



Assignment 1

Mestrado de Engenharia Informática

Verificação e Validação de Software

2023/2024

Autor:

Ricardo Sobral 56332

Índice

LINE & BRANCH COVERAGE	3
EDGE-PAIR COVERAGE E PRIME PATH COVERAGE	4
ALL- COUPLING-USE COVERAGE:.....	5
LOGIC BASED COVERAGE: COMBINATORIAL COVERAGE	8
BASE COISE COVERAGE	9
PIT	10
JUNIT QUICK CHECK.....	13
FALHAS NO CÓDIGO	14

Line & Branch Coverage

Para atingir a cobertura em todas linhas e ramos dos métodos *contains* e *equals* e os seus métodos privados, foi criada a classe *ArrayNTreeTest*, que contem testes essências para atingir as coberturas, mas também testes adicionais.

No caso do método *contains* bastou 3 testes para atingir a cobertura de linhas e ramos. Foi testado o caso em que se verifica a existência de um elemento numa árvore vazia (*testContainsEmptyTree*), o caso do elemento ser a raiz (*testContainsElementAtRoot*) e o caso em que o elemento é está nos *childrens* (*testContainsInChildren*).

No método *equals* não foram encontrados bugs ou redundâncias no código, mas o mesmo não se pode dizer sobre o método *equals* e o seu método privado *equalTrees*. Foram encontrados redundâncias e erros que levam a uma verificação incorreta para alguns casos.

Para o método *equals* foram efetuados mais testes para garantir a cobertura de linha e ramos. No entanto, nem todos os *branches* foram possíveis de alcançar, pois existiam ramos “*unreachables*”, como é o caso no primeiro *if-statement* na função *equalTrees*, pois um dos *branches* já verificado anteriormente na função *equals*, e como a função *equalTrees* é apenas chamada pela função *equals*, então é impossível chegar até a todos os *branches* do primeiro *if-statement* de *equalTrees*.

```
private boolean equalTrees(NTree<T> one, NTree<T> other) {
    if (one == other) // 1/2 unreachable branches
        return true;
```

O mesmo acontece no segundo *if-statement* da função *equalTrees*, onde não é possível alcançar 2 dos 4 *branches* existentes.

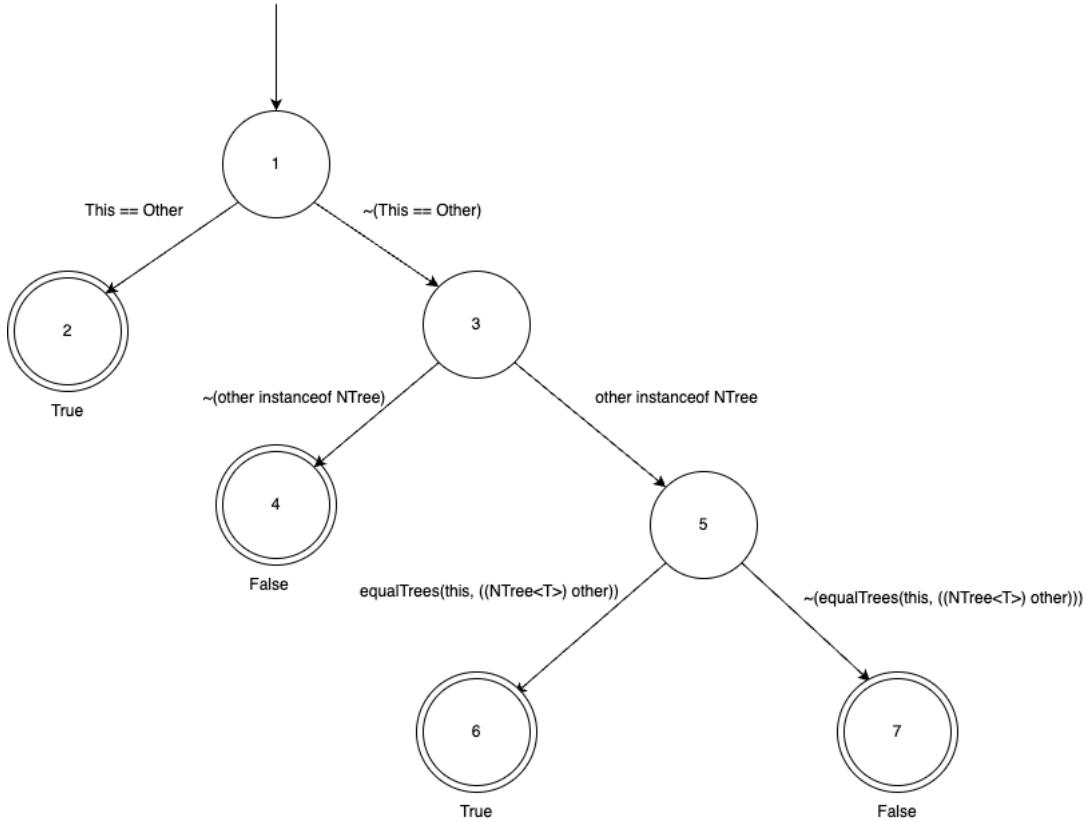
```
    if (one != null && other != null) { // 2/4 unreachable branches
```

