UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

FERRAMENTA DE APOIO À COLETA DE MÉTRICAS EM SOFTWARE ORIENTADO A OBJETOS

BRUNO GÓIS BORGES

BLUMENAU
2006

2006/2-03

BRUNO GÓIS BORGES

FERRAMENTA DE APOIO À COLETA DE MÉTRICAS EM SOFTWARE ORIENTADO A OBJETOS

Proposta de Trabalho de Conclusão de Curso submetida à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso I do curso de Ciências da Computação — Bacharelado.

Prof. Everaldo Artur Grahl - Orientador

BLUMENAU
2006

2006/2-03

FERRAMENTA DE APOIO À COLETA DE MÉTRICAS EM SOFTWARE ORIENTADO A OBJETOS

Por

BRUNO GÓIS BORGES

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Prof. Everaldo Artur Grahl, Mestre
Membro:	Prof. Alexander Roberto Valdameri, Mestre
Membro:	Prof. Paulo César Rodacki Gomes, Doutor

Blumenau, dia de mês de ano [data da apresentação]

Dedico este trabalho a todos que direta ou indiretamente contribuirão para o êxito do mesmo.

AGRADECIMENTOS

Agradeço primeiramente à Deus, por ter concebido serenidade e conhecimento para o desenvolvimento do trabalho.

Aos meus familiares, em especial meus pais Marinete Góis Borges e Roberto Sales Borges, que mesmo longe, sempre incentivaram e acreditaram nos meus estudos em busca de conhecimento.

Aos meus amigos, por suportar as horas difíceis, sempre dando apoio nas minhas decisões.

Ao meu orientador, Everaldo Artur Grahl, por ter acreditado na conclusão deste trabalho.

Há três espécies de cérebros: uns entendem por si próprios; os outros discernem o que os primeiros entendem; e os terceiros não entendem nem por si próprios nem pelos outros; os primeiros são excelentíssimos; os segundos excelentes; e os terceiros totalmente inúteis.

Maquiavel

RESUMO

Este trabalho descreve a criação de uma ferramenta para cálculo de métricas em programas orientados a objetos (OO) codificados na linguagem C# e Java. Entre as métricas utilizadas incluem-se, por exemplo, métodos ponderados por classe, profundidade da árvore de herança e acoplamento entre objetos. O resultado do processo de cálculo é exibido em forma de grade, o qual pode ser armazenado em *Extensible Markup Language* (XML). Além disso, é apresentado gráfico para o entendimento da complexidade das classes.

Palavras Chaves: Orientação a objetos. Métricas de software.

ABSTRACT

This work describes the creation of a tool for calculation of metric in OO programs codified in the language C# and Java. Between the metrics used they are included, for example, weigthed methods per class, depth of inheritance tree and coupling between object classes. The result of the calculation process is shown in a grid, which can be stored in Extensible Markup Language (XML). Moreover, it is presented graphical for the agreement of the complexity of the classes.

Words Keys: Object oriented. Software Metrics.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

CBO – *Coupling between object* classes – Acoplamento entre classes de objetos

CK - Chidamber e Kemerer

COCOMO - *Constructive Const Model*

CS – *Class size* – Tamanho da classe

DIT – *Depth of the inheritance* – Profundidade da árvore de herança

EA – *Enterprise Architect*

GDI+ - *Graphics Device Interface*

HTML – *HyperText Markup Language*

LCOM – *Lack of cohesionin methods* – Falta de coesão em métodos

LOC – *Lines of Code*

LK – Lorenz e Kidd

NOA – Number of operation added by subclass – Número de operações adicionadas por subclasses

NOC – *Number of children* – Número de filhos

NOO – *Number of operations overriden by a subclass* – Número de operações redefinidas por uma subclasse

RFC – *Response for a class* – Respostas de uma classe

SI – *Specialization index* – Índice de especialização

WMC – *Weighted methods per class* – Métodos ponderados por classe

SUMÁRIO

1 INTRODUÇÃO

Segundo Arthur (1994, p. 25), para gerenciar produtividade e qualidade é necessário saber se ambas estão melhorando ou piorando. Isto implica a necessidade de métricas que indiquem as inclinações do desenvolvimento de sistema. De acordo com Amber (1998, p. 72), a gerência de um produto de software atinge um determinado estado de qualidade e precisão se existirem medidas que tornem possível a administração através dos aspectos do sistema. A métrica de software é uma medida de propriedades do sistema, que podem ser definidas como caminhos para determinar quantitativamente a dimensão em que o produto, a seqüência e o projeto de software têm certas características (CÔRTES; CHIOSSI, 2001, p. 30).

De acordo com DeMarco (1989, p. 21), o processo de software estará sob controle se for adotada uma política de coleta de dados e documentação durante o desenvolvimento do projeto. O objetivo da mensuração é abastecer engenheiros e gerentes de produtos com um grupo de informações palpáveis para se projetar, gerenciar, controlar, estimar e melhorar os projetos com maior eficácia (TONINI, 2004). Segundo Côrtes e Chiossi (2001, p. 28), quando são calculadas métricas, pretende-se obter dados que irão proporcionar opções para uma melhoria. Este é o objetivo da métrica de software, o estudo dos fatores que influenciam o rendimento através da qualificação dos projetos de desenvolvimento de software.

Entre as principais inquietações nas fábricas de software encontra-se a possibilidade de se criar um sistema de uma maneira mais rápida e a um custo mais baixo. As práticas baseadas em objetos simplificam o projeto de softwares complexos (PRESSMAN, 1995, p. 42). Para Amber (1998, p. 31), as organizações escolhem a orientação a objetos (OO) porque querem dar às suas aplicações mais qualidade, as quais querem implementar sistemas seguros, com um menor custo e menor tempo.

Este trabalho pretende contribuir para a análise da qualidade e complexidade do código OO, assim como auxiliar no entendimento dos benefícios das métricas.

1.1 OBJETIVO

O objetivo deste trabalho é desenvolver uma ferramenta capaz de efetuar a coleta de métricas de software OO a partir da análise de códigos fontes escrita em linguagem C# e Java.

1.2 ESTRUTURA DO TRABALHO

O trabalho está dividido em cinco capítulos.

O primeiro capítulo apresenta a introdução e os objetivos pretendidos com a elaboração do trabalho.

O segundo capítulo descreve o que são métricas e apresenta a origem dos sistemas métricos, importância e objetivos. O capítulo reúne os conceitos de algumas métricas tradicionais.

O terceiro capítulo apresenta as métricas para orientação a objetos, suas características e diferenças em relação às métricas tradicionais. Em seguida são apresentadas algumas métricas para orientação a objetos divididas em três categorias: análise, projetos e construção.

O quarto capítulo apresenta o desenvolvimento do trabalho, incluindo a descrição da especificação e da implementação do protótipo. Também no quarto capítulo são apresentadas as métricas selecionadas para implementação.

O quinto capítulo finaliza o trabalho, com as conclusões, limitações e sugestões para novos trabalhos nesta área.

2 ORIGEM DO SISTEMA MÉTRICO

As métricas originaram-se da execução prática de avaliação para quantificar indicadores sobre o processo de desenvolvimento de um sistema, sendo adotados a partir de 1970. Existem quatro tendências desta área (MOLLER; PAULISH, 1993, p. 38) que são:

a) medida da complexidade do código: criada em meados de 1970, os conjuntos métricos foram fáceis de se atingir desde que fossem calculados pelo próprio código automatizado;
b) estimativa do custo de um projeto de software: esta técnica foi desenvolvida em meados de 1970, estimando o trabalho e o tempo gasto para se desenvolver um software, baseando-se além de outros fatores, no número de linhas de código utilizados na implementação do sistema;
c) garantia da qualidade do software: a melhoria destas técnicas tiveram maior repercussão entre os anos de 1970 e 1980, dando-se destaque à identificação de informações faltantes, durante as etapas do ciclo de vida do software;
d) processo de desenvolvimento do software: o projeto de software ganhou importância e complexidade, sendo que a necessidade de se administrar este processo foi emergencial. O processo incluiu a definição do ciclo de vida do software pela seqüência das fases e mais destaque no gerenciamento e controle de recursos deste projeto.

A partir do aparecimento destas tendências, os desenvolvedores de sistema começaram a usar métricas no propósito de adequar o processo de desenvolvimento de software.

2.1 IMPORTÂNCIA DAS MEDIÇÕES

Conforme Tonini (2004), se não é conhecida a complexidade de um software não se pode saber o caminho a seguir e nem mesmo o que fazer para solucionar um problema. Uma das maneiras de se controlar o desenvolvimento de um sistema é a utilização da medição de software. As métricas podem medir cada estágio do desenvolvimento e diversos aspectos do produto. Métricas ajudam a compreender o processo utilizado para a implementação de um sistema (JACOBSON et al., 1992, p. 94). De acordo com Pressman (1995, p. 82), o processo

é medido com o propósito de melhorá-lo e o produto é mensurado com o intuito de ampliar sua qualidade.

Medidas são necessárias para examinar a qualidade e o rendimento do processo de desenvolvimento e manutenção do produto de software implementado (CORDEIRO, 2000, p. 37). Para Côrtes e Chiossi (2001, p. 25), “as empresas devem estabelecer métricas apropriadas e manter procedimentos para monitorar e medir as características de suas operações que possam causar impacto significativo na qualidade de seus produtos”.

2.2 OBJETIVOS DA UTILIZAÇÃO DE MÉTRICAS

Segundo Funck (1995), a utilidade das métricas deve ser traçada desde o início da implantação de métricas para avaliação de software. Há várias características importantes associadas com o emprego das métricas de software. Sua escolha requer alguns pré-requisitos (FERNANDES, 1995):

a) os objetivos que se pretende atingir com a utilização das métricas;
b) as métricas devem ser simples de atender e de serem utilizadas para verificar atendimentos de objetivos e para subsidiar processos de tomadas de decisão;
c) as métricas devem ser objetivas, visando reduzir ou minimizar a influência do julgamento pessoal na coleta, cálculo e análise dos resultados.

Para Amber (1998), as métricas podem ser utilizadas para:

a) estimar projetos: baseado em experiências anteriores pode-se utilizar métricas para estimar o tempo, o esforço e o custo de um projeto;
b) selecionar as ferramentas;
c) melhorar a abordagem de desenvolvimento.

De acordo com Fernandes (1995), em uma organização que se dedica ao desenvolvimento de software, seja como atividade-fim seja como de suporte para uma empresa, há vários objetivos que se busca atingir, dependendo do estágio de maturidade em que se encontram essas atividades. Alguns dos objetivos perseguidos geralmente se enquadram na seguinte relação:

a) melhorar a qualidade do planejamento do projeto;

b) melhorar a qualidade do processo de desenvolvimento;
c) melhorar a qualidade do produto resultante do processo;
d) aumentar a satisfação do usuários e clientes do software;
e) reduzir os custos de retrabalho no processo;
f) reduzir os custos de falha externas;
g) aumentar a produtividade do desenvolvimento;
h) aperfeiçoar continuamente os métodos de gestão do projeto;
i) aperfeiçoar continuamente o processo e o produto;
j) avaliar o impacto de atributos no processo de desenvolvimento, tais como novas ferramentas;
k) determinar tendências relativas a certos atributos do processo.

Segundo Fernandes (1995), um dos aspectos que deve ser observado quando da implementação de iniciativas de utilização de métricas é quando a sua utilidade no contexto de um projeto ou do ambiente como todo, além dos tipos e categorias de métricas, usuários das métricas, pessoas para as quais os resultados das métricas são destinados e os seus níveis de aplicação.

Para Funck (1995), o processo de medição e avaliação requer um mecanismo para determinar quais os dados que devem ser coletados e como os dados coletados devem ser interpretados. O processo requer um mecanismo organizado para a determinação do objetivo da medição. A definição de tal objetivo abre caminho para algumas perguntas que definem um conjunto específico de dados a serem coletados. Os objetivos da medição e da avaliação são conseqüências das necessidades da empresa, que podem ser a necessidade de avaliar determinada tecnologia, a necessidade de entender melhor a utilização dos recursos para melhorar a estimativa de custo, a necessidade de avaliar a qualidade do produto para poder determinar sua implementação ou a necessidade de avaliar as vantagens e desvantagens de um projeto de pesquisa.

De acordo com Fernandes (1995), o objetivo primário de se realizar medições no desenvolvimento de software é obter níveis cada vez maiores de qualidade, considerando o projeto, o processo e o produto, visando à satisfação plena dos clientes ou usuários a um custo economicamente compatível.

2.3 MÉTRICAS TRADICIONAIS

Nesta seção serão apresentados alguns exemplos de métricas tradicionais. O assunto métricas de software é muito extenso para ser abordado adequadamente em um único trabalho, para maiores informações sobre métricas tradicionais pode ser consultado (FUNCK, 1995).

2.3.1 CONSTRUCTIVE CONST MODEL

De acordo com Funck (1995), o modelo COCOMO é calculado a partir do número de linhas de código fonte entregue ao usuário. Este modelo foi desenvolvido por Barry Boehm e resulta em estimativas de esforço, prazo, custo e tamanho da equipe para um projeto de software. O COCOMO é um conjunto de submodelos hierárquicos, no qual pode ser dividido em submodelos básicos, intermediários ou detalhados.

2.3.2 LINHAS DE CÓDIGO

Para Koscianski e Soares (2006, p. 229), o modelo LOC, é a técnica de estimativa mais antiga. Ela pode ser aplicada para estimar o custo do software ou para especificar igualdade de analogia. Há muitas discussões e especulações sobre esta técnica. Primeiramente, a definição de linhas de código não é muito claro.

Um exemplo simples seria o caso de ser colocado ou não um comentário ou uma linha em branco como LOC. Alguns autores consideram estes comentários, no entanto, outros não. No caso de programas recursivos, essa técnica falha, porque a recursividade torna o programa mais curto. O sistema LOC é uma técnica genérica e superficial (KOSCIANSKI e SOARES, 2006, p. 229).

Outro problema da técnica LOC, para Pressman (1995), é que esta técnica é fortemente ligada à linguagem de programação utilizada, impossibilitando a utilização de dados históricos para projetos que não utilizam a mesma linguagem.

As vantagens do sistema LOC são (FUNCK, 1995 e POSSOMAI, 2000):

a) é fácil de ser obtido;
b) é utilizado por muitos modelos de estimativa de software como valor básico de entrada;
c) existe farta literatura e dados sobre este sistema de métrica.

As desvantagens são:

a) dependência de linguagem: não é possível comparar diretamente projetos que foram desenvolvidos em linguagens diferentes. Como exemplo, pode-se verifica a quantidade de tempo gasto para gerar uma instrução em uma linguagem de alto nível comparado com uma linguagem de baixo nível;
b) penalizam programas bem projetados: quando um programa é bem projetado o mesmo utiliza poucos comandos para execução de uma tarefa. Assim sendo, um programa que utilizem componentes está mais bem projetado, mas a medição deste tipo de programa através desta métrica é eficiente;
c) difíceis de estima no início do projeto de software: é praticamente impossível estimar o LOC necessário para um sistema saindo da fase de levantamento de requisitos ou da fase de modelagem.

Com estas colocações, nota-se que a métrica LOC não é uma métrica a ser utilizada por si só, ela deveria ser utilizada em conjunto com outras métricas, efetuando um comparativo de resultados. Deste modo uma métrica poderia completar a outra, fornecendo informações que são pertinentes às características de cada uma.

2.3.3 MEDIDA DE CIÊNCIA DO SOFTWARE

Segundo Shepperd (1993), Halstead identificou a Ciência de Software – originalmente chamada de Física do Software – como uma das primeiras manifestações sobre métrica de código baseada num modelo de complexidade do software. A idéia principal deste modelo é a compreensão de que software é um processo de manipulação mental dos símbolos de seus programas.

Estes símbolos podem ser caracterizados como operadores (em um programa executável verbos como: *IF, DIV, READ, ELSE* e os operadores propriamente ditos) ou operandos (variáveis e constantes), visto que a divisão de um programa pode ser considerada como uma seqüência de operadores associados a operandos (Shepperd, 1993).

Para Shepperd (1993), a ciência do software atraiu consideravelmente o interesse das

pessoas em meados de 1970 por ser uma novidade na metrificação do software. Além disso, as entradas básicas do software são todas facilmente extraídas. Após o entusiasmo inicial da ciência do software, foram encontrados sérios problemas. Os motivos podem ser relatados em função da dificuldade que os pesquisadores encontraram na comparação dos trabalhos e evolução da métrica. Outro motivo seria a não associação correta entre o esforço requerido para manipulação do programa e o tempo exigido para conceber o programa e também por tratar um sistema como um simples módulo.

2.3.4 MÉTRICA DA COMPLEXIDADE CICLOMÁTICA

Este método foi proposto por McCabe, que estava particularmente interessado em descobrir o número de caminhos criado pelos fluxos de controle em um módulo do software, desde que fosse relacionado à dificuldade de teste e na melhor maneira de dividir software em módulos (Shepperd, 1993).

De acordo com Jacobson (1992), a idéia é desenhar num grafo a seqüência que um programa pode tomar com todos os possíveis caminhos. A complexidade calculada fornecerá um número designando o quão complexo é um programa (ou seqüência).

Segundo Shepperd (1993), os programas são representados por grafos dirigidos representando o fluxo de controle. De um grafo G, pode ser extraída a complexidade ciclomática v(G). O número de caminhos dentro de um grafo pode ser dado como: o conjunto mínimo de caminhos os quais podem ser utilizados para a construção de outros caminhos através do grafo. A complexidade ciclomática é também equivalente ao número de decisões adicionais dentro de um programa:

v(G) = E – n + 2,
onde,
E: é o número de arestas.
N: é o número de nós.

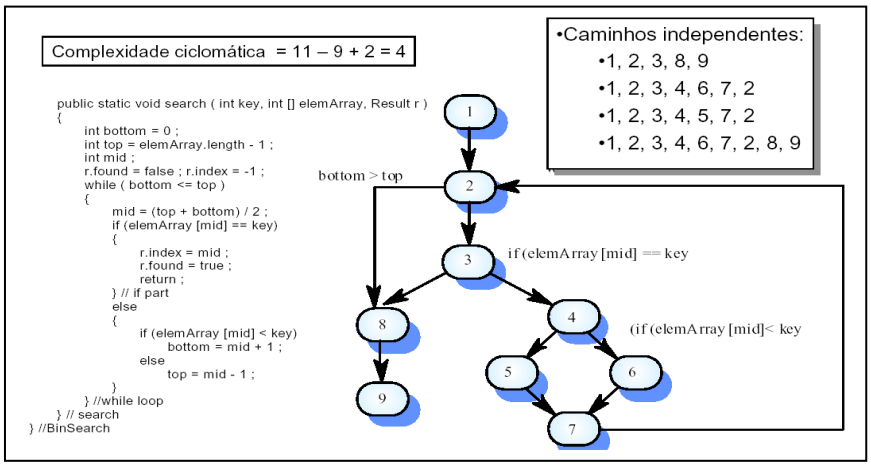
Quadro 1 – Definição das variáveis da complexidade ciclomática

A visão simplificada da métrica de McCabe pode ser questionada em vários pontos. Primeiro, ele tinha uma preocupação especial com os programas escritos em Fortran, onde o mapeamento do código-fonte, para um grafo de fluxo do programa era bem definido, sendo que isto não seria o caso de outras linguagens como Ada. A segunda oposição é que v(G) = 1,

seria verdadeiro em uma seqüência linear de código de qualquer tamanho. Conseqüentemente, a medição não é sensível à complexidade, contribuindo assim na formação de declarações de seqüência lineares.

