Министерство образования и науки
Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и кибербезопасности
Высшая школа технологий искусственного интеллекта
Направление: 02.03.01 Математика и компьютерные науки

Лабораторная работа №1

Тестирование ПО методом белого и черного ящика
по дисциплине «Методы тестирования программного обеспечения»

Выполнил: студент гр. 5130201/20102. _____ Салимли А.
Проверил: к.т.н. _____ Курочкин М.А.

«_____» _____ 202_г.

Санкт-Петербург, 2025 г.

Содержание

Введение

Тестирование является одним из важнейших аспектов современного процесса разработки программного обеспечения. Существует широкий спектр его видов, включая модульное, интеграционное, функциональное, системное и приемочное тестирование. В этом проекте основное внимание будет уделено модульному тестированию.

Модульное тестирование, также известное как unit-тестирование, представляет собой начальную стадию проверки, которая обычно служит отправной точкой для верификации правильности работы программных продуктов. Этот процесс включает тестирование отдельных элементов программного обеспечения, таких как классы, модули и функции. Главной целью модульного тестирования является выявление ошибок, то есть нахождение таких ситуаций, когда поведение модуля не соответствует его предварительно установленным спецификациям.

Существует два основных метода, используемых в unit-тестировании: методы «черного» и «белого» ящика. В этом исследовании детально рассматриваются процедуры, относящиеся к обоим подходам. Эти методы также применяются для тестирования двух программных модулей в соответствии с заранее определенными спецификациями и блок-схемами.

1 Описание методологий тестирования

1.1 Метод черного ящика

Тестирование методом черного ящика - проверка, при которой тестировщик не имеет доступа к коду. Он, как реальный клиент или пользователь, оценивает функции и работу программы, ориентируясь исключительно на интерфейс взаимодействия. Тестирование «черным ящиком» может происходить как вручную, так и автоматически. И, как и в случае «белого ящика», специалист создает test-кейсы, чтобы покрыть все возможные сценарии использования программы. Такое тестирование можно проводить на любом этапе разработки ПО. Часто оно не позволяет выявить скрытые ошибки, но зато доступно начинающим специалистам и помогает посмотреть на продукт глазами обычного пользователя. При использовании метода «черного ящика» тестировщик не видит код программы.

Далее рассматривается три приема тестирования черного ящика: разбиение множества входных данных на эквивалентные классы, анализ значений на их границах и построение причинно-следственных диаграмм.

1.1.1 Эквивалентные разбиения

Спроектированный тест должен обеспечивать значительное покрытие других возможных тестов, предоставляя информацию о наличии или отсутствии ошибок в тех ситуациях, которые не рассматриваются в рамках данного конкретного набора входных значений. Исходя из этого, следует стремиться разделить всю область входных данных программы на конечное число классов эквивалентности. Это упростит процесс тестирования, так как проверка одного представительного значения из класса эквивалентности позволит с высокой вероятностью утверждать, что тестирование любого другого значения из этого же класса даст аналогичные результаты.

Классы эквивалентности определяются путем тщательного анализа каждого входного условия, которое обычно формулируется как утверждение или формулировка в спецификации, и деления его на несколько групп. Это позволяет организовать тестирование более эффективно, минимизируя количество необходимых тестов, сохраняя при этом качество и полноту проверки. Существует два типа классов эквивалентности:

- Допустимые классы эквивалентности - допустимые входные данные программы
- Недопустимые классы эквивалентности - недопустимые входные данные программы

Определение тестов по построенным классам эквивалентности состоят из трех этапов:

- Назначить каждому классу эквивалентности уникальный номер
- Записать новые тесты, охватывающие как можно большее количество оставшихся неохваченными допустимых классов эквивалентности, пока не будут покрыты все допустимые классы
- Записать новые тесты, каждый из которых охватывает один и только один из оставшихся неохваченными недопустимых классов эквивалентности, пока не будут покрыты все недопустимые классы

1.1.2 Анализ граничных значений

Граничные условия - ситуаций, которые возникают на границах входных и выходных классов эквивалентности. Анализ граничных значений отличается от подхода, связанного с разбиением на классы эквивалентности, по двум ключевым аспектам. Во-первых, вместо выбора любого элемента из класса эквивалентности в качестве его представителя, анализ граничных значений требует выделения таких элементов, которые позволяют протестировать каждую границу данного класса. Во-вторых, при разработке тестов акцент делается не только

на входных условиях (области входных значений), но и на диапазоне результатов (выходных классах эквивалентности).

1.1.3 Причинно-следственные диаграммы

Анализ граничных значений и разбиение на классы эквивалентности имеют недостаток - они не рассматривают случаи комбинаций входных условий.

Причинно-следственные диаграммы помогают систематически выбирать высокорезультативные тесты. Так же метод может обнаружить неполноту и неоднозначность исходных спецификаций.

