
Relatório Projeto - Trabalho Prático

Miniprojecto em Engenharia de Software

Escola Superior de Tecnologias e Gestão de Águeda
Tecnologias da Informação - 2ºano
1º Semestre
2019-2020

Jorge Miguel Soares de Oliveira - 83816

Professor: Joaquim Ferreira

1 – Breve Explicação

Este projeto tem como principal objetivo apresentar os testes estruturais e funcionais para a classe **CalculadoraSufixa**. Deverá ainda ser disponibilizada uma solução para que os testes unitários não envolvam a utilização de ficheiros, para serem rápidos a executar. Para tal deve ser feito o *refactoring* do código.

Dessa forma foi facultada a seguinte classe onde podemos verificar que a mesma lê os respetivos ficheiros. Para que seja de rápida execução é viável colocar de modo a que as informações sejam lidas diretamente na consola como indicado na figura.

```
public static void main(String[] args) throws IOException {
    Scanner leitor = new Scanner(System.in);
    System.out.print("Ficheiro de dados? ");
    Scanner ficheiro = new Scanner(new FileReader(leitor.next()));
    System.out.print("Ficheiro de resultados? ");
    BufferedWriter output = new BufferedWriter(new FileWriter(leitor.next()));
    String expressao, sufixa;
    Scanner temp;
    CalculadoraSufixa calculadora;
    while (ficheiro.hasNextLine()) {
        expressao = ficheiro.nextLine();
        if (!CalculadoraSufixa.verificaParenteses(expressao))
            output.write("Número de parênteses não é coincidente");
        else {
            sufixa = CalculadoraSufixa.paraSufixa(expressao);
            output.write(sufixa+"?");
            temp = new Scanner(sufixa);
            calculadora = new CalculadoraSufixa();
            try {
                while (temp.hasNext())
                    calculadora.processaToken(temp.next());
                if (calculadora.haResultado())
                    if (Double.isNaN(calculadora.verResultado()))
                        output.write(" Divisão por zero.");
                    else
                        output.write(" "+calculadora.verResultado());
                else
                    output.write(" Pilha em estado incorrecto.");
            } catch (EmptyStackException e) {
                output.write(" Argumentos insuficientes.");
            }
        }
        output.newLine();
    }
    output.close();
}
```

```
public static void main(String[] args) throws IOException {
    //3 próximas linhas para colocar em vez do que esta comentado
    String resultados="";
    Scanner leitor = new Scanner(System.in);
    String input = leitor.nextLine();

    /*System.out.print("Ficheiro de dados? ");
    Scanner ficheiro = new Scanner(new FileReader(leitor.next()));
    System.out.print("Ficheiro de resultados? ");
    BufferedWriter output = new BufferedWriter(new FileWriter(leitor.next()));*/

    String expressao, sufixa;
```

Como se pode verificar na figura acima, é substituído a forma de ler ficheiros pela forma de fazer as operações diretamente na consola lendo diretamente pelo teclado. Todo o restante código ficará desenvolvido desta forma:

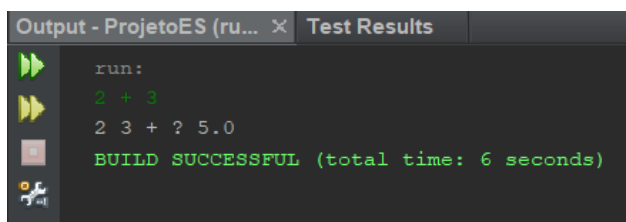
```
String expressao, sufixa;
Scanner temp;
CalculadoraSufixa calculadora;
/*while (ficheiro.hasNextLine()){
    expressao = ficheiro.nextLine();*/
    if (!CalculadoraSufixa.verificaParenteses(input))
        //output.write("Número de parênteses não é coincidente");
        resultados += "Número de parênteses não é coincidente";
    else {
        sufixa = CalculadoraSufixa.paraSufixa(input);
        //output.write(sufixa+"?");
        resultados += sufixa+"?";
        temp = new Scanner(sufixa);
        calculadora = new CalculadoraSufixa();
        try {
            while (temp.hasNext())
                calculadora.processaToken(temp.next());
            if (calculadora.haResultado())
                if (Double.isNaN(calculadora.verResultado()))
                    //output.write(" Divisão por zero.");
                    resultados += " Divisão por zero.";
                else
                    //output.write(" "+calculadora.verResultado());
                    resultados += " "+calculadora.verResultado();
            else
                //output.write(" Pilha em estado incorreto.");
                resultados += " Pilha em estado incorreto.";
        } catch (EmptyStackException e) {
            //output.write(" Argumentos insuficientes.");
            resultados += " Argumentos insuficientes.";
        }
    }
    //output.newLine();
    System.out.println(resultados);
}
//output.close();
```

