Пример разработки через тестирование

История модульного тестирования

Модульное тестирование — довольно старая идея, но особую актуальность она приобрела в последние двадцать лет в связи с развитием методик экстремального программирования.

В 1975-м году оно упоминается в "Мифическом человеко-месяце" Фредерика Брукса

В 1979-м году оно подробно описано в книге "The Art of Software Testing" Гленфорда Майерса.

В 1987-м году IEEE выпустила специальный стандарт модульного тестирования ПО.

В 1999-м году Кент Бек в книге "Extreme Programming Explained" сформулировал основные идеи TDD — методики, которая ставит тестирование во главу угла.

Цикл разработки

Основой TDD является цикл **"red -> green -> refactor"**.

В первой фазе программист пишет тест, во второй — код, необходимый для того, чтобы тест работал, в третьей, при необходимости, производится рефакторинг. Последовательность фаз очень важна.

Пример TDD

В данном примере **не используется никаких библиотек для тестирования** (testing framework), модульные тесты делаются "вручную". При первом чтении это упростит восприятие идеи модульного теста. В следующих материалах будет указано два более элегантных пути создавать тесты.

1. Создаём тесты и заготовку программного кода

1.1. Создаём модульные тесты

Допустим, перед нами стоит задача реализовать сортировку списка. Вместо того, чтобы сразу броситься в бой, мы создадим модульные тесты, и благодаря этому мы сразу продумываем спецификацию функции, фиксируя её в наборе тестов.

```
def test_sort():
    print("Test #1")
    print("testcase #1: ", end="")
    A = [4, 2, 5, 1, 3]
    A_sorted = [1, 2, 3, 4, 5]
    sort_algorithm(A)
    passed = A == A_sorted
    print("Ok" if passed else "Fail")

    print("testcase #2: ", end="")
    A = []
    A_sorted = []
    sort_algorithm(A)
    passed = A == A_sorted
    print("Ok" if passed else "Fail")

    print("testcase #3: ", end="")
    A = [1, 2, 3, 4, 5]
    A_sorted = [1, 2, 3, 4, 5]
    sort_algorithm(A)
    passed = A == A_sorted
    print("Ok" if passed else "Fail")

test_sort()
```

В использованном интерфейсе sort_algorithm() мы заложились на то, что список сортируется на месте. При этом пустой список остаётся пустым, отсортированный по возрастанию список остаётся отсортированным.

1.2. Пишем заглушку

Теперь создаем тестируемую функцию-заглушку, необходимую для того, чтобы код выполнялся. Как только заглушка написана, нужно запустить тесты, и они не должны показывать ОК.

```
def sort_algorithm(A):
    pass

test_sort()
```

Прогоняем тесты:

```
Test #1
testcase #1: Fail
testcase #2: Ok
testcase #3: Ok
```

К сожалению, тест разработан плохо. Заглушка, которая ничего не делает, прошла две трети тестовых сценариев... Может показаться, что функция сортировки почти работает, а это совсем не так.

Значит, во-первых, **мы должны выдавать итоговый Fail, даже если один раз случился Fail**. А во-вторых, нужно сделать **больше тестов** на функциональное действие, а не бездействие. Как минимум, можно добавить случай инвертированного списка, а также случай, когда в списке есть повторяющиеся числа. Так-так!

Хм-м-м... Кстати, не стоит ли проверить устойчивость сортировки (не переставляет ли она местами одинаковые значения)?

И не проверить ли сортировку других объектов — дробных чисел, строк, кортежей, а не только целых чисел?

А что с допустимой длиной списка? Какой длины список должно быть возможно отсортировать нашей функцией за разумное время? Ведь существуют алгоритмы сортировки совершенно разных асимптотик...

Эти вопросы, возникшие уже на первом этапе разработки по TDD, являются самым ценным последствием использования этой методологии! TDD буквально **заставляет сконцентрироваться на спецификации интерфейса функции**. В этот момент мы должны **остановить дальнейшую разработку и уточнить требования**: заглянуть в проектную документацию, пойти к начальнику/заказчику/тимлидеру или, если сам себе начальник, принять чёткие обоснованные решения и зафиксировать их в тестах. При этом хорошо бы написать названия этим тестам так, чтобы было понятно, что именно они тестируют.