Isto acontece, pois, a variável *one* nunca pode ser *null*, portanto não conseguimos alcançar os *branches* em que a variável *one* é *null* (*one == null && other!= null*) e ambas as variáveis são *null* (*one == nul && other == null*).

O *if-statement* referido acima é especialmente preocupante pois não possibilita comparações entre árvores em que a primeira árvore está vazia. Para exemplificar esses casos foram criados dois testes (*testEqualsTwoEmptyTrees* e *testEqualTreesFirstEmpty*) na classe *ArrayNTreeTest*. Nos casos em que ambas são árvores vazias o resultado do *equals* deveria ser *True* se a *capacity* fosse igual, e nos casos em que a primeira árvore é uma árvore vazia e a segunda não, deveria retornar *False*.

Edge-Pair Coverage e Prime Path Coverage

Método Equals:



Edge-Pair Coverage:

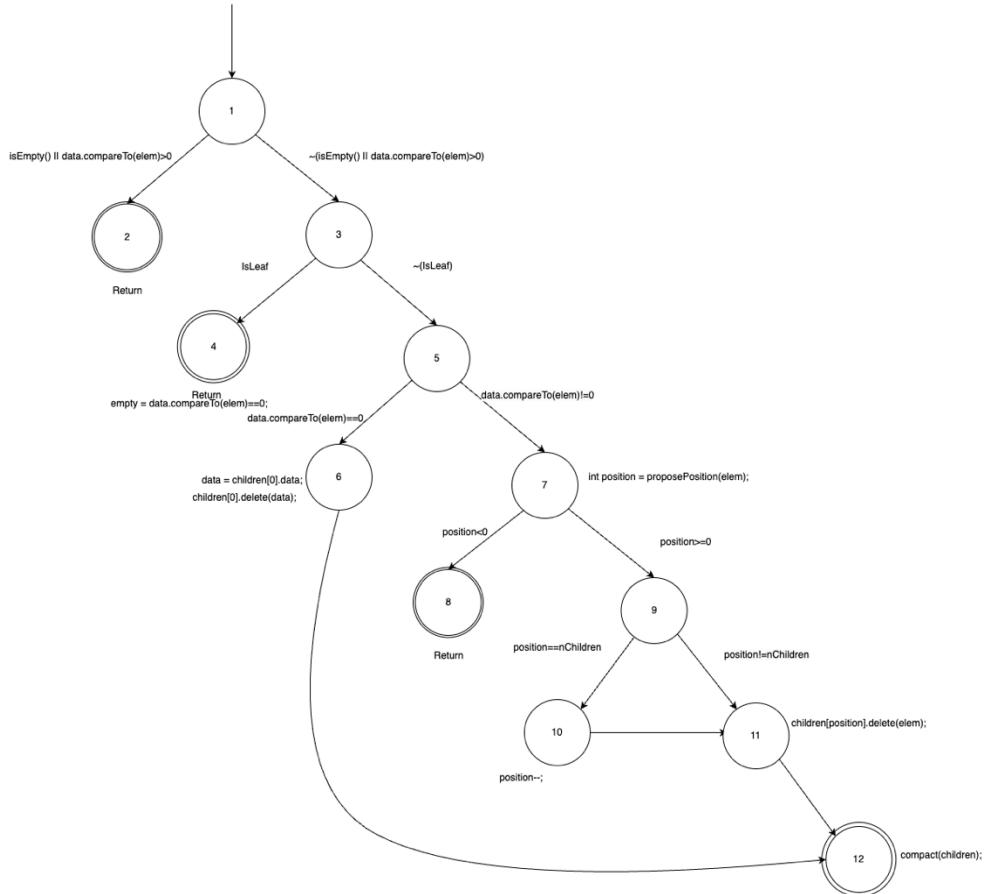
T	Test Case Values (Tree1,Tree2)	EXP. Value	Test Path	Requirements Covered
T1	tree1=tree2	True	[1,2]	[1,2]
T2	([10, 20, 21], 1)	False	[1,3,4]	[1,3,4]
T3	([10, 20, 21], [10, 20, 21])	True	[1,3,5,6]	[1,3,5], [3,5,6]
T4	([10, 20, 21], [10, 20, 23])	False	[1,3,5,7]	[1,3,5], [3,5,7]

