

Факультет программной инженерии и компьютерной техники

Тестирование программного обеспечения

Лабораторная работа №1

Вариант №33138

Преподаватель: Харитонова Анастасия Евгеньевна

Выполнил: Кульбако Артемий Юрьевич Р33112

# Задание

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.

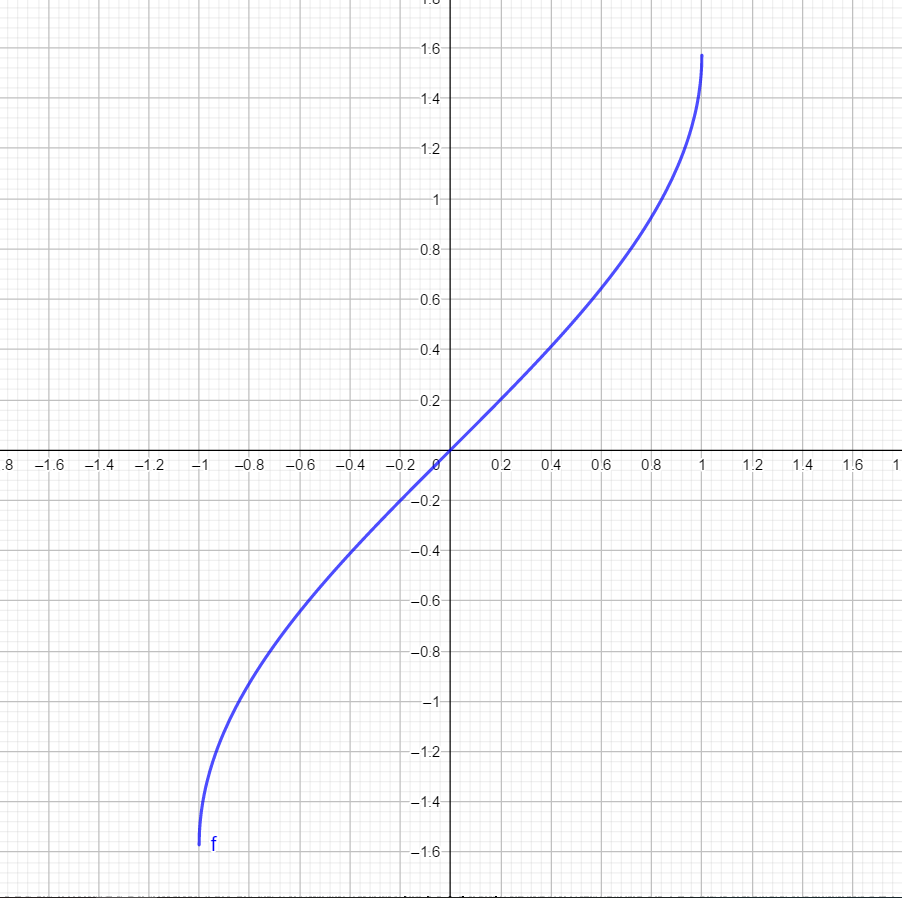
**Вариант 33138**

1. Функция
2. Программный модуль для сортировки массива по алгоритму быстрой сортировки http://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html
3. Описание предметной области:

Еще несколько тысяч лет могучие корабли разрывали огромные космические пространства, чтобы с ревом спикировать на первую планету, встретившуюся на их пути, -- ей оказалась Земля, -- где вследствие роковой ошибки при расчете масштабов весь боевой космический флот был случайно проглочен маленькой собачкой.

