Тестирование программного обеспечения

Вместо введения

«Тестирование программ может использоваться для демонстрации наличия ошибок, но оно никогда не покажет их отсутствие»

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

Эдсгер Вибе Дейкстра

История развития тестирования программного обеспечения

- Первые программные системы разрабатывались в рамках программ научных исследований или программ для нужд министерств обороны.
- Тестирование таких продуктов проводилось строго формализовано с записью всех тестовых процедур, тестовых данных, полученных результатов.
- Тестирование выделялось в отдельный процесс, который начинался после завершения кодирования, но при этом, как правило, выполнялось тем же персоналом.

История развития тестирования программного обеспечения

- В 1960-х много внимания уделялось «исчерпывающему» тестированию, которое должно проводиться с использованием всех путей в коде или всех возможных входных данных.
- В начале 1970-х тестирование ПО обозначалось как «процесс, направленный на демонстрацию корректности продукта»
- В 1980-х тестирование расширилось таким понятием, как предупреждение дефектов.
- В начале 1990-х в понятие «тестирование» стали включать планирование, проектирование, создание, поддержку и выполнение тестов и тестовых окружений, и это означало переход от тестирования к обеспечению качества, охватывающего весь цикл разработки ПО.
- В середине 1990-х с развитием Интернета и разработкой большого количества веб-приложений особую популярность стало получать «гибкое тестирование»

Понятие тестирования

- **Тестирование** - это процесс выполнения системы или компонента, при определенных тестировщиком условиях, для наблюдения и для оценки некоторых аспектов работы системы либо компонента.

- **В процессе тестирования** производится анализ ПО для выявления несоответствия между существующими особенностями выполнения ПО и требованиями к ПО (выявления дефектов).

- **Тестовый случай** – это набор входных данных, условий выполнения системы и ожидаемых результатов, разработанных для определенной цели, как то для оценки определенного пути выполнения приложения или для проверки соответствия определенному требованию.

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.

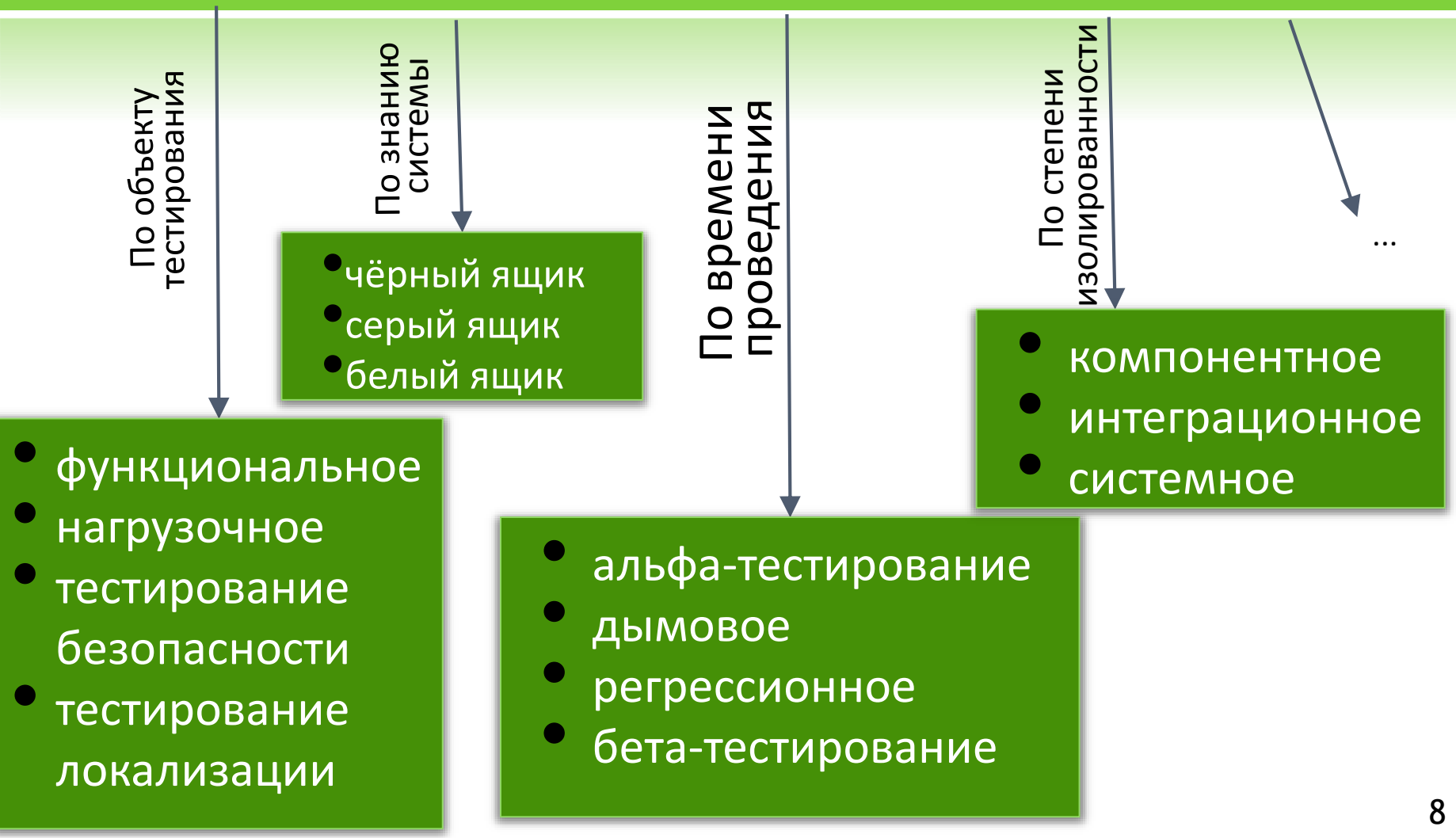
Понятие тестирования: чем тестирование НЕ является

