Geekbrains

Ручное и автоматизированное тестирование веб-приложения “Triangle”: исследовательский подход.

Инженер по тестированию
Ночевной Сергей Алексеевич

Анапа
2023

Дипломный проект “Ручное и автоматизированное тестирование веб-приложения “Triangle”: исследовательский подход.”

Содержание

Введение

С развитием информационных технологий и веб-приложений современное общество сталкивается с неизбежной потребностью обеспечения высокого качества программного обеспечения. Это особенно важно в сферах, где недостаточное качество приложения может иметь серьезные последствия. В рамках данной дипломной работы представляется исследование методов тестирования веб-приложения, разработанного для работы с геометрическими фигурами, а именно треугольниками.
Приложение "Triangle" представляет собой необычный, но информативный вариант калькулятора, который позволяет пользователю ввести значения трех сторон треугольника и получить информацию о типе этого треугольника (равносторонний, равнобедренный, разносторонний). Несмотря на свою простоту, данное веб-приложение предоставляет возможность исследования методов тестирования, которые могут быть применены к более сложным приложениям.

Цель данной работы заключается в исследовании различных методов тестирования веб-приложения "Triangle", включая как ручные, так и автоматизированные подходы. Выявить не только ожидаемые, но и неожиданные аспекты функциональности приложения, проверить его на корректность и надежность, а также закрепить полученные знания на этапе обучения.

Тестирование будет проведено с использованием исследовательского подхода, который позволяет выявить как стандартные функциональные черты приложения, так и его реакцию на разнообразные входные данные и сценарии.

Для автоматизации тестирования будут использованы следующие основные инструменты:
Python: в качестве основного языка программирования.
PyTest: для написания и выполнения автоматизированных тестов.
Selenium WebDriver: для взаимодействия с веб-интерфейсом приложения.
Visual Studio Code и DevTools: для разработки и отладки автотестов.

Ручное тестирование будет проведено с акцентом на интерактивности и надежности приложения, а также выявлении потенциальных слабых мест и аномалий.

В ходе ручного тестирования будут применены чек-листы, но они не будут ограничивать процесс тестирования конкретными шагами и ожиданиями. Вместо этого, чек-листы могут включать в себя общие аспекты и критерии, на которые будет обращено внимание при исследовании приложения.

Этот подход позволяет тестировщику сохранить гибкость в процессе тестирования, освободив его от чрезмерной жесткости в структурированных чек-листах и позволяя более полноценно исследовать приложение в поисках как стандартных, так и неожиданных аспектов функциональности.

Также во время ручного тестирования приложения "Triangle" будут выявлены особенности его поведения и функциональности. Это позволит более грамотно и надежно разрабатывать автоматизированные тесты. Ручное тестирование поможет более точно определить, какие аспекты приложения следует покрыть автотестами, и какие критически важные сценарии использования следует учесть.

Таким образом, результаты ручного тестирования не только помогут в поиске потенциальных проблем и аномалий в приложении, но и обеспечат более качественное и полное автоматизированное тестирование. Это значительно повысит надежность и стабильность приложения "Triangle", а также позволит закрепить полученные навыки на практике.

Данный документ представляет собой обзор исследования, охватывая как теоретические, так и практические аспекты тестирования веб-приложения "Triangle" с акцентом на исследовательском подходе.

Глава 1. Теоретическая часть.

1.1 Фундаментальная теория тестирования.

Основные определения теории тестирования.

Тестирование программного обеспечения — проверка соответствия реальных и ожидаемых результатов поведения программы, проводимая на конечном наборе тестов, выбранном определенным образом.

Цель тестирования — проверка соответствия ПО предъявляемым требованиям, обеспечение уверенности в качестве ПО, поиск очевидных ошибок в программном обеспечении, которые должны быть выявлены до того, как их обнаружат пользователи программы.

Для чего проводится тестирование ПО?

- Для проверки соответствия требованиям.
- Для обнаружение проблем на более ранних этапах разработки и предотвращение повышения стоимости продукта.
- Обнаружение вариантов использования, которые не были предусмотрены при разработке. А также взгляд на продукт со стороны пользователя.
- Повышение лояльности к компании и продукту, т.к. любой обнаруженный дефект негативно влияет на доверие пользователей.

Принципы тестирования

- Принцип 1 — Тестирование демонстрирует наличие дефектов.
  Тестирование только снижает вероятность наличия дефектов, которые находятся в программном обеспечении, но не гарантирует их отсутствия.
- Принцип 2 — Исчерпывающее тестирование невозможно.
  Полное тестирование с использованием всех входных комбинаций данных, результатов и предусловий физически невыполнимо (исключение — тривиальные случаи).
- Принцип 3 — Раннее тестирование
  Следует начинать тестирование на ранних стадиях жизненного цикла разработки ПО, чтобы найти дефекты как можно раньше.

- Принцип 4 — Скопление дефектов.

  Большая часть дефектов находится в ограниченном количестве модулей.
- Принцип 5 — Парадокс пестицида.

  Если повторять те же тестовые сценарии снова и снова, в какой-то момент этот набор тестов перестанет выявлять новые дефекты.
- Принцип 6 — Тестирование зависит от контекста.

  Тестирование проводится по-разному в зависимости от контекста. Например, программное обеспечение, в котором критически важна безопасность, тестируется иначе, чем новостной портал.
- Принцип 7 — Заблуждение об отсутствии ошибок.

  Отсутствие найденных дефектов при тестировании не всегда означает готовность продукта к релизу. Система должна быть удобна пользователю в использовании и удовлетворять его ожиданиям и потребностям.

Обеспечение качества (QA — Quality Assurance) и контроль качества (QC — Quality Control) — эти термины похожи на взаимозаменяемые, но разница между обеспечением качества и контролем качества все-таки есть, хоть на практике процессы и имеют некоторую схожесть.
QC (Quality Control) — Контроль качества продукта — анализ результатов тестирования и качества новых версий выпускаемого продукта.

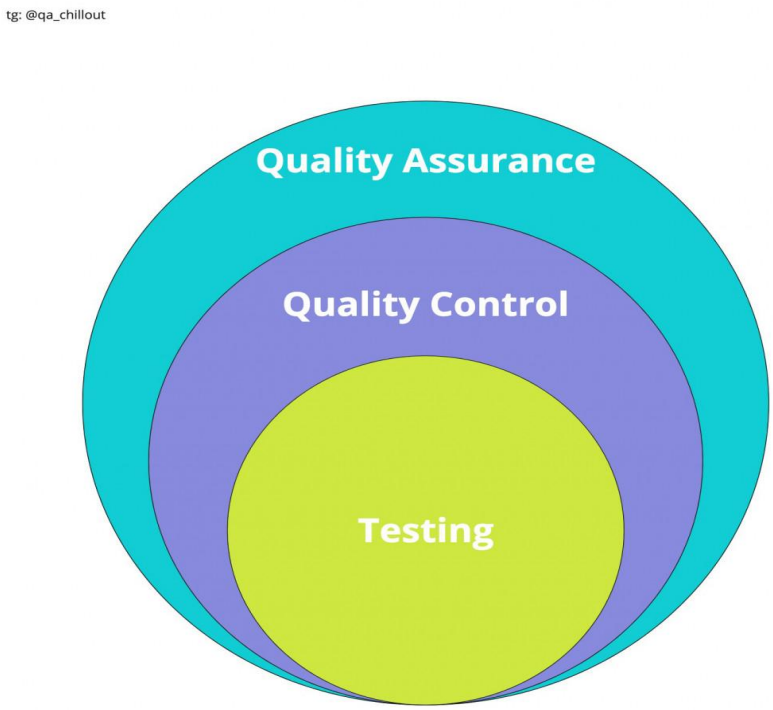
К задачам контроля качества относятся:

- проверка готовности ПО к релизу;
- проверка соответствия требований и качества данного проекта.

QA (Quality Assurance) — Обеспечение качества продукта — изучение возможностей по изменению и улучшению процесса разработки, улучшению коммуникаций в команде, где тестирование является только одним из аспектов обеспечения качества.

К задачам обеспечения качества относятся:

- проверка технических характеристик и требований к ПО;
- оценка рисков;
- планирование задач для улучшения качества продукции;
- подготовка документации, тестового окружения и данных;
- тестирование;
- анализ результатов тестирования, а также составление отчетов и других документов.



Верификация и валидация — два понятия тесно связаны с процессами тестирования и обеспечения качества. К сожалению, их часто путают, хотя отличия между ними достаточно существенны.

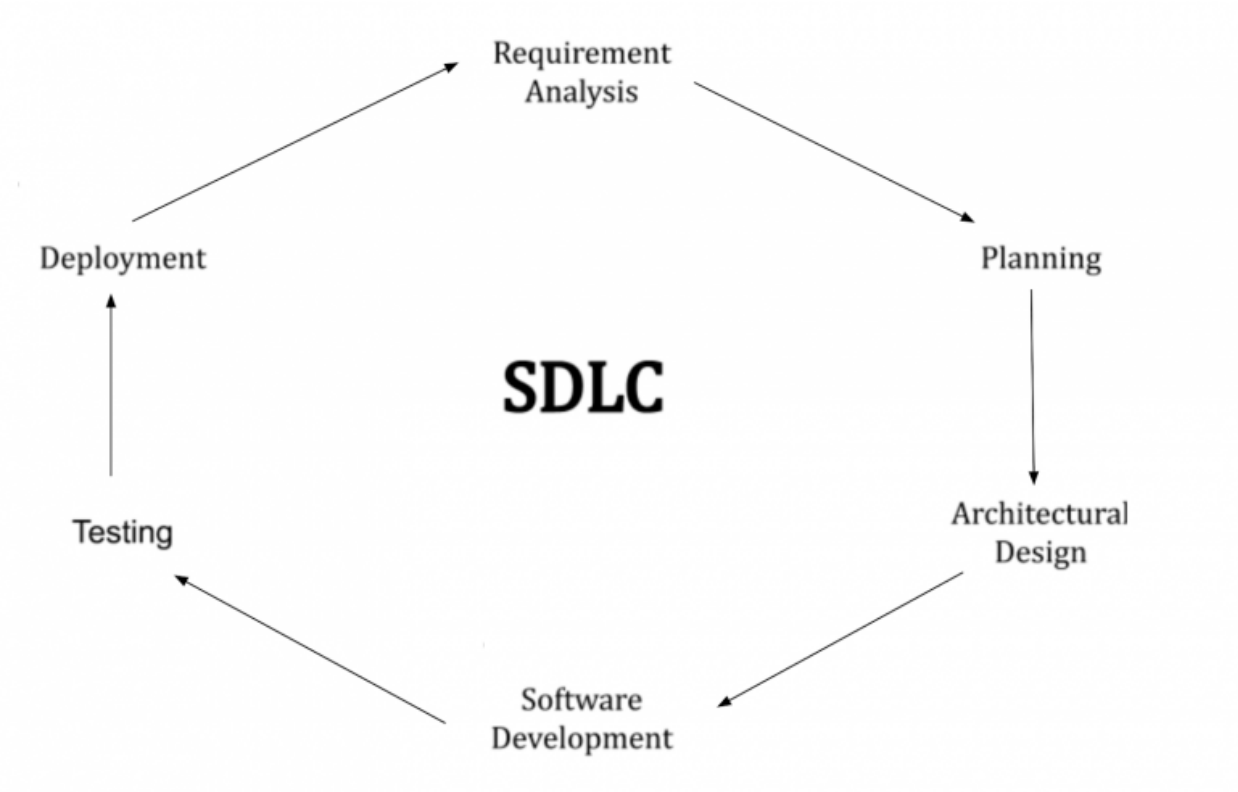
Верификация — это процесс оценки системы, чтобы понять, удовлетворяют ли результаты текущего этапа разработки условиям, которые были сформулированы в его начале.

Валидация — это определение соответствия разрабатываемого ПО ожиданиям и потребностям пользователя, его требованиям к системе.