Prime Path Coverage:

Nodes	Pairs	Triples	Quads
1	[1,2]!, [1,3]	[1,3,4]!, [1,3,5]	[1,3,5,6]!, [1,3,5,7]!
2!	-	-	-
3	[3,4]!, [3,5]	[3,5,6]!, [3,5,7]!	-
4!	-	-	-
5	[5,6]!, [5,7]!	-	-
6!	-	-	-
7!	-	-	-

All-Coupling-Use Coverage:

Método delete:



Node & Edge	Def(i)	Use(i)
1	{elem}	{}
(1,2) (1,3)	{}	{elem}
2	{}	{}
3	{}	{}
(3,4) (3,5)	{}	{elem}
4	{empty}	{elem}
5	{}	{}
(5,6) (5,7)	{}	{elem,data}
6	{data}	{children,data}
-6,12	{}	{}
7	{position}	{elem}
(7,8) (7,9)	{}	{position}
8	{}	{}
9	{}	{}
(9,10) (9,11)	{}	{position, nChildren}
10	{position}	{position}
11	{}	{children,elem, position}
{11,12}	{}	{}
12	{}	{children}

Variable	Last-Def	First-Use
position/index	proposeposition - 1,6,8 , delete - 10	delete - 10,11
children	delete - 1,6,11, compact - 9	delete - 6,11

É de notar que existe uns casos que não possíveis, por exemplo para a variável *position* não é possível fazer (1,10), (6,10),(8,10),pois como o nó 10 também é um nó *last-def*, no nós 1,6 e 8 deixam de ser *last-def*.

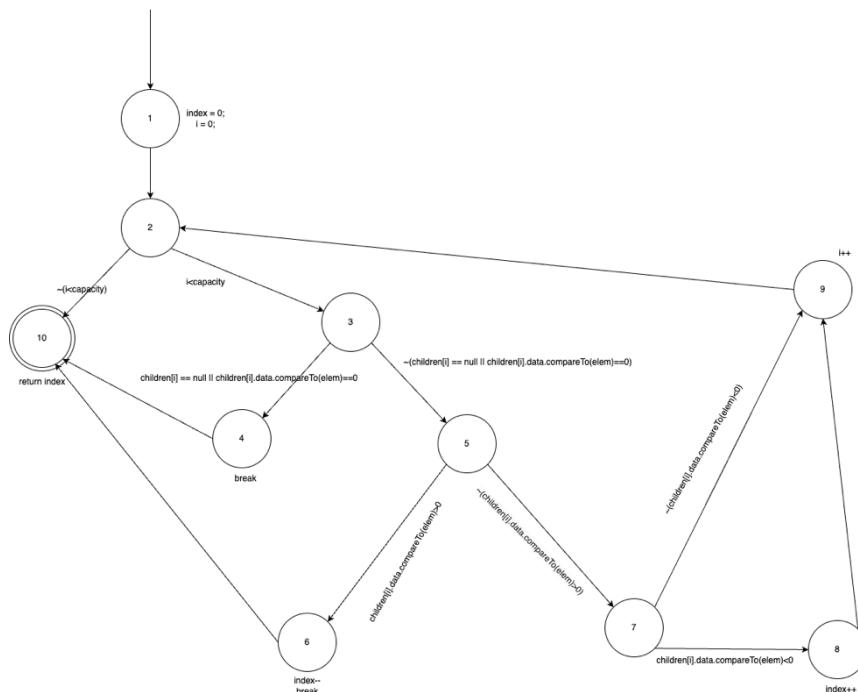
O par (delete- 10, delete- 11) para a variável *position* também não é possível pois o nó 10 é juntamente *last-def* e *first-use*. Algo parecido acontece com o nó 1 para a variável *children*, pelo que o nó 6 e 11 são em simultâneo nós *first-use* e *last-def*.

