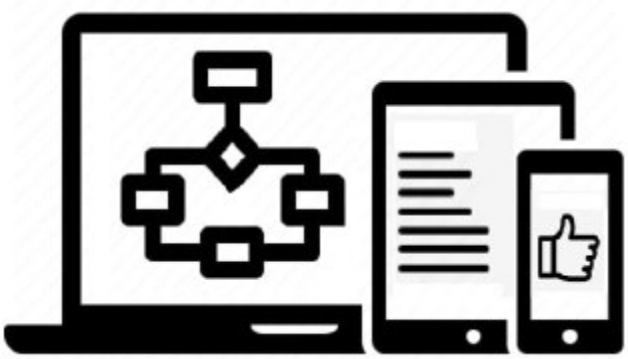
МЕЖГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«БЕЛОРУССКО-РОССИЙСКИЙ УНИВЕРСИТЕТ»

Кафедра «Программное обеспечение информационных технологий»

ТЕСТИРОВАНИЕ И ОТЛАДКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Методические рекомендации к лабораторным работам для студентов направления подготовки 01.03.04 «Прикладная математика» очной формы обучения



Могилев 2025

УДК 621.01
ББК 36.4
Т87

Рекомендовано к изданию
учебно-методическим отделом
Белорусско-Российского университета

Одобрено кафедрой «Программное обеспечение информационных технологий» «11» апреля 2025 г., протокол № 11

Составитель канд. техн. наук, доц. Т. В. Мрочек

Рецензент канд. техн. наук, доц. В. М. Ковальчук

Методические рекомендации содержат описание пяти лабораторных работ по дисциплине «Тестирование и отладка программного обеспечения» для студентов направления подготовки 01.03.04 «Прикладная математика» очной формы обучения. Рассматриваются наиболее известные техники тест-дизайна, основы автоматизации тестирования, составления тестовой документации.

Учебное издание

ТЕСТИРОВАНИЕ И ОТЛАДКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Ответственный за выпуск	В. В. Кутузов
Корректор	А. Т. Червинская
Компьютерная верстка	М. М. Дударева

Подписано в печать 15.08.2025. Формат 60×84/16. Бумага офсетная. Гарнитура Таймс. Печать трафаретная. Усл. печ. л. 2,09. Уч.-изд. л. 2,25. Тираж 21 экз. Заказ № 559.

Издатель и полиграфическое исполнение:
Межгосударственное образовательное учреждение высшего образования
«Белорусско-Российский университет».
Свидетельство о государственной регистрации издателя,
изготовителя, распространителя печатных изданий
№ 1/156 от 07.03.2019.
Пр-т Мира, 43, 212022, г. Могилев.

Содержание

Введение

Цель дисциплины «Тестирование и отладка программного обеспечения» – формирование профессиональных компетенций при работе с основными видами и методами тестирования программного обеспечения (ПО).

Методические рекомендации направлены на освоение технологии тестирования программ, развитие практических навыков создания ручных и автоматизированных тестов, а также формирование компетенций в составлении необходимой документации.

В результате освоения учебной дисциплины студент:

познает:

– модели разработки ПО, место тестирования в жизненном цикле разработки ПО;

– отличительные особенности различных видов тестирования информационных систем;

– виды документов, используемых при проведении тестирования;

– метрики тестирования программного обеспечения;

– приемы отладки и ручного тестирования ПО;

научится:

– планировать тестирование, выявлять риски продукта и проекта, выбирать методы тестирования, осуществлять тестирование, оценивать покрытие кода тестами, составлять итоговую отчетность по тестированию;

– строить набор тестов для тестирования информационной системы;

овладеет:

– методами и техниками разработки тест-дизайна;

– навыками работы с различными инструментами тестирования и отладки;

– навыками использования различных методов ручного и автоматизированного тестирования ПО.

Для защиты лабораторной работы студент подготавливает отчет, включающий титульный лист, формулировку задания, описание входных и выходных данных, а также подробные результаты тестирования.

Защита работы состоит из двух частей: практической, где студент демонстрирует и объясняет выполненную работу, и теоретической, где он отвечает на вопросы по теме лабораторной работы.

Методические рекомендации содержат описание пяти лабораторных работ, задания, контрольные вопросы, список литературы [1–4], которую необходимо использовать при подготовке к защите лабораторных работ.

1 Лабораторная работа № 1. Техники тест-дизайна

Цель работы: освоить техники проектирования тестовых случаев для эффективного выявления дефектов в ПО.

1.1 Теоретические положения

1.1.1 Техники тест-дизайна.

Техники тест-дизайна необходимы при разработке тест-кейсов для следующих целей:

– структурирование процедуры тестирования: тест-дизайн позволяет определить наборы данных и идеи для написания эффективных тест-кейсов;

– оптимизация ресурсов: тест-дизайн позволяет создавать тесты, которые помогут выявить серьезные ошибки, вдумчиво подойти к созданию тестов и не тратить впустую ресурсы;

– сокращение количества тестов: с помощью техник тест-дизайна можно создать меньше тестов.

Существуют следующие основные техники (методики) тест-дизайна: классы эквивалентности, граничные значения, доменное тестирование, попарное тестирование (техника pairwise), тестирование на основе таблиц альтернатив, тестирование на основе состояний и переходов.

1.1.2 Классы эквивалентности.

При изучении данной темы необходимо ознакомиться с [1, с. 237–241].

Класс эквивалентности – набор входных данных, обрабатываемых программой одинаковым образом и приводящих к одинаковому результату.

Метод эквивалентного разбиения (Equivalence Partitioning) заключается в следующем. Область всех возможных наборов входных данных программы по каждому параметру разбивают на конечное число групп классов эквивалентности. Наборы данных такого класса объединяют по принципу обнаружения одних и тех же ошибок: если набор какого-либо класса обнаруживает некоторую ошибку, то предполагается, что все другие тесты этого класса эквивалентности тоже обнаружат эту ошибку и наоборот.

Классы эквивалентности определяются путем анализа ограничений и требований, предъявляемых к каждому входному значению, которые указаны в техническом задании или спецификации. Каждое ограничение анализируется и разбивается на группы, представляющие допустимые и недопустимые классы.

Допустимые классы включают правильные данные, **недопустимые классы** – неправильные данные. Для допустимых и недопустимых классов тесты проектируют отдельно. При построении тестов допустимых классов учитывают, что каждый тест должен проверять по возможности максимальное количество различных входных условий. Такой подход позволяет минимизировать общее число необходимых тестов. Для каждого недопустимого класса эквива-

лентности формируют свой тест. Последнее обусловлено тем, что определенные проверки с ошибочными входами скрывают или заменяют другие проверки с ошибочными входами.

Правила определения классов эквивалентности [2, с. 65].

1 Если входное условие задает диапазон допустимых значений (например, «целое число от 1 до 50»), то следует выделить три класса эквивалентности:

– один допустимый класс, включающий все целые значения, входящие в диапазон (1 ≤ значение ≤ 50);

– два недопустимых класса, одни из которых включает все целые значения, меньшие нижней границы диапазона (значение < 1), а другой включает все целые значения, большие верхней границы диапазона (значение > 50).

2 Если входное условие ограничивает количество элементов, которые обрабатываются одинаковым образом (например, если известно разрешенное количество файлов одного типа для загрузки от 1 до 5 файлов), то определяют:

– один допустимый класс со значениями в пределах диапазона (1 ≤ количество файлов ≤ 5);

– два недопустимых. Первый недопустимый класс включает количество, меньшее минимально допустимого (< 1 файла). Во второй недопустимый класс войдет большее максимально допустимого количество файлов > 5.

3 Если входное условие представляет собой множество отдельных, дискретных значений, и есть основания полагать, что программа обрабатывает каждое значение по-разному, то для каждого допустимого значения создается отдельный допустимый класс эквивалентности, а все остальные варианты ввода – один недопустимый класс. Например, приложение может обрабатывать файлы следующих форматов: JPEG, PNG. Тогда можно выделить два допустимых класса эквивалентности для JPEG и PNG. В недопустимый класс войдут все прочие неподдерживаемые форматы.

4 Если входное условие описывает обязательное требование (например, «первый символ идентификатора – буква»), то выделяются два класса эквивалентности:

– допустимый класс, который включает все значения, соответствующие требованию (первый символ – буква, например, User123, admin);

– недопустимый класс, который включает все значения, не соответствующие требованию (первый символ – не буква, например, 123user, _guest).

5 Если есть основания полагать, что различные элементы внутри одного класса эквивалентности могут обрабатываться программой по-разному, то этот класс следует разбить на более мелкие классы эквивалентности.

Построение тестов на основе классов эквивалентности выполняется следующим образом.

1 Проектирование новых тестов, каждый из которых покрывает как можно большее число непокрытых допустимых классов эквивалентности, до тех пор, пока все допустимые классы эквивалентности не будут покрыты (только не общими) тестами.

2 Запись тестов, каждый из которых покрывает один и только один из непокрытых недопустимых классов эквивалентности, до тех пор, пока все недопустимые классы эквивалентности не будут покрыты тестами.

