

UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA
COMPUTAÇÃO

Wannessa Rocha da Fonseca

**FERRAMENTA DE EXTRAÇÃO DE MÉTRICAS
PARA APOIO À AVALIAÇÃO DE
ESPECIFICAÇÕES ORIENTADAS A OBJETOS**

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos
requisitos para a obtenção do grau de Mestre em Ciência da Computação

Ricardo Pereira e Silva
Orientador

Florianópolis, dezembro de 2002

FERRAMENTA DE EXTRAÇÃO DE MÉTRICAS PARA APOIO À AVALIAÇÃO DE ESPECIFICAÇÕES ORIENTADAS A OBJETOS

Wannessa Rocha da Fonseca

Esta Dissertação foi julgada adequada para obtenção do título de Mestre em Ciência da Computação Área de Concentração e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

Prof. Fernando A. O. Gauthier, Dr.
(Coordenador)

Banca Examinadora

Ricardo Pereira e Silva, Dr.
(orientador)

Guilherme H. Travassos, Dr.

Frank Augusto Siqueira, Dr.

Patrícia Vilain, Dra.

Vitório Bruno Mazzola, Dr.

AGRADECIMENTOS

Ao meu orientador, Professor Ricardo Pereira e Silva, por não poupar esforços na transmissão de conhecimento e pela seriedade como conduziu este trabalho.

A meu amado marido, Evandro, pelo incentivo e pelo exemplo de companheirismo durante todos os momentos do mestrado.

A minha família, em especial a minha querida mãe Elza, pela sua dedicação e compreensão nos momentos em que estive ausente. Deixo aqui registrada a minha eterna gratidão ao meu pai, Zilton, que estava presente fisicamente no início deste trabalho.

Ao CEPROMAT (Centro de Processamento de Dados do Estado de Mato Grosso) pelo apoio recebido.

À UNIRONDON, por ter oferecido este mestrado no Estado de Mato Grosso.

À Universidade Federal de Santa Catarina – em particular ao Departamento de Informática e Estatística por ter viabilizado esse mestrado.

SUMÁRIO

LISTA DE FIGURAS	VI
LISTA DE TABELAS	VII
LISTA DE GRÁFICOS.....	VIII
RESUMO	IX
ABSTRACT	X
1 INTRODUÇÃO.....	11
2 QUALIDADE	15
2.1 INTRODUÇÃO	15
2.2 QUALIDADE DE PROCESSO DE DESENVOLVIMENTO DE SOFTWARE	16
2.3 QUALIDADE DO PRODUTO SOFTWARE	17
2.4 CONCLUSÃO	25
3 AMBIENTE DE DESENVOLVIMENTO DE SOFTWARE SEA	27
3.1 INTRODUÇÃO	27
3.2 ESTRUTURA DO AMBIENTE SEA	27
3.3 TÉCNICAS DE MODELAGEM DO AMBIENTE SEA PARA ESPECIFICAÇÕES OO ...	29
3.4 CONCLUSÃO	35
4 MÉTRICAS PARA SOFTWARE ORIENTADO A OBJETOS.....	36
4.1 INTRODUÇÃO	36
4.2 MÉTRICA DE ACOPLAMENTO	41
4.3 MÉTRICAS DE ENCAPSULAMENTO	51

4.4	MÉTRICAS DE COMPLEXIDADE	55
4.5	MÉTRICA DE COESÃO	67
4.6	MÉTRICAS DE POLIMORFISMO	70
4.7	MÉTRICAS VERSUS ELEMENTOS DE ESPECIFICAÇÃO	77
4.8	CONCLUSÃO	82
5	FERRAMENTA DE EXTRAÇÃO DE MÉTRICAS	83
5.1	INTRODUÇÃO	83
5.2	TRABALHOS CORRELATOS	83
5.3	FERRAMENTA DE EXTRAÇÃO DE MÉTRICAS	89
5.4	FERRAMENTA DE VISUALIZAÇÃO DOS RESULTADOS DAS MÉTRICAS	99
5.5	CONCLUSÃO	104
6	ESTUDO DE CASO	106
6.1	INTRODUÇÃO	106
6.2	OBTENÇÃO E ANÁLISE DOS RESULTADOS	107
6.3	RESUMO	128
6.4	CONCLUSÃO	129
7	CONCLUSÃO	131
7.1	RESULTADOS OBTIDOS	132
7.2	LIMITAÇÕES	132
7.3	TRABALHOS FUTUROS	133
7.4	CONSIDERAÇÕES FINAIS	133
BIBLIOGRAFIA	135	
ANEXO 1 – EXEMPLO DE CÁLCULO DAS MÉTRICAS	139	
ANEXO 2 – EXEMPLOS DE DIAGRAMAS DE SEQÜÊNCIA DA FERRAMENTA MET	175	
ANEXO 3 – FORMATO DO ARQUIVO DE EXPORTAÇÃO DE DADOS .	183	

LISTA DE FIGURAS

FIGURA 3.1 - Estrutura do ambiente SEA [SIL 00]	28
FIGURA 3.2 - Exemplo de Diagrama de Caso de Uso.....	29
FIGURA 3.3 - Exemplo de Diagrama de Atividade	30
FIGURA 3.4 - Exemplo de Diagrama de Classe	31
FIGURA 3.5 - Exemplo de Diagrama de Seqüência	31
FIGURA 3.6 - Exemplo de Diagrama de Transição de Estados	32
FIGURA 3.7 - Exemplo de Diagrama de Corpo de Método.....	33
FIGURA 4.1 – Exemplo de processo ideal de aplicação de métricas	37
FIGURA 4.2 – Eliminando hierarquias largas [DIA 97]	59
FIGURA 4.3 – Eliminando hierarquias profundas [DIA 97].....	61
FIGURA 5.1 - Estrutura do ambiente SEA com a Ferramenta MET.....	90
FIGURA 5.2 – Ferramenta de Extração de Métricas de Especificações Orientadas a Objetos (MET).....	91
FIGURA 5.3 – Diagrama de Caso de Uso – MET.....	93
FIGURA 5.4 – Diagrama de Classes – MET.....	94
FIGURA 5.5 – Interface da ferramenta de extração das métricas.....	97
FIGURA 5.6 – Exemplo - saída de dados formato relatório.....	98
FIGURA 5.7 – Interface da tela de parâmetro	99
FIGURA 5.8– Diagrama de Classes da Ferramenta de Visualização Gráfica	100
FIGURA 5.9 – Exemplo de Visão hierárquica	101
FIGURA 5.10 – Exemplo de Tabela	103
FIGURA 5.11 – Exemplo de Gráfico	104
FIGURA 6.1 - Visão geral - <i>framework</i>	107

LISTA DE TABELAS

TABELA 4.1 – Classificação das métricas.....	40
TABELA 4.2 – Métricas versus Elementos e Relacionamentos.....	81
TABELA 5.1 – Ferramentas de Suporte a Avaliação de Software OO.....	86
TABELA 5.2 – Métricas versus Ferramentas de Extração de Métricas.....	89

LISTA DE GRÁFICOS

GRÁFICO 6.1 - Visão geral das classes - <i>framework</i>	108
GRÁFICO 6.2 - Visão geral dos métodos - <i>framework</i>	109
GRÁFICO 6.3 - Métrica: MHF - <i>framework</i>	110
GRÁFICO 6.4 - Métrica: Tamanho dos métodos - <i>framework</i>	110
GRÁFICO 6.5 - Métrica: Número de argumentos nos métodos - <i>framework</i>	111
GRÁFICO 6.6 - Métricas: Número de métodos disponíveis nas classes – <i>framework</i>	111
GRÁFICO 6.7 - Métrica: Número de métodos definidos nas classes - <i>framework</i>	111
GRÁFICO 6.8 - Métrica: Métodos complexos por classe - <i>framework</i>	112
GRÁFICO 6.9 - Métrica: Referência a SubClasses – <i>framework</i>	112
GRÁFICO 6.10 - Métrica: OVO - <i>framework</i>	113
GRÁFICO 6.11 - Métrica: NIP - <i>framework</i>	114
GRÁFICO 6.12 - Métrica: NOC - <i>framework</i>	114
GRÁFICO 6.13 - Métrica: DIT - <i>framework</i>	114
GRÁFICO 6.14 - Métrica: MIF - <i>framework</i>	115
GRÁFICO 6.15 - Métrica: AIF - <i>framework</i>	115
GRÁFICO 6.16 - Métrica: RFC – <i>framework</i>	116
GRÁFICO 6.17 - Métrica: CF - <i>framework</i>	117
GRÁFICO 6.18 - Métrica: CBO - <i>framework</i>	117
GRÁFICO 6.19 - Métrica: ACAIC - <i>framework</i>	118
GRÁFICO 6.20 - Métrica: DCAEC - <i>framework</i>	118
GRÁFICO 6.21 - Métrica: OCAIC - <i>framework</i>	119
GRÁFICO 6.22 - Métrica: OCAEC - <i>framework</i>	119
GRÁFICO 6.23 - Métrica: ACMIC - <i>framework</i>	120
GRÁFICO 6.24 - Métrica: DCMEC - <i>framework</i>	120
GRÁFICO 6.25 - Métrica: OCMIC - <i>framework</i>	121
GRÁFICO 6.26 - Métrica: OCMEC - <i>framework</i>	121
GRÁFICO 6.27 - Métrica: AMMIC - <i>framework</i>	122
GRÁFICO 6.28 - Métrica: DMMEC - <i>framework</i>	122
GRÁFICO 6.29 - Métrica: OMMIC - <i>framework</i>	123
GRÁFICO 6.30 - Métrica: OMMEC - <i>framework</i>	123
GRÁFICO 6.31 - Métrica: PF - <i>framework</i>	124
GRÁFICO 6.32 - Métrica: DPA - <i>framework</i>	124
GRÁFICO 6.33 - Métrica: DPD - <i>framework</i>	125
GRÁFICO 6.34 - Métrica: DP - <i>framework</i>	125
GRÁFICO 6.35 - Métrica: SPA - <i>framework</i>	126
GRÁFICO 6.36 - Métrica: SPD - <i>framework</i>	126
GRÁFICO 6.37 - Métrica: SP - <i>framework</i>	127
GRÁFICO 6.38 - Métrica: LCOM - modificado – <i>framework</i>	127

RESUMO

O paradigma da orientação a objetos vem sendo adotado em função da expectativa de produzir software mais robusto, com maior flexibilidade e com alto grau de reutilização. No entanto, a adoção do paradigma da orientação a objetos não garante que os sistemas sejam desenvolvidos com qualidade e com todos os benefícios que a orientação a objetos pode proporcionar.

O presente trabalho apresenta um levantamento das métricas voltadas a software orientado a objetos e que podem apoiar a avaliação de qualidade de especificações de artefatos de software orientados a objetos. Também apresenta um protótipo de ferramenta desenvolvido que automatiza a extração dessas métricas a partir de especificações orientadas a objetos. A ferramenta foi inserida no ambiente de desenvolvimento de software SEA, que suporta o desenvolvimento de especificações de projeto orientadas a objetos. As métricas fornecidas pela ferramenta podem ser utilizadas para apoiar o desenvolvedor de software, na avaliação do quanto uma especificação em construção adere aos preceitos da orientação a objeto, ainda na fase de projeto.

Palavras-chave: Métricas para projeto orientado a objetos, ferramenta de extração de métricas, orientação a objetos.

ABSTRACT

The object-oriented paradigm, attract widespread interest because it represents an important way to produce more robust and flexible software, increasing the reuse of software artifacts. The adoption of the object-oriented paradigm does not guarantee the quality of developed systems and all the benefits that the object-oriented can provide.

The present work presents a metrics survey of object-oriented software and can support the quality evaluation of object-oriented software artifacts specification. Also presents a tool that automatizes the extraction of these metrics in a given oriented-object specification. The tool was inserted in SEA environment which supports the development of objects-oriented project specifications. The metrics provided by the tool can be applied to support the software developer in the evaluation of how much the specification being construed fits to the principle of the object-oriented in the project phase.

Keywords: Metrics for object-oriented design, metrics extraction tools, object-oriented.

1 INTRODUÇÃO

O software tornou-se o principal elemento da evolução dos sistemas e produtos baseados em computador. Por outro lado, tornou-se um fator limitante na evolução de sistemas de computador, pois logo a cultura e a história da “programação” criou um conjunto de problemas que persistem até hoje. Problemas como, maximizar a produtividade, realizar estimativas de custo e prazo mais precisos e garantir que todos os erros do software sejam eliminados, fazendo com que a qualidade do software seja suspeita [PRE 02].

Para estudar esses problemas e propor soluções, foi criada uma área específica dentro da Ciência da Computação denominada Engenharia de Software, que integra métodos, ferramentas e procedimentos para o desenvolvimento de software, com o objetivo de aumentar a produtividade, reduzir o custo, fazer estimativas mais precisas e produzir software com mais qualidade.

Um processo de desenvolvimento de software tem como objetivo a obtenção de produtos com maior qualidade. Visa como resultado do esforço produtivo não apenas código, mas também outros produtos de trabalho como a especificação de projeto, planos de teste e outros. A avaliação de qualidade de software, nesse contexto, não pode se ater exclusivamente ao código, mas à totalidade dos produtos desenvolvidos. Quanto mais tarde um defeito for detectado, mais dispendiosa será sua correção. Por exemplo, um defeito não detectado na fase de análise, se for detectado somente na fase de teste, terá um alto custo, pois acarretará correções na fase de análise, projeto e implementação. Um caminho para identificar defeitos é a avaliação contínua dos produtos gerados ao longo do desenvolvimento.

A dificuldade em desenvolver sistemas com qualidade cresceu em função do aumento da complexidade dos sistemas. Avaliar se critérios de qualidade, como reutilização e flexibilidade, estão sendo corretamente explorados é uma tarefa muito

difícil de ser realizada manualmente. Booch enfatiza a limitação da capacidade humana em lidar com sistemas de software de alta complexidade ([BOO 94]¹ Apud [DIA 97]): *A complexidade dos sistemas de software que temos que desenvolver está crescendo, e ainda temos limitações básicas na nossa habilidade para lidar com esta complexidade. Como podemos resolver esse problema?*.

A utilização do paradigma da orientação a objetos fornece mecanismos adequados para que sistemas grandes e complexos sejam flexíveis a mudanças, com arquitetura de software modular e reutilizável, aumentando a produtividade, reduzindo o custo de desenvolvimento e esforço na manutenção. Porém a orientação a objetos por si só não garante esse conjunto de características, como também não garante que sistemas sejam desenvolvidos com qualidade.

Com o advento do paradigma orientado a objetos, iniciaram-se estudos de métricas destinadas especificamente para software orientado a objetos, avaliando características específicas, como herança e polimorfismo. Como resultado desses estudos, têm-se obtido propostas de métricas que podem auxiliar a monitorar a qualidade do software, a serem aplicadas durante as fases de desenvolvimento de software, mais comumente aplicadas sobre o código fonte produzido. Essas métricas geralmente quantificam características da especificação orientada a objetos e em alguns casos são associados significados qualitativos aos valores quantitativos extraídos. Existem vários trabalhos na literatura que propõe essas métricas, outros que contestam o uso das métricas e outros trabalhos que demonstram preocupação quanta validação das métricas, pois não basta que estas sejam propostas.

Pressman [PRE 02] descreve sobre a importância da utilização de métricas:

Métricas de software fornecem uma maneira quantitativa de avaliar a qualidade de atributos internos do produto, habilitando assim o engenheiro de software a avaliar a qualidade antes do produto ser construído. As métricas fornecem a visão aprofundada, necessária

¹ Booch, G; “Object Oriented Analisys and Design with Applications (2nd ed)”, Benjamin / Cummings, Califórnia, 1994

para criar modelos efetivos de análise e projeto, código sólido, e testes rigorosos.

Nesta dissertação é apresentada uma coletânea de métricas técnicas específicas para software orientado a objetos. Discute-se também como essas métricas podem ser extraídas ao longo das etapas de análise e projeto, a partir de especificações produzidas em UML. E ainda a construção de uma ferramenta que automatiza a extração dessas métricas, ainda na fase de projeto. É apresentado um estudo de caso que mostra a aplicação e utilização das métricas e da ferramenta.

A obtenção automática de métricas produz subsídio ao longo das etapas de análise e projeto, com baixo esforço, capaz de apoiar o desenvolvedor em seu esforço de localizar construções que possam gerar algum risco para o projeto, como o uso exagerado de polimorfismo ou classes muito acopladas. Assim, é possível a extração de métricas indicadoras de como a especificação de projeto foi elaborada e o quanto a especificação está aderente aos preceitos da orientação a objeto. Cabe salientar que o papel da ferramenta de extração de métricas é o de fornecimento de subsídios para o processo de avaliação de qualidade e não o de avaliar a qualidade do software tratado.

A ferramenta desenvolvida neste trabalho está inserida no ambiente SEA, que proporciona a construção de especificações de artefatos de software, tais como *frameworks*, componentes, interfaces de componentes, padrões de projeto e aplicações. Esse ambiente foi projetado para possibilitar a inclusão de novas ferramentas de apoio ao processo de desenvolvimento da especificação.

O presente trabalho está organizado em 7 capítulos, cujos conteúdos estão descritos a seguir.

No Capítulo 2 são tratados aspectos de qualidade de software, qualidade de processo, fatores internos e externos da qualidade de software, qualidade especificamente de software orientado a objetos e discussão de como avaliar essa qualidade.

No capítulo 3 é feita uma breve apresentação do ambiente SEA, onde foi integrada a ferramenta de extração de métricas para software orientado a objetos.

O capítulo 4 descreve um conjunto de métricas que são usadas como meio para avaliar a qualidade de especificação de software orientado a objetos. Essas métricas

serviram como subsídio para implementar a ferramenta de extração de métricas - MET, descrita neste trabalho.

No capítulo 5 são apresentados exemplos de ferramentas de extração de métricas, incluindo a ferramenta proposta neste trabalho, descrevendo a arquitetura da ferramenta e sua interface.

No capítulo 6 é apresentado um estudo de caso que usa especificação de um *framework* de jogos de tabuleiro, desenvolvido por Silva [SIL 97].

No capítulo 7 é apresentada a conclusão deste trabalho.

No anexo 1 são apresentados exemplos de cálculo das métricas e como os resultados são apresentados pela ferramenta de extração de métricas, posposta neste trabalho.

No anexo 2 são apresentados exemplos de diagramas de seqüência da ferramenta MET.

No anexo 3 é apresentado o formato do arquivo de exportação de dados da ferramenta MET para a ferramenta de visualização dos resultados das métricas.

2 QUALIDADE

2.1 *Introdução*

Segundo a norma NBR ISO 8402 [NBR 93], qualidade é *a totalidade das características de uma entidade que lhe confere a capacidade de lhe satisfazer às necessidades explícitas e implícitas*.

A entidade é o produto em questão, que pode ser um bem ou um serviço. As necessidades explícitas são as próprias condições e objetivos propostos. As necessidades implícitas incluem as diferenças entre os usuários, a evolução no tempo, as implicações éticas, as questões de segurança e outras visões subjetivas.

No início do século passado a discussão sobre a qualidade estava focada nos produtos, onde era inspecionada apenas a qualidade do produto final. Na década de 40, pensou-se em avaliar a qualidade dos subprodutos gerados no processo. Por volta da década 50, teve início o controle do processo de produção. Na década 60 o enfoque passou a ser a educação das pessoas envolvidas na produção. Na década de 70, a otimização de cada processo de produção. A partir da década 80, a valorização da avaliação do próprio projeto de produção (etapa que ocorre antes de começar a produzir) [BAR 97].

Com o processo de desenvolvimento de software ocorreu algo semelhante. Até a década de 60 os programas eram relativamente simples e o enfoque era a codificação. O desenvolvimento era em pequena escala, dado o custo e as limitações de hardware. com a evolução do hardware, foi possível desenvolver programas cada vez mais complexos. No final da década de 60 foram verificados problemas relativos ao desenvolvimento e manutenção do software, caracterizando a crise do software, em função de problemas tais como: fracasso nas estimativas de custos e prazos para processo de desenvolvimento e manutenção de software, insatisfação do cliente com o produto final. Todos esses problemas ocorreram devido à cultura de desenvolvimento de software utilizada, porque na maioria das vezes software era produzido sem técnicas de

desenvolvimento. A solução foi adotar abordagens metodológicas, que estabelecessem passos e critérios bem definidos para desenvolvimento, manutenção, gerenciamento do processo de desenvolvimento, estimativas de custos e prazos e coordenação do trabalho em equipe [SIL 01].

Começou a surgir uma maior preocupação referente à produtividade dos desenvolvedores, qualidade dos produtos e aspectos de segurança de programas. Para Barreto [BAR 97], uma vez tendo garantido a qualidade no processo, o produto produzido não deixará de tê-la. A seguir será descrito o aspecto da qualidade do processo (processo de desenvolvimento de software) e do produto (o software).

2.2 Qualidade de Processo de Desenvolvimento de Software

Quando se fala em qualidade de software, geralmente vem à mente que o objeto de avaliação é o produto software, mas hoje é sabido a necessidade de avaliar o processo de desenvolvimento de software. Um software com qualidade, gerado com base em um processo de software definido e documentado, tem um maior valor para a empresa que o desenvolveu, pois é a garantia que outros softwares possam ter a mesma qualidade. Por outro lado, a qualidade de software desenvolvido sem seguir um processo de desenvolvimento documentado, pode não ser repetida, pois quando se tem um processo de desenvolvimento com qualidade, a tendência é que todos os softwares desenvolvidos seguindo o processo tenham a mesma qualidade.

A falta de qualidade e produtividade no processo de desenvolvimento está diretamente relacionada ao crescente aumento da complexidade e tamanho do software, assim como ao aumento de novas demandas, sempre com prazo ultrapassado e a falta de processos definidos e instituídos para o desenvolvimento de software que inclua um subprocesso de avaliação da qualidade.

Conforme Silva [SIL 01], a qualidade de processo se divide em:

Aspectos Tecnológicos – que se referem à abordagem de desenvolvimento do sistema, técnicas de modelagem, procedimento de análise e projetos, entre outros.

Aspectos Gerenciais – que se referem à coordenação dos trabalhos, avaliação e cumprimento de prazos, entre outros.

Para realizar a avaliação da qualidade dos processos têm-se como referência algumas normas, tais como: ISO 12207 (Software Life Cycle Process - Norma para a qualidade do processo de desenvolvimento de software) [WEB 99], NBR ISO 9001 (Sistemas de qualidade - Modelo para garantia de qualidade em Projeto, Desenvolvimento, Instalação e Assistência Técnica) [HUT 94], NBR ISO 9000-3 (Gestão de qualidade e garantia de qualidade. Aplicação da norma ISO 9000 para o processo de desenvolvimento de software) [ANT 95], NBR ISO 10011 (Auditoria de Sistemas de Qualidade) e CMM (*Capability Maturity Model*) [PAU 95] [PAU 01].

Em um processo de desenvolvimento de software são descritas quais as medições que serão aplicadas durante o desenvolvimento de um software. São métricas destinadas a garantir a qualidade do produto final (o software), dos produtos intermediários e do processo para identificar melhorias. De acordo com o CMM, por exemplo, devem ser estabelecidas medições durante todo o ciclo de desenvolvimento de software. Os produtos gerados durante todo o ciclo de vida são revisados e as medidas são usadas para quantificar as características do software.

2.3 Qualidade do Produto Software

Qualidade de software segundo Pressman [PRE 02] é:

Conformidade com requisitos funcionais e de desempenho explicitamente declarados, padrões de desenvolvimento explicitamente documentados e características implícitas, que são esperadas em todo software desenvolvido profissionalmente.

Um produto de software deve ser desenvolvido com a qualidade necessária e suficiente para o uso pré-estabelecido e não com o objetivo de alcançar a qualidade perfeita.

Avaliar um produto de software exige planejamento, controle e uso de técnicas de avaliação adequadas. O método SQUID descrito por [BOE 99] trata o uso de medições para planejar e controlar a qualidade do produto durante o desenvolvimento, e avaliar a qualidade do produto final.

Para realizar a avaliação da qualidade dos produtos de software tem-se como referência o conjunto de normas: ISO/IEC 9126, que apresenta um conjunto de

características da qualidade e métricas aplicáveis aos produtos de software, ISO/IEC 14598, que estabelece o processo de avaliação dos produtos de software, e ISO /IEC 12119, que estabelece os requisitos de qualidade de um pacote de softwares e instruções sobre como testar os software com relação a esses requisitos.

Os fatores de qualidade de software podem ser classificados em [SIL 01]:

Fatores internos de qualidade – não são percebidos pelos usuários do sistema e referem-se diretamente à forma como os softwares estão organizados, se estão modularizados, legíveis, etc. Fatores internos são meios para se alcançar os fatores externos.

Fatores externos de qualidade – esses fatores podem ser percebidos pelos usuários do sistema. São eles: correção, robustez, estendibilidade, reusabilidade, compatibilidade, eficiência, portabilidade, verificabilidade, integridade e facilidade de uso.

Neves [NEV 99] discute a identificação da falta de qualidade:

Embora seja difícil medir e definir um software como sendo de boa qualidade, é fácil identificar um software de má qualidade. Os erros freqüentes, o mau funcionamento, ou a inadequação aos seus requisitos são sempre notados.

Ao utilizar os fatores de qualidade externos para avaliar a qualidade de software, deve-se definir prioridades. Os requisitos de qualidade de software podem variar de acordo com o tipo de software, mudando assim a prioridade de cada fator de qualidade, descrita anteriormente.

Para se obter software com qualidade é imprescindível que seja garantida a qualidade dos produtos de cada fase do desenvolvimento de software. Desta forma, é necessário um acompanhamento sobre os processos realizados e produtos gerados. No acompanhamento é necessário realizar medidas para garantir a qualidade do produto ou processo. Algumas características de software (linhas de código, tempo de execução, número de erros na fase de teste) são facilmente medidas, porém, algumas medidas são subjetivas, como, por exemplo, medir a consistência dos dados de um sistema. Conforme Bergamo [BER 00]:

Existem poucas métricas de aceitação geral para as características de software. Grupos ou organizações de normalização podem estabelecer seus próprios modelos de processo de avaliação e métodos para a criação e validação de métricas relacionadas com as características. Além disso, faz-se necessário o estabelecimento de níveis de pontuação específicos para a organização ou para a aplicação.

A avaliação de um software pode ser realizada com base em fatores internos e externos de qualidade. Os fatores internos estão associados à boa organização do software e são bases para a garantia dos fatores externos. Um software bem organizado será mais fácil de ser entendido, exigindo menos esforço e custo de manutenção, assim como maior capacidade de reutilização e redução do número de erros.

2.3.1 Qualidade de Software Orientado a Objetos

Em função da busca constante de melhoria da qualidade de software, buscam-se também formas de avaliar um software produzido a partir do paradigma orientado a objetos. A avaliação pode ser feita sob a visão do usuário, (produto final) ou a avaliação pode ser feita sobre os produtos desenvolvidos durante o ciclo de desenvolvimentos de software e não somente o código. Porém, os critérios utilizados para avaliar os produtos “intermediários” do desenvolvimento de software orientado a objetos diferem dos produtos gerados utilizando o paradigma estruturado. Por outro lado, os fatores externos (usabilidade, confiabilidade, performance) aplicados para avaliar a qualidade de um software orientado a objetos são os mesmos utilizados para o software desenvolvido segundo o paradigma funcional.

O desenvolvimento de software orientado a objetos traz consigo uma possibilidade de melhoria da qualidade e produtividade no desenvolvimento de software, devido ao enfoque da orientação a objetos permitir modelar o problema em termos de objetos, diminuindo o *gap* semântico entre o mundo real e a sua abstração. E ainda as atividades de reutilização, manutenção e teste tendem a ser mais simples em função do não isolamento dos dados dos seus procedimentos [ROC 01]. É freqüentemente usado na tentativa de promover reuso de software, redução de tempo de desenvolvimento, diminuição de custo de manutenção e simplicidade de criação de novos sistemas. Isso é verdade, porém, a orientação a objetos por si só não garante todos

esses benefícios. É necessário o desenvolvimento de componentes de software que permitam flexibilidade, reutilização e adaptações com menor propagação de mudanças possíveis. Para isso é necessário o uso de técnicas que garantam determinadas características no projeto do software. A seguir serão descritas algumas técnicas e características da orientação a objetos que auxiliam na produção de software com os benefícios acima descritos.

Encapsulamento

Essa característica não é própria da orientação a objetos, porém, a capacidade de unir a estrutura de dados e o código que manipula essa estrutura, torna esse modelo de desenvolvimento extremamente poderoso, no que se refere à capacidade de isolamento de componentes de software. Com isso é possível produzir componente de software com capacidade de ser usado em diversas situações, sem provocar mudanças no contexto que está sendo inserido. Outra característica do encapsulamento é a facilidade de adaptação de um componente de software, sob dois aspectos:

- O primeiro diz respeito ao entendimento do componente de software. Como o componente possui os dados e todo o código que os manipulam, assim é mais fácil compreendê-lo.
- O segundo diz respeito à propagação de mudanças. Quanto maior o encapsulamento, menor será o impacto provocado pelas mudanças.

Protocolo

O protocolo de um objeto é definido pelo conjunto de mensagens que podem ser enviadas para ele. Os objetos com o mesmo protocolo podem ser trocados sem que haja necessidade de adequações, além de diminuir o número de mensagens diferentes que os desenvolvedores devem conhecer.

Polimorfismo

É um poderoso instrumento no desenvolvimento de componentes de software genéricos. Com o uso do polimorfismo é possível tratar instâncias de objetos de diferentes classes sem considerar a classe de cada objeto. Isso é possível, caso os objetos entendam um conjunto comum de mensagens. Um exemplo disso é uma lista heterogênea de objetos.

No caso do tratamento de uma lista de objetos heterogêneos é possível, antes de passar uma mensagem, verificar qual é o tipo do objeto (elemento da lista) e invocar o método (módulo) correto, porém, é comum esquecer um tipo de um elemento, ou ainda, quando for inserido um novo tipo de objeto na lista, acarretar mudanças nos trechos que tratam a lista. Assim, o polimorfismo além de reduzir o número de identificadores de mensagens, diminui também o impacto provocado pelas adequações do software, minimizando o tempo, custo e risco de inserções de erros nas adequações.

Por outro lado, o código que possui um alto grau de mensagens iguais, decididas em tempo de execução, dificulta seu entendimento, sendo necessária a avaliação das instâncias de objetos para conhecer a mensagem invocada.

Herança

É uma das características mais fortes do modelo orientado a objetos. Permite que uma nova classe seja produzida a partir da estrutura de dados e operações de outra classe, chamada superclasse. Dessa forma, não é preciso reescrever aquilo que já foi previsto em outra classe. Essa técnica é chamada de desenvolvimento por diferença, visto que é desenvolvido ou redefinido somente o que é diferente da superclasse. O maior benefício é que mantém a superclasse inalterada, não interferindo naquilo que foi desenvolvido anteriormente.

A herança promove uma certa classificação das classes de um sistema. Normalmente se espera que as classes relacionadas por herança sejam próximas conceitualmente. Isso será verdade se a herança não for usada somente com o objetivo de herança de código e sim como especialização de um conceito genérico em algo mais específico.

Outra característica da herança é a possibilidade do desenvolvimento de mensagens/protocolos padrões. Quando é produzida uma classe e nela são especificadas somente as assinaturas dos métodos (classes abstratas), forçará suas subclasses a implementarem esses métodos, definindo um conjunto comum de operações para um grupo de classes relacionadas por herança.

2.3.2 Subsídios para Avaliação de Especificação de Software OO

Na seção anterior foram apresentadas as principais características do modelo orientado a objetos. Nota-se que as três características, encapsulamento, herança e polimorfismo estão relacionadas. É necessário o apoio da herança para o desenvolvimento de componentes polimórficos. O encapsulamento apóia o desenvolvimento por diferença, que é o princípio da herança. Como o objetivo deste trabalho é buscar subsídios para avaliar especificações de software orientado a objetos, serão apresentados a seguir alguns trabalhos, que tratam esse assunto.

Uma das formas mais difundidas para avaliar os produtos de software consiste no uso de mensuração, pois conforme Tom de Marco *não podemos avaliar aquilo que não podemos medir*.

É sabido a importância de avaliar produtos gerados nas fases iniciais do desenvolvimento de software, pois possibilita que ações corretivas ou preventivas possam ser tomadas no sentido de reduzir esforços e custos no desenvolvimento do software como um todo. Vários estudos existem na tentativa de apresentar o que deve ser avaliado nas fases iniciais do projeto, pois, quanto mais tarde um erro for detectado, mais dispendiosa será sua correção. Por exemplo, uma falha não detectada na fase de análise, se for detectada somente na fase de teste, terá um alto custo, pois acarretará correções na fase de análise, projeto e implementação.

Para avaliar a qualidade de especificações de software orientado a objetos freqüentemente utilizam-se métricas. Contudo, as métricas utilizadas no desenvolvimento estruturado não são facilmente adaptadas para o modelo orientado a objetos, com exceção de algumas métricas que avaliam características de métodos de classe, que no modelo funcional avaliam módulos e que podem ser utilizadas nos dois paradigmas. Isso se deve principalmente aos conceitos particulares da orientação a objetos, tais como, classes, herança, polimorfismo, encapsulamento [CHI 93] [BAS 96] [ROS 94].

Relacionado à avaliação da qualidade de software orientado a objetos está a complexidade do software, pois o aumento da complexidade pode influenciar

diretamente a qualidade do produto. Com o objetivo de reduzir a complexidade de projetos orientados a objetos, o trabalho de Andrade [AND 98], propôs a aplicação simultânea de princípios, diretrizes e métricas.

A experiência em desenvolvimento orientado a objetos relatados na literatura, aponta diretrizes de qualidade aplicadas ao paradigma de orientação a objetos. A utilização dessas diretrizes é mais necessária à medida que o tamanho do sistema aumenta [ROC 01].

Para alcançar os benefícios que o paradigma da orientação a objetos pode promover é preciso seguir certos princípios e diretrizes de desenvolvimento. Dessa forma é preciso durante o desenvolvimento do software, verificar se estes estão sendo seguidos. Para verificação do atendimento de alguns princípios e diretrizes é possível utilizar métricas. Por exemplo, é sabido que um software deve ter um baixo acoplamento e existem algumas métricas destinadas a medir o grau de acoplamento em projeto orientado a objetos, ou seja, a métrica quantifica o uso de acoplamento, mas é preciso que o desenvolvedor faça a relação com o princípio ou diretriz.

Alguns exemplos de diretrizes e princípios que podem ser usados como referência no desenvolvimento de software orientado a objetos são descritos a seguir:

- Minimizar o acoplamento entre as operações que pertencem a objetos diferentes [ROC 01]. Com isso diminui-se a dependência entre esses objetos e consequentemente o impacto provocado pela propagação de efeitos colaterais, em decorrência a possíveis modificações. Existe uma certa discussão em relação à herança, que cria acoplamento entre a superclasse e a subclasse, porém, ajuda a abstração de dados [AND 98].
- Maximizar a coesão entre as operações que pertencem ao mesmo objeto [ROC 01]. Quanto mais coeso for um objeto maior é a aderência dessa abstração ao conceito descrito. Em contrapartida, quanto menor a coesão de operações em um objeto pode significar problemas na representação conceitual. Pode indicar que o objeto deva ser particionado, diminuindo as diferenças conceituais entre as operações do objeto.

- Maximizar a coesão entre objetos que pertencem à mesma estrutura hierárquica [ROC 01]. Quando a coesão for alta, indicará que a hierarquia de classes representa a descrição de um conceito genérico, no topo da hierarquia, e os subníveis descrevem refinamentos (especializações) desse conceito. Caso contrário, pode indicar que foi desenvolvida uma hierarquia de classes com o objetivo de reaproveitamento de código, reduzindo as possibilidades de reuso desses componentes.
- Minimizar as conexões entre classes [MEY 97]. Quanto maior for o volume de informações passadas entre pares de classe, por exemplo, o número de argumentos dos métodos, maior será a possibilidade de alto impacto em decorrência de possíveis modificações.
- Ocultar as informações das classes [MEY 97]. Quanto menos expostos são os detalhes de uma classe, menores serão os riscos de propagação de mudanças. Se for levado em consideração o princípio anterior e reduzir ao máximo o conjunto de métodos de uma classe, menor será o risco de uma alteração interna provocar alguma propagação de alteração nos clientes da classe.
- Uma classe deve ser aberta a extensões e fechada a alterações [MEY 97] ([MAR 96]² apud [AND 98]). Os níveis mais altos em uma hierarquia de classes devem ser constituídos por classes abstratas. Assim, quando existir a necessidade de inserir uma nova funcionalidade, será feito na forma de extensão, usando herança, e não alteração na superclasse.
- Classe de alto nível não pode depender de classe de baixo nível ([MAR 96] apud [AND 98]). Quando um software possui camadas de classes que tratam serviços diferentes em cada camada, não pode existir dependência entre classes de níveis diferentes. Para eliminar tais dependências é preciso definir classes que implementam a interface entre esses níveis.

² MARTIN, R., *The Open-Closed Principle, C⁺⁺ Report*, 1996

- Uma classe não deve depender de suas subclasses [ROC 01]. Numa hierarquia de classes relacionadas por herança, uma superclasse generaliza um conceito e uma subclasse especializa esse conceito. Assim, quando é projetada uma superclasse não deve haver a preocupação com algo que ainda não foi definido, no caso, os detalhes das possíveis subclasses, até por que, o projetista de uma classe pode não saber que ela será usada como superclasse. O inverso pode ocorrer, quando a superclasse define um conjunto de métodos abstratos, forçando as subclasses a implementar uma interface padrão.

Existem vários trabalhos na literatura que visam propor métricas, interpretar, questionar a sua utilidade, verificar a sua aplicabilidade. Esses trabalhos mostram a preocupação com a credibilidade científica das métricas propostas. Para que uma métrica tenha validade não depende somente da sua fundamentação teórica. É preciso que ela seja validada estatisticamente, através de um número representativo de estudos de caso que possibilitem uma análise estatística para obter interpretações mais qualitativas do que quantitativas. Vários trabalhos disponíveis na literatura que tratam de métricas são referenciados no capítulo de métricas. Porém, ainda não existem métricas precisas onde o seu resultado possa apresentar um valor que dê uma representação qualitativa do produto avaliado.

O uso de métricas para especificações orientadas a objetos é pouco explorado devido a fatores como, a forma como as métricas são propostas e, consequentemente, em função da dificuldade da interpretação das mesmas. Um outro fator que contribui bastante para a pouca utilização das métricas está relacionado à pouca disponibilidade de ferramentas que dêem suporte a um processo de metrificação.

2.4 Conclusão

A qualidade do processo é diretamente ligada à qualidade do produto. Com o intuito de gerar produtos com maior qualidade, hoje se busca avaliar a qualidade tanto do produto como do processo de desenvolvimento de software. A qualidade dos produtos software pode ser medida com base em características externas ou internas. Pelo fato das características internas não serem visíveis aos usuários, essa responsabilidade, na maioria das vezes, é da equipe de desenvolvimento.

Avaliar os fatores externos de qualidade do produto software independe do paradigma utilizado para o desenvolvimento do software, porém, quando se trata de avaliar fatores internos de qualidade o paradigma utilizado está diretamente relacionado a “*o que*” será avaliado.

Sabe-se ainda que não é preciso e nem aconselhável que somente o produto final, o “*software*”, seja avaliado. O processo de avaliação deve ser iniciado já nas primeiras fases do desenvolvimento do software, pois quanto mais cedo um problema (má decisão de projeto) for identificado, menos dispendiosa será sua alteração.

Sobre a qualidade de especificações orientadas a objetos, tem-se discutido muito, porém, não existem métricas precisas que possam ser utilizadas para avaliar a qualidade de especificações OO e sim métricas que quantificam características do paradigma orientado a objetos e que podem subsidiar um processo de avaliação.

3 AMBIENTE DE DESENVOLVIMENTO DE SOFTWARE SEA

3.1 *Introdução*

O ambiente SEA foi desenvolvido por Silva para desenvolvimento e uso de artefatos de software orientado a objetos. Esse ambiente foi construído como extensão da estrutura de classes do *framework* OCEAN, também desenvolvido por Silva [SIL 00].

No ambiente SEA é possível inserir novas ferramentas, que poderão ser chamadas durante a operação do ambiente. Podem ser inseridas ferramentas que executem tarefas auxiliares para outras ferramentas, como exemplo, as ferramentas de correção de erro, que são utilizadas por ferramentas de análise.

No decorrer deste capítulo são tratadas as principais características do ambiente SEA que são relevantes para este trabalho. Dessa forma será apresentada a estrutura do ambiente e o suporte de modelagem para especificações OO disponíveis no ambiente SEA.

3.2 *Estrutura do Ambiente SEA*

O ambiente SEA permite a especificação de artefatos de software orientado a objetos, a saber, aplicações orientadas a objetos, *frameworks* orientados a objetos e componentes.

Os protótipos do *framework* OCEAN e do ambiente SEA foram implementados em *Smalltalk*, utilizando o ambiente *VisualWorks*. Foi usado ainda o *framework*, *HotDraw* para o desenvolvimento destes protótipos, para o suporte ao desenvolvimento de editores gráficos. Na figura 3.1 é exibido um esquema gráfico da estrutura do ambiente SEA.

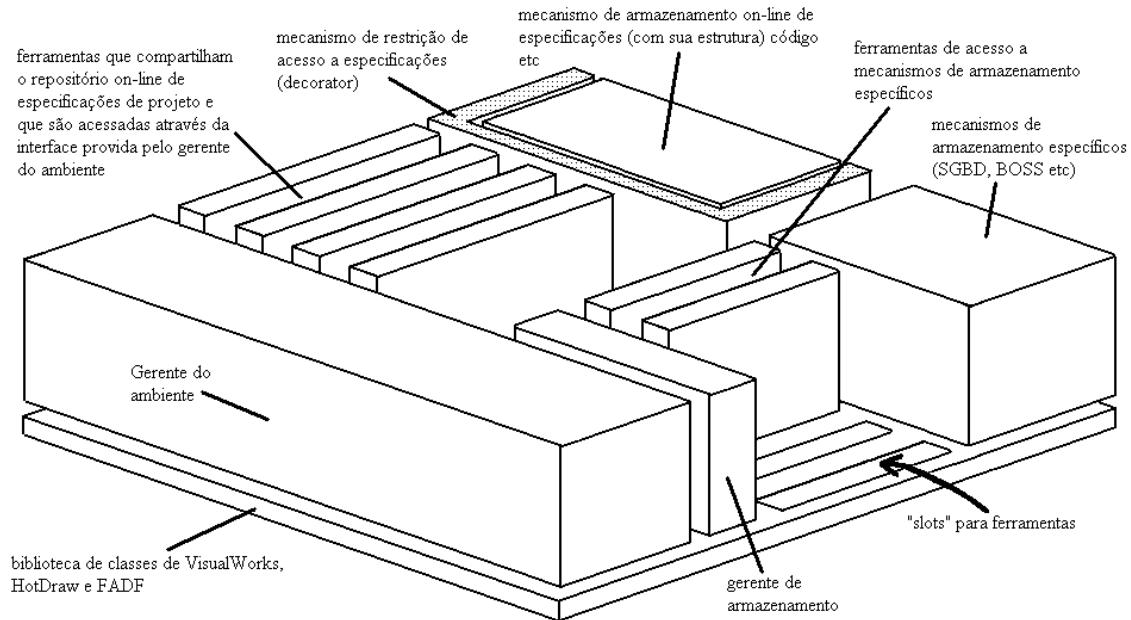


FIGURA 3.1 - Estrutura do ambiente SEA [SIL 00]

O ambiente possui um repositório de especificações, compartilhado por diversas ferramentas. As ferramentas neste ambiente podem ser classificadas como:

Ferramentas de edição: são ferramentas que podem ler e alterar uma especificação. Nesta categoria estão inseridos os editores de modelos e ferramentas de inserção de estrutura de padrões em especificações.

Ferramentas de análise: são ferramentas que podem ler uma especificação, mas não podem alterá-las, podendo aplicar avaliações quanto à consistência (como as disponíveis no ambiente) e extração de métricas de especificação orientadas a objetos (a implementação desta ferramenta é proposta neste trabalho).

Ferramentas de transformação: essas ferramentas fazem a ligação entre as especificações do ambiente e elementos externos. Como exemplo de ferramenta de transformação, tem-se a ferramenta para converter uma especificação em código *Smalltalk*.

3.3 Técnicas de Modelagem do Ambiente SEA para Especificações OO

No ambiente SEA são usadas cinco das oito técnicas de modelagem de UML e uma técnica adicional, proposta pelo autor do ambiente SEA, o Diagrama de Corpo de Método (DCM), para descrever o algoritmo dos métodos, o que não está previsto em UML. As técnicas de modelagem utilizadas são:

- Diagrama de Caso de Uso
- Diagrama de Atividades
- Diagrama de Classes
- Diagrama de Seqüência
- Diagrama de Transição de Estados
- Diagrama de Corpo de Método – não previsto em UML

A seguir são apresentados exemplos dos diagramas contemplados pela ferramenta:

Diagrama de Casos de Uso

O diagrama de caso de uso³ é composto por atores, casos de uso e relações entre atores e casos de uso. Exemplo de um diagrama de caso de uso é descrito na figura 3.2.

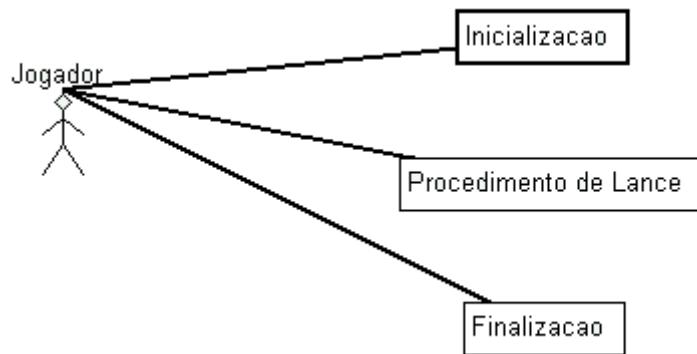


FIGURA 3.2 - Exemplo de Diagrama de Caso de Uso

³ Os diagramas (Caso de Uso, Atividades, Classes, Seqüência, Transição de Estados e Corpo de Métodos) foram modelados no Ambiente SEA.

Diagrama de Atividades

O autor do ambiente SEA propõe o uso do diagrama de atividades para estabelecer restrições de ordem de execução de casos de uso. O diagrama de atividades é composto por atividades (sendo que cada atividade corresponde a um caso de uso definido no diagrama de casos de uso) e transições entre as atividades. Deve ser eleito um caso de uso para ser a primeira situação de processamento, sendo que o processamento dos demais casos de uso ficam condicionados ao processamento do caso de uso inicial. Dessa forma, no ambiente SEA, é possível estabelecer restrições de seqüência de situações de processamento. Exemplo de diagrama de atividades descrito no ambiente SEA é apresentado na figura 3.3.

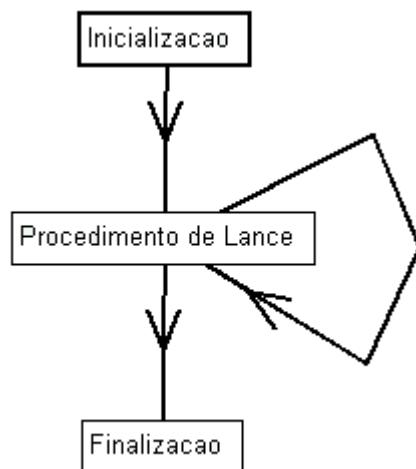
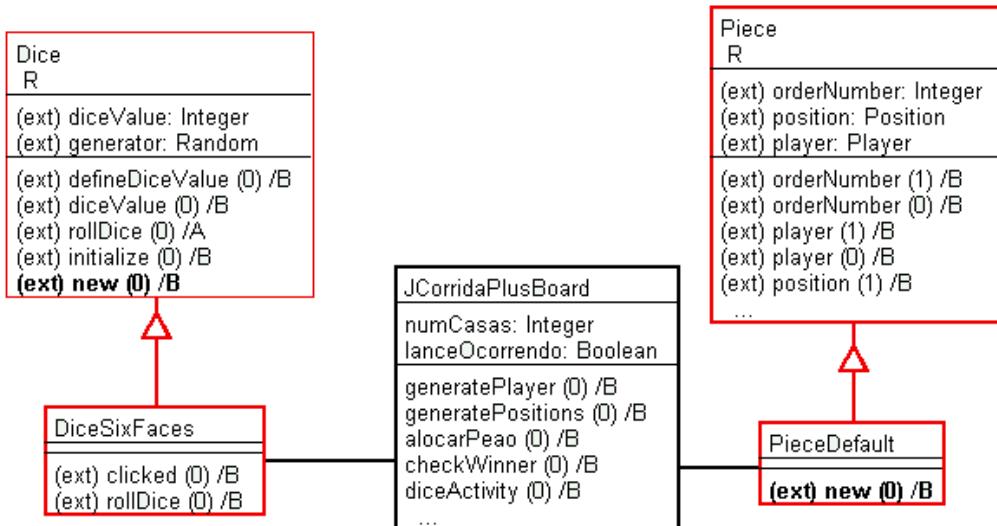


FIGURA 3.3 - Exemplo de Diagrama de Atividade

Diagrama de Classes

No diagrama de classe são representados os conceitos: classe, com seus atributos e métodos, associações binárias, agregação e herança. Esses conceitos podem ser classificados como externos – destacados em vermelho (conceitos criados fora da especificação). Exemplo de diagrama de classes, descrito no ambiente SEA é apresentado na figura 3.4.

FIGURA 3.4 - Exemplo de Diagrama de Classe⁴

Diagramas de Seqüência

O diagrama de seqüência é composto por mensagens e elementos que trocam as mensagens.

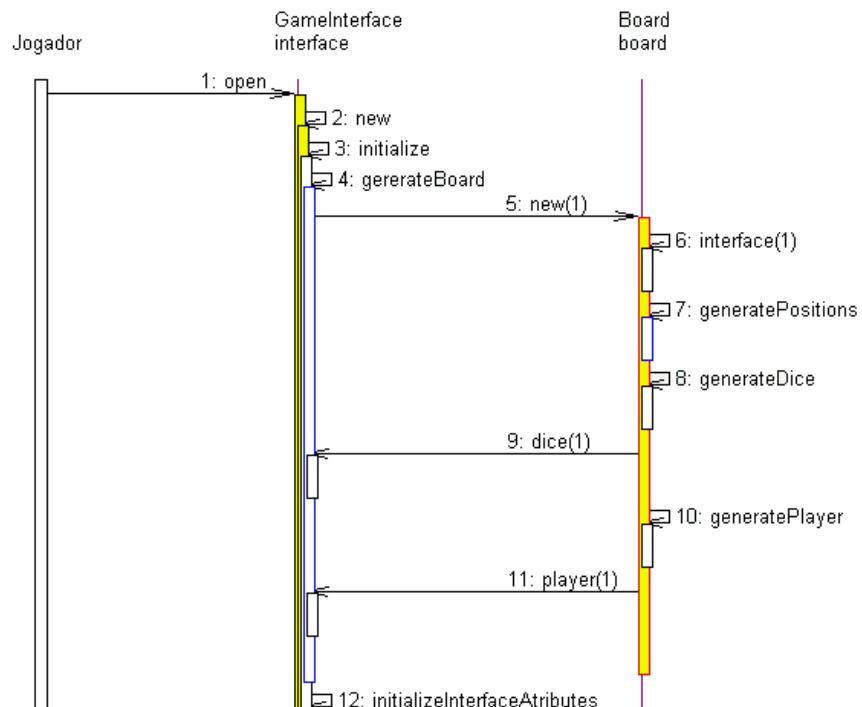


FIGURA 3.5 - Exemplo de Diagrama de Seqüência

⁴ No ambiente SEA foram feitas as seguintes convenções: Os métodos em negrito representam métodos de classe, os símbolos [L] presentes em alguns atributos de classes representam que o atributo é uma lista do tipo que foi definido, os números entre parênteses representam o comprimento da lista de parâmetros do método e a letra após parêntese representa a classificação do método em A- Abstrato, B – Base e T – Template.