Этапы тестирования:

1. Анализ продукта
2. Работа с требованиями
3. Разработка стратегии тестирования и планирование процедур контроля качества
4. Создание тестовой документации
5. Тестирование прототипа
6. Основное тестирование
7. Стабилизация
8. Эксплуатация

Стадии разработки ПО — этапы, которые проходят команды разработчиков ПО, прежде чем программа станет доступной для широкого круга пользователей.



Программный продукт проходит следующие стадии:

1. анализ требований к проекту;
2. проектирование;
3. реализация;
4. тестирование продукта;
5. внедрение и поддержка.

Требования — это спецификация (описание) того, что должно быть реализовано.
Требования описывают то, что необходимо реализовать, без детализации технической стороны решения.

Атрибуты требований:

1. Корректность — точное описание разрабатываемого функционала.
2. Проверяемость — формулировка требований таким образом, чтобы можно было выставить однозначный вердикт, выполнено все в соответствии с требованиями или нет.
3. Полнота — в требовании должна содержаться вся необходимая для реализации функциональности информация.
4. Недвусмысленность — требование должно содержать однозначные формулировки.

5. Непротиворечивость — требование не должно содержать внутренних противоречий и противоречий другим требованиям и документам.
6. Приоритетность — у каждого требования должен быть приоритет(количественная оценка степени значимости требования). Этот атрибут позволит грамотно управлять ресурсами на проекте.
7. Атомарность — требование нельзя разбить на отдельные части без потери деталей.
8. Модифицируемость — в каждое требование можно внести изменение.
9. Прослеживаемость — каждое требование должно иметь уникальный идентификатор, по которому на него можно сослаться.

1.1.2 Что такое баг. Его жизненный цикл.

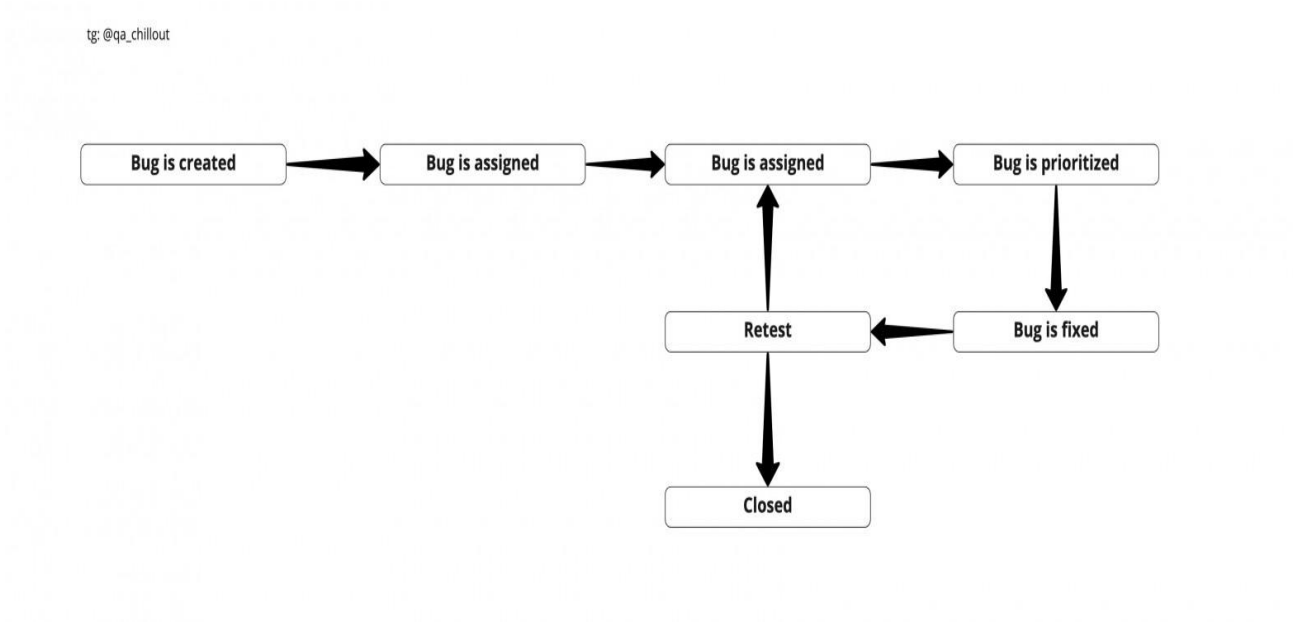
Дефект (bug) — отклонение фактического результата от ожидаемого.
Отчёт о дефекте (bug report) — документ, который содержит отчет о любом недостатке в компоненте или системе, который потенциально может привести компонент или систему к невозможности выполнить требуемую функцию.

Атрибуты отчета о дефекте:

1. Уникальный идентификатор (ID) — присваивается автоматически системой при создании баг-репорта.
2. Тема (краткое описание, Summary) — кратко сформулированный смысл дефекта, отвечающий на вопросы: Что? Где? Когда(при каких условиях)?
3. Подробное описание (Description) — более широкое описание дефекта (указывается опционально).
4. Шаги для воспроизведения (Steps To Reproduce) — описание четкой последовательности действий, которая привела к выявлению дефекта. В шагах воспроизведения должен быть описан каждый шаг, вплоть до конкретных вводимых значений, если они играют роль в воспроизведении дефекта.
5. Фактический результат (Actual result) — описывается поведение системы на момент обнаружения дефекта в ней. чаще всего, содержит краткое описание некорректного поведения(может совпадать с темой отчета о дефекте).
6. Ожидаемый результат (Expected result) — описание того, как именно должна работать система в соответствии с документацией.
7. Вложения (Attachments) — скриншоты, видео или лог-файлы.

8. Серьезность дефекта (важность, Severity) — характеризует влияние дефекта на работоспособность приложения.
9. Приоритет дефекта (срочность, Priority) — указывает на очерёдность выполнения задачи или устранения дефекта.
10. Статус (Status) — определяет текущее состояние дефекта. Статусы дефектов могут быть разными в разных баг-трекинговых системах.
11. Окружение (Environment) – окружение, на котором воспроизвелся баг.

Жизненный цикл бага



Severity vs Priority

Серьёзность (severity) показывает степень ущерба, который наносится проекту существованием дефекта. Severity выставляется тестировщиком.

Градация Серьезности дефекта (Severity):

- Блокирующий (S1 – Blocker)

  тестирование значительной части функциональности вообще недоступно. Блокирующая ошибка, приводящая приложение в нерабочее состояние, в результате которого дальнейшая работа с тестируемой системой или ее ключевыми функциями становится невозможна.
- Критический (S2 – Critical)

  критическая ошибка, неправильно работающая ключевая бизнес-логика, дыра в системе безопасности, проблема, приведшая к временному падению сервера или приводящая в нерабочее состояние некоторую часть системы, то есть не работает

важная часть одной какой-либо функции либо не работает значительная часть, но имеется workaround (обходной путь/другие входные точки), позволяющий продолжить тестирование.

- Значительный (S3 – Major)
  не работает важная часть одной какой-либо функции/бизнес-логики, но при выполнении специфических условий, либо есть workaround, позволяющий продолжить ее тестирование либо не работает не очень значительная часть какой-либо функции. Также относится к дефектам с высокими visibility – обычно не сильно влияющие на функциональность дефекты дизайна, которые, однако, сразу бросаются в глаза.
- Незначительный (S4 – Minor)
  часто ошибки GUI, которые не влияют на функциональность, но портят юзабилити или внешний вид. Также незначительные функциональные дефекты, либо которые воспроизводятся на определенном устройстве.
- Тривиальный (S5 – Trivial)
  почти всегда дефекты на GUI — опечатки в тексте, несоответствие шрифта и оттенка и т.п., либо плохо воспроизводимая ошибка, не касающаяся бизнес-логики, проблема сторонних библиотек или сервисов, проблема, не оказывающая никакого влияния на общее качество продукта.

Срочность (priority) показывает, как быстро дефект должен быть устранён. Priority выставляется менеджером, тимлидом или заказчиком

Градация Приоритета дефекта (Priority):

- P1 Высокий (High)
  Критическая для проекта ошибка. Должна быть исправлена как можно быстрее.
- P2 Средний (Medium)
  Не критичная для проекта ошибка, однако требует обязательного решения.
- P3 Низкий (Low)
  Наличие данной ошибки не является критичным и не требует срочного решения. Может быть исправлена, когда у команды появится время на ее устранение.

Существует шесть базовых типов задач:

- Эпик (epic) — большая задача, на решение которой команде нужно несколько спринтов.

- Требование (requirement) — задача, содержащая в себе описание реализации той или иной фичи.
- История (story) — часть большой задачи (эпика), которую команда может решить за 1 спринт.
- Задача (task) — техническая задача, которую делает один из членов команды.
- Под-задача (sub-task) — часть истории / задачи, которая описывает минимальный объем работы члена команды.
- Баг (bug) — задача, которая описывает ошибку в системе.

Тестовые среды

- Среда разработки (Development Env) – за данную среду отвечают разработчики, в ней они пишут код, проводят отладку, исправляют ошибки
- Среда тестирования (Test Env) – среда, в которой работают тестировщики (проверяют функционал, проводят smoke и регрессионные тесты, воспроизводят.
- Интеграционная среда (Integration Env) – среда, в которой проводят тестирование взаимодействующих друг с другом модулей, систем, продуктов.
- Предпрод (Preprod Env) – среда, которая максимально приближена к продакшену. Здесь проводится заключительное тестирование функционала.
- Продакшн среда (Production Env) – среда, в которой работают пользователи.

Основные фазы тестирования

- Pre-Alpha: прототип, в котором всё ещё присутствует много ошибок и наверняка неполный функционал. Необходим для ознакомления с будущими возможностями программ.
- Alpha: является ранней версией программного продукта, тестирование которой проводится внутри фирмы-разработчика.
- Beta: практически готовый продукт, который разработан в первую очередь для тестирования конечными пользователями.
- Release Candidate (RC): возможные ошибки в каждой из фичей уже устранены и разработчики выпускают версию на которой проводится регрессионное тестирование.
- Release: финальная версия программы, которая готова к использованию.

1.1.3 Основные виды тестирования ПО

Вид тестирования — это совокупность активностей, направленных на тестирование заданных характеристик системы или её части, основанная на конкретных целях.

1. Классификация по запуску кода на исполнение:
    - Статическое тестирование — процесс тестирования, который проводится для верификации практически любого артефакта разработки: программного кода компонент, требований, системных спецификаций, функциональных спецификаций, документов проектирования и архитектуры программных систем и их компонентов.
    - Динамическое тестирование — тестирование проводится на работающей системе, не может быть осуществлено без запуска программного кода приложения.
2. Классификация по доступу к коду и архитектуре:
    - Тестирование белого ящика — метод тестирования ПО, который предполагает полный доступ к коду проекта.
    - Тестирование серого ящика — метод тестирования ПО, который предполагает частичный доступ к коду проекта (комбинация White Box и Black Box методов).
    - Тестирование чёрного ящика — метод тестирования ПО, который не предполагает доступа (полного или частичного) к системе. Основывается на работе исключительно с внешним интерфейсом тестируемой системы.
3. Классификация по уровню детализации приложения:
    - Модульное тестирование — проводится для тестирования какого-либо одного логически выделенного и изолированного элемента (модуля) системы в коде. Проводится самими разработчиками, так как предполагает полный доступ к коду.
    - Интеграционное тестирование — тестирование, направленное на проверку корректности взаимодействия нескольких модулей, объединенных в единое целое.
    - Системное тестирование — процесс тестирования системы, на котором проводится не только функциональное тестирование, но и оценка характеристик качества системы — ее устойчивости, надежности, безопасности и производительности.

- Приёмочное тестирование — проверяет соответствие системы потребностям, требованиям и бизнес-процессам пользователя.