Причинно-следственная диаграмма - формальный графический язык, на который транслируется спецификация написанная на естественном языке.

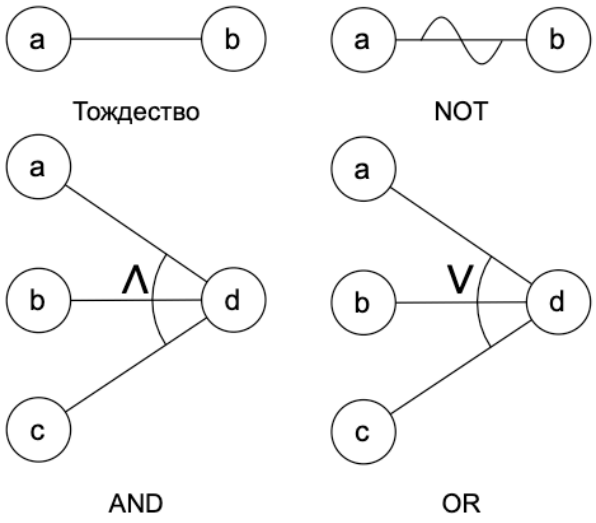

Рис. 1 Базовая нотация в причинно-следственных диаграммах

Тут каждый узел диаграммы может находится в бинарном состоянии: 0 - состояние «отсутствует», а 1 - состояние «присутствует».

Процесс построения теста:

- Спецификация разбивается на части с которыми легче работать
- В спецификации определяются причины и следствия

  Причина - Входное условие или класс эквивалентности входных условий.

  Следствие - это выходное условие или преобразование системы (долговременное воздействие, которое входное условие оказывает на состояние программы или системы).

  Причины и следствия определяются путем последовательного чтения спецификации и

подчеркивания тех слов или фраз, которые описывают причины и следствия. Каждой причине и каждому следствию присваивается уникальный номер.

- Сематическое содержание спецификации преобразуется в булевый граф, связывающий причины и следствия. Этот граф - сама диаграмма.
- Диаграмма снабжается примечаниями, задающими ограничения и описывающими комбинации причин и/или следствий, реализация которых невозможна из-за синтаксических или внешних ограничений
  - Ограничение E требует, чтобы всегда выполнялось условие, в соответствии с которым только a или только b может быть равно 1 (a и b не могут быть равны 1 одновременно, но обе величины могут быть равны 0)
  - Ограничение I требует, чтобы по крайней мере одна из величин, a, b или c, была равна 1 (a, b и c не могут быть равны 0 одновременно)
  - Ограничение O требует, чтобы одна и только одна из величин, a или b, была равна 1
  - Ограничение R требует, чтобы a было равно 1, только если b равно 1 (т.е. a не может быть равно 1, если b равно 0).
- Путем прослежования состояний условий диаграмма преобразуется в таблицу решений с ограниченными входами. Каждый столбец таблицы решений соответствует тесту.
- Столбцы таблицы решений преобразуются в тесты.

1.2 Метод белого ящика

Метод белого ящика - тестирование с управляемой логикой программы (logic-driving testing) Тестировщик подбирает тестовые данные путем анализа логики программы с учетом предоставленной ему спецификации программы и блок-схемы программы.
Это подразумевает, что если все возможные пути выполнения программы были протестированы, можно сделать вывод о том, что программа полностью протестирована.

1.2.1 Покрытие решений

Согласно критерию покрытия решений, количество тестов должно быть таким, чтобы каждое условие в программе хотя бы один раз принимало значение true (истина) и false (ложь). Иными словами, каждая логическая ветвь всех операторов ветвления в программе должна быть выполнена хотя бы один раз. Обычно покрытие решений соответствует критерию покрытия операторов или инструкций. Поскольку каждая инструкция относится к определенному пути, который начинается либо с оператора ветвления, либо с точки входа в программу, тестирование всех ветвей одновременно подразумевает и тестирование всех инструкций. Однако данное правило не применяется, по крайней мере, в двух следующих случаях:

- программы, которые не включают в себя точки ветвления
- программы или методы, имеющие несколько точек входа. В этой ситуации некоторые инструкции будут выполняться только в том случае, если программа начинается с определенной точки входа.

1.2.2 Покрытие условий

Согласно этому критерию, количество тестов должно быть таким, чтобы каждое из элементарных условий, составляющих проверочное выражение в точке ветвления, принимало оба возможных логических значения — true и false — хотя бы один раз. Поскольку покрытие условий, подобно покрытию решений, не всегда гарантирует выполнение каждой инструкции, необходимо дополнительно требовать, чтобы каждая точка входа в программу или подпрограмму была достигнута хотя бы один раз.