2 – Métodos a realizar os testes

Após serem resolvidas estas questões teremos o próximo passo, construir os testes para os 3 métodos pedidos. São eles:

- **public static boolean verificaParenteses(String infixa)**
- **public static boolean eOperador(String token)**
- **public static String paraSufixa(String infixa)**

Agora é possível testar o programa:



De forma automática, o programa tenta converter com sucesso da forma infixa para a forma sufixa, como se verifica.

4 – Caixa Branca e Caixa Preta

Por conseguinte será necessário fazer os testes de caixa branca e caixa preta para cada um dos **3 métodos**. Os testes de caixa branca são baseados na estrutura interna do programa. O objetivo é assegurar que todas as atribuições e condições são executadas pelo menos uma vez. O teste exaustivo está fora de questão realizar pois demoraria anos para percorrer todos os caminhos com todas as instruções possíveis. O teste do tipo caminho base (*Base Path*) - é uma técnica de caixa branca, onde se calcula a complexidade lógica do software e se utiliza esta medida como base para descobrir os caminhos básicos:

1) calcula-se: nº de decisões simples + 1 --> exemplo: $V(G)=N$

- Quanto maior $V(G)$ maior a probabilidade de erros

2) identificam-se os caminhos independentes

Dado que $V(G)=N$, há N caminhos

- Caminho 1: 1,2,3,6,7,8"

- Caminho 2: 1,2,3,5,7,8"

- Caminho 3: 1,2,4,7,8"

- Caminho N : 1,2,4,7,2,4,...,7,8

3) contruir casos de teste

O teste de caixa preta não é baseado na estrutura do programa O seu foco é no comportamento das entradas/saídas. O objetivo deste tipo de teste consiste na redução do número de casos de teste usando partição de equivalências. Para isso deve-se escolher casos de teste para cada classe de equivalência.

5 – Método para verificar parenteses

5.1 – Caixa Branca

Começando pelo primeiro método “verificaParenteses”, contam-se os caminhos, identificam-se e constroem-se os respetivos casos de teste conforme estão identificados.

```
public static boolean verificaParenteses(String infixa){ //verifica se os parenteses estao corretos
    Scanner expressao = new Scanner(infixa); //le os valores --> (1 + 1)
    Stack<String> pilha = new Stack<String>(); //cria a pilha para colocar os valores
    String token; //variavel a usar
    try {
        1 while (expressao.hasNext()){ //PRIMEIRO CAMINHO - entra aqui sempre que expressao nao for vazia
            token = expressao.next(); //token = ( --> primeira posicao
            2 if (token.equals("(")) //SEGUNDO CAMINHO
                pilha.push(token); //envia para a pilha o (
            3 else if (token.equals(")")) //TERCEIRO CAMINHO
                pilha.pop(); //retira da pilha
        }
        4 return (pilha.isEmpty()); //QUARTO CAMINHO - se tiver vazia retorna true
    } catch (EmptyStackException e) {
        return false;
    }
}
```

De seguida o objetivo é criar os casos de teste para cada caminho na classe “CalculadoraSufixaTest” que acabou de ser criada. No método “testVerificaParenteses”. Cada caso de teste deve ser criado com o **assertTrue** ou **assertFalse** que indica se o caso deve dar certo ou errado conforme na imagem seguinte.

```
public void testVerificaParenteses() {  
    //caixa branca - sao 4 caminhos porque há 3 condições simples  
    assertTrue(CalculadoraSufixa.verificaParenteses(" ( ) ")); //faz if e else if e retorna  
    assertTrue(CalculadoraSufixa.verificaParenteses(" ")); //return pilha.isEmpty  
    assertFalse(CalculadoraSufixa.verificaParenteses(" ) ")); //if retorna false e else if retorna true  
    assertFalse(CalculadoraSufixa.verificaParenteses(" ( ")); //if retorna true e else if retorna false  
}
```

Com estes casos de teste verifica-se que sempre que a expressão indicada pelo utilizador tem dois parenteses ou nenhum usa-se o **assertTrue**, pois significa que é sempre que a expressão está correta. Sempre que apenas é indicado um dos parenteses a expressão está incorreta.

