Практическое задание №4

Для тестирования использовалась библиотека JQF. Данная библиотека позволяет генерировать собственные входные данные, а также случайные. В данной работе тестовое покрытие измерялось при помощи инструмента JaCoCo.

Lesson 5

- B lesson5 проводилось тестирование 2 функций, а именно extractRepeats() и findSumOfTwo().
 - Из рисунка ниже можно увидеть результаты тестирования при генерации случайных данных. Причина таких результатов связанна с тем, что диапазон генерируемых данных очень велик, из-за чего вероятность того, что мы получим одинаковые элементы, или сумму двух случайных чисел из коллекции довольно мал. Собственно, это мы и видим и рисунка 2.

Element	ф	Missed Instructions +	Cov.	Missed Branches	\$	Cov. \$	Missed	Cxty	Missed	Lines	Missed	Methods \$
findSumOfTwo(List, int)			82 %			75 %	1	3	2	9	0	1
extractRepeats(List)			76 %		-	58 %	4	7	4	14	0	1

Рис 1. Результаты тестов со случайно сгенерированными данными

```
fun extractRepeats(list: List<String>): Map<String, Int> { |fun findSumOfTwo(list: List<Int>, number: Int): Pair<Int, Int> {
    val result = mutableMapOf<String, Int>()
                                                                        var result = Pair(-1, -1
    val names = hashSetOf<String>()
                                                                        val midRes = mutableMapOf<Int, Int>()
                                                                        for (i in list.indices)
    for (element in list) {
        if (result.containsKey(element)) {
                                                                            val rest = number -
                                                                                                  list[i]
                                                                            if (midRes.containsKey(list[i])) {
   result = Pair(midRes[list[i]], i) as Pair<Int, Int>
             var count = result[element]
if (count != null) {
                 count += 1
                                                                            } else midRes[rest] = i
                 result[element] = count
        } else {
                                                                        return result
            result[element] = 1
        names.add(element)
    for (element in names) {
        if (result[element] == 1)
    result.remove(element)
    return result
```

Рис 2. Покрываемый тестами код

• Решить данную проблему удалось, немного обработав сгенерированные случайные данные. В случае с extractRepeats() производилось дублирование случайных элементов исходной коллекции, а для findSumOfTwo() брались элементы из сгенерированной коллекции и находилась их сумма. Несмотря на это, не удалось добиться полного покрытия. Связанно это с тем, что одна из веток никогда не будет ложной "if (count!= null)"

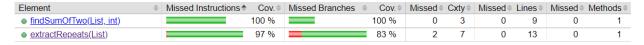


Рис 3. Результаты тестов со структурированными данными

```
fun extractRepeats(list: List<String>): Map<String, Int> { fun findSumOfTwo(list: List<Int>, number: Int): Pair<Int, Int> {
                                                                     var result = Pair(-1, -1)
val midRes = mutableMapOf<Int, Int>()
    val result = mutableMapOf<String, Int>()
val names = hashSetOf<String>()
                                                                     for (i in list.indices)
    for (element in list)
        if (result.containsKey(element)) {
                                                                          val rest = number -
                                                                                               list[i]
                                                                          if (midRes.containsKey(list[i]))
             var count = result[element]
                                                                              result = Pair(midRes[list[i]], i) as Pair<Int, Int>
             if (count != null) {
                                                                              break
                 count +=
                 result[element] = count
                                                                          } else midRes[rest] = i
                                                                      return result
             result[element] = 1
        names.add(element)
    for (element in names) {
        if (result[element] == 1) result.remove(element)
    return result
```

Рис 4. Покрываемый тестами код

Lesson 6

B lesson6 проводилось тестирование функций bestHighJump() и mostExpensive ().

• Случайное тестирование в данном случае показало себя очень плохо. Основной причиной стало то, что на вход оба метода ожидают данные, которые должны иметь определенную структуру. Поэтому оба теста в основном заканчивались на моменте проверки корректности формата входных данных.