Тестирование классов эквивалентности – это самая основная методика тест-дизайна. Она помогает тестировщикам выбрать небольшое подмножество из всех возможных тестовых сценариев и при этом обеспечить приемлемое покрытие. У этой техники есть еще один плюс. Она приводит к идее о тестировании **граничных значений** (Boundary Value Analysis, BVA) – второй ключевой технике тест-дизайна.

1.1.3 Граничные значения.

Граничные значения включают в себя значения, находящиеся на границе классов эквивалентности, а также значения, непосредственно прилегающие к этим границам (ниже и выше). Анализ показывает, что в этих местах часто резко увеличивается возможность обнаружения ошибок.

Правила определения граничных значений.

1 Если входные данные должны находиться в диапазоне [*a*, *b*], то граничные значения включают:

- минимальную границу (*a*);
- максимальную границу (*b*);
- значения чуть ниже минимума (*a* – 1);
- значения чуть выше максимума (*b* +1);
- значения внутри диапазона (например, *a* + 1, *b* – 1).

Пример – Дано требование к полю «Возраст», которое должно принимать значения от 18 до 65 лет. Граничные значения:

- 17 (нижняя недопустимая граница, *a* – 1);
- 18 (нижняя допустимая граница, *a*);
- 19 (внутри диапазона, *a* + 1);
- 64 (внутри диапазона, *b* – 1);
- 65 (верхняя допустимая граница, *b*);
- 66 (верхняя недопустимая граница, *b* + 1).

2 Если допустимое значение плавающей переменной составляет от –1.0 до 1.0, тестируются значения –1.0, 1.0, –1.001 и 1.001.

3 Если входные или выходные данные программы представляют собой упорядоченный набор, тестируются первый и последний элементы набора.

4 Если программа поддерживает список, тестируются значения из этого списка. Все остальные значения считаются недопустимыми.

5 Если программа настроена на работу с прописными символами, тестируются граничные значения для A и Z. Также тестируются @ и [, потому что в коде ASCII символ @ предшествует A, а символ [следует сразу за Z.

6 При считывании из файла или записи в файл тестируются первый и последний символы файла.

7 Если система работает с денежными значениями, где минимальный шаг составляет 1 копейку (0,01 денежной единицы), при тестировании диапазона

от *a* до *b* обязательно проверяются значения на границах этого шага: (*a* – 0,01) и (*b* + 0,01).

8 Если для переменной указано несколько диапазонов значений, каждый диапазон считается отдельным классом. Если подмножества значений из этих диапазонов не пересекаются, тестируются крайние значения, а также граничные значения над верхней границей и под нижней границей.

Специальные значения.

После применения двух указанных выше стратегий анализа граничных значений рекомендуется исследовать программу на наличие специальных значений, с помощью которых можно обнаружить множество ошибок. Некоторые примеры приведены ниже.

1 В случае с целыми числами нужно обязательно протестировать ноль, если он входит в допустимый класс эквивалентности.

2 При тестировании времени необходимо для каждого элемента (час, минута и секунда) протестировать значения 59 и 0, независимо от ограничения, установленного для входной переменной. Помимо граничных значений входной переменной необходимо всегда тестировать значения –1, 0, 59 и 60.

3 При тестировании даты (год, месяц и день) необходимо добавить тестовые сценарии, например, для количества дней в конкретном месяце, количества дней в феврале в високосном году или количества дней в невисокосном году.

1.1.4 Доменное тестирование.

Доменное тестирование (Domain Analysis) – это техника тест-дизайна, основанная на разбиении входных данных на логические группы (домены), где все значения внутри одного домена обрабатываются программой одинаково.

Данная техника используется в случаях, когда несколько входных параметров могут быть протестированы одновременно. В основе этой техники лежат эквивалентное разбиение и анализ граничных значений. Цель – проверить корректность обработки данных на границах между доменами.

В качестве примера рассмотрим тестирование формы поиска с фильтрами по цене и категориям товаров. Вместо того чтобы отдельно тестировать диапазон цен и отдельно – работу категорий, доменное тестирование позволяет сразу проверить критические комбинации: минимальная цена с самой популярной категорией, максимальная цена с редкой категорией, нулевая цена с несуществующей категорией. Такой подход особенно ценен в сложных формах с множеством взаимосвязанных полей, где количество возможных комбинаций растет экспоненциально, и полный перебор всех вариантов становится невозможным. Доменное тестирование дает разумный компромисс между полнотой проверки и временем тестирования, фокусируясь на наиболее вероятных и критически важных комбинациях входных параметров.

Домен – это подмножество входных данных, которые:

– обрабатываются единообразно (например, все числа от 1 до 100);

– имеют общие правила валидации (например, только буквы A–Z);

– вызывают одинаковое поведение системы (например, числа более 1000 приводят к ошибке).

Домен может включать один или несколько классов эквивалентности (если логика обработки совпадает) и граничные значения (как точки перехода между доменами).

Этапы формирования доменов.

1 Анализ требований, в ходе которого необходимо выявить все ограничения на входные данные (диапазоны, форматы, условия).

2 Разбиение входных данных на классы эквивалентности. Класс эквивалентности – это подмножество значений в домене, для которых поведение программы одинаково.

3 Определение границ доменов, в ходе которого необходимо найти точные граничные значения, разделяющие домены.

4 Выбор тестовых значений. Для каждого домена определяются:

– одно значение внутри домена (при необходимости);

– значения на границах (минимальное / максимальное);

– значения за границами (например, 17 и 66 для возраста с диапазоном допустимых значений от 18 до 65).

5 Проверка комбинаций доменов (если они взаимодействуют). Скомбинировать эти значения нужно таким образом, чтобы отдельные входные параметры (или, например, поля формы) можно было протестировать одновременно.

Позитивные значения каждого поддомена объединяются для формирования тестов, которые покрывают сразу несколько условий. Например, объединение корректного имени, возраста и e-mail позволяет протестировать форму регистрации с минимальным количеством тестов.

Для негативных тестов объединение нескольких источников ошибок в один домен недопустимо, т. к. это усложняет диагностику. Поэтому каждое негативное условие всегда проверяется отдельно.

Пример доменного тестирования. Даны требования к двум полям формы:

– поле «Фамилия» должно иметь длину от 2 до 20 символов, допускается ввод только букв (кириллица);

– поле «Возраст» может принимать целые числа от 0 до 120.

В таблицах 1.1 и 1.2 приведены классы эквивалентности и граничные значения, определенные для полей «Фамилия» и «Возраст».

Таблица 1.1 – Классы эквивалентности и граничные значения для поля «Фамилия»

Класс эквивалентности	Описание класса	Граничные значения	Пример
Допустимые	Корректная длина 2–19 символов включительно	Минимальное значение 2	Эн
		Максимальное значение 20	Христорожденственский
	Кириллица	–	Иванов
Недопустимые	Пустое значение	–	""
	Недостаток длины	1 символ	А
	Превышение длины	21 символ	Строка длиннее 20 символов
	Недопустимые символы	–	"Иванов123", "User@"

Таблица 1.2 – Классы эквивалентности и граничные значения для поля «Возраст»

Класс эквивалентности	Описание класса	Граничные значения	Пример
Допустимые	Целое число	–	0
	Корректный диапазон 0–120 включительно	Минимальное значение	0
		Максимальное значение 120	120
Недопустимые	Нецелые числа	–	18,5
	Отрицательные числа	–1	–1
	Превышение значения	Числа > 120	121
	Нечисловой ввод		"двадцать", abc, 18 лет
	Пустое значение		""

В таблице 1.3 приведены домены, определенные для полей «Фамилия» и «Возраст».

Таблица 1.3 – Домены

Тест	Домен	Пример	Результат
Позитивные тесты для поля «Фамилия»			
Тест 1	Кириллица, минимальное значение 2	Эн	Позитивный
Тест 2	Кириллица, максимальное значение 20 символов	Христорожденствен-ский	Позитивный
Негативные тесты для поля «Фамилия»			
Тест 3	Пустое значение	–	""
Тест 4	Недостаток длины	1 символ	А
Тест 5	Превышение длины	21 символ	Строка длиннее 20 символов
Тест 6	Недопустимые символы	–	"Иванов123", "User@"
Позитивные тесты для поля «Возраст»			
Тест 7	Целое минимальное значение	0	Позитивный
Тест 8	Целое максимальное значение	120	Позитивный
Негативные тесты для поля «Возраст»			
Тест 9	Нецелые числа	–	18,5
Тест 10	Отрицательные числа	–1	–1
Тест 11	Превышение значения	Числа > 120	121
Тест 12	Нечисловой ввод		"двадцать", abc, 18 лет
Тест 13	Пустое значение	–	""

В таблице 1.4 приведен финальный чек-лист позитивных и негативных тестов для полей «Фамилия» и «Возраст».

