



# kountable

Мутационное тестирование для Java и Scala на примере pitest

# Мутационное тестирование

- Метод тестирования, который создает небольшие изменения кода программы;
- Было предложено в 1971 году;
- Первая реализация появилась в 1980 году;



# Мутационное тестирование. Зачем?



Допустим это ваш сервис:

```
object A{  
  def apply(x: Boolean, y: Boolean): Int = {  
    if(x && y) 2  
    else if(x || y) 1  
    else 0  
  }  
}
```

Он умеет считать сколько аргументов равно true.

# Как узнать что нужно добавить тесты?



Вариант 1: У вас нет тестов;

Решение: `class MyTest{ def test(): Unit = assert(true) }`

Результат: Все тесты проходят, но смысла ноль.

# Как узнать что нужно добавить тесты?



Вариант 2: Тесты должны работать с кодом который тестируется;

Решение: `class MyTest{ def test(): Unit = A.apply(true, true) > 0 }`

Результат: Все тесты проходят, но покрытие кода низкое.

# Как узнать что нужно добавить тесты?



Вариант 3: Тесты должны покрывать > X% кода;

Решение: `class` MyTest{

```
def test(): Unit = {  
    A(true, true) > 0  
    A(true, false) > 0  
    A(false, false) == 0  
}}
```

Результат: Все тесты проходят, покрытие 100%...

# Как узнать что нужно добавить тесты?



Вариант 4: Тесты должны покрывать > X% кода и делать assert-ы;

Решение: `class` MyTest{

```
def test(): Unit = {  
    assert(A(true, true) > 0)  
    assert(A(true, false) > 0)  
    assert(A(false, false) == 0)  
}
```

# Мутация кода

- Мутируем функцию раз

```
def apply(x: Boolean, y: Boolean): Int =
```

```
  if(false) 2
```

```
  else if (x || y) 1
```

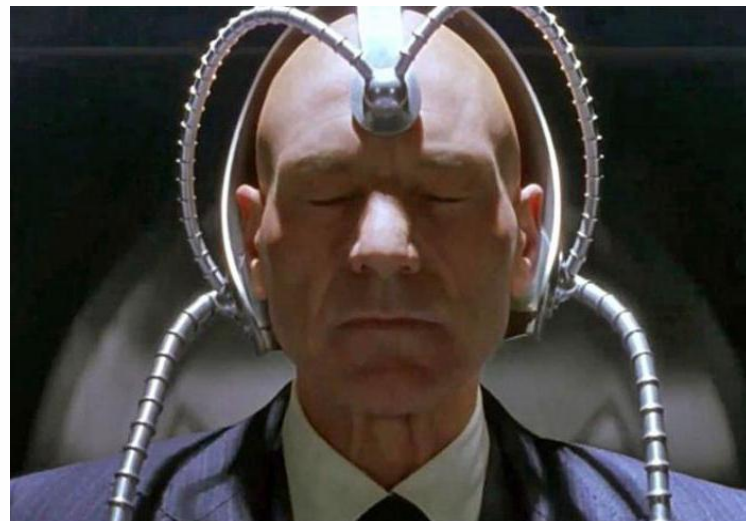
```
  else 0
```

- А тесты проходят...

```
apply(true, true) should be > 0
```

```
apply(false, true) should be > 0
```

```
apply(false, false) shouldBe 0
```





# Мутация кода

- Мутируем функцию два

```
def apply(x: Boolean, y: Boolean): Int =
```

```
  if(x && y) 100
```

```
  else if (x || y) 500
```

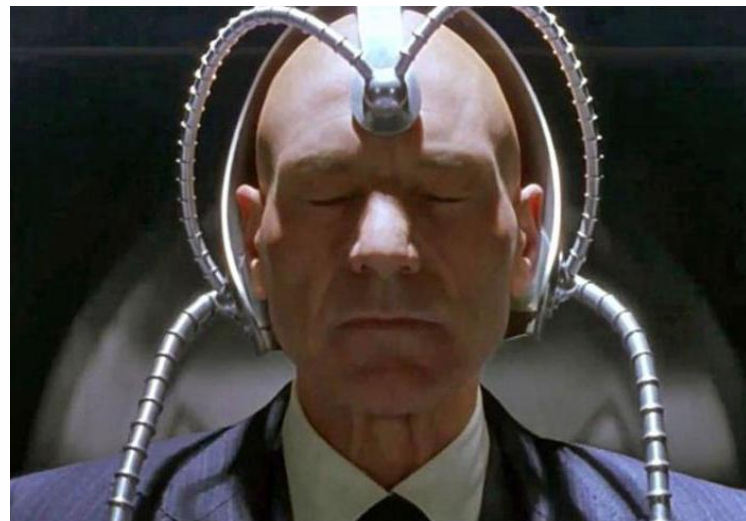
```
  else 0
```

- А тесты всё ещё проходят...

```
apply(true, true) should be > 0
```

```
apply(false, true) should be > 0
```

```
apply(false, false) shouldBe 0
```



