Модульное тестирование

Юрий Литвинов
y.litvinov@spbu.ru

Модульное тестирование: зачем?

1. Любая программа содержит ошибки
2. Если программа не содержит ошибок, их содержит алгоритм, который реализует эта программа
3. Если ни программа, ни алгоритм ошибок не содержат, такая программа даром никому не нужна

Пример

Консольный калькулятор, складывающий два двузначных числа

- Называется adder
- Ввод числа заканчивается нажатием на *Enter*
- Программа должна вывести сумму после ввода второго числа

Смоук-тест

Что делаем	**Что происходит**
Вводим *adder* и жмём на *Enter*	Экран мигает, внизу появляется знак вопроса
Нажимаем 2	За знаком вопроса появляется цифра 2
Нажимаем *Enter*	В следующей строке появляется знак вопроса
Нажимаем 3	За вторым знаком вопроса появляется цифра 3
Нажимаем *Enter*	В третьей строке появляется 5, несколькими строками ниже — ещё один знак вопроса

Выявленные проблемы

- Нет названия программы на экране, может, мы запустили не то
- Нет никаких инструкций, пользователь без идей, что делать
- Непонятно, как выйти

План дальнейших тестов

Ввод	**Ожидаемый результат**	**Замечания**
99 + 99	198	Пара наибольших допустимых чисел
-99 + -99	-198	Отрицательные числа, почему нет?
99 + -14	85	Большое первое число может влиять на интерпретацию второго
-38 + 99	61	Отрицательное плюс положительное
56 + 99	155	Большое второе число может повлиять на интерпретацию первого
9 + 9	18	Два наибольших числа из одной цифры
0 + 0	0	Программы часто не работают на нулях
0 + 23	23	0 — подозрительная штука, его надо проверить и как первое слагаемое,
-78 + 0	-78	и как второе

План дальнейших тестов (2)

Ввод	**Замечания**
100 + 100	Поведение сразу за диапазоном допустимых значений
Enter + *Enter*	Что будет, если данные не вводить вообще
123456 + 0	Введём побольше цифр
1.2 + 5	Вещественные числа, пользователь может решить, что так можно
A + b	Недопустимые символы, что будет?
Ctrl-A, Ctrl-D, F1, Esc	Управляющие клавиши часто источник проблем в консольных программах

Ещё больше тестов!

- Внутреннее хранение данных — двузначные числа могут хранить в **byte**
  - 99 + 99, этот случай покрыли
- Кодовая страница ввода: символы ’/’, ’0’, ’9’ и ’:’
  - Программист может напутать со строгостью неравенства при проверке
  - Не надо вводить A + b, достаточно граничные символы

Информация к размышлению

- Программа из сотни строк может иметь 10^{18} путей исполнения
  - Времени жизни вселенной не хватило бы, чтобы их покрыть
- После передачи на тестирование в программах в среднем от 1 до 3 ошибок на 100 строк кода
- В процессе разработки — 1.5 ошибок на 1 строку кода (!)
- Если для исправления ошибки надо изменить не более 10 операторов, с первого раза это делают правильно в 50% случаев
- Если для исправления ошибки надо изменить не более 50 операторов, с первого раза это делают правильно в 20% случаев

Модульные тесты

- Тестирование отдельного класса
- Проверяют внешнее поведение класса
- Полностью автоматические
- Направлены на поиск ошибок в конкретном методе
- Не влияют на функциональность системы и не поставляются пользователю

Почему модульные тесты полезны

- Помогают искать ошибки
  - Особо эффективны, если налажен процесс Continuous Integration
- Облегчают изменение программы
  - Помогают при рефакторинге
- Тесты — документация к коду
- Помогают улучшить архитектуру
- НЕ доказывают отсутствие ошибок в программе

“Базовые” библиотеки модульного тестирования

- JVM: JUnit 5 (хотя JUnit 4 тоже довольно популярен)
  - Kent Beck и Erich Gamma, аж 1997 год
  - Самая популярная библиотека для JVM
- C++: Google Test, Boost.Test, CTest, Qt Test
- Python: PyTest
- .NET: NUnit, xUnit, Microsoft Unit Test Framework