De acordo com Shepperd (1993), a complexidade ciclomática é sensível ao número de subrotinas dentro de um programa, por este motivo, McCabe sugere que este aspecto seja tratado como componentes não relacionados dentro de um grafo de controle. Este ponto teria um resultado interessante, pois aumentaria a complexidade do programa globalmente, visto que ele é dividido em vários módulos que se imagina serem sempre simples.

A Figura 1 demonstra um exemplo de complexidade ciclomática que ilustra o fluxo do grafo gerado para os caminhos entre 1 e 9.



Fonte: Leite (2006).

Figura 1 – Exemplo de cálculo de complexidade ciclomática

3 MÉTRICAS PARA ORIENTAÇÃO A OBJETOS

Embora exista farta literatura sobre como aplicar os métodos OO, a maioria não apresenta detalhes relativos à qualidade. A razão é simples: o desenvolvimento de software utilizando esse enfoque ainda não dispõe de métricas precisas e bem entendidas que possam ser utilizadas para avaliar produtos e processos de desenvolvimento nesta abordagem. Valores para medidas atributos e formas de prevenção e correção de defeitos ainda não estabelecidos (ROCHA, 2001, p. 55).

Muitas métricas já foram desenvolvidas para gerações passadas de tecnologia e, em muitos casos, são usadas até para desenvolvimento OO, porém não são muito coerentes, pois a diferença com sistemas tradicionais é muito grande (ROCHA; MOLDONADO; WEBER, 2001, p. 56).

Para Rocha (2001, p. 56), o desenvolvimento de software utilizando o paradigma de OO surge como uma possibilidade para a melhoria da qualidade e produtividade, pois permite modelar o problema em termos de objetos capaz de diminuir a distância entre o problema do mundo real e sua abstração.

Segundo Rosenberg (1998, p. 42), A OO requer uma abordagem diferente tanto no desenvolvimento do projeto quanto na implementação do mesmo, como também nas métricas de software, visto que usa objetos e não blocos de construção fundamentais. De acordo com Rocha (2001, p. 55), dadas às diferenças entre as duas visões, é comum constatar que as métricas de software desenvolvidas para serem aplicadas aos métodos tradicionais de desenvolvimento não são facilmente mapeadas para os conceitos OO.

De acordo com Gustafson (2003, p. 112), existem várias propostas para métricas OO que levam em consideração as características básicas e interações do sistema como: número de classes; número de métodos; linhas de código por método; profundidade máxima da hierarquia de classes; a relação existente entre métodos públicos e privados; entre outros. Tais métricas baseiam-se na análise detalhada do projeto.

Segundo Jacobson (1992, p. 58), as métricas OO podem ser separadas em duas categorias: medidas relacionadas com processos e relacionadas com produtos. As métricas relacionadas com processo são utilizadas para mensurar o processo e o status do processo de desenvolvimento do sistema, consistem em medir coisas tais como: cronogramas ou números de falhas encontradas durante o processo de testes. Para Jacobson (1992, p. 58), para aprender a manipular e administrar um processo de desenvolvimento OO é importante iniciar a coleta

de dados destas medições tão metodicamente quanto possível. A seguir alguns exemplos de métricas relacionadas com processo:

a) tempo total de desenvolvimento;
b) tempo de desenvolvimento em cada processo e sub-processo;
c) tempo utilizado modificando modelos de processos anteriores;
d) tempo gasto em todos os tipos de sub-processos como: especificação dos casos de uso, desenho do bloco, teste do bloco e do caso de uso para cada objeto;
e) número de diferentes tipos de falhas encontrados durante revisões;
f) número de mudanças propostas nos modelos anteriores;
g) custo da garantia de qualidade;
h) custo para introduzir novas ferramentas e processo de desenvolvimento.

Estas medições podem formar uma base para o planejamento do desenvolvimento de projetos futuros. Por exemplo, conhecendo o tempo médio gasto para especificar todos os casos de uso. Estas medições, entretanto deveriam sempre vir acompanhadas por uma indicação de exatidão da medição (tal como desvio padrão), caso contrário, não se tem senso de exatidão da previsão. Deve-se observar também que estas medições podem variar muito entre diferentes processos, organizações, aplicação e equipe. Portanto, é perigoso tirar conclusões genéricas sobre dados existentes sem considerar as circunstâncias (Jacobson, 1992, p. 59).

As métricas relacionadas com produtos são aquelas que são utilizadas para controlar a qualidade do produto final. Elas tradicionalmente são aplicadas ao sistema ainda em construção para mensurar sua complexidade e prever propriedades do produto final.

Conforme Jacobson (1992, p. 59), medidas tradicionais de produtos podem ser utilizadas para algumas aplicações OO. Entretanto a métrica mais comum, linhas de código, é a menos interessante para sistemas OO, pois às vezes o menor código escrito é o mais reutilizado e, muitas vezes dá maior qualidade ao produto.

A seguir serão exemplificadas algumas métricas para OO. Estas métricas estão relacionadas em três categorias: métricas de análise, projeto e construção. De acordo com Ambler (1998, p. 62), estas medidas podem ser utilizadas para auxiliar a melhorar os esforços de desenvolvimento. Elas podem identificar áreas com problemas na aplicação antes que elas apareçam como um erro detectado pelo usuário. As métricas de projetos e construção além de mensurar aspectos importantes do sistema, são fáceis de automatizar, tornando-as mais fáceis de coletar (Ambler, 1998, p. 62).

A figura 2 demonstra um exemplo de diagrama de classes que ilustra algumas medidas explicadas em seguida. Este diagrama de classes define criação de pessoas: uma pessoa pode ser do tipo cliente ou funcionário, por sua vez o funcionário pode ser do tipo horista, diarista ou mensalista; um funcionário tem um cargo e deve estar em um departamento.

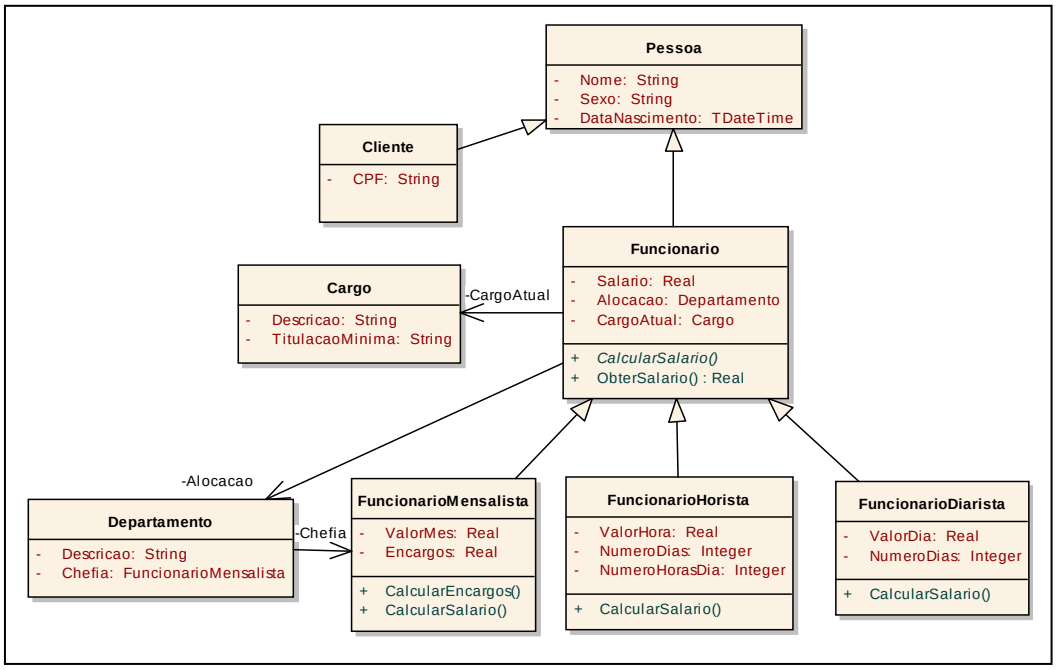


Figura 2 – Exemplo de diagrama de classes

Faz sentido adicionar um peso às métricas das classes para produzir uma medida de complexidade do sistema. A maioria das medidas examina atributos em termos dos conceitos de OO como herança, polimorfismo e encapsulamento. Para tanto, seria necessário coletar um número significativo de contagens, ou seja, seria necessário tomar valores de vários projetos e dimensioná-los selecionando as classes, os métodos e os atributos desejáveis para medir o tamanho e a complexidade de um novo software (GUSTAFSON, 2003, p. 112).

3.1 MÉTRICAS SEGUNDO CHIDAMBER E KEMERER

Chidamber e Kemerer (1994, p. 41), propuseram seis métricas para o cálculo de complexidade de sistema OO. As métricas chamadas de CK são ótimas referências para análise quantitativa, objetivando a concentração de testes em classes que possivelmente contêm maior número de defeitos. A seguir uma descrição de cada métrica:

a) WMC: cálculo do número de serviços por classe. Um alto WMC mostra que a

classe tende a se tornar específica e seus serviços possuem características que atendem a necessidades individuais, restringindo sua reutilização. O número de serviços mostra ainda qual o nível de esforço deve ser despendido para o teste da complexidade da classe;

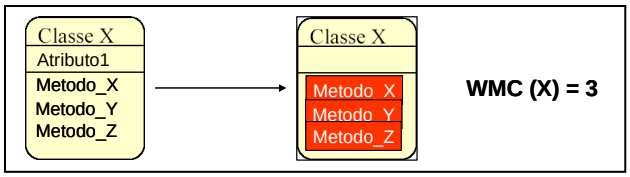


Figura 3 – Exemplo da métrica WMC

b) DIT: é o número máximo de superclasses posicionadas hierarquicamente acima da classe em questão. Um DIT elevado mostra que muitas das características da classe foram adquiridas por herança e, são comuns a outras classes. Isso mostra que as superclasses contêm um alto nível de abstração, o que significa que elas estão possivelmente preparadas para possibilitar uma boa reutilização. Em contrapartida, DIT pode indicar que a classe herda muitos serviços, aumentando a sua complexidade;

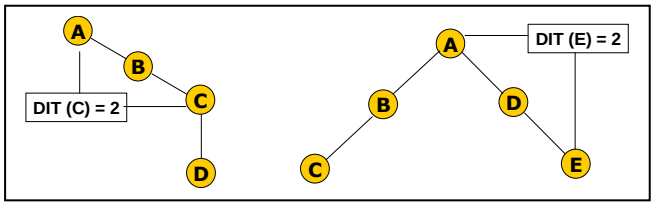


Figura 4 – Exemplo da métrica DIT

c) NOC: número de subclasses posicionadas imediatamente abaixo da superclasse em questão ou filhos diretos. Um NOC elevado indica um baixo nível de abstração, pois uma superclasse com muitos filhos, tende a conter poucas características comuns a todas as subclasses. Um alto número de filhos diretos também pode indicar problemas estruturais, uma vez que as subclasses podem não se adequar à abstração implícita da classe pai;

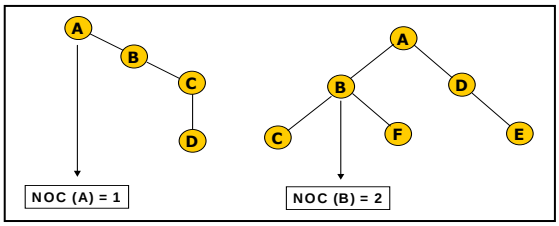


Figura 5 – Exemplo da métrica NOC

d) CBO: mostra qual é o nível de acoplamento entre as classes da aplicação. Quanto maior a ligação entre elas, menor a possibilidade de reutilização, pois a classe torna-se dependente de outras classes para cumprir suas obrigações. Portanto o CBO está diretamente legado ao nível de reaproveitamento. Um alto acoplamento indica uma baixa independência de classe, o que aumenta consideravelmente a complexidade e, em conseqüência, o esforço de teste;

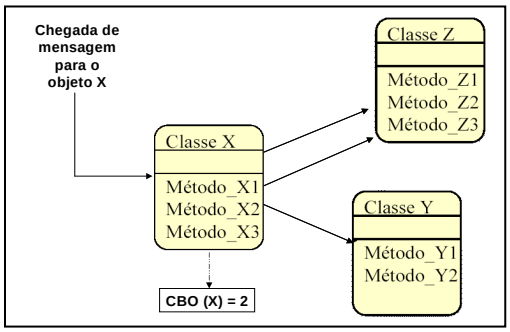


Figura 6 – Exemplo da métrica CBO

e) LCOM: número de acesso a um ou mais atributos em comum pelos serviços da própria classe. Dessa forma, os serviços tornam-se ligados pelos atributos, podendo indicar que foram mal projetados, pois apresentam baixa coesão. Uma das principais características do software OO é apresentar uma alta coesão nos métodos, o que garante que esses exerçam sua função adequadamente. Portanto é importante manter o LCOM da classe baixo;

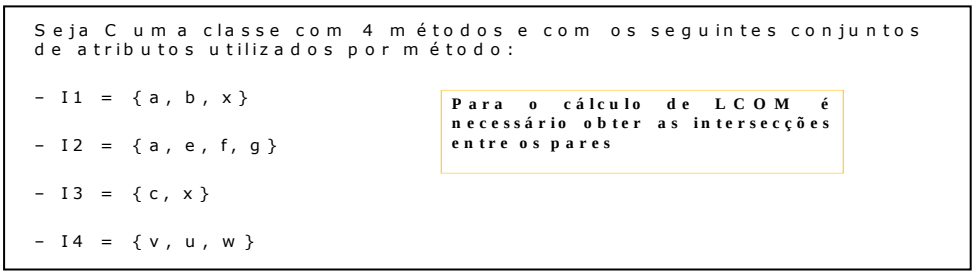


Figura 7 – Exemplo da métrica LCOM

f) RFC: indica a capacidade de resposta que a classe tem ao receber mensagens de seus objetos. Uma maior capacidade de resposta requer uma estrutura de classe projetada para atender a essa particularidade gerando uma maior complexidade, tornando necessário um maior esforço de teste.

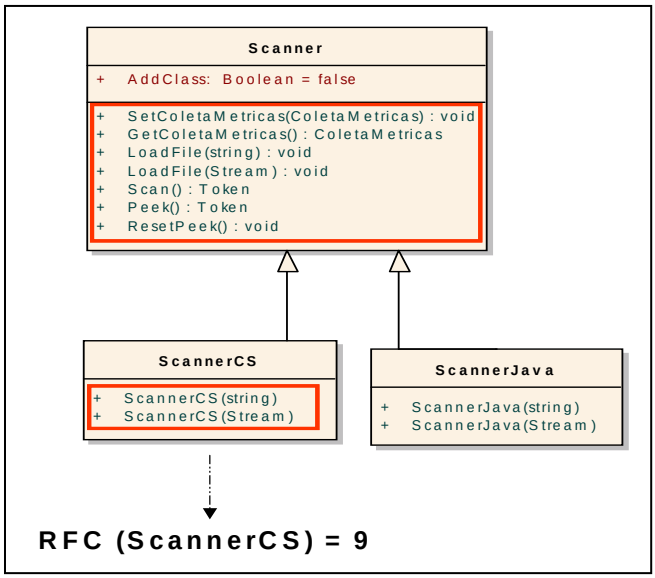


Figura 8 - Exemplo da métrica RFC

Tabela 1 – Exemplo de calculo do modelo de CK para Figura 2

Classe	WCT	DIT	NOC	CBO	RFC	LCOM
Pessoa	0	0	2	0	0	0
Cliente	0	1	0	0	0	0
Funcionário	2	1	3	2	2	0
Cargo	0	0	0	0	0	0
FuncionarioHorista	1	2	0	0	1	0
FuncionarioMensalista	2	2	0	0	2	1
FuncionarioDiarista	1	2	0	0	1	0
Departamento	0	0	0	2	0	0

3.2 MÉTRICAS SEGUNDO LORENZ E KIDD

Uma outra proposta de métricas OO foram criada por Lorenz e kidd, têm como base o cálculo quantitativo de alguns aspectos fundamentais da OO, como os atributos e serviços, herança, coesão e acoplamento. Não diferindo das métricas de CK no foco, mas sim em sua metodologia de calculo. Abaixo segue a descrição de cada métrica definida:

a) CS: número de serviços e atributos locais e herdados de superclasses. Os serviços e atributos públicos das classes localizadas hierarquicamente acima e os da própria

classe em questão compõe o CS. Um grande CS torna a classe muito específica, pois sua estrutura atende a particularidades, o que restringe a reutilização, requerendo ainda maior esforço de testes, já que a classe se torna mais complexa;

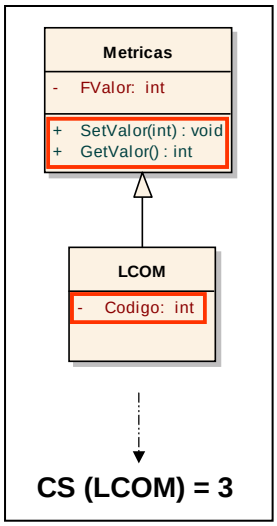


Figura 9 - Exemplo da métrica CS

b) NOO: os métodos definidos nas superclasses são herdados pelas subclasses, mas, quando esses não atendem à necessidade individual da subclasse, podem ser redefinidos, ferindo assim a abstração implícita na superclasse. Um alto índice de NOO indica um problema estrutural. Se muitas subclasses têm serviços redefinidos, as subclasses possivelmente estão hierarquicamente mal projetadas;

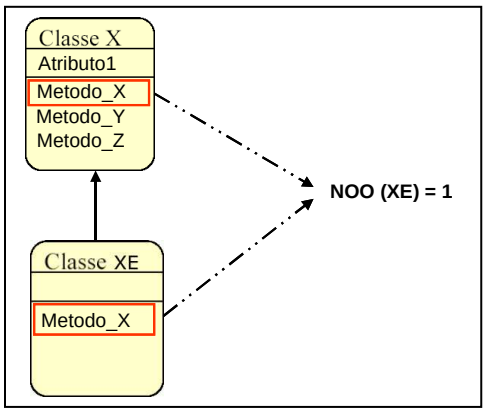


Figura 10 - Exemplo da métrica NOO

c) NOA: se a classe contém um alto número de operações e atributos privados, ela torna-se muito específica, diminuindo as possibilidades de reaproveitamento. Pode-se dizer que um alto NOA pode indicar uma falha de modelo. Muitas particularidades mostram que a classe não está bem posicionada na hierarquia, já

que suas características principais deveriam estar implícitas nos seus ancestrais;

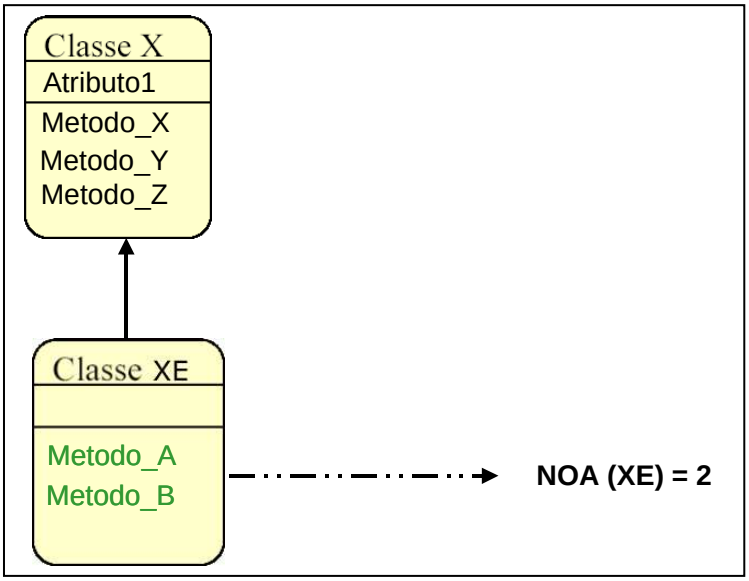


Figura 11 - Exemplo da métrica NOA

d) SI: número de serviços adicionados, eliminados ou redefinidos. Indica o nível de especialização das classes ou as alterações efetuadas para atender à necessidade individual daquela classe.

