2. ПРАКТИЧЕСКАЯ РАБОТА. МОДУЛЬНОЕ И МУТАЦИОННОЕ ТЕСТИРОВАНИЕ ПРОГРАММНОГО ПРОДУКТА

2.1. Цель и задачи практической работы

Цель работы: познакомить студентов с процессом модульного и мутационного тестирования, включая разработку, проведение тестов, исправление ошибок, анализ тестового покрытия, а также оценку эффективности тестов путём применения методов мутационного тестирования.

Для достижения поставленной цели работы студентам необходимо выполнить ряд **задач**:

– изучить основы модульного тестирования и его основные принципы;

– освоить использование инструментов для модульного тестирования (pytest для Python, JUnit для Java и др.);

– разработать модульные тесты для программного продукта и проанализировать их покрытие кода;

– изучить основы мутационного тестирования и освоить инструменты для его выполнения (MutPy, PIT, Stryker);

– применить мутационное тестирование к программному продукту, оценить эффективность тестов;

– улучшить существующий набор тестов, ориентируясь на результаты мутационного тестирования;

– оформить итоговый отчёт с результатами проделанной работы.

2.2. Теоретический раздел

2.2.1. Модульное тестирование

Модульное тестирование (Unit Testing) — это метод тестирования программного обеспечения, при котором проверяются отдельные компоненты программы, такие как функции, методы, классы. Основной целью является изолированная проверка корректности работы каждого компонента.

Основные принципы модульного тестирования:

– каждый компонент тестируется независимо от других;

– тесты должны быть легко воспроизводимыми;

– для каждого теста задаётся ожидаемый результат.

Также для модульного тестирования можно выделить ряд особенностей:

1. Тесты должны быть детерминированными, то есть при повторном запуске с теми же входными данными всегда давать одинаковый результат. Это исключает вероятность ложных срабатываний.

2. Тесты должны выполняться в контролируемом окружении, без зависимости от внешних систем (например, баз данных или сетевых подключений). Для этого часто применяются mock-объекты или заглушки (stubs).

3. Каждый тест должен проверять только одну функцию или метод. Это упрощает поиск причин ошибки, если тест провалится.

4. Тесты должны быть легко читаемыми и понятными для любого члена команды, независимо от его опыта. Это повышает качество кода и упрощает сопровождение.

5. Хорошо написанный тест выполняет не только проверку, но и служит документацией для тестируемого кода, показывая, как он должен работать.

6. Выполнение одного теста не должно влиять на результат другого. Это гарантирует, что тесты можно запускать в любом порядке.

7. Тесты должны быть оптимизированы, чтобы выполняться быстро, особенно если они запускаются в CI/CD-пайплайне.

8. Цель — максимально покрыть функциональность тестами, включая граничные случаи и исключительные ситуации.

9. Особое внимание уделяется тестированию критичных участков системы, таких как алгоритмы обработки данных, сложная бизнес-логика или интерфейсы взаимодействия с внешними системами. И т.д.

Рассмотрим ряд инструментов для проведения модульного тестирования для популярных языков программирования.

Для Python

PyTest является популярным инструментом с гибкими возможностями для написания и запуска тестов. Он поддерживает параметризацию, обработку исключений и удобный вывод результатов.

Для Java

JUnit широко используется в разработке на Java благодаря своей простоте и поддержке аннотаций для настройки тестов. TestNG, вдохновлённый JUnit, предоставляет расширенные возможности, такие как параметризация тестов и управление зависимостями между ними.

Для C#/.NET

NUnit — это надёжный инструмент для тестирования в среде .NET, который поддерживает асинхронное тестирование и предоставляет мощный функционал для работы с атрибутами. xUnit.net является современным аналогом NUnit, разработанным с акцентом на производительность и гибкость.

Для C++

Google Test является мощным фреймворком для тестирования приложений на C++, предоставляющим поддержку модульных и интеграционных тестов, а также расширенные функции для обработки исключений. Catch2 — это лёгкий и удобный инструмент для написания тестов с простым синтаксисом.

Для JavaScript/TypeScript

Mocha — это гибкий фреймворк для тестирования JavaScript, который поддерживает асинхронные тесты и интеграцию с другими библиотеками. Jest предоставляет возможности для тестирования React-приложений, включая снапшот-тесты, и отличается высокой производительностью. Vitest — это современный инструмент для тестирования, идеально подходящий для проектов с использованием TypeScript.

