Тестирование исходного кода

Если отладка — процесс удаления ошибок, то под программированием можно понимать процесс их внесения.

Эдсгер Вибе Дейкстра

Подготовил: Радченко В.И., ВМИ-356

Понятие тестирования

Тестирование программного обеспечения — проверка соответствия между реальным и ожидаемым поведением программы, осуществляемая на конечном наборе тестов, выбранных определённым образом.

Тестирование является одним из этапов разработки ПО:

1. Определение требований
2. Проектирование архитектуры
3. Конструирование
4. ***Тестирование***
5. Внедрение

Понятие тестирования: чем тестирование не является

Тестирование — это не поиск ошибок в программе!

При поиске ошибок:

- цель — найти наибольшее число багов;
- тестируются — самые нестабильные части; программы;
- тесты — самые нестандартные.

При тестировании:

- цель — пропустить как можно меньше важных для пользователя багов;
- тестируются — самые приоритетные для пользователя части программы;
- тесты — стандартные, при отсутствии необходимости в краевых тестах.

Понятие тестирования: чем тестирование не является

Тестирование не может доказать корректность кода!

Тестирование программы весьма эффективно может показать наличие ошибок, но оно безнадёжно неадекватно для демонстрации их отсутствия.

Эдсгер Вибе Дейкстра

Цель тестирования состоит не в том, чтобы показать удовлетворительную работу программы.

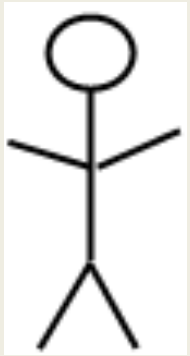
Цель тестирования — чётко определить, в чём работа программы неудовлетворительна.

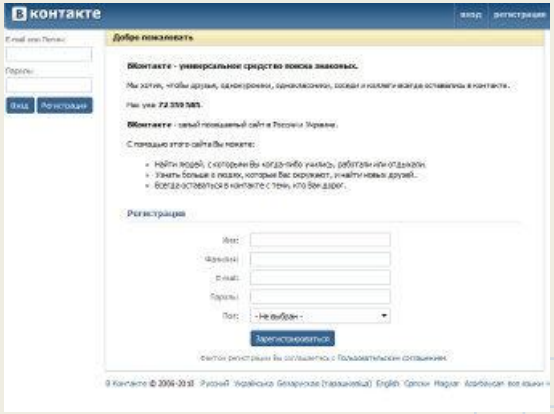
Виды тестирования

Процесс тестирования в рамках жизненного цикла ПО

Чёрный, серый и белый ящики

Конечные пользователи разрабатываемого ПО не имеют представления о его исходном коде, таблицах баз данных. Для них эта система является **чёрным ящиком**.

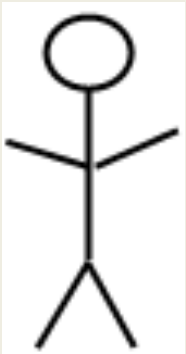


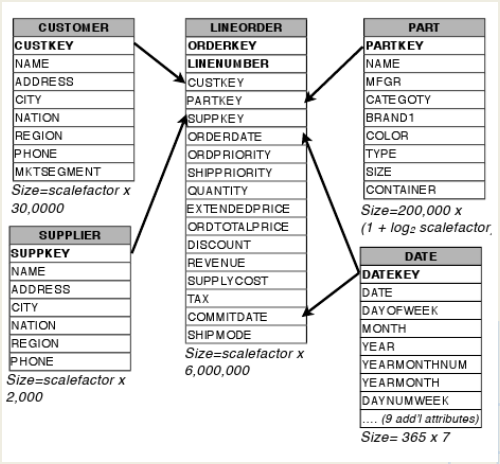


При тестировании по стратегии чёрного ящика руководствуются **спецификацией** системы, и оценивается её **функциональность**.

Чёрный, серый и белый ящики

Тестировщики проверяют не только *что* делает система, но и *как* она это делает. Для них разрабатываемая система является **серым ящиком**.