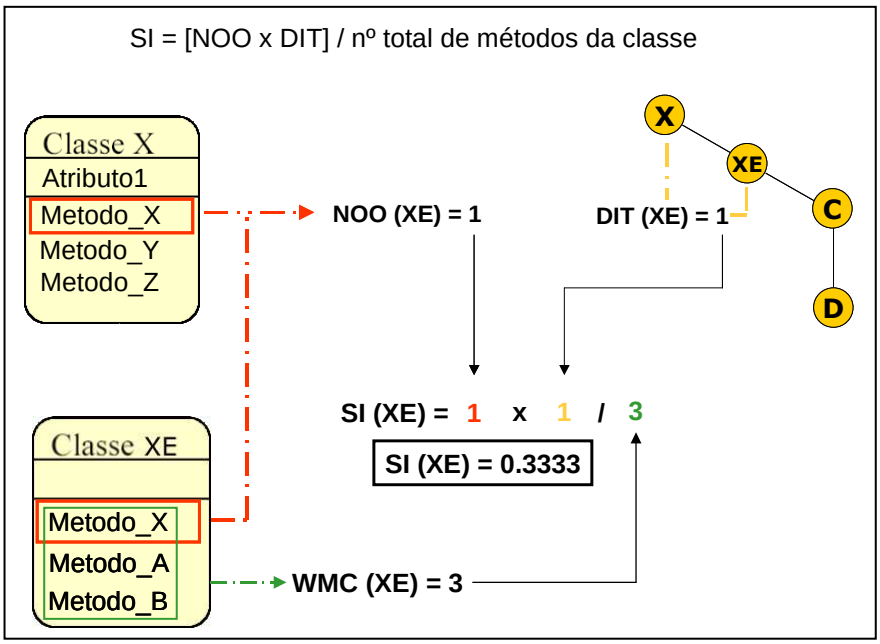


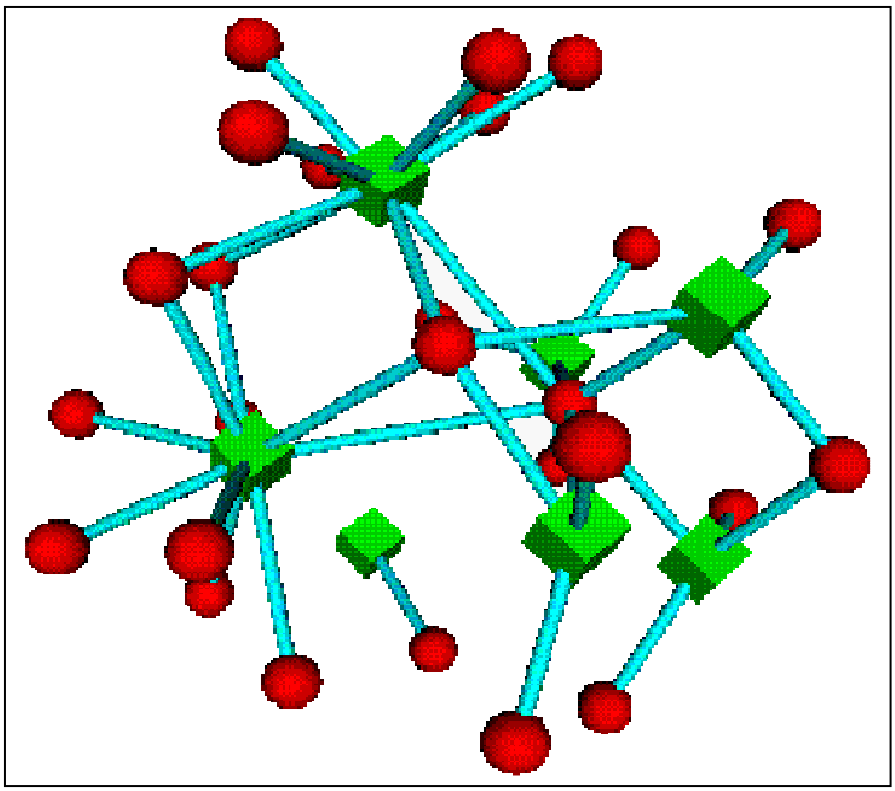
Figura 12 - Exemplo da métrica SI

Tabela 2 – Exemplo de calculo do modelo de LK para Figura 2

Classe	**CS**	**NOO**	**NOA**	**SI**
Pessoa	3	0	3	0
Cliente	1	0	1	0
Funcionário	5	0	3	0
Cargo	2	0	2	0
FuncionarioHorista	6	1	3	0
FuncionarioMensalista	6	1	2	0
FuncionarioDiarista	5	1	2	0
Departamento	2	0	2	0

De acordo com Koscianski e Soares (2006, p. 242), uma técnica muito interessante consiste em pesquisar intuitivamente o acoplamento e a coesão. Um recurso bastante utilizado é por meio de gráficos que demonstram ao analisador a complexidade estrutural do produto e facilitam a identificação de gargalos. A figura 13 é um modelo tridimensional da estrutura de uma classe. Os cubos são atributos, acessados por métodos representados por esferas. Imediatamente é percebido acoplamento ou não entre os métodos do objeto, suas dependências e oportunidades de particionamento da classe em subclasses.

Para Lanusse (2006), outro recurso em forma de gráfico muito interessante é o proposto por Kiviat, auxilia no reconhecimento de problemas de performance, é um gráfico circular em que as métricas são plotadas sobre retas radiais. Neste modelo, tem-se cada métrica representada. A linha vermelha é o limite determinado de acordo com os valores informados para cada métrica, antes da execução do cálculo. Os pontos vermelhos são as métricas que foram violadas. Os verdes significam que os valores máximos definidos na métrica estão de acordo com o que foi calculado. Se houver ponto azul, o valor mínimo estabelecido para métrica não foi atingido. Na figura 14 é apresentado um exemplo do gráfico de Kiviat.



Fonte: Koscianski e Soares (2006)

Figura 13 – Visualização da estrutura de uma classe

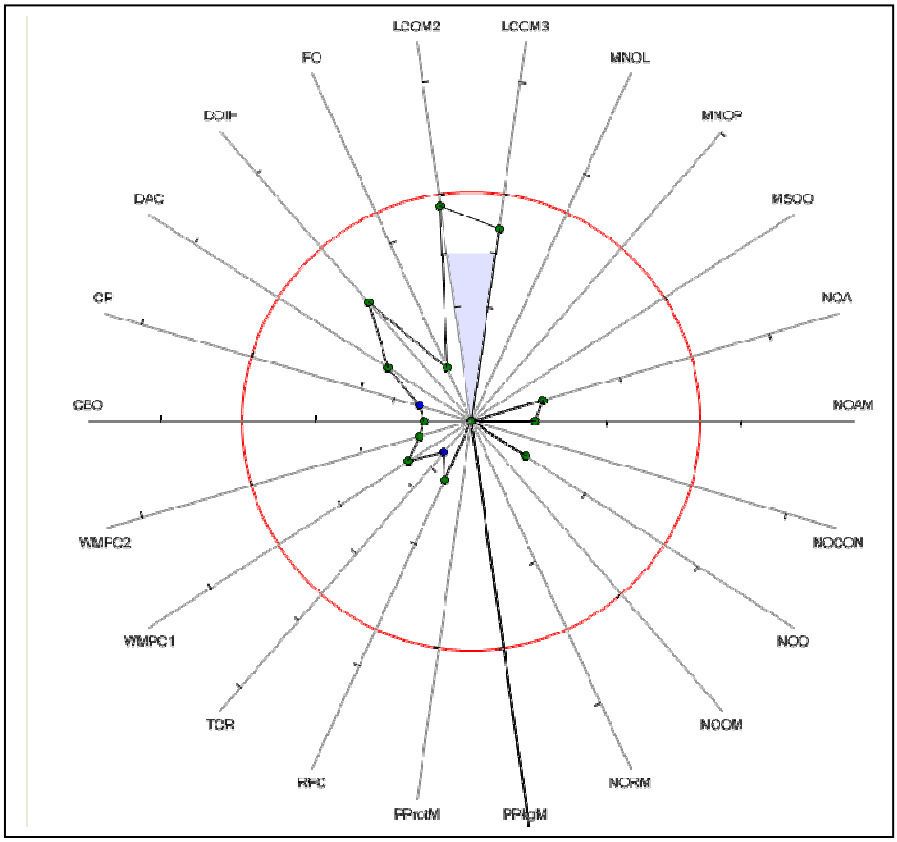


Figura 14 - Exemplo do gráfico de Kiviat

3.3 FERRAMENTAS SOBRE MÉTRICAS ORIENTADAS A OBJETOS

Nesta seção serão apresentadas três ferramentas relacionadas a métricas OO.

3.3.1 JMETRIC - JAVA METRICS ANALYSER

O JMetric propõem a coleta de métricas OO, o projeto foi iniciado em abril de 1998 como parte de uma pesquisa referente a ferramentas de medição de métricas em software OO. A equipe de desenvolvimento concluiu com a pesquisa que as ferramentas disponíveis para o propósito não eram boas. Então o JMetric começou a ser desenvolvida pela Universidade tecnológica de Swinburne para coletar métricas em projetos Java. A ferramenta é *open-source* e continua sendo melhorada. Ao iniciar o JMetric é apresentada a tela principal figura 15.

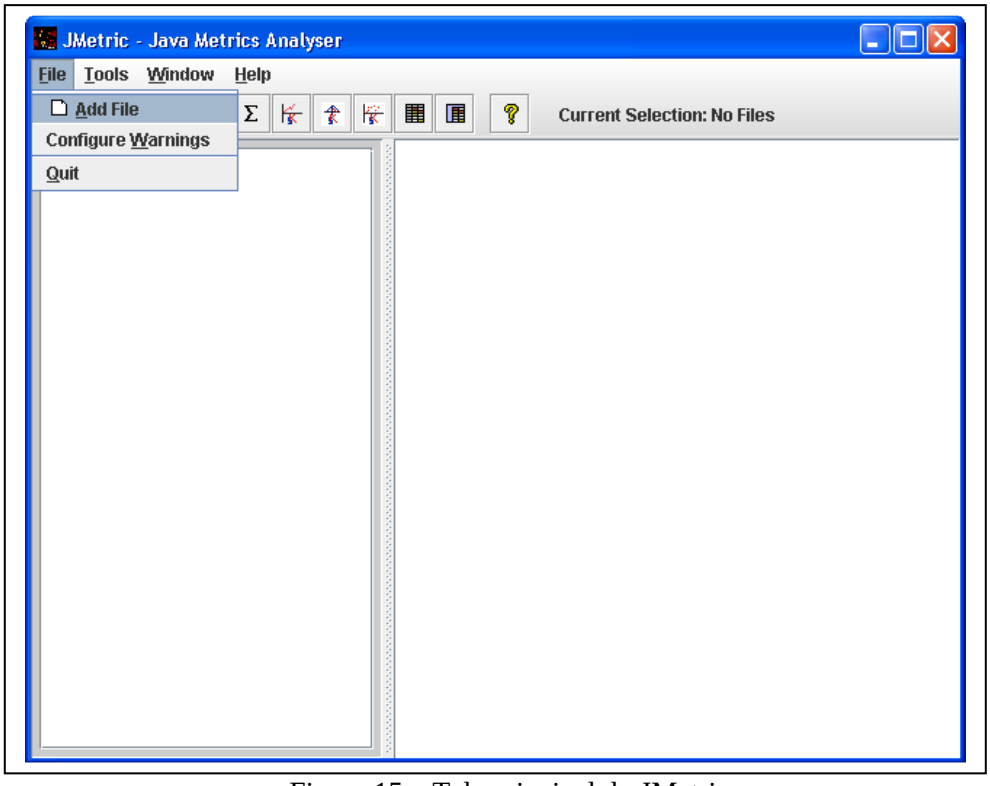


Figura 15 – Tela principal do JMetric

Para efetuar o cálculo das métricas do sistema o usuário deverá selecionar um projeto. Deverá ser adicionado um arquivo escrito em Java. Após adicionar o arquivo, é apresentado uma árvore com a estrutura das classes, neste casso o objeto `TokenUtils` figura 16.

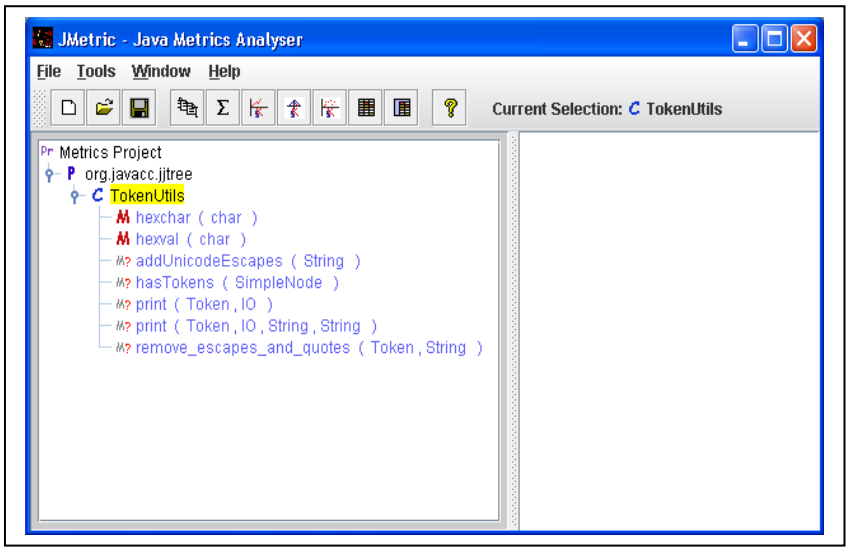


Figura 16 – Árvore com a estrutura da(s) classe(s)

Após a escolha do arquivo para o cálculo, é necessário calcular figura 17.

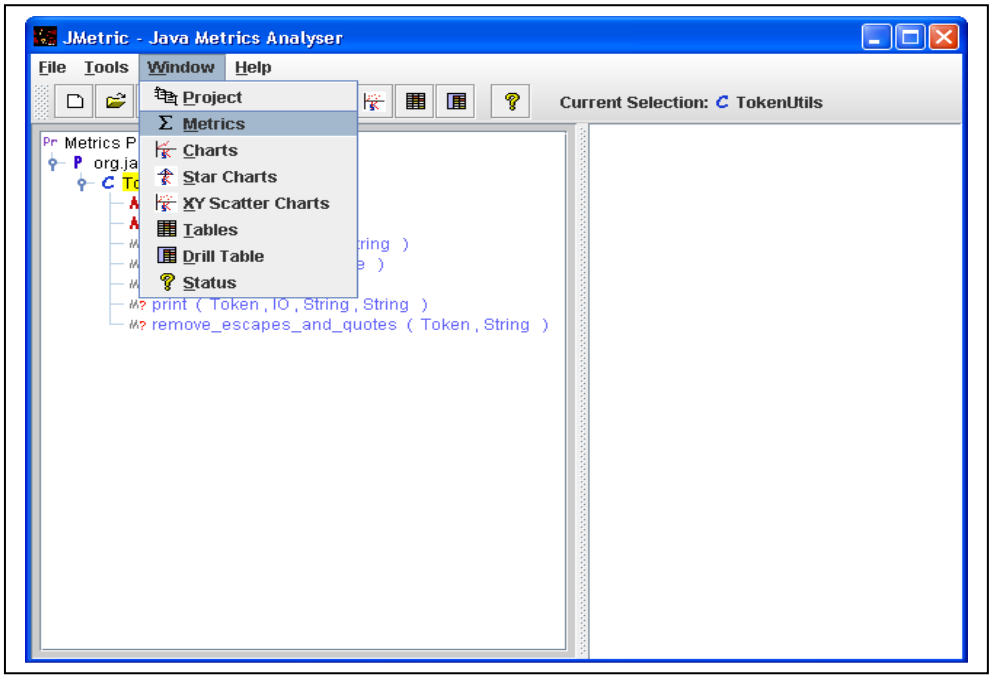


Figura 17 – Calcular as métricas JMetrics

Após calcular as medidas, pose-se exibir de duas maneiras, tabelas e gráficos, de acordo com a documentação do projeto, portanto a exibição em forma de tabela não foi possível, pois a ferramenta não apresentou nenhum resultado o gráfico é demonstrado na figura 18.

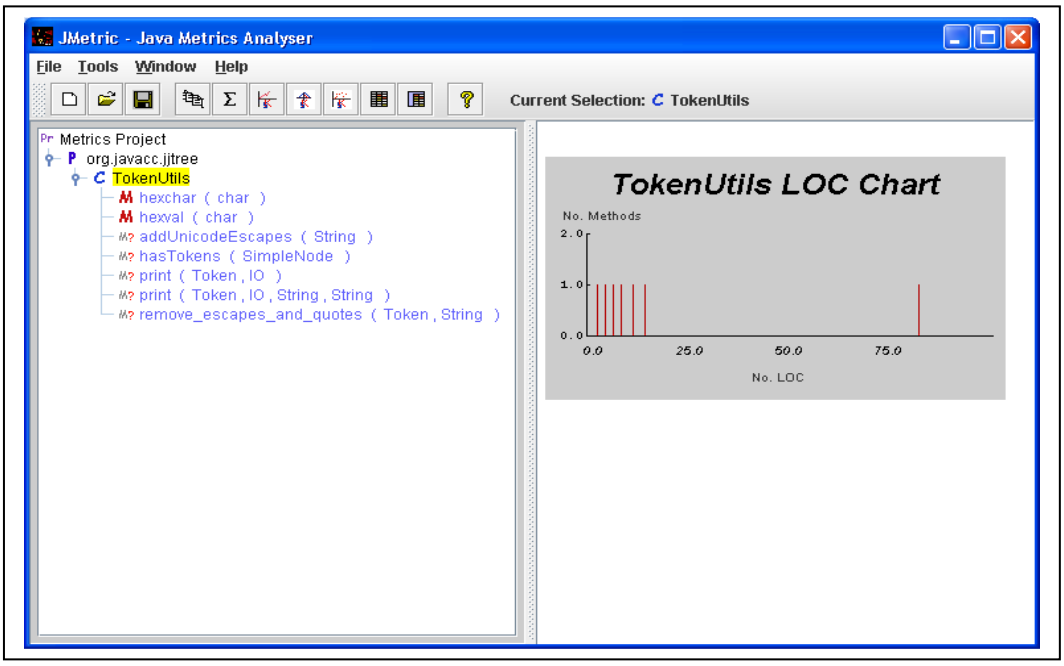


Figura 18 – Demonstração do gráfico do JMetric

3.3.2 FERRAMENTA PARA CÁLCULO DE MÉTRICAS EM SOFTWARE ORIENTADOS A OBJETOS CODIFICADOS EM DELPHI

Em Seibt (2001) é descrito um protótipo de uma ferramenta para cálculo de métricas em software orientado a objetos. O protótipo é capaz de analisar o código fonte de um projeto OO em Delphi, extraindo as classes, seus métodos e atributos para posterior cálculo de métricas para software OO. A ferramenta permite calcular dezenove métricas de projeto e de construção, entre estas, profundidade da árvore de herança e métodos ponderados por classe.

