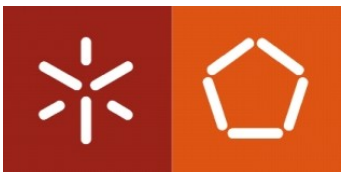

Universidade do Minho

Mestrado Integrado em Engenharia Informática

Métodos Formais em Engenharia de Software

Análise e Teste de Software

Métricas e Smells para C– e MSP

Grupo 11
Carlos Sá - A59905
Filipe Oliveira - A57816
Sérgio Caldas - A57779

Resumo

O presente documento constitui o relatório da primeira fase do projeto desenvolvido no âmbito da UCE de Análise e Teste de Software.

Este projeto consiste no desenvolvimento de um catálogo de métricas para a linguagem C– e MSP. Assim, pretende-se definir um catálogo de métricas que permita avaliar código fonte escrito na linguagem **C–** e MSP. Tal será feito com o recurso ao **Tom** - uma linguagem para manipulação de àrvores e **Gom** que permite a representação da árvore numa especificação algébrica. Através destas métricas poderemos extrair indicadores que permitam localizar pontos de *correção*.

Entre as várias métricas temos, por exemplo, as que vão contabilizar o número de linhas de código, o número de funções, ciclos, chamadas a funções recursivas entre outras.

Com o presente catálogo pretende-se localizar *bad smells* em C– em função de valores dessas métricas. Baseando-nos num conjunto de bons programas em C– definem-se o valor de referência para algumas dessas métricas. Caso seja analisado um programa que não siga esse valor de referência então um "bad smell"é localizado (por exemplo, uma função com muitas linhas). Ao longo deste documento explicaremos como foram criadas essas métricas e como foram utilizadas para um conjunto de programas. O projeto pode ser acedido através do seu respectivo repositório **GitHub** aqui.

1 Catálogo de Métricas de Software

A primeira fase de estudo para este trabalho, foi procurar um conjunto de métricas de software para detecção de *bad smells*.

Após o estudo feito sobre as diferentes métricas de software existentes, o grupo fez uma seleção das métricas de software mais adequadas aos programas na linguagem que C–. Foi então construído um catálogo de métricas de software a desenvolver em *TOM* pelo grupo.

Abaixo estão listadas o catálogo de métricas que o grupo conseguiu construír:

- Cálculo do número de funções;
- Contagem do número de argumentos por função;
- **Complexidade Ciclomática** (**CYCLO**);
- Número complexidade operacional intrinsica (**CYCLO/LOC**)
- Linhas de código por operação (**LOC/NOM**);
- Cálculo do número de **operadores por função**;
- Número total de **operadores únicos por função**;
- Número total de **operandos por função**;
- Número total de **operandos únicos por função**;
- Cálculo da **métrica de Halstead** (Volume, Dificuldade, Esforço, e Bugs);
- **Número de comentários** por função;
- Número total de linhas de código (**LOC**);
- Cálculo do *Path length* de instruções (**IPL**);
- Contagem do Número **Total de Operações de Comparação**;
- Cálculo do número **total de operações de incremento/decremento**;
- Número total de **operações de atribuíção**;
- Número de **argumentos não utilizados** por função;
- Total de declarações não usadas por função;
- Número de packages (**NOP**). (1 ficheiro código C– corresponde a 1 package);
- Número total de Classes (**NOC**). No nosso caso 1;
- Número total de classes por package (**NOC/Package**). No nosso caso 1;
- Número total de operações (**NOM**);

- Número total de operações por classe (**NOM/Class**);
- Número de chamadas (**CALLS**) a operações;
- **Total de classes** invocadas;
- **Coupling intensity**: número de chamadas por operação (CALLS/Operações);
- **Coupling dispersion**: FANOUT/Operation_Call (número de classes envolvidas por operação;
- Número médio de classes derivadas (**ANDC**): Número de subclasses directas de uma classe. No nosso caso, 0;
- **AHH** Altura média da hierarquia (tamanho médio do path length desde a raíz até à subclasse mais profunda). No nosso caso é assumido por omissão 1 subclasse e o respectivo path length máximo da função em estudo.
- Indice Maintainability
- Número de operações por classe (NOM/NOC);

