

FERNANDO DETTONI

***REENGENHARIA DE FERRAMENTAS PARA EXTRAÇÃO E
INTERPRETAÇÃO DE MÉTRICAS EM ESPECIFICAÇÕES DE
APLICAÇÕES E FRAMEWORKS ORIENTADAS A OBJETOS***

Florianópolis

2010

FERNANDO DETTONI

***REENGENHARIA DE FERRAMENTAS PARA EXTRAÇÃO E
INTERPRETAÇÃO DE MÉTRICAS EM ESPECIFICAÇÕES DE
APLICAÇÕES E FRAMEWORKS ORIENTADAS A OBJETOS***

Trabalho de Conclusão de Curso apresentada
como requisito parcial para obtenção de grau de
Bacharel em Ciências da Computação na Uni-
versidade Federal de Santa Catarina.

Orientador:

Ricardo Pereira e Silva

Co-orientador:

Leonardo de Souza Brasil

UNIVERSIDADE FEDERAL DE SANTA CATARINA - UFSC
CENTRO TECNOLÓGICO - CTC
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA - INE

Florianópolis

2010

O presente trabalho de conclusão de curso foi aprovado como requisito parcial para obtenção do grau de Bacharel em Ciências da computação na Universidade Federal de Santa Catarina sob o título “Reengenharia de ferramentas para extração e interpretação de métricas em especificações de aplicações e frameworks orientadas a objetos”, elaborado por Fernando Dettoni e aprovado em 2010, em Florianópolis, no estado de Santa Catarina pela banca examinadora constituída pelos membros:

Prof. Dr. Ricardo Pereira e Silva
Orientador

Leonardo de Souza Brasil
Co-Orientador

Prof. Msc. Evandro César Freiburger
Universidade Federal de Mato Grosso

Prof. Dr. Raul Sidnei Wazlawick
Universidade Federal de Santa Catarina

Sumário

Lista de Figuras	
1 INTRODUÇÃO	p. 8
1.1 OBJETIVOS	p. 9
1.1.1 OBJETIVO GERAL	p. 9
1.1.2 OBJETIVOS ESPECÍFICOS	p. 9
1.2 MOTIVAÇÃO	p. 9
2 FRAMEWORKS ORIENTADOS A OBJETOS	p. 11
2.1 DESENVOLVIMENTO DE FRAMEWORKS ORIENTADOS A OBJETOS	p. 13
2.2 CICLO DE VIDA DE UM FRAMEWORK	p. 14
2.3 ASPECTOS DE QUALIDADE EM <i>FRAMEWORKS</i> ORIENTADOS A OBJETOS	p. 15
3 AMBIENTE DE DESENVOLVIMENTO OCEAN/SEA	p. 17
3.1 FRAMEWORK OCEAN	p. 17
3.2 AMBIENTE SEA	p. 18
3.2.1 REQUISITOS PARA UM AMBIENTE DE APOIO AO DESENVOLVIMENTO DE APLICAÇÕES E <i>FRAMEWORKS</i> ORIENTADOS A OBJETOS	p. 19
3.2.2 ESPECIFICAÇÃO ORIENTADA A OBJETOS NO AMBIENTE SEA	p. 20
4 MÉTRICAS APLICADAS A ESPECIFICAÇÕES ORIENTADAS A OBJETOS	p. 25
4.1 CONJUNTO DE MÉTRICAS APLICADAS AO DESENVOLVIMENTO E USO DE FRAMEWORKS	p. 25
4.1.1 QUANTO À PROPORÇÃO DE SOFTWARE PRODUZIDO E REUSADO	p. 26
4.1.2 QUANTO À ESPECIALIZAÇÃO NO USO DE FRAMEWORKS	p. 33

4.2	CONJUNTO DE MÉTRICAS APLICADAS AO DESENVOLVIMENTO DE APLICAÇÕES ORIENTADAS A OBJETOS	p. 41
4.2.1	MÉTRICAS DE ACOPLAMENTO	p. 41
4.2.2	MÉTRICAS DE ENCAPSULAMENTO	p. 46
4.2.3	MÉTRICAS DE COMPLEXIDADE	p. 47
4.2.4	MÉTRICAS DE COESÃO	p. 51
4.2.5	MÉTRICAS DE POLIMORFISMO	p. 51
5	REENGENHARIA DE FERRAMENTAS DE EXTRAÇÃO DE MÉTRICAS ORIENTADAS A OBJETO	p. 55
5.1	ANÁLISE DAS ESPECIFICAÇÕES	p. 55
5.2	TRABALHOS CORRELATOS	p. 57
5.3	REENGENHARIA DAS FERRAMENTAS	p. 58
5.3.1	OBTENÇÃO DOS DADOS DA ESPECIFICAÇÃO	p. 59
5.3.2	CÁLCULO DAS MÉTRICAS E APRESENTAÇÃO DOS RESULTADOS	p. 64
6	CONCLUSÃO	p. 68
6.1	REALIZAÇÕES	p. 68
6.2	LIMITAÇÕES	p. 69
6.3	TRABALHOS FUTUROS	p. 70
6.4	CONSIDERAÇÕES FINAIS	p. 71
	REFERÊNCIAS	p. 72
	Apêndice A – EXEMPLO DE USO DA FERRAMENTA	p. 74
A.1	ESPECIFICAÇÃO DO FRAMEWORK FRAG	p. 74
A.2	MÉTRICAS REFERENTES AO USO DE FRAMEWORKS ORIENTADOS A OBJETOS	p. 79
A.3	MÉTRICAS SOBRE APLICAÇÕES ORIENTADAS A OBJETOS	p. 83

A.3.1	ACOPLAMENTO	p. 83
A.3.2	ENCAPSULAMENTO	p. 88
A.3.3	COMPLEXIDADE	p. 88
A.3.4	COESÃO	p. 95
A.3.5	POLIMORFISMO.....	p. 95

Lista de Figuras

Figura 2.1 Aplicação desenvolvida com um framework	11
Figura 2.2 Aplicação desenvolvida utilizando bibliotecas	12
Figura 2.3 Ciclo de vida de um <i>framework</i>	14
Figura 3.1 Estrutura do ambiente SEA	18
Figura 3.2 Alguns diagramas presentes na especificação orientada a objetos.	20
Figura 3.3 Diagrama de atividade representando algoritmo de métodos.	23
Figura 5.1 Diagrama de casos de uso da ferramenta.	58
Figura 5.2 Diagrama de sequência da ferramenta.	59
Figura 5.3 Estrutura para armazenamento dos dados da especificação original	60
Figura 5.4 Representação de aplicações e frameworks.	61
Figura 5.5 Representação de classes e métodos na estrutura de armazenamento.	62
Figura 5.6 Representação de métodos e seus agregados na estrutura de armazenamento. ..	63
Figura 5.7 Classes da ferramenta de extração e análise de métricas orientadas a objetos. ..	65

Figura 5.8 Sequenciamento dos métodos a partir da invocação da ferramenta.	66
Figura 5.9 Visualização dos dados na ferramenta.	67
Figura 5.10 Visualização dos dados de forma gráfica.	67
Figura A.1 Classes relacionadas com Position.	75
Figura A.2 Classes relacionadas com Dice.	76
Figura A.3 Classes relacionadas com Player.	77
Figura A.4 Classes pertencentes ao Jogo da Velha.	78
Figura A.5 Classes pertencentes ao Reversi.	79

1 INTRODUÇÃO

A Engenharia de Software é a área dentro da Ciência da computação que fornece métodos, ferramentas e procedimentos, de forma integrada, para o desenvolvimento de softwares com maior qualidade e menos custo (FONSECA, 2002). Engloba não apenas aspectos técnicos do desenvolvimento de software, mas também aspectos gerenciais como a coordenação da equipe de projeto.

Dentro do processo de desenvolvimento do software, o paradigma de orientação a objetos provê mecanismos suficientes para a produção de softwares complexos com alta qualidade, mas não atesta por si só esta qualidade. Sendo assim, são necessárias ferramentas de apoio à verificação da qualidade no desenvolvimento de software nas primeiras etapas do processo de desenvolvimento (SILVA, 2000) (FREIBERGER, 2002).

De uma forma genérica, a qualidade do software é definida por Pressman (2006) como a sua conformidade com os requisitos do cliente, com normas de desenvolvimento explicitamente documentadas, além de características esperadas em qualquer software desenvolvido profissionalmente. Porém, de uma maneira mais específica, é muito difícil, senão impossível, definir formalmente estes parâmetros de qualidade. O estudo de métricas de software iniciou-se para auxiliar na verificação da qualidade de software quantificando alguns parâmetros e associando significados qualificativos aos dados quantitativos obtidos (FREIBERGER, 2002) e (FONSECA, 2002).

O ambiente SEA, desenvolvido por Silva (2000) sob o *framework* OCEAN tem por objetivo o suporte ao desenvolvimento de sistemas orientados a objetos com foco principal no desenvolvimento e uso de artefatos reutilizáveis. Este ambiente provê suporte à adição de ferramentas que possibilitam a edição, transformação e análise de conceitos e modelos orientados a objetos.

Em (FREIBERGER, 2002) e (FONSECA, 2002) os autores apresentam diferentes conjuntos de métricas, bem como a implementação de ferramentas acopladas ao SEA para a extração destas métricas. Porém, o ambiente SEA passou por uma reengenharia de modo a torná-lo mais

presente ao contexto de desenvolvimento de software nos dias de hoje e não foi incluído em seu escopo a reengenharia destas ferramentas (COELHO, 2007) (MACHADO, 2007) (AMORIM, 2006) (VARGAS, 2008).

Vargas (2008) incrementou o ambiente SEA com a possibilidade de definição de especificações orientadas a objetos utilizando a linguagem UML 2.0, que define novos conceitos e documentos, permitindo a definição de especificações mais completas e detalhadas. A utilização de ferramentas de extração de métricas orientadas a objetos sob especificações na linguagem UML 2.0 permite agregar mais qualidades aos artefatos desenvolvidos sob esta linguagem (OMG, 2004).

1.1 OBJETIVOS

1.1.1 OBJETIVO GERAL

O presente trabalho pretende efetuar a reengenharia das ferramentas propostas por Freiburger (2002) e Fonseca (2002) para a extração e visualização de métricas de artefatos de software de modo a integrá-las às novas versões do *framework* OCEAN atualizando-as e provendo novas funcionalidades.

1.1.2 OBJETIVOS ESPECÍFICOS

A partir do objetivo geral deste trabalho foram definidos os seguintes objetivos específicos:

- Analisar os conceitos gerais de métricas de software a fim de conhecer melhor o cenário atual deste campo;
- Analisar as ferramentas de obtenção de métricas de softwares orientados a objetos;
- Efetuar a reengenharia de ferramentas propostas por Fonseca (2002) e Freiburger (2002), a fim de que se tornem presentes nas novas versões do ambiente SEA;
- Atualizar estas ferramentas para integrá-las ao ambiente OCEAN/SEA em seu estado atual que engloba a utilização de UML 2.0.

1.2 MOTIVAÇÃO

Dentre todos os produtos criados pelo ser humano nas últimas décadas, a computação enfrenta, possivelmente, um dos maiores níveis de obsolescência. Cria-se a necessidade de

atualizações constantes de software e hardware. Tendo em vista este caráter evolutivo, é muito importante que os softwares se mantenham em constante evolução para acompanhar o mercado. Neste sentido que surge a reengenharia de software, que efetua fortes mudanças, de uma forma organizada, em um software com características já obsoletas para que venha a atingir melhor os objetivos para o qual foi projetado (PRESSMAN, 2006).

É um consenso entre desenvolvedores de software que a qualidade é um fator extremamente importante no desenvolvimento de softwares (PRESSMAN, 2006). Sendo assim, são necessários meios para a obtenção do nível de qualidade desejado em um software. Ferramentas que analisem especificações orientadas a objetos, obtendo métricas sobre determinados aspectos do projeto, podem ser muito úteis nessa asserção da qualidade.

Após a reengenharia promovida no ambiente SEA, este carece de ferramentas que possam ajudar na busca pela qualidade do software a partir de métricas orientadas a objetos. Como (FREIBERGER, 2002) e (FONSECA, 2002) já propuseram ferramentas com este intuito, a reengenharia das mesmas, além de trazê-las para o contexto atual do ambiente SEA, traz a oportunidade de ir além do que foi proposto anteriormente e complementar o seu desenvolvimento com a utilização de técnicas e conceitos não presentes anteriormente.

2 FRAMEWORKS ORIENTADOS A OBJETOS

Um *framework* captura em sua essência, o conhecimento necessário para a resolver uma classe de problemas em particular (TALIGENT, 1994). Um framework orientado a objetos se utiliza da abordagem orientada a objetos para generalizar um domínio de aplicação a partir da utilização de classes inter-relacionadas (SILVA, 2000).

A abordagem de *frameworks* orientados a objetos, apresentada na figura 2.1, se diferencia da utilização de bibliotecas de classe, apresentada na figura 2.2, justamente pelo inter-relacionamento entre as suas classes (SILVA, 2000). Enquanto na utilização de uma biblioteca cabe ao usuário definir o comportamento entre as classes da aplicação, na abordagem de *frameworks* orientados a objetos este comportamento já está definido e é ele que se utiliza das classes da aplicação.

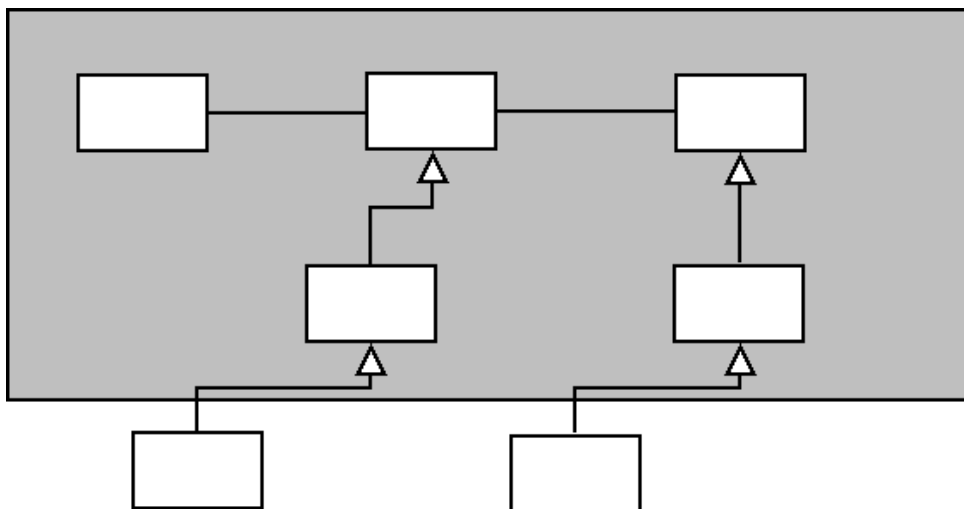


Figura 2.1: Aplicação desenvolvida com um framework

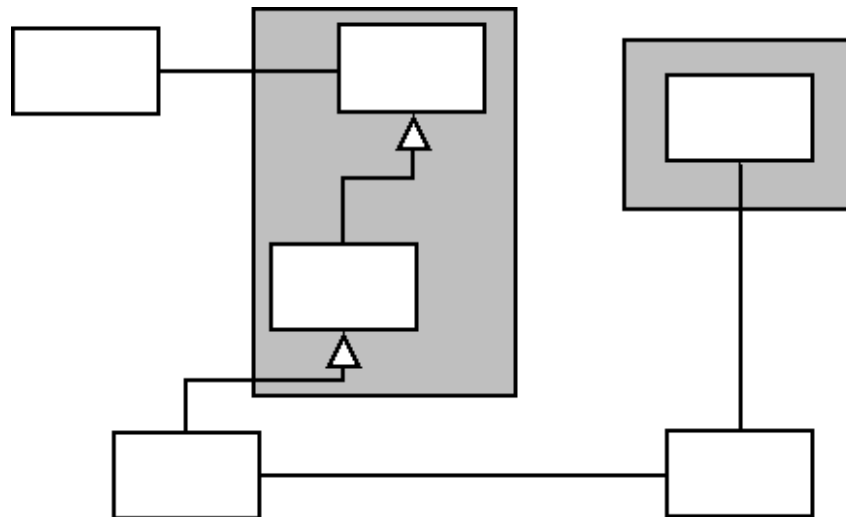


Figura 2.2: Aplicação desenvolvida utilizando bibliotecas

De acordo com Fayad e Schmidt (1997) os principais benefícios apresentados pela utilização da abordagem de *frameworks* orientados a objetos são:

- Modularidade, devido ao encapsulamento de informações de implementações atrás de interfaces estáveis;
- Reusabilidade, definindo componentes genéricos que eliminarão a necessidade de re-desenvolvimento de soluções comuns;
- Extensibilidade a partir de métodos *hook*¹ definidos pela aplicação para a especialização e customização de serviços e características;
- Inversão de controle permite que o *framework* selecione os conjuntos de métodos *hook* caracterizando a utilização de *frameworks* orientados a objeto.

Sendo assim, o desenvolvimento e uso de *frameworks* orientados a objetos pode ser bastante útil para o desenvolvimento de aplicações neste paradigma. Além de prover o reuso do código desenvolvido, o *framework* permite também o reuso de comportamentos que são apresentados por diferentes aplicações de um mesmo domínio.

¹Métodos *hook* possuem uma classificação relativa: são métodos chamados por métodos *template*. Métodos *template* por sua vez são caracterizados por serem definidos apenas em função de outro métodos (métodos *hook*).

2.1 DESENVOLVIMENTO DE FRAMEWORKS ORIENTADOS A OBJETOS

Criar um *framework* de forma que este possua as propriedades de alterabilidade e extensibilidade exige muitos cuidados por parte do projetista para que seja feita a correta identificação das partes flexíveis do *frameworks*. Além disso é necessário uma certa familiaridade com os princípios da orientação a objetos para de modo a utilizar destas características para garantir a reusabilidade do projeto, bem como promover o polimorfismo para possibilitar o acoplamento dinâmico (SILVA, 2000).

Pode ser notado que um *framework* desenvolvido para atender a um domínio específico de aplicações será mais complexo que qualquer uma das aplicações desenvolvidas sobre ele. Isso ocorre devido aos objetivos do *framework*, que deve ser capaz de generalizar as necessidades do domínio, precisar prover a extensibilidade, modularidade e alterabilidade necessária aos artefatos produzidos a partir deste.

Ao iniciar o desenvolvimento de um *framework*, deve-se decidir sobre a abrangência do domínio tratado levando em consideração o custo para a produção do artefato. Frameworks com maior abrangência tendem a custar mais, enquanto *frameworks* com menor abrangência tendem a ser mais sensíveis quanto a mudança de domínio (FREIBERGER, 2002).

2.2 CICLO DE VIDA DE UM FRAMEWORK

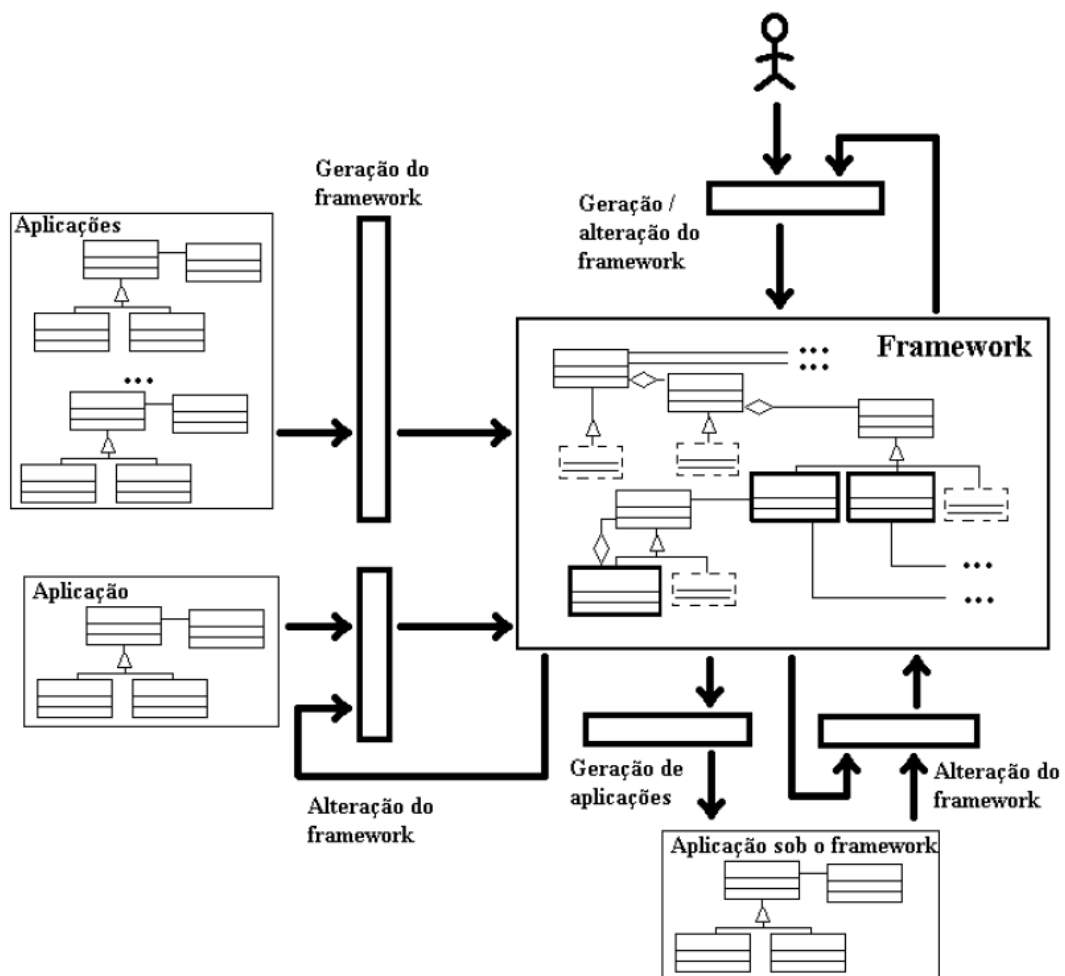


Figura 2.3: Ciclo de vida de um *framework*. Fonte: (SILVA, 2000)

O ciclo de vida de um *framework*, tal como mostrado na figura 2.3, desenvolve-se de forma bastante diferente a uma aplicação comum já que não se trata apenas de um artefato isolado, mas de um artefato em estado intermediário entre as aplicações que deram origem ao *framework* e as aplicações que serão desenvolvidas a partir dele (SILVA, 2000) (FREIBERGER, 2002).

Neste modelo de ciclo de vida, temos o desenvolvedor em um papel fundamental dentro do desenvolvimento de *frameworks* orientados a objetos. Cabe a ele o levantamento da estrutura necessária para que o *framework* apresente a flexibilidade necessária para a criação de aplicações sobre o mesmo, além de gerar a documentação e prover sua manutenção.

As aplicações desenvolvidas em um domínio constituem uma base importante para o desenvolvimento de um *framework* que pretenda generalizar este domínio. Estas aplicações são a base fundamental para o desenvolvimento de acordo com a metodologia de Projeto Dirigido por

Exemplos. Como expressa Silva (2000), em um cenário ideal, seria interessante o uso de uma ferramenta que automaticamente generalizasse um domínio de aplicação, gerando um *framework* a partir de exemplos, mas as vastas possibilidades no desenvolvimento de aplicações torna esta tarefa bastante complexa, se não impossível.

As aplicações geradas pelo *framework* são o principal objetivo de seu projeto. Idealmente, estas aplicações devem implementar apenas o que já foi previsto como flexível no projeto do *framework* mas na prática isto raramente acontece por falta de um total entendimento do domínio em um primeiro momento. Por isso, é relevante ao importante ao *framework* que, durante seu ciclo de vida, incorpore determinadas funcionalidades que não foram implementadas anteriormente. Uma solução para ajudar na descoberta de problemas na utilização de *frameworks* orientados a objetos é a utilização de métricas como as apresentadas por Freiberger (2002).

2.3 ASPECTOS DE QUALIDADE EM *FRAMEWORKS* ORIENTADOS A OBJETOS

Por ter como objetivo final a geração de aplicações, a qualidade de um *framework* se dá, não apenas pela qualidade do próprio *framework*, mas também pela qualidade do uso deste pelas aplicações geradas sobre ele. Além disso, quando encontrado algum indício de problema na qualidade este pode ser, não apenas um problema de projeto do *framework*, mas também um problema gerado pelo mau uso do *framework* pela aplicação (FREIBERGER; SILVA, 2009).

Idealmente, um *framework* deve poder generalizar o domínio de tal forma que não seja necessária a criação de novos conceitos pelas aplicações geradas a partir dele. A criação de conceitos não previstos no *framework* pode ocorrer pela criação de classes não relacionadas por herança, a criação de métodos que não sobrescrevam métodos herdados ou a criação de novos atributos.

Como exemplo, a criação de uma classe não relacionada por herança com uma classe prevista para gerar subclasses do *framework* representa a criação um conceito não reutilizado do *frameworks*. Porém, esta medida por si só não garante um problema no projeto do *framework*, podendo ser também um mau uso do mesmo. Um meio de investigar mais a fundo a causa do problema pode ser a utilização de métricas sobre o histórico de aplicações desenvolvidas sob o *framework*.

Freiberger (2002) defende a utilização de medidas sobre histórico de aplicações desenvolvidas sobre o *frameworks*. Desta forma é possível fazer uma análise comparativa entre as

aplicações desenvolvidas, além da verificação de qualidade ter seu foco no uso do *framework*, e não apenas no seu desenvolvimento. A utilização de medidas para *frameworks* orientados a objetos necessita certos requisitos:

1. Disponibilidade de um ambiente para especificação de *frameworks* e aplicações orientadas a objetos;
2. Disponibilidade de uma base de dados relacionando especificações de um *framework* orientado a objetos e as aplicações desenvolvidas sob este *framework*;
3. Ter um conjunto de medidas aplicadas à asserção da qualidade em *frameworks* e aplicações orientadas a objetos;
4. Disponibilidade de uma ferramenta capaz de extrair as informações necessárias para a geração das medidas definidas.

