольку остальные действия создаютсяпо аналогии.

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**».

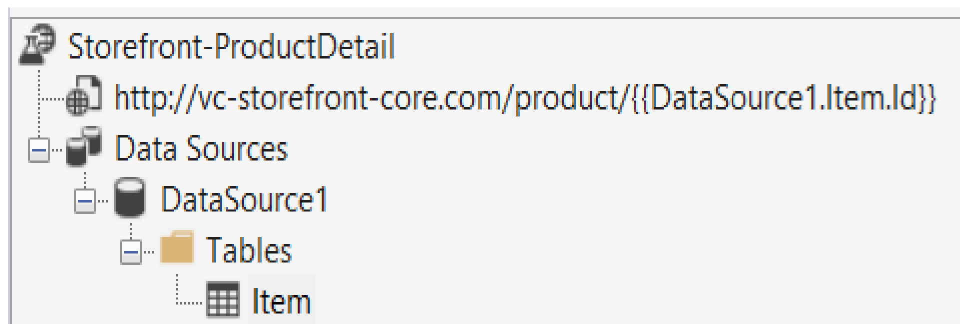


Для данного типа теста Visual Studio сразу же попытается открыть браузер, где можно будет в интерактивном режиме прокли- кать необходимые действия непосредственно на сайте, но мы этого делать не будем, но сразу закроем браузер и остановим запись. В итоге мы получим пустой тест **Storefront-ProductDetail.webtest**.

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 VS **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.



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

Storefront-ProductDetail [16:13]

Error Click here to run again Internet Explorer 9.0 LAN Edit run settings

Request	Status	Total Time	Request Time	Request Bytes	Response Bytes
Run 1					
https://demo.virtocommerce.com/product/1137d81055dc47f3	200 OK	1,776 sec	1,776 sec	0	15 602
Run 2					
https://demo.virtocommerce.com/product/c54dd7f0e74a456c	200 OK	1,496 sec	1,496 sec	0	15 597
Run 3					
https://demo.virtocommerce.com/product/fb46499c4f7e4c78ε	200 OK	1,502 sec	1,502 sec	0	15 575
Run 4					
https://demo.virtocommerce.com/product/ad5ca2cc488348a8	200 OK	2,224 sec	2,224 sec	0	15 569
Run 5					
https://demo.virtocommerce.com/product/ccff63918246463dt	200 OK	1,494 sec	1,494 sec	0	15 570
Run 6					
https://demo.virtocommerce.com/product/dae730451bc745bf	200 OK	1,503 sec	1,503 sec	0	15 587
Run 7					
https://demo.virtocommerce.com/product/7a92cfece20c4c6dε	200 OK	1,515 sec	1,515 sec	0	15 605

Storefront-AddProductToCart.webtest
Storefront-BrowsingCategory.webtest
Storefront-Mixed-recorded.webtest
Storefront-ProductDetail.webtest
Storefront-SearchByBrand.webtest
Storefront-SearchByPhrase.webtest

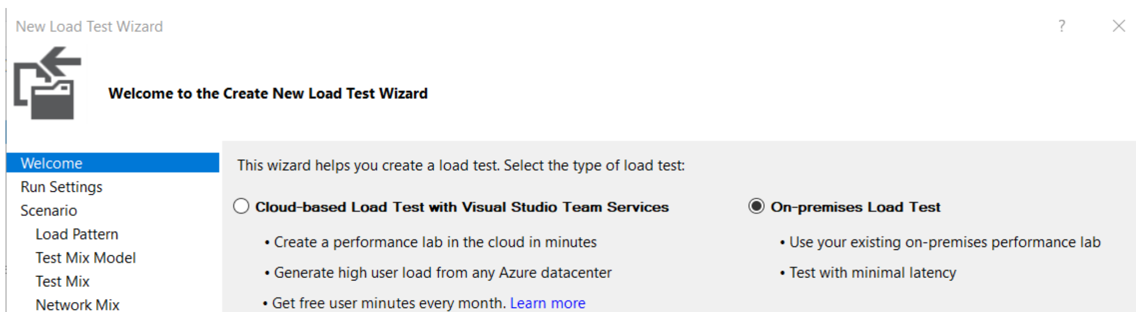
По аналогии создадим тесты для все остальных наших сценариев

