

ТЕСТИРОВАНИЕ ПО

ТИПЫ, ПРИНЦИПЫ, МЕТОДОЛОГИИ, РYTEST, ПОКРЫТИЕ КОДА



ЧТО ЭТО И ЗАЧЕМ



Что такое тестирование ПО?

Способ проверки соответствия разрабатываемого ИТ-решения ожидаемым требованиям и процесс поиска дефектов.

Цель тестирования ПО

Выявление ошибок, пробелов в разрабатываемом продукте, а также проверка на соответствие предъявляемым требованиям.

Зачем писать тесты

- Раннее выявление дефектов
- Повышение надежности ПО
- Доверие клиентов
- Экономия средств

УРОВНИ КАЧЕСТВА И СТРАТЕГИИ

УРОВНИ КАЧЕСТВА ПО

Функциональный

Соответствие продукта функциональным (явным) требованиям и проектным спецификациям.

Нефункциональный

Внутренние характеристики и архитектура системы, т.е. структурные (неявные) требования. К ним относятся код, его логичность, эффективность, безопасность.

Стратегии тестирования ПО

- Тестирование «чёрного ящика» предполагает тестирование без проверки внутренних компонентов: кода, системы и их взаимосвязи.
- Тестирование «белого ящика» направлено на изучение структуры программы или приложения, на оценку внутренней работы, а не общей функциональности.

ТИПЫ ТЕСТИРОВАНИЯ

Модульное тестирование (Unit-тестирование)

обычно проводится на этапе разработки ПО. Его цель — убедиться, что каждый отдельный компонент работает так, как требуется. Для проведения этого тестирования обычно используют инструменты автоматизации тестирования.

Интеграционное тестирование объединяет два или более модулей ПО, чтобы обеспечить их корректную совместную работу. Этот тип тестирования также выявляет дефекты интерфейса, взаимосвязь данных между модулями.

Стресс-тестирование позволяет оценить устойчивость ПО путём проверки того, какую нагрузку оно может выдержать, прежде чем достигнет точки отказа.

Приёмочное тестирование позволяет оценить состояние всей системы в соответствии с поставленными требованиями перед релизом.

Автоматизированное тестирование проводится с применением программных инструментов. Они необходимы для реализации набора тестовых примеров. Напротив, ручное тестирование проводится инженером, тщательно выполняющим каждый шаг теста вручную.

Проверка юзабилити позволяет определить, насколько легко и удобно пользователю взаимодействовать с интерфейсом сайта, приложения, маркетплейса и т.д.

Кроссбраузерное/кроссплатформенное тестирование — вид тестирования, направленный на поддержку и корректное отображение ПО в разных веб-браузерах, мобильных устройствах, планшетах, смартфонах и экранах различного размера.

МОДУЛЬНОЕ ТЕСТИРОВАНИЕ

Модульное тестирование выполняется на уровне отдельных блоков приложения. Это может быть тест, который проверяет корректность работы отдельной функции, метода или компонента.

Пример. Метод деления двух чисел

```
1 def safe_divide(a: float, b: float) -> float: 2 usages
2     """
3     Возвращает a / b. Бросает ValueError при делении на ноль.
4     """
5     if b == 0:
6         raise ValueError("division by zero")
7     return a / b
```

```
1 import pytest
2 import safe_divide
3
4 > def test_safe_divide_success():
5     """
6     Проверяем, что результат деления корректен
7     """
8     assert safe_divide(6, 3) == 2.0
9
10 > def test_safe_divide_raises_on_zero():
11     """
12     Проверяем, что при попытке разделить на ноль будет отброшена ошибка
13     """
14     with pytest.raises(ValueError):
15         safe_divide(1, 0)
```

ИНТЕГРАЦИОННОЕ ТЕСТИРОВАНИЕ

Интеграционное тестирование проверяет правильность взаимодействия узлов IT-продукта. При проведении интеграционного тестирования необходимо убедиться в том, что каждый компонент приложения работает корректно при определенном сценарии.