Os requisitos um e dois são preenchidos pelo ambiente SEA, que tem seu foco na especificação de artefatos reutilizáveis de software. Um conjunto de métricas aplicadas ao histórico de aplicações desenvolvidas sob um *framework*, previsto no requisito três, pode ser encontrado em (FREIBERGER, 2002) e (FREIBERGER; SILVA, 2009). Este trabalho se propõe a efetuar a reengenharia da ferramenta de extração de métricas criada por Freiburger (2002), de modo a preencher também o quarto requisito.

3 AMBIENTE DE DESENVOLVIMENTO OCEAN/SEA

Criado por Ricardo Pereira e Silva (SILVA, 2000), o *framework* OCEAN tem a finalidade de generalizar o domínio de ferramentas de suporte ao desenvolvimento de aplicações e *frameworks* orientados a objetos. Seu principal foco está na flexibilidade, já que é voltado à modelagem de *frameworks* e componentes e necessita se adaptar às mudanças durante seu ciclo de vida.

A partir do *framework* OCEAN, foi desenvolvido ainda no mesmo projeto a aplicação SEA que tem por objetivo ser um ambiente para projeto de aplicações e *frameworks* orientados a objetos com uso de UML. O ambiente permite, além da especificação de aplicações orientadas a objetos, a especificação de *frameworks* orientados a objetos, de componentes e interfaces de componentes orientados a objetos, e a especificação de padrões de projeto.

Originalmente implementado utilizando a linguagem *Smalltalk* e o ambiente de desenvolvimento VisualWorks, tanto o *framework* OCEAN quanto o ambiente SEA passaram por uma reengenharia aonde foram reescritos na linguagem *Java* conforme descrito em Coelho (2007), Machado (2007), Amorim (2006) e Vargas (2008). O desenvolvimento do presente trabalho ocorre na versão mais recente do sistema.

3.1 FRAMEWORK OCEAN

Além de suportar o desenvolvimento de *frameworks* e componentes orientados a objetos, o objetivo do OCEAN é generalizar o domínio de aplicações de suporte ao desenvolvimento de artefatos orientados a objetos, de modo que cada ambiente criado a partir dele possa utilizar sua própria estrutura de especificações e possa manter suas próprias funcionalidades.

A extensão básica do OCEAN se dá a partir da especialização, por herança, de uma classe abstrata, a classe *EnvironmentManager*, além da especialização de especificações, modelos e conceitos necessários. Nesta classe devem ser definidos certos aspectos da aplicação:

- os tipos de especificação tratados, bem como seus modelos e conceitos;

- o mecanismo de visualização associado a cada modelo e conceito;
- o mecanismo de armazenamento a ser utilizado;
- as ferramentas utilizáveis para a manipulação de especificações

A estrutura de uma especificação no OCEAN consiste basicamente em um repositório de modelos, um repositório de conceitos, e informação sobre o relacionamento entre estes, contida nas tabelas de sustentação e referência. Para se definir um tipo de especificação devem-se definir quais os tipos de modelos tratados por esta especificação, sendo que em cada tipo de modelo deve definir os conceitos tratados a partir deste.

3.2 AMBIENTE SEA

O objetivo do SEA é, basicamente, auxiliar na produção de aplicações, *frameworks* e componentes orientados a objetos de forma flexível e extensível. Sua estrutura baseia-se na arquitetura *toaster* conforme apresentado na figura 3.1.

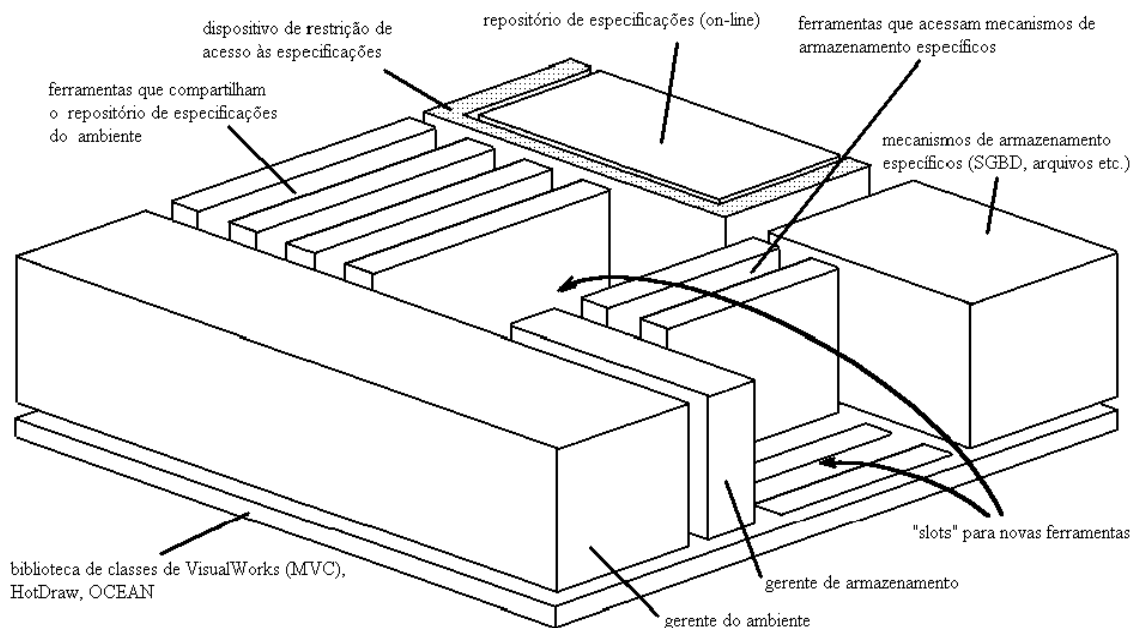


Figura 3.1: Estrutura do ambiente SEA. Fonte: (SILVA, 2000)

Possui um repositório compartilhado de especificações, que pode ser acessado pelas diversas ferramentas do sistema. A manipulação das especificações do repositório compartilhado ocorre a partir das ferramentas utilizando a interface provida pelo gerente do ambiente. A gravação e o carregamento das especificações são administrados pelo gerente de armazenamento.

3.2.1 REQUISITOS PARA UM AMBIENTE DE APOIO AO DESENVOLVIMENTO DE APLICAÇÕES E *FRAMEWORKS* ORIENTADOS A OBJETOS

Silva (2000) define um conjunto de requisitos que devem ser obedecidos na implementação de uma aplicação que pretenda servir de apoio ao desenvolvimento de aplicações, *frameworks* e componentes orientados a objetos, tal como o ambiente SEA (SILVA, 2000). Devido a sua natureza flexível, artefatos de software desenvolvidos a partir deste ambiente terão, normalmente, seu foco voltado ao reuso e por isso, este também deve ser o foco do ambiente de suporte. No que diz respeito ao apoio do desenvolvimento de *frameworks* e componentes, o ambiente deve obedecer aos seguintes requisitos:

- Registro das informações referentes à flexibilidade do artefato, como a definição de classes redefiníveis e essenciais.
- Suporte à produção de artefatos a partir da especialização de artefatos reutilizáveis como *frameworks* e componentes.
- Suporte à adição de novas funcionalidades em um *framework* a partir das funcionalidades implementadas em alguma aplicação desenvolvida sobre este.
- Suporte à geração e edição de *frameworks* a partir de aplicações do domínio do problema, mas que não sejam especializações deste framework.
- Tratamento em separado do desenvolvimento de interfaces de componentes e dos componentes em si para incrementar o nível de reuso possibilitando que mais de um *framework* seja desenvolvido sob uma mesma interface.
- Suporte ao desenvolvimento de aplicações a partir da interconexão entre diferentes componentes.
- Suporte à utilização conjunta das abordagens de desenvolvimento de *frameworks* e componentes.
- Suporte ao uso de padrões na especificação de artefatos, promovendo o reuso da experiência de projeto.
- Suporte ao uso e padrões arquitetônicos de projeto, promovendo também o reuso da experiência de projeto.

3.2.2 ESPECIFICAÇÃO ORIENTADA A OBJETOS NO AMBIENTE SEA

Para a definição de especificações orientadas a objeto, que além de aplicações inclui também *framework* e componentes, o ambiente SEA utiliza basicamente a linguagem UML 1.1 com algumas modificações propostas por Silva (2000) na versão em SmallTalk, e a linguagem UML 2.0 com extensões baseadas em estereótipos propostas por Silva (2007) na versão em Java. Neste trabalho são abordados, principalmente, os aspectos das especificações definidas em linguagem UML 2.0, por estar mais ligada ao contexto atual do ambiente OCEAN/SEA.

Estas modificações se devem principalmente ao fato de as definições da UML ainda não cobrirem de um modo satisfatório a especificações de artefatos flexíveis como *frameworks* e componentes, além da ausência de um diagrama próprio para a especificação de algoritmos de métodos. A figura 3.2 apresenta alguns diagramas de uma especificação orientada a objetos no ambiente SEA.

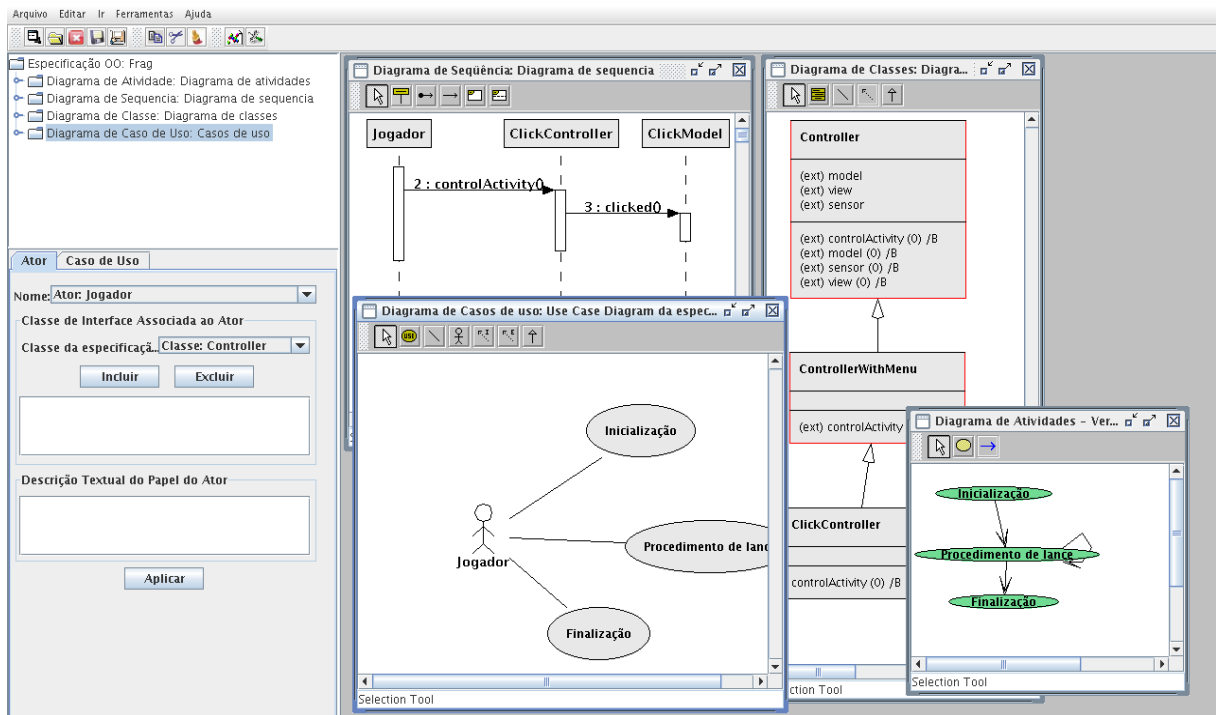


Figura 3.2: Alguns diagramas presentes na especificação orientada a objetos.

Os diagramas que compõe a linguagem de especificação UML 2.0 do ambiente SEA são explicitados em Vargas (2008) e Silva (2007):

- Diagrama de Casos de Uso, como previsto na UML, modela as funcionalidades do sistema;
- Diagrama de Atividades, podendo representar o sequenciamento de casos de uso, ou a descrição de algoritmos de métodos;

- Diagrama de Classes, permite a modelagem de cada classe do sistema, além de seus relacionamentos;
- Diagrama de Sequência, possibilita o refinamento dos casos de uso a partir de interações que podem ser novamente refinadas em outros diagramas de sequência;
- Diagrama de Máquina de Estados, representa o estado dos atributos relevantes de uma classe em determinado momento, além da situação que acarreta sua mudança (chamada de método);
- Diagrama de Objetos, representa um conjunto de instâncias de classes, e suas ligações, auxiliando na modelagem estrutural do sistema e da classe;
- Diagrama de Pacotes, sendo um pacote uma entidade voltada a agrupar elementos sintáticos da especificação, o diagrama de pacotes modela as relações entre os pacotes da especificação;
- Diagrama de Componentes, que representa a relação entre diferentes componentes do sistema;
- Diagrama de Implantação, também definido como Diagrama de Utilização por Silva (2007), define a organização do conjunto de elementos de um sistema;
- Diagrama de Comunicação, apresenta um outro modo para modelar a interação entre objetos do sistema, complementando ou substituindo o Diagrama de Sequência.
- Diagrama de Estrutura Composta, permite detalhar elementos de modelagem estrutural, como classes e componentes, definindo suas interfaces e estruturas internas.

Devido ao foco no desenvolvimento de *frameworks*, o conceito de classe no SEA foi entendido de modo a englobar mais dois atributos: a redefinibilidade e a essencialidade. Estes atributos definem, respectivamente, se a classe pode ser redefinida pela aplicação (a partir de herança), e se a classe deve obrigatoriamente ser redefinida pela aplicação.

Em relação aos métodos modelados, há a possibilidade de definí-los como base, *template* ou abstrato. Indicando, respectivamente, que um método é totalmente definido, é definido em função de outros métodos ou tem apenas sua assinatura definida.

Dentre o conjunto de diagramas definidos, dois se destacam dos demais por possibilitarem a definição da maior parte dos aspectos do sistema. O Diagrama de Classes possibilita, sozinho, a definição da estrutura interna de todas as classes do sistema, e a interconexão entre estas. Complementando este, o Diagrama de Atividades, conforme definido por Silva (2007) permite

a definição individual dos algoritmos do sistema, possibilitando a compreensão do comportamento dinâmico do sistema.

DIAGRAMA DE CLASSES

O mais utilizado dentre o conjunto de diagramas da linguagem UML 2.0, o Diagrama de Classes permite ao desenvolvedor definir, de forma rudimentar, a estrutura do sistema em tempo de desenvolvimento, com suas relações entre as classes.

Permite representar conceitos como a herança, agregação, composição e associação, de uma forma mais explícita que o código desenvolvido. Se por um lado representa muito bem a parte estática da aplicação, por outro não possui meios de representar o sistema em tempo de execução, como a interação entre métodos e troca de mensagens (SILVA, 2007).

Podemos a partir deste obter diversas informações utilizadas no cálculo de métricas orientadas a objetos:

- Nome das classes da especificação;
- Atributos das classes;
- Métodos definidos na classe;
- Relações de herança, agregação, composição e associação.

DIAGRAMA DE ATIVIDADES

Como complemento ao Diagrama de Classes, são necessários modelos capazes de modelar o sistema em sua execução. Silva (2007) propõe a utilização do diagrama de atividades para a definição de algoritmos de métodos, definindo um conjunto de ações e atividades personalizadas para a representação de conceitos como: definição e atribuição de variáveis, chamadas de métodos, fluxos de controle, etc. Desta forma, o Diagrama de Atividades provê ao desenvolvedor um alto nível de detalhamento do sistema, permitindo também a utilização de ferramentas para a geração automatizada de código. A figura 3.3 apresenta a utilização do diagrama de atividades para a definição de algoritmos.

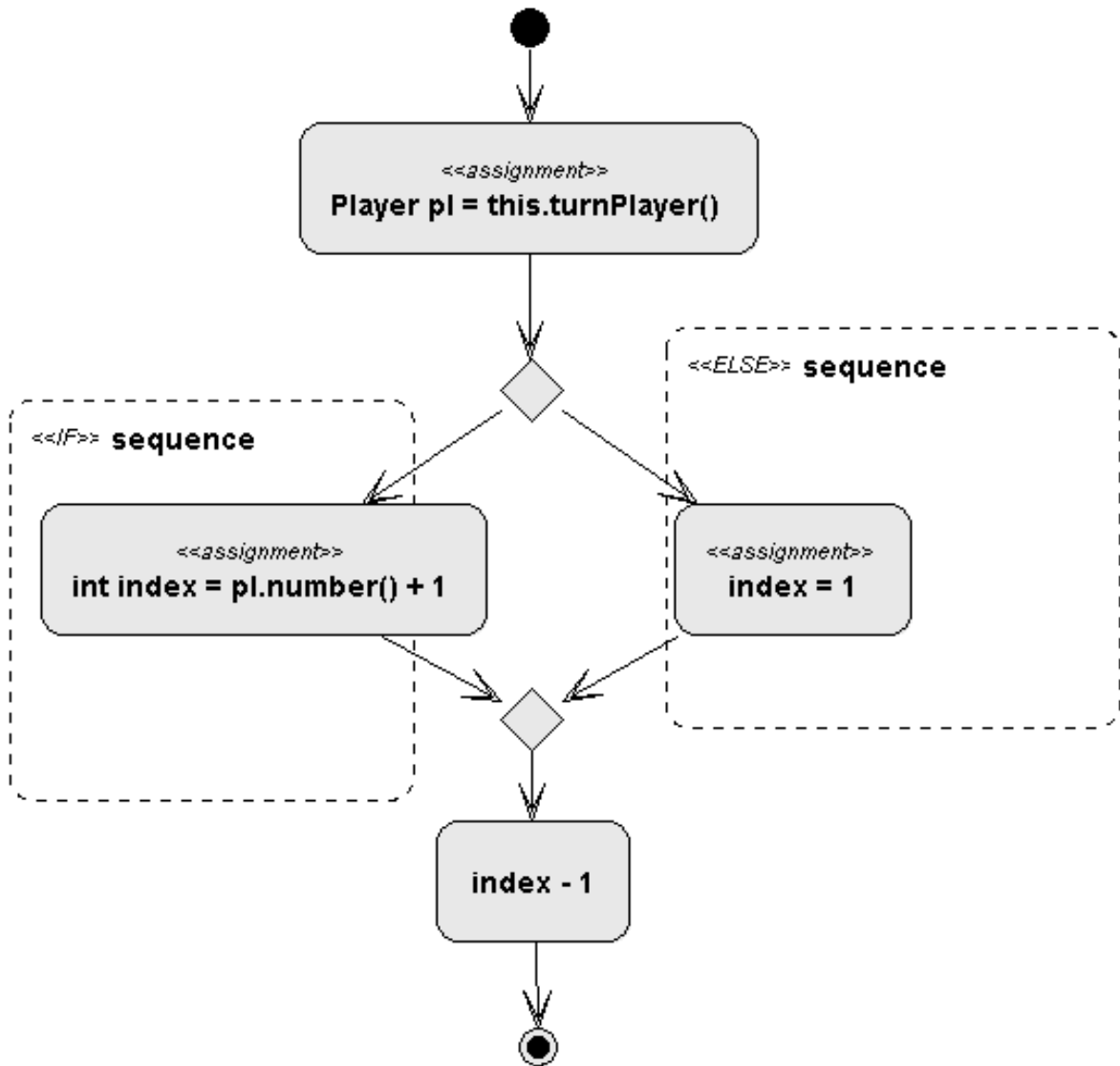


Figura 3.3: Diagrama de atividade representando algoritmo de métodos.

Silva (2007) define os conceitos implementados no ambiente OCEAN/SEA para o suporte do Diagrama de Atividades como ferramenta de modelagem de algoritmos de métodos. São eles:

- *Comment*, correspondente a um comentário no código;
- *Assignment*, representa a atribuição a uma variável local ao método;
- *Return*, apresenta o retorno do método;
- *Message*, define uma chamada a um método;
- *Variable*, definido como uma personalização do comentário, apresenta a declaração de variáveis;

- *Task package*, representa um bloco de código em uma linguagem de programação, sendo copiado diretamente na geração de código;
- *Generic task*, em um alto nível de abstração, corresponde a uma tarefa;
- *If node*, corresponde a um bloco a ser executado no caso da condição explicitada ser verdadeira;
- *If-Else node*, semelhante ao *If node*, define um segundo bloco a ser executado no caso de a condição ser false;
- *Switch node*, define uma escolha entre diversos blocos de código dependendo do valor de determinada variável;
- *While node*, bloco de código a ser executado indefinidamente enquanto determinada condição for verdadeira;
- *Do-While node*, semelhante ao *While node*, tem sua única diferença no teste da condição determinada, que se dá ao final da execução do bloco;
- *For node*, da mesma forma que o *While node*, executa um bloco repetidamente, mas um número limitado de vezes.

Para preencher as atividades correspondentes a comandos do sistema, Silva (2007) propõe a utilização padrão da sintaxe da linguagem Java para comandos que possuem correspondentes nesta linguagem de programação, deixando a critério do desenvolvedor o padrão a ser utilizado nos elementos *Task package* e *Generic task*.

4 MÉTRICAS APLICADAS A ESPECIFICAÇÕES ORIENTADAS A OBJETOS

Conforme citado anteriormente, métricas aplicadas a especificações orientadas a objeto podem ser muito úteis para aferir a qualidade de um artefato de software, seja este um *framework* ou uma aplicação. Pretendem quantificar as características de um software, de modo a identificar problemas no processo de desenvolvimento, além de possibilitar um melhor embasamento para decisões de projeto.

No caso de *frameworks* orientados a objetos, Freiburger (2002) propõe um conjunto de métricas voltadas ao histórico de uso de frameworks orientados a objeto. Já Fonseca (2002) utiliza um conjunto de métricas técnicas¹ voltadas ao apoio no desenvolvimento de aplicações orientadas a objeto.

Este capítulo objetiva apresentar o conjunto de métricas apresentadas por Freiburger (2002) e Fonseca (2002), e que foram implementadas no desenvolvimento deste projeto.

4.1 CONJUNTO DE MÉTRICAS APLICADAS AO DESENVOLVIMENTO E USO DE FRAMEWORKS

Freiburger (2002), em sua definição do conjunto de métricas voltadas ao histórico de uso de *frameworks* orientados a objetos, separa estas métricas em três categorias: quanto a proporção de software produzido e reusado, quanto a especialização de conceitos do *framework* em face aos conceitos desenvolvidos pela aplicação, e quanto as redefinições previstas pelo *framework* e efetuadas pela aplicação em relação aquelas não previstas.

Segue a apresentação do conjunto de métricas desenvolvidas por Freiburger (2002) no que diz respeito ao histórico do uso de *frameworks* orientados a objetos. A definição de cada métrica apresenta seu nome, descrição, e, quando necessário, sua fórmula e modo de interpretação.

¹Concentram-se nas características do software e não no seu processo de desenvolvimento.

4.1.1 QUANTO À PROPORÇÃO DE SOFTWARE PRODUZIDO E REUSADO

A utilização de *frameworks* orientados a objetos se baseia principalmente na reutilização, não apenas do código, mas também da arquitetura do sistema. A utilização de métricas para quantificar o reuso do *framework* pode auxiliar bastante na identificação de problemas como:

- o mau uso *framework*, fazendo com que não seja aproveitado todo seu potencial na aplicação desenvolvida;
- a incompatibilidade entre as funcionalidade providas pelo *framework* e as funcionalidades necessárias à aplicação.

NÚMERO DE CLASSES DO *FRAMEWORK* - FC

Contabiliza o número total de classes exclusivas do *framework*.

NÚMERO DE CLASSES EXCLUSIVAS DA APLICAÇÃO - EC

Contabiliza o número total de classes exclusivas da aplicação.

NÚMERO DE CLASSES DO *FRAMEWORK* REFERENCIADAS DE FORMA DIRETA - DIRC

Refere-se a todas as classes do *framework* que são utilizadas diretamente pela aplicação. São consideradas as situações aonde alguma classe da aplicação referencia a classe do *framework* na forma de:

- um atributo da classe;
- um tipo de retorno de método;
- uma redefinição da classe a partir de herança;
- variáveis temporárias de método;
- chamada a métodos da classe.

NÚMERO DE CLASSES DO *FRAMEWORK* REFERENCIADAS DE FORMA INDIRETA - IRC

Apresenta o número de classes do *framework* utilizadas pela aplicação. O conjunto de classes referenciadas de forma indireta pela aplicação engloba:

1. classes referenciadas diretamente por classes referenciadas diretamente na aplicação; e
2. classes referenciadas diretamente por classes referenciadas indiretamente na aplicação.

TOTAL DE CLASSES DO *FRAMEWORK* REFERENCIADAS PELA APLICAÇÃO - TCFRA

Tomando por CFRD o conjunto de classes referenciadas diretamente pela aplicação (DiRC) e CFRI o conjunto das classes referenciadas indiretamente pela aplicação (IRC), o total de classes do *framework* se dá pela cardinalidade da união destes conjuntos.

$$TCFRA = |CFRD \cup CFRI| \quad (4.1)$$

TOTAL DE CLASSES DA APLICAÇÃO - TC

O total de classes da aplicação se dá pelo total de classes efetivamente utilizadas pela aplicação, dentro do conjunto de todas as classes do projeto.

$$TC = EC + TCFRA \quad (4.2)$$

NÚMERO DE CLASSES DO *FRAMEWORK* NÃO REFERENCIADAS - NORC

Apresenta o número de classes que não foram referenciadas direta ou indiretamente pela aplicação. O valor ideal para esta métrica está próximo de zero, sendo que um valor muito alto pode indicar um pobre aproveitamento das funcionalidades do *framework*.

$$NoRC = TCF - TCFRA \quad (4.3)$$

PERCENTUAL DE REUSO DE CLASSES - CREP

Define o percentual de reuso das classes do *framework* por parte da aplicação. O resultado desta métrica para uma aplicação, pode ser comparado a outras aplicações desenvolvidas sob o mesmo *framework* de forma verificar se esta foge ao padrão de reuso utilizado nas outras aplicações.

A discrepância deste valor entre duas aplicações pode indicar uma utilização do *framework* menor do que o previsto, ou pode indicar que o *framework* não atende a todas as necessidades da aplicação.

$$CReP = \left(\frac{TCFRA}{TC} \right) * 100 \quad (4.4)$$

PERCENTUAL DE DESENVOLVIMENTO DE CLASSES - CPP

Calcula o percentual de classes desenvolvidas para a aplicação, em razão do número total de classes utilizadas pela aplicação.

