Тестирование

Наумов Д.А., доц. каф. КТ

Основы программной инженерии, 2020

Содержание лекции

Тестирование (software testing)

деятельность, выполняемая для оценки и улучшения качества программного обеспечения. Эта деятельность, в общем случае, базируется на обнаружении *дефектов* и *проблем* в программных системах.

Тестирование программных систем состоит из:

- динамической верификации поведения программ
- на конечном (ограниченном) наборе тестов (set of test cases)
- выбранных соответствующим образом из обычно выполняемых действий прикладной области
- и обеспечивающих проверку соответствия ожидаемому поведению системы.

Сегодня тестирование рассматривается как деятельность, которую необходимо проводить **на протяжении всего процесса разработки и сопровождения** и является важной частью конструирования программных продуктов.

Динамичность (dynamic)

предполагает выполнение тестируемой программы с заданными входными данными.

Конечность (ограниченность, finite)

предполагает компромисс между ограниченными ресурсами и заданными сроками, с одной стороны, и практически неограниченными требованиями по тестированию, с другой.

Выбор (selection) сценариев тестирования

на основе техники анализа рисков, анализа требований и экспертизе в области тестирования и заданной прикладной области.

Ожидаемое поведение (expected behaviour)

может рассматриваться в контексте:

- пользовательских ожиданий (testing for validation)
- спецификации (testing for verification)
- на основе неявных требований или обоснованных ожиданий.

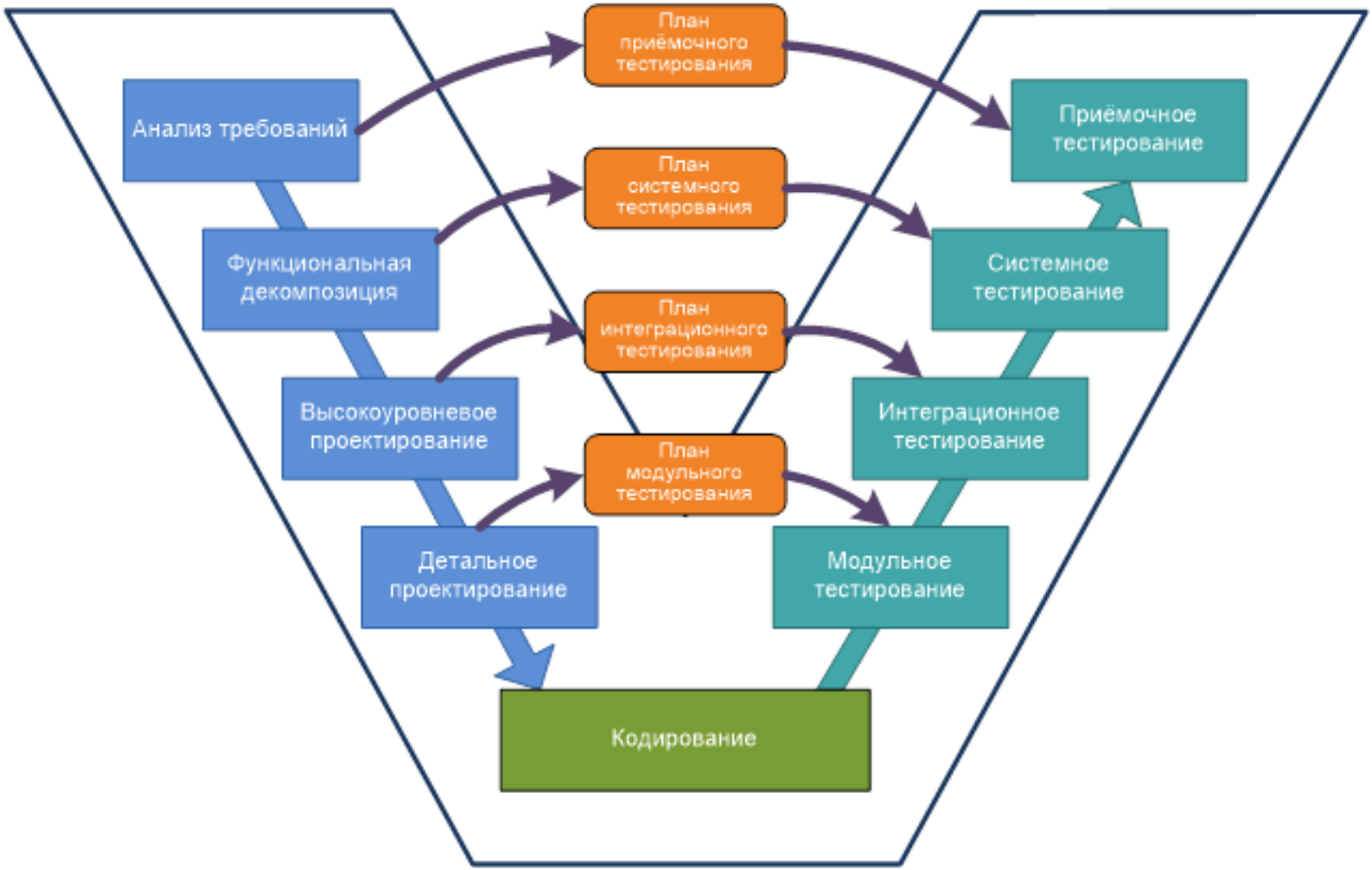
- Важно четко разделять **причину нарушения работы** прикладных систем, обычно описываемую терминами **недостаток** или **дефект**, и наблюдаемый нежелательный эффект, вызываемый этими причинами – **сбой**.
- Термин ошибка, в зависимости от контекста, может описывать и как **причину сбоя**, и сам **сбой**.
- Тестирование позволяет обнаружить дефекты, приводящие к сбоям.