4. Классификация по степени автоматизации:
    - Ручное тестирование.
    - Автоматизированное тестирование.
5. Классификация по принципам работы с приложением
    - Позитивное тестирование — тестирование, при котором используются только корректные данные.
    - Негативное тестирование — тестирование приложения, при котором используются некорректные данные и выполняются некорректные операции.
6. Классификация по уровню функционального тестирования:
    - Дымовое тестирование (smoke test) — тестирование, выполняемое на новой сборке, с целью подтверждения того, что программное обеспечение стартует и выполняет основные для бизнеса функции.
    - Тестирование критического пути (critical path) — направлено для проверки функциональности, используемой обычными пользователями во время их повседневной деятельности.
    - Расширенное тестирование (extended) — направлено на исследование всей заявленной в требованиях функциональности.
7. Классификация в зависимости от исполнителей:
    - Альфа-тестирование — является ранней версией программного продукта. Может выполняться внутри организации-разработчика с возможным частичным привлечением конечных пользователей.
    - Бета-тестирование — программное обеспечение, выпускаемое для ограниченного количества пользователей. Главная цель — получить отзывы клиентов о продукте и внести соответствующие изменения.
8. Классификация в зависимости от целей тестирования:
    - Функциональное тестирование (functional testing) — направлено на проверку корректности работы функциональности приложения.
    - Нефункциональное тестирование (non-functional testing) — тестирование атрибутов компонента или системы, не относящихся к функциональности.

Тестирование производительности (performance testing) — определение стабильности и потребления ресурсов в условиях различных сценариев использования и нагрузок.
Нагрузочное тестирование (load testing) — определение или сбор показателей

производительности и времени отклика программно-технической системы или устройства в ответ на внешний запрос с целью установления соответствия требованиям, предъявляемым к данной системе (устройству).

Тестирование масштабируемости (scalability testing) — тестирование, которое измеряет производительность сети или системы, когда количество пользовательских запросов увеличивается или уменьшается.

Объёмное тестирование (volume testing) — это тип тестирования программного обеспечения, которое проводится для тестирования программного приложения с определенным объемом данных.

Стрессовое тестирование (stress testing) — тип тестирования направленный для проверки, как система обращается с нарастающей нагрузкой (количеством одновременных пользователей).

Инсталляционное тестирование (installation testing) — тестирование, направленное на проверку успешной установки и настройки, обновления или удаления приложения.

Тестирование интерфейса (GUI/UI testing) — проверка требований к пользовательскому интерфейсу.

Тестирование удобства использования (usability testing) — это метод тестирования, направленный на установление степени удобства использования, понятности и привлекательности для пользователей разрабатываемого продукта в контексте заданных условий.

Тестирование локализации (localization testing) — проверка адаптации программного обеспечения для определенной аудитории в соответствии с ее культурными особенностями.

Тестирование безопасности (security testing) — это стратегия тестирования, используемая для проверки безопасности системы, а также для анализа рисков, связанных с обеспечением целостного подхода к защите приложения, атак хакеров, вирусов, несанкционированного доступа к конфиденциальным данным.

Тестирование надёжности (reliability testing) — один из видов нефункционального тестирования ПО, целью которого является проверка работоспособности приложения при длительном тестировании с ожидаемым уровнем нагрузки.

Регрессионное тестирование (regression testing) — тестирование уже проверенной ранее функциональности после внесения изменений в код приложения, для уверенности в том, что эти изменения не внесли ошибки в областях, которые не подверглись изменениям.

Повторное/подтверждающее тестирование (re-testing/confirmation testing) — тестирование, во время которого исполняются тестовые сценарии, выявившие ошибки во время последнего запуска, для подтверждения успешности исправления этих ошибок.

1.1.4 Техники тест-дизайна и методы тестирования

Тест-дизайн — это этап тестирования ПО, на котором проектируются и создаются тестовые случаи (тест-кейсы).

Выделяет следующие техники тест-дизайна:

1. Тестирование на основе классов эквивалентности (equivalence partitioning) — это техника, основанная на методе чёрного ящика, при которой мы разделяем функционал (часто диапазон возможных вводимых значений) на группы эквивалентных по своему влиянию на систему значений.
2. Техника анализа граничных значений (boundary value testing) — это техника проверки поведения продукта на крайних (граничных) значениях входных данных.
3. Попарное тестирование (pairwise testing) — это техника формирования наборов тестовых данных из полного набора входных данных в системе, которая позволяет существенно сократить количество тест-кейсов.
4. Тестирование на основе состояний и переходов (State-Transition Testing) — применяется для фиксирования требований и описания дизайна приложения.
5. Таблицы принятия решений (Decision Table Testing) — техника тестирования, основанная на методе чёрного ящика, которая применяется для систем со сложной логикой.
6. Доменный анализ (Domain Analysis Testing) — это техника основана на разбиении диапазона возможных значений переменной на поддиапазоны, с последующим выбором одного или нескольких значений из каждого домена для тестирования.
7. Сценарий использования (Use Case Testing) — Use Case описывает сценарий взаимодействия двух и более участников (как правило — пользователя и системы).

Методы тестирования

Тестирование белого ящика — метод тестирования ПО, который предполагает, что внутренняя структура/устройство/реализация системы известны тестировщику.

Согласно ISTQB, тестирование белого ящика — это:

- тестирование, основанное на анализе внутренней структуры компонента или системы;
- тест-дизайн, основанный на технике белого ящика — процедура написания или выбора тест-кейсов на основе анализа внутреннего устройства системы или компонента.
- Почему «белый ящик»? Тестируемая программа для тестировщика — прозрачный ящик, содержимое которого он прекрасно видит.

Тестирование серого ящика — метод тестирования ПО, который предполагает комбинацию White Box и Black Box подходов. То есть, внутреннее устройство программы нам известно лишь частично.

Тестирование чёрного ящика — также известное как тестирование, основанное на спецификации или тестирование поведения — техника тестирования, основанная на работе исключительно с внешними интерфейсами тестируемой системы.

Согласно ISTQB, тестирование черного ящика — это:

- тестирование, как функциональное, так и нефункциональное, не предполагающее знания внутреннего устройства компонента или системы;
- тест-дизайн, основанный на технике черного ящика — процедура написания или выбора тест-кейсов на основе анализа функциональной или нефункциональной спецификации компонента или системы без знания ее внутреннего устройства.

1.1.5 Тестовая документация

Стратегия тестирования (Test strategy) – подход к проведению тестирования (STLC). Небольшой статический документ, который предшествует плану тестирования, где сформулированы некоторые базовые подходы к тестированию, что позволяет убедиться в том, что все заинтересованные лица понимают одинаково, что и как будет тестироваться.

Стратегия включает:

- Роли и обязанности в команде тестирования
- Область тестирования
- Тестовые инструменты

- Тестовая среда
- График тестирования
- Сопутствующие риски

Тест план (Test plan) – документ, описывающий весь объем работ по тестированию, начиная с описания объекта, стратегии, расписания, критериев начала и окончания тестирования, до необходимого в процессе работы оборудования, специальных знаний, а также оценки рисков с вариантами их решения.

Тест план должен отвечать на следующие вопросы:

- Что необходимо протестировать?
- Как будет проводиться тестирование?
- Когда будет проводиться тестирование?
- Критерии начала тестирования?
- Критерии окончания тестирования?

Основные пункты тест плана:

1. Идентификатор тест плана (Test plan identifier);
2. Введение (Introduction);
3. Объект тестирования (Test items);
4. Функции, которые будут протестированы (Features to be tested;)
5. Функции, которые не будут протестированы (Features not to be tested);
6. Тестовые подходы (Approach);
7. Критерии прохождения тестирования (Item pass/fail criteria);
8. Критерии приостановления и возобновления тестирования (Suspension criteria and resumption requirements);
9. Результаты тестирования (Test deliverables);
10. Задачи тестирования (Testing tasks);
11. Ресурсы системы (Environmental needs);
12. Обязанности (Responsibilities);
13. Роли и ответственность (Staffing and training needs);
14. Расписание (Schedule);
15. Оценка рисков (Risks and contingencies);
16. Согласования (Approvals).

Чек-лист (check list) – это документ, который описывает что должно быть протестировано с разным уровнем детализации. (часто содержит только действия без ожидаемого результата)

Тестовый сценарий (test case) – это артефакт, описывающий совокупность шагов, конкретных условий и параметров, необходимых для проверки реализации тестируемой функции или ее части.

Атрибуты тест кейса:

- Предусловия (PreConditions) – список действий, которые приводят систему к состоянию пригодному для проведения основной проверки. Либо список условий, выполнение которых говорит о том, что система находится в пригодном для проведения основного теста состояния.
- Шаги (Steps) – список действий, переводящих систему из одного состояния в другое, для получения результата, на основании которого можно сделать вывод о удовлетворении реализации, поставленным требованиям.
- Ожидаемый результат (Expected result) – что по факту должны получить.

1.2 Исследовательское тестирование

Исследовательское тестирование – это одновременное изучение программы, проектирование и выполнение тестов. Этот подход — противоположность сценарного, когда список проверок составлен до проведения тестирования. Исследовательские тесты не определены заранее и не выполняются в точном соответствии с планом.

Исследовательское тестирование — это не методика тестирования. Это подход или образ мыслей, который можно применить к любой методике.

Исследовательское тестирование не должно выполняться небрежно, в спешке. Такой подход требует тщательной подготовки, а знания и умения тестировщика — важная форма этой подготовки. Исследовательское тестирование может проводиться как вручную, так и со вспомогательными инструментами.

Когда применять исследовательское тестирование?

- Нужна быстрая обратная связь о новом продукте.
- Нужно быстро изучить продукт.
- Сценарное тестирование не находит баги, требует разнообразия.
- Нужно принять решение о необходимости покрытия области сценарными тестами.
- Требований нет, они неполные или устарели.
- Продукт маленький, разработка тестовых сценариев займёт больше времени, чем сам процесс тестирования.

Исследовательское и сценарное тестирование

Преимущества сценарного тестирования

1. Тестирование можно планировать: тест-кейсы можно легко поделить между различными тестировщиками или командами.
2. Важные кейсы гарантировано будут пройдены.
3. Можно оценить процент покрытия требований тестами.
4. Тестовые сценарии можно использовать для обучения новых сотрудников.
5. Тестовые сценарии помогают проводить приёмочные испытания и определять критерии готовности.

Преимущества исследовательского тестирования

1. Нестандартные ходы выявляют нестандартные дефекты.
2. Не тратится время на описание всех сценариев.

3. Не нужна поддержка тестовых сценариев.
4. Не наступает «эффект пестицида».
5. Можно тестировать без требований.
6. Тесты могут стать интереснее и креативнее.

Ограничения для исследовательского тестирования.

Чистое исследовательское тестирование хорошо работает на небольших краткосрочных проектах или на начальных этапах жизни продукта. В первом случае объёмная тестовая документация может не оправдать времени, затраченного на её написание. Во втором — требования могут быть очень размытыми, так как нет определённости, в каком направлении будет двигаться разработка. Функциональность будет меняться, так что потребуется постоянная актуализация тест-кейсов.

Чтобы систематизировать исследовательское тестирование можно использовать идею туров. Туры – это идеи и инструкции по исследованию программного продукта, объединенные определённой общей темой или целью. Туры, как правило, ограничены по времени – длительность тестовой сессии не должна превышать 4 часа.

Идею туров развивали в своих работах Канер, Бах, Хендриксон, Болтон, Кохл и другие. В конце статьи есть дополнительные ссылки на их работы.

Тур – это своего рода план тестирования, он отражает основные цели и задачи, на которых будет сконцентрировано внимание тестировщика во время сессии исследовательского тестирования. При этом Виттакер использует метафору, что тестировщик – это турист, а тестируемое приложение – это город. Обычно у туриста (тестировщика) мало времени, поэтому он выполняет конкретную задачу в рамках выбранного тура, ни на что другое не отвлекаясь. Город (ПО) разбит на районы: деловой центр, исторический район, район развлечений, туристический район, район отелей, неблагополучный район.

