Universidade do Minho

Mestrado Integrado em Engenharia Informática

Métodos Formais em Engenharia de Software - 4º ano

Análise e Teste de Software

Fase 3
Métricas e Smells para C– e MSP

Cláudia Ribeiro - A64460
José Ribeiro - A64389
Mário Santos - A64299

20 de Janeiro de 2016

Conteúdo

1 Introdução

De maneira a determinar a qualidade de um certo programa poderão fazer-se "medições" do *software* e, posteriormente, melhorá-lo. A partir disto é também possível fazer uma previsão sobre o sucesso e insucesso do programa em questão. Com as métricas é possível verificar se determinadas características desejadas estão presentes no programa que se está a desenvolver, assim como, apurar se há outras que estão em falta.

Especificamente, com métricas de *software* poderão obter-se resultados objectivos e quantificaveis, tais como, o número de linhas de código, o número de blocos aninhados, o números de parâmetros das funções, entre outras, as quais dão informações tanto sobre o comprimento do programa, como da sua complexidade.

Exprimindo o que se pode entender como *code smells*, estes não devem ser vistos como *bugs*, mas sim como indicadores de algo menos bom nos programas elaborados que devem, portanto, ser melhorados para evitar, por exemplo, que o programa demore mais tempo a executar, assim como, prevenir futuras falhas no *software*.

Os "maus cheiros" podem estar associados a vários factores como, por exemplo, declarações de variavéis que não irão ser utilizadas ao longo do programa, inutilização de argumentos em determinadas funções, blocos aninhados em demasia, entre outros.

2 Enunciado do Projecto

No âmbito da Unidade Curricular de Análise e Teste de Software foi-nos proposto elaborar um projecto que tem como principal objectivo a construção de um catálogo de métricas de código fonte, assim como a detecção de *bad smells*, para duas linguagens, `C--` e `MSP`.

A partir da análise de programas bem construídos, em `C--`, é possível elaborar o catálogo de métricas pois, dessa forma, podemos aferir quais os valores típicos para, por exemplo, o número de argumentos das funções, ou o número de linhas de código, entre outras. O mesmo acontece quando nos referimos à linguagem `MSP`.

Após fixar os vários valores de referência para as métricas, é concebível definir *bad smells* para, posteriormente, fazer a sua detecção, assim como, se possível, fazer *refactoring*, de maneira a eliminá-los.

Para a concretização deste trabalho prático foi utilizado o sistema `TOM`.

2.1 Sistema TOM

O `TOM` pode ser visto como uma extensão a uma linguagem de programação, que foi desenhado para manipular árvores e documentos XML, assim como permite, facilmente, a utilização de *pattern matching* para inspeccionar objectos e devolver valores.

Especificamente no nosso projecto, utilizámos o `TOM` num ambiente `Java`, o que permitiu, para além do referido acima, a utilização de programação estratégica para, por exemplo, a construção das métricas. Foi também possível gerar árvores orientadas ao objecto, denominadas `GOM`.

3 Compromissos para a 1ª Fase

Para esta primeira fase do projecto, propusemo-nos, no início do mês de Novembro, a fazer as seguintes métricas:

- Cyclomatic complexity (McCabe's complexity);
- LOC – lines of code;
- NOF – number of functions/methods;
- NOA – number of arguments of a function.

4 Compromissos para a 2ª Fase

Aquando da entrega da 1ª fase do projecto, um dos objectivos que idealizámos para esta etapa foi o de completar o catálogo de métricas, dando, para cada uma delas, um valor padrão, e dessa forma seria possível cumprir o compromisso de fazer a detecção de `bad smells`. O segundo objectivo passava por construir um *ranking* para cada função, e para o programa na sua totalidade.

5 Compromissos para a 3ª Fase

Para a última fase do projecto os nossos compromissos passam por explorar os *smells*, assim como, fazer o *refactoring* dos mesmos e, para além disso, fazer a leitura dos valores para as métricas a partir de um `.csv`.

6 C–

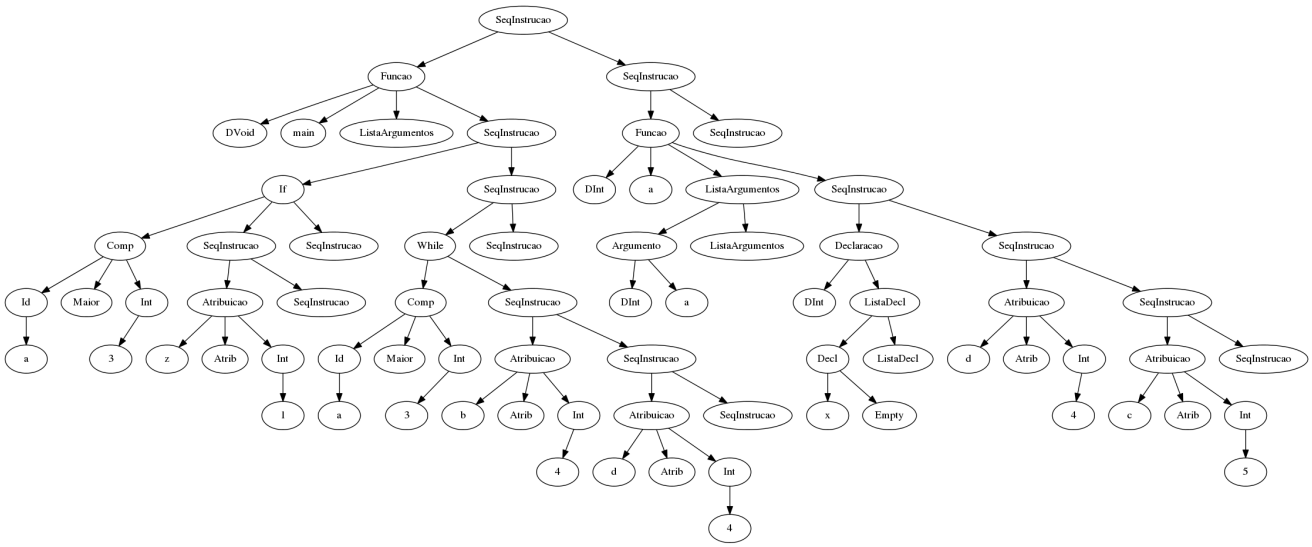
A partir das árvores `GOM` geradas, as instruções são subdivididas por funções, usando um `HashMap<String,Instrucao>` e a partir de um processamento de informação *TopDown*, é usada uma estratégia para, desta maneira, ser possível ler as métricas por função.

```
‘TopDown(visitFuncoes(main.funcoesInst)).visit(p);
```

```
%strategy visitFuncoes(funcoes:HashMap) extends Identity() {
      visit Instrucao {
          Funcao(tipo,nome,argumentos,inst) -> {
              funcoes.put(‘nome, ‘inst);
        }
     }
}
```

Para melhorar o funcionamento e a facilitar a visualização das árvores, decidimos eliminar os comentários do sistema (da gramática e da árvore `GOM`).