1.2.3 Покрытие решений и условий

По этому критерию набор тестов считается достаточно полным, если выполняются следующие условия:

- Каждое условие в решении принимает все возможные значения хотя бы один раз
- Каждый возможный результат решения проверяется как минимум один раз
- Управление передается каждой точке входа хотя бы один раз

Однако, несмотря на то, что данный подход кажется способным охватить все возможные результаты решений для всех условий, это часто не достигается из-за того, что одно условие может скрывать другое.

1.2.4 Покрытие операторов

Покрытие операторов - метод проектирования тестов методом белого ящика, который включает в себя выполнение всех исполняемых операторов (if, for и switch) в исходном коде как минимум один раз. Это необходимое но не достаточное решение проблемы тестирования ПО. Метод довольно слабый, если мы решим отказаться от выше описанных методов.

1.2.5 Комбинаторное покрытие условий

Метод при котором каждая возможная комбинация результатов вычисления условий в каждом решении и каждая точка входа проверяются по крайней мере один раз.

2 Программный модуль 1

Название программы: GCD (Алгоритм Евклида для нахождения НОД).

2.1 Постановка задачи

Дано:

a :: Integer -> Integer -- целое число a :: $a \in \mathbb{Z}$

b :: Integer -> Integer -- целове число b :: $b \in \mathbb{Z}$

Требуется:

Найти наибольший общий делитель (НОД) чисел a и b, алгоритмом Евклида.

Ограничения:

$\forall a, b : a, b \in \mathbb{Z}$ (a, b - целые числа)

$a, b \in [-100000, 100000]$

2.2 Спецификация

Таб.1 Спецификация

№	Входные данные	Результат	Реакция программы
1	a = 14 b = 15	myGCD(14,15) = 1	Вывод на экран результата НОД(14,15)
2	a = -15 b = -45	myGCD(-15,-45) = | -15| = 15	Вывод на экран результата НОД(-15,-45)
3	a = 0.4 b = 0.5	Ошибка: Числа должны быть типа: Integer	stack - сборщик проекта выводит ошибку в консоль среды разработки: No instance for ‘Fractional Int’ arising from the literal ‘0.5’
4	a = 0 b = 0	myGCD(0,0) = 0	Вывод на экран результата НОД(0,0) (математически не определен)

2.3 Тестирование методом черного ящика

2.3.1 Разбиение на классы эквивалентности

В таблице 2 представлено разбиение на классы эквивалентности. В таблицах 3 и 4 приведены тесты, проверяющие случаи допустимых и недопустимых классов соотвественно.

Таб. 2 Разбиение на классы эквивалентности

Входные данные	Допустимые классы	Недопустимые классы
a	a :: Целые числа (Int, Integer) : {1} a < 100000 {2} a != 0 {3}	a :: Вещественные {7} a :: Комплексные {8} a :: Рациональные {9} a :: Не число - NaN {10} a > 10^14 {11} a < -10^14 {12}
b	b :: Целые числа (Int, Integer) {4}, b < 100000 {5} b != 0 {6}	b :: Вещественные {13} b :: Комплексные {14} b :: Рациональные {15} b :: Не число - NaN {16} b > 10^14 {17} b < -10^14 {18}

Таб. 3 Тесты для допустимых классов

№	Класс эквивалентности	Входное значение: a	Входное значение: b	Ожидаемый результат	Фактический результат	Статус теста
1	{{1},{2},{3}}	14	15	1	1	Пройден
2	{{4},{5},{6}}	6	36	6	6	Пройден

Таб.4 Тесты для недопустимых классов

№	Класс эквивалентности	Входное значение: a	Входное значение: b	Ожидаемый результат	Фактический результат	Статус теста
1	{{7}}	1.4	1.45	No instance for ‘Fractional Int’ arising from the literal ‘0.5’	No instance for ‘Fractional Int’ arising from the literal ‘0.5’	Пройден
2	{{8}}	1 :+ 2	3 :+ 52	Couldn't match type ‘Complex’ with ‘Int’ Expected: Int Actual: String	Couldn't match type ‘Complex’ with ‘Int’ Expected: Int Actual: String	Пройден
3	{{9}}	0.5	0.75	No instance for ‘Fractional Float’ arising from the literal ‘0.5’	No instance for ‘Fractional Int’ arising from the literal ‘0.5’	Не пройден
4	{{10}}	«Hello»	«Haskell»	Couldn't match type ‘String’ with ‘Int’ Expected: Int Actual: String	Couldn't match type ‘String’ with ‘Int’ Expected: Int Actual: String	Пройден
5	{{11}}	10^24	10^25	OutOfBound	OutOfBound	Пройден
6	{{15}}	2.5	3.5	No instance for ‘Fractional Int’ arising from the literal ‘0.5’	No instance for ‘Fractional Int’ arising from the literal ‘0.5’	Пройден
7	{{16}}	‘F’	\n -> n	Couldn't match type ‘[Char]’ with ‘Int’ Expected: Int Actual: String	Couldn't match type ‘[Char]’ with ‘Int’ Expected: Int Actual: String	Пройден
8	{{18}}	-10^23	-10^24	OutOfBound	OutOfBound	Пройден