Необходимо понимать, что причина сбоя **не всегда может быть однозначно определена**.

Не существует теоретических критериев, позволяющих гарантированно определить какой именно дефект приводит к наблюдаемому сбою

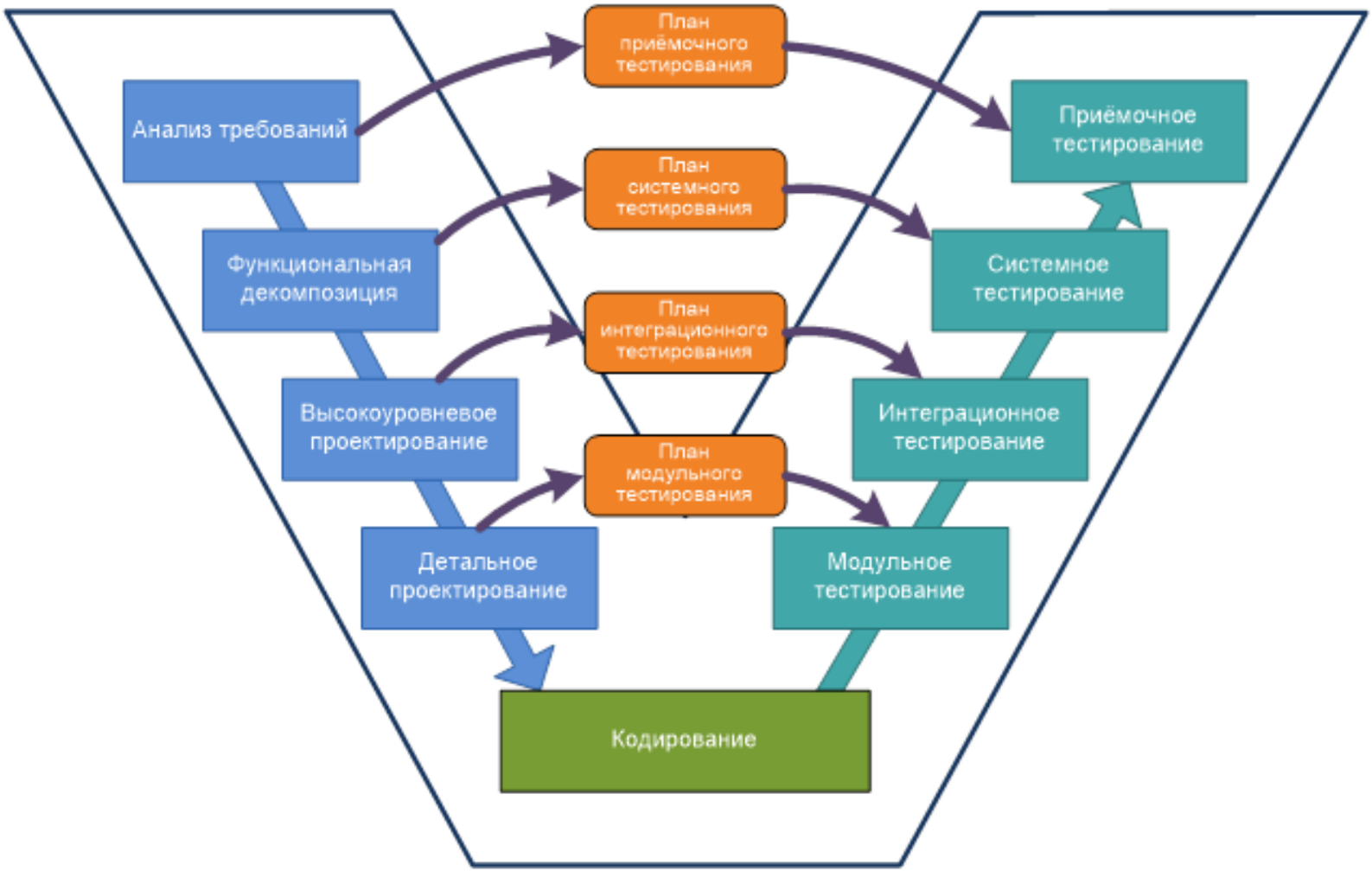
Модульное тестирование (Unit testing)