Um alto percentual de desenvolvimento de classes define que foi necessário o desenvolvimento de muitas classes novas para atender os requisitos da aplicação, podendo indicar o mau uso do *framework*, aonde classes foram desenvolvidas desnecessariamente, ou uma falha no *framework*, que não conseguiu generalizar de forma satisfatória o domínio tratado.

$$CPP = \left(\frac{EC}{TC} \right) * 100 \quad (4.5)$$

NÚMERO DE MÉTODOS DO *FRAMEWORK* - FM

Contabiliza o número total de métodos exclusivos do *framework*.

NÚMERO DE MÉTODOS EXCLUSIVOS DA APLICAÇÃO - EM

Contabiliza o número total de métodos exclusivos da aplicação.

NÚMERO DE MÉTODOS DO *FRAMEWORK* REFERENCIADOS DE FORMA DIRETA - DIRM

Refere-se a todos os métodos do *framework* que são utilizadas diretamente pela aplicação. São considerados aqueles métodos externos à aplicação e utilizados por algum método da aplicação, conforme definido no Diagrama de Atividades de Método.

NÚMERO DE MÉTODOS DO *FRAMEWORK* REFERENCIADOS DE FORMA INDIRETA - IRM

Apresenta a quantidade de métodos que, de algum modo, fazem parte da aplicação. Fazem parte do conjunto de métodos do *framework* referenciados de forma indireta pela aplicação:

1. os métodos referenciados diretamente por algum Diagrama de Atividades dos métodos referenciados diretamente pela aplicação (medida anterior); e
2. os métodos referenciados, através do seu Diagrama de Atividades, indiretamente pela aplicação;

TOTAL DE MÉTODOS DO *FRAMEWORK* REFERENCIADOS PELA APLICAÇÃO - TM-FRA

Tomando por MFRD o conjunto de métodos referenciados diretamente pela aplicação (DiRM) e MFRI o conjunto das métodos referenciados indiretamente pela aplicação (IRM), o total de métodos do *framework* referenciados pela aplicação se dá pela cardinalidade da união destes conjuntos.

$$TMFRA = |MFRD \cup MFRI| \quad (4.6)$$

TOTAL DE MÉTODOS DA APLICAÇÃO - TM

O total de métodos da aplicação se dá pelo total de métodos que podem ser invocados durante a execução da aplicação, dentro do conjunto de todos os métodos do projeto.

$$TM = EM + TMFRA \quad (4.7)$$

NÚMERO DE MÉTODOS DO *FRAMEWORK* NÃO REFERENCIADOS - NORM

Apresenta o número de métodos do *framework* que não foram referenciados direta ou indiretamente pela aplicação. O valor ideal para esta métrica está próximo de zero, sendo que um valor muito alto pode indicar um pobre aproveitamento das funcionalidades do *framework*.

$$NoRM = TM - TMFRA \quad (4.8)$$

PERCENTUAL DE REUSO DE MÉTODOS - MERP

Define o percentual de reuso dos métodos do *framework* por parte da aplicação. A comparação do resultado desta métrica entre diferentes aplicações sob o mesmo *framework* pode demonstrar falhas no projeto da aplicação ou do *framework*.

$$MPP = \left(\frac{TMFRA}{TM} \right) * 100 \quad (4.9)$$

PERCENTUAL DE DESENVOLVIMENTO DE MÉTODOS - MPP

Calcula o percentual de métodos desenvolvidos para a aplicação, em razão do número total de métodos utilizados.

A obtenção de um valor alto no resultado desta métrica pode indicar falhas no projeto do *framework* ou da aplicação.

$$MPP = \left(\frac{EM}{TM} \right) * 100 \quad (4.10)$$

NÚMERO DE ATRIBUTOS DO *FRAMEWORK* - FA

Contabiliza o número total de atributos do *framework*.

NÚMERO DE ATRIBUTOS EXCLUSIVOS DA APLICAÇÃO - EA

Contabiliza o número total de atributos exclusivos da aplicação.

NÚMERO DE ATRIBUTOS DO *FRAMEWORK* REFERENCIADOS DE FORMA DIRETA - DIRA

Refere-se a todos os atributos do *framework* que são utilizados diretamente pela aplicação, a partir de subclasses das classes do *frameworks*.

NÚMERO DE ATRIBUTOS DO *FRAMEWORK* REFERENCIADOS DE FORMA INDIRETA - IRA

Representa a quantidade de atributos do *framework* utilizados pela aplicação. Este número é obtido a partir dos atributos utilizados dentro do Diagrama de Atividades dos métodos refer-

enciados pela aplicação (TMFRA).

TOTAL DE ATRIBUTOS DO *FRAMEWORK* REFERENCIADOS PELA APLICAÇÃO - TAFRA

Tomando por AFRD o conjunto de atributos referenciados diretamente pela aplicação (NiRA) e AFRI o conjunto das atributos referenciados indiretamente pela aplicação (IRA), o total de atributos do *framework* referenciados pela aplicação se dá pela cardinalidade da união destes conjuntos.

$$TAFRA = |AFRD \cup AFRI| \quad (4.11)$$

TOTAL DE ATRIBUTOS DA APLICAÇÃO - TA

O total de atributos da aplicação se dá pelo total de atributos que podem ser utilizado pela aplicação, dentro do conjunto de todos os atributos do projeto.

$$TA = EA + TAFRA \quad (4.12)$$

NÚMERO DE ATRIBUTOS DO *FRAMEWORK* NÃO REFERENCIADOS - NORA

Apresenta o número de atributos do *framework* que não serão acessados durante a execução da aplicação. O valor ideal para esta métrica está próximo de zero, sendo que um valor muito alto pode indicar um pobre aproveitamento das funcionalidades do *framework*.

$$NoRA = TA - TAFRA \quad (4.13)$$

PERCENTUAL DE REUSO DE MÉTODOS - ARP

Define o percentual de reuso dos atributos do *framework* por parte da aplicação. A comparação do resultado desta métrica entre diferentes aplicações sob o mesmo *framework* pode demonstrar falhas no projeto da aplicação ou do *framework*.

$$ARP = \left(\frac{TAFRA}{TA} \right) * 100 \quad (4.14)$$

PERCENTUAL DE DESENVOLVIMENTO DE ATRIBUTOS - APP

Calcula o percentual de atributos criados exclusivamente para a aplicação, em razão do número total de atributos necessários à aplicação.

A obtenção de um valor alto no resultado desta métrica pode indicar falhas no projeto do *framework* ou da aplicação.

$$APP = \left(\frac{EA}{TA} \right) * 100 \quad (4.15)$$

NÚMERO DE COMANDOS DO *FRAMEWORK* -FS

Contabiliza o número total de comandos do *framework*, levando em consideração os comandos referenciados nos Diagramas de Atividades do *framework*.

NÚMERO DE COMANDOS EXCLUSIVOS DA APLICAÇÃO - ES

Contabiliza o número total de comandos exclusivos da aplicação, levando em consideração os comandos referenciados nos Diagramas de Atividades do *framework*.

NÚMERO DE COMANDOS DO *FRAMEWORK* REFERENCIADOS DE FORMA DIRETA - DIRS

Refere-se a todos os comandos do *framework* definidos nos métodos do *framework* referenciados diretamente pela aplicação (DiRM). Desta forma, as métricas que tratam o reuso de comandos do *framework* são semelhantes às métricas que tratam o reuso de métodos, com a diferença que o reuso de métrica trata os métodos de forma ponderada, dando mais importância a métodos maiores.

NÚMERO DE COMANDOS DO *FRAMEWORK* REFERENCIADOS DE FORMA INDIRETA - IRS

Representa a quantidade de comandos do *framework* utilizados pela aplicação. Este número é a contabilização de todos os comandos utilizados pelos métodos referenciados indiretamente pela aplicação (IRM).

TOTAL DE COMANDOS DO *FRAMEWORK* REFERENCIADOS PELA APLICAÇÃO - TCOFRA

Representa o contagem de todos os comandos definidos nos Diagramas de Atividades de todos os métodos do *framework* referenciados pela aplicação (TMFRA).

TOTAL DE ATRIBUTOS DA APLICAÇÃO - TS

O total de comandos da aplicação se dá pela contagem de comandos utilizados na aplicação como um todo.

$$TS = ES + TCOFRA \quad (4.16)$$

NÚMERO DE COMANDOS DO *FRAMEWORK* NÃO REFERENCIADOS - NORS

Apresenta a contagem de comandos do *framework* pertencentes aos métodos não referenciados pela aplicação.

$$NoRS = FS - TCOFRA \quad (4.17)$$

PERCENTUAL DE REUSO DE COMANDOS - SREP

Define o percentual de reuso dos comandos do *framework* por parte da aplicação. A comparação do resultado desta métrica entre diferentes aplicações sob o mesmo *framework* pode demonstrar falhas no projeto da aplicação ou do *framework*.

$$SReP = \left(\frac{TCOFRA}{TS} \right) * 100 \quad (4.18)$$

PERCENTUAL DE DESENVOLVIMENTO DE COMANDOS - SPP

Calcula o percentual de comandos criados exclusivamente para a aplicação, em razão do número total de comandos necessários à aplicação.

A obtenção de um valor alto no resultado desta métrica pode indicar falhas no projeto do *framework* ou da aplicação.

$$SPP = \left(\frac{ES}{TS} \right) * 100 \quad (4.19)$$

4.1.2 QUANTO À ESPECIALIZAÇÃO NO USO DE FRAMEWORKS

Atividade essencial na utilização de *frameworks*, a especialização de classes e métodos permite que certos aspectos sejam especializados na aplicação, moldando seu comportamento

de acordo com seus requisitos.

Medidas aplicadas a quantificação da especialização no uso de *frameworks* podem indicar erros no desenvolvimento da aplicação por má utilização do *framework*, seja por não utilizar todo seu potencial ou por o *framework* não se encaixar perfeitamente com as necessidades da aplicação desenvolvida.

NÚMERO DE CLASSES ESPECIALIZADAS - SP

Define a quantidade de classes do *framework* que foram sobrescritas na aplicação.

NÚMERO DE CLASSES NOVAS - NC

Quantifica as classes que foram desenvolvidas na aplicação, sem nenhuma relação de herança com o *framework*.

PERCENTUAL DE ESPECIALIZAÇÃO DE CLASSES - PEC

Define o percentual de métodos da aplicação que foram especializados a partir de métodos do *framework*.

Quanto maior o percentual obtido, maior é a relação da aplicação desenvolvida com o *framework* utilizado, e provavelmente maior a conformidade entre estes. Um baixo percentual pode indicar que parte do que foi desenvolvido na aplicação já está pronto no *framework*, ou que o *framework* não generaliza corretamente o domínio tratado.

Como a medida não leva em consideração se a classe especializada é definida como redefinível no projeto do *framework*, um alto percentual pode ser gerado também por redefinições inaqueadas e pouco reuso de código, neste caso. Assim, uma avaliação detalhada deve ser feita com a utilização desta em conjunto com outras métricas.

$$PEC = \left(\frac{SC}{EC} \right) * 100 \quad (4.20)$$

PERCENTUAL DE NOVAS CLASSES - PNC

Define o percentual de classes da aplicação que não tem relação de herança com as classes do *framework*.

O objetivo de um *framework* é permitir que várias aplicações dentro de um domínio sejam

desenvolvidas com o menor esforço possível. Um alto percentual de novas classes na aplicação mostra que esta precisou de muitas classes novas, que não foram tratados no *framework* utilizado, ou estão sendo redefinidos desnecessariamente na aplicação.

Várias aplicações com um alto percentual de novas classes, indicam que o *framework* não consegue generalizar corretamente o domínio tratado. Poucas aplicações com alto percentual de novas classes indica que estas aplicações provavelmente não estão fazendo um bom uso do *framework*.

$$PNC = \left(\frac{NC}{EC}\right) * 100 \quad (4.21)$$

NÚMERO DE MÉTODOS ESPECIALIZADOS - SM

Define a quantidade de métodos do *framework* que foram sobrescritos na aplicação.

NÚMERO DE MÉTODOS NOVOS - NM

Quantifica os métodos que foram desenvolvidos na aplicação, sem redefinir nenhum método de sua super classe no *framework*.

PERCENTUAL DE ESPECIALIZAÇÃO DE MÉTODOS - PEM

Define a percentual de métodos da aplicação que foram especializados a partir de métodos do *framework*.

Do mesmo modo que para classes (PEC), um maior percentual indica uma maior conformidade entre o *framework* e a aplicação desenvolvida. Por outro lado um baixo percentual pode indicar uma falha cometida pelo desenvolvedor do *framework*, ao generalizar o domínio tratado, ou pelo desenvolvedor da aplicação, ao reutilizar o código disponível.

$$PEM = \left(\frac{SM}{EM}\right) * 100 \quad (4.22)$$

PERCENTUAL DE NOVOS MÉTODOS - PNM

Define o percentual de métodos da aplicação que não sobrescrevem métodos do *framework*.

De forma semelhante ao percentual de novas classes desenvolvidas, um alto percentual

de métodos novos na aplicação pode indicar uma baixa conformidade entre a aplicação e o framework.

Várias aplicações com um alto percentual de novos métodos indicam que, provavelmente, o *framework* não consegue generalizar corretamente o domínio tratado. Poucas aplicações com alto percentual de novas classes indicam que estas aplicações provavelmente não estão fazendo um bom uso do framework.

$$PNM = \left(\frac{NM}{EM} \right) * 100 \quad (4.23)$$

NÚMERO DE COMANDOS ESPECIALIZADOS - SS

Define a quantidade de comandos do *framework* que foram sobrescritos, a partir da sobrescrita de métodos, na aplicação.

NÚMERO DE COMANDOS NOVOS - NS

Identifica a quantidade de comandos desenvolvidos novos na aplicação a partir de seus métodos não sobrescritos.

PERCENTUAL DE ESPECIALIZAÇÃO DE COMANDOS - PECO

Define a percentual de comandos da aplicação que foram especializados a partir da sobrescrita de métodos do framework.

Esta medida apresenta conclusões semelhantes ao percentual de especialização de métodos (PEM), porém atribui um peso proporcional ao tamanho dos métodos especializados podendo ser, portanto, mais precisa.

$$PECO = \left(\frac{SC}{ES} \right) * 100 \quad (4.24)$$

PERCENTUAL DE NOVOS COMANDOS - PNCO

Define o percentual de comandos da aplicação que estão presentes nos métodos que não foram sobrescritos na aplicação.

Esta medida apresenta conclusões semelhantes ao percentual de novos métodos (PNM),

porém atribui um peso proporcional ao tamanho dos métodos especializados podendo ser, portanto, mais precisa.

$$PNCO = \left(\frac{NS}{ES}\right) * 100 \quad (4.25)$$

NÚMERO DE CLASSES REDEFINÍVEIS DO *FRAMEWORK* - NCRF

Total de classes marcadas como redefiníveis pelo projeto do framework.

NÚMERO DE CLASSES ESPECIALIZADAS E PREVISTAS PELO PROJETO DO *FRAMEWORK* - FOCS

Total de classes especializadas pela aplicação, e definidas como redefiníveis no projeto do framework.

NÚMERO DE CLASSES ESPECIALIZADAS E NÃO PREVISTAS PELO PROJETO DO *FRAMEWORK* - UNCS

Número de classes que foram redefinidas na aplicação, e não foram definidas como redefiníveis pelo projeto do framework.

TOTAL DE CLASSES REDEFINIDAS - TCR

Somatório das classes redefinidas pela aplicação, sendo estas previstas ou não.

$$TCR = NCENP + NCEAP \quad (4.26)$$

PERCENTUAL DE REDEFINIÇÃO PREVISTA - PRP

Define o quanto da redefinição ocorrida no software está dentro do previsto no projeto do framework.

$$PRP = \left(\frac{NCEAP}{TCR}\right) * 100 \quad (4.27)$$

PERCENTUAL DE REDEFINIÇÃO NÃO PREVISTA - PRNP

Define o quanto da redefinição efetuada pela aplicação não foi prevista pelo projeto do framework.

$$PNRN = \left(\frac{NCENP}{TCR} \right) * 100 \quad (4.28)$$

NÚMERO DE MÉTODOS TEMPLATE DO *FRAMEWORK* - TM

Número de métodos definidos explicitamente como template no projeto do framework.

NÚMERO DE MÉTODOS ABSTRATOS DO *FRAMEWORK* - AM

Número de métodos definidos explicitamente como abstratos no projeto do framework.

NÚMERO DE MÉTODOS REGULARES DO *FRAMEWORK* - RM

Número de métodos definidos explicitamente como regulares no projeto do framework.

NÚMERO DE MÉTODOS TEMPLATE SOBRESCRITOS - STEM

Define a quantidade de métodos template do *framework*, redefinidos pela aplicação.

Métodos template tem característica de definir a lógica de um algoritmo através de delegação a métodos hook. Assim, método template devem ser flexibilizados pela sobrescrita de seus métodos hook, e não pela sobrescrita de si mesmo. Qualquer valor diferente de zero alerta para um problema no projeto do *framework*, ou uma falta de entendimento do *framework* por parte do desenvolvedor da aplicação.

NÚMERO DE MÉTODOS ABSTRATOS SOBRESCRITOS - SAM

Quantidade de métodos definidos como abstratos pelo projeto do framework, e sobrescrito pela aplicação.

NÚMERO DE MÉTODOS REGULARES SOBRESCRITOS - SREM

Total de métodos definidos como base pelo projeto do *framework* e sobrescritos na aplicação.

TOTAL DE MÉTODOS SOBRESCRITOS - TMS

Somatório das três classes de métodos sobrescritos pela aplicação.

$$TMS = STeM + SAM + SReM \quad (4.29)$$

PERCENTUAL DE MÉTODOS TEMPLATE SOBRESCRITOS - PMTS

Em razão do total de métodos sobrescritos, calcula qual a porcentagem corresponde a métodos template.

$$PMTS = \left(\frac{STeM}{TMS} \right) * 100 \quad (4.30)$$

PERCENTUAL DE MÉTODOS ABSTRATOS SOBRESCRITOS - PMAS

Em razão do total de métodos sobrescritos, calcula qual a porcentagem corresponde a métodos abstratos.

$$PMAS = \left(\frac{SAM}{TMS} \right) * 100 \quad (4.31)$$

PERCENTUAL DE MÉTODOS REGULARES SOBRESCRITOS - PMRS

Em razão do total de métodos sobrescritos, calcula qual a porcentagem corresponde a métodos regulares.

$$PMRS = \left(\frac{SReM}{TMS} \right) * 100 \quad (4.32)$$

NÚMERO DE MÉTODOS TEMPLATE HOOK SOBRESCRITOS - SHOTEM

Quantidade de métodos definidos no projeto do *framework* como template, e chamados por algum método template, que foram sobrescritos na aplicação.

NÚMERO DE MÉTODOS ABSTRATOS HOOK SOBRESCRITOS - SHAM

Quantidade de métodos declarados como abstratos no projeto do *framework*, e chamados por algum método *template*, que foram sobrescritos na aplicação.

NÚMERO DE MÉTODOS REGULARES HOOK SOBRESCRITOS - SHORM

Quantidade de métodos declarados como regulares no projeto do *framework*, e chamados por algum método *template*, que foram sobrescritos na aplicação.

PERCENTUAL DE MÉTODOS TEMPLATE HOOK SOBRESCRITOS - PMTHS

Define a porcentagem de métodos *template hook* sobrescritos pela aplicação dentro o total de métodos sobrescritos.

$$PMTHS = \left(\frac{SHoTeM}{TMS} \right) * 100 \quad (4.33)$$

PERCENTUAL DE MÉTODOS ABSTRATOS HOOK SOBRESCRITOS - PMAHS

Define a porcentagem de métodos abstratos *hook* sobrescritos pela aplicação dentro o total de métodos sobrescritos.

$$PMAHS = \left(\frac{SHAM}{TMS} \right) * 100 \quad (4.34)$$

PERCENTUAL DE MÉTODOS REGULARES HOOK SOBRESCRITOS - PMRHS

Define a porcentagem de métodos regulares *hook* sobrescritos pela aplicação dentro o total de métodos sobrescritos.

$$PMRHS = \left(\frac{SHoRM}{TMS} \right) * 100 \quad (4.35)$$

4.2 CONJUNTO DE MÉTRICAS APLICADAS AO DESENVOLVIMENTO DE APLICAÇÕES ORIENTADAS A OBJETOS

No que se refere à qualidade de aplicações orientadas a objetos, Fonseca (2002) apresenta um conjunto de métricas aplicadas a verificar a qualidade. Este conjunto de métricas busca informações em relação à qualidade técnica da aplicação produzida, sem levar em consideração o processo utilizado para o desenvolvimento da mesma.

Muitas vantagens podem ser obtidas através da utilização de métricas sobre aplicações orientadas a objetos, podendo verificar a qualidade do produto de acordo com as regras definidas pelo contexto de desenvolvimento do mesmo. O valor obtido em uma métrica pode ser aceitável em um projeto e inadmissível em outro semelhante, tornando necessária uma avaliação humana do resultado das métricas utilizadas. Porém, um possível quadro na utilização destas métricas poderia ser a pré-definição de parâmetros que ativem alertas durante o desenvolvimento quando determinadas medidas alcancem determinados níveis.

Fonseca (2002) separa as medidas em cinco diferentes categorias: acoplamento, encapsulamento, complexidade, coesão e polimorfismo. Conceitos estes bastante difundidos na verificação de qualidade na orientação a objetos. Podem-se descobrir problemas, no projeto de uma aplicação orientada a objetos, que levariam a um menor reuso do código, ou uma maior dificuldade na manutenção do sistema.

4.2.1 MÉTRICAS DE ACOPLAMENTO

As métricas a seguir tem como objetivo quantificar dados sobre o acoplamento entre as classes do projeto da aplicação.

Normalmente, no projeto de aplicações orientadas a objetos, é desejável o menor acoplamento possível. Um alto acoplamento aumenta a chance de ocorrer problemas após modificações na aplicação. Ao serem feitas alterações em uma classe com alto grau de exportação, é necessário que muito mais testes sejam executados para que cubram, além da classe alterada, todas as classes acopladas a classe alterada (FONSECA, 2002).

FATOR DE ACOPLAMENTO - CF

Determina um fator, variando de zero a um, que apresenta o quão acopladas estão as classes de uma aplicação. Um fator de acoplamento zero indica que não ligação entre as classes da aplicação, enquanto um fator de acoplamento máximo indica que todas as classes estão ligadas

entre si.

Apesar de ser praticamente impossível construir uma aplicação sem nenhum acoplamento, é bastante desejável que a aplicação tenha o menor acoplamento possível. Um alto fator de acoplamento aumenta a complexidade da aplicação, dificultando o reuso e manutenibilidade (FONSECA, 2002).

$$CF = \frac{\sum_{i=1}^{TC} [\sum_{j=1}^{TC} cliente(C_i, C_j)]}{TC^2 - TC} \quad (4.36)$$

Onde:

$C = Classe.$

$TC = Total de classes no projeto$

$$cliente = \begin{cases} 1 & \text{se referencia } (C_c, C_s) \text{ e } C_c \neq C_s; \\ 0 & \text{caso contrário.} \end{cases}$$

$referencia(C_c, C_s) = C_c \text{ possui uma referência da classe } C_s.$

ACOPLAMENTO ENTRE CLASSES DE OBJETOS - CBO

Para cada classe, esta métrica obtém o número de outras classes que são referenciadas por esta. Para a obtenção desta métrica foi utilizada a seguinte fórmula:

$$CBO(C) = \sum_{i=1}^{TC} [cliente(C, C_i)] \quad (4.37)$$

ACOPLAMENTO CLASSE-ATRIBUTO POR IMPORTAÇÃO NOS ANCESTRAIS - ACAIC

Identifica as referências do tipo Classe-Atributo que a classe C faz a seus ancestrais, mostrando sua dependência a estas. Referências do tipo Classe-Atributo entre classes C_i e C_j caracterizam-se pelos atributos do tipo C_j presentes na classe C_i.

$$ACAIC(C) = \sum_{C_i \in \text{ancestrais}(C)} ACA(C, C_i) \quad (4.38)$$

$\text{ancestrais}(C) = \text{conjunto de classes ancestrais de } C.$

$ACA(A, B) = \text{o número de interações Classe – Atributo que ocorrem entre as classes } A \text{ e } B.$

ACOPLAMENTO CLASSE-ATRIBUTO POR EXPORTAÇÃO NOS DESCENDENTES - DCAEC

Define a quantidade de referências do tipo Classe-Atributo entre uma classe e seus descendentes.

$$DCAEC(C) = \sum_{C_i \in \text{descendentes}(C)} ACA(C_i, C) \quad (4.39)$$

$\text{descendentes}(C) = \text{conjunto de classes descendentes de } C.$

ACOPLAMENTO POR CLASSE-ATRIBUTO POR IMPORTAÇÃO ENTRE CLASSES SEM RELAÇÃO DE HERANÇA - OCAIC

Define a quantidade de referências do tipo Classe-Atributo, por importação, entre classes que não estão relacionadas hierárquicamente na aplicação. Classes com alto acoplamento por importação tendem a sofrer mais com mudanças ocorridas em outras partes do sistema.

$$OCAIC(C) = \sum_{C_i \in \text{outras}(C)} ACA(C, C_i) \quad (4.40)$$

$\text{outras}(C) = \text{conjunto de classes não relacionadas a } C \text{ por herança.}$

ACOPLAMENTO POR CLASSE-ATRIBUTO POR EXPORTAÇÃO ENTRE CLASSES SEM RELAÇÃO DE HERANÇA - OCAEC

Define a quantidade de referências do tipo Classe-Atributo, por exportação, entre classes que não estão relacionadas hierárquicamente na aplicação. Classes com alto acoplamento por exportação tendem a causar um maior impacto na aplicação ao serem modificadas.

$$OCAEC(C) = \sum_{C_i \in outras(C)} ACA(C_i, C) \quad (4.41)$$

ACOPLAMENTO POR CLASSE-MÉTODO POR IMPORTAÇÃO NOS ANCESTRAIS - ACMIC

Semelhante a medida ACAIC, esta métrica tem por objetivo a quantificação do acoplamento entre uma classe e seus ancestrais, mas desta vez se atendo ao acoplamento do tipo Classe-Método. O acoplamento por Classe-Método entre A e B identifica, dentre os retornos de métodos e parâmetros de A, as referências à classe B.

$$ACMIC(C) = \sum_{C_i \in ancestrais(C)} ACM(C, C_i) \quad (4.42)$$

$ACM(A, B)$ = o número de interações Classe – Método que ocorrem entre as classes A e B.

