UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

VISUALIZAÇÃO DE SOFTWARE BASEADA EM UMA METÁFORA DO UNIVERSO UTILIZANDO O CONJUNTO DE MÉTRICAS CK

RENATO CORREA JULIANO

Uberlândia - Minas Gerais
2014

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

RENATO CORREA JULIANO

VISUALIZAÇÃO DE SOFTWARE BASEADA EM UMA METÁFORA DO UNIVERSO UTILIZANDO O CONJUNTO DE MÉTRICAS CK

Dissertação de Mestrado apresentada à Faculdade de Computação da Universidade Federal de Uberlândia, Minas Gerais, como parte dos requisitos exigidos para obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Banco de Dados e Imagens.

Orientador:
Prof. Dr. Bruno Augusto Nassif Travençolo

Coorientador:
Prof. Dr. Michel dos Santos Soares

Uberlândia, Minas Gerais
2014

Dados Internacionais de Catalogação na Publicação (CIP)
Sistema de Bibliotecas da UFU, MG, Brasil.

J94v
2014

Juliano, Renato Correa,
Visualização de software baseada em uma metáfora do universo utilizando o conjunto de métricas CK / Renato Correa Juliano. - 2014.
121 p. : il.
Orientador: Bruno Augusto Nassif Travençolo.
Coorientador: Michel dos Santos Soares.
Dissertação (mestrado) - Universidade Federal de Uberlândia, Programa de Pós-Graduação em Ciência da Computação.
Inclui bibliografia.

1. Computação - Teses. 2. Engenharia de software - Teses. 3. Medição de software - Teses. I. Travençolo, Bruno Augusto Nassif. II. Soares, Michel dos Santos. III. Universidade Federal de Uberlândia. Programa de Pós-Graduação em Ciência da Computação. IV. Título.

CDU: 681.3

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Os abaixo assinados, por meio deste, certificam que leram e recomendam para a Faculdade de Computação a aceitação da dissertação intitulada "**Visualização de software baseada em uma metáfora do universo utilizando o conjunto de métricas CK**" por **Renato Correa Juliano** como parte dos requisitos exigidos para a obtenção do título de **Mestre em Ciência da Computação**.

Uberlândia, 17 de Março de 2014

Orientador: _____
Prof. Dr. Bruno Augusto Nassif Travençolo
Universidade Federal de Uberlândia

Coorientador: _____
Prof. Dr. Michel dos Santos Soares
Universidade Federal de Sergipe

Banca Examinadora:

_____
Prof. Dr. Moacir Pereira Ponti Junior
Universidade de São Paulo

_____
Prof. Dr. Marcelo de Almeida Maia
Universidade Federal de Uberlândia

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Data: Março de 2014

Autor:	**Renato Correa Juliano**
Título:	**Visualização de software baseada em uma metáfora do universo utilizando o conjunto de métricas CK**
Faculdade:	**Faculdade de Computação**
Grau:	**Mestrado**

_____

Autor

Dedicatória

A Deus, família e amigos, base de tudo.

Agradecimentos

A Deus, pela oportunidade, proteção, carinho e força ao longo desses três anos de estudos e viagens.

À minha esposa, Marina, pelo amor, compreensão, paciência e dedicação.

À minha família, em especial Marcy, Gilberto e Laís, pelo exemplo, amor e suporte dado durante toda a minha vida.

Ao meu padrinho Márcio, que mesmo distante, fez-se presente nesta jornada, abrindo as portas da informática.

Aos professores Bruno e Michel, pelos ensinamentos transmitidos, pela transposição das barreiras que apareceram ao longo do caminho e pelo exemplo de ótimos profissionais da educação.

Aos professores Marcelo e Moacir, por examinar e contribuir com este trabalho.

Aos professores Válter, Pedro, Winicius e Luiz Moço, pela amizade e sabedoria passada ao longo dos meus oito anos de Uniaraxá.

Ao Centro Universitário do Planalto de Araxá, pelo incentivo financeiro, pela flexibilidade de horários, pelo aplicativo e mão de obra disponibilizada para a realização dos estudos.

Aos meus amigos e companheiros de trabalho e mestrado que me ajudaram e tornaram esta realização possível.

"O futuro pertence àqueles que acreditam na beleza de seus sonhos."
Elleanor Roosevelt

Resumo

O processo de desenvolvimento de software é uma atividade complexa, custosa e difícil, sendo tema de diversos estudos ao longo dos anos. Para tornar o software mais tangível, o cálculo de métricas é utilizado para fornecer informações úteis aos desenvolvedores, auxiliando no processo de tomada de decisões. Além do uso de métricas, é crescente o uso da visualização de software para auxílio na compreensão e desenvolvimento de softwares. O presente trabalho utiliza as métricas de Chidamber e Kemerer (métricas CK) para medir sistemas desenvolvidos sob o paradigma da orientação a objetos. Foi realizada uma revisão sistemática que expôs a eficiência das métricas CK em diversos estudos. As métricas CBO, RFC e WMC foram utilizadas com sucesso em todos os estudos analisados, enquanto as demais (LCOM, DIT e NOC) obtiveram sucesso em apenas alguns. Além disso, foi criada uma tabela com valores dessas medidas que apontaram possíveis problemas do software, como a predição da propensão a erros. Por fim, é proposto um novo modelo de visualização de software, baseado em uma metáfora simplista do universo, que visa facilitar a compreensão de programas transformando as classes de um sistema em corpos celestes que possuem características estipuladas de acordo com o valor das métricas. Esse modelo foi implementado no software denominado SUVsoft, que realiza a visualização de software e calcula o conjunto de métricas CK. Cinco softwares, de diferentes tamanhos e contextos foram visualizados e analisados e foi possível observar que a aplicação do conceito de força gravitacional com características definidas por métricas (cor e raio) possibilitaram a identificação de classes que possuem valores de métricas discrepante das demais. Os resultados deste trabalho podem ser utilizados para nortear futuros estudos sobre métricas de CK e também auxiliar nas atividades de Engenharia de Software, por meio do modelo de visualização proposto.

Palavras chave: engenharia de software, métricas de software, anomalias, visualização de software.

Abstract

The software development process is a complex, costly and difficult task, being subject of several studies over the years. In order to turn software more tangible, metrics calculation is used to provide useful informations to support developers in decision-making process. This present study used Chidamber and Kemerer suite metrics (CK metrics) to measure software developed under object oriented paradigm. A sistematic review was conducted and exposed the efficiency of CK metrics in several studies. Some metrics as CBO, RFC and WMC were successfully used in all studies analyzed, while other (LCOM , DIT and NOC) were successful in only a few. In addition, a table with values of these measures indicated possible software problems, such as predicting error prone, was created. A new visualization model is proposed, based on a simplistic metaphor of universe, which aims to facilitate the understanding of softwares transforming the classes of a system in celestial bodies have stipulated characteristics according to the value of metrics. This model was implemented in a software, called SUVsoft, that performs visualization software and calculate the CK metrics suite. Finally, five softwares of different sizes and contexts were visualized and analyzed, and it was observed that the application of gravitational force herewith color and radius allowed the identification of classes with discrepant values in CK metrics. The results of this dissertation can be used to guide future studies of CK metrics and also to assist Software Engineering activities through the visualization using the proposed model.

Keywords: software engineering, software metrics, anomalies, software visualization.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

AH	Área Hierárquica do nó pai
CBO	*Coupling Between Objects*
CC	Complexidade Ciclomática
CK	Chidamber e Kemerer
DIT	*Depth of Inheritance Tree*
IQM	Média Interquartil
K	Constante para definir o tipo de visualização (circular ou árvore)
LCOM	*Lack of Cohesion Methods*
N	Número de nós filhos
NOC	*Number of Children* (métrica)
P	Perímetro do nó pai de acordo com os nós filhos
POO	Programação Orientada a Objetos
R	Raio do nó
RFC	*Response for a Class*
RH	Raio Hierárquico do nó
WMC	Weight Methods per Class
XML	*eXtensible Markup Language*

Capítulo 1

Introdução

Software é um elemento essencial para o mundo moderno. Vários setores da sociedade como, indústria, entidades financeiras e entretenimento fazem uso constante de softwares. Portanto, a engenharia de software é necessária para o funcionamento e manutenção dessas atividades, que são muito importantes para a sociedade [Sommerville 2009].

A vida útil do software precisa ser prolongada ao máximo, mantendo sempre otimizada a relação de custo/benefício. Essa tarefa pode ser alcançada por meio da engenharia de software [Sommerville 2009]. Com isso, técnicas surgiram ao longo dos anos, visando aprimorar a manutenção e qualidade desses softwares. Dentre elas, novos modelos, como o orientado a objetos, eclodiram para facilitar o processo de desenvolvimento e manutenção [Booch et al. 2008]. A Programação Orientada a Objetos (POO) é baseada em três conceitos fundamentais: abstração dos tipos de dados, herança e vinculação dinâmica. Esse paradigma é suportado por classes, métodos, objetos e troca de mensagens [Sebesta 2009]. [Booch et al. 1994] define a POO como um conjunto de características relacionadas a abstração, encapsulamento e hierarquia que facilitam o entendimento e resolução de problemas, pois são próximas a outras atividades inerentes do ser humano.

A POO foi proposta com diversos objetivos, dentre eles aumentar o nível de abstração no desenvolvimento de software e facilitar a decomposição de problemas complexos em partes mais gerenciáveis. Outras propriedades como reuso de componentes e facilidade na realização de mudanças, graças a possibilidade de divisão em módulos, contribuíram para a popularização da POO [Booch et al. 1994].

Com a popularização da POO, uma grande quantidade de produtos de software foi criada usando esse paradigma. Portanto, a necessidade de monitorar, controlar e gerenciar o processo de desenvolvimento se tornou um desafio às instituições [Basili et al. 1996].

Conforme afirmado por DeMarco, "não se pode controlar o que não se pode quantificar" [DeMarco 1982]. As métricas de software possibilitam a quantificação de importantes pontos de controle do processo de desenvolvimento de software e da qualidade do produto [Li e Henry 1993]. Portanto, são necessárias para que sejam tomadas decisões gerenciais como, por exemplo, alocação de recursos [Basili et al. 1996]. Diversas atividades

foram realizadas ao longo dos anos utilizando métricas de software. Em específico, as métricas de Chidamber e Kemerer – métricas CK – ([Chidamber e Kemerer 1994]) foram usadas para diferentes estudos, como predizer a propensão a erros que uma classe pode ter [Subramanyam e Krishnan 2003,Gyimothy et al. 2005,Janes et al. 2006,Olague et al. 2007,Shatnawi e Li 2008,English et al. 2009,Nair e Selvarani 2011,Singh e Kahlon 2011,Johari e Kaur 2012,Singh e Kahlon 2012], analisar o impacto da refatoração nos valores das métricas [Stroggylos e Spinellis 2007], analisar o impacto da refatoração na reusabilidade com base em métricas [Moser et al. 2006] e análise da facilidade de reuso caixa-branca [Kakarontzas et al. 2012].

Vários problemas em softwares orientados a objetos podem aparecer durante o processo de desenvolvimento. Esses problemas podem ser de natureza do próprio modelo. Entre outros, a falta de um modelo de ciclo de vida adequado para suportar reusabilidade, a habilidade em assegurar a qualidade do produto e processo de desenvolvimento e a capacidade de avaliar a produtividade da equipe de desenvolvimento podem dificultar o desenvolvido de software e, portanto, merecem ser tema de estudos [Abreu e Carapua 1994]. Além do mais, como Brooks afirma, a complexidade do software é uma propriedade e não um acidente. E essa complexidade deve ser acompanhada e monitorada para que seja controlada [Brooks Jr. 1987]. Baseando-se nisso, a utilização de métricas durante o processo de desenvolvimento torna-se uma ferramenta útil para a melhoria de atividades gerenciais, como controle, condução e acompanhamento.

A atividade de mensurar características do software pode auxiliar na melhoria estrutural do software, na redução da manutenção do produto e, consequentemente, na melhora da qualidade do software. Entretanto, a atividade de descobrir possíveis problemas nem sempre pode exibir, de maneira nítida, os problemas do software. Software, por ser intangível, dificulta essa descoberta [Koschke 2003].

Portanto, por meio da visualização de software, é possível criar formas gráficas para exibição de métricas, tornando-as mais tangíveis e mais intuitivas [Wettel 2010]. O objetivo é facilitar a compreensão do software por parte do usuário perante os problemas que o software possui, seja esse usuário um gestor de projeto de desenvolvimento, um desenvolvedor ou um testador.

Pela importância atribuída a ela, a visualização de software é um campo da engenharia de software com diversos estudos dirigidos e uma ativa comunidade [Wettel 2010]. Segundo [Price et al. 1993], a visualização de software pode ser definida como a utilização de tipografia, desenho de gráficos, animações e cinematografia, juntamente com técnicas modernas de interação humano-computador e tecnologias computacionais gráficas para facilitar o entendimento e o uso efetivo de um sistema computacional. Já [Koschke 2003] trata a visualização de software como o mapeamento de artefatos de softwares (incluindo código-fonte) por meio de representação gráfica para aprimorar a compreensão. Isso é necessário porque o software é intangível e uma forma gráfica o torna visível.

A atuação deste trabalho está situada na área de visualização de estruturas e métricas de software. As métricas utilizadas são as propostas por [Chidamber e Kemerer 1994]. Por meio da visualização, é possível identificar classes que possuem valores de suas métricas discrepantes das demais, que possam indicar anomalias. Essas informações podem assistir a tomada de decisão durante o ciclo de vida do software.

1.1 Objetivos

O objetivo deste trabalho é propor um modelo de visualização de software baseado em uma metáfora com o universo e guiado pelo conjunto de métricas CK, em que características específicas de softwares orientados a objetos como acoplamento, herança e complexidade sejam visualizadas.

Objetivos específicos

- Analisar a aplicabilidade do conjunto de métricas CK em atividades de Engenharia de Software;
- Identificar limiares em valores das métricas de CK propostos em outros estudos;
- Propor uma técnica de visualização de software baseada na metáfora com universo;

1.2 Estrutura do trabalho

Esta dissertação está organizada em seis capítulos. O presente capítulo apresentou a motivação para a realização desse trabalho e os objetivos do estudo.

No Capítulo 2, um breve referencial teórico sobre os conceitos que foram aplicados no presente trabalho é apresentado.

No Capítulo 3 é realizada uma análise de 13 trabalhos correlatos na área de visualização de software que, conceitualmente, se aproximam deste estudo.

No Capítulo 4 é feita uma revisão sistemática sobre a aplicabilidade e eficiência das métricas CK em atividades de Engenharia de Software como predição da propensão a erros, impacto da refatoração nos valores das métricas e análise da facilidade do reuso caixa-branca.

No Capítulo 5 é apresentado o modelo de visualização proposto que é baseado numa metáfora simplista do universo em que as métricas de CK norteiam a visualização.

Por fim, no Capítulo 6, é realizada a conclusão e são expostos os trabalhos futuros.

Capítulo 2

Referencial teórico

Neste capítulo são apresentados conceitos relevantes para a realização do trabalho proposto nesta dissertação. Inicialmente, os conceitos básicos sobre compreensão de programas e engenharia reversa são explanados. Em seguida, a visualização de software e seus modelos são explicados brevemente e, por fim, as métricas de software orientadas a objetos são expostas.

2.1 Compreensão de programas

Um passo inicial para realizar modificações em um sistema é compreendê-lo primeiro. Uma possível solução para facilitar esse entendimento é a execução de atividades relacionadas à engenharia reversa, que é definida como o processo de analisar sistemas visando a identificação de seus componentes e inter-relacionamentos existentes, com o objetivo de criar representações do sistema em um nível mais alto de abstração [Chikofsky e Cross 1990].

A atividade de compreender programas é difícil e custosa, dispendendo tempo e recursos financeiros. Estima-se que os gastos com a manutenção de softwares estejam no intervalo de 60%–90% do total [McKee 1984,Pressman 2005]. Portanto, atividades que diminuam o trabalho de manter softwares tornam-se importantes para aumentar o ciclo de vida dos aplicativos. Entretanto, a compreensão de programas não é uma tarefa trivial, conforme os três itens a seguir:

Softwares são intangíveis. A própria natureza do software torna-o intangível, não possuindo forma ou tamanho [Ball e Eick 1996]. Tal característica dificulta seu entendimento e análise, impulsionando o estudo e desenvolvimento de novas técnicas para amenizar tal situação ao longo dos anos [Koschke 2003]. Vale ressaltar que, segundo [Ware 2013], os seres humanos conseguem obter mais informações pelo sentido da visão do que todos os demais combinados. Essa afirmação reforça a utilização de técnicas de visualização de software como forma de apoio a atividades de engenharia de software.

Softwares são grandes e complexos. O tamanho e a complexidade dos softwares

dificultam sua compreensão, desde pequenas partes até o sistema como um todo. Além disso, a própria complexidade do software está relacionada ao tamanho do mesmo [Caserta e Zendra 2011].

Softwares evoluem. Para acompanhar a rápida mudança do mercado, as empresas e, consequentemente, seus softwares, devem estar aptos a receber e aceitar mudanças. Com isso, a evolução dos softwares torna-se fator de sucesso para que empresas consigam se destacar em um mercado competitivo [Sommerville 2009]. Conforme as evoluções do software vão acontecendo, sua arquitetura e documentação tendem a aumentar. E, de acordo com a segunda lei de Lehman, a evolução de um software causa um aumento em sua complexidade [Lehman 1980].

2.2 Visualização de software

A visualização de software é uma subárea da visualização da informação, que tem como objetivo representar um software a partir de três pontos de vista distintos: estático, dinâmico e de evolução [Diehl 2007].

2.2.1 Tipos de visualização

A **visualização estática** busca ilustrar as estruturas, relacionamentos e propriedades das entidades que o software possui. Os dados de entrada para o modelo são obtidos, geralmente, do código-fonte dos aplicativos, sem que haja a execução do programa [Caserta e Zendra 2011,Ware 2013].

A **visualização dinâmica** é baseada em informações obtidas durante a execução do software. Dados relacionados à chamada de métodos e troca de mensagens entre objetos compõe as informações inseridas nesse modelo. Este tipo de visualização busca auxiliar a compreensão do comportamento do software [Ware 2013].

A **visualização evolutiva** incrementa a visualização estática utilizando informações temporais ao modelo [Caserta e Zendra 2011]. Em alguns casos, como em [Caudwell 2010] e [Ogawa e Ma 2009], é feita uma análise do software com base nas informações obtidas de repositório de dados e essas informações são mostradas ao longo do tempo. Geralmente os dados que alimentam o modelo são provenientes de controladores de versão e são visualizados em forma de vídeo.

Esses três tipos de visualização são utilizados como forma de classificação dos estudos e modelos analisados no Capítulo 3. Embora todos os tipos de visualização sejam importantes na compreensão de software, o presente modelo descrito nesta dissertação limita seu escopo na visualização estática de software, com ênfase nos valores das métricas e estruturas hierárquicas.

2.2.2 Estágios da visualização

O processo de visualização possui quatro etapas, combinados com um conjunto de *feedbacks*. A Figura 2.1 ilustra essas etapas e a realimentação delas feita pelo usuário.

- Obtenção e entrada dos dados. Nessa etapa, o tipo de dado é diretamente ligado ao tipo de visualização. Informações como estrutura do software e valor de métricas, troca de mensagens e instanciação de objetos e modificações no código-fonte servem como fonte de dados para os modelos estático, dinâmico e evolutivo, respectivamente;
- Transformação dos dados em algo mais fácil de ser manipulado. Geralmente é realizado também um filtro para que somente as informações necessárias sejam enviadas ao modelo de visualização;
- Aplicação do modelo gráfico com base nas informações obtidas, gerando uma imagem na tela como resposta. Algumas informações passadas pelo usuário podem transformar a visualização ou selecionar um conjunto de informações considerada útil;
- Processamento visual e cognitivo por parte do usuário [Ware 2013].

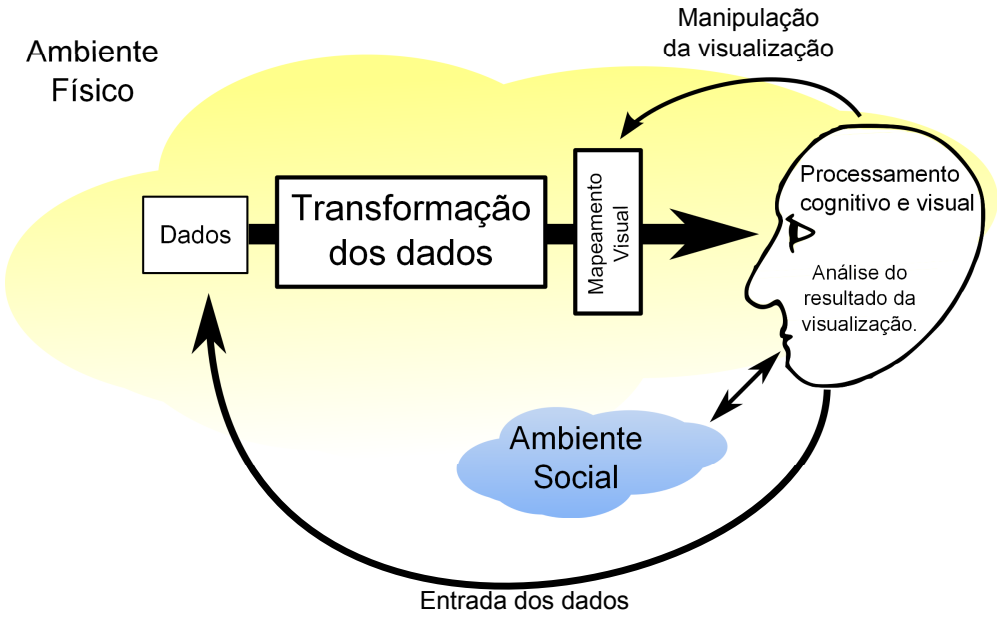


Figura 2.1: Etapas da visualização. Figura adaptada de [Ware 2013].

Durante o processo de visualização, a interação entre o usuário e o modelo gráfico é importante para aprimorar o resultado da visualização. Aplicação de filtros e refinamentos dos dados de entrada podem auxiliar na geração de uma visualização que se torne útil para a tomada de decisão por parte do usuário [Ware 2013].

Sendo assim, a principal questão relacionada à visualização é: como transformar dados em algo que as pessoas possam entender, interpretar e tomar decisões? A construção do

modelo, a interação entre o usuário e a visualização e os ambientes físico e social devem ser considerados e manipulados para responder essa questão e otimizar tal tarefa.

2.3 Métricas de software

A atividade de medir software tornou-se essencial para Engenharia de Software [Fenton e Pfleeger 1998]. Segundo [Sommerville 2009], "a medição do software preocupa-se com a derivação de um valor numérico ou o perfil para um atributo de um componente de software, sistema ou processo".

"O que não for mensurável, transforme-o em mensurável". Essa frase, atribuída a Galileu Galilei (1564–1642), ilustra a importância que é dada a prática de medir, em diferentes atividades da sociedade [Fenton e Pfleeger 1998]. [DeMarco 1982] afirma que "não se pode controlar o que não se pode medir". Então, assim como em outras áreas, as métricas são importantes para analisar e gerenciar as atividades de Engenharia de Software.

Entretanto, o cálculo de métricas deve ser feito com objetivo específico, baseado em o que usuários, desenvolvedores e gerentes querem saber [Fenton e Pfleeger 1998]. Isso é necessário para auxiliar na escolha correta das métricas.

Segundo [Fenton e Pfleeger 1998], diversas avaliações podem ser feitas de acordo com os valores de suas métricas, tais como o grau de qualidade do projeto, se o código está pronto para ser testado ou se os requisitos estão consistentes e completos.