Diagrama de Transição de Estados

O diagrama de transição de estados é associado a uma classe do diagrama de classes e o estado é definido em função de atributos e dos valores assumidos por estes atributos.

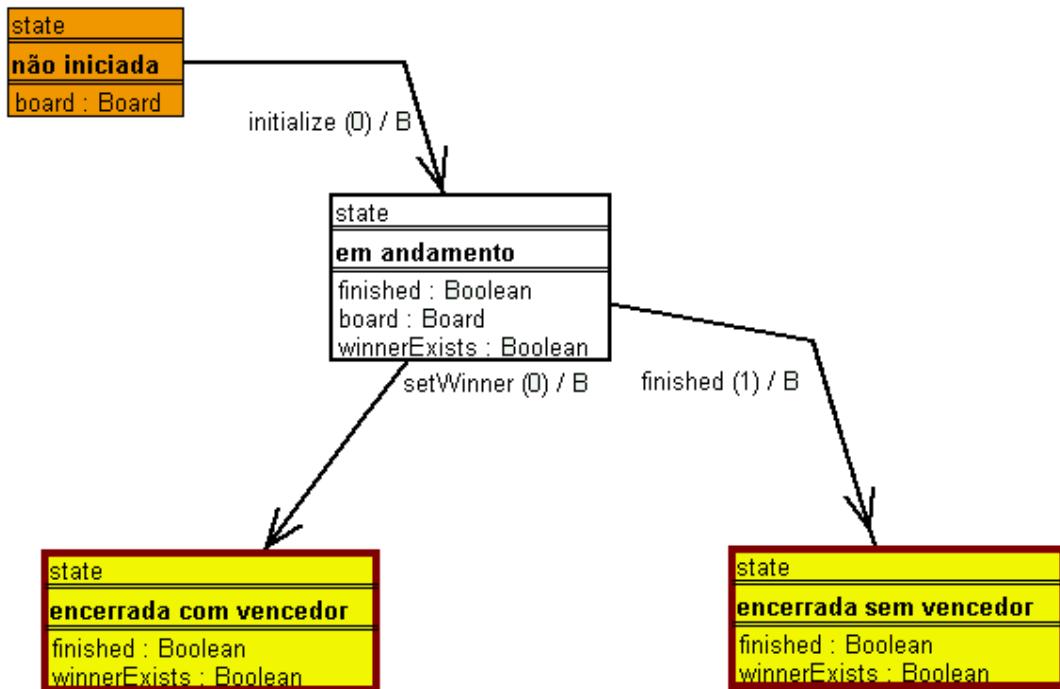


FIGURA 3.6 - Exemplo de Diagrama de Transição de Estados

Diagrama de Corpo de Método

O diagrama de corpo de método, que não é previsto pela UML, porém, disponibilizado no ambiente SEA, proporciona a especificação do algoritmo de cada método da classe, sendo o algoritmo expresso em uma linguagem simbólica, conversível para comandos equivalentes em diferentes linguagens de programação orientadas a objetos. Outra utilidade desse diagrama é na avaliação de referências feitas por uma classe a outras classes da especificação, sendo que essas referências podem ser através de variáveis locais do tipo da classe, parâmetros do método, valores associados a parâmetros, métodos de classes invocados e valores de retorno do método em questão.

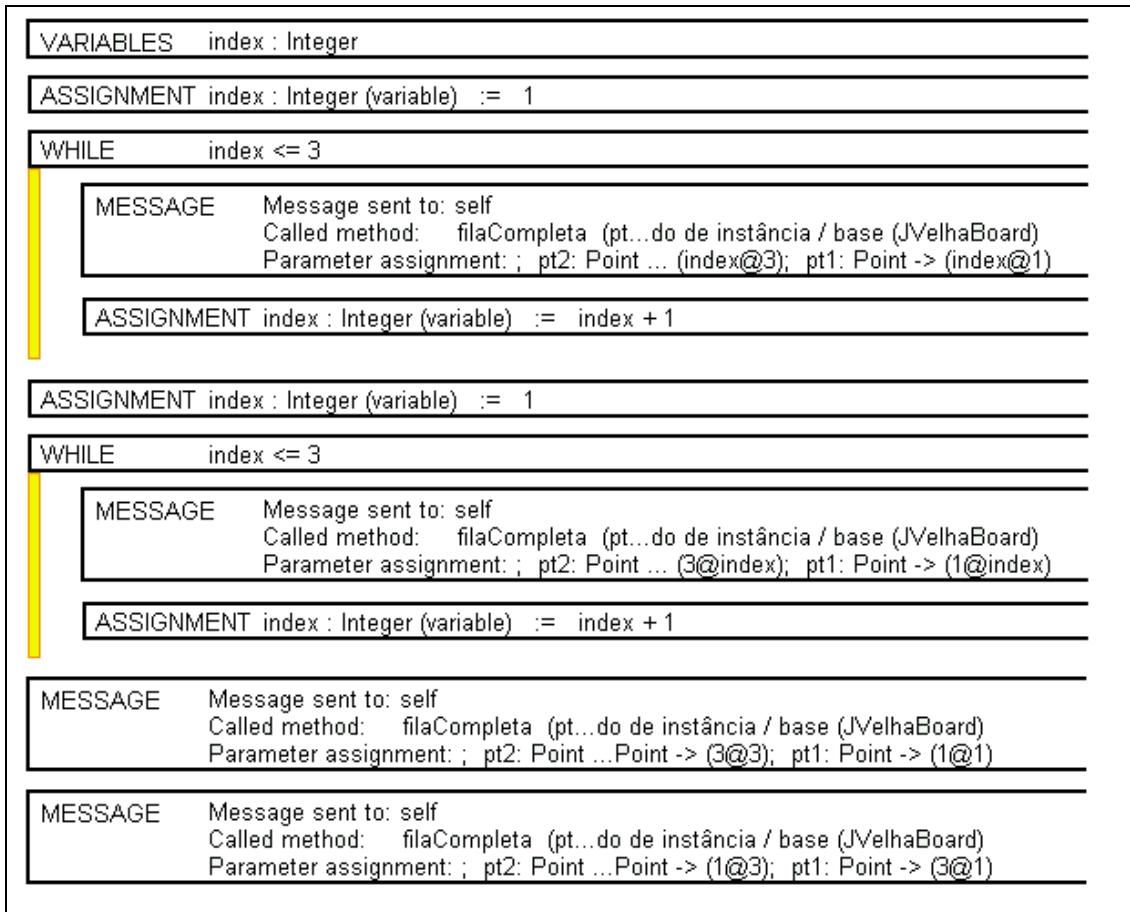


FIGURA 3.7 - Exemplo de Diagrama de Corpo de Método

O diagrama no corpo de métodos contém os seguintes comandos:

Variables - para declaração de variáveis temporárias;

Assignment - para atribuição de atributo, variável temporária, parâmetro, classes ou constante a um atributo da classe ou a uma variável temporária do método modelado;

Return - para retornar um dado;

Comment - para comentário;

If - statement que contém outros comandos para condicionar a execução destes a um predicado;

IfElse - semelhante ao *If*, porém com cláusula *Else*;

Message - comando de envio de mensagem a objeto;

Task - destinado para inclusão de string, que será copiado para o código fonte sem alteração. O objetivo do Task é inclusão de trecho de código gerado automaticamente por um suporte de desenvolvimento de software, como por exemplo, parte do software referente à interface gráfica gerada por ambientes de programação visual, como Delphi e VisualWorks [SIL 00].

While - *statement* destinado a descrever uma estrutura de repetição em que a condição é testada antes de cada execução,

Repeat - *statement* destinado a descrever uma estrutura de repetição em que a condição é testada após cada execução,

nTimes - *statement* destinado a descrever uma estrutura de repetição em que a quantidade de repetições é pré-estabelecida.

ExternalStateTransitionDiagram - equivalente a um comentário, gerado automaticamente pela ferramenta, para registrar a referência ao método descrito em diagramas de transição de estados.

ExternalSequenceDiagram - equivalente a um comentário, gerado automaticamente pela ferramenta, para registrar a referência ao método descrito em diagramas de seqüência.

Foram introduzidas no ambiente SEA algumas extensões para representar conceitos do domínio do *framework*, para agregar recursos⁵ de modelagem e para expressar a ligação semântica entre elementos de uma especificação e entre diferentes especificações.

Foram introduzidas também no ambiente as propriedades de redefinibilidade e essencialidade de classes na especificação de um *framework*. A propriedade de redefinibilidade permite registrar se uma classe de uma especificação de *framework* poderá ser ou não redefinida. Dessa forma, uma aplicação desenvolvida a partir do *framework*, poderá incluir subclasses somente para as classes do *framework* classificadas como redefiníveis. As classes redefiníveis podem ainda ser classificadas como essenciais. Uma classe definida como essencial caracteriza que todo artefato de

⁵ Recursos, como por exemplo, definir restrições na ordem dos casos de uso.

software que utiliza o *framework* deverá utilizar a classe essencial. O ambiente contempla o registro explícito da classificação dos métodos como *base*, *template* ou *abstrato*.

3.4 Conclusão

O ambiente SEA é adequado para inserção de ferramenta que automatize a coleta de métricas de especificações OO, em função dos seguintes itens:

- flexibilidade do ambiente, que suporta acrescentar novas ferramentas,
- disponibilidade das informações das especificações orientadas a objetos no repositório do ambiente,
- disponibilidade de recursos neste ambiente, não previstos em UML, como o diagrama de corpo de método, que viabiliza a extração de algumas métricas ainda na fase de projeto,
- dar continuidade a um trabalho de doutorado de Silva [SIL 00], que propôs e implementou o protótipo do ambiente SEA.

Embora seja justificável a utilização do ambiente SEA neste trabalho como ambiente fornecedor de dados para a extração das métricas, este ambiente possui algumas restrições que merecem ser destacadas, como por exemplo, o ambiente não controla versões dos artefatos nele especificados e a utilização do ambiente SEA se restringe ao meio acadêmico.

4 MÉTRICAS PARA SOFTWARE ORIENTADO A OBJETOS

4.1 *Introdução*

No capítulo 2, discutiu-se qualidade de software, com ênfase à qualidade de software orientado a objetos e a subsídios utilizados para auxiliar a avaliação da qualidade, como métricas. Métricas voltadas para software orientado a objetos serão apresentadas neste capítulo.

Características de um software são avaliadas quantitativamente por várias razões, tais como subsidiar a indicação da qualidade do produto, avaliar o processo, servir de referência para estimativas, avaliar se os recursos de modelagem de projetos e programação foram bem explorados, e até como justificativa para aquisição de novas ferramentas e de capacitação.

Segundo Pressman [PRE 95], *na maioria dos empreendimentos técnicos, as medições e as métricas ajudam a entender o processo técnico usado para se desenvolver produto, como também o próprio produto.*

As métricas de software são classificadas de diferentes maneiras por diversos autores. Nesse trabalho será adotada uma classificação definida por Pressman [PRE 95] denominado métricas técnicas. Essa classificação refere-se a métricas que se concentram especificamente nas características de software (complexidade lógica e grau de modularidade) e não no processo utilizado para desenvolver o software.

As métricas técnicas podem ser divididas em duas categorias, quanto à forma de obtenção dos valores das medidas:

- **Medidas diretas** – referem-se às medidas que podem ser obtidas com medições aplicadas diretamente ao software, tais como, número de linhas de código e tamanho da interface. Essas medidas são fáceis de serem realizadas, desde que sejam adotadas convenções previamente.

- **Medidas indiretas** – são medidas indiretas do produto, por exemplo, o fator de reutilização do software relacionada à utilização de recursos tais como polimorfismo e herança.

Realizar medidas durante o processo de desenvolvimento de software é tão importante quanto em qualquer outro processo, pois através das medidas é possível controlar o que está sendo produzido.

Segundo Erni [ERN 96], o uso de métricas de projetos como parte de um processo de desenvolvimento tem dois problemas:

- Muitas vezes são usadas métricas complexas, ou seja, aquelas que combinam várias métricas simples em uma única fórmula, tornando difícil justificar o valor gerado pela fórmula.
- Para muitas métricas de projetos, não são identificados critérios para definir valores adequados, esperados para os resultados das métricas. Isso torna a interpretação do resultado das medições bastante difícil.

O processo de desenvolvimento ideal seria o desenvolvedor aplicar ao projeto as suas regras, e, através da aplicação permanente de métricas de projeto, obter um alerta imediato quando alguma regra fosse violada, para que adequações fossem realizadas, garantindo assim, a obtenção de um projeto com qualidade. Isto é ilustrado na figura 4.1. Porém, para a maioria das métricas é preciso realizar uma interpretação do resultado em função do tipo do projeto.

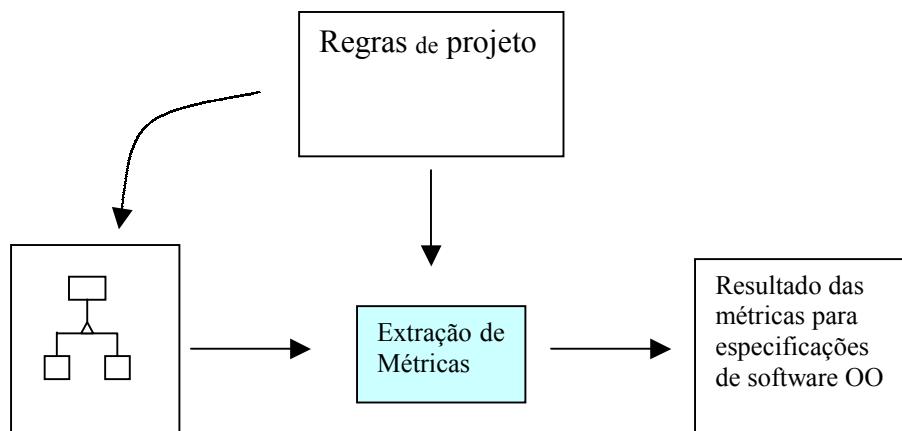


FIGURA 4.1 – Exemplo de processo ideal de aplicação de métricas

Hoje é possível encontrar sistemas desenvolvidos com técnicas e linguagens de programação orientada a objetos, com estruturas rígidas e inflexíveis, sendo difícil incorporar novas funcionalidades, em função das inevitáveis mudanças de requisitos [COR 00]. Não basta que o modelo do sistema esteja semanticamente correto para garantir que sistema será flexível e reutilizável. É necessário que a qualidade do modelo seja garantida, quanto aos critérios de flexibilidade e reutilização. Para isso é necessário que os modelos desenvolvidos sejam avaliados em relação aos critérios de qualidade definidos. Essas avaliações podem ser realizadas utilizando-se das métricas propostas para software orientado a objetos, que quantificam características específicas da orientação a objetos.

As métricas existentes para o desenvolvimento de software com o modelo funcional não são eficientes para desenvolvimento orientado a objetos, em função de conceitos específicos do paradigma, tais como classe, polimorfismo, encapsulamento e herança. Para suprir essa lacuna, novas métricas foram propostas por diversos autores. Algumas delas são descritas neste capítulo. O critério adotado para selecionar as métricas, foi o de produzir uma coletânea ampla com todas as métricas identificadas na literatura para software orientado a objetos.

Exemplo de cálculo das métricas e como os resultados são apresentados pela ferramenta de extração de métricas, posposta neste trabalho, são descritos no anexo 1. As especificações utilizadas para demonstrar os cálculos, apresentados no anexo 1, são pequenas e hipotéticas, porém, uma especificação de uma aplicação real é utilizada no capítulo 6 - Estudo de Caso.

Neste trabalho as métricas foram agrupadas em métricas de polimorfismo, acoplamento, coesão, encapsulamento e complexidade. Este critério foi adotado para tornar a apresentação das métricas mais didática, conforme apresentado na tabela 4.1 .

Métrica	Classificação	Seção
Fator de Acoplamento – COF	Acoplamento	4.2.1
Acoplamento entre Classes de Objetos – CBO	Acoplamento	4.2.2
Acoplamento de classes por meio de interação Classe-Atributo		
Acoplamento Classe-Atributo por importação nos ancestrais– ACAIC	Acoplamento	4.2.3.1
Acoplamento Classe-Atributo por exportação nos descendentes – DCAEC	Acoplamento	4.2.3.2
Acoplamento por Classe-Atributo por importação entre classes sem relação de herança – OCAIC	Acoplamento	4.2.3.3
Acoplamento por Classe-Atributo por exportação entre classes sem relação de herança – OCAEC	Acoplamento	4.2.3.4
Acoplamento de classes por meio de interação Classe-Método		
Acoplamento Classe-Método por importação nos ancestrais– ACMIC	Acoplamento	4.2.4.1
Acoplamento Classe-Método por exportação nos descendentes – DCMEC	Acoplamento	4.2.4.2
Acoplamento Classe-Método por importação entre classes sem relação de herança– OCMIC	Acoplamento	4.2.4.3
Acoplamento Classe-Método por exportação entre classes sem relação de herança – OCMEC	Acoplamento	4.2.4.4
Acoplamento de classes por meio de interação Método-Método		
Acoplamento Método-Método por importação nos ancestrais – AMMIC	Acoplamento	4.2.5.1
Acoplamento Método-Método por exportação nos descendentes – DMMEC	Acoplamento	4.2.5.2
Acoplamento Método-Método por importação entre classes sem relação de herança – OMMC	Acoplamento	4.2.5.3
Acoplamento Método-Método por exportação entre classes sem relação de herança – OMMEC	Acoplamento	4.2.5.4
Fator de Atributos Ocultos – AHF	Encapsulamento	4.3.1
Fator de Métodos Ocultos – MHF	Encapsulamento	4.3.2
Tamanho dos Métodos nas Classes	Complexidade	4.4.1
Número de Argumentos no Método	Complexidade	4.4.2
Número de Métodos nas Classes – NOM	Complexidade	4.4.3
Número de Classes Imediatamente Descendentes – NOC	Complexidade	4.4.4
Profundidade da Árvore de Herança – DIT	Complexidade	4.4.5
Fator de Herança de Métodos – MIF	Complexidade	4.4.6
Fator de Herança de Atributos – AIF	Complexidade	4.4.7
Reação de uma Classe – RFC	Complexidade	4.4.8
Métodos Complexos por Classe - WMC	Complexidade	4.4.9
Referência à Subclasses	Complexidade	4.4.10
Falta de Coesão – LCOM	Coesão	4.5.1
Falta de Coesão – LCOM - Modificado	Coesão	4.52

Métrica	Classificação	Seção
Fator de Polimorfismo – POF	Polimorfismo	4.61
Sobrecarga em Classes Isoladas – OVO	Polimorfismo	4.6.2
Polimorfismo Estático nos Ancestrais – SPA	Polimorfismo	4.6.3
Polimorfismo Estático nos Descendentes – SPD	Polimorfismo	4.6.4
Polimorfismo Estático em Relação de Herança – SP	Polimorfismo	4.6.5
Polimorfismo Dinâmico nos Ancestrais – DPA	Polimorfismo	4.6.6
Polimorfismo Dinâmico nos Descendentes – DPD	Polimorfismo	4.6.7
Polimorfismo Dinâmico – DP	Polimorfismo	4.6.8
Polimorfismo em relação sem herança – NIP	Polimorfismo	4.6.9

TABELA 4.1 – Classificação das métricas

As métricas descritas a seguir foram identificadas, porém, como são utilizadas para cálculo de outras métricas e por serem relativamente simples, estas não possuem um detalhamento individual. Essas medidas quantificam características de classes, métodos e atributos, conforme descrito a seguir:

Classe:

- Quantidade de Classes;
- Quantidade de Classes Abstratas;
- Quantidade de classes Concretas;

Atributo:

- Quantidade de atributos definidos diretamente na classe;
- Quantidade de atributos herdados pela classe;
- Quantidade de atributos disponíveis na classe;
- Quantidade de atributos definidos na especificação;
- Quantidade de atributos herdados na especificação;
- Quantidade de atributos disponíveis na especificação;

Métodos:

- Quantidade de métodos definidos na classe;
- Quantidade de métodos disponíveis na classe (definidos mais herdados);
- Quantidade de métodos herdados na classe (herdados menos sobreescritos);
- Quantidade de métodos ocultos na classe
- Quantidade de métodos visíveis na classe
- Quantidade de métodos sobreescritos na classe
- Quantidade de métodos *templates* na classe,
- Quantidade de métodos abstratos na classe

- Quantidade de métodos regulares na classe
- Quantidade de métodos definidos na especificação
- Quantidade de métodos disponíveis na especificação (definidos mais herdados);
- Quantidade de métodos herdados na especificação (herdados menos sobrescritos);
- Quantidade de métodos ocultos na especificação
- Quantidade de métodos visíveis na especificação
- Quantidade de métodos sobrescritos na especificação
- Quantidade de métodos *templates* na especificação
- Quantidade de métodos abstratos na especificação
- Quantidade de métodos regulares na especificação

A seguir serão detalhadas todas as métricas apresentadas na tabela 4.1.

4.2 Métrica de Acoplamento

Uma característica da estrutura interna diretamente relacionada à qualidade de software é o acoplamento, que se refere ao grau de interdependência entre as classes. Acoplamento entre classes deve ser controlado [CHI 93] [POE 98]:

1. Um alto grau de acoplamento dificulta o reuso, pois classes mais independentes são mais fáceis de serem utilizadas em outras aplicações.
2. Para melhorar a modularidade e o encapsulamento, o acoplamento entre classes deveria ser o menor possível. Um alto grau de acoplamento entre classes de objetos aumenta a sensibilidade de mudanças em partes do projeto, pois mudança em uma classe pode afetar outras classes, aumentando a dificuldade de manutenção.
3. Usualmente a medida de acoplamento é utilizada para avaliar a complexidade do teste a ser feito em parte do projeto, pois quanto maior o acoplamento mais rigoroso deve ser o teste.

Acoplamento pode acontecer por passagem de mensagem entre instâncias de classes (acoplamento dinâmico) ou por *links* de associação semântica (acoplamento estático). É desejável que classes sejam tão pouco dependentes quanto possível uma das outras [ABR 95].

A primeira métrica a ser tratada neste tópico, para avaliar o acoplamento, será o Fator de Acoplamento, que é aplicado à especificação como um todo. Essa métrica foi primeiramente proposta por Abreu e Carapuça [ABR 94]. A segunda métrica será Acoplamento entre Classes de Objetos, proposta por Chidamber e Kemerer [CHI 93].

Em seguida será apresentado um conjunto de métricas de acoplamento, elaborado por Briand, Devandu e Melo [BRI 97]. Essas métricas são utilizadas para medir diferentes tipos de interação entre classes, levando em consideração:

- tipo de relacionamento entre as classes (existência ou não de relação de herança),
- a localização esperada do impacto de mudança, ou seja, se o impacto da mudança flui em direção à classe avaliada (importação) ou em direção oposta à classe (exportação).
- o tipo de interação Classe-Atributo, Classe-Método ou Método-Método.

A partir de todas as combinações anteriores serão apresentadas 12 métricas, para auxiliar o projetista a avaliar o impacto do acoplamento na qualidade do projeto resultante.

As métricas de Briand, Devandu e Melo [BRI 97] foram propostas para avaliar código de programas gerados em linguagem C⁺⁺ e em um outro trabalho [VEI 99], para avaliação de código Java.

O uso dessas métricas servirá para identificar o grau de acoplamento entre classes, seja por importação ou por exportação, sendo que quanto maior o grau de acoplamento de exportação de uma classe, maior será o impacto em outras classes, quando a classe for alterada. E quanto maior o grau de importação em uma classe maior será o impacto na classe quando ocorrer mudanças nas outras classes, ou seja, tem-se uma visão da interdependência de classes. Quando mais uma classe for dependente de outras classes, será mais difícil entendê-la e dessa forma, mais fácil cometer erros, e por não serem entendidas corretamente, serem mal utilizadas.

4.2.1 Fator de Acoplamento – CF (*Coupling Factor*)

Essa métrica⁶ representa a medida de acoplamento entre classes (excluindo acoplamento causado por herança) e indica através de um índice, que varia de 0 (não uso ou ausência total) a 1 (máximo uso ou presença máxima possível) o fator de acoplamento entre as classes da especificação. Foi primeiramente apresentada por Brito e Abreu [ABR 94] e é aplicada à especificação como um todo.

Um alto grau de acoplamento tende a aumentar a complexidade, reduzir o encapsulamento e o potencial de reuso, dificultar o entendimento e a manutenibilidade do software. Dessa forma, é desejado que CF tenha um baixo valor. [ABR 94]. Sobre a avaliação do resultado da métrica, Pressman [PRE 02] descreve:

Apesar de muitos fatores afetarem a complexidade do software, sua inteligibilidade e sua manutenibilidade, é razoável concluir que à medida que o valor CF aumenta, a complexidade do software também vai aumentar, e a inteligibilidade, a manutenibilidade e o potencial de reuso podem piorar como resultado.

A definição da métrica diz que o acoplamento entre classes ocorre por relacionamento entre classe cliente (C_c) e uma classe fornecedora (C_s), ou seja, se uma classe C_c tem pelo menos uma referência a método ou atributo da classe C_s [ABR 94]. Porém, neste trabalho o acoplamento entre classes foi avaliado em função da invocação de mensagens de outra classe, a classe fornecedora pode ser referenciada na classe cliente através de tipo de atributo, tipo de argumentos de métodos, variáveis locais do método e tipo de retorno de método (caso haja um retorno do tipo de uma classe).

Cálculo:

$$CF = \frac{\sum_{i=1}^{TC} \left[\sum_{j=1}^{TC} \text{é_Cliente}(C_i, C_j) \right]}{TC^2 - TC}$$

⁶ Fator de Acoplamento pertence ao conjunto de métricas conhecido como MOOD (*Metrics for Object Oriented Design*)

Onde:

C = Classe;

TC = Número total de classe do projeto em consideração;

$$\text{é_Cliente}(C_c, C_s) = \begin{cases} 1 & \text{se } C_c \Rightarrow C_s \wedge C_c \neq C_s \\ 0 & \text{caso contrário} \end{cases}$$

A relação $C_c \Rightarrow C_s$, representa que a classe C_c contém pelo menos uma referência não hierárquica da classe C_s . [ABR 94]

Em 1995 foi feita uma revisão do conjunto das métricas MOOD [ABR 95a] e a fórmula da métrica CF foi redefinida, se comparada à fórmula proposta originalmente em [ABR 94].

$$\text{e } CF = \frac{\sum_{i=1}^{TC} \left[\sum_{j=1}^{TC} \text{é_Cliente}(C_i, C_j) \right]}{TC^2 - TC - 2 \times \sum_{i=1}^{TC} DC(C_i)}$$

Onde $(TC^2 - TC)$ representa o valor máximo possível de acoplamento entre as classes de uma especificação.

$DC(C_i)$ representa o número de descendentes da classe C_i .

$2 \times \sum_{i=1}^{TC} DC(C_i)$ representa o valor máximo possível de acoplamento, feito

em uma relação de herança.

$$\text{é_Cliente}(C_c, C_s) = \begin{cases} 1 & \text{se } C_c \Rightarrow C_s \wedge C_c \neq C_s \\ \wedge \neg(C_c \rightarrow C_s) \\ 0 & \text{caso contrário} \end{cases}$$

onde a relação $C_c \rightarrow C_s$ representa uma relação de herança (C_c herda de C_s).

Dessa forma é avaliado o acoplamento sem relação de herança, onde C_c e C_s não podem estar ligadas por qualquer relação de herança, seja direta ou indiretamente.

Então o numerador representa o valor real de acoplamento, sem relação de herança e o denominador representa o valor máximo possível de acoplamento, sem relação de herança entre as classes.

4.2.2 Acoplamento entre Classes de Objetos – CBO (*Coupling Between Object Classes*)

Essa métrica provê o número de classes a qual a classe em questão está acoplada. Foi proposta por Chidamber e Kemerer [CHI 93], que diz: *duas classes são acopladas quando métodos declarados em uma classe usam métodos ou atributos definidos por outra classe.*

Cálculo:

Não há uma expressão matemática proposta pelo autor para calcular o valor para métrica. Neste trabalho essa métrica foi implementada da seguinte forma:

Para cada classe é contabilizada a quantidade de classes que são referenciadas na classe, podendo ser através de tipo de atributo, tipo de parâmetro de método, tipo do retorno de método (caso seja um retorno do tipo de uma classe), variável de método ou por invocação de mensagem de outra classe.

Para esta métrica está sendo identificado acoplamento também em relação de herança.

4.2.3 Acoplamento de Classes por Meio de Interação Classe-Atributo

O tipo de interação Classe-Atributo representa o acoplamento entre classes através do atributo, ou seja, uma classe possui um atributo do tipo de outra classe. Supondo uma classe A e uma classe B, se a classe A tem um atributo do tipo da classe B, então a classe A é acoplada à classe B através de atributo. A interação Classe-Atributo será investigada entre classes com e sem relação de herança. Através desse

grupo de métricas será possível visualizar o sentido do impacto da mudança, ou seja, se o impacto flui em direção a classe (de importação) ou em direção oposta a classe (de exportação).

As definições das funções a seguir se fazem necessárias para a descrição das métricas, pertencentes ao conjunto de métrica que buscam o acoplamento entre classes, por meio de interação Classe-Atributo descrito, nesta seção:

- a. $ACA(C_i, C_j)$ é definido como o número de interação Classe-Atributo, que ocorre entre as declarações de atributos da classe C_i e a classe C_j .
- b. $ACA(C_j, C_i)$ é definido como o número de interações Classe-Atributo que ocorrem entre as declarações de atributos da classe C_j e a classe C_i .
- c. $Ancestrais(C_i)$ é um operador que retorna um conjunto de ancestrais da classe C_i .
- d. $Descendentes(C_i)$ é um operador que retorna um conjunto de descendentes da classe C_i .
- e. $Outras(C_i)$ é um operador que retorna o conjunto de classes distintas que não são ascendentes nem descendentes da classe C_i .

4.2.3.1 Acoplamento por Classe-Atributo por Importação nos Ancestrais-ACAIC (*Ancestors Class-Attribute Import Coupling*)

O objetivo dessa métrica é identificar o número de vezes que a classe C_i faz referência à outra classe através de atributo. Considera, para efeitos da avaliação, o conjunto de classes ancestrais da classe em questão, identificando assim se a classe C_i depende de suas ancestrais. Essa medida é obtida segundo a relação matemática:

$$ACAIC(C_i) = \sum_{C_j \in Ancestrais(C_i)} ACA(C_i, C_j)$$

4.2.3.2 Acoplamento por Classe-Atributo por Exportação nos Descendentes - DCAEC (*Descendant Class-Attribute Export Coupling*)

Semelhante à métrica anterior, esta busca identificar o acoplamento através de atributos, porém, nas classes descendentes. Subsídia a identificação do número de vezes que a classe C_i é referenciada nas suas subclasses, ou seja, é possível identificar se mudanças ocorridas na classe C_i serão exportadas para as suas classes descendentes. Essa medida é obtida, segundo a relação matemática:

$$DCAEC(C_i) = \sum_{C_j \in Descendentes(C_i)} ACA(C_j, C_i)$$

4.2.3.3 Acoplamento por Classe-Atributo por Importação entre Classes Sem Relação de Herança – OCAIC (*Others Class-Attribute Import Coupling*)

As duas métricas anteriores avaliam o acoplamento por classe-atributo em classes com relação de herança. Já esta métrica avalia classes sem relação de herança. Assim, é possível identificar se a classe C_i sofrerá impacto de mudanças efetuadas em classes não relacionadas por herança. Essa medida é obtida, segundo a relação matemática:

$$OCAIC(C_i) = \sum_{C_j \in Outras(C_i)} ACA(C_i, C_j)$$

4.2.3.4 Acoplamento por Classe-Atributo por Exportação entre Classes Sem Relação de Herança - OCAEC (*Others Class-Attribute Export Coupling*)

Essa métrica também avalia acoplamento classe-atributo em classes não relacionadas por herança. Mudanças realizadas na classe C_i poderão causar impacto em suas classes descendentes, que a importaram na forma de atributo. Essa medida é obtida, segundo a relação matemática:

$$OCAEC(C_i) = \sum_{C_j \in Outras(C_i)} ACA(C_j, C_i)$$

4.2.4 Acoplamento de Classes por Meio de Interação Classe-Método

O tipo de interação Classe-Método representa o acoplamento entre classes através de método, ou seja, a assinatura de um método de uma classe pode ter referência de outra classe. Supondo uma classe A e uma classe B, se a classe A tem um método cujo parâmetro faz referência à classe B ou retorna uma instância da classe B, então classe A é acoplada à classe B via método. A interação Classe-Método será investigada entre classes com relação de herança e sem relação de herança.

As definições das funções a seguir, fazem necessário para a descrição das métricas pertencentes ao conjunto de métricas que buscam o acoplamento entre classes por meio de interação Classe-Método:

- a. $ACM(C_i, C_j)$ é definido como o número de interação Classe-Método que ocorrem entre métodos da classe C_j e a classe C_i .
- b. $ACM(C_j, C_i)$ é definido como o número de interação Classe-Método que ocorrem entre métodos da classe C_j e a classe C_i .

As funções: Ancestrais(C_i), Descendentes(C_i) e Outras(C_i) já foram definidos na seção 4.1.3 – Acoplamento entre Classes por meio de Interação Classe-Atributo.

4.2.4.1 Acoplamento Classe-Método por Importação nos Ancestrais- ACMIC (*Ancestors Class-Method Import Coupling*)

Essa métrica busca estabelecer parâmetro para avaliação do acoplamento entre uma classe C_i e suas ancestrais através dos métodos. Para tal, avalia os tipos de parâmetros e retorno de métodos. Provê insumos para identificar se a classe C_i será impactada, caso ocorra alteração em classes ancestrais de C_i . Essa medida é obtida, segundo a relação matemática:

$$ACMIC(C_i) = \sum_{C_j \in Ancestrais(C_i)} ACM(C_i, C_j)$$

4.2.4.2 Acoplamento Classe-Método por Exportação nos Descendentes – DCMEC (*Descendant Class-Method Export Coupling*)

Nessa métrica a avaliação do acoplamento entre classes através de métodos é realizada nos descendentes de uma determinada classe C_i . Busca identificar se mudanças ocorridas na classe C_i serão propagadas para as suas classes descendentes. Essa medida é obtida segundo a relação matemática:

$$DCMEC(C_i) = \sum_{C_j \in Descendentes(C_i)} ACM(C_j, C_i)$$

4.2.4.3 Acoplamento Classe-Método por Importação entre Classes Sem Relação de Herança – OCMIC (*Others Class-Method Import Coupling*)

Esta métrica é similar às duas métricas anteriores, porém, são consideradas para análise somente as classes que não possuem relação de herança com a classe C_i . Portanto é possível identificar o número de vezes que classes são referenciadas na classe C_i . Essa medida é obtida, segundo a relação matemática:

$$ACMIC(C_i) = \sum_{C_j \in Outras(C_i)} ACM(C_i, C_j)$$

4.2.4.4 Acoplamento Classe-Método por Exportação entre Classes Sem Relação de Herança – OCMEC (*Others Class-Method Export Coupling*)

Semelhante à métrica anterior, esta visa identificar o quanto a classe C_i está acoplada à classes não relacionadas por herança. Dessa forma é identificado o quanto a classe C_i exporta o impacto de possíveis mudanças às classes não relacionadas por herança. Essa medida é obtida, segundo a relação matemática:

$$OCMEC(C_i) = \sum_{C_j \in Outras(C_i)} ACM(C_j, C_i)$$

4.2.5 Acoplamento de Classes por Meio de Interação Método-Método

O tipo de interação Método-Método representa um acoplamento entre classes através de métodos, ou seja, um método invoca método de outra classe ou um método é passado como parâmetro para outra mensagem. Supondo uma classe A com um método mA e uma classe B com o método mB, se mA chama o método mB ou se mB é passado como parâmetro para mA, portanto é dito que existe uma interação entre as classes A e B. A interação Método-Método será investigada entre classes, relacionadas por herança e não relacionadas por herança. Através desse grupo de métricas será possível visualizar também a localização esperada de um impacto de mudança, por exemplo, uma classe importa impacto de mudança dos seus ancestrais.

As definições das funções a seguir, se fazem necessárias para a descrição das métricas pertencentes ao conjunto de métricas que buscam o acoplamento entre classes, por meio de interação Método-Método:

- a. $AMM(C_i, C_j)$ é definido como o número de interação Método-Método, que ocorrem entre métodos da classe C_i e métodos da classe C_j .
- b. $AMM(C_j, C_i)$ é definido como o número de interações Método-Método, que ocorrem entre métodos da classe C_j e métodos da classe C_i .

As funções: Ancestrais(C_i), Descendentes(C_i) e Outras(C_i) já foram definidos na seção 4.1.3 – Acoplamento entre Classes por meio de Interação Classe-Atributo.

4.2.5.1 Acoplamento Método-Método por Importação nos Ancestrais – AMMIC (*Ancestors Method-Method Import Coupling*)

Essa métrica identifica a quantidade de vezes que métodos da classe C_i fazem referência a métodos de suas classes ancestrais. Dessa forma, é possível identificar quando a classe C_i dependente de suas ancestrais. Essa medida é obtida segundo a relação matemática:

$$AMMIC(C_i) = \sum_{C_j \in Ancestrais(C_i)} AMM(C_i, C_j)$$

4.2.5.2 Acoplamento Método-Método por Exportação nos Descendentes – DMMEC (*Descendants Method-Method Export Coupling*)

Nessa métrica, ao contrário da anterior, busca-se identificar o número de vezes que métodos de uma superclasse C_i são invocados por métodos das subclasses. Assim é possível identificar as subclasses de C_i que serão impactadas, caso sejam realizadas mudanças em C_i . Essa medida é obtida, segundo a relação matemática:

$$DMMEC(C_i) = \sum_{C_j \in Descendentes(C_i)} AMM(C_j, C_i)$$

4.2.5.3 Acoplamento Método-Método por Importação entre Classes Sem Relação de Herança – OMMIC (*Others Method-Method Import Coupling*)

Essa métrica avalia o acoplamento método-método em classes sem relação de herança. Quantifica o número de métodos de uma classe C_i que invocam métodos de outras classes. Possibilita identificar a origem do impacto sobre a classe C_i em função da importação de mudanças de outras classes. Essa medida é obtida, segundo a relação matemática:

$$OMMIC(C_i) = \sum_{C_j \in Outras(C_i)} AMM(C_i, C_j)$$

4.2.5.4 Acoplamento Método-Método por Exportação entre Classes Sem Relação de Herança – OMMEC (*Others Method-Method Export Coupling*)

Também visa quantificar o acoplamento método-método, porém quantifica a exportação do impacto de mudanças ocorridas na classe C_i em outras classes não relacionadas por herança. Essa medida é obtida, segundo a relação matemática:

$$OMMEC(C_i) = \sum_{C_j \in Outras(C_i)} AMM(C_j, C_i)$$

4.3 Métricas de Encapsulamento

A seguir serão apresentadas métricas para avaliar o uso do conceito de encapsulamento de atributos e métodos de uma especificação como um todo.

4.3.1 Fator de Atributos Ocultos – AHF (*Attribute Hiding Factor*)

Essa métrica⁷ foi primeiramente apresentada por Brito e Abreu [ABR 94]. É aplicada à especificação como um todo. Indica, através de um índice, que varia de 0 (não uso) a 1 (máximo uso), o fator de atributos ocultos na especificação. Quando o valor do fator atingir 1 (um), demonstra o máximo grau de utilização do conceito de encapsulamento para atributos, ou seja, que todos os atributos são ocultos.

Essa métrica proporciona a avaliação da especificação quanto ao critério de encapsulamento de atributos no projeto, que no paradigma orientado a objetos determina que todos os atributos sejam ocultos à classe, somente sendo acessíveis por outras classes, através de seus métodos públicos (interfaces). Vale observar que nem toda linguagem de programação obedece tal critério, ficando a cargo do desenvolvedor definir o grau de encapsulamento. Por exemplo, em linguagens como C++ e Java, atributos ocultos podem ser classificados como privados⁸ e protegidos⁹ e os atributos visíveis como atributos públicos¹⁰.

Conforme Correa [COR 00], as heurísticas de projeto orientado a objetos podem ser uma fonte para a identificação de construções problemáticas de projeto, pois quando essas são violadas correspondem a problemas em potencial que um projeto pode apresentar. Um atributo definido como público viola o princípio de que *todos os dados devem estar escondidos dentro de sua classe* [Riel 96¹¹ apud [COR 00]].

Erni [ERN 96] também descreve que *uma classe não deve ter atributos públicos*.

⁷ Essa métrica pertence ao conjunto de métricas conhecido como MOOD (*Metrics for Object Oriented Design*)

⁸ Atributos privados são acessíveis somente dentro do escopo da própria classe e são herdados pelas classes descendentes, mas não são acessíveis.

⁹ Atributos protegidos são acessíveis somente dentro do escopo da própria classe e são herdados pelas classes descendentes.

¹⁰ Atributos públicos são acessíveis por qualquer objeto e são herdados pelas classes descendentes.

¹¹ A .Riel, *Object Oriented Design Heuristics*, Addison-Wesley, 1996.

Em função das afirmações acima, o valor do fator de herança deve ser 1 (um), pois quando esse for menor que 1, é um indicativo que o projeto não atende a essa premissa.

Cálculo:

Essa medida pode ser obtida, segundo a relação matemática proposta por Brito e Abreu [ARB 94] [ARB 95a]:

$$AHF = \frac{\sum_{i=1}^{TC} Ah(C_i)}{\sum_{i=1}^{TC} Ad(C_i)}$$

Sendo:

$A_h(C_i)$ = número de atributos ocultos na classe C_i

$A_d(C_i)$ = número de atributos definidos na classe C_i ,

Sendo A_d obtido com a seguinte relação: $A_d(C_i) = A_v(C_i) + A_h(C_i)$

Onde, $A_v(C_i)$ = número de atributos visíveis na classe C_i

O numerador de AHF é o somatório de todos os atributos ocultos em todas as classes da especificação e o denominador representa o total de todos os atributos definidos em todas as classes da especificação.

Neste trabalho atributo oculto é o mesmo que atributo protegido, visto que no ambiente SEA todos os atributos são definidos como protegidos. Por isso essa métrica não foi implementada, pois o AHF será sempre 1, ou seja 100% dos atributos são ocultos.

4.3.2 Fator de Métodos Ocultos – MHF (*Method Hiding Factor*)

Essa métrica foi primeiramente apresentada por Brito e Abreu [ABR 94] e é aplicada à especificação como um todo¹². Indica, através de um índice, que varia de 0 (zero) a 1 (um) o fator de métodos ocultos na especificação.

¹² Essa métrica pertence ao conjunto de métricas conhecido como MOOD (*Metrics for Object Oriented Design*)

Quanto mais próximo de zero o valor de MHF, maior será a quantidade de métodos acessíveis por outras classes. Rumbaugh [RUM 94] descreve: *O encapsulamento impede que um programa se torne tão interdependente que uma pequena modificação possa causar grandes efeitos de propagação.*

Algumas funcionalidades implementadas na classe são restritas à própria classe. Assim, esses métodos devem ser ocultos às outras classes, evitando que clientes vejam detalhes de implementação interna. Somente os serviços que a classe pode oferecer devem ser públicos, o que são chamados de interface da classe [VEI 99]. Essa métrica auxilia o desenvolvedor a identificar o grau de encapsulamento de métodos no projeto.

Para essa métrica existem dois extremos: Ter uma especificação com 0% de métodos ocultos não é ideal, pois todos os detalhes de implementação da classe estariam visíveis e acessíveis às demais classes. Por outro lado, 100% é um valor impraticável, pois caso isso ocorra, as classes não teriam finalidades, pois não estariam aptas a oferecer serviços.

Em linguagens como C++ e Java, de forma análoga à classificação de atributos, os métodos ocultos são classificados como privados e protegidos e os métodos visíveis são classificados como públicos.

No ambiente SEA os métodos podem ser classificados explicitamente como ocultos ou públicos, o que semanticamente significa protegidos ou públicos, respectivamente. Dessa forma, neste trabalho método oculto significa o mesmo que método protegido.

Cálculo:

Essa medida pode ser obtida, segundo a relação matemática proposta por Brito e Abreu [ARB 94] [ARB 95a]:

$$MHF = \frac{\sum_{i=1}^{TC} Mh(Ci)}{\sum_{i=1}^{TC} Md(Ci)}$$

Sendo:

$M_h(C_i)$ = número de métodos ocultos na classe C_i

$M_d(C_i)$ = número de métodos definidos na classe C_i ,

Sendo M_d obtido com a seguinte relação: $M_d(C_i) = M_v(C_i) + M_h(C_i)$

Onde, $M_v(C_i)$ = número de métodos visíveis (interface) na classe C_i

4.4 Métricas de Complexidade

4.4.1 Tamanho dos Métodos

Métodos longos são mais difíceis de serem lidos e entendidos, dificultando as manutenções e futuras extensões das especificações, por isso a necessidade de avaliar o tamanho do método. Há várias indicações para avaliar o tamanho dos métodos, tais como: número de *statements* executáveis e número de atributos usados, sugeridos em [ABR 94a]. E avaliação do comprimento do método, em função do número de linhas – LOC (*Line of Code*) [ERN 96].

Em [ABR 94a] o LOC é utilizado para calcular a dimensão média de métodos e ressalta que o limite depende da linguagem de programação. O número de linhas de códigos fontes gerados em *Smalltalk* sugerido é 8 e em C++ 24, conforme [LOR 93] apud [ABR 94a].

Segundo Jonhson [JOH 88], é aconselhável que uma classe não tenha métodos longos, ou seja, não deve possuir métodos com mais de 30 linhas de código. Métodos com mais de 30 linhas devem ser subdivididos.

Métricas de tamanho são importantes para processos de estimativas e uma das métricas de tamanho mais tradicionais é o LOC.

Sobre os valores referentes ao número de linhas de código acima apresentados, não foi localizado nenhuma comprovação formal da validade desses.

Há muitas divergências em função da métrica LOC, pois apesar de ser uma medida relativamente fácil de ser extraída, existem argumentações opostas que alegam o fato da métrica ser dependente da linguagem de programação [PRE 02].

Neste trabalho, o tamanho do método será avaliado em função do número de *statements* executáveis. Embora métricas de número de linhas e *statements* sejam geralmente aplicadas sobre o código fonte, essas podem também ser aplicadas ainda em fase de projeto, desde que a metodologia adotada para o processo de desenvolvimento

possua suporte à especificação de algoritmos na fase de projeto. Neste trabalho a métrica, tamanho dos métodos, será aplicada já na fase de projeto, pois o ambiente SEA possibilita a descrição dos algoritmos dos métodos através do diagrama de corpo de método, o que não é previsto pela linguagem UML.

No Ambiente SEA, o corpo de método é composto por *statements*. A extração da métrica foi realizada em função do número de *statements*, e não em função do número de linhas, conforme prevê a métrica. Isso garante mais consistência dos dados extraídos, pois o valor da métrica não varia em função das disposições do código e sim do número de *statements*. Assim, um algoritmo descrito por um desenvolvedor tem mais probabilidade de conter o mesmo número de *statements* que o mesmo algoritmo descrito por outro desenvolvedor.

Para essa métrica foram considerados somente os métodos definidos na classe e não os herdados, evitando dessa forma a reavaliação dos métodos já avaliados nas superclasses. Métodos sobrescritos são considerados como definidos na classe avaliada, consequentemente, serão avaliados.

Cálculo:

Para todas as classes da especificação são extraídos todos os métodos definidos na própria classe (excluindo os métodos abstratos) e contabilizados os números de statements. Não foram contabilizados os Statements do tipo Comment (para conter comentários), Variables (para declaração de variáveis temporárias), ExternalStateTransitionDiagram (equivalente a um comentário, gerado automaticamente pela ferramenta para registrar a referência ao método descrito no diagrama de transição de estados) e ExternalSequenceDiagram (equivalente a um comentário, gerado automaticamente pela ferramenta para registrar a referência ao método descrito no diagrama de seqüência).

4.4.2 Número de Argumentos no Método

O objetivo dessa métrica é identificar o número de argumentos dos métodos. Métodos com um número grande de argumentos são difíceis de serem lidos, entendidos e a comunicação entre objetos se torna mais complexa.

Para se ter um bom entendimento das interfaces o número de argumentos do método deve ficar abaixo de seis, com exceção dos construtores que inicializam o objeto. Métodos com muitos argumentos devem ser particionados em métodos com menor número de argumentos, aumentando a possibilidade de semelhança com outros métodos. Geralmente métodos com menor número de argumentos são mais reutilizáveis [JOH 88].

Para o cálculo dessa métrica foram considerados somente os métodos definidos na classe e não os herdados, para evitar a reavaliação dos métodos já avaliados nas superclasses.

Cálculo:

Para todas as classes da especificação foram extraídos os métodos definidos na própria classe e para cada método foi contabilizado a quantidade de argumentos. Os métodos construtores não foram considerados para extração das métricas.

4.4.3 Número de Métodos nas Classes – NOM (*Number Of Methods*)

Essa métrica identifica o número de métodos definidos localmente na classe e foi proposta por (Li and Henry¹³ Apud [ABO 97]).

Uma classe não deve ter mais de 50 métodos [Johnson 88] Apud [ERN 96]. Vale ressaltar que não foi localizada nenhuma comprovação científica e formal sobre o número de métodos anteriormente descrito, assim com o valor sugerido como limite para o número de linhas de código fonte, apresentado na métrica *Tamanho de Métodos*.

O objetivo de uma classe é representar uma única abstração. Se uma classe tem de 50 a 100 métodos, isto representa uma abstração complexa. Isto significa que provavelmente tal classe não foi bem definida e provavelmente pode ser tratada como um conjunto de abstrações distintas. Classes grandes devem ser visualizadas com desconfiança e provavelmente indicam construções pobres [JOH 88].

¹³ Booch, G; “Object Oriented Analysis and Design with Applications (2nd ed)”, Benjamin / Cummings, Califórnia, 1994

Embora essa métrica fosse destinada para verificação de número de métodos locais na classe, foi implementada diante das seguintes considerações:

1. Avaliando o número de métodos definidos nas classes, ou seja, métodos descritos nas classes, exceto os herdados (métrica proposta pelo autor).
2. Avaliando o número de métodos disponíveis nas classes, entendido como métodos definidos na própria classe mais os herdados.

Cálculo:

Foram executados dois cálculos:

1. *Para todas as classes da especificação foi extraído o número de métodos definidos na própria classe;*
2. *Para todas as classes da especificação foi extraído o número de métodos disponíveis na classe.*

4.4.4 Número de Classes Imediatamente Descendentes – NOC (*Number of Children*)

Essa métrica identifica o número de classes diretamente ligadas à classe em estudo, através do relacionamento do tipo herança. Foi definida por Chidamber e Kemerer [CHI 93], que fizeram as seguintes considerações que justificam o uso da métrica:

- Um grande número de classes descendentes, indicam maior reuso, visto que herança é uma forma de reuso.
- Um grande número de classes descendentes, indicam maior probabilidade de abstração imprópria da super classe. Se uma classe tem um grande número de descendentes, pode ser um caso de mal emprego da sub-classificação.
- O número de classes descendentes dá uma idéia da influência que uma classe tem no projeto. Se uma classe tem um número grande de

descendentes, mais testes podem ser requeridos dos métodos daquela classe.

Uma classe com um grande número de filhos imediatamente descendentes representa a largura hierárquica da classe. O aumento da largura hierárquica de classes pode indicar a existência de grande distância semântica entre a classe e suas subclasses [DIA 97].

Sobre hierarquia de classes Dias [DIA 97] descreve:

Experiência com desenvolvimento de software orientado a objetos mostra que uma hierarquia de classe é geralmente pouco profunda e muito larga. Uma hierarquia muito larga induz à possibilidade de existirem características replicadas por várias classes e com um baixo grau de reutilização.

A largura da hierarquia pode ser reduzida da seguinte maneira:

- Primeiro deve-se buscar um novo nível de abstração, considerando a coesão semântica entre as classes.
- Segundo, caso o item anterior não seja suficiente ou aplicável para reduzir a largura da hierarquia, é indicado criar mais um nível de abstração, buscando grupos de classes semelhantes, usando os seguintes critérios: subclasses que redefiniram as características da superclasse, subclasses que não redefiniram as características da superclasse e subclasses que tiveram novas características adicionadas.