Таблица 1.4 – Чек-лист позитивных и негативных тестов

Тест	Поле «Фамилия»	Поле «Возраст»
Позитивные тесты		
Тест 1	Кириллица, минимальное значение 2 символа	Целое минимальное значение 0
Тест 2	Кириллица, максимальное значение 20 символов	Целое максимальное значение 120
Негативные тесты		
Тест 3	Пустое значение	Целое минимальное значение 0
Тест 4	Недостаток длины	Целое минимальное значение 0
Тест 5	Превышение длины	Целое минимальное значение 0
Тест 6	Недопустимые символы	Целое минимальное значение 0
Тест 7	Кириллица, минимальное значение 2 символа	Нецелые числа
Тест 8	Кириллица, минимальное значение 2 символа	Отрицательные числа
Тест 9	Кириллица, минимальное значение 2 символа	Превышение значения
Тест 10	Кириллица, минимальное значение 2 символа	Нечисловой ввод
Тест 11	Кириллица, минимальное значение 2 символа	Пустое значение

В результате применения техники доменного тестирования были сформированы два позитивных теста вместо шести, полученных по итогам определения классов эквивалентности и граничных значений, а также девять негативных тестов.

1.1.5 Попарное тестирование.

Попарное тестирование (техника Pairwise) – это техника тест-дизайна, которая позволяет сократить количество тест-кейсов за счет проверки всех возможных пар значений параметров вместо полного перебора комбинаций.

Основная идея здесь заключается в том, что большинство дефектов возникает из-за взаимодействия двух параметров, а не всех сразу. Поэтому достаточно проверить все уникальные пары значений.

Этапы попарного тестирования.

1 Выделить тестируемые параметры системы и возможные значения для каждого. Пример: для формы регистрации с параметрами:

- пароль: допустимый, невалидный (короткий), пустой;
- логин: корректный, некорректный (спецсимволы), пустой;
- e-mail: валидный, невалидный (без @), пустой.

2 Построить таблицу комбинаций, где каждая строка – тест-кейс, покрывающий **уникальные пары значений** между разными параметрами. Пример для трех параметров приведен в таблице 1.5.

Количество тест-кейсов можно сократить с помощью специализированных инструментов (AllPairs, PICT).

3 Проверить покрытие пар. Необходимо убедиться, что каждая возможная пара значений между разными параметрами встречается хотя бы в одном тест-кейсе.

Таблица 1.5 – Чек-лист позитивных и негативных проверок

Тест-кейс	Пароль	Логин	E-mail
1	Допустимый	Корректный	Валидный
2	Допустимый	Некорректный	Невалидный
3	Допустимый	Пустой	Пустой
4	Невалидный	Корректный	Невалидный
5	Невалидный	Некорректный	Пустой
6	Невалидный	Пустой	Валидный
7	Пустой	Корректный	Пустой
8	Пустой	Некорректный	Валидный
9	Пустой	Пустой	Невалидный

Пример покрытия пар для «Пароль-Логин»:

– допустимый + корректный (тест-кейс 1);
– допустимый + некорректный (тест-кейс 2);
– допустимый + пустой (тест-кейс 3);
– невалидный + корректный (тест-кейс 4);
– ... и т. д.

Дефекты обычно проявляются во взаимодействии между параметрами (например, «пустой логин + валидный пароль» даст ошибку).

Преимущества Pairwise:

– сокращение количества тест-кейсов на 70 %...90 % по сравнению с полным перебором;

– обнаружение 90 % багов, т. к. большинство ошибок – парные.

– универсальность, т. к. применяется для UI, API, конфигурационного тестирования.

1.1.6 Тестирование на основе таблиц альтернатив.

Тестирование на основе таблиц альтернатив (Decision Table Testing) – техника тестирования (по методу белого ящика), в которой проверяется поведение системы в зависимости от комбинаций входных условий и соответствующих им действий (решений). Это особенно полезно для сложных бизнес-правил, где результат зависит от множества условий. Таблица альтернатив помогает явно отобразить все возможные сценарии и ожидаемые результаты.

Данная техника применяется для систем с множеством условий (например, банковские операции, страховые калькуляторы), когда результат зависит от комбинации входных данных, а также для верификации бизнес-логики и валидации правил.

Этапы применения Decision Table Testing.

1 Определить условия и действия.

Условия (Conditions) – это входные параметры или правила, влияющие на результат.

Действия (Actions) – это выходные результаты или операции системы.

Рассмотрим пример для банковского перевода.

Условия:

– достаточно ли средств на счете? (да / нет);
– введен ли верный pin-код? (да / нет);
– превышен ли дневной лимит? (да / нет).

Действия:

– разрешить перевод;
– отклонить перевод;
– заблокировать карту.

2 Построить таблицу альтернатив (таблица 1.6), где столбцы – это уникальные комбинации условий (тест-кейсы), а строки – условия и соответствующие действия.

Таблица 1.6 – Таблица альтернатив для перевода денег

Условия / Действия	Тест 1	Тест 2	Тест 3	Тест 4
Достаточно средств? (Да/Нет)	Да	Да	Нет	Нет
Верный PIN? (Да/Нет)	Да	Нет	Да	Нет
Превышен лимит? (Да/Нет)	Нет	Нет	Нет	Да
Действия				
Разрешить перевод	Да	Нет	Нет	Нет
Отклонить перевод	Нет	Да	Да	Да
Заблокировать карту	Нет	Нет	Нет	Нет

3 Определить тест-кейсы. Каждый столбец таблицы – это отдельный тест-кейс:

– тест 1: достаточно средств + верный PIN + лимит не превышен → перевод разрешен;
– тест 2: достаточно средств + неверный PIN → перевод отклонен;
– тест 3: недостаточно средств + верный PIN → перевод отклонен;
– тест 4: недостаточно средств + неверный PIN + лимит превышен → перевод отклонен.

4 Проверить покрытие (убедиться, что таблица покрывает все возможные комбинации условий) и выполнить тесты.

Преимуществами Decision Table Testing является покрытие сложных правил, т. к. данная техника явно отображает зависимости между условиями и действиями, а также минимизация пропущенных сценариев – исключаются «слепые зоны» в тестировании. Инструменты: Excel, специализированные программы (TestRail, Qase).

1.1.7 Тестирование на основе состояний и переходов.

Тестирование на основе состояний и переходов (State Transition Testing) – это техника тест-дизайна, которая проверяет поведение системы в зависимости от ее состояний и переходов между ними. Она применяется для систем, где:

– объект может находиться в разных состояниях;
– переход между состояниями зависит от событий или условий;

– поведение системы меняется в зависимости от текущего состояния.

Система рассматривается как конечный автомат, где состояние – это стабильное условие системы (например, «Заказ оплачен»), переход – это изменение состояния под действием события (например, «Получить платеж»), событие – действие или условие, инициирующее переход.

Алгоритм применения тестирования на основе состояний и переходов.

1 Выбор одного тестируемого объекта (например, заказ в интернет-магазине). Объект можно рассматривать, например, как строку в таблице базы данных.

2 Определение списка всех возможных состояний объекта. Пример для заказа «Создан», «Оплачен», «Отправлен», «Доставлен», «Отменен».

3 Определение событий и переходов. Фиксируются все возможные события, вызывающие переходы между состояниями. Указываются условия для каждого перехода.

4 Построение диаграммы состояний и переходов (рисунок 1.1). Визуализируем состояния (узлы) и переходы (стрелки). Стрелки подписываются соответствующими событиями. На диаграмму не добавляют элементы интерфейса, только бизнес-логику.

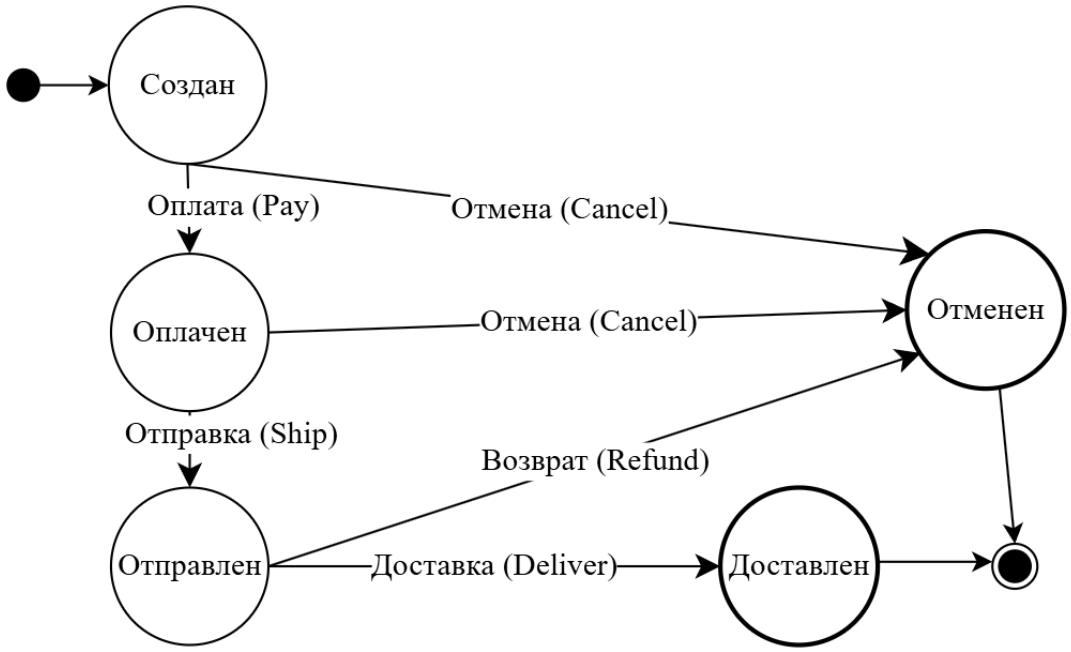


Рисунок 1.1 – Диаграмма состояний и переходов объекта «Заказ» в интернет-магазине

5 Для каждого перехода создается отдельный тест-кейс. Проверяются как валидные, так и невалидные переходы. Чек-лист тестов для рассматриваемого примера приведен в таблице 1.7.