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.**

New Item... Ctrl+Shift+A
Existing Item... Shift+Alt+A
New Folder
REST API Client...
Reference...
Web Reference...
Service Reference...
Connected Service
Analyzer...
Unit Test...
Load Test...
Web Performance Test...
Ordered Test

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

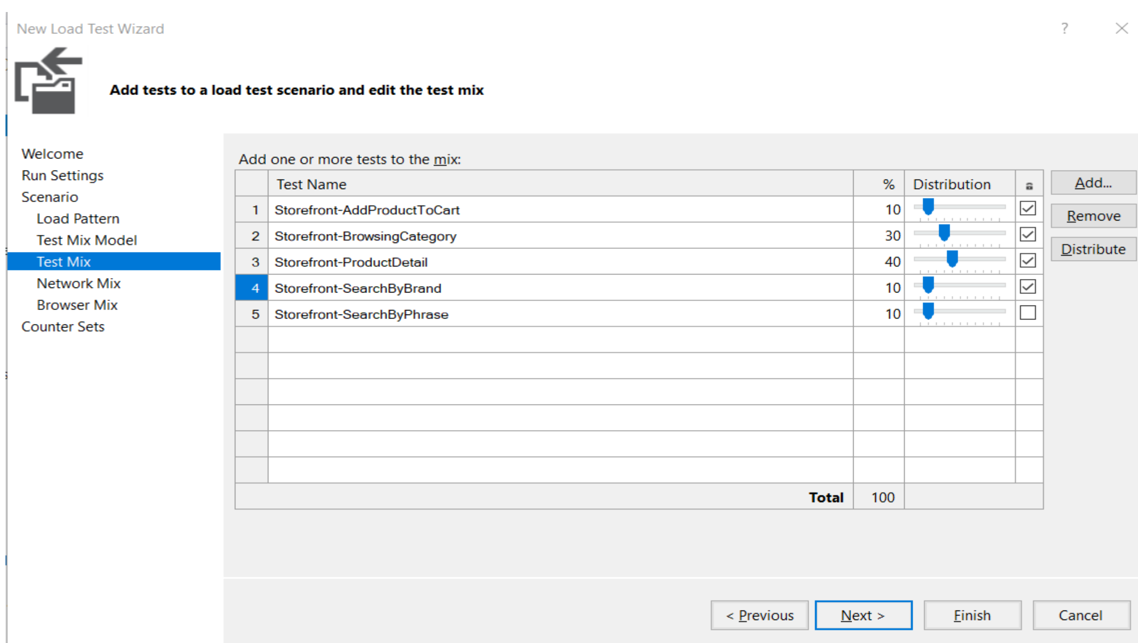


Этот пункт требует определѐнного разъяснения, ведь вы справедливо спросите: «Причем тут on-premise?» Тема статьи о тестиро вании с помощью Teams Services и MS Azure, но тут есть нюанс, так как мы для тестов используем источники данных в виде таблиц или других внешних сервисов, то с этим могут возникнуть 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 записитакого 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.



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.

Storefront-Mixed-recorded [8:52]

