

Федеральное государственное автономное образовательное
учреждение высшего образования
Национальный исследовательский университет ИТМО

Факультет программной инженерии и компьютерной техники
Направление подготовки 09.03.04 Программная инженерия
Дисциплина «Тестирование программного обеспечения»

Отчёт
По лабораторной работе №1

Вариант: 5199

Студент:

Барсуков М. А.

группа *P3315*

Преподаватель:

Цона Е. А.

г. Санкт-Петербург, 2025 г.

Описание задания

1. Для указанной функции провести модульное тестирование разложения функции в степенной ряд. Выбрать достаточное тестовое покрытие.
2. Провести модульное тестирование указанного алгоритма. Для этого выбрать характерные точки внутри алгоритма, и для предложенных самостоятельно наборов исходных данных записать последовательность попадания в характерные точки. Сравнить последовательность попадания с эталонной.
3. Сформировать доменную модель для заданного текста. Разработать тестовое покрытие для данной доменной модели

Вариант 5199

1. Функция **$\arcsin(x)$**
2. Программный модуль для работы с **красно-черным деревом** (<http://www.cs.usfca.edu/~galles/visualization/RedBlack.html>)
3. Описание предметной области:

| |
|---|
| Триллиан в отчаянии схватила его за руку и потянула к двери, которую Форд и Зафод пытались открыть, но Артур был, как труп -- казалось, надвигающиеся воздухоплавающие грызуны загипнотизировали его. |
|---|

Выполнение

Исходный код



<https://github.com/maxbarsukov/itmo/tree/master/6%20тпо/лабораторные/lab1>

Задание 1: $\arcsin(x)$

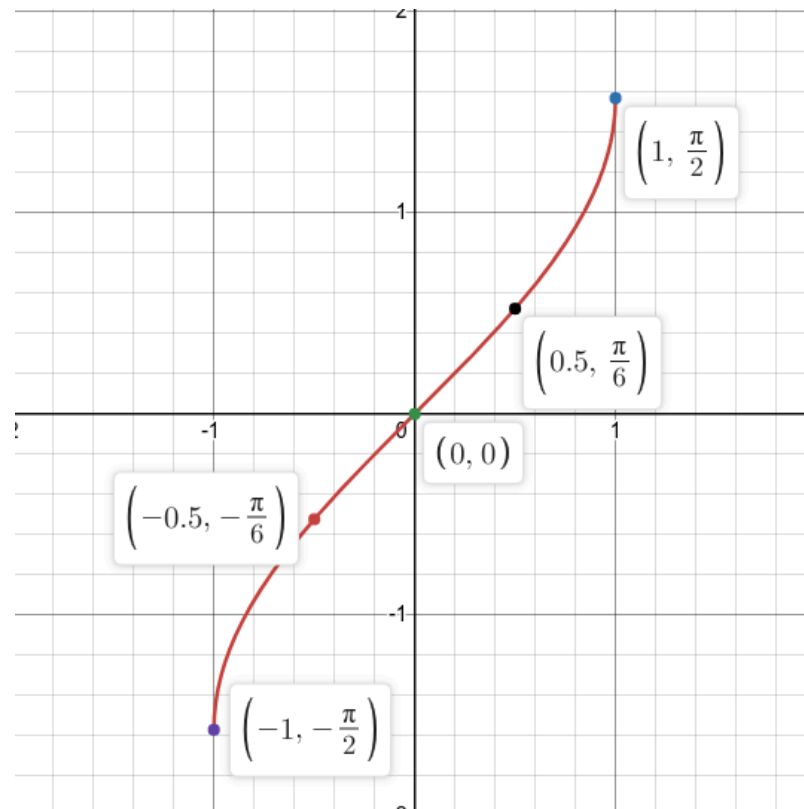


График функции $\arcsin(x)$ с табличными значениями в точках.