6 Проведение тестирования и анализа (сравнивается фактическое поведение с ожидаемым).

Таблица 1.7 – Чек-лист тестирования переходов между состояниями

Номер теста	Начальное состояние	Событие	Ожидаемый результат	Тип проверки
1	Создан	Оплата	Переход в «Оплачен»	Валидный
2	Оплачен	Отправка	Переход в «Отправлен»	Валидный
3	Отправлен	Доставка	Переход в «Доставлен»	Валидный
4	Создан	Отмена	Переход в «Отменен»	Валидный
5	Оплачен	Отмена	Переход в «Отменен»	Валидный
6	Отправлен	Возврат	Переход в «Отменен»	Валидный
7	Создан	Отправка	Ошибка «Нельзя отправить неоплаченный заказ»	Невалидный
8	Оплачен	Доставка	Ошибка «Заказ еще не отправлен»	Невалидный
9	Доставлен	Отмена	Ошибка «Заказ уже доставлен»	Невалидный
10	Отменен	Оплата	Ошибка «Заказ отменен»	Невалидный

Задание

Необходимо получить у преподавателя задачи по номеру своего варианта. В каждый вариант входят задачи на применение следующих техник тест-дизайна:

– классы эквивалентности и граничные значения;
– доменное тестирование;
– попарное тестирование;
– тестирование на основе таблиц альтернатив;
– тестирование на основе состояний и переходов.

Содержание отчета: тема и цель работы; решение задач по варианту.

Контрольные вопросы

1 В чем разница между классами эквивалентности и граничными значениями? Почему важно тестировать и то, и другое?

2 Как классы эквивалентности и граничные значения помогают снизить количество тестовых случаев? Приведите пример.

3 Сколько классов эквивалентности у булевого поля (Да/Нет)?

4 Чем класс эквивалентности отличается от домена?

5 Форма регистрации требует логин (от 5 до 10 символов, латиница и цифры) и пароль (от 8 до 15 символов, латиница, цифры и спецсимволы). Применить доменное тестирование для создания тестовых случаев.

6 В чем суть Pairwise-тестирования? Когда лучше использовать Pairwise, а не полный перебор?

7 Что такое таблица решений? Для правила «Скидка 10 % при сумме свыше1000 руб.» построить простую таблицу.

8 Что такое «состояние» системы? Перечислить три состояния для объекта «Заказ». Как обозначаются переходы на диаграмме состояний и переходов?

2 Лабораторная работа № 2. Юнит-тестирование

Цель работы: изучить основы юнит-тестирования, научиться тестировать отдельные методы, классы без зависимостей.

2.1 Теоретические положения

Юнит-тестирование – это метод тестирования отдельных компонентов (юнитов) программного кода (функций, методов, классов) на корректность работы. Каждый тест проверяет одну конкретную функциональность в изоляции от других частей системы.

Юнит-тест (unit test) – это отдельная часть программы, которая вызывает тестируемый модуль и проверяет корректность его работы. Тест считается непройденным, когда его результат не совпадает с ожидаемым.

Задача сводится к следующему: написать тесты для методов, изменение кода в которых может привести к ошибкам, в том числе в тех местах, которые мы уже протестировали.

Юнит-тест должен быть автоматизированным и повторяемым, давая стабильные результаты при каждом запуске, простым в реализации, чтобы разработчики могли легко создавать и поддерживать его, сохраненным вместе с кодом для последующего регрессионного тестирования и проверки изменений, доступным для запуска всем членам команды, например, с помощью одной команды, и выполняться быстро, чтобы не замедлять процесс разработки.

Юнит-тесты должны располагаться в отдельном проекте. Если в решении несколько проектов, то каждому соответствует проект с тестами.

Рекомендуется проект с тестами называть так же, как и основной проект, но добавлять через точку слово Test в конце наименования. Для проекта MailSender будет создан проект MailSender.Test.

Называть методы нужно так, чтобы было понятно, какой кусок кода он тестирует, каков сценарий проверки, входящие переменные, результат или ожидаемое поведение. Схема наименования может быть такой:

[Тестирующийся метод]_[Сценарий]_[Ожидаемое поведение].

Например, в библиотеке CodePasswordDLL есть метод getCodePassword, где на вход подается пароль, а на выходе получаем его зашифрованным. Тестовый метод для него будет называться getCodePassword_abc_bcd.

Фреймворки тестирования. Чтобы создавать юнит-тесты, в .NET можно использовать фреймворки: xUnit, NUnit, MSTest.

Подход AAA. В литературе по юнит-тестированию как один из вариантов составления юнит-теста описывают подход AAA (Arrange, Act, Assert – Инициализация, Действие, Ожидаемый результат). Если посмотреть на юнит тест, то для большинства можно четко выделить три части кода:

– Arrange (настройка) – в этом блоке кода мы настраиваем тестовое окружение тестируемого юнита;

– Act – выполнение или вызов тестируемого сценария;

– Assert – проверка того, что тестируемый вызов ведет себя определенным образом.

Что и как тестировать.

Следует убедиться, что во время тестирования задействованы все операторы тестируемой функции, т. е. обеспечивайте достаточно большой набор вариантов входных параметров, достаточный для тестирования всех случаев каждого ветвления, входа во все циклы и т. п.

Нужно проверить, что все возможные ветвления и условия кода были протестированы. Так, например, для конструкции switch это означает, что все возможные значения, которые могут быть переданы в switch, должны быть протестированы.

Тестирование циклов (неофициально называемое «тест 0, 1, 2») базируется на том, что если в коде есть цикл, то, чтобы убедиться в его работоспособности, нужно его выполнить 0, 1 и 2 раз. Если он работает корректно во второй итерации, то можно ожидать его корректную работу и для всех последующих итераций (3, 4, 10, 100 и т. д.). Эти тесты помогают убедиться, что цикл правильно обрабатывает крайние случаи:

– тест 0 (0 итераций): проверяет, что цикл работает корректно, когда он не должен выполняться ни разу. Например, это может быть проверка, что функция не вызывает ошибок, если список пустой;

– тест 1 (1 итерация): проверяет, что цикл выполняется ровно один раз. Это позволяет удостовериться, что логика внутри цикла работает правильно, когда есть только один элемент для обработки;

– тест 2 (2 итерации): проверяет, что цикл выполняется дважды. Это важно для проверки логики, которая зависит от обработки нескольких элементов.

Тестируйте разные типы ввода, чтобы убедиться, что проверяемый «кусок кода» правильно их обрабатывает.

Для целых чисел следует убедиться, что было проверено, как тестируемая функция обрабатывает 0, отрицательные и положительные значения. При наличии пользовательского ввода также нужно проверить вариант возникновения переполнения.

Для чисел типа с плавающей запятой следует убедиться, что было проверено, как тестируемая функция обрабатывает значения, которые имеют неточности (значения, которые немного больше/меньше ожидаемых).

Для строк следует убедиться, что было протестировано, как проверяемая функция обрабатывает пустую строку, строку с допустимыми значениями, строку с пробелами и строку, содержимым которой являются одни пробелы.

Все тесты должны быть независимы, т. е. не стоит в одном TestMethod вызывать все функции. Более того, тесты для разных функций можно поместить в разные классы.

Пример – Необходимо написать на C# программу для расчета кинетической энергии E_k, которая рассчитывается по формуле $E_k = m\ v^2/2$, где m – масса тела, кг, допустимый диапазон: 0.1–1000 кг; v – скорость, м/с, допустимый диапазон 0.1–100 м/с.