[Sommerville 2009] afirma que um dos objetivos em se medir software é fornecer a capacidade de, a longo prazo, fazer julgamentos sobre a qualidade de software, ao invés de realizar revisões. A ideia é que sistemas fossem aprovados ou reprovados de acordo com os valores de um conjunto de métricas. Mas essa condição está, até o presente momento, longe de ser atingida e não há indícios de que se torne realidade.

O modelo apresentado nesta dissertação é baseado em conceitos utilizados na orientação a objetos, como hierarquia, complexidade e acoplamento. Portanto, as métricas utilizadas para nortear o modelo devem medir características relacionadas a softwares desenvolvidos sob o paradigma orientado a objetos. Para realizar tal medição, foi escolhido o conjunto de métricas propostas por Chidamber e Kemerer, em 1994 ([Chidamber e Kemerer 1994]). Diante de várias outras como os apresentados por [Abreu e Carapua 1994,Lorenz e Kidd 1994,Pfleeger e Palmer 1990], a escolha pelo conjunto de métricas CK foi embasada no estudo de [Radjenovi et al. 2013], em que os autores afirmam que esse conjunto é o mais popular na predição a propensão a erros entre as métricas orientadas a objetos.

2.3.1 Conjunto de métricas CK

O conjunto de métricas CK foi sugerido em 1994, por Chidamber e Kemerer [Chidamber e Kemerer 1994], para medir atributos de softwares que foram desenvolvidos sob o paradigma da orientação a objetos. Em seu estudo, eles descobriram que as métricas combinadas podem fornecer informações importantes para a tomada de decisão [Harrison et al. 1998].

Kemerer e Darcy [Chidamber et al. 1998] analisaram vários estudos que utilizaram o conjunto de métricas CK. Eles fizeram algumas observações sobre métricas orientadas a objetos [Booch et al. 2008], descritas a seguir.

- Cada métrica tem sido aplicada com sucesso em diversos domínios;
- As métricas, frequentemente, tiveram relação com fatores de qualidade como custo, defeitos, reúso e manutenção;
- As métricas WMC, CBO, RFC e LCOM foram utilizadas com sucesso na maioria dos estudos.

São descritas, a seguir, as métricas que compõe o conjunto de métricas CK.

DIT - *Depth of Inherance Tree* (Profundidade da Árvore de Herança)

Essa métrica abrange casos em que existe herança de classes. É a métrica que calcula a distância entre o nó da classe objetivo (a classe na qual deseja-se encontrar a distância) até o último nó da classe pai. Em casos de herança múltipla, a DIT irá registrar a classe que estiver mais distante [Chidamber e Kemerer 1994]. Essa métrica pode retratar a complexidade dos comportamentos de uma classe, a complexidade do projeto do sistema e se a classe pode ser reutilizada por outras classes [Harrison et al. 1998].

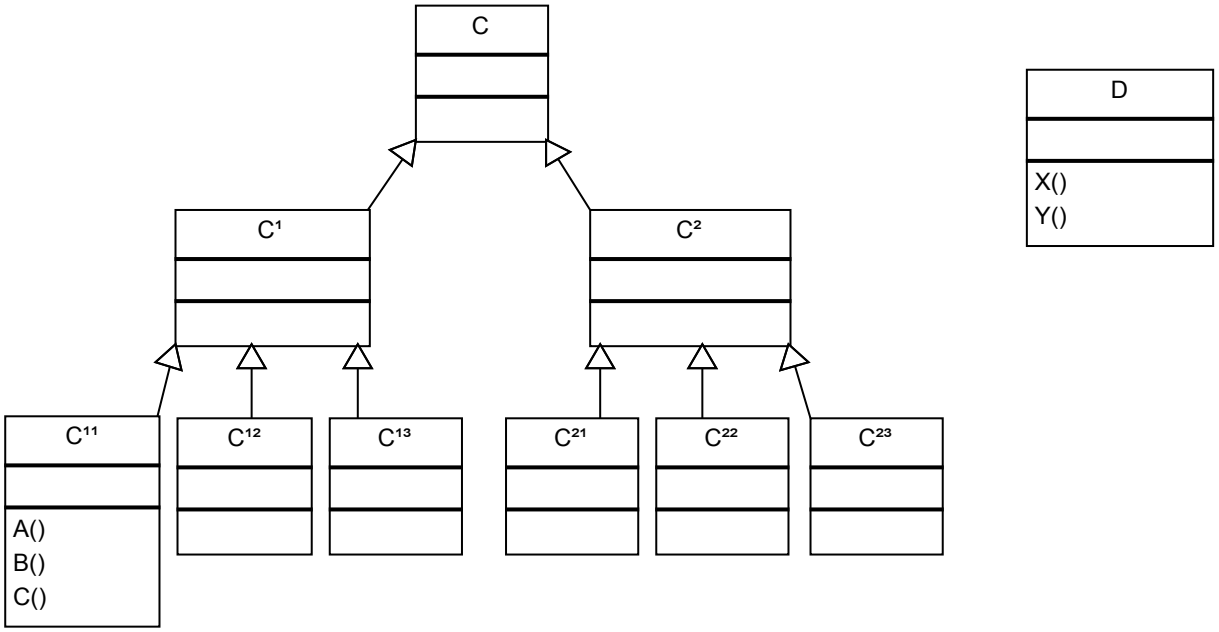
Para exemplificar, considere a Figura 2.2. Nela, a classe C possui o valor de DIT igual a 1, enquanto as classes C^1 e C^2 possuem o valor de DIT igual a 2. E, por fim, as classes C^{11}, C^{12}, C^{13}, C^{21}, C^{22} e C^{23} possuem valor 3 para a métrica DIT.

NOC - *Number of Children* (Número de Filhos)

A métrica NOC quantifica o número de subclasses imediatas de uma determinada classe. Ou seja, essa quantificação exibe quantas classes filhas uma classe tem. De acordo com [Chidamber e Kemerer 1994], quanto maior for o valor desta métrica, maior será a importância do teste. [Harrison et al. 1998] ainda cita que, por meio dessa métrica, é possível identificar o grau de reúso da classe [Chidamber e Kemerer 1994].

Considere novamente a Figura 2.2. Nela, a classe C possui dois filhos diretos (C^1 e C^2) e, portanto, a métrica NOC recebe o valor de 2. As classes C^1 e C^2 possuem três filhos cada e, portanto, a métrica NOC de ambas recebe o valor de 3.

Figura 2.2: Diagrama de classes para exemplificar o conjunto de métricas CK.



CBO - *Coupling Between Object Classes* (Acoplamento entre Classes)

A métrica CBO é definida como o número de classes na qual ela está acoplada. Uma classe está acoplada a outra quando pelo menos um método de uma delas utiliza métodos e/ou variáveis de instância da outra. Por meio dessa métrica é possível identificar o nível de reusabilidade de uma classe e seu grau de modularidade. Quanto mais baixo for o valor de CBO, mais modularizada a classe fica e, consequentemente, maior é a possibilidade de reusabilidade. Ainda é possível analisar o rigor do teste de software, seja ele em nível de importância, tempo dispendido ou quantidade de testes. Isto acontece porque, quanto maior for o valor, mais complexo será o teste [Chidamber e Kemerer 1994].

No exemplo da Figura 2.2, a classe C^{11} possui um método chamado *A()* que, por sua vez, utiliza em seu corpo os métodos *X()* e *Y()*, ambos da classe D. Portanto, o valor da métrica CBO para a classe C^{11} é de 2.

RFC - *Response for a Class* (Resposta para uma Classe)

Essa métrica representa a quantidade total de métodos da própria classe somados a quantidade de métodos invocados de outras classes (conforme Equação 2.1) [Harrison et al. 1998].

$$RFC = \{\mathrm{M}\} \cup_i \{R_i\}, \tag{2.1}$$

onde R_i é o conjunto de métodos chamados pelo método i e M é o conjunto total de métodos da classe em que o método i foi construído.

Para exemplificar o cálculo da métrica RFC, considere C^{11}, da Figura 2.2, como a

classe em análise. Esta classe possui três métodos (A(), B() e C()). O método A(), ao ser executado, invoca os métodos X() e Y() da classe D. Portanto, o valor de RFC é a soma dos métodos da classe C^{11} (3) acrescido dos métodos invocados pelos seus métodos (neste caso é o método A()) (2), totalizando 5.

Assim como a métrica CBO, a RFC também avalia o acoplamento entre classes. Pode-se observar que, quando uma classe está muito acoplada a outra, a depuração e o teste serão trabalhosos e a complexidade do sistema será alta [Chidamber e Kemerer 1994].

LCOM - *Lack of Cohesion in Methods* (Ausência de Coesão nos Métodos)

A métrica LCOM é a medida que quantifica a similaridade dos métodos dentro de uma classe. Dois ou mais métodos são coesos quando compartilham atributos de uma classe [Chidamber e Kemerer 1994].

Para realizar o cálculo, cada dupla de métodos que uma determinada classe possui é analisada. Então, é feita a intersecção entre os atributos utilizados pelos métodos. Caso haja intersecção, Q é incrementado, senão, P é incrementado. A Fórmula 2.2 ilustra o cálculo da métrica LCOM e, a seguir, é exemplificada.

$$LCOM = \begin{cases} 0 & \text{se } |P| > |Q| \\ |P| - |Q| & \text{caso contrário} \end{cases}, \tag{2.2}$$

A classe X possui quatro métodos com os seguintes conjuntos de variáveis de instância:

- $M_1 = \{a, b, c\}$;
- $M_2 = \{a, b\}$;
- $M_3 = \{c, d, e, f\}$;
- $M_4 = \{g, h\}$;

A primeira etapa é criar as tuplas entre os métodos e analisar se ambos compartilham variáveis de instância.

1. $M_1 \cap M_2 = \{a, b\}$;
2. $M_1 \cap M_3 = \{c\}$;
3. $M_1 \cap M_4 = \varnothing$;
4. $M_2 \cap M_3 = \varnothing$;
5. $M_2 \cap M_3 = \varnothing$;
6. $M_3 \cap M_4 = \varnothing$;

Portanto, a variável $P = 4 = \{3, 4, 5, 6\}$ e $Q = 2 = \{1, 2\}$. Com isto, o valor da métrica LCOM da classe X é igual a 2.

Quando o valor de LCOM é alto, significa que a classe não possui uma funcionalidade bem definida, uma vez que vários métodos modificam os mesmos atributos. Por outro lado, se este número for baixo, os métodos da classe serão coesos, e portanto, menos similares [Harrison et al. 1998].

Uma crítica feita sobre a métrica LCOM é que ela ilustra somente a falta de coesão entre os métodos e não se estes métodos são coesos. Por não permitir valor negativo, não é possível identificar alta coesão entre classes, ou dentro de um conjunto de classes, quais são mais coesas que as demais [Harrison et al. 1998]. Outra crítica está relacionada a utilidade da métrica, uma vez que não está muito claro se a informação provida pela métrica agrega valor além das que já são fornecidas por outras métricas [Sommerville 2009].

WMC - *Weighted Methods Per Class* (Métodos Ponderados por Classe)

O WMC é uma medida, inicialmente trabalhada por [Chidamber e Kemerer 1994], que é definida pela soma das complexidades dos métodos da classe. Segundo [Harrison et al. 1998], a definição de complexidade não foi feita por Chidamber e Kemerer, mas foi sugerida que fosse uma unidade (um número natural), deixando esse aspecto livre. Comumente, a métrica WMC é calculada com base na soma da complexidade ciclomática (CC) de cada um dos métodos de uma classe. Esta métrica analisa a quantidade de caminhos possíveis que um procedimento (método ou função) pode percorrer durante a sua execução [McCabe 1976].

A WMC proporciona uma visão sobre alguns pontos, tais como o número de métodos, sendo que a complexidade desses pode indicar quanto tempo seria necessário para desenvolver e dar manutenção em uma determinada classe (quanto maior for o número de métodos numa classe, maior será o impacto sobre suas classes filhas). Classes com muitos métodos geralmente são de aplicações específicas, diminuindo a possibilidade de reuso [Chidamber e Kemerer 1994].

2.4 Considerações finais

A visualização de software tem papel considerável em diversas etapas da Engenharia de Software, auxiliando nas atividades de engenharia reversa a compreensão de software. Portanto, é importante que modelos visuais consigam diminuir o tempo gasto na compreensão de programas e facilitem as tarefas de desenvolvedores e gestores para a tomada de decisão.

Esse capítulo apresentou, brevemente, conceitos de visualização de software. Também foram discutidas as métricas de softwares orientados a objetos do conjunto de CK e a importância delas no processo de desenvolvimento de software.

Capítulo 3

Trabalhos Correlatos

Neste capítulo são descritos os trabalhos correlatos que são referência na área de visualização de software e/ou utilizam métricas como critério de visualização. Os trabalhos estão ordenados cronologicamente, de acordo com a data de publicação.

3.1 SeeSoft

Em 1992, foi desenvolvido o SeeSoft [Eick et al. 1992], um dos trabalhos pioneiros na visualização de estruturas e métricas de software [Holten et al. 2007]. Esse aplicativo é baseado em quatro ideias-chave: redução da representação do código, coloração das métricas, manipulação das visualizações e possibilidade de leitura do código-fonte atual. O SeeSoft mapeia cada linha de código e representa-a por uma linha composta por vários pontos (*pixel*). A coloração é utilizada para indicar estatísticas de interesse do usuário do sistema, assim como uma métrica de software [Eick et al. 1992]. A Figura 3.1 ilustra a visualização da inclusão de linhas de código em um sistema, onde a escala azul-vermelho representa a data de inclusão dessas linhas. Quanto mais próximo da cor vermelha estiver, mais recente é a inclusão da linha de código.

Os conceitos do SeeSoft foram bem recebidos pela comunidade. Dentre eles, é possível citar a capacidade de, por meio da visualização, selecionar os objetos em questão e visualizar o código-fonte, criando uma conexão direta entre a visualização e o código-fonte. Isto cria uma navegação natural entre as representações [Marcus et al. 2003]. Ainda na Figura 3.1 é possível notar o código-fonte em ênfase.

3.2 SHriMP

Outro trabalho de destaque é o SHriMP, criado por [Storey 2001]. O SHriMP ilustra as diversas hierarquias presentes em um sistema orientado a objetos. É possível visualizar as ligações entre Pacotes/Classes e Classes/Classes, bem como analisar o código-fonte ou

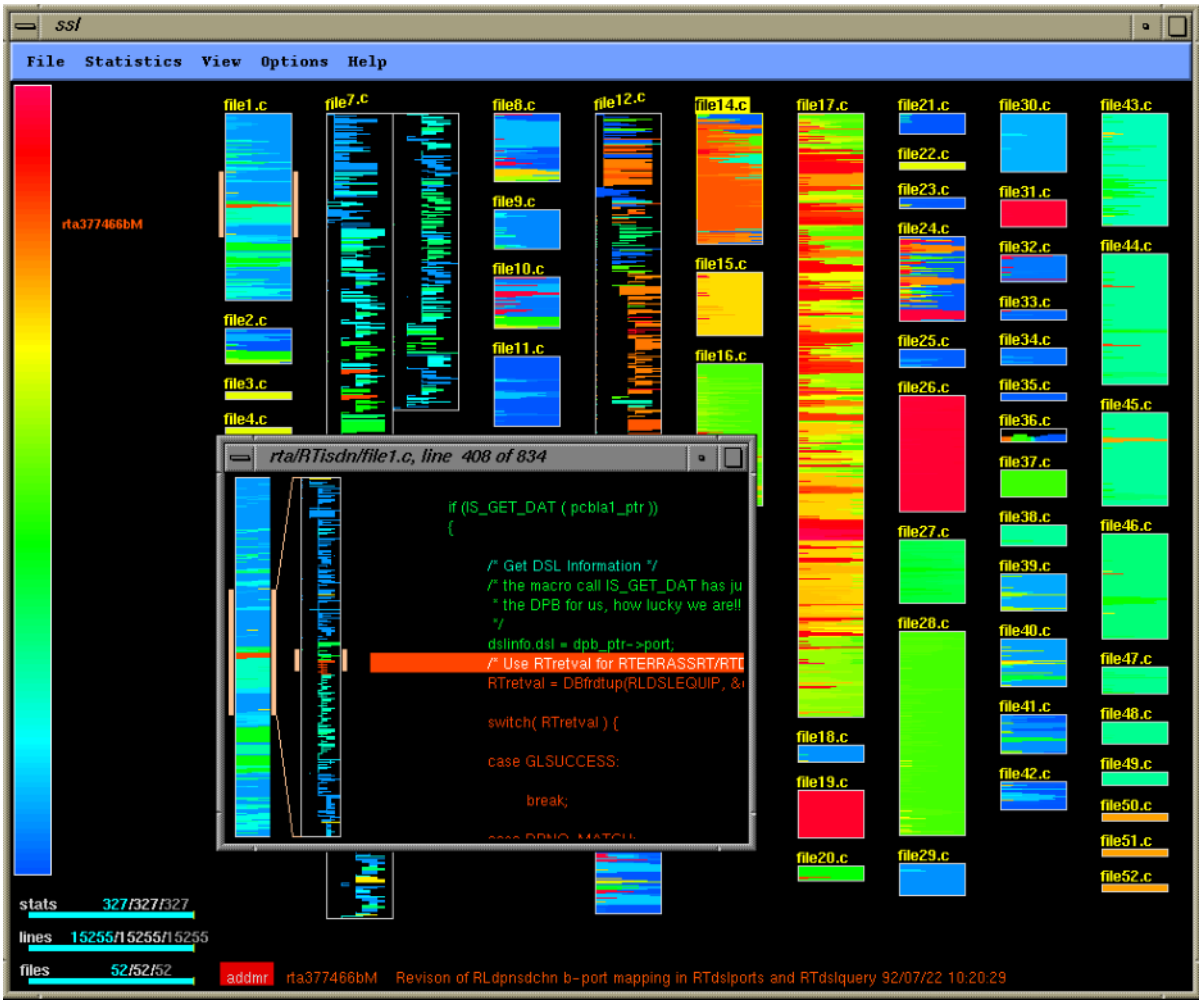


Figura 3.1: Visualização de um sistema pelo SeeSoft exibindo a data de inclusão de linhas de código. Figura obtida de [Eick et al. 1992].

até mesmo a documentação produzida em "Javadoc", conforme ilustrado na Figura 3.2. Entretanto, o tamanho do software a ser analisado impacta diretamente na visualização. Quanto maior for o software que será avaliado, pior fica a visualização [Storey 2001]. Por se tratar de um sistema de visualização de análise estrutural, a estrutura do software possibilita a descoberta de pontos destoantes, conforme [Chidamber e Kemerer 1994] descreve na métrica *Number of Children* - NOC. Porém, SHriMP não exibe de forma explícita este tipo de análise, por não ser o foco do trabalho. Portanto, esse tipo de análise fica por conta do próprio usuário, que pode não ter conhecimento prévio desse tipo de anomalia.

Existem 4 (quatro) formas de visualização possíveis no SHriMP. A visualização por "Classes" exibe a herança entre classes. A visualização por "Source Code" demonstra o código-fonte de alguma classe visualizada. A visualização por "Package" mostra a relação das classes e interfaces de um determinado pacote e a visulização por "Javadoc" exibe os comentários do código-fonte. O ponto de destaque dessas visualizações é a possibilidade de navegar pelo modelo utilizando *hyperlinks*, ampliando ou diminuindo o nível de abstração

da visualização. A Figura 3.2 demonstra os vários tipos de visões possíveis disponibilizadas pelo software SHriMP.

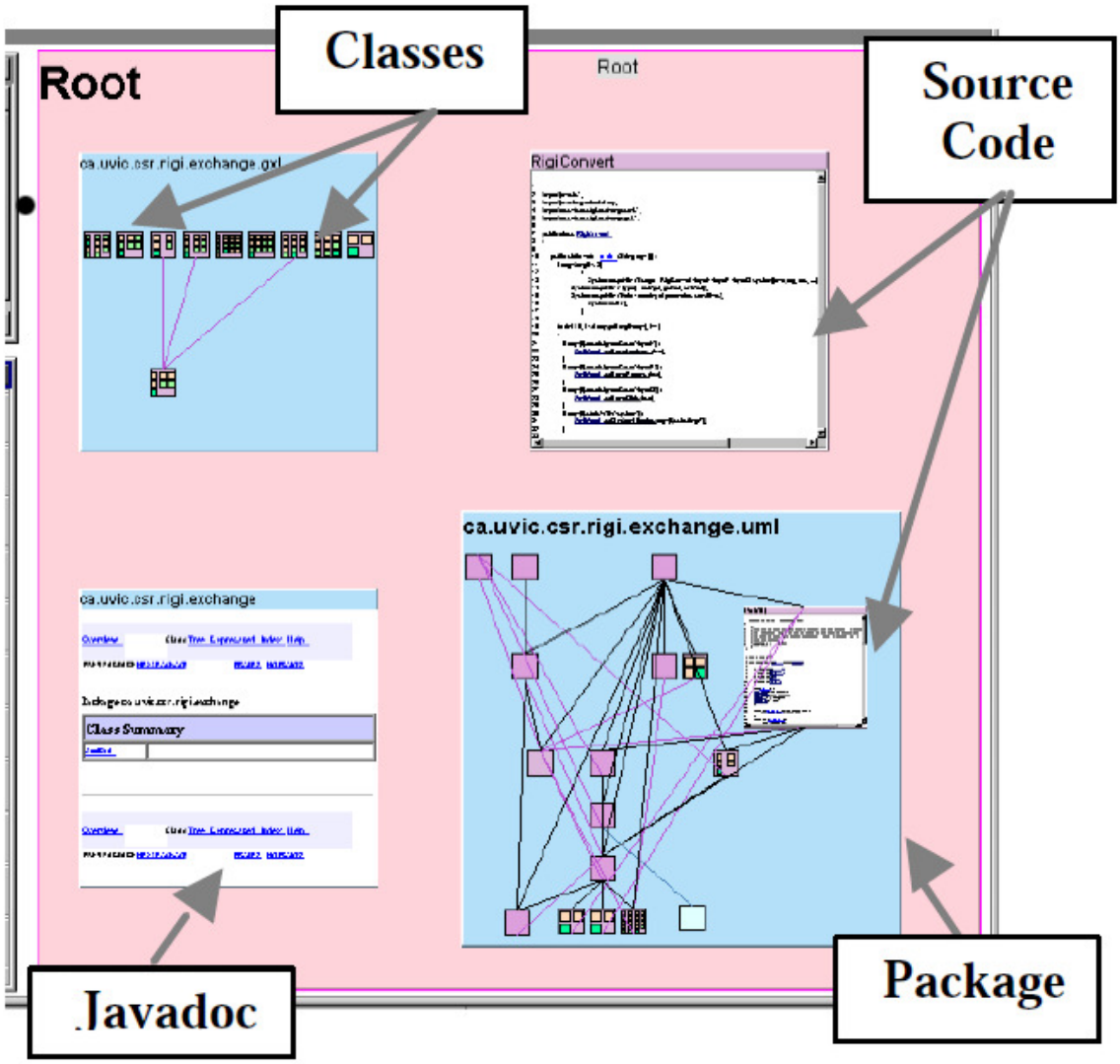


Figura 3.2: Possibilidades de visualização disponibilizadas pelo SHriMP. Figura obtida de [Storey 2001].

3.3 Tarantula

O Tarantula, por meio da atividade de visualização de software, busca auxiliar o trabalho de descoberta de erros e falhas utilizando a depuração de código-fonte. Cada instrução do sistema é representa por um conjunto de *pixels* (similar ao conceito do SeeSoft [Eick et al. 1992]) que são coloridos de acordo com a taxa de sucesso na execução de casos de teste. A Figura 3.3 ilustra a visualização de um aplicativo com aproximadamente 6800 instruções. Quanto maior o tamanho do aplicativo, mais difícil fica a sua visualização

[Jones et al. 2002].