Error Click here to run again Internet Explorer 9.0 LAN Edit run settings

Request	Status	Total Time	Request Time	Request Bytes	Response Bytes
http://localhost:8888/product/1020	200 Fiddler Generated	1,029 sec	1,029 sec	0	816
http://localhost:8888/product/10201	200 Fiddler Generated	0,001 sec	0,001 sec	0	793
http://localhost:8888/verktoy/stasjonare-verktoy/benkslipemaskin	200 Fiddler Generated	0,001 sec	0,001 sec	0	852
http://localhost:8888/product/10202	200 Fiddler Generated	0,000 sec	0,000 sec	0	793
http://localhost:8888/byggtilbehor/batteri	200 Fiddler Generated	0,000 sec	0,000 sec	0	836
http://localhost:8888/storefrontapi/cart/items	200 Fiddler Generated	0,366 sec	0,366 sec	27	950
http://localhost:8888/product/10204	200 Fiddler Generated	0,000 sec	0,000 sec	0	793
http://localhost:8888/storefrontapi/cart/items	200 Fiddler Generated	0,000 sec	0,000 sec	27	950
http://localhost:8888/byggtilbehor/batteri	200 Fiddler Generated	0,000 sec	0,000 sec	0	833
http://localhost:8888/byggtilbehor/industristovsuger	200 Fiddler Generated	0,000 sec	0,000 sec	0	810
http://localhost:8888/byggtilbehor/batteri	200 Fiddler Generated	0,000 sec	0,000 sec	0	829
http://localhost:8888/byggtilbehor/industristovsuger/stovsugerposer	200 Fiddler Generated	0,000 sec	0,000 sec	0	825
http://localhost:8888/storefrontapi/cart/items	200 Fiddler Generated	0,000 sec	0,000 sec	27	950
http://localhost:8888/dewalt	200 Fiddler Generated	0,000 sec	0,000 sec	0	786
http://localhost:8888/Electronics/en-US/matinstrument/avveingsinstrument	200 Fiddler Generated	0,003 sec	0,003 sec	0	830
http://localhost:8888/Electronics/en-US/byggtillbehor	200 Fiddler Generated	0,000 sec	0,000 sec	0	811
http://localhost:8888/product/10205	200 Fiddler Generated	0,000 sec	0,000 sec	0	793

После отработки всех данных получим вот такую картинку в Fiddler.