De seguida, é demonstrada a árvore gerada (já sem os comentários) a partir de um dos ficheiros exemplo.



6.1 Métricas

Nesta secção serão descritas as métricas que elaborámos para esta primeira fase do trabalho prático, no que diz respeito ao código `C--`.

6.1.1 Métrica *LOC - Lines of Code*

Uma vez que para a métrica em questão não existe uma definição standard, esta foi calculada contando o número de vezes que cada instrução aparece. Por exemplo, no caso da instrução em questão ser uma declaração ou uma atribuição conta como 1. Já, no caso de ser um `if` o número de linhas, neste caso instruções, é calculado recursivamente, pois é necessário somar o número de instruções que estão dentro dessa condição.

Tanto nos ciclos `while`, `for`, como em `if`'s são somadas, ao número total de linhas, 2 unidades, que correspondem à abertura e fecho das instruções em questão.

Com o programa que elaborámos é mostrado, para cada ficheiro lido, o número de linhas por função/método, e o número total de linhas de todo o ficheiro.

```
1  private static int linesOfCode(Instrucao i) {
2      int aux = 0;
3      %match(i) {
4          Atribuicao(id,opAtrib,exp) -> { return 1;}
5          Declaracao(tipo,decl) -> { return 1;}
6          If(condicao,inst1,inst2) -> { if(linesOfCode(‘inst2)>0)
               aux=2; return (linesOfCode(‘inst1)+2)+(linesOfCode(‘
               inst2)+aux);}
7          While(condicao,inst) -> { return linesOfCode(‘inst)+2;}
8          For(decl,condicao,exp,inst) -> { return linesOfCode(‘inst)
               +2;}
```

```
9           Return(exp) -> { return 1;}
10          Funcao(tipo,nome,argumentos,inst) -> { return linesOfCode
                (`inst);}
11          Exp(exp) -> { return 1;}
12          SeqInstrucao(inst1, inst*) -> { return linesOfCode(`inst1)
                +linesOfCode(`inst*);}
13          }
14      return aux;
15  }
```

6.1.2 Métrica *NOF - Number of Functions*

Tal como o nome indica, esta métrica é o resultado do número de funções de determinado programa, que se encontre em análise.

No caso do nosso projecto, o resultado para esta métrica é o tamanho de um `HashMap`, que contém cada função e as correspondentes instruções, ou seja, o número de funções é o número de chaves contidas nesse `HashMap`.

6.1.3 Métrica *NOA - Number of Arguments (of a function)*

Com facilidade se percebe que esta métrica é o resultado do número de argumentos de uma função.

```
1   private static int foundArgs(String funcao, Instrucao i) {
2       %match(i) {
3           Funcao(tipo,nome,argumentos,inst) -> { if(funcao.equals(`
                nome)) return foundListArgs(`argumentos); }
4           SeqInstrucao(inst1, inst*) -> { return foundArgs(funcao, `
                inst1)+foundArgs(funcao, `inst*);}
5       }
6       return 0;
7   }
8
9   private static int foundListArgs(Argumentos args) {
10      %match(args) {
11          ListaArgumentos(arg1,tailArg*) -> { return foundListArgs(`
                arg1)+foundListArgs(`tailArg*); }
12          Argumento(_,idArg) -> { return 1; }
13      }
14      return 0;
15  }
```

6.1.4 Métrica *Número de Declarações*

O resultado desta métrica é, tal como o nome indica, o número de declarações que são feitas em cada função do programa.

```
1 private static int foundDecl(Instrucao i) {
2     %match(i) {
3         Declaracao(tipo,decl) -> { return 1; }
4         SeqInstrucao(inst1, inst*) -> { return foundDecl(`inst1)+
              foundDecl(`inst*);}
5     }
6     return 0;
7 }
```

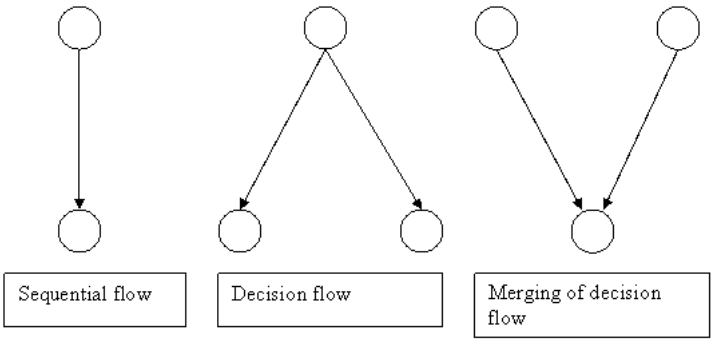
6.1.5 Métrica *Cyclomatic Complexity (McCabe's complexity)*

Como se pode constatar através da sua designação, esta métrica permite saber qual a complexidade de um programa, tal é feito contando o número de fluxos de um pedaço de código.

A complexidade ciclomática "mede" a quantidade de decisões que podem ser tomadas num programa, dando o número mínimo de caminhos que possam levar a outros caminhos do programa.

Tendo em conta esta informação, para fazer o cálculo desta métrica utilizámos a fórmula $D + 1$, sendo que o `D` é o número de pontos de decisão.

Apresentamos de seguida um gráfico de controlo de fluxo, no qual cada nodo representa uma região do programa, e as arestas representam o fluxo do programa de uma região para outra.



Os operadores booleanos podem ou não ser um factor de adição na complexidade, dependendo se geram ou não caminhos, respectivamente.

Posto isto, é possível demonstrar o código elaborado, que respeita o que foi dito até agora sobre a métrica Cyclomatic Complexity.

```
private static int foundCC(Instrucao i) {
    %match(i) {
        If(condicao,inst1,inst2) -> { return 1+foundCC(`inst1)+
           foundCC(`inst2)+foundBoolean(`condicao);}
        While(condicao,inst) -> { return 1+foundCC(`inst)+
           foundBoolean(`condicao);}
        For(decl,condicao,exp,inst) -> { return 1+foundCC(`inst)+
           foundBoolean(`condicao);}
        SeqInstrucao(inst1, inst*) -> { return foundCC(`inst1)+
           foundCC(`inst*);}
    }
    return 0;
}

/* Numero de operacoes booleanas em condicoes */
private static int foundBoolean(Expressao e) {
    %match(e) {
        E(cond1,cond2) -> { return 1+foundBoolean(`cond1)+
           foundBoolean(`cond2);}
        Ou(cond1,cond2) -> { return 1+foundBoolean(`cond1)+
           foundBoolean(`cond2);}
    }
    return 0;
}
```