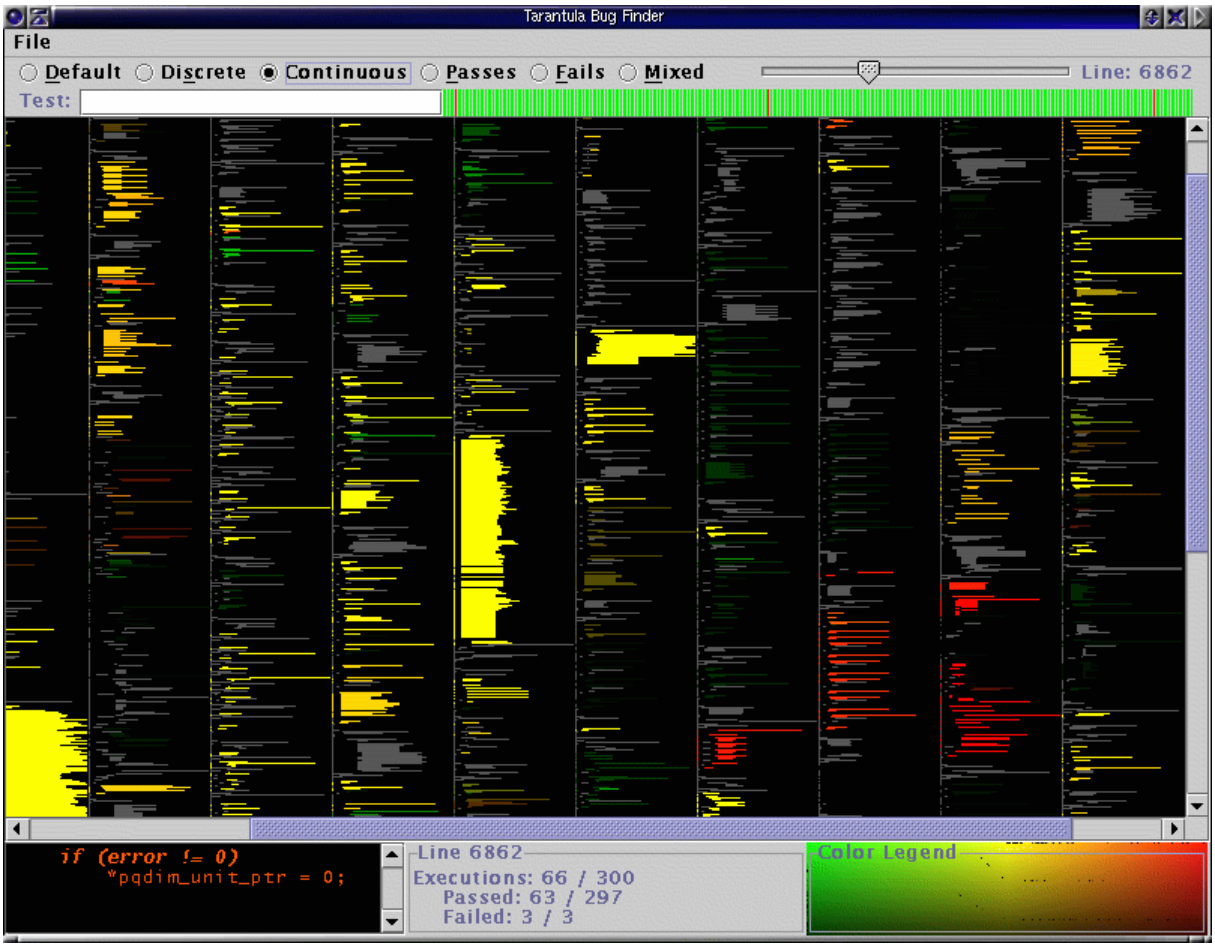


Figura 3.3: Representação das linhas de código pelo Tarantula. Figura obtida de [Jones et al. 2002].

3.4 CodeCrawler

Em 1998, Lanza [Lanza 1999] propôs um sistema de visualização de software caracterizado como visão polimétrica. Em 2003, utilizou este conceito para implementar e aprimorar um sistema chamado CodeCrawler, possibilitando a visualização baseada em hierarquia, conforme ilustrado na Figura 3.4 [Lanza 2003]. O CodeCrawler é baseado em cinco critérios que direcionam a visualização: posição no eixo X, posição no eixo Y, altura, comprimento e cor dos objetos gráficos. Essas características estão associadas a algumas métricas e atributos do software, tais como o pacote em que a classe é criada, quantidade de linhas de código, quantidade de métodos e quantidade de atributos das classes.

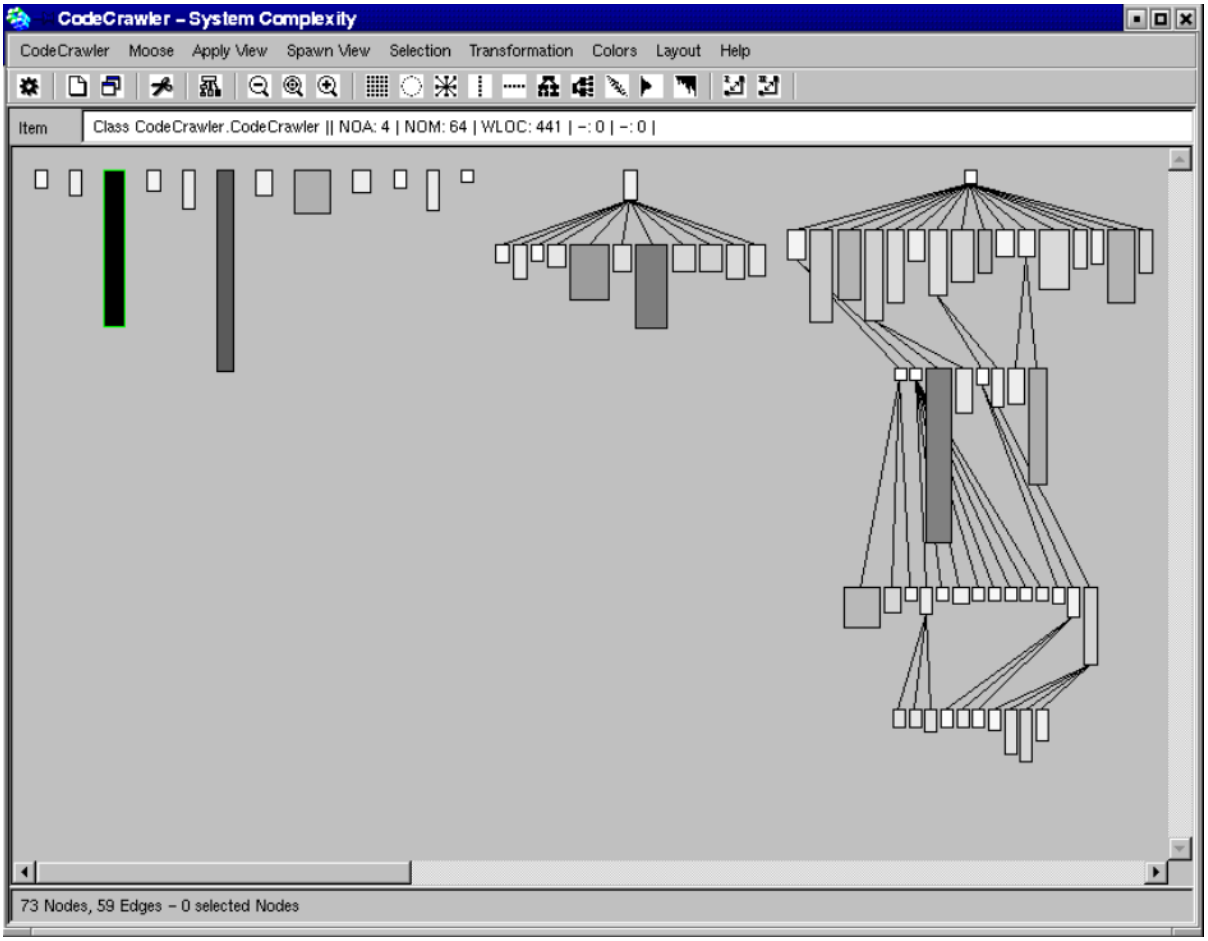


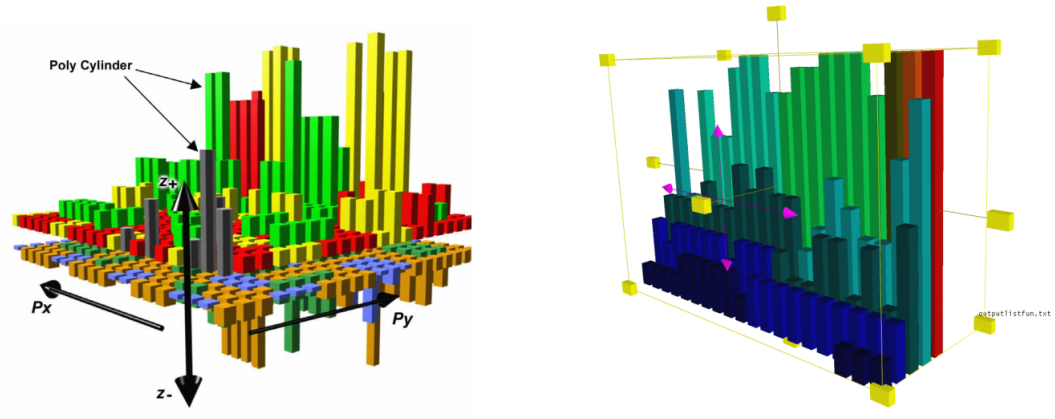
Figura 3.4: Visualização de um sistema pelo CodeCrawler. Figura obtida de [Lanza 2003].

3.5 sv3D

O sv3D (*source viewer 3D*) estende a metáfora de visualização do SeeSoft. O objetivo do sv3D é incrementar a experiência do usuário perante o sistema, aprimorando a facilidade de interação. A visão 2D do SeeSoft é transformada em uma visão 3D, conforme ilustrado na Figura 3.5. A visualização transforma cada arquivo em um conteiner, que por sua vez possui vários policilindros. Cada policilindro representa uma linha de código. O usuário pode rotacionar e ampliar ou reduzir a visualização do sistema [Marcus et al. 2003].

3.6 Solar System

O Solar System é um modelo que realiza uma metáfora com o sistema solar para representar classes e métricas de softwares desenvolvidos sob o paradigma orientado a objetos. Essa visualização representa o software como uma galáxia composta por vários sistemas solares, em que cada sistema solar representa um pacote. A estrela central é o pacote e os planetas que estão em órbita são as classes que compõem o pacote em questão. A profundidade da órbita indica o grau de profundidade da classe, por meio da métrica



(a) (b)

Figura 3.5: Visualização de código-fonte pelo sv3D. (a) Representação policilíndrica. (b) Representação de funções onde a cor indica a quantidade de chamadas para aquela função e a altura representa o tempo de execução da função. Figuras obtidas de [Marcus et al. 2003].

DIT. A cor representa o tipo do dado (as classes são azuis e as interfaces são vermelhas) e o tamanho do planeta é definido de acordo com a quantidade de linhas de código (LOC) das classes [Graham et al. 2004,Caserta e Zendra 2011]. A Figura 3.6 ilustra a visualização de um software no modelo de sistema solar.

Entretanto, alguns problemas nesse modelo podem ser identificados. A ausência de ligações hierárquicas entre as classes impede que esse tipo de informação seja extraída da visualização. Outro problema é a interpretação da profundidade da árvore de herança, já que as classes não possuem a ligação de super/sub classes.

3.7 TraceCrawler

O TraceCrawler é uma extensão do CodeCrawler [Lanza 1999] para representar visualmente informações provenientes de traços de execução de um sistema. Itens como instanciação de objetos e envio de mensagens servem como dados de entrada para geração da visualização. A Figura 3.7 mostra a visualização de um software utilizando o TraceCrawler. Os primeiros blocos representam as classes e os blocos superiores representam os objetos instanciados por aquela classe. As linhas em escala de cinza são as relações de herança e as linhas em vermelho representam a troca de mensagem entre os objetos [Greevy et al. 2006]. A visualização do TraceCrawler é classificada como dinâmica, onde são representados aspectos relacionados a execução e comportamento do software.

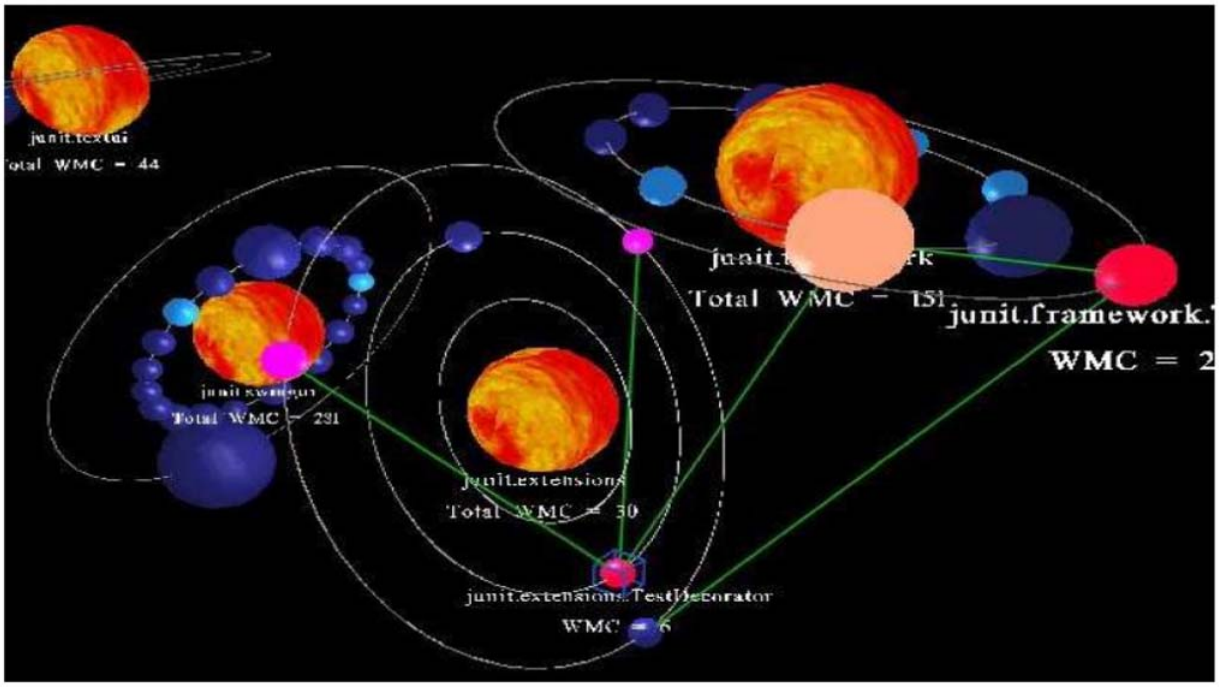


Figura 3.6: Visualização de um software utilizando a metáfora do sistema solar. Figura obtida de [Caserta e Zendra 2011].

3.8 CodeCity

O CodeCity é um sistema de visualização de software que utiliza a metáfora de uma cidade, em que a estrutura física da metrópole faz alusão a estrutura lógica de um software. Algumas características do paradigma de programação orientada a objetos são exibidas durante a visualização, tais como, número de atributos, número de métodos. Ainda existe a possibilidade de destacar classes representadas como *God* e *Brain*. Classes caracterizadas como *God* tendem a concentrar grande parte da inteligência do sistema, que possuem alta complexidade, baixa coesão entre as classes e uma grande quantidade de acessos em outras classes. Isto viola o padrão de projeto do paradigma de orientação a objetos onde cada classe deve ter uma única responsabilidade. Portanto, classes *god* são de díficil manutenção e entendimento. Já classes *Brain* são próximas das classes *god*, só que não utilizam tantos dados vindos de outras classes e são um pouco mais coesas. As classes tratadas como *God* + *Brain* são classes que possuem as duas caraterísticas, isto é, efetuam várias ações utilizando informações contidas em si, em outras classes e acumulam uma vasta quantidade de regras de negócio do sistema [Wettel et al. 2011]. A detecção dessas características citadas anteriormente facilitam a descoberta de classes complexas [Olbrich et al. 2010,Wettel et al. 2011].

Nesse trabalho, os autores sugerem uma abordagem de avaliação empírica de um software utilizando o modelo gráfico sugerido e desenvolvido por eles. Um ponto de destaque do CodeCity é um quadro dos trabalhos correlatos, que compara a aplicabilidade da proposta do CodeCity (onde o software foi testado também na área industrial, ampliando

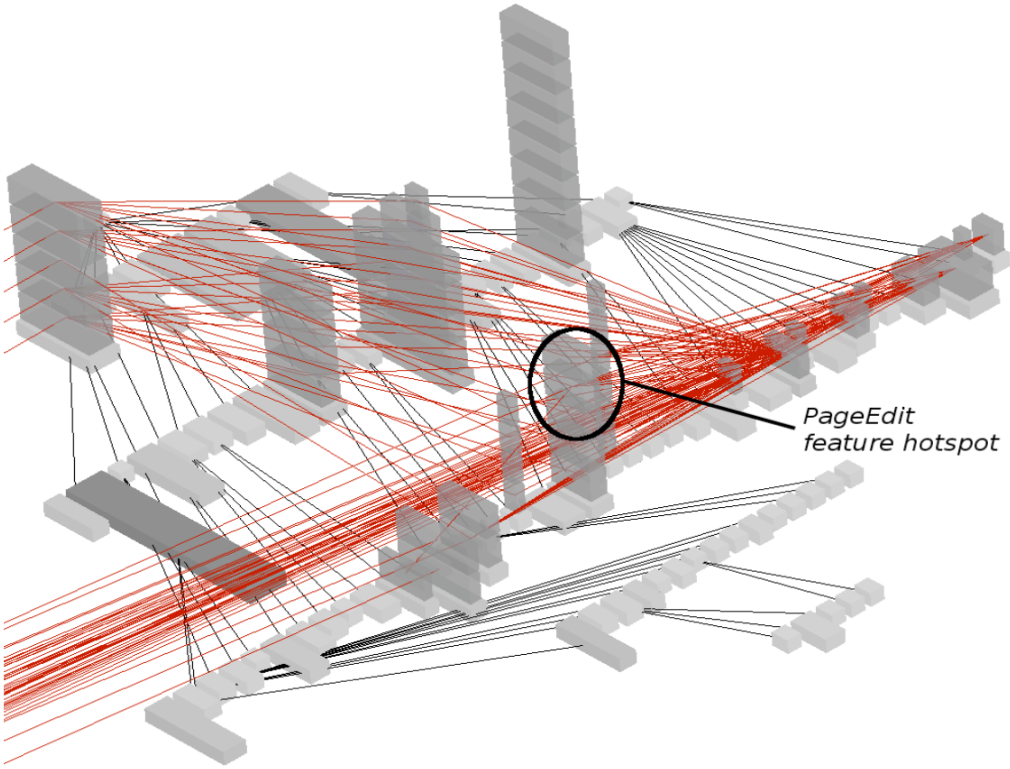


Figura 3.7: Representação do modelo TraceCrawler. Figura obtida de [Greevy et al. 2006].

o alcance do trabalho) com outros trabalhos. Ao final do trabalho, os autores retratam as estatísticas obtidas em pesquisas com profissionais da área acadêmica e desenvolvedores experientes da área industrial, exibindo os ganhos conseguidos após a utilização do CodeCity na análise de corretude e no tempo de conclusão da análise do sistema.

Entretanto, uma limitação do trabalho é o tamanho do software a ser avaliado. Apesar de conseguir resultados importantes na visualização de softwares de grande tamanho (segundo análise dos próprios autores), o tamanho do software define a qualidade da compreensão. Essa qualidade da visualização é inversamente proporcional ao tamanho do software. Outra característica do trabalho é que outras métricas, além das que indicam possíveis anomalias, também são avaliadas. A visualização de classes *God* e *Brain* são indicadores de anomalias, mas as demais (número de atributos, número de métodos e número de linhas sem comentários - KLOC) não retratam características específicas de anomalias de software [Wettel et al. 2011]. Para exemplificar o funcionamento do CodeCity, é exibida a visualização do software JDK (Java Development Kit), versão 1.5, na Figura 3.8.

3.9 ExtraVis

O ExtraVis é um sistema de visualização de software que abrange as áreas de visualização estática e dinâmica. Nesse trabalho, os autores desenvolveram um protótipo que,

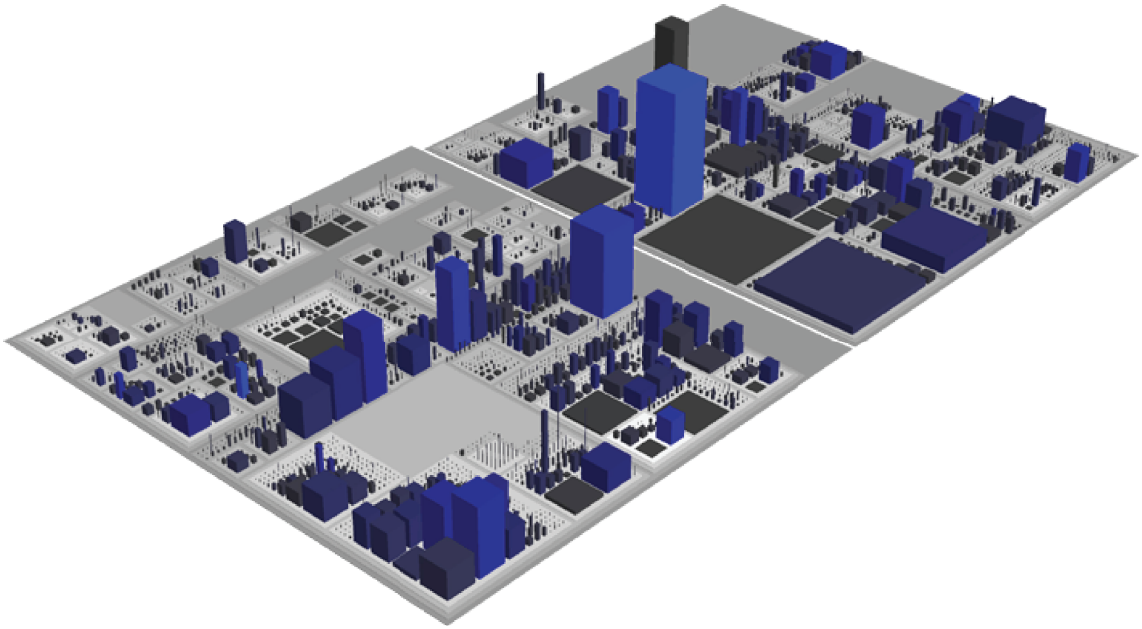


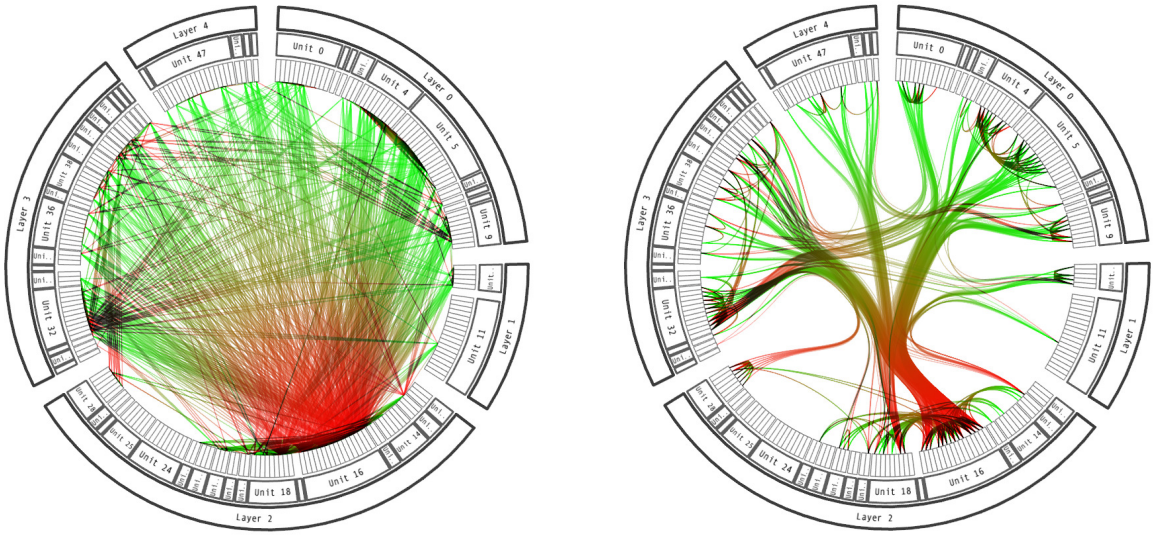
Figura 3.8: Visualização de um sistema pelo CodeCity. Figura obtida de [Wettel et al. 2011].

além de mapear estruturalmente os pacotes, classes e métodos do software, mapeia também a troca de mensagens entre eles. Para melhorar a visualização, é utilizado o conceito de *Hierarchical Edge Bundles* (HEB), proposto por [Holten 2006]. O HEB é uma técnica de visualização para unir uma quantidade de arestas que possuem posições próximas. O objetivo é aprimorar a visualização de grandes quantidades de arestas, agrupando-as, mas mantendo suas características principais [Holten 2006]. As Figuras 3.9a e 3.9b mostram o conceito HEB não aplicado e aplicado, respectivamente. A cor das arestas define o tempo de chamada do método, onde a cor verde representa chamadas mais antigas e a cor vermelha mostra as chamadas mais recentes dos métodos. Pela sua aplicabilidade e aceitação, a técnica HEB é sugerida como um dos itens de trabalhos futuros (Capítulo 6.1 para essa dissertação).