Текст программы:

```
public class KineticEnergyCalculator
{
   /// <summary>
   /// Рассчитывает кинетическую энергию тела
   /// </summary>
   /// <param name="mass">Масса (0.1–1000 кг)</param>
   /// <param name="velocity">Скорость (0.1–100 м/с)</param>
   /// <returns>Кинетическая энергия (Джоули)</returns>
   /// <exception cref="ArgumentOutOfRangeException">При недопустимых значениях</exception>
   public static double CalculateKineticEnergy(double mass, double velocity)
   {
      if (mass < 0.1 || mass > 1000)
         throw new ArgumentOutOfRangeException(nameof(mass), "Масса должна быть от 0.1 до 1000 кг");
      if (velocity < 0.1 || velocity > 100)
         throw new ArgumentOutOfRangeException(nameof(velocity), "Скорость должна быть от 0.1 до 100 м/с");
      return mass * Math.Pow(velocity, 2) / 2;
   }
}
```

В таблице 2.1 приведены классы эквивалентности для решаемой задачи.

Таблица 2.1 – Классы эквивалентности и граничные значения

Параметр	Допустимый диапазон	Допустимые классы	Недопустимые классы	Граничные значения	Пример ввода	Ожидаемый результат	Обоснование
Масса (кг)	0.1–1000	0.1 ≤ mass ≤ 1000	mass < 0.1, mass > 1000	0.1, 0.2, 500, 999, 1000	10	Корректный расчет	Проверка валидного диапазона
Скорость (м/с)	0.1–100	0.1 ≤ velocity ≤ 100	velocity < 0.1, velocity > 100	0.1, 0.2, 50, 99, 100	5	Корректный расчет	Проверка валидного диапазона
Энергия (Дж)	>0	Положительные числа	Отрицательные, Infinity	0.0005 (min), 5 000 000 (max)	2 кг, 3 м/с → 9 Дж	Корректный результат	Проверка формулы

В таблице 2.2 приведен чек-лист проверок для решаемой задачи.

Таблица 2.2 – Чек-лист проверок

Номер проверки	Тип теста	Входные данные (масса, скорость)	Ожидаемый результат	Покрытие
1	Позитивный [Fact]	(2.0, 3.0)	9.0 Дж	Базовый случай
2	Позитивный [Theory]	(0.1, 0.1)	0.0005 Дж	Минимальные значения
3	Позитивный [Theory]	(1000, 100)	5 000 000 Дж	Максимальные значения
4	Негативный [Theory]	(-1.0, 5.0)	ArgumentOutOfRangeException	Отрицательная масса
5	Негативный [Theory]	(10.0, 0.05)	ArgumentOutOfRangeException	Слишком низкая скорость
6	Граничный [Theory]	(999, 99)	4 890 049.5 Дж	Значения, близкие к максимуму

Разработанными тестами покрыты все граничные значения массы и скорости, проверены оба исключительных сценария, учтены минимальные и максимальные комбинации.

В данной задаче тестируемой системой (SUT, System Under Test) является класс KineticEnergyCalculator, а конкретно – его метод Calculate, который вычисляет кинетическую энергию по заданным массе и скорости. Этот класс представляет собой функциональность, которая тестируется [5, с. 71]. **Тестируемая система** предоставляет точку входа для поведения, которое проверяется.

Класс KineticEnergyCalculator является SUT потому, что он представляет собой изолированный объект тестирования: в каждом тесте создается новый экземпляр класса (var sut = new KineticEnergyCalculator()), чтобы гарантировать независимость тестов. Тесты проверяют только его поведение, не затрагивая внешние зависимости (например, базы данных или API). Система SUT создается заново в каждом тесте (не сохраняет состояние между вызовами) и не зависит от внешних систем.

Далее приведены тесты в xUnit.

В данном коде использован атрибут [Fact], поскольку необходимо протестировать один конкретный сценарий без вариаций входных данных.

```
[Fact]
public void CalculateKineticEnergy_PositiveValues_ReturnsCorrectResult()
{
    // Arrange: Инициализация SUT
    var sut = new KineticEnergyCalculator();
    double mass = 2.0, velocity = 3.0;

    // Act: Вызов метода Calculate
    double result = sut.Calculate(mass, velocity);
```

```
    // Assert: Проверка результата
    Assert.Equal(9.0, result, precision: 2);
}
```

Далее приведен тест с атрибутом [Theory], поскольку нужно проверить несколько валидных и невалидных комбинаций – граничные значения и исключения. Атрибут [Theory] позволяет проверить множество сценариев с разными входными данными в одном методе, что удобно для группировки однотипных тестов.

```
[Theory]
[InlineData(0.1, 0.1, 0.0005)]     // Минимальные значения.
[InlineData(1000, 100, 5_000_000)]  // Максимальные значения
[InlineData(10, 5, 125)]            // Нормальные значения
public void CalculateKineticEnergy_ValidInputs_ReturnsExpectedResult(
    double mass, double velocity, double expected)
{
    // Arrange
    var sut = new KineticEnergyCalculator();

    // Act
    double result = sut.Calculate(mass, velocity);

    // Assert
    Assert.Equal(expected, result, precision: 2);
}

[Theory]
[InlineData(-1, 5)]    // Отрицательная масса
[InlineData(10, 0)]    // Нулевая скорость
[InlineData(2000, 10)] // Масса превышает максимум
public void CalculateKineticEnergy_InvalidInputs_ThrowsException(
    double mass, double velocity)
{
    // Arrange
    var sut = new KineticEnergyCalculator();

    // Act & Assert
    Assert.Throws<ArgumentOutOfRangeException>(
        () => sut.Calculate(mass, velocity));
}
```

Задание

Необходимо получить у преподавателя задание, в котором следует:

1) написать код решения на C#;

2) составить перечень классов эквивалентности граничных значений в виде таблицы 2.1;

3) составить чек-лист проверок в виде таблицы 2.2;

4) написать код юнит-тестов в полном соответствии с чек-листом, которые проверяют корректность выполнения каждой операции из условия задачи. В каждом юнит-тесте использовать понятие тестируемой системы sut. Использовать только фреймворк xUnit [3];

5) определить метрики покрытия: code coverage, branch coverage [3, с. 28–31]. Чтобы увидеть процент покрытия ветвей и условий в проекте с использованием xUnit, можно использовать инструменты анализа покрытия кода. В Visual Studio и .NET экосистеме для этого можно использовать фреймворк Coverlet, инструмент командной строки ReportGenerator, расширение Fine Code Coverage;

6) написать код тестовых заглушек (моков и стабов) для решаемой задачи [5, с. 117]. Описать назначение заглушек, обосновать достаточность их количества. Использовать фреймворк Moq.

Содержание отчета: тема и цель работы; ответы на пункты 1–6 задания.

Контрольные вопросы

1 Что такое юнит-тест? По каким правилам строится юнит-тест?
2 Чем юнит-тесты отличаются от интеграционных?
3 Что такое SUT (System Under Test)?
4 Какой атрибут в xUnit отмечает параметризованный тест?
5 Как назвать тест по соглашению AAA?
6 Как рассчитываются метрики покрытия code coverage и branch coverage?

3 Лабораторная работа № 3. Разработка документации для тестирования

Цель работы: освоить принципы и методы разработки комплекта тестовой документации для проверки качества приложений.

3.1 Теоретические положения

Документация для тестирования играет ключевую роль в обеспечении качества программного обеспечения. Она определяет цели, область применения, методы и критерии оценки качества, а также предоставляет четкие инструкции для выполнения тестовых процедур.

В процессе тестирования UI обычно используются следующие виды документации:

– план тестирования (Test Plan): определяет цели, область применения, методы, ресурсы и расписание тестирования;

– тест-кейсы (Test Scenarios): описывают последовательность действий, которые необходимо выполнить для проверки определенной функциональности;

– тестовые наборы (Test Suites): группируют тестовые сценарии по определенным критериям (например, по функциональности, по типу тестирования);

– чек-листы (Checklists): представляют собой списки проверок, которые необходимо выполнить для оценки качества определенного аспекта приложения;

– отчеты об ошибках (Bug Reports): описывают обнаруженные ошибки и предоставляют информацию, необходимую для их исправления;

– отчет о результатах тестирования.

При тестировании UI необходимо учитывать следующие аспекты:

– UI-элементы: проверка корректности отображения и функционирования кнопок, текстовых полей, списков, изображений, иконок и других элементов интерфейса;

– формы: проверка валидации данных, обработки ошибок, сохранения и отправки данных;

– навигация: проверка корректности переходов между страницами, работы меню, ссылок и других элементов навигации;

– адаптивность: проверка корректности отображения и функционирования приложения на различных устройствах и разрешениях экрана;

– доступность: проверка соответствия приложения требованиям доступности для пользователей с ограниченными возможностями.

Этапы выполнения работы.

Этап 1. Анализ требований и спецификаций.

На этом этапе необходимо тщательно изучить требования и спецификации веб-приложения, чтобы понять его функциональность, архитектуру и особенности пользовательского интерфейса. Важно обратить внимание на следующие аспекты:

– функциональные требования: какие функции должно выполнять приложение, какие данные оно должно обрабатывать;

– нефункциональные требования: какие требования предъявляются к производительности, безопасности, надежности и удобству использования приложения;

– требования к пользовательскому интерфейсу: какие элементы интерфейса должны быть реализованы, как они должны выглядеть и функционировать;

– требования к адаптивности: на каких устройствах и разрешениях экрана должно корректно отображаться приложение;

– требования к доступности: какие требования предъявляются к доступности приложения для пользователей с ограниченными возможностями.