# Выполнение

## arcsin(x)

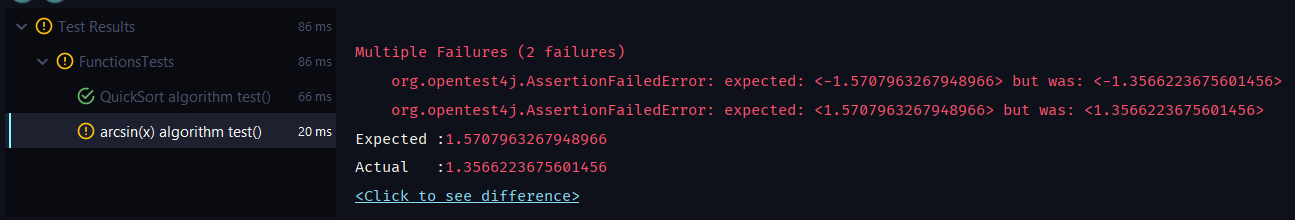
Значения, в которых необходимо проверить функцию: – граничные значения, где функция меняет знак, а также переходит в неопределённость. Для достоверности возьмём также пару значений из промежутка . Итоговая таблица для проверки:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| x | -1 | -0.5 | 0 | 0.5 | 1 |
| arcsin(x) | -PI / 2 | -0.5235 | 0 | 0.5235 | PI / 2 |

*fun* arcsin(x: Number): Double =  
 x.toDouble().let **{** *if* (abs(**it**) <= 1) {  
 *val* fact: (Int) -> Int = **{** (1..**it**).fold(1) **{** acc, i **->** acc \* i **} }** generateSequence(0) **{** n **->** n + 1 **}** .take(33)  
 .map **{** n **->** (fact(2 \* n) / (4.0.pow(n) \* fact(n).toDouble().pow(2) \* (2 \* n + 1))) \* **it**.pow(2 \* n + 1) **}** .sum()  
 } *else* Double.NaN **}**

Написанная функция использует, согласно варианту, разложение в степенной ряд (ряд Тейлора) для нахождения значений.

@Test *fun* `arcsin(x) algorithm test`() = assertAll(  
 **{** assertEquals(Double.NaN, arcsin(-1.1), 0.001) **}**,  
 **{** assertEquals(-*PI* / 2, arcsin(-1), 0.001) **}**,  
 **{** assertEquals(-0.5235987755982989, arcsin(-0.5), 0.001) **}**,  
 **{** assertEquals(0.0, arcsin(0), 0.001) **}**,  
 **{** assertEquals(0.5235987755982989, arcsin(0.5), 0.001) **}**,  
 **{** assertEquals(*PI* / 2, arcsin(1), 0.001) **}**,  
 **{** assertEquals(Double.NaN, arcsin(1.1), 0.001) **}**)



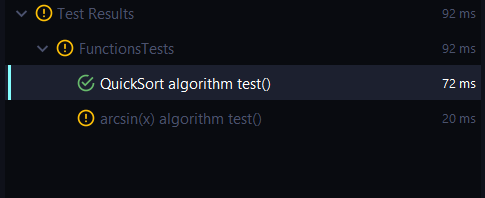
Функция не проходит на двух значениях. Ряд Тейлора (степенной) - плохой способ нахождения , потому что его ошибка увеличивается по мере удаления от аргумента, вокруг которой находится ряд. Нормальная реализация этой функции используют или алгоритмы аппроксимации. Поэтому тесты на граничных значениях не проходят.

## quickSort

*fun* <T: *Comparable*<T>> *List*<T>.quickSort(): *List*<T> =  
 *if* (size >= 2) {  
 *val* pivot = first()  
 *val* (smaller, greater) = drop(1).partition **{ it** <= pivot **}** smaller.quickSort() + pivot + greater.quickSort()  
 } *else this*