2 Programas C– de exemplo utilizados

Para avaliar a correção dos valores obtidos pelas métricas, foram utilizados um conjunto de pequenos programas em C– que passamos a listar abaixo:

- max (Cálculo do valor máximo entre dois números);
- max3numbers (Cálculo do máximo de três números);
- factorialI (Cálculo do factorial *Imperativo*);
- factorialR (Cálculo do factorial *Recursivo*);
- descN (que calcula a sequência de descendentes inteiros de N);
- oddOrEven (que verifica se um valor é par ou impar);
- swap_ab (troca o valor de duas variáveis);
- mult2num (multiplicação de dois valores inteiros);
- printNnaturals (imprime os naturais de 1 até N);
- armstrongnumber (que calcula valores de Armstrong

O código fonte de cada um destes programas em C– criados pelo grupo pode ser encontrado na pasta **exemplos** no código anexo a este relatório.

3 Estratégias de Refactoring

Após criarmos o conjunto de métricas já referidos na secção 1 partimos para o segundo objectivo do trabalho. Um segundo objectivo para o trabalho foi criar um conjunto de estratégias em *TOM* e funções que permitam realizar *refactoring* de pedaços de código fonte. Estas refabricações trazem a potencialidade de, dado um código fonte em C–, poder realizar um conjunto de transformações que permitam tornar o código mais legível, e retirar partes do código fonte que não sejam necessárias. Estas transformações devem ser realizadas de forma a que o código seja alterado ao nível da sua estrutura interna sem que a funcionalidade do programa seja comprometida e a correcção do mesmo seja garantida.

Todas as transformações realizadas, alteram o programa a nível estrutural não acrescentando nem removendo funcionalidade aos programas nem quaisquer tipo de optimizações.

Depois de várias tentativas de definição de diferentes *refactorings* o grupo chegou a um conjunto de 4 refactorings:

- Refactoring da negação das condições dos if's;

```
%strategy RefactorNegIf() extends Identity() {
    visit Instrucao {
        If(c1,c2,c3,Nao(condicao),c4,c5,i1,i2) ->
            { return `If(c1,c2,c3,condicao,c4,c5,i2,i1); }
    }
}
```

- Dupla negação de condições;

```
%strategy RefactorNegNeg() extends Identity() {
    visit Expressao {
        Nao(Nao(c1)) -> { return `c1; }
    }
}
```

- Remoção de argumentos não utilizados por uma função;

```
public static Argumentos removeArgumentosNaoUtilizados(Argumentos args, ↩
    TreeSet<String> idsUtilizados) {
    %match(args) {
        ListaArgumentos(arg1,tailArg*) -> {
            %match(arg1) {
                a@Argumento(_,_,_,idArg,_) -> {
                    if (idsUtilizados.contains(`idArg))
                        return `ListaArgumentos(a,↩
                            removeArgumentosNaoUtilizados(tailArg*,↩
                            idsUtilizados));
                    else return removeArgumentosNaoUtilizados(`tailArg*,↩
                        idsUtilizados);
                }
            }
        }
    }
    return args;
}
```

- Remoção de declarações não utilizadas;

```
public static Instrucao removeDeclaracoesNaoUtilizadas ( Instrucao inst,↩
     TreeSet<String> idsNaoUtilizados ) {
    %match(inst) {
        Declaracao(_,_,_,declaracoes,_,_) -> {
            %match(declaracoes) {
                ListaDecl( dec1,taliDec*) -> {
                    %match(dec1) {
                        Decl(id,_,_,_,_) -> {
                            if ( idsNaoUtilizados.contains(`id))
                            return `Exp(Empty());
                        }
                    }
                }
            }
        }
    }
    return inst;
}
```