6.1.6 Métrica *Nested Block Depth*

O objectivo desta métrica é saber qual a profundidade de blocos aninhados.
No programa que elaborámos, esta métrica é calculada de forma recursiva, ou seja, enquanto houver instruções irá sempre ser verificado se esta tem outras instruções aninhadas, devolvendo no fim o número máximo de blocos aninhados para determinada função.

```
private static int foundNested(Instrucao i) {
    %match(i) {
        If(condicao,inst1,inst2) -> { return 1+max(foundNested(`
           inst1),foundNested(`inst2));}
        While(condicao,inst) -> { return foundNested(`inst)+1;}
        For(decl,condicao,exp,inst) -> { return foundNested(`inst)
           +1;}
        SeqInstrucao(inst1, inst*) -> { return max(foundNested(`
           inst1),foundNested(`inst*));}
    }
    return 0;
}
```

6.2 *Smells*

Para a detecção de ”maus cheiros” foram definidos valores máximos para cada métrica. Portanto, se uma determinada métrica ultrapassa o valor máximo definido, é detectado um *smell*, tal como é possível verificar no exemplo apresentado de seguida.

Os valores considerados foram os seguintes:

- Número máximo de linhas (`maxLinhas`): 15;
- Número máximo de declarações (`maxDecl`): 5;
- Número máximo de argumentos (`maxArgs`): 3;
- Número máximo de blocos aninhados (`maxNested`): 3;
- Complexidade ciclomática máxima (`maxCC`): 5.

Considerando que `funcoesNested` é um `HashMap<String, Integer>` onde são guardadas as várias funções (chave - String) e o corresponde número máximo de blocos aninhados que esta tem (valor - Integer), é detectado um *smell* se o número máximo de blocos aninhados de uma função for maior que o definido em `maxNested`.

```
public int classificaSmellNested(String s) {
    int a, res = 0;
    if((a = this.funcoesNested.get(s)) > this.maxNested){
        res = 1;
        this.nSmells++;
    }
    return res;
}
```

Posto isto, se for detectado um ”mau cheiro” é mostrado no ecrã o seguinte:

```
if(met.classificaSmellNested(s) == 1){
    System.out.println("** Smell Detectado: Bloco com demasiados
       aninhamentos **\n");
}
```

Para os restantes valores definidos, a detecção de *bad smells* é feita de forma semelhante.

No final do programa é também possível visualizar o número total de *smells* detectados.

6.3 *Star Ranking*

Nesta secção irão ser descritos os passos para a construção do *Star Ranking* que foi elaborado de forma a completar o catálogo de métricas, sendo possível classificar os programas testados.

Os Rankings podem ser recolhidos através de um ficheiro *.csv*, que contém todos os rankings necessários para comparar as métricas dos nossos programas. Os rankings que temos são: linhas, declarações, argumentos, bloco e a complexidade ciclomática.

linhas	declaracoes	argumentos	bloco	cyclomatic
15	5	3	3	5

Figura 1: Exemplo de limites do Star Ranking

A cada função é associado o seu *ranking*, sendo isto guardado num `HashMap<String,Double> funcoesRank` (funcao, ranking).

A classificação de uma determinada função é feita da seguinte forma:

```
1  public Double classificaFuncao(String s){
2      Double aux=0.0;
3      int a;
4
5      if((a = this.funcoesLinhas.get(s)) <= this.maxLinhas)
6          aux+=0.5;
7      else
8          aux+=((maxLinhas*0.5)/a);
9
10     if((a = this.funcoesDecl.get(s)) <= this.maxDecl)
11         aux+=0.5;
12     else
13         aux+=((maxDecl*0.5)/a);
14
15     if((a = this.funcoesArgs.get(s)) <= this.maxArgs)
16         aux+=1;
17     else
18         aux+=((maxArgs*1)/a);
19
20     if((a = this.funcoesNested.get(s)) <= this.maxNested)
21         aux+=1.5;
22     else
23         aux+=((maxNested*1)/a);
24
25     if((a = this.funcoesCC.get(s)) <= this.maxCC)
26         aux+=1.5;
27     else
```

```
28          aux+=((maxCC*1)/a);
29
30      return aux;
31  }
```

Isto é, para cada função é comparado o seu número de declarações, argumentos, etc., e é verificado se este número ultrapassa o número máximo estabelecido. No caso de não extrapolar esse valor é somado o valor correspondente à métrica em questão, caso contrário, é feito um cálculo que faz com que o ranking dessa função descresça.

A classificação de todo o programa é obtida da maneira seguinte:

```
1   public String getRank(){
2       Double aux = 0.0;
3
4       for(String s : this.funcoesRank.keySet()){
5           aux+=this.funcoesRank.get(s);
6       }
7
8       DecimalFormat df = new DecimalFormat("#.#");
9       df.setRoundingMode(RoundingMode.FLOOR);
10
11      return df.format(aux/(this.funcoes));
12  }
```

Ou seja, o valor obtido no final irá ser ser a soma do *ranking* de todas as funções que constam em `funcoesRank`, dividindo esse valor pelo número de funções que o programa em questão tem.

Os valores que o *Star Ranking* pode tomar variam entre 1 e 5, sendo que 5 é o melhor e o 1 é o pior e é feito da seguinte forma:

- Número de Linhas: 0,5 valores;
- Número de declarações: 0,5 valores;
- Némero de argumentos: 1 valor;
- Número de blocos aninhados: 1,5 valores;
- *Cyclomatic Complexity*: 1,5 valores.

6.4 *Refactoring*

Através da *estratégia* que se segue é possível obter algo como:
(O programa utilizado para os exemplos que se seguem encontra-se em anexo, no final do relatório).

```
********** Smells ************
Decl("a",Empty()) Utilizado
Decl("b",Empty()) Utilizado
Decl("res",Empty()) Utilizado
Decl("c",Empty()) Não Utilizado
Id("a") Utilizado
Id("b") Utilizado
Id("c") Não Utilizado
Argumento(DInt(),"a") Utilizado
Argumento(DInt(),"b") Utilizado
Argumento(DInt(),"c") Não Utilizado
Decl("res",Empty()) Utilizado
```