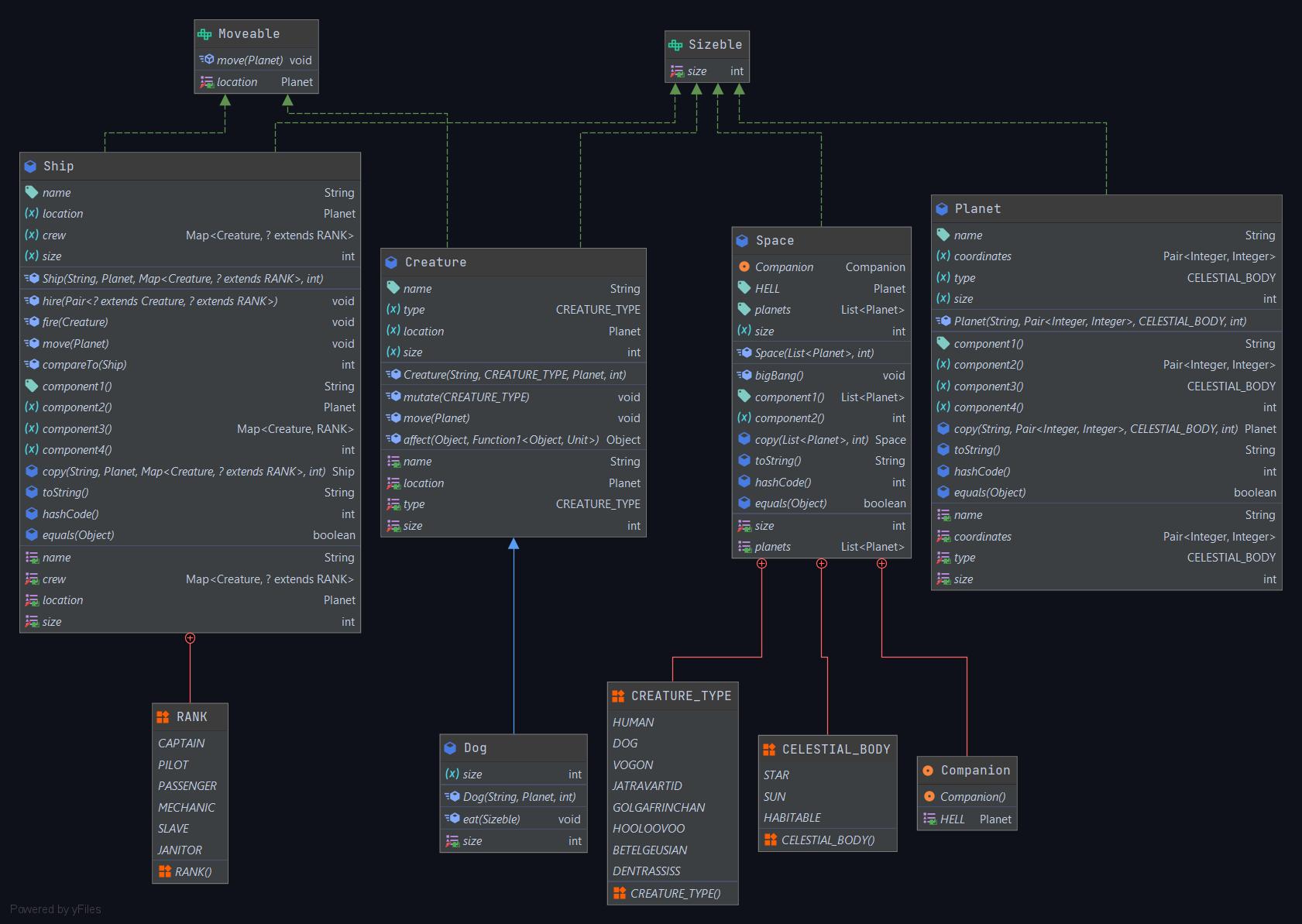
Алгоритм быстрой сортировки проверим двумя способами: вручную отсортированным массивом (с отрицательными, а также повторяющимися числами), а также алгоритмом сортировки пузырьком (в котором якобы сложнее совершить ошибку).