Этап 2. Разработка плана тестирования.

План тестирования – это документ, который определяет цели, область применения, методы, ресурсы и расписание тестирования. Он служит руководством для всех участников команды и помогает обеспечить эффективное и последовательное тестирование. Основные понятия, используемые при составлении тест-плана, и пример тест-плана приведены в [1, с. 211–220].

Особое внимание следует обратить на прямые и расчетные метрики, метрики покрытия.

Этап 3. Разработка чек-листов.

Чек-лист – это список проверок, которые необходимо выполнить для оценки качества определенного аспекта приложения. Чек-листы позволяют убедиться, что все необходимые проверки выполнены и ничего не упущено. Основные понятия, используемые при составлении чек-листа, и пример чек-листа приведены в [1, с. 115–119].

Этап 4. Разработка тест-кейсов.

Тест-кейс – это документ, который описывает последовательность действий, которые необходимо выполнить для проверки определенной функциональности. Тест-кейсы должны быть четкими, подробными и легко воспроизводимыми. Основные понятия, используемые при составлении тест-кейса, и пример тест-кейса приведены в [1, с. 120–166].

Этап 5. Разработка тестовых наборов.

Тестовый набор – это группа тест-кейсов, объединенных по определенному критерию. Тестовые наборы позволяют организовать тестовые сценарии и упростить процесс их выполнения.

Тестовые наборы могут быть сгруппированы по следующим критериям:

– функциональность: тест-кейсы, проверяющие определенную функциональность приложения (например, регистрация пользователей, оформление заказов);

– тип тестирования: тест-кейсы, относящиеся к определенному типу тестирования (например, UI-тестирование, функциональное тестирование, тестирование производительности);

– приоритет: тест-кейсы, имеющие разный приоритет (например, критические, важные, второстепенные);

– компонент: тест-кейсы, проверяющие определенный компонент приложения (например, главная страница, страница товара, форма заказа).

Этап 6. Создание отчетов об ошибках.

Отчет об ошибке – это документ, который описывает обнаруженную ошибку и предоставляет информацию, необходимую для ее исправления. Отчет об ошибке должен быть четким, подробным и легко понятным. Основные понятия, используемые при составлении отчета об ошибках, и пример отчета об ошибках приведены в [1, с. 167–208].

Этап 7. Анализ результатов тестирования и подготовка отчета.

После выполнения всех тестовых процедур необходимо проанализировать результаты тестирования и подготовить отчет. Основные понятия, используемые при составлении отчета о результатах тестирования, и пример отчета приведены в [1, с. 220–227].

Задание

Разработать для приложения по варианту индивидуального задания следующие документы:

– план тестирования;

– тест-кейсы, тестовые наборы, чек-листы для тестирования UI-элементов (поля ввода, кнопки, выпадающие списки, чекбоксы, радиокнопки), форм, навигации, адаптивности, доступности;

– отчеты об ошибках;

– отчет о результатах тестирования UI.

Содержание отчета: тема и цель работы; план тестирования, тест-кейсы, тестовые наборы, чек-листы для тестирования UI-элементов, форм, навигации, адаптивности, доступности, отчеты об ошибках, отчет о результатах тестирования UI.

Контрольные вопросы

1 В чем разница между чек-листом и тест-кейсом?
2 Какие разделы обязательно должны быть в тест-плане?
3 Как определить критерии завершения тестирования?
4 Перечислите базовые проверки, применяемые для поля ввода данных.
5 Перечислите базовые проверки для поля загрузки файлов.
6 Перечислите базовые проверки, применяемые для ввода даты.
7 Перечислите базовые проверки, применяемые для поля со списком.
8 Перечислите базовые проверки, применяемые для радиокнопки.
9 Перечислите базовые проверки для чекбоксов и радиокнопок.
10 Перечислите базовые проверки, применяемые для меню.
11 Перечислите базовые проверки, применяемые для таблиц.
12 Как составить тест-кейс для формы регистрации?
13 Какие негативные сценарии нужно проверить в форме входа?
14 Какие разделы должны быть в отчете о результатах тестирования?

4 Лабораторная работа № 4. Тестирование web-приложения

Цель работы: обеспечить качество web-приложения путем проверки функциональности, юзабилити, производительности, безопасности, совместимости.

4.1 Теоретические положения

В данной работе необходимо обеспечить качество веб-приложения за счет проверки:

– функциональности – корректности работы всех функций;

– юзабилити – удобства интерфейса для пользователей (адаптивность, доступность);

– производительности – скорости загрузки;

– безопасности – защиты от уязвимостей;

– кросс-браузерной совместимости – работы в различных браузерах и на различных устройствах.

Начальный этап проверки – анализ требований, в ходе которого изучаются техническое задание и пользовательские сценарии, выявляются ключевые функции для тестирования (например: авторизация, оплата, фильтры), определяются приоритеты тестирования.

В ходе планирования тестирования определяются проводимые виды тестирования (функциональное, UI, нагрузочное и др.), окружение (браузеры (Chrome, Firefox, Safari), ОС (Windows, macOS, iOS, Android)), составляются чек-листы и тест-кейсы.

В ходе тест-дизайна с помощью различных техник создаются сценарии позитивные (валидные данные, например, корректный логин/пароль) и негативные (невалидные данные, например, пустой пароль).

В ходе функционального тестирования проверяются работоспособность форм (например, регистрация, вход, заказ), корректность API-запросов (интеграция с backend), навигация (ссылки, кнопки, переходы между страницами).

В процессе UI-тестирования проверяются адаптивность верстки (мобильные/десктопные версии), соответствие дизайну (цвета, шрифты, расположение элементов), удобство интерфейса (интуитивность навигации).

При кросс-браузерном и кросс-платформенном тестировании проверяются отображение и функционал в разных браузерах (например, Chrome, Firefox, Edge, Safari), работа на мобильных устройствах (iOS, Android).

В ходе тестирования безопасности проверяются уязвимости (SQL-инъекции, XSS, CSRF), защита данных (HTTPS, шифрование паролей), доступ по ролям (пользователь/админ).

При нагрузочном тестировании проверяются скорость загрузки страниц, стабильность при высокой нагрузке (1000+ пользователей).

Рассмотрим, как проводить тестирование на примере главной страницы Википедии (https://ru.wikipedia.org). В качестве инструмента будет использоваться Playwright – современный фреймворк для автоматизации E2E-тестов.

На этапе планирования тестирования нужно определить:

– что тестируется – главная страница Википедии, поиск, навигация;

– какие браузеры и ОС (Chrome, Firefox, Windows);

– какие виды тестирования применяются (функциональное, UI, кросс-браузерное, безопасности);

– применяемые инструменты – Python и Playwright.

На этапе тест-дизайна выполняется создание сценариев, при этом предусматривается написание:

– позитивных тестов (валидные данные):

– поиск существующей статьи → должна открыться страница с результатом поиска;

– клик на «случайная статья» → должна загрузиться случайная страница Википедии;

– переход в раздел «история» → должна открыться страница истории выполненных правок;

– негативных тестов (невалидные данные):

– поиск пустой строки → должно появиться сообщение об ошибке;

– поиск спецсимволов (например, !@#$) → должна быть обработка некорректного ввода;

– ввод SQL-инъекции (' OR 1=1 --) → должна быть реализована защита от инъекций.

Playwright – это инструмент для комплексного тестирования современных веб-приложений. Установим Playwright через pip (менеджер пакетов Python): pip install playwright. После этого нужно установить браузеры для тестирования: playwright install. Эта команда скачает Chrome, Firefox и WebKit.

Создадим файл test_wikipedia.py и напишем в нем несколько тестов.

```
# Проверка загрузки страницы
from playwright.sync_api import sync_playwright
def test_page_load():
    with sync_playwright() as p:
        # Запускаем браузер (по умолчанию – Chromium)
        browser = p.chromium.launch(headless=False)  # headless=False – чтобы видеть браузер
        page = browser.new_page()
        # Переходим на Википедию
        page.goto("https://ru.wikipedia.org")
        # Проверяем, что заголовок страницы содержит "Википедия"
        assert "Википедия" in page.title()
        # Закрываем браузер
        browser.close()
test_page_load()
```

Запустим тест, для чего выполним инструкцию python test_wikipedia.py. Если все работает, браузер откроется, загрузит Википедию и закроется без ошибок.

