Требования к написанию авто-тестов

Создал(а) Неизвестный пользователь (a.prokopeva), редактировал(а) Неизвестный пользователь (o.poveteva) 25 июл., 2024

Основная сложность тестирования математической модели в том, что нужно отличить функциональность программы от цели, для которой программа была сделана. Эта проблема происходит потому, что главная задача математической модели - выполнять операции с входными данными для получения описания какого-либо объекта или процесса, но процесс программной реализации так же завязан на операциях с входными параметрами. Поэтому для лучшего понимания тестирования в Модели КС выделены тестовые сущности, уровни и виды тестирования:

Тестовая сущность: Тесты (test)

Тесты нужны для уменьшения вероятности появления бага. В проекте используются следующие уровни тестов:

- **юнит-тесты**;
- **интеграционные тесты**:
  - компонентные,
  - высокоуровневые.

А также каждый из уровней может содержать в себе определенный вид тестов:

- **функциональные тесты**:
  - позитивные,
  - негативные.

> Описание уровней тестов приведено на соответствующих вкладках

Тестовая сущность: Пример (example)

Примером использования класса/функции в модели является скрипт с отрисовкой графиков и карт, которые нужно именно увидеть. Либо другой вариант, если правда нужно выделить именно пример использования функции/класса (в таком случае существует дублирующий тест).

Нет меток

Требования к написанию авто-тестов

Создал(а) Неизвестный пользователь (a.prokopeva), редактировал(а) Неизвестный пользователь (o.poveteva) 25 июл., 2024

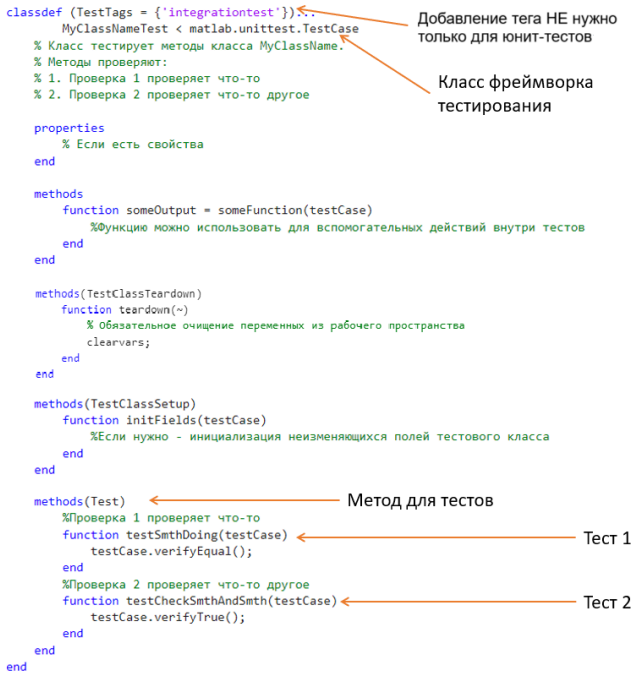
T1	Класс **Test Case** обязательно наследуется от соответствующего класса фреймворка используемого языка программирования (**matlab.unittest.TestCase**)
T2	Название класса теста повторяет название тестируемой функции или класса с большой буквы с добавкой “**Test**” на конце. *Пример 1: MyFunTest.m, MyClassTest.m*
T3	В задачу разработчика обязательно входит разработка кода и позитивное + негативное тестирование этого кода.
T4	Класс **Test Case** обязательно содержит в себе **methods(TestClassTeardown)** для очищения памяти рабочего пространства и по необходимости **methods(TestClassSetup)** для записи тестовых данных перед выполнением тестов.
T5	Для использования путей в тестах выбираем функцию projPath(). Иначе могут быть проблемы при автоматизированном запуске тестов. Это функция из репозитория matlab-models. **Для путей в тестах: projPath()**
T6	Все проверки осуществляются только через вызов verify.
T7	Одна тестовая функция содержит в себе один вызов verify, что соответствует требованию один тест - одна проверка.