# Мутация кода

- Мутируем функцию три

```
def apply(x: Boolean, y: Boolean): Int =
```

```
  if(x && y) 1
```

```
  else if (x || y) 2
```

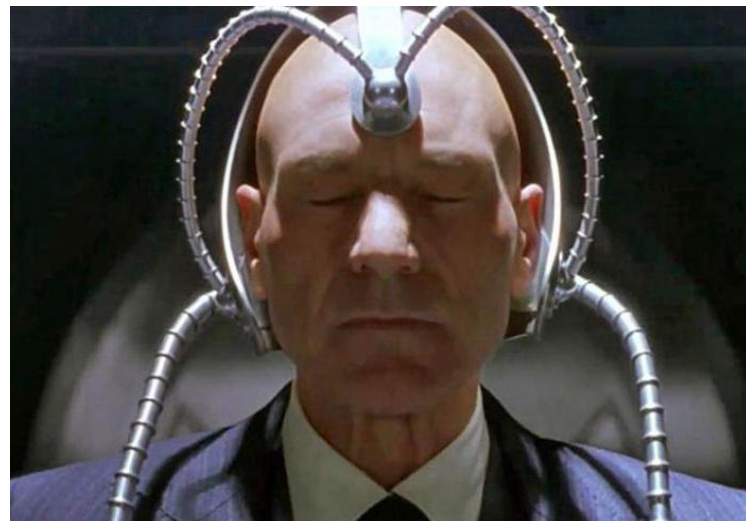
```
  else -1
```

- И наконец-то что-то сломалось

```
apply(true, true) should be > 0
```

```
apply(false, true) should be > 0
```

```
apply(false, false) shouldBe 0
```



# Как узнать что нужно добавить тесты?



Вариант 5: Изменение кода не приводит к падению тестов;

Решение: `class` MyTest{

```
def test(): Unit = {  
    assert(A(true, true) == 2)  
    assert(A(true, false) == 1)  
    assert(A(false, false) == 0)  
}
```

Результат: Не решит все проблемы, но улучшит тесты и найдет баги.

# Framework



- javascript web <https://github.com/knishiura-lab/AjaxMutator>
- javascript nodeJS <https://github.com/stryker-mutator/stryker>
- .Net <https://visualmutator.github.io/web/>
- jvm <http://pitest.org/>

# PIT



- Плагины для систем сборки maven, ant и gradle;
- Командная строка “java -cp ... ..”;
- Для sbt плагина скорее нет, чем есть  
<https://github.com/hcoles/sbt-pit>;
- Рассмотрим использование PIT из maven.

# PIT maven

`<groupId>org.pitest</groupId>`

`<artifactId>pitest-maven</artifactId>`

`<version>1.2.0</version>`

`<configuration>`

`<targetClasses>`

`<param>com.example.pitest*</param>`

`</targetClasses>`

`<targetTests>`

`<param>com.example.pitest*</param>`

`</targetTests>`

`</configuration>`

# PIT ОТЧЕТ

- Killed
- SURVIVED
- No coverage
- Non viable
- Timed Out
- Memory error
- Run error