ACOPLAMENTO CLASSE-MÉTODO POR EXPORTAÇÃO NOS DESCENDENTES - DCMEC

Da mesma forma que a medida DCAEC, esta métrica provê uma quantificação do acoplamento entre cada classe e seus descendentes, mas desta vez a partir das referências do tipo Classe-Método.

$$DCMEC(C) = \sum_{C_i \in descendentes(C)} ACM(C_i, C) \quad (4.43)$$

ACOPLAMENTO CLASSE-MÉTODO POR IMPORTAÇÃO ENTRE CLASSES SEM RELAÇÃO DE HERANÇA - OCMIC

De forma análoga a métrica OCAIC, esta medida proporciona ao desenvolvedor uma idéia do acoplamento por importação entre classes sem relação de herança, desta vez, levando em consideração as referências do tipo Classe-Método.

$$OCMIC(C) = \sum_{C_i \in outras(C)} ACM(C, C_i) \quad (4.44)$$

ACOPLAMENTO CLASSE-MÉTODO POR EXPORTAÇÃO ENTRE CLASSES SEM RELAÇÃO DE HERANÇA - OCMEC

De forma semelhante à métrica OCAEC, define o acoplamento por exportação entre classes não relacionadas hierarquicamente, levando em consideração, desta vez, suas referências Classe-Método.

$$OCMEC(C) = \sum_{C_i \in outras(C)} ACM(C_i, C) \quad (4.45)$$

ACOPLAMENTO MÉTODO-MÉTODO POR IMPORTAÇÃO NOS ANCESTRAIS - AMMIC

Esta métrica tem por objetivo a verificação do acoplamento por importação entre os métodos da classe e os métodos de suas ancestrais. Diferentemente da métrica ACAIC e ACMIC, esta leva em consideração apenas as referências Método-Método que se dá pela invocação de mensagens da classe B pelos métodos da classe A conforme Fórmula 4.2.1.

$$AMMIC(C) = \sum_{C_i \in ancestrais(C)} AMM(C, C_i) \quad (4.46)$$

$AMM(A, B)$ = o número de interações Método – Método que ocorrem entre as classes A e B.

ACOPLAMENTO MÉTODO-MÉTODO POR EXPORTAÇÃO NOS DESCENDENTES - DMMEC

Quantifica, da mesma forma que as métricas DCMEC e DCAEC, o acoplamento por exportação entre métodos da classe e métodos de classes descendentes, mas desta vez utilizando apenas referências do tipo Método-Método.

$$DMMEC(C) = \sum_{C_i \in descendentes(C)} AMM(C_i, C) \quad (4.47)$$

ACOPLAMENTO MÉTODO-MÉTODO POR IMPORTAÇÃO ENTRE CLASSES SEM RELAÇÃO DE HERANÇA - OMMIC

Análogamente às métricas OCAIC e OCMIC, esta medida proporciona uma quantificação do acoplamento por importação entre classes sem relação de herança, levando em consideração apenas as referências do tipo Método-Método.

$$OMMIC(C) = \sum_{C_i \in outras(C)} AMM(C, C_i) \quad (4.48)$$

ACOPLAMENTO MÉTODO-MÉTODO POR EXPORTAÇÃO ENTRE CLASSES SEM RELAÇÃO DE HERANÇA - OMMEC

De forma semelhante às métricas OCMEC e OCAEC, quantifica o acoplamento por exportação entre classes não relacionadas hierarquicamente, levando em consideração apenas suas referências Método-Método.

$$OCMEC(C) = \sum_{C_i \in outras(C)} ACM(C_i, C) \quad (4.49)$$

4.2.2 MÉTRICAS DE ENCAPSULAMENTO

Um dos quatro pilares da Orientação a Objetos, o encapsulamento deve permitir que apenas a própria classe tenha acesso a seus detalhes mais internos, sendo que nenhuma classe deve ser influenciada por mudanças internas a outras classes (BOOCH, 1998).

As métricas definidas abaixo buscam mostrar ao desenvolvedor uma quantificação das partes visíveis ou não da aplicação.

FATOR DE ATRIBUTOS OCULTOS - AHF

Proporciona um fator variando de zero a um, sendo que zero define que todos os atributos são visíveis a todas as classes da aplicação e um define que nenhum atributo está visível a outra classe da aplicação além da própria classe aonde o atributo está declarado.

Idealmente o valor desta métrica deve ser um, pois todo acesso a atributos de uma classe deve ser feito por métodos públicos desta classe. Não atingindo este valor podemos dizer que a aplicação não atende aos critérios de encapsulamento.

$$AHF = \frac{\sum_{i=1}^{TC} Ah(C)}{\sum_{i=1}^{TC} Ad(C)} \quad (4.50)$$

$Ah(C)$ = número de atributos ocultos definidos em C .

$Ad(C)$ = número de atributos definidos em C .

FATOR DE MÉTODOS OCULTOS - MHF

De forma semelhante a métrica AHF, esta medida mostra quanto dos métodos da aplicação estão definidos como ocultos.

Ao contrário de atributos, o resultado desta métrica deve ser tão próximo a um quanto possível, mas nunca igual a um, pois assim as classes não ofereceriam nenhum serviço e não teriam serventia.

$$MHF = \frac{\sum_{i=1}^{TC} Mh(C)}{\sum_{i=1}^{TC} Md(C)} \quad (4.51)$$

$Mh(C)$ = número de métodos ocultos definidos em C .

$Md(C)$ = número de métodos definidos em C .

4.2.3 MÉTRICAS DE COMPLEXIDADE

Uma alta complexidade dos elementos da aplicação desenvolvida não é desejável pois dificulta o reuso e manutenibilidade do código. As métricas a seguir definem a complexidade da aplicação, de acordo com a complexidade de classes e métodos.

TAMANHO DOS MÉTODOS NAS CLASSES - SOM

Calcula a quantidade de *statements* definidos em cada método, descontando-se comentários e a declaração de variáveis. Quanto menor o tamanho de um método, mais fácil é a sua compreensão, manutenção e reuso.

NÚMERO DE ARGUMENTOS NO MÉTODO - NAM

Do mesmo modo que um método grande, um alto número de argumentos no método dificulta a compreensão e manutenção do sistema. Johnson (apud FONSECA, 2002) define que métodos não devem ter mais do que seis argumentos, sendo que métodos que precisam de muitos argumentos devem ser refatorados em métodos menores.

NÚMERO DE MÉTODOS NAS CLASSES - NOM

Calcula a quantidade de métodos definidos na classe. Uma classe com muitos métodos tem uma grande chance de estar classe não abstrair corretamente o contexto proposto, além de dificultar a manutenção, o reuso e a compreensão da classe. Johnson (apud FONSECA, 2002) defende que uma classe deve ter até cinquenta métodos.

NÚMERO DE CLASSES IMEDIATAMENTE DESCENDENTES - NOC

A quantidade de classes imediatamente descendentes a uma classes pode fornecer muitas informações sobre a utilização de herança nas classes. Uma classe com muitos descendentes diretos caracteriza uma hierarquia de classes larga, o que leva a certos indícios sobre a classe:

- Classes com um grande número descendentes indicam uma maior influência na aplicação, causando maiores impactos ao serem efetuadas mudanças;
- Classes com uma uma hierarquia muito larga tendem possuir erros de abstração, pois possuem um menor reuso de classes;
- Classes com uma grande hierarquia denota um maior uso de herança.

O número ideal para o número de classes imediatamente descendentes é muito dependente da classe em questão e deve ser analisado em cada caso.

PROFUNDIDADE DE ÁRVORE DE HERANÇA - DIT

Nesta métrica, atribui-se zero para as classes que não possuem uma superclasse, e a quantidade de classes acima da classe sob análise, na hierarquia de classes. O ideal é que a hierarquia de classes de uma aplicação seja balanceado entre uma hierarquia pouco larga, e pouco profunda.

Uma classe em uma alta profundidade da hierarquia de herança tem sua complexidade aumentada pela quantidade de métodos herdados pela classe. Em contrapartida, indica um maior reuso do código da aplicação. Uma hierarquia pouco larga e pouco profunda geralmente define um bom grau de reuso com limitada complexidade.

FATOR DE HERANÇA DE MÉTODOS - MIF

Variando de zero a um, indica que não há nenhum método não sendo sobrescrito nas subclasses, no caso de zero, mostrando que não há reuso de métodos na aplicação. Este fator é obtido a partir da Fórmula 4.52:

$$MIF = \frac{\sum_{i=1}^{TC} Mi(Ci)}{\sum_{i=1}^{TC} Ma(Ci)} \quad (4.52)$$

$Mi(C) =$ número de métodos herdados e não sobrescritos na classe C .

$Ma(C) =$ número de métodos disponíveis na classe C .

FATOR DE HERANÇA DE ATRIBUTOS - AIF

De forma semelhante ao fator de herança de métodos, esta métrica demonstra a relação entre atributos herdados de outras classes na hierarquia de herança de classes, e os atributos definidos na própria classe.

$$AIF = \frac{\sum_{i=1}^{TC} Ai(Ci)}{\sum_{i=1}^{TC} Aa(Ci)} \quad (4.53)$$

$Ai(C) =$ número de atributos herdados na classe C .

$Aa(C) =$ número de atributos disponíveis na classe C .

REAÇÃO DE UMA CLASSE - RFC

Define o conjunto de métodos potencialmente invocados ao ser efetuada a invocação de algum método da classe. Sendo composta pelos métodos da classe, mais os métodos invocados

em primeiro nível pelos métodos da classe, um alto valor para esta métrica indica uma alta complexidade da classe e faz com que o entendimento e testes da classe sejam dificultados.

$$RFC = |M \cup_{todosi} R_i| \quad (4.54)$$

$M = \text{métodos da classe.}$

$R_i = \text{métodos invocados por métodos da classe.}$

MÉTODOS COMPLEXOS POR CLASSE - WMC

Demonstra um contador para os métodos da classe, ponderando sua complexidade. Métodos pouco complexos, ideais à aplicação, tem peso igual a um. Métodos complexos tem seu peso igual a dois.

Idealmente, o valor para esta métrica deve ser igual ao número de métodos da classe, indicando que não há nenhum método complexo na classe. Para este trabalho, conforme defendido por Johnson (apud FONSECA, 2002) foi utilizado neste trabalho um limite de trinta comandos para que o método não seja considerado complexo, levando em consideração as regras definidas na métrica SOM.

$$WMC = \sum_{i=0}^n c_i \quad (4.55)$$

$$c_i(M) = \begin{cases} 1 & \text{se SOM}(M) \leq 30 \\ 2 & \text{caso contrário.} \end{cases}$$

REFERÊNCIA À SUBCLASSES - RSC

Se caracteriza por quantificar as referências que a superclasse faz às suas subclasses. Qualquer valor diferente de zero nessa métrica considera a abstração de classes incorreta, pois uma classe em um nível superior da hierarquia de herança nunca deve depender de classes em níveis inferiores da hierarquia (ROCHA J. C. MALDONADO, 2001).

4.2.4 MÉTRICAS DE COESÃO

A coesão de uma classe é caracterizada por métodos e atributos que caracterizam bem o domínio modelado. Uma abstração coesa do domínio tratado facilita a compreensão e manutenção da aplicação.

FALTA DE COESÃO MÉTODOS - LCOM

Esta métrica indica a falta de coesão entre os métodos da classe. Isto é medido comparando os atributos referenciados a cada par de métodos e verificando se estes são diferentes, conforme Fórmula 4.56.

$$LCOM(C) = \begin{cases} |P| - |Q| & \text{—P—i—Q—} \\ 0 & \text{caso contrário.} \end{cases} \quad (4.56)$$

$$P = \{(I_i, I_j) \mid I_i \cap I_j = 0\}$$

$$Q = \{(I_i, I_j) \mid I_i \cap I_j \neq 0\}$$

$$I(i) = \text{o conjunto de atributos utilizados pelo método } M_i. \quad (4.57)$$

FALTA DE COESÃO NOS MÉTODOS MODIFICADO - MLCOM

Visando melhorar a precisão da métrica LCOM, Fonseca (2002) propõe a contabilização dos atributos utilizados também por métodos invocados diretamente.

Isso significa que, dada a Fórmula 4.56, substitui-se a função $I(i)$, conforme declarado abaixo:

$I(i)$ = o conjunto de atributos utilizados pelo método M_i , ou por alguns dos métodos invocados diretamente.

4.2.5 MÉTRICAS DE POLIMORFISMO

O polimorfismo traz ao projeto do software a oportunidade de aumentar o reuso, diminuindo a complexidade do projeto, mas aumentando a dificuldade para compreender o projeto.

Benlarbi e Melo (apud FONSECA, 2002) define o polimorfismo em três diferentes grupos:

- Polimorfismo puro: métodos de mesmo nome e diferentes assinaturas no escopo da classe;
- Polimorfismo estático: métodos de mesmo nome e diferentes assinaturas, em diferentes classes;
- Polimorfismo dinâmico: métodos de mesmo nome e mesma assinatura de métodos herdados e sobrescritos na classe.

FATOR DE POLIMORFISMO - PF

De uma forma geral, quantifica o uso de polimorfismo na aplicação. Podendo estar entre zero e um, sendo que zero indica que não foi utilizado polimorfismo na aplicação, e um indicando que todas as possibilidades de uso de polimorfismo foram aproveitadas, esta métrica analisa a especificação como um todo, sem levar em consideração cada classe individualmente.

$$PF = \frac{\sum_{i=0}^{TC} Mo(Ci)}{\sum_{i=1}^{TC} Mn(Ci) * DC(Ci)} \quad (4.58)$$

$Mn(Ci) =$ número de métodos novos na classe Ci ;

$Mo(Ci) =$ número de métodos sobrescritos na classe Ci ;

$DC(Ci) =$ número de descendentes da classe Ci ;

SOBRECARGA DE CLASSES ISOLADAS - OVO

Define a quantidade de sobrecarga de métodos, ou polimorfismo puro, existem em uma classe. Utilizado apenas dentro do escopo de uma única classe, o polimorfismo puro trata apenas dos métodos sobrecarregados, de acordo com o seguinte cálculo:

$$OVO(C) = \sum_{f_i \in C} overl(f_i, C) \quad (4.59)$$

$overl(f_i, C) =$ define o número de vezes que a função f_i é sobrecarregada em C .

POLIMORFISMO ESTÁTICO NOS ANCESTRAIS - SPA

O polimorfismo estático nos ancestrais define se, foram efetuadas sobrecargas de métodos definidos nos seus ancestrais. Ou seja, houve uma redefinição de função com mesmo de uma função em algum ancestral, mas com assinatura diferente.

$$SPA(C) = \sum_{C_i \in \text{Ancestrais}} SPoly(C_i, C) \quad (4.60)$$

$SPoly(C_i, C) =$ define o número de funções polimórficas entre C_i e C .

POLIMORFISMO ESTÁTICO NOS DESCENDENTES - SPD

Do mesmo modo que a métrica anterior, esta tem por objetivo verificar o polimorfismo estático em cada classe, desta vez levando em consideração o conjunto de suas classes descendentes.

$$SPD(C) = \sum_{C_i \in \text{Descendentes}} SPoly(C_i, C) \quad (4.61)$$

POLIMORFISMO ESTÁTICO EM RELAÇÃO DE HERANÇA - SP

Da mesma forma que as métricas SPA e SPD, esta leva em consideração o polimorfismo estático nas relações de herança, desta vez agrupando as classes ancestrais e descendentes.

$$SP(C) = \sum_{C_i \in (\text{Descendentes} \cup \text{Ancestrais})} SPoly(C_i, C) \quad (4.62)$$

POLIMORFISMO DINÂMICO NOS ANCESTRAIS - DPA

Trata-se da sobrescrita de métodos dos ancestrais, correspondendo ao número de métodos herdados e sobrescritos na classe.

$$DPA(C) = \sum_{C_i \in \text{Ancestrais}} DPoly(C_i, C) \quad (4.63)$$

$DPoly(C_i, C) =$ define o número de funções definidas em C_i e sobrescritas em C .

POLIMORFISMO DINÂMICO NOS DESCENDENTES - DPD

De forma análoga à métrica anterior, trata das relações polimórficas dinâmicas, mas levando em consideração desta vez os seus descendentes.

$$DPD(C) = \sum_{C_i \in \text{Descendentes}} DPoly(C_i, C) \quad (4.64)$$

POLIMORFISMO DINÂMICO - DP

Engloba todas as relações polimórficas dinâmicas encontradas em DPA e DPD, tratando assim das duas direções na árvore de herança.

$$DP(C) = \sum_{C_i \in (\text{Descendentes} \cup \text{Ancestrais})} DPoly(C_i, C) \quad (4.65)$$

POLIMORFISMO EM RELAÇÃO SEM HERANÇA - NIP

Apesar de não caracterizar o polimorfismo, métodos com o mesmo nome em diferentes classes não relacionadas por herança podem gerar confusão quanto ao entendimento do contexto das classes. Esta métrica utiliza os mesmos critérios para avaliação de polimorfismo estático e dinâmico das métricas anteriores, mas desta vez leva em consideração apenas classes não relacionadas por herança com a classe sob análise.

$$NIP(C) = \sum_{C_i \in \text{outras}} [DPoly(C_i, C) + SPoly(C_i, C)] \quad (4.66)$$

5 REENGENHARIA DE FERRAMENTAS DE EXTRAÇÃO DE MÉTRICAS ORIENTADAS A OBJETO

Fonseca (2002) e Freiburger (2002) produziram ferramentas para a extração de métricas em aplicações e *frameworks* orientados a objetos, respectivamente. Originalmente desenvolvidas para integrarem o ambiente SEA criado por Silva (2000), não fizeram parte do escopo da reengenharia promovida por Coelho (2007), Machado (2007), Amorim (2006) e Vargas (2008).

Além de efetuar a reengenharia das ferramentas supracitadas, este trabalho propõe-se a integrá-las em uma ferramenta única que aborda todos os aspectos apresentados no capítulo anterior referentes à qualidade de aplicações e *frameworks* orientados a objetos. A ferramenta desenvolvida a partir das criações de Fonseca (2002) e Freiburger (2002) deve ser capaz de calcular todas as métricas especificadas no capítulo anterior, a partir de uma especificação orientada a objetos do ambiente OCEAN/SEA.

5.1 ANÁLISE DAS ESPECIFICAÇÕES

Considerando a semelhança conceitual entre as ferramentas desenvolvidas por Freiburger (2002) e Fonseca (2002), se faz possível a reengenharia destas em apenas uma ferramenta, que englobe o cálculo de métricas voltadas a aplicações orientadas a objetos, e o cálculo sobre o histórico de uso de *frameworks* orientados a objetos. Para tanto, é necessária uma análise de ambas especificações em busca de semelhanças e divergências entre elas.

Em um primeiro nível de análise, ambas as ferramentas leem dados da especificação e, a partir de fórmulas matemáticas, transformam estes dados em métricas orientadas a objetos. Pode-se notar, no entanto, que as principais divergências entre suas especificações estão no modo de obtenção e armazenamento dos dados da especificação.

Freiburger (2002) define um conjunto de classes para o armazenamento de dados de especificações de *frameworks* e aplicações orientadas a objetos, fazendo uma ponte entre a especificação em seu formato original e a ferramenta de cálculo das medidas definidas. Este conjunto permite o armazenamento dos dados da especificação, além de permitir que o conjunto de métricas seja

definido posteriormente à extração de dados da especificação.

Fonseca (2002), por sua vez, não define um conjunto de classes capazes de armazenar os dados da especificação, utilizando para a geração de métricas a própria especificação, extraindo desta um conjunto de dados semelhantes aos utilizados por Freiburger (2002). A especificação é, então, repassada à ferramentas auxiliares que buscam nela os dados necessários a geração de cada métrica específica.

Nota-se então a semelhança nos dados utilizados por ambas as ferramentas e cria-se a possibilidade de uma extração única dos dados da especificação. Podemos identificar as peculiaridades das duas abordagens:

- Extração dos dados para uma estrutura de dados separada:
 - cria o passo adicional da extração dos dados para a estrutura de dados;
 - torna o cálculo das métricas mais simples devido aos dados estarem mais facilmente acessíveis;
 - devido ao cálculo mais simples das métricas, facilita na criação de novas medidas que utilizem o mesmo conjunto de dados.
- Extração dos dados diretamente da especificação:
 - consegue obter os resultados das métricas em apenas um passo;
 - torna desnecessária a criação de mais um conjunto de classes para a representação de uma especificação;

Para este trabalho, foi escolhido utilizar a técnica abordada por Freiburger (2002), adicionando ao conjunto de dados proposto os dados necessários ao cálculo das métricas propostas por Fonseca (2002). Esta decisão foi tomada, principalmente, por permitir que novas métricas sejam facilmente desenvolvidas no sistema.

No que se refere à visualização das informações ambas as ferramentas se utilizam de ferramentas desenvolvidas externamente à ferramenta de extração de métricas. Aproveitando a expressividade e extensividade da linguagem Java, na qual o sistema OCEAN/SEA está desenvolvido atualmente, a visualização dos resultados será feita a partir da própria ferramenta, dentro do ambiente OCEAN/SEA.

5.2 TRABALHOS CORRELATOS

Outras ferramentas, semelhantes a ferramenta desenvolvida neste trabalho, foram criadas com o objetivo de analisar softwares orientados a objetos a partir da análise de métricas.

A ferramenta Odyssey-Metrics (COPPE, 2006) foi desenvolvida pela Equipe de Reutilização de Software do Laboratório de Engenharia de Software da Universidade Federal do Rio de Janeiro com o objetivo de apoiar a utilização de métricas com o ambiente Odyssey, que possui suporte ao projeto de aplicações orientadas a objetos.

Outra ferramenta com propósito semelhante é a SDMetrics, desenvolvida na Alemanha (SDMETRICS, 2007), que utiliza modelos UML em XMI¹ como entrada de uma forma genérica, sem estar integrado a um ambiente de projeto de aplicações orientadas a objetos.

Em uma análise comparativa entre as duas ferramentas apresentadas e a ferramenta desenvolvida neste trabalho, podemos destacar alguns pontos:

- Métricas implementadas na ferramenta: enquanto esta ferramenta implementa setenta diferentes métricas para a análise de projetos de frameworks orientados a objetos e trinta e sete métricas referente a análise de aplicações, a ferramenta Odyssey-Metrics apresenta inicialmente catorze métricas para a análise de aplicações orientadas a objetos, e a ferramenta SDMetrics pode calcular inicialmente cinquenta e oito métricas sobre aplicações orientadas a objetos;
- Suporte ao desenvolvimento de novas métricas pelo usuário: neste contexto, ao contrário da ferramenta desenvolvida neste trabalho, ambas as ferramentas apresentadas possuem uma linguagem para a especificação de métricas não contidas em seu conjunto inicial;
- Integração com ferramenta de projetos: dentre as ferramentas analisadas, a ferramenta SDMetrics é a única que trabalha de forma genérica, utilizando dados genéricos de modelos XMI;
- Suporte a métricas referentes ao uso de frameworks orientados a objetos: nenhuma das ferramentas analisadas prevê a utilização de métricas sobre frameworks orientados a objetos.

Levando em consideração os dados levantados com a análise de ferramentas correlatas, podemos verificar que a ferramenta proposta neste trabalho tem seu principal diferencial no

¹XML Metadata Interchange: padrão desenvolvido pela OMG para a definição de modelos UML.

suporte a métricas que apóiem o desenvolvimento de frameworks orientados a objetos, tendo seu ponto fraco em não possuir um suporte para definição de métricas adicionais ao conjunto original.

5.3 REENGENHARIA DAS FERRAMENTAS

Tendo em vista as decisões tomadas em razão da análise da especificação das ferramentas, a reengenharia se deu com foco na ampla reutilização dos conceitos desenvolvidos na aplicação original, empregando novas técnicas e tecnologias para o devido acoplamento da ferramenta ao ambiente OCEAN/SEA.

Com a união das ferramentas, os casos de uso da ferramenta se resumem a dois, já que com a utilização de apenas uma ferramenta para a obtenção e visualização de todas as métricas definidas, sua utilização se torna mais simples do que o uso de duas ferramentas. A figura 5.1 mostra os casos de uso desenvolvidos na ferramenta.

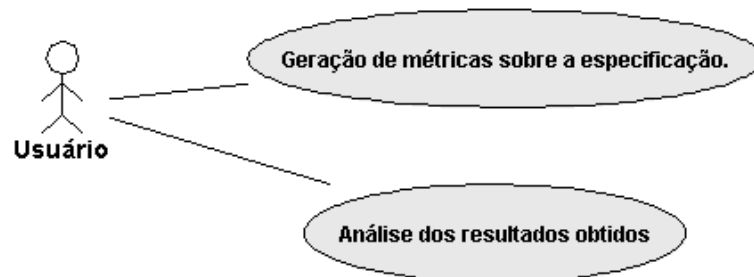


Figura 5.1: Diagrama de casos de uso da ferramenta.

A ferramenta desenvolvida calcula as métricas de aplicações e frameworks orientados a objetos a partir de duas etapas distintas. Primeiramente, são obtidos os dados da especificação sob análise e armazenados em uma estrutura de dados intermediária. Com base nos dados obtidos, a ferramenta calcula o valor das métricas definidas, e apresenta graficamente ao usuário. A figura 5.2 demonstra o início da ferramenta e seu fluxo até a obtenção das métricas.

