М. С. НИКОЛЮКИН, В. В. КОНКИНА, К. И. ПАТУТИН

ТЕСТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Тамбов
Издательский центр ФГБОУ ВО «ТГТУ»
2025

Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования «Тамбовский государственный технический университет»

М. С. НИКОЛЮКИН, В. В. КОНКИНА, К. И. ПАТУТИН

ТЕСТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Утверждено Ученым советом университета в качестве учебного пособия для бакалавров 3 курса направления подготовки 09.03.01 «Информатика и вычислительная техника», изучающих дисциплину «Тестирование программного обеспечения»

Учебное электронное издание

Тамбов
Издательский центр ФГБОУ ВО «ТГТУ»
2025

УДК 004.415.53
ББК 32.971.3
Н63

Рецензенты:

Кандидат технических наук, доцент, доцент кафедры
«Математическое моделирование и информационные технологии»
Института новых технологий и искусственного интеллекта
ФГБОУ ВО «ТГУ имени Г. Р. Державина»
Д. С. Соловьев

Кандидат технических наук, доцент, доцент кафедры
«Информационные системы и защита информации»
ФГБОУ ВО «ТГТУ»
Ю. В. Кулаков

Николюкин, М. С.

Н63 Тестирование программного обеспечения [Электронный ресурс] : учебное пособие / М. С. Николюкин, В. В. Конкина, К. И. Патутин. – Тамбов : Издательский центр ФГБОУ ВО «ТГТУ», 2025. – 1 электрон. опт. диск (CD-ROM). – Системные требования : ПК не ниже класса Pentium II ; CD-ROM-дисковод ; 1,0 Mb ; RAM ; Windows 95/98/XP ; мышь. – Загл. с экрана.

ISBN 978-5-8265-2883-9

Рассматриваются основные понятия облачных технологий, приводятся общие сведения по работе и взаимодействию с ними.

Предназначено для бакалавров 3 курса направления подготовки 09.03.01 «Информатика и вычислительная техника», изучающих дисциплину «Тестирование программного обеспечения».

УДК 004.415.5
ББК 32.971.3

ISBN 978-5-8265-2883-9

ВВЕДЕНИЕ

Данное пособие охватывает основные аспекты тестирования программного обеспечения, включая теоретические основы, категории тестирования, а также ключевые принципы и методы, используемые для обеспечения качества программных продуктов.

В первом разделе особое внимание уделяется теоретическим аспектам, таким как классификация видов тестирования, основные понятия и определения, а также общие подходы к тестированию на различных этапах жизненного цикла программного обеспечения. Читатель сможет получить глубокие знания о том, как правильно организовать и проводить тестирование, а также понять важность тестирования в процессе разработки.

Во втором разделе пособия рассматриваются практические аспекты автоматизированного тестирования. Здесь представлены руководства по созданию и использованию автоматизированных тестов, выбору инструментов для автоматизации, а также процессу интеграции автоматизированных тестов в общий процесс разработки. Читатель научится создавать эффективные автоматизированные тесты, оптимизировать их выполнение и повышать качество программного обеспечения за счет автоматизации повторяющихся задач.

Пособие предназначено для студентов высших учебных заведений, обучающихся по направлению подготовки бакалавров 09.03.01 «Информатика и вычислительная техника» в рамках дисциплины «Тестирование программного обеспечения».

1. ТЕОРЕТИЧЕСКИЕ ОСНОВЫ ТЕСТИРОВАНИЯ

1.1. ОСНОВЫ ТЕСТИРОВАНИЯ

Основные понятия и определения

Люди совершают ошибки, некоторые из которых могут быть незначительными, а другие иметь разрушительные последствия. Все, что производится человеком, может содержать ошибки, и именно поэтому любой продукт нуждается в проверке – тестировании. Тестирование программного обеспечения (ПО) – это процесс, направленный на выявление и устранение дефектов, чтобы обеспечить соответствие продукта требованиям и удовлетворить ожидания пользователей.

Тестирование программного обеспечения представляет собой процесс, охватывающий все активности жизненного цикла программного продукта, включая как динамические, так и статические действия, связанные с планированием, подготовкой и оценкой программного обеспечения. Основной целью тестирования является проверка соответствия программного продукта описанным требованиям, его пригодности для заявленных целей, а также выявление дефектов. В соответствии с глоссарием *ISTQB* (*International Software Testing Qualifications Board*), тестирование определяется как процесс, который включает в себя все виды деятельности, направленные на планирование, подготовку, оценку и документирование программного обеспечения в целях проверки его соответствия требованиям и выявления дефектов.

ISTQB (*International Software Testing Qualifications Board*) – Международный совет по квалификации в области тестирования программного обеспечения, который разрабатывает стандарты и сертификационные программы для специалистов в области

тестирования. Основной целью ISTQB является повышение уровня знаний и навыков тестировщиков по всему миру посредством сертификаций и предоставления доступа к глоссарию терминов, связанных с тестированием.

Дефект (*Bug, Defect*) – отклонение программного продукта от его требований или спецификаций, которое может привести к неправильной работе системы. Дефект может возникнуть как в результате ошибок на этапе разработки, так и вследствие неправильного или неполного понимания требований.

Техническое задание (ТЗ) – документ, который описывает требования и спецификации программного продукта, включающие в себя функциональные и нефункциональные характеристики. ТЗ служит основой для разработки, тестирования и последующей оценки программного обеспечения.

Артефакты тестирования – это побочные продукты, создаваемые в процессе тестирования программного обеспечения и используемые для улучшения коммуникации между членами команды проекта. Артефакты включают документы, такие как тест-планы, отчеты о дефектах, результаты тестов и прочие материалы, которые фиксируют и систематизируют ход тестирования. Они помогают команде проектировать, отслеживать и оценивать процесс тестирования, обеспечивая ясность и прозрачность всех этапов работы.

Тестирование можно рассматривать как процесс исследования или изучения программы в целях проверки ее соответствия ожиданиям. Этот процесс предполагает запуск программы и проверку того, что весь функционал соответствует техническому заданию. Для этого используются заранее написанные или подготовленные проверки.

Таким образом, тестирование является процессом исследования программы, направленным на определение ее соответствия заявленным требованиям с помощью заранее подготовленных проверок.

Цели тестирования зависят от целей самого проекта. Однако существует несколько общих задач, которые можно выделить как основные. Первой задачей является проверка того, что все указанные требования выполнены. Для этого необходимо убедиться, что все элементы программы соответствуют техническому заданию и работают правильно. Следующей задачей является создание уверенности в уровне качества объекта тестирования. Тестирование, хотя и не влияет напрямую на качество, позволяет выявлять и исправлять дефекты, что способствует достижению заданного уровня качества.

Ошибочно считать, что простое нахождение и исправление большого числа дефектов обеспечит успех продукту. Тщательное тестирование всех требований и исправление всех найденных дефектов может привести к созданию системы, которая не будет удовлетворять потребности и ожидания пользователей или окажется менее удобной в использовании по сравнению с конкурирующими продуктами.

Тестирование программного обеспечения – это не только поиск и устранение дефектов, но и комплексный процесс, направленный на обеспечение того, чтобы продукт соответствовал всем заявленным требованиям, был удобен и полезен для конечного пользователя, а также конкурентоспособен на рынке.

Основные принципы тестирования

Тестирование программного обеспечения играет ключевую роль в обеспечении его качества, выявляя дефекты и подтверждая соответствие продукта требованиям. Принципы тестирования, разработанные на основе многолетнего опыта, представляют собой основные руководства, которые помогают командам *QA* эффективно организовать процесс тестирования. В этом разделе рассматриваются основные принципы, которые необходимо учитывать при тестировании программного обеспечения.

Первый принцип утверждает, что тестирование демонстрирует наличие дефектов, а не их отсутствие. Этот подход подразумевает, что тестирование снижает вероятность наличия ошибок, однако не может гарантировать, что программное обеспечение полностью свободно от дефектов. Даже если тесты не выявляют ошибок, это не означает, что их нет – возможно, они просто не были обнаружены.

Второй принцип заключается в том, что исчерпывающее тестирование невозможно. Проведение полного тестирования всех возможных комбинаций входных данных, условий и сценариев является физически невыполнимым. Вместо этого следует сосредоточиться на оптимизации объема тестирования, основываясь на анализе рисков и приоритетов, чтобы наиболее эффективно использовать ресурсы.

Принцип раннего тестирования подчеркивает необходимость интеграции тестирования на самых ранних этапах жизненного цикла разработки программного обеспечения. Обнаружение и исправление дефектов на начальных стадиях разработки значительно снижает затраты на их устранение в дальнейшем. Раннее тестирование позволяет минимизировать риски и предотвратить возникновение критичных ошибок на поздних этапах разработки.

Четвертый принцип, известный как кластеризация дефектов, указывает на то, что большинство ошибок сосредоточены в небольшом числе модулей. Этот принцип основан на наблюдении, что значительная часть дефектов связана с относительно небольшой частью кода. Таким образом, тестирование должно быть сосредоточено на наиболее проблемных участках, чтобы обеспечить максимальную эффективность процесса.

Пятый принцип – это парадокс пестицида. Он утверждает, что многократное выполнение одних и тех же тестов со временем теряет свою эффективность, поскольку они перестают выявлять новые дефекты. Для поддержания эффективности

тестирования необходимо регулярно обновлять и адаптировать тестовые сценарии, чтобы они оставались актуальными и могли обнаруживать новые ошибки.

Принцип заблуждения об отсутствии ошибок подчеркивает, что отсутствие обнаруженных дефектов не гарантирует готовности продукта к релизу. Программное обеспечение может быть технически корректным, но при этом не удовлетворять потребностям пользователей или не соответствовать их ожиданиям. Тестирование должно охватывать не только технические аспекты, но и учитывать пользовательский опыт и бизнес-требования.

Наконец, последний принцип гласит, что тестирование зависит от контекста. Подходы к тестированию могут варьироваться в зависимости от типа программного обеспечения и специфики проекта. Для каждого проекта необходимо разрабатывать уникальную стратегию тестирования, учитывающую особенности продукта, его цели и аудиторию.

В совокупности эти принципы обеспечивают основу для построения эффективной стратегии тестирования программного обеспечения. Понимание и применение этих принципов позволяет командам *QA*-инженеров достигать высокого уровня качества программного обеспечения, минимизировать риски и обеспечивать удовлетворенность конечных пользователей.

Типы тестирования

Тестирование программного обеспечения включает в себя множество подходов и методов, каждый из которых решает свои задачи и применяется в различных контекстах разработки. Чтобы систематизировать все многообразие методов, тестирование можно классифицировать по нескольким критериям: глубина покрытия, ширина охвата, знание кода, время и место проведения, степень подготовки, изолированность компонентов, степень автоматизации и объект тестирования.

Начнем с классификации по глубине покрытия. Дымовое тестирование (*Smoke Testing*) представляет собой поверхностную проверку основной функциональности приложения, направленную на подтверждение того, что ключевые функции работают корректно. Например, после внесения изменений в приложение тестировщик проверяет, открываются ли главные страницы и работает ли базовая навигация. Позитивное тестирование (*Positive Testing*) нацелено на проверку корректной работы системы с ожидаемыми и правильными данными. Например, если пользователь вводит корректные данные при регистрации, система должна успешно создать учетную запись. В этом случае проверяется, что система функционирует согласно ожиданиям. Полное тестирование (*Complete Testing*) охватывает как позитивные, так и негативные сценарии. Это более глубокий уровень проверки, при котором тестировщик проверяет, что система справляется как с корректными, так и с некорректными данными. Например, проверка того, что система отклоняет некорректно введенный адрес электронной почты.

По ширине охвата можно выделить регрессионное тестирование (*Regression Testing*), которое направлено на проверку корректности работы приложения после внесения изменений. Например, после добавления новой функции в приложение тестировщик проверяет, что существующий функционал не был нарушен. Валидация дефектов (*Defect Validation*) фокусируется на проверке того, что исправленный дефект действительно устранен. Например, если разработчик сообщает, что баг исправлен, тестировщик проверяет, что ошибка больше не возникает. Тестирование новых функций (*New Features Testing*) направлено на проверку нового функционала. Например, если добавлена новая функция фильтрации товаров в интернет-магазине, тестировщик проверяет, что она работает корректно и не нарушает работу других функций.

Классификация по знанию кода включает методы тестирования «белого ящика» (*White-box Testing*), «черного ящика» (*Black-box Testing*) и «серого ящика» (*Gray-box Testing*). В тестировании «белого ящика» тестировщик имеет доступ к исходному коду и может проверять внутреннюю логику работы системы. Например, проверка работы всех ветвей условия if-else в коде. В тестировании «черного ящика» акцент делается на проверке функциональности без доступа к коду. Например, тестировщик проверяет, что при вводе данных в форму регистрации они корректно обрабатываются, не анализируя при этом, как работает код за кадром. В тестировании «серого ящика» тестировщик частично знает внутреннюю структуру системы, например, имеет доступ к базам данных и проверяет, что данные корректно сохраняются и извлекаются.

По времени и месту проведения различают альфа-тестирование (*Alpha Testing*), бета-тестирование (*Beta Testing*) и приемочное тестирование (*User Acceptance Testing*). Альфа-тестирование проводится на стороне разработчика в условиях, максимально приближенных к реальной эксплуатации. Например, внутренние тестировщики проверяют функциональность приложения, выявляя оставшиеся баги. Бета-тестирование проводится пользователями на стороне клиента для оценки готовности продукта к использованию в реальных условиях. Например, ограниченная группа пользователей тестирует приложение, проверяя его соответствие требованиям и ожиданиям. Приемочное тестирование проводится заказчиком для подтверждения, что система соответствует всем критериям и требованиям. Например, заказчик проверяет, что все функции приложения работают в соответствии с его ожиданиями перед окончательным запуском.