Добавим тест, который проверяет работу поиска:

```
def test_search():
    with sync_playwright() as p:
        browser = p.chromium.launch(headless=False)
        page = browser.new_page()
        page.goto("https://ru.wikipedia.org")
        # Находим поле поиска (по CSS-селектору) и вводим запрос
        search_input = page.locator("#searchInput")
        search_input.fill("Python")
        # Нажимаем кнопку "Поиск"
        search_button = page.locator("#searchButton")
        search_button.click()
        # Проверяем, что перешли на страницу с результатами
        assert "Python" in page.title()
        browser.close()
```

```
test_search()
```

Проверим, что можно перейти в раздел "Случайная статья":

```
def test_random_article():
    with sync_playwright() as p:
        browser = p.chromium.launch(headless=False)
        page = browser.new_page()
        page.goto("https://ru.wikipedia.org")
        # Кликаем на ссылку "Случайная статья"
        random_link = page.locator("text=Случайная статья")
        random_link.click()
        # Ждем загрузки новой страницы
        page.wait_for_load_state("networkidle")
        # Проверяем, что заголовок страницы не пустой
        assert page.title() != ""
        browser.close()
test_random_article()
```

Playwright поддерживает несколько браузеров. Изменим код, чтобы тесты запускались в Chrome и Firefox:

```
browsers = ["chromium", "firefox"]
for browser_type in browsers:
    def run_tests():
        with sync_playwright() as p:
            browser = getattr(p, browser_type).launch(headless=False)
            page = browser.new_page()
            page.goto("https://ru.wikipedia.org")
            print(f"Тестирование в {browser_type}:")
            print(f"Заголовок: {page.title()}")
            browser.close()
    run_tests()
```

Playwright позволяет эмулировать мобильные устройства:

```
def test_mobile_responsive():
    with sync_playwright() as p:
        # Эмулируем iPhone 11
        iphone = p.devices["iPhone 11"]
        browser = p.chromium.launch(headless=False)
        context = browser.new_context(**iphone)
        page = context.new_page()
        page.goto("https://ru.wikipedia.org")
        # Проверяем, что мобильное меню отображается
```

```
        assert page.locator(".main-menu-button").is_visible()
        context.close()
```

Для проверки соответствия дизайну можно делать скриншоты и сравнивать с эталоном:

```
def test_design_layout():
    with sync_playwright() as p:
        browser = p.chromium.launch(headless=False)
        page = browser.new_page()
        page.goto("https://ru.wikipedia.org")
        # Делаем скриншот и сохраняем
        page.screenshot(path="wikipedia_layout.png")
        # Проверяем цвет элемента (например, синий заголовок)
        header_color = page.locator(".main-heading").evaluate("el => getComputedStyle(el).color")
        assert header_color == "rgb(6, 69, 173)"  # Синий цвет Википедии
        browser.close()
```

Выполним кросс-браузерное тестирование:

```
# Определяем список браузеров, которые будем тестировать
browsers = ["chromium", "firefox"]
# Перебираем каждый браузер из списка
for browser_type in browsers:
    # Определяем функцию для тестирования в текущем браузере
    def test_cross_browser():
        # Используем sync_playwright для синхронной работы с браузерами
        with sync_playwright() as p:
            # Получаем доступ к нужному браузеру через getattr
            # Например, если browser_type="chromium", то получим p.chromium
            # Запускаем браузер с видимым интерфейсом (headless=False)
            browser = getattr(p, browser_type).launch(headless=False)
            # Создаем новую вкладку/страницу в браузере
            page = browser.new_page()
            # Переходим на главную страницу Википедии
            page.goto("https://ru.wikipedia.org")
            # Проверяем, что элемент с классом .logo (логотип Википедии) виден на странице
            # Если элемент не виден, assert вызовет ошибку AssertionError
            assert page.locator(".logo").is_visible()
            # Закрываем браузер после завершения теста
            browser.close()
        # Вызываем функцию тестирования для текущего браузера
    test_cross_browser()
```

Напишем функцию для тестирования базовой защиты от XSS-атак:

```
def test_xss_protection():
# применим контекстный менеджер sync_playwright для работы с браузером
    with sync_playwright() as playwright:
        # Запускаем браузер Chromium в видимом режиме (headless=False)
        browser = playwright.chromium.launch(headless=False)
        # Создаем новую страницу (вкладку) в браузере
        page = browser.new_page()
        # Переходим на главную страницу Википедии
        page.goto("https://ru.wikipedia.org")
        # Вводим XSS-инъекцию в поисковую строку:
        # <script>alert(1)</script> - это классическая тестовая XSS-атака,
        # которая при уязвимости сайта выведет alert-окно
        page.fill("#searchInput", "<script>alert(1)</script>")
        # Нажимаем кнопку поиска
        page.click("#searchButton")
        # Проверяем защиту от XSS:
        # 1. Получаем весь HTML-код страницы через page.content()
        # 2. Проверяем, что строка "alert" не содержится в коде страницы
        # Это означает, что:
        # - либо скрипт был санитизирован (очищен)
        # - либо выполнение скриптов заблокировано
        # - либо сайт корректно обработал недопустимые символы
        assert "alert" not in page.content(), "Обнаружена уязвимость XSS!"
        # Закрываем браузер
        browser.close()
```

Выполним проверку скорости загрузки страницы в Playwright.

```
from playwright.sync_api import sync_playwright
import time
def test_page_load_performance():
    """Тест для измерения скорости загрузки страницы и анализа метрик производительности"""
    with sync_playwright() as p:
        # Инициализируем браузер (можно использовать chromium, firefox или webkit)
        browser = p.chromium.launch(headless=True)  # headless=True для ускорения тестов
        # Создаем новый контекст и страницу
        context = browser.new_context()
        page = context.new_page()
        # 1. Измерение общего времени загрузки
```

```
        start_time = time.time()
        # Переходим на тестируемую страницу с ожиданием полной загрузки
        response = page.goto(
            "https://example.com",
            wait_until="networkidle"  # Ожидаем завершения сетевых запросов
        )
        # Вычисляем общее время загрузки
        load_time = time.time() - start_time
        print(f"\nОбщее время загрузки: {load_time:.2f} секунд")
        # 2. Получаем метрики производительности из браузера
        metrics = page.evaluate("""() => {
            const timing = performance.timing;
            return {
                // Время от начала навигации до полной загрузки DOM
                domComplete: timing.domComplete - timing.navigationStart,
                // Время от начала навигации до полной загрузки страницы
                loadEventEnd: timing.loadEventEnd - timing.navigationStart,
                // Время от начала навигации до первого байта (TTFB)
                ttfb: timing.responseStart - timing.navigationStart
            };
        }""")
        print("\nДетальные метрики производительности:")
        print(f"- DOM загружен за: {metrics['domComplete']} мс")
        print(f"- Полная загрузка страницы: {metrics['loadEventEnd']} мс")
        print(f"- Time to First Byte (TTFB): {metrics['ttfb']} мс")
        # 3. Проверяем: время загрузки не превышает допустимых пределов
        MAX_LOAD_TIME = 3000  # 3 секунды - максимально допустимое
                              # время загрузки в мс
        assert metrics['loadEventEnd'] <= MAX_LOAD_TIME, (
         f"Страница загружается слишком долго: {metrics['loadEventEnd']} мс"
         f"(максимум {MAX_LOAD_TIME} мс)"
        )
        # 4. Анализ сетевых запросов
        print("\nАнализ сетевых запросов:")
        for request in page.request_log():
            if request.timing:
                print(f"Запрос: {request.url}")
print(f"- Время: {request.timing['responseEnd'] - request.timing['requestTime']} мс")
        # Закрываем браузер
        context.close()
        browser.close()
    # Запускаем тест
    test_page_load_performance()
```

Если тесты прошли без ошибок – значит, основные функции Википедии работают. Если есть ошибки (например, AssertionError), нужно:

– проверить, правильно ли указаны селекторы;

– убедиться, что сайт доступен;

– проверить код на опечатки.

Задание

Настроить окружение для тестирования (Python + Playwright).

Написать автоматизированные тесты для проверки функциональности, юзабилити, производительности, безопасности, совместимости веб-приложения по варианту индивидуального задания.

Содержание отчета: тема и цель работы; тест-план, скриншоты кода в Playwright для проверки основных функций приложения, отчет о результатах тестирования, который содержит описание найденных багов (шаги воспроизведения, скриншоты), рекомендации по доработкам, метрики качества (количество дефектов, покрытие тестами).

Контрольные вопросы

1 Какие виды тестирования можно автоматизировать с помощью Playwright?

2 Как проверить, что элемент присутствует на странице?

3 Как проверить адаптивность сайта на разных устройствах с помощью Playwright?

4 Какие элементы нужно тестировать в многошаговой форме (например, регистрация на сайте)?

5 Какие параметры верстки проверяют при UI-тестировании?

6 Как эмулировать мобильное устройство в Playwright?

7 Как определить, является ли найденный баг критическим?

8 Какие метрики важны для оценки качества тестирования?

5 Лабораторная работа № 5. Автоматизированное тестирование

Цель работы: изучить методы автоматизированного тестирования REST API с использованием Postman.

5.1 Теоретические положения

REST (Representational State Transfer) – это архитектурный стиль для распределенных систем, который широко используется при разработке веб-API. Основные принципы REST включают: четкое разделение между клиентом и сервером; отсутствие состояния (stateless): каждый запрос содержит всю необходимую информацию; ответы могут быть кэшированы.