2.3.2 Анализ граничных условий

В программе присутствуют 4 граничных условия:

$$a,\ b :: \begin{cases} a < 10^{14} \\ a > -10^{14} \\ b < 10^{14} \\ b > 10^{14} \end{cases} \approx \begin{cases} a \leftarrow maxBound :: Int \\ a \leftarrow minBound :: Int \\ b \leftarrow maxBound :: Int \\ b \leftarrow minBound :: Int \end{cases}$$

Для каждой из границ определим тесты:

- тест соответствующий наиболее близкому к границе числу
- тест соответствующий целому числу, которое выходит за границу maxBound
- тест соответствующий дробному числу, которое выходит за границу minBound

Таб. 5 Тесты граничных условий

№	Входное значение: a	Входное значение: b	Ожидаемый результат	Фактический результат	Статус теста
1	999999999999	-999999999999	999999999	999999999	Пройден
2	-999999999999	999999999999	999999999	999999999	Пройден
3	99999999999999	-999999999999	OutOfBound	OutOfBound	Пройден
4	99999999999999	99999999999999	OutOfBound	OutOfBound	Пройден
5	-1000000000000	99999999999999	OutOfBound	OutOfBound	Пройден
6	-9999999999999	-1000000000000	OutOfBound	OutOfBound	Пройден

2.3.3 Причинно-следственная диаграмма

Для диаграммы нужно определить причины, промежуточные данные, следствия:

Причины:

A) a - Целое число: $a \in \mathbb{Z}$

B) b - Целое число: $b \in \mathbb{Z}$

C) $-10^{14} < a < 10^{14} \approx a \leftarrow maxBound :: Int$

D) $-10^{14} < b < 10^{14} \approx b \leftarrow maxBound :: Int$

Промежуточные:

E) a - целое и не больше 10^{14} и не меньше -10^{14} — $a : a \in \mathbb{Z} \& -10^{14} < a < 10^{14}$

F) b - целое и не больше 10^{14} и не меньше -10^{14} — $b : b \in \mathbb{Z} \& -10^{14} < b < 10^{14}$

Следствия:

G) Программа выводит НОД для *a, b*

H) Программа выводит ошибку

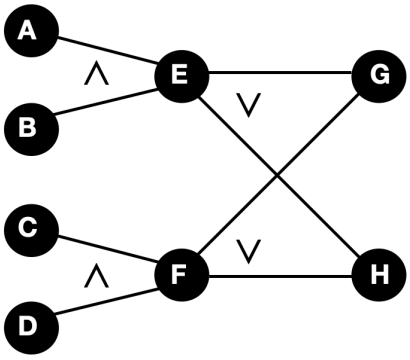

Рис.3 Причинно-следственная диаграмма

Таблица состоит из 0 и 1, где:

$$1_{\mathbb{A}} = \begin{cases} 1, E \wedge F = G \\ 0, E \vee F = H \end{cases}$$

Таб. 6 Таблица решений

№	1	2	3	4	5
A	1	0	1	1	1
B	0	1	0	1	1
C	1	1	1	0	1
D	1	1	0	1	1
E	1	1	1	0	1
F	1	1	1	1	1
G	1	1	1	1	1
H	1	1	1	1	0

Тесты на основе причинно-следственных диаграмм:

{A, 2} : a = 0.4, b = 3

{B, 1} : a = 4, b = 0.2

{B, 3} : a = 3, b = 0.45

{C, 4} : a = 10^15, b = 2

{D, 3} : a = 4, b = 10^19

{H, 5} : 1

2.4 Тестирование методом белого ящика

На рисунке 4 представлена блок-схема алгоритма. Буквами будем отмечать пути ветвления.

Условия в ветвлениях программы:

- *b == 0*
- *a != b*
- *a > b*

2.4.1 Покрытие операторов

Критерием покрытия является выполнение каждого оператора программы не менее одного раза. Критерий является необходимым но недостаточным условием для полного тестирования по методу белого ящика.

Тесты:

a = 6, b = 36;

Таблица теста покрытия операторов:

Таб. 7 Таблица теста покрытия операторов

Значение: a	Значение: b	Ожидаемый результат	Фактический результат	Путь выполнения	Условия в пути выполнения	Статус теста
6	36	6	6	6,36 -> 36,6 -> 6,0	b != 0 b != 0 b = 0	Пройден

Замечание* - Данный подход не рассматривает все возможные варианты входных данных, а значит является неэффективным для нахождения ошибок.