Пример, юнит-тесты в C#

- Для тех, кто всё пропустил: https://msdn.microsoft.com/en-us/library/hh694602.aspx#BKMK_Quick_starts

Data-driven-тесты

```
[TestCase(12, 3, 4)]
[TestCase(12, 2, 6)]
public void DivideTest(int n, int d, int q)
{
    Assert.AreEqual(q, n / d);
}
```

Или даже

```
[TestCase(12, 3, ExpectedResult = 4)]
[TestCase(12, 2, ExpectedResult = 6)]
public int DivideTest(int n, int d)
{
    return n / d;
}
```

Best practices (1)

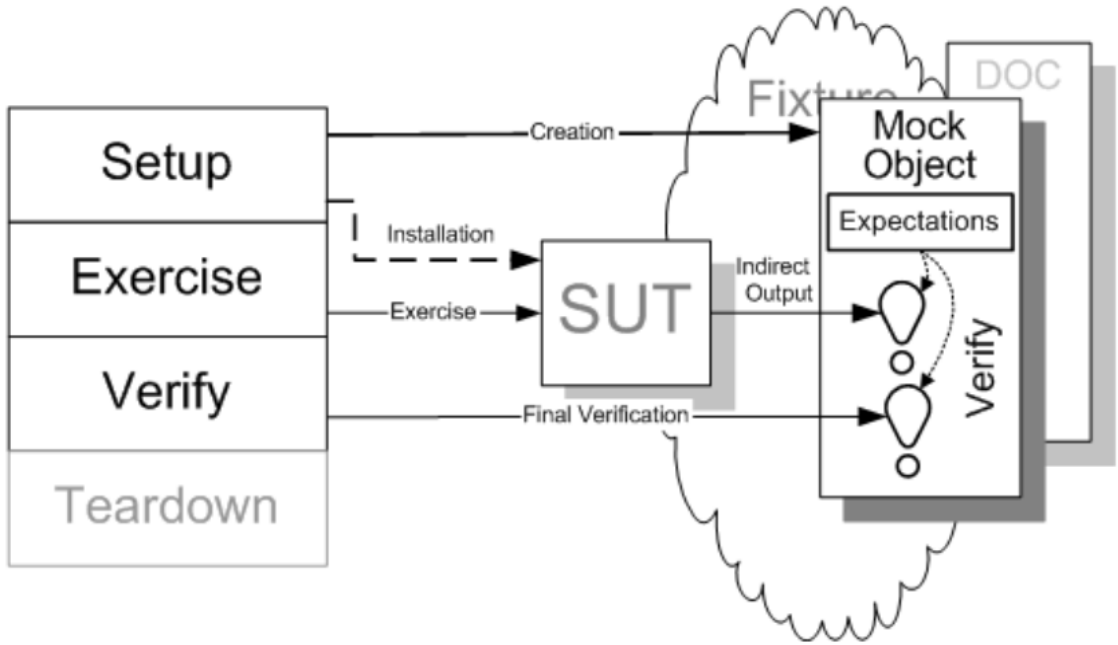
- Чёткое разделение на три фазы (Arrange-Act-Assert)
  - Настройка тестового окружения и тестируемой системы (SUT)
  - Выполнение действия
  - Проверка результатов
- Именование тестов: “в таких-то условиях should происходить то-то”
  - https://stackoverflow.com/questions/155436/unit-test-naming-best-practices
- Модульные тесты тестируют один модуль
  - Mock-объекты

Best practices (2)

- Независимость тестов
  - Желательно, чтобы поломка одного куска функциональности ломала один тест
  - Подчищать за собой данные, не использовать глобальное состояние
  - Тесты могут исполняться параллельно!
- Тесты должны работать быстро
  - И запускаться после каждой сборки
    - Continuous Integration!
- Тестов должно быть много
  - Следить за Code coverage
- Каждый тест должен проверять конкретный тестовый сценарий
  - Исключить случайность, не использовать try/catch в тестах
- Test-driven development