Тестирование не может доказать корректность кода!

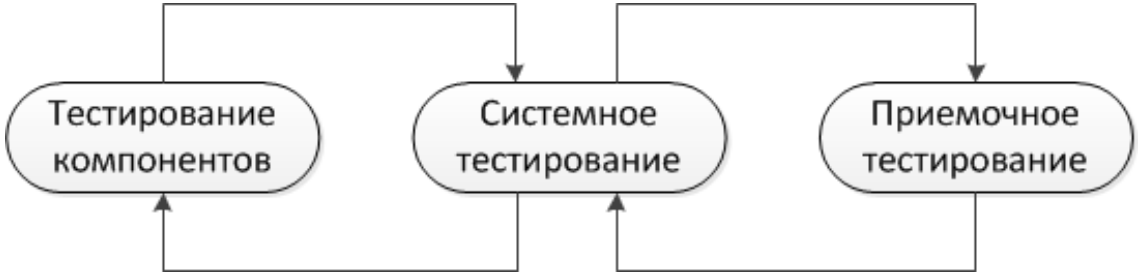
Цель тестирования состоит не в том, чтобы показать удовлетворительную работу программы.

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

Виды тестирования



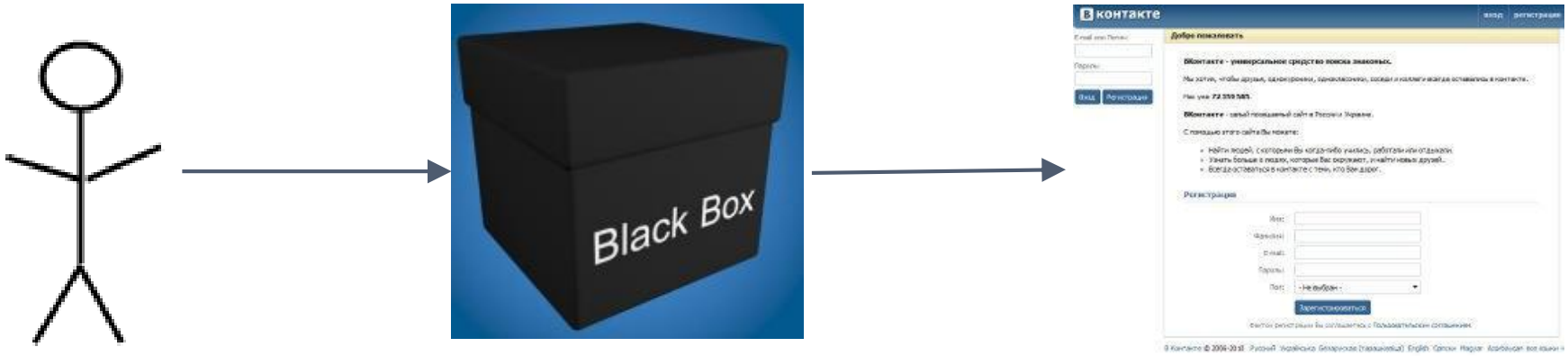
Процесс тестирования



- **Компонентное тестирование (юнит-тестирование):** тестируются отдельные компоненты системы для определения корректности их работы. Каждый компонент тестируется независимо от других. Компонентами могут быть отдельные функции, классы, библиотеки.
- **Системное тестирование (интеграционное тестирование)**: компоненты объединяются в общую систему и она тестируется