2.4.2 Покрытие решений

Согласно этому критерию требуется составить такое множество тестов, при которых каждое условие программы будет выполнено как в истинном, так и в ложном варианте. Иными словами, к тестам, разработанным по принципу покрытия операторов, нужно добавить проверки всех возможных путей выполнения. Таким образом, к уже имеющемуся набору тестов необходимо включить следующие случаи:

- a = 10, b = 0 (A)
- a = 10, b != 0 (B)
- a = 14, b = 3 (C)
- a = 25, b = 25 (D)
- a = 40, b = 30 (E)
- a = 30, b = 45 (F)

Эти 6 входных значения проверяют все условия в программном модуле.

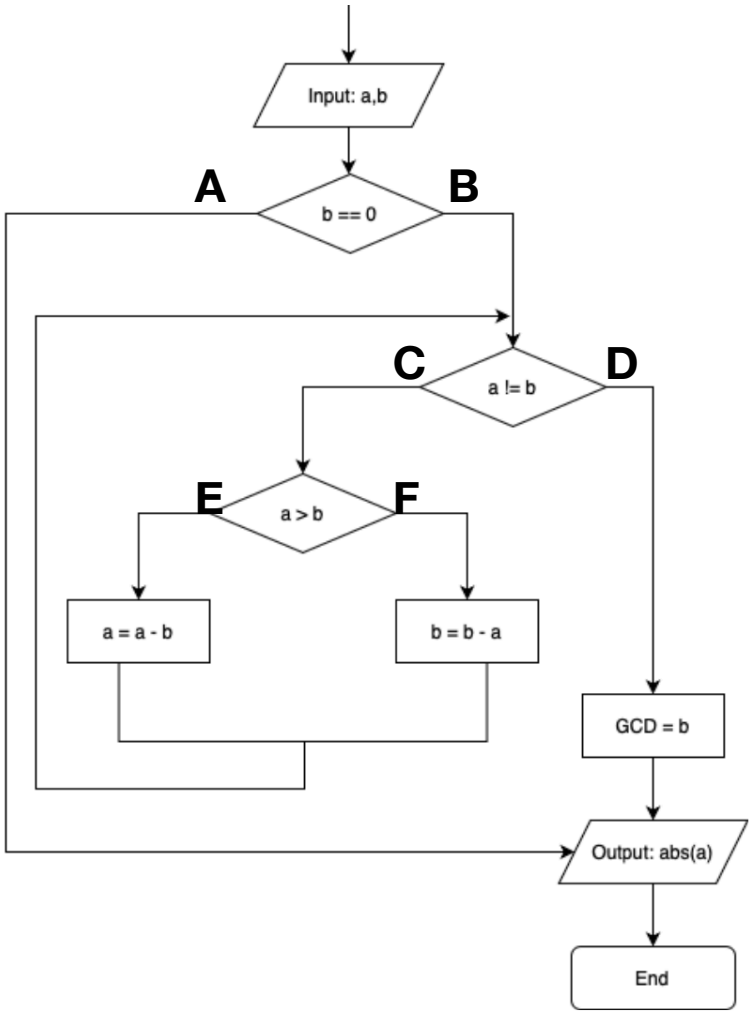

Рис. 4 Пути и их варианты

Таблица тестов покрытия решений содержит наборы входных данных для прохождения всех ветвей.

Таб. 8 Тесты покрытия решений

Путь	Входное значение: a	Входное значение: b	Ожидаемый результат	Фактический результат	1	2	3	4	Статус теста
A	10	0	10	10	T	-	-	-	Пройден
BD	25	25	25	25	F	F	-	-	Пройден
BCED	40	30	10	10	F	T	F	-	Пройден
BCFD	30	45	15	15	F	T	F	F	Пройден

2.4.3 Покрытие условий

Результаты покрытия условий чаще дают результат лучше чем покрытия решений. В покрытиях условий - записываются число тестов, достаточное для того, чтобы все возможные результаты каждого условия в решении выполнялись по крайней мере один раз.
Можно добавить к тестам:

a = «Hello!» :: не число.

Полученная таблица:

Таб. 9 Тесты покрытия условий

Входное значение: a	Входное значение: b	Ожидаемый результат	Фактический результат	Путь	1	2	3	4	Статус теста
«Hello!»	3	Couldn't match type ‘String’ with ‘Int’ Expected: Int Actual: String	Couldn't match type ‘String’ with ‘Int’ Expected: Int Actual: String	-	-	-	-	-	Пройден

2.4.4 Покрытие решений и условий

Согласно этому критерию тесты необходимо составить таким образом, чтобы каждое условие проверялось хотя бы один раз, каждое решение выполнялось не менее одного раза, и каждый оператор срабатывал хотя бы один раз. Совокупность всех этих тестов обеспечивает полное покрытие условий и решений. Таким образом, все уже составленные тесты относятся к данному методу.
Таблица покрытия решений и условий:

Таб. 10 Покрытия решений и условий

Путь	Входное значение: a	Входное значение: b	Ожидаемый результат	Фактический результат	1	2	3	4	Статус теста
A	10	0	10	10	T	-	-	-	Пройден
BD	25	25	25	25	F	F	-	-	Пройден
BCED	40	30	10	10	F	T	F	-	Пройден
BCFD	30	45	15	15	F	T	F	F	Пройден
-	«Haskell»	3	Couldn't match type ‘String’ with ‘Floatt’ Expected: Int Actual: String	Couldn't match type ‘String’ with ‘Int’ Expected: Int Actual: String	-	-	-	-	Не пройден

2.4.5 Комбинаторное покрытие условий

Этот критерий требует создания такого количества тестов, при котором каждая возможная комбинация результатов вычисления условий в каждом решении и каждая точка входа проверяется по крайней мере один раз.