É possível, de maneira implícita, observar uma anomalia de software, em que um método é chamado diversas vezes durante a execução por vários outros métodos. Esta anomalia, segundo [Chidamber e Kemerer 1994] pode ser obtida utilizando a métrica RFC [Holten et al. 2007]. Assim como nos outros trabalhos citados anteriormente, o tamanho do software interfere na visualização do sistema.

3.10 code_swarm

O code_swarm é um software para a visualização de informações oriundas de sistemas controladores de versões, com o objetivo de mostrar as interações realizadas pelos desenvolvedores durante a produção de um sistema. É utilizada a técnica de visualização conhecida como árvores orgânicas, que busca aprimorar a visualização de dados proveni-



(a) (b)

Figura 3.9: Tipos de visões do ExtraVis. (a) Exemplo do ExtraVis **sem** a aplicação do conceito de HEB. (b) Exemplo do ExtraVis **com** a aplicação do conceito de HEB. Figuras obtidas de [Holten et al. 2007].

entes de fontes dinâmicas de informação. Propriedades como o crescimento e resposta a entrada de novos dados norteiam esse modelo [Fry 2000]. As informações que alimentam a visualização são provenientes de sistemas controladores de versões, tais como SVN, CSV e Mercurial. Um ponto de destaque do trabalho é o histograma que é criado no canto esquerdo da tela, juntamente com a visualização em forma de árvore orgânica. O modelo de visualização foi bem aceito pela comunidade, segundo os autores [Ogawa e Ma 2009]. A Figura 3.10 ilustra os dois modelos de visualização citados anteriormente. Outro exemplo interessante, que retrata o processo de construção do jogo *Little Planet 2*, está disponível no site YouTube, por meio do caminho *http://www.youtube.com/watch?v=jVPZv631cqE*.

3.11 Gource

O Gource utiliza a técnica de visualização orgânica, assim como o code_swarm, para a geração de seu modelo [Caudwell 2010]. É um aplicativo que possibilita a visualização da etapa de gerência de configuração por meio de uma linha do tempo, onde todas as atividades efetuadas dentro do projeto são exibidas de forma gráfica, mostrando as modificações, criações e remoções ao longo do tempo. O Gource é um projeto *Open Source* voltado para a leitura de arquivos criados pelo aplicativo de repositório de dados SVN (*Subversion* [Collins-Sussman et al. 2004]).

O Gource proporciona a visualização do histórico do controle de versão por meio da árvore de pastas do projeto. Os usuários que efetuam quaisquer tipos de operação dentro do projeto são representados graficamente e suas ações são mapeadas por meio das cores

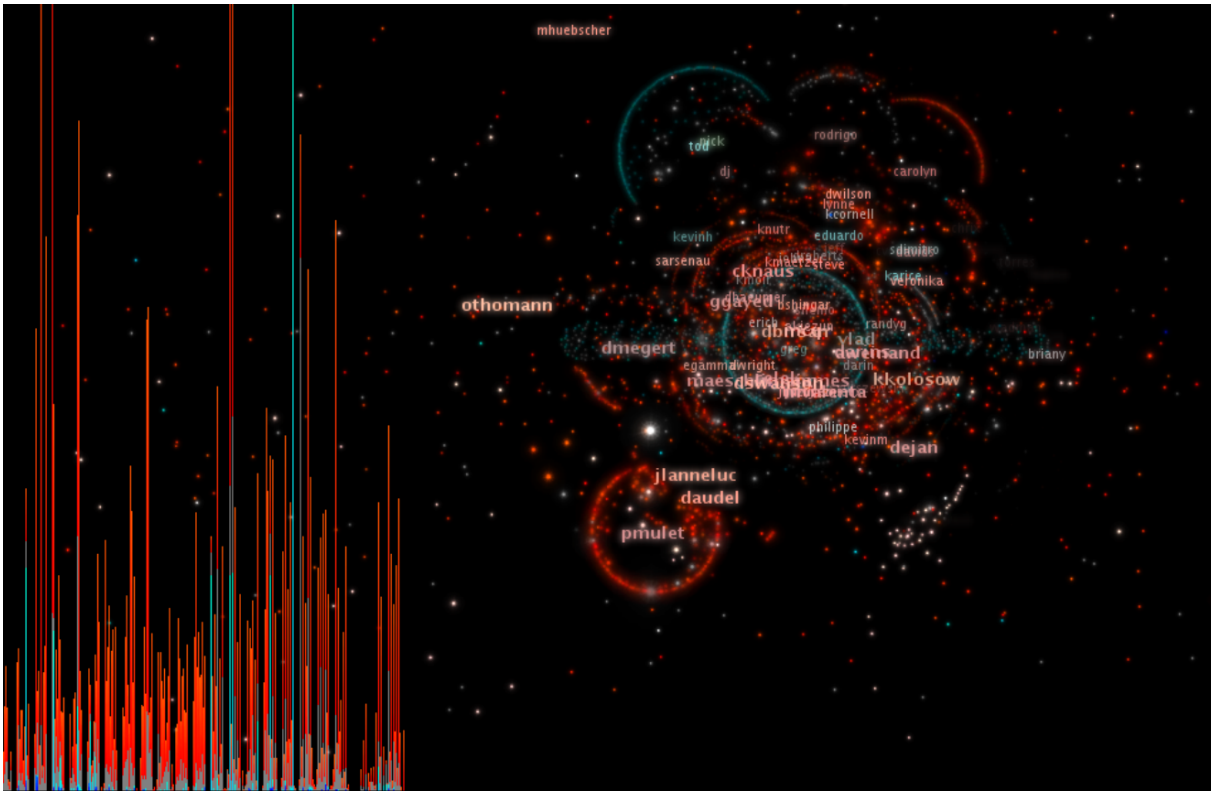


Figura 3.10: Visualização do projeto Eclipse no code_swarm. Figura obtida de [Ogawa e Ma 2009].

vermelho (exclusão), laranja (modificação) e verde (inclusão) assim que o COMMIT[1] é realizado [Caudwell 2010]. A Figura 3.11 ilustra o aplicativo Gource sendo aplicado no projeto *Mozilla Firefox*. Em outro exemplo, é ilustrada a visualização do processo de desenvolvimento da linguagem de programação Python, desde 1990, disponível em *http://www.youtube.com/watch?v=aPk1BqK8zzI*.

Uma das diferenças entre Gource e code_swarm está na forma como as modificações realizadas pelo usuário são exibidas. No code_swarm o objeto gráfico que representa o usuário movimenta-se menos do que no Gource. Outra diferença é que no code_swarm os arquivos não possuem a representação das hierarquias entre pastas e arquivos. Já no Gource esta representação acontece, conforme Figura 3.11.

3.12 Manhattan

O aplicativo Manhattan segue a metáfora de cidades proposta no trabalho de [Wettel e Lanza 2007], gerando a visualização do software em desenvolvimento para o usuário em tempo real. Conforme as modificações vão acontecendo no código-fonte, a visualização é atualizada [Lanza et al. 2013]. Este conceito busca diminuir o tempo de descoberta de problemas, reduzindo o custo de manutenção. A Figura 3.12 ilustra o funcionamento

[1]Atividade de registrar a operação dentro do sistema de gerenciamento de versão [Collins-Sussman et al. 2004]

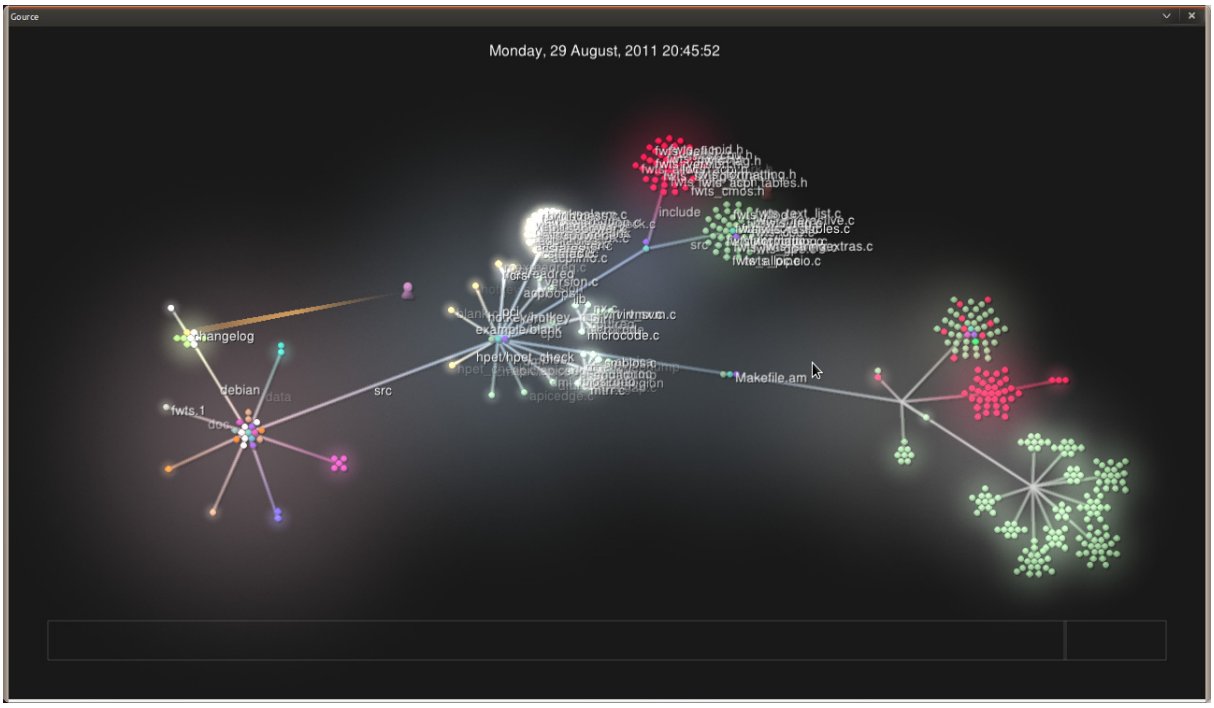


Figura 3.11: Exemplo do mapeamento feito pelo Gource no projeto Mozilla Firefox. Figura obtida de [Caudwell 2010].

do Manhattan onde o usuário possui o aplicativo de desenvolvimento em um monitor e a visualização em outro. Apesar de utilizar a metáfora do CodeCity, o Manhattan foi reescrito para funcionar como um *plugin* da ferramenta Eclipse[2].

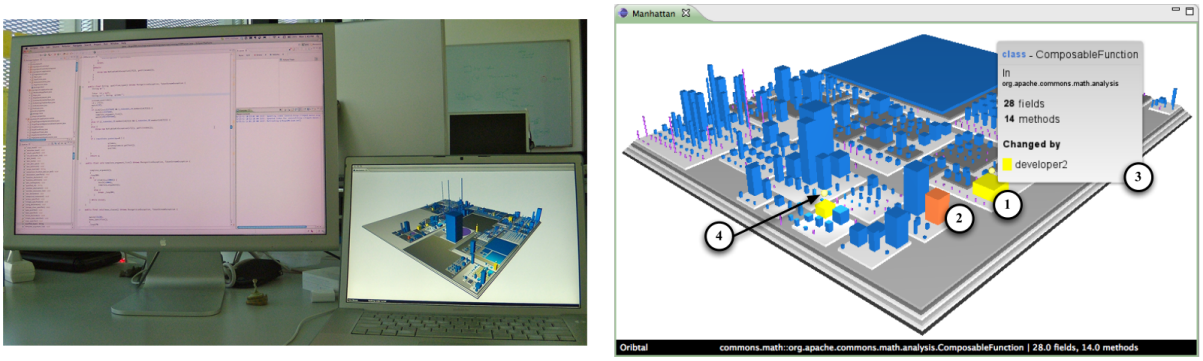


Figura 3.12: Utilização do Manhattan em tempo real. Figura obtida de [Lanza et al. 2013].

3.13 SArF Map

O SArF Map (***S**oftware **A**rchitecture **F**inder*) é um aplicativo de visualização que busca analisar critérios de arquitetura do software, juntamente com valores de métricas. Os autores utilizaram como dados de entrada informações produzidas de outros trabalhos para separar o sistema em características (*features*). Feito isto, a visualização é orientada

[2]www.eclipse.org

com base nas características e nas métricas das classes. A metáfora acontece de forma parecida com os trabalhos de [Lanza 2003,Wettel e Lanza 2007,Lanza et al. 2013], utilizando o conceito de cidades, quadras e prédios. A principal diferença entre SArF e CodeCity é que no SArF as cidades representam características e no CodeCity representam pacotes [Kobayashi et al. 2013]. A Figura 3.13 exibe o funcionamento do SArF aplicado ao software Weka.

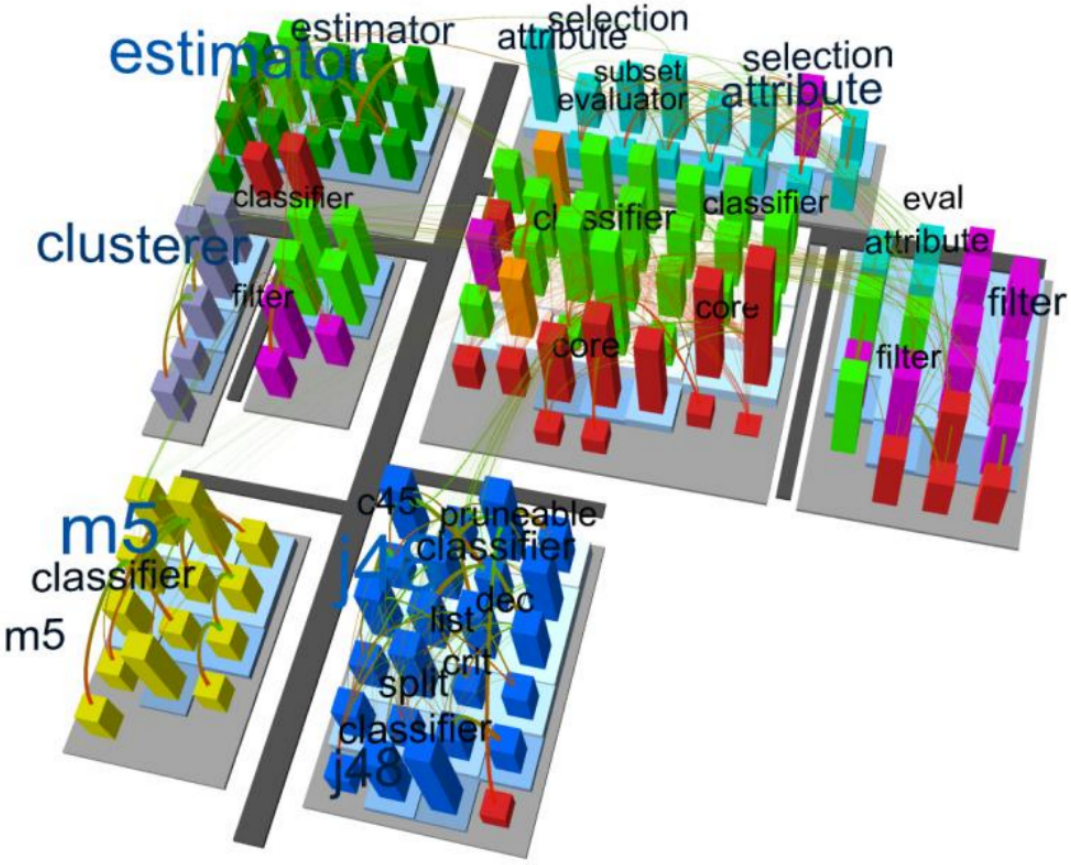


Figura 3.13: Leitura do Weka pelo SArF. Figura obtida de [Kobayashi et al. 2013].

3.14 Considerações Finais

A Tabela 3.1 expõe os trabalhos correlatos identificados durante a construção deste trabalho. As características apresentadas na tabela são descritas a seguir.

1. **Estrutura:** Visualização com base na estrutura do código-fonte (exemplo: herança entre classes).
2. **Métrica:** Visualização com base nos valores das métricas do sistema (exemplo: altura baseada em linhas de código (LOC); cor baseada na quantidade de métodos (NOM - *Number of Methods*));
3. **Comportamento:** Visualização baseada em instanciação de objetos e/ou troca de mensagens.

4. **Evolução:** Visualização baseada nas mudanças ocorridas durante o desenvolvimento do software.

A coluna "# citações" representa a quantidade de citações que os artigos obtiveram de acordo com o site *scholar.google.com*, até a data de 18/12/2013. Esta métrica tem como objetivo dar uma informação superficial sobre a aceitação do estudo na comunidade. É válido ressaltar que outras informações podem interferir na análise dos valores desta métrica, como a data de realização do estudo.

É importante observar que grande parte dos trabalhos utilizam a técnica de ampliar ou reduzir o campo de visão para melhorar a visualização [Storey 2001,Marcus et al. 2003,Jones et al. 2002,Wettel e Lanza 2007,Kobayashi et al. 2013,Lanza et al. 2013]. Esta característica é importante para incrementar a interação com o usuário e facilitar a atividade de interpretação do sistema.

Após a análise dos trabalhos correlatos, percebe-se que a visualização de software possui uma dificuldade comum: a visualização de grandes softwares. Por isso, o modelo de visualização presente neste trabalho aborda esse problema, tentando minimizar a importância de classes que não possuem um valor elevado das métricas de CK e enfatizando as que poderão gerar erros no sistema. Estudos, conforme citado no capítulo 4, mostram que classes com altos valores de suas métricas possuem uma propensão a erros maior do que as demais.

A visualização de software depende de uma série de fatores para que o objetivo de auxiliar as atividades de compreensão de software seja realizado com êxito. Quanto mais próxima da realidade for a visão do usuário diante do sistema, maior será sua familiarização com o sistema.

O modelo proposto neste trabalho, detalhado no capítulo 5 e também relacionado na Tabela 3.1, busca facilitar o entendimento de programas desenvolvidos em linguagens orientadas a objetos, dando ênfase na herança (DIT e NOC), acoplamento (CBO e RFC), coesão (LCOM) e complexidade (WMC) de classes. A intenção é que estes três itens despertem a curiosidade do usuário para indicar que determinadas classes possuem valores de suas métricas que estão discrepantes das demais em algum dos quesitos, com base nas métricas citadas anteriormente.

Além disso, a característica do modelo em evitar sobreposição de objetos gráficos pode deixar a visualização mais clara para o usuário. Quando os nós não ficam sobrepostos, é possível identificá-los mais claramente, aumentando a assistência da visualização.

Tabela 3.1: Trabalhos correlatos de visualização de software e comparação com a proposta apresentada neste trabalho.

ID	Software	Referência	Ano	Tipo de Visualização	Características*	Dados de Entrada	3D	# citações
S1	SeeSoft	[Eick et al. 1992]	1992	Estática	1;	Dados do Controlador de Versão	Não	725
S2	SHriMP	[Storey 2001]	2001	Estática	1;	Java	Não	79
S3	Tarantula	[Jones et al. 2002]	2002	Dinâmica	3;	C e Casos de Teste	Não	539
S4	CodeCrawler	[Lanza 2003]	2003	Estática	1;2;	Modelo FAMIX	Sim	98
S5	sv3D	[Marcus et al. 2003]	2003	Estática	1;2;	XML	Sim	31
S6	Solar System	[Graham et al. 2004]	2004	Estática	1;2;	Java	Sim	32
S7	TraceCrawler	[Greevy et al. 2006]	2006	Dinâmica	1;3;	Modelo FAMIX	Sim	43
S8	CodeCity	[Wettel e Lanza 2007]	2007	Estática	1;2;	Modelo FAMIX	Sim	103
S9	ExtraVis	[Holten et al. 2007]	2007	Dinâmica	1;2;3;	RSF (RigiStandardFormat)	Sim	75
S10	Code Swarm	[Ogawa e Ma 2009]	2009	Evolutiva	4;	Dados do Controlador de Versão	Não	39
S11	Gource	[Caudwell 2010]	2009	Evolutiva	4;	Dados do Controlador de Versão	Não	10
S12	SArF Map	[Kobayashi et al. 2013]	2013	Estática	1;2;	Weka database	Sim	–
S13	Manhattan	[Lanza et al. 2013]	2013	Estática	1;2;4;	Java	Sim	–
–	SUVSoft	–	–	Estática	1;2;	C#; Vb.NET; J#; XML	Sim	–

Capítulo 4

Conjunto de Métricas CK: uma revisão sistemática

Este capítulo mostra uma revisão sistemática sobre trabalhos que utilizaram as métricas de Chidamber e Kemerer (métricas CK) [Chidamber e Kemerer 1994] como fonte de informação para a realização de atividades de Engenharia de Software, tais como predição a propensão a erros, impacto da refatoração nas métricas de um software, facilidade de reuso caixa-branca e análise de índices de manutenibilidade ou qualidade. Dessa revisão surgiram dois estudos: (i) um levantamento estatístico dos valores das métricas de CK utilizadas nos artigos e uma sugestão de valores que sejam discrepantes dos demais, chamados nesse estudo de "anomalias" e (ii) uma lista de estudos que utilizaram as métricas CK com eficiência na realização de estudos nas áreas citadas anteriormente.

O objetivo é fornecer informações sobre quais métricas foram utilizadas com sucesso em outros estudos e sugerir valores dessas métricas que mereçam atenção dos desenvolvedores nas atividades de Engenharia de Software.

O presente estudo é baseado no artigo "*Detection of Software Anomalies using Object-Oriented Metrics*", publicado na conferência *16th International Conference on Enterprise Information Systems - ICEIS 2014* [Juliano et al. 2014].