позволяет проверить функционирование отдельно взятого элемента системы. Что считать элементом – модулем системы определяется контекстом.



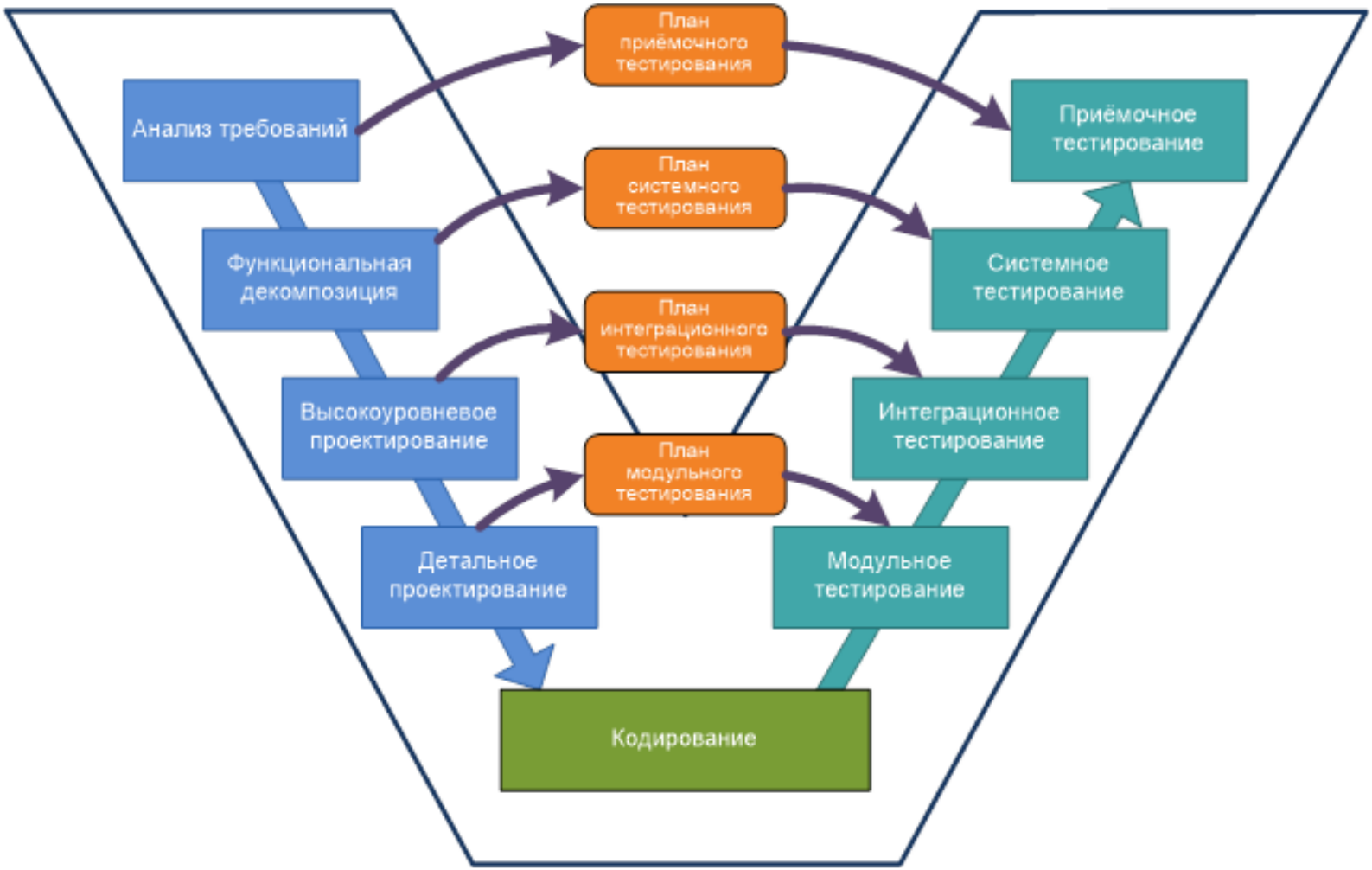
Интеграционное тестирование (Integration testing)