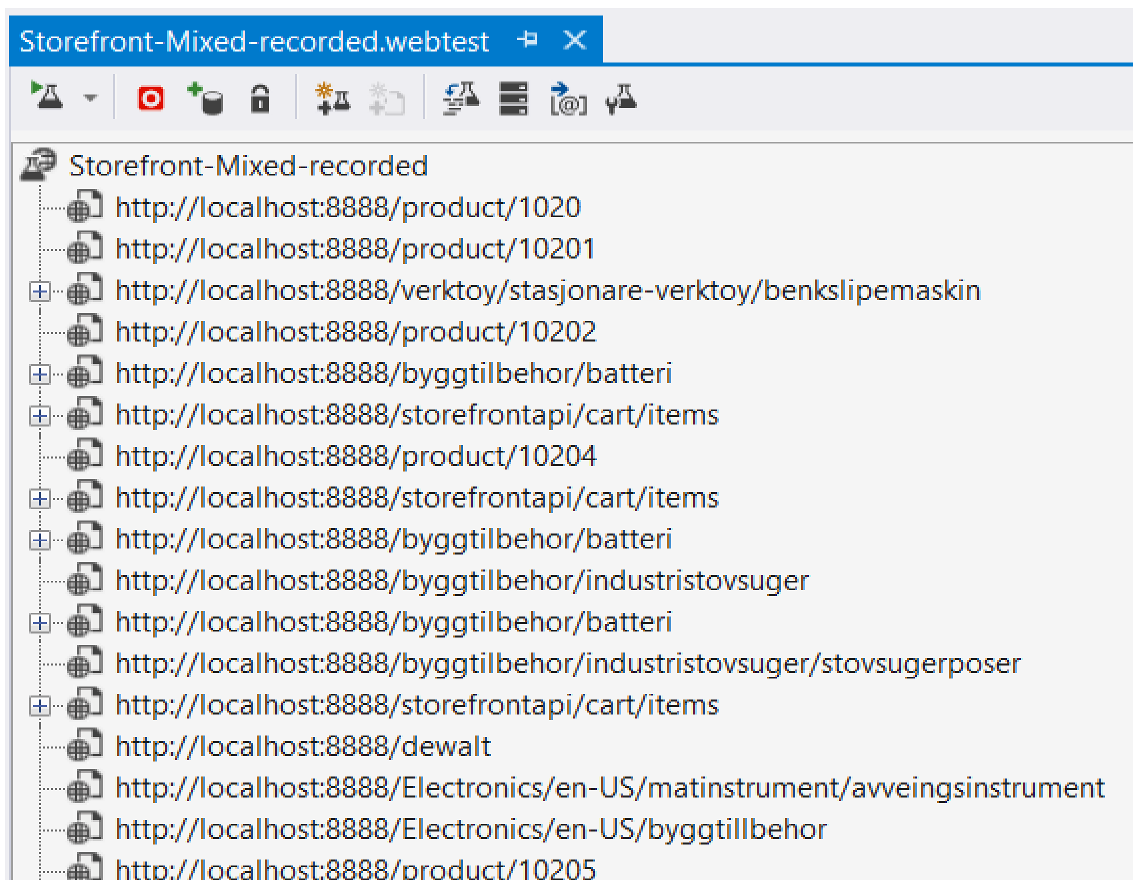
Progress Telerik Fiddler Web Debugger

File Edit Rules Tools View Help GET /book GeoEdge

WinConfig Replay Go Stream Decode Keep: All sessions Any Process Find Save Brows

#	ClientBeginRe...	Reques...	Result	Host	URL	Overall_Ela
3682	08:53:14.073	GET	200	localhost:8888	/search?type=product&q=2608604494	
3683	08:53:14.087	GET	200	localhost:8888	/maleinstrumenter/miljoinstrument/radonmaler?ter...	
3684	08:53:14.096	GET	200	localhost:8888	/search?type=product&q=5+x+400+mm	
3685	08:53:14.105	POST	200	localhost:8888	/storefrontapi/cart/items	0:00:00.00
3686	08:53:14.115	GET	200	localhost:8888	/product/12184	0:00:00.00
3687	08:53:14.124	GET	200	localhost:8888	/product/12185	
3688	08:53:14.133	GET	200	localhost:8888	/product/12186	
3689	08:53:14.141	POST	200	localhost:8888	/storefrontapi/cart/items	
3690	08:53:14.150	GET	200	localhost:8888	/verneutstyr/forsta-hjalpen/refill?terms=Brand:ce...	
3691	08:53:14.159	GET	200	localhost:8888	/product/12187	
3692	08:53:14.168	GET	200	localhost:8888	/product/12189	0:00:00.00
3693	08:53:14.181	GET	200	localhost:8888	/product/1219	0:00:00.00
3694	08:53:14.190	GET	200	localhost:8888	/product/12190	
3695	08:53:14.198	GET	200	localhost:8888	/Electronics/en-US/matinstrument/miljoinstrument/r	

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.

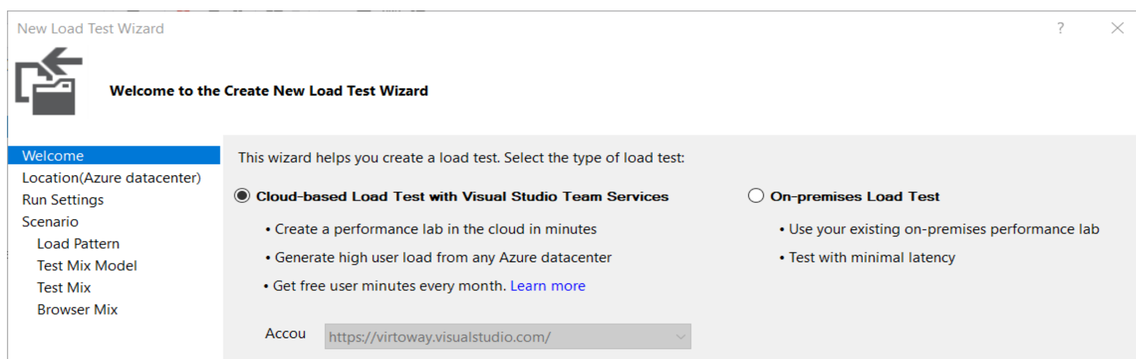


Теперь практически все готово для запуска нашего теста в об- лаке, последним шагом по подготовке теста нужно в любом тексто- вом редакторе открыть «записанный» **MixedLoadTest** и заменить в нем localhost:8888 (адрес прокси, Fiddler-а) на адрес нашего магази- на в облаке.