Ou seja, com esta `strategy` e os métodos auxiliares que esta invoca, é possível fazer o *refactoring* quando aparecem *bad smells* tais como: `if's` em que a condição é negada; a não utilização de determinados argumentos; variavéis declaradas que não são usadas posteriormente, parametros, em expressões, que não são utilizados. Por último, também respeitante aos `if's`, se condição for absoluta, por exemplo, se esta for um `true` o programa apaga o `if` e deixa só as instruções que estão dentro deste, se a condição do `if` for `false`, ficam as instruções do `else` (se ele existir). Para o *smell* descrito em primeiro lugar, é retirada a negação e as instruções são trocadas (por exemplo, `if(!a) then b else c` irá ser transformado em `if(a) then c else b`). Já nos outros tipos de *bad smells* irão ser retirados os argumentos, as declarações e os parametros não utilizados.

```
1  %strategy stratBadSmells(Set idsUtilizados) extends Identity() {
2      visit Instrucao {
3          If(Nao(condicao),inst1,inst2) -> {
4              return `If(condicao,inst2,inst1);
5          }
6          Funcao(tipo,nome,argumentos,inst) -> {
7              idsUtilizados = new TreeSet<String>();
8              `TopDown(stratCollectIds(idsUtilizados, 0)).visit(`
                   inst);
9              Argumentos args = removeArgumentosNaoUtilizados(`
                   argumentos, idsUtilizados);
10             return `Funcao(tipo,nome,args,inst);
11         }
12         Declaracao(tipo,decls) -> {
```

```
13              Boolean decl = verificaDeclaracoesNaoUtilizados(‘decls
                    , idsUtilizados);
14              if(decl)
15                  return ‘Declaracao(tipo, decls);
16              else
17                  return ‘SeqInstrucao(Exp(Empty()));
18          }
19      }
20      visit Expressao{
21          Call(id,param)->{
22              Parametros para = removeParametrosNaoUtilizados(‘param
                    , idsUtilizados);
23              return ‘Call(id,para);
24          }
25      }
26  }
```

Pegando no exemplo acima, o resultado final irá ser o seguinte:

```
********** Smells ************
Decl("a",Empty()) Utilizado
Decl("b",Empty()) Utilizado
Decl("res",Empty()) Utilizado
Id("a") Utilizado
Id("b") Utilizado
Argumento(DInt(),"a") Utilizado
Argumento(DInt(),"b") Utilizado
Decl("res",Empty()) Utilizado
```

Outro exemplo para demonstrar como é feito o *refactoring* pode observar-se de seguida, onde inicialmente temos o ficheiro em `C--` sem *refactoring* e depois é apresentado o ficheiro já com as devidas ”refabricações” efectuadas.

```
 1 void main() {
 2     int a;
 3     int res;
 4     int c;
 5     a = input(int);
 6     res = refac(a,c);
 7     print(';');
 8     print(res);
 9 }
10
11 int refac(int a, int c){
12     int res;
13     if (1 || 0) {
14         res = a;
15     }
16     else {
17         res = c;
18     }
19     return res;
20 }
```

```
 1 void main() {
 2     int a;
 3     int res;
 4     a = input(int);
 5     res = refac(a);
 6     print(';');
 7     print(res);
 8 }
 9
10 int refac(int a){
11     int res;
12     res = a;
13     return res;
14 }
```

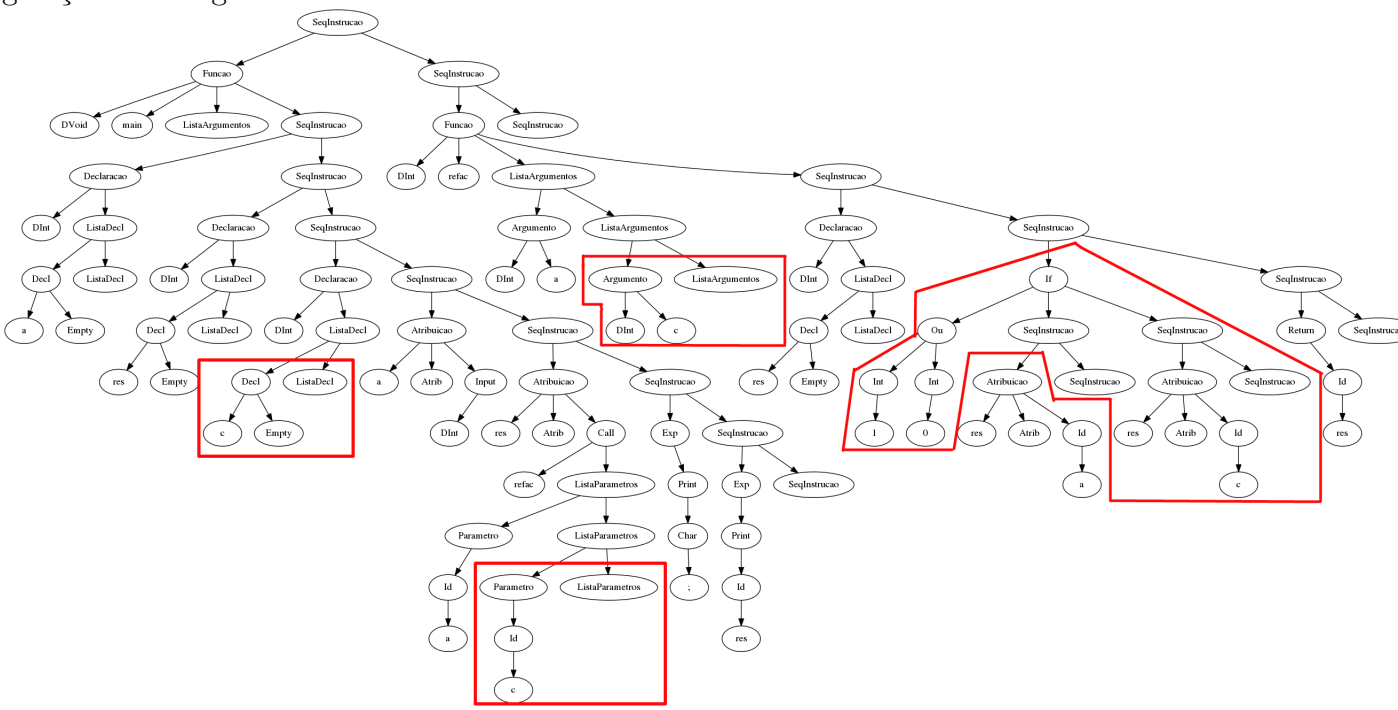
No exemplo mostrado acima é possível verificar os vários *refactorings* que são realizados, sendo eles:

- **if com condição absoluta:** na função `refact` existe um `if` no qual a condição é `1 || 0`, ou seja, para todas as execuções, apenas a instrução `res = a` vai ser utilizada, logo é eliminado todo o bloco de código correspondente ao `else`;
- **Argumentos não utilizados**: devido ao *refactoring* descrito acima, o argumento `c` nunca é utilizado, logo é retirado;