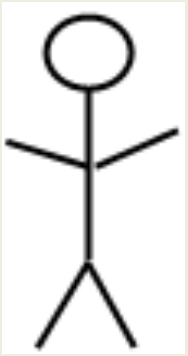


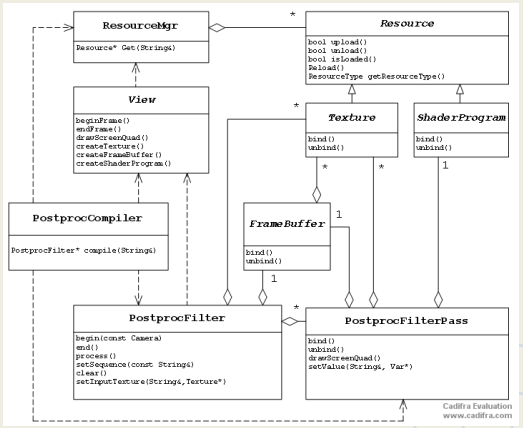


При тестировании по стратегии серого ящика руководствуются не только **спецификацией**, но и **ключевыми элементами проектирования**. Тестируется как **функционал**, так и **ожидаемое поведение программы.**

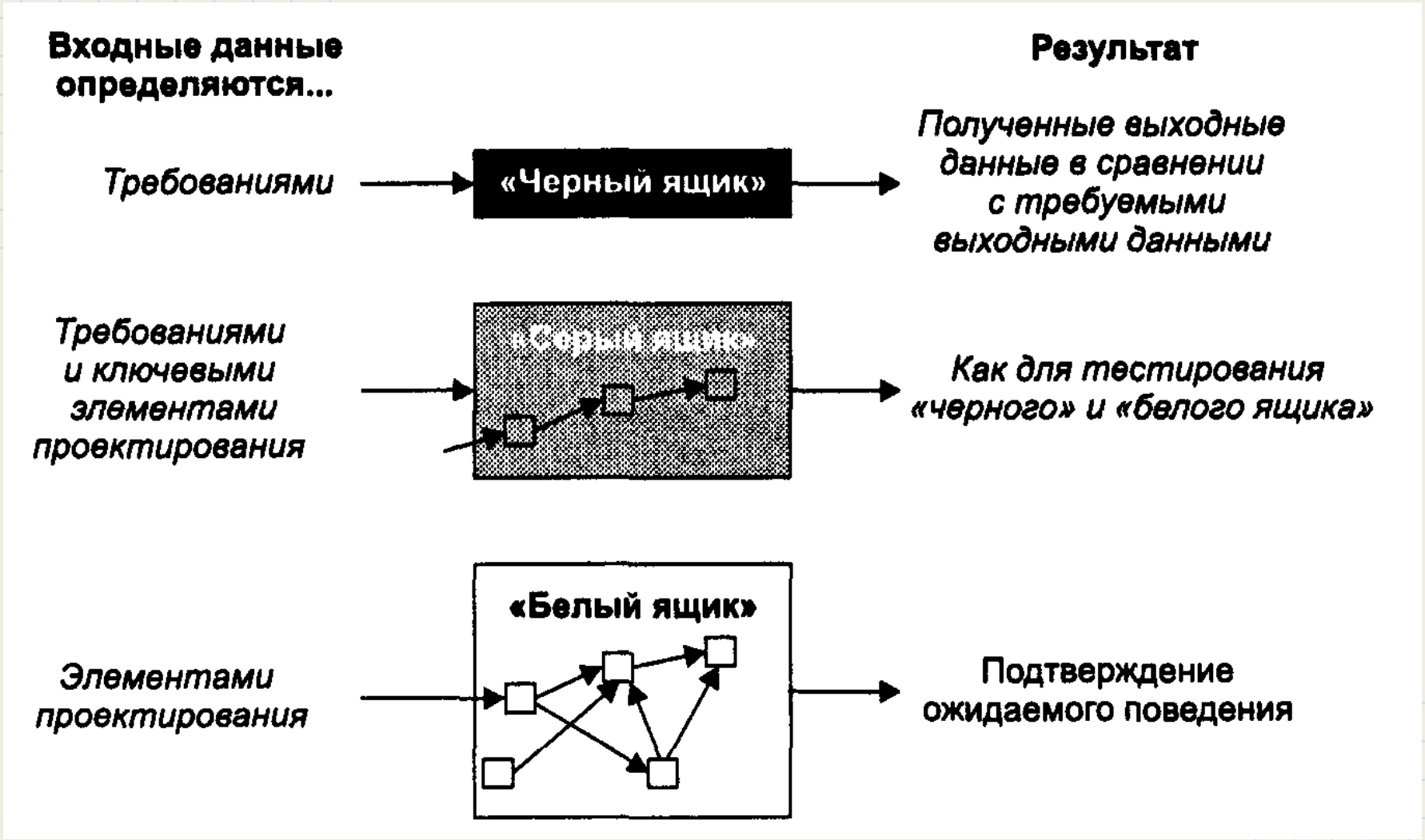
Чёрный, серый и белый ящики

Разработчики знают код. Они определяют уместные или неуместные паттерны проектирования, структуры классов. Для них разрабатываемая ими система — **белый ящик**.