3.3.3 MÉTRICAS PARA PROGRAMAÇÃO ORIENTADA A OBJETOS

Em Cardoso (1999) é descrito o software implementado para facilitar o cálculo de métricas em sistemas OO. O sistema analisa o código fonte de projetos em Delphi e fornece cálculos de métricas como contagem de métodos, classes sucessoras, ascendentes e descendentes específicas para OO.

4 DESENVOLVIMENTO DO TRABALHO

O presente trabalho resultou na criação de um software que possibilita analisar código fonte OO em C# e Java e fornecer algumas das métricas estudadas.

A seguir serão informados detalhes sobre requisitos, especificação e implementação do software.

4.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

O objetivo do desenvolvimento deste trabalho foi disponibilizar uma ferramenta capaz de calcular métricas pré-definidas a partir da análise de código fonte de projetos codificados em C# e Java. As informações para cálculo das métricas são coletadas através da extração das classes, métodos e de seus atributos através da análise dos arquivos de um projeto em C# e Java, exemplificada na figura 19. Após a coleta destes dados são calculadas as métricas estudadas.

Para extrair e identificar as informações do código fonte os dados das classes, foram utilizadas o *Scanner* e o *Parser* gerados pela ferramenta *CocoR for C#* de acordo com a gramática das classes do Apêndice “A”. O *Scanner* separa os *Tokenz* e o *Parser* verifica a sintaxe do projeto.

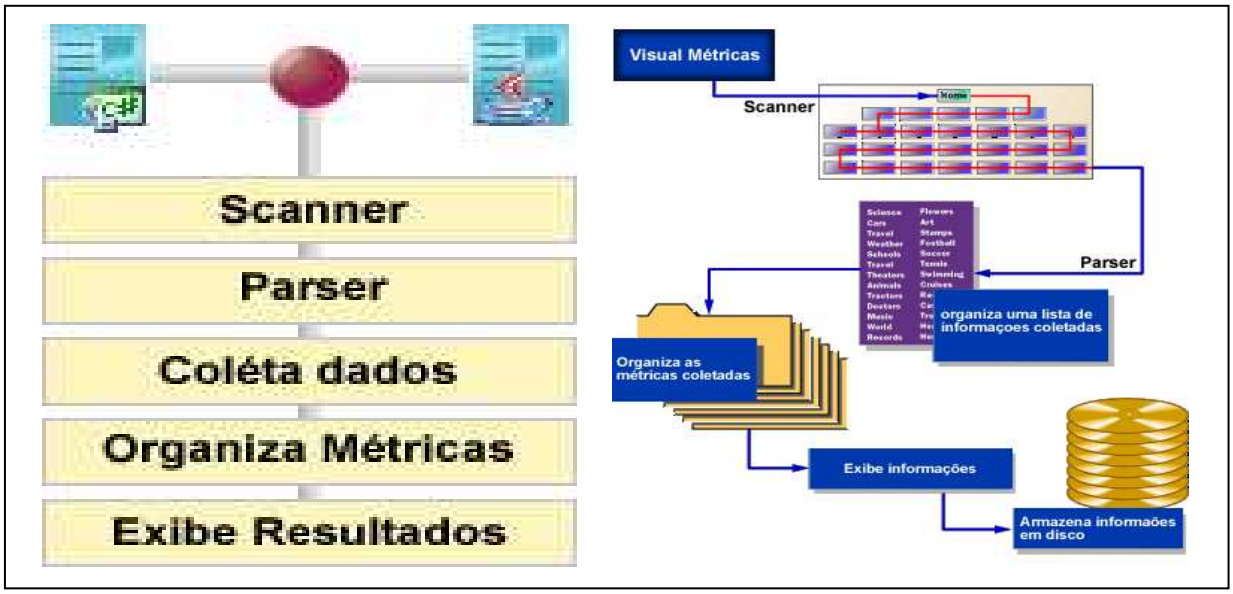


Figura 19 – Esquema de funcionamento do protótipo

4.2 MÉTRICAS SELECIONADAS

As métricas de projeto e de construção são as mais indicadas para se obter através da análise do código fonte, pois a maioria das informações necessárias para o cálculo destas métricas pode ser obtida através de análise automática do código fonte. As métricas a seguir detalhadas no item 3.1 e 3.2 foram as implementadas pelo software:

a) número de serviços por classe;
b) número máximo de superclasses;
c) número de subclasses;
d) nível de acoplamento entre as classes;
e) número de acesso a um ou mais atributos em comum pelos serviços da própria classe.;
f) capacidade de resposta da classe;
g) número de serviços e atributos locais e herdados;
h) métodos definidos nas superclasses herdados pelas subclasses;
i) número de operações e atributos privados;
j) número de serviços adicionados.

4.3 ESPECIFICAÇÃO DO SOFTWARE

Para a especificação do software foi utilizada a UML, que é apresentado através do diagrama de caso de uso, diagrama de classes e do diagrama de seqüência. Estes diagramas serão apresentados a seguir construídos na ferramenta EA.

4.3.1 DIAGRAMA DE CASO DE USO

A ferramenta criada tem por objetivo facilitar a coleta de métricas em códigos fontes. A figura 20 exibe o diagrama de caso de uso da ferramenta. O Quadro 2 descreve os casos de uso exibidos na figura 20, que se refere à iteração do arquiteto com os casos de uso.

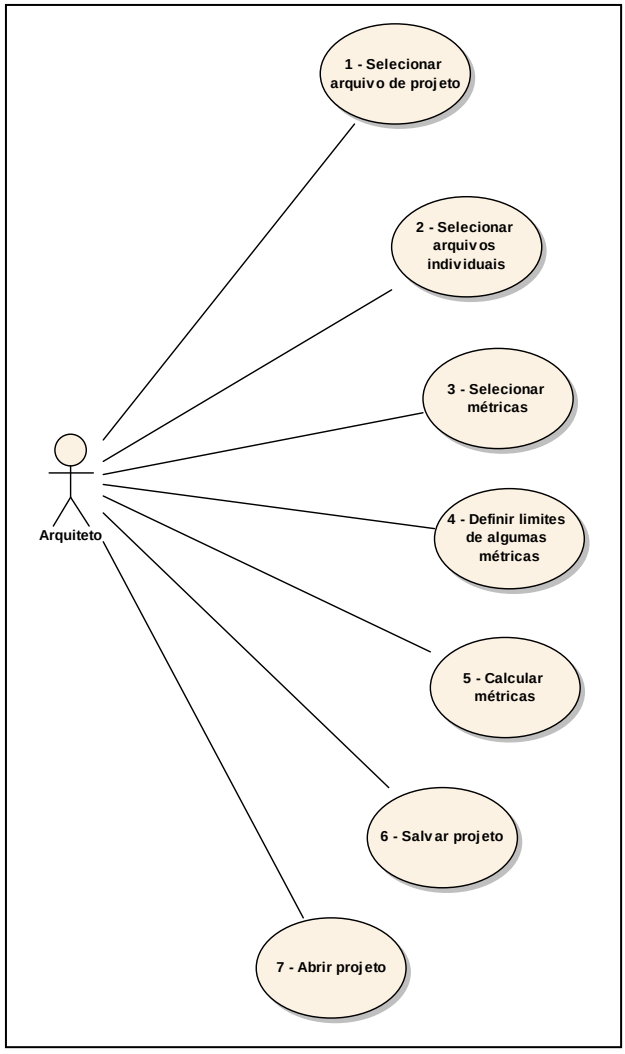


Figura 20 – Diagrama de caso de uso

Caso de Uso	**Descrição**
Selecionar arquivos de projeto	Permite ao arquiteto de software escolher o projeto que será analisado, que pode ser um escrito em C# ou Java.
Seleciona arquivos individuais	Permite ao arquiteto de software selecionar arquivos individualmente, para uma análise específica de um arquivo escrito em Java ou C#.
Selecionar métricas	Permite ao arquiteto de software escolher as métricas que serão calculadas.

Salvar projeto	Permite ao arquiteto de software gravar o resultado do calculo das métricas.
Abrir projeto	Permite ao arquiteto de software carregar um projeto gravado anteriormente.
Definir limites de algumas métricas	Permite ao arquiteto de software definir o limite mínimo e máximo que uma determinada métrica pode atingir.
Calcular métricas	Permite ao arquiteto de software calcular as métricas a partir das informações selecionadas anteriormente.

Quadro 2 – Descrição dos casos de uso do ambiente de coleta de métricas

4.3.2 DIAGRAMA DE CLASSES

No desenvolvimento do software foram identificadas cinqüenta e três classes, destas serão exemplificas métodos e atributos públicos de vinte e sete classes (figura 21) que são utilizadas para armazenar as informações coletadas do código fonte e posteriormente auxiliar na obtenção das métricas. As outras vinte e seis classes estão dividias entre interfaces e auxiliares, não influenciam na exemplificação do funcionamento da ferramenta.

A classe `ColetaMetricas` é responsável por iniciar a análise do código fonte dos arquivos do projeto selecionado e a partir desta análise alimentar a classe `ColetaMetricas`, com objetos `Classe`, que por sua vez será composta com as classes `Atributos`, `Servicos` e `Assinatura`. O Quadro 3 descreve os métodos da classe `ColetaMetricas`.

Métodos	**Descrição**
ColetaMetricas()	Operação responsável pela criação da classe.
AddClasse()	Adiciona uma nova classe para coleta das métricas.
GetClasse()	Retorna um objeto `Classe`.
Coletar()	Operação responsável por iniciar a análise do código fonte para extração dos dados do projeto para posterior cálculo das métricas.

Quadro 3 – Descrição dos métodos da classe `ColetaMetricas`.

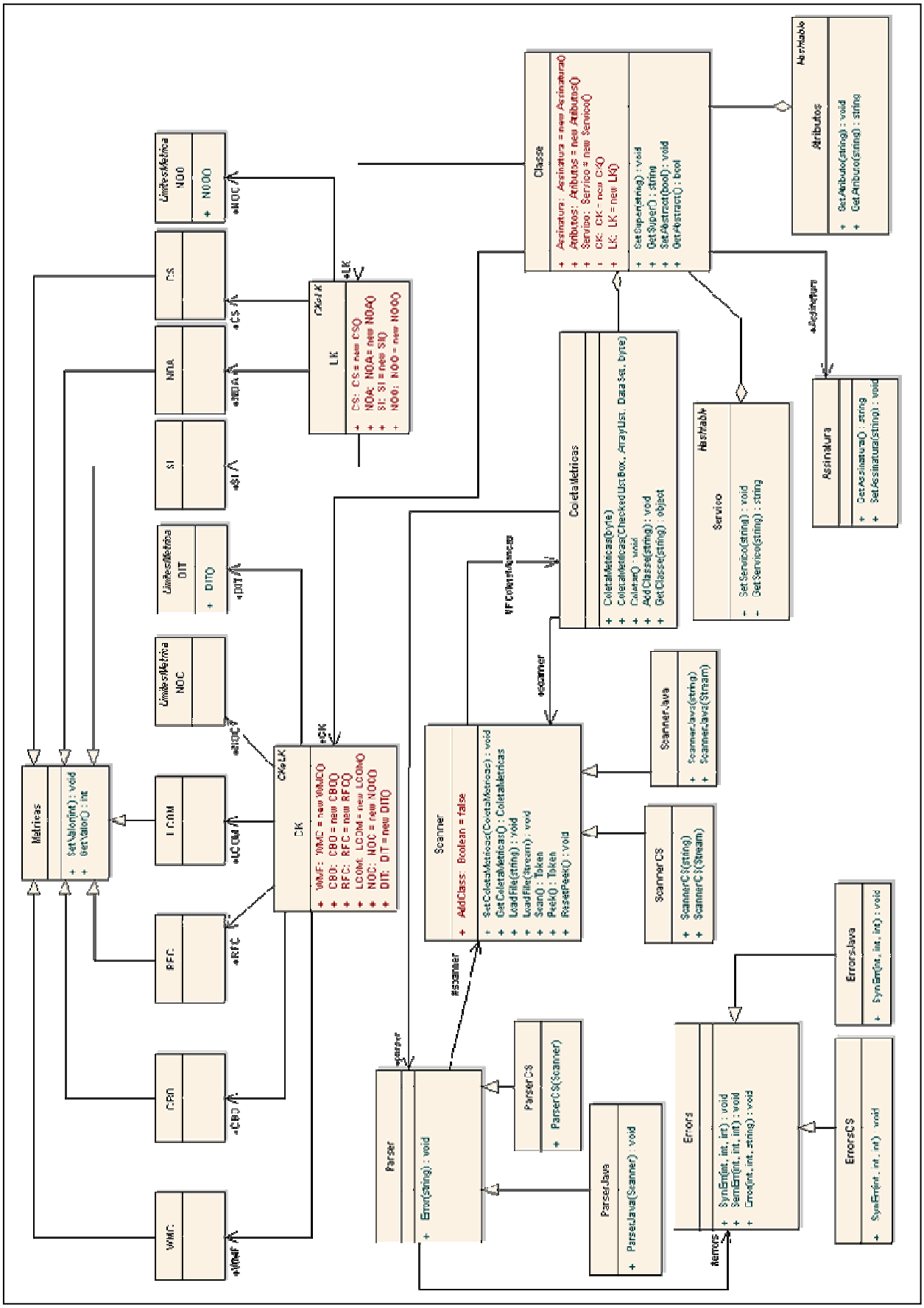


Figura 21 – Diagrama de classes

A classe `Scanner` é a que auxilia na análise do código fonte. A mesma faz uma busca e extração de identificadores, símbolos especiais e palavras reservadas. O Quadro 4 descreve os métodos e atributos da classe `Scanner`.

Métodos	**Descrição**
SetColetaMetricas()	Informa para a classe `Scanner` o objeto `ColetaMetricas` descrito anteriormente.
GetColetaMetricas()	Retorna um objeto `ColetaMetricas`.
LoadFile()	Responsável por carregar o arquivo a ser analisado.
Scan()	Operação responsável por pegar o próximo símbolo para a análise.
Peek()	Avança para o próximo símbolo.
ResetPeek()	Certifica-se de que o Peek() comece pela posição atual da busca.
Atributos	**Descrição**
AddClass	Armazena se foi adicionado uma classe.

Quadro 4 – Descrição dos métodos da classe `Scanner`.

A classe `ScannerCS` é uma extensão da classe `Scanner`, é a que auxilia na análise do código fonte escrito em C#. O Quadro 5 descreve os métodos da classe `ScannerCS`.

Métodos	**Descrição**
ScannerCS(string)	Operação responsável pela criação da classe, informando o caminho do arquivo que será analisado.
ScannerCS(Stream)	Operação responsável pela criação da classe, informando o arquivo já carregado que será analisado.

Quadro 5 – Descrição dos métodos da classe `ScannerCS`.

A classe `ScannerJava` é uma extensão da classe `Scanner`, é a que auxilia na análise do código fonte escrito em Java. O Quadro 6 descreve os métodos da classe `ScannerJava`.

Métodos	**Descrição**
ScannerJava(string)	Operação responsável pela criação da classe, informando o caminho do arquivo que será analisado.
ScannerCS(Stream)	Operação responsável pela criação da classe, informando o arquivo já carregado que será analisado.

Quadro 6 – Descrição dos métodos da classe `ScannerJava`.

A classe `Parser` é responsável pela análise léxica e sintática do código fonte. O Quadro 7 descreve os métodos da classe `Parser`.

Métodos	**Descrição**
Error()	Operação responsável pelo disparo da mensagem de erro caso o *parsing* retorne algum erro.

Quadro 7 – Descrição dos métodos da classe `Parser`.

A classe `ParserCS` é uma extensão da classe `Parser`, é a que auxilia na análise léxica e sintática do código fonte escrito em C#. O Quadro 8 descreve os métodos da classe `ParserCS`.

Métodos	**Descrição**
ParserCS()	Operação responsável pela criação da classe e iniciar a análise.

Quadro 8 – Descrição dos métodos da classe `ParserCS`.

A classe `ParserJava` é uma extensão da classe `Parser`, é a que auxilia na análise léxica e sintática do código fonte escrito em Java. O Quadro 9 descreve os métodos da classe `ParserJava`.

Métodos	**Descrição**
ParserJava()	Operação responsável pela criação da classe e iniciar a análise.

Quadro 9 – Descrição dos métodos da classe `ParserJava`.

A classe `Errors` é responsável pela notificação dos erros encontrados na análise do código fonte. O Quadro 10 descreve os métodos da classe `Errors`.

Métodos	**Descrição**
SynErr()	Operação responsável pela notificação dos erros sintáticos.
SemErr()	Operação responsável pela notificação dos erros semânticos em uma futura implementação.
Error()	Operação responsável pela notificação de erros diversos como arquivo corrompido.

Quadro 10 – Descrição dos métodos da classe `Errors`.

A classe `ErrorsCS` é uma extensão da classe `Errors`, é a que auxilia nos erros da análise do código fonte escrito em C#. O Quadro 11 descreve os métodos da classe `ErrorsCS`.

Métodos	**Descrição**
SynErr()	Operação responsável pela notificação dos erros sintáticos do código fonte C#.

Quadro 11 – Descrição dos métodos da classe `ErrorsCS`.

A classe `ErrorsJava` é uma extensão da classe `Errors`, é a que auxilia nos erros da análise do código fonte escrito em Java. O Quadro 12 descreve os métodos da classe `ErrorsJava`.

Métodos	**Descrição**
SynErr()	Operação responsável pela notificação dos erros sintáticos do código fonte Java.

Quadro 12 – Descrição dos métodos da classe `ErrorsJava`.

A classe `Classe` contém as informações das classes encontradas no projeto analisado, sempre que uma classe é encontrada, um objeto `Classe` é instanciado a partir do método `AddClasse()` da classe `ColetaMetricas` (Quadro 14). O Quadro13 descreve os métodos e atributos da classe `Classe`.

Métodos	**Descrição**
SetSuper()	Informa para o objeto `Classe` qual é a classe antecessora da mesma.
GetSuper()	Retorna a descrição da classe antecessora.
SetAbstract()	Informar se a classe é abstrata ou não.
SetAbstract()	Retorna se a classe é abstrata ou não.
Atributos	**Descrição**
Assinatura	Contém o nome da classe.
Atributos	Contém os atributos da classe.
Serviço	Contém os serviços ou métodos da classe.
CK	Contém as métricas previstas por Chidamber e Kemerer.
LK	Contém as métricas previstas por Lorenz e Kidd.