4 Identificação de Bad Smells e Refactoring Automático

Para que fosse possível identificar Bad Smells no código dos programas C– e transformá-lo num código mais legivel e estruturalmente melhorado. Para tal é preciso utilizar as métricas criadas em 1 e comparar os valores obtidos por cada uma das métricas com os seus valores de referência. Cada métrica, possui os seus próprios valores de referência que permitem classificar se o código está estruturalmente bem construído ou não de acordo com o valor obtido para a métrica em causa. Assim, foi necessário estudar para cada métrica, quais os intervalos de valores optimos.

De acordo com a obra [3] chegamos aos seguintes intervalos de valores para as métricas:

	Low	**Average**	**High**	**Fonte**
Cyclo/Lines of Code	0.20	0.25	0.30	OOMP[3]
LOC/Operation	5	10	16	OOMP [3]
NOM/Class	4	9	15	OOMP [3]
NOC/Package	3	19	35	OOMP [3]
CALLS/Operation	1.17	1.58	2	OOMP [3]
Fanout/CALL	0.20	0.34	0.48	OOMP [3]
ANDC	0.19	0.28	0.37	OOMP [3]
AHH	0.05	0.13	0.21	OOMP [3]
Indice Maintainability	0.05	0.10	0.95	MICROSOFT [1]
LOC	9	19	39	NDEPEND CODE ANALYSER [2]
Number of Arguments	3	5	8	NDEPEND CODE ANALYSER [2]
Unused Arguments	0	0	1	NDEPEND CODE ANALYSER [2]

Tabela 1: *Valores de referência para as métricas*

Os intervalos de valores de referência para algumas das métricas mostradas em 1 foram obtidas a partir de [1] e [2] de acordo com a coluna fonte da tabela acima.

Dado um código em C–, se um dado valor obtido para um métrica através do nosso programa está acima do valor tabelado *High* então um bad smell é detetado. A partir do momento em que um *bad smell* é detetado pelo nosso programa é necessário classificar negativamente o valor da métrica obtida através de um sistema de classificação.

De igual modo, é necessário dar uma classificação positiva se o valor da métrica estiver dentro do intervalo de valores estabelecido. Mais informação sobre a classificação das métricas é apresentada na secção 4.1. No que toca à parte de *refactoring*, sempre que é identificado um *bad smell*, é aplicada uma das transformações de código já apresentadas em 3.

Como não existem refabricações para todo o tipo de smells identificados, são aplicadas transformações de código apenas nos casos previstos nessa mesma secção. A transformação do código fonte é feita de forma automática pelo programa e será analisada com mais profundidade na secção 4.7.

4.1 Sistema de classificação do Programa

Com base nos valores de referência obtidos para as métricas referidas no ponto 4, temos então 12 operandos distintos para poder atribuir uma classificação ao código em análise.

Assim, e visto que nessa mesma tabela, os valores de referência apontam para 3 situações distintas por métrica, achamos por bem criar uma sistema de classificação global de 3 estrelas.

Considere que a cada métrica com valor de referência estão associados 3 possíveis pontos. Temos então 36 pontos possíveis para as 12 métricas. Para obter os 3 pontos uma métrica tem que registar um valor abaixo do valor tido como limite de LOW. Valores da métrica que estejam entre os valores de referência LOW e AVERAGE dão 1 ponto à pontuação global. Valores acima da referência HIGH dão à métrica -7 pontos.