При тестировании по стратегии белого руководствуются **элементами проектирования** системы. Тестируется **ожидаемое поведение программы.**



Чёрный, серый и белый ящики

Тестирование по стратегии чёрного ящика

- Функционал системы. Делает ли система то, что она должна делать согласно спецификации?
- Контроль вводимых данных. Как реагирует система на некорректный ввод данных?
- Выводимые данные. Правильно ли система реагирует на рутинные действия пользователя?

Тестирование на всевозможных наборах данных — *исчерпывающее тестирование* — как правило невозможно. Вместо этого следует выбрать несколько максимально покрывающих проверяемый функционал тестов.

Тестирование по стратегии чёрного ящика

Метод эквивалентного разбиения:

- разбиение тестов на такие *классы эквивалентности*, что если один тест из него не выполняется, то другие также не будут выполнены, и наоборот;
- каждый тест должен входить в максимальное число классов эквивалентности.

Метод анализа граничных условий:

- выбор любого элемента в классе эквивалентности в качестве представительного осуществляется таким образом, чтобы проверить границы этого класса.

Тестирование по стратегии белого ящика

- Тестирование всех возможных веток в коде. В каком случае условие этого условного оператора не будет выполняться? Может ли этот цикл стать вечным?
- Надлежащая обработка исключтельных ситуаций: намеренно "ломаем" методы, чтобы убедиться в том, что в этих случаях будут вызваны `exception`-ы.
- "Краевые" тесты: как поведёт себя метод, если в `*args` ничего не передавать? Что будет, если он вдруг столкнётся с нехваткой ресурсов?

С тестированием по стратегии белого ящика тесно связано вычисление процента тестового покрытия по критерию исходного кода программы.

Тестирование по стратегии белого ящика

Метод покрытия операторов:

- выполнение каждого оператора рассматриваемой части кода по крайней мере один раз на всём наборе тестов.

Метод покрытия решений (ветвей):

- подбор таких тестов, что каждое условие в коде принимает как значение `true`, так и `false`.

Метод покрытия путей:

- подбор таких тестов, что каждый путь в программе выполняется хотя бы раз.

Тестирование по стратегии белого ящика

```
void div_by_zero_test(int x)
{
    int tmp = 0;
    /*критерий покрытия операторов может быть
удовлетворён,
    а последующая ошибка — оказаться невыявленной*/
    if (x > 0) tmp = 1;
    float b = 1.0 / tmp;
    return;
}
void div_by_zero_test(int x)
{
    int tmp = 7;
    /*а здесь недостаточно точным оказывается метод
    покрытия ветвей*/
    while (tmp > x) tmp--;
    float b = 1.0 / tmp;
    return;
}
```

Тестирование по стратегии серого ящика

- Контроль ведения аудита по вводимой информации. Ведутся ли логи во время использования системы?
- Проверка информации, создаваемой самой системой: таймштампов с учётом временных зон, хэш-сумм, внешних ключей баз данных...
- Удаление временных файлов и очистка памяти. Не возникает ли утечка памяти во время исполнения программы?

Эта стратегия объединяет в себе цели белого и чёрных ящиков. Программно её можно реализовать, добавив `print`-ы или `assert`-ы в стабильных частях кода, или же добавив их при тестировании белого ящика.

Тестовое покрытие

Тестовое покрытие — метрика оценки качества тестирования, представляющая собой плотность покрытия тестами...

- ...*требований*, то есть проверка соответсвтия набора проводимых тестов требованиям к программе,
- ...либо *исполняемого кода*, то есть полнота проверки тестами уже разработанной части программы.

Тестовое покрытие: покрытие требований

Для измерения покрытия требований необходим анализ спецификации и разбивка требований на пункты. В соответствие каждому пункту следует поставить набор тестов.