Вывод:

Таким образом, исследовательское тестирование оказывает весьма положительное воздействие на тестирование веб-приложений, особенно в ситуациях, когда требования недостаточно ясны или меняются. Этот метод позволяет тестировщикам сохранить гибкость и творчество, исследуя приложение, находя интересные и неожиданные аспекты функциональности. Более того, использование исследовательского тестирования может способствовать обогащению знаний и навыков в области тестирования и разработки ПО, что соответствует целям дипломной работы.

1.3 Основы тестирования веб-приложений

Структура веб-приложений

Веб-приложение, или веб-сайт, представляет собой пакет, загруженный на клиентской стороне и содержащий множество слоев взаимозависимых модулей, построенных на основе веб-технологий:

- *HTML* – язык разметки гипертекста, представляющий основанную на тегах систему разметки документов, определяющую структуру и отдельные компоненты того, что в итоге компилируется в объектную модель документа, или DOM.
- *CSS* – каскадная таблица стилей, являющаяся фреймворком для стилизации приложения, с помощью которого идентифицируют и оформляют различные части DOM видимой области страницы. CSS предоставляет возможности вроде выбора элементов по их ID, классу и отношению к другим элементам DOM.
- *JavaScript* – высокоуровневый интерпретируемый скриптовый язык, с помощью которого прописывается и выполняется все поведение приложения.

Помимо слоя фронтенда в большинстве приложений также присутствует серверная сторона, или бэкенд, с API, построенным на микросервисах и базах данных. Серверная часть содержит все данные и бизнес-логику, абстрагируя всю эту информацию в соответствующие контракты, к которым фронтенд может обращаться через HTTP-методы, используя нужную форму запроса и учетные данные.

В зависимости от применяемых инструментов и внутренней сути сайта ему потребуется подходящая стратегия хостинга и инфраструктура. Сайты могут размещаться на различных системах, которые в общем можно разделить на два вида:

- *Статические веб-хосты*: используются для статических сайтов и представляют собой платформы доставки на базе хранилищ файлов, предоставляющие домены, электронные адреса, DNS и прочие возможности вроде SSL, шифрования и интеграции сторонних сервисов. В качестве примеров можно привести Godaddy, Hostinger и прочие.
- *Динамические веб-хосты*: служат для размещения динамических веб-приложений. К наиболее популярным относятся облачные платформы вроде AWS, Google Cloud, Azure, Salesforce и IBM cloud, которые предоставляют в качестве сервисов различные вычислительные возможности, включая виртуальные машины, базы данных,

масштабирование ресурсов по требованию и т.д. Эти платформы являются стандартом для развертывания веб-приложений в бизнес-целях ввиду их высокой производительности и безопасности. Кроме того, они предлагают к использованию передовые технологии ИИ и машинного обучения.

Веб-приложение, даже при небольшой начальной конфигурации по мере добавления в него страниц, контента и функциональности будет постепенно разрастаться. И однажды, когда будет достигнут определенный порог сложности, значительно затруднится управление приложением, а также отслеживание потребления и выделения ресурсов.

В зависимости от типа создаваемого приложения разработчик может использовать для него различную структуру:

- *Одностраничное приложение*. Состоит из одной модели документа, в рамках которой реализуется вся функциональность. Такие проекты задействуют большой объем функциональной логики, упаковываемой и отправляемой на компьютер клиента с соответствующими оптимизациями безопасности и производительности. В качестве известных примеров можно назвать Gmail, Facebook, GitHub и пр.
- *Многостраничное приложение*. Наиболее частый выбор, при котором приложение разделяется на множество страниц, доступных по различным путям URL. Создаются такие приложения с помощью серверных фреймворков и механизмов шаблонизации. При этом они имеют характерное преимущество в безопасности перед одностраничными решениями.
- *Прогрессивное веб-приложение*. Это современный способ применения WebView на мобильных устройствах для выполнения веб-приложений в качестве нативных, используя service worker, манифесты и оболочку.

Тестирование веб-приложений

Хороший план тестирования сайта включает в себя стратегию, задачи тестирования, подход, расписание тестов и среду их выполнения. Стратегия должна быть выстроена так, чтобы сайт в итоге отвечал всем бизнес-требованиям и соответствовал своему назначению.

Что касается подхода к тестированию, то он должен включать в себя:

- *Модульное тестирование*: проверку отдельных частей базы кода с помощью модульных тестов. Подробнее об этом можете почитать в руководстве по JavaScript (англ.) и документации Python (англ.). При этом современные подходы

программирования по принципу no-code/low-code и инструменты на основе ИИ позволяют написание модульных тестов частично автоматизировать.

- *Интеграционное тестирование*: подразумевает проверку различных сегментов кода сайта в виде независимых функций или модулей с помощью тестовой программы или прочих инструментов, активируемых при слиянии кода с родительским репозиторием. Все ведущие хостинги вроде GitHub, Gitlab и Bitbucket имеют встроенную поддержку реализации CI/CD.
- *Системное тестирование*: тестирование сайта на уровне пользовательского интерфейса и функционала вроде авторизации, регистрации и прочих потоков. В ходе этого процесса проверяется корректность совместной работы различных элементов сайта. Чаще всего с этой целью используется Selenium, наиболее популярный фреймворк для автоматизации действий браузера.
- *Приемочное тестирование*: как правило, это последняя стадия тестирования, на которой полностью собранное приложение с данными проверяется в продакшн-среде или среде интеграции (стейджинг). Этот этап включает в себя тестирование с привлечением фактических или мнимых пользователей. При этом наиболее эффективным способом управления процессом утверждения изменений в быстро меняющемся UI продукта является автоматизированное визуальное тестирование.
- *Тестирование производительности*: при постепенном увеличении числа пользователей необходимо обеспечить, чтобы сервера справлялись с нагрузкой в пиковые часы использования приложения. Помимо этого, данный этап также подразумевает поддержание сквозной безопасности в каждой точке контакта между сайтом и пользователями с помощью правильного использования HTTP-заголовков и анализа метаданных.

Роль браузеров

Для просмотра любого сайта необходим браузер, который запускается в операционной системе устройства. Эти особые приложения разрабатываются несколькими компаниями и обычно являются бесплатными.

Браузеры играют важную роль, не только тем, что делают сайты доступными для пользователей, но и тем, что помогают разработчикам, предоставляя широкий арсенал инструментов для тестирования и отладки различных аспектов веб-приложений.

Большинство браузеров предлагают в качестве вспомогательного интерфейса инструменты разработки, позволяя при желании заглянуть в код отрисовываемых страниц приложений.

Доступ к этим инструментам обычно можно получить через правый клик по странице и выбор опции «Просмотреть код элемента», либо комбинацией клавиш Ctrl+Chift+I.

Ниже некоторые основные компоненты инструментов разработчика в Chrome:

- *Elements*: проводник по элементам, который предоставляет доступ к скомпилированной DOM с множеством функций для добавления/удаления компонентов и установки состояний вроде наведения (hover), фокуса (focus) и т.д.
- *Console*: журнал вывода консоли при выполнении JavaScript. Очень полезен при отладке. Эту вкладку также можно использовать для выполнения фрагментов JS-кода на активной странице и просмотра его вывода.
- *Sources*: в этой вкладке вы увидите список всех файлов исходного кода, загруженных текущей страницей. Справа от нее находится отладчик скриптов, с помощью которого можно расставлять точки останова и корректировать выполнение сайта в реальном времени.
- *Network*: эта вкладка логирует все сетевые вызовы к странице и от нее, отражая большой список подробностей вроде типа, статуса, запроса/ответа, тайминга и т.д. Здесь также есть возможность эмуляции сценариев сетевой доступности с помощью функции ограничения скорости (throttling).
- *Performance*: на этой вкладке у вас есть возможность проводить запись и анализ быстродействия страницы во времени.
- *Memory*: эта вкладка позволяет делать моментальные снимки кучи, визуализировать использование памяти во времени, обнаруживать ее утечки, просматривать размеры объектов и т.д.
- *Application*: с помощью этого инструмента можно оценивать, редактировать и отлаживать service worker, кэш и прочие параметры.
- *Security*: предоставляет краткую сводку по валидности сертификата SSL.

В браузере также присутствует встроенная панель эмуляции устройств, которая позволяет эмулировать сценарии пользовательского интерфейса различных устройств, предлагая список предустановленных профилей разрешения, сетевой задержки, масштабирования, поворота экрана, а также возможность устанавливать собственное разрешение для тестирования отзывчивости.

Раздел разработчика в Chrome постоянно получает новые и более совершенные инструменты вроде Lighthouse, Recorder и прочих, которые предоставляют углубленные возможности анализа общего состояния приложения.

1.4 Основы автоматизированного тестирования.

Автоматизированное тестирование (Automation Testing, Test Automation) — техника тестирования, в которой для выполнения тест кейсов используются специальные программы. Это отличает ее от ручного тестирования, в котором тест кейсы выполняются вручную тестировщиком.

Программы для автоматизации сравнивают полученные результаты с актуальными и генерируют подробные тест-репорты.

Разработка продукта циклична и итерационна — и на каждой итерации, как правило, требуется выполнение одного и того же набора тестов. С помощью инструментов автоматизированного тестирования можно записывать наборы тестов (test suites) и выполнять, когда это необходимо. Как только набор тестов автоматизирован, участие человека в выполнении тестов практически не требуется. Это делает автоматизированное тестирование эффективной техникой.

Цель автоматизации — уменьшить количество тестов, которые нужно выполнять вручную.

Автоматизированное тестирование — лучший способ улучшить эффективность, покрытие продукта тестами, уменьшить время на тестирование. Автоматизированное тестирование очень важно, и вот почему:

- Ручное тестирование всех возможных сценариев использования требует много времени (и, следовательно, денег)
- Автоматизированное тестирование увеличивает скорость тестирования
- Автоматизированное тестирование не требует участия человека для выполнения тестов. Автоматизированные тесты могут быть запущены в любое время (днем, ночью, в выходные и праздники)
- Многократное ручное тестирование одной и той же функциональности скучно

Для максимальной эффективности, для определения сценариев, подходящих под автоматизацию, пользуйтесь следующими критериями:

- Критически важная бизнес-функциональность
- Тест кейсы, которые нужно выполнять много раз
- Тест кейсы, которые сложно воспроизвести вручную
- Тест кейсы, воспроизведение которых занимает много времени

Следующие критерии не подходят для автоматизации:

- Новые тест кейсы, которые еще не были выполнены вручную

- Тест кейсы для функциональности, требования к которой часто меняются
- Тест кейсы, которые выполняются редко

Процесс автоматизированного тестирования

Шаг 1: Выбор инструмента для автоматизации

Шаг 2: Определение функциональности, которую нужно автоматизировать

Шаг 3: Планирование, тест дизайн и разработка тестов

Шаг 4: Выполнение тестов

Шаг 5: Поддержка написанных тестов

Определение функциональности, которую нужно автоматизировать

Область для автоматизации может быть определена по следующим критериям:

- Функциональность, которая важна для бизнеса
- Сценарии, для тестирования которых нужны большие объемы входных данных
- Функциональность, использующаяся в нескольких частях приложения
- Целесообразность с технической точки зрения
- Сложность написания тест кейсов
- Возможность использования одних и тех же тест кейсов для кроссбраузерного тестирования

Планирование, тест дизайн и разработка

На этом этапе создается тест стратегия и тест-план, которые содержат следующие детали:

- Выбранный инструмент автоматизации
- Фреймворк с описанием его особенностей
- Описание функциональности, тестирование которой будет автоматизировано
- Подготовка стендов для выполнения тестов
- Расписание выполнение автотестов
- Результаты автоматизированного тестирования

Выполнение тестов

Во время этой стадии происходит выполнение автотестов. После выполнения генерируется подробный тест репорт.

Выполнение тестов может быть запущено как из инструмента автоматизации напрямую, так и с помощью системы управления тестированием (Test Management Tool), который запустит инструмент автоматизации.