A pontuação global é então calculada e a atribuídas estrelas tendo em conta esse mesmo valor:

- 1 estrela – valores entre 0 e 12 pontos;
- 2 estrelas – valores entre 13 e 24 pontos;
- 3 estrelas – valores entre 25 e 36 pontos;

4.2 Overview Pyramid

Tendo em conta que as primeiras 8 métricas representam a chamada "Overview Pyramid"apresentada no livro **Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems [3]**, podemos calcular essa mesma pirâmide para o código em análise pela nossa aplicação, e organizar-las por 3 super grupos iguais aos apresentados no livro:

- Métricas que relacionam Herança;
- Métricas que relaccionam Tamanho e Complexidade;
- Métricas que relaccionam Coupling (relação entre classes);

Atente nos seguintes exemplos do livro:

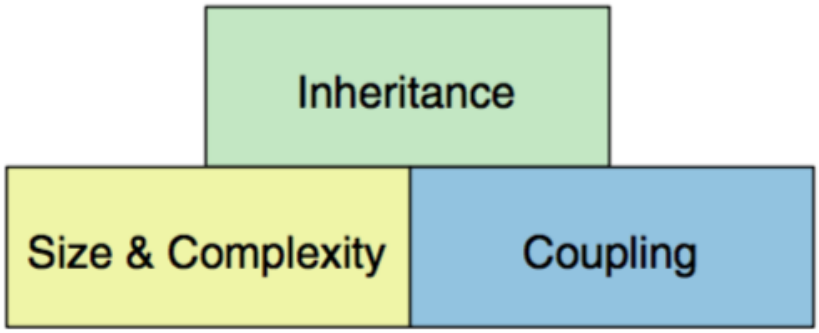

Figura 1: *Overview dos 3 super-grupos da pirâmide:* ***Herança, Complexidade e Tamanho, e Coupling***

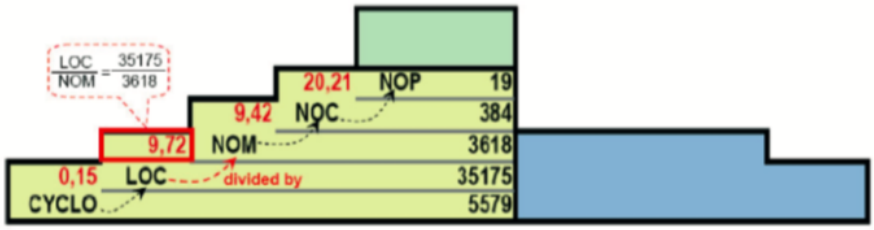

Figura 2: *Exemplo de cálculo de uma métrica pertencente ao super-grupo* ***Complexidade e Tamanho***

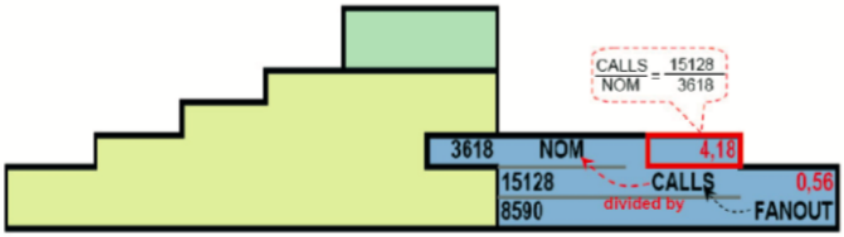

Figura 3: *Exemplo de cálculo de uma métrica pertencente ao super-grupo* ***Coupling***

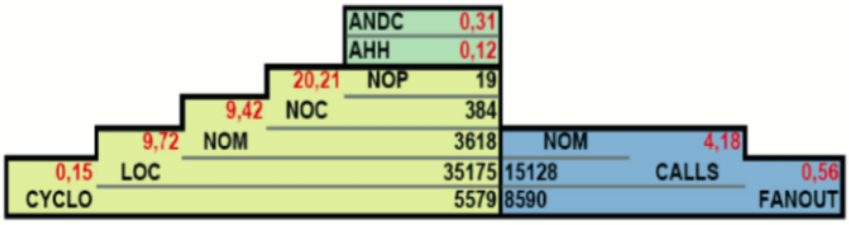

Figura 4: *Overview completa da Pirâmide*

4.3 Overview Gráfica das Métricas

Dado o número elevado de métricas calculadas, assim como a forma pensada para atribuir classificações aos códigos em análise, julgamos essencial conseguir visualizar e consequentemente interpretar essas mesmas métricas de uma forma simples.