1. Выберем “интересные” для тестирования точки:

- **-1.0, 1.0** – граничные значения, вне которых функция не существует, в них значение должно верно определяться с достаточной точностью, несмотря на то, что в этих точках для разложения функции в степенной ряд быстро сходится и значения могут быть неточными;
- **-999, 999** – значения далеко от граничных, функция должна возвращать NaN, как эталонная реализация **Math.asin**;
- **-1.0000001, 1.0000001** – значения близкие к граничным, должны быть верно определены как не входящие в область определения.
- **-0.99, 0.99** – близкие к граничным точки, должны верно определяться несмотря на быструю сходимость в окрестностях граничных точек;
- **-0.5, 0.5** – стандартные табличные значения далеко от угловых случаев;
- **-0.000001, 0.000001** – близкие к **0** точки, должны считаться верные значения, а не 0;
- **-0.0** – должно определяться как **-0** ровно, т.е. знак сохраняется (как в эталонной реализации);

- **0.0** – должно определяться как **0** ровно, а не какое-то приближение (как в эталонной реализации);
- **Double.NaN** – должно определяться как **NaN** (как в эталонной реализации);
- **Double.POSITIVE_INFINITY** – должно определяться как **NaN** (так как выход за граничные значения);
- **Double.MIN_VALUE** – должно определяться как малое double число, а не как 0.

Таблица тестовых данных для “интересных” точек:

| x | arcsin(x) |
|--------------------------|--|
| -1.0 | $-\pi/2$ |
| -0.99 | -1.5608 |
| -0.5 | -0.5236 |
| -0.000001 | -0.000001 |
| 0.0 | 0.0 |
| 0.000001 | 0.000001 |
| 0.5 | 0.5236 |
| 0.99 | 1.5608 |
| 1.0 | $\pi/2$ |
| -999 | NaN |
| 999 | NaN |
| -1.0000001 | NaN |
| 1.0000001 | NaN |
| Double.NaN | NaN |
| Double.POSITIVE_INFINITY | NaN |
| Double.MIN_VALUE | 0.0000000000 000002220446 049250313080 847263336181 640625 |

- Кроме этих “особенных” точек, будем также тестировать значения в промежутке $[-1, 1]$ с разрывом 0.1, т.е. **(-1.0, -0.9, ... 0.9, 1.0)** и сравнивать значения нашей функцией с эталонной реализацией, чтобы проверять правильность поведения функции на всем протяжении.
- Также будем проводить **property-based/fuzzy** тестирование. Выберем промежуток для тестирования **[-0.999, 0.999]** (числа вне промежутка можем не проверять, так как в п.1 граничные условия проверены), количество прогонов 1 000 000 (достаточно для покрытия большого числа различных точек промежутка, и тесты выполняются за адекватное время).

Test Summary

39
tests

0
failures

0
ignored

0.678s
duration

100%

successful

Packages

Classes

| Package | Tests | Failures | Ignored | Duration | Success rate |
|------------------|-------|----------|---------|----------|--------------|
| lab1.task1.utils | 39 | 0 | 0 | 0.678s | 100% |

Тесты успешно проходятся.

task1 > lab1.task1.utils

lab1.task1.utils

| Element | Missed Instructions | Cov. | Missed Branches | Cov. | Missed | Cxty | Missed | Lines | Missed | Methods | Missed | Classes |
|---------------|---------------------|------|-----------------|------|--------|------|--------|-------|--------|---------|--------|---------|
| Trigonometric | <div></div> | 97 % | <div></div> | 92 % | 2 | 10 | 1 | 20 | 1 | 3 | 0 | 1 |
| Total | 3 of 127 | 97 % | 1 of 14 | 92 % | 2 | 10 | 1 | 20 | 1 | 3 | 0 | 1 |

Итоговое тестовое покрытие.

Функция **arcsin(x)** корректно обрабатывает все тестовые случаи, включая граничные и специальные значения. Все результаты совпадают с эталонной реализацией **Math.asin**.