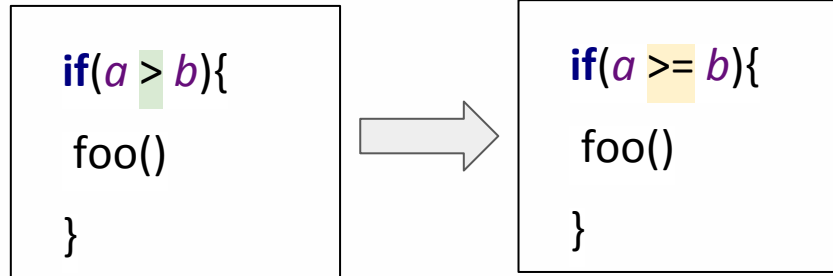
# PIT типы мутаций



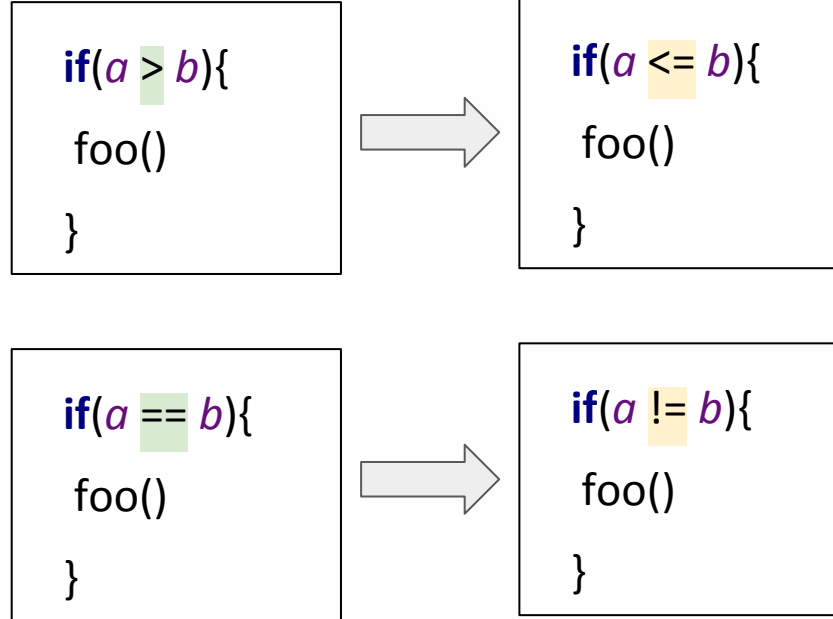
- CONDITIONALS\_BOUNDARY
- NEGATE\_CONDITIONALS
- REMOVE\_CONDITIONALS
- MATH
- INCREMENTS
- INVERT\_NEGS
- INLINE\_CONSTS
- ...