Desta forma ficam concluídos os testes de caixa branca.

5.2 – Caixa Preta

Para os testes de caixa preta devem-se identificar aqueles que se consideram necessários e críticos para o correto funcionamento do programa. Os escolhidos foram os da figura seguinte, sendo que o objetivo foi selecionar expressões com mais parenteses.

```
//caixa preta - todos os casos que se pretender  
assertTrue(CalculadoraSufixa.verificaParenteses(" ( ) ")); //esta certo  
assertTrue(CalculadoraSufixa.verificaParenteses(" "));  
assertFalse(CalculadoraSufixa.verificaParenteses(" ) ")); //esta errado  
assertFalse(CalculadoraSufixa.verificaParenteses(" ( "));  
assertFalse(CalculadoraSufixa.verificaParenteses(" ( ( "));  
assertFalse(CalculadoraSufixa.verificaParenteses(" ( ( ) "));  
assertTrue(CalculadoraSufixa.verificaParenteses(" ( ( ) ) "));  
assertFalse(CalculadoraSufixa.verificaParenteses(" ( ) ) "));
```

6 – Método eOperador

6.1 – Caixa Branca

O próxima método a ser testado é o **public static boolean eOperador(String token)** que é responsável por verificar que um dos 4 operados está contido na expressão indicada pelo utilizador.

```
public static boolean eOperador(String token){ //verifica se o operador é um das 4 opções  
    return token.equals("+") || token.equals("-") ||  
           token.equals("*") || token.equals("/"); //retorna true se for um dos 4 operadores  
}
```

Para este caso é bastante fácil identificarem-se os caminhos, pois as condições simples são 4, portanto os caminhos serão automaticamente 5, conforme indicado na figura seguinte.

```
public void testEOperador() {  
    //caixa branca - sao 5 caminhos porque há 4 condições simples  
    assertTrue(CalculadoraSufixa.eOperador("+")); //Caminho 1 --> faz o +  
    assertTrue(CalculadoraSufixa.eOperador("-")); //Caminho 2 --> faz o -  
    assertTrue(CalculadoraSufixa.eOperador("/")); //Caminho 3 --> faz o /  
    assertTrue(CalculadoraSufixa.eOperador("*")); //Caminho 4 --> faz o *  
    assertFalse(CalculadoraSufixa.eOperador(".")); //Caminho 5 --> faz o return false  
}
```

6.1 – Caixa Preta

Agora, como no exemplo anterior, nos de caixa preta considerou-se necessário colocar os “:” pois pode ser confundido com o símbolo de divisão que é “/”.

```
//caixa preta
assertTrue(CalculadoraSufixa.eOperador("+"));
assertTrue(CalculadoraSufixa.eOperador("-"));
assertTrue(CalculadoraSufixa.eOperador("/"));
assertTrue(CalculadoraSufixa.eOperador("*"));
assertFalse(CalculadoraSufixa.eOperador("."));
assertFalse(CalculadoraSufixa.eOperador(":"));
assertFalse(CalculadoraSufixa.eOperador("1"));
```

