Модульное тестирование

Понятие модульного тестирования

- Модуль – это минимальная смысловая единица исходного кода, пригодная для тестирования (функция, процедура, метод, в отдельных случаях – класс).
- Модульный тест проверяет работоспособность модуля в условиях изоляции от других модулей приложения.

Цели тестирования

- Рефакторинг – изменение (улучшение ясности и эффективности, профилирование) кода без изменения функциональности
- Расширение кода с проверкой неизменности задуманной функциональности

За

- Лучше качество кода. По сути, комбинация “основного” кода и модульных тестов – это двойная запись одной и той же функциональности.
- Модульные тесты описывают прототип приложения – заложенную структуру, базовые алгоритмы. Соответственно, код реализует идеи разработчика, выраженные в модульных тестах (перевод бизнес-языка в язык программного кода).
- Рефакторинг. Цель не пропустить баги в старой функциональности при добавлении новой.
- Скорость нахождения ошибок.
- Возможность тестирования базовой функциональности без UI.
- Модульные тесты служат дополнением к документации.
- Точечная настройка производительности системы и обнаружение утечек памяти.

Против

- Время – деньги. Вместо написания кода, который можно продать, разработчик будет трудиться над тестами.
- Модульных тестов недостаточно для качественного тестирования приложений.
- Модульные тесты выполняются в стерильных условиях.
- Модульные тесты покажут наличие ошибок, но не докажут их отсутствие.

Тесты оправданы

- Бизнес-требования меняются часто, код меняется часто (добавляется новая функциональность, проводится рефакторинг, выполняется починка багов).
- Большая информационная система, которая образована модулями, находящимися в зоне ответственности нескольких групп разработчиков.
- Высококритичные системы (медицина), системы повышенной опасности (ядерные объекты).
- Существует потребность быстро получить оценку текущей работоспособности системы.
- Возможность чинить баги с минимальными издержками.
- Крупные, серьезные проекты.

Разработка тестов не оправдана

- Код-однодневка (в частности фриланс). Модульные тесты отнимут время и деньги.
- Идея модульных тестов не принимается частью разработчиков. Надежность системы равна надежности самого слабого звена.
- Не предполагается возможность обновлять модульные тесты.
- Модульные тесты для старого, унаследованного кода.
- Не нужно писать модульные тесты “для галочки”.

Пример

- Решение квадратного уравнения в действительных числах

Пример

- ```
  from math import *

  def SquareEquation(a,b,c):
      if a!=0:
          d=b**2-4*a*c
          if d==0:
              return [-b/(2*a)]
          elif d>0:
              return [(-b-sqrt(d))/(2*a), (-b+sqrt(d))/(2*a)]
          else:
              raise ArithmeticError("Действительных корней нет.")
      else:
          if b==c==0:
              raise ArithmeticError("Корень любое число.")
          elif b!=0:
              return [-c/b]
          else:
              raise ArithmeticError("Корней нет.")
  ```
  ```

Тест примера

```
import unittest
    from SquareEquation import *

    class TestSquareEquation(unittest.TestCase):

        def setUp(self): pass

        def testEqualRoots(self):
            self.assertListEqual(SquareEquation(1,2,1),[-1])

        def testTwoRoots(self):
            self.assertListEqual(SquareEquation(1,0,-1),[-1, 1])

        def testOneRoot(self):
            self.assertListEqual(SquareEquation(0,1,-1),[1])

        def testErrorRaises(self):
            with self.assertRaises(ArithmeticError):
                SquareEquation(0, 0, 0)