Такие связи часто объединяют в единую матрицу, называемую *матрицей трассировки требований*, что способствует наглядности.

$$T = Lc / Lt,$$

где T — покрытие требований, Lc — количество проверенных тестами требований, Lt — общее количество требований.

Тестовое покрытие: покрытие кода

$$T = Ltc/Lcode,$$

где T — покрытие требований, Ltc — количество покрытых тестами строк кода, Lcode — общее количество строк кода.

Вычисление этой характеристики возможно автоматизировать. Существуют инструменты (например, Clover), позволяющие проанализировать, в какие строки кода были вхождения во время тестов.

Проведение такого анализа легко реализуется в рамках тестирования по стратегии белого ящика при модульном или интеграционном подходе.

Автоматизированное тестирование

Автоматизация тестирования заключается в использовании готовых программных средств для выполнения тестов и проверки результатов выполнения.

- Python — unittest, doctest
- Java — JUnit
- C++ — Boost::Test, Google Test
- C — UniTESK
- web-приложения — Selenium
- ...

Автоматизированное тестирование: Python — unittest

```
class ArbitraryPrecisionCalculator(object):
    """
    Синглтон, осуществляющий вычисления для случая
    длинной арифметки.
    """
    __metaclass__ = SingletonMeta

    def addition(self, *args):
    """
    Длинное сложение.
    На вход подаются длинные числа в виде списка.
    Метод возвращает их сумму в виде списка либо 0, если
    ни одного числа не было передано.
    """
        ...

    def division(self, dividend, divisor):
    """
    Длинное деление.
    На вход подаются два длинных числа: делимое и делитель.
    Метод возвращает результат их деления.
    """
        ...
```

Автоматизированное тестирование: Python — unittest

```
from django.test import TestCase
from somemodulename import ArbitraryPrecisionCalculator

class DivisionTest(TestCase):
    """
    Тестирование деления калькулятора длинной арифметики.
    """
    def test_simple_div(self):
        d1 = [6,]
        d2 = [3,]
        res = [2,]
        self.assertEqual(ArbitraryPrecisionCalculator(d1,d2),res)

    <...>
    def test_division_by_zero(self):
        d1 = [4,2,]
        d2 = [0,]
        self.assertRaises(DivisionByZeroException,
                          ArbitraryPrecisionCalculator(d1,d2))
```

Автоматизированное тестирование: Python — unittest

```
<...>
class SingletonTest(TestCase):
    """
    Тестирование единственности объекта ArbitraryPrecisionCalculator
    """
    def test_only_obj(self):
        APC1 = ArbitraryPrecisionCalculator()
        APC2 = ArbitraryPrecisionCalculator()
        self.assertTrue(APC1 is APC2)
<...>
```

Разработка через тестирование

Разработка через тестирование (англ. TDD, test-driven development) — техника разработки ПО, основанная на повторении коротких циклов разработки:

- написание теста, покрывающего желаемое изменение кода;
- написание кода, позволяющего пройти этот и ранее написанные тесты;
- рефакторинг кода.

Разработка через тестирование

Рассмотрим технику TDD на примере написания функции для разложения числа на простые множители:

- 2 — [2]
- 3 — [3]
- 4 — [2,2]
- 16 — [2,2,2,2]
- 21 — [3,7]
- ...

Разработка через тестирование

```
from django.test import TestCase
from prime import getFactors

class SimpleTest(TestCase):
   def test_one(self):
      self.assertItemsEqual([],getFactors(1))
```

Разработка через тестирование

```
from django.test import TestCase
from prime import getFactors

class SimpleTest(TestCase):
   def test_one(self):
      self.assertItemsEqual([],getFactors(1))
```

```
def getFactors(num):
      return []
```

Разработка через тестирование

```
from django.test import TestCase
from prime import getFactors

class SimpleTest(TestCase):
   def test_one(self):
      self.assertItemsEqual([],getFactors(1))
```

```
def getFactors(num):
      return []
```

Разработка через тестирование

```
from django.test import TestCase
from prime import getFactors

class SimpleTest(TestCase):
   def test_one(self):
      self.assertItemsEqual([],getFactors(1))
   def test_two(self):
      self.assertItemsEqual([2],getFactors(2))
```

```
def getFactors(num):
      return []
```