Assim sendo, decidimos criar uma interface gráfica para a aplicação, que se assemelhe e englobe a noções da Overview Pyramid, assim como as métricas por nós adicionadas que não estão presentes na mesma.

Denote ainda, que em caso de necessidade de *refactoring* a aplicação gera uma novo código máquina e permite a comparação das métricas aplicadas ao código já sem os*Smells* possíveis de serem automaticamente removidos.

Temos então uma forma gráfica de visualizar os resultados de cada métrica, assim como a influência da refabricação do código em termos da sua classificação final.

Atente nos seguintes mockups da aplicação contendo exemplos por nós considerados pertinentes:

4.4 Exemplo de classificação de um código do repositório com 3 Estrelas

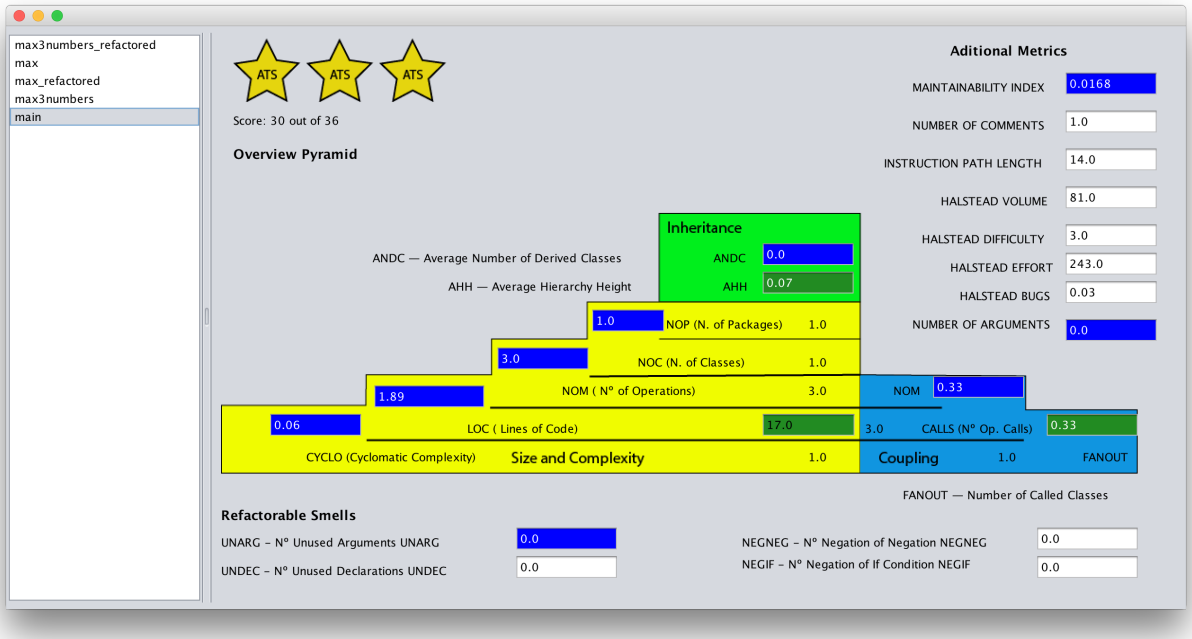

Figura 5: *Código classificado com 3 Estrelas*

4.5 Exemplo de classificação de um código do repositório com 2 Estrelas