*private fun* <T: *Comparable*<T>> *List*<T>.bubbleSort(): *List*<T> =  
 *this*.toMutableList().apply **{** *for* (i *in* 0..*this*.size - 2)  
 *for* (j *in* 0..*this*.size - i - 2)  
 *if* (*this*[j] > *this*[j + 1])  
 *this*[j] = *this*[j + 1].also **{** *this*[j + 1] = *this*[j] **} }**@Test *fun* `QuickSort algorithm test`() = assertAll(  
 **{** List(50) **{** Random.nextInt() **}**.let **{** assertEquals(**it**.bubbleSort(), **it**.quickSort(), "😭") **} }**,  
 **{** assertEquals(listOf(-2, -2, 5, 7, 10, 15, 23 ,45), listOf(10, 23, 45, 15, 5, -2, -2, 7).quickSort()) **}**



Алгоритм работает абсолютно корректно.

## Unit-тестирование объектов



### Код

*package* world  
  
*/\*\*  
 \* Allow object to move from* ***[Planet]*** *to* ***[Planet]*** *\*/  
interface Moveable* {  
  
 *var* location: Planet  
 *infix fun* move(dist: Planet)  
}  
  
  
*interface Sizeble* { *var* size: Int }

*package* world  
  
*/\*\*  
 \* Represents the living* ***[Creature]*** *in* ***[Space]*** *\*/  
open class* Creature(*val* name: String,  
 *var* type: Space.CREATURE\_TYPE,  
 *override var* location: Planet,  
 *override var* size: Int = 0): *Moveable*, *Sizeble* {  
  
 */\*\*  
 \* Change* ***[Creature.type]*** *\*/  
 infix fun* mutate(t: Space.CREATURE\_TYPE) { *this*.type = t }  
  
 */\*\*  
 \* Move* ***[Creature]*** *to another* ***[Planet]****. Without* ***[Ship]*** *he can move only to* ***[Space.CELESTIAL\_BODY.HABITABLE]*** *\*/  
 override infix fun* move(dist: Planet) { *if* (dist.type == Space.CELESTIAL\_BODY.*HABITABLE*) *this*.location = dist }  
  
 */\*\*  
 \* Universal action for affecting* ***[Creature]*** *on every possible object  
 \*/  
 fun* affect(obj: Any, f: (Any) -> Unit): Any { *return* f(obj) }  
}

*package* world  
  
*class* Dog(name: String,  
 location: Planet,  
 *override var* size: Int): Creature(name, Space.CREATURE\_TYPE.*DOG*, location, size) {  
  
 *fun* eat(obj: *Sizeble*) {  
 affect(obj) **{** println("Eating...")  
 *if* (obj *is Moveable*)  
 *if* (obj.size > *this*.size) *throw* Exception("Too big!") *else* obj.location = Space.HELL  
 *else throw* Exception("Dog is not god")  
 *return*@affect Unit  
 **}** }  
}

*package* world  
  
*/\*\*  
 \* Represents planet in* ***[Space]*** *\*/  
data class* Planet(*val* name: String,  
 *var* coordinates: Pair<Int, Int>,  
 *var* type: Space.CELESTIAL\_BODY, *override var* size: Int = 0): *Sizeble*

*package* world  
  
*/\*\*  
 \* Space ship for moving between planets  
 \*/  
data class* Ship(*val* name: String,  
 *override var* location: Planet,  
 *var* crew: *Map*<Creature, RANK> = mutableMapOf(),  
 *override var* size: Int = 0): *Moveable*, *Comparable*<Ship>, *Sizeble* {  
  
 */\*\*  
 \* Add* ***[Creature]*** *to* ***[Ship.crew]*** *on* ***[RANK]*** *\*/  
 infix fun* hire(candidate: Pair<Creature, RANK>) { *this*.crew = *this*.crew + candidate }  
  
 */\*\*  
 \* Remove* ***[Creature]*** *from* ***[Ship.crew]*** *\*/  
 infix fun* fire(c: Creature) { *this*.crew = crew - c }  
  
 */\*\*  
 \* Move* ***[Ship]*** *and all it's* ***[Creature]*** *to another* ***[Planet]*** *\*/  
 override infix fun* move(dist: Planet) {  
 *this*.location = dist  
 crew.forEach **{ it**.key.move(dist) **}** }  
  