Для Kotlin

KotlinTest (или Kotest) предоставляет мощные функции для тестирования на Kotlin, включая поддержку спецификаций (например, BDD, Gherkin), аннотаций и интеграцию с JUnit. Он идеально подходит для создания читаемых и лаконичных тестов.

Для Ruby

RSpec — это мощный инструмент для тестирования на Ruby, поддерживающий поведенческое тестирование (BDD) и позволяющий писать тесты в человекочитаемом формате.

2.2.2. Мутационное тестирование

Мутационное тестирование (Mutation Testing) — это метод тестирования, при котором создаются изменённые версии программы (мутанты) с внесением небольших ошибок. Цель — проверить, способны ли существующие тесты обнаружить эти ошибки. Впервые оно было предложено в 1970-х годах Ричардом Демилло, Ричардом Липтоном и Фредом Сэйфором.

В данном виде тестирования выделяют следующую терминологию:

1) ***мутант*** — это модифицированная версия кода;
2) ***убийство мутанта*** — процесс нахождения ошибки в коде с помощью разработанного теста, вызванную мутантом;
3) ***выживший мутант*** — мутант, не обнаруженный тестами.

Преимущества мутационного тестирования заключаются в том, что данный метод помогает определить, насколько тесты способны обнаруживать ошибки, при этом выжившие мутанты указывают на участки кода, не охваченные тестами. Большинство процессов, включая создание мутантов, можно автоматизировать.

История и развитие мутационного тестирования проходила в несколько основных этапов:

– **1970-1980-е годы.** Разработка теоретической основы. Метод активно изучался в академической среде, но его практическое применение было ограничено из-за низкой вычислительной мощности.

– **1990-е годы.** Разработка первых инструментов, таких как Mothra. Мутационное тестирование использовалось в исследовательских и критически важных проектах.

– **2000-е годы.** Улучшение автоматизации и рост вычислительных мощностей сделали метод более доступным. Появились инструменты, такие как Proteum и MuJava.

– **2010-е годы.** Интеграция в CI/CD-пайплайны и популяризация Agile и DevOps ускорили внедрение мутационного тестирования в разработку ПО.

– **2020-е годы.** Современные инструменты, такие как PIT (для Java), Stryker (для JavaScript, TypeScript, C#), сделали метод стандартной практикой для многих компаний.

В настоящее время мутационное тестирование применяется в различных областях:

1. Критически важные системы, например, такие как:

– ***Авионика***. Проверка алгоритмов управления и систем безопасности. Например, мутационное тестирование используется для проверки реакции систем управления полётом на некорректные входные данные.

– ***Медицинские системы***. Тестирование программного обеспечения для медицинских приборов, таких как устройства мониторинга состояния пациентов.

– ***Финансовые системы***. Проверка алгоритмов обработки транзакций и шифрования. Реальный пример — тестирование работы систем автоматического расчёта кредитных ставок.

2. API и библиотеки.

Мутационное тестирование помогает убедиться, что API и библиотеки устойчивы к ошибкам и некорректным данным. Например, библиотека для работы с JSON может тестироваться на обработку повреждённых или некорректных данных.

3. Интеграция в CI/CD.

Мутационное тестирование становится частью автоматизированных пайплайнов. Например, при каждом коммите запускается тестирование на мутантах, чтобы убедиться, что изменения в коде не снижают качество тестового покрытия.

4. Искусственный интеллект и машинное обучение.

Используется для тестирования алгоритмов ИИ, где требуется высокая степень надёжности. Например, мутационное тестирование может применяться для проверки устойчивости моделей машинного обучения к некорректным данным.

2.2.3. Сочетание с модульным тестированием

Мутационное тестирование и модульное тестирование идеально дополняют друг друга. Существуют следующие варианты совместного использования:

1. Оценка модульных тестов. Благодаря использования мутационного тестирования в процессе проверки программного продукта возможно понять, насколько эффективно модульные тесты выявляют ошибки.

2. Создание мутантов на уровне модулей упрощает анализ результатов. Например, в модуле расчёта налогов можно проверить, что тесты выявляют ошибки в изменённых формулах расчёта.

3. Выжившие мутанты указывают на пробелы в модульных тестах, которые можно устранить.

4. Автоматизированные инструменты запускают мутационное тестирование в рамках пайплайнов, что обеспечивает регулярный контроль качества.