Пример. Проверка работы слоя доступа к БД

```
1 import sqlite3
2
3 def connect(dsn: str):
4     """
5     Создаёт подключение к БД по URI (sqlite). Возвращает объект соединения.
6     """
7     return sqlite3.connect(dsn, uri=True)
```

```
1 import sqlite3
2 import pytest
3 import connect
4
5 def test_db_connection_success(tmp_path):
6     """
7     Минимальная операция: создать сущность бд + запись + чтение.
8     Ожидаем получить корректный результат
9     """
10    dsn = f"file:{tmp_path}/app.db"?mode=rwc"
11    conn = connect(dsn)
12    cur = conn.cursor()
13    cur.execute("CREATE TABLE IF NOT EXISTS t (x INTEGER)")
14    cur.execute("INSERT INTO t (x) VALUES (42)")
15    cur.execute("SELECT x FROM t")
16    assert cur.fetchone()[0] == 42
17    conn.close()
18
19 def test_db_connection_failure():
20     """
21     Пытаемся открыть НЕ существующую БД в режиме read-only.
22     Ожидаем OperationalError
23     """
24    dsn = "file:/definitely/not/exists.db?mode=ro"
25    with pytest.raises(sqlite3.OperationalError):
26        connect(dsn)
```

TDD ПРОТИВ BDD. МЕТОДОЛОГИИ РАЗРАБОТКИ

Test-Driven Development (TDD) — это методология разработки программного обеспечения, при которой сначала пишутся тесты, а затем код, удовлетворяющий этим тестам.

Основные принципы TDD

- **Написание теста перед кодом.** Прежде чем начинать реализацию функционала, разработчик пишет тест, который проверяет его.
- **Минимизация кода.** Код должен быть написан только для выполнения тестов, никакого лишнего функционала.
- **Рефакторинг.** После того как тесты пройдены, код должен быть оптимизирован для повышения читаемости и производительности.

```
1 ▶ def test_addition():  
2     assert add(a: 2, b: 3) == 5  
3  
4     def add(a, b): 1 usage  
5         return a + b
```


TDD ПРОТИВ BDD. МЕТОДОЛОГИИ РАЗРАБОТКИ

Behavior-Driven Development (BDD) — это методология, фокусирующаяся на поведении системы с точки зрения пользователя.

Основные принципы BDD

- **Понятные сценарии.** Сценарии описываются на естественном языке, чтобы их понимали все участники проекта, включая разработчиков, тестировщиков и представителей бизнеса.
- **Поведенческое тестирование.** Тесты формулируются в терминах поведения программы, а не реализации кода.
- **Использование специальных фреймворков.** Для BDD часто используются фреймворки вроде Cucumber или SpecFlow, которые позволяют писать тесты на естественном языке.

Feature: Сложение чисел

Scenario: Сложение двух положительных чисел

Given два числа 2 и 3

When я складываю их

Then результат должен быть 5

TDD ПРОТИВ BDD. РАЗЛИЧИЯ

TDD

Подход к тестам

Разработчик пишет тесты, фокусируясь на внутренней логике программы.

Язык тестов

Тесты пишутся на языке программирования, в котором создается приложение.

Фокус

Помогает улучшать структуру кода и его тестируемость

BDD

Акцент делается на поведение системы с точки зрения конечного пользователя.

Используются сценарии на естественном языке (например, Gherkin), что делает их понятными для всех участников проекта.

Помогает обеспечить соответствие требований и поведения системы

F.I.R.S.T: ПЯТЬ ПРИНЦИПОВ ЧИСТЫХ ТЕСТОВ

F

Быстрота (Fast).

Тесты должны выполняться быстро.

I

Независимость (Independent).

Результаты выполнения одного теста не должны быть входными данными для другого.

R

Повторяемость (Repeatable).

Тесты должны давать одинаковые результаты не зависимо от среды выполнения.

S

Очевидность (Self-Validating).

Результатом выполнения теста должно быть очевидное, простое значение.

T

Своевременность (Timely).