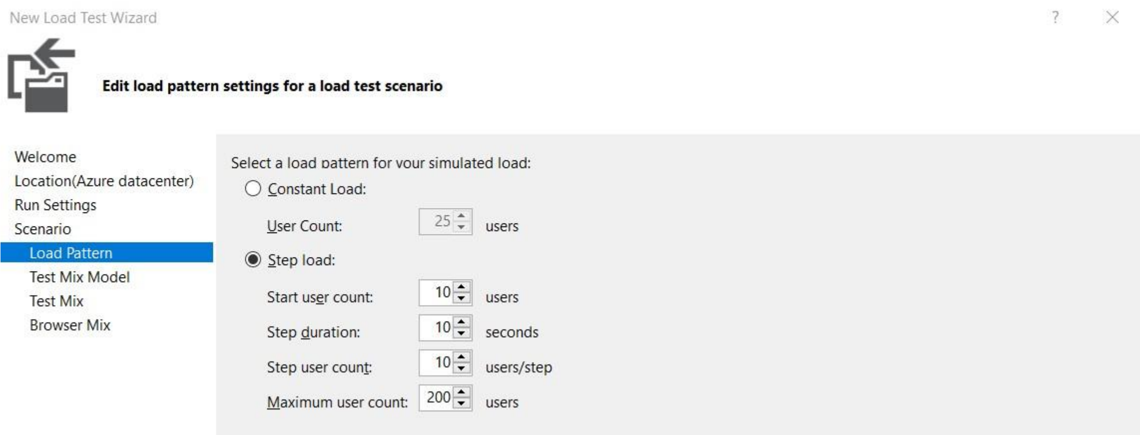
Запуск теста в облаке

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**.