По степени подготовки ПО выделяют интуитивное тестирование (*Ad hoc Testing*) и исследовательское тестирование (*Exploratory Testing*). Интуитивное тестирование предполагает

неформальный подход, когда тестировщик проверяет приложение без заранее подготовленных сценариев, полагаясь на свой опыт и интуицию. Например, тестировщик случайным образом взаимодействует с элементами интерфейса, чтобы выявить неожиданные ошибки. Исследовательское тестирование предполагает создание тестов «на лету», исходя из текущих наблюдений. Например, тестировщик, проверяя одну функцию, замечает аномалию в другой части системы и решает исследовать ее более подробно.

По изолированности компонентов системы тестирование может быть модульным (*Module Testing*), интеграционным (*Integration Testing*) и системным (*System Testing*). Модульное тестирование фокусируется на проверке отдельных компонентов системы. Например, проверка работы функции сложения в калькуляторе. Интеграционное тестирование проверяет взаимодействие между модулями, например, проверка, что после интеграции модуля оплаты с модулем корзины товаров оформление заказа проходит без сбоев. Системное тестирование охватывает всю систему в целом, проверяя ее соответствие бизнес-требованиям. Например, проверка, что процесс покупки от выбора товара до получения подтверждения заказа работает корректно.

По степени автоматизации различают автоматизированное тестирование (*Test Automation*) и ручное тестирование (*Manual Testing*). Автоматизированное тестирование используется для ускорения и упрощения проверки повторяющихся задач. Например, автоматическое тестирование формы входа в систему при каждом обновлении кода. Ручное тестирование, напротив, проводится без инструментов автоматизации, например, тестировщик вручную проверяет работу всех элементов интерфейса на различных устройствах.

Наконец, по объекту тестирования различают функциональное тестирование (*Functional Testing*) и нефункциональное

тестирование (*Non-Functional Testing*). Функциональное тестирование направлено на проверку того, как система выполняет свои основные задачи. Например, проверка, что кнопка «Отправить» на форме обратной связи работает, и данные отправляются на сервер. Нефункциональное тестирование охватывает аспекты, не связанные с функциональностью, такие как производительность, безопасность и удобство использования. Например, проверка, насколько быстро загружается сайт при одновременном доступе большого количества пользователей (Performance Testing), или тестирование безопасности, чтобы убедиться, что данные пользователей надежно защищены от несанкционированного доступа (*Security Testing*).

Таким образом, тестирование программного обеспечения – это многоуровневый процесс, включающий множество различных подходов, каждый из которых позволяет выявить и устранить ошибки на разных этапах разработки и использования продукта. Эти методы помогают обеспечить высокое качество и надежность конечного продукта, соответствие его требованиям и удовлетворение ожиданий пользователей.

Жизненный цикл дефекта

Чтобы эффективно управлять ошибками в программном обеспечении, важно понимать их жизненный цикл (*Defect Lifecycle*). Этот цикл описывает стадии, которые проходит ошибка с момента ее обнаружения до полного разрешения. Знание и понимание жизненного цикла дефекта позволяет правильно описывать, оформлять и закрывать ошибки в системах отслеживания, что, в свою очередь, помогает улучшить качество программного обеспечения.

Жизненный цикл дефекта начинается с момента его обнаружения и продолжается до полного устранения. Этот процесс охватывает несколько этапов, каждый из которых обозначается соответствующим статусом. Понимание этих этапов помогает

эффективно управлять дефектами, правильно описывать их, оформлять и закрывать в системах отслеживания ошибок.

Первым этапом является появление нового дефекта, когда ошибка впервые записывается в систему управления. Этот статус называется «*New*» (новый). Далее отчет об ошибке назначается на определенного пользователя, чаще всего на разработчика, и получает статус «*Assigned*» (назначен). После назначения разработчик приступает к работе над исправлением ошибки, что отражается в статусе «*Open*» (открыт). На этом этапе проводится анализ и редактирование кода, чтобы устранить выявленную проблему.

Когда разработчик вносит необходимые изменения в код и тестирует их самостоятельно, дефект получает статус «*Fixed*» (исправлен). Далее ошибка передается обратно тестировщику для повторного тестирования, в этот момент статус изменяется на «*Pending retest*» (тестирование в режиме ожидания). Тестировщик повторно проверяет измененный код, чтобы убедиться в устранении дефекта, этот этап называется «*Retesting*» (повторное тестирование). Если дефект действительно исправлен, тестировщик ставит статус «*Verified*» (проверен). Если же ошибка снова появляется, она возвращается разработчику с повторным назначением, при этом статус изменяется на «*Reopened*» (переоткрыт). Процесс повторяется до полного устранения дефекта.

В случае успешного устранения дефекта и его подтверждения тестировщиком, ошибка получает статус «*Closed*» (закрыт), что означает, что проблема решена и больше не воспроизводится. Если же в системе обнаруживается идентичный дефект или два разных дефекта возникают из-за одной и той же проблемы, один из них может получить статус «*Duplicate*» (дубликат).

Иногда разработчик может отклонить ошибку, если считает, что она не существенна или не требует исправления, в таком случае дефект получает статус «*Rejected*» (отклонен). В ситуа-

циях, когда исправление ошибки откладывается на будущее, например, из-за низкого приоритета или недостатка времени, дефекту присваивается статус «*Deferred*» (отсрочен). Также возможна ситуация, когда выявленный дефект на самом деле не является ошибкой, а представляет собой запрос на изменение функциональности, такой дефект получает статус «*Not a bug*» (не является багом).

Этапы жизненного цикла дефекта включают обнаружение ошибки тестировщиком, ее запись в систему управления дефектами, назначение разработчику, анализ и корректировку кода, повторное тестирование и окончательное закрытие. Этот процесс может варьироваться в зависимости от используемой системы управления дефектами и требований проекта. По мере прохождения дефекта через различные статусы, в системе управления создаются отчеты, которые позволяют оценивать эффективность работы как программистов, так и тестировщиков, что в итоге способствует повышению качества программного обеспечения.

1.2. ПЛАНИРОВАНИЕ И ПРОЕКТИРОВАНИЕ ТЕСТОВ

Анализ требований и его роль в тестировании

Требования к программному обеспечению – это совокупность запросов и утверждений, касающихся атрибутов, свойств или качеств системы, которую необходимо реализовать. Требования могут быть представлены как в виде текстовых утверждений, так и графических моделей. Обычно они подразделяются на три уровня: бизнес-требования, пользовательские требования и функциональные требования. Также существуют нефункциональные требования, которые описывают атрибуты качества продукта.

Бизнес-требования – это высокоуровневые цели, которые ставит перед собой организация или заказчик. Эти требования

отвечают на вопросы, зачем создается продукт, какие проблемы он решает и какие возможности предоставляет пользователям. Например, система должна собирать данные клиентов и хранить их в базе данных.

Пользовательские требования описывают цели и задачи, которые пользователи смогут выполнять с помощью системы для достижения бизнес-целей. В этих требованиях указывается, как именно пользователи будут взаимодействовать с продуктом, какие действия они смогут выполнять. Например, это может быть описание интерфейса для ввода и отправки данных.

Функциональные требования определяют поведение системы, которое необходимо разработать для выполнения пользовательских и бизнес-требований. Эти требования часто формулируются с использованием слов «должен» или «не должен». Например, «Система должна отправлять данные пользователя в базу данных» или «Система не должна обрабатывать данные при незаполненном поле ввода».

Нефункциональные требования описывают атрибуты качества, такие как производительность, надежность и удобство использования. Эти требования не связаны напрямую с функциями системы, но важны для ее общей работоспособности и удовлетворенности пользователей. Они проверяются через соответствующие виды тестирования.

После определения видов требований тестировщик приступает к их анализу. Главная задача на этом этапе – провести анализ требований, который представляет собой вид статического тестирования. В этом процессе тестировщик работает не с самим программным обеспечением, а с документацией, проверяя требования на соответствие важным критериям. Эти критерии включают полноту, корректность, осуществимость, необходимость, недвусмысленность, модифицируемость и прослеживаемость.

Полнота требований означает, что они должны полностью описывать функциональность, которую следует реализовать. Например, каждое требование должно охватывать все аспекты работы системы, от процессов до отдельных кнопок и полей ввода. Корректность предполагает точное и правильное описание желаемого функционала, соответствующего поставленным задачам. Осуществимость означает, что каждое требование должно быть реальным и выполнимым в рамках известных ограничений и возможностей системы. Необходимость предполагает, что каждое требование должно отражать действительную потребность пользователей и быть актуальным для проекта. Недвусмысленность исключает возможность разной интерпретации требований разными участниками проекта, что особенно важно для разработчиков. Модифицируемость обеспечивает легкость внесения изменений в требования без необходимости корректировать другие связанные элементы. Прослеживаемость позволяет отследить связь между требованием и другими элементами документации, что упрощает управление изменениями.

После завершения анализа требований тестировщик переходит к их декомпозиции. Декомпозиция – это процесс разделения требований на составляющие части, что позволяет легче анализировать и тестировать систему. Декомпозиция часто представляется в виде иерархического древа, которое используется для создания чек-листа проверок программы.

В процессе декомпозиции тестировщик может выбрать различные признаки для разделения требований, такие как интерфейс, пользователи, компоненты программы, используемые данные, состояние сущностей и другие параметры. На каждом уровне декомпозиции можно использовать разные принципы разделения, что позволяет детально анализировать систему и структурировать процесс тестирования. Глубина декомпозиции определяется удобством использования и опытом тестировщика. В среднем, 3 – 6 уровней достаточно для создания чек-листа,

где нижние уровни представляют собой конкретные проверки элементов и функций системы.

Таким образом, анализ и декомпозиция требований – это фундаментальные шаги в процессе тестирования, которые позволяют заранее выявить ошибки и сократить время на их исправление. Правильно проведенный анализ и грамотная декомпозиция обеспечивают качественное тестирование и повышают вероятность успешного релиза продукта.

Разработка тест-плана

Тест-план – это артефакт тестирования, который описывает все действия, происходящие в процессе тестирования, начиная с разработки стратегии и заканчивая критериями завершения задач и оценкой рисков. Он структурирует тестирование, придавая ему определенную логику и обеспечивая четкое понимание задач и ожидаемых результатов всеми членами команды.

Согласно стандарту IEEE 829, тест-план должен состоять из 19 пунктов:

1. Идентификатор тест-плана.
2. Ссылки.
3. Введение.
4. Объекты тестирования.
5. Проблемы и риски.
6. Функции, которые нужно протестировать.
7. Функции, которые не нужно тестировать.
8. Подходы.
9. Критерии прохождения тестов.
10. Критерии остановки и требования для возобновления тестирования.
11. Результаты тестирования.
12. Оставшиеся задачи тестирования.
13. Требования среды.
14. Требования по части кадров и их обучения.

15. Обязанности.
16. Расписание.
17. Планирование рисков и непредвиденных обстоятельств.
18. Утверждение.
19. Глоссарий.

В начале документа указывается идентификатор тест-плана, включающий название компании, название документа, его версию и год создания. Следующий раздел посвящен ссылкам, где фиксируется история документа с указанием всех изменений, что позволяет команде отслеживать процесс его разработки.

Введение служит пояснительной запиской для клиента. В нем кратко описываются цели тестирования и предоставляемые командой *QA* услуги. Это помогает клиенту понять, что будет протестировано и зачем.

Далее следуют объекты тестирования – это те функциональные возможности, которые будут протестированы. Это своего рода краткое содержание тест-плана, которое в дальнейшем будет подробно расписано.

Раздел «Проблемы и риски» описывает возможные трудности, с которыми может столкнуться команда во время тестирования, будь то кадровые вопросы или технологические аспекты. Это важно для того, чтобы заранее предусмотреть возможные препятствия и быть готовыми к их преодолению.

После этого идет описание функций, которые нужно протестировать, с подробным списком и временем, отведенным на их проверку. Также указываются функции, которые не будут тестироваться, с объяснением причин их исключения.

Затем описываются подходы и методы тестирования, которые будут применяться, включая тест-кейсы. Это помогает клиенту получить полное представление о процессе тестирования.

Критерии прохождения тестов определяют, как будут оцениваться результаты каждого тест-кейса, а также устанавливают критерии начала и окончания тестирования. Критерии останов-

ки и возобновления тестирования описывают ситуации, при которых тестирование может быть приостановлено из-за критических багов.

Результаты тестирования фиксируются и представляются клиенту, чтобы засвидетельствовать проведенную работу. Если какие-то задачи остаются невыполненными, они включаются в раздел «Оставшиеся задачи тестирования».

Требования среды и оборудования, используемого для тестирования, описываются в соответствующем разделе. Это включает как аппаратное обеспечение, так и программные инструменты, необходимые для выполнения тестов. Кроме того, если требуется обучение команды для понимания специфики проекта, это также указывается в тест-плане.

Обязанности членов команды описываются в отдельном разделе, чтобы четко распределить задачи и обеспечить эффективное выполнение работы. Расписание тестирования с дедлайнами и этапами работы также должно быть указано, чтобы команда могла оценивать прогресс и соответствовать срокам.

Планирование рисков и непредвиденных обстоятельств описывает меры по преодолению потенциальных проблем и действия в случае форс-мажорных ситуаций. После этого тест-план должен быть утвержден ответственными лицами, что официально дает старт процессу тестирования.

Завершает документ глоссарий, который помогает избежать неправильного толкования используемых терминов.

