|  |  |
| --- | --- |
|  | Министерство науки и высшего образования Российской Федерации  Федеральное государственное бюджетное образовательное учреждение высшего образования «Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»  (МГТУ им. Н.Э. Баумана) |
| ФАКУЛЬТЕТ «ИНЖЕНЕРНЫЙ БИЗНЕС И МЕНЕДЖМЕНТ»  КАФЕДРА «ПРОМЫШЛЕННАЯ ЛОГИСТИКА» (ИБМ-3)  Домашнее задание  На тему: «Тестирование программного обеспечения в Python**»**  По дисциплине: «Парадигмы и конструкции языков программирования»  38.03.05. Бизнес-информатика, Маркетинг цифровых технологий (уровень бакалавриата)  Студент ИБМ3- 34Б  \_\_\_\_\_\_\_\_\_\_ Д.А.Артёмова  (Подпись, дата)  Руководитель А.Н.Нардид  (Подпись, дата)  2024 г.  **Введение**  Качество программного обеспечения (ПО) является критическим фактором в современном мире. Независимо от размера и сложности проекта, тестирование играет ключевую роль в обеспечении надежности, корректности и безопасности разрабатываемого продукта. Язык программирования Python, благодаря своей простоте, читаемости и богатому набору библиотек, широко используется для разработки тестовых сценариев и автоматизации процессов тестирования. Данный реферат посвящен обзору различных подходов и инструментов для тестирования ПО, доступных в экосистеме Python.  В рамках данного реферата будут рассмотрены следующие задачи:  1. Определение ключевых понятий и методологий тестирования ПО: Будут определены основные термины и классификации методов тестирования (модульное, интеграционное, системное, приемочное тестирование), а также рассмотрены различные стратегии интеграционного тестирования (top-down, bottom-up, big bang).  2. Обзор инструментов и фреймворков для тестирования в Python: Будет проведен сравнительный анализ популярных фреймворков для тестирования в Python, таких как unittest и pytest, с акцентом на их сильных и слабых сторонах, а также рассмотрены дополнительные инструменты, такие как mock и nose2.  3. Демонстрация практического применения инструментов: В реферате будут представлены примеры кода, иллюстрирующие использование unittest и pytest для написания различных типов тестов, включая unit-тесты и использование фикстур.  4. Анализ стратегий проектирования тестов: Будут описаны принципы написания эффективных тестов (FIRST), а также кратко рассмотрены методологии TDD (Test-Driven Development) и BDD (Behavior-Driven Development).  5. Обсуждение лучших практик тестирования: Реферат затронет важные аспекты эффективного тестирования, такие как покрытие кода тестами, управление тестовой средой и анализ результатов тестирования.  **Основные подходы и стратегии тестирования**  Тестирование ПО — это комплексный процесс, включающий в себя различные стратегии и техники, направленные на выявление дефектов и обеспечение соответствия программного продукта заданным требованиям. Ключевые методологии включают:   * Модульное тестирование (Unit Testing): Этот уровень тестирования фокусируется на изолированных компонентах кода, таких как отдельные функции, классы или методы. Цель — проверить корректность работы каждого компонента независимо от остальных частей системы. В Python unittest и pytest предлагают мощные средства для организации и проведения unit-тестов, включая утверждения (assertions) для проверки ожидаемых результатов и фикстуры (fixtures) для управления тестовой средой. Ключевым аспектом unit-тестирования является принцип "быстрого обратной связи": тесты должны выполняться быстро, чтобы разработчики могли оперативно получать информацию о найденных ошибках. * Интеграционное тестирование (Integration Testing): После того, как модульные тесты успешно пройдены, переходят к интеграционному тестированию. На этом этапе проверяется взаимодействие между различными модулями, чтобы убедиться в их корректном взаимодействии и обмене данными. Существуют различные стратегии интеграционного тестирования:   \* Top-down: Тестирование начинается с верхних уровней системы, постепенно переходя к нижним.  \* Bottom-up: Тестирование начинается с нижних уровней, постепенно интегрируя компоненты и проверяя взаимодействие.  \* Big Bang: Все компоненты интегрируются одновременно, и затем проводится комплексное тестирование.   * Системное тестирование (System Testing): На этом этапе тестируется вся система как единое целое. Цель — проверить соответствие системы заданным требованиям, включая функциональность, производительность, безопасность и надежность. Системное тестирование часто включает в себя тестирование на различных платформах и конфигурациях. * Приемочное тестирование (Acceptance Testing): Это заключительный этап тестирования, на котором система проверяется конечными пользователями или представителями заказчика. Цель — убедиться, что система соответствует их ожиданиям и бизнес-требованиям. Существуют различные виды приемочного тестирования, такие как альфа-тестирование (тестирование внутри компании) и бета-тестирование (тестирование конечными пользователями).   **Инструменты и фреймворки для тестирования в Python**  Python предлагает огромный выбор инструментов для автоматизации тестирования:   * unittest: Стандартный модуль Python, предоставляющий объектно-ориентированный фреймворк для написания unit-тестов. Он прост в освоении, но может быть менее удобным для больших проектов, чем pytest. * pytest: Один из самых популярных фреймворк для тестирования в Python. Он характеризуется гибкостью, минимализмом и расширяемостью за счет множества плагинов. pytest поддерживает различные виды тестирования, включая параметризованное тестирование, фикстуры и мощные средства отладки. * nose2: Расширение для unittest, предоставляющее дополнительные возможности, такие как автоматическое обнаружение тестов и более удобный вывод результатов. * mock: Библиотека для создания mock-объектов, позволяющих имитировать поведение зависимостей и изолировать тестируемый код. Это особенно полезно при интеграционном тестировании, когда сложно или невозможно использовать реальные зависимости (например, базы данных, внешние API). Mock-объекты позволяют контролировать поведение зависимостей, устанавливать ожидаемые вызовы методов и возвращать заданные значения. Это значительно упрощает тестирование и повышает скорость выполнения тестов, особенно в сложных системах. * Selenium: Фреймворк для автоматизированного тестирования веб-приложений. Selenium позволяет управлять веб-браузером и взаимодействовать с веб-элементами, что делает его незаменимым инструментом для тестирования веб-интерфейсов. * Requests: Библиотека для отправки HTTP-запросов, используется для тестирования веб-сервисов (API).   Рассмотрим примеры тестирования класса Rectangle, Circle и Square (предполагается, что они определены в модуле geometric\_figures и имеют метод area() для вычисления площади), используя два популярных фреймворка: pytest и unittest.  *Пример 1:* pytest  pytest известен своим лаконичным синтаксисом и удобством использования. Вот пример нескольких тестов:  import pytest import math from geometric\_figures import Rectangle, Circle, Square  @pytest.fixture def rectangle\_fixture():  return Rectangle(5, 10, "blue")  @pytest.fixture def circle\_fixture():  return Circle(5, "green")  @pytest.fixture def square\_fixture():  return Square(5, "red")  def test\_rectangle\_area\_bdd(rectangle\_fixture):  assert rectangle\_fixture.area() == 50  def test\_circle\_area\_bdd(circle\_fixture):  assert math.isclose(circle\_fixture.area(), 78.5398, abs\_tol=0.001)  def test\_square\_area\_bdd(square\_fixture):  assert square\_fixture.area() == 25  def test\_rectangle\_error\_bdd():  with pytest.raises(ValueError):  Rectangle(-5, 10, "blue")  def test\_circle\_error\_bdd():  with pytest.raises(ValueError):  Circle(-5, "green")  *Пример 2:* unittest  unittest — стандартный модуль Python, предоставляющий более традиционный, объектно-ориентированный подход к тестированию:  import unittest from geometric\_figures import Rectangle, Circle, Square  class TestFigures(unittest.TestCase):  def test\_rectangle\_area(self):  rect = Rectangle(5, 10, "blue")  self.assertEqual(rect.area(), 50)   def test\_circle\_area(self):  circ = Circle(5, "green")  self.assertAlmostEqual(circ.area(), 78.5398, places=4)   def test\_square\_area(self):  square = Square(5, "red")  self.assertEqual(square.area(), 25)   def test\_rectangle\_error(self):  with self.assertRaises(ValueError):  Rectangle(-5, 10, "blue")   def test\_circle\_error(self):  with self.assertRaises(ValueError):  Circle(-5, "green")  if \_\_name\_\_ == '\_\_main\_\_':  unittest.main()  Как видно из примеров, pytest обычно считается более кратким и удобным (без включения фиктур), в то время как unittest предоставляет более структурированный подход, особенно полезный для больших проектов.  **Организация тестового кода**  Эффективное тестирование требует не только правильного выбора инструментов, но и грамотной организации тестового кода. Хорошо структурированный тестовый код легко поддерживать, расширять и понимать. В Python часто используются следующие подходы:   * Разделение тестов по модулям: Тесты для каждого модуля помещаются в отдельные файлы или директории, что улучшает читаемость и упрощает поиск нужных тестов. * Использование именных соглашений: Следование соглашениям об именовании файлов и тестов (например, test\_\*.py для файлов с тестами) помогает автоматическим инструментам правильно обнаруживать и запускать тесты. * Фикстуры (fixtures): pytest позволяет создавать фикстуры – функции, которые выполняют подготовку тестовой среды (например, создание тестовых данных, подключение к базе данных). Фикстуры повышают читаемость и переиспользуемость кода тестов. * Параметризация: Позволяет запускать один и тот же тест с различными наборами входных данных.   **Стратегии проектирования тестов**  Эффективное тестирование требует не только выбора правильных инструментов, но и продуманной стратегии проектирования тестов:   * FIRST: Принципы написания хороших тестов (Fast, Independent, Repeatable, Self-Validating, Thorough). * Test-Driven Development (TDD): Методология разработки, где тесты пишутся перед реализацией кода. * Behavior-Driven Development (BDD): Методология, которая фокусируется на поведении системы с точки зрения пользователя или заказчика. Инструменты, такие как Behave, позволяют писать тесты на более понятном, близком к естественному языку.   **Отчетность о результатах тестирования**  После запуска тестов важно проанализировать результаты. Инструменты тестирования предоставляют различные отчеты, которые помогают оценить качество тестирования:   * Процент покрытия кода: Показывает, какая часть кода покрыта тестами. * Количество пройденных и не пройденных тестов: Дает общее представление об успешности тестирования. * Детальные отчеты об ошибках: Предоставляют информацию об ошибках, включая стек вызовов, что помогает в отладке. * Генерация отчетов в различных форматах: Многие инструменты позволяют генерировать отчеты в формате HTML, XML и других, что упрощает их анализ и хранение.   **Заключение**  В заключение, тестирование ПО является неотъемлемой частью успешной разработки качественного и надежного программного продукта. Python предоставляет мощный набор инструментов и фреймворков, которые позволяют автоматизировать процесс тестирования на всех этапах, от модульного тестирования до сложных системных проверок. Использование различных стратегий и методов, а также правильный выбор инструментов, позволяют значительно повысить эффективность процесса тестирования и гарантировать высокое качество конечного продукта.  **Литература**   1. Unit testing frameworke. Python 3.13.0 documentaion. Документация по Python, модуль unittest. URL: <https://docs.python.org/3/library/unittest.html> (дата обращения 24.11.2024). 2. Pytest documantaion. Документация модуля pytest. URL: <https://docs.python.org/3/library/unittest.html> (дата обращения 24.11.2024). 3. Selenium. Описание фреймворка. URL: <https://www.selenium.dev/about/> (дата обращения 24.11.2024). 4. Requests: HTTP for Humans. Документация. URL: <https://requests.readthedocs.io/en/latest/> (дата обращения 24.11.2024). 5. Эффективная работа с легаси – кодом. Майкл Физерс. Статья. URL: <https://bespoyasov.ru/blog/working-effectively-with-legacy-code/> (дата обращения 24.11.2024). | |