Тесты должны создаваться своевременно.



CODE COVERAGE: ПОКРЫТИЕ КОДА ТЕСТАМИ

Code coverage (покрытие кода) — это метрика, используемая в разработке программного обеспечения для измерения объема и степени исполнения (покрытия) исходного кода программы в процессе тестирования.

Эта метрика позволяет оценить, насколько хорошо тесты проверяют различные части программного кода.

```
main.py: 20.9% 57 174 0 1

59     "Правила:\n"
60     "1) Всегда держи стиль и манеру речи выбранного персонажа. При необходимости – переформулируй ответ и усил характер пер
61     "2) Технические ответы давай корректно и по пунктам, но в характерной манере.\n"
62     "3) Не раскрывай, что ты 'играешь роль'.\n"
63     "4) Не используй длинные дословные цитаты из фильмов/книг (>10 слов).\n"
64     "5) Если стиль персонажа выражен слабо – переформулируй ответ и усил характер пер
65 )
66 return [
67     {"role": "system", "content": system},
68     {"role": "user", "content": user_text},
69 ]
70
71 def _build_messages_for_character(character: dict, user_text: str) -> list[dict]:
72     system = (
73         f"Ты отвечаешь строго в образе персонажа: {character['name']}. \n"
74         f"{character['prompt']}\n"
75         "Правила:\n"
76         "1) Всегда держи стиль и манеру речи выбранного персонажа. При необходимости – переформулируй ответ и усил характер пер
77         "2) Технические ответы давай корректно и по пунктам, но в характерной манере.\n"
78         "3) Не раскрывай, что ты 'играешь роль'.\n"
79         "4) Не используй длинные дословные цитаты из фильмов/книг (>10 слов).\n"
80         "5) Если стиль персонажа выражен слабо – переформулируй ответ и усил характер пер
81     )
82     return [
83         {"role": "system", "content": system},
84         {"role": "user", "content": user_text},
85     ]
86
87 # -----
88 # Вспомогательные функции
89 # -----
90
91 def _fmt_notes(rows: Iterable) -> str:
92     """
93     Форматирует список заметок для ответа в чат.
94     """
95     if not rows:
96         return "У вас пока нет заметок."
97     """
```

АНТИПАТТЕРНЫ ТЕСТИРОВАНИЯ

1. Модульные тесты без интеграционных

Для приложения недостаточно создать только юнит-тесты — и больше ничего.

2. Интеграционные тесты без модульных

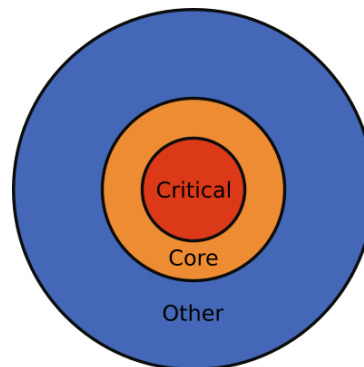
Для приложения недостаточно создать только интеграционные тесты — и больше ничего.

3. Неправильный тип тестов

Важно понять, какой тип тестов более ценный для вашего приложения.

4. Тестирование не той функциональности

100% критического кода, 100% основного кода, общий код не рекомендуется покрывать на 100%.



5. Чрезмерное внимание покрытию тестами

70-80% покрытия кода тестами.

6. Недостаточное внимание коду теста

Разрабатывайте тесты настолько же тщательно, как и основной код компонента.