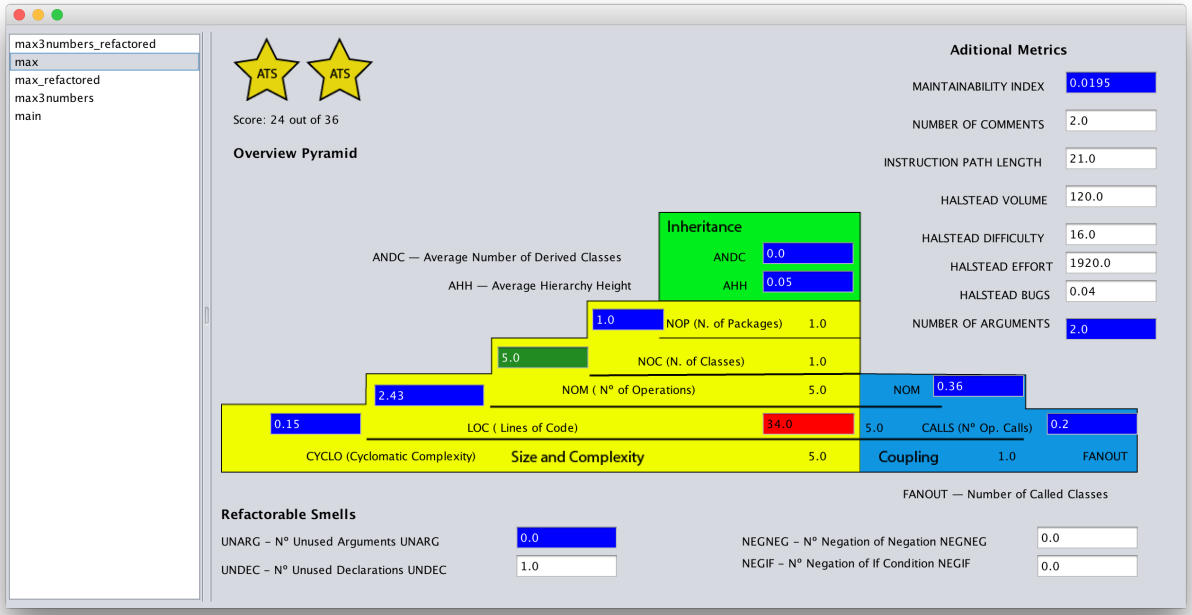

Figura 6: *Código classificado com 2 Estrelas*

4.6 Exemplo de classificação de um código do repositório com 1 Estrela

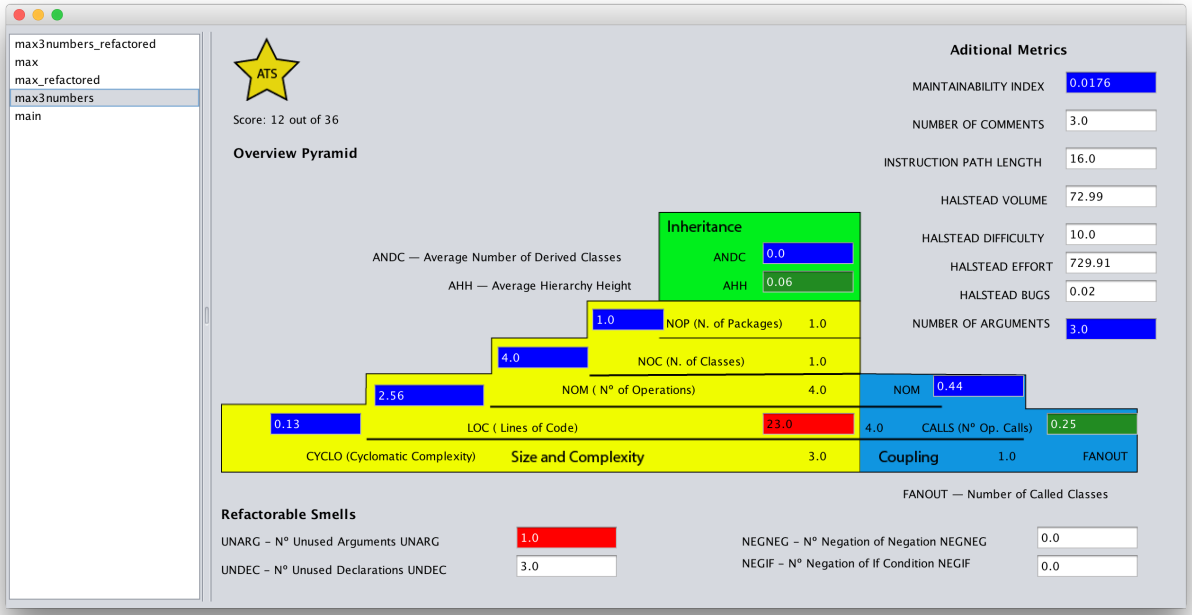

Figura 7: *Código classificado com 1 Estrela*

4.7 Exemplo de Refactoring Automático de código

4.7.1 Antes de Refactoring (1 estrela)

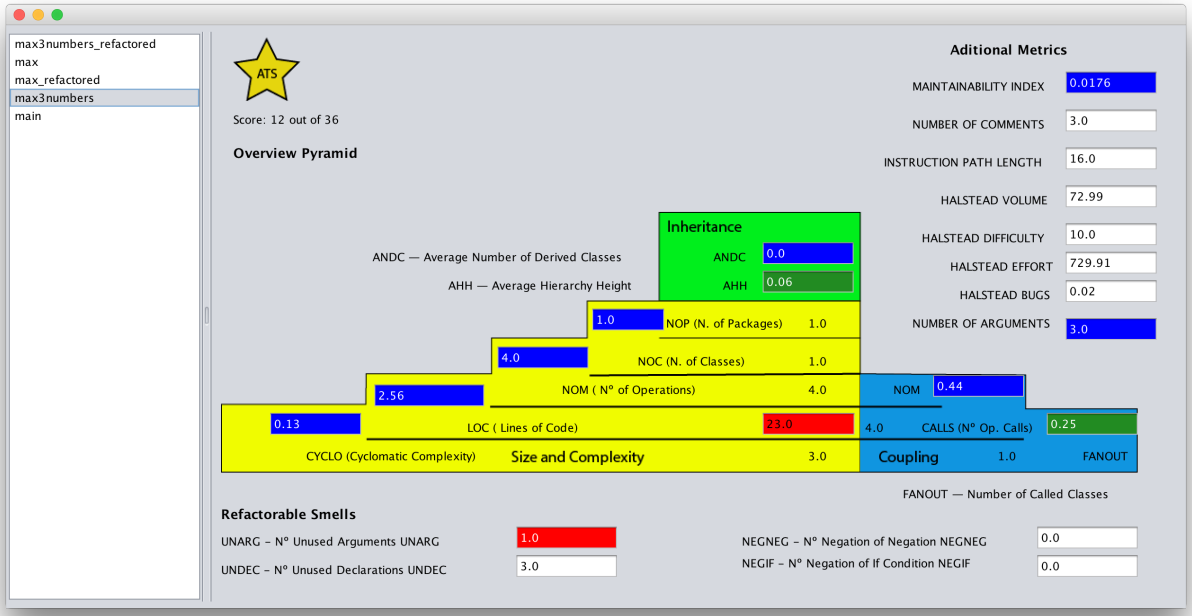

Figura 8: *Código classificado com 1 estrela antes de Refactoring*

4.7.2 Após Refactoring (2 estrelas)

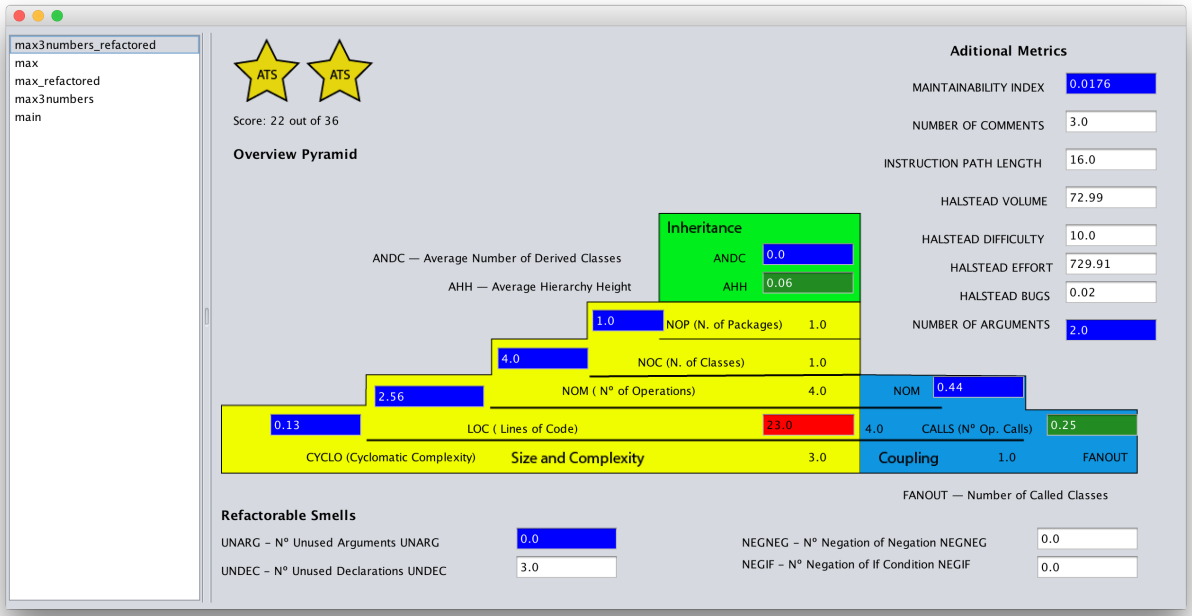

Figura 9: *Código classificado com 2 estrelas após Refactoring*

5 Dificuldades e limitações

Ao longo de todo o projecto o grupo foi confrontado com algumas dificuldades que foram limitando a progressão do trabalho. Essas dificuldades estão relacionados, na sua maioria, com perceber como funciona o *TOM*. Foi também necessário muito tempo para perceber como é que acedíamos e manipulávamos a informação através da árvore gerada e poder definir as métricas e os refactorings.