Несмотря на строгую структуру, существуют разные виды тест-планов, которые могут отличаться в зависимости от уровня и типа тестирования. Например, существуют тест-планы по уровням (модульное, интеграционное, системное, приемочное тестирование) и по типам (функциональное тестирование, тестирование производительности и т.д.). Мастер тест-план – это комплексный документ, который охватывает высокоуровневую информацию и используется для всего проекта.

Важно также различать тест-план и стратегию тестирования. Тест-план более детализирован и охватывает больше аспектов проекта, тогда как стратегия тестирования чаще используется на организационном уровне и редко изменяется. Стратегия обычно является частью тест-плана.

Подводя итоги, можно сказать, что тест-план – это не просто документ, а важный артефакт, который структурирует процесс тестирования и помогает избежать множества возможных проблем. Написание тест-плана требует внимания к деталям, аналитических навыков и способности продумывать действия на несколько шагов вперед. Однако результат стоит затраченных усилий, поскольку хорошо составленный тест-план делает процесс тестирования более организованным, эффективным и прозрачным для всех участников проекта.

Тест-кейсы

Тест-кейсы являются одним из ключевых инструментов в арсенале любого специалиста по тестированию программного обеспечения. Они позволяют структурировать процесс проверки, минимизировать риски пропуска ошибок и обеспечивать высокое качество конечного продукта.

Тест-кейс – это алгоритм действий, которые нужно выполнить для проверки работы программы или ее отдельных компонентов, таких как кнопки, поля ввода и пр. Каждый тест-кейс включает в себя несколько обязательных элементов: предусловия, шаги проверки, ожидаемый результат и постусловия. Основной целью тест-кейса является обеспечение последовательной и детализированной проверки определенной функциональности программы.

Согласно определению, данному в словаре терминов *ISTQB*, тест-кейс представляет собой «набор входных значений, предусловий выполнения, ожидаемых результатов и

постусловий выполнения, разработанный для определенной цели или тестового условия, таких как выполнение определенного пути программы или проверка соответствия определенному требованию».

Тест-кейсы используются в крупных проектах, где некорректная работа системы может иметь серьезные последствия. Например, в проектах, связанных с медицинским оборудованием, финансовыми системами или системами обеспечения безопасности. В таких случаях тест-кейсы обеспечивают высокую степень детализации и точности в процессе проверки программного обеспечения.

Каждый тест-кейс должен содержать ряд обязательных атрибутов, которые обеспечивают его эффективность и пригодность к использованию. К ним относятся:

– уникальный идентификационный номер состоит из комбинации букв и цифр, что позволяет однозначно идентифицировать тест-кейс среди других;

– заголовок описывает цель тест-кейса, например, «Проверка входа пользователя с верными данными»;

– предусловия – шаги, которые должны быть выполнены перед началом тест-кейса. Например, авторизация пользователя или запуск приложения;

– шаги: подробное изложение действий, которые необходимо выполнить для проведения проверки;

– постусловия – действия, необходимые для восстановления системы до исходного состояния после завершения тестирования;

– ожидаемый результат – описание того, что должно произойти после выполнения всех шагов тест-кейса;

– фактический результат – состояние системы после выполнения тест-кейса. Этот атрибут не всегда обязателен;

– статус: успех (Success), провал (Failed) или блокировка (Blocked) тест-кейса. Отмечается, если требуется.

К некоторым тест-кейсам могут добавляться дополнительные атрибуты, такие как требования к среде (специфическое оборудование или ПО, необходимые для выполнения тест-кейса), специальные процедурные требования (особые настройки или процессы) и межкейсовые зависимости (тест-кейсы, которые должны быть выполнены до данного).

Тест-кейсы особенно важны в тех случаях, когда некорректная работа системы может привести к серьезным последствиям, например, в системах, связанных с пожарной безопасностью, медицинским обслуживанием или финансовой сферой. В таких проектах требуется более глубокое тестирование, и тест-кейсы играют важную роль в обеспечении качества продукта.

В проектах, где некорректная работа системы не несет критических рисков, а команда тестировщиков небольшая и хорошо знакома с продуктом, часто используются чек-листы. Это позволяет сэкономить время и ресурсы, так как написание и обслуживание тест-кейсов может быть трудозатратным и не всегда оправданным.

Тест-кейсы имеют свои преимущества и недостатки, которые важно учитывать при выборе метода тестирования.

Преимущества:

1. Универсальность. Тест-кейс может выполнить любой сотрудник, даже если он не принимал участия в разработке проекта. Достаточно следовать инструкциям, чтобы провести тестирование.

2. Структурированность. Тест-кейсы обеспечивают систематизированный подход к тестированию, что снижает вероятность пропуска ошибок.

Недостатки:

1. Затраты времени. Написание и обслуживание тест-кейсов требует значительных временных ресурсов, особенно если проект динамично развивается и часто меняется.

2. Монотонность работы. Выполнение одинаковых шагов в разных тест-кейсах может быть утомительным и демотивирующим.

3. Риск неактуальности. Тест-кейсы могут устаревать, особенно если проект быстро меняется. Это требует постоянного обновления и проверки актуальности тест-кейсов.

Тест-кейсы можно разделить на несколько групп в зависимости от типа проверяемых данных и предполагаемого поведения системы:

1. Позитивные тест-кейсы. Проверяют, что система работает корректно при введении правильных данных и выполнении всех условий.

2. Отрицательные тест-кейсы. Проверяют, как система реагирует на некорректные данные или действия, например ввод неправильного пароля или оставление обязательного поля пустым.

3. Деструктивные тест-кейсы. Проверяют способность системы справляться с экстремальными условиями, такими как высокая нагрузка или неожиданное отключение питания.

Пример позитивного тест-кейса может включать проверку добавления нового филиала в системе записи клиентов салона красоты при введении корректных данных. Отрицательный тест-кейс проверит, как система реагирует на ввод неполного адреса или дублирование информации. Деструктивный тест-кейс проверит устойчивость системы к критическим нагрузкам, таким как одновременное добавление большого числа мастеров.

Для создания эффективных тест-кейсов необходимо соблюдать ряд правил:

– заголовок должен быть четким и кратким, не содержащим шагов выполнения или ожидаемого результата;

– предусловие должно подробно описывать состояние системы перед началом тестирования, но не включать ссылки на конкретные среды или данные авторизации;

– шаги проверки должны быть описаны четко, последовательно и доступным языком, без излишней детализации;

– ожидаемый результат должен быть указан для каждого шага проверки и описывать состояние системы после его выполнения;

Соблюдение этих правил поможет составить тест-кейсы, которые будут одинаково удобны в использовании для всех сотрудников проекта и помогут избежать распространенных ошибок.

Среди наиболее частых ошибок при создании тест-кейсов можно выделить следующие:

– слишком обобщенное название – это создает путаницу между различными тест-кейсами одного проекта;

– употребление повелительного наклонения – важно использовать нейтральные формулировки, чтобы избежать лишнего давления на исполнителя;

– некликабельные ссылки – все ссылки должны быть кликабельными для удобства использования;

– слишком подробные описания действий – не нужно описывать очевидные шаги, это усложняет восприятие;

– слишком краткое описание действий – недостаток информации также может привести к непониманию и ошибкам.

Тест-кейсы – это важный инструмент, который делает процесс тестирования программного обеспечения более структурированным и доступным для неспециалистов. Они позволяют обеспечить высокое качество проверки и минимизировать риски возникновения ошибок в работе системы. Однако важно помнить, что тест-кейсы требуют регулярного обновления и тщательной проверки, чтобы оставаться актуальными и эффективными.

При правильном подходе к разработке и использованию тест-кейсов они становятся мощным средством обеспечения

качества программного обеспечения, помогая командам достигать поставленных целей и удовлетворять требования пользователей.

Чек-листы

Чек-лист представляет собой структурированный список проверок, который используется для тестирования программного обеспечения. Он содержит перечень элементов, таких как блоки, секции, страницы и другие компоненты приложения, которые подлежат проверке. Чек-листы служат инструментом для организации процесса тестирования, обеспечивая упорядоченный подход к проверке функциональности и характеристик системы.

Классический чек-лист обычно включает в себя три основных элемента: заголовок, статус и заметку. Заголовок определяет, какой именно элемент системы подлежит проверке, статус отображает результат тестирования, а заметка может содержать дополнительные комментарии или уточнения по проведенному тесту. Статусы в чек-листе могут варьироваться, включая такие категории, как «Пройдено» (*Passed*), «Не пройдено» (*Failed*), «Заблокировано» (*Blocked*), «Пропущено» (*Skipped*) и «Не проводился» (*Not run*). Эти статусы позволяют легко отслеживать текущее состояние тестирования и быстро идентифицировать проблемные области, требующие внимания.

Одним из ключевых преимуществ использования чек-листов является их наглядность и компактность. Чек-листы предоставляют ясное и удобное представление о том, какой объем работы уже выполнен, а также о том, какие проверки еще предстоит выполнить. Это особенно важно в условиях ограниченного времени, когда необходимо быстро оценить прогресс и определить оставшиеся задачи перед сдачей или приемкой проекта.

Однако важно понимать, что чек-лист не заменяет собой тест-кейсы. В отличие от тест-кейсов, которые предоставляют подробное описание шагов тестирования, включая алгоритмы и сценарии, чек-листы фокусируются на перечне направлений тестирования. Они указывают, что именно нужно протестировать, но не дают подробных инструкций по тому, как это делать. В этом заключается основное различие между этими двумя инструментами.

Чек-листы проще и быстрее в составлении, что делает их удобными для использования на начальных этапах тестирования или в случаях, когда требуется провести общую проверку системы. Однако их применение может быть сложнее для новичков, так как они требуют от тестировщика глубокого понимания функционала и контекста работы системы. Опытному тестировщику, знакомому с продуктом и процессом, будет легко ориентироваться по чек-листу и проводить тестирование, тогда как новый специалист может столкнуться с трудностями в понимании сути тестируемых элементов без детальной информации, предоставляемой тест-кейсами.

Таким образом, чек-листы являются полезным инструментом в арсенале тестировщика, особенно в тех случаях, когда необходимо быстро и эффективно оценить состояние системы. Однако для полноценного и глубокого тестирования они должны использоваться в сочетании с тест-кейсами, которые предоставляют более детальные инструкции и помогают избежать пропуска важных проверок.

Варианты использования

Варианты использования (*Use case*) представляют собой сценарии взаимодействия пользователя с системой, в которых описывается, как именно программа должна работать в конкретных ситуациях. Они служат важным инструментом в процессе разработки программного обеспечения, помогая каждому

участнику проекта лучше понимать, как система должна функционировать и какие задачи она решает.

Для заказчика варианты использования особенно полезны, поскольку в них отражается конечная бизнес-ценность, которую система должна предоставить. Вариант использования описывает действия пользователя и результаты, которых он ожидает, что делает реализацию сценария понятной даже для тех, кто не обладает техническими знаниями. Например, заказчик может легко понять, как пользователь должен оформить заказ в интернет-магазине или как система обрабатывает запросы на получение отчетов. Это помогает заказчику убедиться, что разрабатываемая система действительно решает его бизнес-задачи.

Для разработчиков варианты использования предоставляют наглядное представление бизнес-логики и поведения системы. Они описывают, как различные компоненты системы взаимодействуют друг с другом и с пользователем, какие данные вводятся и какие результаты должны быть получены. Это помогает разработчикам лучше понять требования и эффективно реализовать их в коде. Варианты использования также облегчают процесс планирования и определения объема работ, поскольку четко показывают, какие функциональные возможности должны быть разработаны.

Для тестировщиков варианты использования являются ценной основой для создания тест-кейсов. Варианты использования представляют собой пригодные для тестирования требования, где четко указаны цели, которые система должна достигать, и пути их достижения. Тестирование по сценариям использования, или use case testing, позволяет выявить в приложении недостатки, которые могут быть упущены при других видах тестирования, таких как юнит-тестирование. Например, тестировщик может использовать вариант использования для проверки полного сценария оформления заказа в интернет-магазине, включая добавление товаров в корзину, выбор способа оплаты, подтвер-

ждение заказа и получение уведомления о его успешном оформлении. Такой подход позволяет обнаружить проблемы, связанные с интеграцией различных частей системы, а также с логикой обработки данных.

Таким образом, варианты использования играют ключевую роль в процессе разработки программного обеспечения, обеспечивая ясность и понимание требований для всех участников проекта – от заказчика до разработчика и тестировщика. Они помогают создать продукт, который отвечает бизнес-требованиям и удовлетворяет потребности пользователей.

Баг-репорты

Баг-репорт представляет собой документ, в котором содержится полная информация о выявленном дефекте в программном обеспечении. Он включает в себя все необходимые данные, такие как шаги воспроизведения ошибки, ее описание, локализация и другие важные детали. Подробное и точное описание ошибки в баг-репорте играет ключевую роль в ее быстром устранении и корректной перепроверке.

Создавая баг-репорт, важно максимально точно локализовать ошибку, т.е. определить, в какой части программы или на каком этапе ее использования возникает проблема. Также рекомендуется проверить наличие ошибки на разных устройствах и версиях программного обеспечения, чтобы определить, насколько она распространена и в каких условиях возникает. Описание несоответствия фактического поведения программы ожидаемому результату должно быть максимально четким и понятным, чтобы разработчики могли быстро понять суть проблемы.

Баг-репорт является неотъемлемой частью любого проекта, вне зависимости от того, пишутся ли другие тестовые документы. Правильность составления баг-репорта напрямую влияет

на скорость понимания проблемы разработчиками и, соответственно, на качество ее устранения. Если баг-репорт составлен нечетко или содержит недостаточную информацию, это может привести к задержкам в исправлении ошибки или даже к ее неправильной интерпретации. Таким образом, качественно составленный баг-репорт способствует более эффективной работе команды и улучшению общего качества программного продукта.