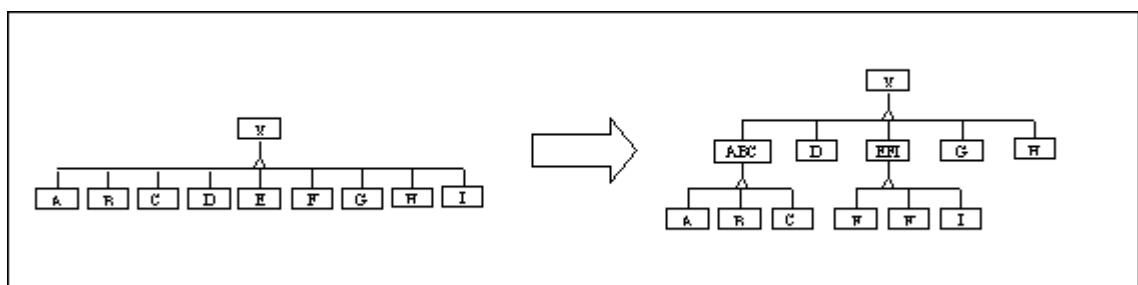


FIGURA 4.2 – Eliminando hierarquias largas [DIA 97]

Para essa métrica não há indicação de um valor ideal de número de descendentes imediatos.

Cálculo:

Para todas as classes da especificação é identificado o número de descendentes imediatos.

4.4.5 Profundidade da Árvore de Herança – DIT (*Depth of Inheritance Tree*)

Essa métrica mede a profundidade da hierarquia, ou seja, dada uma classe, verifica o nível de profundidade entre os seus descendentes. Foi definida por Chidamber e Kemerer [CHI 93]. O valor de DIT no nó raiz vale 0 (zero), estabelecendo o menor valor possível para o índice. Nos demais níveis da hierarquia este índice irá mostrar o número de ascendentes que a classe em questão possui. Os autores fazem as seguintes considerações:

- Uma classe em um nível de profunda alta na hierarquia de herança de classes, pode ter um grande número de métodos herdados, aumentando a complexidade do entendimento do seu comportamento. Portanto, árvores de hierarquias profundas constituem maior complexidade de projeto.
- Quanto mais alto o nível de profundidade de uma classe na hierarquia, maior o potencial de reuso dos métodos herdados.

Aumentar o nível de hierarquia de classes implica em maior esforço para entendimento e manutenção das classes, pois ao invés de localizar todas as características da classe na própria classe e encapsular todos os seus dados, estas características aparecem dispersas ao longo de toda a hierarquia. Por outro lado, uma estrutura hierárquica profunda aumenta o potencial de reutilização [REZ 00].

Alteração em uma classe, situada no topo de uma hierarquia, requer teste em classes descendentes em toda a hierarquia. Para reduzir a profundidade, Dias [DIA 97], sugere:

- Primeiro deve-se verificar as semelhanças entre as subclasses, compará-las à superclasse, analisando a possibilidade de incorporar as características da subclass na superclasse.

- Se a profundidade ainda persistir, após aplicar o item anterior, deve-se procurar formas semânticas para substituir o relacionamento de herança por outro tipo de relacionamento.

Conforme Booch [BOO 00],

Mantenha os relacionamentos de generalização geralmente equilibrados; as redes de heranças não deverão ser muito profundas (questione a existência de cinco ou mais níveis), nem muito extensas (ao contrário, busque a possibilidade de classes abstratas intermediárias).

A métrica não dará um indicativo de conformidade com um valor considerado ideal e sim terá caráter de demonstrar a situação hierárquica da especificação, pois se tem variantes que devem ser levadas em consideração, por exemplo o tamanho do projeto. Dessa forma, uma hierarquia de classes com profundidade hierárquica máxima igual a dois, pode ser considerada crítica para um projeto com três classes, porém, para um projeto com dez classes poderia ter outra interpretação.

A eliminação de hierarquias profundas pode refletir em métricas como: largura da hierarquia e número de classes.

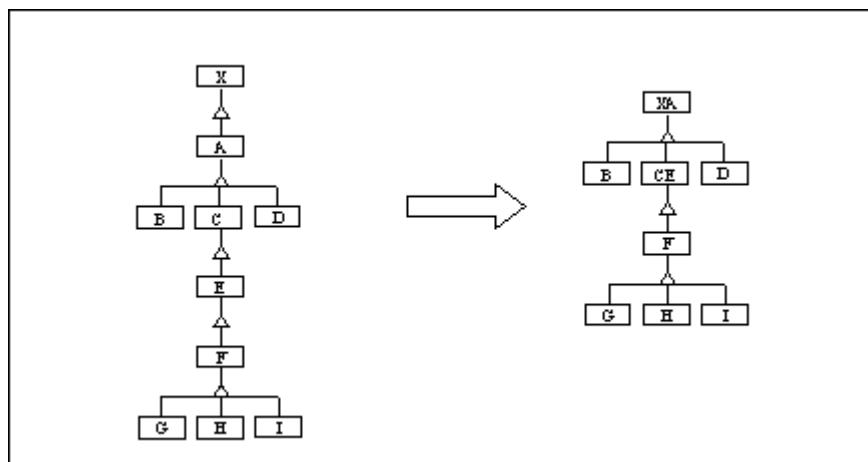


FIGURA 4.3 – Eliminando hierarquias profundas [DIA 97]

Cálculo:

Para todas as classes identificar o seu nível hierárquico na árvore de herança.

4.4.6 Fator de Herança de Métodos – MIF (*Method Inheritance Factor*)

Essa métrica foi proposta por Brito e Abreu [ABO 97] e é aplicada à especificação como um todo. O resultado é apresentado na forma de um índice, que pode variar de 0 (zero) a quase 1 (um), ou seja, este índice, no limite máximo, tende a 1 (um).

Caso o valor de MIF seja igual a 0 (zero) pode significar que:

- **Não há hierarquia de herança na especificação avaliada**

Não tendo relação de herança na especificação, o somatório de métodos herdados, que representa o numerador da expressão será 0 (zero), o que acarreta em fator de herança igual a zero.

- **Todos os métodos herdados foram sobreescritos**

Para essa métrica os métodos herdados e sobreescritos não são contabilizados no somatório de métodos herdados. Dessa forma, se todos os métodos herdados forem sobrepostos, o numerador da expressão será 0 (zero), o que acarreta em fator de herança igual a zero.

A interpretação do resultado da métrica em uma especificação que possui relacionamento de herança, pode feita da seguinte maneira:

- Um baixo valor do índice indica um baixo nível de reuso de implementação de método, pois métodos herdados podem ter sido sobreescritos, consequentemente, havendo um indicativo de um alto nível de reuso de interface.
- Valor de MIF tendendo a 1 (um), significa que o nível de reuso de implementação de método foi alto. Isto significa que muitos métodos foram herdados e não tiveram sua implementação alterada, ou seja, não foram sobreescritos, e, consequentemente, houve um indicativo de baixo reuso de interface.

Cálculo:

Essa medida pode ser obtida, segundo a relação matemática proposta por Brito e Abreu [ABO 97]:

$$MIF = \frac{\sum_{i=1}^{TC} Mi(C_i)}{\sum_{i=1}^{TC} Ma(C_i)} \quad (4)$$

Sendo:

$M_i(C_i)$ = número de métodos herdados na classe C_i (métodos herdados e não sobrescritos).

$M_a(C_i)$ = número de métodos disponíveis na classe C_i (métodos descritos mas classes assim como os herdados)

Sendo M_a obtido com a seguinte relação: $M_a(C_i) = M_d(C_i) + M_i(C_i)$

Onde, $M_d(C_i)$ = número de métodos definidos na classe C_i

4.4.7 Fator de Herança de Atributos – AIF (*Attribute Inheritance Factor*)

Essa métrica foi proposta por Brito e Abreu [ABO 97] e é aplicada à especificação como um todo. O fator de herança de atributos é apresentado na forma de um índice que pode variar de 0 (zero) a quase 1 (um), ou seja, este índice tende a 1 (um).

Caso o valor de AIF seja igual a 0 (zero) significa que não foi utilizada a herança de atributos na especificação, e quando o valor do índice tende a 1, significa a herança de atributos muito utilizada.

Cálculo:

Essa medida pode ser obtida, segundo a relação matemática proposta por Brito e Abreu [ABO 97]:

$$AIF = \frac{\sum_{i=1}^{TC} Ai(C_i)}{\sum_{i=1}^{TC} Aa(C_i)}$$

Sendo:

$A_i(C_i)$ = número de atributos herdados na classe C_i .

$A_a(C_i)$ = número de atributos disponíveis na classe C_i (atributos definidos localmente na classe mais os herdados)

Sendo A_a obtido com a seguinte relação: $A_a(C_i) = A_d(C_i) + A_i(C_i)$

Onde, $A_d(C_i)$ = número de atributos definidos na classe C_i

4.4.8 Reação de uma Classe – RFC (*Response For a Class*)

Essa métrica foi proposta por Chidamber e Kemerer [CHI 93]. RFC é o conjunto de métodos que podem ser executados potencialmente em resposta a uma mensagem recebida por um objeto daquela classe, independente do método pertencer ou não à classe. O conjunto de resposta ou reação de uma classe é avaliado somente em um primeiro nível de relacionamento.

Quando o valor de RFC aumenta, o esforço necessário para a atividade de teste também tende a aumentar, pois exige um maior nível de entendimento por parte do testador. O aumento de RFC eleva também a complexidade do projeto da classe.

Cálculo:

RFC é definido como $|RS|$ onde RS é o conjunto de respostas para a classe. E o conjunto de respostas para a classe é expresso como:

$$RS = \{M\} \bigcup_{\text{todos } i} \{R_i\}$$

onde $\{R_i\}$ é o conjunto de métodos chamados pelo método i

e $\{M\}$ é o conjunto de método na classe.

4.4.9 Métodos Complexos por Classe - WMC (*Weighted Method Count*)

A métrica WMC representa o somatório das complexidades ponderadas de todos os métodos definidos na classe. Essa métrica foi proposta por Chidamber e Kemerer [CHI 93]. Se a complexidade de todos os métodos da classe é considerada 1 (um), o resultado de WMC é igual ao número de métodos definidos na classe.

O número de métodos e sua complexidade são indicadores razoáveis do tempo e esforço necessários para desenvolver, manter e testar a classe. Além disso um grande número de métodos na classe aumenta o potencial de impacto nas classes descendentes. E ainda, à medida que o número de métodos aumenta, provavelmente a classe torna-se mais específica para a aplicação, limitando a possibilidade de reuso [PRE 02] [CHI 93] [ROS 98]. Em função desses fatores, Pressman [PRE 02] afirma que *o valor de WMC deve ser mantido razoavelmente baixo*

Os autores da métrica acrescentaram que a medida de complexidade utilizada no cálculo de WMC propositalmente não foi definida, permitindo assim, que diferentes critérios possam ser adotados para a medida de complexidade do método.

Em [BAS 96] foi considerado que todos os métodos da classe têm complexidades iguais, onde o valor de WMC é simplesmente o número de métodos definidos na classe. Essa hipótese foi assumida com o objetivo de simplificar o cálculo. Outra razão dessa adoção é o fato de que a escolha de uma medida de complexidade é um tanto arbitrário, visto que essa medida não foi especificamente definida pelos autores da métrica. Em alguns trabalhos como [ROS 98] e [BAS 96] o valor adotado para a complexidade foi 1.

Nesse trabalho a complexidade dos métodos foi avaliada em função do número de *statements*¹⁴, adotando o seguinte critério:

Complexidade método igual a 1, quando a quantidade de *statements* do método avaliado for menor ou igual à quantidade de *statements* definido pelo usuário, como parâmetro de avaliação.

Complexidade método igual a 2, quando a quantidade de *statements* do método avaliado for maior que a quantidade de *statements* definido pelo usuário, como parâmetro de avaliação.

Os valores sugeridos anteriormente para a complexidade dos métodos não foram validados, ou seja, não é comprovado que a complexidade realmente dobra quando se trata de métodos com um maior número de *statements*. Dessa forma, o segundo valor da

¹⁴ A contagem do número de *statements* do método é descrita na métrica **Tamanho do Método**.

complexidade, aqui definido como 2, pode ser alterado pelo usuário da ferramenta na tela de parâmetros.

Cálculo:

Conforme definido pelo autor da métrica a Classe C_1 , possui métodos $M_1 \dots M_n$ e esses métodos possuem complexidades $c_1 \dots c_n$.

$$WMC = \sum_{i=1}^n c_i$$

Se todos os métodos têm complexidade igual, então $WMC = n$, ou seja, igual ao número de métodos.

4.4.10 Referência à Subclasses

Essa métrica tem como objetivo localizar se existem superclasses que dependem de suas subclasses. Essa métrica não foi localizada na literatura utilizada para identificar as demais métricas descritas neste capítulo, porém, será implementada na ferramenta de extração de métricas, por se tratar de uma diretriz de qualidade aplicada ao paradigma orientado a objetos, conforme descrito em [ROC 01], *os objetos perto da raiz da árvore hierárquica não devem depender de objetos inferiores.*

O valor ideal para essa métrica é zero, ou seja, não existir superclasses referenciando subclasses.

Em [PRE 95a] esse princípio é reforçado quando é descrito que não faz sentido que uma superclasse dependa de uma subclasse.

Cálculo:

Para cada classe da especificação que é especializada, busca-se na superclasse referência da subclasse, podendo ser através de tipo de atributo, tipo de parâmetro de método, tipo do retorno de método (caso seja um retorno do tipo de uma classe), variável de método ou por invocação de mensagem de outra classe.

4.5 Métrica de Coesão

Uma classe que possui um conjunto de características que contribuem para abstração de tipo representada pela classe, é dita como alta coesão. Porém, uma classe que apresenta um conjunto de características incoerentes é dita pouco coesa.

Determinar a coesão de uma classe é possível pelo exame do grau em que *o conjunto propriedades que ela possui é parte do problema ou do domínio do projeto* ([WHI 97]¹⁵ apud [PRE 02]).

4.5.1 Falta de Coesão nos Métodos – LCOM (*Lack of Cohesion in Methods*)

Essa métrica foi proposta por Chidamber e Kemerer [CHI 93], com o objetivo de medir a falta de coesão entre os métodos de uma classe.

Pressman [PRE 02] descreveu LCOM como *o número de métodos que tem acesso a um ou mais dos mesmos atributos*. LCOM será igual a zero, caso o par de métodos não acesse pelo menos uma vez o mesmo atributo, caso contrário será diferente de zero. O autor acrescentou que:

Em geral, valores altos de LCOM implicam que a classe pode ser melhor projetada quebrando-a em duas ou mais classes separadas. Apesar de haver casos nos quais um alto valor de LCOM é justificável, manter a coesão alta, isto é, manter o LCOM baixo é desejável.

Métodos em uma classe devem estar coesos. Falta de coesão indica que classes provavelmente podem ser divididas em duas ou mais classes. Qualquer medida de disparidade de método ajuda a identificar falha no projeto de classe. Baixa coesão aumenta a complexidade, aumentando a probabilidade de erros durante o desenvolvimento [CHI 93].

Sobre o LCOM, Muller [MUL 02] disse que *o maior problema com essa medida de coesão é que requer pelo menos a implementação parcial dos métodos do sistema*. E o autor acrescenta que *é importante compreendermos que coesão é um aspecto central*

¹⁵ Whitmire, S., *Object-Oriented Design Measurement*, Wiley , 1997.

da capacidade de reutilização. Quanto mais coesas as suas classes, é mais provável que você consiga reutilizar as classes em circunstâncias diferentes.

Como neste trabalho as especificações avaliadas são elaboradas no ambiente SEA, que possui o diagrama de corpo de métodos (DCM), é possível extrair o valor dessa métrica ainda na fase de projeto.

A métrica LCOM procura semelhança entre pares de métodos, obtendo todos os possíveis pares de métodos da classe. Para cada par de métodos em que não são identificados atributos comuns, a semelhança é zero. Caso exista algum atributo comum entre o par de métodos, existe alguma semelhança. Chidamber e Kemerer [CHI 93] definem LCOM como a contagem de par de métodos que tem semelhança igual a zero menos a contagem de par de métodos com semelhança diferente de zero.

Neste trabalho para verificar se um método faz referência a um atributo da classe procede-se da seguinte forma:

- Receber um valor ou ser atribuído.
- Receber o resultado de uma mensagem
- Ser passado como parâmetro de método
- Ser retornado por um método
- Ser o objeto destino de uma mensagem

Cálculo:

Considerando a Classe C_1 , com métodos $M_1, M_2..M_n$ e $\{I_j\}$ o conjunto de atributos usados pelo método M_i . Se todos os conjuntos de $\{I_1\}... \{I_n\}$ são \emptyset (vazio) então $P = \emptyset$.

Tendo que $P = \{(I_i, I_j) | I_i \cap I_j = \emptyset\}$ e $Q = \{(I_i, I_j) | I_i \cap I_j \neq \emptyset\}$

O valor de LCOM será:

$$\begin{aligned} LCOM &= |P| - |Q|, \quad \text{if } |P| > |Q| \\ &= 0, \quad \text{caso contrário} \end{aligned}$$

Considerações sobre a métrica

Após implementar a métrica LCOM, foi observado que a mesma não gera informações satisfatórias, visto que LCOM somente avalia o uso de atributo pelos métodos da classe.

Sobre a definição da coesão de métodos, utilizando como base métodos que utilizam as variáveis internas da classe, Page-Jones [PAG 01] não considera essa abordagem muito atrativa, descrevendo duas razões:

A primeira razão é que coesão é uma propriedade que deveria estar aparente a partir do “exterior” de uma unidade de software encapsulada. Portanto, parece errado termos de examinar as partes internas de uma classe a fim de estimarmos sua coesão. A segunda razão é que tal medição é instável e bastante dependente do desenho interno particular de método, que pode mudar durante sua existência de uma classe.

Uma alternativa já identificada, seria fazer a verificação das referências aos atributos através de seus métodos de acesso. De qualquer forma, é preciso buscar novos critérios para avaliar a coesão dos métodos em uma classe.

O resultado dessa métrica não será avaliado no estudo de caso.

4.5.2 Falta de Coesão nos Métodos – LCOM - Modificado

Em função das considerações feitas para a métrica LCOM descrita anteriormente, a mesma teve o cálculo alterado, passando a considerar também a invocação de métodos, e não somente o uso de atributos, conforme definido na métrica originalmente proposta pelo autor. Devido à alteração na forma de cálculo da métrica LCOM, esta passará a ser identificada por LCOM – Modificado, para diferenciá-la da métrica original.

Cálculo:

Ao cálculo original de LCOM foi acrescida a busca de referências aos atributos através de seus métodos de acesso, somente a um nível, por exemplo, se um método X possui uma chamada a um método mA de acesso a atributo A, é contabilizado que o método X acessa o atributo A.

4.6 Métricas de Polimorfismo

O uso de polimorfismo tem aspectos favoráveis e desfavoráveis. Um ponto favorável ao uso de polimorfismo é a redução da complexidade de projeto, pois permite que mensagens com a mesma assinatura sejam tratadas por um conjunto de classes diferentes em uma relação de herança, reduzindo assim o número de assinaturas de métodos diferentes do projeto. Outro aspecto favorável diz respeito à reutilização de classes, visto que novas classes podem ser criadas por especialização, redefinindo as mensagens herdadas, sem interferir no funcionamento das mensagens já existentes na relação de herança. Um ponto negativo, é que um software desenvolvido com um alto grau de polimorfismo pode gerar uma maior dificuldade no entendimento das invocações das mensagens, visto que deve ser conhecida a instância do objeto que invoca a mensagem, para um posterior entendimento do resultado obtido com a mensagem [ABR 95].

A primeira métrica a ser tratada neste tópico, para avaliar o uso de polimorfismo, é o Fator de Polimorfismo, que é aplicado à especificação como um todo. Essa métrica foi primeiramente proposta por Abreu e Carapuça [ABR 94].

Embora o conceito de polimorfismo mais amplamente utilizado é *métodos com mesmas assinaturas em classes relacionadas por herança*, neste trabalho será tratado também o conjunto de métricas proposto por Benlarbi e Melo [BEN 99], que classifica o polimorfismo em três grupos descritos a seguir:

Polimorfismo puro – definição de métodos com mesmo nome, mas com assinaturas diferentes dentro do escopo da mesma classe.

Polimorfismo estático – definição de métodos com mesmo nome e com assinaturas diferentes em classes, ligadas ou não por relação de herança.

Polimorfismo dinâmico - definição de método com mesmo nome e mesma assinatura, de um método herdado e sobreescrito¹⁶ na classe avaliada.

¹⁶ O termo sobreescrito, também conhecido como redefinição (*overriding*)

No contexto deste trabalho as funções ou “funções membros do polimorfismo” são composta por *Nome*, *Assinatura*¹⁷ e *Tipo de Retorno*, podendo cada um dos três elementos anteriores mudar em cada nova declaração [BEN 99].

As métricas a serem aplicadas para cada tipo de polimorfismo, descrito anteriormente, serão tratadas nas próximas sessões.

4.6.1 Fator de Polimorfismo – PF (*Polymorphism Factor*)

Essa métrica indica, através de um índice que varia de 0 (zero) a 1 (um), o fator de polimorfismo de métodos na especificação. Ela foi primeiramente apresentada por Brito e Abreu [ABR 96]. É aplicada à especificação como um todo e visa quantificar a ausência ou presença do polimorfismo, podendo variar de 0% (não uso ou ausência total) a 100% (máximo uso ou presença máxima possível).

Conforme Harrison [HAR 98], PF é definido informalmente como:

O número de métodos que redefinem métodos herdados dividido pelo número máximo de possíveis situações polimórficas diferentes. PF é uma medida indireta da quantidade relativa de ligação dinâmica em um sistema.

Para o cálculo dessa métrica é considerado o conceito de polimorfismo dinâmico, definido anteriormente.

Cálculo:

$$PF = \frac{\sum_{i=1}^{TC} Mo(Ci)}{\sum_{i=1}^{TC} [Mn(Ci) \times DC(Ci)]}$$

¹⁷ Outras definições de assinatura de métodos:

“A assinatura define a interface para a operação: os argumentos que ela exige (número, ordem e tipo) e os valores que ela retorna (número, ordem e tipo) .” [RUM 94]

“Cada assinatura formal comprehende o nome da operação, juntamente com a lista de argumentos de dados de entrada e saída formais da operação” [PAG 01]

Onde:

C = Classe;

TC = Número total de classe do projeto em consideração;

$M_h(C_i)$ = número de métodos novos na classe C_i ;

$M_o(C_i)$ = número de métodos sobrescritos na classe C_i ;

$DC(C_i)$ = Número de descendentes de C_i .

Conforme análise feita por Harrison [HAR 98], quando na especificação avaliada não há relação de herança, a métrica apresenta uma inconsistência, pois o valor de $DC(C_i)$ é zero.

4.6.2 Polimorfismo Puro ou Sobrecarga em Classes Isoladas – OVO (*Overloading in Stand-Alone Classes*)

Medir o polimorfismo puro em uma classe é o mesmo que medir o número de métodos sobrecarregados¹⁸, ou seja, métodos com mesmo nome, mas com assinaturas diferentes, dentro do escopo da mesma classe.

Há algumas divergências de nomenclatura, quando o assunto é polimorfismo. Polimorfismo puro, também conhecido como Sobreposição, geralmente não é classificado como polimorfismo, conforme [PAG 01]:

A distinção normal entre polimorfismo e sobreposição é que o polimorfismo permite que o mesmo nome de operação seja definido diferentemente pelas diferentes classes, enquanto que a sobreposição permite que o mesmo nome de operação seja definido diferentemente diversas vezes dentro da mesma classe.

Nesse trabalho, sobreposição será o mesmo polimorfismo puro.

Na ferramenta descrita neste trabalho, não é feita a verificação do tipo de parâmetros dos métodos, pois no ambiente SEA não é possível descrever métodos com

¹⁸“O nome e os parâmetros de uma operação.” [BOO 00]

¹⁸ Neste trabalho os temos sobrecarga e sobreposição são utilizados com o mesmo significado.

mesmo número de parâmetros e com tipos de parâmetros diferentes ou somente o tipo de retorno diferente na mesma classe ou em classes relacionadas por herança. A verificação de polimorfismo puro ou estático é feita em função do número de parâmetros.

A métrica OVO levanta o número total de sobrecarga¹⁹ de métodos em uma classe.

Essa métrica tem como objetivo, conforme descrito por Veiga [VEI 99], *medir o grau de genericidade dos métodos em uma classe, contando o número de funções membros que implementam a mesma operação*. Ou seja, essa métrica irá obter qual é o número de métodos implementados na mesma classe, com mesmo nome, porém, com uma lista de argumentos e/ou tipos diferentes. Essa característica irá mostrar se a classe é flexível a diferentes tipos de entrada.

Cálculo:

Essa métrica é definida da seguinte forma:

$$OVO(C) = \sum_{f_i \in C} \text{overl}(f_i, C)$$

C = Classe

f_i = representa uma função da classe C.

$\text{Overl}(f_i, C)$ = retorna o número de vezes que a função f_i é sobrecarregada na classe C.

4.6.3 Polimorfismo Estático nos Ancestrais - SPA (*Static Polymorphism in Ancestors*)

O objetivo dessa métrica é identificar a existência do polimorfismo estático nos ancestrais para cada membro de função da classe avaliada, e não identificar o número de ocorrências de cada membro de funções polimórficas estáticas nos ancestrais. Ou seja, essa métrica irá identificar se houve ocorrência de métodos implementados com mesmo

¹⁹ Neste contexto sobrecarga é o mesmo que sobreposição.

nome, porém, com uma lista de argumentos e/ou tipos diferentes, entre a classe avaliada e as suas classes ancestrais. Quando essa métrica resulta em um valor maior que zero indica que foi usado o recurso de polimorfismo estático nos ancestrais.

Cálculo:

O polimorfismo estático nos ancestrais é identificado da seguinte forma:

$$SPA(C) = \sum_{C_i \in \text{Ancestrais}} SPoly(C_i, C)$$

onde,

$\text{Ancestrais}(C)$ é um operador que retorna um conjunto de ancestrais distintos da classe C .

$SPoly(C_i, C)$ é um operador que retorna o número de membros de funções polimórficas estáticas que aparecem entre C_i e C .

4.6.4 Polimorfismo Estático nos Descendentes – SPD (*Static Polymorphism in Descendents*)

O objetivo dessa métrica é identificar o polimorfismo estático nos descendentes para cada membro de função da classe avaliada. Assim, essa métrica irá obter o número de métodos implementados com o mesmo nome, porém, com uma lista de argumentos e/ou tipos diferentes, entre a classe avaliada e as suas classes descendentes. Neste sentido, quando essa métrica resulta um valor maior que zero, indica que foi usado o recurso de polimorfismo estático nos descendentes.

Cálculo:

Polimorfismo Estático nos descendentes é identificado da seguinte forma:

$$SPD(C) = \sum_{C_i \in \text{Descendentes}} SPoly(C_i, C)$$

onde,

$\text{Descendentes}(C)$ é um operador que retorna um conjunto de descendentes distintos da classe C .

$SPoly(C_i, C)$ é um operador que retorna o número de membros de funções polimórficas estáticas que aparecem entre C_i e C .

4.6.5 Polimorfismo Estático em Relação de Herança – SP (*Static Polymorphism*)

O polimorfismo estático em relação de herança será calculado, sendo que o conjunto de classes a ser utilizado é composto pelo conjunto de classes ascendentes e descendentes.

Cálculo:

$$SP(C) = \sum_{C_i \in (Ancestrais \cup Descendentes)(C)} SPoly(C_i, C)$$

onde,

$(Ancestrais(U) \cup Descendentes(C))$ é um operador que retorna um conjunto de todas as classes ancestrais de C mais as descendentes de C .

4.6.6 Polimorfismo Dinâmico nos Ancestrais – DPA (*Static Polymorphism in Ancestors*)

O objetivo é medir o impacto do polimorfismo dinâmico entre uma classe e suas classes ancestrais. Esta métrica mostra a quantidade de métodos que foram herdados das classes ancestrais e sobreescritos.

Cálculo:

$$DPA(C) = \sum_{C_i \in Ancestrais} DPoly(C_i, C)$$

$Ancestrais(C)$ é um operador que retorna um conjunto de ancestrais distintos da classe C .

$DPoly(C_i, C)$ = retorna o número de membros de funções polimórficas dinâmicas que aparecem em C_i e C .

Relações polimórficas dinâmicas são simétricas $\Rightarrow DPoly(C_i, C) = DPoly(C, C_i)$

4.6.7 Polimorfismo Dinâmico nos Descendentes – DPD (*Dynamic Polymorphism in Descendents*)

O objetivo é medir o impacto do polimorfismo dinâmico entre uma classe e suas classes descendentes.

Cálculo:

$$DPD(C) = \sum_{C_i \in Descendentes} DPoly(C_i, C)$$

Descendentes (C) é um operador que retorna um conjunto de descendentes distintos da classe C.

$Dpoly(C_i, C)$ = retorna o número de membros de funções polimórficas dinâmicas que aparecem C_i e C.

4.6.8 Polimorfismo Dinâmico – DP (*Dynamic Polymorphism*)

O polimorfismo dinâmico será calculado, sendo que o conjunto de classes a ser utilizado é composto pelo conjunto de classes ascendentes e descendentes.

Cálculo:

$$DPA(C) = \sum_{C_i \in (Ancestrais \cup Descendentes)(C)} DPoly(C_i, C)$$

4.6.9 Polimorfismo em relação sem herança – NIP (*Polymorphism in non-inheritance relations*)

Essa métrica está descrita no grupo das métricas de polimorfismo, mas o seu objetivo é identificar nomes iguais de métodos em classes sem relação de herança. Tais métodos podem ter ou não a mesma assinatura.

Conforme Benlarbi e Melo [BEM 99], o NIP não é um polimorfismo real, simplesmente age como um indicador potencial para confusão humana. Métodos com nomes iguais em classes sem relação de herança, podem levar à dificuldade de manutenção. Essa dificuldade é gerada, em função da confusão que pode ocorrer quando o desenvolvedor ler o código fonte.

Neste trabalho, esta métrica foi implementada e classificada como polimorfismo para manter a mesma classificação do autor da métrica. A identificação de métodos com nomes iguais, em classes não relacionadas por herança, pode ser tratado como uma indicação de maior dificuldade para a manutenção dependendo do tipo de negócio que está sendo implementado. Por exemplo, quando se trabalha com interface gráfica, segundo VEIGA [VEI 99] é comum ter métodos com mesmos nomes (em classes sem relação de herança).

Conforme Rumbaugh [RUM 94], quando existem métodos em várias classes que executam a mesma operação, é importante que esses métodos possuam a mesma assinatura. E quando os métodos são destinados à operações semanticamente diferentes, é preferível evitar utilizar o mesmo nome de métodos.

Cálculo:

O polimorfismo em relação sem herança é definido da seguinte forma :

$$NIP(C) = \sum_{C_i \in Outras(C)} SPoly(C_i, C) + DPoly(C_i, C)$$

Sendo Outras(c) um operador que retorna o conjunto de classes distintas que não são ascendentes nem descendentes da classe C.

Para o cálculo do NIP a expressão DPoly(C_i, C) retorna o número de métodos com mesmo nome e assinatura que aparecem entre C_i e C, não verificando se o método é sobreescrito, pois trata-se de classes não relacionadas por herança.

4.7 Métricas versus Elementos de Especificação

As métricas apresentadas neste trabalho utilizam elementos de especificação, tais como classe, atributos, métodos, relação de herança. Na tabela 4.2, são apresentadas todas as métricas descritas neste capítulo, os elementos utilizados para extração das métricas e a forma como foi implementada na ferramenta MET.

Métrica	Elemento					Herança					
	Classe	Algoritmo de Método		Atributo	Assinatura método	Variáveis de Método	Classes Sem Relação de Herança	Classes com Relação de Herança		Herança de Atributos	Herança de Métodos
		Instância	Classe					Classes Ancestrais	Classes Descendentes		
Fator de Acoplamento – COF	X	X	X	X	X	X	X				
Acoplamento entre Classes de Objetos – CBO	X	X	X	X	X	X	X	X	X		
Acoplamento Classe-Atributo por importação nos ancestrais – ACAIC	X			X				X			
Acoplamento Classe-Atributo por exportação nos descendentes – DCAEC	X			X					X		
Acoplamento por Classe-Atributo por importação entre classes sem relação de herança – OCAIC	X			X			X				
Acoplamento por Classe-Atributo por exportação entre classes sem relação de herança – OCAEC	X			X			X				
Acoplamento Classe-Método por importação nos ancestrais – ACMIC	X							X			
Acoplamento Classe-Método por exportação nos descendentes – DCMEC	X								X		
Acoplamento Classe-Método por importação entre classes sem relação	X						X				

Métrica	Elemento					Herança					
	Classe	Algoritmo de Método		Atributo	Assinatura método	Variáveis de Método	Classes Sem Relação de Herança	Classes com Relação de Herança		Herança de Atributos	Herança de Métodos
		Instância	Classe					Classes Ancestrais	Classes Descendentes		
de herança– OCMIC											
Acoplamento Classe-Método por exportação entre classes sem relação de herança – OCMEC	X						X				
Acoplamento Método-Método por importação nos ancestrais – AMMIC		X	X					X			
Acoplamento Método-Método por exportação nos descendentes – DMMEC		X	X						X		
Acoplamento Método-Método por importação entre classes sem relação de herança – OMMIC		X	X				X				
Acoplamento Método-Método por exportação entre classes sem relação de herança – OMMEC		X	X				X				
Fator de Polimorfismo – POF	X				X					X	X
Sobrecarga em Classes Isoladas OVO	X				X						
Polimorfismo Estático em Relação de Herança – SP	X				X			X	X	X	
Polimorfismo Estático nos Ancestrais – SPA	X				X			X		X	
Polimorfismo Estático nos Descendentes – SPD	X				X				X	X	
Polimorfismo Dinâmico - DP	X				X			X	X	X	X
Polimorfismo Dinâmico nos Ancestrais – DPA	X				X			X		X	X

Métrica	Elemento					Herança						
	Classe	Algoritmo de Método		Atributo	Assinatura método	Variáveis de Método	Classes Sem Relação de Herança	Classes com Relação de Herança		Herança de Atributos	Herança de Métodos	Sobrescrição Métodos
		Instância	Classe					Classes Ancestrais	Classes Descendentes			
Referência à Subclasses	X								X			
LCOM – Modificado	X	X	X	X								

TABELA 4.2 – Métricas versus Elementos e Relacionamentos

4.8 Conclusão

Neste capítulo foi apresentada uma coletânea de métricas levantadas na literatura, sendo que métrica LCOM teve o seu cálculo alterado, sendo então denominado de LCOM – Modificado, para diferenciá-la da métrica original. É importante ressaltar que este trabalho não tem como objetivo validar a forma de cálculo das métricas ou a sua eficácia e sim, identificá-las e automatizar sua obtenção através de uma ferramenta que faça a extração das métricas nas fases de análise e projeto.

As métricas de especificação de software orientado a objetos, descritas neste capítulo, têm como objetivo auxiliar a avaliação de projeto e não funcionam com o propósito de fornecer um diagnóstico da qualidade e sim como indicadores. O ideal seria se o resultado de uma métrica apresentasse um diagnóstico exato da qualidade, porém, os resultados das métricas devem ser analisados pelo desenvolvedor, considerando aspectos específicos do projeto, o que pode ser complexo para definir. Este é um dos motivos da pouca utilização das métricas. Um outro fator que contribui para a pouca utilização é relacionado à baixa disponibilidade de ferramentas que dêem suporte a um processo rápido de metrificação.

As métricas podem ser aplicadas nas fases de análise e projeto, o que traz vantagens como a não necessidade de aguardar que o código fonte seja gerado e ainda uma redução de esforço e custo considerável, em função das possíveis mudanças originadas em função da interpretação dos resultados das métricas.

Existe uma grande preocupação quanto à validade das métricas, pois não basta que uma métrica seja proposta é preciso que ela seja validada.

No próximo capítulo serão apresentados alguns trabalhos que tratam da automação da extração de métricas e também o protótipo de uma ferramenta, desenvolvido neste trabalho, que automatiza a extração das métricas que foram descritas nesse capítulo.

5 FERRAMENTA DE EXTRAÇÃO DE MÉTRICAS

5.1 *Introdução*

O uso de ferramentas para apoiar a avaliação de software é de inegável importância, pois é bastante trabalhoso e, em certas situações, técnica ou economicamente inviável fazer a extração manual de dados a partir de modelos de projetos orientados a objetos ou programas fonte – principalmente à medida que a complexidade do software aumenta.

Este capítulo descreve trabalhos que tratam de suporte ferramental para auxiliar a avaliação de software orientado a objetos. Após a apresentação dessas ferramentas é apresentado o protótipo da ferramenta desenvolvida neste trabalho, que na verdade é composto por duas ferramentas, uma de extração de métricas, que foi incorporada ao ambiente SEA e uma outra utilizada para a visualização gráfica dos resultados das métricas.

5.2 *Trabalhos Correlatos*

Os trabalhos identificados apresentam automação de algumas operações de extração de dados. Esses dados são usados pelas ferramentas sob dois aspectos:

- para verificação de diretriz de qualidade, por exemplo, *uma classe base não depender de suas classes derivadas*, ou de atendimento de *Patterns* e *AntiPatterns*.
- para extração de métricas, que tem como objetivo apresentar um resultado que quantifica algumas características, tais como acoplamento, polimorfismo e coesão.

A – OOPDTool

No trabalho descrito por Correa [COR 00], é apresentada uma ferramenta OOPDTool, que dá suporte à reengenharia de sistemas legados desenvolvidos com o

paradigma de orientação a objetos, e permite a avaliação de projetos ainda em desenvolvimento, com o objetivo de identificar pontos no sistema que devem ser modificados, de modo a tornar o sistema mais flexível e reutilizável. Esta ferramenta é integrada a uma ferramenta CASE - Rose (*Rational Software Corporation*). A arquitetura da ferramenta OOPDTool é composta por quatro módulos principais:

Extrator de Projeto - apesar de na ferramenta Rose já existir módulo de engenharia reversa, para linguagens como C+ e Java, que extraem informações estruturais do sistema, os autores de OOPDTool optaram por desenvolver um módulo de engenharia reversa, para que, além de informações estruturais, fossem extraídas informações relativas à implementação dos métodos das classes avaliadas. Neste módulo do sistema é armazenado para cada método, um diagrama de colaboração em apenas um nível de chamada, ou seja, são extraídas de cada método apenas as mensagens diretamente disparadas a partir deste, armazenando as informações obtidas, através do extrator de projeto na ferramenta Rose.

Gerador de Fatos - gera uma base de dados em Prolog, que representa o conjunto de fatos capturados do modelo de projeto no Rose. Nesse conjunto de fatos são descritas as classes, interfaces, tipos, atributos, operações e relacionamentos.

Captura de conhecimento - este módulo captura o conhecimento sobre boas e más maneiras de construção de projetos orientados a objeto, formando assim uma base de conhecimento sobre heurísticas e *Design Patterns*²⁰ e *AntiPatterns*²¹. A base de conhecimento serve tanto como base de consulta para o projetista como base para detecção automática de design patterns e antipatterns em modelo de projetos orientados a objetos.

²⁰ Design Patterns – conforme Christopher Alexander [AIS 77] Apud [GAM 00]: “cada padrão descreve um problema no nosso ambiente e o núcleo da sua solução, de tal forma que você possa usar esta solução mais de um milhão de vezes, sem nunca fazê-lo da mesma maneira”.

²¹ Antipatterns – conforme Alexandre Correa [COR 00]: “Um AntiPattern descreve uma solução para um problema recorrente que traz consequências negativas para o projeto. Um AntiPattern pode resultar da falta de conhecimento de uma solução melhor, ou ainda da aplicação de um design Pattern no contexto incorreto”.

Analizador - é feita a análise da base de fatos do projeto em estudo, utilizando a base de conhecimento. Neste módulo são detectados os fragmentos do projeto que podem gerar problemas referentes à manutenção, à reutilização e à flexibilidade, sendo apresentadas sugestões de melhoria.

B – JMetrics

Outro trabalho que trata de uma ferramenta para auxiliar no suporte de avaliação de software orientado a objetos é [VEI 99], que apresenta a ferramenta JMetrics – Java Metrics Analyser, que extrai métricas de código em linguagem Java. Essa ferramenta foi desenvolvida na linguagem Java, é *Free*²² e tem como principais objetivos [VEI 99]:

- Extrair um conjunto de métricas de projeto, pacotes, métodos e atributos,
- Possibilitar a extração de métricas a partir de bibliotecas de classes, utilizadas no projeto. Classes que não pertencem ao projeto, mas que foram reescritas por algumas das classes do projeto são também analisadas pela ferramenta. Por isso faz-se necessário a inclusão de um arquivo com extensão “jar” ou “zip” ou a pasta que contém estas classes.
- Gerar planilha com as métricas resultantes.

A relação de métricas apresentada para a ferramenta JMetrics foi descrita com base nos trabalhos [BRI 02] [VEI 99]. Neste último trabalho é apresentado um conjunto de métricas intermediárias, ou seja, utilizadas para calcular outras métricas. Porém, não está claro se estes resultados são disponibilizados pela ferramenta.

C – Ferramenta para Aplicação de Métricas em Projeto Orientado a Objetos

O trabalho [AND 98] trata da redução de complexidade de projetos orientados a objetos, com aplicação simultânea de princípios, diretrizes e métricas. E, como parte deste trabalho, foi automatizada a extração das métricas.

A extração das métricas é realizada sob modelos contidos na ferramenta *CASE Rose*, implementada com a linguagem nativa da ferramenta *CASE (Rose Scripting*

²² A ferramenta JMetrics pode ser encontrada no endereço <http://jmetric.it.swin.edu.au/>.

Language). Os resultados gerados pelos *scripts* podem ser formatados, como relatórios ou arquivo texto de entrada para MS Excel.

As métricas extraídas a partir dos *scripts* estão relacionadas na tabela 5.2.

D – Ferramenta para cálculo de métrica em softwares orientados a objetos codificados em Object-Pascal

Essa ferramenta foi descrita em [SEI 02] e tem como objetivo a extração de métricas a partir do código fonte Object-Pascal – Delphi. A ferramenta foi desenvolvida em Delphi e as métricas implementadas por ela estão descritas na tabela 5.2.

A tabela 5.1 apresenta e compara as ferramentas levantadas com a ferramenta MET.

Ferramenta	Utiliza	Análise é feita sobre	Linguagem	Visualização dos Resultados
OOPDTool (1)	Heurísticas, <i>Design Patterns</i> e <i>AntiPatterns</i>	Código Fonte (C ++ e Java) Modelos de Projetos – (Rose)	Não disponível em [COR 00]	– Relatório texto
JMetrics (2)	Métricas	Código fontes (Java)	Java	– Gráficos e tabelas – Exporta arquivo texto
Ferramenta para Aplicação de Métricas em Projeto Orientado a Objetos (3)	Métricas	Modelo de Classes (Rose)	<i>Rose Scripting Language</i>	– Relatório – Excel
Ferramenta para cálculo de métrica em softwares orientados a objetos codificados em Object-Pascal (4)	Métricas	Código fonte (Object Pascal)	<i>Delphi</i>	– Gráficos – Tabelas
MET	Métricas	Modelos de especificação OO do ambiente SEA	<i>Smalltalk</i>	– Relatório – Exporta arquivo de dados para a Ferramenta de Visualização dos Resultados

TABELA 5.1 – Ferramentas de Suporte a Avaliação de Software OO.

Três das ferramentas descritas acima necessitam do código fonte para fazer extração das métricas. Nesse contexto, a avaliação do software só ocorre na fase final do desenvolvimento do software. Assim, as adequações necessárias de serem realizadas após a interpretação das métricas, são mais dispendiosas em relação a tempo e recurso. Mesmo que seja usado um gerador automático de código, a dificuldade da adequação persiste, pois na maioria das vezes o código gerado automaticamente não inclui

algoritmos completos, mas apenas esqueletos de algoritmo ou algoritmos óbvios, como de acesso a atributos, o que possibilita uma análise apenas parcial de potenciais problemas de desenvolvimento antes da implementação.

A tabela 5.2 apresenta as métricas que cada ferramenta implementa, comparando com a ferramenta MET.

Métricas	Ferramentas ²³			
	(B)	(C)	(D)	MET
Fator de Acoplamento – COF				X
Acoplamento entre Classes de Objetos – CBO			X	X
Acoplamento Classe-Atributo por importação nos ancestrais– ACAIC	X			X
Acoplamento Classe-Atributo por exportação nos descendentes – DCAEC	X			X
Acoplamento por Classe-Atributo por importação entre classes sem relação de herança – OCAIC	X			X
Acoplamento por Classe-Atributo por exportação entre classes sem relação de herança – OCAEC	X			X
Acoplamento Classe-Método por importação nos ancestrais– ACMIC	X			X
Acoplamento Classe-Método por exportação nos descendentes – DCMEC	X			X
Acoplamento Classe-Método por importação entre classes sem relação de herança– OCMIC	X			X
Acoplamento Classe-Método por exportação entre classes sem relação de herança – OCMEC	X			X
Acoplamento Método-Método por importação nos ancestrais – AMMIC				X
Acoplamento Método-Método por exportação nos descendentes – DMMEC				X
Acoplamento Método-Método por importação entre classes sem relação de herança – OMMIC				X
Acoplamento Método-Método por exportação entre classes sem relação de herança – OMMEC				X
Fator de Atributos Ocultos – AHF	X			X
Fator de Métodos Ocultos – MHF	X			X
Tamanho dos Métodos nas Classes				X
Número de Argumentos no Método				X
Número de Métodos nas Classes – NOM				X
Número de Classes imediatamente descendentes – NOC	X	X	X	X
Profundidade da Árvore de Herança – DIT	X	X	X	X
Fator de Herança de Métodos – MIF	X			X

²³ A ferramenta OOPDTool não está relacionada nesta tabela, pois não extrai métricas como as demais ferramentas.

Métricas	Ferramentas²³			
	(B)	(C)	(D)	MET
Fator de Herança de Atributos – AIF	X			X
Reação de uma Classe – RFC			X	X
Métodos Complexos por Classe - WMC			X	X
Referência à Subclasses				X
Falta de Coesão – LCOM			X	X
Falta de Coesão – LCOM - Modificado				X
Fator de Polimorfismo – POF				X
Sobrecarga em Classes Isoladas – OVO	X			X
Polimorfismo Estático nos Ancestrais – SPA	X			X
Polimorfismo Estático nos Descendentes – SPD	X			X
Polimorfismo Estático em Relação de Herança – SP	X			X
Polimorfismo Dinâmico nos Ancestrais – DPA	X			X
Polimorfismo Dinâmico nos Descendentes – DPD	X			X
Polimorfismo Dinâmico - DP	X			X
Polimorfismo em Relação Sem Herança – NIP	X			X
Numero de Redefinição de Métodos		X		
Número de Métodos Disponíveis nas classes			X	X
Número de Métodos Sobrescritos			X	X
Número de Métodos Herdados			X	X
Número de Métodos Adicionados nas Classes			X	X
Número de Operações e Atributos por Classe		X		X
Número de Pais por Classe		X		
Média de Métodos por Classe				X
Média de Métodos Públicos por Classe			X	
Média de Atributos por Classe			X	
Quantidade de Atributos	X		X	X
Tamanho Médio dos Métodos			X	
Percentual de Comentários			X	
Quantidade de Métodos de Classe			X	
Quantidade de Métodos de Instância			X	X
Índice de Especialização			X	
Total de Atributos Privados	X			
Total de Métodos Privados	X			
Quantidade de Classes			X	X
Quantidade de Classes Abstratas				X
Quantidade de Classes Concretas				X
Quantidade de Atributos Definidos Diretamente na Classe				X

Métricas	Ferramentas ²³			
	(B)	(C)	(D)	MET
Quantidade de Atributos Herdados pela Classe				X
Quantidade de Atributos Definidos na Especificação				X
Quantidade de Atributos Herdados na Especificação				X
Quantidade de Atributos Disponíveis na Especificação				X
Quantidade de Métodos Definidos na Classe				X
Quantidade de Métodos Disponíveis na Classe (Definidos mais Herdados)	X			X
Quantidade de Métodos Herdados na Classe (Herdados menos Sobrescritos)				X
Quantidade de Métodos Ocultos na Classe				X
Quantidade de Métodos Visíveis na Classe				X
Quantidade de Métodos Sobrescritos na Classe				X
Quantidade de Métodos <i>Templates</i> na Classe				X
Quantidade de Métodos Abstratos na Classe				X
Quantidade de Métodos Regulares na Classe				X
Quantidade de Métodos Definidos na Especificação				X
Quantidade de Métodos Disponíveis na Especificação (definidos mais herdados)				X
Quantidade de Métodos Herdados na Especificação (herdados menos sobrescritos)				X
Quantidade de Métodos Ocultos na Especificação				X
Quantidade de Métodos Visíveis na Especificação				X
Quantidade de Métodos Sobrescritos na Especificação				X
Quantidade de Métodos <i>Templates</i> na Especificação				X
Quantidade de Métodos Abstratos na Especificação				X
Quantidade de Métodos Regulares na Especificação				X

TABELA 5.2 – Métricas versus Ferramentas de Extração de Métricas

5.3 Ferramenta de Extração de Métricas

Este capítulo apresenta o protótipo da ferramenta MET (*Metrics Extraction Tool*), desenvolvida neste trabalho, que extrai, a partir de análise estática, métricas²⁴ para subsidiar a avaliação da qualidade de artefatos de software orientados a objetos. A ferramenta MET foi inserida no ambiente SEA [SIL 00] e faz uso dos modelos de especificação gerados pelo ambiente, conforme descrito no capítulo 3, e assim como o ambiente SEA, a ferramenta também foi desenvolvida utilizando a linguagem *Smalltalk*.

²⁴ As métricas implementadas na ferramenta MET foram descritas no capítulo 4.

O ambiente SEA proporciona a construção de especificação de artefatos de software, tais como, *frameworks*, componentes, interfaces de componentes, padrões de projeto e aplicações. Foi projetado para possibilitar a inclusão de novas ferramentas de apoio ao processo de desenvolvimento da especificação, garantindo assim a viabilidade da ferramenta desenvolvida neste trabalho. A figura 5.1 mostra como a ferramenta MET foi inserida no contexto do ambiente SEA.

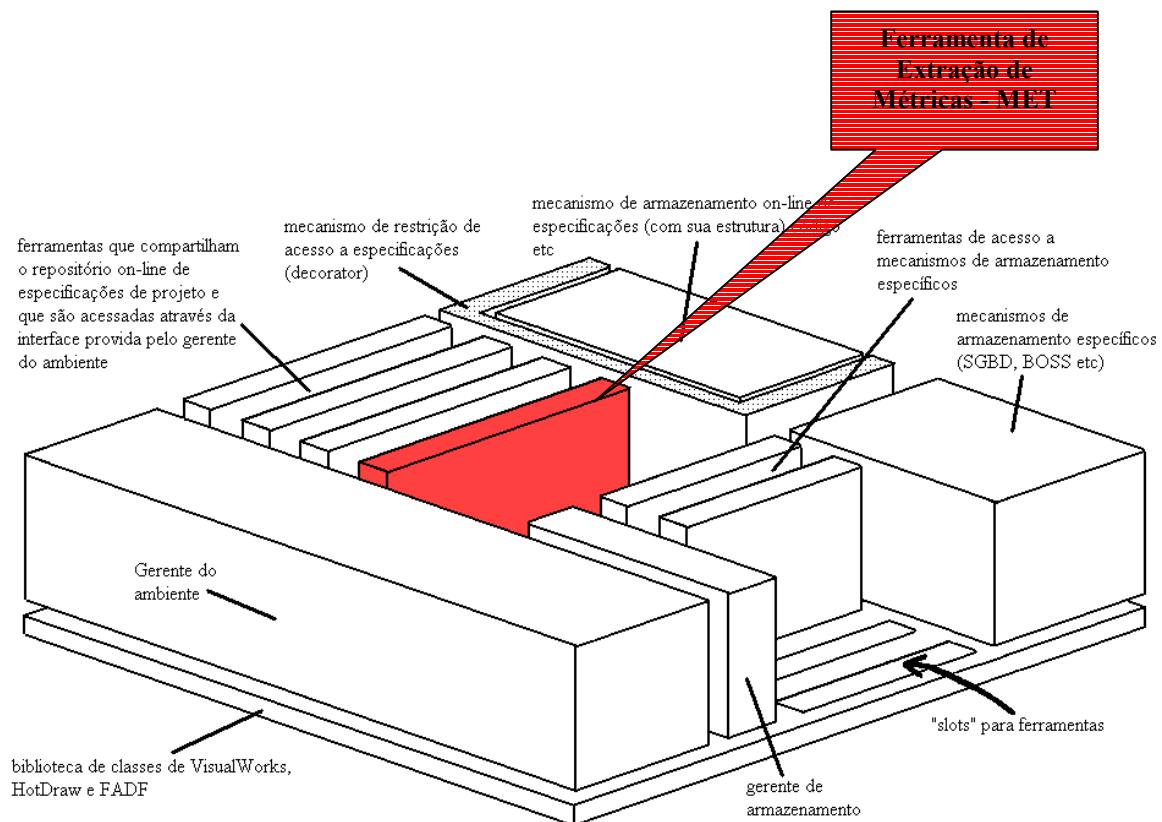


FIGURA 5.1 - Estrutura do ambiente SEA com a Ferramenta MET.