является процессом проверки взаимодействия между программными компонентами/модулями.



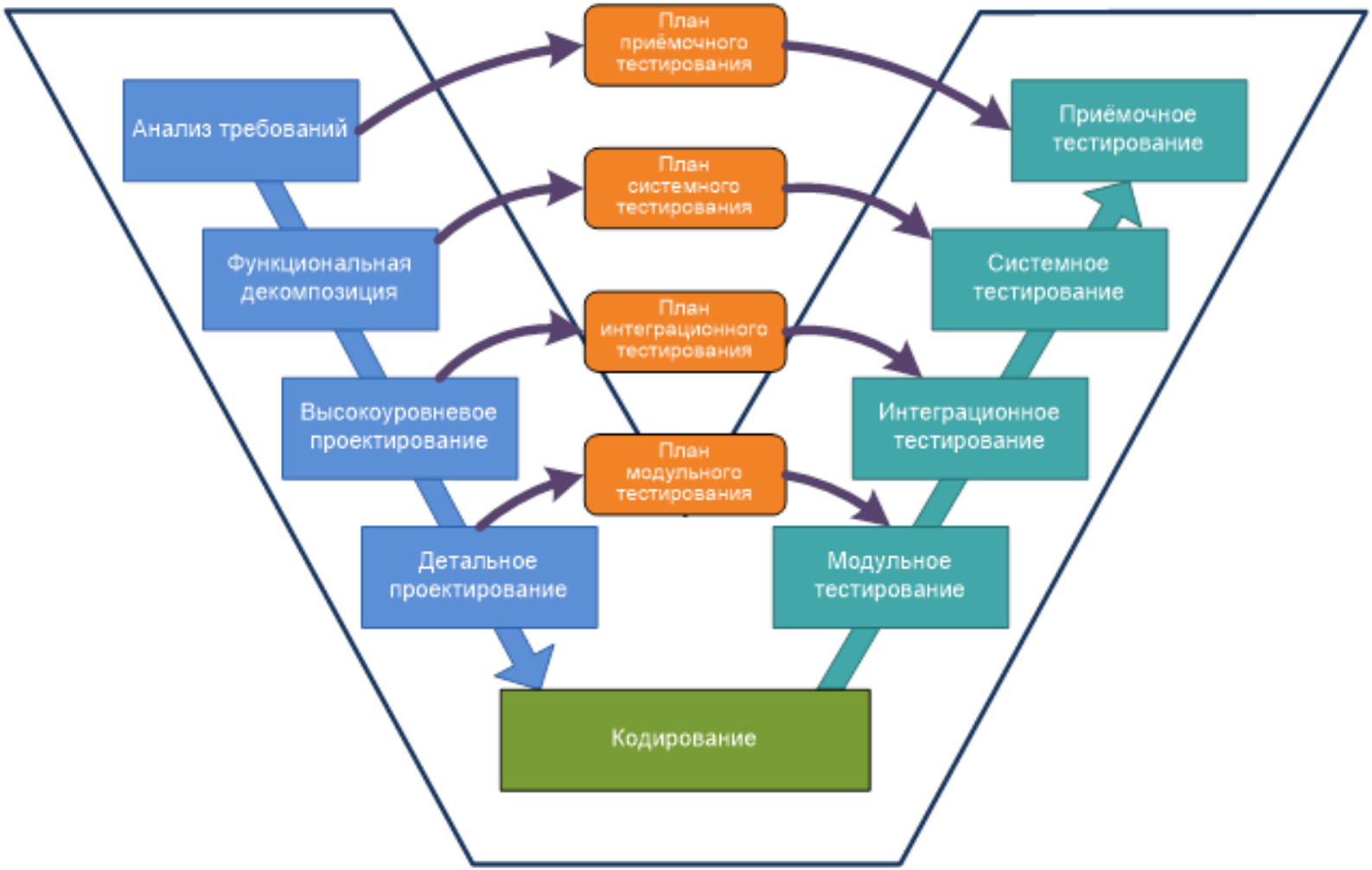
Системное тестирование (System testing)