O par (delete – 6, delete – 6) não é possível pois para entrar no nó 6 o elemento a apagar tem que esta na *root* , se o *root* tiver filhos a última *def* vai ser no *compact* - 9, se não tiver filhos entra no nó 4 e termina execução, logo não podemos ter (delete – 6, delete – 6).

O par (*compact*- 9, *delete*-11) não foi considerado pois só possível obtê-lo através da capacidade de recursividade da função *delete*, sendo assim obter as seguintes combinações:

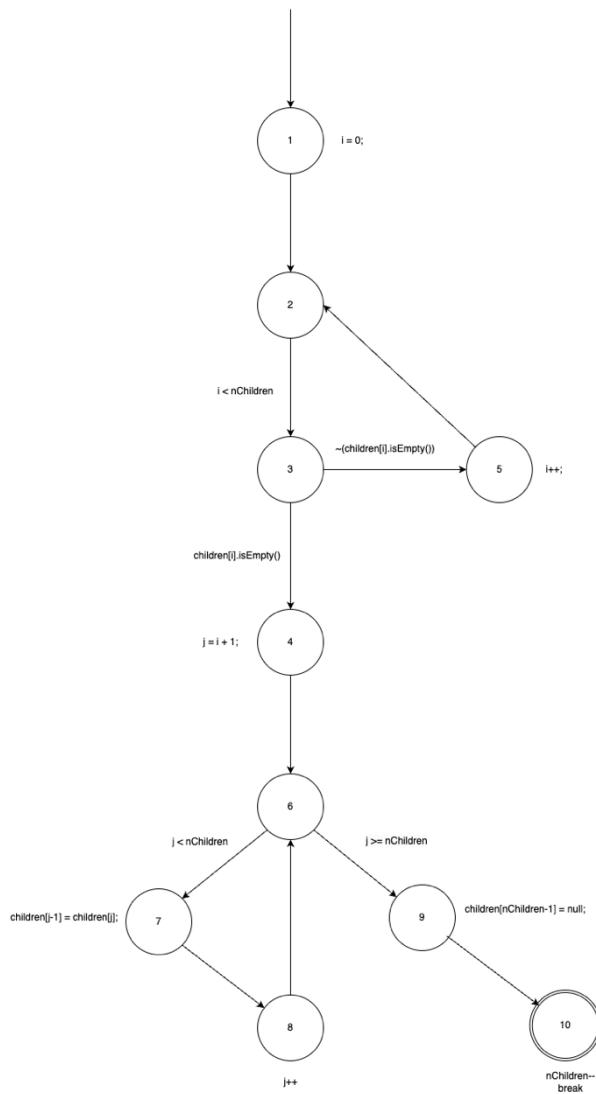
Variables	Combinations (Last-def, First-use)
position/index	(proposeposition - 1, delete - 11)
	(proposeposition - 6, delete - 11)
	(proposeposition - 8, delete - 11)
	(delete - 10, delete - 10)
Children	(delete - 11, delete - 11)
	(compact - 9, delete - 6)

Método *proposePosition*:



Node Edge	Def(i)	Use(i)
1	{elem,index,i}	{}
(1,2)	{}	{}
2	{}	{}
(2,3),(2,10)	{}	{i}
3	{}	{}
(3,4),(3,5)	{}	{children,elem,i}
4	{}	{}
(4,10)	{}	{}
5	{}	{}
(5,6),(5,7)	{}	{children,elem,i}
6	{index}	{index}
(6,10)	{}	{}
7	{}	{}
(7,8),(7,9)	{}	{children,elem,i}
8	{index}	{index}
9	{i}	{i}
(8,9)	{}	{}
(9,2)	{}	{}
10	{}	{index}

Método Compact:



Node Edge	Def	Use
1	{children,i}	{}
(1,2)	{}	{}
2	{}	{}
(2,3)	{}	{i,nChildren}
3	{}	{}
(3,4),(3,5)	{}	{children,i}
4	{j}	{i}
5	{i}	{i}
(4,6)	{}	{}
6	{}	{}
(6,7),(6,9)	{}	{j,nChildren}
7	{}	{children,j}
(7,8)	{}	{}
8	{j}	{i}
(8,6)	{}	{}
9	{children}	{nChildren}
(9,10)	{}	{}
10	{nChildren}	{nChildren}