1.3. ВЫПОЛНЕНИЕ ТЕСТОВ И ОТЧЕТНОСТЬ

Подготовка к выполнению тестов

Подготовка к тестированию является важным этапом в процессе обеспечения качества программного обеспечения. Хотя этот этап часто может пересекаться с этапом разработки тестов и выполняться параллельно с ним, для лучшего понимания мы рассматриваем его отдельно. На этом этапе обычно выполняются несколько ключевых задач.

Во-первых, создается тестовое окружение, в котором настраивается необходимая конфигурация для проведения тестов. Это включает установку всех компонентов системы, настройку серверов и других элементов инфраструктуры. Также на этом этапе происходит развертывание версии приложения, которая включает в себя все необходимые изменения для проведения тестирования. Это позволяет убедиться, что тестируемая версия содержит все исправления и обновления, которые нужно проверить.

Кроме того, важно подготовить тестовые данные, которые будут использоваться в процессе тестирования. Тестовые данные могут включать в себя реальные или синтетические данные, которые необходимы для проверки работы системы в различных сценариях. Важно, чтобы эти данные были актуальными и соответствовали требованиям тестов.

Для проектов, где используется автоматизация тестирования, на этом этапе также могут дорабатываться скрипты автоматизации, чтобы они соответствовали новой версии системы. Это включает в себя обновление тестовых сценариев и их адаптацию к измененным компонентам системы.

После завершения всех подготовительных работ проводится проверка работоспособности приложения. Это делается с помощью базового набора тестов, которые позволяют убедиться, что система корректно функционирует после развертывания. Эта проверка помогает выявить грубые ошибки, которые могли возникнуть при развертывании, и минимизировать риски.

Этот этап подготовки к тестированию является универсальным для любого подхода к разработке, но его продолжительность и сложность могут варьироваться в зависимости от типа приложения, его архитектуры и зрелости процессов разработки. В некоторых случаях все подготовительные работы могут занять всего несколько минут, в других – могут потребовать несколько дней.

Результатом этого этапа является готовое тестовое окружение с подготовленными данными и установленным приложением, которое готово к полноценному тестированию.

Ручное и автоматизированное тестирование

Ручное и автоматизированное тестирование – два основных подхода, которые используют для обеспечения качества программного обеспечения. Каждый из них имеет свои особенности, преимущества и ограничения, что делает их подходящими для разных задач в зависимости от требований проекта.

Ручное тестирование предполагает выполнение всех шагов тестового сценария вручную, что означает непосредственное участие тестировщика в процессе проверки. Этот подход позволяет глубже понять, как работает приложение, и выявить про-

блемы, которые могут быть незаметны при автоматизированных тестах. Например, если тестировщик проверяет функцию входа в систему на веб-сайте, он вручную вводит данные, такие как имя пользователя и пароль, нажимает на кнопку «Войти» и оценивает, что происходит далее. Если при этом обнаруживается, что система не перенаправляет пользователя на главную страницу или отображает некорректное сообщение об ошибке, тестировщик фиксирует проблему и сообщает о ней разработчикам.

Один из главных плюсов ручного тестирования – это возможность оценки субъективных факторов, таких как удобство использования и интуитивность интерфейса. Например, в процессе тестирования мобильного приложения тестировщик может обнаружить, что кнопки на экране слишком малы для удобного нажатия или что навигация по меню запутанная и нелогичная. Такие нюансы трудно учесть при автоматизированном тестировании, но они могут существенно повлиять на пользовательский опыт.

Однако у ручного тестирования есть и свои минусы. Оно требует значительных временных и трудовых затрат, так как каждый тест выполняется вручную. Например, проверка функции корзины на сайте интернет-магазина может включать в себя последовательное добавление товаров, изменение их количества, удаление и повторное добавление, а также проверку работы скидок и промокодов. Выполнение всех этих действий вручную может занять много времени, особенно, если таких сценариев много. Это делает процесс довольно медленным и дорогим, особенно в крупных проектах с большим количеством функциональных сценариев.

Автоматизированное тестирование, в свою очередь, использует специализированные инструменты и фреймворки для автоматического выполнения тестовых сценариев. Этот подход особенно полезен в тех случаях, когда необходимо часто повторять

одни и те же тесты, например, при регрессионном тестировании. Регрессионное тестирование необходимо для проверки того, что после внесения изменений или добавления новых функций, существующий функционал продолжает работать корректно. Автоматизация позволяет сократить время тестирования и повысить его эффективность, что особенно важно в условиях жестких сроков.

Например, в случае автоматизированного тестирования системы управления складом, можно настроить скрипты, которые автоматически проверят все основные функции: добавление новых товаров на склад, учет их количества, перемещение между различными секциями склада и списание. Такие тесты можно запускать после каждого обновления системы, и они будут выполняться гораздо быстрее, чем если бы все эти проверки проводились вручную.

Автоматизация также исключает возможность человеческой ошибки. Например, если тестировщику необходимо проверить, как система обрабатывает различные комбинации входных данных, автоматизированный скрипт сможет сделать это с высокой точностью, тогда как ручное тестирование может привести к пропуску некоторых комбинаций или некорректному вводу данных.

Однако автоматизированное тестирование требует значительных первоначальных инвестиций. Например, для разработки и поддержки тестовых скриптов требуется время и ресурсы, а также необходимы специальные навыки у сотрудников. Инструменты автоматизации могут быть сложными в освоении, и их настройка требует значительных усилий. Кроме того, автоматизация подходит не для всех видов тестов. Некоторые аспекты, такие как визуальные элементы интерфейса или пользовательский опыт, сложно оценить с помощью автоматизированных инструментов.

Рассмотрим конкретные примеры использования ручного и автоматизированного тестирования в реальных проектах.

Ручное тестирование часто используется в проектах, где важна оценка пользовательского интерфейса или специфических сценариев, которые сложно автоматизировать. Например, при тестировании приложения для онлайн-банкинга, ручное тестирование может включать проверку, как система реагирует на нестандартные действия пользователя, такие как быстрое переключение между вкладками или попытка одновременно выполнить несколько транзакций. Такие ситуации сложно предсказать и автоматизировать, поэтому ручное тестирование позволяет выявить проблемы, которые могут остаться незамеченными при автоматизированных тестах.

Автоматизированное тестирование эффективно используется в крупных проектах с повторяющимися задачами. Например, в случае разработки веб-приложения для электронной коммерции, автоматизация может включать проверку процесса оформления заказа, интеграции с платежными системами и генерации отчетов о продажах. Каждый раз, когда в системе вносятся изменения, автоматизированные тесты могут быть запущены для проверки того, что все эти процессы работают корректно, что значительно ускоряет процесс тестирования и снижает риск пропуска ошибок.

Выбор между ручным и автоматизированным тестированием зависит от специфики проекта и целей тестирования. Ручное тестирование лучше подходит для ситуаций, где важны субъективные оценки и гибкость, а автоматизация эффективна для задач, требующих частого повторения одних и тех же проверок. В большинстве случаев наилучшие результаты достигаются при комбинировании обоих подходов, что позволяет использовать их сильные стороны для обеспечения высокого качества программного продукта.

Создание отчетной документации

Отчет по тестированию – это важный документ, который фиксирует проделанную работу и результаты тестирования программного обеспечения. Такой отчет играет ключевую роль в процессе разработки и контроля качества, поскольку он помогает участникам проекта – разработчикам, менеджерам и заказчикам – понимать текущее состояние продукта, оценивать его готовность к выпуску и принимать решения о дальнейших действиях. Отчеты могут содержать разнообразную информацию, представленную в виде текста, таблиц, графиков и диаграмм, что позволяет эффективно передать результаты работы и акцентировать внимание на ключевых аспектах.

Форма и содержание отчета по тестированию могут варьироваться в зависимости от аудитории, для которой он предназначен. Для руководства могут быть важны общие выводы и ключевые показатели, тогда как для команды разработчиков и тестировщиков необходима более детализированная информация о выявленных дефектах, выполненных тестах и оставшихся задачах. Соответственно, каждый отчет может быть адаптирован под конкретные потребности, что делает его полезным инструментом для всех участников процесса.

Отчет по инциденту – это документ, который фиксирует события, произошедшие во время тестирования, и требует дальнейшего изучения. В процессе тестирования программного обеспечения могут возникнуть различные инциденты, такие как неожиданные сбои системы, проблемы с производительностью, некорректное поведение программы или ошибки в коде. Каждый такой инцидент необходимо задокументировать, чтобы можно было проанализировать его причины и разработать меры по их устранению. В отчете по инциденту подробно описывается, что именно произошло, при каких условиях, как это повлия-

ло на работу системы и какие шаги были предприняты для устранения проблемы. Важно не только зафиксировать сам факт инцидента, но и дать рекомендации по предотвращению подобных ситуаций в будущем. Такой отчет помогает команде быстро реагировать на проблемы, устранять их и минимизировать влияние на общий процесс разработки.

Отчет о результатах тестирования представляет собой периодический документ, в котором фиксируется подробная информация о проведенных тестах, их результатах и оставшихся задачах. Этот отчет создается по завершении определенного этапа тестирования или в рамках регулярной отчетности, например, ежедневно или еженедельно. В нем содержится информация о том, какие тесты были выполнены, какие из них прошли успешно, а какие завершились с ошибками, сколько дефектов было обнаружено и устранено. Кроме того, в отчете о результатах тестирования часто указывается статус оставшихся тестов, их приоритет и планируемые сроки выполнения. Этот документ помогает команде отслеживать прогресс тестирования, выявлять узкие места и принимать решения о необходимости дополнительных проверок или изменений в процессе. Отчет о результатах тестирования также полезен для управления рисками, так как позволяет своевременно обнаруживать потенциальные проблемы и корректировать план работы.

Отчет о ходе тестирования – это документ, в котором подводятся промежуточные итоги тестирования в целях отслеживания общего прогресса проекта. Такой отчет обычно включает в себя обзор текущего состояния тестирования, описание достигнутых целей и задач, а также информацию о том, какие этапы еще предстоит завершить. Важно, что отчет о ходе тестирования позволяет увидеть динамику процесса, понять, насколько проект продвигается по плану, и где могут возникнуть задержки или трудности. Этот отчет часто используется

на встречах команды или при общении с заказчиком для предоставления актуальной информации о состоянии проекта. В нем могут быть приведены данные о выполнении ключевых тестов, количество обнаруженных и исправленных дефектов, а также оценка того, насколько продукт готов к релизу. Отчет о ходе тестирования помогает поддерживать прозрачность процесса разработки и вовремя вносить необходимые корректировки в план работы.

Итоговый отчет о тестировании представляет собой всеобъемлющий документ, который содержит полную информацию о тестировании, проведенном на протяжении всего жизненного цикла разработки программного обеспечения. В этом отчете подробно описывается каждый этап тестирования, начиная с планирования и подготовки тестовой среды, и заканчивая итоговыми результатами и выводами. Итоговый отчет включает в себя сводку всех обнаруженных и устраненных дефектов, описание использованных методологий и инструментов, а также оценку качества продукта на основе проведенных тестов.

Итоговый отчет о тестировании важен для подведения окончательных итогов проекта и принятия решения о готовности продукта к выпуску. Он служит основой для анализа эффективности процесса тестирования, помогает выявить сильные и слабые стороны работы команды и разработать рекомендации для будущих проектов. Кроме того, этот отчет может использоваться как официальный документ для передачи заказчику или другим заинтересованным сторонам, подтверждающий, что продукт прошел все необходимые проверки и соответствует установленным требованиям.

Таким образом, отчеты по тестированию являются неотъемлемой частью процесса разработки программного обеспечения, обеспечивая прозрачность, контроль качества и поддержку принятия решений на всех этапах жизненного цикла продукта.

1.4. ТЕСТИРОВАНИЕ НА РАЗЛИЧНЫХ ЭТАПАХ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Тестирование программного обеспечения – это многоуровневый процесс, который охватывает все этапы жизненного цикла разработки продукта. Он начинается с анализа требований и продолжается до приемки готового продукта, обеспечивая его качество и соответствие ожиданиям пользователей и бизнеса. Тестирование является неотъемлемой частью каждого этапа разработки, начиная с проверки правильности и полноты требований и заканчивая окончательной проверкой продукта перед его выпуском.

Первым важным этапом является тестирование требований. Это критически важный процесс, который позволяет убедиться, что все требования к системе четко определены, являются полными и корректными. На этом этапе тестировщики проверяют, чтобы требования не содержали двусмысленностей и не допускали различных интерпретаций. Например, если требование описывает скорость обработки данных системой, оно должно содержать конкретные цифры, чтобы избежать разночтений между заказчиком и разработчиком. Тестирование требований помогает заранее выявить потенциальные проблемы и уточнить области, которые могут вызвать трудности на последующих этапах разработки.

После того, как требования сформулированы и утверждены, начинается разработка дизайна системы. Тестирование на этом этапе направлено на проверку проектных решений, архитектуры и макетов системы. Здесь важно убедиться, что предложенные решения соответствуют требованиям и смогут поддерживать все необходимые функции и нагрузки. Например, если в дизайне предусмотрена работа с большими объемами данных, тестировщики проверяют, насколько архитектура системы способна справляться с этими нагрузками. Это позволяет выявить потен-

циальные проблемы еще до начала кодирования, что в дальнейшем сокращает затраты на исправление ошибок.