Mock-объекты

- Объекты-заглушки, подставляемые вместо реальных



Когда использовать

- Когда тестируемый модуль вызывает много чего ещё — непонятно, что сломалось, если тест не прошёл
- Когда выполнение реальной операции долго или небезопасно
- Когда есть механизм внедрения mock-объекта — dependency injection
  - Хороший код и так должен это делать

Библиотеки mock-объектов

- JVM: Mockito, EasyMock
- C++: Google Test
- Python: unittest.mock
- .NET: Moq, NSubstitute

Пример, Mockito, классический тест

```
@Test
public void test() throws Exception {
    // Arrange
    var testee = new UnitToTest();
    var helper = new Helper();
    // Act
    testee.doSomething(helper);
    // Assert
    assertTrue(helper.somethingHappened());
}
```

Пример, Mockito, тест с mock-объектом

```
@Test
public void test() throws Exception {
    // Arrange, prepare behaviour
    var testee = new UnitToTest();
    Helper aMock = mock(Helper.class);
    when(aMock.isCalled()).thenReturn(true);
    // Act
    testee.doSomething(aMock);
    // Assert - verify interactions (optional)
    verify(aMock).isCalled();
}
```

Матчеры

- Писать Assert.AreEqual неудобно и не всегда типобезопасно
- Сейчас модна объектная модель условия, те самые Matchers (или Constraints)
  - Пример (NUnit): Assert.That(array, Has.Exactly(1).EqualTo(3));
- Библиотеки: Hamcrest (JUnit) / NHamcrest (NUnit)
- Многие библиотеки юнит-тестирования умеют что-то такое “из коробки”

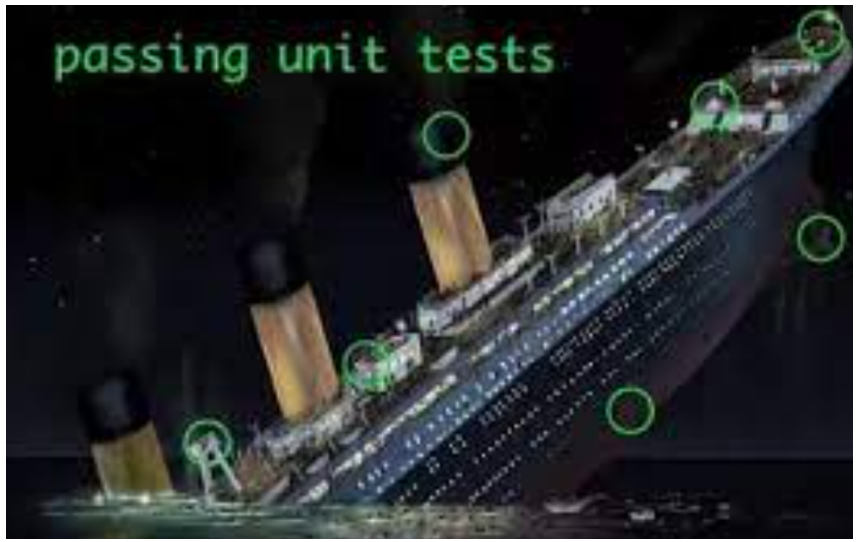
Property-driven testing, фаззинг

- А пусть тестовая система сама генерирует данные на вход!
- Пример, FsCheck:

```
open FsCheck

let revRevIsOrig (xs:list<int>) = List.rev (List.rev xs) = xs

Check.Quick revRevIsOrig
// Ok, passed 100 tests.

let revIsOrig (xs:list<int>) = List.rev xs = xs

Check.Quick revIsOrig
// Falsifiable, after 2 tests (2 shrinks) (StdGen (338235241,296278002)):
// Original:
// [3; 0]
// Shrunk:
// [1; 0]
```

UI-тестирование

- Есть ещё интеграционные тесты и UI-тесты
- UI-тестирование: Selenium (Web), FlaUI (или White) и подобные штуки — настольные приложения



© Интернеты