Разработка через тестирование

```
from django.test import TestCase
from prime import getFactors

class SimpleTest(TestCase):
   def test_one(self):
      self.assertItemsEqual([],getFactors(1))
   def test_two(self):
      self.assertItemsEqual([2],getFactors(2))
```

```
def getFactors(num):
      return []
```

Разработка через тестирование

```
from django.test import TestCase
from prime import getFactors

class SimpleTest(TestCase):
   def test_one(self):
      self.assertItemsEqual([],getFactors(1))
   def test_two(self):
      self.assertItemsEqual([2],getFactors(2))
```

```
def getFactors(num):
      res = []
      if num > 1:
            res.append(num)
      return res
```

Разработка через тестирование

```
from django.test import TestCase
from prime import getFactors

class SimpleTest(TestCase):
   def test_one(self):
      self.assertItemsEqual([],getFactors(1))
   def test_two(self):
      self.assertItemsEqual([2],getFactors(2))
```

```
def getFactors(num):
      res = []
      if num > 1:
            res.append(num)
      return res
```

Разработка через тестирование

```
from django.test import TestCase
from prime import getFactors

class SimpleTest(TestCase):
   def test_one(self):
      self.assertItemsEqual([],getFactors(1))
   def test_two(self):
      self.assertItemsEqual([2],getFactors(2))
   def test_four(self):
      self.assertItemsEqual([2,2],getFactors(4))
```

```
def getFactors(num):
      res = []
      if num > 1:
            res.append(num)
      return res
```

Разработка через тестирование

```
from django.test import TestCase
from prime import getFactors

class SimpleTest(TestCase):
   def test_one(self):
      self.assertItemsEqual([],getFactors(1))
   def test_two(self):
      self.assertItemsEqual([2],getFactors(2))
   def test_four(self):
      self.assertItemsEqual([2,2],getFactors(4))
```

```
def getFactors(num):
      res = []
      if num > 1:
            res.append(num)
      return res
```

Разработка через тестирование

```
from django.test import TestCase
from prime import getFactors

class SimpleTest(TestCase):
   def test_one(self):
      self.assertItemsEqual([],getFactors(1))
   def test_two(self):
      self.assertItemsEqual([2],getFactors(2))
   def test_four(self):
      self.assertItemsEqual([2,2],getFactors(4))
```

```
def getFactors(num):
      res = []
      if num > 2:
            res.append(2)
            num = 2
      if num > 1:
            res.append(num)
      return res
```

Разработка через тестирование

```
from django.test import TestCase
from prime import getFactors

class SimpleTest(TestCase):
   def test_one(self):
      self.assertItemsEqual([],getFactors(1))
   def test_two(self):
      self.assertItemsEqual([2],getFactors(2))
   def test_four(self):
      self.assertItemsEqual([2,2],getFactors(4))
```

```
def getFactors(num):
      res = []
      if num > 2:
            num = num / 2
            res.append(num)
      if num > 1:
            res.append(num)
      return res
```

Разработка через тестирование

```
from django.test import TestCase
from prime import getFactors

class SimpleTest(TestCase):
   def test_one(self):
      self.assertItemsEqual([],getFactors(1))
   def test_two(self):
      self.assertItemsEqual([2],getFactors(2))
   def test_four(self):
      self.assertItemsEqual([2,2],getFactors(4))
```

```
def getFactors(num):
      res = []
      if num > 2:
            num = num / 2
            res.append(num)
      if num > 1:
            res.append(num)
      return res
```

Разработка через тестирование

```
from django.test import TestCase
from prime import getFactors

class SimpleTest(TestCase):
   def test_one(self):
      self.assertItemsEqual([],getFactors(1))
   def test_two(self):
      self.assertItemsEqual([2],getFactors(2))
   def test_four(self):
      self.assertItemsEqual([2,2],getFactors(4))
   def test_five(self):
      self.assertItemsEqual([5],getFactors(5))
```

```
def getFactors(num):
      res = []
      if num > 2:
            num = num / 2
            res.append(num)
      if num > 1:
            res.append(num)
      return res
```

Разработка через тестирование

```
from django.test import TestCase
from prime import getFactors

class SimpleTest(TestCase):
   def test_one(self):
      self.assertItemsEqual([],getFactors(1))
   def test_two(self):
      self.assertItemsEqual([2],getFactors(2))
   def test_four(self):
      self.assertItemsEqual([2,2],getFactors(4))
   def test_five(self):
      self.assertItemsEqual([5],getFactors(5))
```

```
def getFactors(num):
      res = []
      if num > 2:
            num = num / 2
            res.append(num)
      if num > 1:
            res.append(num)
      return res
```