Функция правильно возвращает **NaN** для значений, выходящих за пределы допустимого диапазона, и сохраняет знак для нулевых значений.

Быстрая сходимость функции в окрестностях граничных значений не привела к ошибкам в вычислениях, что указывает на стабильность реализации.

Задание 2: Красно-чёрное дерево

Эталонная реализация по варианту:

<http://www.cs.usfca.edu/~galles/visualization/RedBlack.html>

Красно-чёрное дерево — это самобалансирующееся бинарное дерево поиска, которое гарантирует логарифмическую сложность операций вставки, удаления и поиска. Для обеспечения корректности работы дерева необходимо тщательно протестировать его основные операции:

- Вставка элемента.
- Удаление элемента.
- Поиск элемента.
- Проверка свойств красно-черного дерева (инвариантов).

Цель тестирования — убедиться, что дерево корректно выполняет все операции и сохраняет свои свойства после каждой операции.

Будем проверять нашу реализацию структуры данных red-black tree следующим образом:

- для каждого публичного метода структуры (добавление/удаление/поиск элемента – интерфейс совпадает с эталонной реализацией) попытаемся выделить основные сценарии работы метода (то есть возможные ветвления для метода);
- проверять приватные методы, мы будем как “белый ящик” – будем проверять их соответствие эталонным данным через вызов публичных методов с заданными исходными данными, которые обеспечат выполнение проверяемого участка кода/метода.

Для тестирования красно-черного дерева используется комбинация подходов:

1. Покрытие кода (Code Coverage):
 - Убедиться, что все строки кода (включая ветви условий и циклов) выполнены хотя бы один раз.
 - Особое внимание уделяется сложным методам, таким как балансировка после вставки и удаления.
 - Покрытие граничных значений (Boundary Value Analysis):
2. Тестирование вставки и удаления на граничных случаях:
 - Вставка в пустое дерево.
 - Удаление единственного элемента.
 - Вставка и удаление элементов в порядке возрастания и убывания.
3. Покрытие свойств (Property-Based Testing):

- Проверка инвариантов красно-черного дерева после каждой операции:
 - Корень дерева всегда чёрный.
 - Нет двух последовательных красных узлов.
 - Все пути от корня до листьев содержат одинаковое количество чёрных узлов (чёрная высота).
4. Покрывание сценариев использования (Scenario-Based Testing):
- Тестирование типичных сценариев использования дерева:
 - Вставка случайных элементов.
 - Удаление случайных элементов.
 - Поиск элементов в дереве.

В ходе тестирования была обнаружена ошибка (из-за того, что результаты выполнения для эталонной и нашей реализации отличались):

- Ошибка балансировки при удалении:
 - При удалении элемента в определённых случаях нарушалось свойство чёрной высоты.
 - Исправлено путём добавления дополнительных проверок в метод delete.

Как мы можем видеть, все работает корректно:

Test Summary

41

0

0

0.067s

100%

successful

tests

failures

ignored

duration

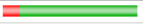
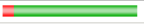








































Packages

Classes

| Package | Tests | Failures | Ignored | Duration | Success rate |
|---------------------------|-------|----------|---------|----------|--------------|
| lab1.task2.datastructures | 41 | 0 | 0 | 0.067s | 100% |

Тесты успешно проходятся.