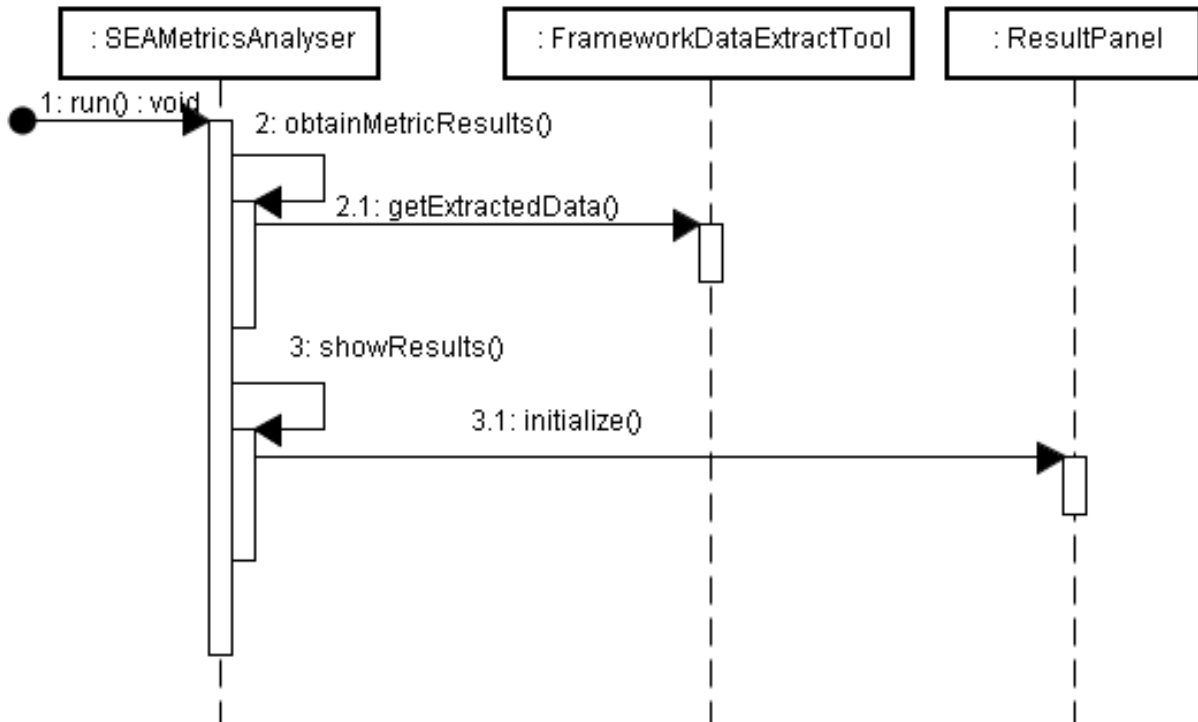


Figura 5.2: Diagrama de sequência da ferramenta.

5.3.1 OBTENÇÃO DOS DADOS DA ESPECIFICAÇÃO

Utilizando a estrutura de classes proposta por Freiburger (2002), foi possível obter e armazenar as informações da especificação sendo tratada, facilitando o cálculo de métricas e separando a ferramenta em duas partes distintas, com menor complexidade. A figura 5.3 demonstra esta estrutura sem as modificações propostas neste trabalho.

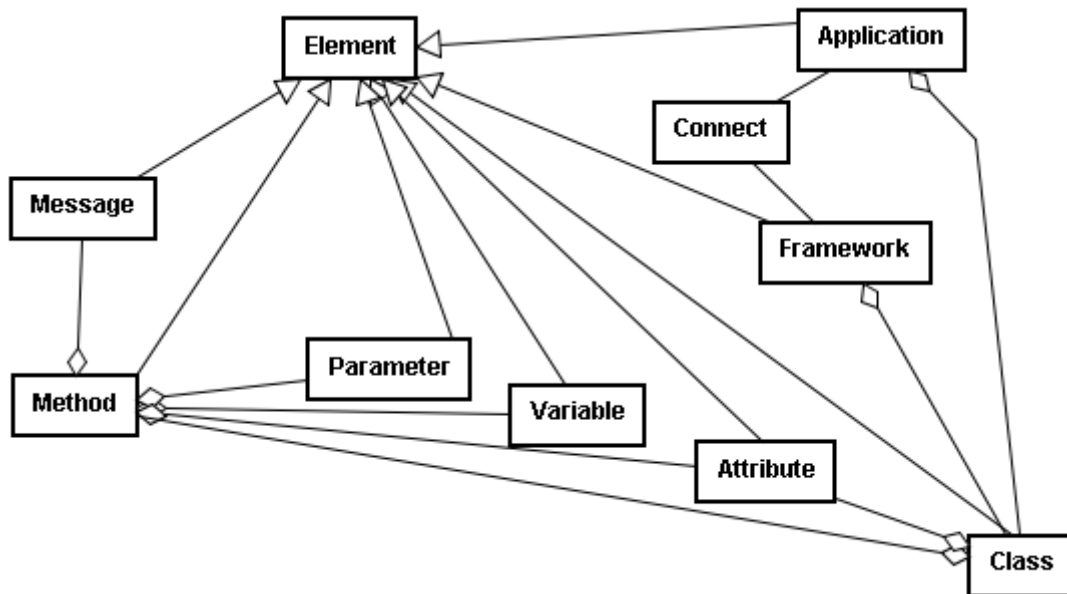


Figura 5.3: Estrutura para armazenamento dos dados da especificação original. Fonte: (FREIBERGER, 2002)

A utilização neste trabalho se deu com algumas modificações pontuais, de forma a permitir tanto o armazenamento dos dados de ambos os tipos de especificações orientadas a objetos, além de facilitar a implementação da ferramenta.

O armazenamento de aplicações e *framework* se dá a partir das classes *Application* e *Framework*, descendentes de classe *ClassContainer*, criada com o objetivo de generalizar os pontos em comum entre aplicações e frameworks e promover o reuso de código na obtenção dos dados da especificação. Já que a ferramenta é desenhada com foco em *frameworks* orientados a objetos, além de aplicações, a classe *Connect* promove a ligação entre *framework* e aplicação, conforme apresentado na figura 5.4. Nos diagramas apresentados, os métodos de acesso direto aos atributos foram omitidos para diminuir o tamanho da representação das classes.

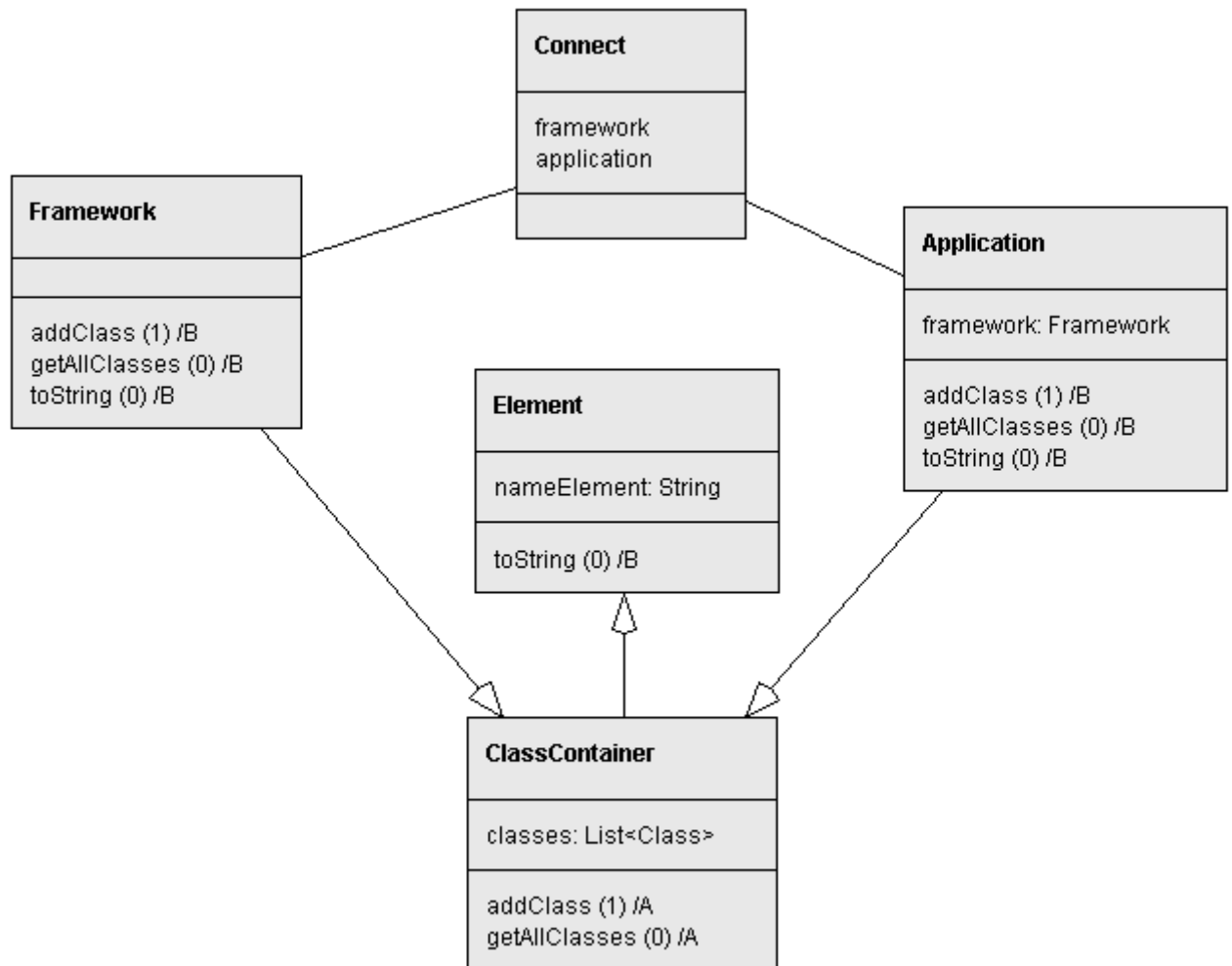


Figura 5.4: Representação de aplicações e frameworks.

Seguindo a estrutura de armazenamento, a figura 5.5 mostra o relacionamento das classes, com seus atributos e métodos. A estrutura de classes e suas associações permanecem pouco alteradas, já que ambas as ferramentas se utilizavam praticamente dos mesmos dados da especificação. Os dados de classe obtidos da especificação são basicamente as suas propriedades, como sua superclasse, e suas associações, como seus atributos e métodos.

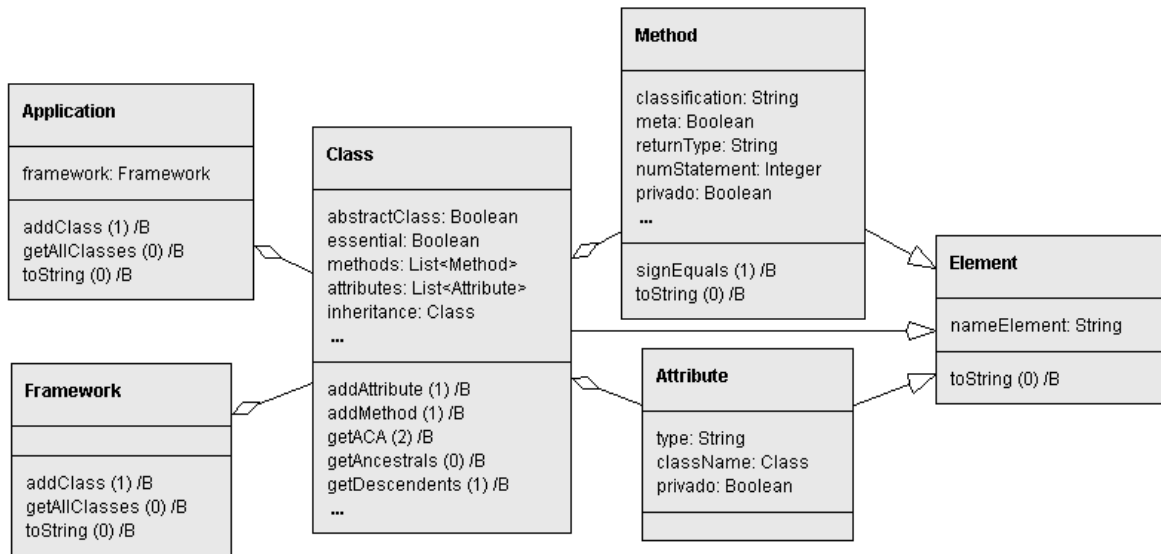


Figura 5.5: Representação de classes e métodos na estrutura de armazenamento.

Dentre os conceitos presentes em especificações orientadas a objetos, a representação de um método é, provalmente, a mais complexa delas, pois engloba além do seu estado, suas mensagens, atributos, variáveis e parâmetros, conforme apresentado na figura 5.6. A lista de atributos contida em cada método se refere aos atributos acessados, ou alterados por este durante a sua execução e é obtida durante a avaliação do diagrama de atividades correspondente ao método.

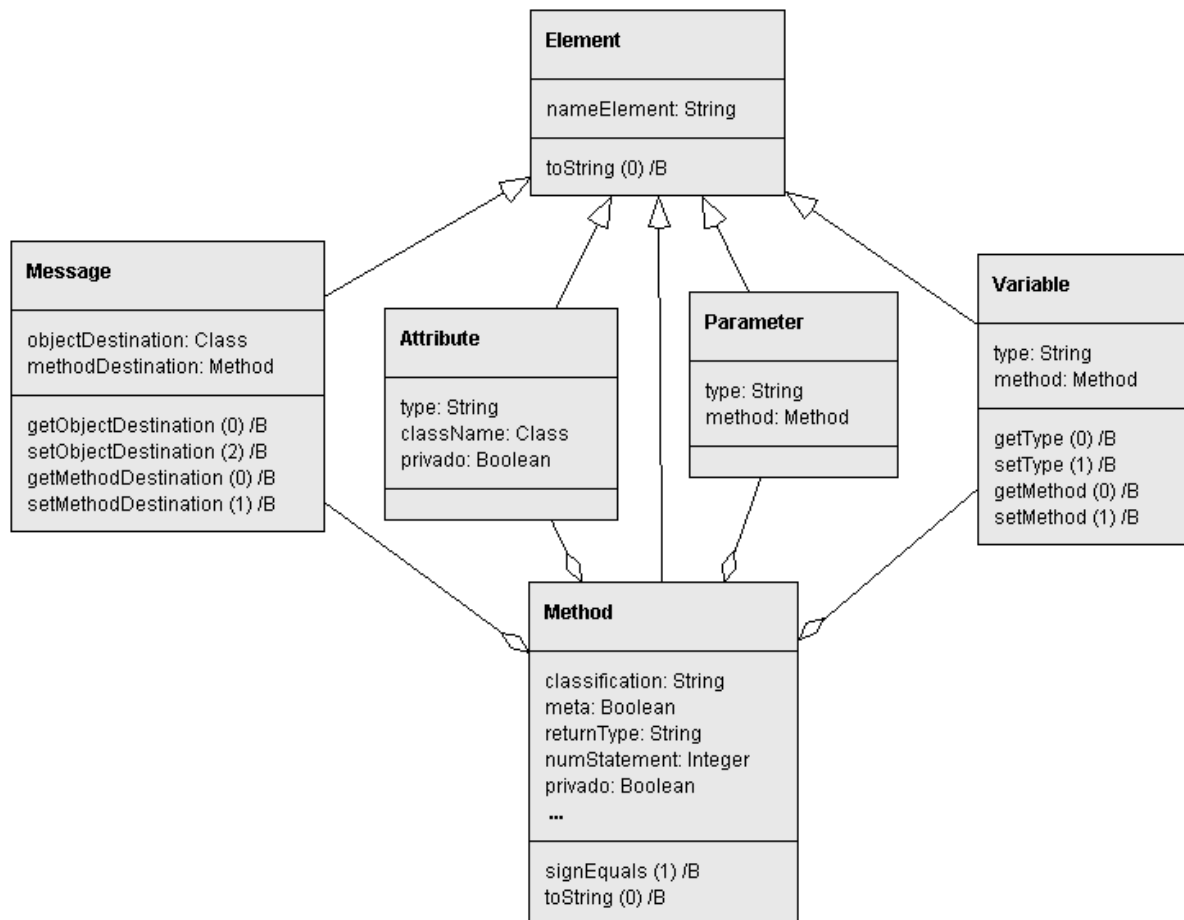


Figura 5.6: Representação de métodos e seus agregados na estrutura de armazenamento.

A extração dos dados da especificação se dá a partir de dois diagramas chave. O Diagrama de Classes permite a obtenção os dados sobre classes, com seus métodos e atributos, e as conexões entre estas. O Diagrama de Atividades, conforme proposto por Silva (2007), possui os dados referentes aos algoritmos de métodos da especificação, e permite a ferramenta obter relacionamentos muitas vezes ocultos entre os objetos do sistema.

Como o Diagrama de Atividades, conforme proposto por Silva (2007) para a definição de algoritmos, não possui no ambiente SEA uma ligação com os elementos que cada atividade representa, esta conexão entre os conceitos presentes no diagrama é feita pela ferramenta de extração de métricas a partir da comparação das referências nas atividades com os conceitos obtidos a partir do Diagrama de Classes da especificação. Por padrão cada elemento é analisado levando-se em consideração a utilização da linguagem Java, e cada elemento do tipoTask package e *Generic task* considera um incremento de uma unidade no número de *statements* no método, mas não tem seu conteúdo interno analisado.

Sendo voltado para a especificação de artefatos reusáveis como *frameworks* e compo-

mentes, além de aplicações orientadas a objetos, o ambiente OCEAN/SEA permite que uma especificação defina um conjunto de outras especificações, como *frameworks*, de onde herdará seus conceitos. Assim, a partir de uma especificação, a ferramenta busca especificações definidas na especificação analisada, e especificações que definem a especificação analisada como “pai”. Assim, é possível analisar simultaneamente as métricas dos artefatos em um nível hierárquico superior ou inferior à especificação analisada, permitindo, principalmente, a análise comparatória entre diferentes aplicações desenvolvidas sob um mesmo framework.

5.3.2 CÁLCULO DAS MÉTRICAS E APRESENTAÇÃO DOS RESULTADOS

Após analisar as especificações de ambas as ferramentas foi definida e elaborada a nova especificação, que reúne os conceitos em comum e tratando suas divergências de forma a promover os resultados de ambas as ferramentas em uma única. Além da ferramenta em si, foi necessário desenvolver funcionalidades do ambiente para permitir à ferramenta a obtenção de seus resultados.

Após a extração dos dados das especificações, a ferramenta se divide basicamente em duas tarefas distintas: calcular as métricas de aplicações e frameworks orientados a objetos, e apresentar o resultado obtido ao usuário. A figura 5.7 demonstra a interconexão entre as classes que compõe a ferramenta, deixando clara a divisão entre: a extração dos dados da especificação (*FrameworkDataExtractTool*), o cálculo de métricas de aplicações orientadas a objetos (*ApplicationMetricsCalculator*), o cálculo de métricas de frameworks orientados a objetos (*FrameworkMetricsCalculator*) e a visualização dos resultados (*ResultPanel*).

SEAMetricsAnalyser descende diretamente da classe abstrata *OceanTool*, o que a define como uma ferramenta do ambiente OCEAN/SEA, responsável pelo controle da ferramenta, em mais baixo nível, e pela visualização de sua interface gráfica.

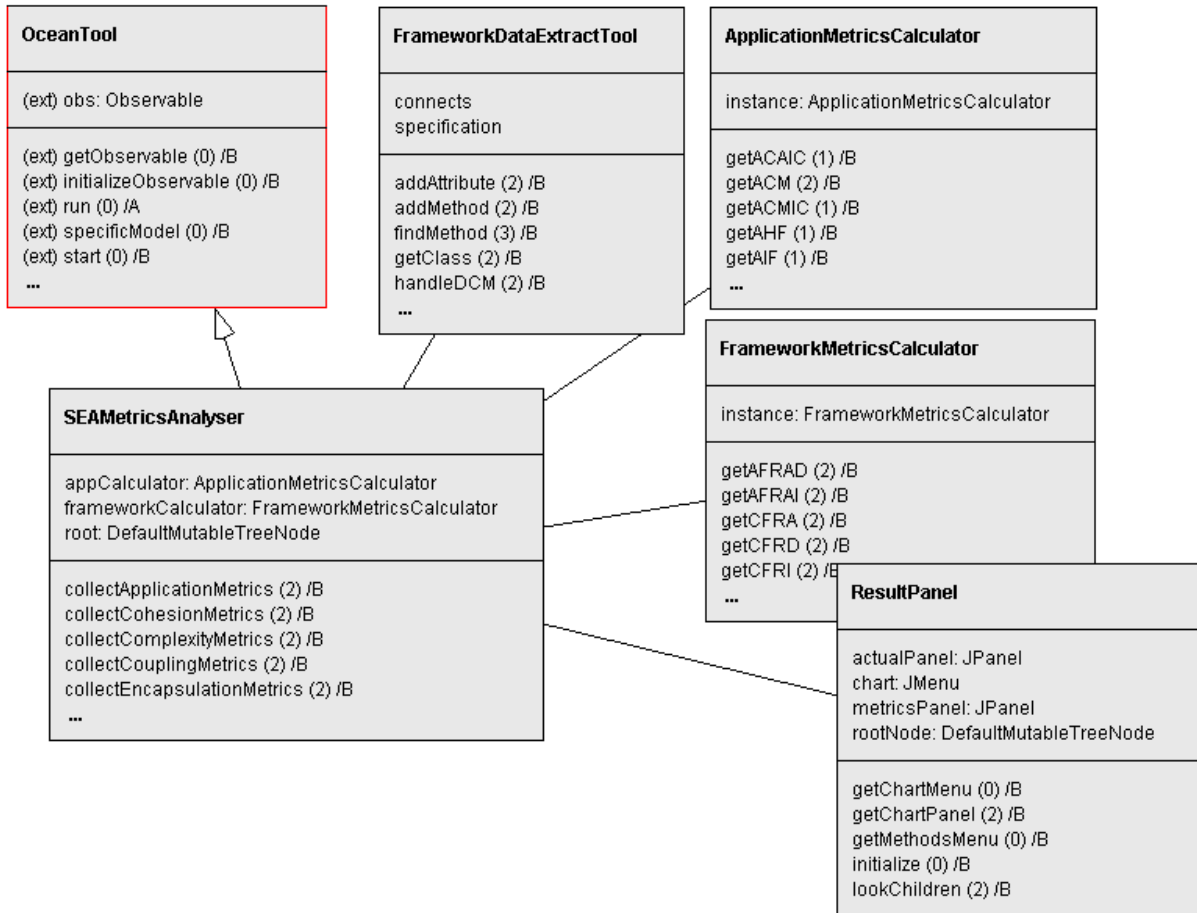


Figura 5.7: Classes da ferramenta de extração e análise de métricas orientadas a objetos.

A figura 5.8 apresenta a sequência de métodos chamados a partir da invocação da ferramenta, refinando o diagrama mostrado na figura 5.2 no que diz respeito ao cálculo das métricas a partir dos dados da especificação. Não estão inclusos na imagem a obtenção de cada métrica individualmente, nem a obtenção dos dados da especificação.

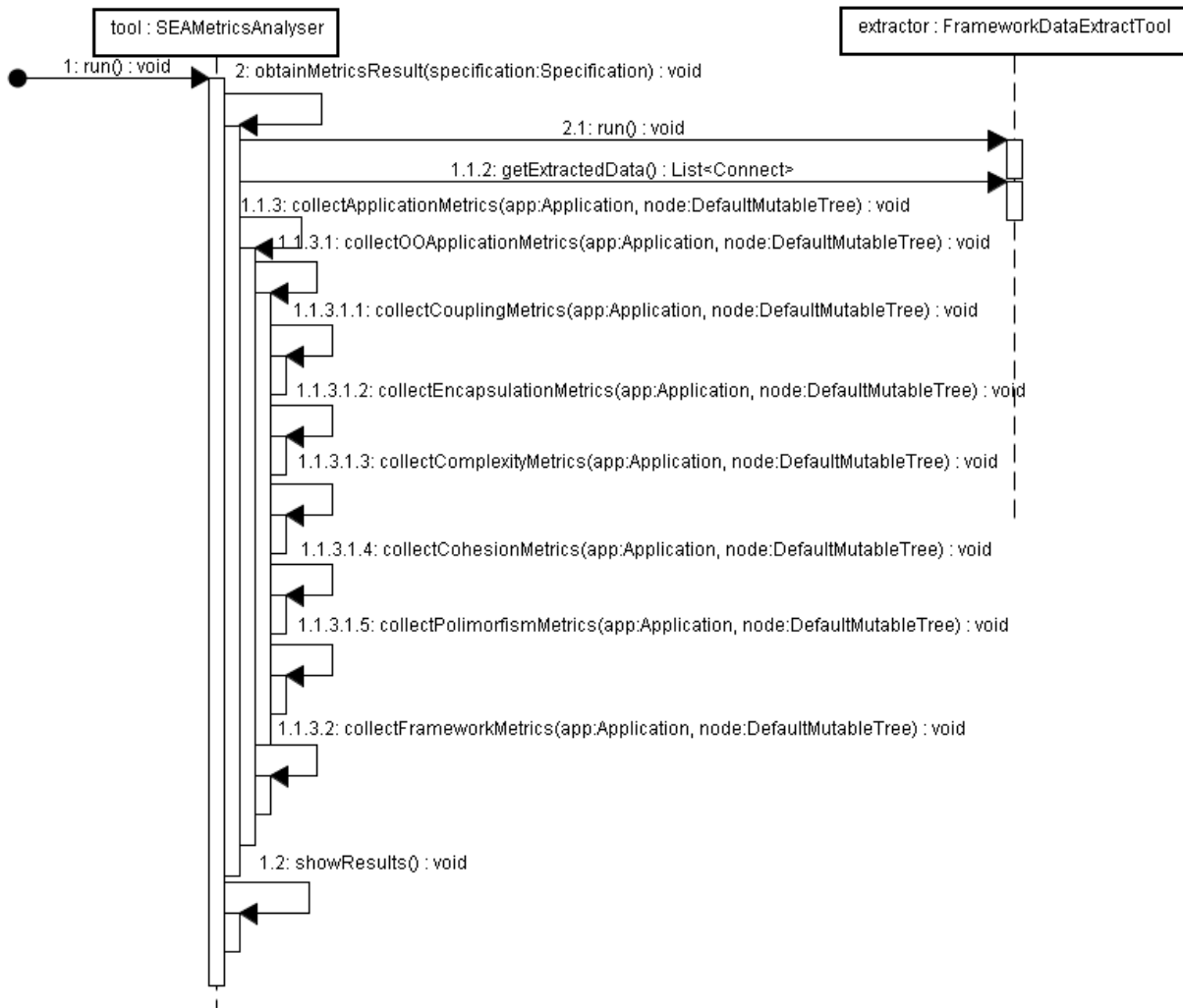


Figura 5.8: Sequenciamento dos métodos a partir da invocação da ferramenta.