Нет меток

Требования к написанию авто-тестов

Создал(а) Неизвестный пользователь (a.prokopeva), редактировал(а) Неизвестный пользователь (o.poveteva) 25 июл., 2024

На примере MATLAB:




```
classdef (TestTags = {'integrationtest'}) ...
        MyClassNameTest < matlab.unittest.TestCase
    % Класс тестирует методы класса MyClassName.
    % Методы проверяют:
    % 1. Проверка 1 проверяет что-то
    % 2. Проверка 2 проверяет что-то другое

    properties
        % Если есть свойства
    end

    methods
        function someOutput = someFunction(testCase)
            %Функцию можно использовать для вспомогательных действий внутри тестов
        end
    end

    methods(TestClassTeardown)
        function teardown(~)
            % Обязательное очищение переменных из рабочего пространства
            clearvars;
        end
    end

    methods(TestClassSetup)
        function initFields(testCase)
            %Если нужно - инициализация неизменяющихся полей тестового класса
        end
    end

    methods(Test)
        %Проверка 1 проверяет что-то
        function testSmthDoing(testCase)
            testCase.verifyEqual();
        end
        %Проверка 2 проверяет что-то другое
        function testCheckSmthAndSmth(testCase)
            testCase.verifyTrue();
        end
    end
end
```

Скачать шаблон.

Нет меток

Требования к написанию авто-тестов

Создал(а) Неизвестный пользователь (a.prokopeva), редактировал(а) Неизвестный пользователь (o.poveteva) 25 июл., 2024

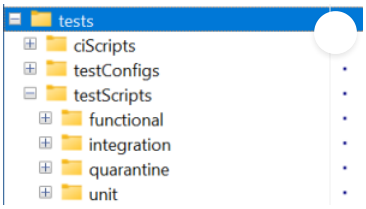
P1	Список всех возможных проверок для тестов тут.
P2	Тревожный звонок, когда необходимо переписывать тесты при рефакторинге класса/функции или добавлении новой фичи. Тесты должен быть независим от внутренней имплементации и структуры.
P3	Чтобы лучше понимать работу с тестами изучите методологию, изложенную в статье 10 эффективных практик реализации тестирования.
P4	Тесты не должны обращаться к одному и тому же изменяемому ресурсу (файлу, массиву, т.д.).
P5	Тесты не должны создавать/изменять один и тот же ресурс (файл, массив, т.д.).
P6	Для настройки глобальных переменных среды для запуска тестов можно пользоваться Fixture.
P7	Именование тестовых функций в коде должно совпадать с именованием тестовых случаев в Jira, но в латинской раскладке. Если именования тестовых случае в Jira нет, тогда название должно быть уникальным и без добавления каких-либо лишних деталей, которые не требуются для идентификации. Все правила для именования методов также распространяются и на тесты.
P8	Необходимо в комментариях к коду указывать ссылку на источник эталонных значений, если это возможно

Нет меток

Требования к написанию авто-тестов

Создал(а) Неизвестный пользователь (a.prokopeva), редактировал(а) Неизвестный пользователь (o.poveteva) 25 июл., 2024

Все тестовые файлы располагаются в папке test/testScripts/, для юнит-тестов папка unit/, для интеграционных папка integration/, а функциональные тесты располагаются в папке functional/ без подпапки. В папке quarantine/ располагаются тесты, которые сломались в следствии корректировок в коде разработки и их следует починить.



Примечание: Также, в этой папке test должна располагаться папка testConfig для хранения конфигурационных файлов при необходимости. Конфиги в папке должны разделяться папками, если они используются для разных тестовых классов.

Нет меток

Требования к написанию авто-тестов

Создал(а) Неизвестный пользователь (a.prokopeva), редактировал(а) Неизвестный пользователь (o.poveteva) 25 июл., 2024

Юнит-тесты (Unit tests) - проверяют смысловую работу отдельного фрагмента кода, который можно логически изолировать от другого фрагмента кода в системе. Фрагмент кода тестируется на предмет того, пригоден ли он для использования и работает ли как ожидалось.