4.1 Introdução

O processo de desenvolvimento de sistemas é uma tarefa árdua e complexa, onde a criatividade e rigor no cumprimento de metas (prazos e custos) precisam ser balanceadas ao longo do processo. Essa dificuldade em desenvolver, implantar e manter software é bem reconhecida e tem sido amplamente estudada [Brooks 1995,Glass 1999,Berry 2004,Boehm 2006,Wirth 2008]. Não é incomum que os prazos e os custos estipulados inicialmente sejam extrapolados e que o controle das mudanças de requisitos se torne falho. Esses problemas podem resultar em falhas no software, que por sua vez podem trazer impactos

negativos às instituições, gerando, entre outros problemas, perdas financeiras [Bar-Yam 2003,Charette 2005].

Para tentar amenizar os problemas citados anteriormente, algumas atividades podem ser realizadas durante o ciclo de vida dos softwares, com o intuito de diminuir suas consequências. Dentre elas, pode-se citar a utilização de métricas como fonte de dados para a tomada de decisão ou para a geração de novas informações. Um exemplo disso é a associação de métricas de software à análise de propensão a erros. Nessa análise, os valores das métricas são correlacionados com a quantidade de erros encontrada nas classes medidas. Então, são procurados valores de métricas que possam predizer futuros erros. Com tal atividade, espera-se reduzir a quantidade de falhas no software em futuras implementações [Fenton e Pfleeger 1998]. Outras características de software como manutenibilidade, testabilidade e compreensibilidade também podem ser medidas usando métricas de software [Olbrich et al. 2009]. Portanto, a utilização e análise de métricas durante o processo de desenvolvimento de softwares pode aprimorar os produtos gerados.

Em [Johari e Kaur 2012] foi analisada a aplicabilidade de métricas orientadas a objetos na estimativa do esforço de manutenção de softwares. Foi conduzido um estudo empírico, utilizando softwares de código aberto, para verificar a aplicabilidade das métricas em estimar o esforço necessário para a realização de revisões nas classes de um sistema. As métricas WMC, RFC e CBO foram eficientes ao predizer a propensão a erros em classes.

Em [Dallal 2012], foram estudadas as habilidades de várias métricas de qualidade, consideradas individualmente e combinadas para prever classes que precisarão de refatoração. Os autores concluíram que a métrica LCOM foi usada com sucesso para tal atividade.

A revisão sistemática em Engenharia de Software proposta por [Kitchenham 2010] descreveu a importância de avaliar softwares usando métricas. De acordo com os autores, embora haja uma vasta quantidade de pesquisas relacionadas às métricas de software, futuras pesquisas devem aprimorar as metodologias empíricas antes de conseguir responder, de maneira assertiva, questões empíricas relacionadas à utilização de métricas no processo de desenvolvimento de software.

Em outra revisão sistemática, proposta por [Radjenovi et al. 2013], métricas orientadas a objetos (49%) foram usadas frequentemente duas vezes mais do que métricas tradicionais (27%) e métricas de processo (24%). Outra análise interessante é que o conjunto de métricas CK é o mais popular entre as métricas orientadas a objeto.

Além da análise das métricas, é importante ressaltar também que vários itens podem interferir na construção do sistema e nos valores de suas métricas. De acordo com [Subramanyam e Krishnan 2003], a relação entre métricas e defeitos de software varia dependendo da linguagem de programação. Portanto, avaliações de métricas de software tornam-se sensíveis para linguagens de programação específicas.

Nesse contexto, as principais contribuições desse estudo são: (i) a revisão de trabalhos experimentais baseados nas métricas CK e uma sugestão de "anomalias de software"

baseada em estudos empíricos e análise estatística e (ii) um estudo, baseado em artigos científicos, sobre quais métricas foram eficientes na realização de algumas tarefas de Engenharia de Software, tais como predição de propensão a erros, análise do impacto de refatoração ou facilidade de reuso caixa-branca.

Para a obtenção das informações necessárias para a confecção desse estudo, somente artigos que utilizaram as métricas CK foram considerados. As métricas CK foram escolhidas porque são bem difundidas no meio acadêmico e são frequentemente aplicadas em estudos [Radjenovi et al. 2013]. As métricas lidas de softwares desenvolvidos em Java e C++, ambos utilizados com frequência pelos setores acadêmico e profissional [Tiobe 2013], foram aplicadas. São descritas, para cada métrica, uma base de dados empírica com valores médio e desvio padrão. Com esses valores, é possível inferir estatisticamente o conceito de anomalia proposto nesse trabalho.

O conteúdo desse capítulo está organizado da seguinte forma: na Seção 4.2 é descrita a metodologia; na Seção 4.3 são expostos os resultados do presente trabalho; na Seção 4.4 os resultados são discutidos e comparados aos demais trabalhos e na Seção 4.6 as considerações finais são apresentadas.

4.2 Metodologia

Com o objetivo de estabelecer o que será considerado como anomalia, de acordo com as métricas CK, foi realizada uma pesquisa entre diversas revistas científicas e anais de conferências por artigos descrevendo resultados na análise de software que utilizaram as métricas CK. Os dados foram extraídos dos artigos escolhidos, tabulados e analisados utilizando métodos estatísticos. Por fim, foi criada uma classificação das métricas baseada em seus valores e uma análise sobre o grau de importância das métricas CK para a realização de alguns estudos. Esses passos são descritos a seguir e esquematizados na Figura 4.1.

No primeiro passo (Etapa 1), foi realizada a escolha das revistas científicas e conferências que nortearam este estudo, utilizando as seguintes base de dados:

- ACM Digital Library;
- IEEE Xplore;
- ScienceDirect - Elsevier;
- SpringerLink.

O critério para a escolha desses veículos foi baseado no escopo dessas publicações, que incluem tópicos como manutenção, refatoração, qualidade e métricas de software, e também porque estes são considerados de excelente qualidade na comunidade de Engenharia de Software.

As revistas científicas escolhidas foram:

- EMSE (*Empirical Software Engineering*);
- Information Sciences;
- IST (*Information and Software Technology*);
- JSS (*Journal of System and Software*);
- SCP (*Science of Computer Programming*);
- TOSEM (*ACM Transactions on Software Engineering Methodology*);
- TSE IEEE (*IEEE Transactions on Software Engineering*).

As conferências escolhidas foram:

- CSMR (*European Conference on Software Maintenance and Reengineering*);
- ECOOP (*European Conference on Object-Oriented Programming*);
- ICPC (*International Conference on Program Comprehension*).
- ICSE (*International Conference on Software Engineering*);
- ICSM (*International Conference on Software Maintenance*);
- OOPSLA (*International Conference on Object Oriented Programming*);
- WCRE (*Working Conference on Reverse Engineering*);

Para limitar a abrangência da pesquisa, somente estudos realizados durante os anos de 2003 à 2013 foram selecionados. Esse critério foi escolhido para deixar o presente estudo com informações de artigos atuais.

As palavras-chave utilizadas foram:

1. “software” AND “metrics”;
2. “CK” AND “metrics”;
3. “design” AND “defects”;
4. “proneness” AND “error”.
5. “proneness” AND “fault”.

Assim que os artigos foram selecionados seguindo os critérios já expostos (Etapa 1), foi aplicado um filtro manual, com base nos títulos dos artigos (Etapa 2). Nessa etapa, 48 estudos foram selecionados. Na Etapa 3 foi realizada a leitura dos resumos e os que não se encaixavam na proposta desse estudo foram descartados. Dos 48 estudos, 16 foram escolhidos.

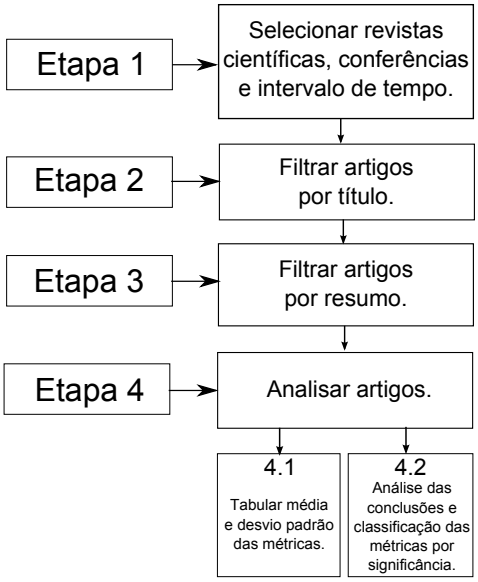


Figura 4.1: Etapas do processo de seleção e análise dos estudos.

Após a escolha dos artigos, a média (AVG) e o desvio padrão (STDEV) de cada uma das métricas dos softwares presente nos estudos selecionados foi tabulada (Etapa 4.1). Dos 16 estudos selecionados, 14 apresentavam os valores de média e desvio padrão para os softwares analisados. Nessa atividade, valores discrepantes dos demais foram desconsiderados. Foram considerados como discrepante valores que estavam nas extremidades (25% menores e 25% maiores). Esse tipo de média é conhecido como medida do intervalo interquartil (IQM - *Interquartile Mean*) [Huck 2012].

Assim que os valores foram obtidos da Etapa 4.1, foi criada uma classificação das métricas, baseada em um estudo proposto por [Lanza e Marinescu 2006].

Para cada métrica CK (DIT, NOC, CBO, RFC, LCOM e WMC), foram definidas quatro classes de valores: **baixo**, **normal**, **alto** e **anomalia**. São classificados como **baixo** os valores das métricas que são menores ou iguais ao resultado da subtração do IQM da média (AVG) e IQM do desvio padrão (STDEV). Os valores **normais** são superiores aos valores baixos e inferiores aos valores altos. Os valores das métricas que estiverem acima da classificação **baixo** e abaixo da soma do IQM da média (AVG) e IQM do desvio padrão (STDEV) são consideradas como **alta**. Por fim, é considerado como anomalia os valores das métricas que são iguais ou superiores a soma do IQM da média (AVG) e IQM do desvio padrão (STDEV) acrescido de 30%. Na proposta de [Lanza e Marinescu 2006], são utilizados somente os valores de média e desvio padrão (sem o uso de IQM) e a classificação de anomalia é feita pela soma entre média e desvio padrão acrescido de 50%. A Figura 4.2 ilustra a classificação das métricas.

O modelo proposto por [Lanza e Marinescu 2006] é usado em algumas atividades de Engenharia de Software, tais como critérios visuais (cor, e tamanho, por exemplo) para modelos de visualização de software (CodeCrawler [Lanza 2003] (Seção 3.4) e CodeCity [Wettel e Lanza 2007] (Seção 3.8)) e para definir critérios de análise de reusabilidade

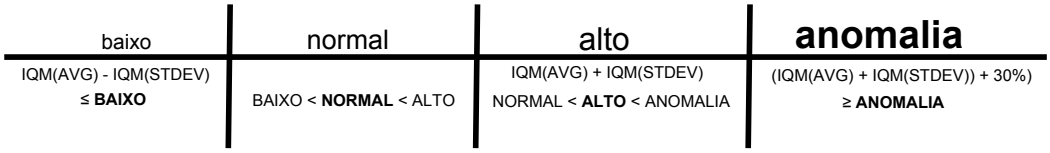


Figura 4.2: Proposta de classificação das métricas.

[Kakarontzas et al. 2012].

Após a classificação das métricas, os resultados dos valores das métricas para os diferentes tipos de linguagem de programação (C++ e Java) são comparados, com o objetivo de encontrar valores diferentes de anomalias, devido as características particulares de cada uma das linguagens.

Um segundo estudo realizado neste capítulo é uma revisão literária sobre quão significativa as métricas de CK são em algumas atividades de Engenharia de Software, como predição de propensão a erros, análise do impacto da refatoração ou facilidade de reuso caixa-branca (Etapa 4.2). Dentre os 48 estudos, 15 apresentaram conclusões sobre a utilização de métricas.

O objetivo é mostrar que os artigos encontrados na revisão sistemática possuem uma relação nos valores que são indicados como propensos a erros e quais métricas foram utilizadas com sucesso. Se existirem valores próximos nas métricas, será possível antecipar algumas atividades para aprimorar o código-fonte, como refatoração ou análise por especialistas humanos do código. Quando uma classe atingir esse valor, ela deverá passar por um processo de avaliação e validação, antes de ser colocada em versões finais do sistema.

4.3 Resultados

O resultado da análise dos artigos selecionados estão sumarizados na Tabelas 4.1 e 4.2, referentes aos artigos que utilizaram softwares desenvolvidos em Java e C++, respectivamente. Essas tabelas mostram a média e o desvio padrão dos valores das métricas CK obtidos dos softwares utilizados como fonte de estudo nos artigos analisados. O valor médio e o IQM desses valores também compõem as tabelas. É importante ressaltar que os softwares estudados são provenientes da indústria ou da academia e código aberto ou proprietário. Não foi feita restrição sobre as características citadas anteriormente para deixar o estudo o mais abrangente possível.

Dos estudos que utilizaram softwares desenvolvidos em Java, oito foram considerados e estão listados na Tabela 4.1.

1. [Subramanyam e Krishnan 2003][1];
2. [Benestad et al. 2006];

[1]Esse estudo analisa softwares desenvolvidos em Java e C++. Ele foi incluído na lista de estudos em Java por possuir os valores da média e desvio padrão de aplicativos desenvolvidos em Java.

3. [Stroggylos e Spinellis 2007];
4. [Shatnawi e Li 2008];
5. [Nair e Selvarani 2011];
6. [Johari e Kaur 2012];
7. [Abuasad e Alsmadi 2012];
8. [Kakarontzas et al. 2012].

No total, esses artigos relatam 20 softwares desenvolvidos em Java. Alguns estudos como [Stroggylos e Spinellis 2007] não utilizaram todas as métricas CK, entretanto eles foram considerados devido a importância de seus resultados. Em [Kakarontzas et al. 2012], existem 29 aplicativos sumarizados como um.

Na Tabela 4.2, seis artigos que avaliaram 21 sistemas desenvolvidos em C++ foram considerados.

1. [Gyimothy et al. 2005];
2. [Zhou e Leung 2006];
3. [Janes et al. 2006];
4. [Olague et al. 2007];
5. [Singh e Kahlon 2011];
6. [Singh e Kahlon 2012].

Após a construção das Tabelas 4.1 e 4.2, outras duas tabelas foram elaboradas, com as classificações das métricas, conforme proposto na Seção 4.2. Nas Tabelas 4.3 e 4.4 são mostrados os valores das métricas já classificados dos softwares desenvolvidos em Java e C++, respectivamente. As classificações **baixo**, **normal**, **alto** e **anomalia** foram feitas usando as fórmulas expostas na Figura 4.2.

Vale ressaltar que os valores gerados nas Tabelas 4.3 e 4.4 receberam dois tratamentos. Na classificação de valores considerados como **baixo**, os valores abaixo de zero foram alterados para zero, uma vez que nenhuma das métricas CK possui valores negativos. Uma segunda modificação foi o arrendondamento dos valores considerados como **anomalia**, visto que as métricas CK possuem somente valores inteiros.

O segundo resultado do presente estudo é uma revisão acerca de quais métricas propostas por [Chidamber e Kemerer 1994] foram significativas na execução de diversas atividades de Engenharia de Software, como a predição da propensão a erros, o impacto da refatoração em métricas ou facilidade de reuso caixa-branca. A Tabela 4.5 exibe quais foram as métricas significativas na execução dos estudos em questão. As métricas receberam uma entre quatro classificações de acordo com a sua utilidade perante ao objetivo do estudo, proposto nos respectivos artigos que são expostas a seguir.

Tabela 4.1: Métricas obtidas dos sistemas desenvolvidos em Java.

Software	**Artigo**	**DIT**		**NOC**		**CBO**		**RFC**		**LCOM**		**WMC**	
		AVG	**STDEV**	**AVG**	**STDEV**	**AVG**	**STDEV**	**AVG**	**STDEV**	**AVG**	**STDEV**	**AVG**	**STDEV**
–	[Subramanyam e Krishnan 2003]	1,02	1			2,94	3,45					12,15	15,84
–	[Benestad et al. 2006]	0,46	0,5	0,46	2,75							6,9	11,2
–	[Benestad et al. 2006]	0,59	0,81	0,59	2,37							7,8	10,3
–	[Benestad et al. 2006]	0	0	0	0							11,4	12,5
–	[Benestad et al. 2006]	0,76	0,54	0,76	3,81							4,9	4,5
Hibernate	[Stroggylos e Spinellis 2007]	1,32		0,36		16,57		64,83		538,46		23,18	
Connector/J	[Stroggylos e Spinellis 2007]	0,3		0,15		12,15						60,65	
Log4J/ckjm	[Stroggylos e Spinellis 2007]	1,16		0,11		6,95		32,68		46,26		12,74	
Log4J/CCCC	[Stroggylos e Spinellis 2007]	1,08		0,13		12,36						18	
Java (Eclipse 2.0)	[Shatnawi e Li 2008]	1,98	1,37	1,39	8,85	9,72	11,36	41,77	71,28	85,26	476,29		
Java (Eclipse 2.1)	[Shatnawi e Li 2008]	1,97	1,37	1,35	8,91	10,25	12,1	45,08	79,18	101,15	560,96		
Java (Eclipse 3.0)	[Shatnawi e Li 2008]	1,59	1,25	1	6,71	8,31	10,11	40,03	71,53	101,78	652,44		
(1) java commercial	[Nair e Selvarani 2011]	2,48	1,159	0,08	0,2768	10,64	5,415	14,32	15,978	0,9	0	8,6	7,88
(2) java commercial	[Nair e Selvarani 2011]	1,65	0,745	0,05	0,2236	15,4	5,335	18,55	18,409	0,86	0	10,15	9,178
(3) java commercial	[Nair e Selvarani 2011]	1,833	0,753	11,83	11,21	0	0	6,5	3,51	0,9	0	7	4,899
(4) java commercial	[Nair e Selvarani 2011]	3,545	1,508	19,82	11,91	0,27	0,47	10,8	6,75	0,92	0	7,455	4,525
(5) java commercial	[Nair e Selvarani 2011]	2,6	1,35	43,5	27,8	0,2	0,42	14	8,33	0,74	0	32	20
jHotDraw	[Johari e Kaur 2012]	1,23	1,61	0,31	1,27	6,18	7,61	36,7	36,5			11,7	11,65
jEdit	[Abuasad e Alsmadi 2012]			0,44	2,351	14,96	24,483	32,72	48,714				
29 Open Source Softwares	[Kakarontzas et al. 2012]	2,46	1,78	0,51	5,8	6,54	10,14	30,9	44,24	174,37	2819,68	10,39	18,53
IQM		**1,44**	**1,07**	**1,30**	**5,11**	**8,44**	**6,64**	**28,87**	**35,75**	**56,93**	**241,38**	**11,71**	**10,65**
Média		1,48	1,05	4,36	6,28	8,34	7,57	29,91	36,77	95,6	501,04	15,31	10,92

Tabela 4.2: Métricas obtidas dos sistemas desenvolvidos em C++.

Software	**Artigo**	**DIT**		**NOC**		**CBO**		**RFC**		**LCOM**		**WMC**	
		AVG	**STDEV**	**AVG**	**STDEV**	**AVG**	**STDEV**	**AVG**	**STDEV**	**AVG**	**STDEV**	**AVG**	**STDEV**
Firefox 1.0	[Gyimothy et al. 2005]	2,89	2,97			6,52	7,95	59,12	86,01	322,37	1754,78	15,88	23,68
Firefox 1.1	[Gyimothy et al. 2005]	2,88	2,98			6,55	7,99	59,23	86,19	324,93	1776,92	15,89	23,77
Firefox 1.2	[Gyimothy et al. 2005]	2,76	3,03			6,92	8,05	61,45	88,53	332,63	1603,85	16,76	24,31
Firefox 1.3	[Gyimothy et al. 2005]	2,75	3,16			7,13	8,14	61,39	89,04	332	1626,51	16,58	24,28
Firefox 1.4	[Gyimothy et al. 2005]	2,75	3,25			6,98	7,98	60,63	90,59	314,54	1609,34	16,03	23,79
Firefox 1.5	[Gyimothy et al. 2005]	2,77	3,25			6,97	7,99	61,1	93,67	319,15	1726,65	15,98	23,96
Firefox 1.6	[Gyimothy et al. 2005]	2,76	3,42	0,92	21,65	6,99	8,1	60,69	92,44	321,7	1752,48	15,99	24,06
A	[Janes et al. 2006]	0,9	1,27	0,27	1,39	11,68	12,17	24,59	25,93	59,44	115,94		
B	[Janes et al. 2006]	0,97	1,12	0,16	0,62	4,17	8,02	17,59	35,36	87,9	234,98		
C	[Janes et al. 2006]	0,97	0,96	0,16	1,2	23,17	24,09	67,46	71,9	1041,3	3197,59		
D	[Janes et al. 2006]	0,25	0,44	0,14	0,63	11,05	18,24	33,23	52,38	256,57	701,88		
E	[Janes et al. 2006]	0,26	0,55	0,05	0,32	17,82	22,67	17,82	22,67	1606,03	3585,45		
–	[Zhou e Leung 2006]	1	1,26	0,21	0,7	8,32	6,38	34,38	36,2	68,72	36,89	17,42	17,45
14R3	[Olague et al. 2007]	0,41	0,69	0,32	1,65	2,33	3,58	13,59	18,28	2,13	2,23	41,8	72,93
15R1	[Olague et al. 2007]	0,42	0,7	0,32	1,55	2,78	4,34	14,22	20,1	2,22	2,38	47,43	86,35
15R2	[Olague et al. 2007]	0,49	1,05	0,25	1,08	2,23	3,93	13,36	18,61	2,42	2,27	43,64	80,3
15R3	[Olague et al. 2007]	0,49	1,05	0,25	1,09	2,28	4,05	13,48	19,15	2,44	2,27	44,33	81,6
15R4	[Olague et al. 2007]	0,47	1,01	0,24	1,07	2,42	4,19	13,24	19,59	2,37	2,12	42,84	80,13
15R5	[Olague et al. 2007]	0,49	0,99	0,25	1,13	2,25	3,86	13,28	19,21	2,47	2,27	44,09	83,48
Firefox 2.0	[Singh e Kahlon 2011]	1,97	1,926	0,97	16	9,26	13,975	25,2	50,049	254,6	2264,38	36,8	125,523
Firefox 3.0	[Singh e Kahlon 2012]	2,14	2,02	1,08	16,4	10,37	14,2	26,91	48,99	225,85	1451,15	40,04	116,3
IQM		**1,44**	**1,73**	**0,34**	**3,42**	**6,63**	**8,57**	**35,09**	**50,7**	**190,01**	**980,29**	**29**	**52,41**
Média		1,47	1,77	0,37	4,43	7,53	9,52	35,81	51,66	280,08	1116,78	29,47	56,99

Tabela 4.3: Valores das classificações baixo, normal, alto e anomalia de softwares desenvolvidos em Java.

	DIT	**NOC**	**CBO**	**RFC**	**LCOM**	**WMC**
Baixo	0	0	2	0	0	1
Normal	1	1–5	3–14	1–64	1–297	2–21
Alto	2	6–7	15–19	65–83	298–387	22–28
Anomalia	≥ 3	≥ 8	≥ 20	≥ 84	≥ 388	≥ 29

Tabela 4.4: Valores das classificações baixo, normal, alto e anomalia de softwares desenvolvidos em C++.

	DIT	**NOC**	**CBO**	**RFC**	**LCOM**	**WMC**
Baixo	0	0	0	0	0	0
Normal	1–2	1–3	1–14	1–85	1–1169	1–80
Alto	3	4	15–19	86–111	1170–1521	81–105
Anomalia	≥ 4	≥ 5	≥ 20	≥ 112	≥ 1521	≥ 106

● A métrica é significativa de acordo com o objetivo do estudo.

◓ A métrica é significativa, mas inversamente proporcional ao objetivo do estudo.

○ A métrica não é significativa de acordo com o objetivo do estudo.