Пример: HP Quality Center — система управления тестированием, которая управляет QTP для выполнения автотестов.

Поддержка написанных тестов

На стадии поддержки происходит изменение существующих тестов (в случае планируемого изменения функциональности) или добавление новых тестов.

Советы по использованию инструментов автоматизации

- Функциональность, подходящая для автоматизации, должна быть определена до начала разработки проекта.
- Инструмент для автоматизации должен быть выбран исходя из требований конкретного продукта, а не из популярности.
- Придерживайтесь стандартов написания кода, когда разрабатываете автотесты. Вот некоторые из них:
    - Придерживайтесь гайдлайнов при написании кода
    - Оставлйте комментарии
    - Обрабатывайте ошибки — при разработке думайте о том, как отработает ваша система в случае некорректного поведения приложения.
- Собирайте метрики, чтобы определить эффективность автоматизированного тестирования. Вот некоторые из них:
    - Процент найденных багов

- Время, затраченное на выполнение автотестов для каждого релиза

Преимущества автоматизированного тестирования

- На 70% быстрее, чем ручное тестирование
- Надежность
- Сохраняет время и деньги
- Не требует участия человека для выполнения тестов
- Возможность повторного использования написанных скриптов

Типы автоматизированного тестирования

- Smoke Testing
- Unit Testing
- Integration Testing
- Functional Testing
- Keyword Testing
- Regression Testing
- Data Driven Testing
- Black Box Testing

1.4.1 Автоматизированное тестирование веб-приложений

Термины

API (Application Programming Interface) — программный интерфейс приложения, с помощью которого одна программа может взаимодействовать с другой. API 3 позволяет слать информацию напрямую из одной программы в другую, минуя интерфейс взаимодействия с пользователем.

UI-тестирование — этап тестирования ПО, проверяющий графический интерфейс.

Data Driven Testing (DDT) — подход к созданию/архитектуре автоматизированных тестов (юнит, интеграционных, чаще применимо к backend-тестированию), при котором тест умеет принимать набор входных параметров и эталонный результат или эталонное состояние, с которым он должен сравнить результат, полученный в ходе прогонки входных параметров.

Тестирование API, UI, UX

Как правило, при тестировании веб-приложений проводят:

- UI-тестирование — тестирование графического интерфейса. Его задача — проверить функции приложения, имитируя действия пользователей. В процессе

тестирования элементы интерфейса проверяют на корректность, вводя данные в приложение через устройства ввода или средства автоматизированного UI-тестирования.

- UX-тестирование (юзабилити-тестирование) — тестирование качества интерфейса, его удобства для пользователя. Задача — понять, насколько хорошо, понятно, логично, удобно, правильно разработан ресурс, работают ли на нем все элементы и функции.
- Тестирование API. API — это интерфейс, позволяющий двум независимым компонентам программного обеспечения обмениваться информацией. Не у всех веб-приложений есть доступный для тестирования API, но там, где он есть, его обязательно нужно протестировать.

Тестирование UI хорошо поддается автоматизации с использованием библиотеки Selenium. Она позволяет работать с элементами интерфейса — получать их свойства и взаимодействовать с ними.
Хуже всего автоматизируется UX-тестирование, так как оценивать удобство для пользователя автоматически довольно сложно. Лучше, чтобы это делал человек.

Виды API (REST, SOAP)
API выполняет роль посредника между внутренними и внешними программными функциями, обеспечивая эффективный обмен информацией. Конечные пользователи могут не замечать работу API, но этот механизм очень важен. Рассмотрим два вида API — REST API и SOAP API.
REST API (от англ. Representational State Transfer, передача состояния представления или передача репрезентативного состояния) — это архитектурный подход, который устанавливает ограничения для API: как они должны быть устроены и какие функции должны поддерживать. Позволяет стандартизировать работу программных интерфейсов, сделать их более удобными и производительными.
В отличие от SOAP API, REST API — не протокол, а список рекомендаций, которым можно следовать или не следовать. Поэтому у него нет собственных методов.
SOAP (от англ. Simple Object Access Protocol, простой протокол доступа к объектам) — это протокол, по которому веб-сервисы взаимодействуют друг с другом или с клиентами.
SOAP API — веб-сервис, использующий протокол SOAP для обмена сообщениями между серверами и клиентами. При этом сообщения должны быть написаны на языке XML в соответствии со строгими стандартами, иначе сервер вернет ошибку.

Сравним SOAP и REST API в таблице:

SOAP API	REST API
Использование XML, WSDL	Ресурс-ориентированная технология
Работа с методами	HTTP-запросы
Поддержка транзакций и уровней безопасности	Для несложной бизнес-модели
Сложнее разрабатывать	Легче разрабатывать

Что такое DDT

Data Driven Testing (DDT) — подход к созданию/архитектуре автоматизированных тестов (юнит, интеграционных, чаще всего применимо к backend-тестированию), при котором тест умеет принимать набор входных параметров и эталонный результат или эталонное состояние, с которым он должен сравнить результат, полученный в ходе прогонки входных параметров (Data Driven Testing).

Такое сравнение и есть assert такого теста. При том как часть входных параметров могут передаваться опции выполнения теста или флаги, которые влияют на его логику.

Особый плюс хорошо спроектированного DDT — возможность ввести входные значения и эталонный результат в виде, удобном для всех ролей на проекте: от мануального тестировщика до менеджера проекта (тест-менеджера) и даже product owner’a.

Соответственно, когда вы способны загрузить мануальных тестировщиков увеличением покрытия и увеличением наборов данных, это удешевляет тестирование. Кроме того, удобный и понятный формат позволяет более наглядно видеть, что покрыто, а что нет. Это, по сути, и есть документация тестирования. К примеру, это может быть XLS-файл с понятной структурой (хотя чаще всего properties файла достаточно).

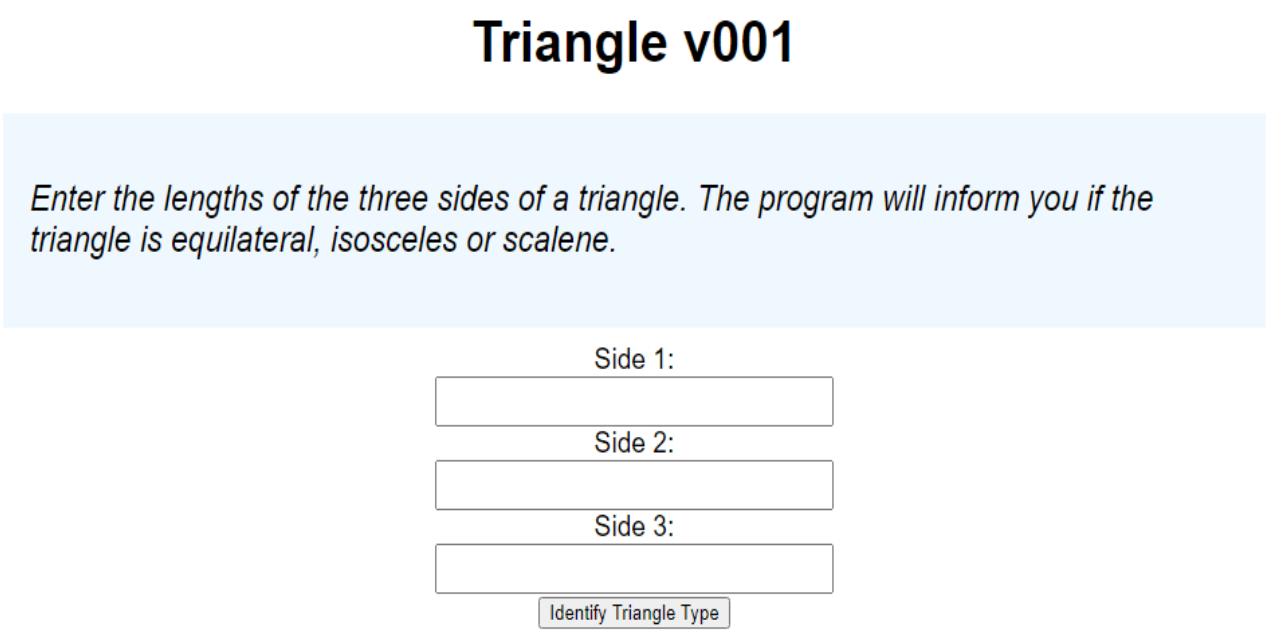
Или же создатель такого теста может дать коллегам workflow, по которому можно просто подготовить эталонные значения. То есть желательно избежать зависимости в создании наборов данных от программистов и автоматизаторов — любой на проекте должен суметь их подготовить.

Глава 2. Практическая часть

2.1 Ручное исследовательское тестирование веб-приложения "Triangle"

Целью данной работы является закрепление основ и сути процесса тестирования веб-приложений с использованием простого приложения "Triangle". На примере этого приложения, мы сосредотачиваемся на том, как применение теоретических знаний об исследовательском тестировании и о тестировании в целом может помочь в улучшении практических навыков и лучшем понимании основных аспектов веб-тестирования.

Описание веб-приложения "Triangle".
Веб-приложение "Triangle" представляет собой простой калькулятор, который принимает три стороны треугольника от пользователя и определяет тип треугольника на основе введенных данных. Основные функции приложения включают в себя определение равностороннего, равнобедренного и разностороннего треугольника.
Ссылка на приложение: https://testpages.eviltester.com/styled/apps/triangle/triangle001.html



Веб-приложение имеет максимально простой и интуитивно понятный интерфейс. Имеется описание, в котором говорится: *“Введите длины трех сторон треугольника. Программа сообщит вам, является ли треугольник равносторонним, равнобедренным или разносторонним.”*
Есть три поля для ввода значений и кнопка для получения результата при нажатии на неё.

Таким образом веб-приложение уже автоматически проходит UX (пользовательский опыт) тестирование.

Процесс ручного тестирования будет включать:

- Запуск приложения и ознакомление с пользовательским интерфейсом.
- Ввод различных значений для сторон треугольника и анализ результатов приложения.
- Проверка реакции приложения на некорректные данные.
- Кросс-браузерное тестирование в различных веб-браузерах.

Начнём составлять условный чек-лист для тестирования веб-приложения “Triangle”, без привязки к требованиям. Чек-лист будет содержать графы: проверка, ожидание, результат, комментарий.

Все тесты в чек-листе сопровождаются комментариями, в которых описывается ожидаемое поведение программы и оценивается, насколько она справилась с этими ожиданиями. Эта информация будет полезной для разработки автотестов "по методу черного ящика", так как она позволит легче определить, какие тесты следует автоматизировать и как оценивать их успешность.

Первая группа тестов состоит из введения валидных значений. Вводятся значения, соответствующие разностороннему, равностороннему и равнобедренному треугольнику.

Вторая группа тестов состоит из ввода невалидных значений. Таких как - нули для всех сторон; отрицательные значения сторон; буквы/символы вместо цифр; пустые строки; значения сторон из которых невозможно получить треугольник; ввода больших чисел; и как вариант проверки на XSS (межсайтовый скриптинг) уязвимость.

Также для наглядности применён метод “Pairwise” для ввода пустых полей в различной комбинации. Учитывая относительную простоту веб-приложения, было использовано два ключевых сценария для демонстрации тестирования: случаи, когда поля заполнены валидными значениями, и случаи, когда поля оставлены пустыми.

Цель - продемонстрировать применение метода “Pairwise”, для наглядного представления того, как можно проверить различные комбинации ввода данных.

Коротко о **Pairwise** (попарное тестирование) – это техника тест-дизайна, при которой тест-кейсы создаются так, чтобы выполнить все возможные отдельные комбинации каждой пары входных параметров. Достаточно проверить комбинации пар входных параметров, потому что ошибки чаще всего находятся именно на перекрестке двух параметров.

Далее в таблице отображено применение метода, где видно, как сократилось кол-во проверок с 8 до 4 благодаря “попарному тестированию”.