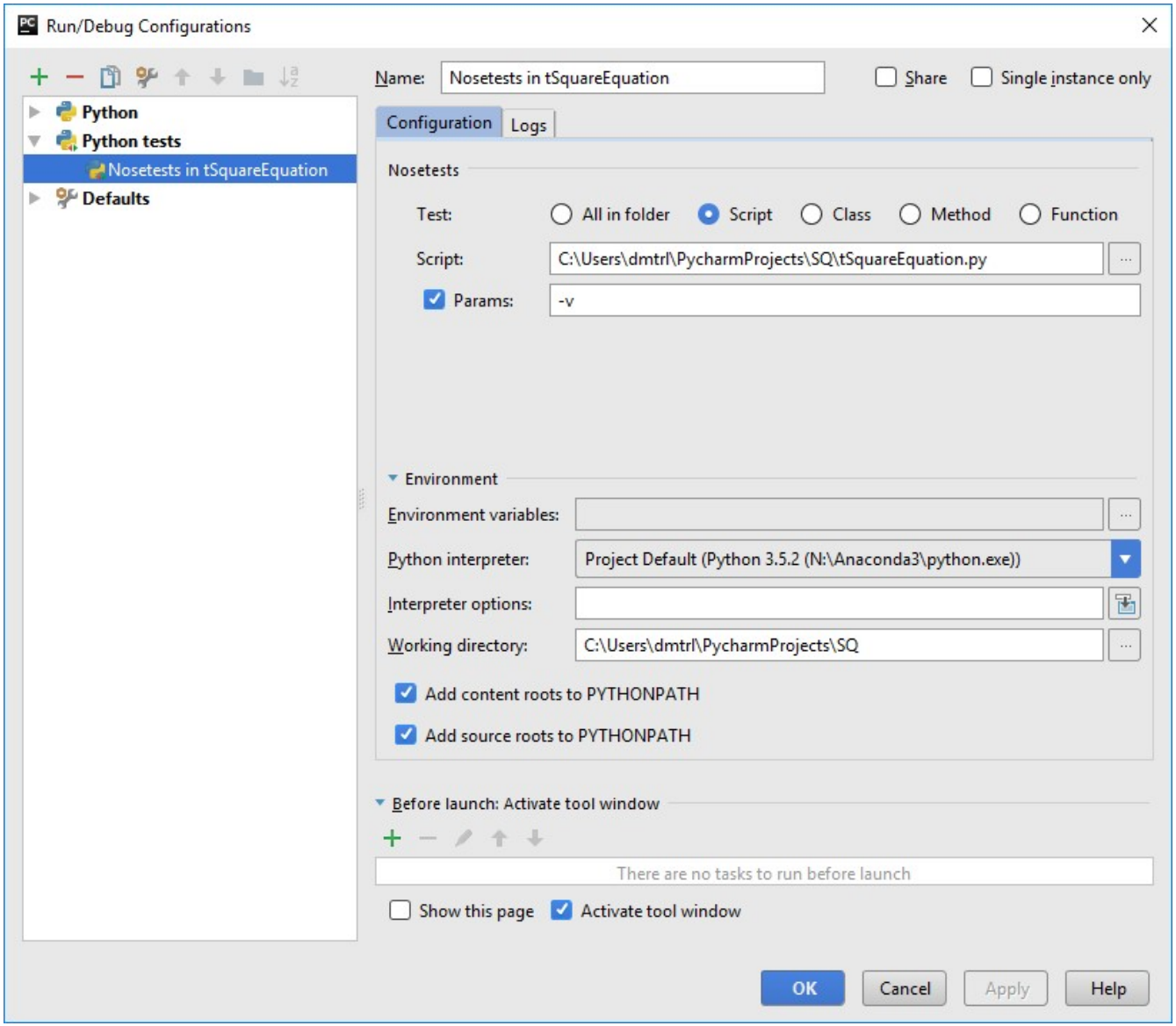
        def testErrorRaises2(self):
            self.assertRaises(ArithmeticError, SquareEquation, 0, 0, -1)

        def testErrorRaises3(self):
            self.assertRaises(ArithmeticError, SquareEquation, 4, 1, 4)

        def tearDown(self):   pass

    if __name__ == '__main__':
        unittest.main()
```

Настройки в PyCharm



Результаты работы теста

```
N:\Anaconda3\python.exe "C:\Program Files (x86)\JetBrains\PyCharm
    Community Edition 2016.3.2\helpers\pycharm\noserunner.py"
    C:\Users\dmtrl\PycharmProjects\SQ\tSquareEquation.py -v
Testing started at 2:36 ...
testEqualRoots (tSquareEquation.TestSquareEquation) ... .ok
testErrorRaises (tSquareEquation.TestSquareEquation) ... .ok
testErrorRaises2 (tSquareEquation.TestSquareEquation) ... .ok
testErrorRaises3 (tSquareEquation.TestSquareEquation) ... .ok
testOneRoot (tSquareEquation.TestSquareEquation) ... .ok
testTwoRoots (tSquareEquation.TestSquareEquation) ... .ok
----------------------------------------------------------------------
Ran 6 tests in 0.018s