Quadro 13 – Descrição dos métodos da classe `Classe`.

```
public void AddClasse(string prClasse)
{
    Classe classe = new Classe();
    classe.Assinatura.SetAssinatura(prClasse);
    Classes.Add(prClasse.ToUpper(), classe);
}
```

Quadro 14 – Instancia um objeto de `Classe`

A classe `Atributo` contém as informações dos atributos das classes encontradas no projeto analisado. O Quadro 15 descreve os métodos da classe `Atributos`.

Métodos	**Descrição**
SetAtributo()	Adiciona um atributo no objeto `Atributo`.
GetAtributo()	Retorna a descrição do atributo solicitado.

Quadro 15 – Descrição dos métodos da classe `Atributos`.

A classe `Servico` contém as informações dos serviços ou métodos das classes encontradas no projeto analisado. O Quadro 16 descreve os métodos da classe `Servico`.

Métodos	**Descrição**
SetServico()	Adiciona um serviço no objeto `Serviço`.
GetServico()	Retorna a descrição do serviço solicitado.

Quadro 16 – Descrição dos métodos da classe `Servico`.

A classe `Assinatura` é a que auxilia para definir a descrição de outras classes como exemplo a classe `Classe`. O Quadro 17 descreve os métodos da classe `Assinatura`.

Métodos	**Descrição**
SetAssinatura()	Informa para o objeto `Assinatura` a descrição da mesma.
GetAssinatura()	Retorna a descrição do objeto `Assinatura`.

Quadro 17 – Descrição dos métodos da classe `Assinatura`.

A classe `CK` contém as informações das métricas previstas por Chidamber e Kemerer descritas no item 3.1.1. O Quadro 18 descreve os atributos da classe `CK`.

Atributos	**Descrição**
WMC	Contém o objeto de métrica `WMC`.
CBO	Contém o objeto de métrica `CBO`.
RFC	Contém o objeto de métrica `RFC`.
LCOM	Contém o objeto de métrica `LCOM`.
NOC	Contém o objeto de métrica `NOC`.
DIT	Contém o objeto de métrica `DIT`.

Quadro 18 – Descrição dos atributos da classe `CK`.

A classe `LK` contém as informações das métricas previstas por Lorenz e Kidd descritas no item 3.1.2. O Quadro 19 descreve os atributos da classe `LK`.

Atributos	**Descrição**
CS	Contém o objeto de métrica `CS`.
NOA	Contém o objeto de métrica `NOA`.
SI	Contém o objeto de métrica `SI`.

NOO	Contém o objeto de métrica NOO.

Quadro 19 – Descrição dos atributos da classe LK.

A classe Metricas contém as informações referentes à quantidade calculada da métrica. A Quadro 20 descreve os métodos da classe Metrica.

Atributos	**Descrição**
SetValor()	Informar para o objeto Metricas o valor calculado.
GetValor()	Retorna o valor calculado.

Quadro 20 – Descrição dos métodos da classe Metricas.

As classes WMC, CBO, RFC, LCOM, NOC, DIT, SI, NOA, CS e NOO são extensões da classe Métricas descrita no Quadro 20, dando um destaque para as classes DIT e NOO que especializaram o construtor da classe Metricas. O Quadro 21 e 22 descreve os métodos das classes DIT e NOO respectivamente.

Atributos	**Descrição**
DIT()	Operação responsável pela criação da classe.

Quadro 21 – Descrição dos métodos da classe DIT.

Atributos	**Descrição**
NOO()	Operação responsável pela criação da classe.

Quadro 22 – Descrição dos métodos da classe NOO.

4.3.3 DIAGRAMA DE ATIVIDADES

Na figura 22 é apresentado o diagrama de atividades, que representa os passos para a realização do cálculo das métricas. O Quadro 23 descreve as atividades deste processo.

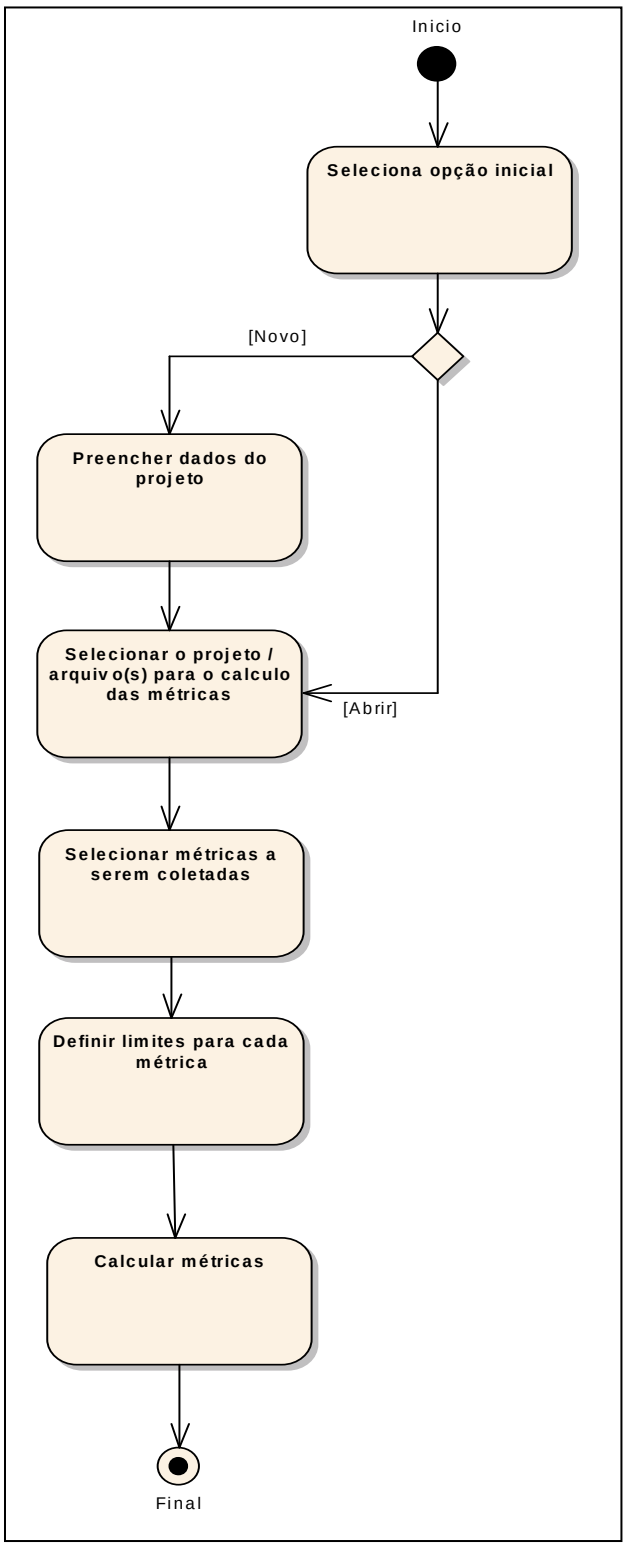


Figura 22 – Diagrama de atividades do processo de coleta das métricas

Atividades	**Descrição**
Seleciona opção inicial	O arquiteto de software escolhe criar um novo projeto ou abrir um projeto existente.
Preencher dados do projeto	O arquiteto de software preencher os dados referentes ao projeto criado.
Selecionar projeto / arquivo(s) para o cálculo das métricas	O arquiteto de software escolhe um projeto escrito em C# ou Java para a coleta das métricas.
Selecionar métricas a serem coletadas	O arquiteto de software define dentre as métricas propostas, as que serão coletadas.
Definir limites para cada métrica	O arquiteto de software informa o limite mínimo e máximo que uma determinada métrica pode atingir.
Calcular métricas	O arquiteto de software executa o cálculo das métricas.

Quadro 23 – Descrição das atividades do processo de coleta das métricas.

4.4 IMPLEMENTAÇÃO

Considerações sobre as técnicas utilizadas para implementação do software, bem como a forma de operação do mesmo, serão apresentadas nesta seção.

No construtor da classe `ColetaMetricas,` é atribuído dentre outros parâmetros, a lista de arquivos e o tipo do projeto a ser coletada as métricas (Quadro 24).

```
public ColetaMetricas(CheckedListBox prListFiles, ArrayList prListaArquivos, DataSet prDataSet, byte prTypeProject)
{
    FListaArquivos = prListaArquivos;
    FListFiles = prListFiles;
    FDataSet = prDataSet;
    FTypeProject = prTypeProject;
}

public void Coletar()
{
    for (int i = 0 ; i < FListFiles.Items.Count ; i++)
    {
        this.parse(FListaArquivos[i].ToString().Replace("'\'", "\\").ToString() + "\\" + FListFiles.Items[i].ToString());
    }
}
```

Quadro 24 – Atribuindo a lista de arquivos para coleta das métricas e o tipo do projeto

O processor de coleta de medidas é iniciado através do método `Coletar()` da classe `ColetaMetricas` (Quadro 24). Este método não recebe parâmetro porque no construtor da

classe já foram passadas todas as informações necessárias para a coleta das métricas.

O método `Coletar()` faz um laço sobre a lista de arquivos informado no construtor da classe `ColetaMetricas` no parâmetro `prListaArquivos` que foi atribuido ao atributo `FlistaArquivos` (Quadro 24). A cada iteração do procedimento `Coletar()` é chamado o método `parse()` que carrega o arquivo informado no parâmetro do mesmo (Quadro 25), é iniciado o processo de análise léxica e sintática do texto do arquivo segundo gramática definida.

```
private void parse(string s)
{
    fstream = File.Open(s, FileMode.Open, FileAccess.Read, FileShare.Read);
    switch (FTypeProject)
    {
        case 0: //C#
            scanner = new ScannerCS(s);
            parser = new ParserCS(scanner);
            break;

        case 1: // Java
            scanner = new ScannerJava(s);
            parser = new ParserJava(scanner);
            break;
    }

    scanner.SetColetaMetricas(this);
    parser.Parse(); // Analisa o código
}
```

Quadro 25 – Executando o analisador léxico e sintático no arquivo informádo

O método `parse()` instancia as classes o atributo `parser` e `scanner` (Quadro 25), de acordo com o atributo `FtypeProject` informado no construtor da classe `ColetaMetricas` (Quadro 24).

Caso o processo de *parsing* ocorra sem nenhum erro, o atributo `Classes` que está definido na classe `ColetaMetricas` (Quadro 26), conterá as métricas coletadas do projeto.

```
public class ColetaMetricas
{
    private ArrayList FListaArquivos;
    private CheckedListBox FListFiles;
    private FileStream fstream;
    private Scanner scanner;
    private Parser parser;
    private Hashtable Classes = new Hashtable();
    private DataSet FDataSet;
    private byte FTypeProject;
```

Quadro 26 – Definição do atributos da classe `ColetaMetricas`

Se existirem erros durante a análise léxica e sintática do texto do arquivo, estes erros

são informados conforme o código ilustrado no Quadro 27. Neste caso, nenhuma métrica é coletada e, portanto, não será exibido as medidas do projeto.

```
public void SynErr(int line, int col, int n)
{
    string s;
    switch (n)
    {
        case 0:
            s = "EOF expected";
            break;
        case 1:
            s = "ident expected";
            break;
        case 2:
            s = "intCon expected";
            break;
        case 3:
            s = "realCon expected";
            break;
        case 4:
            s = "charCon expected";
            break;
        case 5:
            s = "stringCon expected";
            break;
        case 6:
            s = "abstract expected";
            break;
        case 7:
            s = "as expected";
            break;
        case 8:
            s = "base expected";
            break;
        case 9:
            s = "bool expected";
            break;
        case 10:
            s = "break expected";
            break;
        case 11:
            s = "byte expected";
            break;
        case 12:
            s = "case expected";
            break;
        .
        .
        .
```

Quadro 27 – Tratamento de erro

A gramática ilustrada no Apêndice “A” ajudou na construção do analisador para à coleta das métricas, ela identifica de uma classe como por exemplo a declaração, estrutura, declaração da estrutura, métodos e atributos.

4.4.1 TÉCNICAS E FERRAMENTAS UTILIZADAS

O software foi implementado no ambiente de desenvolvimento Microsoft Visual C# 2005 *Express Edition*, utilizando a OO para o desenvolvimento do projeto. Para geração do analisador léxico e sintático foi utilizado o *Coco/R for C#*, o que facilitou bastante para que

fosse dada total atenção à coleta de métricas, a geração das palavras reservadas e da lista de *tokens* foi gerada a partir da gramática do C# e Java Apêndice “A”, para geração do gráfico foi utilizado a biblioteca GDI+ que está disponível no C# e para elaboração da ajuda, foi utilizado o *HelpNDoc*.

4.4.2 OPERACIONALIDADE DA IMPLEMENTAÇÃO

Serão apresentadas as telas do software com suas respectivas funcionalidades. Com o intuito de facilitar a demonstrar e compreensão, será realizado a coleta das métricas de um projeto C# a partir do código fonte das classes da figura 2 que contém as classes `Pessoa`, `Cliente`, `Funcionario`, `Cargo`, `Departamento`, `FuncionarioMensalista`, `FuncionarioHorista` e `FuncionarioDiarista`.

Ao iniciar o sistema será apresentada a tela principal do programa ao usuário como ilustra a figura 23, a ferramenta conta com opções de *menu*, barra de atalho, local destinado a projetos recentes e três opções de *site* com assuntos relacionados a este trabalho.

Figura 23 – Tela principal do software

Para iniciar um novo projeto no sistema o arquiteto de software deverá selecionar a opção “Novo Projeto...” ou clicar em “Projeto...” conforme a figura 24.

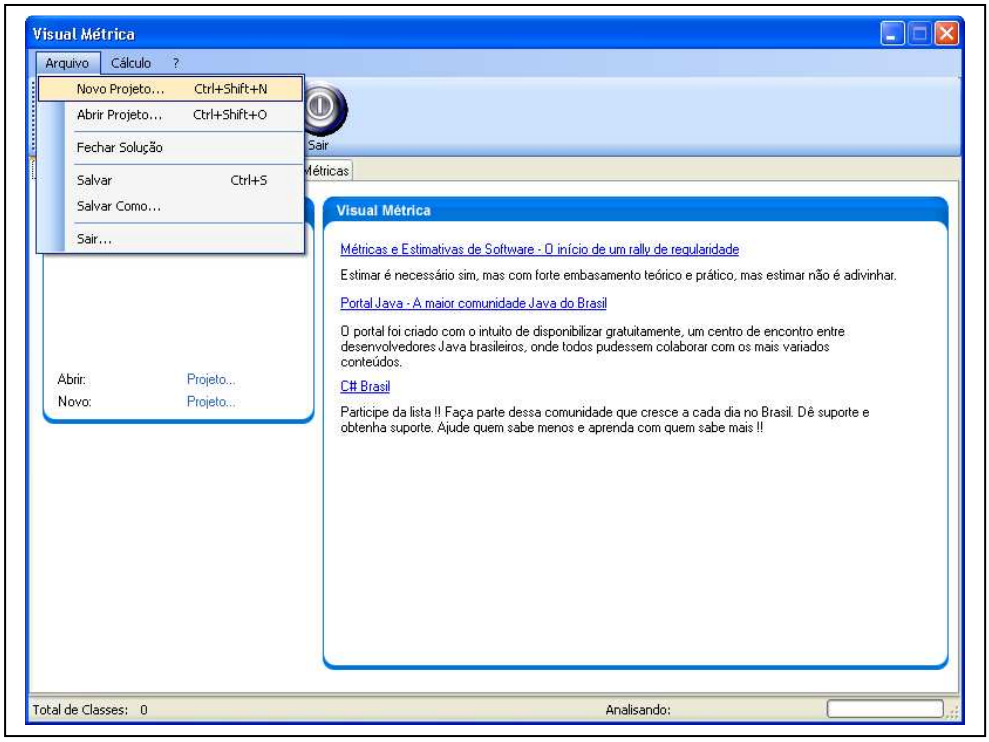


Figura 24 – Criação de um novo projeto no Visual Métrica

Para efetuar o cálculo das métricas de um sistema o arquiteto de software deverá selecionar um projeto. Esta seleção pode ser de duas formas. A primeira é selecionar um novo projeto para coleta de métricas. Após a escolha, a guia “Projeto” é apresentada para o preenchimento dos dados do projeto como pode ser visto na figura 25.

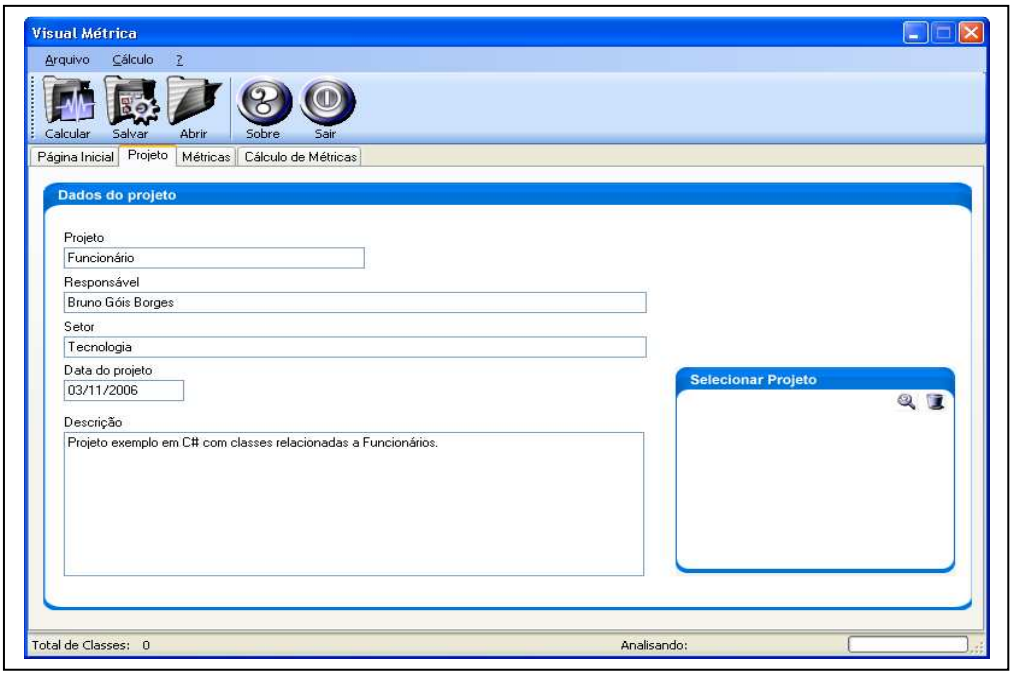


Figura 25 - Guia de informação dos dados do projeto

Após informar os dados do projeto, deverá ser selecionado o projeto ou o(s) arquivo(s) para o cálculo das métricas. Na janela ao lado dos dados do projeto, encontra-se uma outra janela com o título “Selecionar Projeto”. Para escolher um projeto basta clicar no ícone que representa uma lupa e será apresentada uma caixa de diálogo para abrir um arquivo (as extensões dos arquivos a serem aberto são referentes a projeto C#, arquivos individuais do C# e arquivos do Java consecutivamente, “.vcproj”, “.cs”, “.java”) conforme figura 26.