В REST API используются стандартные HTTP-методы: GET – получение ресурса, POST – создание ресурса, PUT – полное обновление ресурса, PATCH – частичное обновление ресурса, DELETE – удаление ресурса.

Важные HTTP-статусы: 200 OK – успешный запрос, 201 Created – ресурс создан, 400 Bad Request – неверный запрос, 401 Unauthorized – требуется аутентификация, 403 Forbidden – доступ запрещен, 404 Not Found – ресурс не найден, 500 Internal Server Error – ошибка сервера.

Postman – это мощная платформа для работы с API, которая позволяет: отправлять HTTP-запросы различных типов, сохранять историю запросов, организовывать запросы в коллекции, писать автоматизированные тесты, генерировать документацию, выполнять мониторинг API.

Основные компоненты Postman: коллекции – группы связанных запросов, окружения – наборы переменных, тестовые скрипты – код JavaScript для проверки ответов, предварительные скрипты – код, выполняемый перед запросом, а также переменные для хранения данных.

При выполнении лабораторной работы необходимо скачать установщик Postman с официального сайта https://www.postman.com/downloads/. После установки запустить Postman и создать учетную запись.

Для создания GET-запроса к тестовому API JSONPlaceholder следует:

– нажать кнопку «New» в левом верхнем углу;

– выбрать «Request»;

– ввести имя запроса «Get Posts» и создать новую коллекцию «Lab Work»;

– в поле URL ввести: https://jsonplaceholder.typicode.com/posts;

– убедиться, что выбран метод GET;

– нажать кнопку «Send», после чего можно увидеть ответ от сервера в формате JSON со списком постов. Следует обратить внимание на статус код 200 OK в верхней части ответа.

Теперь добавим простую проверку к этому запросу, для чего следует перейти на вкладку «Tests» и ввести следующий код:

```
// Проверка статус кода
pm.test("Status code is 200", function() {
    pm.response.to.have.status(200);
});
// Проверка времени ответа
pm.test("Response time is less than 200ms", function() {
    pm.expect(pm.response.responseTime).to.be.below(200);
});
// Проверка структуры ответа
pm.test("Response has expected structure", function() {
    const jsonData = pm.response.json();
    pm.expect(jsonData).to.be.an('array');
    pm.expect(jsonData[0]).to.have.property('userId');
    pm.expect(jsonData[0]).to.have.property('id');
    pm.expect(jsonData[0]).to.have.property('title');
```

```
    pm.expect(jsonData[0]).to.have.property('body');
});
```

После отправки запроса следует перейти на вкладку «Test Results», чтобы увидеть результаты выполнения тестов.

Разберем этот код подробно:

– pm.test() – функция для создания теста, принимает два параметра: строку с описанием теста и функцию, содержащую проверки;

– pm.response – объект, содержащий информацию об ответе сервера: to.have.status() – проверка статус кода, responseTime – время ответа в миллисекундах;

– pm.response.json() – парсит тело ответа как JSON;

– pm.expect() – функция для утверждений (аналогично assert): to.be.an() – проверка типа, to.have.property() – проверка наличия свойства.

Postman поддерживает несколько типов переменных:

– глобальные – доступны во всех запросах;
– окружения – зависят от выбранного окружения;
– коллекции – доступны только в рамках коллекции;
– локальные – только в рамках одного запроса.

Создадим переменную для базового URL, для чего следует:

– нажать на кнопку «Environments» в левой панели;
– создать новое окружение с именем «Lab Environment»;
– добавить переменную «base_url» со значением «https://jsonplaceholder.typicode.com»;
– активировать это окружение, выбрав его из выпадающего списка в правом верхнем углу.

Теперь модифицируем наш запрос, для чего изменим URL на: {{base_url}}/posts и отправим запрос – он должен работать, как прежде.

Создадим новый POST-запрос для добавления поста, для чего следует:

– создать новый запрос в коллекции «Lab Work»;
– назвать его «Create Post»;
– выбрать метод POST;
– установить URL: {{base_url}}/posts;
– перейти на вкладку «Body»;
– выбрать «raw» и «JSON» из выпадающих списков;
– ввести тело запроса:

```
{
  "title": "My test post",
  "body": "This is a test post created from Postman",
  "userId": 1
}
```

Добавим следующие тесты.

```
// Проверка статус кода
pm.test("Status code is 201", function() {
    pm.response.to.have.status(201);
});

// Проверка структуры ответа
pm.test("Response contains created post", function() {
    const jsonData = pm.response.json();
    pm.expect(jsonData).to.have.property('id');
    pm.expect(jsonData.title).to.eql("My test post");
    pm.expect(jsonData.body).to.eql("This is a test post created from Postman");
    pm.expect(jsonData.userId).to.eql(1);
});

// Сохраняем ID созданного поста в переменную
const jsonData = pm.response.json();
pm.environment.set("created_post_id", jsonData.id);
```

После отправки запроса следует проверить результаты тестов и убедиться, что переменная created_post_id установлена.

Часто тесты требуют выполнения нескольких запросов последовательно. Создадим тест, который реализует цепочку запросов следующим образом: создает пост (POST), получает созданный пост (GET), обновляет пост (PUT), удаляет пост (DELETE).

Создадим новый запрос для получения поста, для чего следует:

- создать новый GET запрос «Get Created Post»;
- установить URL: {{base_url}}/posts/{{created_post_id}};
- добавить тесты.

```
pm.test("Status code is 200", function() {
    pm.response.to.have.status(200);
});

pm.test("Post matches created data", function() {
    const jsonData = pm.response.json();
    pm.expect(jsonData.id).to.eql(pm.environment.get("created_post_id"));
    pm.expect(jsonData.title).to.eql("My test post");
});
```

Теперь создадим запрос для обновления поста, для чего следует:

- создать новый PUT-запрос «Update Post»;
- установить URL: {{base_url}}/posts/{{created_post_id}};
- в теле запроса указать

```
{
    "id": {{created_post_id}},
```

```
    "title": "Updated post title",
    "body": "Updated post body",
    "userId": 1
}
```

– добавить тесты

```
pm.test("Status code is 200", function() {
    pm.response.to.have.status(200);
});

pm.test("Post was updated", function() {
    const jsonData = pm.response.json();
    pm.expect(jsonData.title).to.eql("Updated post title");
    pm.expect(jsonData.body).to.eql("Updated post body");
});
```

Наконец, создадим запрос для удаления поста, для чего следует:

– создать новый DELETE запрос «Delete Post»;
– установить URL: {{base_url}}/posts/{{created_post_id}};
– добавить тесты

```
pm.test("Status code is 200", function() {
    pm.response.to.have.status(200);
});

pm.test("Post is really deleted", function() {
    // Отправляем запрос на получение удаленного поста
    pm.sendRequest({
        url: pm.environment.get("base_url") + "/posts/" + pm.environment.get("created_post_id"),
        method: 'GET'
    }, function (err, response) {
        pm.expect(response.code).to.eql(404);
    });
});
```

Postman позволяет запускать все запросы в коллекции последовательно, для чего нужно нажать на кнопку «Runner» в левом верхнем углу, выбрать коллекцию «Lab Work», выбрать окружение «Lab Environment», нажать «Start Lab Work».

Задание

Необходимо в Postman создать коллекцию, окружение с переменными, реализовать запросы GET, POST, PUT, DELETE. Для каждого запроса написать не менее трех тестов. Реализовать цепочку запросов (создание → чтение → обновление → удаление) и запустить коллекцию через Runner.

Содержание отчета: тема и цель работы; прокомментированный код выполнения задания.

Контрольные вопросы

1 Что такое REST API и каковы его основные принципы? Какие HTTP-методы используются в REST API и для чего?

2 Как создать и отправить GET-запрос в Postman?

3 Как написать тест для проверки статуса кода в Postman?

4 Какие виды переменных поддерживает Postman и как их использовать?

5 Как создать POST-запрос с телом в формате JSON?

6 Как запустить всю коллекцию тестов в Postman?

Список литературы

1 **Куликов, С. С.** Тестирование программного обеспечения. Базовый курс / С. С. Куликов. – Минск : Четыре четверти, 2023. – 303 с. – URL: http://svyatoslav.biz/software_testing_book/ (дата обращения: 29.04.2025).

2 **Проскуряков, А. В.** Качество и тестирование программного обеспечения. Метрология программного обеспечения : учеб. пособие / А. В. Проскуряков ; Южный федеральный университет. – Ростов-н/Д ; Таганрог : Изд-во Южного федерального университета, 2022. – 197 с. – URL: https://znanium.com/catalog/product/2057599 (дата обращения: 29.04.2025).

3 **Хориков, В.** Принципы юнит-тестирования / В. Хориков. – СПб. : Питер, 2021. – 320 с.: ил.

4 **Орлов, С. А.** Программная инженерия. Стандарт третьего поколения : учебник для вузов / С. А. Орлов – СПб. : Питер, 2016. – 640 с.