Logic Based Coverage: Combinatorial Coverage

Combinatorial Coverage (CoC) foi escolhido por várias razões como ter um bom “custo benefício”, percorre todos os casos possíveis, foca-se em casos reais, e como o método *equals* não é muito complexo então nunca chega a ser exaustivo este tipo de teste mas para além disso, o facto de metade dos casos possíveis não puderem ser alcançados (4 de 8 casos), devido à natureza das clausulas então a maioria dos outros testes lógicos tornar-se-iam impraticáveis (CC, PC, CACC) e outros não teriam tanta utilidade (GACC, RACC).

Como o método *Equals* é simples usar CoC nunca a ser impraticável!

Explicação porque de alguns casos não serem fazíveis:

Sendo que Predicado \rightarrow A || B && C; A \rightarrow This == Other; B \rightarrow other instanceof NTree; C \rightarrow equalTrees(this, ((NTree<T>) other));

A	B	C	Value
T	T	T	T
T	T	F	T
T	F	T	T
T	F	F	T
F	T	T	T
F	T	F	F
F	F	T	F
F	F	F	F

2º caso) A:T B:T C:F V:T Não é fazível, pois se os objetos tem a mesma referência e são do tipo *Ntree* então tem que ser obrigatoriamente iguais;

3º caso) A:T B:F C:T V:T Não é fazível, pois para que os objetos seja iguais por *equalsTree* então tem que ser de instancia *Ntree*;

4º caso) A:T B:F C:F V:T Não é fazível, pois se tem a mesma referencia e o *other* não é do tipo *Ntree*, logo o *this* não é *Ntree* e assim não podemos usar a função *equals* pois é apenas aplicada a objetos tipo *Ntree*;

7º caso) A:F B:F C:T V:F Não é fazível, pois para *EqualTrees* ser verdadeiro então *other* tem que ser do tipo *Ntree*

Base Coise Coverage

Método equals:

1. Tree 1 está fazia:

tree1empty

!tree1empty

2. Tree 2 está fazia:

tree2empty

!tree2empty

3. Tree 2 está null:

tree2null

!tree2null

4. Comparar interseção de duas árvores, Tree 1 e Tree 2:

Não existe interseção (empty)

Tree 1 e Tree 2 partilham elementos (partial)

Tree 1 e Tree 2 são iguais (full)

Partitions	Base Choice	Tests
[tree1empty, !tree1empty] [tree2empty, !tree2empty] [tree2null, !tree2null] [empty, partial, full]	[!tree1empty, !tree2empty, !tree2null, empty]	[!tree1empty, !tree2empty, !tree2null, empty]
	[!tree1empty, !tree2empty, !tree2null, parcial]	[!tree1empty, !tree2empty, !tree2null, parcial]
	[!tree1empty, !tree2empty, !tree2null, full]	[!tree1empty, !tree2empty, !tree2null, full]
	[tree1empty, !tree2empty, !tree2null, empty]*	[tree1empty, !tree2empty, !tree2null, empty]*
	[!tree1empty, tree2empty, !tree2null, empty]	[!tree1empty, tree2empty, !tree2null, empty]
	[!tree1empty, !tree2empty, tree2null, empty]	[!tree1empty, !tree2empty, tree2null, empty]

* [tree1empty, !tree2empty, !tree2null, empty] este teste está como assertThrows, pois a função equals não está preparada para os casos em que a "tree1" está vazia

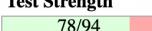
O caso [!tree1empty, tree2empty, tree2null, empty], no entanto percebe-se que não é possível realizar este teste pois não é possível ter uma *tree null* e vazia ao mesmo tempo pois se uma é *null* não é uma *tree*, logo não pode ser uma *tree* vazia.