	Сторона 1	пусто	валидное значение
	Сторона 2	пусто	валидное значение
	Сторона 3	пусто	валидное значение

До применения метода попарного тестирования

	Сторона 1	**Сторона 2**	**Сторона 3**
1	Пусто	Пусто	Пусто
2	Пусто	Пусто	Валидное значение
3	Пусто	Валидное значение	Пусто
4	Пусто	Валидное значение	Валидное значение
5	Валидное значение	Пусто	Пусто
6	Валидное значение	Пусто	Валидное значение
7	Валидное значение	Валидное значение	Пусто
8	Валидное значение	Валидное значение	Валидное значение

После применения метода попарного тестирования

	Сторона 1	**Сторона 2**	**Сторона 3**
1	Пусто	Пусто	Пусто
2	Пусто	Валидное значение	Валидное значение
3	Валидное значение	Валидное значение	Пусто
4	Валидное значение	Пусто	Валидное значение

Подробный чек-лист тестирования доступен в гугл-таблице по ссылке Диплом. Условный чек-лист. Исследовательское тестирование.

А также в приложении к данной дипломной работе с названием “Таблица 1. Чек-лист.”

Вывод:

Все 19 тестов веб-приложения “Triangle” были успешно завершены. Эти тесты включали в себя 4 позитивных сценария, охватывающих валидные входные данные, и 15 негативных сценариев, где проверялись невалидные входные данные.

Результаты тестов соответствовали ожидаемому поведению приложения. Это подтверждает, что приложение способно корректно обрабатывать разнообразные ситуации и остается надежным в отношении ввода данных.

Следующий шаг - провести аналогичное тестирование с использованием автотестов. Это позволит не только подтвердить надежность веб-приложения, но и применить навыки автотестирования на практике.

2.2 Автоматизированное тестирование веб-приложения "Triangle"

Завершив ручное тестирование, можно переходить к следующему этапу в области тестирования программного обеспечения. На основе ожидаемых выводов программы, полученных в результате ручных проверок, теперь есть возможность попрактиковаться в написании автотестов методом “чёрного ящика”.
Автоматизированные тесты предоставляют эффективный инструмент для повторения и расширения уже пройденых проверок, что позволит более систематически и масштабно проверить функциональность веб-приложения “Triangle”.

Для автотестов были использованы следующие инструменты и паттерн.
Python - язык программирования, выбранный для автоматизации тестирования, известен своей простотой, читаемостью кода и многочисленными библиотеками, предназначенными для автоматизации. Python идеально подходит для разработки автотестов благодаря своей простоте и четкому синтаксису.
Pytest - фреймворк для написания и запуска тестов в Python. Его простота в использовании и расширяемость делают его популярным выбором для тестирования. Pytest обеспечивает мощный механизм для организации и выполнения тестов, генерации отчетов и управления параметрами тестовых запусков.
Selenium WebDriver - инструмент для автоматизации действий веб-приложений, который имитирует взаимодействие пользователя с браузером. Selenium WebDriver позволяет выполнять действия, такие как нажатие кнопок, ввод текста, выбор элементов и многое другое. Он поддерживает разные браузеры, что делает его универсальным инструментом для тестирования веб-приложений.
Page Object - концепция проектирования, которая помогает организовать автоматизацию тестирования веб-приложений. Она предполагает создание отдельных классов для каждой веб-страницы или компонента приложения, что упрощает управление элементами и действиями на странице. Page Object улучшает читаемость кода, повторное использование и обслуживание автотестов.
Девтулс (DevTools) - инструменты разработчика, встроенные в большинство современных веб-браузеров, которые предоставляют возможность анализа, отладки и тестирования веб-страниц и веб-приложений. Они доступны при тестировании методом чёрного ящика и предоставляют ряд важных возможностей, которые делают их неотъемлемой частью процесса автоматизации тестирования.

DevTools предоставляют широкий спектр функций и инструментов, таких как анализ производительности, мониторинг сетевых запросов, отладка JavaScript и другие. Однако в рамках нашей дипломной работы мы активно задействуем конкретно анализ HTML-кода страницы и идентификацию уникальных идентификаторов, такие как CSS-селекторы, для эффективного взаимодействия с элементами при создании автотестов.

Приступим к созданию нашего проекта, опираясь на паттерн Page Object.

Для начала создадим класс BasePage в файле BaseApp.py, который представляет базовую страницу для автотестов веб-приложения “Triangle” и дадим описание методам, которое можно будет вызвать через “help(BasePage)”.

```
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC


class BasePage:
    def __init__(self, driver):
        """
        Конструктор класса BasePage.
        Args:
            driver: Объект WebDriver для взаимодействия с браузером.
        """
        self.driver = driver
                                                    self.base_url       =
"https://testpages.eviltester.com/styled/apps/triangle/triangle001.html"

    def find_element(self, locator, time=10):
        """
        Метод для поиска элемента на странице.
        Args:
            locator: Локатор элемента (например, (By.ID, 'element_id'))
            time: Время ожидания (по умолчанию 10 секунд).
        Returns:
            Найденный элемент.
        Raises:
               TimeoutException: Если элемент не найден в течение указанного
времени.
        """
        return WebDriverWait(self.driver, time).until(
            EC.presence_of_element_located(locator),
            message=f"Can't find element by locator {locator}")

    def get_element_property(self, locator, property):
```

```
        """
        Метод для получения значения CSS-свойства элемента.
        Args:
            locator: Локатор элемента.
            property: Имя CSS-свойства (например, 'color').
        Returns:
            Значение указанного CSS-свойства элемента.
        """
        element = self.find_element(locator)
        return element.value_of_css_property(property)

    def go_to_site(self):
        """
        Метод для перехода на базовый URL веб-приложения.
        """
        return self.driver.get(self.base_url)


help(BasePage)
```

Далее определим фикстуру *browser* в файле “conftest.py”.

Фикстуры в PyTest используются для подготовки и очистки состояния для выполнения тестов. В данном случае, фикстура *browser* создает экземпляр браузера, который будет использоваться в тестах.

Создание фикстуры browser в отдельном файле conftest.py имеет несколько преимуществ:

Повторное использование: Фикстуры могут быть использованы во многих тестах, и можно определить их один раз в conftest.py и использовать в различных тестовых файлах.

Чистота кода тестов: Это помогает сделать тестовые файлы более чистыми и читаемыми, так как код для создания браузера выносится в отдельную фикстуру.

Модульность и поддержка: Это облегчает обслуживание и обновление кода фикстуры, так как он находится в одном месте (conftest.py), и нам не нужно повторять код в каждом тестовом файле.

Пользовательская конфигурация: если нужно внести изменения в способ создания браузера (например, добавить настройки или обработку исключений), то можно сделать это в одном месте (файл conftest.py).

```
import yaml
import pytest
from selenium import webdriver
from selenium.webdriver.chrome.service import Service
from webdriver_manager.chrome import ChromeDriverManager
from webdriver_manager.microsoft import EdgeChromiumDriverManager
```

```
from webdriver_manager.firefox import GeckoDriverManager

with open("Web_application_Triangle/testdata.yaml", encoding="utf-8") as f:
    testdata = yaml.safe_load(f)
    browser = testdata["browser"]

@pytest.fixture(scope="session")
def browser():
    """
    Фикстура для создания экземпляра браузера.
    Returns:
        webdriver: Экземпляр веб-драйвера для тестирования.
    """
    if browser == "microsoft":
        # Использование Microsoft Edge
        service = Service(
            executable_path=EdgeChromiumDriverManager().install())
        options = webdriver.EdgeOptions()
        driver = webdriver.Edge(service=service, options=options)
    elif browser == "firefox":
        # Использование Firefox
        service = Service(executable_path=GeckoDriverManager().install())
        options = webdriver.FirefoxOptions()
        driver = webdriver.Firefox(service=service, options=options)
    else:
        # Использование Chrome
        service = Service(executable_path=ChromeDriverManager().install())
        options = webdriver.ChromeOptions()
        driver = webdriver.Chrome(service=service, options=options)
    yield driver
    driver.quit()
```

Далее в отдельном файле “testpage.py” создадим классы для удобной работы с локаторами элементов страницы. И так же дадим краткое описание классам.

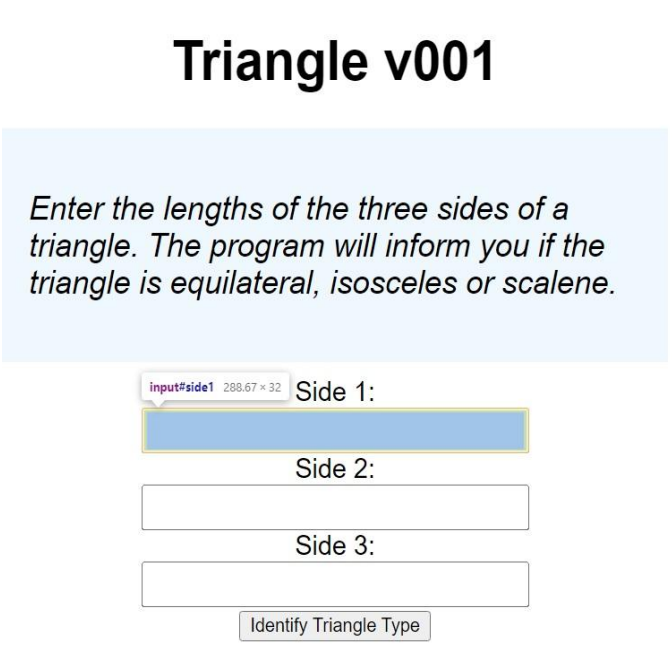
Локаторы - это специальные инструкции или выражения, используемые в автоматизации тестирования веб-приложений, чтобы найти и взаимодействовать с элементами на веб-странице. Их цель:

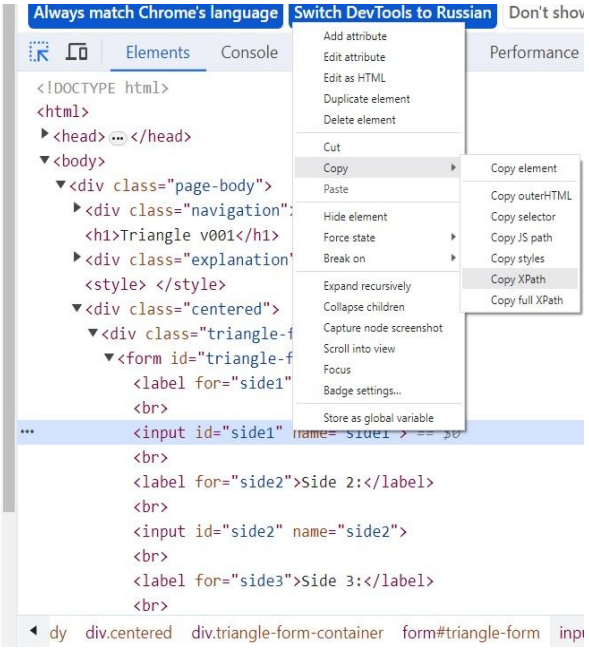
- Определить, где искать элементы на веб-странице.
- Указать, какие именно элементы следует выбрать и использовать в тестах.

Локаторы могут быть выражены с использованием разных методов и синтаксисов, таких как XPath, CSS-селекторы, идентификаторы элементов, и т. д. Они играют важную роль в

создании и поддержке автоматизированных тестов, позволяя находить и взаимодействовать с элементами веб-страницы в точно определенных местах.

Для получения локаторов элементов страницы воспользуемся инструментом разработчика DevTools. На скриншоте ниже пример получения нужных локаторов.