OK
```

Термины

- **Test Case** (Тестовый сценарий) – минимальный блок тестирования. Он проверяет ответы для разных наборов данных. Это тестовый сценарий (набор проверяемых условий, переменных, состояний системы или режимов). Обычно является логически неделимым и может содержать одну или более проверок (asserts). Модуль unittest предоставляет базовый класс TestCase, который можно использовать для создания новых тестовых случаев.

Термины

- **Test Suite** (Набор тестов) – набор Test Case в рамках одного класса либо в рамках модуля.
- **Test Fixture** (Испытательный стенд) – набор средств для подготовки, необходимой для выполнения тестов, и действия для очистки после выполнения тестов. Например, создание временных баз данных или запуск серверного процесса.
- **Test Runner** (Исполнитель тестов) – компонент, который управляет выполнением тестов и предоставляет пользователю результат. Исполнитель может использовать графический или текстовый интерфейс или возвращать специальное значение, которое сообщает о результатах выполнения тестов.

Термины

- **setUp** – функция, реализующая предварительную подготовку (создания экземпляров классов, открытие соединений) для прогона тестов. Может относится к конкретному тесту (setUp), к набору тестов (setUpClass) или к модулю (setUpModule).
- **tearDown** – функция, реализующая окончательную очистку данных и закрытие всех ресурсов.

Использование assert

```
import unittest

class BaseTestClass(unittest.TestCase):
    def test_add(self):
        self.assertEquals(120, 100 + 20)
        self.assertFalse(10 > 20)
        self.assertGreater(120, 100)
    def test_sub(self):
        self.assertEquals(100, 140 - 40)

if __name__ == '__main__':
    unittest.main()
```

Проверки на успешность

assertEqual(a, b) — a == b
assertNotEqual(a, b) — a != b
assertTrue(x) — bool(x) is True
assertFalse(x) — bool(x) is False
assertIs(a, b) — a is b
assertIsNot(a, b) — a is not b
assertIsNone(x) — x is None
assertIsNotNone(x) — x is not None
assertIn(a, b) — a in b
assertNotIn(a, b) — a not in b
assertIsInstance(a, b) — isinstance(a, b)
assertNotIsInstance(a, b) — not isinstance(a, b)
assertRaises(exc, fun, *args, **kwds) — fun(*args, **kwds) порождает исключение exc
assertRaisesRegex(exc, r, fun, *args, **kwds) — fun(*args, **kwds) порождает исключение exc и сообщение соответствует регулярному выражению r
assertWarns(warn, fun, *args, **kwds) — fun(*args, **kwds) порождает предупреждение
assertWarnsRegex(warn, r, fun, *args, **kwds) — fun(*args, **kwds) порождает предупреждение и сообщение соответствует регулярному выражению r
assertAlmostEqual(a, b) — round(a-b, 7) == 0
assertNotAlmostEqual(a, b) — round(a-b, 7) != 0
assertGreater(a, b) — a > b
assertGreaterEqual(a, b) — a >= b
assertLess(a, b) — a < b
assertLessEqual(a, b) — a <= b
assertRegex(s, r) — r.search(s)
assertNotRegex(s, r) — not r.search(s)
assertCountEqual(a, b) — a и b содержат те же элементы в одинаковых количествах, но порядок не важен

Два класса – два тестовых сценария