RedBlackTree

| Element | Missed Instructions | Cov. | Missed Branches | Cov. | Missed Cxty | Missed Lines | Missed Methods |
|---|---|-------|--|-------|-------------|--------------|----------------|
| fixDelete(RedBlackTree.Node) |  | 86 % |  | 90 % | 2 12 | 6 44 | 0 1 |
| fixInsert(RedBlackTree.Node) |  | 90 % |  | 83 % | 3 10 | 4 28 | 0 1 |
| deleteNodeHelper(RedBlackTree.Node, int) |  | 92 % |  | 93 % | 1 9 | 2 32 | 0 1 |
| rightRotate(RedBlackTree.Node) |  | 91 % |  | 83 % | 1 4 | 1 13 | 0 1 |
| hasEqualBlackHeight(RedBlackTree.Node) |  | 97 % |  | 75 % | 2 5 | 0 7 | 0 1 |
| insert(int) |  | 100 % |  | 100 % | 0 7 | 0 24 | 0 1 |
| leftRotate(RedBlackTree.Node) |  | 100 % |  | 100 % | 0 4 | 0 13 | 0 1 |
| printTreeHelper(RedBlackTree.Node, String, boolean) |  | 100 % |  | 100 % | 0 4 | 0 11 | 0 1 |
| isValidBSTHelper(RedBlackTree.Node, int, int) |  | 100 % |  | 80 % | 2 6 | 0 5 | 0 1 |
| isProperlyColored(RedBlackTree.Node) |  | 100 % |  | 83 % | 2 7 | 0 6 | 0 1 |
| isValidRedBlackTree() |  | 100 % |  | 100 % | 0 5 | 0 9 | 0 1 |
| searchTreeHelper(RedBlackTree.Node, int) |  | 100 % |  | 100 % | 0 4 | 0 5 | 0 1 |
| rbTransplant(RedBlackTree.Node, RedBlackTree.Node) |  | 100 % |  | 100 % | 0 3 | 0 7 | 0 1 |
| countBlackHeight(RedBlackTree.Node) |  | 100 % |  | 100 % | 0 3 | 0 5 | 0 1 |
| inOrderHelper(List, RedBlackTree.Node) |  | 100 % |  | 100 % | 0 2 | 0 5 | 0 1 |
| countNodesHelper(RedBlackTree.Node) |  | 100 % |  | 100 % | 0 2 | 0 3 | 0 1 |
| RedBlackTree() |  | 100 % | n/a | n/a | 0 1 | 0 5 | 0 1 |
| inOrder() |  | 100 % | n/a | n/a | 0 1 | 0 3 | 0 1 |
| minimum(RedBlackTree.Node) |  | 100 % |  | 100 % | 0 2 | 0 3 | 0 1 |
| printTree() |  | 100 % | n/a | n/a | 0 1 | 0 2 | 0 1 |
| search(int) |  | 100 % | n/a | n/a | 0 1 | 0 1 | 0 1 |
| delete(int) |  | 100 % | n/a | n/a | 0 1 | 0 2 | 0 1 |
| isValidBST(RedBlackTree.Node) |  | 100 % | n/a | n/a | 0 1 | 0 1 | 0 1 |
| countNodes() |  | 100 % | n/a | n/a | 0 1 | 0 1 | 0 1 |
| getRoot() |  | 100 % | n/a | n/a | 0 1 | 0 1 | 0 1 |
| Total | 48 of 953 | 94 % | 13 of 144 | 90 % | 13 97 | 13 236 | 0 25 |