Разработка через тестирование

```
from django.test import TestCase
from prime import getFactors

class SimpleTest(TestCase):
   def test_one(self):
      self.assertItemsEqual([],getFactors(1))
   def test_two(self):
      self.assertItemsEqual([2],getFactors(2))
   def test_four(self):
      self.assertItemsEqual([2,2],getFactors(4))
   def test_five(self):
      self.assertItemsEqual([5],getFactors(5))
```

```
def getFactors(num):
      res = []
      if num > 3:
            if num % 2 == 0:
                  num = num / 2
                  res.append(num)
      if num > 1:
            res.append(num)
      return res
```

Разработка через тестирование

```
from django.test import TestCase
from prime import getFactors

class SimpleTest(TestCase):
   def test_one(self):
      self.assertItemsEqual([],getFactors(1))
   def test_two(self):
      self.assertItemsEqual([2],getFactors(2))
   def test_four(self):
      self.assertItemsEqual([2,2],getFactors(4))
   def test_five(self):
      self.assertItemsEqual([5],getFactors(5))
```

```
def getFactors(num):
      res = []
      if num > 3:
            if num % 2 == 0:
                  num = num / 2
                  res.append(num)
      if num > 1:
            res.append(num)
      return res
```

Разработка через тестирование

```
from django.test import TestCase
from prime import getFactors

class SimpleTest(TestCase):
   def test_one(self):
      self.assertItemsEqual([],getFactors(1))
   def test_two(self):
      self.assertItemsEqual([2],getFactors(2))
   def test_four(self):
      self.assertItemsEqual([2,2],getFactors(4))
   def test_five(self):
      self.assertItemsEqual([5],getFactors(5))
   def test_six(self):
      self.assertItemsEqual([2,3],getFactors(6))
```

```
def getFactors(num):
      res = []
      if num > 3:
            if num % 2 == 0:
                  num = num / 2
                  res.append(num)
      if num > 1:
            res.append(num)
      return res
```

Разработка через тестирование

```
from django.test import TestCase
from prime import getFactors

class SimpleTest(TestCase):
   def test_one(self):
      self.assertItemsEqual([],getFactors(1))
   def test_two(self):
      self.assertItemsEqual([2],getFactors(2))
   def test_four(self):
      self.assertItemsEqual([2,2],getFactors(4))
   def test_five(self):
      self.assertItemsEqual([5],getFactors(5))
   def test_six(self):
      self.assertItemsEqual([2,3],getFactors(6))
```

```
def getFactors(num):
      res = []
      for i in xrange(2,num):
            if num % i == 0:
                  num = num / i
                  res.append(i)
      if num > 1:
            res.append(num)
      return res
```

Разработка через тестирование

```
from django.test import TestCase
from prime import getFactors

class SimpleTest(TestCase):
   def test_one(self):
      self.assertItemsEqual([],getFactors(1))
   def test_two(self):
      self.assertItemsEqual([2],getFactors(2))
   def test_four(self):
      self.assertItemsEqual([2,2],getFactors(4))
   def test_five(self):
      self.assertItemsEqual([5],getFactors(5))
   def test_six(self):
      self.assertItemsEqual([2,3],getFactors(6))
```

```
def getFactors(num):
      res = []
      for i in xrange(2,num):
            if num % i == 0:
                  num = num / i
                  res.append(i)
      if num > 1:
            res.append(num)
      return res
```

Разработка через тестирование

```
from django.test import TestCase
from prime import getFactors

class SimpleTest(TestCase):
   def test_one(self):
      self.assertItemsEqual([],getFactors(1))
   def test_two(self):
      self.assertItemsEqual([2],getFactors(2))
   def test_four(self):
      self.assertItemsEqual([2,2],getFactors(4))
   def test_five(self):
      self.assertItemsEqual([5],getFactors(5))
   def test_six(self):
      self.assertItemsEqual([2,3],getFactors(6))
   def test_seven(self):
      self.assertItemsEqual([7],getFactors(7))
```

```
def getFactors(num):
      res = []
      for i in xrange(2,num):
            if num % i == 0:
                  num = num / i
                  res.append(i)
      if num > 1:
            res.append(num)
      return res
```