Когда начинается процесс написания кода, наступает время для модульного тестирования. Этот этап фокусируется на проверке отдельных компонентов системы. Модульное тестирование позволяет убедиться, что каждый модуль программы работает корректно в соответствии с требованиями. Например, если разработчик создает функцию для обработки платежей, модульное тестирование проверяет, правильно ли она работает со всеми возможными комбинациями входных данных. Этот этап особенно важен, так как он позволяет выявить и устранить ошибки на ранних стадиях разработки, что значительно упрощает интеграцию компонентов и снижает риски.

Следующим важным этапом является интеграционное тестирование, которое проверяет взаимодействие между различными модулями системы. Даже если каждый модуль работает корректно по отдельности, их совместная работа может вызывать ошибки. Например, проблемы могут возникнуть при передаче данных между модулями или при взаимодействии с внешними сервисами. Интеграционное тестирование позволяет выявить и исправить эти проблемы, прежде чем они приведут к сбоям на более поздних этапах разработки.

Нагрузочное тестирование – важная часть процесса, которая проверяет, как система ведет себя под высокой нагрузкой. Этот вид тестирования особенно важен для приложений, которые будут использоваться большим числом пользователей одновременно. Нагрузочное тестирование помогает определить, как система справляется с пиковыми нагрузками, как быстро она реагирует на запросы и не теряет ли данные при обработке большого объема информации. Например, для веб-приложения нагрузочное тестирование может включать проверку работы сайта при одновременном доступе к нему тысяч пользователей.

Это тестирование позволяет выявить узкие места в производительности системы и внести необходимые улучшения до выпуска продукта.

Системное тестирование является следующим этапом, на котором проверяется работа всей системы в целом. Этот этап охватывает все аспекты работы программного обеспечения, включая функциональность, производительность, безопасность и удобство использования. Системное тестирование проводится в условиях, максимально приближенных к реальной среде эксплуатации, что позволяет выявить ошибки, которые могли остаться незамеченными на предыдущих этапах. Например, тестировщики могут проверять, насколько стабильно работает система при длительном использовании, как она справляется с большими объемами данных и защищена ли она от внешних угроз.

Завершающим этапом является приемочное тестирование, которое проводится с целью окончательной проверки системы перед ее передачей заказчику. На этом этапе проверяются ключевые функции и процессы системы, чтобы убедиться, что они работают корректно и соответствуют требованиям. Например, заказчик может запросить проверку процесса оформления заказа в интернет-магазине, включая все этапы от выбора товара до получения подтверждения о покупке. Успешное прохождение приемочного тестирования означает, что продукт готов к выпуску и может быть внедрен в эксплуатацию.

Таким образом, тестирование на различных этапах разработки программного обеспечения обеспечивает высокое качество конечного продукта, помогает выявлять и устранять ошибки на ранних стадиях, минимизирует риски и повышает удовлетворенность пользователей. Каждая фаза тестирования вносит свой вклад в создание надежной и функциональной системы, которая соответствует ожиданиям бизнеса и пользователей.

1.5. АВТОМАТИЗИРОВАННЫЕ СРЕДСТВА УПРАВЛЕНИЯ ТЕСТИРОВАНИЕМ

Инструменты тестирования играют важную роль в этом процессе, помогая разработчикам и тестировщикам находить и устранять дефекты, проверять функциональность, производительность и надежность ПО в различных условиях. Эти инструменты позволяют оптимизировать процесс тестирования, ускоряя его и делая более точным, что особенно важно для создания качественных и конкурентоспособных продуктов.

Одним из наиболее распространенных методов тестирования является ручное тестирование. Этот метод предполагает, что тестировщик вручную выполняет тест-кейсы, имитируя действия пользователя. Ручное тестирование особенно полезно на ранних стадиях разработки или в ситуациях, когда необходимо проверить новый функционал или пользовательский интерфейс. Например, тестировщик вручную проверяет процесс регистрации на веб-сайте, вводя данные в поля формы, нажимая кнопку «Зарегистрироваться» и проверяя, что система корректно обрабатывает введенные данные и создает учетную запись.

Для организации ручного тестирования и управления процессом тестирования используются различные инструменты. Одним из таких инструментов является ***Trello*** – платформа для управления проектами, которая помогает тестировщикам отслеживать задачи и организовывать рабочий процесс. ***TestRail*** – еще один важный инструмент, позволяющий структурировать тест-кейсы, отслеживать их выполнение и анализировать результаты. ***Test IT*** – современная система управления тестированием, которая упрощает работу с тест-кейсами и хранит результаты в базе данных, что позволяет легко анализировать и просматривать их. ***Jira*** – популярный инструмент для управления проектами и отслеживания багов, который широко используется разработчиками и тестировщиками. Этот инструмент

позволяет эффективно управлять задачами и интегрироваться с другими системами тестирования. ***Zephyr*** – система управления тестированием, основанная на концепции рабочих столов и приборных панелей, что позволяет команде тестировщиков работать с централизованного сервера и быть в курсе всего процесса тестирования в режиме реального времени. ***Postman*** – мощный инструмент для тестирования API, который позволяет тестировщикам легко создавать, тестировать и документировать API, выполняя различные типы HTTP-запросов и преобразовывая их в код на различных языках программирования.

Автоматизированное тестирование является не менее важным методом, особенно в крупных проектах, где необходимо многократное выполнение тестов. Этот метод тестирования предполагает использование специальных инструментов и скриптов для автоматизации процесса проверки функциональности ПО. Автоматизация позволяет значительно ускорить тестирование, особенно когда речь идет о повторяющихся или рутинных задачах. Например, с помощью автоматизированного тестирования можно проверить форму входа на веб-сайт, где необходимо многократно вводить различные комбинации данных и проверять реакцию системы.

Для автоматизированного тестирования существуют различные инструменты. ***Appium*** – это фреймворк с открытым исходным кодом, который используется для автоматизации тестирования мобильных приложений на платформах *Android*, *iOS* и Windows. Этот инструмент поддерживает автоматизацию тестирования как нативных, так и гибридных мобильных приложений, а также веб-приложений, доступ к которым можно получить через мобильные браузеры. ***Jenkins*** – это платформа автоматизации с открытым исходным кодом, которая используется для непрерывной интеграции и тестирования программного обеспечения. Этот инструмент позволяет разработчикам внед-

рять изменения в проект, автоматизировать сборку, тестирование и развертывание кода, что ускоряет выпуск новых версий продукта. ***Pytest*** – популярный фреймворк для тестирования на языке Python, который упрощает написание и выполнение тест-кейсов и помогает составлять отчеты. ***Selenium*** – один из самых известных инструментов для автоматизированного тестирования веб-приложений. Он поддерживает написание тестов на разных языках программирования и их выполнение на различных браузерах и операционных системах. ***Cucumber*** – это инструмент для автоматизированного тестирования с открытым исходным кодом, который позволяет создавать тест-кейсы на понятном языке, что делает его доступным даже для специалистов без глубоких технических знаний.

Нагрузочное тестирование играет ключевую роль в проверке производительности программного обеспечения, особенно в условиях высокой нагрузки. Этот метод тестирования позволяет симулировать одновременную работу большого числа пользователей и оценить, насколько хорошо система справляется с увеличением объема запросов. Например, с помощью нагрузочного тестирования можно проверить, как интернет-магазин справляется с резким увеличением числа посетителей во время распродаж, что помогает выявить потенциальные проблемы с производительностью и стабильностью системы.

Среди инструментов для нагрузочного тестирования можно выделить ***Apache JMeter*** – мощный инструмент с открытым исходным кодом, который предназначен для нагрузочного тестирования поведения программного обеспечения и измерения его производительности. Этот инструмент позволяет имитировать высокую нагрузку на сервер или группу серверов, чтобы проверить их стабильность и проанализировать производительность при различных типах нагрузки. ***LoadView*** – облачный инструмент для нагрузочного тестирования, который симу-

лирует поведение пользователей на сайте или сервере, выполняя определенные действия, такие как просмотр страниц, поиск, добавление товаров в корзину, и генерируя запросы для проверки производительности системы. **LoadNinja** – еще один облачный инструмент, который позволяет тестировщикам имитировать реальные действия пользователей через веб-браузеры, а также записывать и выполнять тесты без необходимости написания кода. **Gatling** – современный инструмент для нагрузочного тестирования, который использует исходный код для моделирования взаимодействия пользователей с веб-приложением, а также предоставляет интерактивные HTML-отчеты в режиме реального времени. **Neoload** – инструмент, который помогает тестировщикам проверять стабильность и производительность ПО под различными условиями рабочей нагрузки и интегрируется с процессами непрерывной интеграции (CI/CD).

Таким образом, выбор правильных инструментов для тестирования программного обеспечения играет решающую роль в успехе разработки. Использование современных инструментов тестирования помогает ускорить процессы проверки, обеспечить надежность и стабильность программного обеспечения, а также предоставить пользователям качественный и конкурентоспособный продукт. Важно учитывать совместимость инструментов с различными платформами, наличие технической поддержки, возможности по генерации отчетов и их стоимость, чтобы выбрать наиболее подходящие решения для конкретного проекта.

2. ПОГРУЖЕНИЕ В АВТОМАТИЗИРОВАННОЕ ТЕСТИРОВАНИЕ

2.1. ВВЕДЕНИЕ В АВТОМАТИЗИРОВАННОЕ ТЕСТИРОВАНИЕ

Основные понятия и определения

Автоматизированное тестирование – это набор методик, подходов и инструментов, призванных сократить рутинные задачи тестирования, освобождая время и ресурсы для более сложных задач. Одной из областей применения автоматизации является регрессионное тестирование. С каждой новой сборкой проекта растет число тест-кейсов, необходимых для проверки того, что ранее работавшая функциональность по-прежнему работает корректно. Автоматизация позволяет избежать ручного повторения этих тестов.

Инсталляционное тестирование и настройка тестового окружения также выигрывают от автоматизации. Подготовка тестового окружения часто включает множество повторяющихся операций, таких как проверка работы инсталлятора, размещение файлов, настройка конфигурационных файлов и реестра. Автоматизация этих задач делает их более эффективными.

Конфигурационное тестирование и тестирование совместимости требуют выполнения одних и тех же тест-кейсов на большом множестве входных данных под разными платформами и в разных условиях. Например, если в файле настроек есть 100 параметров, каждый из которых может принимать 100 значений, существует 100^{100} вариантов конфигурационного файла. Ручная проверка всех этих вариантов нереальна, тогда

как автоматизация позволяет эффективно проверить все возможные сценарии.

Комбинаторные техники тестирования, включая доменное тестирование, предусматривают генерацию комбинаций значений и многократное выполнение тест-кейсов с использованием этих сгенерированных комбинаций в качестве входных данных. Модульное тестирование, направленное на проверку корректности работы атомарных участков кода и их элементарных взаимодействий, практически не выполнимо для человека при необходимости выполнить тысячи таких проверок без ошибок.

Интеграционное тестирование фокусируется на глубокой проверке взаимодействия компонентов, особенно, когда сложно отследить процессы на уровне низкоуровневых взаимодействий глубже, чем пользовательский интерфейс. Тестирование безопасности включает необходимость проверки прав доступа, паролей по умолчанию, открытых портов, уязвимостей текущих версий ПО и т.д. Автоматизация позволяет быстро и точно выполнить множество таких проверок, что невозможно сделать вручную без риска пропустить важные ошибки.

Тестирование производительности предполагает создание нагрузки с интенсивностью и точностью, не доступной человеку, а также сбор большого набора параметров работы приложения с высокой скоростью и анализ большого объема данных из журналов системы автоматизации. Дымовое тестирование для крупных систем включает выполнение при получении каждого билда большого количества простых для автоматизации тест-кейсов, позволяющих быстро проверить основные функции системы и определить готовность к более глубокому тестированию.

Для приложений или их частей без графического интерфейса автоматизация особенно полезна. Это включает проверку консольных приложений на больших наборах значений параметров командной строки и их комбинаций, а также проверку приложе-

ний и компонентов, не предназначенных для взаимодействия с человеком, таких как веб-сервисы, серверы, библиотеки и т.д.

Длительные, рутинные и утомительные для человека операции, требующие повышенного внимания, также являются кандидатами для автоматизации. Это включает проверки, требующие сравнения больших объемов данных, высокой точности вычислений, обработки большого количества файлов, распределенных по дереву каталогов, и занимающие ощутимо большое время выполнения, особенно, когда такие проверки повторяются часто.

Проверка «внутренней функциональности» веб-приложений, таких как ссылки и доступность страниц, может быть эффективно автоматизирована. Например, проверка 30 000 ссылок на предмет их действительности становится управляемой задачей благодаря существованию многих готовых решений, обусловленных стандартностью этой задачи.

Стандартная и однотипная функциональность для многих проектов, даже при высокой сложности первичной автоматизации, окупается благодаря возможности многократного использования полученных решений в разных проектах. Технические задачи, такие как проверки корректности протоколирования, работы с базами данных, корректности поиска, файловых операций, а также форматов и содержимого генерируемых документов, также эффективно решаются с помощью автоматизации.

Преимущества и недостатки автоматизации

Автоматизация тестирования способна значительно ускорить процесс проверки программного обеспечения. Например, если вручную сравнивать несколько файлов объемом в десятки мегабайт каждый, это может занять месяцы, а то и годы. В то время как автоматизированные скрипты для дымового тестирования способны выполнить 36 проверок менее чем

за 5 секунд. Тестировщику остается только запустить скрипт, а все остальное делает автоматика.

Автоматизация тестирования исключает влияние человеческого фактора в процессе выполнения тест-кейсов. Представьте задачу сравнения двух текстовых файлов, каждый из которых по объему сопоставим с книгой в сто страниц, в целях выявления всех, даже самых незначительных, различий. Вероятность совершения ошибки при ручном выполнении такой задачи крайне высока, особенно, если учесть, что число таких файлов может достигать десяти или двадцати, а процедура сравнения должна повторяться многократно. Автоматизированные системы справляются с этой задачей без единой ошибки.