O usuário do ambiente SEA, durante a modelagem dos artefatos de software, terá disponível a ferramenta de extração de métricas de especificações orientadas a objetos, que irá obter os dados para extração das métricas no repositório de especificações do ambiente, sobre os quais serão aplicadas as métricas descritas no capítulo 4.

5.3.1 Descrição da Ferramenta

A ferramenta extrai dados de especificação de artefatos de software desenvolvidos a partir do paradigma de orientação a objetos, tais como, *frameworks*, componentes, padrões de projeto e aplicações. Os artefatos podem ser submetidos à ferramenta a qualquer momento de sua elaboração, conforme apresentado na figura 5.2.

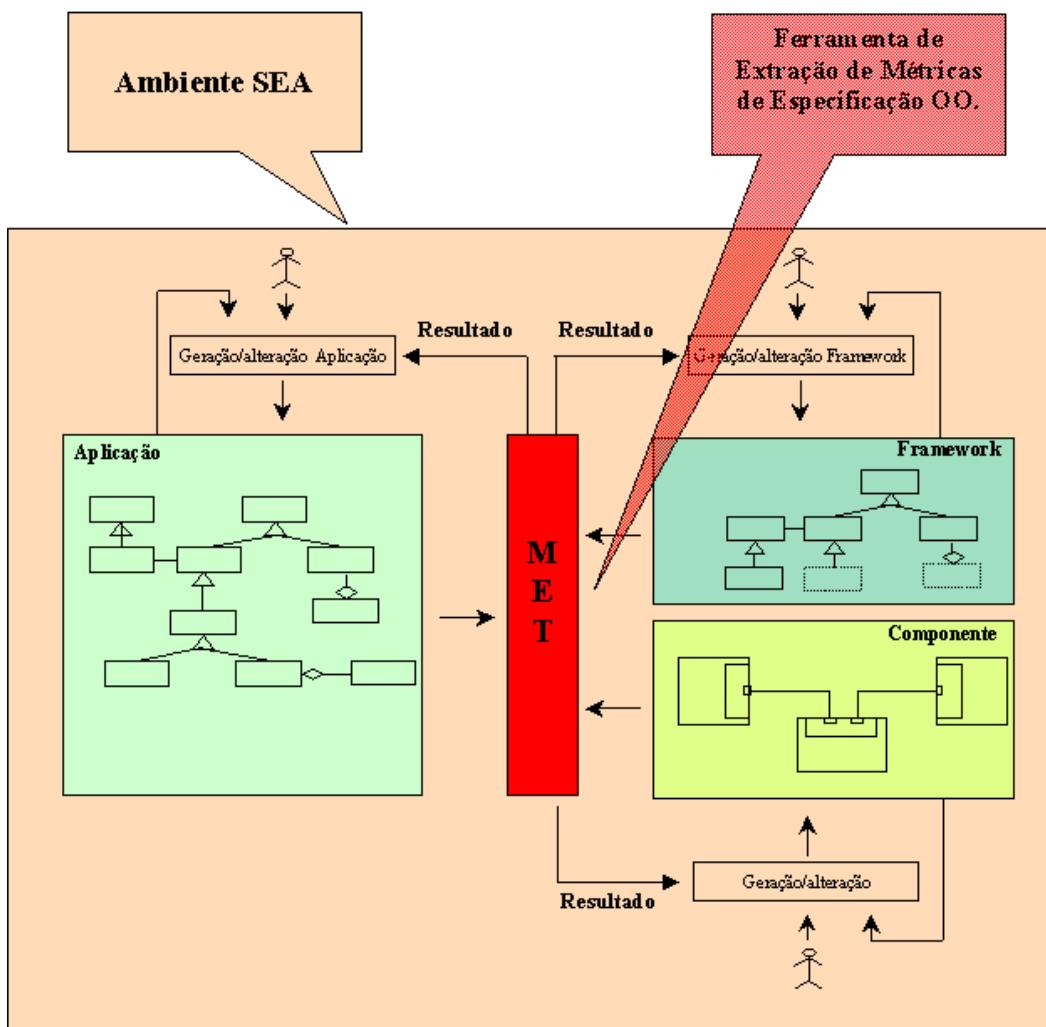


FIGURA 5.2 – Ferramenta de Extração de Métricas de Especificações Orientadas a Objetos (MET)

Após o modelo ter sido submetido à ferramenta MET, o usuário receberá os resultados das métricas. Esses resultados têm como propósito fornecer valores quantitativos e não qualitativos. Dessa forma, os valores não devem ser tomados como imperativos e sim como indicadores, sendo necessária uma interpretação por parte do

desenvolvedor. Caso o desenvolvedor verifique a necessidade de alterar o modelo, as mudanças podem ser realizadas e o modelo deverá novamente submetido à ferramenta.

Dessa forma, a ferramenta MET dá suporte para o diagnóstico, ainda na fase de análise e projeto, de construções que possam comprometer tanto a finalização do projeto, quanto as suas possíveis extensões. Problemas que somente apareceriam numa fase final do projeto podem ser detectados e modificados, ainda nas fases iniciais do projeto, reduzindo assim, o custo e o esforço de adequação.

A ferramenta MET extrai métricas, levando em consideração aspectos, como:

- Análise individual de classe, como por exemplo, o número de métodos, o tamanho da interface, o número de atributos ocultos.
- Análise de especificação dos métodos, como por exemplo, o número de linhas do método. Essa análise é possível de ser realizada ainda na fase de projeto, em virtude do ambiente SEA possibilitar modelagem do algoritmo do método, através do diagrama de corpo do método, proposto por Silva [SIL 00], como extensão da linguagem UML.
- Análise dos relacionamentos existentes entre as classes, tais como o uso de recursos de herança, polimorfismo, acoplamento, coesão e encapsulamento.

A ferramenta extrai métricas referentes à especificação em edição no ambiente SEA. Abaixo serão descritos os dados utilizados para extração das métricas:

Especificação

- Identificação;
- Lista de classes;

Classes

- Identificação;
- Lista de atributos definidos na própria classe e seus respectivos tipos;
- Classificação (abstrata ou concreta);
- Origem (da própria especificação ou externa);
- Superclasse (caso exista);
- Lista de métodos definidos na própria classe;

Atributos

- Identificação;
- Tipo;

Métodos

- Identificação;
- Lista de parâmetros e seus respectivos tipos;

- Tipo de retorno;
- Classificação (abstrato, *template* ou regular);
- Natureza do método (de instância ou de classe);
- Classe (identificação da classe onde foi definido);
- Sobrescrito (se sobrepõe algum método das superclasses);
- Número de *statements* executáveis, especificados no corpo de método (descreve o algoritmo do método em questão);
- Lista de variáveis temporárias do corpo de método e seus respectivos tipos;

Parâmetros de métodos

- Identificação;
- Tipo;
- Origem (da própria especificação ou externa).

Variáveis temporárias de métodos

- Identificação;
- Tipo;
- Origem (da própria especificação ou externa).

Mensagens

- Identificação;
- Objeto destino (objeto que implementa o método invocado);
- Método (método invocado do objeto destino).

5.3.2 Especificação da Ferramenta

Serão apresentados nesta seção, o diagrama de casos de uso e o diagrama de classes da ferramenta MET. Alguns exemplos de diagramas de seqüência são descritos no anexo 2.

Diagrama de Casos de Uso

A figura 5.3 ilustra os casos de uso para a ferramenta MET.

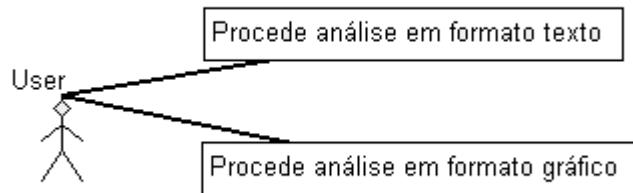


FIGURA 5.3 – Diagrama de Caso de Uso – MET

Diagrama de Classes

A inserção de uma nova ferramenta no ambiente SEA se dá em função da extensão das funcionalidades do *framework* OCEAN. Para isto, todas as novas ferramentas devem ser subclasses da classe *OceanTool*.

No diagrama de classes da figura 5.4 pode ser observado o relacionamento da ferramenta MET com o *framework* *Ocean*. As classes do diagrama na cor preta representam as classes da ferramenta (MET) implementada neste trabalho e as classes na cor vermelha representam as classes do *framework* *Ocean*.

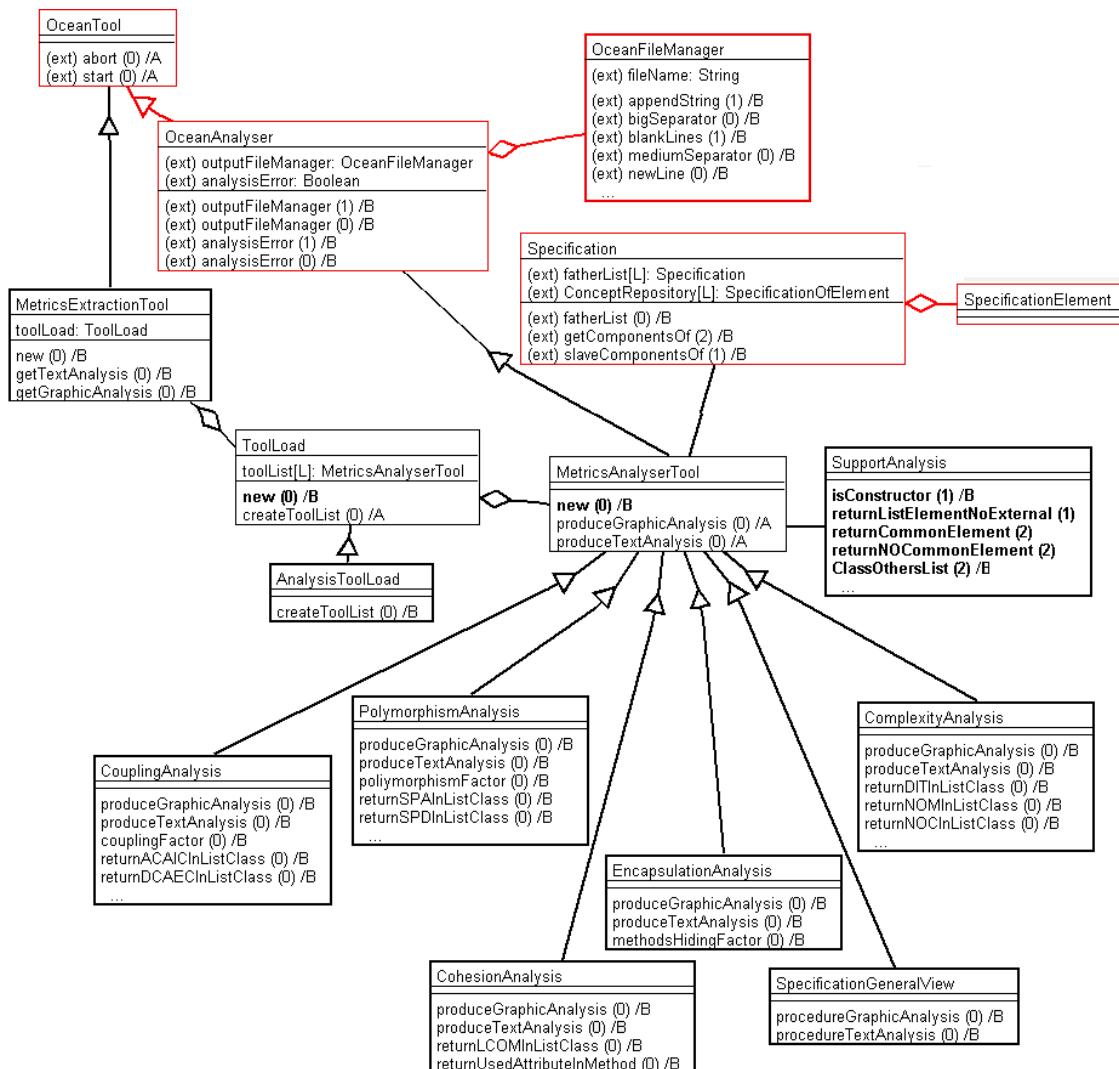


FIGURA 5.4 – Diagrama de Classes – MET

A classe *OceanFileManager* implementa serviços de escrita em arquivos de saída das ferramentas, definidas a partir do *OceanTool*. A classe *OceanTool* define as funcionalidades genéricas das ferramentas desenvolvidas sob o *framework*. Por isso, todas as ferramentas a serem inseridas no ambiente SEA devem ser subclasses de *OceanTool*. A classe *OceanAnalyser* é uma especialização da classe *OceanTool* e representa a superclasse para as ferramentas de análise (como a desenvolvida neste trabalho - MET). A classe *SpecificationElement* é a superclasse de todos os elementos de especificação, tais como: classes, atributos, métodos, relacionamentos, etc. A classe *Specification* representa a especificação propriamente dita e agrupa os elementos de especificação (*SpecificationElement*) que compõem a especificação.

A especificação e seus elementos foram representados neste diagrama de uma forma simplificada, pois não é objetivo deste trabalho entrar nesse nível de detalhe. Maiores detalhes podem ser vistos em Silva [SIL 00].

A classe *MetricsAnalysisTool* é uma superclasse abstrata das classes concretas responsáveis pela implementação das métricas de suporte à avaliação de softwares orientados a objetos: *CouplingAnalysis*, *PolymorphismAnalysis*, *CohesionAnalysis*, *EncapsulationAnalysis*, *ComplexityAnalysis* e *SpecificationGeneralView*. Essas subclasses possuem as seguintes finalidades:

CouplingAnalysis é responsável por implementar as métricas de acoplamento. Identifica o quanto à classe analisada está acoplada à outras classes ou o quanto outras classes estão acopladas à classe analisada (considerando o relacionamento com herança e sem herança) e fator de acoplamento.

PolymorphismAnalysis é responsável por implementar as métricas de polimorfismo estático, polimorfismo dinâmico, sobrecarga em classes isoladas e métodos com mesmo nome e assinatura diferente, em classes que não possuem relação de herança.

CohesionAnalysis é responsável por implementar métrica de coesão nos métodos das classes.

EncapsulationAnalysis é responsável por implementar métrica de encapsulamento nos métodos das classes.

ComplexityAnalysis é responsável por implementar métricas de complexidade, tais como, número de métodos, profundidade da árvore hierárquica, fator de herança de métodos e fator de herança de atributos.

SpecificationGeneralView é responsável por extrair medidas que apresentam uma visão geral da especificação (busca informações sobre classes, atributos e métodos) e visão geral por classe da aplicação (busca informações sobre atributos e métodos)

Pode-se observar que a arquitetura da ferramenta MET possibilita que novos tipos de análise possam ser inseridos facilmente na ferramenta, bastando definir uma nova subclasse de *MetricsAnalyserTool* e acrescentá-la à lista de ferramentas *ToolLoad*.

A classe *ToolLoad* é superclasse abstrata da classe concreta *AnalysisToolLoad*, responsável pela criação da lista de ferramentas que implementa métricas que dão suporte para avaliar a qualidade de artefatos de software orientados a objetos. A classe *AnalysisToolLoad* é responsável por instanciar cada elemento da lista. Essa estrutura garante que outras subclases de *ToolLoad* possam ser inseridas, sem causar alterações na classe *MetricsExtractionTool*. Isso seria justificável, caso fosse necessário o instanciamento de grupos distintos de ferramentas que implementam as métricas.

A classe *MetricsExtractionTool* é responsável pela interface da ferramenta MET com o usuário. Esta classe agrupa a classe *ToolLoad* e instancia a classe *AnalysisToolLoad* que possui a lista de ferramentas desejadas.

5.3.3 Interface da Ferramenta MET

Na figura 5.5 é apresentada a interface de ferramenta MET, destinada à extração de métricas.

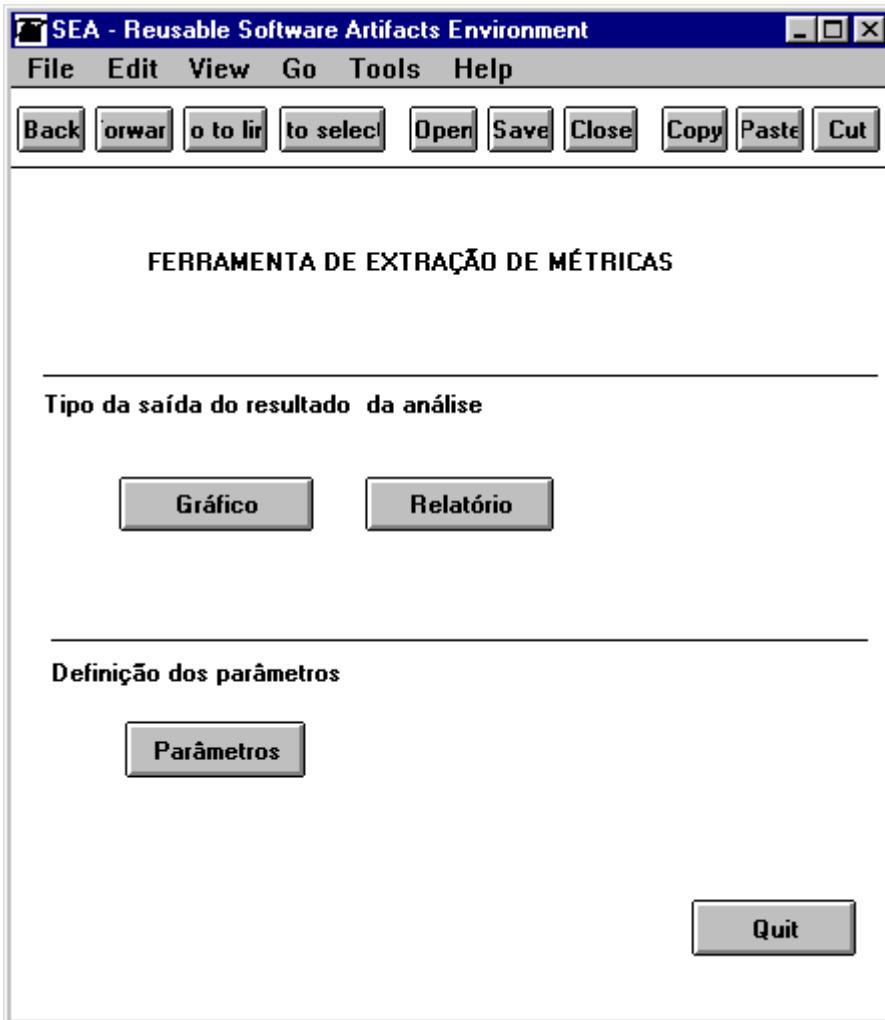


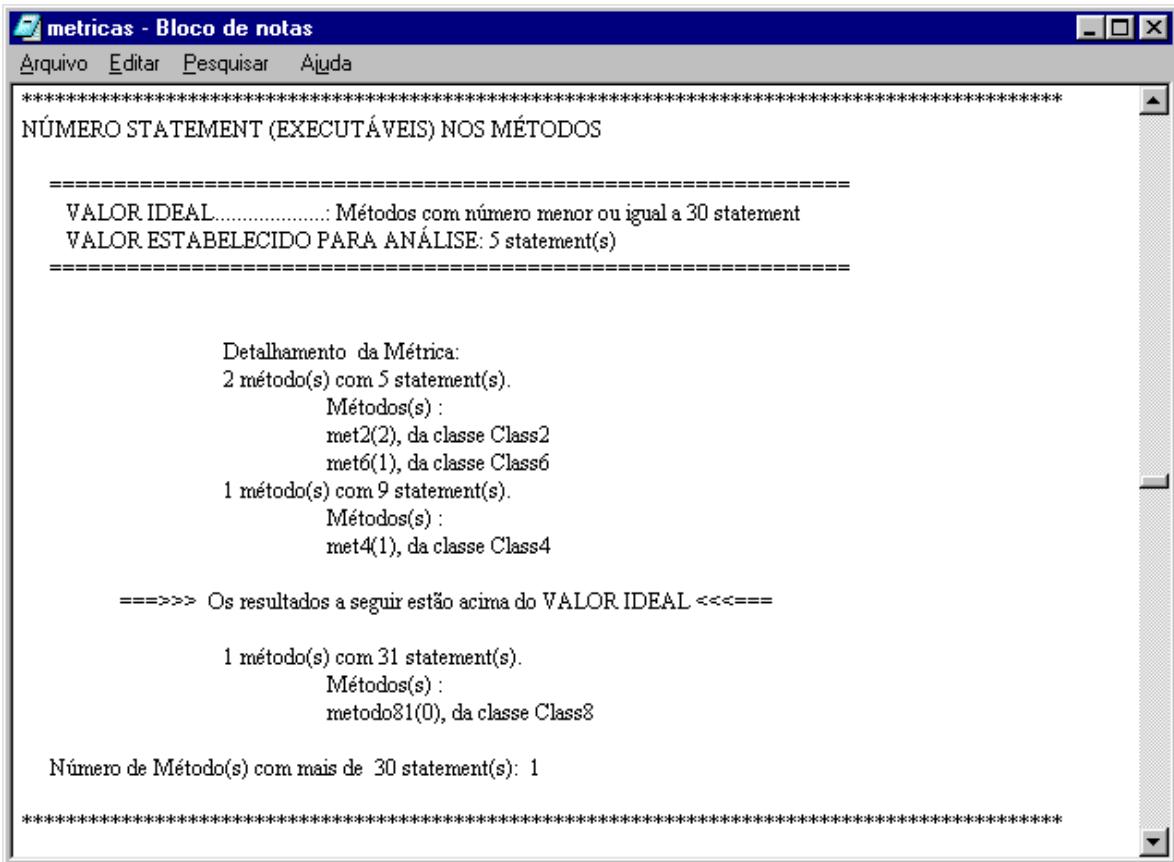
FIGURA 5.5 – Interface da ferramenta de extração das métricas

Nesta tela são apresentadas duas opções de saída do resultado:

Gráfico: gera arquivo que serve como entrada para a ferramenta *Visualização dos Resultados das Métricas*²⁵. A descrição desta ferramenta será tema do próximo assunto a ser trabalhado.

Relatório: gera o resultado das métricas em um arquivo texto. Um exemplo de um relatório emitido pela ferramenta MET é apresentado na figura 5.6.

²⁵ Ferramenta desenvolvida na linguagem *Delphi* para proporcionar a visualização do resultado das métricas em uma interface gráfica.



```

metricas - Bloco de notas
Arquivo  Editar  Pesquisar  Ajuda
=====
NÚMERO STATEMENT (EXECUTÁVEIS) NOS MÉTODOS

=====
VALOR IDEAL.....: Métodos com número menor ou igual a 30 statement
VALOR ESTABELECIDO PARA ANÁLISE: 5 statement(s)
=====

Detalhamento da Métrica:
2 método(s) com 5 statement(s).
Métodos(s) :
met2(2), da classe Class2
met6(1), da classe Class6
1 método(s) com 9 statement(s).
Métodos(s) :
met4(1), da classe Class4

====>>> Os resultados a seguir estão acima do VALOR IDEAL <<<====

1 método(s) com 31 statement(s).
Métodos(s) :
metodo81(0), da classe Class8

Número de Método(s) com mais de 30 statement(s): 1
=====
```

FIGURA 5.6 – Exemplo - saída de dados formato relatório

Para algumas métricas (número de métodos na classe, tamanho do método e número de argumentos) pode ser informado um parâmetro, conforme descrito na figura 5.7, caracterizando uma investigação sob algum patamar de interesse do usuário. Por exemplo, para a métrica “*Tamanho do Método*”, caso o usuário queira fazer a análise de métodos que possuem 20 ou mais *statements*, este valor deve ser informado no parâmetro. Estabelecer esse parâmetro implica alteração na saída do resultado da métrica:

- no relatório, serão apresentados somente os métodos com 20 ou mais *statements*;
- no gráfico, os valores que são iguais a 20 ou mais *statements* são destacados, caracterizando a faixa de investigação do usuário.

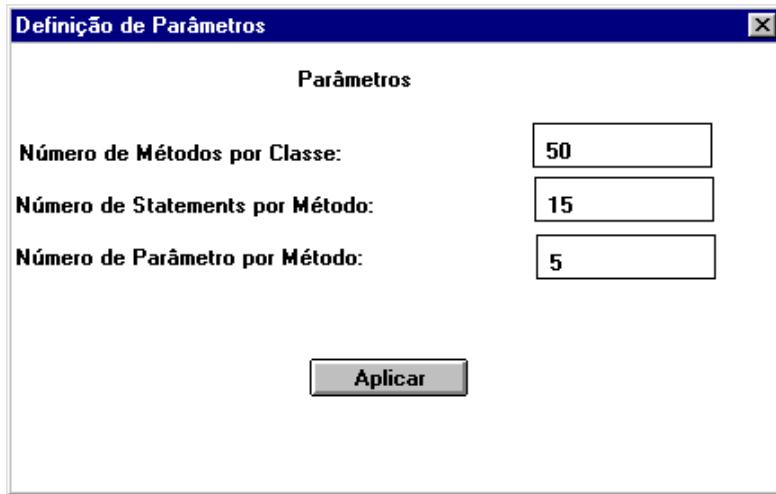


FIGURA 5.7 – Interface da tela de parâmetro

5.4 Ferramenta de Visualização dos Resultados das Métricas

Essa ferramenta foi construída com o objetivo de apresentar os resultados das métricas extraídas pela ferramenta MET, em uma interface mais amigável.

A comunicação entre as duas ferramentas ocorre através de um arquivo texto, exportado pela ferramenta MET e que é usado como entrada de dados para a ferramenta de visualização dos resultados das métricas. O formato do arquivo de exportação de dados é descrito no anexo 3.

A seguir será apresentado o diagrama de classes para a ferramenta de visualização gráfica dos resultados das métricas. A descrição dessa ferramenta não será apresentada no mesmo nível de detalhe que foi utilizado para apresentar a ferramenta MET, uma vez que essa ferramenta caracteriza-se como apoio à ferramenta MET.

5.4.1 Diagrama de Classes

No diagrama de classes da figura 5.8, a classe *GraphicViewTool* é responsável pela interface da ferramenta de visualização gráfica dos resultados das métricas com o usuário. Esta classe agrupa a classe *SpecificationView* que trata os dados da especificação como um todo, reúne também os dados das métricas previstas na ferramenta, através da classe *MetricView* e ainda as informações específicas de cada classe, através da *ClassView*.

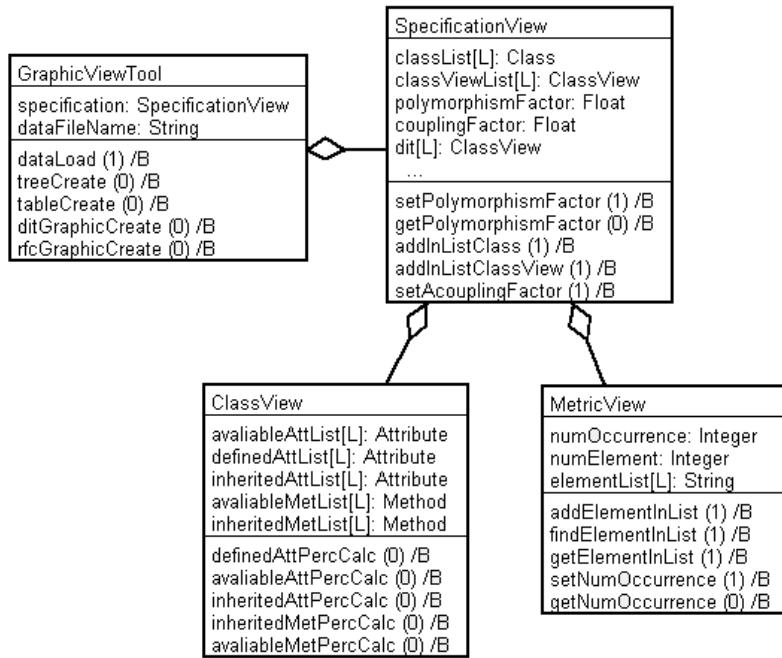


FIGURA 5.8– Diagrama de Classes da Ferramenta de Visualização Gráfica

5.4.2 Interface da Ferramenta

A interface da ferramenta de *Visualização Gráfica dos Resultados das Métricas* é composta por visualização dos resultados das métricas nos formatos: árvore hierárquica, gráficos e tabelas.

A seguir são apresentados alguns exemplos das três formas de visualização dos resultados das métricas, extraídas de especificações, que estão disponíveis na ferramenta de visualização gráfica:

Árvore hierárquica

Todas as medidas extraídas pela ferramenta MET são apresentadas na forma de visão de árvore hierárquica, sendo possível além da visualização da quantidade do resultado das métricas, uma visão dos elementos usados para chegar ao valor do resultado obtido. Por exemplo, para a métrica *Número de Argumentos nos Métodos* é possível identificar que 13 métodos não possuem parâmetros junto à lista dos métodos com essa característica.

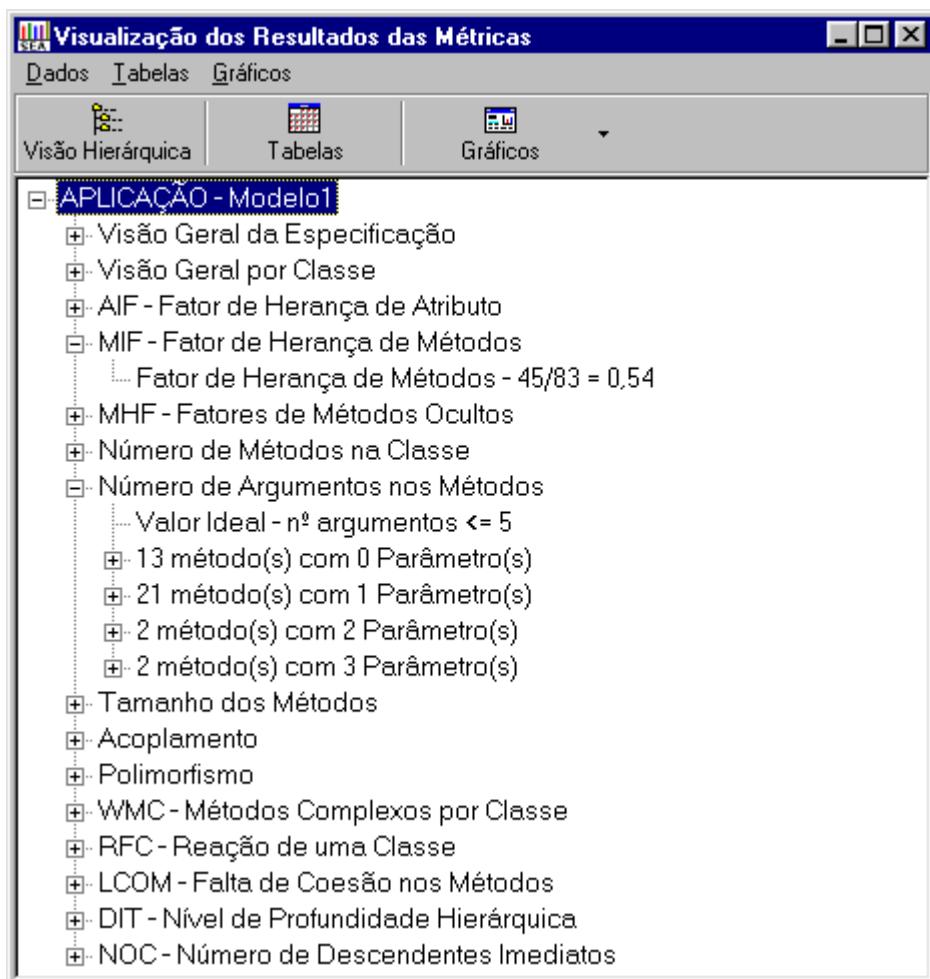


FIGURA 5.9 – Exemplo de Visão hierárquica

Nesta árvore é possível obter também uma visão geral da especificação, onde se têm as seguintes informações:

Em relação a **classes**, pode-se obter a quantidade de classes abstratas e concretas.

Em relação a **atributos**, pode ser verificada a quantidade de atributos definidos diretamente nas classes da especificação, a quantidade total de atributos herdados e quantidade total de atributos disponíveis, que representa o somatório dos atributos definidos na classe mais os atributos herdados.

Em relação a **métodos**, é possível ter a seguinte visualização:

- Quanto à origem, a quantidade de métodos definidos diretamente nas classes da especificação, a quantidade total de métodos herdados (métodos herdados menos os sobrescritos), e quantidade total de métodos disponíveis, que representa o somatório dos métodos definidos na classe mais os métodos herdados.
- Quanto à visibilidade, a quantidade de métodos ocultos e visíveis da especificação
- Quanto à sobrescrição, a quantidade de métodos sobrescritos e métodos novos da especificação.
- E a classificação quanto aos métodos que podem ser: *templates*, abstratos ou regulares.

A visão dos métodos também é possível de ser visualizada na figura 5.1.

A visão geral por classe, apresenta as mesmas informações para os atributos e métodos.

Tabelas

Medidas extraídas pela ferramenta de análise são agrupadas em *Visão Geral da Especificação*, *Visão Geral por Classe* e *Métricas*. Medidas para a especificação como um todo é apresentado na figura a seguir:

Visualização dos Resultados das Métricas

	Abstratas	Concretas
Classes	6	15

	Disponíveis	Definidos	Herdados	Ocultos	Visíveis	Sobrescritos	Novos	Template	Regulares	Abstratos
Atributos	72	41	31	0	0	0	0	0	0	0
Métodos	250	65	185	0	185	17	168	2	173	10

	Fatores
Herança Atributos	0,43
Herança Métodos	0,26
Métodos Ocultos	0,00
Polimorfismo	0,21
Acoplamento	0,08

FIGURA 5.10 – Exemplo de Tabela

Gráficos

Todas as métricas implementadas neste trabalho podem ser visualizadas em gráficos, conforme é apresentado no capítulo de métricas. A seguir é apresentado um gráfico que mostra uma visão geral dos métodos da especificação. É possível também ter uma visão dos métodos e atributos para cada classe da especificação.

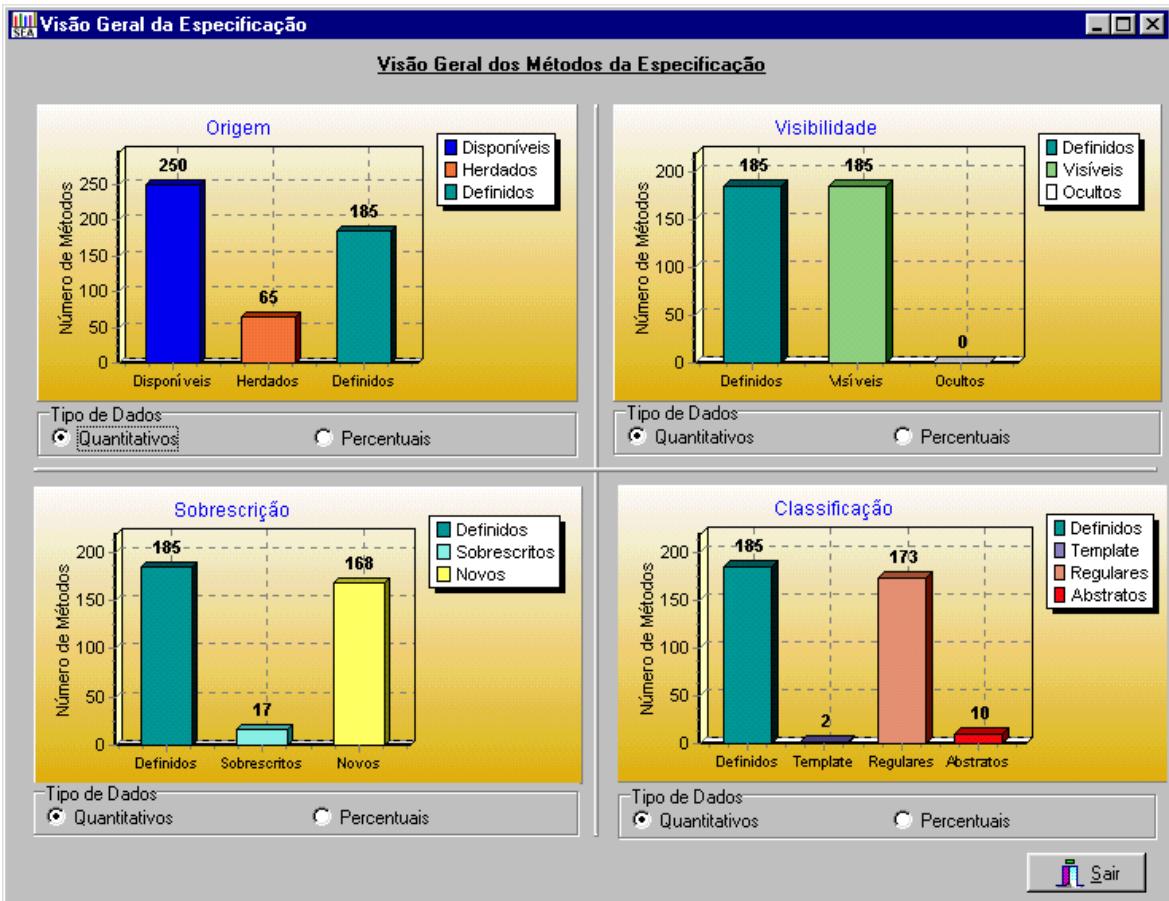


FIGURA 5.11 – Exemplo de Gráfico

5.5 Conclusão

Neste capítulo foram apresentadas algumas ferramentas que têm como objetivo o suporte à avaliação de software orientado a objetos. Foi apresentada uma relação dessas ferramentas com a ferramenta MET.

Alguns trabalhos, como [LYO 99] que descrevem a utilização de ferramentas para extração de métricas, foram localizados, porém, alguns não possuíam a documentação necessária para o entendimento da proposta, assim não foram descritos neste trabalho.

A automação do cálculo das métricas é importante para que o processo de metrificação seja viável, visto que a obtenção desses valores manualmente é extremamente dispendiosa.

As ferramentas MET e a de visualização dos resultados das métricas também foram apresentadas. A inserção da ferramenta MET no ambiente SEA possibilitou que as métricas possam ser extraídas já nas fases de análise e projeto. Um recurso importante disponibilizado pelo ambiente é o diagrama de corpo de método. Através deste diagrama foi possível a extração de algumas métricas, que somente seria possível avaliando o código fonte.

Um aspecto negativo de a ferramenta estar inserida no ambiente SEA é que a extração de métricas ficou restrita aos modelos documentados no ambiente.

Um exemplo de uso da ferramenta MET e da ferramenta de visualização dos resultados das métricas será descrito no Estudo de Caso, no próximo capítulo.

6 ESTUDO DE CASO

6.1 *Introdução*

Como parte do trabalho de pesquisa foi desenvolvido um estudo de caso, tendo como objetivo extrair as métricas de uma especificação de um software existente, para subsidiar sua avaliação. A metrificação foi realizada sobre a especificação de um *framework*, FraG – *framework* para jogos de tabuleiro [SIL 97]. A especificação do *framework* no ambiente SEA foi desenvolvida por Freiberger [FRE 02], através de engenharia reversa do *framework* implementado em *smalltalk*.

O *framework* avaliado generaliza os conceitos (classes), as características (atributos) e ações (métodos) do domínio jogos de tabuleiro.

Em relação a conceitos internos e externos o *framework* possui:

- 28 classes, sendo 07 classes externas e 21 classes do próprio *framework* de jogos.
- 46 atributos, sendo 05 atributos de classes externas e 41 atributos de classes do próprio *framework* de jogos.
- 206 métodos, sendo 21 Métodos de classes externas e 185 Métodos de classes do próprio *framework* de jogos.

A ferramenta MET extrai métricas das classes definidas no *framework* e consequentemente dos métodos, dos atributos e dos relacionamentos. As classes externas à especificação não serão consideradas para análise, pois o objetivo é extrair métricas da especificação corrente, no caso o *framework* de jogos. Por exemplo, as classes *Model*, *ApplicationModel*, *Controller*, *ControllerWithMenu*, *View* externas ao *framework* de jogos não foram consideradas para a extração das métricas.

6.2 Obtenção e Análise dos Resultados

A seguir são apresentados os resultados das métricas e a interpretação desses resultados, feita pela autora deste trabalho. A ferramenta MET não emite parecer de qualidade da especificação tratada. A avaliação da qualidade da especificação tratada demanda atuação humana para análise dos resultados das métricas.

6.2.1 Visão geral da especificação

Na figura 6.1 é possível visualizar dados da especificação²⁶ como um todo.

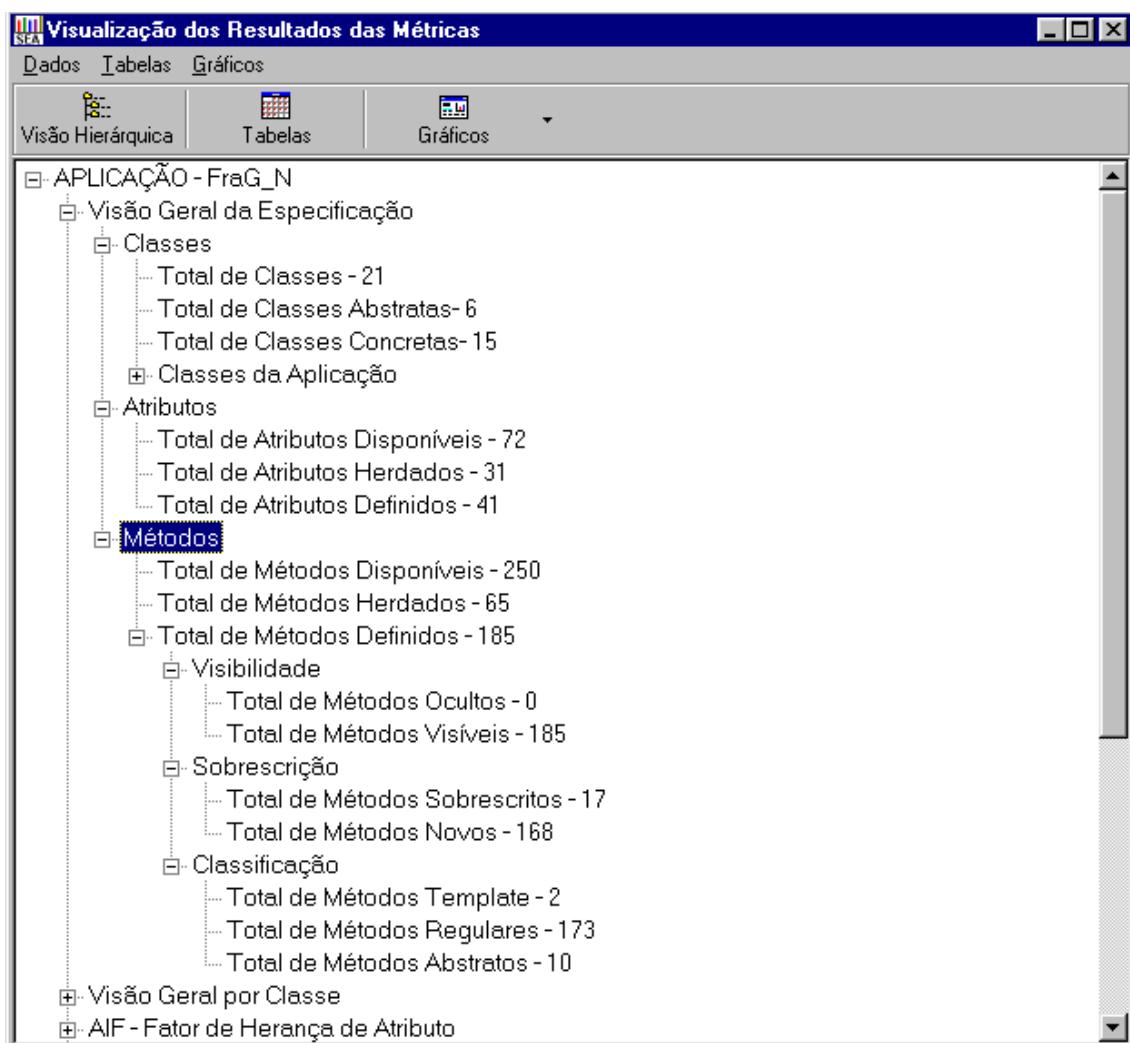


FIGURA 6.1 - Visão geral - *framework*

²⁶ Especificação no contexto do estudo de caso refere-se à especificação do framework de jogos de tabuleiro.

Os dados que são apresentados na árvore podem também ser visualizados no gráfico, como ilustrado a seguir:

Gráfico: Visão geral das classes da aplicação

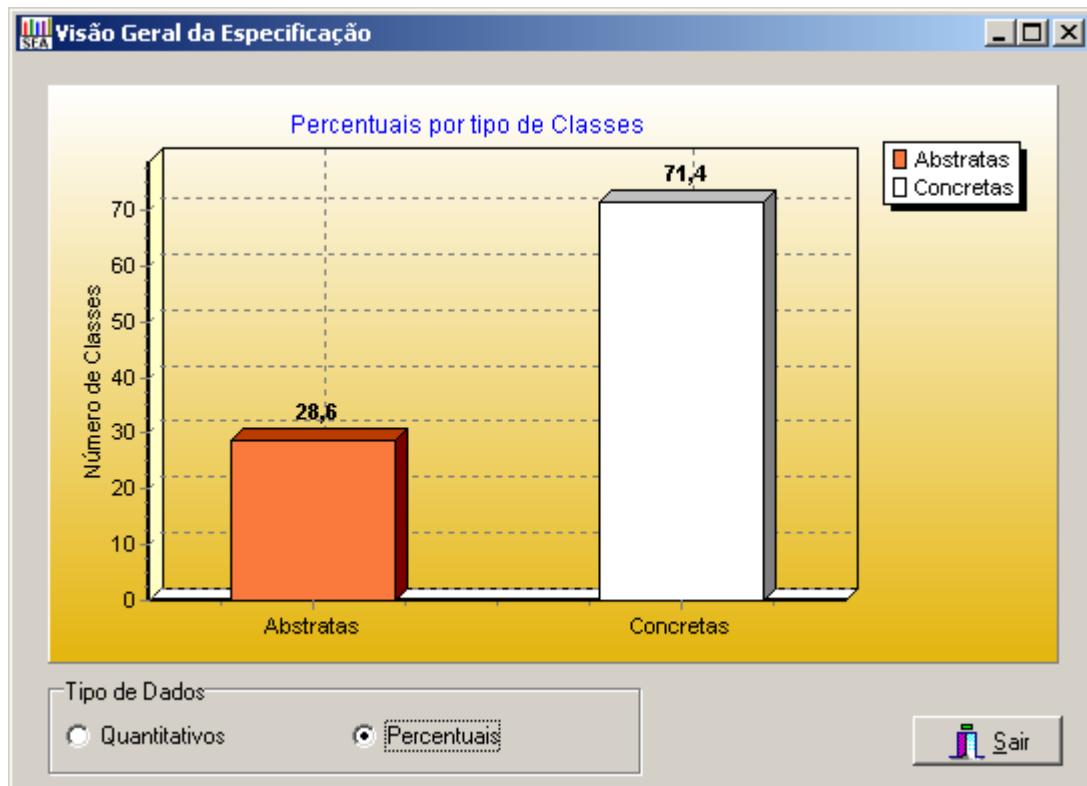


GRÁFICO 6.1 - Visão geral das classes - *framework*

O gráfico anterior apresenta uma visão de percentuais, sendo possível observar que do total de classes da aplicação, 28,6% são abstratas e 71,4% são concretas.

Gráfico: Visão geral dos métodos da aplicação

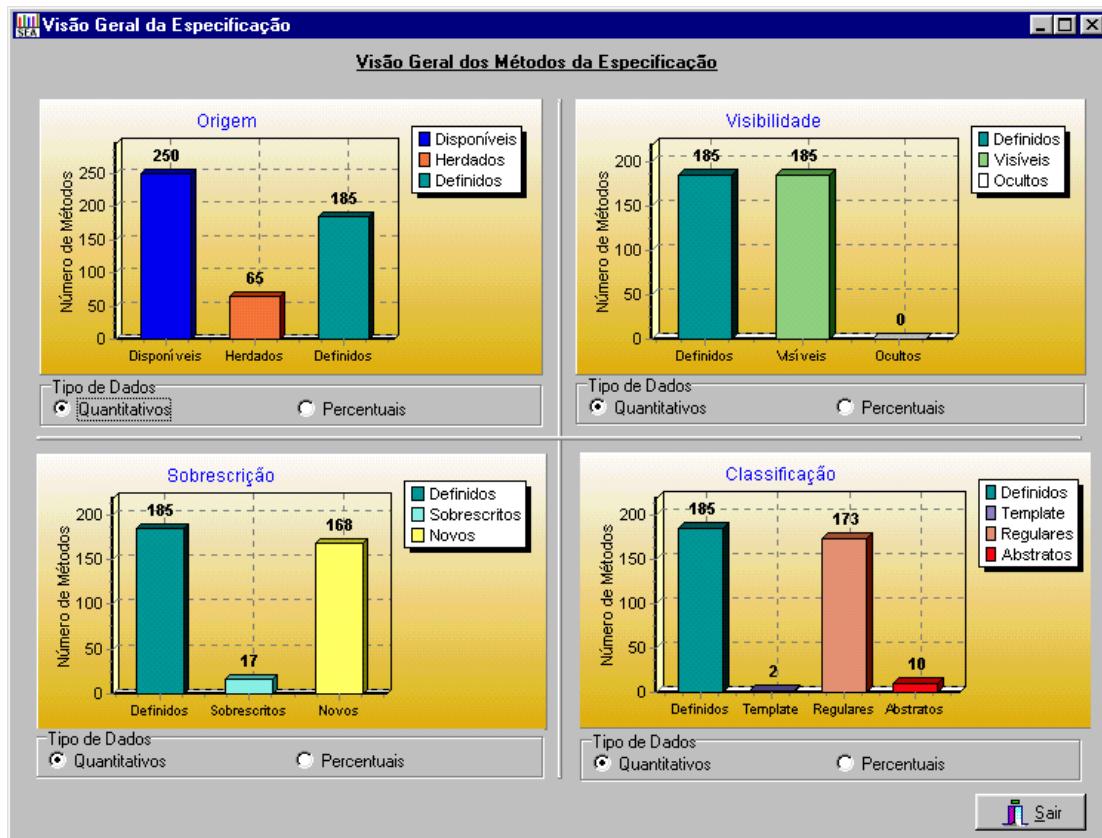


GRÁFICO 6.2 - Visão geral dos métodos - *framework*

Esse gráfico oferece uma visão geral dos métodos da especificação do *framework*.