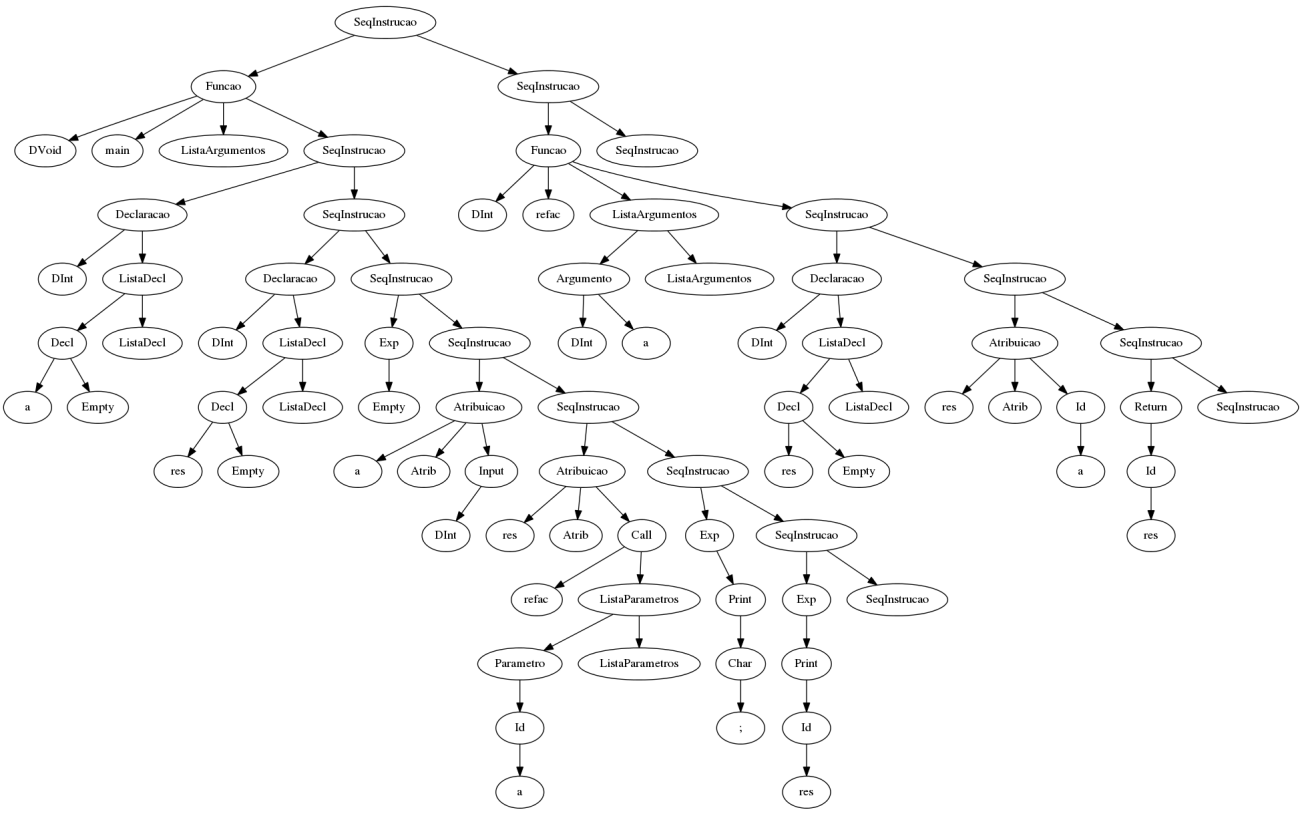
- **Parâmetros não utilizados**: em consequência da eliminação do argumento da função `refact`, na função `main` o parâmetro `c` não vai ser usado aquando da invocação da função `refact`, pois esta não necessita dele, logo este também vai ser removido;

- **Declarações não utilizadas**: Eliminação da declaração da variável `c` pois esta nunca é utilizado ao longo do programa, devido aos *refactorings* efectuados anteriormente.

Na imagem que se segue, a vermelho estão assinalados os elementos que irão ser "refabricados" que, neste caso, corresponde a serem eliminados, tanto na árvore `GOM`, como na geração de código `MSP`.



Árvore `GOM` gerada após o *refactoring*.


SeqInstrucao
Funcao
SeqInstrucao
DVoid
main
ListaArgumentos
SeqInstrucao
Funcao
SeqInstrucao
Declaracao
SeqInstrucao
DInt
refac
ListaArgumentos
SeqInstrucao
DInt
ListaDecl
Declaracao
SeqInstrucao
Argumento
ListaArgumentos
Declaracao
SeqInstrucao
Decl
ListaDecl
DInt
ListaDecl
Exp
SeqInstrucao
DInt
a
DInt
ListaDecl
Atribuicao
SeqInstrucao
a
Empty
Decl
ListaDecl
Empty
Atribuicao
SeqInstrucao
Decl
ListaDecl
res
Atrib
Id
Return
SeqInstrucao
res
Empty
a
Atrib
Input
Atribuicao
SeqInstrucao
res
Empty
a
Id
DInt
res
Atrib
Call
Exp
SeqInstrucao
res
refac
ListaParametros
Print
Exp
SeqInstrucao
Parametro
ListaParametros
Char
Print
Id
;
Id
a
res

7 MSP

7.1 Métricas

Na parte que se segue estará exposto o trabalho que realizámos para o código `MSP`. Para esta parte do trabalho, como as métricas anteriores de `C--` contêm métricas mais complexas, nesta parte mantivemos apenas o número de ocorrências de cada tipo de instrução *.msp*. As instruções todas foram guardas num `HashMap<String,Interger> metricas`, em que a chave é o nome da instrução e o valor é o número de ocorrências dessa mesma instrução. Estas métricas foram criadas separadamente pois era desnecessário percorrer todo o tipo de instruções se o utilizador apenas pediu um determinado tipo de instruções.

7.1.1 Métricas *Declarações*

Nesta métrica, verificamos que quando existem declarações, quer sejam de funções ou de variáveis, tem como instrução "Decl". Para contar o número de declarações, foi criada uma *strategy* do tipo ilustrado em baixo:

```
%strategy visitDecl(metricas:HashMap,n:int) extends Identity() {
    visit Instrucao {
        Decl(id,initMemAddress,size) -> {
                n++;
                metricas.put("Decl",n);
        }
    }
}
```

Esta métrica visita intruções do tipo "Decl" incrementa o contador e vai inserindo no *HashMap* esse valor.