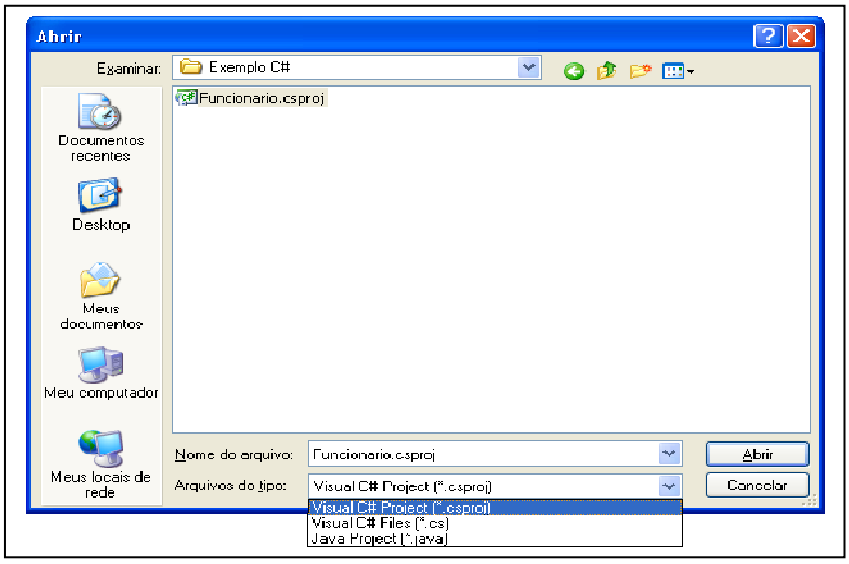


Figura 26 - Tela de abertura do projeto para análise

Após a escolha do projeto a ser analisado, é listado o(s) arquivo(s) na janela “Selecionar Projeto” conforme a Figura 27.

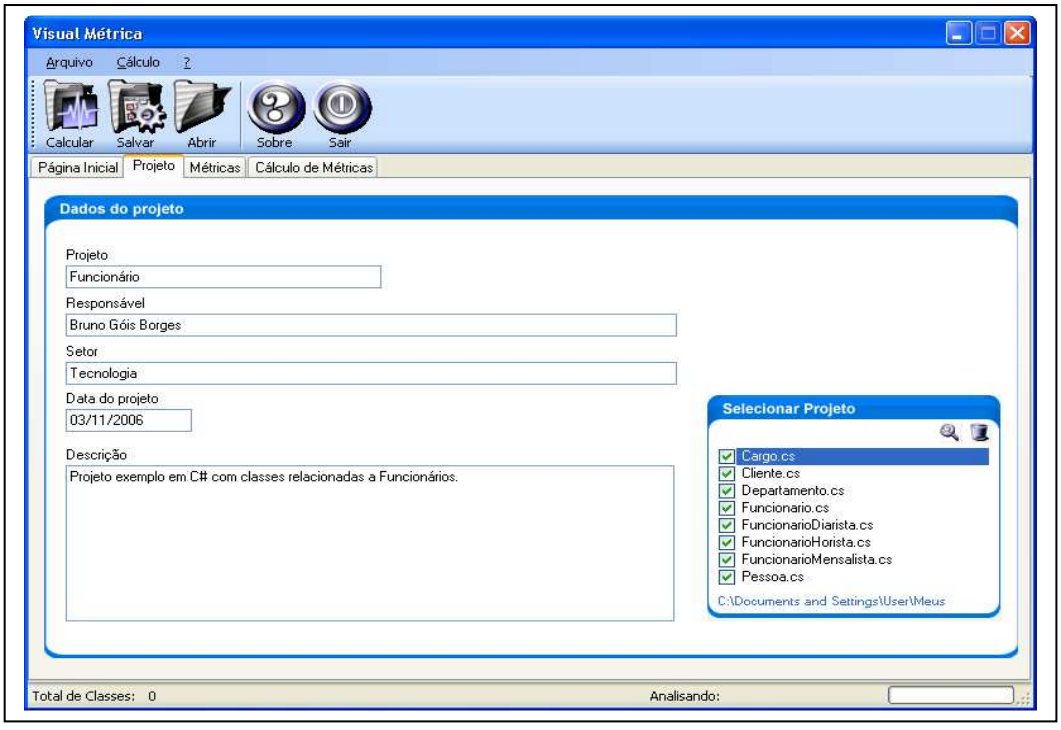


Figura 27 - Lista dos arquivos referentes ao projeto selecionado

Com todos os dados do projeto preenchidos e o projeto escolhido, deverá ser selecionada a guia “Métrica” para a escolha das medidas que serão calculadas para o projeto conforme visto na figura 28.

Figura 28 – Tela de seleção de projetos salvo anteriormente

Outra funcionalidade muito interessante da ferramenta é a possibilidade de definir limite máximo e mínimo de algumas métricas. Esta parametrização é por projeto e influencia diretamente o resultado do cálculo. Ao navegar pelas métricas, a descrição da mesma é descrita na parte inferior da tela na janela Descrição como pode ser visto na figura 29.

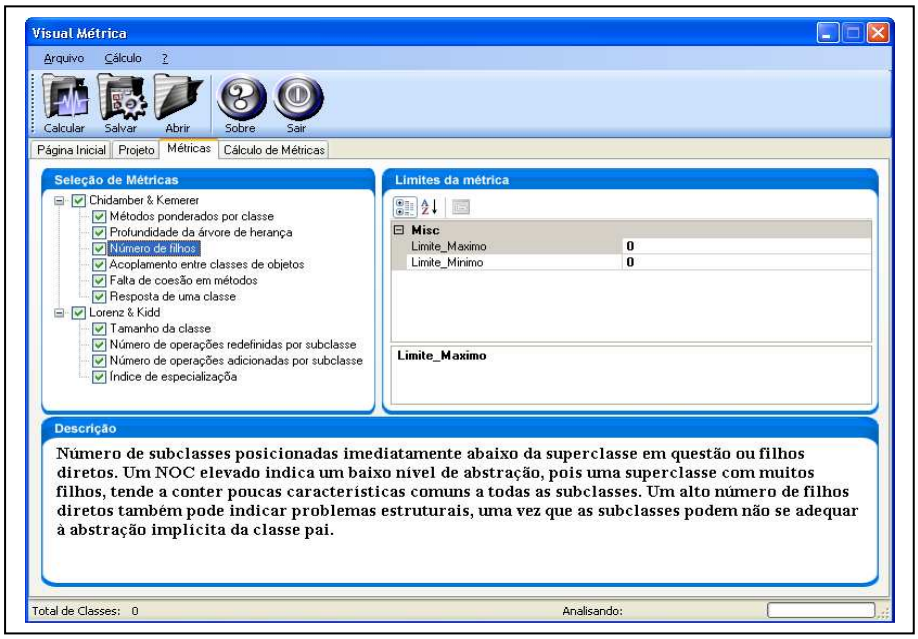


Figura 29 – Tela de parametrização de limite máximo e mínimo por métrica

Após a definição de todos os parâmetros do projeto, informações referentes ao mesmo, métricas a serem calculadas e os limites das métricas, deverão ser calculadas as medidas deste código conforme ilustrada na figura 30.

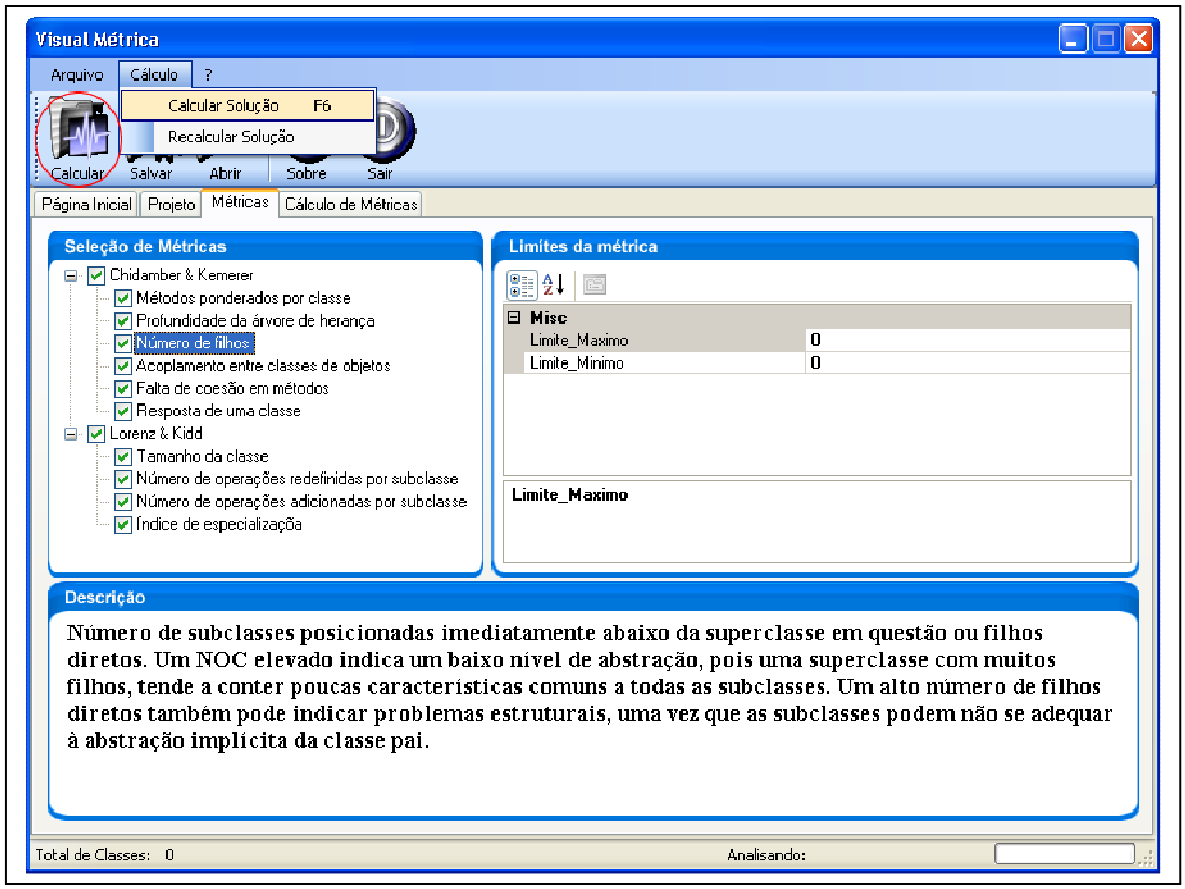


Figura 30 – Tela para o cálculo das métricas

Durante o cálculo das métricas, algumas informações são atualizadas na tela como qual o arquivo que está sendo analisado no momento e uma barra de progressão para informar o percentual do que já foi calculada conforme ilustrada na figura 31.

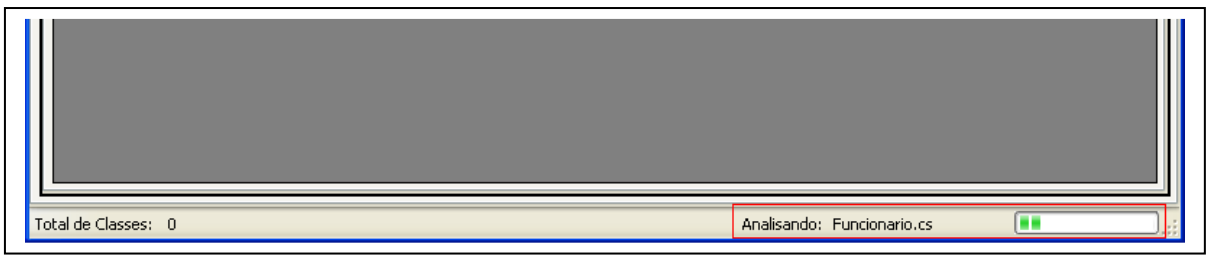


Figura 31 - Tela com informações durante o cálculo

Após o cálculo das métricas os resultados são apresentados como mostra a figura 32 e 33. Estas telas listam todas as métricas selecionadas e calculadas para cada classe do projeto.

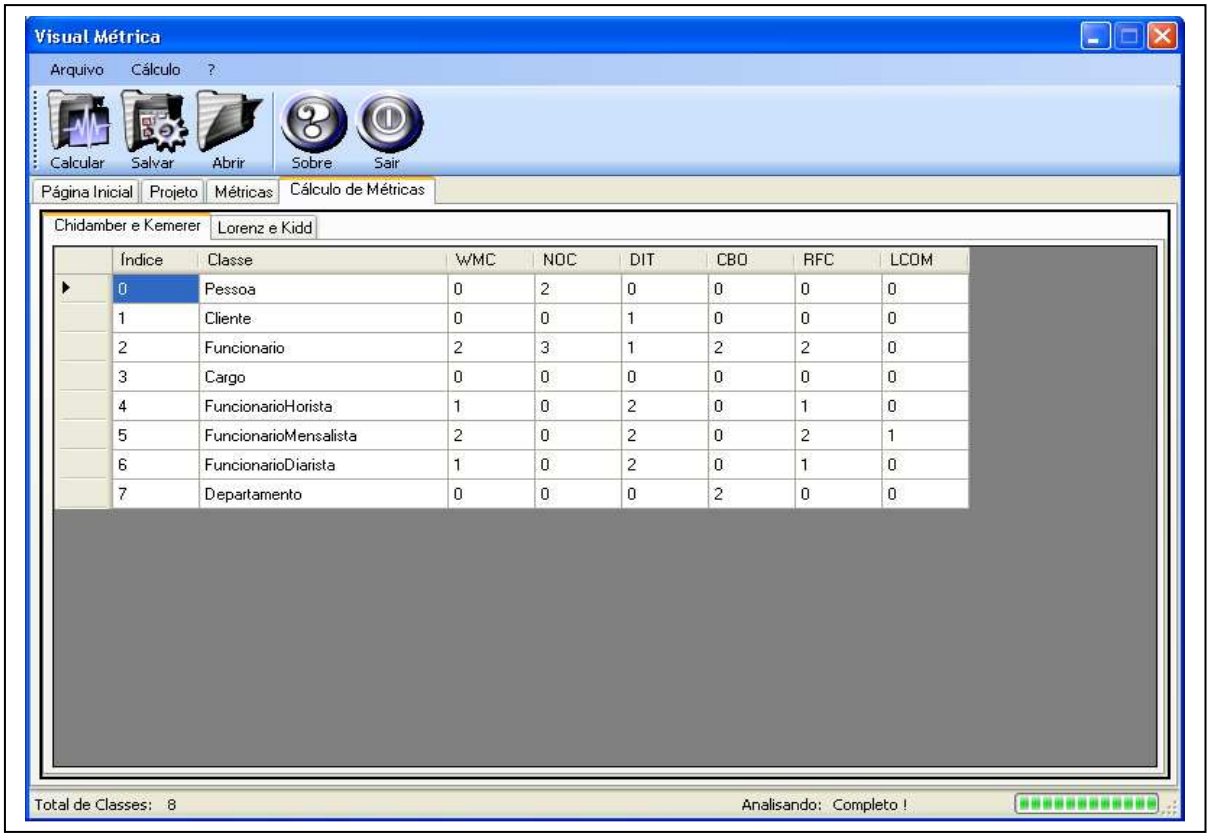


Figura 32 – Resultado do cálculo segundo Chidamber e Kemerer

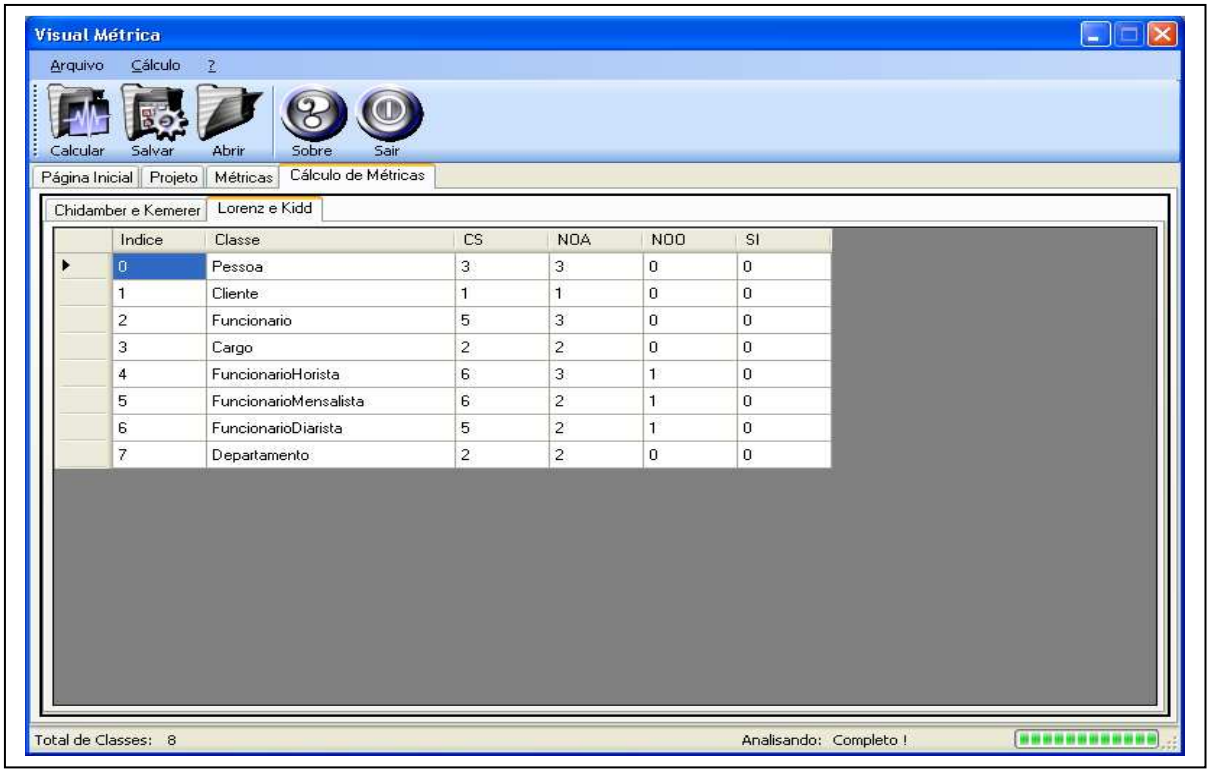


Figura 33 – Resultado do cálculo segundo Lorenz e Kidd

Outra opção interessante desta ferramenta é a possibilidade de analisar os resultados obtidos do cálculo com o auxílio do gráfico proposto por Kiviat. Para analisar a classe escolhida basta clicar com o botão direito do *mouse* na grade dos resultados em cima da classe e será exibida a opção de gráfico conforme é ilustrado na figura 34.