Для нашей задачи будем копировать локаторы элементов по XPATH и CSS. Вставим полученные локаторы в код.

```
from BaseApp import BasePage
from selenium.webdriver.common.by import By
import logging


class TestSearchLocators:
    """
    Класс, содержащий локаторы элементов страницы, используемых для поиска элементов.
    """
    LOCATOR_SIDE1_FIELD = (
        By.XPATH, """/html/body/div[1]/div[3]/div[1]/form/input[1]""")
    LOCATOR_SIDE2_FIELD = (
        By.XPATH, """/html/body/div[1]/div[3]/div[1]/form/input[2]""")
    LOCATOR_SIDE3_FIELD = (
        By.XPATH, """/html/body/div[1]/div[3]/div[1]/form/input[3]""")
    LOCATOR_BUTTON = (By.CSS_SELECTOR, """#identify-triangle-action""")
    LOCATOR_TEXT_FIELD = (By.XPATH, """/html/body/div[1]/div[3]/div[2]""")


class OperationsHelper(BasePage):
    """
```

```
    Класс `OperationsHelper`, наследуется от класса `BasePage` и
предоставляет методы
    для выполнения операций на веб-странице, такие как ввод данных и нажатие
кнопок.
    """

    def enter_side1(self, side):
        logging.info(
            f"Send {side} to element
{TestSearchLocators.LOCATOR_SIDE1_FIELD[1]}")
        side_field =
self.find_element(TestSearchLocators.LOCATOR_SIDE1_FIELD)
        side_field.clear()
        side_field.send_keys(side)

    def enter_side2(self, side):
        logging.info(
            f"Send {side} to element
{TestSearchLocators.LOCATOR_SIDE2_FIELD[1]}")
        side_field =
self.find_element(TestSearchLocators.LOCATOR_SIDE2_FIELD)
        side_field.clear()
        side_field.send_keys(side)

    def enter_side3(self, side):
        logging.info(
            f"Send {side} to element
{TestSearchLocators.LOCATOR_SIDE3_FIELD[1]}")
        side_field =
self.find_element(TestSearchLocators.LOCATOR_SIDE3_FIELD)
        side_field.clear()
        side_field.send_keys(side)

    def click_button(self):
        logging.info("Click button")
        self.find_element(TestSearchLocators.LOCATOR_BUTTON).click()

    def get_text(self):
        text_field = self.find_element(
            TestSearchLocators.LOCATOR_TEXT_FIELD, time=3)
        text = text_field.text
        logging.info(
            f"We find text {text} in text field
{TestSearchLocators.LOCATOR_TEXT_FIELD[1]}")
        return text
```

Для удобства и гибкости автоматизированного тестирования пропишем некоторые настройки в файле YAML (YAML - это удобный формат для структурирования данных), который позволит легко изменять различные параметры для последующего тестирования. Одной из важных настроек может быть кроссбраузерность, то есть возможность запускать тесты в разных веб-браузерах.

Преимущество использования YAML-файла заключается в том, что мы можем легко настраивать тестовые параметры, такие как выбор браузера, ссылки, данные для ввода и многие другие параметры, без необходимости изменения кода тестов. Это делает автоматизированное тестирование более гибким и облегчает поддержку и масштабирование тестовых сценариев.

Таким образом, использование YAML-файла для хранения настроек - это еще один плюс автоматизированного тестирования, который делает процесс более управляемым и адаптируемым к изменяющимся требованиям и средам выполнения.

Файл “testdata.yaml” будет выглядеть так:

```yaml
# Тестирование веб-приложения "Triangle", которое принимает на вход
# три стороны треугольника и выдает информацию о типе этого треугольника
# (равнобедренный, равносторонний, разносторонний).
# Веб-приложение расположено по ссылке -
# https://testpages.eviltester.com/styled/apps/triangle/triangle001.html

browser: chrome
test1: [7, 7, 4] # равнобедренный треугольник
test2: [5, 5, 5] # равносторонний треугольник
test3: [3, 4, 5] # разносторонний треугольник

test4: [0, 0, 0] # невалидные значения - нули
test5: [-2, 3, 4] # невалидные значения с отрицательным числом
test6: [10, 20, 10] # невалидные значения с несуществующем треугольником

test7: ["аба", 20, 10] # ввод букв в 1ый столбец
test8: [10, "аба", 10] # ввод букв во 2ой столбец
test9: [10, 20, "аба"] # ввод букв в 3ий столбец

test10: [4294967295, 4294967295, 4294967000] # ввод большого валидного числа
test11: [] # ввод пустых полей
test12:  ["<script>alert('XSS')</script>",  "<script>alert('XSS')</script>",
"<script>alert('XSS')</script>"] # проверка на XSS уязвимость
```

Основные настройки сделаны, теперь напишем автотесты для веб-приложения “Triangle”

Перед этим сделаем ещё одну небольшую настройку в отдельном файле “pytest.ini”

```
[pytest]
log_file_format = %(asctime)s %(levelname)s %(message)s
log_file_date_format = %Y-%m-%d %H:%M:%S
log_file = log.txt
log_file_level = 20
addopts = -v --html-report=report.html
```

Будем сохранять логи в режиме INFO, а также создадим подробный отчёт о тестировании в HTML формате.

Создаем отдельный файл с автотестами “test_app.py”

```
from testpage import OperationsHelper
import logging
import yaml
import pytest

with open("testdata.yaml", encoding="utf-8") as f:
    testdata = yaml.safe_load(f)


def test_1(browser):
    """
    Проверяем треугольник на равнобедренность,
    вводя валидные данные в приложение.
    """
    logging.info("Test Starting")
    testpage = OperationsHelper(browser)
    testpage.go_to_site()
    testpage.enter_side1(testdata.get("test1")[0])
    testpage.enter_side2(testdata.get("test1")[1])
    testpage.enter_side3(testdata.get("test1")[2])
    testpage.click_button()
    assert "isosceles" in testpage.get_text().lower(), "test FAILED"


def test_2(browser):
    """
    Проверяем треугольник на равносторонность,
    вводя валидные данные в приложение.
    """
    logging.info("Test Starting")
    testpage = OperationsHelper(browser)
    testpage.go_to_site()
    testpage.enter_side1(testdata.get("test2")[0])
    testpage.enter_side2(testdata.get("test2")[1])
```

```
    testpage.enter_side3(testdata.get("test2")[2])
    testpage.click_button()
    assert "equilateral" in testpage.get_text().lower(), "test FAILED"


def test_3(browser):
    """
    Проверяем треугольник на разносторонность,
    вводя валидные данные в приложение.
    """
    logging.info("Test Starting")
    testpage = OperationsHelper(browser)
    testpage.go_to_site()
    testpage.enter_side1(testdata.get("test3")[0])
    testpage.enter_side2(testdata.get("test3")[1])
    testpage.enter_side3(testdata.get("test3")[2])
    testpage.click_button()
    assert "scalene" in testpage.get_text().lower(), "test FAILED"


@pytest.mark.parametrize("test_name, side_values", [
    ("test4", [0, 0, 0]),
    ("test5", [-2, 3, 4]),
    ("test6", [10, 20, 10])
])
def test_4_5_6(browser, test_name, side_values):
    """
    Вводим невалидные данные в приложение,
              объединив   три   однотипных   теста   с   помощью   фикстуры
@pytest.mark.parametrize.
    """
    logging.info("Test Starting")
    testpage = OperationsHelper(browser)
    testpage.go_to_site()
    side1, side2, side3 = side_values
    testpage.enter_side1(side1)
    testpage.enter_side2(side2)
    testpage.enter_side3(side3)
    testpage.click_button()
    assert "error: not a triangle" in testpage.get_text(
    ).lower(), f"{test_name} FAILED"


@pytest.mark.parametrize("test_name, side_values", [
    ("test7", ["aба", 20, 10]),
```

```
    ("test8", [10, "аба", 10]),
    ("test9", [10, 20, "аба"])
])
def test_7_8_9(browser, test_name, side_values):
    """
    Вводим невалидные данные в приложение,
              объединив   три   однотипных   теста   с   помощью   фикстуры
@pytest.mark.parametrize.
    """
    logging.info("Test Starting")
    testpage = OperationsHelper(browser)
    testpage.go_to_site()
    side1, side2, side3 = side_values
    testpage.enter_side1(side1)
    testpage.enter_side2(side2)
    testpage.enter_side3(side3)
    testpage.click_button()
      assert "error: side 1 is not a number" or "error: side 2 is not a
number" or "error: side 3 is not a number" in testpage.get_text(
    ).lower(), f"{test_name} FAILED"


def test_10(browser):
    """
    Ввод большого валидного числа.
    """
    logging.info("Test Starting")
    testpage = OperationsHelper(browser)
    testpage.go_to_site()
    testpage.enter_side1(testdata.get("test10")[0])
    testpage.enter_side2(testdata.get("test10")[1])
    testpage.enter_side3(testdata.get("test10")[2])
    testpage.click_button()
    assert "isosceles" in testpage.get_text().lower(), "test FAILED"


def test_11(browser):
    """
    Ввод пустых полей.
    """
    logging.info("Test Starting")
    testpage = OperationsHelper(browser)
    testpage.go_to_site()
    testpage.enter_side1(testdata.get("test11"))
    testpage.enter_side2(testdata.get("test11"))
```

```
    testpage.enter_side3(testdata.get("test11"))
    testpage.click_button()
    assert "error: side 1 is missing" in testpage.get_text().lower(), "test
FAILED"


def test_12(browser):
    """
    Проверка на XSS уязвимость
    """
    logging.info("Test Starting")
    testpage = OperationsHelper(browser)
    testpage.go_to_site()
    testpage.enter_side1(testdata.get("test12")[0])
    testpage.enter_side2(testdata.get("test12")[1])
    testpage.enter_side3(testdata.get("test12")[2])
    testpage.click_button()
    assert "error: side 1 is not a number" in testpage.get_text().lower(),
"test FAILED"
```

В тестах стоит обратить внимание на то, что были объединены некоторые однотипные тесты с ожидаемыми одинаковыми ответами программы, используя фикстуру *@pytest.mark.parametrize*. Эта фикстура позволяет параметризировать тестовые функции, предоставляя разные наборы входных данных для одного и того же тестового сценария.

Когда мы используем *@pytest.mark.parametrize*, мы определяем наборы параметров, которые будут переданы в тестовую функцию. Функция будет выполнена несколько раз - по разу для каждого набора параметров. Это позволяет создавать компактные и поддерживаемые тесты, когда у есть несколько схожих случаев, требующих проверки.

Таким образом, фикстура *@pytest.mark.parametrize* позволяет эффективно организовать однотипные тесты с ожидаемыми одинаковыми результатами и уменьшить дублирование кода, делая тестовую базу более читаемой и легко поддерживаемой.

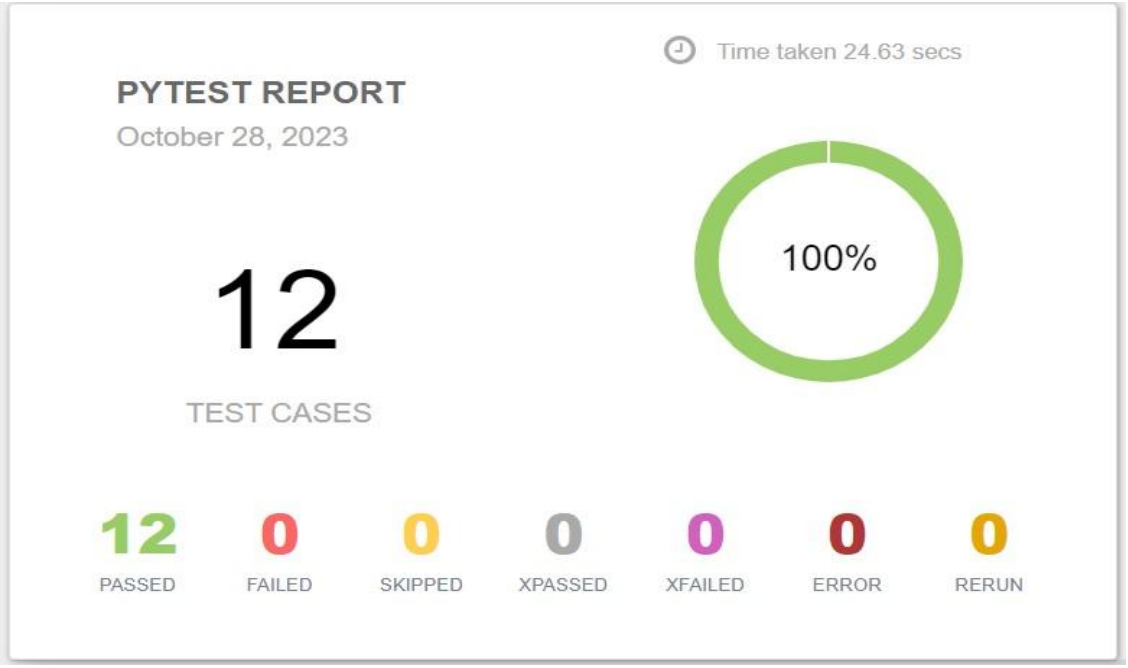
Когда всё подготовлено к тестированию, запускаем тест в терминале с помощью команды “*pytest*”.