6.2.2 Encapsulamento

Em relação ao encapsulamento dos atributos, todos os atributos do *framework* são ocultos, atendendo um dos preceitos do paradigma da orientação a objetos. O gráfico não foi gerado para esta métrica, pois no ambiente SEA todos os atributos são definidos como protegidos.

Em relação ao encapsulamento dos métodos, no *framework* todos os métodos foram classificados como públicos, ou seja, visíveis às outras classes. Essa não é uma prática muito recomendada pelo paradigma da orientação a objetos, pois somente os serviços que a classe pode oferecer, devem ser públicos.

Vale ressaltar, que a especificação do *framework* originou-se de engenharia reversa, a partir de um código escrito em *SmallTalk*, que não distingue métodos públicos dos privados, ou seja, todos os métodos foram considerados públicos.

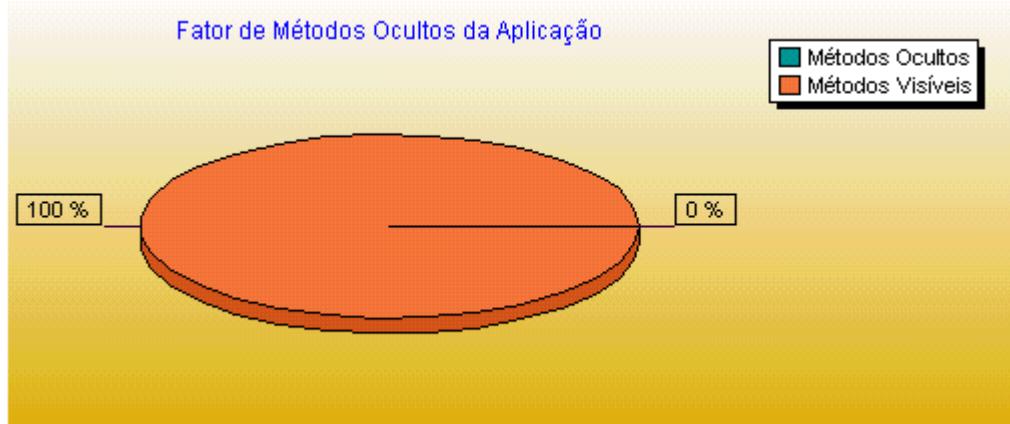


GRÁFICO 6.3 - Métrica: MHF - *framework*

6.2.3 Tamanho dos Métodos

Todos os métodos do *framework* não ultrapassaram a quantidade de 30 *statements* (valor adotado como crítico para o presente estudo de caso).

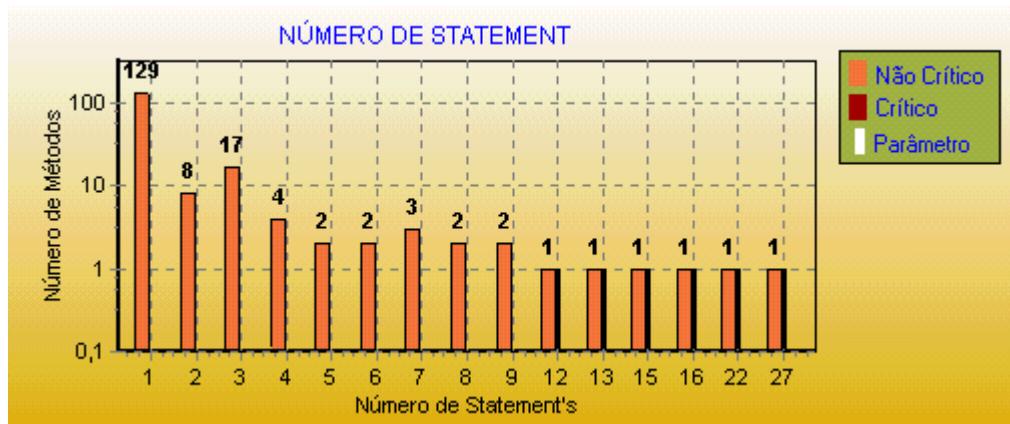
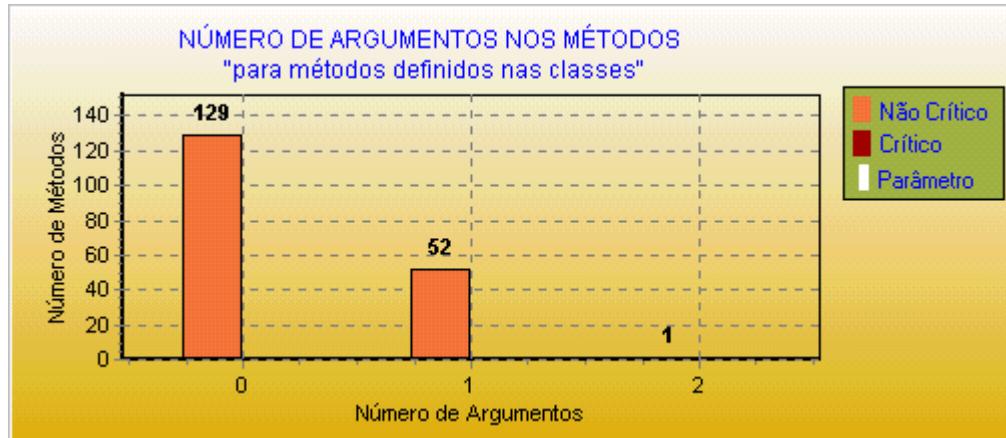


GRÁFICO 6.4 - Métrica: Tamanho dos métodos - *framework*

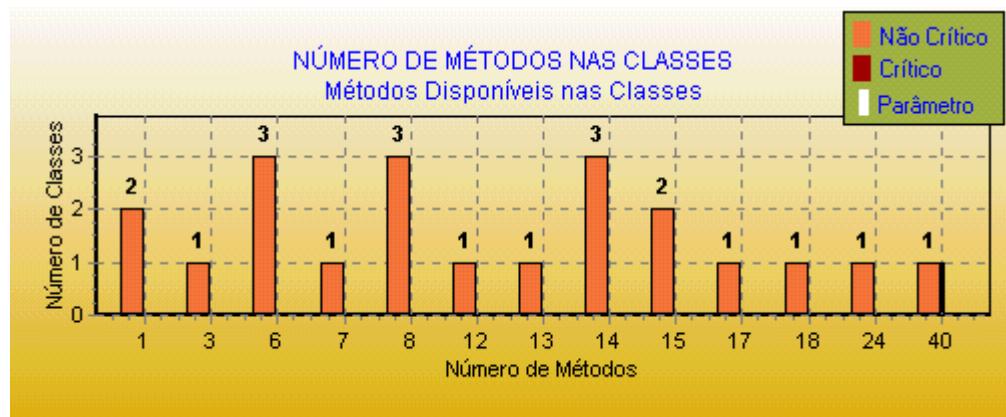
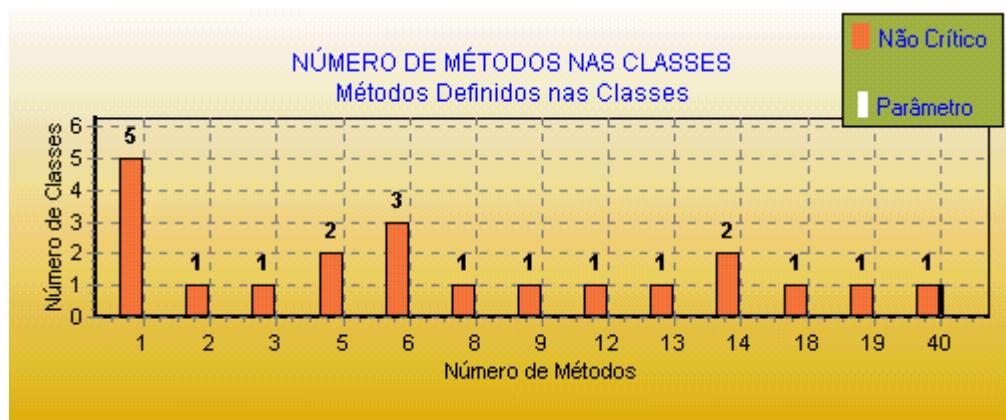
6.2.4 Número de Argumentos nos Métodos

Todos os métodos do *framework* possuem número de argumentos que não ultrapassam a quantidade de 5 argumentos (valor adotado como limite para o presente estudo de caso).

GRÁFICO 6.5 - Métrica: Número de argumentos nos métodos - *framework*

6.2.5 Número de Métodos nas Classes

O número de métodos das classes do *framework* não ultrapassa o valor crítico adotado para o presente estudo de caso, que é de 50 métodos disponíveis por classe. A seguir os gráficos apresentam as quantidades de métodos disponíveis e definidos nas classes.

GRÁFICO 6.6 - Métricas: Número de métodos disponíveis nas classes – *framework*GRÁFICO 6.7 - Métrica: Número de métodos definidos nas classes - *framework*

6.2.6 Métodos Complexos por Classe

Os valores para a complexidade dos métodos das classes são considerados baixos e coincidentes com o número de métodos definidos nas classes, pois o critério utilizado para avaliar a complexidade do método foi baseado no número de *statements*. Sendo que o valor adotado para análise, foi complexidade 1 para métodos com até 30 *statements*. Neste estudo de caso, todos os métodos têm complexidade 1.

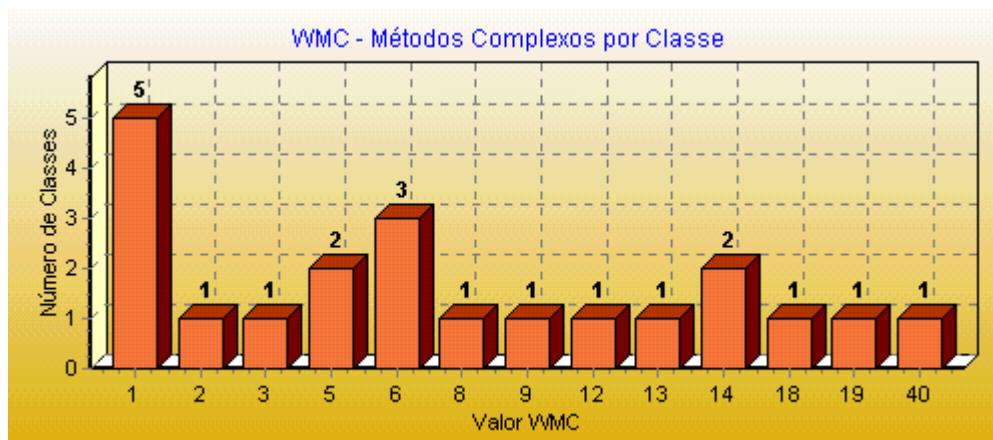


GRÁFICO 6.8 - Métrica: Métodos complexos por classe - *framework*

6.2.7 Referência a SubClasses

Através deste resultado é possível verificar que não há nenhuma classe na especificação que referencia as suas subclasses, atendendo assim, uma das diretrizes de qualidade descritas no capítulo 3.

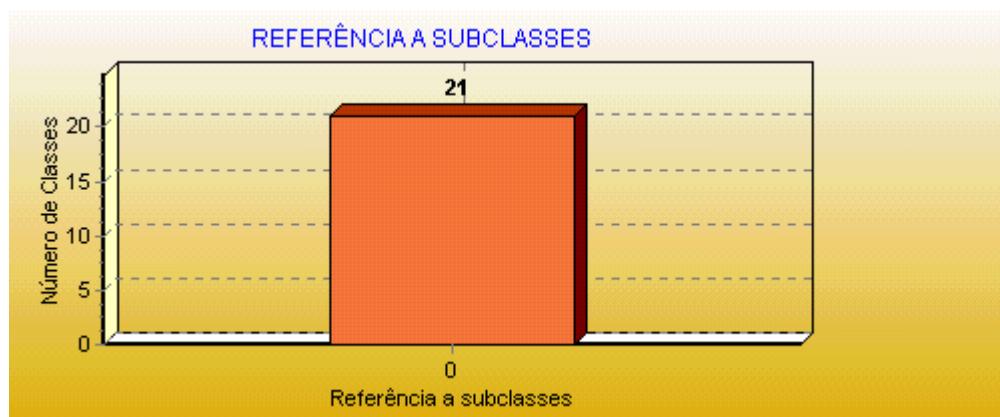


GRÁFICO 6.9 - Métrica: Referência a SubClasses – *framework*

6.2.8 Sobrecarga em Classes Isoladas

Os resultados do gráfico representam que várias classes do *framework* de jogos apresentam sobrecarga de métodos, por isso estão mais suscetíveis a atenderem determinada invocação de métodos com diferentes entrada de dados. Por exemplo, o resultado de OVO igual a 2, está presente na classe *PlayerAutomatic*, pois *esta classe* possui o método *board*, com 2 sobrecargas: *board(0)* e *board(1)*.

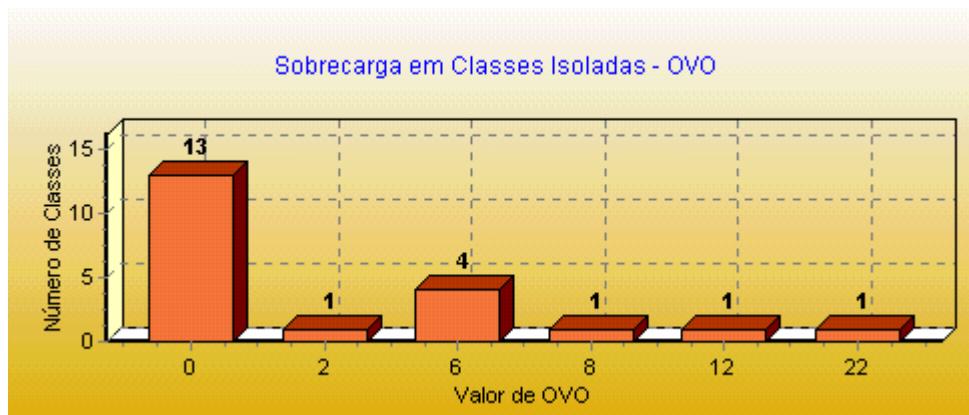


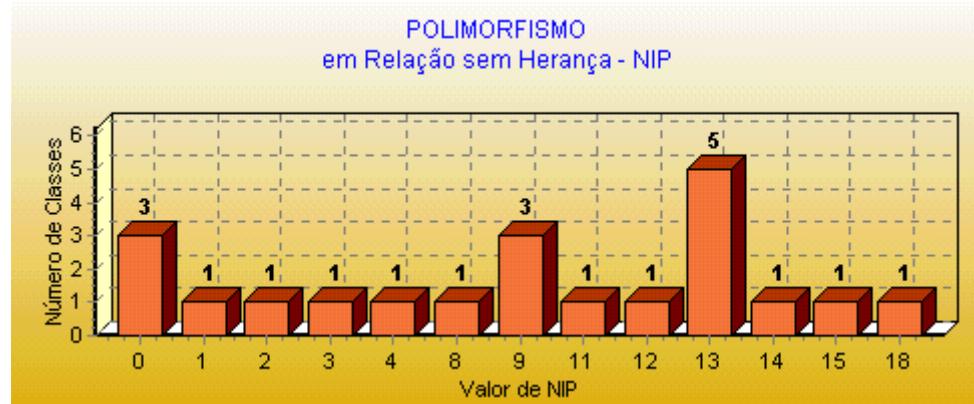
GRÁFICO 6.10 - Métrica: OVO - *framework*

6.2.9 Nome de Métodos

O gráfico mostra que o *framework* possui vários métodos com mesmo nome em classes sem relação de herança.

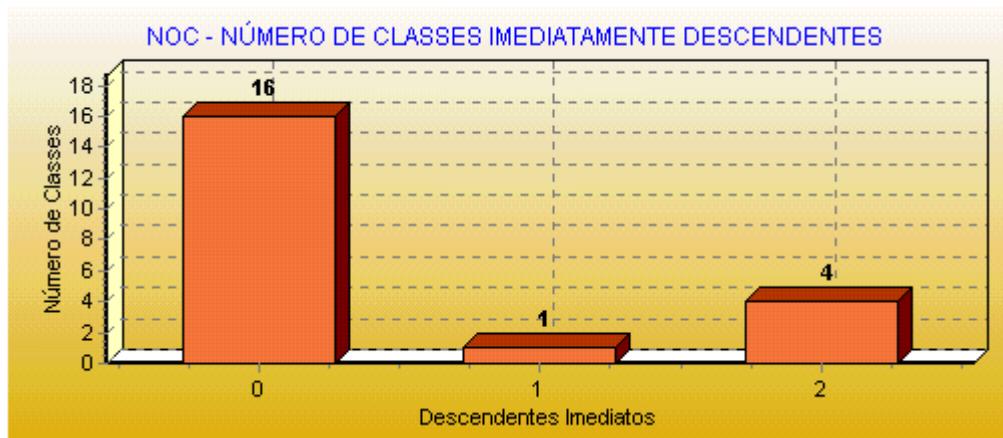
Por exemplo, o resultado de NIP igual a 2, está presente na classe *Player*, pois esta classe os métodos *equipament(0)* e *equipament(1)*, que também estão presentes na classe *PositionDecoration*. Porém, estes métodos, nas duas classes, são utilizados para executar as mesmas operações.

O resultado dessa métrica funciona como um alerta para que o desenvolvedor possa verificar se os nomes dados aos métodos realmente estão coerentes, pois é importante que executem operações similares e possuam a mesma assinatura, caso contrário, é preferível evitar utilizar o mesmo nome de métodos.

GRÁFICO 6.11 - Métrica: NIP - *framework*

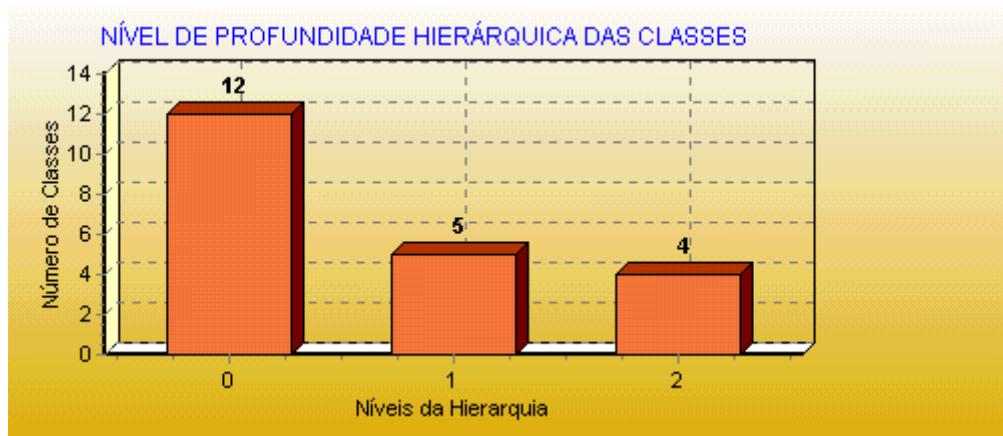
6.2.10 Número de Classes Imediatamente Descendentes

A maior largura de hierarquia de classes no *framework* de jogo é 2.

GRÁFICO 6.12 - Métrica: NOC - *framework*

6.2.11 Profundidade da Árvore de Herança

O maior nível de profundidade hierárquica de herança encontrado no *framework* foi 2.

GRÁFICO 6.13 - Métrica: DIT - *framework*

6.2.12 Fator de Herança de Métodos

No *framework* houve 26% de aproveitamento de código, através de herança de métodos (métodos herdados e não sobrescritos) pelas classes e 74% dos métodos sem reaproveitamento do código (definidos nas próprias classes).

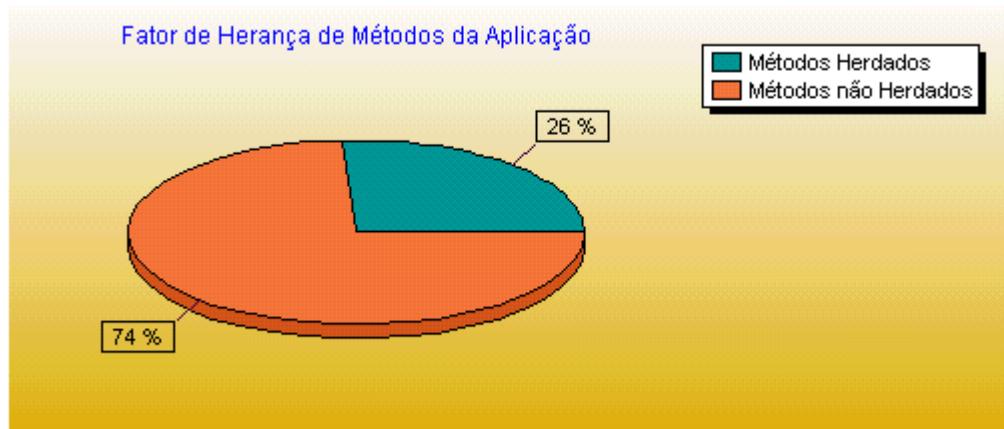


GRÁFICO 6.14 - Métrica: MIF - *framework*

6.2.13 Fator de Herança de Atributos

No *framework* do total de atributos, cerca de 40% são herdados, isto é, são definidos em uma classe e podem ser usados em mais de uma classe.

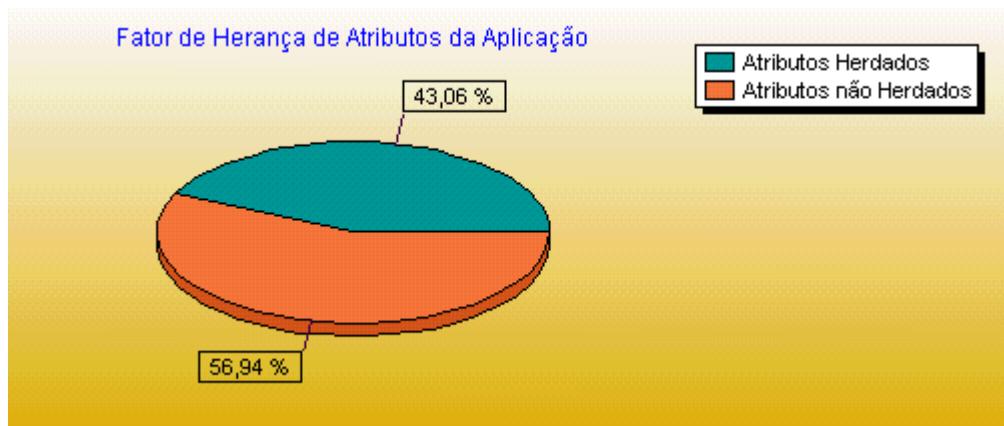


GRÁFICO 6.15 - Métrica: AIF - *framework*

6.2.14 Reação de uma Classe

O conjunto de respostas previsto para as classes pode ser visto no gráfico.



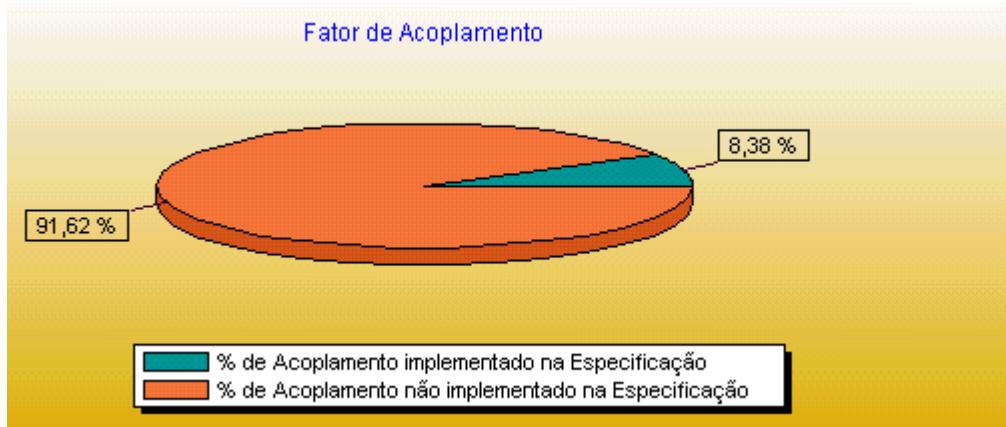
GRÁFICO 6.16 - Métrica: RFC – *framework*

6.2.15 Acoplamento

Os dados extraídos em função das métricas de acoplamento mostram que o *framework* de jogos apresenta um baixo acoplamento. Isto é um fator positivo da especificação do *framework*, pois um baixo grau de acoplamento tende a facilitar o entendimento, a manutenção, a diminuir a dificuldade em realizar testes e ainda aumentar o potencial de reuso (classes mais independentes são mais fáceis de serem utilizadas em outras aplicações).

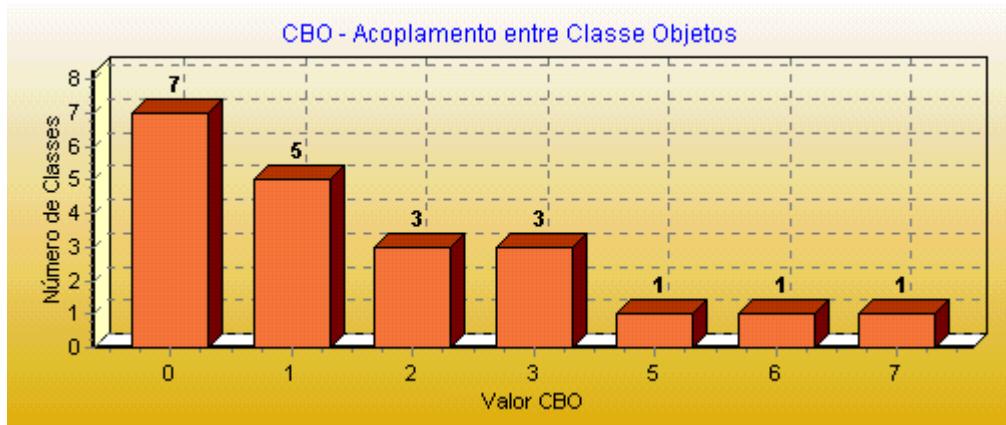
A seguir são apresentados os resultados extraídos em função das métricas de acoplamento. Cada métrica quantifica aspectos diferentes, como tipo de acoplamento e sentido do impacto da mudança, em função do acoplamento.

Fator de Acoplamento

GRÁFICO 6.17 - Métrica: CF - *framework*

Para essa métrica é avaliado o acoplamento entre classes, excluindo o acoplamento existente em uma relação de herança. Como esse percentual representa o fator de acoplamento para a especificação como um todo, confirma a análise anterior.

Acoplamento entre Classes de Objetos

GRÁFICO 6.18 - Métrica: CBO - *framework*

O gráfico mostra o quanto as classes estão acopladas a outras classes, considerando todos os tipos de acoplamento, ou seja, através de tipo de atributo, tipo de parâmetro de método, tipo do retorno de método, variável de método ou por invocação de mensagem de outra classe. Para extração desses valores é considerado também o relacionamento entre classes relacionadas por herança. Pode-se observar que o acoplamento realmente foi baixo.

O foco principal das métricas apresentadas a seguir é uma avaliação pontual sobre o tipo de acoplamento existente, o sentido do impacto da mudança nas classes.

Acoplamento por Classe-Atributo por importação nos ancestrais

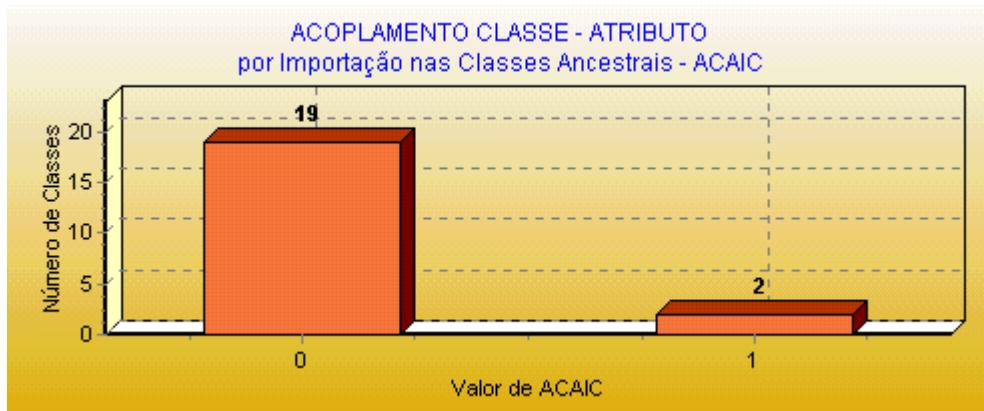


GRÁFICO 6.19 - Métrica: ACAIC - *framework*

No gráfico anterior pode-se observar que 2 classes, no caso, *DiceDecoration* e *PositionDecoration*, que implementam o *padrão de projeto Decorator* ([GAM 94]²⁷ apud [SIL 00]), dependem de suas ancestrais. Essas duas classes importarão o impacto da mudança realizado na classe ancestral, pois as duas classes possuem um atributo do tipo de uma classe ancestral.

Acoplamento por Classe-Atributo por exportação nos descendentes

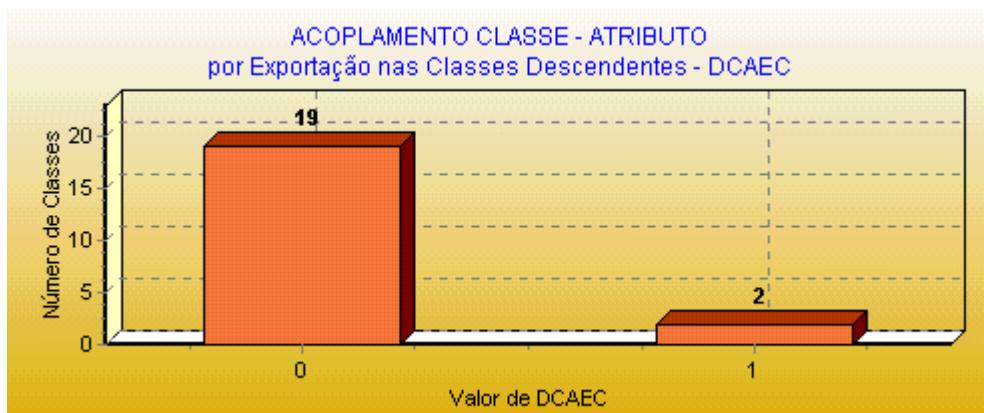


GRÁFICO 6.20 - Métrica: DCAEC - *framework*

No gráfico anterior pode-se observar que 2 classes são referenciadas 1 vez nas classes descendentes na forma de atributo. As classes *Position* e *Dice* são referenciadas nas classes descendentes. Essas duas classes merecem mais atenção no momento da manutenção, pois alterações realizadas nestas classes afetarão as classes descendentes.

²⁷ GAMMA, Erich. Design Pattern: elements of reusable object-oriented software. Addison –Wesley, 1994

Acoplamento por Classe-Atributo por importação entre classes sem relação de herança

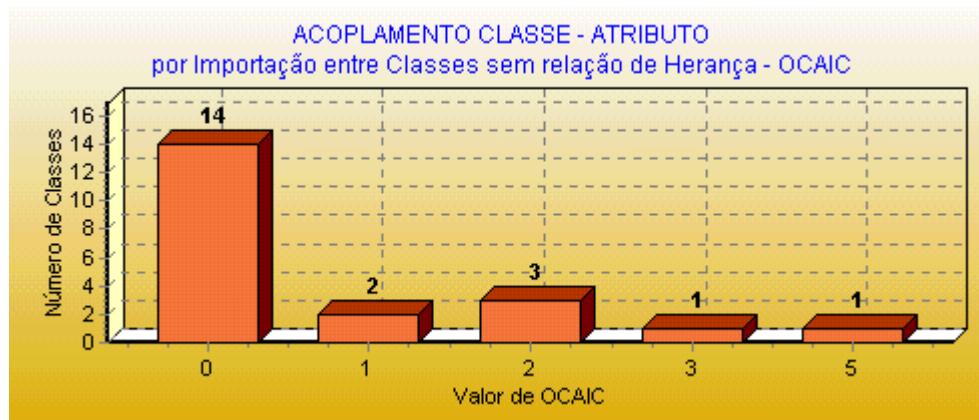


GRÁFICO 6.21 - Métrica: OCAIC - *framework*

Pode-se identificar a quantidade de classes que importarão o impacto de mudanças realizadas em classes sem relação de herança.

Acoplamento por Classe-Atributo por exportação entre classes sem relação de herança

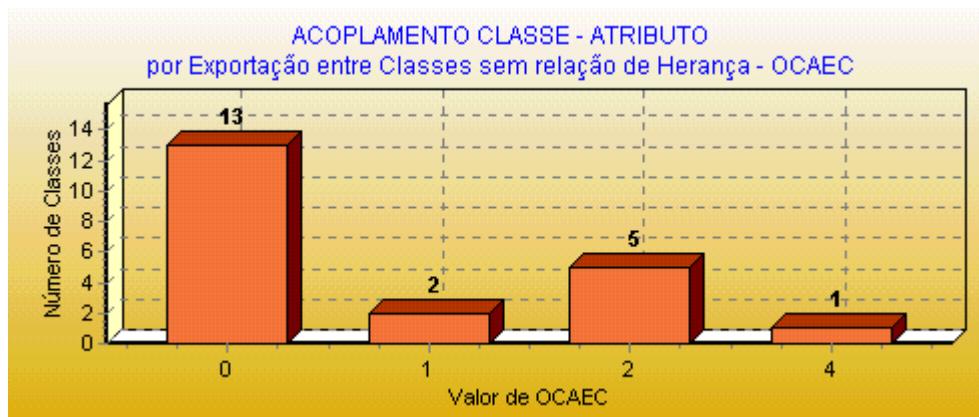


GRÁFICO 6.22 - Métrica: OCAEC - *framework*

Nesse gráfico, pode-se observar a quantidade de classes que foram referenciadas na forma de atributo em classes não relacionadas por herança.

Acoplamento Classe-Método por importação nos ancestrais

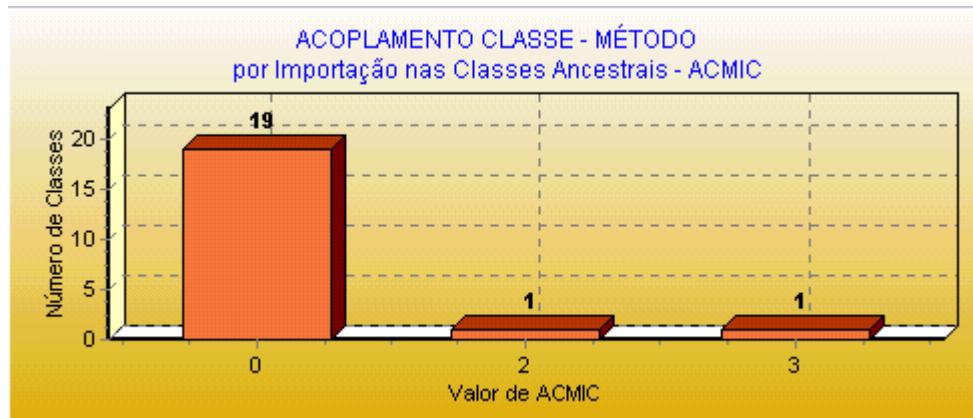


GRÁFICO 6.23 - Métrica: ACMIC - *framework*

No gráfico anterior pode-se observar que 1 classe (*DiceDecorator*) faz 3 referências a classes ancestrais, na forma de parâmetro ou retorno de um método e 1 classe (*PositionDecoration*) faz 3 referências.

Acoplamento Classe-Método por exportação nos descendentes

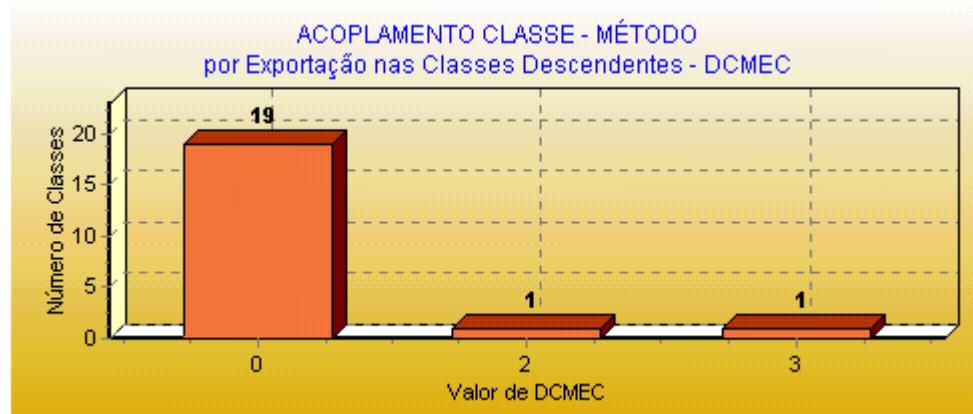


GRÁFICO 6.24 - Métrica: DCMEC - *framework*

Nesse gráfico, pode-se observar que 1 classe foi referenciada em 2 classes descendentes na forma de parâmetro ou como o retorno de um método.

As classes que foram referenciadas merecem mais atenção no momento da manutenção, pois alterações realizadas nestas classes afetarão as classes descendentes.

Acoplamento Classe-Método por importação entre classes sem relação de herança

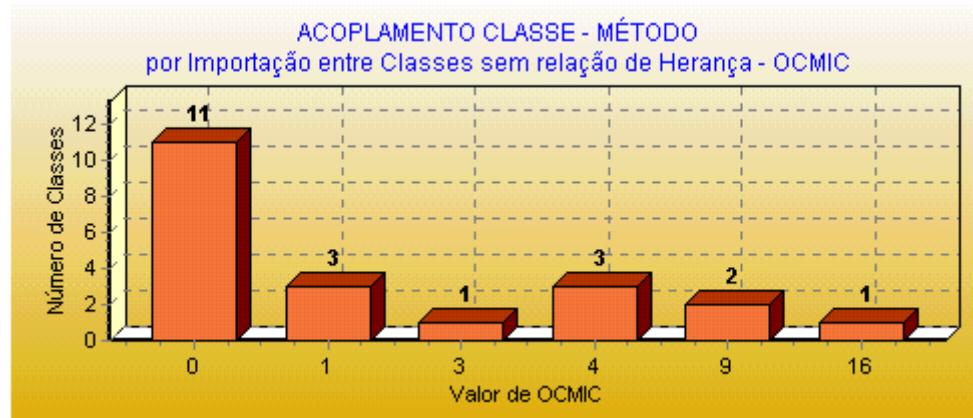


GRÁFICO 6.25 - Métrica: OCMIC - *framework*

Pode-se identificar a quantidade de classes que estarão importando o impacto de mudanças realizadas em classes sem relação de herança.

Acoplamento Classe-Método por exportação entre classes sem relação de herança

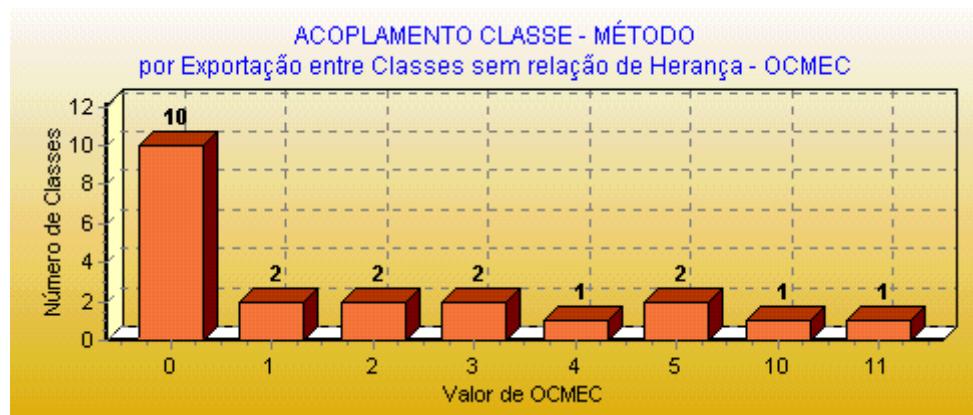


GRÁFICO 6.26 - Métrica: OCMEC - *framework*

Nesse gráfico, pode-se observar que somente 10 classes não foram referenciadas em nenhuma classe descendente na forma de parâmetro ou como o retorno de um método.

As classes que foram referenciadas, merecem mais atenção no momento da manutenção, pois alterações realizadas nestas classes poderão afetar outras classes.

Acoplamento Método-Método por importação nos ancestrais

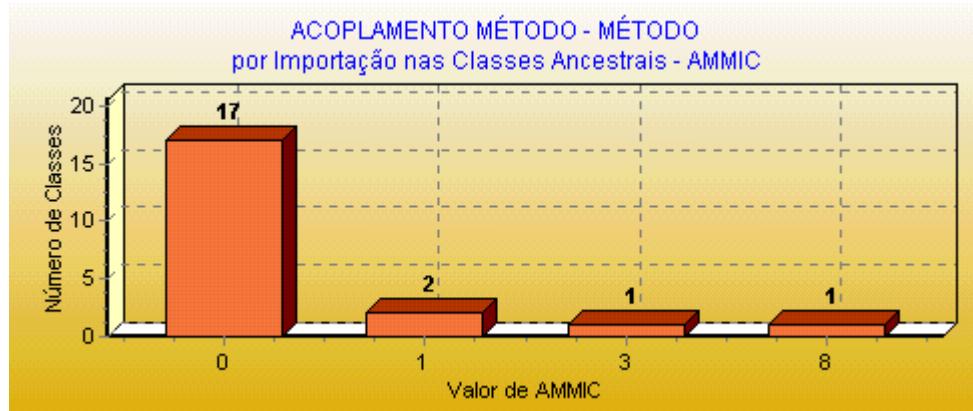


GRÁFICO 6.27 - Métrica: AMMIC - *framework*

No gráfico anterior pode-se observar que 17 classes não fazem referência as suas classes ancestrais, 1 classe (*PositionDefault*) faz 3 referências e 1 classe (*PositionDecoration*) faz 8 referências às classes ancestrais através de invocação de métodos.

Acoplamento Método-Método por exportação nos descendentes

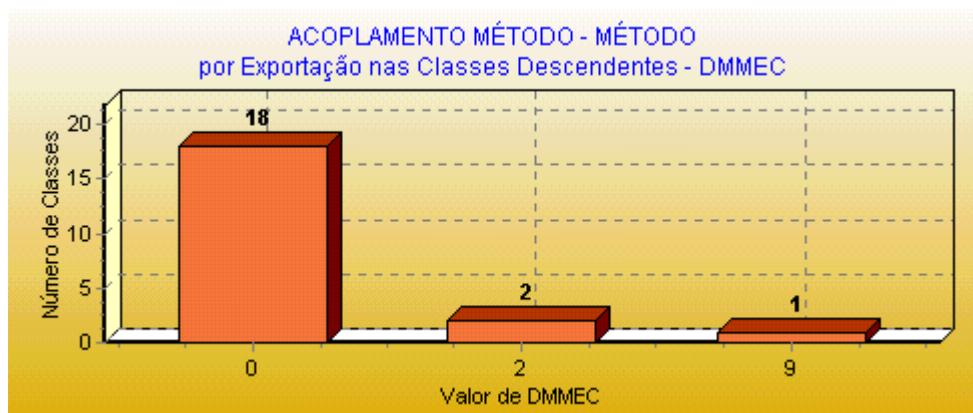


GRÁFICO 6.28 - Métrica: DMMEC - *framework*

Segue a mesma análise dos gráficos anteriores (DCAEC e DCMEC), porém, considerando a interação método-método, ou seja, métodos da classe analisada que foram invocados pelos métodos das classes descendentes. Pode-se observar que 18 classes não foram referenciadas nas classes descendentes através da invocação de métodos.

Acoplamento Método-Método por importação entre classes sem relação de herança

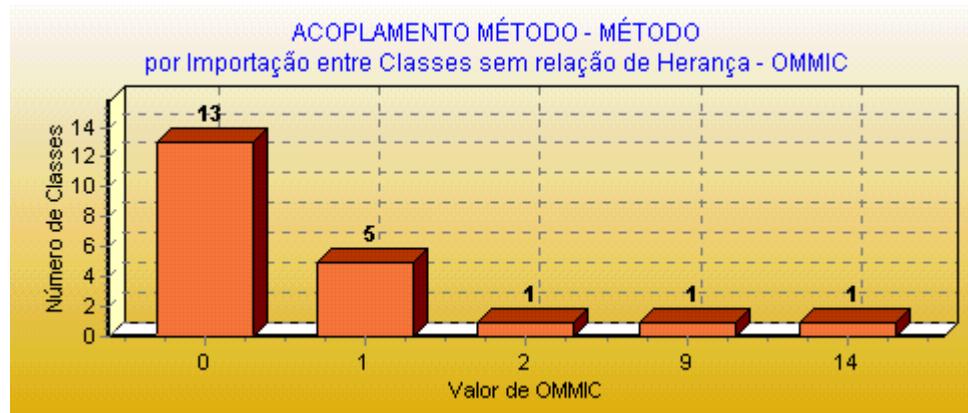


GRÁFICO 6.29 - Métrica: OMMIC - *framework*

No gráfico anterior pode ser observado que 13 classes não invocaram método de classes não relacionadas por herança e que 1 classe invocou 9 métodos de outras classes e 1 classe invocou 14 métodos de classes não relacionadas por herança.

Acoplamento Método-Método por exportação entre classes sem relação de herança

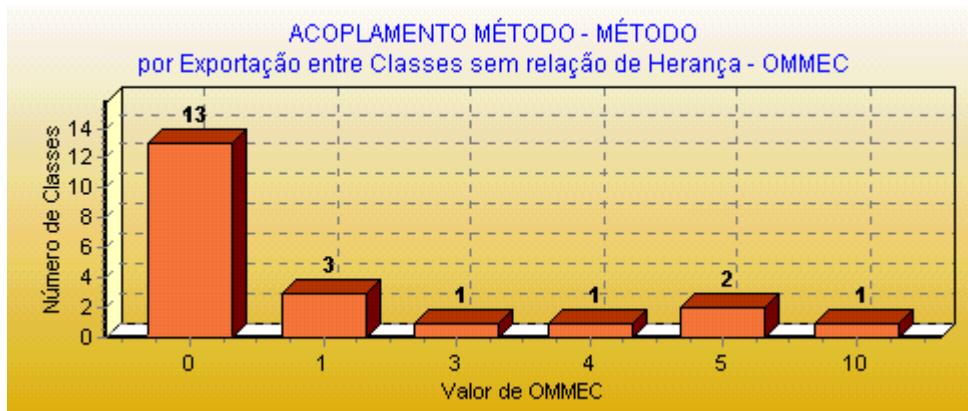


GRÁFICO 6.30 - Métrica: OMMEC - *framework*

Nesse gráfico, pode-se observar que 13 classes não foram referenciadas em classes não relacionadas por herança, através da invocação de seus métodos. Por outro lado, 1 classe (*Player*) foi referenciada nas 10 vezes e ela merece mais atenção no momento da manutenção, pois alterações realizadas nesta classe poderão afetar as outras classes.

6.2.16 Polimorfismo

Dentro da possibilidade máxima de polimorfismo, o *framework* implementou 20% de métodos polimórficos. Isto pode ser observado no gráfico de fator do polimorfismo.

O polimorfismo dinâmico (método com mesmo nome e mesma assinatura) foi bastante explorado pelo *framework*. Por outro lado, o polimorfismo estático (métodos com mesmo nome e com assinaturas diferentes) foi menos explorado.

Fator de Polimorfismo

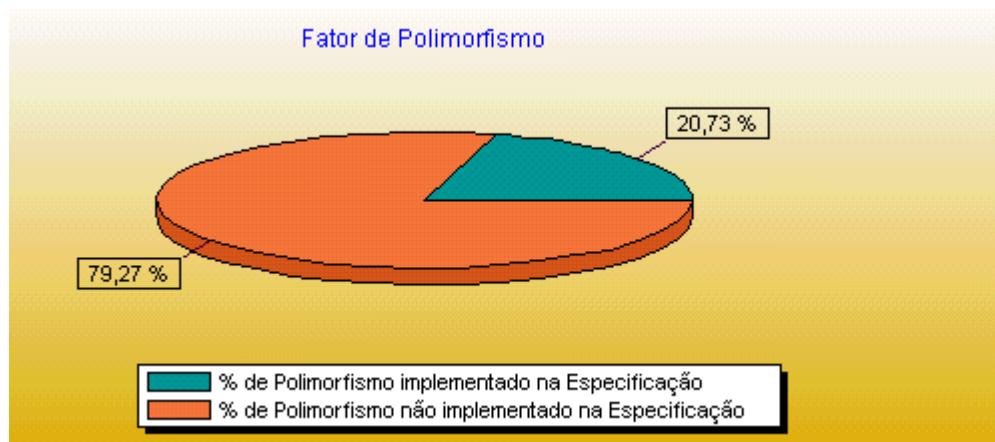


GRÁFICO 6.31 - Métrica: PF - *framework*

Representa o percentual de métodos que tiveram sua implementação alterada nos métodos herdados.

Polimorfismo Dinâmico nos Ancestrais

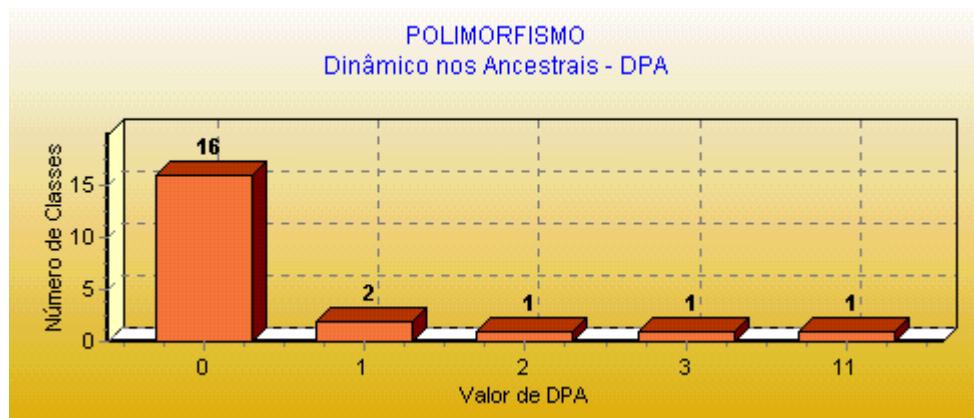


GRÁFICO 6.32 - Métrica:DPA - *framework*

O objetivo é medir o impacto do polimorfismo dinâmico entre uma classe e suas classes ancestrais. Duas classes sobrescreveram 1 método de uma de suas subclasses.

Polimorfismo Dinâmico nos Descendentes

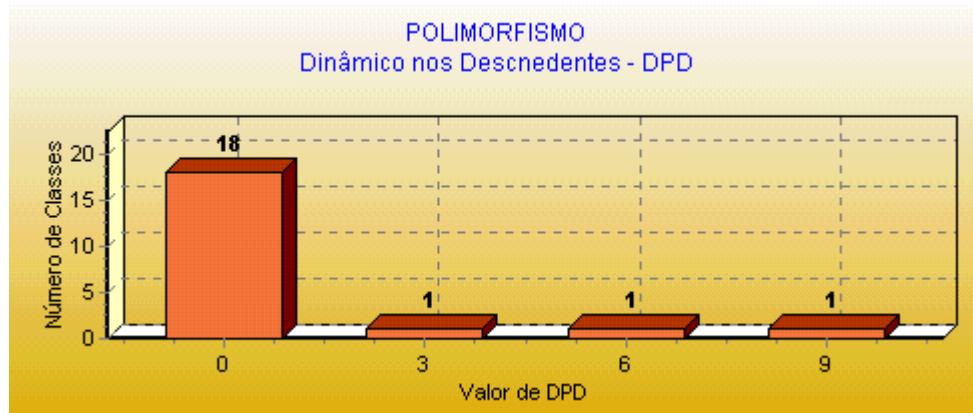


GRÁFICO 6.33 - Métrica: DPD - *framework*

O objetivo é medir o impacto do polimorfismo dinâmico entre uma classe e suas classes descendentes. É identificado no gráfico, que em 1 classe 2 métodos foram sobrescritos e assim sucessivamente.