охватывает целиком всю систему и фокусируется на нефункциональных требованиях -– безопасности, производительности, точности, надежности т.п.



Приёмочное тестирование (Acceptance/qualification testing)

Проверяет поведение системы на предмет удовлетворения требований заказчика.



Цели тестирования

- **Установочное тестирование** (Installation testing) – проверка процедуры инсталляции системы в целевом окружении.
- **Альфа- и бета-тестирование** (Alpha and beta testing) – перед массовым использованием необходимо пройти стадии альфа (внутреннее пробное использование) и бета (пробное использование с привлечением отобранных внешних пользователей) версий.
- **Функциональные тесты**/тесты соответствия (Conformance testing/Functional testing/Correctness testing) – проверка соответствия системы, предъявляемым к ней требованиям, описанным на уровне спецификации поведенческих характеристик.
- **Достижение и оценка надежности** (Reliability achievement and evaluation)
- **Регрессионное тестирование** (Regression testing) – повторное выборочное тестирование системы или компонент для проверки сделанных модификаций.

Цели тестирования

- **Тестирование производительности** (Performance testing) – проверка удовлетворения специфических требований, предъявляемых к параметрам производительности.
- **Нагрузочное тестирование** (Stress testing) – выполнение программной системы с повышением нагрузки.
- **Сравнительное тестирование** (Back-to-back testing) – сравнение двух версий системы.
- **Восстановительные тесты** (Recovery testing) – проверка возможностей рестарта системы в случае непредусмотренной катастрофы (disaster).
- **Конфигурационное тестирование** (Configuration testing) – проверка поведения и работоспособности системы в различных конфигурациях.
- **Тестирование удобства** и простоты использования (Usability testing) – проверка, насколько легко конечный пользователь системы может ее освоить.

Техники тестирования

1. Техники, базирующиеся на интуиции и опыте инженера
   - Специализированное тестирование
   - Исследовательское тестирование
2. Техники, базирующиеся на спецификации
   - Эквивалентное разделение приложения
   - Анализ граничных значений
   - Таблицы принятия решений
   - Тесты на основе конечного автомата
   - Тестирование на основе формальной спецификации
   - Случайное тестирование
3. Техники, ориентированные на код
   - Тесты, базирующиеся на блок-схеме
   - Тесты на основе потоков данных
4. Тестирование, ориентированное на дефекты
   - Предположение ошибок
   - Тестирование мутаций
5. Техники, базирующиеся на условиях использования
   - Операционный профиль
   - Тестирование, базирующееся на надежности инженерного процесса

Процесс и документирование

Работы по тестированию должны быть организованы в единый процесс, на основе учета четырех элементов и связанных с ними факторов:

1. людей (в том числе, в контексте организационной структуры и культуры),
2. инструментов,
3. регламентов,
4. количественных оценок (измерений).

Документация – составная часть формализации процесса тестирования. Стандарт IEEE 829-98 «Standard for Software Test Documentation», предоставляющий описание тестовых документов, их связей между собой и с процессом тестирования:

- план тестирования;
- спецификация процедуры тестирования;
- спецификация тестов;
- лог тестов;
- и др.