 *override infix fun* compareTo(other: Ship) = *this*.crew.size - other.crew.size  
  
 */\*\*  
 \* Available positions on ship  
 \*/  
 enum class* RANK { *CAPTAIN*, *PILOT*, *PASSENGER*, *MECHANIC*, *SLAVE*, *JANITOR* }  
}

*package* world  
  
*/\*\*  
 \* Contains* ***[Planet]*** *and types of existed object in it  
 \*/  
data class* Space(*val* planets: *MutableList*<Planet>,  
 *override var* size: Int = Int.MAX\_VALUE): *Sizeble* {  
  
 *companion object* { *val* HELL = Planet("Hell", -1 to -1, Space.CELESTIAL\_BODY.*SUN*) }  
  
 */\*\*  
 \* Destroy all planets  
 \*/  
 fun* bigBang() = planets.clear()  
  
 */\*\*  
 \* Types of* ***[Planet]*** *\*/  
 enum class* CELESTIAL\_BODY { *STAR*, *SUN*, *HABITABLE* }  
  
 */\*\*  
 \* Types of* ***[Creature]*** *\*/  
 enum class* CREATURE\_TYPE { *HUMAN*, *DOG*, *VOGON*, *JATRAVARTID*, *GOLGAFRINCHAN*, *HOOLOOVOO*, *BETELGEUSIAN*, *DENTRASSISS* }  
}

*import* world.\*  
*import* org.junit.jupiter.api.\*  
*import* kotlin.test.\*  
*import* org.junit.jupiter.api.Test  
  
*private fun* createPlanet() = Planet("test\_planet", 1 to 1, Space.CELESTIAL\_BODY.*HABITABLE*)  
  
*private fun* createTester() = Creature("tester", Space.CREATURE\_TYPE.*HUMAN*, createPlanet())  
  
*private fun* createShip() = Ship("test\_ship", createPlanet(), emptyMap())  
  
*/\*\*  
 \* Tests for* ***[Ship]*** *\*/*@TestInstance(TestInstance.Lifecycle.*PER\_CLASS*)  
*class* ShipTest {  
  
 @Test *fun* `Test hire creature to the ship's crew`() {  
 *val* tester = createTester()  
 *val* testShip = createShip()  
 testShip hire (tester to Ship.RANK.*SLAVE*)  
 assert(testShip.crew.contains(tester))  
 }  
  
 @Test *fun* `Test fire creature from ship's crew`() {  
 *val* tester = createTester()  
 *val* testShip = createShip()  
 testShip fire tester  
 assert(testShip.crew.contains(tester).not())  
 }  
  
 @Test *fun* `Test compare two ship's`() {  
 *val* testPlanet = createPlanet()  
 *val* ship1 = Ship("ship1", testPlanet, mapOf(  
 Creature("t1", Space.CREATURE\_TYPE.*HOOLOOVOO*, testPlanet) to Ship.RANK.*CAPTAIN*,  
 Creature("t2", Space.CREATURE\_TYPE.*BETELGEUSIAN*, testPlanet) to Ship.RANK.*PASSENGER*))  
 *val* ship2 = Ship("ship2", createPlanet(), mapOf(  
 Creature("t1", Space.CREATURE\_TYPE.*HOOLOOVOO*, testPlanet) to Ship.RANK.*CAPTAIN*,  
 Creature("t2", Space.CREATURE\_TYPE.*BETELGEUSIAN*, testPlanet) to Ship.RANK.*PASSENGER*,  
 Creature("te", Space.CREATURE\_TYPE.*DENTRASSISS*, testPlanet) to Ship.RANK.*MECHANIC*))  
 assertEquals(-1, ship1 compareTo ship2)  
 }  
 @Test @AfterAll  
 *fun* `Test move all ship's creatures`() {  
 *val* originPlanet = createPlanet()  
 *val* distinationPlanet = Planet("dist\_planet", 1 to 1, Space.CELESTIAL\_BODY.*HABITABLE*)  
 *val* testShip = Ship("test\_ship", originPlanet, mapOf(  
 Creature("t1", Space.CREATURE\_TYPE.*HOOLOOVOO*, originPlanet) to Ship.RANK.*JANITOR*,  
 Creature("t2", Space.CREATURE\_TYPE.*BETELGEUSIAN*, originPlanet) to Ship.RANK.*PILOT* ))  
 testShip move distinationPlanet  
 assert(testShip.crew.map **{ it**.key.location **}**.all **{ it** == distinationPlanet **}**)  
 }  
  