Polimorfismo Dinâmico

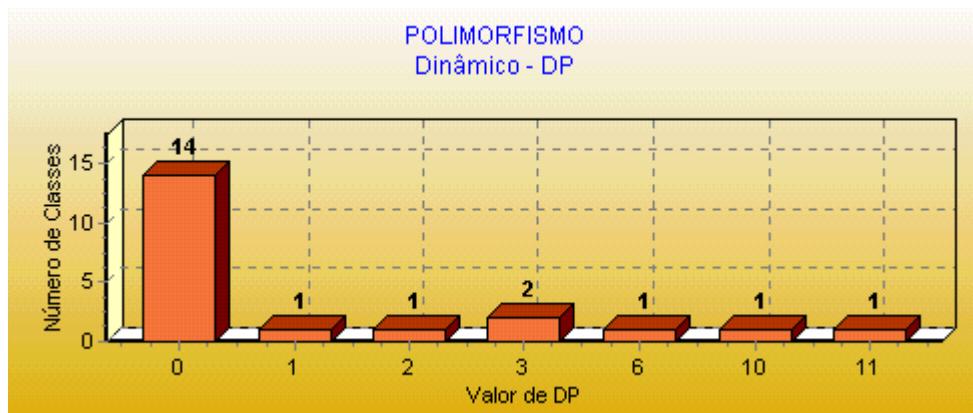


GRÁFICO 6.34 - Métrica: DP - *framework*

Pode ser visualizado no gráfico anterior que em 14 classes não ocorreu polimorfismo dinâmico (nem em relação aos ancestrais e nem em relação aos descendentes), ou seja, nestas classes métodos não sobrescrevem métodos herdados dos ancestrais e nem tiveram seus métodos sobrescritos nos descendentes.

Polimorfismo Estático nos Ancestrais



GRÁFICO 6.35 - Métrica: SPA - *framework*

Pode-se observar, através do gráfico, que em 19 classes não houve ocorrência de polimorfismo estático, que em 1 classe houve 1 polimorfismo estático (um método com mesmo nome e assinatura diferente) e assim sucessivamente.

Polimorfismo Estático nos Descendentes

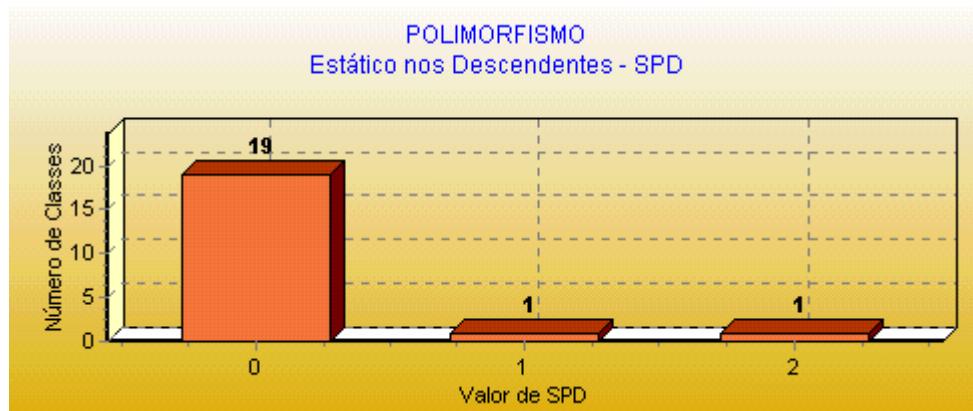


GRÁFICO 6.36 - Métrica: SPD- *framework*

Pode-se observar no gráfico, que em 19 classes não houve ocorrência de polimorfismo estático, em 1 classe houve um método polimórfico estático (1 método com mesmo nome e assinaturas diferentes) e assim sucessivamente.

Polimorfismo Estático

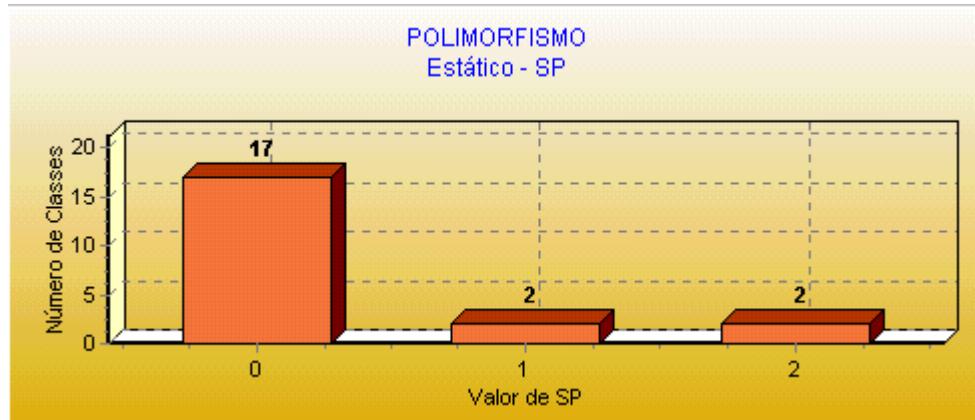


GRÁFICO 6.37 - Métrica: SP – *framework*

Avaliando os nomes dos métodos das subclasses e superclasses, foi detectado que em 17 classes não houve ocorrências de polimorfismo estático, em 2 classes houve 1 ocorrência de polimorfismo estático e assim sucessivamente.

6.2.17 Coesão

Os métodos em uma classe devem ser coesos e para isso o valor de LCOM deve ser baixo. A extração desse resultado foi em função da métrica LCOM – Modificado.

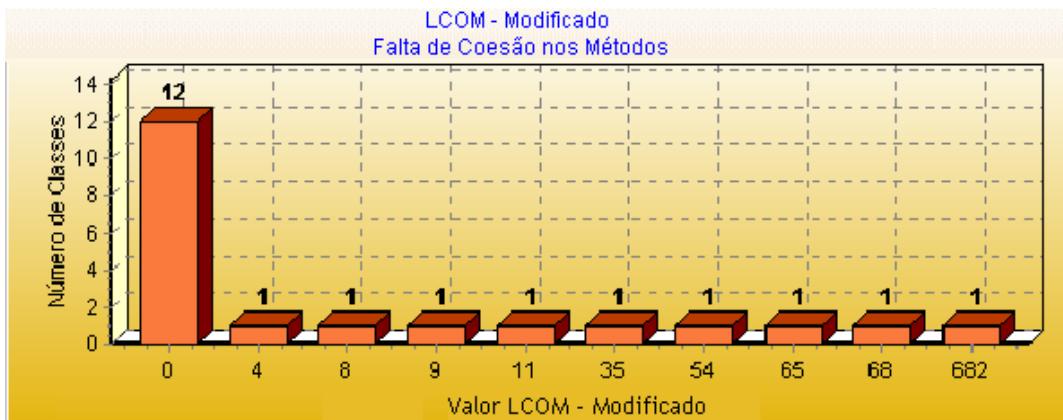


GRÁFICO 6.38 - Métrica: LCOM - modificado – *framework*

No gráfico anterior observa-se que para a maioria das classes tem-se um valor baixo para LCOM, o que é indicativo que os métodos das classes estão coesos. Porém 5 classes têm um valor que se destacam em relação aos demais. Assim, foi analisado o motivo da falta de coesão da classe *GameInterface* que apresentou o valor 682. É uma classe que apresenta 14 atributos e 28 métodos de acesso a atributos, porém, os demais

métodos são abstratos ou tratam de interface gráfica. Embora o resultado da métrica sugere a falta de coesão, a investigação mostra que não há necessidade de alteração nessa classe.

6.3 Resumo

Em síntese, a interpretação dos resultados obtidos das métricas extraídas da especificação do *framework* de jogos está descrita a seguir:

- Todos os atributos são ocultos às classes, atendendo assim, o princípio de que uma classe não deve ter atributos públicos. Com relação aos métodos, observou-se que todos métodos são visíveis as demais classes, porém, isto é justificável em função da especificação avaliada ser fruto de uma engenharia reversa de um código fonte produzido em *smalltalk*, que não permite a definição de métodos ocultos.
- Baixo acoplamento entre as classes do framework, atendendo assim, a diretriz de qualidade que descreve que o acoplamento entre as operações que pertencem a objetos diferentes deve ser minimizado.
- O número de métodos nas classes, número de argumentos nos métodos e tamanho dos métodos satisfazem os critérios/limites adotados neste trabalho.
- Não há nenhuma classe na especificação que referencia as suas subclasses, atendendo assim, a diretriz de qualidade de diz que: *uma classe não deve depender de suas subclasses*.
- Sobre os métodos das classes, observou-se uma baixa complexidade, considerando o critério adotado neste trabalho (tamanho dos métodos).
- Sobrecarga em classes isoladas foi bastante utilizada, mostrando que classe é flexível a diferentes tipos de interface.
- Foram definidos muitos métodos com nomes iguais em classes sem relação de herança. Porém, foi verificado que tais os métodos com nomes iguais são utilizados para executar as mesmas operações.
- Largura e profundidade hierárquica baixas.

- Em relação ao fator de herança de métodos (26%), observa-se que cerca de um quarto dos métodos definidos foram aproveitados por mais de uma classe, o que é um indicador do uso de herança para reuso de implementação.
- Confrontando o valor do fator de herança de métodos com o fator de polimorfismo (20%), observa-se que em acréscimo aos 26% de métodos cujas implementações foram reusadas, 20% dos métodos herdados foram sobreescritos, o que indica também a aplicação do conceito de herança para reuso de interface na especificação analisada.
- Através do valor do fator de herança de atributos, observa-se um aproveitamento de atributos definidos em uma classe, em mais de uma classe da especificação. Isto ocorre em quase metade dos atributos da especificação, o que é um indicador de que o conceito de herança foi utilizado para generalizar características.
- Comparando os valores obtidos através da métrica *Reação de uma classe* com a métrica *Número de métodos na classe*, observa-se na especificação analisada, invocação de outros métodos, pelos métodos das classes durante a execução destes. Isso indica uso de delegação, para divisão de responsabilidades entre mais de um método.

Os resultados exibidos pela ferramenta MET, não correspondem a uma avaliação da especificação tratada, ou seja, dependem de uma interpretação humana. Em geral, a interpretação dos resultados das métricas não apresentou nenhuma situação alarmante. Isso se deve ao fato da especificação utilizada neste estudo de caso corresponder a um *framework* estruturalmente bem construído [SIL 97]. Foi concebido com uso de padrões de projeto e de bons princípios da orientada a objetos.

6.4 Conclusão

O estudo de caso mostra que é possível fazer a interpretação dos resultados das métricas e de certa forma realizar uma avaliação da especificação. Para realizar a interpretação dos resultados das métricas é necessário ter experiência em

desenvolvimento de software orientado a objetos e conhecer os princípios e diretrizes de qualidade.

O estudo de caso não tinha como objetivo efetuar um processo de validação das métricas e sim mostrar uma aplicação prática da extração de métricas através da ferramenta MET. Em [BAS 96] [ROS 98] [GLA 00] podem ser encontrados propostas de validação de métricas.

7 CONCLUSÃO

Neste trabalho foi tratado o desenvolvimento de software utilizando o paradigma de orientação a objetos. Foram tratados aspectos referentes ao processo de desenvolvimento de software e à qualidade do software. Buscou-se identificar critérios de avaliação de qualidade específicos para artefatos de software orientados a objetos, visto que métodos e métricas de análise de produtos elaborados a partir de outras abordagens, na maioria dos casos, não são aplicáveis aos produtos baseados no paradigma de orientação a objetos. Dessa forma, foram localizados trabalhos na literatura que propõem o uso de métricas para controle da qualidade de software, específicas para mensurar características da orientação a objetos.

O controle da qualidade pode ser realizado nas fases iniciais do desenvolvimento do software, evitando despender esforços com a codificação de especificações problemáticas, o que pode significar economia, pois quanto mais tarde ocorrer o tratamento de um defeito, maior será o esforço e custo necessários.

O uso de métricas para o controle da qualidade é bastante difundido, porém, não é uma prática simples de ser realizada. Em primeiro lugar, devido ao esforço de obtenção e, por vezes, a natureza monótona da atividade de extração, o que sugere a adequação do uso de ferramentas que façam a extração das métricas, que, preferencialmente, apóiem a extração das métricas dos produtos gerados em todas as fases do desenvolvimento do software – e não apenas sobre código. Segundo, os resultados das métricas representam valores quantitativos e não qualitativos, o que acarreta a necessidade do desenvolvedor entender a métrica e ainda estabelecer um patamar para o resultado das métricas, considerado ideal para o projeto.

Na tentativa de contribuir para a solução dos problemas acima mencionados, este trabalho propôs o agrupamento de métricas definidas em outros trabalhos e a implementação de uma ferramenta que automatizasse a sua extração. Foi desenvolvido um estudo de caso que analisou uma especificação de um *framework* orientado a objetos. Após obter os resultados das métricas com o uso da ferramenta MET, foi

desenvolvido um exemplo de diagnóstico da especificação, levando em consideração os dados fornecidos pela ferramenta e as características de projeto do *framework*.

7.1 Resultados obtidos

Este trabalho destacou como principais resultados:

- A produção de uma coletânea de métricas técnicas específicas para software orientado a objetos, com o objetivo de auxiliar no controle da qualidade.
- A ferramenta MET, desenvolvida em *SmallTalk* e inserida no ambiente SEA [SIL 00], que tem a finalidade de extrair resultados das métricas de especificações de software orientado a objetos.
- A ferramenta para visualização gráfica dos resultados das métricas extraídas pela ferramenta MET, que foi desenvolvida em Object-Pascal (Delphi).

7.2 Limitações

Algumas limitações foram encontradas ao desenvolver este trabalho:

- O ambiente SEA não controla versões das especificações dos artefatos nele especificadas. Devido a essa limitação, as métricas não puderam ser aplicadas a diferentes versões de um mesmo software, o que seria útil para apoiar a avaliação do impacto de uma alteração.
- Como a ferramenta MET foi incorporada ao ambiente SEA, esta ficou restrita a avaliar somente especificações do seu repositório. Esta limitação pode ser contornada a partir de ferramentas do desenvolvimento de ferramentas que convertam especificações produzidas em outros ambientes para o ambiente SEA. A inserção desse tipo de ferramenta está prevista na arquitetura do ambiente, porém, em sua versão atual não dispõe de tal recurso.
- Ferramenta MET não oferece um parecer sobre a qualidade da especificação orientada a objetos, e sim valores quantitativos de

características da orientação a objetos, que devem ser interpretados pelo usuário da ferramenta.

7.3 Trabalhos Futuros

A seguir são apresentados alguns trabalhos possíveis de serem feitos a partir deste trabalho:

- Identificar novas métricas para quantificar características que não estão contempladas neste trabalho.
- Ampliar a ferramenta apresentada neste trabalho, buscando: suportar versões diferentes da mesma especificação de software orientado a objetos e implementar novas métricas não vislumbradas até o presente momento.
- Realizar novos estudos de caso com especificações de tamanhos variados, utilizando o ambiente SEA para a modelagem e a ferramenta MET para efetuar a extração das métricas.
- Levantar heurísticas em torno das métricas que permitam ir além da coleta de métricas, automatizando a emissão de diagnósticos de qualidade.

7.4 Considerações Finais

A utilização do paradigma da orientação a objetos propicia software mais robusto, com maior flexibilidade, maior potencial de reutilização e maior produtividade, reduzindo o custo de desenvolvimento e o esforço na manutenção. Porém, existem poucos recursos para medir aspectos da tecnologia orientada a objetos de forma automatizada, aplicáveis aos artefatos na fase de projeto. Geralmente, nesta fase de desenvolvimento a avaliação dos artefatos desenvolvidos é feita subjetivamente, com base na experiência das pessoas e a avaliação quantitativa é protelada para depois da geração de código. Nesse contexto, mostram-se úteis técnicas, metodologias e ferramentas que proporcionem mecanismos de suporte à avaliação quantitativa de um software em etapas anteriores à geração de código.

Baseado nesta convicção, este trabalho de pesquisa produziu uma coletânea de métricas para projeto de software orientado a objetos e um protótipo de uma ferramenta que automatiza a extração de métricas a partir de análise estática, ainda na fase de projeto, dando suporte para a identificação de vícios de desenvolvimento antes do esforço de geração de código.

BIBLIOGRAFIA

- [ABO 97] ABOUNADER, Joe Raymond; Lamb, David A. **A Data Model Object-Oriented Design Metrics**. Department of Computing and Information Science, (External Technical Report ISSN-0836-0227-1997/409), Queen's University, Kingston, Ontario, Canada, 1997.
- [ABR 94] ABREU, Fernando Brito; Carapuça, Rogério. **Object-Oriented Software Engineering: Measuring and Controlling the Development Process**. Proceedings of the 4th International Conference on Software Quality, McLean, Virginia, USA, October 1994.
- [ABR 94a] ABREU, Fernando Brito e Carapuça, Rogério; *Candidate Metrics for Object-Oriented Software within a Taxonomy Framework*. Journal of Systems and Software, vol 26, no. 1, July 1994.
- [ABR 95] ABREU, Fernando Brito. **Talk on "Design Metrics for Object-Oriented Software Systems"**. INESC/ISEG . 7th ERCIM Database Research GroupWorkshop on Object Oriented Darabases, Lisbon, 15-16 May 1995
- [ABR 95a] ABREU, Fernando Brito e Abreu; Goulão, Miguel; Esteves, Rita; **Toward the Design Quality Evaluation of Object-Oriented Software Systems**. USA, Proceedings of the 5th International Conference on Software Quality, 1995.
- [ABR 96] ABREU, Fernando Brito e Melo, Walcélia. **Evaluating the Impact of Object-Oriented Design on Software Quality**. Originally published in Proceedings of the 3rd International. Software Metrics Symposium (METRICS'96), Berlin, Germany, IEEE , March 1996.
- [ABR 96a] ABREU, Fernando Brito e Abreu; Esteves, Rita; Goulão, Miguel **Design of Eiffel Programs: Quantitative Evaluation Using the MOOD Metrics**. Santa Bárbara, Califórnia, Originally published in Proceedings of TOOLS'96 USA, July, 1996.
- [AND 98] ANDRADE, R. S. **Aplicações simultâneas de princípios, diretrizes e métricas para redução de complexidade de projeto orientado a objetos**. Rio de Janeiro, Dissertação de Mestrado COPPE/RJ, 1998.
- [ANT 95] ANTONIONI, José, **Qualidade em Software: manual de aplicação da ISO 9000**. São Paulo, Makron Books, 1995.
- [BAR 97] BARRETO, José Júnior. **Qualidade do Produto X Processo**. Disponível por www em <http://www.iso9000.com.br>.
- [BAS 96] BASILI, Victor B; BRIAND, Lionel; MELO, Walcélia. **A Validation of object-Oriented Design Metrics as Quality Indicators**. IEEE, vol 22, no.10, 1996.
- [BEN 99] BENLARBI, Saïda e Melo, Walcélia. **Polymorphism Measures for Early Design Metrics**. Proceedings of the 21st International Conference on Software Engineering, 1999.

- Risk Prediction.** Proc. of the 21th Int'l Conf. on Software Engineering, , Los Angeles, CA, IEEE Press, May, 1999.
- [BER 00] BERGAMO, Marilia L e Melo, Walcélío. **Tutorial sobre Usabilidade de Software.** Universidade Católica de Brasília, 2000.
- [BOE 99] BOEGH, J. et al. **A Method for Software Quality Planning, Control and Evaluation.** IEEE, 1999.
- [BOO 00] BOOCH, Grady; **UML - Guia do Usuário.** Rio de Janeiro, Campus, 2000.
- [BRI 02] BRIAND, Lionel C; Melo, Walcélío; Wüst, Jürgen. **Assessing the Applicability of Fault-Proneness Models Across Object-Oriented Software Projects.** IEEE Transactions on Software Engineering, pp 706-720, Julho, 2002.
- [BRI 97] BRIAND, Lionel; DEVANBU, Prem and MELO, Walcélío. **An Investigation into Coupling Measures for C++.** 19th International Conference on Software Engineering, Boston, USA, 1997.
- [CHI 93] CHIDAMBER, R.S. e Kemerer, C.F. **A Metrics Suite for Object-Oriented Design.** IEEE Transaction on Software Engineering , 20(6), June, 1993.
- [COR 00] CORREA, A. L. et al. **Uma Arquitetura de suporte à avaliação de modelos orientados a objetos.** Anais do Terceiro Workshop Ibero-Americanico de Engenharia Requisitos, 2000.
- [DIA 97] DIAS, M. S., Andrade, R., Travassos, G. H. **Diretrizes para Redução de Complexidade de Software Orientado a Objetos.** Curitiba, In: VIII Conferência Internacional de Tecnologia de Software, 1997.
- [ERN 96] ERNI, K., Lewerentz, C. **Applying design-metrics to object-oriented frameworks,** in Proc. 3rd International Software Metrics Symposium. Los Alamitos, CA, USA, IEEE Comput. Soc. Press, 1996.
- [FRE 02] FREIBERGER, Evandro C. **Supporte ao uso de frameworks orientados a objetos com base no histórico do desenvolvimento de aplicações.** Dissertação de Mestrado, CPGCC/UFSC, 2002.
- [GLA 00] GLASBERG, D.; Erman, E.E, Melo; W.L. e Madhavji, N., **Validating Object-oriented Design Metrics on a Commercial Java Application.** (NRC – CNRC), 2000.
- [HAR 98] HARRISON, R; Counsell, S.J; Nithi and R.V. **An Evaluation of the MOOD Set of Object-Oriented Software Metrics.** University of Southampton, Southampton, 1998.
- [HUT 94] HUTCHINS, Greg, *ISO 9000: Um Guia Completo para o Registro, as Diretrizes da Auditoria e a Certificação Bem-Sucedida.* São Paulo, Makron Books, 1994.
- [JOH 88] JOHNSON, Ralph E.; FOOTE, Brian. **Designing reusable classes.** Journal of Object-Oriented Programming, p. 22-35, June/july, 1988.
- [LAR 00] LARMAN, Craig. **Utilizando UML e padrões – uma introdução à análise e ao projeto orientados a objetos.** Porto Alegre, Bookman, 2000.
- [LYO 99] Lyons, Brian. **Partener Perspective: Quantifying Qulity in Your OO Model.**

- Models**, Disponível em
www.therationaledge.com/rosearchitect/mag/archives/9810/f6.html, 1999.
- [MEY 97] MEYER, B. **Object-Oriented Software Construction**, Cambridge, Prentice-Hall, 1997.
- [MUL 02] MULLER, Robert J. **Projeto de banco de dados: usando UML para modelagem de dados**. Berkeley Brasil, 2002.
- [NBR 93] NBR ISO 8402:1993, **Gestão da Qualidade e Garantia da Qualidade**. 1993.
- [NEV 99] NEVES, Luciane. **A Atividade de Teste Durante o Ciclo de Vida do Software**. Bate Byte, Julho, 1999.
- [PAG 01] PAGE-JONES, Meilir. **Fundamentos do Desenho Orientado a Objetos com UML**. São Paulo, MAKRON Books, 2001.
- [PAU 01] PAULA, Wilson Pádua Filho, **Engenharia de Software – Fundamentos, Métodos e Padrões**. LTC – Livros Técnicos e Científicos, 2001.
- [PAU 95] PAULK, Mark C; WEBER, Charles V; et al **The Capability Maturity Model: Guidelines for Improving the Software Process**. Carnegie Mellon University, Addison-Wesley, 1995.
- [POE 98] POELS, Geert. **An Analytical Evaluation of Static Coupling Measures for Domain Object Classes**. Department of Applied Economic Sciences, Katholieke Universiteit Leuven, ECOOP98-OOPM, Belgium, 1998.
- [PRE 02] PRESSMAN, Roger S. **Engenharia de Software**. Rio de Janeiro, McGraw-Hill, 2002.
- [PRE 95] PRESSMAN, Roger S. **Engenharia de Software**. São Paulo, Makron Books, 1995.
- [PRE 95a] PREE, Wolfgang **Design Patterns for Object-Oriented Software Development**. Addison – Wesley Publishing Company, 1994/1995.
- [REZ 00] REZENDE, Denia Kuhn. **Um Modelo de Avaliação de Qualidade de Software voltado para Especificações Orientadas a Objetos**. Brasília, Trabalho desenvolvido no Mestrado em Informática na Universidade Católica de Brasília, 2000.
- [ROC 01] ROCHA, A. R. C; MALDONADO, J. C; WEBER, K. C. **Qualidade de Software**. São Paulo, Prentice Hall, 2001.
- [ROS 98] ROSENBERG, Linda H. **Applying and Interpreting Object Oriented Metrics**. EUA, Software Technology Conference, 1998.
- [RUM 94] RUMBAUCH, James et al; **Modelagem e projetos baseados em objetos**. Rio de Janeiro, Campus, 1994.
- [SEI 02] SEIBT, P. R. R. S.; HUGO, M.; GRAHL, E. **Ferramenta para cálculo de métrica em softwares orientados a objetos codificados em Object-Pascal**. Blumenau, SC, XI SEMINCO / SC, 2002.
- [SIL 00] SILVA, Ricardo Pereira. **Supporte ao desenvolvimento e uso de frameworks e componentes**. Porto Alegre, PPGC da UFRGS, 2000.

- [SIL 01] SILVA, Ricardo Pereira. **Qualidade de Software e CMM – notas de aulas.** Disponível por www em <http://www.inf.ufsc.br/~ricardo/qualidade1.zip>, 2001.
- [SIL 97] SILVA, Ricardo Pereira, PRICE, Roberto Tom. **O uso de técnicas de modelagem no projeto de frameworks orientados a objetos.** In: Proceedings of 26th International Conference of the Argentine Computer Science and Operational Research Society (26th JAIIO) / First Argentine Symposium on Object Orientation (ASOO'97). Buenos Aires, Argentine: aug. 1997. p.87-94.
- [VEI 99] VEIGA, A; Barbosa, Farnese, R.; Melo, W, **JMetrics Java Metrics Extractor: An Overview.** University of Brasilia, Dep. of Computer Science, Under-Graduating Final Project, Brasilia, DF, Brazil, 1999. Also published as a Catholic University of Brasilia, Master on Informatics, Software Quality Engineering Group, Technical Report, UCB-QSW-TR-1999/01.
- [WEB 99] WEBER, K. C.; ROCHA, A. R. C. **Qualidade e produtividade em software.** São Paulo, Makron Books, 1999.

ANEXO 1 –EXEMPLO DE CÁLCULO DAS MÉTRICAS

Neste anexo são apresentados exemplos de cálculo das métricas e a forma de visualização dos resultados das métricas implementadas na ferramenta de extração de métricas - MET. Alguns diagramas de classes e corpo de métodos aqui apresentados foram criados apenas para demonstrar o cálculo das métricas e para ilustrar os resultados produzidos pela ferramenta MET, sendo gerados a partir de pequenas especificações hipotéticas produzidas no ambiente SEA, com o intuito de simplificar a demonstração dos exemplos.

MÉTRICA DE ACOPLAMENTO

O diagrama de classes, descrito a seguir, será usado para exemplificar o cálculo das métricas de acoplamento.

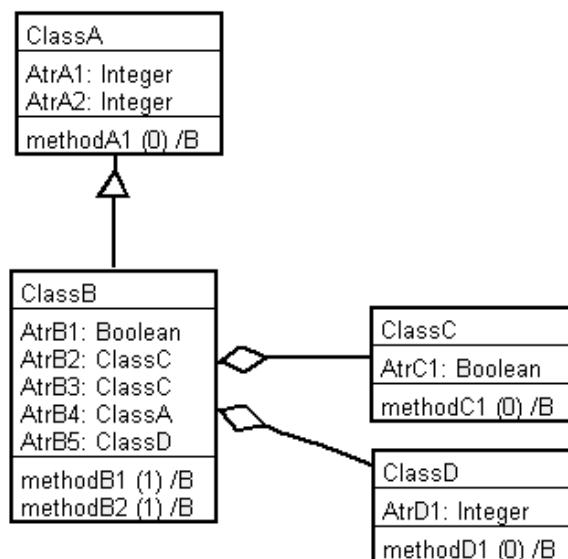


FIGURA 1 - Diagrama de classe – Exemplo de acoplamento

Detalhamento da assinatura dos métodos das classes:

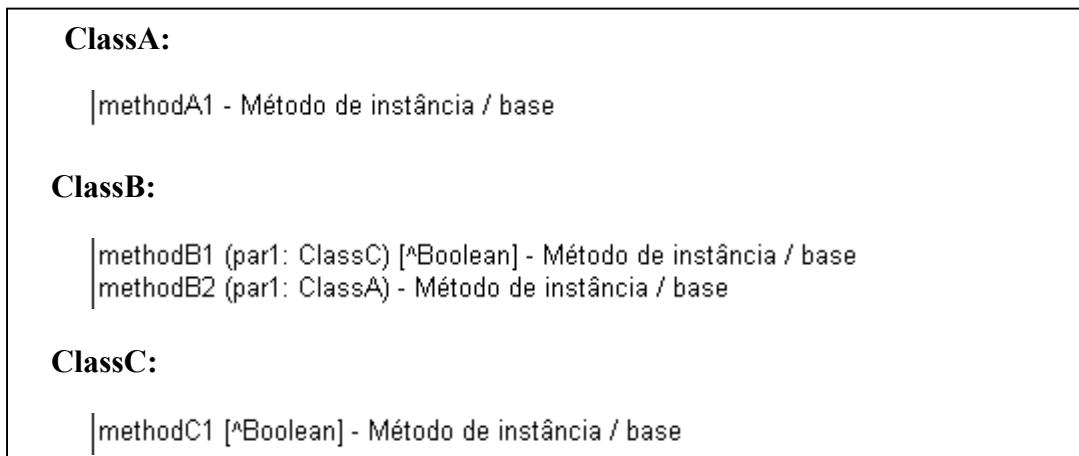


FIGURA 2 – Assinatura dos métodos – Exemplo de Acoplamento

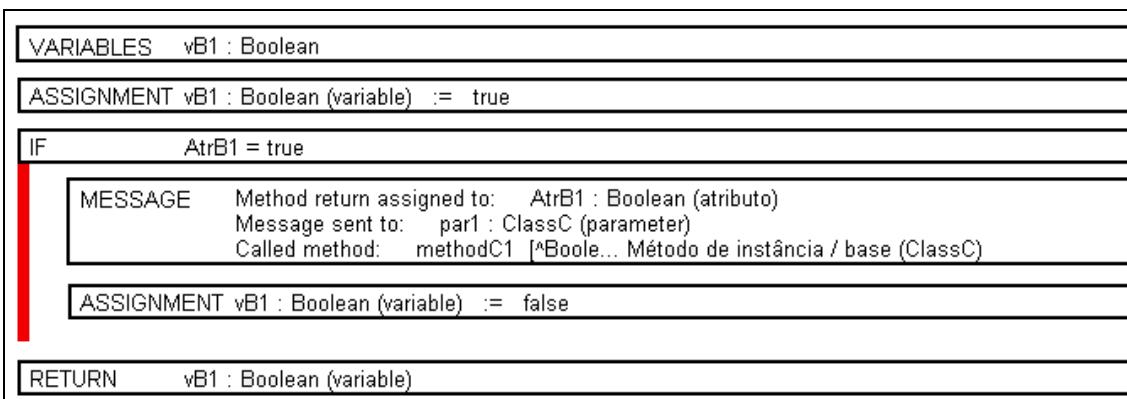


FIGURA 3 – Parte do diagrama de corpo de método do methodB1

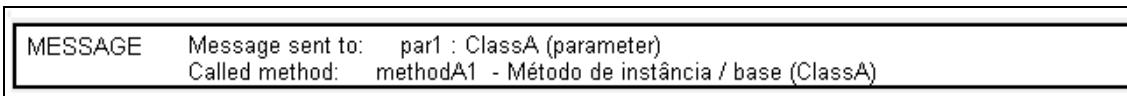


FIGURA 4 – Parte do diagrama de corpo de método do methodB2

Fator de Acoplamento – CF (*Coupling Factor*)

Com base no diagrama de classes da figura 1, e, usando a fórmula de CF redefinida em [ABR 95], pode-se obter os seguintes valores para CF:

é_Cliente(B,C)	1, pois a classe <i>ClassB</i> é cliente da classe <i>ClassC</i> , através dos atributos <i>AtrB2</i> , <i>AtrB3</i> , e parâmetro do método <i>methodB1(par1: ClassC)[^Boolean]</i>
é_Cliente(B,D)	1, pois a classe <i>ClassB</i> é cliente da classe <i>ClassD</i> , através do atributo <i>AtrB5</i>
$\sum_{i=1}^{TC} \left[\sum_{j=1}^{TC} \text{é_Cliente}(C_i, C_j) \right] = 2$	

TABELA 1 – Exemplo de Cálculo de CF

As demais combinações entre as classes, para verificar se uma classe é cliente da outra, não foram apresentadas, pois os seus valores são iguais a zero ou formam relação de herança, o que não é previsto pela métrica.

O valor de TC é igual a 4 e valor da expressão $2x\sum_{i=1}^{TC} DC(C_i)$ é 2, dessa forma o valor CF é 0,2. Esse resultado pode ser visualizado no gráfico a seguir:

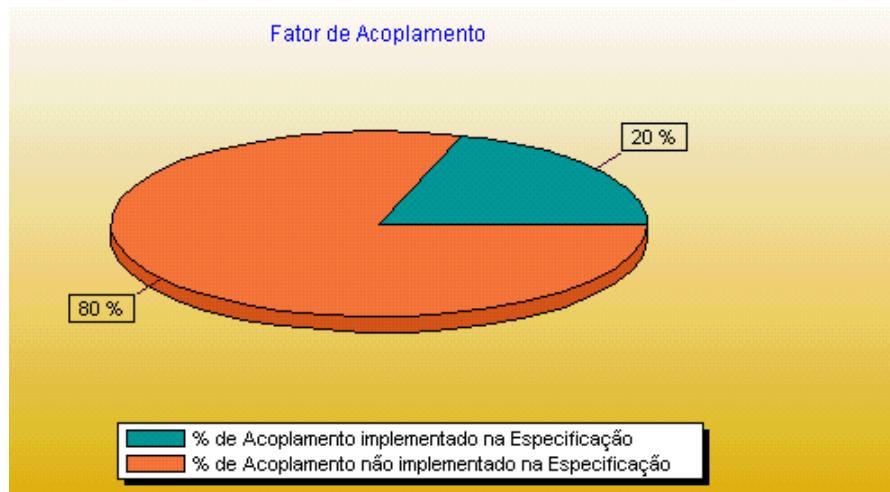


GRÁFICO 1 - Métrica CF

O gráfico anterior mostra que a especificação avaliada no exemplo, possui 20% de acoplamento implementado. Essas métricas visam quantificar a ausência ou presença do acoplamento, podendo variar de 0% (não uso ou ausência total) a 100% (máximo uso ou presença máxima possível). Para o modelo avaliado o valor de acoplamento não atinge 100%, pois existe uma relação de herança no modelo.

Acoplamento entre Classes de Objetos – CBO (Coupling Between Object Classes)

O cálculo da métrica demonstrado a seguir, é efetuado com base no diagrama de classes da figura 1.

CBO(A)	0, não possui referência de outras classes.
CBO(B)	3, a classe <i>ClassB</i> faz referência: 1 - a classe <i>ClassA</i> através do atributo <i>AtrB4</i> e parâmetro do método <i>methodB2(par1: ClassA)</i> 1 - a classe <i>ClassC</i> através dos atributos <i>AtrB2</i> , <i>AtrB3</i> , e parâmetro do método <i>methodB1(par1: ClassC)[^Boolean]</i> 1 - a classe <i>ClassB</i> faz referência à classe <i>ClassD</i> (através do atributo <i>AtrB5</i>)
CBO (C)	0, não possui referência de outras classes.
CBO (D)	0, não possui referência de outras classes.

TABELA 2 – Exemplo de Cálculo de CBO

No gráfico a seguir, é apresentado o valor de CBO para o exemplo apresentado:

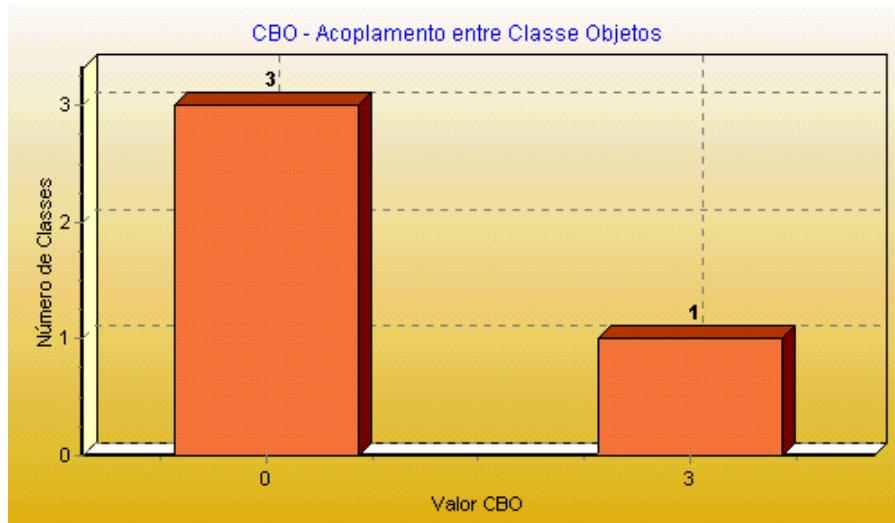


GRÁFICO 2 - Métrica CBO

Neste gráfico, pôde-se observar que 3 classes não fizeram referências a outras classes e uma classe fez referência a 3 outras classes.

Acoplamento por Classe-Atributo por Importação nos Ancestrais – ACAIC (*Ancestors Class-Attribute Import Coupling*)

Todos os exemplos de cálculos apresentados para as métricas de acoplamento, descritas a partir desta, também estão baseados no diagrama de classes da figura 1.

Exemplo de cálculo para ACAIC:

ACAIC(A)	0, não tem ancestral
ACAIC(B)	1, A classe <i>ClassB</i> faz referência à classe <i>ClassA</i> na forma de atributo.
ACAIC(C)	0, não tem classe ancestral
ACAIC(D)	0, não tem classe ancestral

TABELA 3 – Exemplo de Cálculo de ACAIC

No gráfico a seguir, é apresentado o valor de ACAIC para o exemplo apresentado:

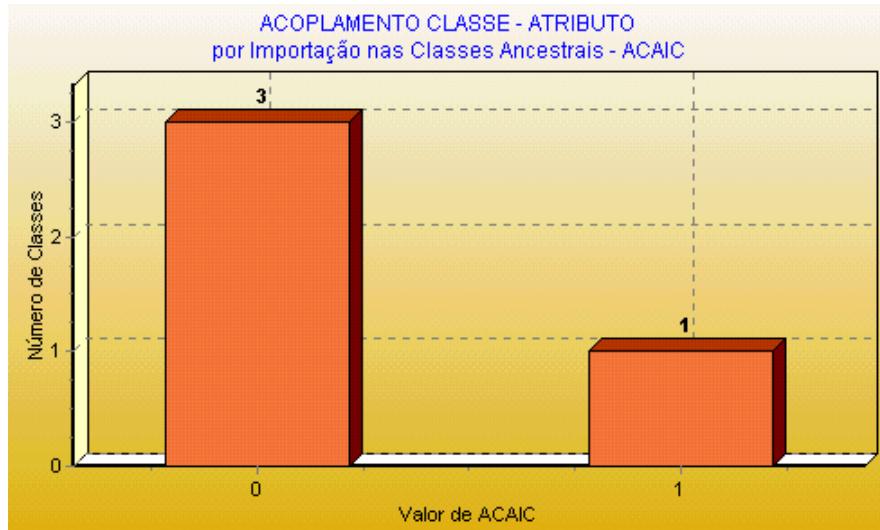


GRAFICO 3 – Métrica ACAIC

Neste gráfico, pode-se observar que 1 classe tem 1 dependência da classe ancestral, por outro lado, tem 3 classes que não possuem dependência das classes ancestrais. Dessa forma, a classe importará o impacto da mudança realizado na classe ancestral, pois existe interação entre as classes através de atributos.

Acoplamento por Classe-Atributo por Exportação nos Descendentes - DCAEC (Descendant Class-Attribute Export Coupling)

Exemplo de cálculo para DCAEC:

DCAEC(A)	1, A classe <i>ClassA</i> é referenciada na classe <i>ClassB</i> na forma de atributo.
DCAEC(B)	0, não tem descendente
DCAEC(C)	0, não tem descendente
DCAEC(D)	0, não tem descendente

TABELA 4 – Exemplo de Cálculo de DCAEC

O gráfico que mostra o resultado da métrica DCAEC não será apresentado, por ser semelhante ao gráfico da métrica ACAIC e gerar a mesma interpretação. Por esse motivo, os gráficos que são gerados para apresentar o resultado para as demais métricas que compõe o grupo de acoplamento não serão apresentados.

Acoplamento por Classe-Atributo por Importação entre Classes sem Relação de Herança – OCAIC (Others Class-Attribute Import Coupling)

Exemplo de cálculo para OCAIC:

OCAIC (A)	0
OCAIC (B)	3, a classe <i>ClassB</i> possui os seguintes atributos do tipo de classes sem relação de herança: <i>AttrB2: ClassC</i> , <i>AttrB3: ClassC</i> e <i>AttrB4: ClassD</i> .
OCAIC (C)	0
OCAIC (D)	0

TABELA 5 – Exemplo de Cálculo de OCAIC

Acoplamento por Classe-Atributo por Exportação entre Classes sem Relação de Herança - OCAEC (*Others Class-Attribute Export Coupling*)

Exemplo de cálculo para OCAEC:

OCAEC (A)	0
OCAEC (B)	0
OCAEC (C)	2, a classe <i>ClassC</i> foi exportada 2 vezes para outras classes na forma de atributos, em relação sem herança: <i>AtrB2: ClassC</i> (na classe <i>ClassB</i>) e <i>AtrB3: ClassC</i> (na classe <i>ClassB</i>).
OCAEC (D)	1, a classe <i>ClassC</i> foi exportada 1 vez para outra classe, na forma de atributo, em relação sem herança: <i>AtrB5: ClassD</i> (na classe <i>ClassB</i>)

TABELA 6 – Exemplo de Cálculo de OCAEC

Acoplamento Classe-Método por Importação nos Ancestrais – ACMIC (*Ancestors Class-Method Import Coupling*)

Exemplo de cálculo para ACMIC:

ACMIC(A)	0, não tem ancestral
ACMIC(B)	1, a classe <i>ClassB</i> possui um método com um parâmetro do tipo da classe ancestral <i>ClassA: MethodB2 (par1: ClassA)</i>
ACMIC(C)	0, não tem classe ancestral
ACMIC(D)	0, não tem classe ancestral

TABELA 7 – Exemplo de Cálculo de ACMIC

Acoplamento Classe-Método por Exportação nos Descendentes – DCMEC (*Descendant Class-Method Export Coupling*)

Exemplo de cálculo para DCMEC:

DCMEC(A)	1, a classe <i>ClassA</i> é referenciada na classe descendente <i>ClassB</i> na forma de parâmetro do método a seguir: <i>MethodB2 (par1: ClassA)</i>
DCMEC(B)	0, não tem descendente
DCMEC(C)	0, não tem descendente
DCMEC(D)	0, não tem descendente

TABELA 8 – Exemplo de Cálculo de DCMEC

Acoplamento Classe-Método por Importação entre Classes sem Relação de Herança – OCMIC (*Others Class-Method Import Coupling*)

Exemplo de cálculo para OCMIC:

OCMIC (A)	0
OCMIC (B)	1, a classe <i>ClassB</i> possui o método <i>methodB1</i> que faz referência através de parâmetro, a uma classe não relacionada por herança, a classe <i>ClassB: MethodB1 (par1: ClassC)</i>
OCMIC (C)	0
OCMIC (D)	0

TABELA 9 – Exemplo de Cálculo de OCMIC

Acoplamento Classe-Método por Exportação entre Classes sem Relação de Herança – OCMEC (Others Class-Method Export Coupling)

Exemplo de cálculo para OCMEC:

OCMEC (A)	0
OCMEC (B)	0
OCMEC (C)	1, a classe <i>ClassC</i> é referenciada em um método da classe <i>ClassB</i> , na forma de parâmetro do método conforme a seguir: <i>MethodB1 (par1: ClassC)</i>
OCMEC (D)	0

TABELA 10 – Exemplo de Cálculo de OCMEC

Acoplamento Método-Método por Importação nos Ancestrais – AMMIC (Ancestors Method-Method Import Coupling)

Exemplo de cálculo para AMMIC:

AMMIC(A)	0, não tem ancestral
AMMIC(B)	1, no <i>methodB2</i> da classe <i>ClassB</i> é invocado o método <i>methodA1</i> , da classe <i>ClassA</i> .
AMMIC(C)	0, não tem classe ancestral
AMMIC(D)	0, não tem classe ancestral

TABELA 11 – Exemplo de Cálculo de AMMIC

Acoplamento Método-Método por Exportação nos Descendentes – DMMEC (Descendants Method-Method Export Coupling)

Exemplo de cálculo para DMMEC:

DMMEC(A)	1, um método da classe <i>ClassA</i> é invocado no corpo de método do <i>MethodB2</i> , da classe <i>ClassB</i> .
DMMEC(B)	0, não tem descendente
DMMEC(C)	0, não tem descendente
DMMEC(D)	0, não tem descendente

TABELA 12 – Exemplo de Cálculo de DMMEC

Acoplamento Método-Método por Importação entre Classes sem Relação de Herança – OMMIC (Others Method-Method Import Coupling)

Exemplo de cálculo para OMMIC:

OMMIC (A)	0
OMMIC (B)	1, na classe <i>ClassB</i> o método <i>methodB1</i> invoca o método <i>methodC1</i> , da classe <i>ClassC</i> .
OMMIC (C)	0
OMMIC (D)	0

TABELA 13 – Exemplo de Cálculo de OMMIC

Acoplamento Método-Método por Exportação entre Classes sem Relação de Herança – OMMEC (Others Method-Method Export Coupling)

Exemplo de cálculo para OMMEC:

OMMEC (A)	0
OMMEC (B)	0
OMMEC (C)	1, um método da classe <i>ClassC</i> é invocado no corpo de método do <i>MethodB1</i> , da classe <i>ClassB</i> .
OMMEC (D)	0

TABELA 14 – Exemplo de Cálculo de OMMEC

MÉTRICAS DE ENCAPSULAMENTO

Fator de Métodos Ocultos – MHF (Method Hiding Factor)

Para essa métrica não será apresentada a demonstração detalhada do cálculo, em função da simplicidade do mesmo, pois dado um exemplo de uma especificação hipotética, que possui 86 métodos, tendo 1 método oculto e 85 métodos visíveis. O cálculo é representado da seguinte forma:

$MHF = 1 / 86 = 0,0116$, onde o resultado de MHF será apresentado no gráfico a seguir:

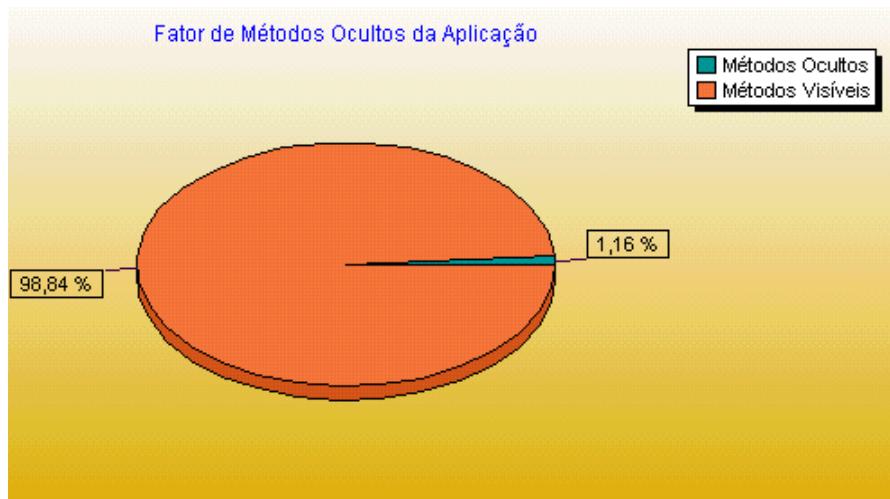


GRÁFICO 15 - Métrica: MHF

No gráfico anterior foi apresentado, o percentual de métodos ocultos e visíveis da especificação, onde 1,16% dos métodos da especificação são ocultos à classe e 98,84% dos métodos são visíveis a outras classes.

Essa métrica quantifica a ausência ou presença de encapsulamento de métodos e é expressa em percentual, podendo variar de 0% (não uso ou ausência total) a 100% (máximo uso ou presença máxima possível).

MÉTRICAS DE COMPLEXIDADE

Tamanho dos Métodos

O exemplo de cálculo para esta métrica não será apresentado, pois seria uma descrição muito grande que não agregaria muita contribuição ao trabalho.

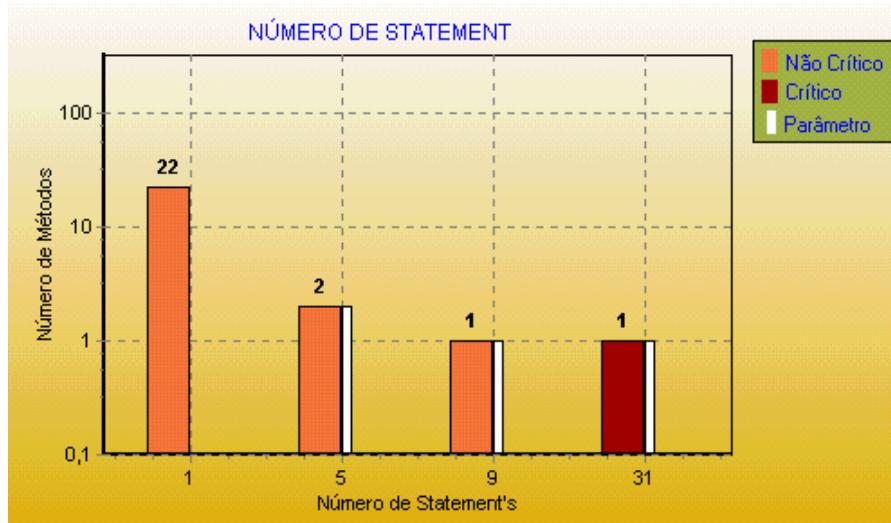


GRÁFICO 16 – Métrica: Tamanho dos Métodos

Na análise do gráfico anterior pôde-se extrair as seguintes informações:

1. Número de *statements* nos métodos

Na especificação avaliada existem 22 métodos que possuem 1 *statement*, 2 métodos que possuem 5, 1 método que possui 9 *statements* e 1 método que possui 31 *statements*.