PIT

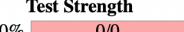
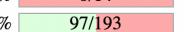
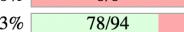
Utilização da ferramenta PIT na classe EdgePairCoverageTest.java

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage	Mutation Coverage	Test Strength
3	47%  97/207	43%  78/182	83%  78/94

Breakdown by Package

Name	Number of Classes	Line Coverage	Mutation Coverage	Test Strength
startup	1	0%  0/14	0%  0/8	0%  0/0
sut	2	50%  97/193	45%  78/174	83%  78/94

Report generated by [PIT](#) 1.6.8

```
293     public boolean equals(Object other) {
294 3 |         return this == other ||
295 1 |             other instanceof NTree &&
296 1 |                 equalTrees(this, ((NTree<T>) other));
297     }
298
299     // Compares the elements between two NTrees
300     private boolean equalTrees(NTree<T> one, NTree<T> other) {
301 1 |         if (one == other) // 1/2 unreachable branches
302 1 |             return true;
303
304         if (one != null && other != null) { // 2/4 unreachable branches
305 2 |             Iterator<T> it1 = one.iterator();
306 2 |             Iterator<T> it2 = other.iterator();
307
308             while(it1.hasNext() && it2.hasNext())
309 2 |                 if(!it1.next().equals(it2.next()))
310 1 |                     return false;
311 1 |
312             if(!it1.hasNext() && !it2.hasNext())
313 2 |                 return true;
314 1 |
315         }
316
317 1 |         return false;
318 }
```

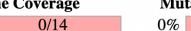
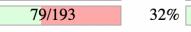
Utilização da ferramenta PIT na classe EdgePairCoverageTest.java:

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage	Mutation Coverage	Test Strength
3	38%  79/207	30%  55/182	73%  55/75

Breakdown by Package

Name	Number of Classes	Line Coverage	Mutation Coverage	Test Strength
startup	1	0%  0/14	0%  0/8	0%  0/0
sut	2	41%  79/193	32%  55/174	73%  55/75

Report generated by [PIT](#) 1.6.8

```

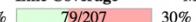
293     public boolean equals(Object other) {
294 3         return this == other ||
295 1             other instanceof NTree &&
296 1                 equalTrees(this, ((NTree<T>) other));
297     }
298
299     // Compares the elements between two NTrees
300     private boolean equalTrees(NTree<T> one, NTree<T> other) {
301 1         if (one == other) // 1/2 unreachable branches
302 1             return true;
303
304
305 2         if (one != null && other != null) { // 2/4 unreachable branches
306             Iterator<T> it1 = one.iterator();
307             Iterator<T> it2 = other.iterator();
308
309 2             while(it1.hasNext() && it2.hasNext())
310 1                 if(!it1.next().equals(it2.next()))
311 1                     return false;
312
313 2             if(!it1.hasNext() && !it2.hasNext())
314 1                 return true;
315     }
316
317 1         return false;
318

```

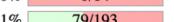
Utilização da ferramenta PIT na classe PrimePathCoverage.java:

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage	Mutation Coverage	Test Strength
3	38%  79/207	30%  55/182	73%  55/75

Breakdown by Package

Name	Number of Classes	Line Coverage	Mutation Coverage	Test Strength
startup	1	0%  0/14	0%  0/8	0%  0/0
sut	2	41%  79/193	32%  55/174	73%  55/75

Report generated by [PIT](#) 1.6.8

```

293     public boolean equals(Object other) {
294 3         return this == other ||
295 1             other instanceof NTree &&
296 1                 equalTrees(this, ((NTree<T>) other));
297     }
298
299     // Compares the elements between two NTrees
300     private boolean equalTrees(NTree<T> one, NTree<T> other) {
301 1         if (one == other) // 1/2 unreachable branches
302 1             return true;
303
304
305 2         if (one != null && other != null) { // 2/4 unreachable branches
306             Iterator<T> it1 = one.iterator();
307             Iterator<T> it2 = other.iterator();
308
309 2             while(it1.hasNext() && it2.hasNext())
310 1                 if(!it1.next().equals(it2.next()))
311 1                     return false;
312
313 2             if(!it1.hasNext() && !it2.hasNext())
314 1                 return true;
315     }
316
317 1         return false;
318

```

Utilização da ferramenta PIT na classe LogicBaseEqualsTest.java:

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage	Mutation Coverage	Test Strength
3	19%	14%	93%

Breakdown by Package

Name	Number of Classes	Line Coverage	Mutation Coverage	Test Strength
startup	1	0%	0%	0%
sut	2	21%	14%	93%