7.1.2 Métrica *Aritméticas*

Com esta métrica queremos contar o número de contas aritméticas que o código pode ter. Para isso, utilizamos uma *strategy* semelhante ao acima representado, com o seguinte aspeto.

```
%strategy visitDecl(metricas:HashMap,n:int) extends Identity() {
    %strategy visitAritm(metricas:HashMap,add:int,sub:int,mul:int,
       div:int,mod:int) extends Identity() {
        visit Instrucao {
                Add() -> {
                    add++;
                    metricas.put("Add",add);
                }
                Sub() -> {
                    sub++;
```

```
                    metricas.put("Sub",sub);
                }
                Div() -> {
                    div++;
                    metricas.put("Div",div);
                }
                Mul() -> {
                    mul++;
                    metricas.put("Mul",mul);
                }
                Mod() -> {
                    mod++;
                    metricas.put("Mod",mod);
                }
        }
}
```

7.1.3 Métrica *Condicionais*

De seguida, mostramos a *strategy* que lê quantas instruções do tipo condicional, como por exemplo, maior ou igual, igual, menor que, etc.

```
%strategy visitRelationals(metricas:HashMap,eq:int,neq:int,gt:int,
   goEq:int,lt:int,loEq:int,nott:int) extends Identity() {
    visit Instrucao {
        Eq() -> {
            eq++;
            metricas.put("Eq",eq);
        }
        Neq() -> {
            neq++;
            metricas.put("Neq",neq);
        }
        Gt() -> {
            gt++;
            metricas.put("Gt",gt);
        }
        GoEq() -> {
            goEq++;
            metricas.put("GoEq",goEq);
        }
        Lt() -> {
            lt++;
            metricas.put("Lt",lt);
        }
```

```
23          LoEq() -> {
24              loEq++;
25              metricas.put("LoEq",loEq);
26          }
27          Nott() -> {
28              nott++;
29              metricas.put("Nott",nott);
30          }
31      }
32  }
```

7.1.4 Métrica *Relacionais*

Esta métrica conta o número de operações relacionais encontradas no ficheiro. Sempre que encontra um "And" ou um "Or", representando respetivamente "&&"e |", incrementa os contadores de cada tipo.

```
1  %strategy visitConditionals(metricas:HashMap,and:int,or:int)
       extends Identity() {
2      visit Instrucao {
3          And() -> {
4              and++;
5              metricas.put("And",and);
6          }
7          Or() -> {
8              or++;
9              metricas.put("Or",or);
10         }
11     }
12 }
```

7.1.5 Métrica *I/O*

Por fim , esta métrica calcula número de ocorrências que *inputs* e *outputs* que o programa contém.

```
1  %strategy visitIO(metricas:HashMap,in:int,out:int) extends
       Identity() {
2      visit Instrucao {
3          IIn(tipo) -> {
4              in++;
5              metricas.put("IIn",in);
6          }
7          IOut() -> {
8              out++;
```

```
9                 metricas.put("IOut",out);
10            }
11        }
12    }
```

7.2 Comparações Original-*Refactored*

Nesta parte do MSP, ficou decicido que se faria a comparação das métricas entre um ficheiro sem *refactoring* e um ficheiro já *refactored*, obviamente, do mesmo programa, sendo que tanto um código, como outro é obtido através da geração de código C-- para MSP (o ficheiro utilizado foi o mesmo que usámos na parte de C--).

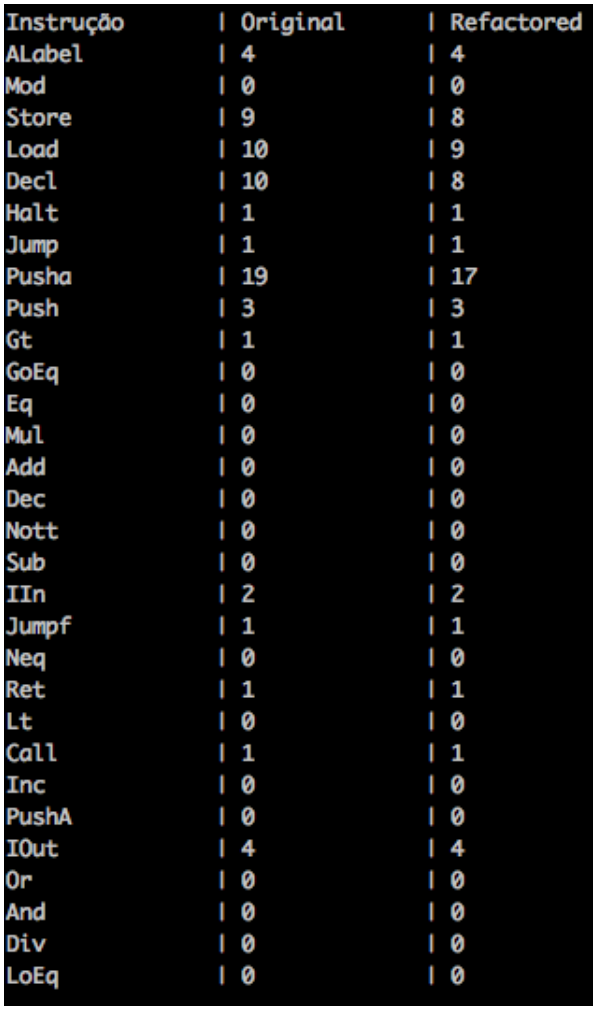
Considerando o seguinte código MSP:

```
1 Decl "f:main" 0 1,Decl "main_a" 1 1,Decl "main_b" 2 1,Decl "
    main_res" 3 1,Decl "main_c" 4 1,Decl "f:max" 5 1,Decl "max_a" 6
     1,Decl "max_b" 7 1,Decl "max_c" 8 1,Decl "max_res" 9 1,Pushi
    0,IOut,Pushc '#',IOut,ALabel "f:main",Pusha "main_a",IIn int,
    Store,Pusha "main_b",IIn int,Store,Pusha "main_res",Pusha "
    max_a",Pusha "main_a",Load,Store,Pusha "max_b",Pusha "main_b",
    Load,Store,Pusha "max_c",Pusha "main_c",Load,Store,Call "f:max"
    ,Pusha "f:max",Load,Store,Pushc ';',IOut,Pusha "main_res",Load,
    IOut,Halt,ALabel "f:max",Pusha "max_a",Load,Pusha "max_b",Load,
    Gt,Jumpf "senao1",Pusha "max_res",Pusha "max_a",Load,Store,Jump
     "fse1",ALabel "senao1",Pusha "max_res",Pusha "max_b",Load,
    Store,ALabel "fse1",Pusha "f:max",Pusha "max_res",Load,Store,
    Ret
```

O respectivo código, já com *refactoring*, é:

```
1 Decl "f:main" 0 1,Decl "main_a" 1 1,Decl "main_b" 2 1,Decl "
    main_res" 3 1,Decl "f:max" 4 1,Decl "max_a" 5 1,Decl "max_b" 6
    1,Decl "max_res" 7 1,Pushi 0,IOut,Pushc '#',IOut,ALabel "f:main
    ",Pusha "main_a",IIn int,Store,Pusha "main_b",IIn int,Store,
    Pusha "main_res",Pusha "max_a",Pusha "main_a",Load,Store,Pusha
    "max_b",Pusha "main_b",Load,Store,Call "f:max",Pusha "f:max",
    Load,Store,Pushc ';',IOut,Pusha "main_res",Load,IOut,Halt,
    ALabel "f:max",Pusha "max_a",Load,Pusha "max_b",Load,Gt,Jumpf "
    senao1",Pusha "max_res",Pusha "max_a",Load,Store,Jump "fse1",
    ALabel "senao1",Pusha "max_res",Pusha "max_b",Load,Store,ALabel
     "fse1",Pusha "f:max",Pusha "max_res",Load,Store,Ret
```

O output correspondente:



Instrução	Original	Refactored
ALabel	4	4
Mod	0	0
Store	9	8
Load	10	9
Decl	10	8
Halt	1	1
Jump	1	1
Pusha	19	17
Push	3	3
Gt	1	1
GoEq	0	0
Eq	0	0
Mul	0	0
Add	0	0
Dec	0	0
Nott	0	0
Sub	0	0
IIn	2	2
Jumpf	1	1
Neq	0	0
Ret	1	1
Lt	0	0
Call	1	1
Inc	0	0
PushA	0	0
IOut	4	4
Or	0	0
And	0	0
Div	0	0
LoEq	0	0

Figura 2: Comparações das métricas entre o ficheiro maiorDeDoisNumeros.msp original e refactored.