Все тесты успешно пройдены!

Полностью удобочитаемый код с автотестами расположен по ссылке на GitHub - **https://github.com/1stFunt/Autotest_web_applications/tree/main/Web_application_Triangle**

Полный отчёт о пройденных тестах находится в файле “report.html”.

Краткий отчёт.



Вывод:

В ходе этой практики проведено углубленное изучение автоматизации тестирования, подтверждающее, что автоматизация процесса тестирования значительно упрощает и ускоряет его выполнение. Особенно важно это при кроссбраузерном тестировании, когда требуется проверить работоспособность в разных браузерах и на разных платформах. Вместо ручного перебора браузеров и платформ, достаточно изменить один параметр в тестовом скрипте, и автотесты сами выполнят все необходимые проверки.

Следующим этапом будет интеграция полученных знаний в повседневную работу, использование автотестов для более крупных и ответственных проектов. Мы также продолжим развивать свои навыки в области автоматизации тестирования.

Важным аспектом этой практики было соблюдение принципов DDT (Data-Driven Testing), позволяющих структурировать и организовать тесты более эффективно. DDT предоставляет возможность отделения тестовых данных от тестового кода, что значительно облегчает поддержку и сопровождение автотестов.

Кроме того, наши автотесты разрабатывались с учетом принципов Page Object, делая код более модульным и понятным. Применение Page Object позволяет абстрагироваться от деталей веб-страницы, что способствует улучшению читаемости и структурированности кода.

Итак, завершая эту практику, мы готовы применить наши навыки автоматизации тестирования в повседневной работе.

Заключение

Теоретическая часть является краеугольным камнем нашей дипломной работы. Она предоставила нам обширный и глубокий обзор фундаментальных принципов тестирования программного обеспечения. Мы начали с изучения фундаментальной теории, что дало нам понимание основных концепций и принципов, которые лежат в основе процесса тестирования.

Что такое баг и его жизненный цикл помогли нам осознать важность выявления и управления дефектами в процессе разработки ПО.

Основные виды тестирования ПО представили нам разнообразные методы тестирования, которые используются для проверки различных аспектов приложений.

Техники тест-дизайна предложили разнообразные подходы к планированию и созданию тестовых случаев.

Тестовая документация описала важность документирования тестовых процедур и результатов.

Все это сформировало нам прочный теоретический фундамент.

Исследовательское тестирование представило нам современные методы, где тестирование требует гибкости и креативности. Здесь мы поняли, что не всегда существуют четкие границы и требования, и исследовательское тестирование становится неотъемлемой частью процесса.

Основы тестирования веб-приложений расширила наше понимание, сфокусировавшись на особенностях тестирования веб-приложений. Мы изучили, как обеспечить надежность и безопасность веб-приложений, а также как проверить их совместимость с различными браузерами и устройствами.

Основы автоматизированного тестирования открыла перед нами мир автоматизации, предоставив обзор инструментов и методов, которые позволяют оптимизировать процесс тестирования веб-приложений.

В итоге, теоретическая часть дипломной работы представляет собой устойчивую основу для дальнейшего исследования и практического применения методов и навыков тестирования программного обеспечения. Эти знания позволят более глубоко понимать и успешно применять тестирование в практике, повышая надежность и безопасность веб-приложений.

В практической части данной дипломной работы мы были в состоянии успешно применить полученные теоретические знания и навыки в деле, несмотря на относительно небольшой объем приложения, который был предметом нашего исследования. Этот опыт позволил нам

не только понимать принципы и методы тестирования на уровне теории, но и применять их на практике, создавая условия для работающего веб-приложения.

Прежде всего, мы провели ручное тестирование, составив условный чек-лист, который базировался на исследовательском подходе. Наш тест включал в себя 19 разнообразных проверок, и радостно отметить, что каждая из них завершилась успешно. Этот опыт подтверждает не только наши теоретические знания, но и способность применять их на практике с учетом особенностей конкретного веб-приложения.

Далее, на основе результатов ручного тестирования, мы перешли к созданию автоматизированных тестов с использованием мощных инструментов, таких как Python, Selenium WebDriver и PageObject. Эти автотесты также успешно прошли проверку и сформировали надежную основу для дальнейшей автоматизации тестирования. Мы пришли к пониманию важности автоматизации в области тестирования веб-приложений.

Важно отметить, что данный опыт не только подготовил нас к работе с более сложными веб-приложениями, но и создал почву для дальнейшего развития исследований в области тестирования программного обеспечения. Это отличная отправная точка для нашей будущей карьеры, и мы гордимся тем, что смогли внести свой вклад в область тестирования.

Список используемой литературы

1. Фундаментальная теория тестирования: сайт Habr - https://habr.com/ru/articles/549054/ (25.03.2021)

2. Исследовательское тестирование: сайт (17.09.2015) - https://www.software-testing.by/blog/exploratory-testing-exploratory-tours/

4. Знакомство с тестированием веб-приложений: сайт Habr - https://habr.com/ru/companies/ruvds/articles/676752/ (17.07.2022)

4. Основы автоматизированного тестирования: сайт Habr - https://testengineer.ru/chto-takoe-avtomatizirovannoe-testirovanie/ (25.07.2021)

5. Автоматизированное тестирование веб-приложений - конспекты Geekbrains (2023)

6. Техники тест-дизайна взяты из книги "A Practitioner’s Guide to Software Test Design", Lee Copeland (2003)

Приложения

Таблица 1. Чек-Лист.

	A	B	C	D
1	**Проверка**	**Ожидание**	**Результат**	**Комментарий**
2	Проверка кнопки "Indentefy Triangle Type"	Кнопка реагирует на "клик"	passed	При нажатии кнопки выводится сообщение, что говорит о её работоспособности
3	Ввод валидных значений (3, 4, 5)	Приложение определило треугольник как разносторонний	passed	Выводится сообщение "Scalene", что значит разносторонний
4	Ввод валидных значений (5, 5, 5)	Приложение определило треугольник как равносторонний	passed	Выводится сообщение "Equilateral", что значит равносторонний
5	Ввод валидных значений (7, 7, 4)	Приложение определило треугольник как равнобедренный	passed	Выводится сообщение "Isosceles", что значит равнобедренный
6	Ввод невалидных значений (0, 0, 0)	Приложение либо не выдаёт овет, либо сообщает о некорректных значениях.	passed	Выводится сообщение "Error: Not a Triangle", что говорит о правильной обработке ввода приложением некорректных данных
7	Ввод невалидных значений с отрицательными числами (-2, 3, 4)	Приложение либо не выдаёт овет, либо сообщает о некорректных значениях.	passed	Выводится сообщение "Error: Not a Triangle", что говорит о правильной обработке ввода приложением некорректных данных
8	Ввод невалидных значений с невозможным соотношением сторон треугольника (10, 20, 10)	Приложение либо не выдаёт овет, либо сообщает о некорректных значениях.	passed	Выводится сообщение "Error: Not a Triangle", что говорит о правильной обработке ввода приложением некорректных данных
9	Ввод невалидных значений в виде букв/символов в графу "Side 1"	Приложение либо не выдаёт овет, либо сообщает о некорректных значениях.	passed	Выводится сообщение "Error: Side 1 is not a Number", что говорит о правильной обработке ввода приложением некорректных данных
10	Ввод невалидных значений в виде букв/символов в графу "Side 2"	Приложение либо не выдаёт овет, либо сообщает о некорректных значениях.	passed	Выводится сообщение "Error: Side 2 is not a Number", что говорит о правильной обработке ввода приложением некорректных данных
11	Ввод невалидных значений в виде букв/символов в графу "Side 3"	Приложение либо не выдаёт овет, либо сообщает о некорректных значениях.	passed	Выводится сообщение "Error: Side 3 is not a Number", что говорит о правильной обработке ввода приложением некорректных данных
12	Ввод больших чисел в графы (4294967295, 4294967295, 4294967000)	Приложение либо не выдаёт овет, либо выдаёт предупреждение, либо считывает треугольник, если такой получается.	passed	Выводится сообщение "Isosceles", что значит равнобедренный. Это говорит о том, что приложение не только справилось с потонциально большим числом, но и смогло определить треугольник, верно считывая данные
13	Ввод пустого поля/полей в след комбинации (пусто, пусто, пусто)	Приложение либо не выдаёт овет, либо предупреждает о пустом поле/полях.	passed	Выводится сообщение "Error: Side 1 is missing", что говорит о правильной обработке ввода некорректных данных
14	Ввод пустого поля/полей в след комбинации (пусто, валидное значение, валидное значение)	Приложение либо не выдаёт овет, либо предупреждает о пустом поле/полях.	passed	Выводится сообщение "Error: Side 1 is missing", что говорит о правильной обработке ввода некорректных данных
15	Ввод пустого поля/полей в след комбинации (валидное значение, валидное значение, пусто)	Приложение либо не выдаёт овет, либо предупреждает о пустом поле/полях.	passed	Выводится сообщение "Error: Side 3 is missing", что говорит о правильной обработке ввода некорректных данных
16	Ввод пустого поля/полей в след комбинации (валидное значение, пусто, валидное значение)	Приложение либо не выдаёт овет, либо предупреждает о пустом поле/полях.	passed	Выводится сообщение "Error: Side 2 is missing", что говорит о правильной обработке ввода некорректных данных
17	Ввод скрипта <script>alert('XSS')</script> в графу "Side 1"	Приложение либо не выдаёт овет, либо сообщает о некорректных значениях.	passed	Выводится сообщение "Error: Side 1 is not a Number", что говорит о правильной обработке ввода приложением некорректных данных и оно не подвержено XSS уязвимости.
18	Ввод скрипта <script>alert('XSS')</script> в графу "Side 2"	Приложение либо не выдаёт овет, либо сообщает о некорректных значениях.	passed	Выводится сообщение "Error: Side 2 is not a Number", что говорит о правильной обработке ввода приложением некорректных данных и оно не подвержено XSS уязвимости.
19	Ввод скрипта <script>alert('XSS')</script> в графу "Side 3"	Приложение либо не выдаёт овет, либо сообщает о некорректных значениях.	passed	Выводится сообщение "Error: Side 3 is not a Number", что говорит о правильной обработке ввода приложением некорректных данных и оно не подвержено XSS уязвимости.

Отчёт

Pytest HTML Reporter

Suite	Test Case	Status	Time (s)
test_app.py	test_1	PASS	14.63
test_app.py	test_7_8_9[test7-side_values0]	PASS	0.51
test_app.py	test_10	PASS	0.35
test_app.py	test_12	PASS	0.35
test_app.py	test_2	PASS	0.34
test_app.py	test_3	PASS	0.34
test_app.py	test_4_5_6[test4-side_values0]	PASS	0.33
test_app.py	test_4_5_6[test5-side_values1]	PASS	0.33
test_app.py	test_4_5_6[test6-side_values2]	PASS	0.33
test_app.py	test_11	PASS	0.33
test_app.py	test_7_8_9[test8-side_values1]	PASS	0.32
test_app.py	test_7_8_9[test9-side_values2]	PASS	0.32