как единое целое. В результате выявляются ошибки взаимодействия компонентов и проблемы определения интерфейсов.
- **Приемочное тестирование**: финальная стадия процесса тестирования. Система тестируется на данных, предоставленных пользователем (покупателем). Могут выявиться ошибки интерпретации данных и требований к системе, т.к. система может вести себя по-другому на реальных данных.

Чёрный, серый и белый ящики

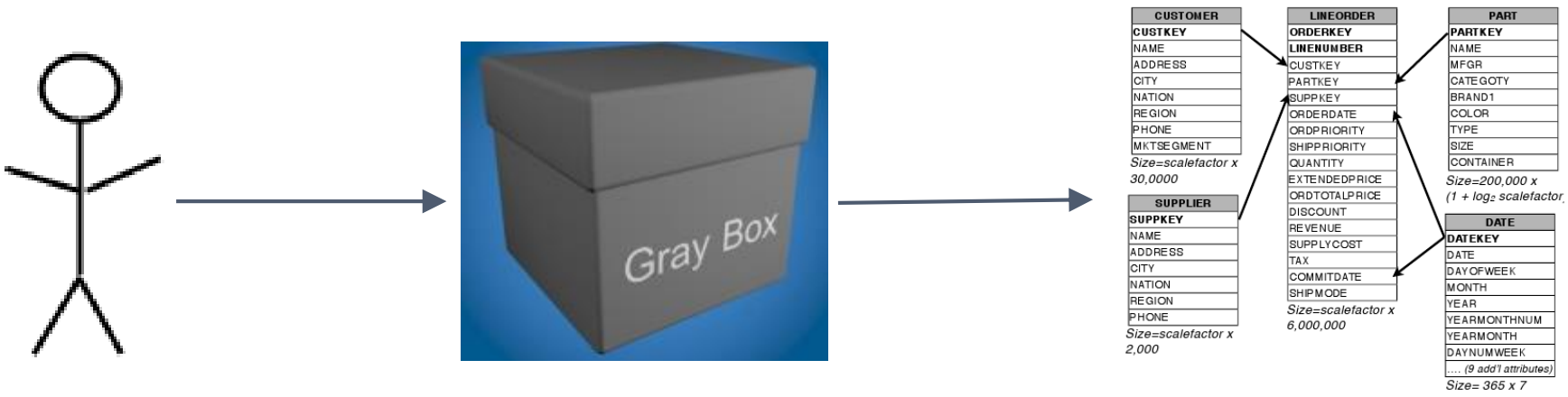
Конечные пользователи разрабатываемого ПО не имеют представления о его исходном коде, таблицах баз данных. Для них эта система является **чёрным ящиком**.



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**.