7 – Método paraSufixa

7.1 – Caixa Branca

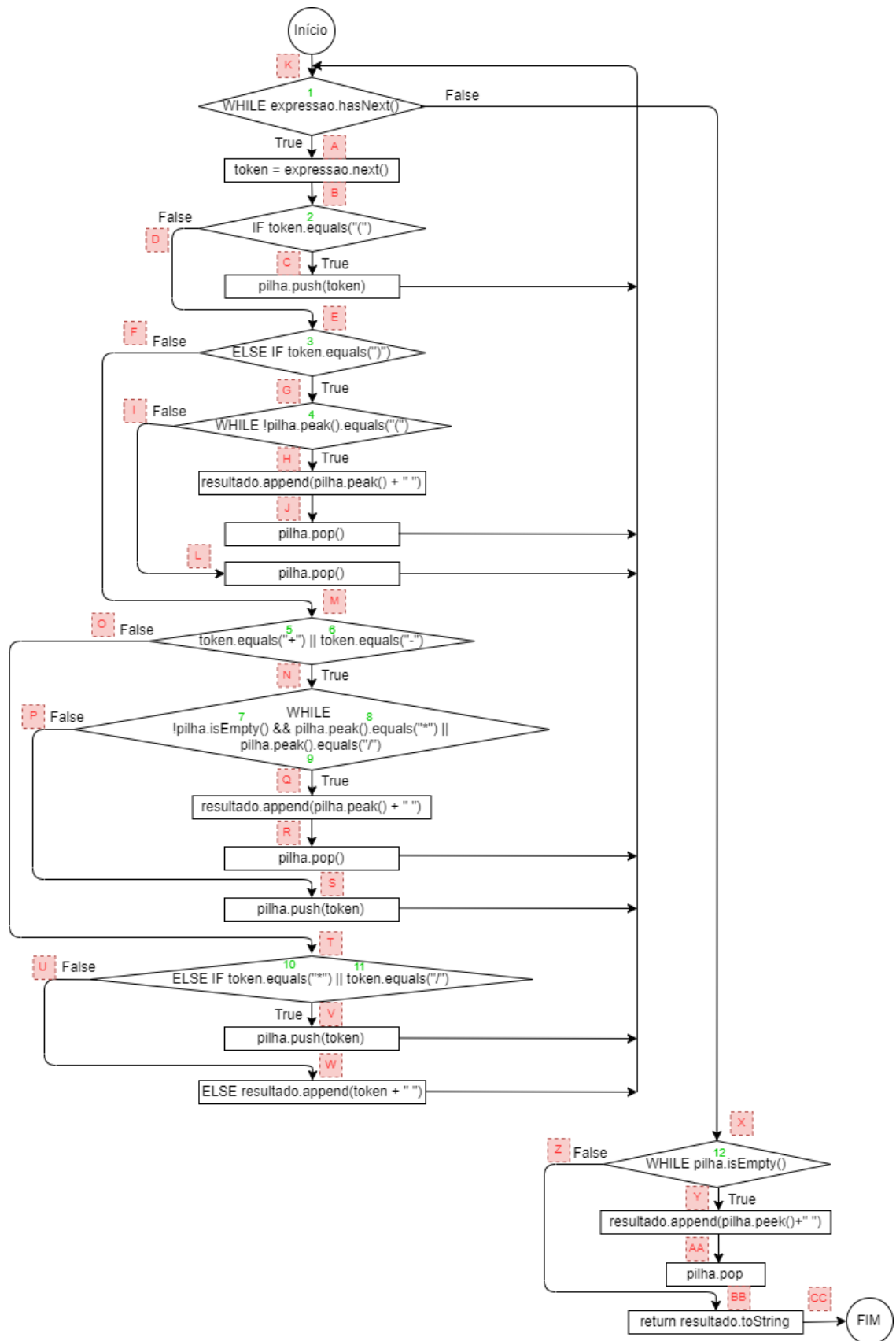
Para último método falta apenas explicar o **public static String paraSufixa(String infixa)** que consiste na técnica de converter a expressão infixa para sufixa. Exemplo: 1 + 2 → 1 2 +

```
public static String paraSufixa(String infixa){
    Scanner expressao = new Scanner(infixa);
    String token;
    Stack<String> pilha = new Stack<String>();
    StringBuilder resultado = new StringBuilder();
    while (expressao.hasNext()){ //enquanto houver a seguir - string nao vazia
        token = expressao.next(); //andar para o lado
        if (token.equals("("))
            pilha.push(token); //mete na pilha
        else if (token.equals(")")){
            while (!pilha.peek().equals("(")){
                resultado.append(pilha.peek()+" ");
                pilha.pop(); //tira da pilha
            }
            pilha.pop();
        }
        else if (token.equals("+") || token.equals("-")){
            while (!pilha.isEmpty() && (pilha.peek().equals("+") || pilha.peek().equals("/"))){
                resultado.append(pilha.peek()+" ");
                pilha.pop();
            }
            pilha.push(token);
        }
        else if (token.equals("*") || token.equals("/"))
            pilha.push(token);
        else // é um operando
            resultado.append(token+" ");
    }
    while (!pilha.isEmpty()){
        resultado.append(pilha.peek()+" ");
        pilha.pop();
    }
    return resultado.toString();
}
```