Foi também difícil de encontrar alguns valores "Standard"credíveis para definir os valores de referência para certas métricas, sendo que para outras métricas optamos por não definir valores standard mas apresentar apenas o valor das mesmas ao utilizador.

6 Trabalho Futuro

Sendo que o grupo enfrentou algumas dificuldades como as que foram descritas na secção 5 existe algum trabalho futuro que poderia ser realizado com vista a enriquecer este projeto.

A primeira tarefa está relacionada com a criação de um maior número de métricas. Este trabalho reúne um conjunto de várias métricas para situações de deteção de smells distintas, mas pelo que estudamos, existem muitas outras métricas que poderiam ser incluídas.

Outra funcionalidade futura sugerida é a definição de um maior conjunto de técnicas de *refactoring* para realizar transformações que permitam obter código mais limpo e legível.

7 Conclusão

Globalmente o grupo está satisfeito com o resultado final do trabalho.

Abordamos o trabalho por forma a cumprir todos os objetivos do enunciado e acreditamos que esses objetivos foram cumpridos. Graças a este trabalho aprendemos mais sobre como deve ser feita a análise de qualidade de software com recurso a métricas, e técnicas de *refactoring* de código por forma a melhorar a qualidade dos programas de entrada.

Referências

[1] Microsoft code metrics values: `https://msdn.microsoft.com/en-us/library/bb385914.aspx`.

[2] Ndepend code analyzer `http://www.ndepend.com/default-rules/webframe.html`.

[3] Michele Lanza, Radu Marinescu, and Stéphane Ducasse. *Object-Oriented Metrics in Practice*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.