При составлении юнит тестов нужно ориентироваться на смысл кода. Можно обращать внимание на следующие аспекты:

- Если проводятся вычисления по каким-то формулам, то для эталонных значений они вернут ожидаемый результат;
- При разбиении возможных значений на диапазоны, результат программы совпадает с эталонными как на диапазонах, так и на граничных значениях (см. Техники тест-дизайна);
- Если программа должна правильно реагировать на нефизичные наборы значений, то это происходит;
- и т.д.

Требования к Unit-тестам:

T1	Один Test Case класс содержит несколько Unit-тестов как методы в этом классе.
T2	Один Unit-тест должен проверять одну единицу функциональности. Не стоит писать длинные Unit-тесты, лучше много маленьких.
T3	Название Unit-теста должно отражать суть теста. Чтобы сразу было видно, где ошибка, если она будет выявлена. Избегайте сокращений.
T4	Unit-тесты должны быть атомарные и изолированные. Порядок запуска не должен иметь значения.

T5	Атомарность Unit-тестов сохраняется в неявном виде, если внутри одной функции содержится два или более вызова verify для одной и той же переменной в случае неравенств.
	
	Список методов, которые соответствуют исключению:
	verifyGreaterThan
	verifyGreaterThanOrEqual
	verifyLessThan
	verifyLessThanOrEqual
	
	Пример: если нужно проверить диапазон, можно вызвать verify единожды, а можно записать как два вызова:
	
	`function testResultEccentricityValues(testCase)`
	`testCase.verifyTrue((a <= b )&&(a >= c));`
	`end`
	
	`function testResultEccentricityValues(testCase)`
	`testCase.verifyGreaterThanOrEqual(a,c);`
	`testCase.verifyLessThanOrEqual(a,b);`
	`end`

Рекомендации к Unit-тестам:

P1	В юнит-тестах проверять всегда лучше как можно больше значений, даже самые очевидные. Если вы пишете тест для класса, проверяйте все его поля с данными.

Нет меток

Требования к написанию авто-тестов

Создал(а) Неизвестный пользователь (a.prokopeva), редактировал(а) Неизвестный пользователь (o.poveteva) 25 июл., 2024

Есть два уровня интеграционных тестов в зависимости от размеров тестовых объектов:

1. Компонентные интеграционные тесты - проверяется взаимодействие функций внутри класса или взаимодействие функций двух разных классов.
2. Высокоуровневые интеграционные тесты - проверяется взаимодействие всех классов между собой.

Требования к интеграционным тестам:

T1	Интеграционный тест проверяет либо взаимодействие функций между собой, либо взаимодействие классов.
T2	Интеграционные тесты должны концентрироваться на самой интеграции. *Пример 1:* при интеграции функции A с функцией B тесты должны быть сосредоточены на обмене данными между двумя функциями, а не на функциональности отдельной функции(это юнит тестирование).
T3	Одна функция тестового класса за раз тестирует одну интеграцию между двумя функциями/классами.
T4	Интеграционные тесты должны быть изолированные. Порядок запуска не должен иметь значения.

T5	Следует размещать несколько проверок в одном тесте, если они проверяют связанный функционал.
	Пример:
	function testApplyFilters(testCase)
	% Подготовка данных
	img = imread('test_image.png');
	processed = applyFilters(img);
	% Проверки
	testCase.verifySize(processed, size(img)); %размер
	testCase.verifyTrue(max(processed(:)) <= 255); %верхняя граница
	testCase.verifyFalse(any(processed(:) < 0)); %нижняя граница
	testCase.verifyEqual(mean(img), mean(processed)); %среднее значение
	end

T6	Для интеграционных тестов обязательно добавление тега перед названием класса. *Пример:* classdef (TestTags = {'integrationtest'})... NameTestClass < matlab.unittest.TestCase

Нет меток

Требования к написанию авто-тестов

Создал(а) Неизвестный пользователь (a.prokopeva), редактировал(а) Неизвестный пользователь (o.poveteva) 25 июл., 2024

Для функционального тестирования можно выделить:

1. Позитивное тестирование (positive testing) — это тестирование сценариев кода, которые соответствуют нормальному поведению системы, класс/функция делает то, для чего создавалась.
2. Негативное тестирование (negative testing) — это тестирование сценариев кода, которые соответствуют ожидаемому некорректному поведению системы — различные сообщения об ошибках, валидация неверного набора данных, проверка обработки исключительных ситуаций (как в реализации самих программных алгоритмов, так и в логике бизнес-правил).