De forma hierárquica, a obtenção das métricas ocorre pela chamada aos métodos das classes *ApplicationMetricsCalculator* e *FrameworkMetricsCalculator*, sendo que cada uma das métricas implementadas têm seu acesso a partir de um método específico.

A visualização dos dados da ferramenta se dá em formato de árvore, apresentando hierarquicamente as métricas obtidas, em uma janela interna ao ambiente OCEAN/SEA. A ferramenta suporta também a visualização de diversas aplicações, desenvolvidas sobre um mesmo *framework*, permitindo uma análise comparativa sobre o uso de *frameworks* orientados a objetos. A figura 5.9 demonstra a ferramenta desenvolvida em uso, apresentando dados sobre duas aplicações desenvolvidas sobre um mesmo framework.

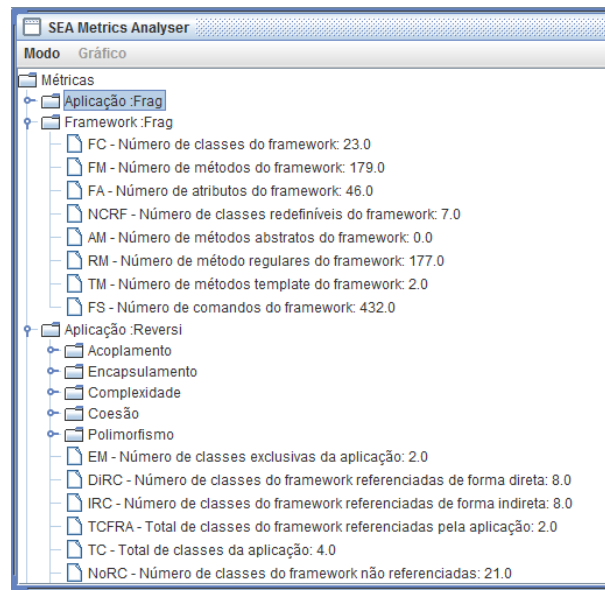


Figura 5.9: Visualização dos dados na ferramenta.

As métricas referentes à *frameworks* orientados a objetos, podem ser visualizadas de forma gráfica, conforme mostrado na figura 5.10, permitindo uma comparação visual entre os dados de diversas aplicações desenvolvidas sob um mesmo framework.

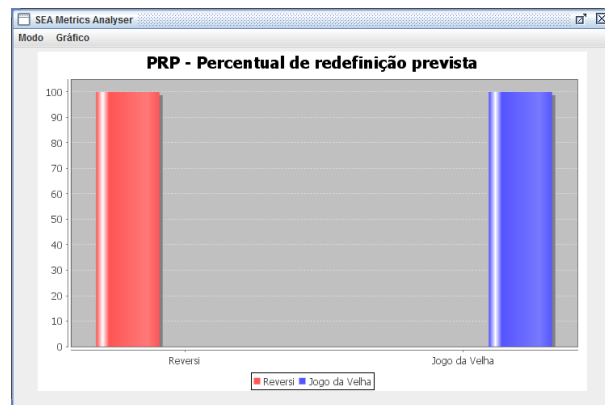


Figura 5.10: Visualização dos dados de forma gráfica.

Dentro do ambiente OCEAN/SEA, uma principal alteração se fez necessária de modo a comportar a ferramenta de geração e análise de métricas criadas. No início do desenvolvimento, notou-se a ausência de uma forma para associar a especificação de uma aplicação aos seus *frameworks* referentes, tornando inválidas as métricas referentes ao uso de *frameworks* orientados a objetos. Este detalhe foi contornado transpondo a janela de especificação pai da implementação original, desenvolvida por Silva (2000), para a implementação atual, a qual já estava preparada internamente para trabalhar com este tipo de especificação.

6 CONCLUSÃO

A reengenharia proposta das ferramentas de extração e análise de métricas voltadas a frameworks e aplicações orientadas a objetos permitiu ao ambiente OCEAN/SEA, mais uma possibilidade para a asserção da qualidade na especificação e desenvolvimento de sistemas orientados a objetos. Acoplando-se a ferramenta às especificações desenvolvidas na linguagem UML 2.0 trouxe esta ao contexto mais atual no que se refere à especificação de sistemas orientados a objetos.

Garantir a qualidade de especificações orientadas a objetos se torna essencial para a produção de sistemas mais robustos, confiáveis e manuteníveis. A utilização de métricas permite ao desenvolvedor focar não apenas na qualidade das funcionalidades desenvolvidas no *software*, mas também na qualidade do artefato produzido, permitindo um mais fácil entendimento do sistema desenvolvido e aumentando o reuso.

6.1 REALIZAÇÕES

O trabalho consistiu basicamente na análise das ferramentas desenvolvidas por Freiburger (2002) e Fonseca (2002), atualizando nestas as definições propostas pelos autores em Freiburger e Silva (2009).

Com base nas definições atualizadas das ferramentas foi necessária uma análise da abordagem da linguagem UML 2.0 definida por Silva (2007), e implementada no ambiente por Vargas (2008), principalmente no que se refere à modelagem de algoritmos de métodos.

Foi possível, com isso, a definição e implementação de uma ferramenta de extração e análise de métricas que, ao mesmo tempo que envolve traços de ambas as ferramentas desenvolvidas por Freiburger (2002) e Fonseca (2002), permite encaixar as métricas definidas ao contexto da implementação atual do ambiente OCEAN/SEA, permitindo a extração de métricas sobre especificações UML 2.0.

A ferramenta obtida ao final do processo de desenvolvimento apresenta as seguintes carac-

terísticas:

- Capacidade de analisar especificações UML 2.0, incluindo dados de algoritmos;
- Capacidade de calcular setenta métricas referentes ao reuso de *frameworks* orientados a objetos;
- Capacidade de calcular trinta e sete métricas referentes aplicações orientadas a objetos;
- Dezoito novas classes desenvolvidas para o ambiente OCEAN/SEA;
- Possibilidade de visualização das métricas de forma gráfica ou hierárquica.
- Capacidade de analisar, simultaneamente, diversas aplicações desenvolvidas sob um mesmo framework.

A validação da ferramenta foi feita com base no *framework* Frag, e as aplicações Jogo da Velha e Reversi, desenvolvidos sobre ele. Este *framework* foi desenvolvido inicialmente por Silva (2000), e convertido para a linguagem Java por (COELHO, 2007) e esta última versão foi utilizada por este trabalho, sendo transformada em uma especificação UML 2.0 a partir da ferramenta de suporte à engenharia reversa do ambiente. Os resultados das métricas obtidos a partir destas especificações está apresentado no Anexo A.

6.2 LIMITAÇÕES

Conforme o escopo proposto, e devido a certas decisões tomadas no decorrer do desenvolvimento da ferramenta, certas limitações se mostram evidentes:

- Assume-se que todos os diagramas de atividades estarão utilizando em suas ações o padrão da sintaxe Java, conforme adotado em Silva (2007), e a ferramenta não é capaz de reconhecer comandos escritos em outro padrão;
- Como as especificações têm por padrão a definição de seu pai, mas não de seus filhos, fica impossível para a ferramenta descobrir, dado um framework, quais aplicações foram desenvolvidas sob este framework. Para contornar este problema foi assumido que todas as aplicações filhas de um framework encontram-se no mesmo diretório do *framework* e por isso a ferramenta não é capaz de utilizar especificações armazenadas em outras partes do disco;

- Devido ao modelo de desenvolvimento adotado para a obtenção das métricas, é necessária a alteração do código-fonte da ferramenta, adicionando-se novos métodos, e modificação de métodos existentes, tornando difícil ao usuário do sistema incluir novas métricas ao conjunto apresentado;
- Do mesmo modo que nas ferramentas originais, não é possível, atualmente, a definição de diferentes versões para um mesmo aplicativo, de forma a obter métricas que comparem a evolução da qualidade ao longo do tempo;
- Apesar de quantificar características de aplicações e *frameworks* orientados a objetos, a ferramenta não apresenta ao usuário uma visão clara da qualidade do sistema, fazendo-se necessária a análise humana detalhada das informações obtidas.

6.3 TRABALHOS FUTUROS

De forma a evoluir a ferramenta desenvolvida, e ir além do escopo proposto neste trabalho, são apresentados alguns projetos possíveis de serem implementados a partir deste trabalho:

- Criação e implementação de uma DSL¹ para a definição das métricas de forma textual, permitindo ao usuário definir novas métricas e estender o conjunto atual. Além da possibilidade de modificar as métricas existentes de forma a se ajustar melhor ao contexto do sistema desenvolvido;
- No escopo deste projeto a ferramenta foi utilizada para a obtenção das métricas de apenas um framework, e duas aplicações desenvolvidas sob este framework. Para uma maior garantia do funcionamento da ferramenta é necessário que diferentes casos de uso sejam produzidos de forma a testar o comportamento da ferramenta ao analisar uma maior variedade de situações;
- Como citado na seção anterior, após a análise das especificações orientadas a objetos, a ferramenta não analisa por si só os resultados obtido de forma a avisar pontualmente ao usuário sobre a falta de qualidade no código. Heurísticas podem ser desenvolvidas para analisar os resultados obtidos e decidir automaticamente sobre a falta de qualidade na especificação, indicando ao desenvolvedor aonde se encontram os problemas e sugerindo possíveis soluções.

¹*Domain-Specific Language: linguagem criada para tratar um domínio específico.*

6.4 CONSIDERAÇÕES FINAIS

A rápida evolução no mercado de desenvolvimento de softwares exige que as ferramentas sejam desenvolvidas cada vez mais rápido, e com mais qualidade. O incremento na qualidade gerado a partir da atualização de suas definições motivou a reengenharia do ambiente OCEAN/SEA, que criou um ambiente moderno e tornou obsoletas as ferramentas desenvolvidas anteriormente sobre ele. A reengenharia destas ferramentas se deu pela mesma razão, e possibilitou a criação de uma ferramenta moderna e mais integrada ao ambiente.

Converter uma especificação orientada a objetos em métricas que a identifiquem é extremamente importante, principalmente em aplicações de maior porte aonde a verificação visual da especificação já não permite mais identificar falhas de qualidade no projeto. Em se tratando de frameworks, o cenário fica ainda mais complicado quando se tem, além de um *framework* de grande porte, várias aplicações igualmente grandes sob desenvolvimento. A utilização de métricas se torna essencial para garantir a qualidade do sistema desenvolvido.

Este trabalho, baseado na necessidade de métricas orientadas a objetos, implementou uma ferramenta para extração e análise de métricas, possibilitando ao desenvolvedor a verificação da qualidade do artefato desenvolvido ainda na fase de projeto do sistema.

REFERÊNCIAS

- AMORIM, J. de. *Integração dos frameworks JHotDraw e OCEAN para a produção de objetos visuais a partir do framework OCEAN*. 2. ed. [S.l.: s.n.], 2006.
- BENLARBI, S.; MELO, W. L. Polymorphism measures for early risk prediction. *International Conference on Software Engineering*, v. 21, p. 334–344, 1999.
- BOOCH, G. *Object-Oriented Analysis and Design*. [S.l.]: ADDISON-WESLEY, 1998.
- COELHO, A. *Reengenharia do framework OCEAN*. [S.l.: s.n.], 2007.
- COPPE. Odyssey-metrics. 2006. Disponível em: <<http://reuse.cos.ufrj.br/odyssey>>.
- FAYAD, M.; SCHMIDT, D. C. Object-oriented application frameworks. *Communications of the ACM*, v. 40, n. 10, p. 3138, oct 1997.
- FONSECA, W. R. da. *Ferramenta de extração de métricas para apoio à avaliação de especificações orientadas a objetos*. Dissertação (Mestrado) — Universidade Federal de Santa Catarina, Florianópolis, dez. 2002.
- FREIBERGER, E. C. *Suporte ao uso de frameworks orientados a objetos com base no histórico do desenvolvimento de aplicações*. Dissertação (Mestrado) — Universidade Federal de Santa Catarina, Florianópolis, dez. 2002.
- FREIBERGER, E. C.; SILVA, R. P. e. Metrics to evaluate the use of object oriented frameworks. *The Computer Journal*, v. 52, n. 3, maio 2009. Disponível em: <<http://comjnl.oxfordjournals.org/cgi/content/abstract/bxm125v2>>.
- JOHNSON, B. F. R. E. Designing reusable classes. *Journal of Object-Oriented Programming*, p. 22–25, 1988.
- MACHADO, T. A. A. S. da R. *Reengenharia da Interface do Ambiente SEA*. [S.l.: s.n.], 2007.
- OMG. Uml 2.0 superstructure specification. 2004. Disponível em: <<http://www.omg.org/cgi-bin/doc?ptc/03-08-02.pdf>>.
- PRESSMAN, R. S. *Engenharia de Software*. 5. ed. [S.l.]: McGraw-Hill, 2006. 720 p.
- ROCHA J. C. MALDONADO, K. C. W. A. R. *Qualidade de Software*. São Paulo: Prentice Hall, 2001.
- SDMETRICS. Sdmetrics tool. 2007. Disponível em: <<http://www.sdmetrics.com>>.
- SILVA, R. P. e. *Suporte ao desenvolvimento de frameworks e componentes*. Tese (Doutorado) — Universidade Federal do Rio Grande do Sul, Porto Alegre, mar. 2000.
- SILVA, R. P. e. *UML2 em Modelagem Orientada a Objetos*. [S.l.]: Visual Books, 2007. 232 p.

TALIGENT, I. Building object-oriented frameworks. 1994.

VARGAS, T. C. de S. *Suporte à Edição de UML 2 no ambiente SEA*. [S.l.: s.n.], 2008.

APÊNDICE A – EXEMPLO DE USO DA FERRAMENTA

Para a verificação da validade da ferramenta desenvolvida, foram obtidas métricas a partir um *framework* orientados a objetos, e duas aplicações desenvolvidas sob este *framework*. O *framework* utilizado é o mesmo utilizado por Freiburger (2002) e Fonseca (2002) para estudo de caso, porém desenvolvido agora em linguagem *Java*. As aplicações utilizadas, também modificadas para este novo contexto, são dois jogos também presentes em (FREIBERGER, 2002), o Jogo da Velha e Reversi.

A.1 ESPECIFICAÇÃO DO FRAMEWORK FRAG

O *framework* Frag foi utilizado por este trabalho para o teste das métricas orientadas a objetos. Segue abaixo o diagrama de classes que representa o framework, dividido em três partes de acordo com suas relações. A figura A.1 apresenta a classe *Position* e suas relações.

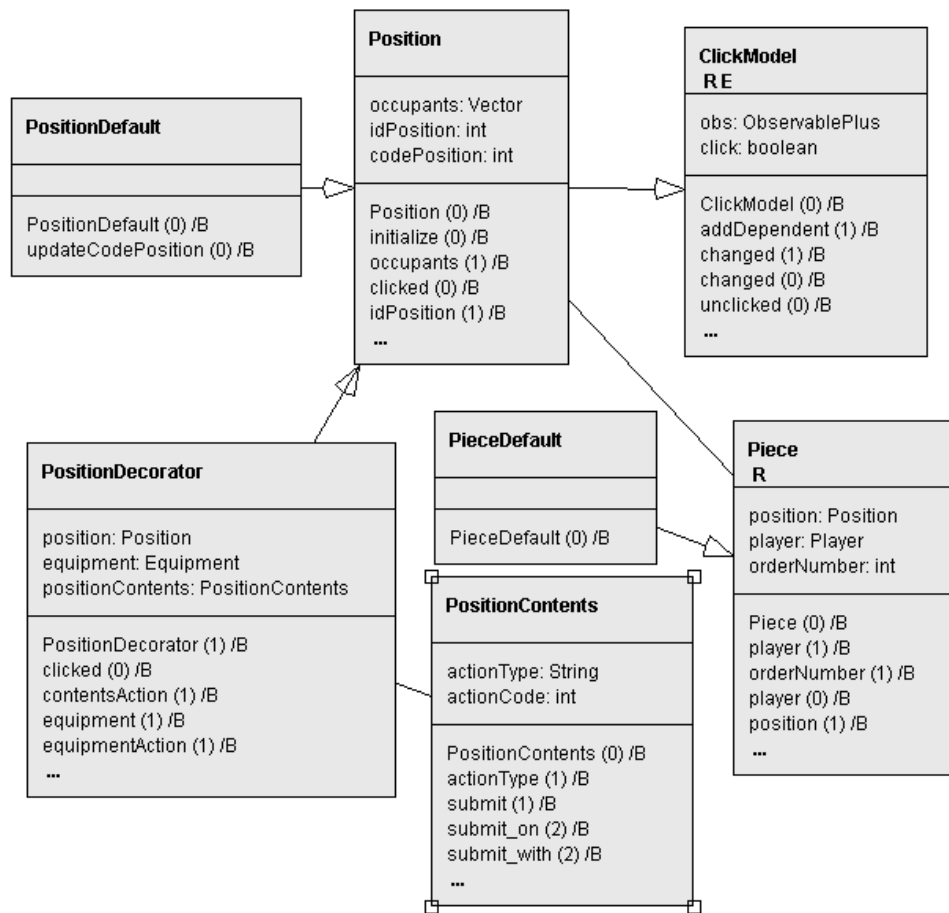


Figura A.1: Classes relacionadas com Position.

Na figura A.2, é apresentada a classe *Dice* e suas relações.

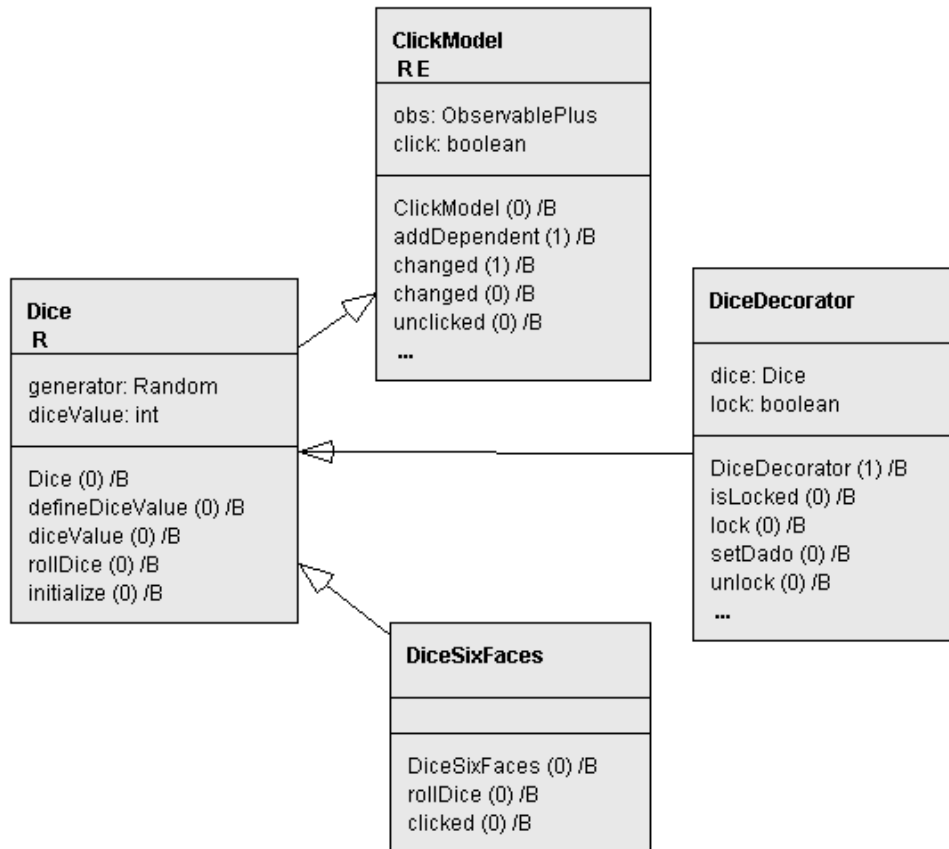


Figura A.2: Classes relacionadas com Dice.

A classe *Player* é apresentada na figura A.3, a seguir, em conjunto com suas classes relacionadas.

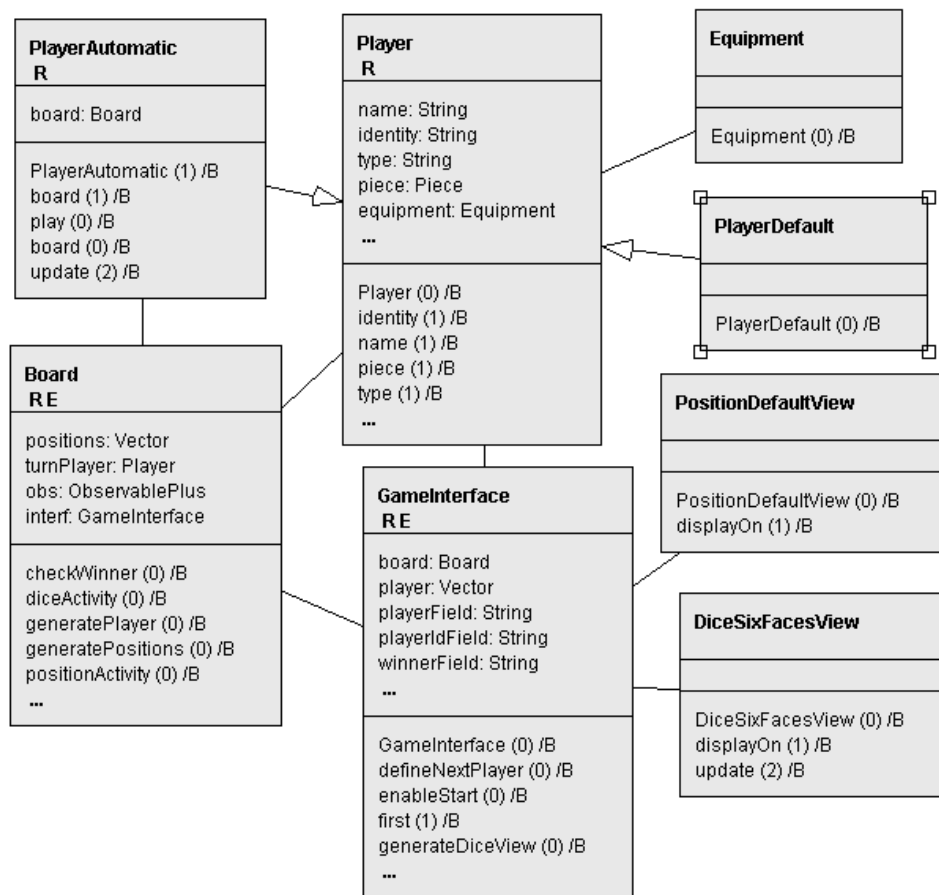


Figura A.3: Classes relacionadas com Player.

A primeira aplicação utilizada para a verificação das métricas, foi uma implementação do Jogo da Velha, em Java, tendo como base o framework Frag. Na figura A.4 são apresentadas suas classes e conexões com o framework.

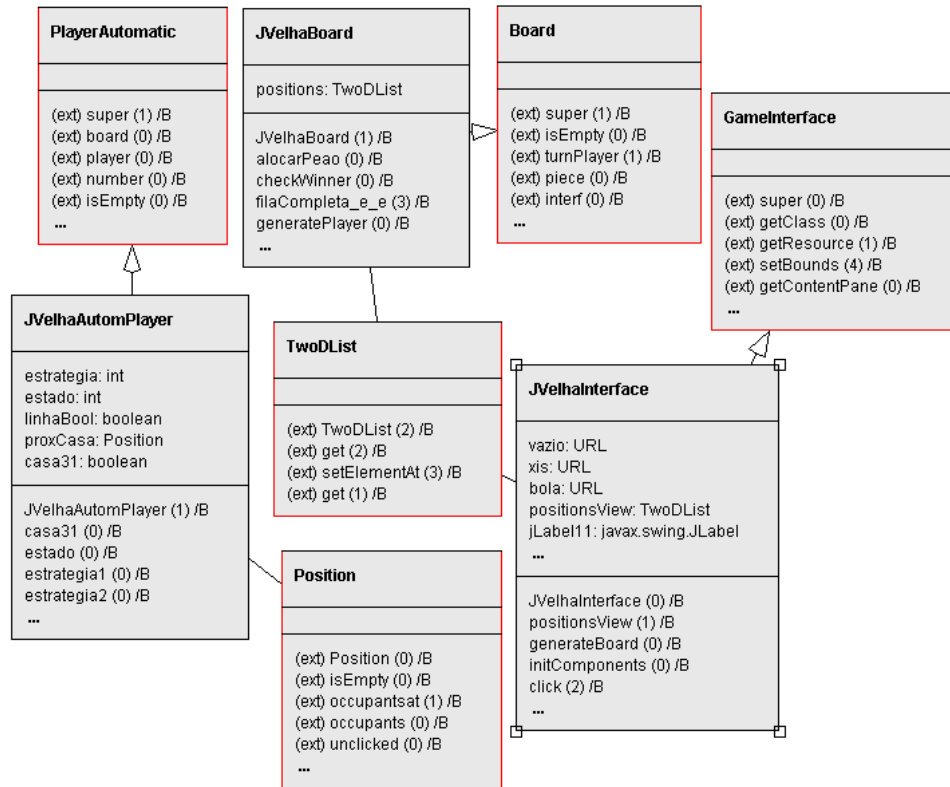


Figura A.4: Classes pertencentes ao Jogo da Velha.

Outra aplicação utilizada nos testes da ferramenta é o Reversi, que tem sua especificação mostrada abaixo, na figura A.5.