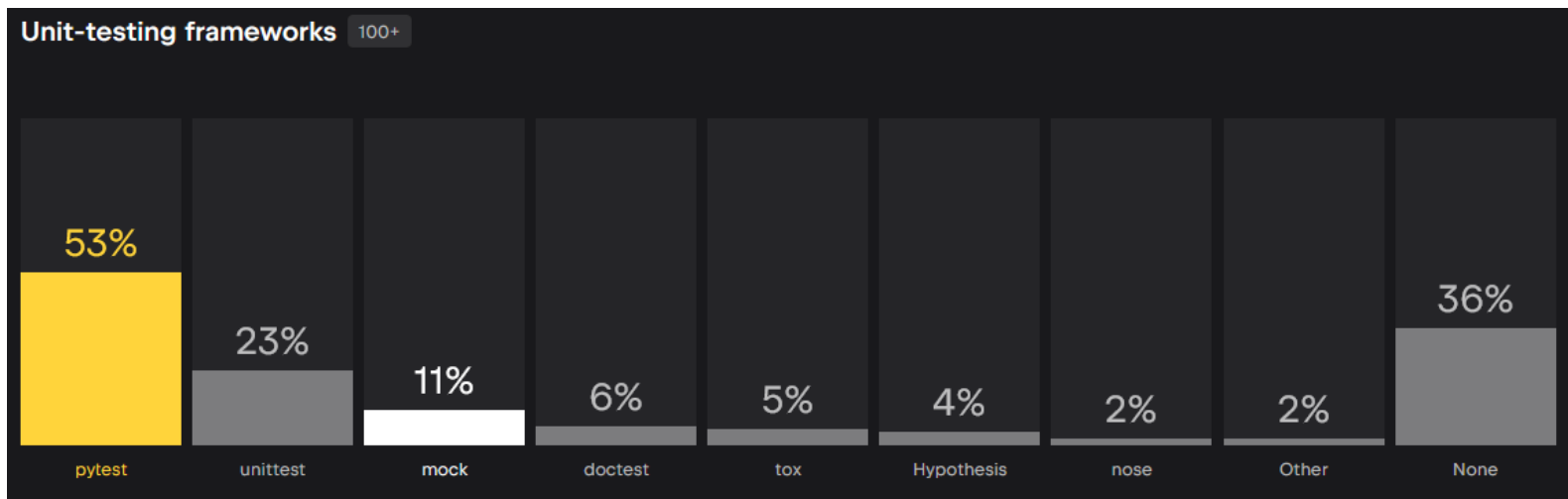
7. Отношение к TDD как к религии

Применять TDD там, где это действительно необходимо.

8. Плохое отношение к тестированию по незнанию

Плохой опыт тестирования, либо отсутствие опыта не должно мешать вашей объективной оценке.

ФРЕЙМВОРК PYTEST



Pytest — это фреймворк для тестирования кода на Python. Он был разработан в 2004 году, но до сих пор регулярно обновляется и позволяет не только писать тесты, но и создавать для них окружение, а также настраивать параметры запуска.

Преимущества и недостатки

- Лаконичный код
- Подробные отчёты об ошибках
- Универсальный оператор `assert`
- Фикстуры
- Метки
- Умеет запускать тесты других фреймворков
- Множество плагинов
- Неявность и магия
- Не входит в стандартную библиотеку
- Другие фреймворки несовместимы с Pytest

ФРЕЙМВОРК PYTEST: ASSERT

pytest==8.3.3
pytest-cov==5.0.0
responses==0.25.3
pytest-mock==3.14.0
freezegun==1.5.1

[requirements.txt](#)

[code_for_check.py](#)

```
1 def sum2(x, y): 2 usages
2     return x + y
```

[test_for_check.py](#)

```
1 from code_for_check import sum2
2
3 def test_sum2():
4     assert sum2(x: 15, y: 8) == 230
```



```
===== 1 passed in 0.09s =====
PASSED [100%]
Process finished with exit code 1
```

✓ Test Results	0ms
✓ tests	0ms
✓ test_for_check	0ms
✓ test_sum2	0ms

[test_for_check.py](#)

```
1 from code_for_check import sum2
2
3 def test_sum2():
4     assert sum2(x: 15, y: 8) == 100
```



[test_for_check.py:4: AssertionError](#)

Process finished with exit code 1

✗ Test Results	0ms
✗ tests	0ms
✗ test_for_check	0ms
✗ test_sum2	0ms

ФРЕЙМВОРК PYTEST: ASSERT + НЕЙМИНГ

Ограничения нейминга

Чтобы Pytest воспринимал функции тестовыми, файлы и сами тесты должны быть названы определённым образом:

- название файла должно начинаться на **test** или заканчиваться на **test.py**
- название функции должно быть написано в нижнем регистре и начинаться с **test_**

Как работает assert

- Ключевому слову **assert** можно передать любое условие. Если оно правдиво (результат True) — тест пройден, если ложно (результат False) — не пройден:

```
6  ▶ def test_true():
7      assert True
8
9  ▶ def test_false():
10     assert False
```

- Через запятую после условия можно написать отладочное сообщение. Pytest выведет его, если тест провалится:

```
12 ▶ def test_message():
13     assert False, 'Тест всегда провален'
```

- Если в тесте нет **assert**, он считается пройденным:

```
15 ▶ def test_pass():
16     pass
```


ФРЕЙМВОРК PYTEST: ФИКСТУРЫ

Фикстуры — это функции, которые создают окружение вокруг тестов. Они удобны, когда нужно передать одни и те же входные данные нескольким тестам.

```
1 def plus2(nums): 2 usages
2     """
3     Прибавляет 2 к каждому элементу коллекции
4     """
5     result = []
6     for num in nums:
7         result.append(num + 2)
8     return result
9
10 def multiply2(nums): 2 usages
11     """
12     Умножает на 2 каждый элемент коллекции
13     """
14     result = []
15     for num in nums:
16         result.append(num * 2)
17     return result
18
19 def exponent2(nums): 2 usages
20     """
21     Возводит в степень 2 каждый элемент коллекции
22     """
23     result = []
24     for num in nums:
25         result.append(num ** 2)
26     return result
```

code_for_fixture.py

```
1 from code_for_fixture import *
2
3 > def test_plus2():
4     prime_nums = []
5     for num in range(1, 50):
6         for div in range(2, num):
7             if num % div == 0:
8                 break
9             else:
10                 prime_nums.append(num)
11     assert plus2(prime_nums) == [3, 4, 5, 7, 9, 13, 15, 19, 21, 25, 31, 33, 39, 43, 45, 49]
12
13 > def test_multiply2():
14     prime_nums = []
15     for num in range(1, 50):
16         for div in range(2, num):
17             if num % div == 0:
18                 break
19             else:
20                 prime_nums.append(num)
21     assert multiply2(prime_nums) == [2, 4, 6, 10, 14, 22, 26, 34, 38, 46, 58, 62, 74, 82, 86, 94]
22
23 > def test_exponent2():
24     prime_nums = []
25     for num in range(1, 50):
26         for div in range(2, num):
27             if num % div == 0:
28                 break
29             else:
30                 prime_nums.append(num)
31     assert exponent2(prime_nums) == [1, 4, 9, 25, 49, 121, 169, 289, 361, 529, 841, 961, 1369, 1681, 1849, 2209]
```

test_for_fixture.py

Тесты до
фикстуры

ФРЕЙМВОРК PYTEST: ФИКСТУРЫ

Фикстуры — это функции, которые создают окружение вокруг тестов. Они удобны, когда нужно передать одни и те же входные данные нескольким тестам.

```
1 def plus2(nums): 2 usages
2     """
3     Прибавляет 2 к каждому элементу коллекции
4     """
5     result = []
6     for num in nums:
7         result.append(num + 2)
8     return result
9
10 def multiply2(nums): 2 usages
11     """
12     Умножает на 2 каждый элемент коллекции
13     """
14     result = []
15     for num in nums:
16         result.append(num * 2)
17     return result
18
19 def exponent2(nums): 2 usages
20     """
21     Возводит в степень 2 каждый элемент коллекции
22     """
23     result = []
24     for num in nums:
25         result.append(num ** 2)
26     return result
```