Можно добавить к тестам:
a = [2,4,5]
b = Functor v
a = Monad m

Таб. 11 Комбинаторное покрытие условий

Путь	Входное значение: a	Входное значение: b	Ожидаемый результат	Фактический результат	1	2	3	4	Статус теста
A	10	0	10	10	T	-	-	-	Пройден
BD	25	25	25	25	F	F	-	-	Пройден
BCED	40	30	10	10	F	T	F	-	Пройден
BCFD	30	45	15	15	F	T	F	F	Пройден
-	«Haskell»	3	Couldn't match type ‘String’ with ‘Int’ Expected: Int Actual: String	Couldn't match type ‘String’ with ‘Int’ Expected: Int Actual: String	-	-	-	-	Пройден
-	[2,4,5]	3	Couldn't match type ‘List’ with ‘Int’ Expected: Int Actual: String	Couldn't match type ‘List’ with ‘Int’ Expected: Int Actual: String	-	-	-	-	Пройден
-	4	Functor v	Couldn't match type ‘Functor’ with ‘Int’ Expected: Int Actual: String	Couldn't match type ‘Functor’ with ‘Int’ Expected: Int Actual: String	-	-	-	-	Пройден

-	Monad m	4	Couldn't match type ‘Monad’ with ‘Int’ Expected: Int Actual: String	Couldn't match type ‘Monad’ with ‘Int’ Expected: Int Actual: String	-	-	-	-	Пройден

3 Программный модуль 2

Название программы: Факториал числа.

3.1 Постановка задачи

Дано: F - целое число.

Требуется: Найти факториал числа F и вывести результат в консоль среды разработки.

Ограничения:

$F \in \mathbb{Z}$ - целое число,

$0 \leq F < 100$.

3.2 Спецификация

Входное значение: F	Результат	Реакция программы
F = 3	6	Вывод на экран: 3! = 6 Завершение программы
F = 0	1	Вывод на экран: 0! = 1 Завершение программы
F = -3	Ошибка! Число должно быть больше нуля!	Вывод на экран: Ошибка! Число должно быть больше нуля! Завершение программы
F = 100	Ошибка! Слишком большое число! Пожалейте процессор!	Вывод на экран: Ошибка! Слишком большое число! Пожалейте процессор! Завершение программы
F = 0.5	Ошибка! Тип числа не Int или оно не Integer	Вывод на экран: Ошибка! Тип числа не Int или оно не Integer Завершение программы

3.3 Тестирование методом черного ящика

3.3.1 Разбиение на классы эквивалентности

Ниже представлено разбиение на классы эквивалентности. А так же приведены тесты в таблицах 12, 13, проверяющие случаи допустимых и недопустимых классов соотвественно.

Входное значение: F

Допустимые классы: $\forall F : \begin{cases} F \in \mathbb{Z} \text{ (целое число)} \\ F \in [0,100] \end{cases} :: (A)$

Недопустимые классы: $\begin{bmatrix} \forall F : F \in \mathbb{R} := (B) \\ \forall F : F \in \mathbb{Q} := (C) \\ \forall F : F \in \mathbb{C} := (D) \\ \forall F : F \notin Integer := (E) \end{bmatrix}$

Таб. 12 Тесты для допустимых классов

№	Класс эквивалентности	Входное значение: F	Ожидаемый результат	Фактический результат	Статус теста
1	A	3	6	6	Пройден

Таб. 13 Тесты для недопустимых классов

№	Класс эквивалентности	Входное значение: F	Ожидаемый результат	Фактический результат	Статус теста
1	B	0.3221	Ошибка! Тип числа не Int или оно не Integer	Ошибка! Тип числа не Int или оно не Integer	Пройден
2	C	0.5	Ошибка! Тип числа не Int или оно не Integer	Ошибка! Тип числа не Int или оно не Integer	Пройден
3	D	2 :+ 3	Ошибка! Тип числа не Int или оно не Integer	Ошибка! Тип числа не Int или оно не Integer	Пройден
4	E	«GHCI»	Ошибка! Тип числа не Int или оно не Integer	Ошибка! Тип числа не Int или оно не Integer	Пройден