```
import unittest

    class FirstTestClass(unittest.TestCase):

        def test_add(self):
            self.assertEquals(120, 100 + 20)

    class SecondTestClass(unittest.TestCase):

        def test_sub(self):
            self.val = 210
            self.assertEquals(210, self.val)
            self.val = self.val - 40
            self.assertEquals(170, self.val)

        def test_mul(self):
            self.val = 210
            self.assertEquals(420, self.val * 2)


    if __name__ == '__main__':
        unittest.main()
```

Подготовка теста

```
class SecondTestClass(unittest.TestCase):

    def setUp(self):
        self.val = 210

    def test_sub(self):
        self.assertEquals(210, self.val)
        self.val = self.val - 40
        self.assertEquals(170, self.val)

    def test_mul(self):
        self.assertEquals(420, self.val * 2)
```

Средства испытательного стенда (Test Fixture)

- **setUp** – подготовка прогона теста; вызывается перед каждым тестом.
- **tearDown** – вызывается после того, как тест был запущен и результат записан. Метод запускается даже в случае исключения (exception) в теле теста.
- **setUpClass** – метод вызывается перед запуском всех тестов класса.
- **tearDownClass** – вызывается после прогона всех тестов класса.
- **setUpModule** – вызывается перед запуском всех классов модуля.
- **tearDownModule** – вызывается после прогона всех тестов модуля.

Средства испытательного стенда (Test Fixture)

setUpClass и **tearDownClass** создаются с декоратором **@staticmethod**

setUpModule и **tearDownModule** - реализуются в виде отдельных функций в модуле и не входят ни в один класс модуля.

unittest - статусы тестов

.	ok
F	FAIL
E	ERROR
x	expected failure
s	skipped 'msg'
u	unexpected success

Декораторы unittest

@unittest.skip(reason)
– пропустить тест. reason описывает причину пропуска.

@unittest.skipIf(condition, reason)
– пропустить тест, если condition истинно.

@unittest.skipUnless(condition, reason)
– пропустить тест, если condition ложно.

@unittest.expectedFailure
– пометить тест как ожидаемая ошибка.

Для пропущенных тестов не запускаются setUp() и tearDown().

Для пропущенных классов не запускаются setUpClass() и tearDownClass().

Для пропущенных модулей не запускаются setUpModule() и tearDownModule().

Декораторы unittest

```
import unittest

    class BaseTestClass(unittest.TestCase):

        @unittest.skip('not supported')
        def test_skip(self):
            self.assertEquals(1000, 10 * 10 * 10)

        @unittest.expectedFailure
        def test_expected(self):
            raise ZeroDivisionError('Error! Division by zero')