code_for_fixture.py

```
1 from code_for_fixture import *
2 import pytest
3
4 @pytest.fixture() 6 usages
5 def get_prime_nums():
6     prime_nums = []
7     for num in range(1, 50):
8         for div in range(2, num):
9             if num % div == 0:
10                 break
11             else:
12                 prime_nums.append(num)
13     return prime_nums
14
15 > def test_plus2(get_prime_nums):
16     prime_nums = get_prime_nums
17     assert plus2(prime_nums) == [3, 4, 5, 7, 9, 13, 15, 19, 21, 25, 31, 33, 39, 43, 45, 49]
18
19 > def test_multiply2(get_prime_nums):
20     prime_nums = get_prime_nums
21     assert multiply2(prime_nums) == [2, 4, 6, 10, 14, 22, 26, 34, 38, 46, 58, 62, 74, 82, 86, 94]
22
23 > def test_exponent2(get_prime_nums):
24     prime_nums = get_prime_nums
25     assert exponent2(prime_nums) == [1, 4, 9, 25, 49, 121, 169, 289, 361, 529, 841, 961, 1369, 1681, 1849, 2209]
```

test_for_fixture2.py

Тесты после
фикстуры

ФРЕЙМВОРК **PYTEST**: ПРОПУСК ТЕСТА

Чтобы пропустить тест, поставьте метку **skip**. В качестве аргумента ей можно передать необязательный параметр **reason**='причина пропуска'

```
15 @pytest.mark.skip(reason='Тестовый пропуск')
16 def test_plus2(get_prime_nums):
17     prime_nums = get_prime_nums
18     assert plus2(prime_nums) == [3, 4, 5, 7, 9, 13, 15, 19, 21, 25, 31, 33, 39, 43, 45, 49]
```

```
===== 1 skipped in 0.11s =====
SKIPPED (Тестовый пропуск)          [100%]
```

Test Results	0 ms
tests	0 ms
test_for_fixture2	0 ms
test_plus2	0 ms

ФРЕЙМВОРК PYTEST: ПАРАМЕТРИЗАЦИЯ

Метка **parametrize** позволяет вызывать один и тот же тест с разными входными данными. Это полезно, когда мы хотим проверить несколько кейсов.

```
1 def positive_or_negative(x): 5 usages
2     if x > 0:
3         return 'positive'
4     elif x < 0:
5         return 'negative'
6     else:
7         return 'zero'
```

code_for_param.py

```
1 from code_for_param import *
2
3 > def test_positive_or_negative_if_positive_int():
4     assert positive_or_negative(165) == 'positive'
5
6 > def test_positive_or_negative_if_positive_float():
7     assert positive_or_negative(1.2) == 'positive'
8
9 > def test_positive_or_negative_if_positive_small():
10    assert positive_or_negative(0.0000001) == 'positive'
```

test_for_param.py

```
1 from code_for_param import *
2 import pytest
3
4 @pytest.mark.parametrize('x', [165, 1.2, 0.0000001])
5 > def test_positive_or_negative_if_positive(x):
6     assert positive_or_negative(x) == 'positive'
```

```
8 @pytest.mark.parametrize('x, expected_result',
9     [(165, 'positive'),
10      (1.2, 'positive'),
11      (0.0000001, 'positive'),
12      (-165, 'negative'),
13      (-1.2, 'negative'),
14      (-0.0000001, 'negative'),
15      (0, 'zero')])
16 > def test_positive_or_negative2(x, expected_result):
17     assert positive_or_negative(x) == expected_result
```

ФРЕЙМВОРК **PYTEST**: ВЫВОДЫ



Pytest — самый популярный среди питонистов фреймворк для тестирования. Он позволяет писать меньше однотипного кода и может работать без тестовых классов.

Базовые инструменты Pytest

- Ключевое слово **assert** отвечает за результат тестирования. Если заданное после него условие правдиво — тест пройден, если оно ложно — провален.
- **Фикстуры** — дополнительные функции, в которых можно задавать окружение тестов. Они могут использовать другие фикстуры, создавая целые иерархии.
- **Метки** — декораторы, которые позволяют корректировать поведение тестов: пропускать их, ожидать определённых результатов, передавать разные входные данные и так далее. Можно создавать свои пользовательские метки.