Таким образом, при составлении функциональных тестов, нужно обратить внимание именно на код, а не смысл кода, стараясь просто создавать условия, чтобы срабатывала та или иная часть программы. “Работать - работает, как - не важно”. При этом можно обращать внимание на следующие аспекты реализации:

- Работоспособность программы без конкретики по результатам;
- Взаимодействие программы с входными и выходными параметрами хотя бы при каких-то условиях происходит так, как ожидается;
- Если программа должна что-то вывести в командную строку, то удаётся создать ситуацию, когда это происходит;
- Если иногда программа должна вернуть, не N, а M переменных, то удаётся создать такую ситуацию;
- Если программа иногда должна выдать ошибку, то удаётся создать такую ситуацию;
- и т.д.

Такие тесты практически не зависят от мировых констант, используемых в задаче; от порядка вычислений; от используемых систем единиц и т.д

Требования к функциональным тестам:

T1	Позитивные функциональные тесты неразрывны с разработкой, без них невозможно сделать релиз.
T2	Для позитивных и негативных тестов важно указывать в названии "**Functional**". *Пример: MyFunFunctionalTest.m, MyClassFunctionalTest.m*
T3	И позитивные и негативные тесты должны быть атомарные и изолированные. Порядок запуска не должен иметь значения.
T4	Для позитивных и негативных тестов обязательно добавление тега "functionaltest" перед названием класса. *Пример:* classdef (TestTags = {'functionaltest'})... NameTestClass < matlab.unittest.TestCase

Нет меток

Требования к написанию авто-тестов

Создал(а) Неизвестный пользователь (a.prokopeva), редактировал(а) Неизвестный пользователь (o.poveteva) 25 июл., 2024

Рассмотрим пример, как написать тесты на языке MATLAB для функции, которая решает квадратное уравнение.

1. Функция принимает на вход 3 входных параметра (a, b, c);
2. Вычисляет дискриминант и корни уравнения по формулам;
3. Возвращает массив с двумя корнями;

При написании тестов нужно рассматривать код с двух сторон, чтобы не запутаться где юнит тесты, а где функциональные:

1. Программная реализация
2. Математическая реализация

Всё, что касается программной реализации, относится к функциональному тестированию. Что касается математической реализации – к юнит тестированию.

> Конечно, такое разделение не является абсолютным и возможны исключения. Но к преимуществам такого подхода можно отнести:
>
> 1. Упрощение разбиения возможностей кода на тесты;
> 2. Возможность точной локализации ошибок. Если “упали” и функциональные тесты, и компонентные, то что-то не так с программой в целом. Если “упали” только компонентные тесты, то проблема где-то в математике.

В данном примере при функциональном тестировании проверяется:

- Выполнение программы от начала до конца;
- Взаимодействие с входными/выходными данными;
- Вывод информации в консоль;
- Возвращение переменных;

При проверке компонентов в юнит тестировании:

- Проверка вычислений по известным формулам;
- Проверка результата в разных диапазонах значений;
- Обработка нефизичных значений;

Скачать подробный разбор примера.

Нет меток

ПРИМЕР РАБОТЫ С КОДОМ ДЛЯ РЕАЛИЗАЦИИ ТЕСТОВ

Важно: *здесь и далее в листингах используются англоязычные комментарии и содержимое на вывод исключительно из-за особенностей пакета отображения кода в pdf-файле, что не поддерживает кириллицу. При действительной разработке и написании тестов необходимо придерживаться правил и использовать только русскоязычные пометки.*