Средства автоматизации способны выполнять тест-кейсы, в принципе, непосильные для человека в силу своей сложности, скорости или иных факторов. Например, нагрузочное тестирование веб-сервиса, где необходимо одновременно эмулировать тысячи пользователей, отправляющих запросы с различными параметрами. Человек физически не сможет выполнить такое количество операций за короткий промежуток времени, тогда как автоматизированные инструменты легко справятся с этой задачей. Другим примером является тестирование мобильных приложений на множестве устройств с разными версиями операционных систем. Автоматизация позволяет быстро прогнать тесты на эмуляторах и реальных устройствах, обеспечивая покрытие всех необходимых комбинаций.

Средства автоматизации способны собирать, сохранять, анализировать, агрегировать и представлять в удобной для восприятия человеком форме колоссальные объемы данных. К примеру, в реальных проектах журналы работы систем автоматизированного тестирования могут занимать десятки гигабайт по каждой итерации. Человек не в состоянии вручную проанализировать такие объемы данных, но правильно настроенная

среда автоматизации сделает это сама, предоставив аккуратные отчеты в 2–3 страницы, удобные графики и таблицы. Если требуется более глубокий анализ, можно легко перейти от агрегированных данных к подробностям.

Средства автоматизации способны выполнять низкоуровневые действия с приложением, операционной системой, каналами передачи данных и т.д. Классический пример – тестирование реакций приложения на нехватку ресурсов. Автоматизированные средства могут искусственно создавать условия, при которых приложению не хватает оперативной памяти или места на диске, и фиксировать его поведение. Еще одним примером является эмуляция потери сетевого соединения для проверки устойчивости приложения к сбоям в сети. Даже если у тестировщика будет достаточно квалификации, чтобы самостоятельно выполнить подобные операции, ему все равно понадобится инструментальное средство.

Автоматизация тестирования – это мощный инструмент, но, как и у любой технологии, у нее есть свои недостатки.

Во-первых, необходимость наличия высококвалифицированного персонала. Автоматизация – это «проект внутри проекта» со своими требованиями, планами, кодом и т.д. Техническая квалификация сотрудников, занимающихся автоматизацией, как правило, должна быть ощутимо выше, чем у их коллег, занимающихся ручным тестированием. Например, для написания автоматизированных тестов на языках программирования, таких как Java или Python, требуется хорошее понимание этих языков и соответствующих фреймворков.

Во-вторых, разработка и сопровождение как самих автоматизированных тест-кейсов, так и всей необходимой инфраструктуры занимает много времени. Ситуация усугубляется тем, что при серьезных изменениях в проекте или в случае ошибок

в стратегии всю соответствующую работу приходится выполнять заново с нуля. Например, при переходе с одного фреймворка веб-разработки на другой все тесты, завязанные на структуру старого фреймворка, могут стать бесполезными.

Автоматизация требует более тщательного планирования и управления рисками, так как в противном случае проекту может быть нанесен серьезный ущерб. Например, если не учесть возможные изменения в интерфейсах приложения, можно потратить много ресурсов на создание тестов, которые быстро устареют. В реальном проекте это может привести к задержкам в релизах и дополнительным затратам на переработку тестов.

Коммерческие средства автоматизации стоят ощутимо дорого, а имеющиеся бесплатные аналоги не всегда позволяют эффективно решать поставленные задачи. Например, приобретение лицензий на инструменты вроде HP UFT или IBM Rational Functional Tester может оказаться непосильным для небольших компаний. При ошибочном выборе инструментов придется не только переделывать всю работу, но и нести дополнительные финансовые затраты на покупку новых средств автоматизации. Кроме того, средств автоматизации крайне много, что усложняет проблему выбора. Это затрудняет планирование и определение стратегии тестирования, может повлечь за собой дополнительные временны́е и финансовые затраты, а также необходимость обучения персонала или найма соответствующих специалистов. Например, если команда выбрала для автоматизации Selenium WebDriver, но не имеет опыта в программировании на языках, поддерживаемых этим инструментом, им придется либо обучаться, либо искать других специалистов.

Несмотря на все сложности, автоматизация тестирования при правильном подходе и грамотном планировании способна значительно повысить эффективность процесса разработки и качества программного обеспечения. Практические примеры

из реальной жизни показывают, что инвестиции в автоматизацию окупаются за счет сокращения времени на тестирование, уменьшения количества ошибок и повышения удовлетворенности пользователей.

Невозможность автоматизации

Учитывая вышесказанное, существуют ситуации, в которых автоматизация может не только не помочь, но и ухудшить положение. В основном это касается областей, где требуется человеческое мышление, а также некоторых технологических сфер.

Задачи планирования, разработки тест-кейсов, написания отчетов о дефектах, анализа результатов тестирования и составления отчетности лучше выполнять вручную. Если функциональность или тест-кейсы нужно проверить всего несколько раз и это может сделать человек, затраты на автоматизацию не оправдаются.

Когда существующие инструменты автоматизации имеют низкий уровень абстракции, возникает необходимость писать большое количество кода. Это не только сложно и занимает много времени, но и повышает риск появления ошибок в самих тест-кейсах. Если средства автоматизации обладают слабыми возможностями по протоколированию процесса тестирования и сбору технических данных о приложении и окружении, существует риск получить неинформативные данные вроде «что-то где-то сломалось», что затрудняет диагностику проблем.

При низкой стабильности требований приходится часто переделывать работу, и в случае автоматизации это обходится дороже, чем при ручном тестировании. Сложные комбинации большого количества технологий делают автоматизацию сложной, снижают надежность тест-кейсов и затрудняют оценку трудозатрат и прогнозирование рисков.

Если существуют проблемы с планированием и ручным тестированием, автоматизация хаотичного процесса лишь создаст автоматизированный хаос и потребует дополнительных ресурсов. Поэтому сначала необходимо решить существующие проблемы, а затем приступать к автоматизации. При нехватке времени и угрозе срыва сроков автоматизация не принесет мгновенных результатов и изначально будет потреблять ресурсы команды, включая время. Существует афоризм: «лучше вручную протестировать хоть что-то, чем автоматизированно не протестировать ничего».

В областях тестирования, требующих оценки ситуации человеком, таких как тестирование удобства использования или доступности, можно попытаться создать алгоритмы, которые будут оценивать ситуацию так же, как человек. Однако на практике живой человек выполнит эту работу быстрее, проще, надежнее и дешевле.

2.2. ИНСТРУМЕНТЫ ТЕСТИРОВАНИЯ PYTHON-ПРИЛОЖЕНИЙ

В главе 1.5 было представлено множество различных инструментов для проведения автоматизированного тестирования приложений. **Python**, благодаря своей простоте и читабельности, стал одним из самых популярных языков для тестирования программного обеспечения. Его богатая экосистема библиотек, среди которых особенно выделяется **Pytest**, предоставляет разработчикам мощные инструменты для написания, запуска и анализа тестов. **Pytest**, с его гибким синтаксисом, возможностями параметризации и интеграцией с другими инструментами, позволяет создавать как простые, так и сложные тестовые сценарии, эффективно проверяя отдельные функции и целые системы.

Чтобы познакомиться с данной библиотекой, важно понять базовые конструкции языка, которые лежат в основе тестов с использованием **Pytest**.

Одним из ключевых элементов в любом языке программирования, особенно в Python, являются **функции** – блоки кода, предназначенные для выполнения определенной задачи. Функции могут принимать данные для обработки (аргументы) и возвращать результат своей работы. Основные моменты, касающиеся функций:

– объявление функции начинается с ключевого слова def, за которым следует имя функции и параметры в круглых скобках;

– тело функции размещается под ее объявлением и выделяется отступом;

– функции могут возвращать значения с помощью ключевого слова return.

Пример объявления функции:

```
def add(a, b):
    return a + b
result = add(2, 3)  # result будет равен 5
```

Чтобы уменьшить количество дублирования в коде и добавить дополнительное поведение к функциям, используются **декораторы**. Декораторы в Python – это функции, которые оборачивают другие функции или методы, модифицируя или расширяя их функциональность. Они помогают повторно использовать код и следовать принципу DRY (Don't Repeat Yourself). Декораторы часто применяются для логирования, обработки ошибок или кэширования результатов.

Декоратор применяется путем добавления символа @ перед объявлением функции, которую он декорирует.

Пример декоратора:

```
def my_decorator(func):
    def wrapper():
        print("Что-то делается перед вызовом функции.")
        func()
        print("Что-то делается после вызова функции.")
    return wrapper

@my_decorator
def say_hello():
    print("Привет!")

        say_hello()
```

Обработка ошибок в Python является неотъемлемой частью написания надежного и стабильного кода. Ключевые слова try...except позволяют «перехватить» исключения, возникающие при выполнении кода, и выполнить альтернативные действия. Блок try содержит код, который может вызвать ошибку, а except – код, который выполняется в случае ошибки. Это позволяет избежать аварийного завершения программы и обеспечить более плавный пользовательский опыт.

Пример использования try...except:

```
def division(a, b):
    try:
        result = a / b
        print(f"{a} / {b} = {result}")
    except ZeroDivisionError:
        print("Деление на ноль!")

division(10, 5)  # 10 / 5 = 2.0
division(10, 0)  # Деление на ноль!
```

Иногда возникает необходимость вызвать исключение вручную. Ключевое слово raise используется для генерации собственных ошибок, что позволяет прервать выполнение кода и передать информацию об ошибочной ситуации.

Пример использования raise:

```
def check_age(age):
    if age < 18:
        raise ValueError("Недостаточный возраст")
    else:
        print("Доступ разрешен")

try:
    check_age(15)
except ValueError as e:
    print(f"Ошибка: {e}")
```

Ключевое слово yield используется для создания генераторов – специальных функций, которые возвращают значения по одному за раз, сохраняя свое состояние между вызовами. Генераторы эффективны с точки зрения памяти, поскольку они не хранят всю последовательность в памяти одновременно, а генерируют значения по мере необходимости.

Пример генератора:

```
def my_generator(n):
    for i in range(n):
        yield i * 2

for num in my_generator(5):
    print(num)
```

В этом примере my_generator – генератор, который возвращает удвоенные значения в диапазоне от 0 до n-1. Цикл for использует генератор для итерации по его значениям.

Ключевое слово assert используется в Python для проверки истинности выражения и выявления ошибок на ранней стадии разработки. Если выражение оказывается ложным, генерируется исключение AssertionError. Assert помогает разработчикам убедиться, что код работает так, как ожидается, и быстро обнаружить возможные проблемы.

Пример использования Assert:

```
def divide(x, y):
    assert y != 0, "На ноль делить нельзя"  # Проверка на ноль
    return x / y

print(divide(10, 2))  # Вывод: 5.0
print(divide(10, 0))  # Генерирует AssertionError с сообщением "На ноль делить нельзя"
```

Базовые инструменты языка Python – функции, декораторы, блоки try...except, генераторы и утверждения assert – обеспечивают прочную и эффективную основу для тестирования кода. Используя эти конструкции, можно писать чистые и понятные тесты, проверять корректность работы программы, своевременно выявлять ошибки и гарантировать соответствие кода заданным требованиям.

Знание этих основ является важным шагом на пути к эффективному использованию библиотеки Pytest для автоматизированного тестирования. Pytest позволяет расширить возможности стандартных инструментов Python, предоставляя удобный синтаксис для написания тестов, мощные механизмы параметризации и гибкие средства для организации и запуска тестовых наборов.

Понимание того, как использовать функции и декораторы, поможет создавать повторно используемые и расширяемые

тесты. Умение работать с обработкой ошибок и генераторами позволит эффективно тестировать различные сценарии и условия. А применение Assert обеспечит простую и понятную валидацию ожидаемых результатов.

Освоив эти базовые концепции, вы сможете более глубоко погрузиться в мир автоматизированного тестирования с Python и Pytest, создавая надежные и эффективные тестовые наборы для любых проектов.

Библиотека Pytest

Для более сложных и комплексных проектов может потребоваться систематизированный и автоматизированный подход к тестированию. В таких случаях стоит обратить внимание на библиотеку **Pytest**. Чтобы начать работать с ней, необходимо создать файл с расширением .py, импортировать библиотеку Pytest и определить функции, названия которых начинаются с test_.

Запуск тестов осуществляется с помощью команды Pytest в терминале. Библиотека автоматически обнаруживает файлы с тестами, если их имена начинаются с test_. Если нужно запустить тесты из файла с другим именем, достаточно указать путь к файлу после команды pytest, например:

```
pytest ./testing/file_with_test.py.
```

Для запуска конкретного теста в файле с несколькими тестами можно использовать аргумент -k, указав имя теста:

```
pytest ./testing/file_with_test.py -k test_division.
```

Дополнительные параметры позволяют получить более подробную информацию о тестах (Pytest -v), запускать только определенные тесты или группы тестов.

Рассмотрим пример файла с тестами:

```
def test_division():
    y = 0
    assert y != 0, "Делитель не должен быть равен нулю"

def test_addition():
    assert 2 + 3 == 5, "Сложение работает некорректно"
```

В этом примере мы определили два теста: test_division и test_addition. При запуске Pytest оба теста будут выполнены, и в терминале отобразится результат их выполнения. Тест test_division не пройдет, поскольку y равен нулю, что нарушает условие y != 0. Это демонстрирует, как pytest выявляет ошибки в коде.

Одной из ключевых возможностей библиотеки являются **фикстуры**. Фикстуры – это функции, которые подготавливают тестовую среду и обеспечивают повторное использование ресурсов. Например, при тестировании функций, взаимодействующих с базой данных, перед каждым тестом может потребоваться установить соединение, создать необходимые таблицы и данные. После теста важно очистить базу данных и закрыть соединение. Фикстуры автоматизируют эти повторяющиеся операции.