3.3.2 Анализ граничных условий

В программе есть два граничных условия:

$$\forall F : \begin{cases} F \leq 100 \\ F \geq 0 \end{cases}$$

Для каждой из границ определим тесты:

- тест соответствующий наиболее близкому к границе числу
- тест соответствующий целому числу, которое выходит за границу (+1)
- тест соответствующий дробному числу, которое выходит за границу (+0.05)

Результат:

Таб.14 Тесты граничных условий

№	Входное значение: F	Ожидаемый результат	Фактический результат	Статус теста
1	0	1	1	Пройден
2	99	Значение 99!	Значение 99!	Пройден
3	101	Ошибка! Число должно быть меньше 100!	Ошибка! Число должно быть меньше 100!	Не пройден

4	100.4	Ошибка! Число должно быть меньше 100!	Ошибка! Число должно быть меньше 100!	Пройден
5	-1	Ошибка! Число должно быть больше!	Ошибка! Число должно быть больше!	Пройден

3.3.3 Причинно-следственная диаграмма

Для построения причинно-следственной диаграммы, нужно ввести:

Причины:

$$F: \begin{cases} F \in \mathbb{Z} := D \\ F \in \mathbb{R} := E \\ F \notin Integer := G \\ 0 \leq F \geq 100 := H \\ F < 0 := I \\ F > 100 := G \end{cases}$$

Следствия:

(A) : Программа вычисляет F!

(B) : Вывод ошибки связанной с граничными значениями F

(C) : Вывод ошибки связанной с типом F

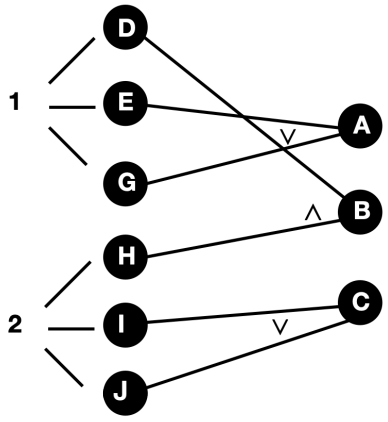

Рис. 5 Причинно-следственная диаграмма

Таблица решений по которой составленны тесты:

Таб. 15 Таблица решений

	1	2	3	4
1	1	0	0	0
2	0	0	1	0

3	1	0	0	1
4	0	0	0	1

Таб. 16

№	Причины	Входное F	Ожидаемый результат	Фактический результат	Статус теста
1	D, H	5	120	120	Пройден
2	G	Monad m	Ошибка! Число не Int и не Integer!	Ошибка! Число не Int и не Integer!	Пройден
3	E, I	22.5	Ошибка! Число не Int и не Integer!	Ошибка! Число не Int и не Integer!	Пройден
4	E, J	-244	Ошибка! Число должно быть больше 0!	Ошибка! Число должно быть больше 0!	Пройден

3.4 Тестирование методом белого ящика

На рисунке рис. 6 представлена блок-схема программы. Буквами отмечены пути ветвления.

Условия в ветвлениях:

- F - целое число
- F = 0
- F < 0
- F > 100
- i = F

3.4.1 Покрытие операторов

Критерием покрытия является выполнение каждого оператора программы хотя бы один раз.

Это необходимое, но недостаточное условие для приемлемого тестирования по методу белого ящика. Необходимо реализовать тесты:

- F - не целое : F = 0.5
- F = 0
- F < 0 : F = -1
- F > 0 && F < 100 : F = 3

В таблице представлен набор тестов, покрывающие все операторы.

Таб.17 Тесты покрытия операторов

№	Входное значение F	Ожидаемый результат	Фактический результат	Путь	1	2	3	4	5	Статус тестирования
1	0.5	Ошибка! F не Int и не Integer	Ошибка! F не Int и не Integer	A	F	-	-	-	-	Пройден