Допустим, мы сходили к руководителю группы разработчиков (тимлидеру) и получили ответы на все наши вопросы:

1) Должна ли сортировка быть устойчивой? — **Да.**

2) Должна ли сортировка быть универсальной? — **Да.**

3) Максимальная длина сортируемого списка? — **100 элементов.**

4) Какая требуется асимптотика? — **Квадратичная, O(N^2)**.

*(Сейчас не будем обсуждать, что в Python есть стандартная универсальная прагматическая сортировка за O(N*logN). Наша задача — на примере Bubble sort, известного "велосипеда", показать ход разработки TDD.)*

1.3. Исправляем тесты

Исправить их нужно так, чтобы тестирование заглушки выдавало отрицательный результат. Но главное — чтобы все наши утверждения про сортировку были зафиксированы в тестах.

В разработке тестов будем следовать структурному программированию и движению "сверху-вниз". Разобьём тесты на отдельные функции с человеко-понятными именами (соответствующими спецификации, определённой выше). Вызывать их будем из главной функции:

```
from random import shuffle  # it randomizes order of elements


def test_sort():
    print("Test sorting algorithm:")
    passed = True

    passed &= test_sort_works_in_simple_cases()
    passed &= test_sort_algorithm_stable()
    passed &= test_sort_algorithm_is_universal()
    passed &= test_sort_algorithm_scalability()

    print("Summary:", "Ok" if passed else "Fail")


def test_sort_works_in_simple_cases():
    print("- sort algorithm works in simple cases:", end=" ")
    passed = True

    for A1 in ([1], [], [1, 2], [1, 2, 3, 4, 5],
               [4, 2, 5, 1, 3], [5, 4, 4, 5, 5],
               list(range(20)), list(range(20, 1, -1))):
        A2 = sorted(list(A1))  # yes, we are cheating here to shorten example
        sort_algorithm(A1)
        passed &= all(x == y for x, y in zip(A1, A2))

    print("Ok" if passed else "Fail")
    return passed


def test_sort_algorithm_stable():
    print("- sort algorithm is stable:", end=" ")
    passed = True

    for A1 in ([[100] for i in range(5)],
               [[1, 2], [1, 2], [2, 2], [2, 2], [2, 3], [2, 3]],
               [[5, 2] for i in range(30)] + [[10, 5] for i in range(30)]):
        shuffle(A1)
        A2 = sorted(list(A1))  # here we are cheating: standard sort is stable
        sort_algorithm(A1)
        # to test stability we will check A1[i] not equals A2[i], but is A2[i]
        passed &= all(x is y for x, y in zip(A1, A2))

    print("Ok" if passed else "Fail")
    return passed


def test_sort_algorithm_is_universal():
    print("- sort algorithm is universal:", end=" ")
    passed = True

    # testing types: str, float, list
    for A1 in (list('abcdefg'),
               [float(i)**0.5 for i in range(10)],
               [[1, 2], [2, 3], [3, 4], [3, 4, 5], [6, 7]]):
        shuffle(A1)
        A2 = sorted(list(A1))
        sort_algorithm(A1)
        passed &= all(x == y for x, y in zip(A1, A2))

    print("Ok" if passed else "Fail")
    return passed


def test_sort_algorithm_scalability(max_scale=100):
    print("- sort algorithm on scale={0}:".format(max_scale), end=" ")
    passed = True

    for A1 in (list(range(max_scale)),
               list(range(max_scale//2, max_scale)) + list(range(max_scale//2)),
               list(range(max_scale, 0, -1))):
        shuffle(A1)
        A2 = sorted(list(A1))
        sort_algorithm(A1)
        passed &= all(x == y for x, y in zip(A1, A2))

    print("Ok" if passed else "Fail")
    return passed


def sort_algorithm(A):
    "Sorting of list A on place."
    pass


test_sort()
```

Теперь тестирование сделано достаточно подробно, чтобы специфицировать задачу данной конкретной сортировки. Чтение главной функции test_sort() позволяет понять свойства алгоритма кратко, а изучение той или иной тестирующей функции даёт нам детальное понимание того или иного свойства.

Важно и то, что эта в кавычках "документация" является действующими критериями, которые одобрят наш код только тогда, когда он будет им соответствовать.

1.4. Убеждаемся, что заглушка не проходит обновлённые тесты

Запустите код, приведённый выше, и получите такое резюме тестирования:

```
Test sorting algorithm:
- sort algorithm works in simple cases: Fail
- sort algorithm is stable: Fail
- sort algorithm is universal: Fail
- sort algorithm on scale=100: Fail
Summary: Fail
```

Это победа! "Красный цвет" достигнут! Модульные тесты написаны, и можно переходить к реализации.