– A métrica não foi levada em consideração no estudo.

Já a Tabela 4.6 mostra, referente aos mesmos artigos da Tabela 4.5, características dos estudos, tais como linguagem em que o software foi escrito, número de softwares analisados, tipo de licença, objetivo do estudo e possíveis *Thresholds* escontrados de acordo com o tema abordado pelo artigo.

4.4 Discussão

Para facilitar o entendimento dos estudos apresentados neste capítulo, um resumo dos artigos utilizados é apresentado a seguir.

Em [Stroggylos e Spinellis 2007] foi analisado o impacto da atividade de refatoração no aprimoramento da qualidade de software. Quatro sistemas de código aberto desenvolvidos em Java foram estudados: Hibernate, Log4J, MySQL e Connector/J. Eles concluíram que, com o passar dos anos e das atividades de refatoração, os valores das métricas RFC, CBO e LCOM aumentaram, deixando o software mais acoplado e menos coeso. Portanto, os autores concluíram que a refatoração não foi positiva para os valores das métricas.

Em [Shatnawi e Li 2008] foi analisada a propensão a erros do projeto Eclipse. Um item de destaque desse artigo é que a análise do software foi feita durante a fase de

Tabela 4.5: Representação das métricas nos objetivos dos estudos.

Seq	Artigo	CBO	DIT	LCOM	NOC	RFC	WMC
S1	[Subramanyam e Krishnan 2003]	●	◓	–	–	–	●
S2	[Gyimothy et al. 2005]	●	●	●	○	●	●
S3	[Benestad et al. 2006]	–	●	–	●	–	●
S4	[Zhou e Leung 2006]	●	○	●	◓	●	●
S5	[Janes et al. 2006]	●	–	○	–	●	–
S6	[Moser et al. 2006]	●	○	○	○	●	●
S7	[Stroggylos e Spinellis 2007]	●	○	●	○	●	●
S8	[Olague et al. 2007]	●	○	●	○	●	●
S9	[Shatnawi e Li 2008]	●	○	–	○	●	●
S10	[English et al. 2009]	●	○	–	○	●	–
S11	[Shatnawi 2010]	●	○	–	○	●	●
S12	[Nair e Selvarani 2011]	●	●	●	●	●	●
S13	[Singh e Kahlon 2011]	●	○	●	–	–	●
S14	[Johari e Kaur 2012]	●	○	○	○	●	●
S15	[Singh e Kahlon 2012]	●	○	●	–	–	●

desenvolvimento, e não nas etapas finais. Os autores também mencionaram que, o quanto antes possível, os valores das métricas devem ser utilizados para nortear o desenvolvimento e reduzir tempo e custo dos projetos. Embora algumas métricas tenham uma significativa associação com a propensão a erros, sua eficiência é questionável, uma vez que outros problemas podem aumentar a propensão a erros, tais como experiência e tempo para conclusão do projeto.

No estudo de [Nair e Selvarani 2011] é discutida a propensão a erros em projetos de software e o seu impacto na qualidade do produto. Para realizar tal avaliação, cinco projetos comerciais foram utilizados e as seis métricas de CK foram analisadas. De acordo com esse trabalho, quando o valor da métrica NOC de uma classe é maior do que cinco, existe 95% de probabilidade da classe possuir algum erro. Quando a métrica DIT é maior do que cinco, existe 81% de probabilidade da classe possuir defeitos e quando a métrica CBO tem seu valor entre 20 e 24, existe de 78% a 98% de probabilidade de uma classe apresentar defeitos. Esses valores são comparados com os estudos obtidos pelo presente estudo, expostos na Seção 4.5.

Em [Benestad et al. 2006], foi investigado o impacto de métricas estruturais, de acoplamento e coesão na avaliação da manutenibilidade de software. O estudo mostrou que a análise das métricas, quando combinadas com outras análises estatísticas, pode ter um resultado mais eficiente do que os métodos separados. Uma das estratégias classificou valores considerados altos para algumas métricas de CK, tais como WMC (>23), DIT (>3) e NOC (>3). Esses valores indicaram maior dificuldade na realização de manutenções no software.

[Subramanyam e Krishnan 2003] analisaram as métricas de CK em softwares utilizados por indústrias e como essas métricas estão associadas com defeitos. Eles concluíram que

Tabela 4.6: Características dos estudos da Tabela 4.5

Seq	Estudo	Linguagem	nº de softwares	Licença	Objetivo	Outliers
1	[Subramanyam e Krishnan 2003]	Java	1	Código Aberto	Propensão a Erros	
2	[Gyimothy et al. 2005]	C++	7	Código Aberto	Identificação de predição de falhas	
3	[Benestad et al. 2006]	Java	4	Código Proprietário	Impacto das métricas na manutenibilidade	DIT > 3 NOC > 3 WMC > 23
4	[Zhou e Leung 2006]	C++	1	Código Proprietário	Impacto das Falhas nas Métricas	
5	[Janes et al. 2006]	C++	5	Código Proprietário	Propensão a erros em softwares de tempo real	
6	[Moser et al. 2006]	Java	1	Código Proprietário	Impacto da refatoração na reusabilidade	
7	[Stroggylos e Spinellis 2007]	Java	4	Código Aberto	Refatoração como Atividade de Melhoria de Qualidade	
8	[Olague et al. 2007]	C++	6	Código Aberto	Propensão a Erros em Metodologias Ágeis	
9	[Shatnawi e Li 2008]	Java	3	Código Aberto	Propensão a Erros	
10	[English et al. 2009]	C++	1	Código Aberto	Predição de Falhas em Softwares de Código Aberto	
11	[Shatnawi 2010]	C++	2	Código Aberto	Análise de valores aceitáveis para métricas com base na identificação de classes com falhas	CBO = 9 RFC = 40 WMC = 29
12	[Nair e Selvarani 2011]	Java	5	Código Proprietário	Propensão a Erros	NOC > 5 DIT > 5 CBO > 20-24 RFC > 160
13	[Singh e Kahlon 2011]	C++	1	Código Aberto	Propensão a Erros & *Bad Smells*	
14	[Johari e Kaur 2012]	Java	1	Código Aberto	Propensão a Erros	
15	[Singh e Kahlon 2012]	C++	3	Código Aberto	Propensão a Erros	

as métricas DIT e CBO estão correlacionadas. Quanto menor for o valor de DIT, maior é o valor de CBO e mais defeitos a classe apresenta. Entretanto, se for isoladamente analisada, a medida DIT é inversamente proporcional a quantidade de defeitos de uma classe (quanto maior for o valor de DIT, menos defeitos a classe possuirá). Eles também concluíram que a associação das métricas com defeitos é diferente de acordo com cada linguagem de programação.

[Johari e Kaur 2012] estudaram o impacto nas métricas causado por modificações durante o ciclo de vida em aplicativos de código-aberto. Foram utilizadas para fazer essa avaliação as métricas de CK. As medidas WMC, RFC e CBO foram eficientes na predição da propensão a falhas e na quantidade de revisões realizadas nas classes. As métricas LCOM e DIT foram menos eficientes. Além disso, a métrica NOC foi ineficiente para realizar tal atividade.

[Zhou e Leung 2006] discutiram acerca do gravidade do impacto de falhas em softwares, baseado em métricas orientadas a objetos. Eles concluíram que baixos valores de métricas podem ser mais previsíveis do que altos valores. Outra conclusão é que CBO, WMC, RFC e LCOM são estatisticamente significativas e DIT não.

[Singh e Kahlon 2012] analisaram os valores de métricas e como eles podem predizer classes com falhas identificadas como *bad smells* em aplicativos de código aberto. Eles concluíram que algumas métricas podem predizer com alta precisão, auxiliando na melhoria dos índices de manutenibilidade, testabilidade e refatoração. Foi criado um modelo de métrica para detectar *bad smells* em softwares e para a validação foi utilizado o modelo investigando o Mozilla Firefox (Versões 2.0 e 3.0).

[Olague et al. 2007] compararam e validaram três conjuntos de métricas orientada a objetos ([Chidamber e Kemerer 1994,Abreu e Carapua 1994,Bansiya e Davis 2002]). Eles concluíram que as métricas CK foram melhores e mais confiáveis em prever propensão a erros, em especial WMC e RFC. Outra conclusão dos autores é que as métricas não foram eficientes nas fases iniciais do desenvolvimento de software. Portanto, quando o desenvolvimento é ágil ou altamente iterativo, métricas podem não ser efetivas. Essa conclusão é análoga a [Shatnawi e Li 2008] e, portanto, merece ser tema de estudos futuros.

[Janes et al. 2006] analisaram a relação entre métricas orientadas a objetos e falhas em sistemas, focados em sistemas de tempo real no domínio de telecomunicações. Eles concluíram que a comunicação entre classes (RFC e CBO, por exemplo) aumenta a probabilidade de defeitos aparecerem. Quanto mais acopladas as classes estiverem, mais alta é a chance de que defeitos apareçam.

[Gyimothy et al. 2005] analisaram a predição a falhas em softwares de código aberto entre sete versões do Mozilla Firefox (1.0 até 1.6). Eles concluíram que a medida NOC não pode ser utilizada para algumas predições, como a propensão a falhas. As demais métricas, exceto LCOM foram eficientes para realizar tal atividade e a métrica LCOM,

apesar do resultado positivo, precisa ser estudada futuramente.

Em [Kakarontzas et al. 2012], os autores propuseram uma nova métrica para facilitar o reuso caixa-branca. Essa nova métrica é baseada nas métricas CK. Eles concluíram que métricas de acoplamento (RFC e CBO) são eficientes para limitar o reuso caixa-branca das classes. A métrica LCOM é menos eficiente e a NOC não fornece informações substanciais. Já a métrica DIT é eficiente para projetos com tamanho médio, mas ineficiente para grandes softwares.

Portanto, as medidas de acoplamento e complexidade das classes podem fornecer informações úteis durante o processo de desenvolvimento de softwares. Atividades como análise da propensão a erros e facilidade de manutenção podem ser suportadas por valores das métricas que abrangem tais características. A seguir, as classificações das métricas presente nas Tabelas 4.3 e 4.4 são comparadas às conclusões dos estudos selecionados e, posteriormente, as métricas serão analisadas de acordo com a eficiência na execução dos objetivos traçados pelos artigos estudados.

4.5 Análise dos resultados das métricas

São apresentados, individualmente, nesta seção os resultados das métricas acerca de sua eficiência em outros estudos e uma comparação com os valores das métricas obtidos por este trabalho e os valores dos demais trabalhos.

DIT

O valor mínimo para que a métrica DIT seja considerada como anomalia é de 3 e 4 para softwares desenvolvidos em Java e C++, respectivamente. Estes valores são próximos do trabalho de [Nair e Selvarani 2011] (maior que 5) e [Benestad et al. 2006] (maior que 3). Entretanto, altos valores de DIT podem não ser significativos em alguns casos, como descrito na Tabela 4.5.

Somente 20% dos artigos discutidos ([Gyimothy et al. 2005,Benestad et al. 2006,Nair e Selvarani 2011]) no presente estudo usaram DIT com sucesso para predizer propensão a erros. Essa conclusão precisa ser aprimorada para analisar se essa métrica pode ser útil para tal atividade.

Embora a métrica DIT não tenha sido usada com sucesso frequentemente na análise da propensão a erros, ela auxilia em importantes análises estruturais do software, como, por exemplo, no grau de especialização de uma classe [Harrison et al. 1998].

NOC

O valor mínimo para que a métrica NOC seja considerada como anomalia é de 8 para softwares desenvolvidos em Java. Esses valores são próximos aos trabalhos de [Nair e

Selvarani 2011] (maior que 5) e [Benestad et al. 2006] (maior que 3). Para os aplicativos construídos em C++, o valor é 5, que também está próximo desses estudos. Uma importante consideração é que a linguagem C++ provê múltipla herança e o valor da métrica tende a ser maior do que as métricas de softwares desenvolvidos em Java, que possui herança simples. Esse aumento pode estar relacionado a capacidade de uma classe herdar várias outras.

Apenas 33% dos estudos concluíram que a métrica NOC foi eficiente. Em [Subramanyam e Krishnan 2003,Zhou e Leung 2006] foi observado que a métrica NOC é inversamente proporcional a propensão a erros. Quanto mais filhos uma classe possuir, maior é a importância dela no sistema, e possivelmente, mais testada ela foi. Assim como a métrica DIT, a NOC não possui uma eficiência elevada. Métricas hierárquicas podem auxiliar na análise de algumas tarefas como compreensão de programas mas foram pouco eficientes quando relacionadas a análise de propensão a erros.

CBO

O valor mínimo para que a métrica CBO seja considerada como anomalia é de 20 para softwares desenvolvidos em Java e C++. Esses valores são próximos do trabalho de [Nair e Selvarani 2011], que concluiu que classes com CBO entre 20-24 possuem de 78% à 98% de probabilidade de conter defeitos. Em [Shatnawi 2010] a métrica CBO teve o valor 9, bem distante do valor encontrado no presente trabalho.

A métrica CBO é considerada a mais relevante para realizar atividades como predição de propensão a erros, análise do impacto da refatoração em métricas ou estipular o grau de facilidade de reuso caixa-branca. Em todos os estudos, a métrica CBO foi eficiente. Isso é uma valiosa informação para mostrar a importância das métricas de acoplamento na análise de softwares.

Além disso, esse resultado confirma o estudo realizado em [Shatnawi 2010] e fornece uma base para a escolha dessa métrica como fonte de futuros trabalhos.

RFC

O valor mínimo para que a métrica RFC seja considerada como anomalia é de 84 para softwares desenvolvidos em Java e 112 para C++. [Shatnawi 2010] encontrou o valor 40 como *threshold* e [Nair e Selvarani 2011] encontraram uma propensão a defeitos de 82% quando o valor de RFC é maior do que 160. Esses valores são diferentes do encontrado no presente trabalho.

Assim como a CBO, a métrica RFC foi utilizada com eficiência em todos os trabalhos, reforçando o resultado sobre métricas que são baseadas no acoplamento entre classes.

WMC

É considerado como anomalia softwares desenvolvidos em Java que possuem WMC maior do que 29 e para C++ o valor encontrado foi 106. O valor para softwares desenvolvidos em Java é bem próximo aos trabalhos de [Benestad et al. 2006] (maior que 23) e [Shatnawi 2010] (maior que 29). Entretanto, os valores dos aplicativos desenvolvidos em C++ são maiores. Um ponto que precisa ser analisado é o tipo de métrica utilizada para calcular a métrica WMC. A quantidade de métodos que uma classe possui ou a quantidade de caminhos que um método pode ter (Complexidade Ciclomática) interfere diretamente na análise.

Assim como as medidas RFC e CBO, a métrica WMC foi eficiente em todos os artigos analisados. Além de considerar o acoplamento entre as classes, a complexidade delas também interfere na predição de propensão a erros. Essa conclusão reforça os estudos de [Zhou e Leung 2006,Shatnawi 2010].

LCOM

O valor mínimo para que a métrica LCOM seja considerada como anomalia é de 388 para softwares desenvolvidos em Java e 1521 em C++. Essa discrepância pode gerar a seguinte conclusão: a análise da métrica LCOM com base nos valores de outros softwares pouco agrega na análise de sistemas, uma vez que seus valores variam excessivamente (o desvio padrão do IQM da métrica em C++ é de 980,29 para softwares desenvolvidos em C++) de acordo com o contexto do software, experiência da equipe, tempo e custo disponíveis, entre outros itens que podem interferir no processo de desenvolvimento de software.

No trabalho de [Gyimothy et al. 2005], concluiu-se que a métrica LCOM possui boa precisão, mas sua completude é baixa. Alguns trabalhos como [Moser et al. 2006,Janes et al. 2006,Kakarontzas et al. 2012,Johari e Kaur 2012] concluíram que a métrica LCOM é ineficiente para realizar algumas atividades como propensão a erros, análise de refatoração ou facilidade de reuso caixa-branca, conforme estudos relacionados na Tabela 4.5.

4.6 Considerações Finais

Vale ressaltar que o uso de informações sobre métricas vindas de outros softwares é uma tarefa não trivial. De acordo com [Kocaguneli et al. 2010], existem vários fatores que podem dificultar a análise de softwares com base em métricas de outros sistemas. Características como experiência da equipe de desenvolvimento, tempo disponível, custo [Subramanyam e Krishnan 2003] e até a localização geográfica da equipe de desenvolvimento podem afetar em tal análise [Kocaguneli et al. 2010].

No trabalho de [Menzies et al. 2011], foi descoberto que a utilização de dados globais para estimativas de esforços e predição de defeitos pode ser ineficaz na maioria das vezes. Frequentemente, a utilização de dados locais sobre o software é mais eficiente para a realização de tais atividades.

Portanto, encontrar valores para métricas de software que indiquem problemas é uma tarefa difícil e complexa. Por outro lado, é possível notar que, apesar de todas as diferentes características (experiência da equipe, localização, metodologia de desenvolvimento, linguagem de programação, arquitetura, tempo e custo) que podem afetar a construção de software, alguns estudos indicam valores próximos quando relacionados a detecção de falhas ou propensão a erros.

A busca por valores que indiquem futuros problemas ou dificuldades merece estudos contínuos, uma vez que a construção de um software é uma atividade única [Pressman 2009] e a origem dos dados pode ser proveniente de outros projetos.

A primeira contribuição deste estudo são as tabelas com a classificação das métricas. Os estudos escolhidos com base no processo descrito na Seção 4.2 foram tabulados e caracterizados como baixo, normal, alto e anomalia. Essa classificação pode ajudar no desenvolvimento ou manutenção de softwares, servindo como base para a análise das métricas. Vale ressaltar que os valores propostos por esse trabalho não foram validados em outros softwares, apenas nos sistemas já estudados. Portanto, é necessário ter cautela ao julgar os valores das métricas. O intuito é que essa classificação sirva de base para algumas atividades de Engenharia de Software.

Outra contribuição do presente estudo é a discussão sobre as métricas e suas eficiências de acordo com os objetivos propostos em seus respectivos artigos. As métricas de acoplamento (RFC e CBO) e complexidade (WMC) foram eficientes em diversas atividades como a predição da propensão a erros, análise do impacto da refatoração ou facilidade de reuso caixa-branca. A medida LCOM não foi tão eficiente quanto as demais e as métricas DIT e NOC (ambas relacionadas às características hierárquicas) foram pouco eficientes.

Embora as métricas de CK sejam amplamente utilizadas para mensurar softwares, a escolha das métricas certas em determinadas situações é essencial para o sucesso de atividades em Engenharia de Software.

Para aprimorar esse estudo, faz-se necessário a validação dos valores classificados como anomalia em diversos estudos, associando esses valores às falhas de software, índices de manutenibilidade, testabilidade ou compreensibilidade. Outra atividade é separar os softwares de acordo com o tipo (biblioteca, interfaces, jogos) e analisar se os valores das métricas possuem relação. Outra melhoria é aumentar a quantidade de conferências e revistas científicas utilizadas. Revistas como JSME (*Journal of Surveying and Mapping Engineering*) e SPE (*Software: Practice and Experience*) e as conferências FSE (*Foundations of Software Engineering*), ASE (*Automated Software Engineering*) e SCAM (*Source Code Analysis and Manipulation*) poderiam ser incluídos.

O presente estudo é aplicado na área de visualização de software, suportando a construção do modelo de visualização (proposto no Capítulo 5, onde os valores das métricas representarão características da visualização, como cor e raio dos objetos gráficos. As métricas visualizadas no modelo são CBO e WMC. Essa escolha é baseada nas informações contidas na Tabela 4.5.

Capítulo 5

SUVSoft - Uma metáfora do universo para compreensão de programas

5.1 Introdução

Nesse capítulo serão expostos dois resultados importantes desse trabalho: (i) o modelo proposto baseado em uma metáfora do universo com a aplicação de forças gravitacionais e (ii) o aplicativo que o implementa. Para suportar tal modelo, foi construído o SUVsoft (*System Universe Visualization Software*), uma ferramenta de visualização de software baseada no modelo do universo para realização de engenharia reversa e que calcula as métricas CK de bibliotecas escritas em C#, J# ou Visual Basic.NET. Além disso, será feita uma análise do presente modelo aplicado em cinco softwares de tamanhos e tipos variados.

Quanto maior for a familiarização com um aplicativo, mais fácil é a compreensão que uma mudança pode vir a gerar. Essa familiarização tem importante papel nas atividades relacionadas a compreensão de programas [Storey et al. 1999]. O foco deste trabalho é propor uma representação gráfica de sistemas desenvolvidos sob o paradigma orientado a objetos, buscando assistir às atividades de entendimento e compreensão de programas, por meio de engenharia reversa e visualização de software.

A representação gráfica dar-se-á baseada em uma metáfora do universo e seus corpos celestes. O universo é o espaço disponível para a visualização e as classes, interfaces e enumeradores são os corpos celestes. A escolha por esse modelo foi motivada na visualização de grandes softwares, em que o tamanho do espaço de visualização do aplicativo interfere negativamente na compreensão de programas [Storey 2001,Holten 2009,Wettel 2010]. A ideia é que a visualização disponibilize uma visão ampla do software e, quando o usuário julgar necessário, ele focalize nos pontos que lhe chamem a atenção, conforme feito em outros trabalhos como [Eick et al. 1992,Storey 2001,Jones et al. 2002,Wettel 2010].

5.2 Modelo de visualização

O modelo gráfico desenvolvido busca auxiliar a visualização das estruturas e métricas de um determinado conjunto de classes. As classes são transformadas em estruturas baseadas em grafos (quando as classes possuem ligações de herança), onde cada classe é um nó e as arestas do grafo são as ligações de herança entre as classes SUPER e as classes SUB. Os nós que representam as classes recebem características específicas como cor e raio, que fornecem ao usuário a possibilidade de diferenciar os nós por meio de seus atributos. O objetivo é que o usuário, por meio da visualização, tenha condições de extrair informações relacionadas às métricas e estruturas de um software, proporcionando um melhor entendimento do mesmo.

Para complementar a visualização e auxiliar nas atividades de engenharia de software, foram propostas inicialmente as métricas WMC e CBO (descritos na Seção 2.3) como atributos para definir cor e tamanho dos nós, respectivamente. Quanto maior o valor de CBO, maior será o raio do nó, enquanto que os valores de WMC são usados para mapear as cores dos nós de acordo com uma escala variando entre as cores azul (menor valor) até vermelho (maior valor). Os mapeamentos dos atributos dos nós com as métricas citadas anteriormente podem ser alterados de acordo com a necessidade do usuário, uma vez que são passíveis de modificação via sistema.

A Figura 5.1 ilustra a visualização da biblioteca *AjaxControlToolkit*[1], utilizando o modelo proposto. Esse sistema possui classes com relacionamento de herança e classes que não possuem qualquer tipo de relação de herança. Nessa visualização, os valores da métrica WMC foram aplicados como cor e os valores da métrica CBO como tamanho do raio dos corpos celestes.