Пример использования фикстуры:

```
import pytest

@pytest.fixture
def db_connection():
    # Устанавливаем соединение с базой данных
    conn = establish_connection()
    yield conn  # Передаем соединение в тест
    # После выполнения теста
    conn.close()  # Закрываем соединение
```

```
def test_insert_data(db_connection):
    # Тестируем добавление данных в базу
    result = db_connection.insert("users", {"name": "Alice"})
    assert result == True

def test_query_data(db_connection):
    # Тестируем запрос данных из базы
    data = db_connection.query("SELECT * FROM users")
    assert len(data) > 0
```

В этом примере фикстура db_connection устанавливает соединение с базой данных перед каждым тестом и закрывает его после. Тестовые функции используют это соединение для проверки функциональности базы данных.

Параметризация тестов в Pytest позволяет запускать один и тот же тест с разными наборами входных данных, что делает тестирование более эффективным и уменьшает дублирование кода.

Пример параметризации:

```
import pytest

@pytest.mark.parametrize("dividend, divisor, expected", [
    (10, 2, 5),
    (9, 3, 3),
    (5, 0, None),  # Ожидаем, что деление на ноль вернет None
])
def test_division(dividend, divisor, expected):
    try:
        result = dividend / divisor
    except ZeroDivisionError:
        result = None
    assert result == expected
```

В этом тесте функция test_division проверяет операцию деления с различными значениями dividend и divisor. Параметризация позволяет легко добавить новые наборы данных для проверки дополнительных случаев.

Маркировка тестов дает возможность присваивать им теги для группировки и управления запуском. Это особенно полезно при большом количестве тестов.

Пример использования маркеров:

```
import pytest

@pytest.mark.database
def test_db_connection():
    # Тестируем соединение с базой данных
    assert db.connect() == True

@pytest.mark.api
def test_api_response():
    # Тестируем ответ API
    response = api.get("/status")
    assert response.status_code == 200
```

Чтобы запустить только тесты, помеченные маркером database, используется команда: pytest -m database.

Pytest также позволяет пропускать тесты на основании определенных условий с помощью декоратора @pytest.mark.skipif.

Пример пропуска теста:

```
import pytest
import sys

@pytest.mark.skipif(sys.platform == "darwin", reason="Тест не поддерживается на macOS")
def test_windows_feature():
    # Тест специфичен для Windows
    assert windows_specific_function() == True
```

Здесь тест test_windows_feature будет пропущен, если тестирование происходит на macOS.

Одной из самых мощных особенностей Pytest является поддержка плагинов, расширяющих функциональность библиотеки. Например:

– pytest-cov: измеряет покрытие кода тестами и генерирует отчеты;

– pytest-xdist: позволяет параллельно запускать тесты, ускоряя процесс тестирования;

– pytest-html: создает отчеты о тестах в формате HTML для удобного просмотра результатов;

– pytest-selenium: интегрируется с Selenium для автоматизированного тестирования веб-приложений.

Хотя Pytest не единственный инструмент для автоматизации тестирования в Python (существуют также unittest, nose и другие), он выделяется своей простотой, гибкостью и богатым набором функций. Это делает его отличным выбором для проектов любого масштаба. Использование Pytest позволяет разработчикам писать более качественный код, быстрее находить и исправлять ошибки, а также сокращать общее время разработки.

Представим, что мы тестируем функцию расчета налогов в финансовом приложении. Нам нужно убедиться, что функция правильно рассчитывает налог для разных уровней дохода и учитывает особые условия.

```
import pytest

@pytest.fixture
def tax_calculator():
    # Инициализируем объект калькулятора налогов
    return TaxCalculator()
```

```
@pytest.mark.parametrize("income, expected_tax", [
    (30000, 4500),    # 15% налог для дохода до 50,000
    (60000, 12000),     # 20% налог для дохода от 50,000 до 100,000
    (150000, 37500),  # 25% налог для дохода свыше 100,000
])
def test_tax_calculation(tax_calculator, income, expected_tax):
    tax = tax_calculator.calculate(income)
    assert tax == expected_tax, f"Неверный расчет налога для дохода {income}"

@pytest.mark.skipif(not is_holiday(), reason="Тест актуален только в праздничные дни")
def test_tax_holiday(tax_calculator):
    # В праздничные дни налог снижается на 5%
    income = 50000
    expected_tax = income * 0.10  # Вместо обычных 15%
    tax = tax_calculator.calculate(income)
    assert tax == expected_tax, "Неверный расчет налога в праздничный день"
```

В этом примере мы используем фикстуру tax_calculator для инициализации объекта перед тестами, параметризуем тесты для проверки разных случаев и пропускаем тесты, если не выполнены определенные условия.

Pytest предоставляет богатый набор инструментов для эффективного и гибкого тестирования. Его возможности по автоматизации, параметризации, маркировке и расширению через плагины делают его мощным инструментом в арсенале разработчика. Благодаря простоте использования и широким возможностям, Pytest способствует повышению качества программного обеспечения и ускоряет процесс разработки.

2.3. АВТОМАТИЗАЦИЯ ВЕБ-ПРИЛОЖЕНИЙ С SELENIUM

Автоматизация тестирования веб-приложений является важной частью процесса разработки, и одним из наиболее популярных инструментов для этой цели является Selenium. В сочетании с языком программирования Python, Selenium предоставляет мощные возможности для создания эффективных тестовых сценариев.

Selenium позволяет автоматизировать взаимодействие с веб-браузерами, эмулируя действия пользователя, такие как переходы по страницам, ввод текста, нажатие кнопок и другие взаимодействия с элементами веб-интерфейса. Для начала работы с Selenium необходимо установить библиотеку Selenium с помощью команды:

pip install selenium

Также необходимо установить веб-драйвер для используемого браузера, например, ChromeDriver для Google Chrome. Его можно скачать с официального сайта и добавить в переменную окружения PATH или указать путь к нему в коде. После установки необходимых компонентов можно приступить к написанию первого тестового скрипта. Например, откроем браузер, перейдем на страницу и получим заголовок страницы:

```
from selenium import webdriver
# Инициализируем веб-драйвер для Chrome
driver = webdriver.Chrome()
# Открываем страницу
driver.get("https://www.example.com")
# Получаем заголовок страницы
title = driver.title
print(f"Заголовок страницы: {title}")
# Закрываем браузер
driver.quit()
```

В этом примере мы импортируем модуль webdriver из библиотеки Selenium, создаем экземпляр драйвера для Chrome, открываем страницу по заданному URL, получаем заголовок страницы и выводим его, а затем закрываем браузер.

Для взаимодействия с элементами на странице необходимо уметь их находить. Selenium предоставляет различные методы для поиска элементов, такие как find_element_by_id, find_element_by_name, find_element_by_xpath, find_element_by_css_selector и другие. Например, чтобы найти поле ввода и ввести в него текст, можно использовать следующий код:

```
from selenium import webdriver
from selenium.webdriver.common.by import By
driver = webdriver.Chrome()
driver.get("https://www.google.com")
# Находим поле ввода по имени
search_box = driver.find_element(By.NAME, "q")
# Вводим текст в поле ввода
search_box.send_keys("Selenium Python")
# Отправляем форму
search_box.submit()
# Ждем загрузки результатов
driver.implicitly_wait(5)
print("Поиск выполнен")
driver.quit()
```

В этом примере мы переходим на страницу Google, находим поле ввода поиска по атрибуту name, вводим запрос «Selenium Python» и отправляем форму. Затем ждем несколько секунд, пока загрузятся результаты поиска.

Иногда необходимо дождаться появления элемента на странице перед взаимодействием с ним. Для этого можно использовать явные ожидания:

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
driver = webdriver.Chrome()
driver.get("https://www.example.com/login")
# Ожидаем появления поля ввода логина
username_field = WebDriverWait(driver, 10).until(
    EC.presence_of_element_located((By.ID, "username"))
)
# Вводим логин и пароль
username_field.send_keys("my_username")
password_field = driver.find_element(By.ID, "password")
password_field.send_keys("my_password")
# Нажимаем кнопку входа
login_button = driver.find_element(By.ID, "loginButton")
login_button.click()
print("Авторизация выполнена")

driver.quit()
```

Работа с выпадающими списками также часто встречается в веб-приложениях. Для взаимодействия с ними можно использовать класс Select из модуля selenium.webdriver.support.ui:

```
from selenium.webdriver.support.ui import Select
select_element = driver.find_element(By.ID, "dropdown")
select = Select(select_element)
# Выбираем опцию по видимому тексту
select.select_by_visible_text("Option 1")
# Выбираем опцию по значению
select.select_by_value("value1")
```

При автоматизации тестирования важно обрабатывать возможные исключения и ошибки. Например, если элемент не найден, Selenium выбросит исключение NoSuchElement Exception. Для обработки таких ситуаций можно использовать конструкцию try-except:

```
from selenium.common.exceptions import
NoSuchElementException
try:
    element = driver.find_element(By.ID, "nonexistent")
except NoSuchElementException:
    print("Элемент не найден")
```

Также важно корректно закрывать браузер после выполнения тестов, используя driver.quit(), чтобы освободить системные ресурсы.

Для организации тестов и повышения их повторного использования рекомендуется применять фреймворки для тестирования, такие как Pytest. С его помощью можно создавать наборы тестов, использовать фикстуры и удобно запускать тесты из командной строки.

Пример теста с использованием Pytest и Selenium:

```
import pytest
from selenium import webdriver

@pytest.fixture
def driver():
    driver = webdriver.Chrome()
    yield driver
    driver.quit()

def test_example_com_title(driver):
    driver.get("https://www.example.com")
    assert driver.title == "Example Domain", "Заголовок страницы не соответствует ожидаемому"
```

В этом примере мы определяем фикстуру driver, которая инициализирует браузер перед тестом и закрывает его после теста. Затем пишем тестовую функцию test_example_com_title, которая проверяет, что заголовок страницы соответствует ожидаемому значению.

При выполнении большого количества тестов может возникнуть необходимость в параллельном запуске тестов для ускорения процесса. Это можно реализовать с помощью плагина pytest-xdist, который позволяет запускать тесты в нескольких потоках.

Для генерации отчетов о тестировании можно использовать плагин pytest-html, который создает HTML-отчеты с подробной информацией о результатах выполнения тестов.

Автоматизация тестирования с помощью Selenium и Python предоставляет мощные инструменты для обеспечения качества веб-приложений. Используя практические примеры кода и применяя лучшие практики, можно создавать надежные и эффективные тестовые сценарии, которые помогут своевременно обнаруживать и устранять ошибки в приложениях.

2.4. НАГРУЗОЧНОЕ ТЕСТИРОВАНИЕ И ЕГО ИНСТРУМЕНТЫ

Нагрузочное тестирование играет ключевую роль в обеспечении качества программного обеспечения, особенно для систем, которым необходимо обрабатывать большой объем запросов или обслуживать множество пользователей одновременно. Цель этого вида тестирования состоит в том, чтобы определить, как приложение ведет себя под различными уровнями нагрузки, выявить узкие места и убедиться, что оно соответствует требованиям производительности.

Одним из наиболее популярных инструментов для проведения нагрузочного тестирования является Apache JMeter. Это открытое программное обеспечение, написанное на Java, которое позволяет моделировать разнообразные сценарии нагрузки и анализировать поведение приложения в условиях интенсивного использования. JMeter поддерживает тестирование как статических, так и динамических ресурсов, включая веб-сервисы, базы данных и другие компоненты.

Для начала работы с JMeter необходимо установить его, скачав с официального сайта и распаковав архив. После установки можно приступить к созданию простого тестового плана. Например, если требуется протестировать веб-приложение, можно настроить HTTP-запросы к серверу и определить количество виртуальных пользователей, которые будут выполнять эти запросы. В JMeter открывается новый тестовый план, в который добавляется группа потоков (Thread Group). Эта группа определяет количество виртуальных пользователей, параметры нагрузки и поведение сценария.

Настройка параметров группы потоков включает указание числа потоков (users), например, 50 виртуальных пользователей. Период разгона (Ramp-Up Period) может быть установлен в 20 секунд, что означает, что все пользователи начнут выполнение в течение этого времени. Также можно задать количество повторений (Loop Count), например, 100, чтобы каждый пользователь выполнял действия несколько раз.

Внутри группы потоков добавляется HTTP-запрос, где указываются параметры сервера, такие как имя хоста или IP-адрес (например, www.example.com), путь к ресурсу (например, /api/test) и метод запроса (GET или POST). Это позволяет имитировать реальные запросы к веб-серверу.

Для сбора и анализа результатов тестирования используются слушатели, такие как «View Results Tree» или «Summary

Report». Они отображают подробную информацию о выполнении тестов, время отклика, процент ошибок и другие важные метрики, что позволяет выявить узкие места в производительности приложения.

JMeter также позволяет запускать тестирование из командной строки, что удобно для автоматизации и интеграции с системами непрерывной интеграции. Например, команда *jmeter -n -t test_plan.jmx -l test_results.jtl* запускает тестовый план в нелокальном режиме (без графического интерфейса), используя указанный файл тестового плана и записывая результаты в файл.

Другим мощным инструментом для нагрузочного тестирования является LoadRunner от компании Micro Focus. Это коммерческое решение предоставляет широкий спектр возможностей для моделирования различных типов нагрузки и анализа производительности систем. В LoadRunner сценарии тестирования создаются с помощью Virtual User Generator (VuGen), где записываются действия виртуальных пользователей. Записанные сценарии могут быть отредактированы и параметризованы для более реалистичного моделирования.

Например, сценарий на языке C в LoadRunner может выглядеть следующим образом:

```
Action()
{
    lr_start_transaction("HomePage");

    web_url("OpenHomePage",
        "URL=http://www.example.com/",
        "TargetFrame=",
        "Resource=0",
        "RecContentType=text/html",
        "Referer=",
```

```
        "Snapshot=t1.inf",
        "Mode=HTML",
        LAST);

    lr_end_transaction("HomePage", LR_AUTO);

    return 0;
}
```

В этом коде функции lr_start_transaction и lr_end_transaction используются для измерения времени выполнения определенных действий, а функция web_url отправляет HTTP-запрос на указанный URL. Такой подход позволяет детально анализировать производительность отдельных частей приложения.

Кроме JMeter и LoadRunner, существует инструмент Locust, написанный на Python. Он позволяет описывать сценарии нагрузки программно, что дает гибкость и простоту в настройке сложных сценариев. В Locust сценарии описываются с использованием классов и методов, что делает их легкочитаемыми и поддерживаемыми.

Например, сценарий в Locust может быть следующим:

```
from locust import HttpUser, task, between

class WebsiteUser(HttpUser):
    wait_time = between(1, 5)

    @task(3)
    def view_items(self):
        self.client.get("/items")
        self.client.get("/items/1")
        self.client.get("/items/2")

    @task(1)
    def add_to_cart(self):
        self.client.post("/cart", {"item_id": 1, "quantity": 1})
```

В этом примере класс WebsiteUser представляет пользователя, выполняющего определенные задачи. Метод view_items имитирует просмотр товаров на сайте, выполняя GET-запросы к различным ресурсам. Метод add_to_cart имитирует добавление товара в корзину посредством POST-запроса. Параметр wait_time задает случайное время ожидания между задачами, что помогает смоделировать поведение реальных пользователей.

Для запуска тестирования с помощью Locust код сохраняется в файл locustfile.py, после чего команда *locust -f locustfile.py --host=http://www.example.com* позволяет начать тестирование. Затем, открыв веб-интерфейс Locust по адресу http://localhost:8089, можно задать количество пользователей и скорость их появления, а также наблюдать за результатами тестирования в режиме реального времени.

Нагрузочное тестирование позволяет выявить проблемы, связанные с производительностью, масштабируемостью и устойчивостью системы. Оно помогает определить, где система может не справляться с нагрузкой, как она ведет себя при увеличении числа пользователей и насколько быстро восстанавливается после перегрузок.

При проведении нагрузочного тестирования важно создавать реалистичные сценарии, моделирующие поведение реальных пользователей. Начинать следует с небольшой нагрузки, постепенно увеличивая ее, чтобы наблюдать за реакцией системы. Мониторинг ресурсов, таких как использование CPU, памяти и дисковой системы на серверах, является неотъемлемой частью процесса. Внимательный анализ результатов, отчетов и лог-файлов позволяет сделать выводы о производительности и стабильности приложения.

Например, после проведения тестирования в JMeter можно использовать такие инструменты, как «Aggregate Report» или

«Response Time Graph», чтобы визуализировать результаты и определить среднее время отклика, процент ошибок и другие ключевые показатели. Это помогает разработчикам и тестировщикам принять обоснованные решения о необходимости оптимизации или изменения архитектуры приложения.

Использование инструментов для нагрузочного тестирования, таких как Apache JMeter, LoadRunner и Locust, способствует обеспечению высокой производительности и надежности приложений. Они позволяют выявить и устранить потенциальные проблемы до того, как приложение будет запущено в эксплуатацию, что повышает удовлетворенность пользователей и снижает риски сбоев в работе системы. Грамотное применение этих инструментов в сочетании с тщательным анализом результатов является важным шагом на пути к созданию качественного программного обеспечения.

2.5. ПРАКТИЧЕСКИЙ ПРИМЕР СОСТАВЛЕНИЯ ТЕСТОВОЙ ДОКУМЕНТАЦИИ

Практика составления тестовой документации является неотъемлемой частью процесса автоматизированного тестирования. Она позволяет обеспечить прозрачность тестового покрытия, облегчить поддержку тестов и гарантировать, что все функциональные требования проверены должным образом. Рассмотрим, как создавать четкие и понятные тест-кейсы, документировать результаты тестирования и организовывать тестовую документацию на протяжении всего жизненного цикла проекта, используя Python и библиотеку Pytest на простом примере.

Представим простое приложение – функцию is_prime(n), которая определяет, является ли заданное число n простым. Нам необходимо написать автоматизированные тесты для этой функции и соответствующую тестовую документацию.

Исходный код функции:

```
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True
```

Для обеспечения полного покрытия функциональности необходимо определить тестовые сценарии, которые проверяют различные случаи использования функции. Тест-кейсы должны быть четко описаны, включая входные данные, ожидаемый результат и описание.

1. Тест-кейсы

№	Тестовый сценарий	Входные данные	Ожидаемый результат	Описание
1	Тестирование отрицательных чисел и нуля	$n = -5$	False	Проверка, что функция возвращает False для отрицательных чисел
		$n = 0$	False	Проверка, что ноль не является простым числом
2	Тестирование числа 1	$n = 1$	False	Проверка, что число 1 не является простым
3	Тестирование простых чисел	$n = 7$	True	Проверка, что число 7 определяется как простое
		$n = 13$	True	Проверка, что число 13 определяется как простое

Продолжение табл. 1

№	Тестовый сценарий	Входные данные	Ожидаемый результат	Описание
4	Тестирование составных чисел	$n = 10$	False	Проверка, что число 10 определяется как составное
		$n = 15$	False	Проверка, что число 15 определяется как составное

На основе составленных тест-кейсов реализуем автоматизированные тесты.

Файл тестов test_is_prime.py:

```
import pytest
from prime import is_prime
def test_negative_numbers():
    assert is_prime(-5) == False, "Отрицательные числа не являются простыми"
    assert is_prime(0) == False, "Ноль не является простым числом"
def test_one():
    assert is_prime(1) == False, "Число 1 не является простым"
def test_prime_numbers():
    assert is_prime(7) == True, "7 является простым числом"
    assert is_prime(13) == True, "13 является простым числом"
def test_composite_numbers():
    assert is_prime(10) == False, "10 не является простым числом"
    assert is_prime(15) == False, "15 не является простым числом"
```

Для выполнения тестов используем команду в терминале:

Pytest test_is_prime.py-v

Результаты выполнения тестов:

test_is_prime.py::test_negative_numbers PASSED
test_is_prime.py::test_one PASSED
test_is_prime.py::test_prime_numbers PASSED
test_is_prime.py::test_composite_numbers PASSED

Все тесты успешно пройдены, что подтверждает корректность работы функции для заданных случаев.

После выполнения тестов необходимо задокументировать результаты для последующего анализа и хранения в истории проекта. Документация должна содержать информацию о выполненных тестах, их статусе, дате и времени выполнения, а также об окружении, в котором проводилось тестирование.

2. Отчет о тестировании функции is_prime

Параметр	Значение
Дата	25 ноября 2024 года
Ответственный	Иван Иванов
Версия приложения	1.0
Окружение	Python 3.9, pytest 6.2.5, Windows 10

3. Результаты тестов

Тестовый сценарий	Статус
test_negative_numbers	Пройден
test_one	Пройден
test_prime_numbers	Пройден
test_composite_numbers	Пройден

Все тесты успешно пройдены. Функция is_prime работает корректно для протестированных случаев.

Для эффективного управления тестовой документацией важно придерживаться определенных рекомендаций. Во-первых, следует поддерживать четкую структуру проекта, храня тесты в отдельной директории, например, tests/. Это облегчает навигацию по проекту и упрощает процесс поддержки тестов в будущем.

Во-вторых, используйте понятные и единообразные наименования файлов и функций, которые отражают суть проводимых тестов. Такой подход улучшает читаемость кода и позволяет быстрее ориентироваться в наборе тестов, что особенно важно при работе в команде.

Третьим аспектом является документирование тестов. Добавляйте комментарии и описания к каждому тесту, поясняя его цель и особенности. Это помогает другим разработчикам и тестировщикам понимать назначение тестов и упрощает процесс их обновления или исправления.

Также рекомендуется хранить тестовую документацию и сами тесты в системе контроля версий, такой как Git, вместе с исходным кодом приложения. Это обеспечивает отслеживание изменений, совместную работу над проектом и возможность отката к предыдущим версиям при необходимости.

Автоматизация играет важную роль в поддержке качества кода. Настройте автоматический запуск тестов при изменениях в коде с помощью инструментов непрерывной интеграции и доставки (CI/CD). Это позволяет своевременно обнаруживать и исправлять ошибки, а также поддерживать стабильность приложения на всех этапах разработки.

Наконец, при изменении функциональности приложения необходимо обновлять соответствующие тесты и документацию. Это гарантирует, что тестовый набор всегда актуален и полностью соответствует текущему состоянию приложения, что

в свою очередь повышает надежность и качество программного обеспечения.

Для автоматизации процесса документирования результатов тестирования можно использовать плагины к Pytest, например, pytest-html.

Установка плагина:

pip install pytest-html

Генерация HTML-отчета:

pytest test_is_prime.py --html=report.html

Полученный файл report.html содержит подробный отчет о выполненных тестах и может быть открыт в браузере.

Если в ходе развития проекта меняются требования, необходимо своевременно обновлять тесты и документацию. Пример изменения требования: Число 1 теперь считается простым.

Обновление теста test_one:

```
def test_one():
    assert is_prime(1) == True, "Число 1 теперь считается простым"
```

Обновление документации:

– измените ожидаемый результат в тест-кейсе для числа 1;
– обновите отчет о тестировании, отразив изменения;
– зафиксируйте причину изменения в системе контроля версий с соответствующим сообщением коммита.

Систематический подход к составлению тестовой документации и организации автоматизированных тестов с использованием Python и Pytest способствует повышению качества программного обеспечения. Четко описанные тест-кейсы, регулярное документирование результатов и грамотная организация тестовой документации облегчают поддержку проекта и позволяют оперативно реагировать на изменения требований.

ЗАКЛЮЧЕНИЕ

Тестирование программного обеспечения играет ключевую роль в обеспечении его качества, надежности и соответствия требованиям пользователей. В данном учебном пособии рассмотрены как теоретические основы, так и практические аспекты тестирования, включая создание тестовой документации и автоматизацию. Особое внимание уделено современным инструментам автоматизации, таким как Pytest, Selenium и JMeter, которые позволяют эффективно организовать процесс проверки функциональности, производительности и безопасности программного обеспечения.

Изучение материалов пособия дает студентам возможность освоить полный цикл тестирования: от анализа требований и разработки тест-кейсов до проведения тестов и документирования их результатов. Полученные знания помогут лучше понять значение тестирования в разработке и внедрении программных продуктов, а также освоить навыки, необходимые для автоматизации рутинных задач и повышения эффективности работы.

Тестирование – это не только поиск ошибок, но и важный процесс обеспечения качества, влияющий на успешность продукта на рынке. Коллектив авторов надеется, что материалы пособия помогут студентам стать востребованными специалистами, способными применять современные подходы и инструменты в реальных проектах.

СПИСОК ЛИТЕРАТУРЫ

1. **Логачева, Н. В.** Важность тестирования программного обеспечения в процессе разработки программного обеспечения / Н. В. Логачева, М. Л. Ладонычева, К. С. Пузырева // Инновационная наука. – 2022. – № 2-2. – С. 23 – 26.

2. **Косов, Н. А.** Применение нейронных сетей для автоматизации тестирования программного обеспечения / Н. А. Косов, П. С. Мазепин, Н. А. Гришин // Наукосфера. – 2020. – № 6. – С. 152 – 156.

3. **Бубненков, Н. А.** Анализ существующих средств управления конфигурациями и методов тестирования программного обеспечения / Н. А. Бубненков // Наука молодых – наука будущего. – 2023. – С. 275 – 280.

4. **Валиуллина, Д. И.** Применение фреймворка PyTest для тестирования программного кода на языке Python / Д. И. Валиуллина, И. А. Зиганшин // Ученый XXI века. – 2022. – С. 35 – 37.

5. **Шумилин, С. С.** Мировые тенденции развития тестирования на Python / С. С. Шумилин // Научный аспект. – 2021. – Т. 1, № 3. – С. 88.

6. **Пакшин, А. В.** Организация процесса автоматизированного тестирования web-приложений на базе фреймворка SELENIUM / А. В. Пакшин, С. А. Фирсова // Огарев-Online. – 2021. – № 12(165). – С. 9.

7. **Колот, А. В.** Тестирование сайта с помощью онлайн-инструмента нагрузочного тестирования / А. В. Колот, М. Е. Щелкунова // Наука, инновации и технологии: от идей к внедрению. – 2022. – С. 26 – 28.

ОГЛАВЛЕНИЕ

Учебное электронное издание

НИКОЛЮКИН Максим Сергеевич
КОНКИНА Виктория Викторовна
ПАТУТИН Кирилл Игоревич

ТЕСТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Учебное пособие

Редактирование И. В. К а л и с т р а т о в о й
Графический и мультимедийный дизайнер Т. Ю. З о т о в а
Обложка, упаковка, тиражирование И. В. К а л и с т р а т о в о й

ISBN 978-5-8265-2883-9

Подписано к использованию 04.03.2025.
Тираж 50 шт. Заказ № 41

Издательский центр ФГБОУ ВО «ТГТУ»
392000, г. Тамбов, ул. Советская, д. 106, к. 14
Тел./факс (4752) 63-81-08.
E-mail: izdatelstvo@tstu.ru