Вовремя будет сказать, что использование библиотеки unittest или библиотеки doctest позволяет существенно упростить оформление и укоротить длину поведенческих требований к функции, оформленных в юнит-тесты. Возможности двух этих библиотек будут показаны в следующих материалах курса.

2. Реализуем требуемую функциональность

Здесь нужно действовать быстро и решительно, ведь спецификация требований у нас на кончиках пальцев, задача предельно ясна и конкретна.

В соответствии с принципом "Test First", следует писать только такой код, который **абсолютно необходим**, чтобы тесты выполнялись успешно. Можно даже "вставить костыль", который не приводит к падению тестов.

Ради простоты демонстрации вначале сделаем небольшую ошибку в сортировке методом пузырька:

```
def sort_algorithm(A):
    """
    Sorting of list on place. Using Bubble Sort algorithm.
    """
    N = len(A)
    for i in range(N-1):
        for k in range(N-1):
            if A[k] >= A[k+1]:
                A[k], A[k+1] = A[k+1], A[k]
```

Run
Reset

Результат тестирования показывает, что данная версия неустойчива:

```
Test sorting algorithm:
- sort algorithm works in simple cases: Ok
- sort algorithm is stable: Fail
- sort algorithm is universal: Ok
- sort algorithm on scale=100: Ok
Summary: Fail
```

Найдите ошибку (опечатку) и добейтесь "зелёного цвета" самостоятельно. Не переделывайте алгоритм полностью, ведь радикальное усовершенствование нам предстоит делать уже на "зелёный цвет".

3. Делаем рефакторинг

Когда типичный программист приступает к рефакторингу, он нервничает и чрезвычайно напряжён. Ведь причиной рефакторинга чаще всего является то, что программный код стал уже *настолько* неразборчив, что перестал работать, а где именно — большой вопрос... Такой рефакторинг на "красный свет" может не только не решить проблемы, но и сломать программу, вплоть до развала проекта по сценарию "Вавилонской башни". В любой момент рефакторинга при переписывании кода программист может что-то сломать и даже не заметить этого. Неспроста слово "рефакторинг" для многих ИТ-менеджеров как красная тряпка для быка: как руководитель, он не понимает смысла этой "возни с уже написанным кодом", а также боится регрессии.

Выход один — рефакторинг нужно начинать только при наличии модульных тестов и только на "зелёный цвет", чтобы всегда быть готовым откатиться на рабочую версию кода, если что-то пойдёт не так.

У нас как раз "зелёный цвет", поэтому **спокойно и смело переписываем код, запуская время от времени наши тесты**, пока не убедимся, что та версия сортировки, которая нам визуально нравится (понятная, чистая и практичная), проходит все тесты и получает итоговый ОК.

Остановимся на таком варианте:

```
def sort_algorithm(A):
    """
    Sorting of list on place. Using Bubble Sort algorithm.
    """
    N = len(A)
    list_is_sorted = False
    bypass = 1
    while not list_is_sorted:
        list_is_sorted = True
        for k in range(N - bypass):
            if A[k] > A[k+1]:
                A[k], A[k+1] = A[k+1], A[k]
                list_is_sorted = False
        bypass += 1
```

Запуск тестов:

```
Test sorting algorithm:
- sort algorithm works in simple cases: Ok
- sort algorithm is stable: Ok
- sort algorithm is universal: Ok
- sort algorithm on scale=100: Ok
Summary: Ok
```

Заключение

Результат применения TDD — счастье программиста, а именно:

1. уверенность в собственном коде,
2. отсутствие страха и спокойствие при рефакторинге,
3. наличие актуальной и автоматически проверяющейся спецификации поведения данной функции.

При этом не следует превращать TDD в догму, считая, что код, написанный по другим методологиям, заведомо плох. Любую самую прекрасную идею можно довести до абсурда.

Статьи для дополнительного чтения

1. "Модульное тестирование и Test-Driven Development, или Как управлять страхом в программировании", Сергей Белов
2. "Applying TDD in Your Company is More Important than Ever!", Dennis Nerush
3. "Эволюция юнит-теста", Андрей Солнцев

Ниже Вы можете скачать PDF версию данного материала.

Пример разработки через тестирование…

Mark as completed