Пусть имеется функция, которая решает квадратное уравнение:

```
function [roots] = quadraticSolver(a,b,c)
        arguments
            a (:, 1) {mustBeNonempty, mustBeNumeric, mustBeFinite}
            b (:, 1) {mustBeNonempty, mustBeNumeric, mustBeFinite}
            c (:, 1) {mustBeNonempty, mustBeNumeric, mustBeFinite}
        end

        if any(a == 0)
            error('quadraticSolver:IncorrectInput', ...
            'Incorrct input! a must not be equal zero!')
        end

        if any(b.^2 - 4.*a.*c < 0)
            error('quadraticSolver:ScaryInput', ...
            'The creator of this function has a phobia of complex
               numbers!')
        end

        roots = zeros(numel(a), 2);

        roots(:, 1) = (-b + sqrt(b.^2 - 4.*a.*c)) ./ (2.*a);
        roots(:, 2) = (-b - sqrt(b.^2 - 4.*a.*c)) ./ (2.*a);

        if any(roots(:, 1) == roots(:, 2))
            warning('quadraticSolver:WowWarning', ...
            'Amazing! Some set of numbers gave a discriminant equal
                to zero!')
        end
    end
```

Рассмотрим результат её позитивного выполнения (т.е. без ошибок), начиная с самого абстрактного и заканчивая максимально конкретным.

1. Программа выполняется до конца;
2. Программа выполняется до конца и возвращает одну непустую переменную;
3. Программа выполняется до конца и возвращает одну непустую переменную типа double;
4. Программа выполняется до конца и возвращает одну непустую переменную типа double с размером по строкам, равным длине первого входного аргумента, и 2-мя столбцами;
5. Программа может выполниться до конца, удовлетворяя вышеуказанным условиям, и дополнительно вернуть warning;
6. Программа выполняется до конца и возвращает одну непустую переменную типа double с размером по строкам, равным длине первого входного аргумента, и 2-мя столбцами, где содержится результат применения формулы $\frac{-b \pm \sqrt{b^2-4ac}}{2a}$ со знаком плюс или минус в 1 или 2 столбце соответственно;
7. Программа выполняется до конца и возвращает одну непустую переменную типа double с размером по строкам, равным длине первого входного аргумента, и 2-мя столбцами, где содержится результат применения формулы $\frac{-b \pm \sqrt{b^2-4ac}}{2a}$ со знаком плюс или минус в 1 или 2 столбце соответственно, а в случае наличия одинаковых решений в одной строке в командной строке возникает предупреждение.

Кажется, что описание основного для математической модели функционала программы заключено именно в 6 или 7 пункте. Но в случае, когда не работает функционал из пунктов 1-5, то и пункты 6-7 так же не работают.

Поэтому при написании тестов предлагается рассматривать код “с двух сторон”:

1. Программная реализация;
2. Математическая реализация.

Таким образом, при составлении functional tests, нужно обратить внимание именно на код, а не смысл кода, стараясь просто создавать условия, чтобы срабатывала та или иная часть программы. “Работать - работает, как

- не важно”. При этом можно обращать внимание на следующие аспекты реализации:

- Работоспособность программы без конкретики по результатам;
- Взаимодействие программы с входными и выходными параметрами хотя бы при каких-то условиях происходит так, как ожидается;
- Если программа должна что-то вывести в командную строку, то удаётся создать ситуацию, когда это происходит;
- Если иногда программа должна вернуть, не N, а M переменных, то удаётся создать такую ситуацию;
- Если программа иногда должна выдать ошибку, то удаётся создать такую ситуацию;
- и т.д.

Такие тесты практически не зависят от мировых констант, используемых в задаче; от порядка вычислений; от используемых систем единиц и т.д.

Для рассматриваемого кода functional tests будут основываться на пунктах 1-5 общего функционала задачи.

```
classdef (TestTags = {'functionaltest'}) ...
    QuadraticSolverPositiveFunctionalTest < matlab.unittest.TestCase

    methods (TestClassTeardown)
        function teardown(~)
            clearvars;
        end
    end

    methods
        function programmWorkingTest(testCase)
        % 1. The programm runs to completion

            expectedExceptionIdentifier = '';
            actualException.identifier = '';

            try
                quadraticSolver(1, 0, -1);
            catch actualException
            end