 @Test *fun* `Move to current planet`() {  
 *val* earth = Planet("earth", 1 to 1, Space.CELESTIAL\_BODY.*HABITABLE*)  
 *val* testShip = Ship("ship", earth, emptyMap())  
 testShip.move(earth)  
 assertEquals(earth, testShip.location)  
 }  
}  
  
*/\*\*  
 \* Tests for* ***[Creature]*** *\*/  
class* CreatureTest {  
  
 @Test *fun* `Test creature mutation process`() {  
 *val* tester = createTester()  
 *val* newType = Space.CREATURE\_TYPE.*DOG* tester mutate newType  
 assertEquals(newType, tester.type, "After mutate type changes incorrect")  
 }  
  
 @Test *fun* `Test move creature to planets`() {  
 *val* tester = createTester()  
 assertAll(  
 **{** *val* prohibitedPlanet = Planet("prohibited", 2 to 1, Space.CELESTIAL\_BODY.*SUN*)  
 tester move prohibitedPlanet  
 assertNotEquals(prohibitedPlanet, tester.location, "The creature unexpectedly moved to an uninhabited planet without technology") **}**,  
 **{** *val* permittedPlanet = Planet("permitted", 2 to 2, Space.CELESTIAL\_BODY.*HABITABLE*)  
 tester move permittedPlanet  
 assertEquals(permittedPlanet, tester.location, "Tester unexpectedly didn't move to permitted planet")  
 **}** )  
 }  
  
 @Test *fun* `Test affect creature process to another object`() {  
 *val* tester = createTester()  
 *val* testObject = createPlanet()  
 tester.affect(testObject) **{** (**it** *as* Planet).coordinates = -1 to -1 **}** assertEquals(-1 to -1, testObject.coordinates, "Creature correctly affect on another object")  
 }  
  
 @Test *fun* `Test dog can eat`() {  
 *val* victim = createShip().apply **{** size = 200 **}** assertAll(  
 **{** *val* bigDog = Dog("big\_dog", createPlanet(), 10000)  
 bigDog.eat(victim)  
 assertEquals(Space.HELL, victim.location)  
 **}**,  
 **{** *val* littleDog = Dog("little\_dog", createPlanet(), 5)  
 assertThrows<Exception> **{** littleDog.eat(victim) **}  
 }** )  
 }  
}  
  
  
*/\*\*  
 \* Tests for* ***[Space]*** *\*/  
class* SpaceTest {  
  
 @Test *fun* `Test explode space`() {  
 *val* planets = mutableListOf(createPlanet(), createPlanet(), createPlanet())  
 *val* testSpace = Space(planets)  
 testSpace.bigBang()  
 assert(testSpace.planets.isEmpty())  
 }  
}

# Вывод

В процессе выполнения лабораторной работы я научился писал юнит-тесты (метод чёрного ящика) для разработанных классов. Сложность заключается в необходимости проявить гибкость мышления при проверке ожидаемого поведения, т.е. придумать альтернативный способ достижения результата, либо вручную формировать как исходные, так и ожидаемые данные для сравнения. Важно отметить, что достижение 100%-го покрытия практически невозможно, поэтому необходимо проверять лишь «избранные» входные данные.