Содержание лекции

REPL (read-eval-print loop – цикл «чтение-вычисление-вывод»)

форма организации простой интерактивной среды программирования в рамках средств интерфейса командной строки.

Пользователь может вводить выражения, которые среда тут же будет вычислять, а результат вычисления отображать пользователю.

- функция *read* читает одно выражение и преобразует его в соответствующую структуру данных в памяти;
- функция *eval* принимает одну такую структуру данных и вычисляет соответствующее ей выражение;
- функция *print* принимает результат вычисления выражения и печатает его пользователю.

Doctest

Модуль Doctest

ищет фрагменты текста, которые выглядят как интерактивные *python*-сессии. Далее выполняет сеансы и проверяет, совпадает ли с тем что указано в docstring.

- использование *doctest* выглядит, как будто пишем код в *REPL*.
- можно применять для тестирования как функций, так и классов;

Напишем тесты для функции *moo*:

```
def moo(oos=2, end=""):
    '''Издать мычание длиной oos с end в конце'''
    return "M"+"o"*oos+end
```

Ручное тестирование

```
> python -i Moo.py

>>> moo()
'Moo'
>>> moo(4)
'Moooo'
>>> moo(0)
'M'
>>> moo(end='!')
'Moo!'
>>> moo(0,'?')
'M?'
```

```
def moo(oos=2, end=""):
''' Издать мычание длиной oos с end в конце
    Оба параметра необязательны:
>>> moo()
'Moo'
    Первый задаёт количество букв 'o' в слове 'Moo'
>>> moo(4)
'Mooooo'
    Букв 'o' может и не быть
>>> moo(0)
'M'
    Второй задаёт символ после всех 'o' (по умолчанию - ничего,
>>> moo(end='!')
'Moo!'
>>> moo(0,'?')
'M?'
    '''
    return "M"+"o"*oos+end
```

Тестирование

```
> python -m doctest Moo.py

**********************************************************************
File "Moo.py", line 13, in Moo.moo
Failed example:
    moo(4)
Expected:
    'Mooooo'
Got:
    'Moooo'
**********************************************************************
1 items had failures:
   1 of   5 in Moo.moo
***Test Failed*** 1 failures.
```

Отчет с успешными тестами

```
> python3 -m doctest -v Moo.py

Trying:
    moo()
Expecting:
    'Moo'
ok
Trying:
    moo(4)
Expecting:
    'Mooooo'
**********************************************************************
File "Moo.py", line 13, in Moo.moo
Failed example:
    moo(4)
Expected:
    'Mooooo'
Got:
    'Moooo'
```

Отчет с успешными тестами

```
Trying:
    moo(0)
Expecting:
    'M'
ok
Trying:
    moo(end='!')
Expecting:
    'Moo!'
ok
Trying:
    moo(0,'?')
Expecting:
    'M?'
ok
```

Отчет с успешными тестами

```
1 items had no tests:
    Moo
**********************************************************************
1 items had failures:
   1 of   5 in Moo.moo
5 tests in 2 items.
4 passed and 1 failed.
***Test Failed*** 1 failures.
```

Тестирование исключений

```
> python -i Moo

>>> moo("QQ")

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "Moo.py", line 33, in moo
    return "M"+"o"*moos+end
TypeError: can't multiply sequence by non-int of type 'str'

>>>
```

Тестирование исключений

Важны только три строчки, остальные можно не включать в *docstring*:

```
def moo(oos=2, end=""):
    ''' Издать мычание длиной oos с end в конце
                ...
                Здесь должно быть исключение:

>>> moo("QQ")
Traceback (most recent call last):
TypeError: can't multiply sequence by non-int of type 'str'
        '''
```

Перенос тестов во внешний файл

Файл exttest.rst (.rst – для Sphinx)

```
External test
=============

Using Moo
---------

Start from importing 'Moo' module:

        >>> import Moo

Then call 'moo':
        >>> Moo.moo(5)
        'Mooooo'
```

Файл exttest.py

```
import doctest
doctest.testfile("exttest.rst")
```

Файл для запуска тестов

```
> python exttest.py -v

Trying:
    import Moo
Expecting nothing
ok
Trying:
    Moo.moo(5)
Expecting:
    'Mooooo'
ok
1 items passed all tests:
   2 tests in exttest.rst
2 tests in 1 items.
2 passed and 0 failed.
Test passed.
```