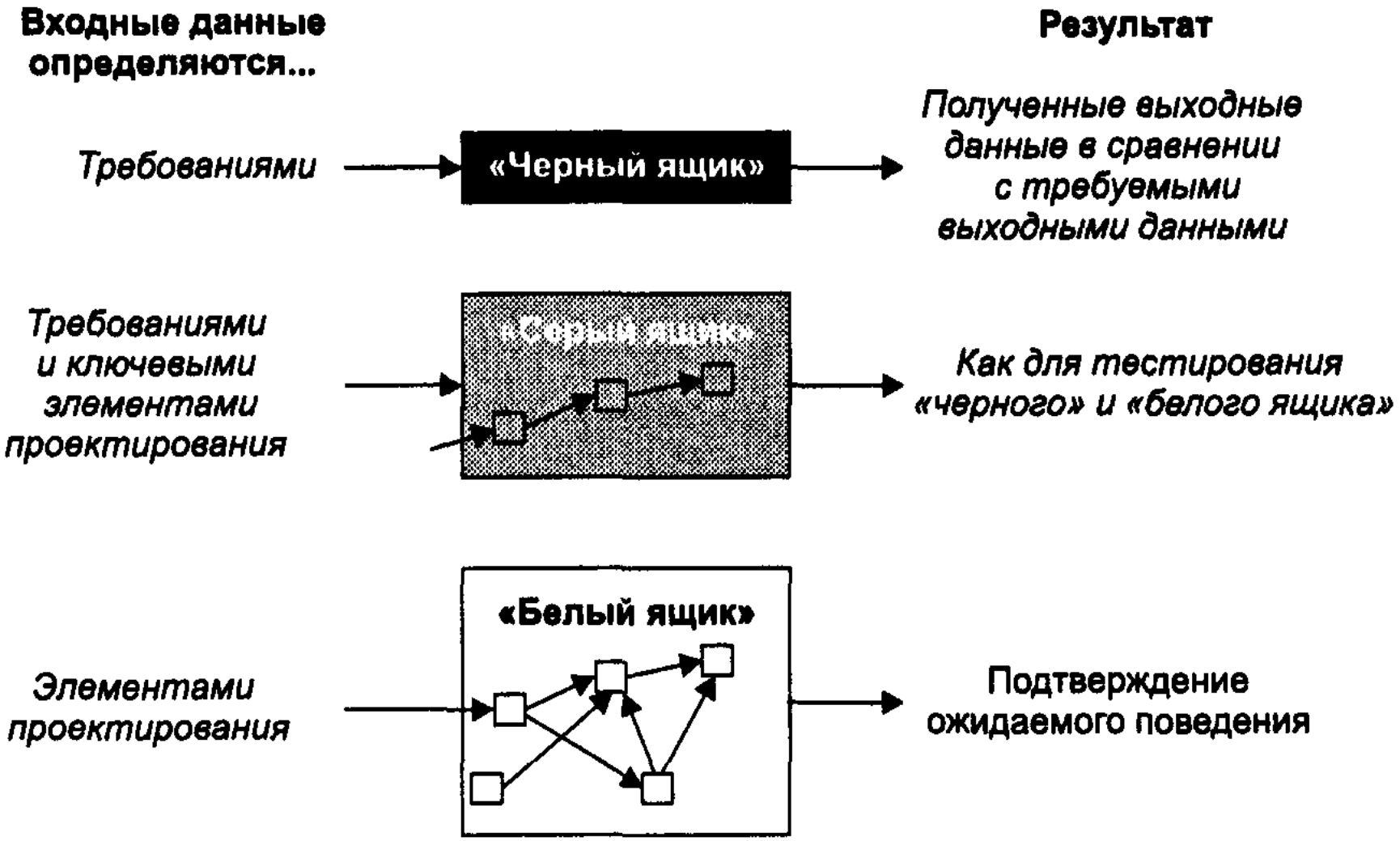
При тестировании по стратегии серого ящика руководствуются не только **спецификацией**, но и **ключевыми элементами проектирования**. Тестируется как **функционал**, так и **ожидаемое поведение программы.**

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 всевозмножных 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.