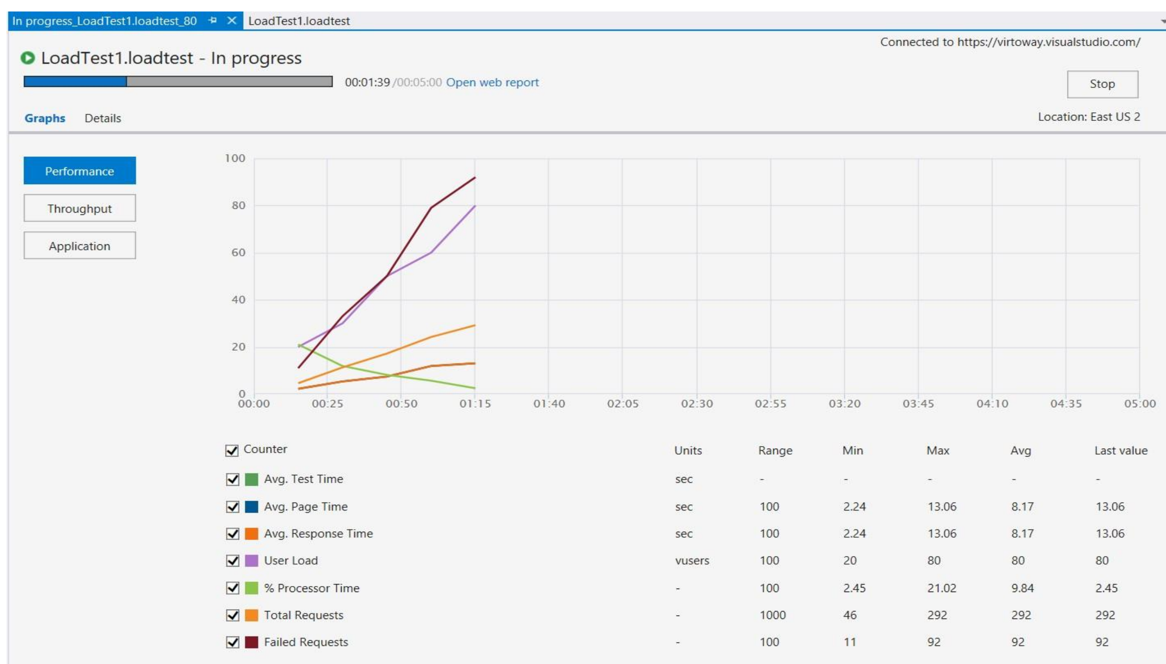


На следующих шагах выбираем дата-центр, с которого будет генерироваться трафик на тестируемый ресурс, а также максималь- ное количество агентов (пользователей) для константного паттерна, либо, если мы хотим использовать постепенное увеличение нагруз- ки, то необходимо задать соответствующие параметры.

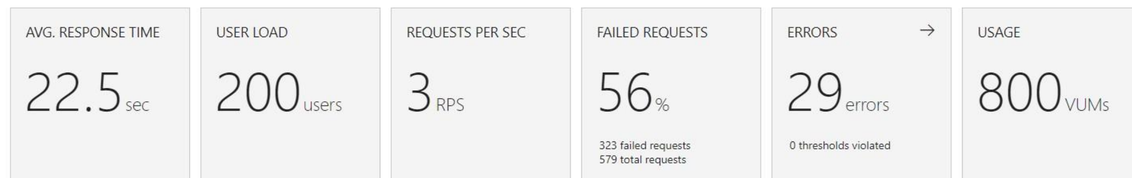


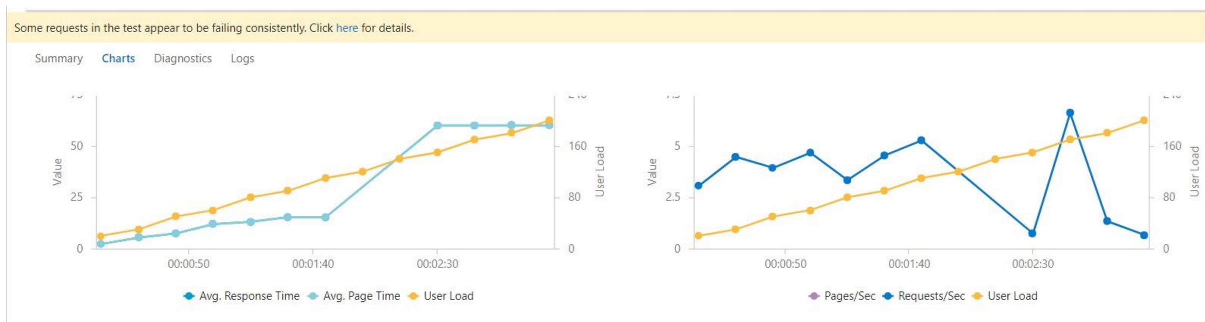
На шаге выбора тестов, выбираем единственный тест, который мы записали ранее с помощью **Fiddler**, он и будет эмулировать «ре- альную» нагрузку на тестируемый ресурс.

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.