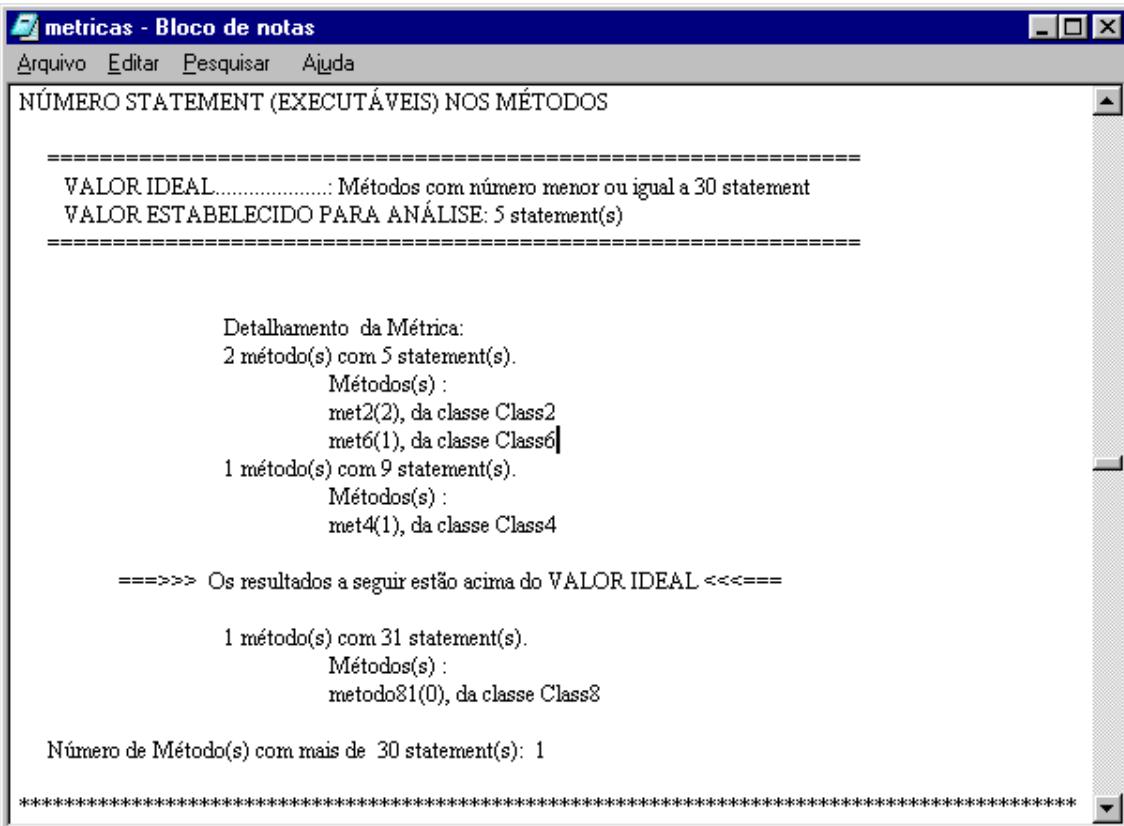
2. Número de métodos que possuem quantidade de *statements* acima de 30 *statements*

É possível visualizar no gráfico que somente 1 método possui 31 *statements*, ultrapassando assim o número de 30 *statements*, considerado neste trabalho como limite, ou seja, este método é considerado longo, devendo ser revisto. Este valor pode ser visto destacado em barra vermelha.

3. Número de métodos com número de *statements* igual ou maior que o valor indicado pelo usuário

É possível visualizar no gráfico que 4 métodos possuem 5 ou mais *statements*, caracterizando uma investigação sob algum patamar de interesse do usuário. Esses valores são destacados com uma barra lateral branca.

A interpretação da métrica pode ser feita também através do resultado em formato texto, conforme apresentado a seguir:



```

metricas - Bloco de notas
Arquivo Editar Pesquisar Ajuda
NÚMERO STATEMENT (EXECUTÁVEIS) NOS MÉTODOS

=====
VALOR IDEAL.....: Métodos com número menor ou igual a 30 statement
VALOR ESTABELECIDO PARA ANÁLISE: 5 statement(s)
=====

Detalhamento da Métrica:
2 método(s) com 5 statement(s).
Métodos(s):
met2(2), da classe Class2
met6(1), da classe Class6
1 método(s) com 9 statement(s).
Métodos(s):
met4(1), da classe Class4

====>>> Os resultados a seguir estão acima do VALOR IDEAL <<<=====

1 método(s) com 31 statement(s).
Métodos(s):
metodo81(0), da classe Class8

Número de Método(s) com mais de 30 statement(s): 1
=====

```

FIGURA 5 – Exemplo relatório: Tamanho dos Métodos

No relatório da figura 5 optou-se por mostrar somente o número de métodos com quantidade de *statements* igual ou maior que o valor estipulado pelo usuário. Dessa forma são apresentados somente os métodos que possuem 5 ou mais *statements*. São destacados ainda, os métodos que possuem mais de 30 *statements*, considerado neste trabalho como limite ideal, ou seja, este método é considerado longo, devendo ser revisto.

Número de Argumentos no Método

Para essa métrica não será apresentado o cálculo, considerado bastante elementar. O gráfico a seguir, apresenta o número de argumentos dos métodos para um dado exemplo:

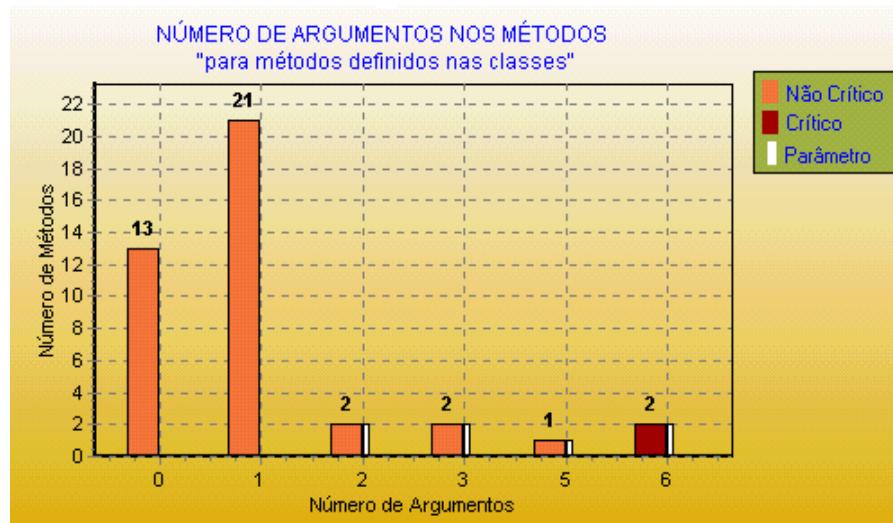


GRÁFICO 17 – Métrica: Número de Argumentos nos Métodos

Na análise do gráfico 17, pode-se extrair as seguintes informações:

1. Número argumentos nos métodos

Existem 13 métodos sem argumentos, 21 métodos com 1 argumento, 2 métodos com 2 argumentos, 2 métodos com 3 argumentos, 1 método com 5 argumentos e, assim, sucessivamente.

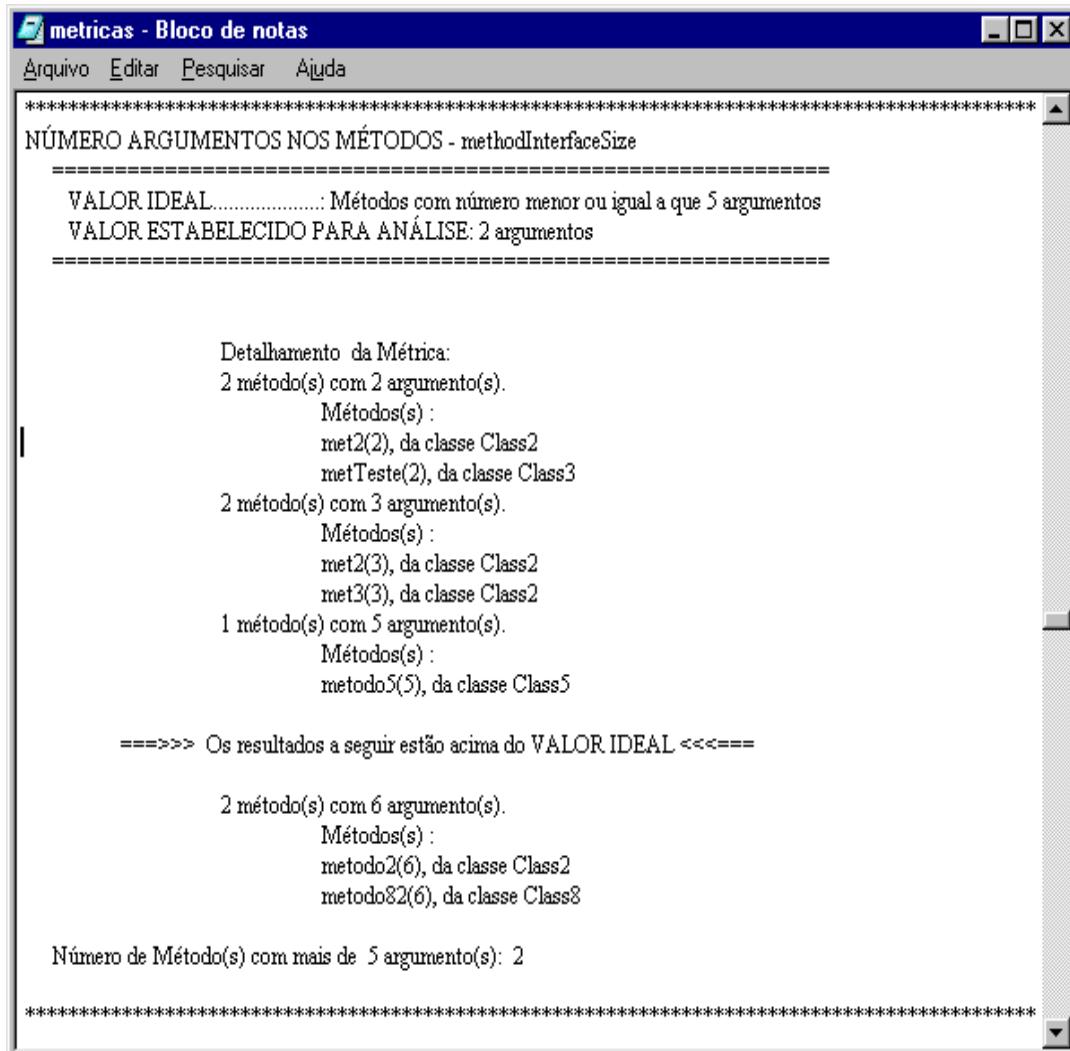
2. Número de métodos que possuem quantidade de argumentos acima do valor sugerido por Johnson [JOH 88]

É possível visualizar no gráfico 2 métodos que possuem 6 argumentos, ultrapassando assim, o valor sugerido por Johnson [JOH 88]. Estes métodos devem ser reavaliados para viabilizar a construção de novos métodos com menor número de argumentos. Esses valores são destacados em barras vermelhas.

4. Número de métodos com número de argumentos igual ou maior que o valor indicado pelo usuário

É possível visualizar no gráfico 7 métodos que possuem 2 ou mais argumentos, caracterizando uma investigação sob algum patamar de interesse do usuário. Esses valores são destacados com uma barra lateral branca.

A interpretação da métrica pode ser feita através do resultado em formato textual, apresentado a seguir:



```

metricas - Bloco de notas
Arquivo  Editar  Pesquisar  Ajuda
=====
NÚMERO ARGUMENTOS NOS MÉTODOS - methodInterfaceSize
=====
VALOR IDEAL.....: Métodos com número menor ou igual a que 5 argumentos
VALOR ESTABELECIDO PARA ANÁLISE: 2 argumentos
=====

Detalhamento da Métrica:
2 método(s) com 2 argumento(s).
Métodos(s) :
met2(2), da classe Class2
metTeste(2), da classe Class3
2 método(s) com 3 argumento(s).
Métodos(s) :
met2(3), da classe Class2
met3(3), da classe Class2
1 método(s) com 5 argumento(s).
Métodos(s) :
metodo5(5), da classe Class5

====>>> Os resultados a seguir estão acima do VALOR IDEAL <<<====

2 método(s) com 6 argumento(s).
Métodos(s) :
metodo2(6), da classe Class2
metodo82(6), da classe Class8

Número de Método(s) com mais de 5 argumento(s): 2
=====
```

FIGURA 6— Exemplo de relatório: Número de Argumentos nos Métodos

No relatório da figura 6 são apresentados somente os métodos que possuem 2 ou mais argumentos, em função do valor estabelecido para análise.

Número de Métodos nas Classes – NOM (*Number Of Methods*)

O exemplo de cálculo para essa métrica não será apresentado, assim como não foi apresentado para as métricas: *Tamanho de Métodos* e *Número de Argumento no Método*.

O gráfico a seguir, apresenta o número de métodos disponíveis nas classes de um dado exemplo:

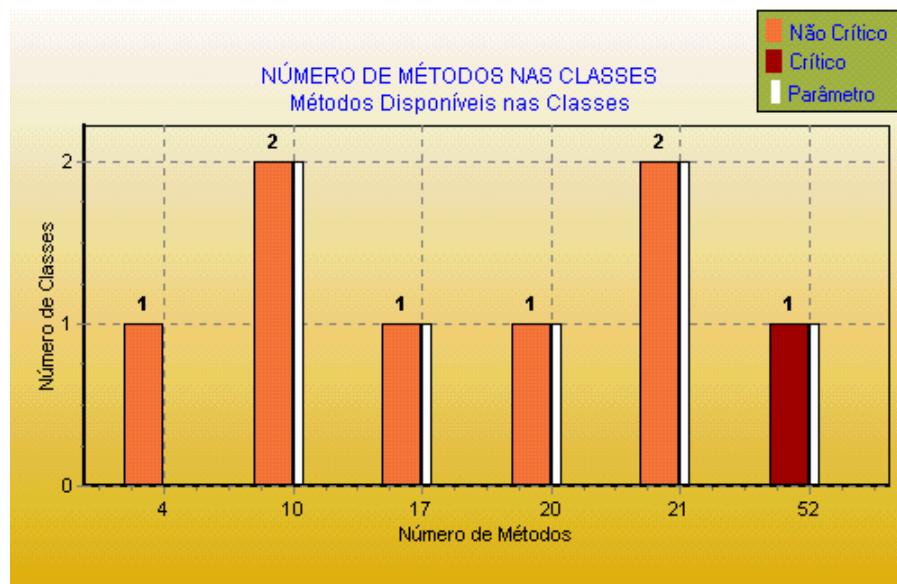


GRÁFICO 18 – Métrica: Número de Métodos nas Classes (Disponíveis)

Na análise do gráfico 18 pode-se extrair as seguintes informações:

1. Número de métodos nas classes

É possível visualizar que 1 classe possui 4 métodos, 2 classes possuem 10 métodos, 1 classe possui 17 métodos, 1 classe possui 20 métodos e assim sucessivamente.

2. Número de classes que possuem quantidade de métodos acima do valor sugerido por Johnson [JOH 88]

É possível visualizar no gráfico 1 classe que possui 52 métodos, ultrapassando assim, a quantidade de métodos sugerida por Johnson [JOH 88]. Essa classe deve ser reavaliada, pois isto é indicativo de possibilidade de subdividida. Esse valor é destacado com uma barra vermelha.

3. Número de classes com número de métodos igual ou maior que o valor indicado pelo usuário

É possível visualizar no gráfico anterior que 7 classes possuem 10 ou mais métodos, caracterizando uma investigação sob algum patamar de interesse do usuário. Esses valores são destacados com uma barra lateral branca.

O gráfico a seguir apresenta o número de métodos definidos nas próprias classes, utilizando o mesmo modelo de classes que gerou o resultado do gráfico anterior:

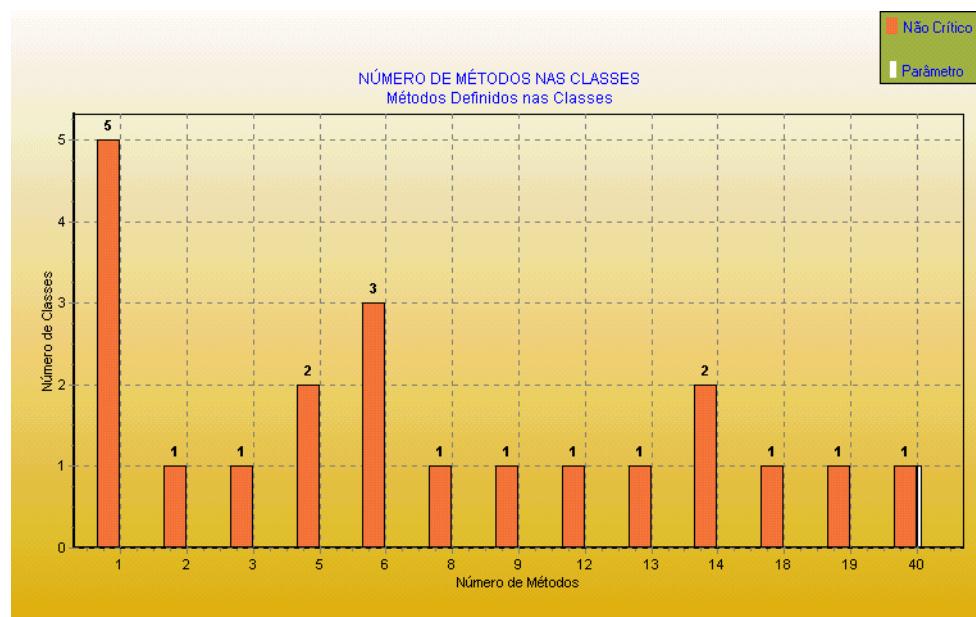


GRÁFICO 19 - Métrica: Número de Métodos nas Classes (Definidos)

Na análise do gráfico 19 pode-se extrair as seguintes informações:

1. Número de métodos nas classes

É possível visualizar que 5 classes possuem 1 método, 1 classe possui 2 métodos, 1 classe possui 2 métodos, 2 classes possuem 5 métodos e assim sucessivamente.

2. Número de classes com número de métodos igual ou maior que o valor indicado pelo usuário

É possível visualizar no gráfico que 1 classe possui 35 ou mais métodos, caracterizando uma investigação sob algum patamar de interesse do usuário. Esses valores são destacados com uma barra lateral branca.

Número de Classes Imediatamente Descendentes – NOC (Number of Children)

Como foi utilizado *framework* de jogos de tabuleiro [SIL 97] para extrair esta métrica, então o cálculo não será demonstrado:

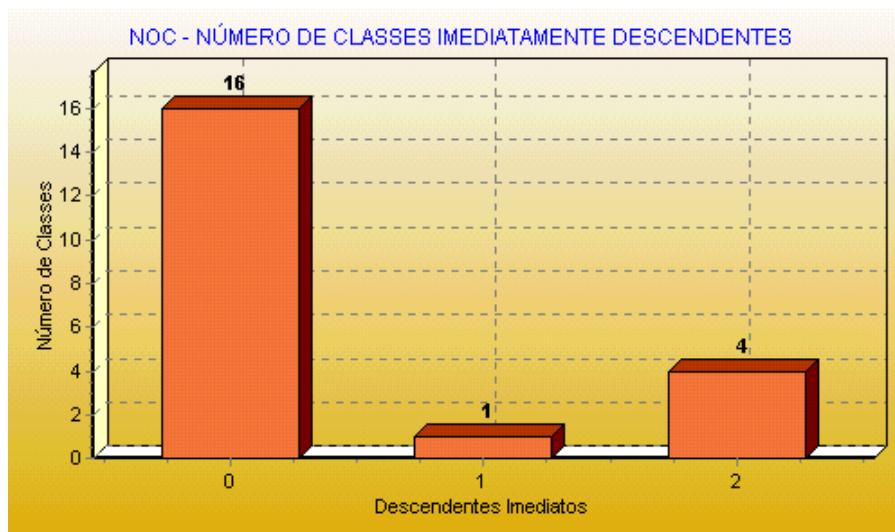


GRÁFICO 20 – Métrica: NOC

O resultado desta avaliação mostra número de classes e seus respectivos números de descendentes imediatos. Existem: 16 classes sem descendentes imediatos, 1 classe com 1 descendente e 4 classes com 2 descendentes.

Profundidade da Árvore de Herança – DIT (Depth of Inheritance Tree)

Para esta métrica não será representado o cálculo, pois a especificação utilizada como referência foi *framework* para jogos de tabuleiro, que possui um tamanho relativamente grande.

No gráfico a seguir é apresentado o valor de DIT.

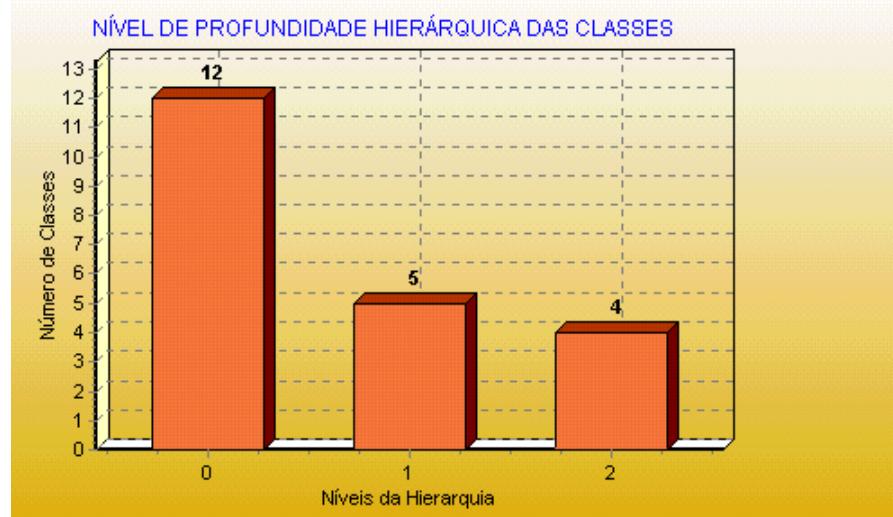


GRÁFICO 21 – Métrica: DIT

O gráfico anterior mostra que existem: 12 classes no nível de hierarquia 0, ou seja, não têm uma superclasse, 5 classes no nível 1 (primeiro nível de herança), 4 classe no nível 2 (segundo nível de herança). O maior nível de profundidade hierárquica da árvore da especificação avaliada é 2.

Fator de Herança de Métodos – MIF (Method Inheritance Factor)

O diagrama de classes da figura 7 não é um diagrama que representa uma aplicação real e será utilizado para exemplificar o cálculo de algumas métricas.

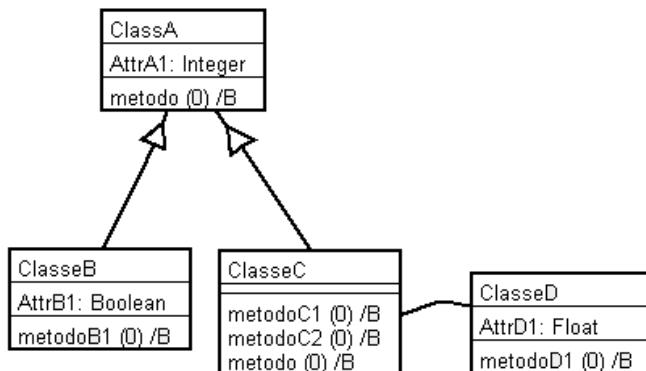


FIGURA 7: Diagrama de Classe – Exemplo MIF

Classe	Mi	Ma
ClasseA	0	1
ClasseB	1	2
ClasseC	0	3
ClasseD	0	1
Total	1	7

TABELA 15 – Exemplo de Cálculo de MIF

$$MIF = 1/7=0,14$$

Embora o autor da métrica tenha afirmado que esta métrica pode ser expressa em porcentagens, que variam de 0% (não uso) a aproximadamente 100% (máximo uso) [ABR 95], o resultado dessa métrica não pode atingir o valor 100%, onde o denominador é a soma do numerador (número de métodos herdados) mais o número de métodos definidos na classe.

O valor de MIF é apresentado no gráfico 22 e, com base no gráfico, o projetista poderá avaliar o fator utilização de herança dos métodos das classes da especificação.

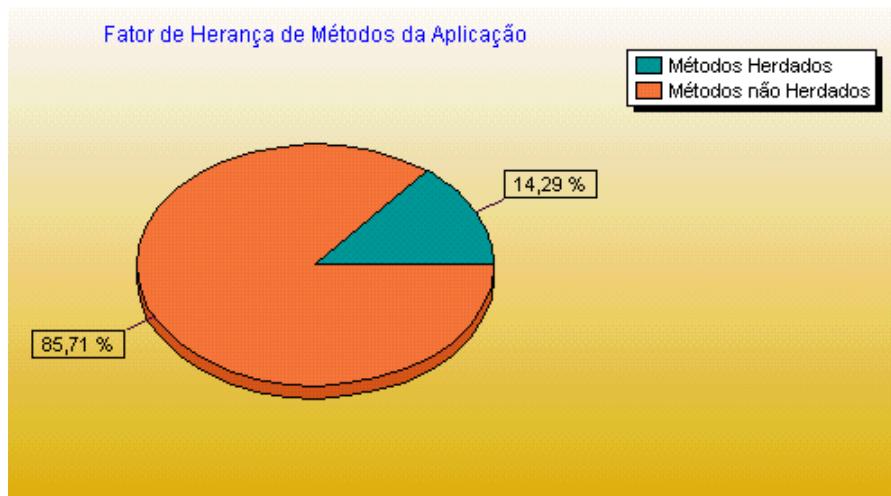


GRÁFICO 22 – Métrica: MIF

O gráfico anterior mostra que na especificação avaliada, somente 14,29% de aproveitamento de código dos métodos herdados (e não sobreescritos) pelas classes e 85,71% dos métodos sem reaproveitamento do código.

Fator de Herança de Atributos – AIF (Attribute Inheritance Factor)

Com base no diagrama de classes da figura 7, pode se obter o seguinte:

Classe	Ai	Aa
ClasseA	0	1
ClasseB	1	2
ClasseC	1	1
ClasseD	0	1
Total	2	5

TABELA 16 – Exemplo de Cálculo de AIF

$$AIF = 2/5=0,40$$

Assim como a métrica AIF, o valor dessa métrica não irá atingir o valor 100%.

O valor de AIF é apresentado no gráfico a seguir:

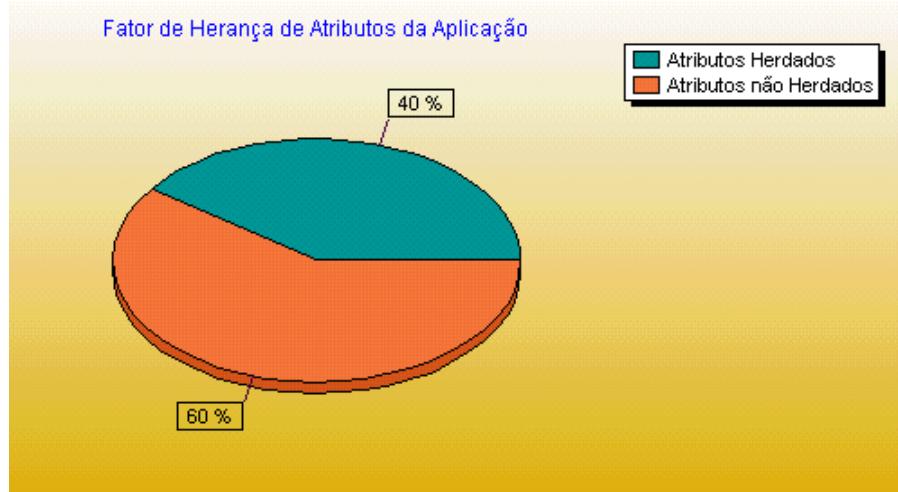


GRÁFICO 23 – Métrica: AIF

No gráfico anterior foi apresentado o percentual de atributos herdados na especificação, sendo que 40% dos atributos são herdados, isto é, são definidos em uma classe e usados em mais de uma classe.

Reação de uma Classe – RFC (Response For a Class)

Com base no diagrama de classes da figura 1, pode-se obter os seguintes valores para RFC:

Classe	$\{M\}$	$\{R_i\}$	RS
ClassA	1, método: methodA1	0, pois o método <i>methodA1</i> não invoca outros métodos	1
ClassB	2, métodos: methodB1 methodB2	2, pois 1 - no corpo de método de <i>methodB1</i> , é invocado o método <i>methodC1</i> . 1 - no corpo de método de <i>methodB2</i> , é invocado o método <i>methodA1</i>	4
ClassC	1, método: methodC1	0, pois <i>methodC1</i> não invoca outros métodos	1
ClassD	1, método: methodD1	0, pois <i>methodD1</i> não invoca outros métodos	1

TABELA 17 – Exemplo de Cálculo de RCF

O gráfico a seguir apresenta o número de RFC para as classes em relação ao exemplo anteriormente apresentado.

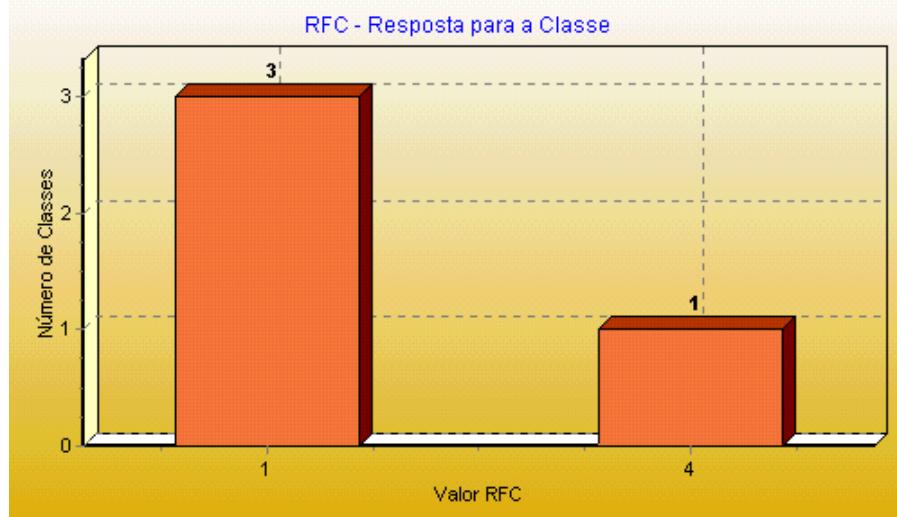


GRÁFICO 24 – Métrica: RFC

No gráfico anterior foi apresentado o seguinte resultado: 3 classes possuem 1 método como o conjunto de resposta para a classe e 1 classe possui 4 métodos que compõem o conjunto de respostas para a classe.

Métodos Complexos por Classe - WMC (Weighted Method Count)

O cálculo dessa métrica é similar ao cálculo da métrica *Número de Métodos na Classe*, porém, para cada método é avaliada a complexidade do método em função do número de *statements*.

No gráfico a seguir é apresentado o valor de WMC, para especificação de um *framework* de jogos de tabuleiro.

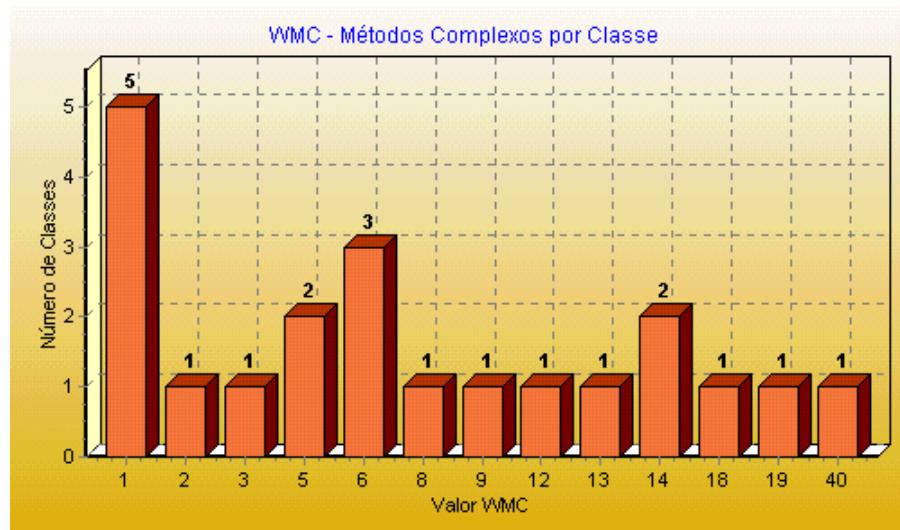


GRÁFICO 25 – Métrica: WMC

No gráfico anterior foram apresentadas 5 classes que possuem WMC igual a 1, 1 classe com WMC igual a 2, e assim sucessivamente.

Referência à Subclasses

Com base no diagrama de classes da figura 7 tem-se o seguinte resultado para a métrica:

Classe	Referência
ClasseA	0, não possuí referência a subclasse
ClasseB	0, não tem subclasse
ClasseC	0, não tem subclasse
ClasseD	0, não tem subclasse

Dessa forma, constata-se que nenhuma classe referenciou uma subclasse.

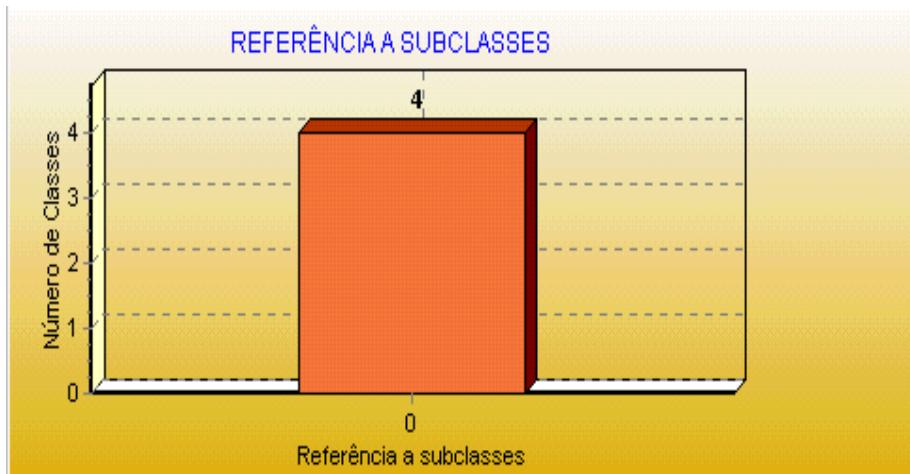


GRÁFICO 26 – Métrica: Referência à Subclasses

MÉTRICA DE COESÃO

Falta de Coesão nos Métodos – LCOM (Lack of Cohesion in Methods)

O diagrama de classes descrito a seguir, será usado para exemplificar o cálculo da métrica.

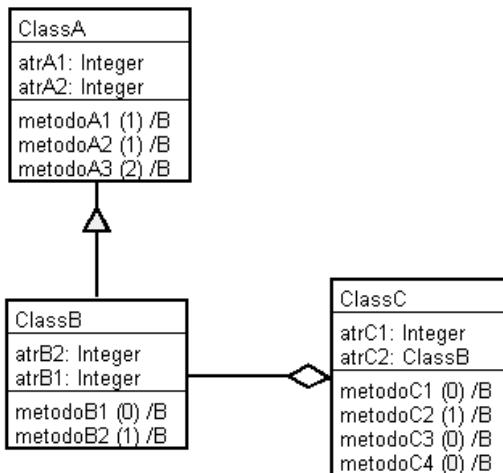


FIGURA 8 - Diagrama de Classes – Exemplo LCOM

Detalhamento da assinatura dos métodos das classes:

```

ClassA:
| metodoA1 (pA1: Integer) - Método de instância / base
| metodoA2 (pA1: Integer) - Método de instância / base
| metodoA3 (pA1: Integer, pA2: Integer) - Método de instância / base

ClassB:
| metodoB1 - Método de instância / base
| metodoB2 (ParB: Integer) - Método de instância / base

ClassC:
| metodoC1 [^Integer] - Método de instância / base
| metodoC2 (parC: Integer) - Método de instância / base
| metodoC3 - Método de instância / base
| metodoC4 [^Integer] - Método de instância / base

ClassD:
| methodD1 - Método de instância / base
  
```

FIGURA 9 – Assinatura dos métodos – Exemplo LCOM

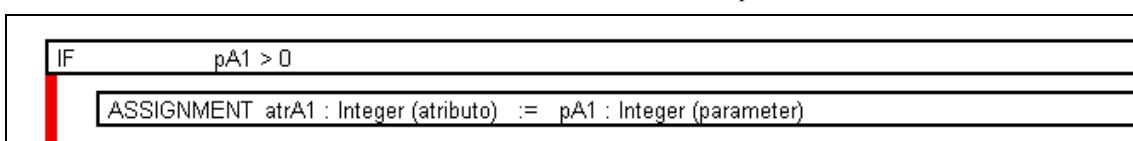


FIGURA 10 - Parte do diagrama de corpo de método metodoA1

```

VARIABLES  vA : Integer
ASSIGNMENT vA : Integer (variable) := 50
IF          pA1 = 0
  ASSIGNMENT atrA2 : Integer (atributo) := pA1 : Integer (parameter)
ELSE
  ASSIGNMENT atrA2 : Integer (atributo) := vA : Integer (variable)

```

FIGURA 11 – Parte do diagrama de corpo de método metodoA2

```

IF          pA1 > pA2
  ASSIGNMENT atrA2 : Integer (atributo) := pA2 : Integer (parameter)
  ASSIGNMENT atrA1 : Integer (atributo) := pA1 : Integer (parameter)
ELSE
  ASSIGNMENT atrA2 : Integer (atributo) := pA1 : Integer (parameter)
  ASSIGNMENT atrA1 : Integer (atributo) := pA2 : Integer (parameter)

```

FIGURA 12 – Parte do diagrama de corpo de método metodoA3

```

MESSAGE  Message sent to: self
Called method:  metodoB2 (ParB: ... Método de instância / base (ClassB)
Parameter assignment: ; ParB: Integer -> atrB1 : Integer

```

FIGURA 13 – Parte do diagrama de corpo de método metodoB1

```

IF          ParB > atrB2
  ASSIGNMENT atrB2 : Integer (atributo) := ParB : Integer (parameter)
ELSE
  ASSIGNMENT atrB2 : Integer (atributo) := 0

```

FIGURA 14 – Parte do diagrama de corpo de método metodoB2

```

RETURN      atrC1 : Integer (atributo)

```

FIGURA 15 – Parte do diagrama de corpo de método do metodoC1

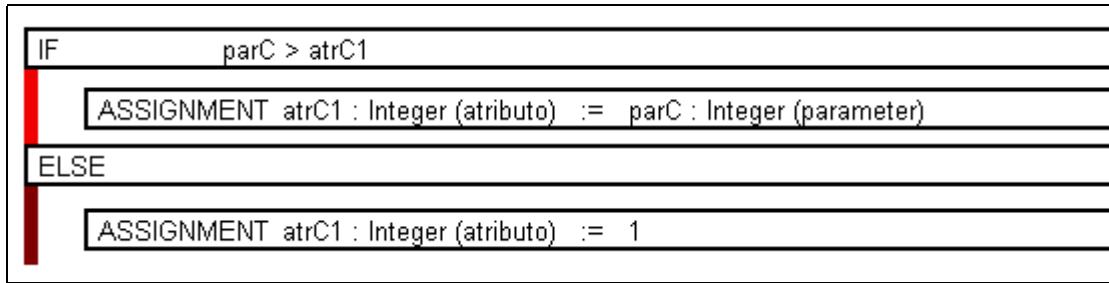


FIGURA 16 – Parte do diagrama de corpo de método metodoC2

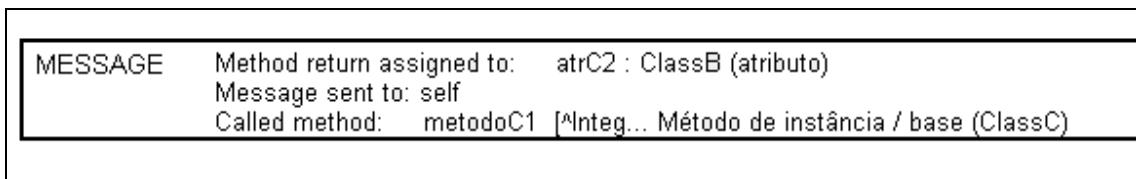


FIGURA 17 – Parte do diagrama de corpo de método metodoC3

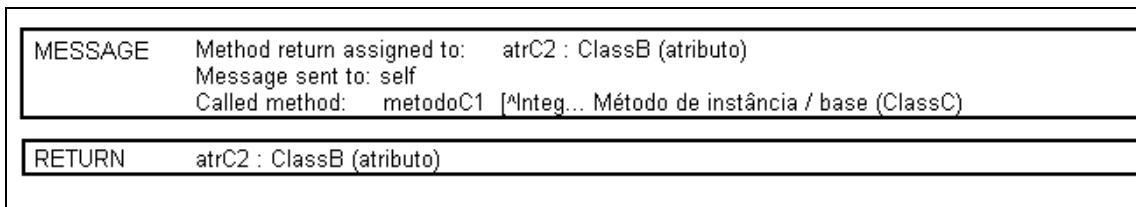


FIGURA 18 – Parte do diagrama de corpo de método metodoC4

Com base nos diagramas acima apresentados (figura 8 à figura 18), pode-se obter os seguintes valores para LCOM:

ClasseA	metodoA1={atrA1}={I1} metodoA2={atrA2}={I2} metodoA3={atrA1, atrA2}={I3} Tendo: $\{I1\} \cap \{I2\} = \emptyset$, $\{I1\} \cap \{I3\} \neq \emptyset$, $\{I2\} \cap \{I3\} \neq \emptyset$ $P=1$ e $Q=2$, como $P < Q$, então LCOM= 0
ClasseB	metodoB1={atrB1}={I1} metodoB2={atrB2}={I2} Tendo: $\{I1\} \cap \{I2\} = \emptyset$ $P=1$ e $Q=0$, como $P > Q$, então LCOM= 1
ClasseC	metodoC1={atrC1}={I1} metodoC2={atrC1}={I2} metodoC3={atrC2}={I3} metodoC4={atrC2}={I4} Tendo: $\{I1\} \cap \{I3\} = \emptyset$, $\{I2\} \cap \{I4\} = \emptyset$, $\{I2\} \cap \{I3\} = \emptyset$, $\{I2\} \cap \{I4\} = \emptyset$, $\{I1\} \cap \{I2\} \neq \emptyset$, $\{I3\} \cap \{I4\} \neq \emptyset$ $P=4$ e $Q=2$, como $P > Q$, então LCOM= 4 - 2 = 2

TABELA 18 – Exemplo de Cálculo de LCOM

Resultado exibido pela ferramenta:

No gráfico a seguir, é apresentado o valor de LCOM para o exemplo apresentado.

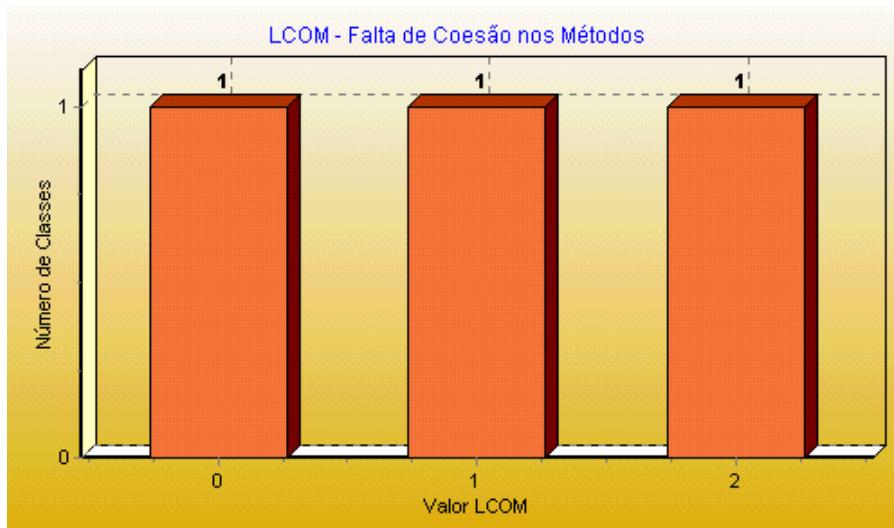


GRÁFICO 26 – Métrica: LCOM

No gráfico anterior foi apresentado que há 1 classe com LCOM igual a 0, 1 classe com LCOM igual a 1 e 1 classe com LCOM igual a 2. Valores baixos indicam coesão e valores altos indicam a falta de coesão.

MÉTRICAS DE POLIMORFISMO

Fator de Polimorfismo – PF (*Polymorphism Factor*)

O diagrama de classes, a seguir, foi elaborado para exemplificar o cálculo da métrica *Fator de Polimorfismo*.

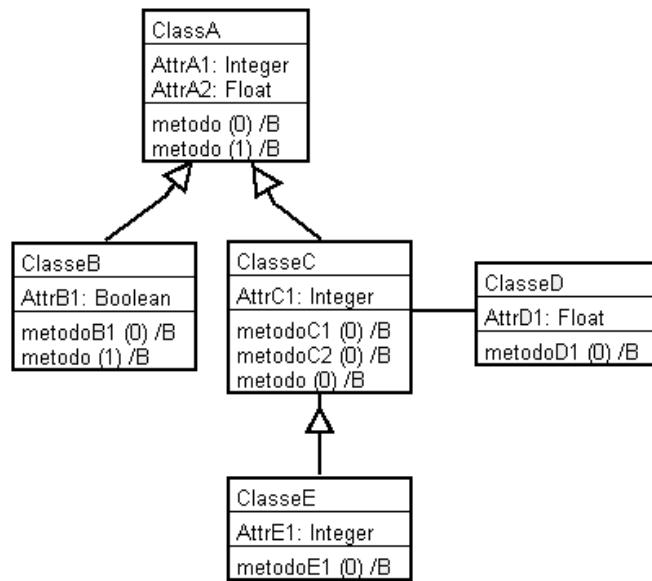


FIGURA 19 - Diagrama de classes - Exemplo PF

Com base no diagrama de classes da figura 19, o valor de PF é calculado da seguinte forma:

Classe	Mo	Mn x DC
ClasseA	0	(2 x 3)
ClasseB	1 método sobrescrito: <i>metodo (par1: Integer) da ClassA</i>	(1 x 0)
ClasseC	1 método sobrescrito: <i>metodo da ClassA</i>	(2 x 1)
ClasseD	0	(1 x 0)
ClasseE	0	(1 x 0)
Total	2	8

TABELA 19 – Exemplo de Cálculo de FP

$$\mathbf{PF} = 2/8=0,25$$

O resultado de PF é apresentado na ferramenta MET, no gráfico a seguir:

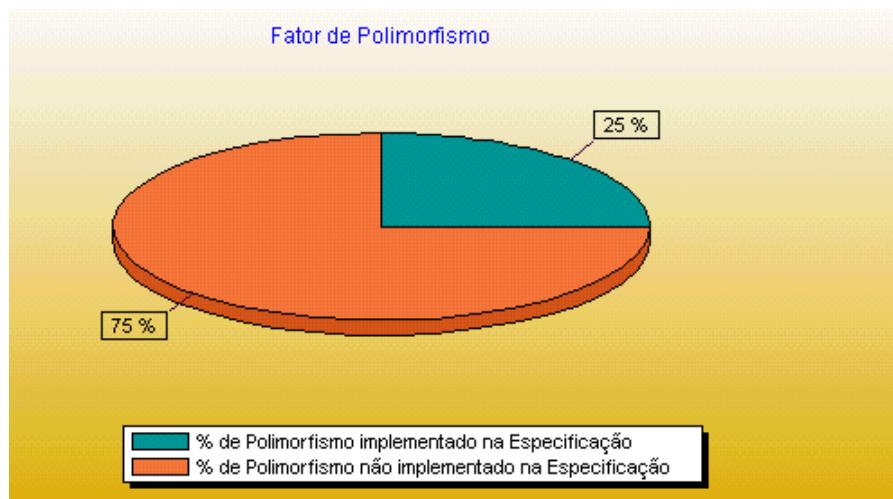


GRÁFICO 27 - Métrica PF

O gráfico acima mostra que na especificação avaliada no exemplo, tem-se 25% de polimorfismo implementado, sendo que 0% representa ausência do conceito de polimorfismo e 100% representa o máximo de uso de polimorfismo.

Polimorfismo Puro ou Sobrecarga em Classes Isoladas – OVO (*Overloading in Stand-Alone Classes*)

O diagrama de classes e a assinatura dos métodos descritos a seguir, serão usados para exemplificar o cálculo das demais métricas de polimorfismo.

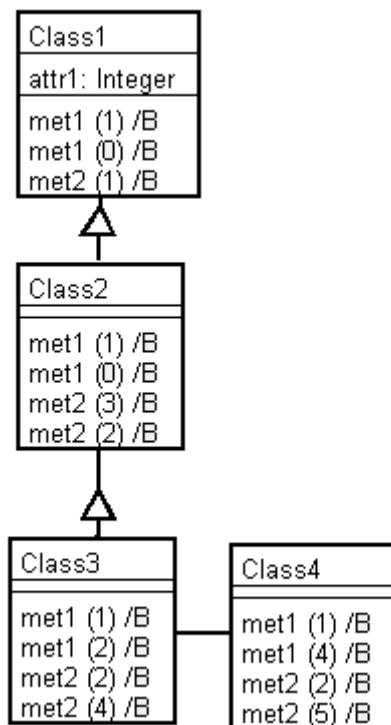


FIGURA 20 - Diagrama de classe – Exemplo de polimorfismo

Detalhamento da assinatura dos métodos das classes:

Class1:

```
met1 (par1: Integer) - Método de instância / base
met1 - Método de instância / base
met2 (par1: Integer) - Método de instância / base
```

Class2:

```
met1 (par1: Integer) - Método de instância / base
met1 - Método de instância / base
met2 (par1: Integer, par2: Float) - Método de instância / base
met2 (par1: Integer, par2: Float, par3: Boolean) - Método de instância / base
```

Class3:

```
met1 (par1: Integer) - Método de instância / base
met1 (par1: Integer, par2: Float) - Método de instância / base
met2 (par1: Integer, par2: Float) - Método de instância / base
met2 (par1: Integer, par2: Float, par3: Boolean, par4: Account) - Método de instância /
```

Class4:

```
met1 (par1: Integer) - Método de instância / base
met1 (par1: Integer, par2: Float, par3: Boolean, par4: Account) - Método de instância ,
```

FIGURA 21 – Assinatura dos métodos –Exemplo polimorfismo

Com base no diagrama de classes da figura 20, pode-se obter os seguintes valores para OVO:

OVO(Class1)	2, 1 função sobrecarregada 2 vezes <i>met1(par1: Integer)</i> <i>met1</i>
OVO (Class2)	4, 1 função sobrecarregada 2 vezes <i>met1(par1: Integer)</i> <i>met1(par1: Integer, par2:Float)</i> 1 função sobrecarregada 2 vezes <i>met2(par1: Integer, par2:Float)</i> <i>met2(par1: Integer, par2:Float, par3: Boolean)</i>
OVO (Class3)	4, 1 função sobrecarregada 2 vezes <i>met1(par1: Integer)</i> <i>met1</i> 1 função sobrecarregada 2 vezes <i>met2(par1: Integer, par2:Float)</i> <i>met2(par1: Integer, par2:Float, par3: Boolean, par4: Account)</i>
OVO (Class4)	4, 1 função sobrecarregada 2 vezes

	$met1(par1: Integer)$ $met1(par1: Integer, par2: Float, par3: Boolean, par4: Account)$ 1 função sobrecarregada 2 vezes $met2(par1: Integer, par2: Float)$ $met2(par1: Integer, par2: Float, par3: Boolean, par4: Account, par5: Integer)$
--	---

TABELA 20 – Exemplo de Cálculo de OVO

O resultado de OVO é apresentado da seguinte forma:

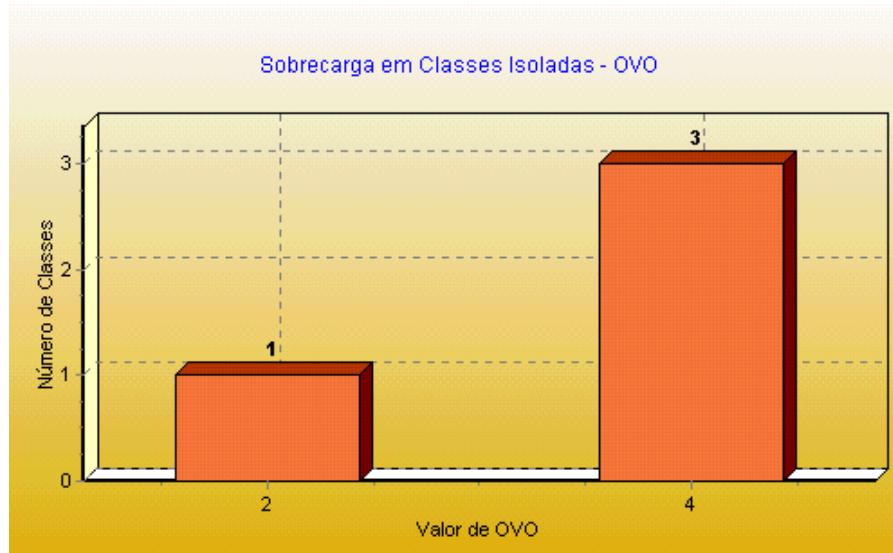


GRÁFICO 28 – Métrica OVO

Neste gráfico, pode-se observar que em todas as classes ocorre a característica de polimorfismo puro. Todas as classes têm chance de atender uma invocação de métodos, a partir de diferentes tipos de entrada de dados. Uma classe apresenta 2 sobrecargas de métodos, ou seja, 2 métodos com o mesmo nome e 3 classes com sobrecarga igual a 4.