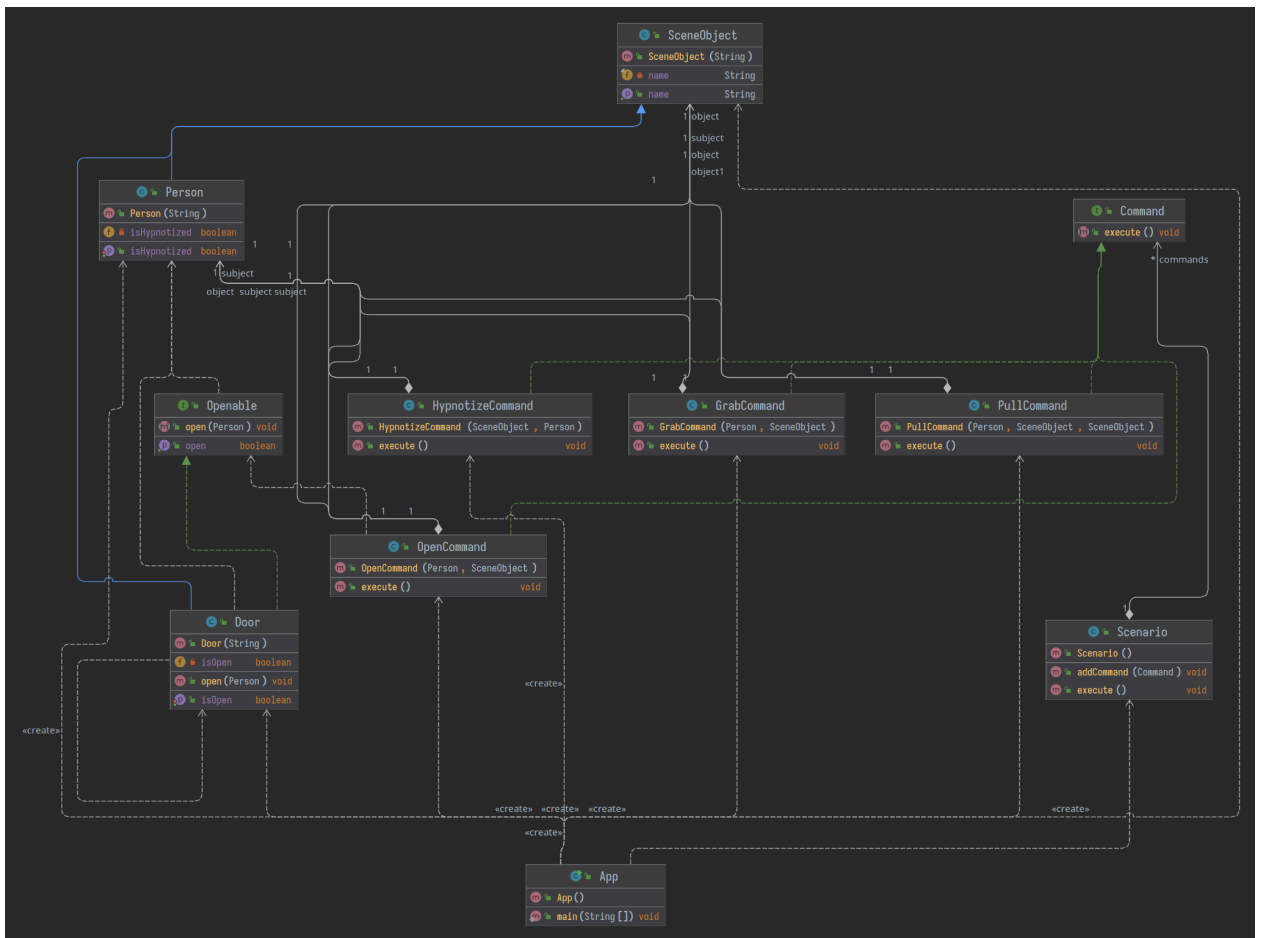
Итоговое тестовое покрытие методов класса RedBlackTree.

Результаты тестирования:

- Красно-черное дерево реализовано корректно и проходит все тесты.
- Код покрыт модульными тестами.
- Тестирование подтвердило, что дерево корректно выполняет все основные операции и сохраняет свои свойства.

Задание 3: Описание предметной области

Для начала, спроектируем нашу доменную область в соответствии с заданным текстом.



Спроектированная UML диаграмма предметной области.

Предметная область включает в себя:

- Персонажи (например, Триллиан, Артур).
- Объекты (например, дверь, рука).
- Действия (например, схватить, потянуть, открыть).
- Сценарии (последовательности действий).

Далее, в соответствии с TDD напомним тесты по спроектированной модели.

Цель тестирования — убедиться, что:

- Все команды выполняются корректно.
- Состояния персонажей и объектов изменяются в соответствии с командами.
- Сценарии выполняются последовательно и без ошибок.