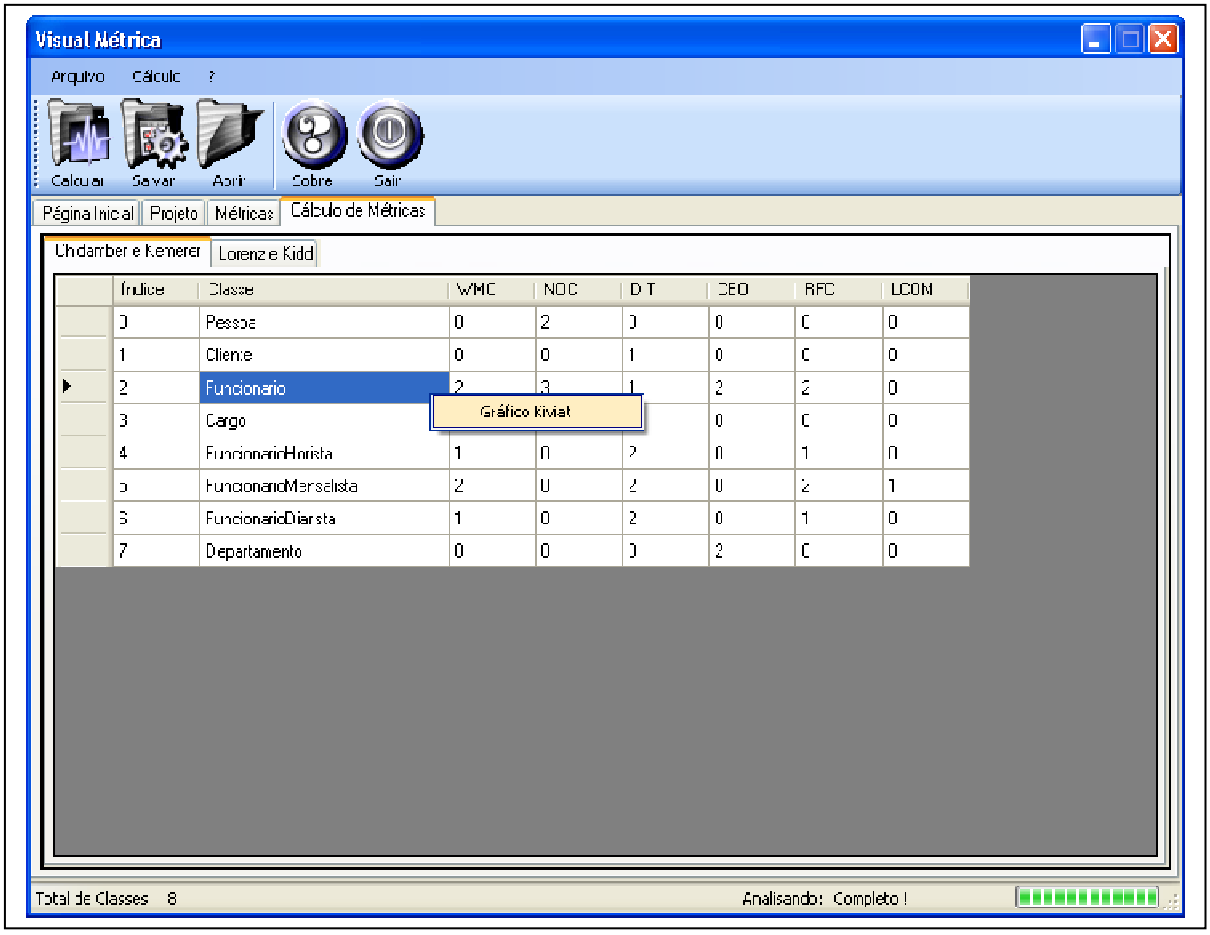


Figura 34 – Opção de análise com o gráfico de Kiviat

O gráfico será gerado com base nas informações calculas para a classe selecionada, respeitando os limites escolhidos na figura 29. Esta opção propõe uma visualização dos dados de uma maneira mais intuitiva e não só com valores em grade que pode se tornar confuso e pouco apresentável. As métricas que influenciam no gráfico são: WMC, DIT, NOO, CBO e LCOM. A Figura 35 demonstra os dados em forma de gráfico exemplificado no item 3.2.

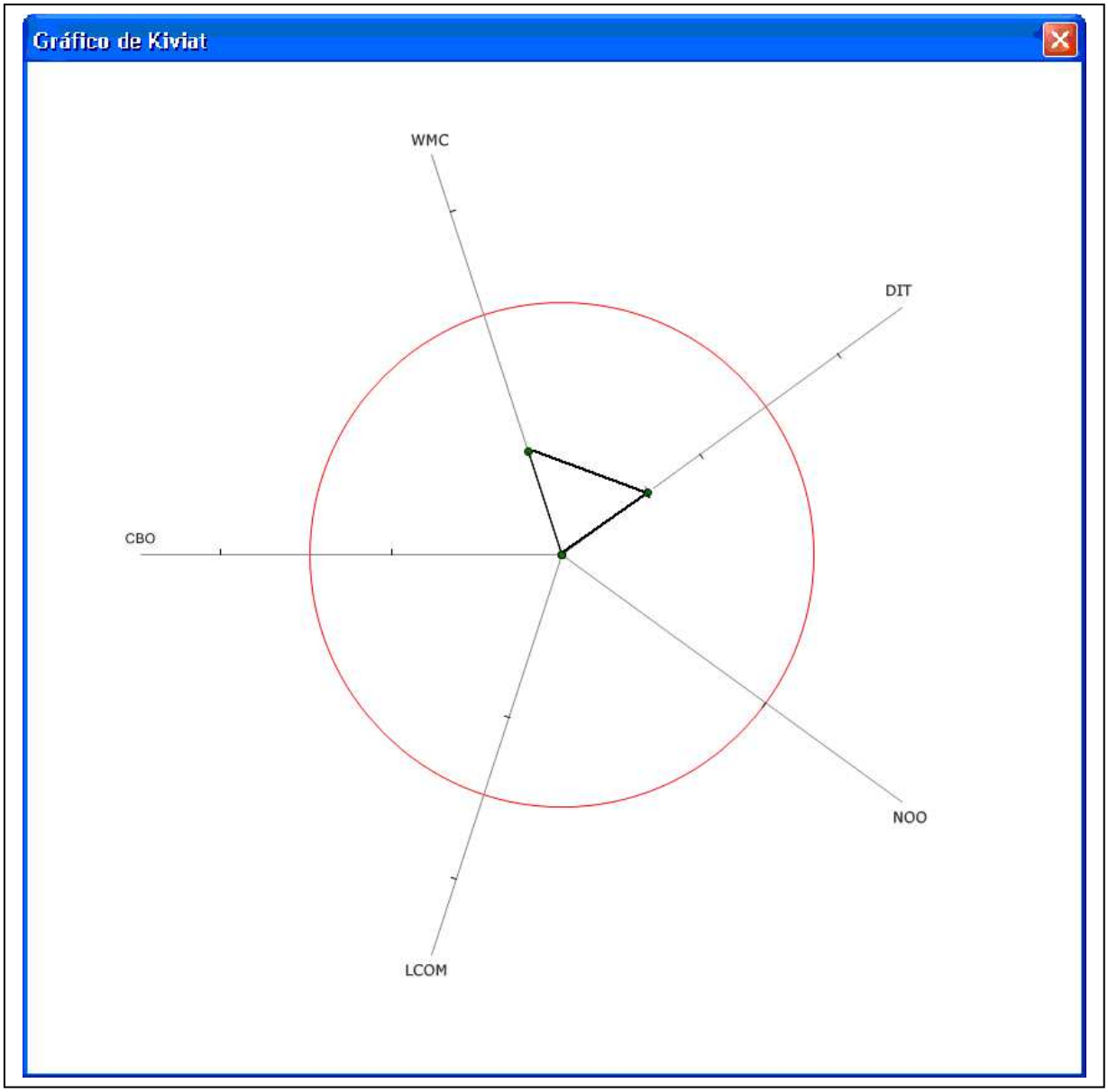


Figura 35 – Gráfico de Kiviat

Depois de analisar um projeto, o arquiteto de software pode salvar suas informações em disco. Para isto basta escolher a opção Arquivo e em seguida Salvar (figura 36). Em seguida é apresentada a tela para escolha do nome que será salvo, o projeto e o seu diretório (figura 37).

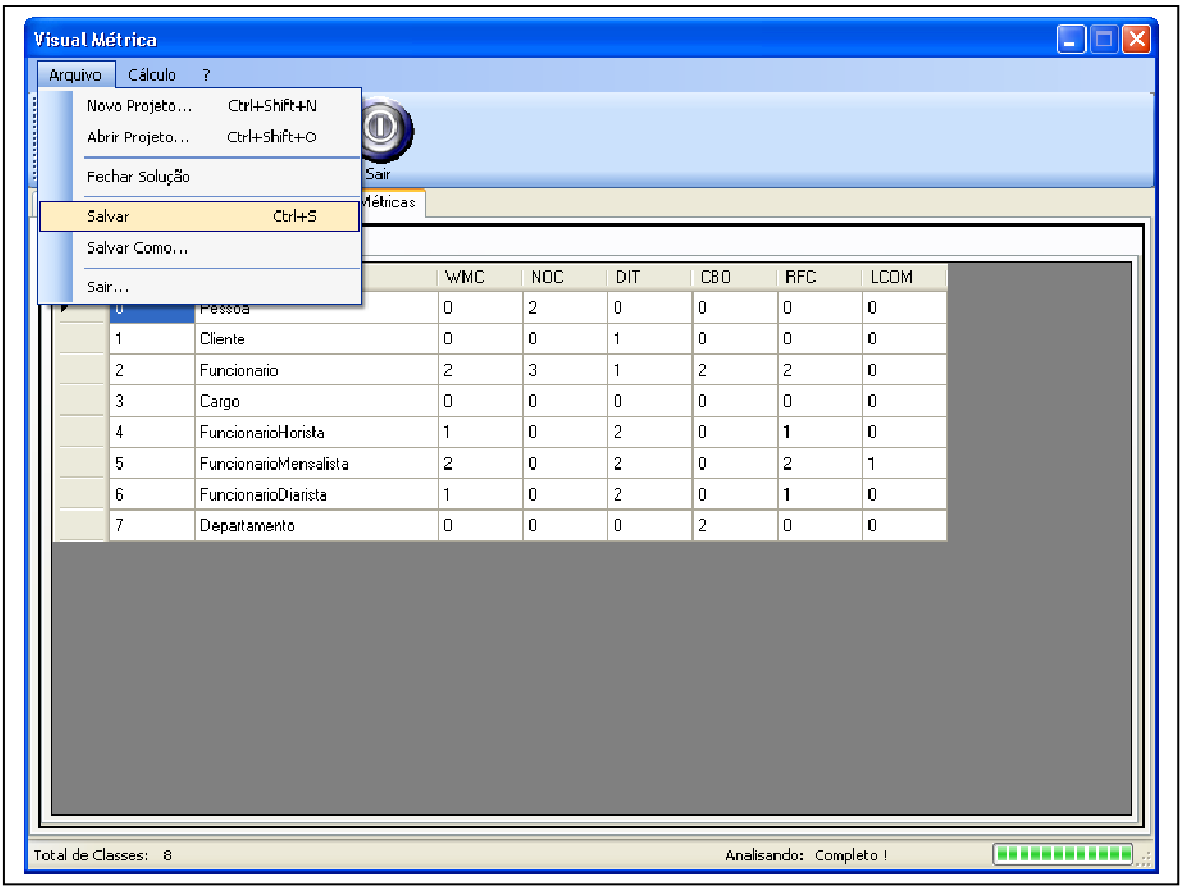


Figura 36 - Tela de opções do projeto

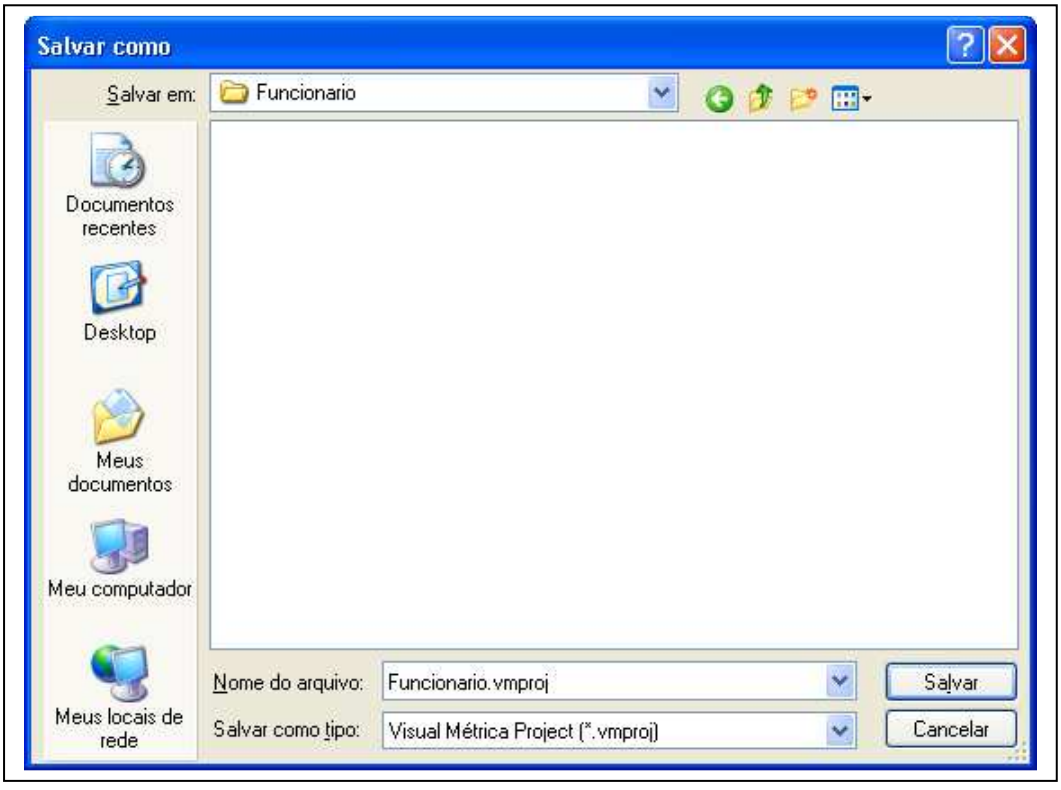


Figura 37 – Tela para escolher o nome do projeto a ser salvo

Outra maneira de escolher um projeto para o cálculo dessas medidas é abrir um projeto analisado previamente e armazenado em disco. Para tanto o arquiteto de software deve selecionar a opção Abrir na barra de atalho ou no menu escolher Arquivo e posteriormente Abrir Projeto. Ainda existe a opção de utilizar as teclas de atalho CTRL+SHIFT+O. O sistema mostrará a tela da figura 38, que permite a escolha de um projeto salvo anteriormente.

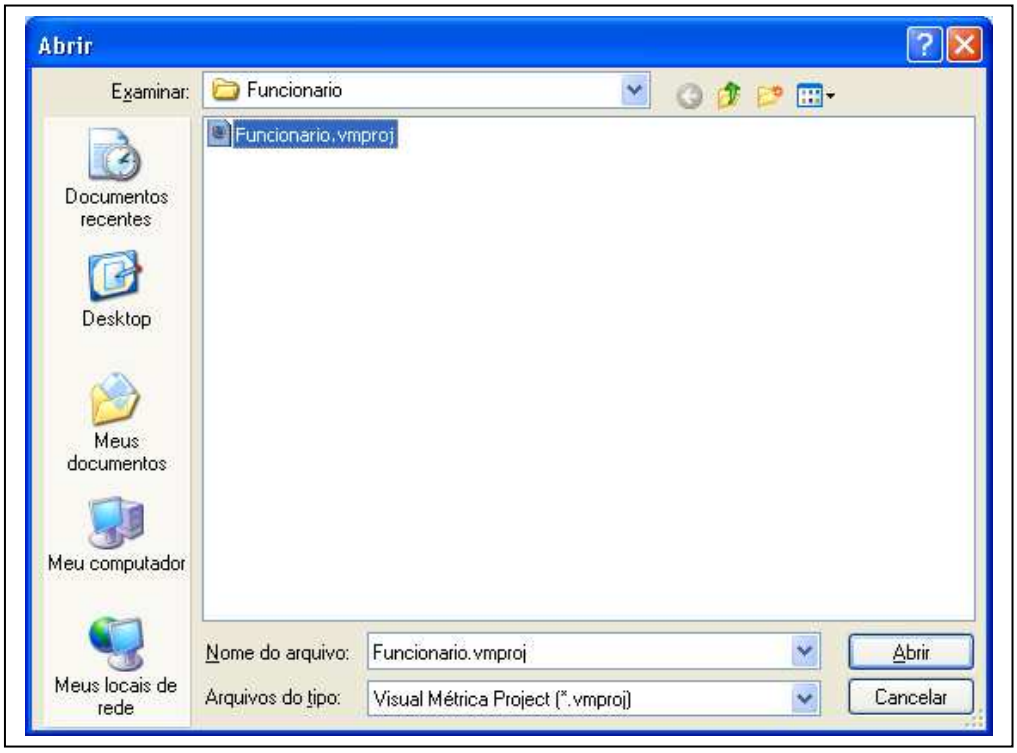
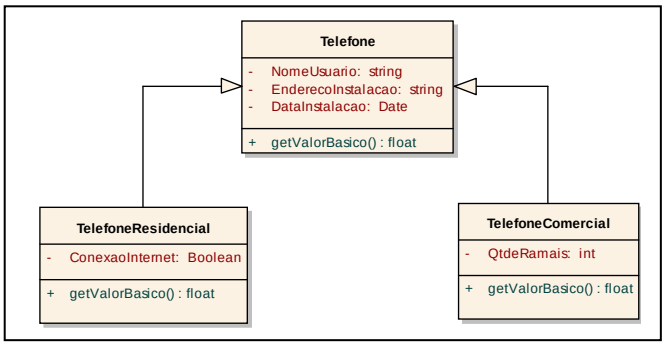


Figura 38 – Tela de seleção de projeto salvo anteriormente

Com o intuito de demonstrar o funcionamento da ferramenta em um projeto escrito em Java, será realizada a coleta das métricas das classes da Figura 39.



Fonte: Hugo e Hübner

Figura 39 – Exemplo para demonstração

Os passos são similares ao demonstrado nas telas anteriores. Os resultados do cálculo são apresentados nas figuras 40 e 41.

