

Программирование: как писать тесты, которые действительно проверяют код

01.11.2026 · PDF-версия статьи с сайта cms.dolinais.ru

В 2026 году автоматические проверки стали неотъемлемой частью разработки. Однако даже большое количество тестов не гарантирует, что они действительно что-то проверяют. История из проекта kb-engine, где 21 автоматическая проверка давала зелёный результат, но некоторые из них были бесполезны, — не единичный случай. В этой статье разберём, как писать тесты, которые ловят ошибки, а не создают иллюзию безопасности.

Почему тесты могут «врать»

Тест — это код, который проверяет другой код. Как и любой код, он может содержать ошибки. Рассмотрим типичные причины ложноположительных результатов:

- **Отсутствие утверждений (assertions).** Тест выполняется, но не проверяет результат. Например, функция вызывается, но её возвращаемое значение не сравнивается с ожидаемым.
- **Слишком общие утверждения.** Проверка, что результат не равен null или не пуст, не гарантирует корректность.
- **Игнорирование исключений.** Если тест ожидает исключение, но оно не выбрасывается, а тест всё равно проходит — это ошибка.
- **Зависимость от внешнего состояния.** Тесты, которые полагаются на порядок выполнения или данные из базы, могут проходить случайно.
- **Неправильная настройка моков.** Заглушки могут возвращать захардкоженные значения, которые не соответствуют реальному поведению.

Все эти проблемы приводят к тому, что тесты становятся «зелёными», но не обнаруживают дефекты.

Как проектировать эффективные тесты

Чтобы тесты приносили пользу, следуйте этим принципам.

1. Один тест — одна проверка

Каждый тест должен проверять одну конкретную вещь. Это упрощает диагностику: если тест упал, сразу понятно, что именно сломалось. Например, вместо теста «функция возвращает правильный результат и не выбрасывает исключение» сделайте два отдельных теста.

2. Используйте чёткие утверждения

Сравнивайте фактические значения с ожидаемыми. Для сложных структур данных применяйте глубокое сравнение (deep equal). Например, в JavaScript можно использовать `expect(result).toEqual(expected)`, а в Python — `assert result == expected`.

3. Проверяйте граничные случаи

Тесты должны покрывать не только типичные сценарии, но и краевые условия: пустые входные данные, максимальные значения, отрицательные числа, некорректные типы. Это помогает выявить ошибки на ранних этапах.

4. Изолируйте тесты

Каждый тест должен быть независим от других. Используйте фикстуры (fixtures) для подготовки данных и очищайте состояние после выполнения. Например, в pytest фикстуры с областью видимости function гарантируют свежее окружение для каждого теста.

5. Применяйте мутационное тестирование

Мутационное тестирование — это метод, при котором в код вносятся небольшие изменения (мутации), и тесты должны их обнаружить. Если тесты проходят на мутировавшем коде, значит, они неэффективны. Инструменты вроде Stryker (JavaScript), PIT (Java), mutmut (Python) автоматизируют этот процесс. В 2026 году мутационное тестирование стало доступнее и чаще включается в CI-пайплайны.

6. Регулярно пересматривайте тесты

Тесты — это тоже код, и они требуют поддержки. При изменении требований обновляйте тесты. Удаляйте устаревшие проверки, которые больше не актуальны. Проводите ревью тестов так же тщательно, как и основного кода.

Пример: как улучшить плохой тест

Рассмотрим функцию, которая возвращает сумму двух чисел. Плохой тест может выглядеть так:

```
def test_add():
    result = add(2, 3)
    assert result is not None
```

Этот тест пройдёт, даже если функция вернёт 0 или строку. Хороший тест проверяет конкретное значение:

```
def test_add():
    assert add(2, 3) == 5
```

Ещё лучше — добавить проверку граничных случаев:

```
def test_add_negative():
    assert add(-2, -3) == -5

def test_add_zero():
    assert add(0, 0) == 0
```

Инструменты и практики в 2026 году

Современные фреймворки для тестирования предоставляют удобные средства для написания надёжных проверок. Например, в Jest (JavaScript) появились улучшенные матчеры для асинхронного кода, в pytest (Python) — более гибкие параметризованные тесты. Встроенные в IDE инструменты анализа покрытия помогают выявить непокрытые участки, но помните: 100% покрытие не гарантирует отсутствие ошибок.

Для CI/CD используйте параллельный запуск тестов, чтобы ускорить обратную связь. Например, в GitLab CI можно настроить разделение тестов на несколько джобов. Однако следите за изоляцией: параллельные тесты не должны мешать друг другу.

Тесты — это инвестиция в будущее проекта. Лучше меньше тестов, но таких, которые действительно проверяют поведение системы, чем сотни бесполезных проверок, создающих ложное чувство защищённости.

Заключение

Автоматические тесты — мощный инструмент, но только если они правильно спроектированы. Избегайте распространённых ошибок: добавляйте конкретные утверждения, изолируйте тесты, проверяйте граничные случаи и применяйте мутационное тестирование. Регулярно пересматривайте тестовый набор, чтобы он оставался актуальным. Помните, что цель тестов — находить ошибки, а не создавать видимость благополучия.

Источник: <https://cms.dolnais.ru/programmirovanie-kak-pisat-testy-kotorye-deystvitelno-proveryayut-kod>