Составим следующие тесты:

Класс: **Person**

- *testPersonCreation*: Проверяет корректность создания объекта Person (имя и начальное состояние).
- *testSetHypnotized*: Проверяет, что метод setHypnotized корректно изменяет состояние персонажа.

Класс: **SceneObject**

- *testSceneObjectCreation*: Проверяет корректность создания объекта SceneObject (имя объекта).

Класс: **Door** (наследует SceneObject и реализует Openable)

- *testDoorCreation*: Проверяет корректность создания объекта Door (имя и начальное состояние).
- *testOpenDoor*: Проверяет, что метод open корректно открывает дверь.
- *testOpenAlreadyOpenDoor*: Проверяет, что повторный вызов метода open не изменяет состояние уже открытой двери.

Класс: **GrabCommand**

- *testGrabCommandExecution*: Проверяет, что команда GrabCommand корректно выполняет действие и выводит ожидаемое сообщение.

Класс: **PullCommand**

- *testPullCommandExecution*: Проверяет, что команда PullCommand корректно выполняет действие и выводит ожидаемое сообщение.

Класс: **OpenCommand**

- *testOpenCommandExecution*: Проверяет, что команда OpenCommand корректно открывает объект, реализующий интерфейс Openable, и выводит ожидаемое сообщение.
- *testOpenNonOpenableObject*: Проверяет, что команда OpenCommand корректно обрабатывает попытку открыть объект, который не реализует интерфейс Openable.

Класс: **HypnotizeCommand**

- *testHypnotizeCommandExecution*: Проверяет, что команда HypnotizeCommand корректно изменяет состояние персонажа и выводит ожидаемое сообщение.

Класс: **Scenario**

- *testScenarioExecution*: Проверяет, что сценарий корректно выполняет последовательность команд и выводит ожидаемые сообщения.

После написания тестов, реализуем собственно сами классы в соответствии с TDD.

Как мы можем видеть, все работает корректно:

Test Summary

| | | | | |
|-------|----------|---------|----------|------------|
| 12 | 0 | 0 | 0.048s | 100% |
| tests | failures | ignored | duration | successful |

Packages Classes

| Package | Tests | Failures | Ignored | Duration | Success rate |
|------------------------|-------|----------|---------|----------|--------------|
| lab1.task3.commands | 5 | 0 | 0 | 0.039s | 100% |
| lab1.task3.controllers | 1 | 0 | 0 | 0.002s | 100% |
| lab1.task3.models | 6 | 0 | 0 | 0.007s | 100% |

Тесты успешно проходятся.

| task3 | | | | | | | | | | | |
|------------------------|---------------------|-------|-----------------|-------|-------------|--------------|----------------|----------------|--|--|--|
| task3 | | | | | | | | | | | |
| Element | Missed Instructions | Cov. | Missed Branches | Cov. | Missed Cxty | Missed Lines | Missed Methods | Missed Classes | | | |
| lab1.task3 | | 0 % | | n/a | 3 3 | 16 16 | 3 3 | 2 2 | | | |
| lab1.task3.commands | | 100 % | | 100 % | 0 9 | 0 28 | 0 8 | 0 4 | | | |
| lab1.task3.models | | 100 % | | 100 % | 0 10 | 0 21 | 0 9 | 0 3 | | | |
| lab1.task3.controllers | | 100 % | | 100 % | 0 4 | 0 10 | 0 3 | 0 1 | | | |
| Total | 123 of 337 | 63 % | 0 of 6 | 100 % | 3 26 | 16 75 | 3 23 | 2 10 | | | |

Итоговое тестовое покрытие.

Выводы

Во время выполнения лабораторной работы я углубил свои знания в JUnit5, научился писать юнит-тесты, использовать параметризированные тесты и тесты на проверку составленной объектной модели.