Polimorfismo Estático nos Ancestrais - SPA (Static Polymorphism in Ancestors)

Com base no diagrama de classes da figura 20, pode-se obter os seguintes valores para SPA:

SPA(Class1)	0, pois classe <i>Class1</i> não tem ancestral.
SPA(Class2)	1, pois: $SPoly(Class1, Class2) = 1$ Pares: <i>met2</i> da <i>Class1</i> com <i>met2</i> da <i>Class3</i>
SPA(Class3)	4, pois: $SPoly(Class1, Class3) = 2$ Pares: <i>met1</i> da <i>Class1</i> com <i>met1</i> da <i>Class3</i> <i>met2</i> da <i>Class1</i> com <i>met2</i> da <i>Class3</i> $SPoly(Class2, Class3) = 2$ Pares: <i>met1</i> da <i>Class2</i> com <i>met1</i> da <i>Class3</i> <i>met2</i> da <i>Class2</i> com <i>met2</i> da <i>Class3</i>
SPA(Class4)	0, pois a classe <i>Class4</i> não tem ancestral.

TABELA 21 – Exemplo de Cálculo de SPA

No gráfico a seguir, é apresentado o resultado de SPA:

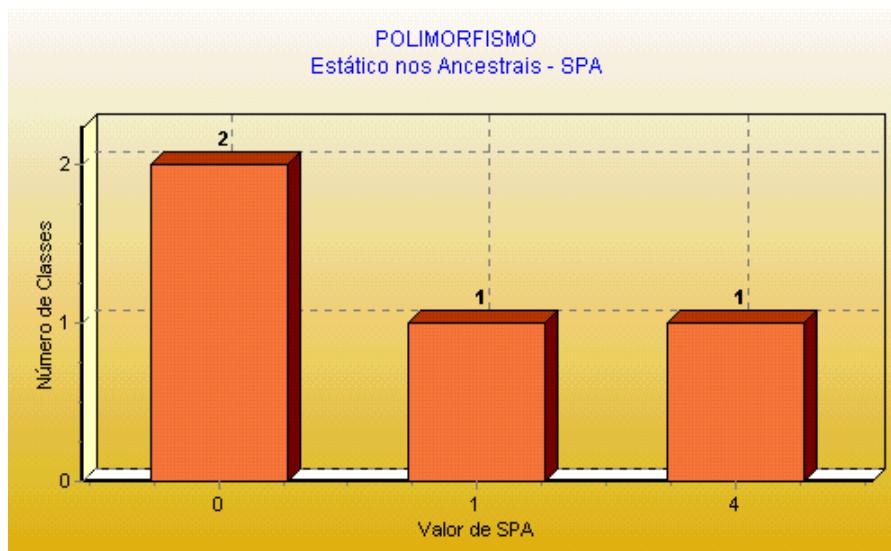


GRÁFICO 29 – Métrica SPA

Pode-se observar, através do gráfico, que em 2 classes não houve ocorrência de polimorfismo estático, que em 1 classe houve 1 polimorfismo estático (um método com mesmo nome e assinatura diferente) e assim sucessivamente.

Polimorfismo Estático nos Descendentes – SPD (Static Polymorphism in Descendents)

Com base no diagrama de classes da figura 20, pode-se obter os seguintes valores para SPD:

SPD(Class1)	3, pois: SPoly(Class2,Class1)=1 Pares: met2 da Class2 com met2 da Class1 SPoly(Class3,Class1)=2 Pares: met1 da Class3 com met2 da Class1 met2 da Class3 com met2 da Class1
SPD(Class2)	2, pois: SPoly(Class3, Class2) =2 Pares: met1 da Class2 com met1 da Class3 met2 da Class2 com met2 da Class3
SPD(Class3)	0, pois a classe Class3 não tem descendente.
SPD(Class4)	0, pois a classe Class4 não tem descendente.

TABELA 22 – Exemplo de Cálculo de SPD

No gráfico 30 é apresentado o valor de SPD:

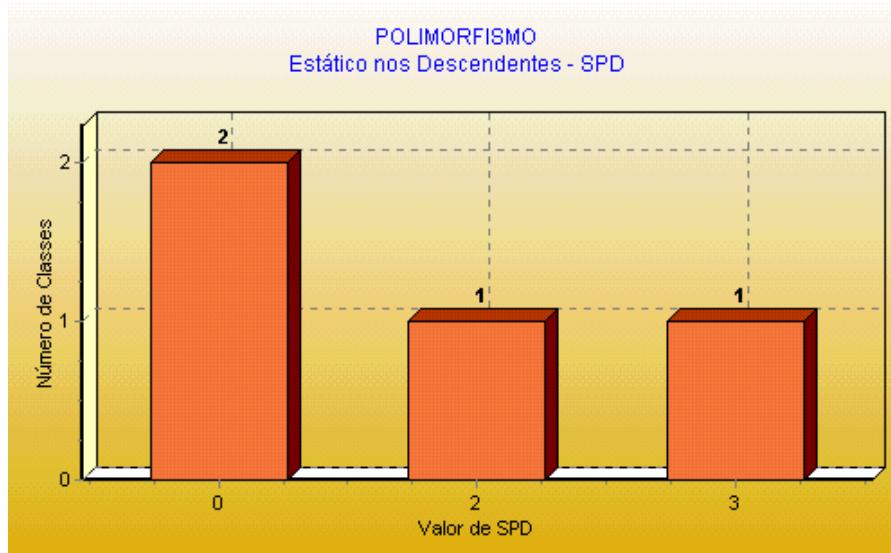


GRÁFICO 30 – Métrica SPD

Pôde-se observar no gráfico que em 2 classes não houve ocorrência de polimorfismo estático, em 1 classe houve 2 métodos polimórficos estáticos (2 métodos com mesmo nome e assinaturas diferentes) e assim sucessivamente.

Polimorfismo Estático em Relação de Herança – SP (Static Polymorphism)

A tabela, a seguir, mostra o detalhamento do cálculo de SP, que é em função do cálculo SPA e SPD apresentado anteriormente:

	SPA	SPD	SP
Class1	0	3	3
Class2	1	2	3
Class3	4	0	4
Class4	0	0	0

TABELA 23 – Exemplo de Cálculo de SP

No gráfico, a seguir, é apresentado o resultado de SP:

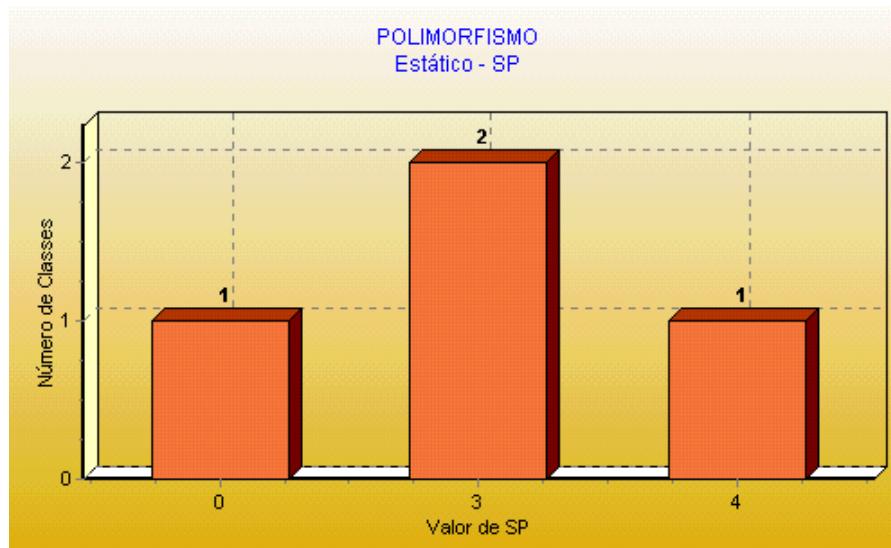


GRÁFICO 31 – Métrica SP

Pôde-se observar no gráfico, que em 1 classe não houve ocorrência de polimorfismo estático (nem em relação aos ancestrais e nem em relação aos descendentes), em 2 classes houve 3 ocorrências de polimorfismo estático e assim sucessivamente.

Polimorfismo Dinâmico nos Ancestrais – DPA (Static Polymorphism in Ancestors)

Com base no diagrama de classes da figura 20, pode-se obter os seguintes valores para DPA:

DPA(Class1)	0, pois a classe <i>Class1</i> não tem ancestral.
DPA(Class2)	2, pois: DPoly(Class1,Class2)=2 Pares: <i>met1 (par1: Integer)</i> da Class2 com <i>met1(par1: Integer)</i> da Class1 <i>met1</i> da Class2 com <i>met1</i> da Class1

DPA(Class3)	3, pois: DPoly(Class1, Class3)=1 Pares: met1 da Class1 com met1 da Class3 DPoly(Class2, Class3)=2 Pares: met1 da Class2 com met1 da Class3 met2(par1:Integer, par2:Float) da Class2 com met2(par1:Integer, par2:Float) da Class3
DPA(Class4)	0, pois a classe Class4 não tem ancestral.

TABELA 24 – Exemplo de Cálculo de DPA

O resultado de DPA é apresentado da seguinte forma:

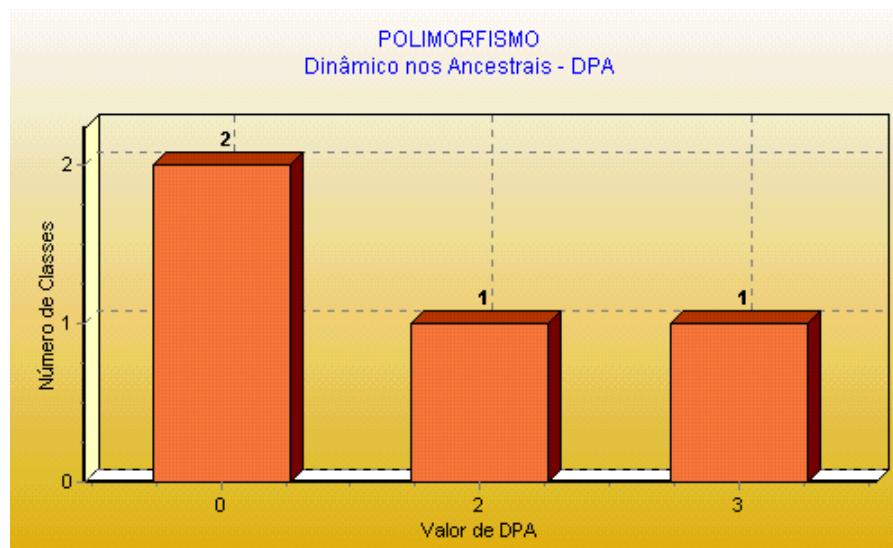


GRÁFICO 32 – Métrica DPA

Pôde-se verificar no gráfico anterior que em 2 classes não ocorreu polimorfismo dinâmico, em 1 classe houve polimorfismo dinâmico igual a 2 e assim sucessivamente.

Polimorfismo Dinâmico nos Descendentes – DPD (Dynamic Polymorphism in Descendents)

Com base no diagrama de classes da figura 20, pode se obter os seguintes valores para DPD:

DPD(Class1)	3, pois: DPoly(Class2, Class1)=2 Pares: met1 (par1: Integer) da Class2 com met1(par1: Integer) da Class1 met1 da Class2 com met1 da Class1 DPoly(Class3, Class1)=1 Pares: met1 da Class1 com met1 da Class3
DPD(Class2)	2, pois DPoly(Class3, Class2)=2 Pares: met1 da Class2 com met1 da Class3

	<i>met2(par1:Integer, par2:Float)</i> da Class2 com <i>met2(par1:Integer, par2:Float)</i> da Class3
DPD(Class3)	0, pois a classe <i>Class3</i> não tem descendente.
DPD(Class4)	0, pois a classe <i>Class4</i> não tem descendente.

TABELA 25 – Exemplo de Cálculo de DPD

No gráfico, a seguir, é apresentado o valor de DPD para cada classe do diagrama de classes apresentada na figura 20:

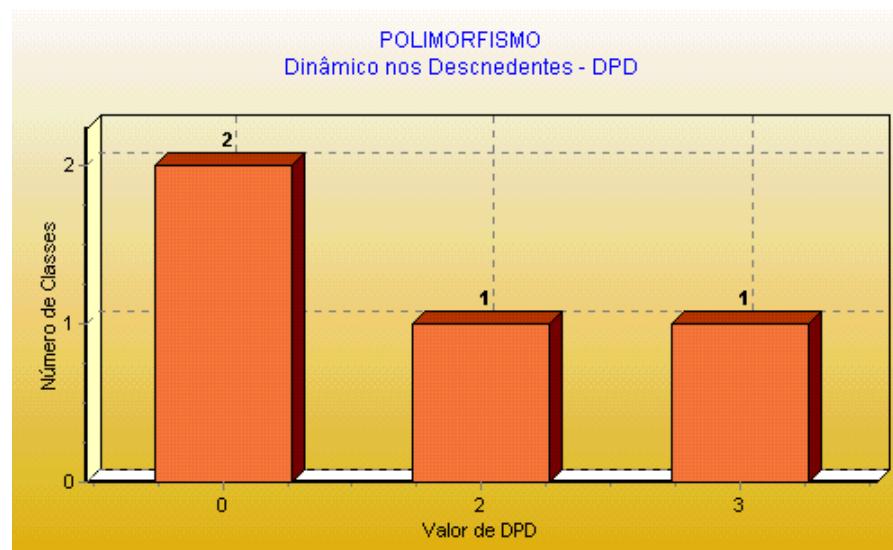


GRÁFICO 33 – Métrica DPD

Pode-se observar no gráfico anterior que em 2 classes que não houve ocorrência de polimorfismo dinâmico nos descendentes, que em 1 classe 2 métodos foram sobrescritos e assim sucessivamente.

Nota-se que os gráficos apresentados para o Polimorfismo Dinâmico nos Ancestrais e Polimorfismo Dinâmico nos Descendentes, apresentaram resultados visuais iguais, por se tratar somente de valores quantitativos, porém, as classes apresentadas são diferentes, como pôde ser observado na visualização hierárquica (outra forma de visualização na ferramenta desenvolvida e descrita neste trabalho), transcrita, a seguir:

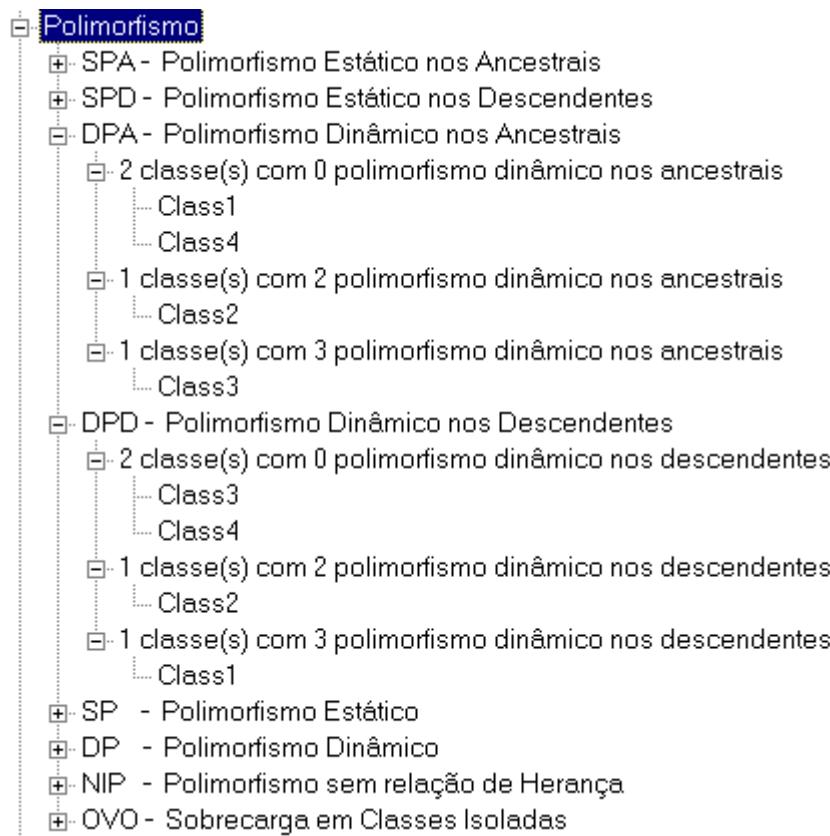


FIGURA 22 – Visualização hierárquica – Métrica DPD

Polimorfismo Dinâmico – DP (*Dynamic Polymorphism*)

O cálculo de DP é em função dos valores de DPA e DPD de cada classe do diagrama de classe da figura 20, conforme apresentado na tabela 26:

	DPA	DPD	DP
Class1	0	3	3
Class2	2	2	4
Class3	3	0	3
Class4	0	0	0

TABELA 26 – Exemplo de Cálculo de DP

No gráfico, a seguir, é apresentado o valor de DP:

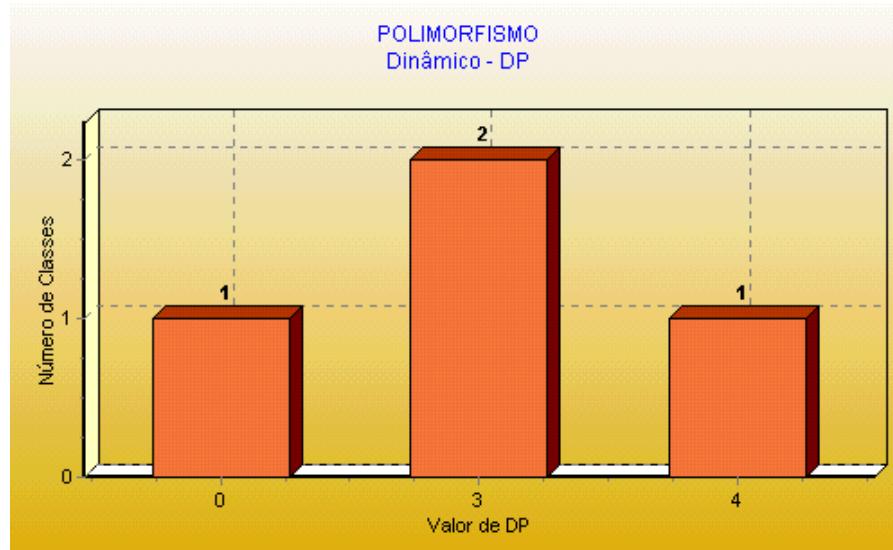


GRÁFICO 34 – Métrica DP

No gráfico anterior foi apresentada uma classe sem ocorrência de polimorfismo dinâmico (nem em relação aos ancestrais e nem em relação aos descendentes), em 2 classes 3 polimorfismos dinâmicos (3 métodos foram sobreescritos) e assim sucessivamente.

Polimorfismo numa relação sem herança – NIP (*Polymorphism in non-inheritance relations*)

Com base no diagrama de classes da figura 20, pode-se obter os seguintes valores para NIP:

NIP(Class1)	3 pois, DPoly(Class4,Class1)=1 Pares: met1 (par1: Integer) da Class4 com met1(par1: Integer) da Class1 SPoly(Class4,Class1)=2 Pares: met1 da Class4 com met1 da Class1 met2 da Class4 com met2 da Class1
NIP(Class2)	4 DPoly(Class4,Class2)=2 Pares: met1 (par1: Integer) da Class4 com met1(par1: Integer) da Class2 met2(par1:Integer, par2:Float) da Class4 com met2(par1:Integer, par2:Float) da Class2 SPoly(Class4,Class2)=2 Pares: met1 da Class4 com met1 da Class2 met2 da Class4 com met2 da Class2

NIP(Class3)	<p>4 $DPoly(Class4, Class3)=2$ Pares: $met1(par1: Integer)$ da Class4 com $met1(par1: Integer)$ da Class3 $met2(par1: Integer, par2: Float)$ da Class4 com $met2(par1: Integer, par2: Float)$ da Class3 $SPoly(Class4, Class3)=2$ Pares: $met1$ da Class4 com $met1$ da Class3 $met2$ da Class4 com $met2$ da Class3</p>
NIP(Class4)	<p>11, simplificadamente: $DPoly(Class1, Class4)=1$ $SPoly(Class1, Class4)=2$</p> <p>$DPoly(Class2, Class4)=2$ $SPoly(Class2, Class4)=2$</p> <p>$DPoly(Class3, Class4)=2$ $SPoly(Class3, Class4)=2$</p>

TABELA 27 – Exemplo de Cálculo de NIP

O valor de NIP é apresentado no gráfico a seguir:

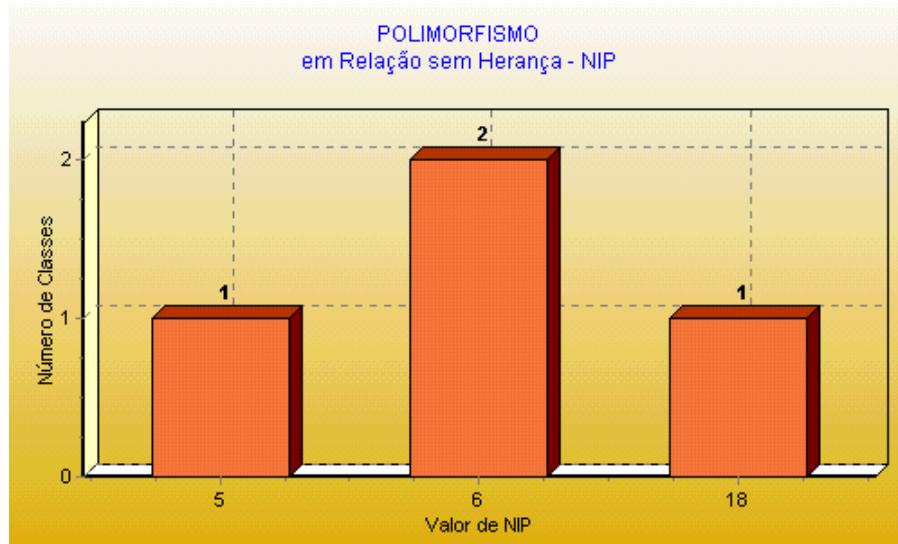


GRÁFICO 35 – Métrica NIP

No gráfico anterior observou-se 1 classe com valor de NIP igual a 3, e 2 classes com valor igual a 4 e 1 classe com valor igual a 11.

ANEXO 2 – EXEMPLOS DE DIAGRAMAS DE SEQÜÊNCIA DA FERRAMENTA MET

A figura 1 mostra o diagrama de seqüência para a operação *getTextAnalysis* da classe *MetricsExtractionTool*, que refina o caso de uso “*Procede Análise em Formato Texto*”. Esse método é inicialmente disparado pelo usuário através da interface da ferramenta MET, e é responsável pela criação da lista de ferramentas através do método *createToolLoad* da classe *AnalysisToolLoad*. Uma característica importante dessa estrutura é que a classe *MetricsExtractionTool* não precisa conhecer as ferramentas (subclasse que reúne um conjunto de métricas), tampouco a quantidade delas. Assim, podem ser inseridas ou removidas novas ferramentas, sem alterar o funcionamento da classe *MetricsExtractionTool*.

Após a instanciação das ferramentas, o *getTextAnalysis* invoca o método *produceTextAnalysis*, que é abstrato na classe *MetricsAnalyserTool* e será sobreposto em todas as ferramentas desenvolvidas. A invocação desse método nas classes concretas, que são subclasses da classe *MetricsAnalyserTool*, ocorre em outros diagramas que refinam o método *produceTextAnalysis*.

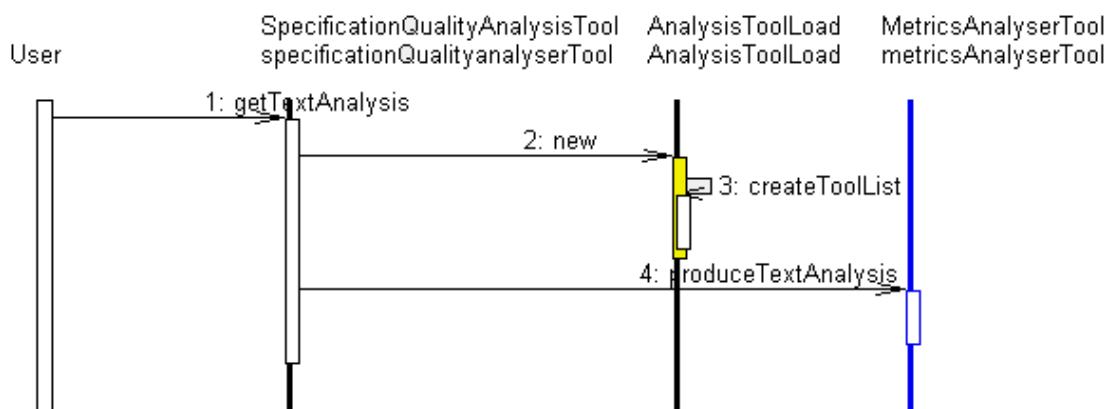


FIGURA 1 – Diagrama de Seqüência (MetricsExtractionTool / GetTextColorAnalysis)

A figura 2 mostra o diagrama de seqüência para a operação *getGraphicAnalysis* da classe *MetricsExtractionTool*, que refina o caso de uso “*Procede Análise em Formato Gráfico*”. Após este método ter sido disparado pela interface da ferramenta, a seqüência de ações a serem executadas é semelhante à descrita para o diagrama de seqüência anterior, exceto pelo método invocado, que neste caso é o método *produceGraphicAnalysis*.

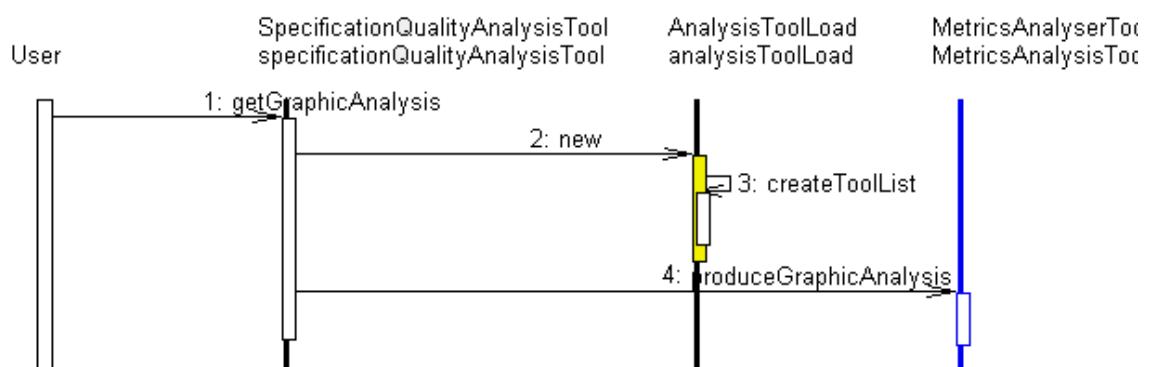


FIGURA 2 – Diagrama de Seqüência (MetricsExtractionTool / GetGraphicAnalysis)

A seguir, são apresentados diagramas de seqüência que mostram a execução dos métodos *procedureTextAnalysis* nas ferramentas *PolymorphismAnalysis*, *CouplingAnalysis*, *EncapsulationAnalysis* e *CohesionAnalysis*.

A figura 3 apresenta parte do diagrama de seqüência que refina o método *produceTextAnalysis* da classe *PolymorphismAnalysis*, que sobrescreve o método abstrato da super classe *MetricsAnalysisTool*. O método *returnDPDInListClass* é responsável por calcular o polimorfismo dinâmico nos descendentes. O método *produceTextAnalysis* invoca também todos os métodos necessários para cálculos das demais métricas de polimorfismo descritas neste trabalho, que não explicitados no diagrama de seqüência.

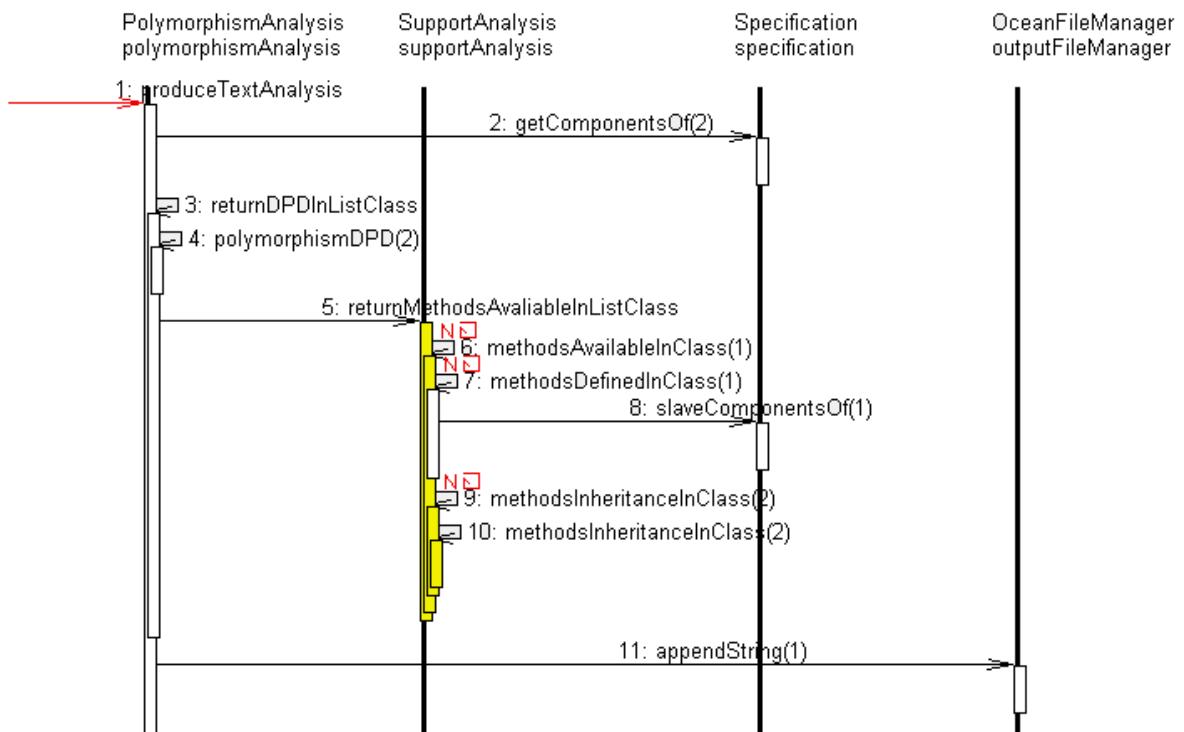


FIGURA 3 – Diagrama de Seqüência (PolymorphismAnalysis / produceTextAnalysis)

A figura 4 apresenta parte do diagrama de seqüência que refina o método *produceTextAnalysis* da classe *CouplingAnalysis*, que sobrescreve o método abstrato da super classe *MetricsAnalysisTool*. O método *returnOCAICInListClass* é responsável por calcular o acoplamento através da interação classe-atributo por importação nos ancestrais e o método *returnDCAECInListClass*, responsável por calcular o acoplamento através da interação classe-atributo por exportação nos descendentes. O método *produceTextAnalysis* invoca também todos os métodos necessários para cálculos das demais métricas de acoplamento descritas neste trabalho, estes não explicitados no diagrama de seqüência.

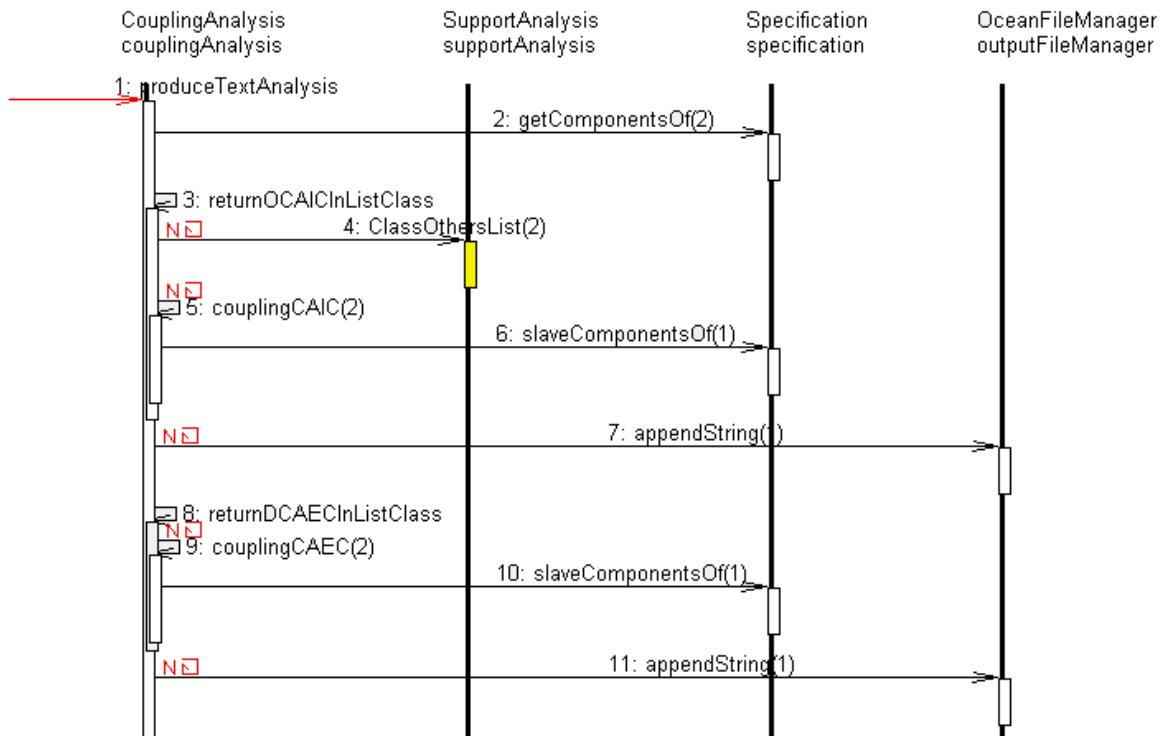


FIGURA 4 – Diagrama de Seqüência (CouplingAnalysis / produceTextAnalysis)

A figura 5 apresenta parte do diagrama de seqüência que refina o método *produceTextAnalysis* da classe *EncapsulationAnalysis*, que sobrescreve o método abstrato da super classe *MetricsAnalysisTool*. A execução do método *methodsHidingFactor* calcula o fator de métodos ocultos da especificação avaliada.

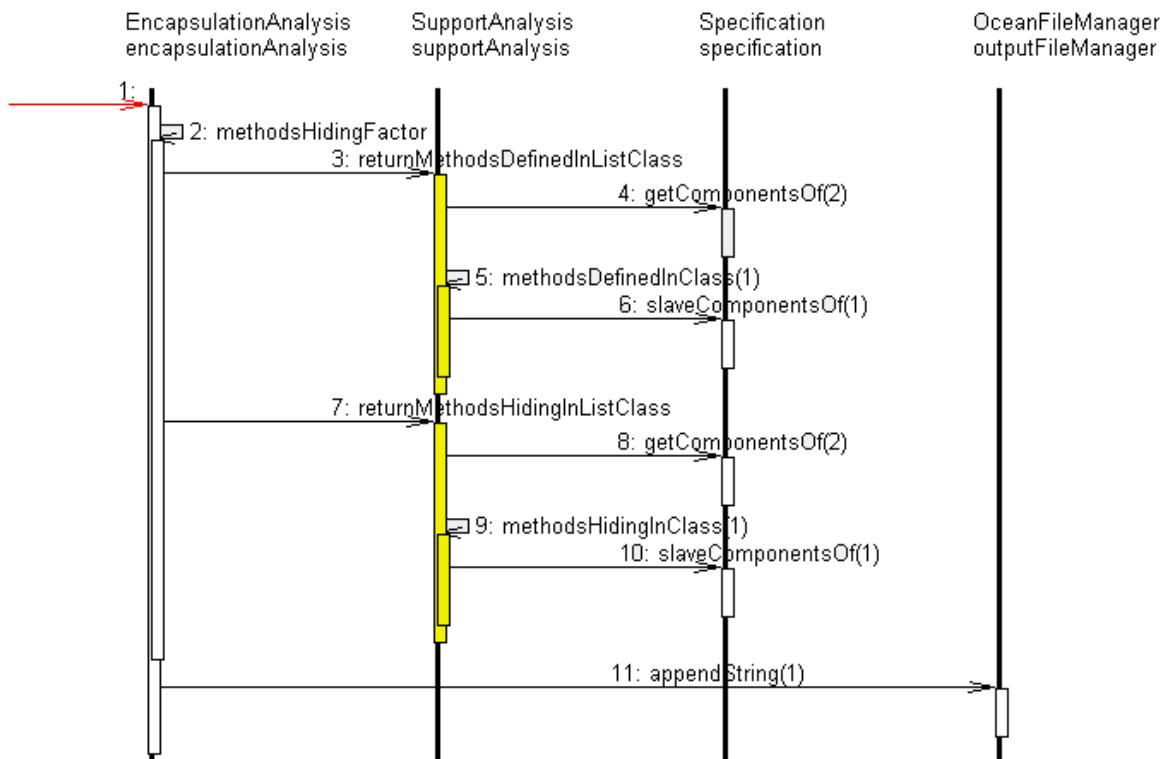


FIGURA 5 – Diagrama de Seqüência (EncapsulationAnalysis / produceTextAnalysis)

A figura 6 apresenta parte do diagrama de seqüência, que refina o método *produceTextAnalysis* da classe *CohesionAnalysis*, que sobrescreve o método abstrato da super classe *MetricsAnalysisTool*. A execução do método *returnLCOMInListClass* calcula a coesão entre os métodos das classes da especificação avaliada.

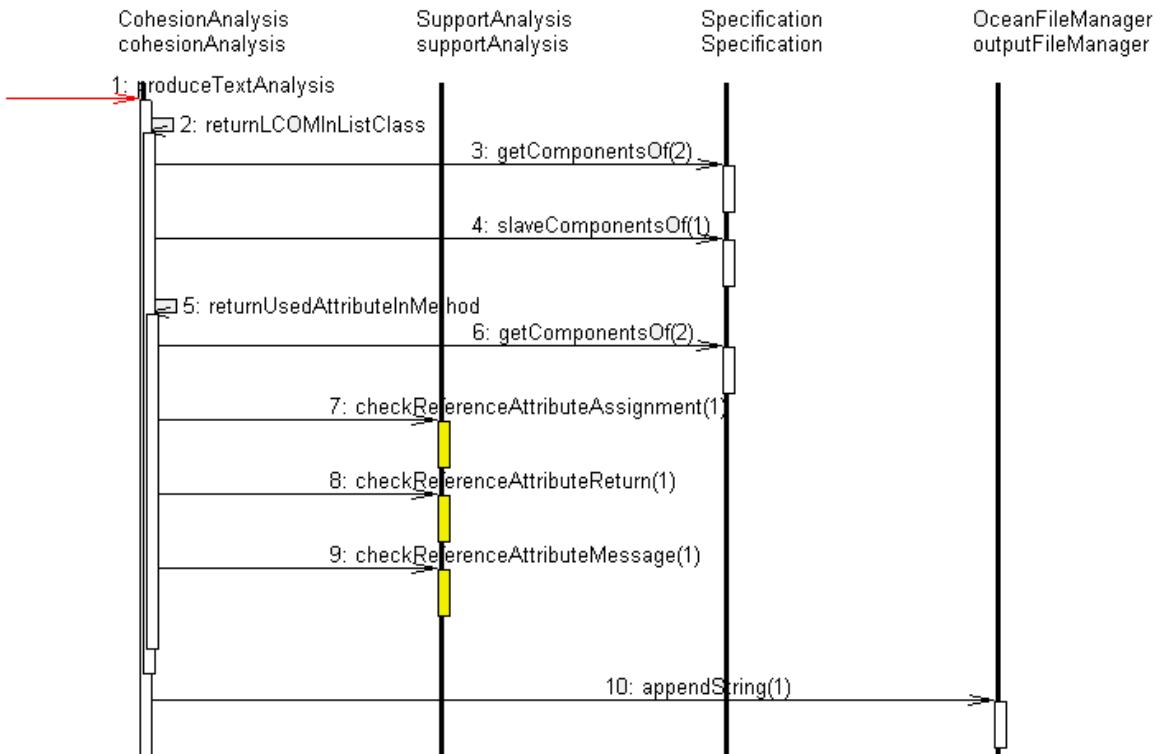


FIGURA 6 – Diagrama de Seqüência (CohesionAnalysis / produceTextAnalysis)

Diagramas de Corpo de Método

Serão apresentados somente alguns diagramas de corpo de método para ilustrar como a ferramenta MET é iniciada e como a lista de ferramentas é carregada.

Na figura 7 é apresentado parte do corpo de método referente ao método *new* da classe *MetricsExtractionTool*, responsável por criar uma instância da classe *AnalysisToolLoad*, que será usada para instanciar a lista de ferramentas.

MESSAGE	Method return assigned to: toolLoad : ToolLoad (atributo)
	Message sent to: AnalysisToolLoad (classe)
	Called method: new [^L]ToolLoad - Método de classe / base (ToolLoad)

FIGURA 7 – Diagrama de Corpo de Método (*MetricsExtractionTool* / *new*)

Na figura 8 é apresentado parte do corpo de método referente ao método *new* da classe *AnalysisToolLoad*, responsável por criar a lista de ferramentas.

MESSAGE	Method return assigned to: toolL... [L]: MetricsAnalyserTool (atributo)
	Message sent to: self
	Called method: createToolList [...o de instância / abstract (ToolLoad)]

FIGURA 8 – Diagrama de Corpo de Método (*AnalysisToolLoad / new*)

Dando continuidade, a figura 9 apresenta parte do corpo de método, referente ao método *getTextAnalysis* da classe *MetricsExtractionTool*, responsável por executar o método *produceTextAnalysis* de todas as ferramentas pertencentes à lista de ferramentas disponibilizada através da instância de *ToolLoad* nesta classe.

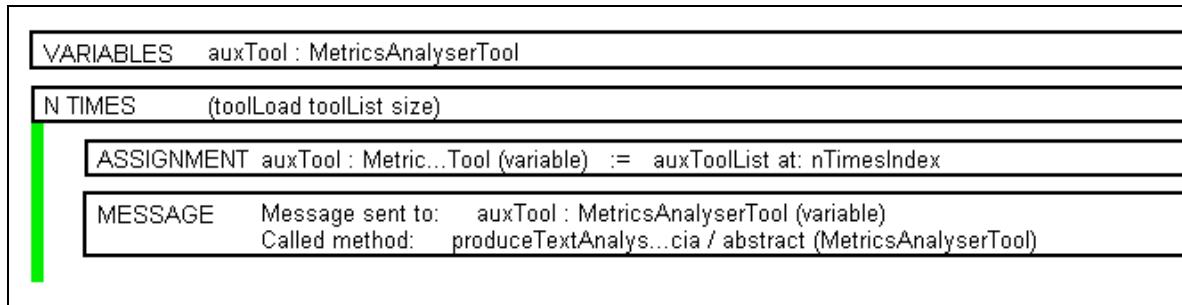


FIGURA 9 – Diagrama de Corpo de Método (MetricsExtractionTool / getTextAnalysis)

A seguir é apresentado parte do corpo de método referente ao método `createToolList` da classe `AnalysisToolLoad`. Este método é responsável por definir e instanciar as ferramentas (*PolymorphismAnalysis*, *PouplingAnalysis*, *CohesionAnalysis*, *EncapsulationAnalysis* e *complexityAnalysis*). Essa implementação possibilita a flexibilidade de inclusão ou remoção de ferramentas, pois a classe `MetricsExtractionTool` não precisa conhecer as ferramentas a serem executadas.

VARIABLES	auxTool : MetricsAnalyserTool auxToolList [L]: MetricsAnalyserTool
MESSAGE	Method return assigned to: auxTool : MetricsAnalyserTool (variable) Message sent to: CouplingAnalysis (classe) Called method: new - Método de classe / base (MetricsAnalyserTool)
TASK	auxToolList add: auxTool
MESSAGE	Method return assigned to: auxTool : MetricsAnalyserTool (variable) Message sent to: PolymorphismAnalysis (classe) Called method: new - Método de classe / base (MetricsAnalyserTool)
TASK	auxToolList add: auxTool
MESSAGE	Method return assigned to: auxTool : MetricsAnalyserTool (variable) Message sent to: CohesionAnalysis (classe) Called method: new - Método de classe / base (MetricsAnalyserTool)
TASK	auxToolList add: auxTool
MESSAGE	Method return assigned to: auxTool : MetricsAnalyserTool (variable) Message sent to: EncapsulationAnalysis (classe) Called method: new - Método de classe / base (MetricsAnalyserTool)
TASK	auxToolList add: auxTool
MESSAGE	Method return assigned to: auxTool : MetricsAnalyserTool (variable) Message sent to: ComplexityAnalysis (classe) Called method: new - Método de classe / base (MetricsAnalyserTool)
TASK	auxToolList add: auxTool
MESSAGE	Method return assigned to: auxTool : MetricsAnalyserTool (variable) Message sent to: SpecificationGeneralView (classe) Called method: new - Método de classe / base (MetricsAnalyserTool)
TASK	auxToolList add: auxTool
RETURN	auxToolList [L]: MetricsAnalyserTool (variable)

FIGURA 10 – Diagrama de Corpo de Método (*AnalysisToolLoad / createToolList*)

ANEXO 3 – FORMATO DO ARQUIVO DE EXPORTAÇÃO DE DADOS

Neste anexo é apresentado o formato do arquivo texto gerado pela ferramenta MET e que é utilizado com entrada de dados na ferramenta de visualização gráfica dos resultados das métricas.

O arquivo texto é formado por registros, sendo que a letra R denota início do registro e a letra F fim do registro. Os formatos dos registros são descritos a seguir:

[R1] // *Visão Geral da Especificação*
 Nome da Aplicação
 [R11]
 Nome da Classe 1
 Nome da Classe 2
 Nome da Classe N
 [F11]
 Número de classes na aplicação
 Número de Classes Abstratas
 Número de Classes Concretas
 Número de atributos Disponíveis na aplicação
 Número de atributos Definidos na aplicação
 Número de atributos Herdados na aplicação
 Número de métodos Disponíveis na aplicação
 Número de métodos Definidos na aplicação
 Número de métodos Herdados na aplicação
 Número de métodos Ocultos na aplicação
 Número de métodos Visíveis na aplicação
 Número de métodos Sobrescritos
 Número de métodos Novos
 Número de Métodos *Template*
 Número de Métodos Base
 Número de Métodos Abstrato
 Fator de Herança de Atributos – numerado
 Fator de Herança de Atributos – denominador
 Fator de Herança de Métodos – numerador
 Fator de Herança de Métodos – denominador
 Fator de Métodos Ocultos – numerador
 Fator de Métodos Ocultos – denominador
 Fator de Polimorfismo – numerador
 Fator de Polimorfismo – denominador
 [F1]

[R2] // *Visão Detalhada por Classe da Especificação*

Nome da Classe
Número de Atributos Disponíveis
[R21]
Nome atributo 1
Nome atributo 2
Nome atributo N
[F21]
Número de Atributos Definidos
[R22]
Nome atributo 1
Nome atributo 2
Nome atributo N
[F22]
Número de Atributos Herdados
[R23]
Nome atributo 1
Nome atributo 2
Nome atributo N
[F23]
Número de Métodos Disponíveis
[R24]
Nome método 1
Nome método 2
Nome método N
[F24]
Número de Métodos Definidos
[R25]
Nome método 1
Nome método 2
Nome método N
[F25]
Número de Métodos Herdados
[R26]
Nome método 1
Nome método 2
Nome método N
[R26]
Número de Métodos Ocultos
[R27]
Nome método 1
Nome método 2
Nome método N
[F27]
Número de Métodos Visíveis
[R28]
Nome método 1
Nome método 2
Nome método N

[F28]
 Número de Métodos Sobrescrito
 [R29]
 Nome método 1
 Nome método 2
 Nome método N
 [F29]
 Número de Métodos Novos
 [R210]
 Nome método 1
 Nome método 2
 Nome método N
 [F210]
 Número de Métodos *Template*
 [R211]
 Nome método 1
 Nome método 2
 Nome método N
 [F211]
 Número de Métodos Base
 [R212]
 Nome método 1
 Nome método 2
 Nome método N
 [F212]
 Número de Métodos Abstrato
 [R213]
 Nome método 1
 Nome método 2
 Nome método N
 [F213]
 [F2]

[R3] // **Métrica: Número de Métodos Disponíveis nas classes**
 Número de métodos (Refere-se ao valor do parâmetro indicado pelo usuário)
 Número de métodos (Refere-se ao valor proposto como ideal para a métrica)
 Número de classes (Coordenada Y)
 Valor da métrica - Número de Métodos nas classes (Coordenada X)
 [R31]
 Nome da classe 1
 Nome da classe 2
 Nome da classe N
 [F31]
 [F3]

[R4] // **Métrica: Número de Métodos Definidos nas Classes**
 Número de métodos (Refere-se ao valor do parâmetro indicado pelo usuário)
 Número de classes (Coordenada Y)

Valor da métrica - Número de Métodos nas classes (Coordenada X)

[R41]

Nome da Classe 1

Nome da Classe 2

Nome da Classe N

[F42]

[F4]

[R5] // ***Métrica: Número de Argumentos nos Métodos nas Classes***

Número de argumentos (Refere-se ao valor do parâmetro indicado pelo usuário)

Número de argumentos (Refere-se ao valor proposto como ideal para a métrica)

Número de métodos (Coordenada Y)

Valor da métrica - Número de argumentos (Coordenada X)

[R51]

Nome do Método 1

Nome do Método 2

Nome do Método N

[F51]

[F5]

[R6] // ***Métrica: Número de Classes Imediatamente Descendentes - NOC***

Número (Refere-se ao valor proposto como ideal para a métrica)

Número de classes (Coordenada Y)

Valor da métrica - Valor de NOC (Coordenada X)

[R61]

Nome da classe 1

Nome da classe 2

Nome da classe N

[F61]

[F6]

[R7] // ***Métricas: Tamanho dos métodos***

Número de *statements* (Refere-se ao valor do parâmetro indicado pelo usuário)

Número de *statements* (Refere-se ao valor proposto como ideal para a métrica)

Número de métodos (Coordenada Y)

Valor da métrica - Número de *statements* (Coordenada X) . . .

[R71]

Nome do método 1

Nome do método 2

Nome do método N

[F71]

[F7]

A lista de registro de métricas apresentadas deste ponto em diante, segue a interface descrita a seguir:

[RX] (sendo X o número do registro da métrica)

Número de ocorrência do elemento (Coordenada Y)

Valor da métrica – (Coordenada X)

[RX1]

Nome do elemento 1

Nome do elemento 2

Nome do elemento N

[FX1]

[FX]

Os registros que obedecem a estrutura descrita anteriormente são:

[R8] //Métrica: OVO

[R9] //Métrica: ACAIC

[R10] //Métrica: OCAIC

[R11] //Métrica: DCAEC

[R12] // Métrica: OCAEC

[R13] //Métrica: ACMIC

[R14] // Métrica: OCMIC

[R15] //Métrica: DCMEC

[R16] // Métrica: OCMEC

[R17] // Métrica: AMMIC

[R18] // Métrica: OMMIC

[R19] // Métrica: DMMEC

[R20] // Métrica: OMMEC

[R21] // Métrica: CBO

[R22] // Métrica: SPA

[R23] // Métrica: SPD

[R24] // Métrica: DPA

[R25] // Métrica: DPD

[R26] // Métrica: SP

[R27] // Métrica: DP

[R28] // Métrica: NIP

[R29] // Métrica: DIT

[R30] // Métrica: WMC

[R31] // Métrica: RFC

[R32] // Métrica: LCOM - Modificado

[R33] // Métrica: Referência a Subclasses