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



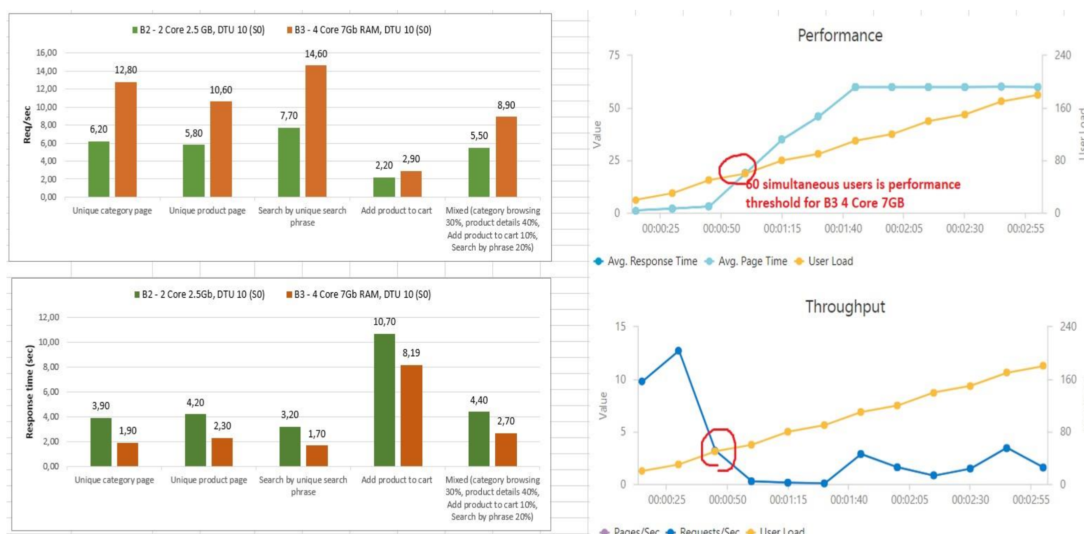


Анализ результатов

Самый важный момент – это обработка и анализ полученных результатов теста. Для рассматриваемой задачи требовалось оценить производительность приложения, работающего на различных конфигурациях 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:



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 определѐнного порогового значения количества запросов.

ПРАКТИЧЕСКАЯ РАБОТА № 8.
Тема: Тестирование интеграции.

Цель работы: проведение тестирования интеграции

Интеграционное тестирование — это процесс, при котором отдельные модули программного обеспечения объединяются и тестируются как группа. Целью такого тестирования является выявление ошибок во взаимодействии между интегрированными модулями. Давайте рассмотрим пример интеграционного тестирования для веб-приложения:

1. **Подготовка тестового окружения**:

– Настройте тестовое окружение, которое имитирует продуктивную среду.

– Убедитесь, что все необходимые сервисы и базы данных работают и доступны.

2. **Создание тестовых случаев**:

– Определите ключевые сценарии использования вашего приложения, которые требуют взаимодействия между модулями.

– Напишите тестовые случаи, которые покрывают эти сценарии.

3. **Интеграция модулей**:

– Поочередно интегрируйте модули, начиная с наиболее низкоуровневых и заканчивая высокоуровневыми.

– После каждой интеграции запускайте тестовые случаи, чтобы проверить корректность взаимодействия.

4. **Выполнение тестов**:

– Запустите тестовые случаи, используя инструменты автоматизации тестирования, если это возможно.

– Внимательно отслеживайте результаты тестов для выявления ошибок в интеграции.

5. **Анализ результатов**:

– Проанализируйте неудачные тесты и определите причину сбоев.

– Проверьте логи и отладочную информацию для диагностики проблем.

6. **Исправление ошибок**:

– Исправьте обнаруженные ошибки в коде.

– После исправлений повторно запустите тесты, чтобы убедиться, что проблемы устранены.

7. **Регрессионное тестирование**:

– Проведите регрессионное тестирование, чтобы убедиться, что новые изменения не повлияли на уже протестированные части системы.

8. **Документирование**:

– Запишите все найденные ошибки и исправления.

– Создайте отчеты о тестировании для будущей оценки и аудита.