Тестирование классов

```
class Test(object):
    """
    >>> a=Test(5)
    >>> a.multiply_by_2()
    10
    """
    def __init__(self, number):
        self.number = number

    def multiply_by_2(self):
        return self.number*2

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Тестирование классов

```
> python example.py -v

Trying:
    a=Test(5)
Expecting nothing
ok
Trying:
    a.multiply_by_2()
Expecting:
    10
ok
3 items had no tests:
    __main__
    __main__.Test.__init__
    __main__.Test.multiply_by_2
1 items passed all tests:
   2 tests in __main__.Test
2 tests in 4 items.
2 passed and 0 failed.
Test passed.
```

Содержание лекции

Модульное тестирование при помощи unittest

Возможности *unittest*:

- обнаружение и автоматическое исполнение тестов;
- настройка теста и его завершение;
- группирование тестов;
- статистика тестирования.

Чтобы создать тестовый случай, нужно:

1. создать класс, отнаследованный от *unittest.TestCase*;
2. внутри этого класса добавить несколько методов, начинающихся со слова *test*.
3. каждый из этих методов должен тестировать какой-то из аспектов кода.

Тестирование свойств строк в Python

```
import unittest

class StringTestCase(unittest.TestCase):
    def test_sum(self):
        self.assertEqual("" + "", "")
        self.assertEqual("foo" + "bar", "foobar")

    def test_lower(self):
        self.assertEqual("FOO".lower(), "foo")
        self.assertTrue("foo".islower())
        self.assertFalse("Bar".islower())

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

Unittest

Самые распространенные проверки:

- **self.assertEqual** – непосредственно проверяет, чтобы первый аргумент равнялся второму. Если это будет не так, то тест будет провален, и появится сообщение о том, что и где пошло не так.
- **self.assertTrue** – ожидает, что аргумент будет эквивалентен правде (True), а self.assertFalse – проверяет на ложь (False).

Запуск делается либо непосредственно из самой программы:

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

А можно из консоли:

```
> python -m unittest my_unittest.py
```

Модуль *unittest* сам найдет все тестовые случаи и выполнит в них все тестовые функции.

Unittest: Fixtures

Тестовые фикстуры (test fixtures) — особые условия, которые создаются для выполнения тестов:

- подготовка тестовых данных;
- создание подключений к БД, сервисам и т.п.
- создание заглушек (mock) для имитации компонентов программы;
- другие действия по поддержке рабочего окружения для проведения теста.

Фикстуры могут быть созданы:

- на уровне модуля с тестами;
- на уровне отдельного класса (от unittest.TestCase);
- для каждого метода в классе теста.

Unittest: Fixtures

Фикстуры unittest:

- Метод **setUp()** вызывается перед каждым вызовом метода **test*** в классе тестового случая.
- Классовый метод **setUpClass()** вызывается один раз перед запуском тестов в классе тестового случая.
- Функция **setUpModule()** вызывается перед выполнением тестовых случаев в этом модуле.

У них есть пары, предназначенные для освобождения ресурсов (закрытия соединений, удаления временных файлов и т.п.):

- **tearDown()** – после каждого метода-теста в классе.
- **tearDownClass()** – после всех тестов в классе.
- **tearDownModule()** – после всех классов в модуле.

```
import unittest
class StringTestCase(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        print(' - set up class')
    def setUp(self):
        print(' - - set up method')
        self.foo = "foo"
        self.bar = "bar"
    def test_sum(self):
        self.assertEqual(self.foo + self.bar, "foobar")
    def test_lower(self):
        self.assertTrue(self.foo.islower())
    def tearDown(self):
        print(' - - tear down method')
    @classmethod
    def tearDownClass(cls):
        print(' - tear down class')
def setUpModule():
    print('set up module')
def tearDownModule():
    print('tear down module')
```

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

Схема вызовов:

```
set up module
 - set up class
 - - set up method
 - - tear down method
 - - set up method
 - - tear down method
 - tear down class
tear down module
```