Для того, чтобы мутационного тестирования было максимально эффективным необходимо придерживаться ряда рекомендаций. При проверке программного продукта стоит фокусироваться на критически важных модулях, т.е. начинать с компонентов, где ошибки наиболее критичны. Например, в системах обработки платежей это могут быть модули расчёта комиссии. Также особое внимание стоит уделить анализу выживших мутантов. Данный процесс помогает понять, какие аспекты требуют доработки. Например, если мутант выживает из-за отсутствия теста на граничное значение, необходимо добавить соответствующий тест. Но стоит обратить особое внимание на то, что количество мутантов должно быть ограниченным, чтобы процесс оставался управляемым. Также хорошим тоном является интеграция мутационного тестирования в CI/CD. Регулярный запуск тестов позволяет своевременно выявлять ошибки. Например, использование мутационного тестирования на этапе проверки pull request помогает гарантировать качество новых изменений.

Для успешного применения мутационного тестирования для популярных языков программирования разработан ряд инструментов. Такие, например, как PIT (Pitest), который является популярным инструментом в экосистеме Java, MutPy для Python или набирающий популярность в последнее время Stryker для JavaScript, TypeScript и C#. Но также нужно учесть и тот факт, что данный вид тестирования начал набирать свою популярность относительно недавно, в связи с чем не для всех языков программирования существуют дополнительные инструменты, помогающие в проведении процесса мутаций кода.

2.2.4. Пример, использования мутационного тестирования.

Исходный код программы представлен функцией для проверки чётности числа в листинге 1:

Листинг 1. Функция для проверки чётности числа

```
def is_even(n):
 return n % 2 == 0
```

Для данной функции был разработан модульный тест (листинг 2):

Листинг 2. Модульный тест для функции проверки чётности числа

```
def test_is_even():
  assert is_even(2) == True # проверка, что чётное число 2 правильно определяется как True
  assert is_even(3) == False # проверка, что нечётное число 3 правильно определяется как False
  assert is_even(0) == True # проверка, что 0 правильно определяется как чётное число (True), так как 0 % 2 == 0
```

Для более полного процесса проверки функции и теста в код были добавлены «мутанты»:

1. Замена оператора проверки остатка на деление:

```
def is_even(n):
  return n / 2 == 0
```

Результат — Мутант «убит», так как тест *assert is_even(3) == False* не прошёл.

2. Изменение условия на противоположное:

```
def is_even(n):
```

return n % 2 != 0

Результат — Мутант «убит», так как тест *assert is_even(2) == True* провалился.

3. Возврат константного значения:

def is_even(n):

return True

Результат — Мутант «выжил», так как тесты не проверяют случаи, где результат должен быть False, кроме одного.

Вывод по проведению дополнительно мутационного тестирования для базовой функции, проверенной изначально модульным тестом, заключается в том, что данная проверка выявила, что начальный тест недостаточно охватывает случаи с отрицательными числами. Например, нужно добавить проверку assert is_even(-3) == False, чтобы улучшить покрытие и убить мутанта.

Мутационное тестирование — это эффективный и мощный инструмент, который помогает улучшить качество тестов и повышает надёжность программного обеспечения. История его развития показывает, как метод эволюционировал от теоретической концепции до широко используемого подхода в индустрии. Современные инструменты и автоматизация сделали метод доступным и практичным для компаний, стремящихся минимизировать ошибки и создать устойчивые системы.

Реальные примеры и лучшие практики подтверждают, что мутационное тестирование незаменимо для обеспечения качества в критически важных системах, API, библиотеках и даже в проектах с использованием искусственного интеллекта. Интеграция этого подхода в процессы разработки делает его важной частью современных стандартов тестирования и разработки ПО.

2.3. Описание работы

Практическая работа состоит из двух частей. Работа проходит внутри ранее сформированной команды. Каждый из участников выступает в роли «Разработчика» и «Тестировщика».

2.3.1. Часть 1. Модульное тестирование

Внутри команды каждый участник разрабатывает свой программный модуль, содержащий минимум 5 функций. Одна из функций должна содержать преднамеренную ошибку. Далее участникам необходимо произвести обмен про-

граммным продуктом и документацией. Для полученного модуля разрабатываются тесты для проверки функциональности. После нахождения ошибки член команды, выступающий в роли тестировщика, возвращает приложение с описанием найденного дефекта на доработку разработчику. Баг исправляется и программный продукт возвращается на повторное тестирование. На этапе разработки документации к ПП между разработчиком и тестровщиком должен быть согласован язык разработки, чтобы избежать дальнейших конфликтных ситуаций.

Ниже приведены примеры модулей, которые могут быть разработаны в рамках практической работы.

– Конвертер единиц измерения (перевод величин между различными системами, например, температуры, длины, массы).