Вот пример кода для тестового случая, который проверяет интеграцию между модулем аутентификации и модулем профиля пользователя в веб-приложении:

```
import requests
from assertpy import assert_that

def test_user_profile_integration():
    # Аутентификация пользователя
    auth_response = requests.post('https://yourapp.com/api/auth', data={'username': 'user', 'password': 'pass'})
    assert_that(auth_response.status_code).is_equal_to(200)

    # Получение токена аутентификации
    auth_token = auth_response.json()['token']

    # Запрос профиля пользователя с использованием токена
    profile_response = requests.get('https://yourapp.com/api/profile', headers={'Authorization': f'Bearer {auth_token}'})

    # Проверка успешного получения профиля
    assert_that(profile_response.status_code).is_equal_to(200)
    assert_that(profile_response.json()).contains('username')
```

Этот тестовый случай сначала выполняет аутентификацию пользователя, затем использует полученный токен для запроса данных профиля. Если в процессе интеграции возникнут ошибки, тест выявит их, проверив статус ответа и содержание полученных данных.

ПРАКТИЧЕСКАЯ РАБОТА № 9.
Тема: Конфигурационное тестирование

Цель работы: изучение конфигурационного тестирования ПО. Результатом работы является отчет, в котором должны быть приведены исходные коды программы, конфигурационноготестирования 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 различ- ных конфигурациях компьютеров и т. д.).

В зависимости от типа проекта конфигурационное тестирова- ние может иметь разные цели:

1. Проект по профилированию работы системы

Цель Тестирования: определить оптимальную конфигурацию оборудования, обеспечивающую требуемые характеристики произ водительности 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**. **porta- bility testing**: The process of testing to determine the portability of a software product.

1. Уровни проведения тестирования

Для клиент-серверных приложений конфигурационное тести- рование 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, удовлетворяю- щего требуемым характеристикам качества (эффективность, порта- тивность, удобство сопровождения, надежность).

На следующем (клиентском) уровне, программное обеспече- ние тестируется с позиции его конечного пользователя и конфигу- рации его рабочей станции. На этом этапе будут протестированы следующие характеристики: удобство использования, функцио- нальность. Для этого необходимо будет провести ряд тестов с раз- личными 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 приложенийверсия

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 ра-бочих 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.

ПРАКТИЧЕСКАЯ РАБОТА № 10.
Тема: Тестирование установки

Цель работы: 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.

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**.

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 неѐ (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 коректной 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 пользо- вателя).

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 (InstallationWizard.)
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 также 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 обеспечучения 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 составь-те список после установки ПО, там все верно» – это провокация ина нее лучше не поддаваться;

b. если программа содержит файлы, подписанные сертификатом от MS, то не исключено, что могут попадаться временные фай- лы, 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, врезультате 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 так- же 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 доступом к общим ресурсам на диске, в памяти и/или в системе. Все это должно быть протестировано тщательнейшим образом.

Список литературы

1. Алименков Н. А. Тестирование программного обеспечения и систем / Н. А. Алименков. – М.: Издательство "Финансы и статистика", 2022.
2. Бейзер Б. Тестирование IT-проектов по методу Black Box / Б. Бейзер. – М.: Издательство "ДМК Пресс", 2021.
3. Бек К. Разработка через тестирование / К. Бек. – М.: Издательство "Символ-Плюс", 2023.
4. Куликов С. В. Тестирование программного обеспечения. Базовый курс / С. В. Куликов. – М.: Издательство "Литтест", 2022.
5. Макконнелл С. Совершенный код / С. Макконнелл. – СПб.: Издательство "Питер", 2022.
6. Савин Р. А. Тестирование dot com, или Пособие по жестокому обращению с багами в интернет-стартапах / Р. А. Савин. – М.: Издательский дом "Вильямс", 2021.

МАМХЯГОВ Давлет Фралевич

МДК 05.03 ТЕСТИРОВАНИЕ ИНФОРМАЦИОННЫХ СИСТЕМ

ПРАКТИКУМ
для студентов III курса специальности
09.02.07 Информационные системы и программирование

Корректор Чагова О.Х.
Редактор Чагова О.Х.

Сдано в набор 12.08.2024 г.
Формат 60х84/16
Бумага офсетная
Печать офсетная
Усл. печ. л. 2,55
Заказ № 4937
Тираж 100 экз.

Оригинал-макет подготовлен
в Библиотечно-издательском центре СКГА
369000, г. Черкесск, ул. Ставропольская, 36