Report generated by [PIT](#) 1.6.8

```
293     public boolean equals(Object other) {
294 3       return this == other ||
295 1       other instanceof NTree &&
296 1       equalTrees(this, ((NTree<T>) other));
297   }
298
299   // Compares the elements between two NTrees
300   private boolean equalTrees(NTree<T> one, NTree<T> other) {
301 1     if (one == other) // 1/2 unreachable branches
302 1       return true;
303
304     if (one != null &amp; other != null) { // 2/4 unreachable branches
305 2       Iterator<T> it1 = one.iterator();
306
307       Iterator<T> it2 = other.iterator();
308
309 2       while(it1.hasNext() && it2.hasNext())
310 1         if(!it1.next().equals(it2.next()))
311 1           return false;
312
313 2       if(!it1.hasNext() && !it2.hasNext())
314 1         return true;
315     }
316
317 1     return false;
318 }
```

Utilização da ferramenta PIT na classe BaseChoiceCoverageEquals.java:

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage	Mutation Coverage	Test Strength
3	38%	25%	61%

Breakdown by Package

Name	Number of Classes	Line Coverage	Mutation Coverage	Test Strength
startup	1	0%	0%	0%
sut	2	41%	26%	61%

Report generated by [PIT](#) 1.6.8

```

292     @SuppressWarnings("unchecked")
293     public boolean equals(Object other) {
294         3 return this == other || 
295         1 other instanceof NTree &&
296         1 equalTrees(this, ((NTree<T>) other));
297     }
298
299     // Compares the elements between two NTrees
300     private boolean equalTrees(NTree<T> one, NTree<T> other) {
301         1 if (one == other) // 1/2 unreachable branches
302         1 return true;
303
304
305         2 if (one != null && other != null) { // 2/4 unreachable branches
306             Iterator<T> it1 = one.iterator();
307             Iterator<T> it2 = other.iterator();
308
309         2 while(it1.hasNext() && it2.hasNext())
310         1 if(!it1.next().equals(it2.next()))
311         1 return false;
312
313         2 if(!it1.hasNext() && !it2.hasNext())
314         1 return true;
315     }
316
317         1 return false;
318 }

```

Apesar de aparentar haver diferenças sobre a *mutation coverage* nas várias classes, quando de facto avaliamos a *mutation coverage* apenas do método *equals* percebemos que o resultado é o mesmo para as várias classes que efetuam testes sobre o método *equals*. Em todos os casos o *mutation coverage* apena não consegui alcançar o primeiro *return true* e o segundo *return false*.

JUnit Quick Check

Para efetuar os testes *Junit* com *Quick Check* foi criada uma classe *RandomArrayNTreeGenerator*, que gera n-trees de forma aleatória, com tamanho máximo de 25 e o maior valor possível que um nó pode ter é 200. O número de filhos por nó varia entre 2 e 4, este número é relativamente pequeno para evitar o aumento de complexidade.

Para verificar as propriedades e executar os testes foi criada a classe *QuickCheckTest*, que para além dos 5 testes relacionado com as propriedades a classe *QuickCheckTest* tem a função *stillInvariant* que verifica se a propriedade de invariância da árvore se mantém.

Falhas no código

As falhas encontradas no código estão relacionadas à função equals e equalTrees, que foram referidas anteriormente no capítulo Line & Branch Coverage. E estas devem-se ao facto da função equalTrees ser uma função privada que é apenas chamada pela função equals, a qual executa verificações previamente antes de chamar a sua função privada, levando a que está não consiga cobrir todos os casos possíveis. Por sua vez a função equals não é capaz de lidar com casos em que a tree1 é null, levando a que alguns testes, nomeadamente aqueles com a primeira árvore vazia, não pudessem ser avaliados devidamente.

```
/*
@SuppressWarnings("unchecked")
public boolean equals(Object other) {
    return this == other ||
           other instanceof NTree &&
           equalTrees(this, ((NTree<T>) other));
}

// Compares the elements between two NTrees
private boolean equalTrees(NTree<T> one, NTree<T> other) {
    if (one == other) // 1/2 unreachable branches
        return true;

    if (one != null && other != null) { // 2/4 unreachable branches
        Iterator<T> it1 = one.iterator();
        Iterator<T> it2 = other.iterator();

        while(it1.hasNext() && it2.hasNext())
            if(!it1.next().equals(it2.next()))
                return false;

        if(!it1.hasNext() && !it2.hasNext())
            return true;
    }

    return false;
}
```