Este método, pelo que se comprovou é o mais difícil de realizar ambos os testes pois dispõe de muitos mais caminhos. As condições simples são 12, logo os caminhos serão 13. Para facilitar a construção dos caminhos foi necessário recorrer à construção de um fluxograma para ser mais simples de os identificar.

7.1.1 – Fluxograma

O fluxograma final é o seguinte, com os caminhos já identificados



7.1.2 – Caminhos

Os caminhos que foram descobertos são:

- Caminho 1 --> K-A-B-C-K-X-Z-BB-CC
- Caminho 2 --> K-A-B-C-K-X-Y-AA-BB-CC
- Caminho 3 --> K-A-B-D-E-G-H-J-K-X-Z-BB-CC
- Caminho 4 --> k-A-B-D-E-G-I-L-K-X-Z-BB-CC
- Caminho 5 --> K-A-B-D-E-F-M(1)-N-Q-R-K-X-Z-BB-CC
- Caminho 6 --> K-A-B-D-E-F-M(2)-N-P-S-K-X-Z-BB-CC
- Caminho 7 --> K-A-B-D-E-F-M-N(1)-Q-R-S-K-X-Z-BB-CC
- Caminho 8 --> K-A-B-D-E-F-M-N(2)-P-S-K-X-Z-BB-CC
- Caminho 9 --> K-A-B-D-E-F-M-N(3)-P-S-K-X-Y-AA-BB-CC
- Caminho 10 --> k-A-B-D-E-F-M-O-T(1)-U-W-K-X-Z-BB-CC
- Caminho 11 --> k-A-B-D-E-F-M-O-T(2)-V-K-X-Z-BB-CC
- Caminho 12 --> K-X-Y-AA-BB-CC
- Caminho 13 --> K-X-Z-BB-CC

Na figura seguinte são colocados os casos de teste para cada caminho, sendo que por vezes para um caminho são ativados vários casos de teste.

```
public void testParaSufixa() {  
    //caixa branca  
    assertEquals("1 1 + ", CalculadoraSufixa.paraSufixa("( 1 + 1 )")); //Caminho 1 --> )  
    assertEquals("2 3 + ", CalculadoraSufixa.paraSufixa("( 2 + 3 )")); //Caminho 2 --> encontrar vazio apos fazer 1 vez  
    assertEquals("3 4 + ", CalculadoraSufixa.paraSufixa("( 3 + 4 )")); //Caminho 3 --> (  
    assertEquals("3 4 + ", CalculadoraSufixa.paraSufixa("( 3 + 4 )")); //Caminho 4  
    assertEquals("1 1 + ", CalculadoraSufixa.paraSufixa("( 1 + 1 )")); //Caminho 5 --> +  
    assertEquals("2 1 - ", CalculadoraSufixa.paraSufixa("( 2 - 1 )")); //Caminho 6 --> -  
    assertEquals("2 1 - ", CalculadoraSufixa.paraSufixa("( 2 - 1 )")); //Caminho 7 --> vazio depois  
    assertEquals("2 1 * ", CalculadoraSufixa.paraSufixa("( 2 * 1 )")); //Caminho 8 --> vazio depois e *  
    assertEquals("2 1 / ", CalculadoraSufixa.paraSufixa("( 2 / 1 )")); //Caminho 9 --> vazio depois e /  
    assertEquals("2 3 * ", CalculadoraSufixa.paraSufixa("( 2 * 3 )")); //Caminho 10 --> *  
    assertEquals("2 1 / ", CalculadoraSufixa.paraSufixa("( 2 / 1 )")); //Caminho 11 --> /  
    assertEquals("2 1 / ", CalculadoraSufixa.paraSufixa("( 2 / 1 )")); //Caminho 12 --> encontrar vazio apos 1 vez  
    assertEquals("", CalculadoraSufixa.paraSufixa("")); //Caminho 13 --> encontra logo vazio na 1a vez  
}
```