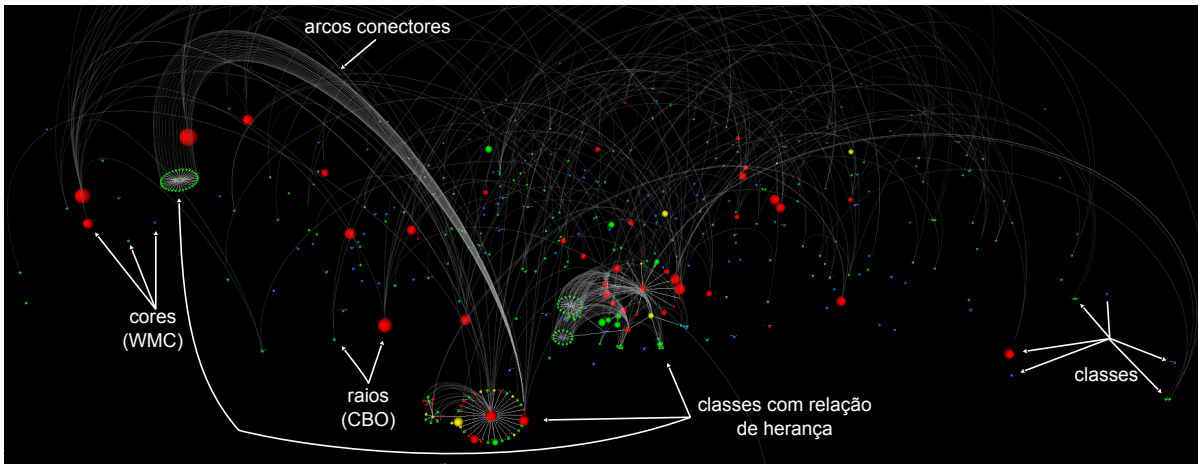


Figura 5.1: Exemplo dos princípios do modelo gráfico.

O modelo de visualização descrito neste trabalho foi construído com base em uma metáfora simplista do universo, ilustrado na Figura 5.2. Vale ressaltar que essa metáfora

[1]Disponível em: https://ajaxcontroltoolkit.codeplex.com/

não leva em consideração as leis da física e que a visualização não foi construída baseando-se em modelos físicos. A metáfora somente fornece a nomenclatura dos elementos usados no modelo de visualização e uma analogia de uma imagem estática do universo. Existem os seguintes itens no modelo:

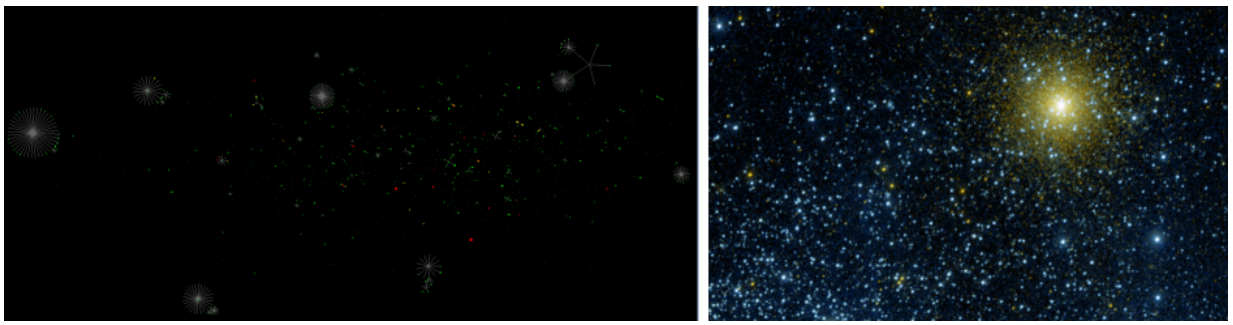


Figura 5.2: Comparação do modelo de visualização com uma imagem do universo. Disponível em [Schiavon 2007].

- **Universo:** Ambiente de visualização dos nós e arestas que representam o software;
- **Corpos Celestes:** Cada corpo celeste representa um nó. Não existe distinção entre os corpos celestes, como planetas, satélites ou cometas. Todo nó é um corpo celeste;
- **Pontes Conectoras:** Quando dois nós possuem a relação pai/filho, estes são ligados por uma aresta, para indicar essa relação;
- **Arcos Conectores:** Quando dois nós possuem uma relação de acoplamento, eles serão conectados por um arco, que é construído no eixo Z do espaço. O acoplamento dos nós é caracterizado pelo acoplamento das classes que os representam;
- **Posicionamento dos Corpos Celestes:** Pode ocorrer de quatro formas. As duas primeiras são baseadas na distribuição Gaussiana e as duas últimas são baseadas na quantidade de filhos que um nó possui.
  - **Posicionamento Livre:** Os nós são distribuídos de forma livre, onde o tamanho do universo limita a distribuição dos nós;
  - **Força Gravitacional:** Quando dois ou mais nós possuem forças gravitacionais relacionadas, o posicionamento é afetado por esta força, fazendo com que corpos sejam posicionados próximos uns dos outros. Essa força gravitacional é representada pela quantidade de métodos que uma classe invoca das outras;
  - **Visualização Circular:** Ocorre quando um nó possuir uma quantidade de filhos superior a cinco;
  - **Visualização em Árvore:** Ocorre quando um nó possuir até quatro filhos.

Essa representação busca amenizar um problema recorrente nos sistemas de visualização de software: a visualização de aplicativos com uma quantidade elevada de objetos

gráficos. Quanto maior for o número de objetos gráficos, mais difícil fica a interpretação do modelo de visualização por parte do usuário [Storey 2001,Holten 2009,Wettel 2010]. Outro ponto de interesse deste trabalho é tratar o raio de cada nó como critério de construção para o modelo. Quando o raio do nó é desconsiderado no posicionamento dos objetos gráficos na visualização, nós com raios maiores que os demais podem sobrepor os outros nós. Esta característica busca fazer com que os nós não se sobreponham, fornecendo uma visão mais limpa de todos os nós e consequentemente, classes. Além disso, as métricas CBO e WMC, ambas usadas com sucesso em diversos trabalhos (conforme Capítulo 4), norteiam o modelo de visualização.

A Figura 5.3 ilustra os itens do modelo apresentado nesta dissertação relacionados à metáfora do universo. O sistema é representado pelo universo, as classes são ilustradas pelos corpos celestes e as métricas WMC, CBO e RFC definem os atributos de cor, raio e posição, respectivamente. As métricas DIT e NOC também atuam na posição, em que a profundidade dos grafos e a forma de visualização são afetadas.

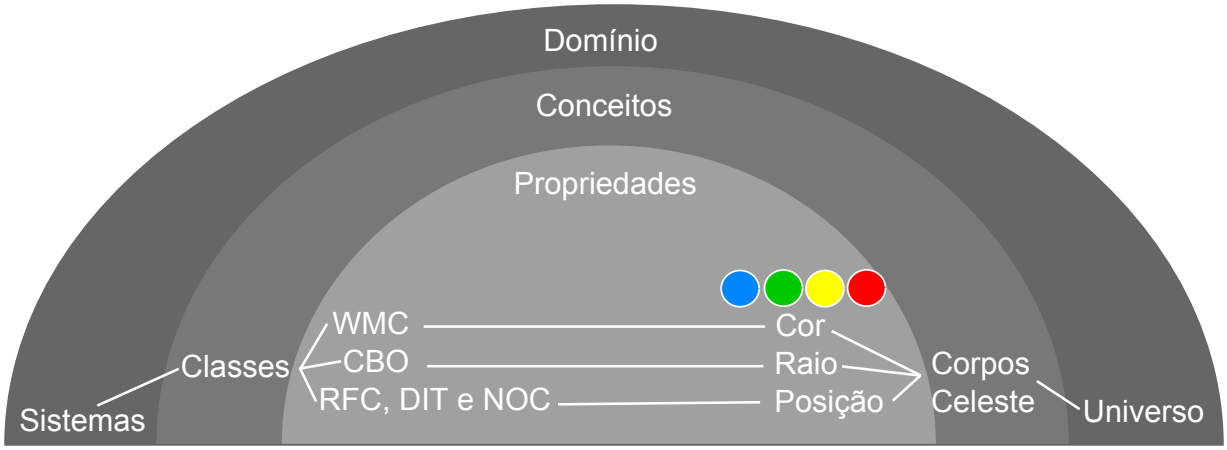


Figura 5.3: Relação entre o modelo e a metáfora do universo. Figura criada baseando-se no modelo apresentado em [Wettel 2010].

5.3 Algoritmo para a construção do modelo

Nesta seção, o algoritmo para a construção do modelo será descrito a partir do fluxograma ilustrado na Figura 5.4. Para facilitar o entendimento do algoritmo, os corpos celestes definidos no modelo serão tratados como "nós".

5.3.1 Limitando o tamanho do universo

O tamanho do universo da visualização é limitado. Essa limitação é feita para evitar uma dispersão excessiva da visualização. Com base na quantidade de nós que a visualização terá e os valores informados pelo usuário para a delimitação dos eixos X e Y, os limites

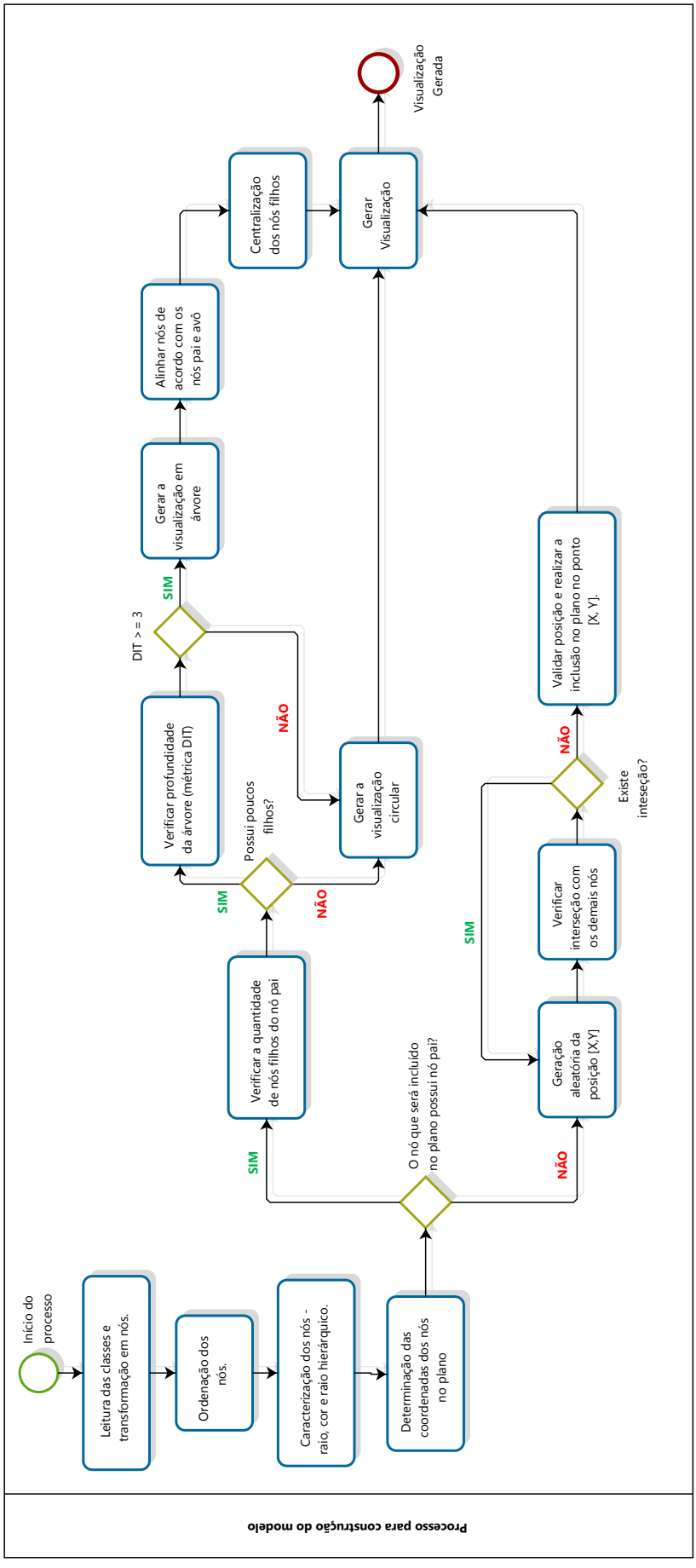


Figura 5.4: Fluxograma do algoritmo de construção do modelo.

do universo são criados de acordo com as equações 5.1a e 5.1b a seguir. As variáveis Sx e Sy representam os tamanhos dos eixos X e Y, respectivamente.

$$Sx = \text{número de nós} \cdot \text{parâmetro de dispersão do eixo X} \tag{5.1a}$$

$$Sy = \text{número de nós} \cdot \text{parâmetro de dispersão do eixo Y} \tag{5.1b}$$

Para que a visualização possa ser aprimorada, é necessário que o valor do parâmetro de dispersão X e Y sigam uma determinada proporção. Se os valores de X e Y forem próximos, a visualização terá uma forma quadrada. O modelo sugere que esta proporção seja 3/1, mas esses valores podem ser alterados no sistema. Essa afirmação é feita porque o campo de visualização disponível pelo *SUVsoft* sofre interferência do menu do aplicativo, que diminui o espaço do eixo Y e também para acompanhar a proporção de grande parte dos monitores de vídeo.

Outro ponto de distinção é que os limites abrangem somente os eixos X e Y, sem considerar o eixo Z. Esse eixo suporta apenas o conceito dos "Arcos Conectores", descritos na Seção 5.4.3.

5.3.2 Transformando as classes em nós

O primeiro passo do algoritmo é transformar o conjunto de classes provenientes no arquivo .dll ou XML em nós. Cada classe presente no arquivo é transformada em um nó que, posteriormente, será incluído no universo. Vale ressaltar que os nós do modelo possuem a mesma relação de herança de suas respectivas as classes. Ou seja, se a classe B herda as características da classe A, o nó B será filho do nó A. Assim que todas as classes foram transformadas em nós, esse conjunto de nós é ordenado, conforme explicado na próxima seção.

5.3.3 Ordenando os nós

Após o recebimento do conjunto de nós, esses são ordenados de acordo com os seguintes critérios:

1. Nós que **não possuem** PAI mas **possuem** FILHOS;
2. Nós que **possuem** PAI e **possuem** FILHOS, ordenados pela métrica DIT (ou profundidade da árvore);
3. Nós que **possuem** PAI mas **não possuem** FILHOS, ordenados pela métrica DIT (ou profundidade da árvore);
4. Nós que **não possuem** PAI e **não possuem** FILHOS.

Esses critérios são subdivididos em dois grupos: (i) nós que não possuem pai (1 e 4) e (ii) nós que possuem pai (2 e 3). Isso acontece para que o modelo consiga distinguir a forma de posicionamento dos nós. Quando o nó não possui um nó pai, sua posição é gerada de forma livre. Caso o nó possua um nó pai, então seu posicionamento será baseada na posição do nó pai.

Entretanto, para que o algoritmo de visualização fique mais eficiente, a disposição dos nós no espaço de visualização ocorre em quatro passos.

Primeiramente, os nós que são referência para outros nós são incluídos na visualização (nós sem pai mas com filhos). A inclusão dos nós do item 1 acontece de forma livre, conforme será descrito na seção 5.3.5.

Logo após, os nós dos itens 2 e 3 são gerados e ordenados pela profundidade em que ficarão no grafo. Quanto menor for a profundidade, mais rápida é a inclusão no modelo.

Por fim, os nós soltos (sem pai e filho) são adicionados ao espaço de visualização. Esse passo de ordenação foi necessário para que a inclusão dos nós no espaço não fosse recursiva e para que a análise de sobreposição entre nós fosse aprimorada. É mais fácil analisar se somente um nó (referente ao item 4) está sobrepondo algum outro nó do que um conjunto de nós sobrepondo outro nó. Essa afirmação é feita porque o raio de um nó solto, na maioria dos casos, é menor do que o raio hierárquico (conceito explicado adiante) dos nós que possuem filhos.

5.3.4 Caracterizando os nós

Nessa etapa os nós são caracterizados de acordo com os valores das métricas de suas respectivas classes. Os atributos raio, cor, nó pai, raio hierárquico e posicionamento são descritos a seguir.

Raio

O tamanho do raio de um nó é baseado no valor da métrica CBO. Até o presente momento, o raio de um nó é o valor da métrica CBO acrescido de 10. Sendo assim, todo nó possui um raio mínimo (r_{min}) de tamanho 10. Ainda estão sendo desenvolvidas estratégias para tornar o raio escalável com o tamanho do universo.

Cor

A cor de cada nó é definida com base no valor da métrica WMC. De acordo com a classificação do valor da métrica (valor baixo, normal, alto ou anomalia, descrito no Capítulo 4), o nó recebe uma cor na escala azul-vermelho. Essa escala também pode ser considerada como uma derivação da escala azul-verde-vermelho [Diehl 2007], em que a escala de cores passa apenas em quatro cores: azul, verde, amarelo e vermelho. A Figura

5.5 ilustra a escala azul-verde-vermelho adotada. Vale ressaltar que esta escala não é contínua, ou seja, é segmentada em azul, verde, amarelo e vermelho.

Figura 5.5: Escala de cores dos objetos gráficos. Figura obtida de [Diehl 2007].

Essa escala passa por quatro cores, descritas a seguir:

- Azul: quando a métrica possui um valor até o nível considerado "**baixo**";
- Verde: quando a métrica possui um valor acima do nível "**baixo**" e inferior ao nível "**alto**", considerado "normal";
- Amarelo: quando a métrica possui um valor igual ou superior ao nível "**alto**" mas inferior ao nível "**anomalia**";
- Vermelho: quando a métrica possui um valor igual ou superior ao nível "**anomalia**";

Figura 5.6: Possíveis cores dos nós de acordo com o valor da métrica WMC.

A Figura 5.6 exemplifica o esquema de cores aplicado aos nós no modelo de visualização. Nos trabalhos de [Lanza 2003] e [Wettel e Lanza 2007], a coloração dos objetos gráficos é realizada na escala cinza-azul. Enquanto em [Eick et al. 1992,Marcus et al. 2003,Holten et al. 2007] é utilizada a escala verde-vermelho, com algumas adaptações. O presente modelo utiliza o esquema azul-verde-vermelho.

Nó pai

O nó pai corresponde ao nó que representa a classe pai. A Figura 5.7 ilustra a conexão entre nós baseada na herança de suas respectivas classes. O nó A possui dois nós filhos, B e C.

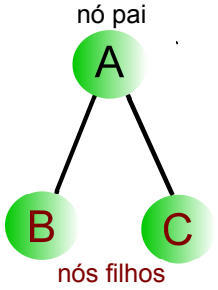


Figura 5.7: Caracterização dos nós pais e filhos.

Porém, existe um item que deve ser aprimorado em trabalhos futuros. Quando a classe SUPER possui apenas uma classe SUB, a distinção entre o nó pai e nó filho torna-se difícil, conforme Figura 5.8. Uma seta indicando a direção da herança pode resolver esse problema. Entretanto, é necessário ver o impacto dessa mudança na visualização. Vale ainda observar que no SUVSoft o usuário pode selecionar um nó e obter suas propriedades. Com isso, é possível determinar quem é o nó pai e o nó filho.

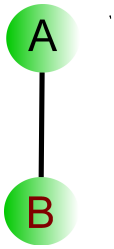


Figura 5.8: Classe SUPER com apenas uma classe SUB. Exemplo da dificuldade em descobrir qual objeto é pai ou filho interfere na visualização.

Nós que possuem poucos filhos

Um ponto importante no modelo de visualização é quando o nó possui poucos filhos. Uma constante K define a quantidade mínima de filhos que um nó precisa ter para se encaixar na visualização do tipo árvore ou circular, conforme explicado a seguir. Essa constante recebeu o valor quatro no presente trabalho.

Raio hierárquico

O raio hierárquico (RH) representa o raio do círculo em que o nó pai e os nós filhos estão alocados. Esse raio é utilizado para impedir a sobreposição de objetos, conceito importante para a construção do modelo. O raio hierárquico entre o nó pai e toda a árvore de herança define o espaçamento entre os centros dos nós que possuem ligação de hierarquia.

Existem duas formas de calcular o raio hierárquico: quando o nó em questão possui poucos filhos ou muitos filhos.

Quando possuir poucos filhos, o raio hierárquico é a soma de R do nó pai (nó i) com o maior raio hierárquico dos nós filho (nós $filhos(i)$)s, acrescido em 20% (Equação 5.2). A escolha do maior valor de RH foi feita para que o algoritmo mantenha um padrão de distribuição entre os nós filhos, fazendo com que a visualização se torne homogênea.

A Figura 5.9 mostra o raio hierárquico do nó A, que leva em consideração o maior raio hierárquico dos nós filhos. Nesse caso, o $max(RH(filhos(A)))$ é do nó B. Esse cálculo é realizado, de forma recursiva, para todas as classes que possui(em) classe(s) filha(s).

$$RH(i) = (R(i) + max(RH(filhos(i)))) \cdot 1,2 \tag{5.2}$$

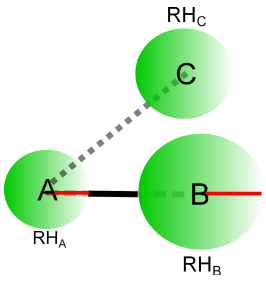


Figura 5.9: Ilustração dos nós que fazem parte da operação recursiva para cálculo do raio hierárquico em nós que possuem poucos filhos.

Uma característica do raio hierárquico para nós que possuem poucos filhos é que, quando o raio hierárquico dos nós filhos possuírem diferentes tamanhos, e provavelmente terão, o raio assumido para todos os nós filhos será o maior entre eles.

Quando o nó em questão possuir muitos filhos, o cálculo do raio hierárquico é baseado no perímetro (P) do círculo criado pelos seus filhos. Esse perímetro é a soma dos diâmetros hierárquicos (baseados nos raios hierárquicos) dos nós filhos. A Figura 5.10 mostra o perímetro P de acordo com o raio hierárquico dos nós filhos. A Equação 5.3a apresenta o cálculo do perímetro e a Equação 5.3b é o cálculo do raio hierárquico do nó.

$$P(i) = \sum_{n} (RH(filhos(i)) \cdot 2) \tag{5.3a}$$

$$RH(i) = \frac{P(i)}{2\pi} \cdot 1,2, \tag{5.3b}$$

onde P é o perímetro do círculo criado de acordo com o raio hierárquico dos nós filhos, $RH(filhos(i))$ é o raio hierárquico dos nós filhos do nó i e n é a quantidade total de nós filhos do nó i.

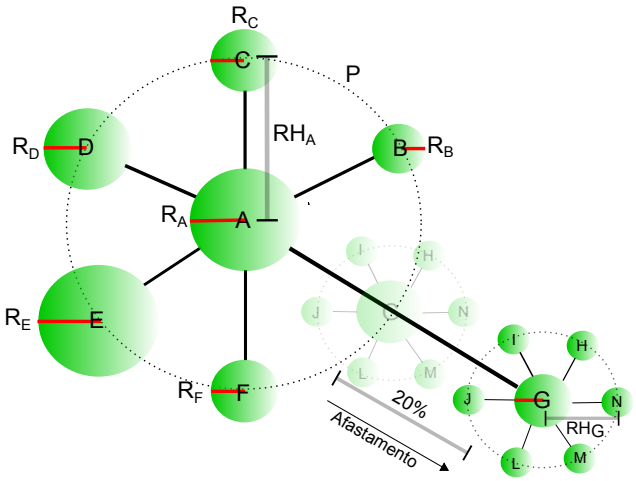


Figura 5.10: Ilustração dos nós que fazem parte da operação recursiva para cálculo do raio hierárquico. O nó G não está centrado no raio hierárquico pois ele possui filhos. Com isso, é feito um afastamento de sua posição.

Por ser uma operação recursiva, a condição de parada é quando um nó não possuir nó(s) filho(s). Nesse caso, o valor do raio hierárquico é igual ao valor do raio do próprio nó.

O motivo pelo qual o maior raio hierárquico dos filhos foi escolhido como padrão é para tornar a distância entre o eixo dos objetos homogênea. Se a distância fosse calculada individualmente, alguns nós ficariam mais próximos ou mais distante do nó pai, dificultando a visualização.

Existe também um afastamento extra dos nós que possuem filhos. Tal característica tem como objetivo deixar a visualização mais limpa e menos agrupada. Esse afastamento é calculado da seguinte forma: o raio hierárquico do nó pai é somado ao raio hierárquico do nó filho, com um acréscimo de 20%.