    if __name__ == '__main__':
        unittest.main()
```

Работа с модулями

Проблема: есть несколько тестовых сценариев (модулей), находящихся в разных файлах. Как их запустить все сразу?

Работа с модулями

- Для прогона тестов из нескольких файлов, создается набор тестов (**Test Suite**) и в него вносятся тесты из классов (модулей).
- Для занесения тестов с разных модулей мы используем класс **Testloader**, имеющий методы:

  loadTestsFromModule()

  loadTestsFromName ()

  loadTestsFromNames ()

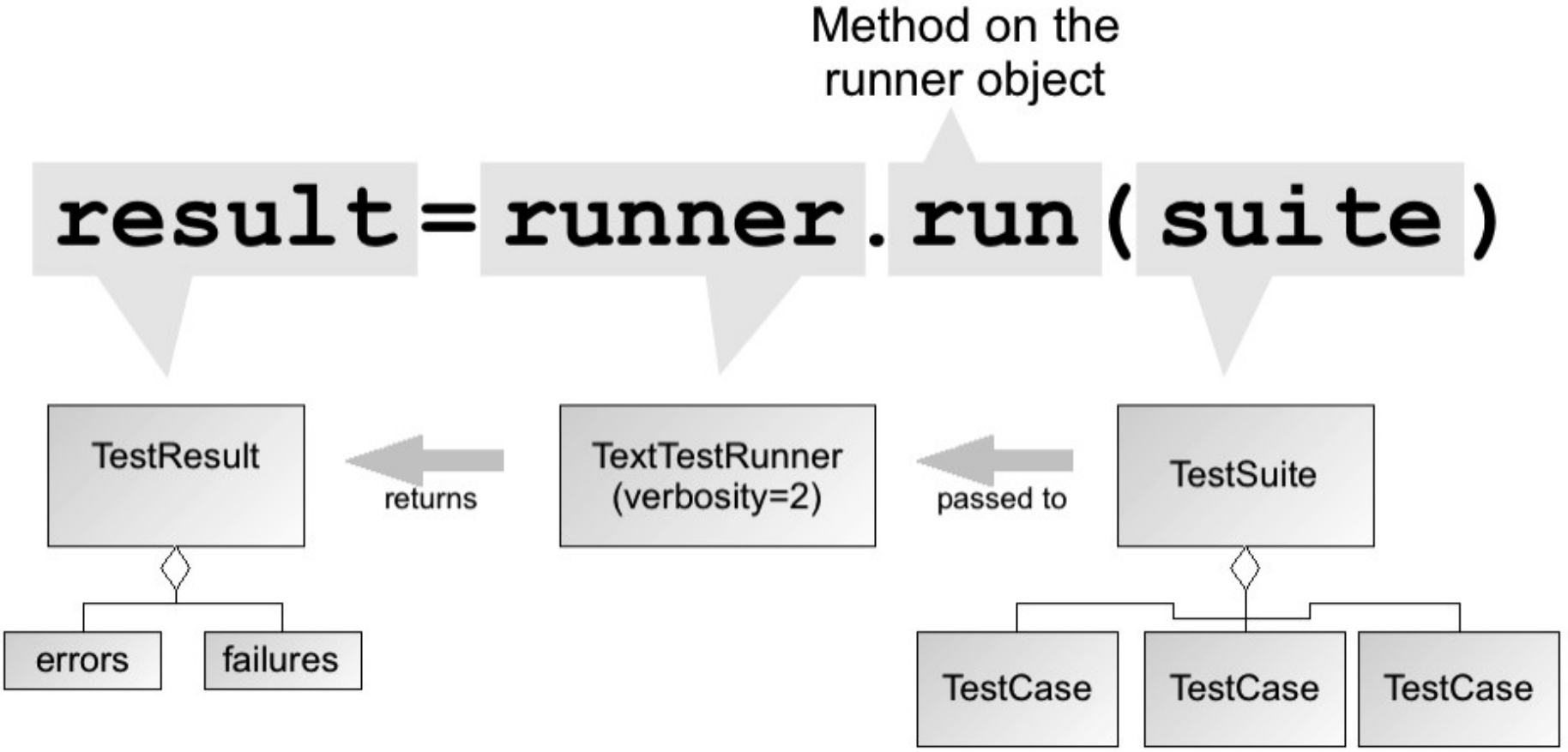
  loadTestsFromTestCase ()

Базовые блоки unittest

- **TestResult** и унаследованный от него **TextTestResult**:
хранят результаты выполнения тестов.
- **TextTestRunner –** запускает тесты на прогон и работает с **TextTestResult** для оповещения об успехе/провале прохождения теста.
- **TestLoader –** класс предназначен для создания коллекций тестов (**Test Suite**)

Тесты подгружаются через **TestLoader** в **TestSuite**,**TestRunner** принимает на вход сформированный на первом шаге **TestSuite** и запускает тесты, далее, по факту прохождения тестов, заполняется **TestResult**.

Работа исполнителя тестов



Пример

```
import unittest

    import test_1
    import test_ 2

    loader = unittest.TestLoader()

    suite = loader.loadTestsFromModule(test_ 1)
    suite.addTests(loader.loadTestsFromModule(test_ 2))

    runner = unittest.TextTestRunner(verbosity=2)
    result = runner.run(suite)
```

Запустить все тесты сразу

```
import unittest

loader = unittest.TestLoader()

# Если мои тесты начинаются со слова test_
suite = loader.discover(start_dir='.', pattern='test_ *.py')
runner = unittest.TextTestRunner(verbosity=2)
result = runner.run(suite)
```

Запустить все тесты сразу

Из папки проекта

python -m unittest discover

Источники

- https://docs.python.org/3.2/library/unittest.html
- http://blog.openquality.ru/unit-tests-why/
- https://pythonworld.ru/moduli/modul-unittest.html
- http://python-lab.ru/documentation/27/stdlib/unittest.html
- http://www.voidspace.org.uk/python/articles/introduction-to-unittest.shtml#loaders-runners-and-all-that-stuff