Пример

```
class MyTestCase(unittest.TestCase):
    @unittest.skip("всегда пропустить")
    def test_nothing(self):
        self.fail("не случится")
    @unittest.skipIf(mylib.__version__ < (1, 3),
        "эта версия библиотеки не поддерживается")
    def test_format(self):
        # этот тест работает только для определенных версий
        pass
    @unittest.skipUnless(sys.platform.startswith("win"),
        "надо Windows")
    def test_windows_support(self):
        # тест работает только на Windows
        pass
    def test_maybe_skipped(self):
        if not external_resource_available():
            self.skipTest("ресурс недоступен")
        # код дальше будет тестировать, если ресурс доступен
        pass
```

Пример пропуска класса

```
@unittest.skip("как пропустить класс")
class MySkippedTestCase(unittest.TestCase):
    def test_not_run(self):
        pass
```

Декоратор @unittest.expectedFailure говорит системе тестирования, что следующий метод должен провалиться (один из self.assert должен не сработать).

Таким образом, разработчик говорит, что он осведомлен, что данный тест пока проваливается, и в будущем к этому примут меры.

```
class ExpectedFailureTestCase(unittest.TestCase):
    @unittest.expectedFailure
    def test_fail(self):
        self.assertEqual(1, 0, "сломано")
```

Метод **self.assertRaises(SomeException)** проверяет, возбуждает ли код нужное исключение.

```
import unittest

def my_div(a, b):
    return a // b

class MyDivTestCase(unittest.TestCase):
    def test_1(self):
        self.assertEqual(my_div(10, 2), 5)
        # при делении на 0 ждем исключение:
        with self.assertRaises(ZeroDivisionError):
            my_div(7, 0)
        # или так: исключение, функция, аргументы
        self.assertRaises(ZeroDivisionError, my_div, 5, 0)

unittest.main()
```

Если из исключения нужно извлечь данные (к примеру, код ошибки):

```
with self.assertRaises(SomeException) as cm:
    do_something()
self.assertEqual(cm.exception.error_code, 3)
```

Метод	Проверяет, что	Добавлен
`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`	3.1
`assertIsNot(a, b)`	`a is not b`	3.1
`assertIsNone(x)`	`x is None`	3.1
`assertIsNotNone(x)`	`x is not None`	3.1
`assertIn(a, b)`	`a in b`	3.1
`assertNotIn(a, b)`	`a not in b`	3.1
`assertIsInstance(a, b)`	`isinstance(a, b)`	3.2
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`	3.2

Содержание лекции

PyTest

Библиотека *PyTest* предоставляет более лаконичный и удобный инструментарий для написания тестов.

Установка:

```
pip install pytest
```

Преимущества *PyTest*:

- более краткий и красивый код;
- используется только один стандартный *assert*;
- возможность формирования подробного отчета;
- разнообразие фикстур;
- имеет плагин и интеграции с другими системами.

В окружении, где установлен *PyTest*, появится команда *py.test*. Из терминала пишем:

```
py.test my_test_cases.py
```

```python
import pytest

def setup_module(module):
    #init_something()
    pass

def teardown_module(module):
    #teardown_something()
    pass

def test_upper():
    assert 'foo'.upper() == 'FOO'

def test_isupper():
    assert 'FOO'.isupper()

def test_failed_upper():
    assert 'foo'.upper() == 'FOo'
```

PROBLEMS 2 OUTPUT TERMINAL DEBUG CONSOLE 1: Python

```
D:\work\git\ds\examples\python\testing\pytest>py.test fixtures.py
============================= test session starts =============================
platform win32 -- Python 3.8.3, pytest-6.1.2, py-1.9.0, pluggy-0.13.1
rootdir: D:\work\git\ds\examples\python\testing\pytest
collected 3 items

fixtures.py ..F

================================== FAILURES ===================================
_____________________________ test_failed_upper ______________________________

    def test_failed_upper():
>       assert 'foo'.upper() == 'FOo'
E       AssertionError: assert 'FOO' == 'FOo'
E         - FOo
E         + FOO

fixtures.py:18: AssertionError
=========================== short test summary info ===========================
FAILED fixtures.py::test_failed_upper - AssertionError: assert 'FOO' == 'FOo'
============================== 1 failed in 0.07s ==============================

D:\work\git\ds\examples\python\testing\pytest>
```

См. также: https://habr.com/ru/post/269759/