Рис 5. Результаты тестов со случайными данными

```
val parts = jumps.split(" ")
var result = -1
jf (:...
fun mostExpensive(description: String): String {
                                                                     fun bestHighJump(jumps:
    var result
var max = 0
                                                                          if (jumps.contains(Regex("""[^-+%0-9\s\d]"""))) {
    if (description.isNotEmptv())
         for (i in goods) {
   val product = i.split(" ")
                                                                               result = -1
                                                                          } else {
    for (i in 0..parts.size - 2) {
                   (product[1].toDouble() v= null) {
  if (product[1].toDouble() >= max) {
    result = product[0]
                                                                                    if (parts.size >= 2) {
    if (parts[i].toIntOrNull() != null && parts[i + 1].contains(Regex("\\+"))) {
                                                                                              if (parts[i].toInt() > result) result = parts[i].toInt()
                         max = product[1].toDouble()
             } else break
                                                                               }
                                                                          return result
     return result
```

Рис 6. Покрываемый тестами код

Так как нам заранее известен формат, который ожидает на вход метод, мы можем генерировать отдельные его части. В результате этого удалось добиться большего покрытия кода. Однако, добиться полного покрытия не удалось из-за метода bestHighJump(), который имел не нужную проверку "if (parts.size >= 2)" (всегда будет больше или равно, иначе не попадем в тело цикла).

Element	Missed Instructions #	Cov.	Missed Branches		Missed	Cxty	Missed	Lines	Missed	Methods
bestHighJump(String)		100 %		92 %	1	8	0	9	0	1
mostExpensive(String)		100 %		100 %	0	6	0	12	0	1

Рис 7. Результаты тестов со структурированными данными

```
fun mostExpensive(description: String): String {
                                                                              fun bestHighJump(jumps: String): Int {
                                                                                   val parts = jumps.split("
var result = -1
      /ar result
     var max = 0.0
if (description.isNotEmpty())
                                                                                   if (jumps.contains(Regex("""[^-+%0-9\s\d]"""))) {
           description.iswotempty()) {
  val goods = description.split("; ")
  for (i in goods) {
     val product = i.split(" ")
                                                                                         result = -1
                                                                                   if (product[1].toDouble() >= muxl) {
   if (product[1].toDouble() >= max) {
                                                                                               if (parts.size >= 2) {
   if (parts[i].toIntOrNull() != null && parts[i + 1].contains(Regex("\\+"))) {
     if (parts[i].toInt() > result) result = parts[i].toInt()
                            result = product[0]
max = product[1].toDouble()
                } else break
                                                                                         }
                                                                                   return result
     return result
```

Рис 8. Покрываемый тестами код

Lesson 7

B lesson7 проводилось тестирование функции, а именно printMultiplicationProcess().

• Первое, что хотелось бы отметить, это специфика самого метода. Он принимает на вход переменные и записывает в файл их умножение в столбик. Основная проблема заключается в том, что мы не можем заранее подготовить результаты для данного теста, так как все данные генерируются случайно. Если же мы решим составить умножение в столбик исходя из сгенерированных чисел, то нам понадобится реализовать функцию, делающую это, а мы её уже тестируем. К сожалению, придумать подхода для тестирования на случайных данных подобной функции не удалось. Поэтому ниже представлены только результаты тестирования на случайных данных. Случайным данным не удалось покрыть случай нулевого множителя и множимого.

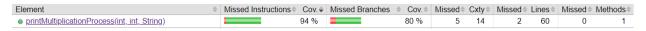


Рис 9. Результаты тестов со случайными данными

Вывод

Первое, что хотелось бы отметить, это огромная мощность фазинга как вида тестирования. Пока сам не прикоснешься не поймешь всю суть. Фазинг частично решает проблему невозможности тестирования собственного кода (ломать собственный код — это сложно). Также, фазинг явно является мощным инструментов для поиска багов в приложении (случаи, когда из всех наборов данных тест ломается только на числах 5 и 154867). Уменьшение влияния человеческого фактора на итоговый результат это в основном и круто.