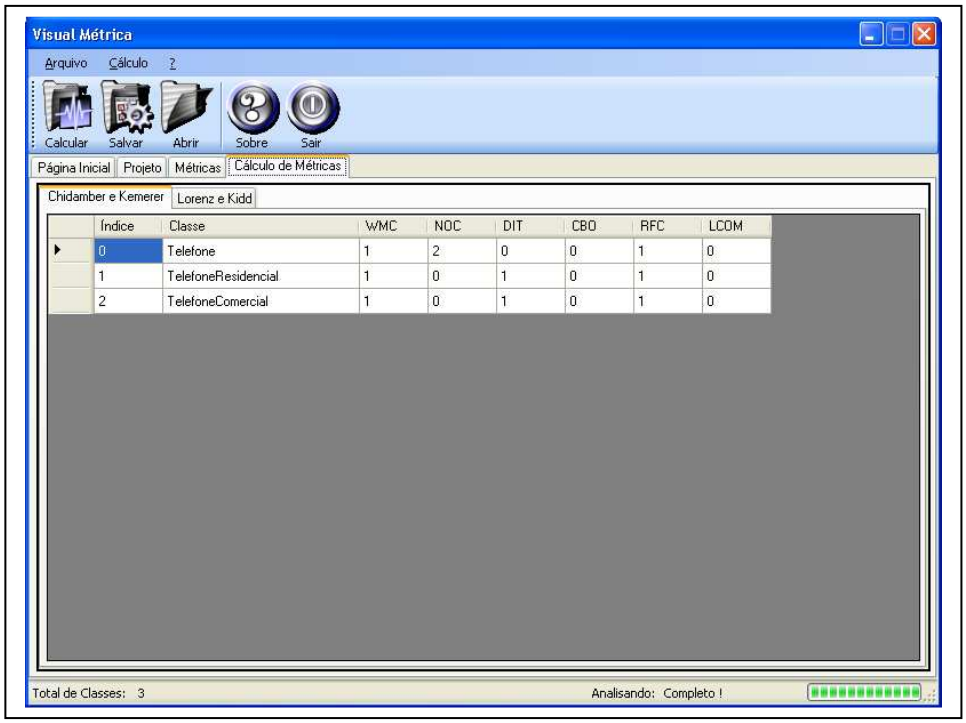


Figura 40 – Resultado do cálculo segundo Chidamber e Kemerer Java

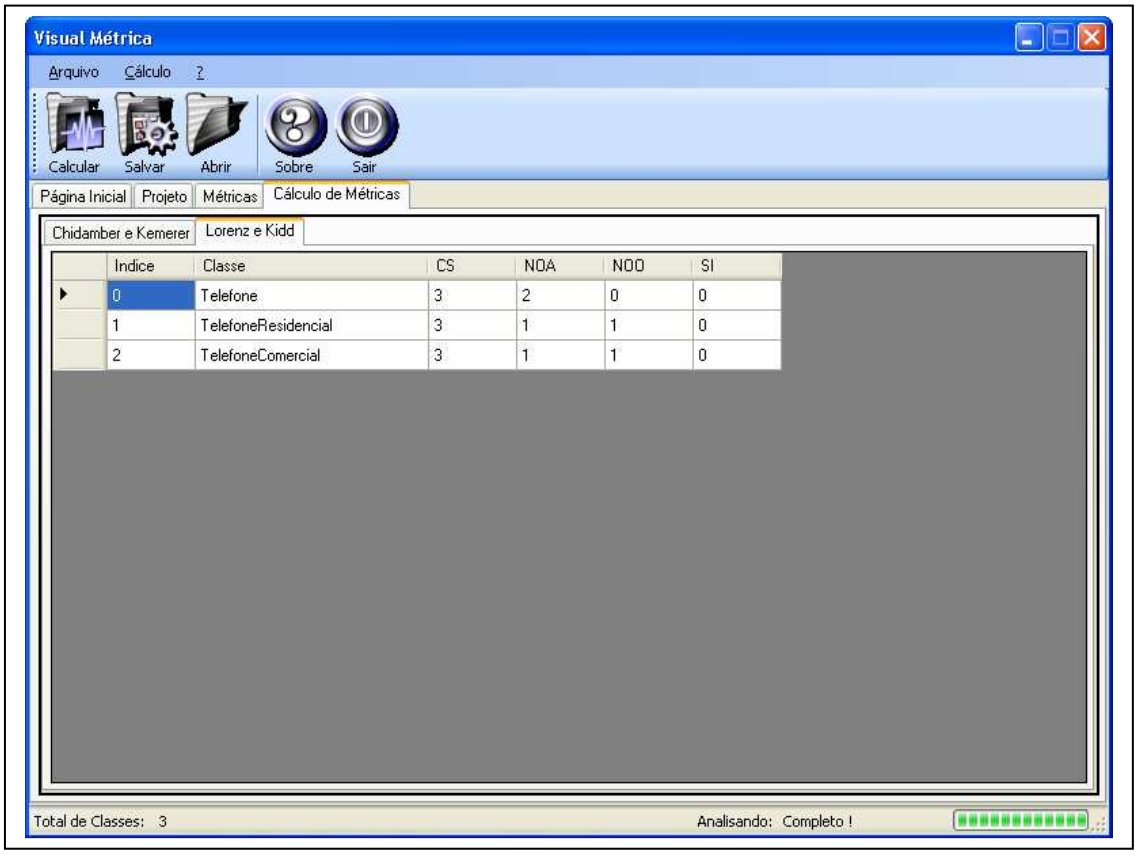


Figura 41 – Resultado do cálculo segundo Lorenz e Kidd Java

O protótipo se dispõe do recurso de ajuda, podendo ser acessado pelo *menu* do sistema Figura 42 ou pela tecla de atalho “F1”.

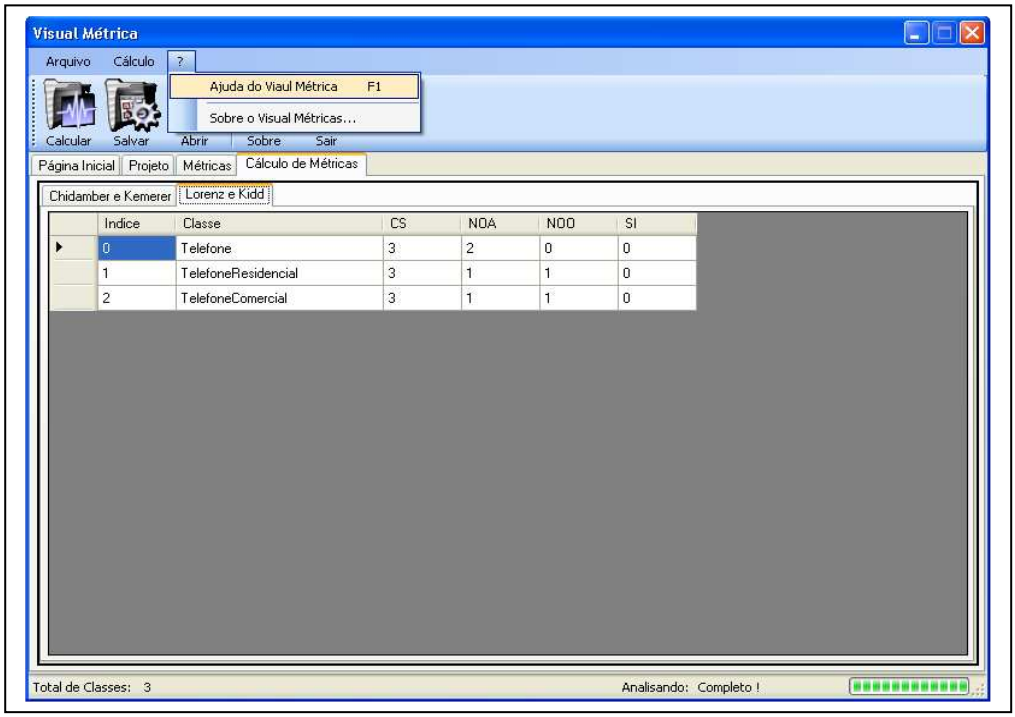


Figura 42 – Opção de ajuda do protótipo

A ajuda do sistema é disponibilizada no formato HTML e tem um *menu* com as opções no lado esquerdo do vídeo e as informações do outro lado. A Figura 43 exemplifica a ajuda do sistema.

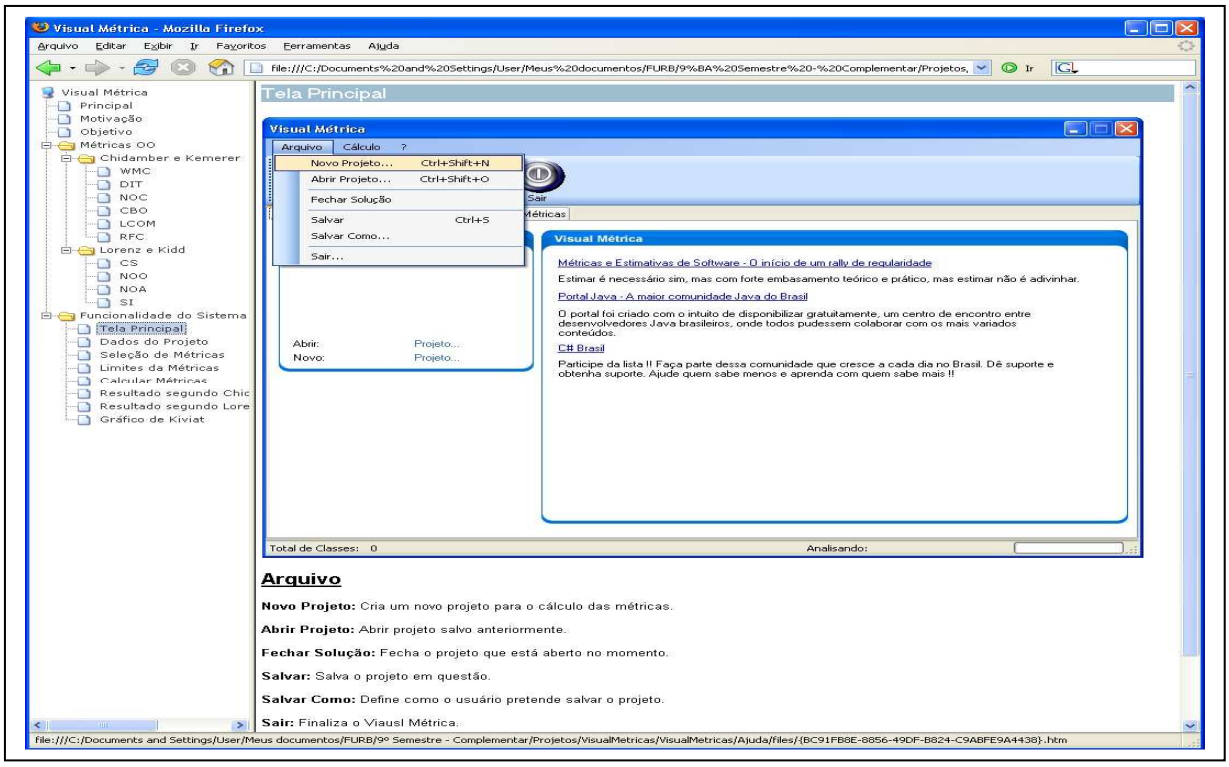


Figura 43 – Tela de ajuda do sistema

4.5 RESULTADOS E DISCUSSÕES

O Quadro 28 é um comparativo entre quatro ferramentas de coleta de métricas em software OO, e tem o objetivo de demonstrar algumas das métricas calculadas por essas ferramentas, linguagens suportadas e demonstrativo de resultados em forma de gráfico.

Ferramenta	**Métrica**										**Linguagem**			**Gráfico**
	WMC	DIT	NOC	CBO	LCOM	RFC	CS	NOO	NOA	SI	C#	Java	Delphi	
Visual Métrica	X	X	X	X	X	X	X	X	X	X	X	X		X
JMetric	X	X	X	X	X	X						X		X
Protótipo Seibt (2001)	X	X	X	X	X	X	X	X	X	X			X	
Protótipo Cardoso (1999)	X		X				X	X		X			X	

Quadro 28 – Comparativo entre as ferramentas

É visto neste comparativo que a ferramenta Visual métrica supera os resultados das demais ferramentas em número de linguagens e geração de gráfico que é apresentado por apenas uma das ferramentas além da Visual Métrica.

No entanto o Visual Métrica conta com o auxílio de uma ferramenta que de geração de analisador léxico e sintático o *Coco/R for C#*, o que não foi utilizado em Cardoso (1999) e Seibit (2001), tornando o desenvolvimento dos protótipos um trabalho bastante árduo.

O uso da ferramenta C# na implementação do trabalho se deu por sua portabilidade e pelo fato de ser uma das primeiras ferramentas de coleta de métricas escrita na linguagem C#.

5 CONCLUSÕES

O objetivo principal do trabalho, construir um software para coleta de métricas em software OO escritos em C# e Java foi atingido. Os resultados alcançados se diferenciam os obtidos em Cardoso (1999), Possamai (2000), Seibt (2001) e JMetric por terem sido calculadas métricas em linguagens diferentes dos trabalhos anteriores e com a geração de gráfico para a análise detalhada das classes. A ferramenta calcula dez métricas previstas por CK e LK.

A utilização da ferramenta *CocoR for C#* para a construção do analisador léxico e sintático foi importante para o trabalho, pois como foi uma ferramenta de fácil aprendizado, acelerou e simplificou o processo de desenvolvimento do analisador. As utilizações da ferramenta Microsoft Visual C# Express Edition e Enterprise Architect 4.5 também auxiliaram consideravelmente no desenvolvimento do trabalho.

Para a implementação da classe de coleta de métricas foram necessários estudos detalhados da estrutura das classes C# e Java. Isto foi possível graças à vasta bibliografia disponível.

A principal contribuição deste trabalho é o estudo das métricas OO, análise dos resultados com o gráfico proposto por Kiviat e a possibilidade da coleta das medidas em C# e Java. A partir das informações disponibilizadas neste trabalho, outras linguagens poderão ser analisadas.

5.1 EXTENSÕES

Como possíveis extensões para o trabalho, destacam-se:

a) disponibilizar o protótipo como *plug-in* para ferramentas CASE para realizar o cálculo a partir de diagramas;
b) adotar um número maior de métricas para um estudo mais detalhado do software analisado como LOC e COCOMO ;
c) ampliar a ferramenta para análise de outras linguagens OO como por exemplo Rubi, C++ e SmallTalk.

REFERÊNCIAS BIBLIOGRÁFICAS

AMBER, Scott W. **Análise e projeto orientado a objeto:** seu guia para desenvolver sistemas robustos com tecnologia de objetos. Tradução Oswaldo Zanelli. Rio de Janeiro: Infobook, 1998.

ARTHUR, Lowell J. **Melhorando a qualidade de software:** um guia para o TQM. Rio de Janeiro. Infobook, 1994.

CARDOSO, Eduardo J. **Métricas para programação orientada a objetos.** 1999. 45 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Universidade Regional de Blumenau, Blumenau.

CORDEIRO, Marco A. **Métricas de software.** Curitiba, 2000. Disponível em: <http://www.pr.gov.br/batebyte/edicoes/2000/bb101/metricas.htm>. Acesso em: 12 nov. 2005.

CÔRTES, Mario L.; CHIOSSI, Thelma C. S. **Modelos de qualidade de software.** Campinas: Editora da UNICAMP, 2001.

DEMARCO, Tom. **Controle de projetos de software:** gerenciamento, avaliação, estimativa. Rio de Janeiro: Campus, 1989.

FUNCK, Mônica Andréa. **Estudo e aplicação das métricas da qualidade do processo de desenvolvimento de aplicações em banco de dados.** 1995. 104 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

GUSTAFSON, David A. **Teoria e problema de engenharia de software.** São Paulo: Bookman, 2003.

HÜBNER Jomi F.; HUGO Marcel. **Prática de laboratório – lista 4.** Blumenau, 2003. Disponível em: < http://www.inf.furb.br/~poo/listas/poo-praticaLab4.pdf>. Acesso em: 3 nov. 2006.

JACOBSON, Ivar et al. **Object oriented software engineering:** a use case driven approach Wokingham: Addison Wesley, 1992.

KOSCIANSKI, Andre; SOARES, Michel dos S. **Qualidade de software:**Aprenda as metodologias e técnicas mais modernas para o desenvolvimento de software. São Paulo: Novatec, 2006.

LORENZ, Mark; KIDD, Jeff. **Object-oriented software metrics:** a practical guide. New Jersey: PTR Prenticel Hall, 1994.

MOLLER, Kurt. H., PAULISH, Daniel J. **Software metrics**: a practitioneris guide to improved product development. Los Alamitos: IEEE, 1993.

POSSAMAI, Roque César. **Ferramenta de análise de estruturas básicas em linguagem Pascal para o fornecimento de métricas.** 2000. 71 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

PRESSMAN, Roger S. **Engenharia de software.** São Paulo: Makron Books, 1995.

ROCHA, Ana R.; MOLDONADO, José C.; WEBER, Kival C. **Qualidade de software:** teoria e prática. São Paulo: Prentice Hall, 2001.

ROSENBERG, Linda. **Applying and interpreting object oriented metrics.** Utah, abr. 1998. Disponível em: <http://satc.gsfc.nasa.gov/support/STC_APR98/apply_oo apply_oo.html>. Acesso em: 07 jun. 2006

SEIBT, Patrícia R. R. S. **Ferramenta para cálculo de métricas em softwares orientados a objetos codificados em Delphi.** 2001. 86 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Universidade Regional de Blumenau, Blumenau.

SHEPPERD, Martin. **Foundation of software measurement.** New York: Prentice Hall, 1995.

TONINI, Antonio C. **Métricas de software.** [S.l.], 2004. Disponível em: <http://www.spin.org.br/Pdf/metricas%202.ppt>. Acesso em: 12 nov. 2005.

APÊNDICE A – GRAMÁTICA DA CLASSE PARA LINGUAGEM C# E JAVA

```
ClassDeclaration
= "class" ident ["extends" Type] ["implements" TypeList] ClassBody
.

ClassBody
= '{' {ClassBodyDeclaration} '}'
.

ClassBodyDeclaration
= ';'
| ["static"] ( Block
              | [ Modifier1 {Modifier} ] MemberDecl
              )
.

MemberDecl
= IF(identAndLPar()) ident ConstructorDeclaratorRest
| MethodOrFieldDecl
| "void" ident VoidMethodDeclaratorRest
| ClassDeclaration
| InterfaceDeclaration
.

MethodOrFieldDecl
= Type ident MethodOrFieldRest
.

MethodOrFieldRest
= VariableDeclaratorsRest ';'
| MethodDeclaratorRest
.

VariableDeclaratorsRest
= VariableDeclaratorRest {',' VariableDeclarator}
.

ArrayInitializer
= '{' [VariableInitializer { IF(commaAndNoRBrace()) ',' VariableInitializer }] [','] '}'
.

MethodDeclaratorRest
= FormalParameters BracketsOpt ["throws" QualidentList] (Block | ';')
.

VoidMethodDeclaratorRest
= FormalParameters ["throws" QualidentList] (Block | ';')
.

ConstructorDeclaratorRest
= FormalParameters ["throws" QualidentList] Block
.

FormalParameters
= '(' [FormalParameter {',' FormalParameter}] ')'
.
```

Quadro 29 – Gramática da classe para linguagem Java

```
NamespaceMemberDeclaration                                  (. Modifiers m = new Modifiers(this); .)
=
  "namespace" ident { "." ident }
  "{" { IF (IsExternAliasDirective()) ExternAliasDirective } { UsingDirective } { NamespaceMemberDeclaration } "}" [ ";" ]
  | { Attributes } ModifierList<m> TypeDeclaration<m>
.

TypeDeclaration<Modifiers m>                                (. TypeKind dummy; .)
=
  ( [ "partial" ]
    (                                                       (. m.Check(Modifier.classes); .)
      "class" ident [ TypeParameterList ] [ ClassBase ]
      { TypeParameterConstraintsClause } ClassBody [ ";" ]
    |                                                       (. m.Check(Modifier.nonClassTypes); .)
      "struct" ident [ TypeParameterList ]
      [ ":" TypeName { "," TypeName } ]
      { TypeParameterConstraintsClause } StructBody [ ";" ]
    |                                                       (. m.Check(Modifier.nonClassTypes); .)
      "interface" ident [ TypeParameterList ]
      [ ":" TypeName { "," TypeName } ]
      { TypeParameterConstraintsClause }
      "{" { InterfaceMemberDeclaration } "}" [ ";" ]
    )
  |                                                         (. m.Check(Modifier.nonClassTypes); .)
    "enum" ident [ ":" IntegralType ] EnumBody [ ";" ]
  |                                                         (. m.Check(Modifier.nonClassTypes); .)
    "delegate" Type<out dummy, true> ident [ TypeParameterList ]
    "(" [ FormalParameterList ] ")"
    { TypeParameterConstraintsClause } ";"
  )
.

ClassBase
=
  ":" ClassType { "," TypeName }
.

ClassBody
=
  "{" { { Attributes }                                                  (. Modifiers m = new Modifiers(this); .)
        ModifierList<m>
        ClassMemberDeclaration<m>
      }
  "}"
```

Quadro 30 – Gramática da classe para linguagem C#