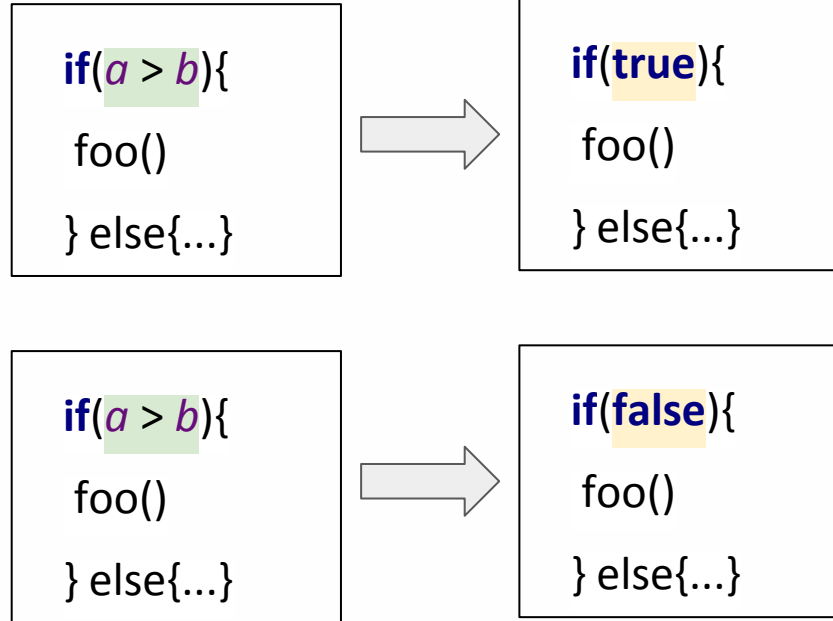
# CONDITIONALS\_BOUNDARY



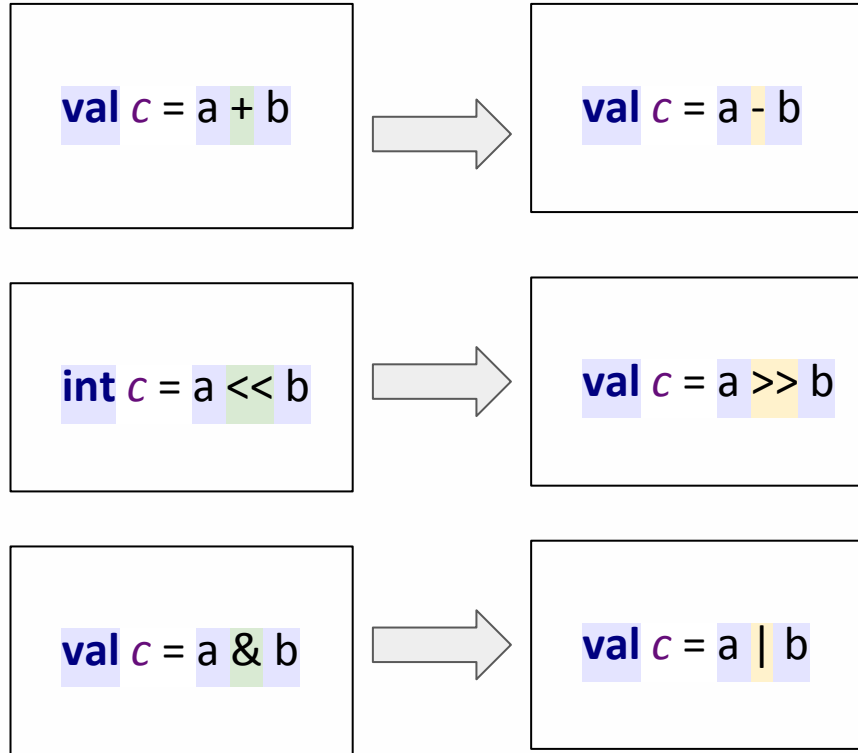
# NEGATE\_CONDITIONALS



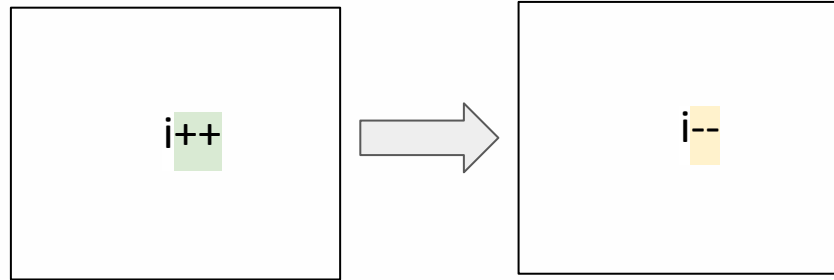
# REMOVE\_CONDITIONALS



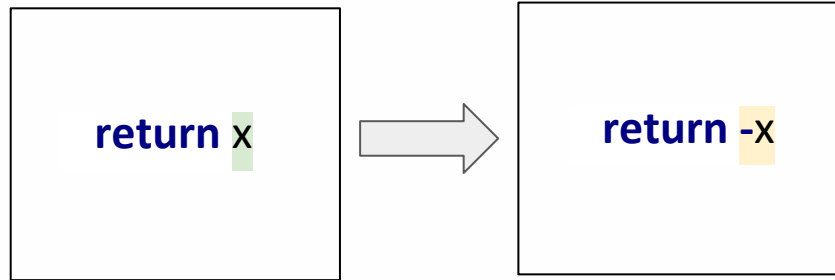
# MATH



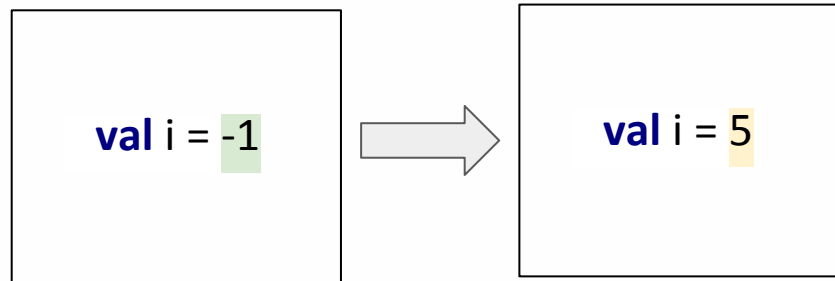
# INCREMENTS



# INVERT\_NEGS



# INLINE\_CONSTS



# RETURN\_VALS



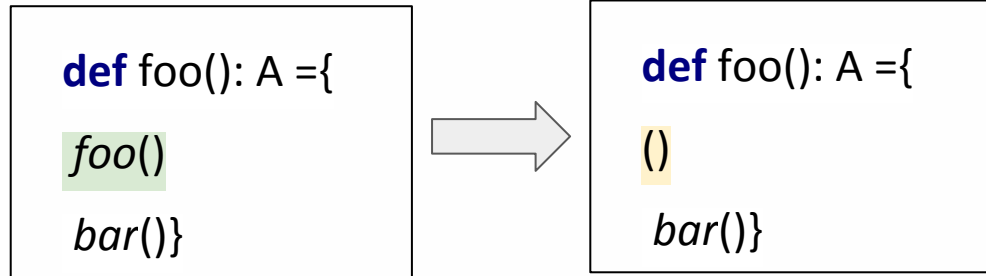
```
class A()  
def foo(): A = A()
```



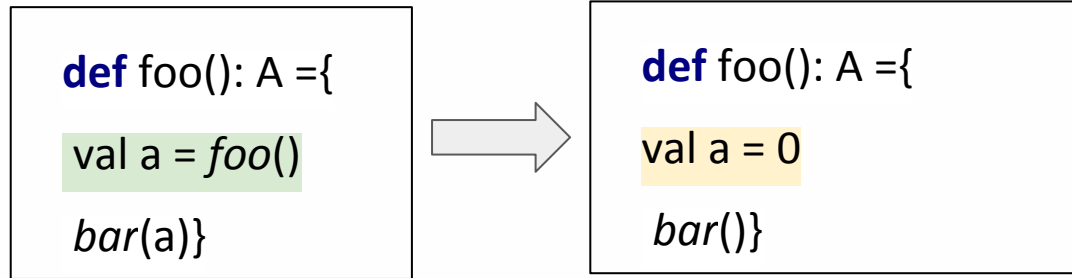
```
class A()  
def foo(): A = null
```



# VOID\_METHOD\_CALLS



# NON\_VOID\_METHOD\_CALLS



# Расстояние Левенштейна




Это минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую.

$\text{distance}(\text{“слон”}, \text{“кот”}) = 3$

слон -> лон -> кон -> кот

# РІТ в деле



Потестируем три реализации функции для расчета расстояния Левенштейна:

1. Реализация от apache  
`org.apache.commons.text.similarity.LevenshteinDistance;`
2. Реализация на scala найденная на Gist в императивном стиле;
3. Реализация на scala найденная на Gist в функциональном стиле.

# Unit ТЕСТЫ

*distance("слон", "кот")* shouldBe 3

an [IllegalArgumentException] should *be* thrownBy *distance*(null, "abc")

an [IllegalArgumentException] should *be* thrownBy *distance*("abc", null)

*distance*("abc", "abc") shouldBe 0

*distance*("abcd", "abc") shouldBe 1

*distance*("abc", "abcd") shouldBe 1

*distance*("", "") shouldBe 0

*distance*("abcdefg", "abczefg") shouldBe 1

*distance*("aaafaaa", "aaaaaaa") shouldBe 1