– Генератор паролей (создание случайных паролей с заданными параметрами, проверка их сложности).

– Анализатор текста (подсчёт слов, символов, определение частоты использования слов).

– Программа для работы с матрицами (сложение, умножение, транспонирование матриц).

– Фильтр чисел (поиск простых чисел, чисел Фибоначчи, деление чисел на чётные и нечётные).

– Программа для работы с датами (вычисление разницы между датами, определение дня недели по дате).

– Калькулятор BMI (Body Mass Index) (расчёт индекса массы тела и рекомендации по здоровому весу).

– Симулятор банковского счёта (начисление процентов, снятие и пополнение средств, ведение истории операций).

– Шифратор и дешифратор текста (реализация простых алгоритмов шифрования, например, шифра Цезаря или XOR).

– Программа для работы с файлами (чтение, запись, копирование, поиск определённых данных в файле).

– Калькулятор (с функциями сложения, вычитания, умножения и деления).

– Программа для работы с массивами (поиск максимального значения, сортировка массива).

– Программа для работы со строками (проверка палиндромов, подсчет символов) и т.д.

Этапы работы:

1) разработка программного продукта (модуля);
2) написание модульных тестов;
3) исправление ошибки, выявленных тестами;
4) формирование отчёта об ошибке;
5) повторное тестирование.

Пример описания ошибки для её передачи разработчику.

Краткое описание ошибки: «Неверное преобразование заглавных букв в строчные».

Статус ошибки: открыта («Open»).

Категория ошибки: серьезная («Major»).

Тестовый случай: «Проверка алгоритма функционирования программы».

Описание ошибки:

1. Загрузить программу.

2. В поле ввода ввести строку «ABCD».

3. Нажать кнопку «Пуск».

4. Полученный результат: «ABCD».

Ожидаемый результат: «abcd».

Примечание: Категория ошибки может принимать следующие значения: блокирующие («Blocker»), критические («Critical»), серьезные («Major»), незначительные («Minor»), тривиальные («Trivial»).

2.3.2. Часть 2. Мутационное тестирование

На втором этапе практической работы участники команды применяют мутационное тестирование для оценки качества тестов, разработанных на предыдущем этапе. Данный этап помогает проверить, насколько эффективно тесты обнаруживают ошибки в программном модуле.

Порядок выполнения работы:

1. Создание мутантов.

Участники применяют инструменты мутационного тестирования для автоматического внесения небольших изменений (мутаций) в код модуля.

Мутации могут включать:

– Изменение операторов (+ → -, * → /).
– Изменение значений (return 0 → return 1).
– Замена логических условий (> → <, == → !=).

2. Запуск тестов на мутантах.

а) Тестовый набор, разработанный в Части 1, применяется к мутантам.

б) Если тест обнаруживает изменение и не проходит — мутант «убит» (тест эффективен).

в) Если тест проходит успешно, несмотря на внесённую мутацию — мутант «выжил», что говорит о недостаточном покрытии тестами.

3. Анализ выживших мутантов.

а) Разбираются причины выживания мутантов.

б) Выявляются пробелы в тестировании, добавляются новые тесты для их устранения.

4. Доработка тестов.

а) Дополняются и уточняются модульные тесты.

б) Повторно выполняется мутационное тестирование для проверки улучшенного тестового покрытия.

2.4. Итоговый отчёт

По результатам командной работы отчёт должен содержать:

1. Титульный лист, включающий в себя наименование работы, название команды, состав команды, дату выполнения, название учебной дисциплины.

2. Цель и задачи работы, где указываются основная цель практической работы и перечень выполненных задач.

3. Практическую часть, включающую:

а) разработку модуля: описание функциональности, исходный код;

б) модульное тестирование: описание тестов, методология, анализ покрытия кода, исправление ошибок;

в) мутационное тестирование: создание мутантов, анализ их выживания, корректировка тестов.

4. Анализ и выводы, включающие:

а) оценку качества тестирования;

б) анализ недостатков и их устранение;

в) итоговые выводы по результатам работы.

5. Приложения (при необходимости): дополнительные материалы (коды, логи тестирования, скриншоты и т. д.).

Отчёт должен быть оформлен в соответствии с ГОСТ 19.401-78 и ГОСТ 34.602-2020. Требования включают стандарты на титульный лист, использование единообразного шрифта, оформление таблиц и диаграмм, наличие нумерации страниц.

Итоговая оценка работы будет зависеть от полноты отчёта, качества выполнения задания, соответствия оформления стандартам и презентации результатов на защите.