Разработка через тестирование

```
from django.test import TestCase
from prime import getFactors

class SimpleTest(TestCase):
   def test_one(self):
      self.assertItemsEqual([],getFactors(1))
   def test_two(self):
      self.assertItemsEqual([2],getFactors(2))
   def test_four(self):
      self.assertItemsEqual([2,2],getFactors(4))
   def test_five(self):
      self.assertItemsEqual([5],getFactors(5))
   def test_six(self):
      self.assertItemsEqual([2,3],getFactors(6))
   def test_seven(self):
      self.assertItemsEqual([7],getFactors(7))
```

```
def getFactors(num):
      res = []
      for i in xrange(2,num):
            if num % i == 0:
                  num = num / i
                  res.append(i)
      if num > 1:
            res.append(num)
      return res
```

Разработка через тестирование

```
from django.test import TestCase
from prime import getFactors

class SimpleTest(TestCase):
    def test_one(self):
        self.assertItemsEqual([],getFactors(1))
    def test_two(self):
        self.assertItemsEqual([2],getFactors(2))
    def test_four(self):
        self.assertItemsEqual([2,2],getFactors(4))
    def test_five(self):
        self.assertItemsEqual([5],getFactors(5))
    def test_six(self):
        self.assertItemsEqual([2,3],getFactors(6))
    def test_seven(self):
        self.assertItemsEqual([7],getFactors(7))
    def test_eight(self):
        self.assertItemsEqual([2,2,2],getFactors(8))
```

```
def getFactors(num):
      res = []
      for i in xrange(2,num):
            if num % i == 0:
                  num = num / i
                  res.append(i)
      if num > 1:
            res.append(num)
      return res
```

Разработка через тестирование

```
from django.test import TestCase
from prime import getFactors

class SimpleTest(TestCase):
   def test_one(self):
      self.assertItemsEqual([],getFactors(1))
   def test_two(self):
      self.assertItemsEqual([2],getFactors(2))
   def test_four(self):
      self.assertItemsEqual([2,2],getFactors(4))
   def test_five(self):
      self.assertItemsEqual([5],getFactors(5))
   def test_six(self):
      self.assertItemsEqual([2,3],getFactors(6))
   def test_seven(self):
      self.assertItemsEqual([7],getFactors(7))
   def test_eight(self):
      self.assertItemsEqual([2,2,2],getFactors(8))
```

```
def getFactors(num):
      res = []
      for i in xrange(2,num):
            while num % i == 0:
                  num = num / i
                  res.append(i)
      if num > 1:
            res.append(num)
      return res
```

Разработка через тестирование

```
from django.test import TestCase
from prime import getFactors

class SimpleTest(TestCase):
   def test_one(self):
      self.assertItemsEqual([],getFactors(1))
   def test_two(self):
      self.assertItemsEqual([2],getFactors(2))
   def test_four(self):
      self.assertItemsEqual([2,2],getFactors(4))
   def test_five(self):
      self.assertItemsEqual([5],getFactors(5))
   def test_six(self):
      self.assertItemsEqual([2,3],getFactors(6))
   def test_seven(self):
      self.assertItemsEqual([7],getFactors(7))
   def test_eight(self):
      self.assertItemsEqual([2,2,2],getFactors(8))
```

```
def getFactors(num):
      res = []
      for i in xrange(2,num):
            while num % i == 0:
                  num = num / i
                  res.append(i)
      if num > 1:
            res.append(num)
      return res
```

Разработка через тестирование

Три "золотых правила" TDD.

1. Код пишется исключительно для того, чтобы провальный тест был пройден.
2. Тесты пишутся исключительно до того момента, пока не был написан провальный тест.
3. После написанного провального теста дописывается ровно столько кода, сколько необходимо для прохождения провального теста.

Ссылки на источники

- http://butunclebob.com
- IEEE Guide to Software Engineering Body of Knowledge, SWEBOK, 2004
- http://www.protesting.ru
- http://habrahabr.ru/post/121162/
- http://habrahabr.ru/post/149903/
- http://habrahabr.ru/post/110307/
- http://habrahabr.ru/post/122098/
- Dan Pilone, Russ Miles. Head First Software Development
- http://glebradchenko.ru/
- http://project.dovidnyk.info
- https://www.google.ru/