Nesta imagem, é possível observar as diferenças mencionadas acima, isto é, depois de fazer *refactoring*, o número de **Load**, **Store**, **Decl** e **Pusha** reduziram em relação ao ficheiro original.

8 Conclusão

Começámos este projecto fazendo uma gramática que ia fazer o parsing aos ficheiros gerados, no entanto, depois de falarmos com o Professor sobre a nossa interpretação e proposta de resolução do problema, chegámos à conclusão que nos encontrávamos no caminho errado e fomos obrigados a recomeçar. Posto isto, pegámos no código fornecido e adaptámo-lo, fazendo as alterações necessárias, maioritariamente, nos ficheiros `Main.t` que nos permitissem obter os resultados para as métricas.

Sobre os compromissos para a primeira entrega, é-nos possível afirmar que foram todos cumpridos e, para além disso, conseguimos também elaborar soluções para mais métricas, tais como, a *Nested Block Depth*. Para as métricas `MSP`, implementámos aquelas que nos pareceram mais adequadas, tendo em conta a estrutura da linguagem.

Após a elaboração da primeira etapa do projecto, é possível concluir que já nos sentimos mais familiarizados com o sistema `TOM`, e a maneira como este deve ser manuseado.

No que diz respeito aos compromissos para a segunda fase do projecto, podemos dizer que foram todos cumpridos, pois completámos o catálogo de métricas com o *ranking*, assim como, o nosso programa já faz a detecção de *smells* no código. Para além disso, desenvolvemos também soluções para fazer *refactoring* ao código com *bad smells*, apesar das dificuldades que nos trouxeram.

Por fim, com a fase três, conseguimos finalizar o projeto, pois alcançámos os compromissos prometidos, em junção às funcionalidades que o professor aconselhou para aprimorar o programa.

Visto que nos sentimos à vontade para o desenvolvimento deste trabalho, para além de implementar todas as funcionalidades pedidas, foi interessante criar um programa que verifique todas as métricas de um código, bem como optimizá-lo, deixando-o mais limpo e mais legível.

9 Anexos

De seguida são apresentados ficheiros exemplo para mostrar a execução do programa elaborado, tanto para `C--` como para `MSP`.

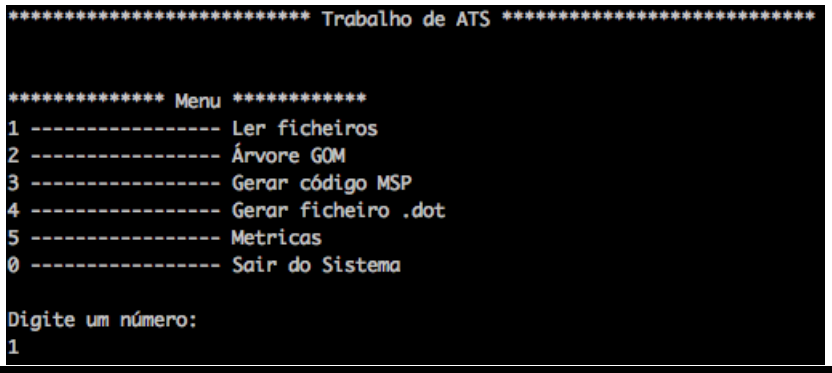
9.1 Métricas

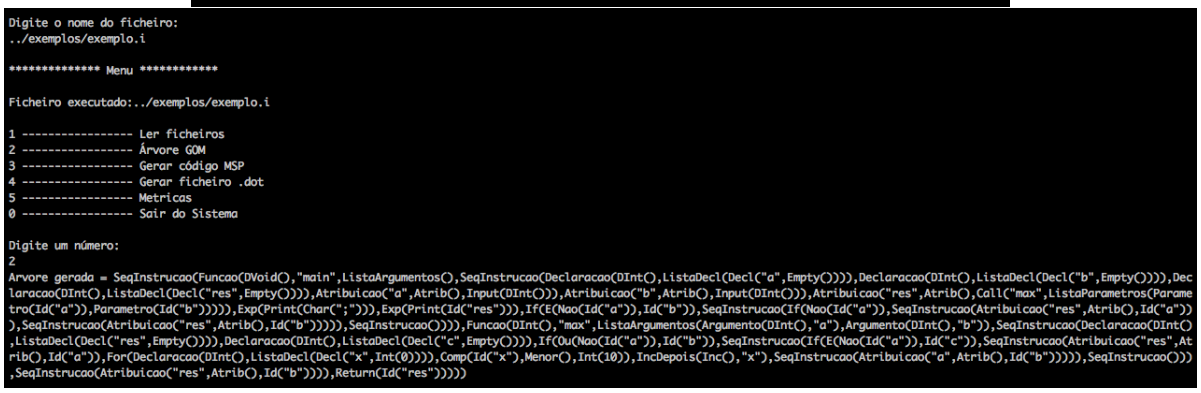
9.1.1 Exemplo C–

```
void main() {
  int a;
  int b;
  int res;

  a = input(int);
  b = input(int);
  res = max(a,b);

  print(';');
  print(res);

  if(!a && b) {
    if(!a) {
      res = a;
    }
    else {
      res = b;
    }
  }
}


int max(int a, int b) {
  int res;
  int c;

  if(!a || b) {
    if(!a && c) {
      res = a;
      for(res = 0; res < 10 ; res++){
        a = b;
      }
    }
  }
  else{
```

```
37      res = b;
38    }
39    return res;
40  }
```