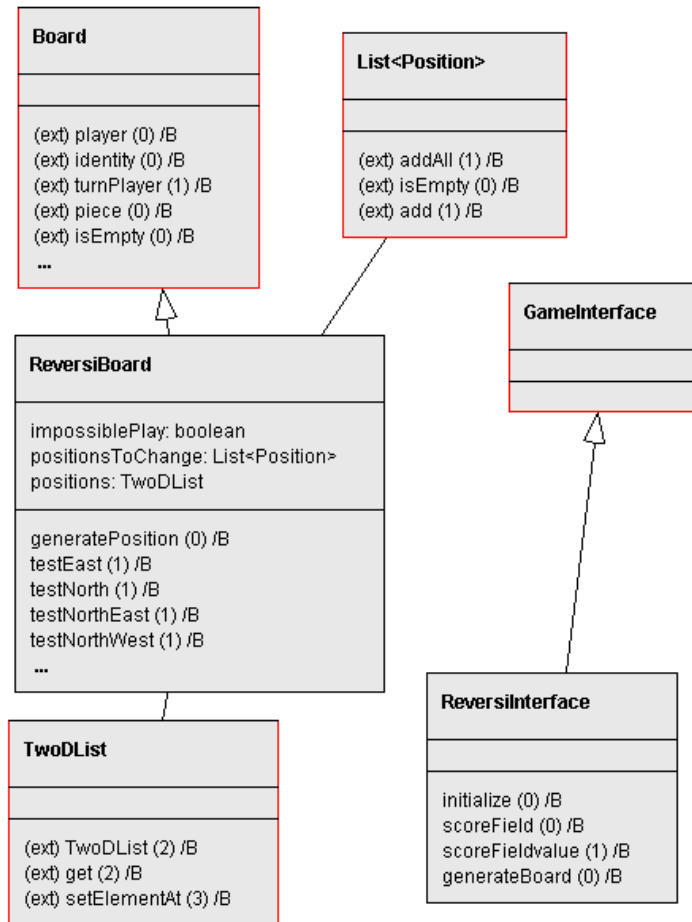


Figura A.5: Classes pertencentes ao Reversi.

A.2 MÉTRICAS REFERENTES AO USO DE FRAMEWORKS ORIENTADOS A OBJETOS

No que concerne a métricas referentes ao uso de frameworks orientados a objetos, foram utilizadas como aplicações de teste uma implementação do Jogo da Velha, e uma implementação do Reversi, desenvolvidas sob o mesmo framework Frag.

Seguem os resultados das métricas aplicadas ao framework como um todo:

- FC - Número de classes do framework: 23.0
- FM - Número de métodos do framework: 179.0
- FA - Número de atributos do framework: 46.0
- NCRF - Número de classes redefiníveis do framework: 7.0
- AM - Número de métodos abstratos do framework: 0.0
- RM - Número de método regulares do framework: 177.0
- TM - Número de métodos template do framework: 2.0
- FS - Número de comandos do framework: 432.0

Estão explicitados abaixo os resultados obtidos na execução das métricas voltadas ao uso de frameworks orientados a objetos e aplicados a cada aplicação individual:

- SHoAM - Número de métodos abstratos hook sobrescritos
 - Reversi: 0.0
 - Jogo da Velha: 0.0
- IRC - Número de classes do framework referenciadas de forma indireta
 - Reversi: 8.0
 - Jogo da Velha: 11.0
- NoRS - Número de comandos do framework não referenciados
 - Reversi: 420.0
 - Jogo da Velha: 414.0
- NoRA - Número de atributos do framework não referenciados
 - Reversi: 38.0
 - Jogo da Velha: 18.0
- TM - Total de métodos da aplicação
 - Reversi: 49.0
 - Jogo da Velha: 88.0
- SReP - Percentual de reuso de comandos
 - Reversi: 4.411764705882353
 - Jogo da Velha: 3.896103896103896
- PMTHS - Percentual de métodos template hook sobrescritos
 - Reversi: 0.0
 - Jogo da Velha: 0.0
- NoRM - Número de métodos do framework não referenciados
 - Reversi: 160.0
 - Jogo da Velha: 147.0
- PMAS - Percentual de métodos abstratos sobrescritos
 - Reversi: 0.0
 - Jogo da Velha: 0.0
- TCFRA - Total de classes do framework referenciadas pela aplicação
 - Reversi: 2.0
 - Jogo da Velha: 2.0
- TCOFR - Total de comandos do framework referenciados pela aplicação
 - Reversi: 12.0
 - Jogo da Velha: 18.0
- CReP - Percentual de reuso de classes
 - Reversi: 50.0
 - Jogo da Velha: 40.0
- PRNP - Percentual de redefinição não prevista
 - Reversi: 0.0
 - Jogo da Velha: 0.0
- PTMS - Percentual de métodos template sobrescritos
 - Reversi: 0.0
 - Jogo da Velha: 0.0
- PMRS - Percentual de métodos regulares sobrescritos
 - Reversi: 100.0
 - Jogo da Velha: 100.0
- PNM - Percentual de novos métodos
 - Reversi: 90.0
 - Jogo da Velha: 89.28571428571429
- TCR - Total de classes redefinidas
 - Reversi: 2.0
 - Jogo da Velha: 3.0
- TMS - Total de métodos sobrescritos
 - Reversi: 0.0
 - Jogo da Velha: 0.0

- Reversi: 3.0
- Jogo da Velha: 6.0
- IRM - Número de métodos do framework referenciados de forma indireta
 - Reversi: 8.0
 - Jogo da Velha: 9.0
- NC - Número de classes novas
 - Reversi: 0.0
 - Jogo da Velha: 0.0
- EM - Número de classes exclusivas da aplicação
 - Reversi: 2.0
 - Jogo da Velha: 3.0
- CPP - Percentual de desenvolvimento de classes
 - Reversi: 50.0
 - Jogo da Velha: 60.0
- FoCS - Número de classes especializadas e previstas pelo projeto do framework
 - Reversi: 2.0
 - Jogo da Velha: 3.0
- SReM - Número de métodos regulares sobrescritos
 - Reversi: 3.0
 - Jogo da Velha: 6.0
- TA - Total de atributos da aplicação
 - Reversi: 8.0
 - Jogo da Velha: 28.0
- SM - Número de métodos especializados
 - Reversi: 3.0
 - Jogo da Velha: 6.0
- STeM - Número de métodos template sobrescritos
 - Reversi: 0.0
 - Jogo da Velha: 0.0
- PMRHS - Percentual de métodos regulares hook sobrescritos
 - Reversi: 0.0
 - Jogo da Velha: 0.0
- MPP - Percentual de desenvolvimento de métodos
 - Reversi: 61.224489795918366
 - Jogo da Velha: 63.63636363636363
- EA - Número de atributos exclusivos da aplicação
 - Reversi: 3.0
 - Jogo da Velha: 22.0
- PRP - Percentual de redefinição prevista
 - Reversi: 100.0
 - Jogo da Velha: 100.0
- SPP - Percentual de desenvolvimento de comandos
 - Reversi: 95.58823529411765
 - Jogo da Velha: 96.1038961038961
- SP - Número de classes especializadas
 - Reversi: 2.0
 - Jogo da Velha: 3.0
- SHoTeM - Número de métodos template hook sobrescritos
 - Reversi: 0.0
 - Jogo da Velha: 0.0
- SAM - Número de métodos abstratos sobrescritos
 - Reversi: 0.0
 - Jogo da Velha: 0.0
- DiRM - Número de métodos do framework referenciados de forma direta
 - Reversi: 12.0

- Jogo da Velha: 24.0
- NM - Número de métodos novos
 - Reversi: 27.0
 - Jogo da Velha: 50.0
- PEC - Percentual de especialização de classes
 - Reversi: 100.0
 - Jogo da Velha: 100.0
- DiRC - Número de classes do framework referenciadas de forma direta
 - Reversi: 8.0
 - Jogo da Velha: 11.0
- MeRP - Percentual de reuso de métodos
 - Reversi: 38.775510204081634
 - Jogo da Velha: 36.36363636363637
- PNC - Percentual de novas classes
 - Reversi: 0.0
 - Jogo da Velha: 0.0
- NoRC - Número de classes do framework não referenciadas
 - Reversi: 21.0
 - Jogo da Velha: 21.0
- APP - Percentual de desenvolvimento de atributos
 - Reversi: 37.5
 - Jogo da Velha: 78.57142857142857
- SS - Número de comandos especializados
 - Reversi: 11.0
 - Jogo da Velha: 98.0
- SHoRM - Número de métodos regulares hook sobrescritos
 - Reversi: 0.0
 - Jogo da Velha: 0.0
- IRS - Número de comandos do framework referenciados de forma indireta
 - Reversi: 7.0
 - Jogo da Velha: 10.0
- TS - Total de atributos da aplicação
 - Reversi: 272.0
 - Jogo da Velha: 462.0
- TAFRA - Total de atributos do framework referenciados pela aplicação
 - Reversi: 5.0
 - Jogo da Velha: 6.0
- EM - Número de métodos exclusivos da aplicação
 - Reversi: 30.0
 - Jogo da Velha: 56.0
- DiRA - Número de atributos do framework referenciados de forma direta
 - Reversi: 1.0
 - Jogo da Velha: 2.0
- NS - Número de comandos novos
 - Reversi: 249.0
 - Jogo da Velha: 346.0
- ARP - Percentual de reuso de métodos
 - Reversi: 62.5
 - Jogo da Velha: 21.428571428571427
- PMAHS - Percentual de métodos abstratos hook sobrescritos
 - Reversi: 0.0
 - Jogo da Velha: 0.0
- IRA - Número de Atributos do framework referenciados de forma indireta
 - Reversi: 4.0
 - Jogo da Velha: 4.0
- PEM - Percentual de especialização de métodos

- Reversi: 10.0
- Jogo da Velha: 10.714285714285714
- TMFRA - Total de métodos do framework referenciados pela aplicação
 - Reversi: 19.0
 - Jogo da Velha: 32.0
- PNCO - Percentual de novos comandos
 - Reversi: 95.76923076923077
 - Jogo da Velha: 77.92792792792793
- ES - Número de comandos exclusivos da aplicação
 - Reversi: 260.0
 - Jogo da Velha: 444.0
- PECO - Percentual de especialização de comandos
 - Reversi: 4.230769230769231
 - Jogo da Velha: 22.07207207207207
- UnCS - Número de classes especializadas e não previstas pelo projeto do framework
 - Reversi: 0.0
 - Jogo da Velha: 0.0
- DiRS - Número de comandos do framework referenciados de forma direta
 - Reversi: 7.0
 - Jogo da Velha: 10.0
- TC - Total de classes da aplicação
 - Reversi: 4.0
 - Jogo da Velha: 5.0

A.3 MÉTRICAS SOBRE APLICAÇÕES ORIENTADAS A OBJETOS

No que se refere a aplicações orientadas a objetos, foi utilizado como estudo de caso o *framework* frag, sem considerar suas relações com aplicações desenvolvidas sobre ele.

A.3.1 ACOPLAMENTO

- CF - Fator de Acoplamento: 0.065
- CBO - Acoplamento entre classes de objetos
 - Classe: ClickModel: 1.0
 - Classe: DiceSixFacesView: 3.0
 - Classe: FraGDefaultImageRepository: 1.0
 - Classe: Piece: 3.0
 - Classe: PieceDefault: 1.0
 - Classe: Player: 3.0
 - Classe: PlayerDefault: 1.0
 - Classe: Position: 2.0
 - Classe: PositionDefault: 3.0
 - Classe: PositionDefaultView: 1.0
 - Classe: ObservablePlus: 1.0
 - Classe: TwoDList: 1.0
 - Classe: Account: 1.0
 - Classe: Board: 7.0
 - Classe: ClickController: 1.0
 - Classe: Equipment: 1.0
 - Classe: PlayerAutomatic: 4.0
 - Classe: PositionContents: 4.0
 - Classe: PositionDecorator: 6.0
 - Classe: Dice: 1.0
 - Classe: DiceDecorator: 2.0
 - Classe: DiceSixFaces: 1.0
 - Classe: GameInterface: 7.0
- ACAIC - Acoplamento por Classe-Atributo por importação nos ancestrais

- Classe: ClickModel: 0.0
- Classe: DiceSixFacesView: 0.0
- Classe: FraGDefaultImageRepository: 0.0
- Classe: Piece: 0.0
- Classe: PieceDefault: 0.0
- Classe: Player: 0.0
- Classe: PlayerDefault: 0.0
- Classe: Position: 0.0
- Classe: PositionDefault: 0.0
- Classe: PositionDefaultView: 0.0
- Classe: ObservablePlus: 0.0
- Classe: TwoDList: 0.0
- Classe: Account: 0.0
- Classe: Board: 0.0
- Classe: ClickController: 0.0
- Classe: Equipment: 0.0
- Classe: PlayerAutomatic: 0.0
- Classe: PositionContents: 1.0
- Classe: PositionDecorator: 1.0
- Classe: Dice: 0.0
- Classe: DiceDecorator: 1.0
- Classe: DiceSixFaces: 0.0
- Classe: GameInterface: 0.0

●DCAEC - Acoplamento por Classe-Atributo por exportação nos descendentes

- Classe: ClickModel: 0.0
- Classe: DiceSixFacesView: 0.0
- Classe: FraGDefaultImageRepository: 0.0
- Classe: Piece: 0.0
- Classe: PieceDefault: 0.0
- Classe: Player: 0.0
- Classe: PlayerDefault: 0.0
- Classe: Position: 1.0
- Classe: PositionDefault: 0.0

- Classe: PositionDefaultView: 0.0
- Classe: ObservablePlus: 0.0
- Classe: TwoDList: 0.0
- Classe: Account: 0.0
- Classe: Board: 0.0
- Classe: ClickController: 0.0
- Classe: Equipment: 0.0
- Classe: PlayerAutomatic: 0.0
- Classe: PositionContents: 0.0
- Classe: PositionDecorator: 0.0
- Classe: Dice: 1.0
- Classe: DiceDecorator: 0.0
- Classe: DiceSixFaces: 0.0
- Classe: GameInterface: 0.0

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

- Classe: ClickModel: 1.0
- Classe: DiceSixFacesView: 0.0
- Classe: FraGDefaultImageRepository: 0.0
- Classe: Piece: 2.0
- Classe: PieceDefault: 0.0
- Classe: Player: 2.0
- Classe: PlayerDefault: 0.0
- Classe: Position: 0.0
- Classe: PositionDefault: 0.0
- Classe: PositionDefaultView: 0.0
- Classe: ObservablePlus: 0.0
- Classe: TwoDList: 0.0
- Classe: Account: 0.0
- Classe: Board: 3.0
- Classe: ClickController: 0.0
- Classe: Equipment: 0.0
- Classe: PlayerAutomatic: 1.0
- Classe: PositionContents: 0.0
- Classe: PositionDecorator: 2.0
- Classe: Dice: 0.0
- Classe: DiceDecorator: 0.0

- Classe: DiceSixFaces: 0.0
- Classe: GameInterface: 5.0
- OCAEC - Acoplamento por Classe-Atributo por exportação entre classes sem relação de herança
 - Classe: ClickModel: 0.0
 - Classe: DiceSixFacesView: 0.0
 - Classe: FraGDefaultImageRepository: 0.0
 - Classe: Piece: 0.0
 - Classe: PieceDefault: 0.0
 - Classe: Player: 0.0
 - Classe: PlayerDefault: 0.0
 - Classe: Position: 1.0
 - Classe: PositionDefault: 0.0
 - Classe: PositionDefaultView: 0.0
 - Classe: ObservablePlus: 0.0
 - Classe: TwoDList: 0.0
 - Classe: Account: 0.0
 - Classe: Board: 0.0
 - Classe: ClickController: 0.0
 - Classe: Equipment: 0.0
 - Classe: PlayerAutomatic: 0.0
 - Classe: PositionContents: 0.0
 - Classe: PositionDecorator: 2.0
 - Classe: Dice: 0.0
 - Classe: DiceDecorator: 2.0
 - Classe: DiceSixFaces: 0.0
 - Classe: GameInterface: 0.0
- DCMEC - Acoplamento Classe-Método por exportação nos descendentes
 - Classe: ClickModel: 0.0
 - Classe: DiceSixFacesView: 0.0
 - Classe: FraGDefaultImageRepository: 0.0
 - Classe: Piece: 0.0
 - Classe: PieceDefault: 0.0
 - Classe: Player: 0.0
 - Classe: PlayerDefault: 0.0
 - Classe: Position: 2.0
 - Classe: PositionDefault: 0.0
 - Classe: PositionDefaultView: 0.0
 - Classe: ObservablePlus: 0.0
 - Classe: TwoDList: 0.0
 - Classe: Account: 0.0
 - Classe: Board: 0.0
 - Classe: ClickController: 0.0
 - Classe: Equipment: 0.0
 - Classe: PlayerAutomatic: 0.0
- ACMIC - Acoplamento Classe-Método por importação nos ancestrais
 - Classe: ClickModel: 0.0
 - Classe: DiceSixFacesView: 0.0
 - Classe: FraGDefaultImageRepository: 0.0
 - Classe: Piece: 0.0
 - Classe: PieceDefault: 0.0

- Classe: PositionContents: 0.0
- Classe: PositionDecorator: 0.0
- Classe: Dice: 2.0
- Classe: DiceDecorator: 0.0
- Classe: DiceSixFaces: 0.0
- Classe: GameInterface: 0.0
- OCMIC - Acoplamento Classe-Método por importação entre classes sem relação de herança
 - Classe: ClickModel: 0.0
 - Classe: DiceSixFacesView: 0.0
 - Classe: FraGDefaultImageRepository: 0.0
 - Classe: Piece: 2.0
 - Classe: PieceDefault: 0.0
 - Classe: Player: 2.0
 - Classe: PlayerDefault: 0.0
 - Classe: Position: 2.0
 - Classe: PositionDefault: 0.0
 - Classe: PositionDefaultView: 0.0
 - Classe: ObservablePlus: 0.0
 - Classe: TwoDList: 0.0
 - Classe: Account: 0.0
 - Classe: Board: 3.0
 - Classe: ClickController: 0.0
 - Classe: Equipment: 0.0
 - Classe: PlayerAutomatic: 2.0
 - Classe: PositionContents: 8.0
 - Classe: PositionDecorator: 6.0
 - Classe: Dice: 0.0
 - Classe: DiceDecorator: 0.0
 - Classe: DiceSixFaces: 0.0
 - Classe: GameInterface: 7.0
- OCMEC - Acoplamento Classe-Método por exportação entre classes sem relação de herança
 - Classe: ClickModel: 0.0
- Classe: DiceSixFacesView: 2.0
- Classe: FraGDefaultImageRepository: 0.0
- Classe: Piece: 12.0
- Classe: PieceDefault: 0.0
- Classe: Player: 4.0
- Classe: PlayerDefault: 0.0
- Classe: Position: 1.0
- Classe: PositionDefault: 0.0
- Classe: PositionDefaultView: 0.0
- Classe: ObservablePlus: 0.0
- Classe: TwoDList: 0.0
- Classe: Account: 0.0
- Classe: Board: 5.0
- Classe: ClickController: 0.0
- Classe: Equipment: 4.0
- Classe: PlayerAutomatic: 0.0
- Classe: PositionContents: 1.0
- Classe: PositionDecorator: 0.0
- Classe: Dice: 1.0
- Classe: DiceDecorator: 1.0
- Classe: DiceSixFaces: 0.0
- Classe: GameInterface: 1.0
- AMMIC - Acoplamento Método-Método por importação nos ancestrais
 - Classe: ClickModel: 0.0
 - Classe: DiceSixFacesView: 0.0
 - Classe: FraGDefaultImageRepository: 0.0
 - Classe: Piece: 0.0
 - Classe: PieceDefault: 1.0
 - Classe: Player: 0.0
 - Classe: PlayerDefault: 1.0
 - Classe: Position: 3.0
 - Classe: PositionDefault: 5.0
 - Classe: PositionDefaultView: 0.0
 - Classe: ObservablePlus: 0.0
 - Classe: TwoDList: 1.0

- Classe: Account: 0.0
- Classe: Board: 0.0
- Classe: ClickController: 0.0
- Classe: Equipment: 0.0
- Classe: PlayerAutomatic: 1.0
- Classe: PositionContents: 0.0
- Classe: PositionDecorator: 13.0
- Classe: Dice: 2.0
- Classe: DiceDecorator: 3.0
- Classe: DiceSixFaces: 2.0
- Classe: GameInterface: 1.0
- DMMEC - Acoplamento Método-Método por exportação nos descendentes
 - Classe: ClickModel: 11.0
 - Classe: DiceSixFacesView: 0.0
 - Classe: FraGDefaultImageRepository: 0.0
 - Classe: Piece: 1.0
 - Classe: PieceDefault: 0.0
 - Classe: Player: 2.0
 - Classe: PlayerDefault: 0.0
 - Classe: Position: 13.0
 - Classe: PositionDefault: 0.0
 - Classe: PositionDefaultView: 0.0
 - Classe: ObservablePlus: 0.0
 - Classe: TwoDList: 0.0
 - Classe: Account: 0.0
 - Classe: Board: 0.0
 - Classe: ClickController: 0.0
 - Classe: Equipment: 0.0
 - Classe: PlayerAutomatic: 0.0
 - Classe: PositionContents: 0.0
 - Classe: PositionDecorator: 0.0
 - Classe: Dice: 4.0
 - Classe: DiceDecorator: 0.0
 - Classe: DiceSixFaces: 0.0
- Classe: GameInterface: 0.0
- OMMIC - Acoplamento Método-Método por importação entre classes sem relação de herança
 - Classe: ClickModel: 3.0
 - Classe: DiceSixFacesView: 0.0
 - Classe: FraGDefaultImageRepository: 0.0
 - Classe: Piece: 0.0
 - Classe: PieceDefault: 0.0
 - Classe: Player: 1.0
 - Classe: PlayerDefault: 0.0
 - Classe: Position: 1.0
 - Classe: PositionDefault: 2.0
 - Classe: PositionDefaultView: 0.0
 - Classe: ObservablePlus: 1.0
 - Classe: TwoDList: 12.0
 - Classe: Account: 1.0
 - Classe: Board: 22.0
 - Classe: ClickController: 0.0
 - Classe: Equipment: 0.0
 - Classe: PlayerAutomatic: 2.0
 - Classe: PositionContents: 0.0
 - Classe: PositionDecorator: 7.0
 - Classe: Dice: 2.0
 - Classe: DiceDecorator: 4.0
 - Classe: DiceSixFaces: 0.0
 - Classe: GameInterface: 48.0
- OMMEC - Acoplamento Método-Método por exportação entre classes sem relação de herança
 - Classe: ClickModel: 2.0
 - Classe: DiceSixFacesView: 2.0
 - Classe: FraGDefaultImageRepository: 0.0
 - Classe: Piece: 3.0
 - Classe: PieceDefault: 2.0
 - Classe: Player: 19.0
 - Classe: PlayerDefault: 2.0

- Classe: Position: 1.0
- Classe: PositionDefault: 0.0
- Classe: PositionDefaultView: 1.0
- Classe: ObservablePlus: 4.0
- Classe: TwoDList: 12.0
- Classe: Account: 1.0
- Classe: Board: 4.0
- Classe: ClickController: 0.0
- Classe: Equipment: 0.0
- Classe: PlayerAutomatic: 1.0
- Classe: PositionContents: 0.0
- Classe: PositionDecorator: 7.0
- Classe: Dice: 2.0
- Classe: DiceDecorator: 9.0
- Classe: DiceSixFaces: 2.0
- Classe: GameInterface: 32.0

A.3.2 ENCAPSULAMENTO

- AHF - Fator de Atributos Ocultos: 1.0
- MHF - Fator de métodos ocultos: 0.0

A.3.3 COMPLEXIDADE

- SOM - Tamanho dos Métodos nas Classes
 - Classe: ClickModel
 - *Método: ClickModel: 1.0
 - *Método: addDependent: 1.0
 - *Método: changed: 1.0
 - *Método: changed: 1.0
 - *Método: unclicked: 2.0
 - *Método: click: 1.0
 - *Método: clicked: 3.0
 - Classe: DiceSixFacesView
 - *Método: DiceSixFacesView: 0.0
 - *Método: displayOn: 2.0
 - *Método: update: 0.0
 - Classe: FraGDefaultImageRepository
 - *Método: FraGDefaultImageRepository: 0.0
 - Classe: Piece
 - *Método: Piece: 0.0
 - *Método: player: 2.0
 - *Método: orderNumber: 2.0
 - *Método: player: 1.0
 - *Método: position: 2.0
- *Método: orderNumber: 1.0
- *Método: position: 1.0
- Classe: PieceDefault
 - *Método: PieceDefault: 1.0
- Classe: Player
 - *Método: Player: 1.0
 - *Método: identity: 2.0
 - *Método: name: 2.0
 - *Método: piece: 2.0
 - *Método: type: 2.0
 - *Método: equipment: 2.0
 - *Método: extraMoves: 2.0
 - *Método: identity: 1.0
 - *Método: name: 1.0
 - *Método: number: 2.0
 - *Método: piece: 1.0
 - *Método: type: 1.0
 - *Método: equipment: 1.0
 - *Método: extraMoves: 1.0
 - *Método: number: 1.0
- Classe: PlayerDefault
 - *Método: PlayerDefault: 1.0
- Classe: Position
 - *Método: Position: 2.0
 - *Método: initialize: 3.0
 - *Método: occupants: 2.0