*distance*("fj", "") shouldBe 2

*distance*("", "a") shouldBe 1


*distance*("abc", "def") shouldBe 3

# Отчет

- Имплементация от apache
  - Mutation Coverage **78%**
- scala императивно
  - Mutation Coverage 94%
- scala функционально
  - Mutation Coverage 94%
- < 100% и это нормально
- <https://braginivan.github.io/index.html>



# Почему же 78% это норма




1. Вот эта строчка при замене 1 на 0 выживает.

```
var11[i] = Math.min(Math.min(var11[i - 1] + 1, var11[i] + 1), upperLeft + cost);
```

2.  $\geq$  или  $>$  не влияет на результат

```
def bubbleSort(seq: Seq){  
  ...  
  if(a > b) или if(a >= b)  
    replace (a, b)  
  ...  
}
```

# Почему же 78% это норма



## 3. Упрощенный расчет по условию:

```
if(str1.length == 0)
    return str2.length;
if(str2.length == 0)
    return str1.length;
```

## 4. Заполнение матрицы удобными значениями:

```
for(i = 1; i <= n; var11[i] = i++)
```

То есть все вычисления которые не влияют на результат а только на скорость.



# Серебряная пуля?

mutationCoverage 100% == quality 100%?

На этот вопрос нам ответит scalacheck.

```
forAll(Gen.alphaLowerStr, Gen.alphaChar) { (word, char) =>  
  distance(word, char + word) == 1  
}
```

*Нашел баг там где PIT подсветил зеленым*

Примеры из доклада [https://github.com/BraginIvan/scala\\_day\\_2017](https://github.com/BraginIvan/scala_day_2017)

Отчет из доклада <https://braginivan.github.io/index.html>

Доклад на Joker 2013 <https://www.youtube.com/watch?v=gGZ-5uHYAi4>

Спасибо за внимание.

Вопросы?



Thank you!