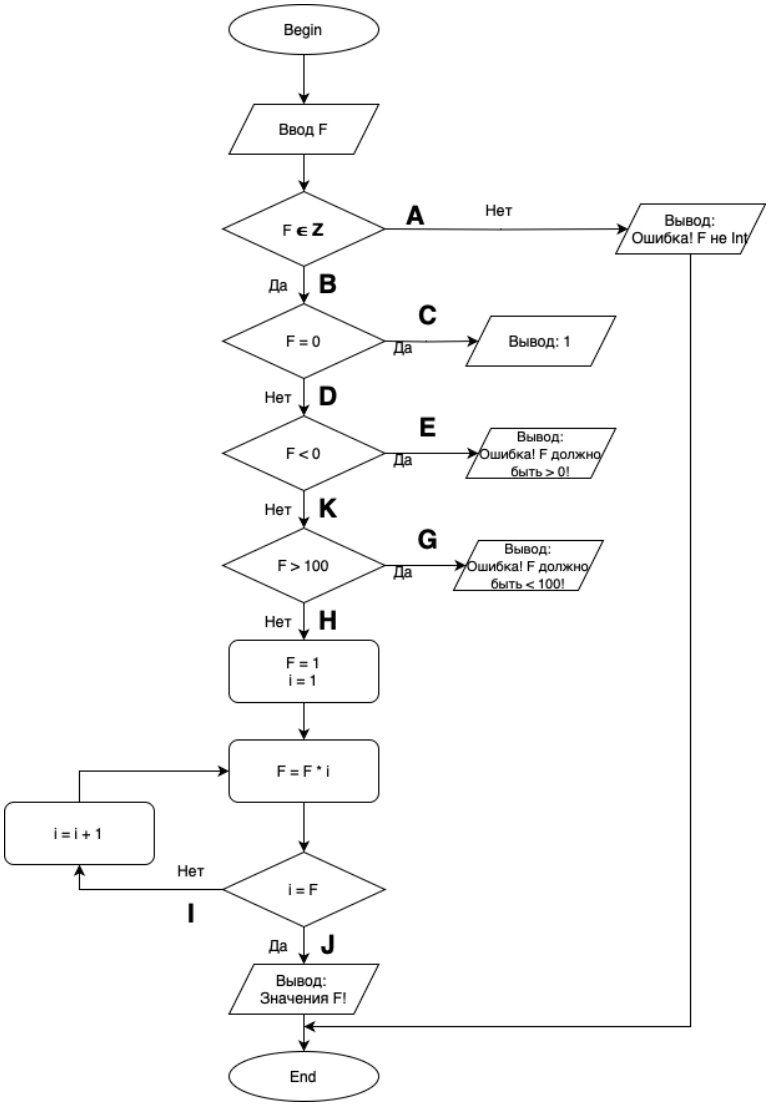

Рис. 6 Блок-схема программного модуля 2

2	0	1	1	BC	T	T	-	-	-	Пройден
3	-1	Ошибка! F не Int и не Integer	Ошибка! F не Int и не Integer	BDKG	T	F	T	-	-	Пройден
4	3	6	6	BDHKJI	T	F	F	F	T	Пройден

Замечание* - Данный подход не рассматривает все возможные варианты входных данных, а значит является неэффективным для нахождения ошибок.

3.4.2 Покрытие решений и условий

В соответствии с этим критерием необходимо составить такое число тестов, при которых каждое условие в программе примет как истинное значение, так и ложное значение. Таким образом, к тестам, составленным для метода покрытия операторов, необходимо добавить

тесты, которые будут проверять все возможные переходы. К уже написанным тестам необходимо добавить следующий тест:

F > 100 : F = 101

Таб. 18 Тесты покрытия решений

№	Входное значение F	Ожидаемый результат	Фактический результат	Путь	1	2	3	4	5	Статус тестирования
1	101	Ошибка! Число должно быть меньше 100!	Ошибка! Число должно быть меньше 100!	BDKG	T	F	F	T	-	пройден

4 Результаты тестирования

В ходе тестирования первой программы методами белого и черного ящика было составлено

- 11 теста для анализа граничных условий (0 не пройдено),
- 5 тестов для метода эквивалентного разбиения (1 не пройдено),
- 5 тестов по методу причинно-следственной диаграммы (0 не пройдено),
- 1 тест для метода покрытия операторов (0 не пройдено),
- 3 теста для метода покрытия решений (0 не пройден),
- 1 тест для метода покрытия условий (0 не пройдено),
- 5 тестов для метода покрытия решений и условий (1 не пройден),

3 теста для комбинаторного метода покрытия условий (0 не пройден).

В ходе тестирования второй программы методами белого и черного ящика было составлено

- 6 тестов для анализа граничных условий (0 не пройдено),
- 5 тестов для метода эквивалентного разбиения (0 не пройдено),
- 7 тестов по методу причинно-следственной диаграммы (0 не пройдено),
- 4 теста для метода покрытия операторов (0 не пройдено),

1 тест для метода покрытия решений и условий (0 не пройдено)

Заключение

В рамках лабораторной работы были изучены методы модульного тестирования: метод черного ящика и метод белого ящика.

Были спроектированы тесты при помощи изученной методологии для двух программ. При составлении тестов использовались методы:

- разбиение на классы эквивалентности;
- анализа граничных значений;
- причинно-следственная диаграмма;
- критерий покрытия оператора;
- критерий покрытия решений;
- критерий покрытия условий;
- критерий покрытия решений и условий;
- критерий комбинаторного покрытия условий.

На рассмотренных программах тесты, составленные по методу черного ящика показали себя лучше — метод позволил выявить больше несоответствий программ спецификации, чем метод белого ящика.

Список использованных источников

1. Майерс, Г. Искусство тестирования программ. - Санкт-Петербург: Диалектика, 2012. - С. 272.