- *Método: clicked: 3.0
- *Método: idPosition: 2.0
- *Método: occupants: 1.0
- *Método: remove: 3.0
- *Método: occupantsat: 1.0
- *Método: codePosition: 3.0
- *Método: idPosition: 1.0
- *Método: isEmpty: 2.0
- *Método: codePosition: 1.0
- *Método: add: 5.0
- Classe: PositionDefault
 - *Método: PositionDefault: 1.0
 - *Método: updateCodePosition: 11.0
- Classe: PositionDefaultView
 - *Método: PositionDefault-View: 0.0
 - *Método: displayOn: 2.0
- Classe: ObservablePlus
 - *Método: ObservablePlus: 0.0
 - *Método: notifyObservers: 2.0
 - *Método: setChanged: 1.0
- Classe: TwoDList
 - *Método: TwoDList: 3.0
 - *Método: getIndex: 3.0
 - *Método: add: 0.0
 - *Método: get: 4.0
 - *Método: setElementAt: 0.0
 - *Método: setElementAt: 0.0
 - *Método: main: 13.0
 - *Método: TwoDList: 1.0
 - *Método: add: 2.0
 - *Método: get: 3.0
- Classe: Account
 - *Método: Account: 1.0
 - *Método: balance: 1.0
 - *Método: deposit: 2.0
 - *Método: draft: 2.0
 - *Método: draftAll: 1.0
 - *Método: initialize: 2.0
- Classe: Board
 - *Método: checkWinner: 2.0
 - *Método: diceActivity: 2.0
 - *Método: generatePlayer: 20.0
 - *Método: generatePositions: 2.0
 - *Método: positionActivity: 2.0
 - *Método: positions: 1.0
 - *Método: Board: 0.0
 - *Método: generateDice: 7.0
 - *Método: interf: 2.0
 - *Método: addDependent: 1.0
 - *Método: interf: 1.0
 - *Método: turnPlayer: 5.0
 - *Método: turnPlayer: 1.0
 - *Método: update: 0.0
- Classe: ClickController
 - *Método: ClickController: 0.0
- Classe: Equipment
 - *Método: Equipment: 0.0
- Classe: PlayerAutomatic
 - *Método: PlayerAutomatic: 4.0
 - *Método: board: 2.0
 - *Método: play: 2.0
 - *Método: board: 1.0
 - *Método: update: 0.0
- Classe: PositionContents
 - *Método: PositionContents: 0.0
 - *Método: actionType: 2.0
 - *Método: submit: 2.0
 - *Método: submit_on: 2.0
 - *Método: submit_with: 2.0
 - *Método: actionCode: 2.0
 - *Método: actionType: 1.0
 - *Método: submit_at: 2.0
 - *Método: submit_at_with: 0.0
 - *Método: actionCode: 1.0
- Classe: PositionDecorator
 - *Método: PositionDecorator: 2.0
 - *Método: clicked: 4.0

- *Método: contentsAction: 2.0
- *Método: equipment: 2.0
- *Método: equipmentAction: 2.0
- *Método: occupants: 2.0
- *Método: position: 2.0
- *Método: positionContents: 2.0
- *Método: remove: 5.0
- *Método: unclicked: 3.0
- *Método: updateCodePosition: 3.0
- *Método: click: 2.0
- *Método: codePosition: 4.0
- *Método: equipment: 1.0
- *Método: isEmpty: 2.0
- *Método: position: 1.0
- *Método: positionContents: 1.0
- *Método: codePosition: 2.0
- *Método: add: 13.0

–Classe: Dice

- *Método: Dice: 2.0
- *Método: defineDiceValue: 4.0
- *Método: diceValue: 1.0
- *Método: rollDice: 1.0
- *Método: initialize: 3.0

–Classe: DiceDecorator

- *Método: DiceDecorator: 3.0
- *Método: isLocked: 1.0
- *Método: lock: 2.0
- *Método: setDado: 2.0
- *Método: unlock: 2.0
- *Método: defineDiceValue: 6.0
- *Método: dice: 2.0
- *Método: diceValue: 1.0
- *Método: clicked: 2.0

–Classe: DiceSixFaces

- *Método: DiceSixFaces: 1.0
- *Método: rollDice: 12.0
- *Método: clicked: 2.0

–Classe: GameInterface

- *Método: GameInterface: 2.0
- *Método: defineNextPlayer: 7.0
- *Método: enableStart: 8.0
- *Método: first: 7.0
- *Método: generateDiceView: 2.0
- *Método: generatePositionView: 2.0
- *Método: playerField: 3.0
- *Método: playerIdField: 3.0
- *Método: playerFieldvalue: 1.0
- *Método: playerIdFieldvalue: 1.0
- *Método: playersOrder: 25.0
- *Método: setWinner: 5.0
- *Método: winnerField: 3.0
- *Método: winnerFieldvalue: 1.0
- *Método: auxDice: 2.0
- *Método: auxDiceView: 2.0
- *Método: board: 2.0
- *Método: dice: 2.0
- *Método: diceView: 2.0
- *Método: eliminatePlayer: 6.0
- *Método: finished: 2.0
- *Método: generateBoard: 4.0
- *Método: initialize: 7.0
- *Método: initializeInterfaceAttributes: 12.0
- *Método: numbPlayers: 2.0
- *Método: player: 2.0
- *Método: positionsView: 2.0
- *Método: turnPlayer: 2.0
- *Método: winnerExists: 2.0
- *Método: auxDice: 1.0
- *Método: auxDiceView: 1.0
- *Método: board: 1.0
- *Método: dice: 1.0
- *Método: diceView: 1.0
- *Método: finished: 1.0

- *Método: numbPlayers: 1.0
 - *Método: player: 1.0
 - *Método: positionsView: 1.0
 - *Método: turnPlayer: 1.0
 - *Método: winnerExists: 1.0
 - *Método: update: 0.0
- NAM - Número de Argumentos no Método
- Classe: ClickModel
 - *Método: ClickModel: 0.0
 - *Método: addDependent: 1.0
 - *Método: changed: 1.0
 - *Método: changed: 0.0
 - *Método: unclicked: 0.0
 - *Método: click: 0.0
 - *Método: clicked: 0.0
 - Classe: DiceSixFacesView
 - *Método: DiceSixFacesView: 0.0
 - *Método: displayOn: 1.0
 - *Método: update: 2.0
 - Classe: FraGDefaultImageRepository
 - *Método: FraGDefaultImageRepository: 0.0
 - Classe: Piece
 - *Método: Piece: 0.0
 - *Método: player: 1.0
 - *Método: orderNumber: 1.0
 - *Método: player: 0.0
 - *Método: position: 1.0
 - *Método: orderNumber: 0.0
 - *Método: position: 0.0
 - Classe: PieceDefault
 - *Método: PieceDefault: 0.0
 - Classe: Player
 - *Método: Player: 0.0
 - *Método: identity: 1.0
 - *Método: name: 1.0
 - *Método: piece: 1.0
 - *Método: type: 1.0
 - *Método: equipment: 1.0
 - *Método: extraMoves: 1.0
 - *Método: identity: 0.0
 - *Método: name: 0.0
 - *Método: number: 1.0
 - *Método: piece: 0.0
 - *Método: type: 0.0
 - *Método: equipment: 0.0
 - *Método: extraMoves: 0.0
 - *Método: number: 0.0
- Classe: PlayerDefault
 - *Método: PlayerDefault: 0.0
 - Classe: Position
 - *Método: Position: 0.0
 - *Método: initialize: 0.0
 - *Método: occupants: 1.0
 - *Método: clicked: 0.0
 - *Método: idPosition: 1.0
 - *Método: occupants: 0.0
 - *Método: remove: 1.0
 - *Método: occupantsat: 1.0
 - *Método: codePosition: 1.0
 - *Método: idPosition: 0.0
 - *Método: isEmpty: 0.0
 - *Método: codePosition: 0.0
 - *Método: add: 1.0
 - Classe: PositionDefault
 - *Método: PositionDefault: 0.0
 - *Método: updateCodePosition: 0.0
 - Classe: PositionDefaultView
 - *Método: PositionDefaultView: 0.0
 - *Método: displayOn: 1.0
 - Classe: ObservablePlus
 - *Método: ObservablePlus: 0.0
 - *Método: notifyObservers: 1.0
 - *Método: setChanged: 0.0
 - Classe: TwoDList
 - *Método: TwoDList: 3.0
 - *Método: getIndex: 1.0

*Método: add: 3.0
 *Método: get: 2.0
 *Método: setElementAt: 3.0
 *Método: setElementAt: 2.0
 *Método: main: 1.0
 *Método: TwoDList: 2.0
 *Método: add: 2.0
 *Método: get: 1.0

–Classe: Account

*Método: Account: 0.0
 *Método: balance: 0.0
 *Método: deposit: 1.0
 *Método: draft: 1.0
 *Método: draftAll: 0.0
 *Método: initialize: 0.0

–Classe: Board

*Método: checkWinner: 0.0
 *Método: diceActivity: 0.0
 *Método: generatePlayer: 0.0
 *Método: generatePositions: 0.0
 *Método: positionActivity: 0.0
 *Método: positions: 0.0
 *Método: Board: 0.0
 *Método: generateDice: 0.0
 *Método: interf: 1.0
 *Método: addDependent: 1.0
 *Método: interf: 0.0
 *Método: turnPlayer: 1.0
 *Método: turnPlayer: 0.0
 *Método: update: 2.0

–Classe: ClickController

*Método: ClickController: 0.0

–Classe: Equipment

*Método: Equipment: 0.0

–Classe: PlayerAutomatic

*Método: PlayerAutomatic: 1.0
 *Método: board: 1.0
 *Método: play: 0.0

*Método: board: 0.0
 *Método: update: 2.0

–Classe: PositionContents

*Método: PositionContents: 0.0
 *Método: actionType: 1.0
 *Método: submit: 1.0
 *Método: submit_on: 2.0
 *Método: submit_with: 2.0
 *Método: actionCode: 1.0
 *Método: actionType: 0.0
 *Método: submit_at: 2.0
 *Método: submit_at_with: 3.0
 *Método: actionCode: 0.0

–Classe: PositionDecorator

*Método: PositionDecorator: 1.0
 *Método: clicked: 0.0
 *Método: contentsAction: 1.0
 *Método: equipment: 1.0
 *Método: equipmentAction: 1.0
 *Método: occupants: 0.0
 *Método: position: 1.0
 *Método: positionContents: 1.0
 *Método: remove: 1.0
 *Método: unclicked: 0.0
 *Método: updateCodePosition: 0.0
 *Método: click: 0.0
 *Método: codePosition: 1.0
 *Método: equipment: 0.0
 *Método: isEmpty: 0.0
 *Método: position: 0.0
 *Método: positionContents: 0.0
 *Método: codePosition: 0.0
 *Método: add: 1.0

–Classe: Dice

*Método: Dice: 0.0
 *Método: defineDiceValue: 0.0
 *Método: diceValue: 0.0
 *Método: rollDice: 0.0

- *Método: initialize: 0.0
 - Classe: DiceDecorator
 - *Método: DiceDecorator: 1.0
 - *Método: isLocked: 0.0
 - *Método: lock: 0.0
 - *Método: setDado: 0.0
 - *Método: unlock: 0.0
 - *Método: defineDiceValue: 0.0
 - *Método: dice: 1.0
 - *Método: diceValue: 0.0
 - *Método: clicked: 0.0
 - Classe: DiceSixFaces
 - *Método: DiceSixFaces: 0.0
 - *Método: rollDice: 0.0
 - *Método: clicked: 0.0
 - Classe: GameInterface
 - *Método: GameInterface: 0.0
 - *Método: defineNextPlayer: 0.0
 - *Método: enableStart: 0.0
 - *Método: first: 1.0
 - *Método: generateDiceView: 0.0
 - *Método: generatePositionView: 0.0
 - *Método: playerField: 0.0
 - *Método: playerIdField: 0.0
 - *Método: playerFieldvalue: 1.0
 - *Método: playerIdFieldvalue: 1.0
 - *Método: playersOrder: 1.0
 - *Método: setWinner: 0.0
 - *Método: winnerField: 0.0
 - *Método: winnerFieldvalue: 1.0
 - *Método: auxDice: 1.0
 - *Método: auxDiceView: 1.0
 - *Método: board: 1.0
 - *Método: dice: 1.0
 - *Método: diceView: 1.0
 - *Método: eliminatePlayer: 1.0
 - *Método: finished: 1.0
 - *Método: generateBoard: 0.0
 - *Método: initialize: 0.0
 - *Método: initializeInterfaceAttributes: 0.0
 - *Método: numbPlayers: 1.0
 - *Método: player: 1.0
 - *Método: positionsView: 1.0
 - *Método: turnPlayer: 1.0
 - *Método: winnerExists: 1.0
 - *Método: auxDice: 0.0
 - *Método: auxDiceView: 0.0
 - *Método: board: 0.0
 - *Método: dice: 0.0
 - *Método: diceView: 0.0
 - *Método: finished: 0.0
 - *Método: numbPlayers: 0.0
 - *Método: player: 0.0
 - *Método: positionsView: 0.0
 - *Método: turnPlayer: 0.0
 - *Método: winnerExists: 0.0
 - *Método: update: 2.0
- NOM - Número de Métodos nas Classes
- Classe: ClickModel: 7.0
 - Classe: DiceSixFacesView: 3.0
 - Classe: FraGDefaultImageRepository: 1.0
 - Classe: Piece: 7.0
 - Classe: PieceDefault: 1.0
 - Classe: Player: 15.0
 - Classe: PlayerDefault: 1.0
 - Classe: Position: 13.0
 - Classe: PositionDefault: 2.0
 - Classe: PositionDefaultView: 2.0
 - Classe: ObservablePlus: 3.0
 - Classe: TwoDList: 10.0
 - Classe: Account: 6.0
 - Classe: Board: 14.0
 - Classe: ClickController: 1.0
 - Classe: Equipment: 1.0

- Classe: PlayerAutomatic: 5.0
- Classe: PositionContents: 10.0
- Classe: PositionDecorator: 19.0
- Classe: Dice: 5.0
- Classe: DiceDecorator: 9.0
- Classe: DiceSixFaces: 3.0
- Classe: GameInterface: 41.0
- NOC - Número de classes Imediatamente Descendentes
 - Classe: ClickModel: 2.0
 - Classe: DiceSixFacesView: 0.0
 - Classe: FraGDefaultImageRepository: 0.0
 - Classe: Piece: 1.0
 - Classe: PieceDefault: 0.0
 - Classe: Player: 2.0
 - Classe: PlayerDefault: 0.0
 - Classe: Position: 2.0
 - Classe: PositionDefault: 0.0
 - Classe: PositionDefaultView: 0.0
 - Classe: ObservablePlus: 0.0
 - Classe: TwoDList: 0.0
 - Classe: Account: 0.0
 - Classe: Board: 0.0
 - Classe: ClickController: 0.0
 - Classe: Equipment: 0.0
 - Classe: PlayerAutomatic: 0.0
 - Classe: PositionContents: 0.0
 - Classe: PositionDecorator: 0.0
 - Classe: Dice: 2.0
 - Classe: DiceDecorator: 0.0
 - Classe: DiceSixFaces: 0.0
 - Classe: GameInterface: 0.0
- DIT - Profundidade de Árvore de Herança
 - Classe: ClickModel: 1.0
- Classe: DiceSixFacesView: 1.0
- Classe: FraGDefaultImageRepository: 1.0
- Classe: Piece: 1.0
- Classe: PieceDefault: 1.0
- Classe: Player: 1.0
- Classe: PlayerDefault: 1.0
- Classe: Position: 1.0
- Classe: PositionDefault: 1.0
- Classe: PositionDefaultView: 1.0
- Classe: ObservablePlus: 1.0
- Classe: TwoDList: 1.0
- Classe: Account: 1.0
- Classe: Board: 1.0
- Classe: ClickController: 1.0
- Classe: Equipment: 1.0
- Classe: PlayerAutomatic: 1.0
- Classe: PositionContents: 1.0
- Classe: PositionDecorator: 1.0
- Classe: Dice: 1.0
- Classe: DiceDecorator: 1.0
- Classe: DiceSixFaces: 1.0
- Classe: GameInterface: 1.0
- MIF - Fator de Herança de Métodos: 0.23595505617977527
- AIF - Fator de Herança de Atributos: 0.0
- RFC - Reação de uma Classe
 - Classe: ClickModel: 7.0
 - Classe: DiceSixFacesView: 3.0
 - Classe: FraGDefaultImageRepository: 1.0
 - Classe: Piece: 7.0
 - Classe: PieceDefault: 1.0
 - Classe: Player: 15.0
 - Classe: PlayerDefault: 1.0
 - Classe: Position: 13.0
 - Classe: PositionDefault: 2.0
 - Classe: PositionDefaultView: 2.0

- Classe: ObservablePlus: 3.0
- Classe: TwoDList: 10.0
- Classe: Account: 6.0
- Classe: Board: 14.0
- Classe: ClickController: 1.0
- Classe: Equipment: 1.0
- Classe: PlayerAutomatic: 5.0
- Classe: PositionContents: 10.0
- Classe: PositionDecorator: 19.0
- Classe: Dice: 5.0
- Classe: DiceDecorator: 9.0
- Classe: DiceSixFaces: 3.0
- Classe: GameInterface: 41.0

- WMC - Métodos Complexos por Classe

- Classe: ClickModel: 7.0
- Classe: DiceSixFacesView: 3.0
- Classe: FraGDefaultImageRepository: 1.0
- Classe: Piece: 7.0
- Classe: PieceDefault: 1.0
- Classe: Player: 15.0
- Classe: PlayerDefault: 1.0
- Classe: Position: 13.0
- Classe: PositionDefault: 2.0
- Classe: PositionDefaultView: 2.0
- Classe: ObservablePlus: 3.0
- Classe: TwoDList: 10.0
- Classe: Account: 6.0
- Classe: Board: 14.0
- Classe: ClickController: 1.0
- Classe: Equipment: 1.0
- Classe: PlayerAutomatic: 5.0
- Classe: PositionContents: 10.0
- Classe: PositionDecorator: 19.0

- Classe: Dice: 5.0
- Classe: DiceDecorator: 9.0
- Classe: DiceSixFaces: 3.0
- Classe: GameInterface: 41.0

- RSC - Referência à Subclasses: 0.0

A.3.4 COESÃO

- LCOM - Falta de coesão métodos

- Classe: ClickModel: 0.0
- Classe: DiceSixFacesView: 0.0
- Classe: FraGDefaultImageRepository: 0.0
- Classe: Piece: 0.0
- Classe: PieceDefault: 0.0
- Classe: Player: 0.0
- Classe: PlayerDefault: 0.0
- Classe: Position: 4.0
- Classe: PositionDefault: 0.0
- Classe: PositionDefaultView: 0.0
- Classe: ObservablePlus: 0.0
- Classe: TwoDList: 0.0
- Classe: Account: 0.0
- Classe: Board: 2.0
- Classe: ClickController: 0.0
- Classe: Equipment: 0.0
- Classe: PlayerAutomatic: 0.0
- Classe: PositionContents: 0.0
- Classe: PositionDecorator: 0.0
- Classe: Dice: 0.0
- Classe: DiceDecorator: 0.0
- Classe: DiceSixFaces: 0.0
- Classe: GameInterface: 18.0

A.3.5 POLIMORFISMO

- PF - Fator de Polimorfismo: 0.11504424778761062
- OVO - Sobrecarga de Classes Isoladas
 - Classe: ClickModel: 1.0
 - Classe: DiceSixFacesView: 0.0
 - Classe: FraGDefaultImageRepository: 0.0
 - Classe: Piece: 3.0
 - Classe: PieceDefault: 0.0
 - Classe: Player: 7.0
 - Classe: PlayerDefault: 0.0
 - Classe: Position: 3.0
 - Classe: PositionDefault: 0.0
 - Classe: PositionDefaultView: 0.0
 - Classe: ObservablePlus: 0.0
 - Classe: TwoDList: 4.0
 - Classe: Account: 0.0
 - Classe: Board: 2.0
 - Classe: ClickController: 0.0
 - Classe: Equipment: 0.0
 - Classe: PlayerAutomatic: 1.0
 - Classe: PositionContents: 2.0
 - Classe: PositionDecorator: 4.0
 - Classe: Dice: 0.0
 - Classe: DiceDecorator: 0.0
 - Classe: DiceSixFaces: 0.0
 - Classe: GameInterface: 11.0
- SPA - Polimorfismo estático nos ancestrais
 - Classe: ClickModel: 0.0
 - Classe: DiceSixFacesView: 0.0
 - Classe: FraGDefaultImageRepository: 0.0
 - Classe: Piece: 0.0
 - Classe: PieceDefault: 0.0
 - Classe: Player: 0.0
- Classe: PlayerDefault: 0.0
- Classe: Position: 0.0
- Classe: PositionDefault: 0.0
- Classe: PositionDefaultView: 0.0
- Classe: ObservablePlus: 2.0
- Classe: TwoDList: 4.0
- Classe: Account: 0.0
- Classe: Board: 0.0
- Classe: ClickController: 0.0
- Classe: Equipment: 0.0
- Classe: PlayerAutomatic: 0.0
- Classe: PositionContents: 0.0
- Classe: PositionDecorator: 6.0
- Classe: Dice: 0.0
- Classe: DiceDecorator: 0.0
- Classe: DiceSixFaces: 0.0
- Classe: GameInterface: 0.0
- SPD - Polimorfismo estático nos descendentes
 - Classe: ClickModel: 0.0
 - Classe: DiceSixFacesView: 0.0
 - Classe: FraGDefaultImageRepository: 0.0
 - Classe: Piece: 0.0
 - Classe: PieceDefault: 0.0
 - Classe: Player: 0.0
 - Classe: PlayerDefault: 0.0
 - Classe: Position: 6.0
 - Classe: PositionDefault: 0.0
 - Classe: PositionDefaultView: 0.0
 - Classe: ObservablePlus: 0.0
 - Classe: TwoDList: 0.0
 - Classe: Account: 0.0
 - Classe: Board: 0.0
 - Classe: ClickController: 0.0
 - Classe: Equipment: 0.0
 - Classe: PlayerAutomatic: 0.0
 - Classe: PositionContents: 0.0
 - Classe: PositionDecorator: 0.0

- Classe: Dice: 0.0
- Classe: DiceDecorator: 0.0
- Classe: DiceSixFaces: 0.0
- Classe: GameInterface: 0.0

●SP - Polimorfismo estático em relação de herança

- Classe: ClickModel: 0.0
- Classe: DiceSixFacesView: 0.0
- Classe: FraGDefaultImageRepository: 0.0
- Classe: Piece: 0.0
- Classe: PieceDefault: 0.0
- Classe: Player: 0.0
- Classe: PlayerDefault: 0.0
- Classe: Position: 6.0
- Classe: PositionDefault: 0.0
- Classe: PositionDefaultView: 0.0
- Classe: ObservablePlus: 0.0
- Classe: TwoDList: 0.0
- Classe: Account: 0.0
- Classe: Board: 0.0
- Classe: ClickController: 0.0
- Classe: Equipment: 0.0
- Classe: PlayerAutomatic: 0.0
- Classe: PositionContents: 0.0
- Classe: PositionDecorator: 0.0
- Classe: Dice: 0.0
- Classe: DiceDecorator: 0.0
- Classe: DiceSixFaces: 0.0
- Classe: GameInterface: 0.0

●DPA - Polimorfismo Dinâmico nos Ancestrais

- Classe: ClickModel: 0.0
- Classe: DiceSixFacesView: 0.0
- Classe: FraGDefaultImageRepository: 0.0

- Classe: Piece: 0.0
- Classe: PieceDefault: 0.0
- Classe: Player: 0.0
- Classe: PlayerDefault: 0.0
- Classe: Position: 0.0
- Classe: PositionDefault: 0.0
- Classe: PositionDefaultView: 0.0
- Classe: ObservablePlus: 0.0
- Classe: TwoDList: 0.0
- Classe: Account: 0.0
- Classe: Board: 0.0
- Classe: ClickController: 0.0
- Classe: Equipment: 0.0
- Classe: PlayerAutomatic: 0.0
- Classe: PositionContents: 0.0
- Classe: PositionDecorator: 0.0
- Classe: Dice: 0.0
- Classe: DiceDecorator: 0.0
- Classe: DiceSixFaces: 0.0
- Classe: GameInterface: 0.0

●DPD - Polimorfismo Dinâmico nos Descendentes

- Classe: ClickModel: 0.0
- Classe: DiceSixFacesView: 0.0
- Classe: FraGDefaultImageRepository: 0.0
- Classe: Piece: 0.0
- Classe: PieceDefault: 0.0
- Classe: Player: 0.0
- Classe: PlayerDefault: 0.0
- Classe: Position: 0.0
- Classe: PositionDefault: 0.0
- Classe: PositionDefaultView: 0.0
- Classe: ObservablePlus: 0.0
- Classe: TwoDList: 0.0
- Classe: Account: 0.0
- Classe: Board: 0.0
- Classe: ClickController: 0.0
- Classe: Equipment: 0.0

- Classe: PlayerAutomatic: 0.0
- Classe: PositionContents: 0.0
- Classe: PositionDecorator: 0.0
- Classe: Dice: 0.0
- Classe: DiceDecorator: 0.0
- Classe: DiceSixFaces: 0.0
- Classe: GameInterface: 0.0

●DP - Polimorfismo Dinâmico

- Classe: ClickModel: 0.0
- Classe: DiceSixFacesView: 0.0
- Classe: FraGDefaultImageRepository: 0.0
- Classe: Piece: 0.0
- Classe: PieceDefault: 0.0
- Classe: Player: 0.0
- Classe: PlayerDefault: 0.0
- Classe: Position: 0.0
- Classe: PositionDefault: 0.0
- Classe: PositionDefaultView: 0.0
- Classe: ObservablePlus: 0.0
- Classe: TwoDList: 0.0
- Classe: Account: 0.0
- Classe: Board: 0.0
- Classe: ClickController: 0.0
- Classe: Equipment: 0.0
- Classe: PlayerAutomatic: 0.0
- Classe: PositionContents: 0.0
- Classe: PositionDecorator: 0.0
- Classe: Dice: 0.0

- Classe: DiceDecorator: 0.0
- Classe: DiceSixFaces: 0.0
- Classe: GameInterface: 0.0

●NIP - Polimorfismo em Relação sem Herança

- Classe: ClickModel: 10.0
- Classe: DiceSixFacesView: 7.0
- Classe: FraGDefaultImageRepository: 1.0
- Classe: Piece: 21.0
- Classe: PieceDefault: 1.0
- Classe: Player: 33.0
- Classe: PlayerDefault: 1.0
- Classe: Position: 26.0
- Classe: PositionDefault: 3.0
- Classe: PositionDefaultView: 3.0
- Classe: ObservablePlus: 3.0
- Classe: TwoDList: 22.0
- Classe: Account: 9.0
- Classe: Board: 26.0
- Classe: ClickController: 1.0
- Classe: Equipment: 1.0
- Classe: PlayerAutomatic: 14.0
- Classe: PositionContents: 14.0
- Classe: PositionDecorator: 40.0
- Classe: Dice: 8.0
- Classe: DiceDecorator: 14.0
- Classe: DiceSixFaces: 6.0
- Classe: GameInterface: 83.0