Abaixo são mostradas imagens dos vários menus que são apresentados durante a execução do programa, assim como os resultados obtidos para as métricas.





```
Digite um número:
5

********** Métricas ***********

Número de funcoes: 2
1 ---------------- Linhas
2 ---------------- Variaveis Declaradas
3 ---------------- Numero de Argumentos
4 ---------------- Blocos Aninhados
5 ---------------- Cyclomatic Complexity
6 ---------------- Todas as Métricas
0 ---------------- Voltar
6

----> Funcao: max
Numero de Linhas: 14
Numero de Declaracoes: 2
Numero de Argumentos: 2
Maior Bloco Aninhado: 3
Cyclomatic Complexity: 6

----> Funcao: main
Numero de Linhas: 16
Numero de Declaracoes: 3
Numero de Argumentos: 0
Maior Bloco Aninhado: 2
Cyclomatic Complexity: 4


Total de Linhas: 34
Total de Declaracoes: 5
Total de Argumentos: 2
```

9.1.2 Exemplo MSP

O ficheiro utilizado para gerar as métricas para MSP, foi o seguinte, que corresponde ao cálculo do fatorial de um número:

```
1 Decl "f:main" 0 1,Decl "main_num" 1 1,Decl "main_resI" 2 1,Decl "
    main_resR" 3 1,Decl "main_aux" 4 1,Decl "f:fatorialI" 5 1,Decl
    "fatorialI_num" 6 1,Decl "fatorialI_fat" 7 1,Pusha "
    fatorialI_fat",Pushi 1,Store,Decl "f:fatorialR" 8 1,Decl "
    fatorialR_num" 9 1,Decl "f:fatorialT" 10 1,Decl "fatorialT_num"
     11 1,Pushi 0,IOut,Pushc '#',IOut,ALabel "f:main",Pusha "
    main_num",IIn int,Store,Pusha "main_resI",Pusha "fatorialI_num"
    ,Pusha "main_num",Load,Store,Call "f:fatorialI",Pusha "f:
    fatorialI",Load,Store,Pusha "main_resR",Pusha "fatorialR_num",
    Pusha "main_num",Load,Store,Call "f:fatorialR",Pusha "f:
    fatorialR",Load,Store,Pusha "main_aux",Pusha "fatorialT_num",
    Pusha "main_num",Load,Store,Call "f:fatorialT",Pusha "f:
    fatorialT",Load,Store,Pusha "main_resI",Load,IOut,Pusha "
    main_resR",Load,IOut,Pusha "main_aux",Load,IOut,Halt,ALabel "f:
```

```
fatorialI",Pusha "fatorialI_num",Load,Not,Jumpf "senao2",ALabel
 "for1",Pusha "fatorialI_num",Load,Pushi 1,Gt,Jumpf "ffor1",
Pusha "fatorialI_fat",Pusha "fatorialI_fat",Load,Pusha "
fatorialI_num",Load,Mul,Store,Pusha "fatorialI_num",Dec,Jump "
for1",ALabel "ffor1",Jump "fse2",ALabel "senao2",ALabel "fse2",
Pusha "f:fatorialI",Pusha "fatorialI_fat",Load,Store,Ret,ALabel
 "f:fatorialR",Pusha "fatorialR_num",Load,Pushi 1,Eq,Pusha "
fatorialR_num",Load,Pushi 0,Eq,Or,Jumpf "senao3",Pusha "f:
fatorialR",Pushi 1,Store,Jump "fse3",ALabel "senao3",Pusha "f:
fatorialR",Pusha "fatorialR_num",Load,Pusha "fatorialR_num",
Pusha "fatorialR_num",Load,Pushi 1,Sub,Store,Call "f:fatorialR"
,Pusha "f:fatorialR",Load,Mul,Store,ALabel "fse3",Ret,ALabel "f
:fatorialT",Pusha "fatorialT_num",Load,Pushi 1,Eq,Pusha "
fatorialT_num",Load,Pushi 0,Eq,Or,Jumpf "senao4",Pusha "f:
fatorialT",Pushi 1,Store,Jump "fse4",ALabel "senao4",Pusha "f:
fatorialT",Pusha "fatorialT_num",Load,Pusha "fatorialR_num",
Pusha "fatorialT_num",Load,Pushi 1,Sub,Store,Call "f:fatorialR"
,Pusha "f:fatorialR",Load,Mul,Store,ALabel "fse4",Ret
```

Os resultados obtidos foram os apresentados nas imagens que se seguem:

Métricas *Declarações*

```
********** Menu Métricas *********
1 ----------------- Ler métricas todas
2 ----------------- Métricas Declarações
3 ----------------- Métricas Aritméticas
4 ----------------- Métricas Condicionais
5 ----------------- Métricas Relacionais
6 ----------------- Métricas I/O
0 ----------------- Voltar atrás

Digite um número:
2
#Declarações = 9
```

Métricas *Aritméticas*

```
********** Menu Métricas *********
1 ----------------- Ler métricas todas
2 ----------------- Métricas Declarações
3 ----------------- Métricas Aritméticas
4 ----------------- Métricas Condicionais
5 ----------------- Métricas Relacionais
6 ----------------- Métricas I/O
0 ----------------- Voltar atrás

Digite um número:
3
#Adições = 0
#Subtrações = 1
#Multiplicações = 2
#Divisões = 0
#Resto = 0
```

Métricas *Condicionais*

```
********** Menu Métricas *********
1 ----------------- Ler métricas todas
2 ----------------- Métricas Declarações
3 ----------------- Métricas Aritméticas
4 ----------------- Métricas Condicionais
5 ----------------- Métricas Relacionais
6 ----------------- Métricas I/O
0 ----------------- Voltar atrás

Digite um número:
4
#And = 0
#Or = 1
```

Métricas ***Relacionais***

```
********** Menu Métricas *********
1 ----------------- Ler métricas todas
2 ----------------- Métricas Declarações
3 ----------------- Métricas Aritméticas
4 ----------------- Métricas Condicionais
5 ----------------- Métricas Relacionais
6 ----------------- Métricas I/O
0 ----------------- Voltar atrás

Digite um número:
5
#Igualdades = 2
#Diferentes = 0
#Maior = 1
#Maior ou Igual = 0
#Menor = 0
#Menor ou Igual = 0
#Negação = 1
```

Métricas *I/O*

```
********** Menu Métricas *********
1 ----------------- Ler métricas todas
2 ----------------- Métricas Declarações
3 ----------------- Métricas Aritméticas
4 ----------------- Métricas Condicionais
5 ----------------- Métricas Relacionais
6 ----------------- Métricas I/O
0 ----------------- Voltar atrás

Digite um número:
6
#Inputs = 1
#Outputs = 4
```

9.2 *Bad Smells* e *Refactoring*

9.2.1 Exemplo C–

```
1  void main() {
2      int a;
3      int b;
4      int res;
5      int c;
6      a = input(int);
7      b = input(int);
8      res = max(a,b,c);
9      print(';');
10     print(res);
11 }
12
13 int max(int a, int b, int c){
14     int res;
15     if (a > b) {
16         res = a;
17     }
18     else {
19         res = b;
20     }
21     return res;
22 }
```