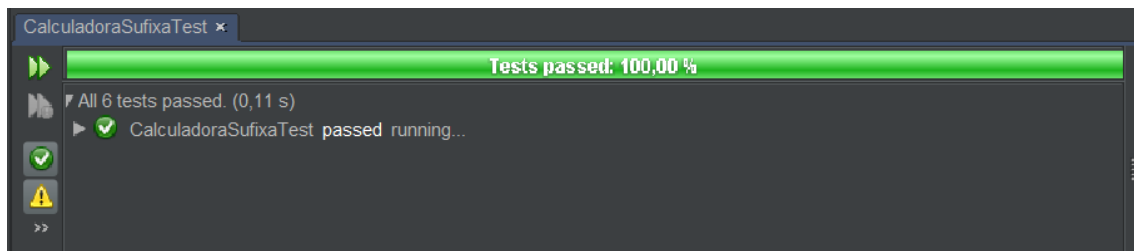
7.2 – Caixa Preta

Para caixa preta foram feitos os seguintes

```
//caixa preta  
assertEquals("1 + ", CalculadoraSufixa.paraSufixa("1 +"));  
assertEquals("1 - ", CalculadoraSufixa.paraSufixa("1 -"));  
assertEquals("1 / ", CalculadoraSufixa.paraSufixa("1 /"));  
assertEquals("1 * ", CalculadoraSufixa.paraSufixa("1 *"));  
  
assertEquals("1 1 + ", CalculadoraSufixa.paraSufixa("1 + 1"));  
assertEquals("2 1 - ", CalculadoraSufixa.paraSufixa("2 - 1"));  
assertEquals("5 5 / ", CalculadoraSufixa.paraSufixa("5 / 5"));  
assertEquals("2 3 * ", CalculadoraSufixa.paraSufixa("2 * 3"));  
  
assertEquals("3 4 * 2 - ", CalculadoraSufixa.paraSufixa("3 * 4 - 2"));  
assertEquals("3 4 / 2 - ", CalculadoraSufixa.paraSufixa("3 / 4 - 2"));  
  
assertEquals("3 4 / 2 - ", CalculadoraSufixa.paraSufixa("( 3 / 4 ) - 2"));  
assertEquals("3 4 - 2 - ", CalculadoraSufixa.paraSufixa("( 3 - 4 ) - 2"));  
assertEquals("3 4 * 2 - ", CalculadoraSufixa.paraSufixa("( 3 * 4 ) - 2"));  
assertEquals("3 4 + 2 - ", CalculadoraSufixa.paraSufixa("( 3 + 4 ) - 2"));  
  
//assertEquals("3 4 * 5 / ", CalculadoraSufixa.paraSufixa("3 * 4 / 5")); //ele faz o 4/5 primeiro nao sei porque  
//assertEquals("3 4 + 2 - ", CalculadoraSufixa.paraSufixa("3 + 4 - 2"));  
//assertEquals("3 4 - 2 - ", CalculadoraSufixa.paraSufixa("3 - 4 - 2"));
```


8 – Conclusão

Todos os testes foram realizados com sucesso



No entanto foi detetado um problema!

```
assertEquals("3 4 * 5 / ", CalculadoraSufixa.paraSufixa("3 * 4 / 5")); //ele faz o 4/5 primeiro nao sei porque
assertEquals("3 4 + 2 - ", CalculadoraSufixa.paraSufixa("3 + 4 - 2"));
assertEquals("3 4 - 2 - ", CalculadoraSufixa.paraSufixa("3 - 4 - 2"));
```

A calculadora faz a segunda parte das expressões em primeiro sempre. Pelo que consegui comprovar não deveria ser assim

Por exemplo: $2-1 \times 5/4 = 1,75$

Colocando na forma sufixa ficaria: $2154/*1+-$

Ao colocar a expressão do exemplo a calculadora converte para $215*4/-1+-$

$215*4/-1+- \neq 2154/*1+-$

Conclui-se por isso que existirá um pequeno problema de código com a calculadora.