```

```
22          testCase.verifyEqual(expectedExceptionIdentifier,
                actualException.identifier);
23      end
24
25      function programmReturningTest(testCase)
26      % 2. The program runs to completion and returns one non-empty
            variable
27
28          actualResult = quadraticSolver(1, 0, -1);
29          testCase.verifyTrue(~isempty(actualResult));
30      end
31
32      function programmReturningDoubleTest(testCase)
33      % 3. The program runs to completion and returns one non-empty
            double variable;
34
35          actualResult = quadraticSolver(1, 0, -1);
36          testCase.verifyClass(actualResult, 'double');
37      end
38
39      function programmReturninNx2MatrixTest(testCase)
40      % 4. The program runs to completion and returns one non-empty
41      % double variable with a row size equal to the length of the
            first
42      % input argument and 2 columns;
43
44          expectedResult = [1, 2];
45          tmpResult = quadraticSolver(1, 0, -1);
46          actualResult = size(tmpResult);
47          testCase.verifyEqual(actualResult, expectedResult);
48      end
49
50      function programmReturningWarningTest(testCase)
51      % 5. The program can execute to the end, satisfying the above
52      % conditions, and additionally return a warning;
53          expectedResult = {'quadraticSolver:WowWarning'; true};
54
55          actualResultPart1 = quadraticSolver(1, 0, 0);
56          [~, actualResultPart2] = lastwarn;
57          actualResult = {actualResultPart2; ~isempty(
                actualResultPart1)};
```

```
58
59              testCase.verifyEqual(actualResult, expectedResult);
60          end
61
62      end
63  end
```

При составлении unit tests необходимо уже ориентироваться на смысл кода. Можно обращать внимание на следующие аспекты кода:

- Если проводятся вычисления по каким-то формулам, то для эталонных значений они вернут ожидаемый результат;
- При разбиении возможных значений на диапазоны, результат программы совпадает с эталонными как на диапазонах, так и на граничных значениях. (см.техники тест-дизайна);
- Если программа должна правильно реагировать на нефизичные наборы значений, то это происходит;
- и т.д.

Такие тесты позволяют отслеживать “тонкие материи”, в которых, однако, и заключается весь смысл математической модели.

Для рассматриваемого кода unit tests будут основываться на пунктах 6-7 общего функционала задачи.

```
1   classdef QuadraticSolverTest < matlab.unittest.TestCase
2
3       methods (TestClassTeardown)
4           function teardown(~)
5               clearvars;
6           end
7       end
8
9       methods
10           function programmCountingTest(testCase)
11          % 6. The program runs to the end and returns one non-empty
12          % double variable with a row size equal to the length of the
13          % first input argument, and 2 columns containing the result of
14          % applying the formula $\frac{-b \pm \sqrt{b^2 - 4ac} }{2a}$
15          % with a plus or minus sign in column 1 or 2, respectively;
16
17              expectedResult = [1, -1];
```

```
18              actualResult = quadraticSolver(1, 0, -1);
19              testCase.verifyEqual(expectedResult, actualResult);
20          end
21
22          function programmCountingTestV2(testCase)
23          % 7. The program runs to the end and returns one non-empty
24          % double variable with a row size equal to the length of the
25          % first input argument, and 2 columns containing the result of
26          % applying the formula $\frac{-b \pm \sqrt{b^2 - 4ac} }{2a}$
27          % with a plus or minus sign in column 1 or 2, respectively,
28          % and if there are identical solutions on the same line,
29          % a warning appears on the command line.
30              expectedResult = {'quadraticSolver:WowWarning'; [0, 0]};
31
32              actualResultPart1 = quadraticSolver(1, 0, 0);
33              [~, actualResultPart2] = lastwarn;
34              actualResult = {actualResultPart2; actualResultPart1};
35
36              testCase.verifyEqual(actualResult, expectedResult);
37          end
38      end
39  end
```

При этом можно ярко видеть разницу между “математической” и “программной” реализацией, рассматривая, к примеру, 5-ый functional test и 2-ой unit test. В каждом из этих тестов вызывается warning. Однако в functional test не важно, какие значения вернулись из функции вместе с warning, поэтому мы проверяем ответ просто на непустоту. А в unit test уже важно, что warning возникает именно тогда, когда возвращается два одинаковых значения ([0, 0] в данном случае), поэтому происходит проверка с эталоном, а не на непустоту.

Отметим, что выше обсуждались исключительно позитивные functional test. Поэтому проверки на возвращение error не были включены в представленные примеры.