Покрытие операторов

- Критерии покрытия операторов подразумевает выполнение каждого оператора программы, по крайней мере, один раз.

```
void func (int a, int b, float x)
{
    if ((a > 1) && (b == 0)) x = x/a;
    if (a == 2 || x > 1) x++;
}
```

$$a = 2\ b = 0\ x = 3$$

Покрытие решений.

- В соответствии с этим критерием необходимо составить такое число тестов, при которых каждое условие в программе примет как истинное, так и ложное значение.
- В нашем примере, при тестировании каждый логический оператор должен принять значение 1 в одном тесте, и значение 0 в другом.

```
void func(int a, int b, float x) {
    if ((a > 1) && (b == 0)) x = x/a;
    if (a == 2 || x > 1) x++;
}
```

(1) (2) (3) (4)

a=2, b=0, x=3

a=0, b=1, x=1.

Покрытие путей.

- Записывается число тестов достаточное для того, чтобы все возможные пути выполнения программы были пройдены, по крайней мере, один раз.

```
void func(int a, int b, float x) {
          (1)              (2)
    if ((a > 1) && (b == 0)) x = x/a;
          (3)        (4)
    if (a == 2 || x > 1) x++;

}
```

Достаточно ли этого, чтобы утверждать, что программа не содержит ошибок?

(1)	(2)	(3)	(4)
1	1	1	1
0	1	1	1
1	0	1	1
1	1	0	1
1	1	1	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1
0	0	0	0

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.

Документация тестирования

- Оформление тестовых случаев

№	Предусловие	Действие	Ожидаемый результат	Результат теста
4	Пользователь авторизован в системе как «студент». Есть оценки за аттестацию.	Нажатие на кнопку «Вывод оценок за аттестацию»	Вывод оценок за аттестацию в соответствии с ожидаемым форматом.	Пройден.

Автоматизированное тестирование компонентов

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)
<...>
```

JUnit

JUnit — библиотека для тестирования программного обеспечения на языке Java, созданная Кентом Беком и Эриком Гаммой

Пример JUnit

```
public class MathFunc {
    private int variable;

    public MathFunc()
    { variable = 0; }

    public MathFunc(int var)
    { variable = var; }

    public int getVariable()
    { return variable; }

    public void setVariable(int variable)
    { this.variable = variable; }
```

```
public long factorial()
{
    long result = 1;
    if (variable > 1) {
        for (int i=1; i<=variable;
i++)
        result = result*i; }
    return result; }

    public long plus(int var) {
    long result = variable + var;
    return result;
    }
```

Пример JUnit

Для написания тестового класса нам нужно создать наследника junit.framework.TestCase. Затем необходимо определить конструктор, принимающий в качестве параметра строку (String) и передающую ее родительскому классу.

Пример JUnit

```
public class TestClass extends TestCase {
   public TestClass(String testName) { super(testName);  }

   public void testFactorialNull() {
      MathFunc math = new MathFunc();
      assertTrue(math.factorial() == 1);
   }

  public void testFactorialPositive() {
      MathFunc math = new MathFunc(5);
      assertTrue(math.factorial() == 120);
   }

   public void testPlus() {
      MathFunc math = new MathFunc(45);
      assertTrue(math.plus(123) == 168);
   }
}
```

Пример JUnit

Метод assertTrue проверяет, является ли результат выражения верным. Присутствуют и следующие методы - assertEquals, assertFalse, assertNull, assertNotNull, assertSame.

Пример JUnit

Для того, чтобы объединить тесты, можно воспользоваться классом TestSuite с его методом addTest.

```
public static void main(String[] args) {
    TestRunner runner = new TestRunner();
    TestSuite suite = new TestSuite();
    suite.addTest(new TestClass(“testFactorialNull”));
    suite.addTest(new TestClass(“testFactorialPositive”));
    suite.addTest(new TestClass(“testPlus”));
    runner.doRun(suite);
}
```