5.3.5 Posicionando os nós no universo

Para a geração do posicionamento dos nós, existe apenas uma condição que deve ser analisada: se o nó que será incluído no universo possui relação de hierarquia com algum outro nó (a classe em questão possui classe pai). Quando esse novo nó possuir tal relacionamento, a sua posição será de acordo com o posicionamento do nó pai. Caso não possua, seu posicionamento será livre. Entretanto, quando existir a atuação de alguma força gravitacional sobre o novo nó, sua posição continuará livre, mas regida por uma distribuição de probabilidade diferente. Nas seções a seguir serão descritos os três tipos de posicionamento possíveis: (i) geração livre; (ii) visualização circular e (iii) visualização em árvore.

Geração livre de posições

A distribuição dos nós que não possuem pai é feita utilizando valores fornecidos pela distribuição Gaussiana para os eixos X e Y, objetivando gerar uma visualização bidimensional. No eixo Z são traçados somente os arcos conectores entre classes que possuem acoplamento, conforme Seção 5.4.3. Existem dois tipos de geração de posição, que diferenciam entre si apenas pelos valores adotados para definir a média da distribuição Gaussiana. A primeira forma de distribuição é chamada de livre, enquanto que a segunda é baseada na força gravitacional entre nós.

Na geração livre a posição dos nós é feita baseando-se em uma distribuição Gaussiana centrada centro da espaço de visualização. Para gerar o ponto X da posição, é produzido um número aleatório, com base na distribuição Gaussiana com média igual a zero e desvio padrão igual a Sx. Para gerar o ponto Y da posição, é produzido outro número aleatório, com base na distribuição Gaussiana, com média igual a zero e desvio padrão igual a Sy. A escolha da distribuição Gaussiana foi feita para que os objetos presentes na visualização ficassem concentrados na parte central da tela e fossem, gradativamente, diminuindo em quantidade. Isso é possível graças a característica dessa distribuição em que a probabilidade de pontos serem gerados no centro (em torno da média) é maior do que nas periferias [Bishop e Nasrabadi 2006].

Na geração de posições com base na força gravitacional entre os corpos celestes, um peso de ligação entre nós é atribuído a todos os nós. Nesse caso, a ligação entre os nós é representada pela quantidade de métodos que uma classe utiliza de outras classes. Para cada uma das classes que possuem ligação entre si é gerado um peso para os nós correspondentes. Dado esses pesos, é feita uma média com as posições de cada uma das classes que possuem ligação, multiplicado pelos seus respectivos pesos. As Equações 5.4a e 5.4b mostram como são feitos os cálculos para determinar essa média para um determinado nó i.

$$C_x = \frac{\sum_j P_x(j) fg(i,j)}{\sum_j fg(i,j)} \tag{5.4a}$$

$$C_y = \frac{\sum_j P_y(j) fg(i,j)}{\sum_j fg(i,j)}, \tag{5.4b}$$

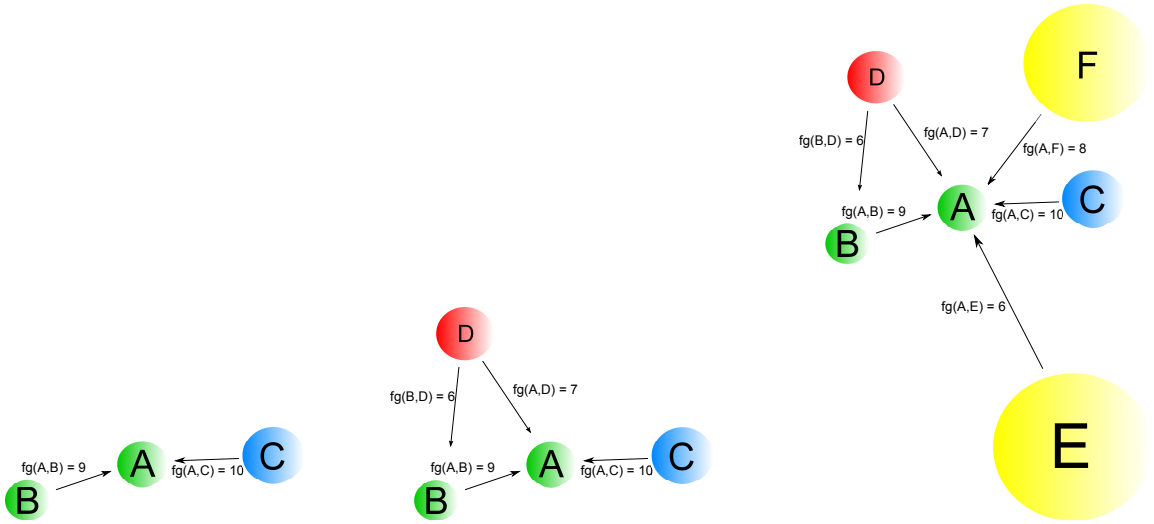
onde $(P_x(j), P_y(j))$ é a posição do nó j, $fg(i,j)$ é o peso da ligação entre o no i com o nó j e C_x e C_y é o ponto médio definido pelo peso gravitacional.

São levadas em consideração somente posições de nós que possuem ligação com a classe em questão que já foram posicionadas no universo. Ou seja, classes que possuem ligação com a classe que será incluída no universo mas ainda não foram posicionadas não entram no cálculo. As Figuras 5.11a, 5.11b e 5.11c exibem a inclusão dos nós com forças

gravitacionais (fg) de seis nós que foram adicionados na seguinte ordem: A → B → C → D → E → F. As classes B, C, D, E e F possuem força gravitacional (fg) com A. A classe D possui força gravitacional com o nó A e com o nó B e, consequentemente, sua posição ficará próxima de ambas. Portanto, assim que estas classes forem inseridas no universo, a distribuição Gaussiana ficará limitada a proximidade da classe a qual ela possui o maior número de atributos e/ou métodos compartilhados. A Figura 5.12 mostra a visualização de um software comercial sem (Figura 5.12a) e com (Figura 5.12b) a atuação da força gravitacional. É possível notar que a distribuição dos nós está fortemente ligada na Figura 5.12b, em que os nós estão agrupados pela força gravitacional em diversos pontos.

Para a geração do novo ponto no plano, são necessários dois novos valores para x e y. O valor de x é obtido por meio de geração aleatória de um valor que segue uma distribuição Gaussiana com média dada pelo valor de C_x e desvio padrão Sx dividido por dez. O valor de y é obtido por meio de geração aleatória de uma distribuição Gaussiana onde a média é representada pelo valor de C_y e o desvio padrão é o valor de Sy dividido por dez. Quanto maior for o peso entre os nós, maior será a probabilidade de eles estarem próximos. A escolha pela constante de valor dez não foi estudada. O valor refere-se a testes realizados durante a dissertação que forneceram uma boa visualização nos softwares apresentados a seguir.

A visão proporcionada pela aplicação da força gravitacional pode ser relevante na análise do acoplamento entre as classes. Dentre os trabalhos correlatos, nenhum tinha como objetivo orientar a visualização com base no acoplamento, critério importante em vários estudos para análise de questões relacionadas a identificação de classes propensas a erros [Shatnawi e Li 2008,Subramanyam e Krishnan 2003,Johari e Kaur 2012,Nair e Selvarani 2011,Zhou e Leung 2006,Olague et al. 2007,Janes et al. 2006,Gyimothy et al. 2005,English et al. 2009]. Portanto, espera-se que esse tipo de informação, aliada as características dos nós (cor, raio e hierarquia) possa assistir aos desenvolvedores em atividades de engenharia de software.



(a) (b) (c)

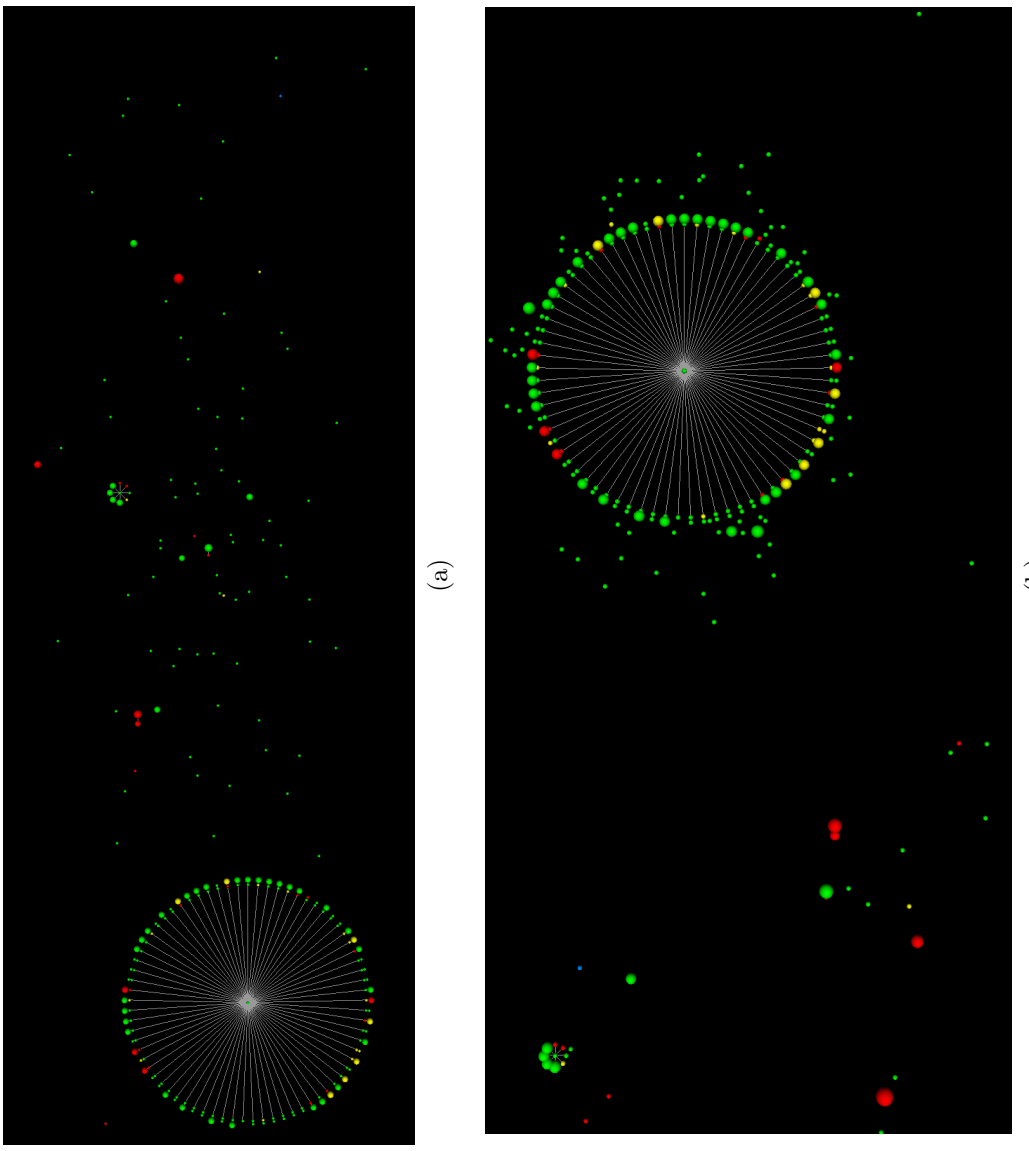
Figura 5.11: (a) Os nós B e C posicionados próximo ao nó A de acordo com a força gravitacional. (b) O nó D está posicionado próximo a A e B. (c) Os nós E e F são posicionados próximo ao nó A, que possui força gravitacional com ambos.

Geração da posição com base no nó pai

Quando um nó possui um nó pai, seu posicionamento pode acontecer de duas formas: (i) visualização circular ou (ii) visualização em árvore. As condições a seguir ilustram as situações em que esses dois tipos de visualizações são utilizados.

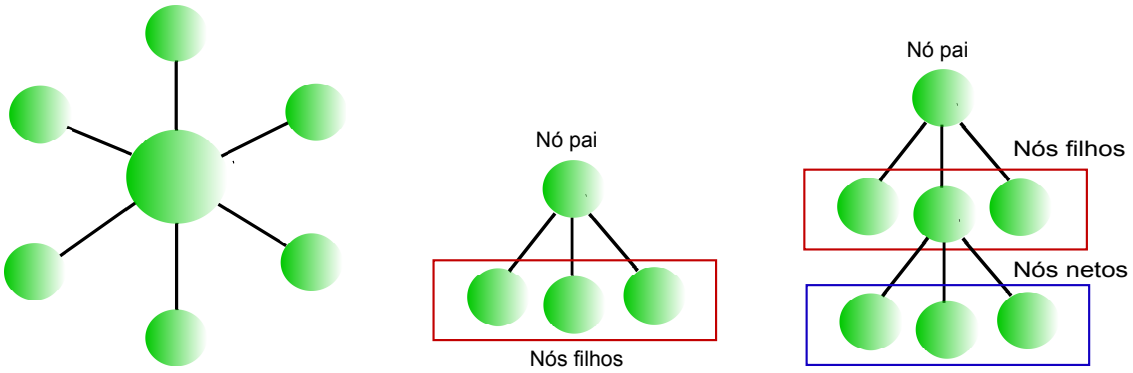
1. Quando a quantidade de filhos de um nó é maior do que K, a visualização tem uma forma circular, conforme Figura 5.13a.
2. Quando o número de filhos de um nó é menor ou igual a K, a visualização tem forma de árvore, conforme Figura 5.13b.
3. Quando a profundidade do grafo é maior ou igual a 2 (ou a classe representada pelo nó possuir o valor de 3 para a métrica DIT) e do tipo árvore, os nós filhos são orientados de acordo com a direção do nó avô/pai (Figura 5.13c).
4. Quando a profundidade do grafo é maior ou igual a 2 (ou a classe representada pelo nó possuir o valor de 3 para a métrica DIT) e do tipo árvore, os nós filhos são centralizados.

As condições citadas anteriormente foram escolhidas por motivos estéticos, objetivando uma melhor visualização.



(a) (b)

Figura 5.12: (a) Atuação da Força Gravitacional. (b) Ausência da Força Gravitacional.



(a) (b) (c)

Figura 5.13: (a) Visualização Circular. (b) Visualização em Árvore. (c) Visualização em Árvore com Filhos.

Visualização circular

A visualização circular acontece quando a quantidade de filhos de um nó é maior do que a constante K. Para realizar a distribuição homogênea dos nós, o ângulo entre os nós filhos é calculado. A Equação 5.5 ilustra o cálculo do ângulo (γ) entre os nós filhos.

$$\gamma = 360/n, \tag{5.5}$$

onde n é a quantidade de nós filhos que o nó possui.

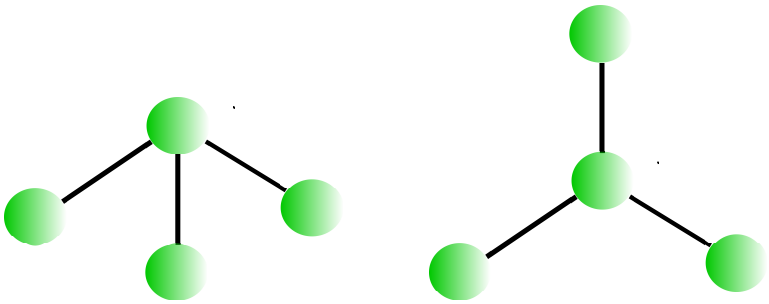
A distância entre o nó pai e seus filhos é dada pelo raio hierárquico. Como a visualização circular só acontece quando o número de filhos de uma classe é maior do que K e o nó pai que servirá como referência já está alocado no universo, todos os filhos da primeira profundidade do grafo são incluídos. Com isso, uma lista de nós filhos é criada e inserida no universo de acordo com a ordem. Cada nó nessa lista possui um índice, que inicia em zero e termina na quantidade de itens da lista menos um. Conforme o valor do índice aumenta, o valor do ângulo do nó o acompanha, possibilitando a inclusão dos nós filhos em diferentes posições. Para efetuar a alocação dos nós ao redor do nó pai, as Equações 5.6a e 5.6b são realizadas para a geração das posições X e Y do novo nó filho N.

$$N_x = F_x + (RH(F) \cdot \cos(\gamma \cdot i)) \tag{5.6a}$$
$$N_y = F_y + (RH(F) \cdot \text{sen}(\gamma \cdot i)), \tag{5.6b}$$

onde F_x é a posição no eixo X do nó pai, F_y é a posição no eixo Y do nó pai, RH_F é o raio hierárquico do nó pai, γ é o ângulo dos objetos filhos e i é o índice do nó filho na lista.

Visualização em forma de árvore

A visualização em forma de árvore acontece quando a quantidade de nós filhos é menor ou igual ao valor da constante K. A motivação por essa diferença no tipo de visualização é estética. Quando uma classe possui poucos nós filhos, a visualização circular não mostrou-se esteticamente útil, deixando algumas informações importantes difíceis de serem extraídas. As Figuras 5.14b e 5.14a ilustram exemplos de visualização circular e em forma de árvore com poucos filhos, respectivamente. É difícil saber, pela Figura 5.14b quem é o nó pai e quem são os nós filhos. Já na Figura 5.14a, é mais intuitivo identificar quem é o nó pai.



(a) (b)

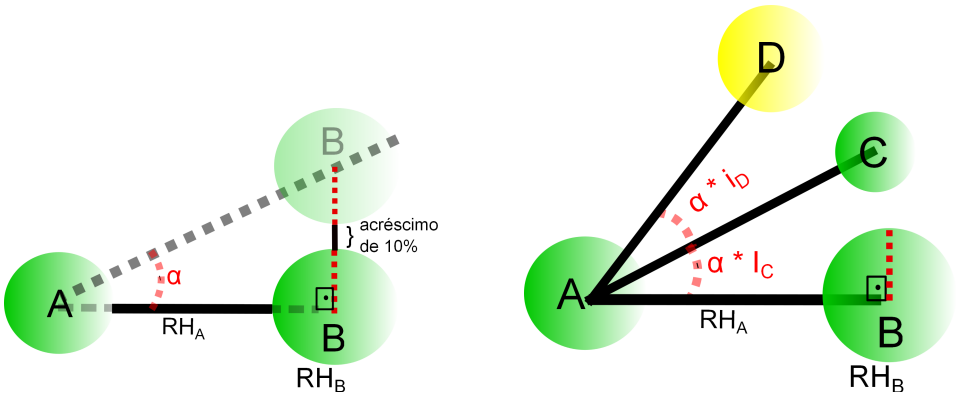
Figura 5.14: (a) Visualização em árvore com poucos filhos. (b) Visualização circular com poucos filhos.

Para realizar a visualização em forma de árvore, o maior raio hierárquico dos nós filhos é escolhido para servir como medida para distanciar todos os nós filhos. As Figuras 5.15a e 5.15b ilustram o cálculo da visualização em forma de árvore. O nó A possui três nós filhos B, C e D, que, por sua vez, não possuem filhos. O raio do nó B (R_B) é maior do que os raios dos nós C e D. Portanto, para determinar o ângulo α que distanciará os nós filhos, é definido um triângulo formado pelo raio hierárquico do nó pai (nó A) e o dobro do maior raio hierárquico dos nós filhos (raio do nó B) acrescido em 10% (Figura 5.15a). Em seguida, é obtido o ângulo α. Essa duplicação acontece para simular o maior ângulo α possível, já que existe a possibilidade de existir, entre os nós filhos, outro nó que possua um raio hierárquico de mesmo tamanho.

Esse ângulo é usado para alocar todos os demais nós filhos. Assim, quando o próximo nó filho for inserido (o nó C do atual exemplo), o ângulo α é usado para posicionar o novo nó, de acordo com seu índice (i) perante à lista. Esse passo é realizado até que todos os nós filhos tenham sido posicionados. Todo primeiro nó da lista de filhos possui índice zero e, portanto, será incluído no ângulo α com valor zero.

É possível notar que existe um acréscimo no raio hierárquico para que os nós não se colidam, caso o raio hierárquico seja igual aos dos demais. Esse acréscimo é de 10% do tamanho do cateto oposto. É importante ressaltar que esse tipo de visualização só

acontece quando a profundidade da árvore é maior ou igual a dois, ou o valor da métrica DIT da classe representada pelo nó for maior ou igual a três.



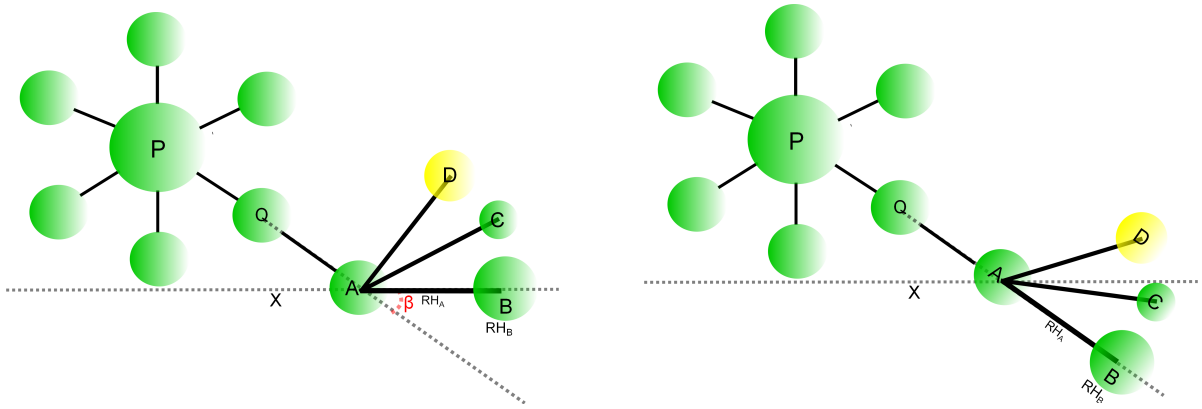
(a) (b)

Figura 5.15: (a) Alocação do primeiro nó e cálculo do ângulo α. (b) Inclusão do segundo nó de acordo com o ângulo α e o índice i.

Após a alocação de todos os nós filhos, é necessário alinhar os nós filhos, conforme seção a seguir.

Alinhamento dos nós filhos

A orientação dos nós na visualização em árvore é alinhada de acordo com os nós pai e avô (nó pai do nó pai). Esse é o motivo pelo qual existe a pré-condição de que a visualização em árvore só aconteça quando um nó pai possuir poucos filhos e sua profundidade for maior ou igual a dois (ou $DIT \geq 3$), uma vez que é necessário a existência de um nó pai e um nó avô. Para realizar o alinhamento dos nós em relação aos nós pai e avô, é feito o seguinte cálculo: o vetor $\vec{PQ}$ orienta a posição do nó A. Por sua vez, o vetor $\vec{QA}$ orienta a construção dos filhos do nó A. Para realizar tal atividade, é calculado o ângulo β, em relação ao vetor $\vec{QA}$ e $\vec{X}$ (na posição [1,0]). Com o ângulo β obtido, a distância entre os nós (ângulo α) realiza uma alteração no ângulo β, fazendo com que a visualização fique alinhada. A Figura 5.16a ilustra os nós desalinhados e as medidas para o cálculo de alinhamento e a Figura 5.16b ilustra o resultado dos nós já alinhados.



(a) (b)

Figura 5.16: (a) Visualização em árvore **sem** alinhamento dos nós filhos. (b) Visualização em árvore **com** alinhamento dos nós filhos.

Existem duas possibilidade para o sinal da posição do eixo Y do vetor $\vec{QA}$. Caso seja negativo, o ângulo α receberá uma subtração, dado o sinal de β. Se for positivo, o ângulo α será incrementado por β. A Equação 5.7 ilustra o resultado de α, levando em consideração o sinal de $\vec{QA}$.

$$\alpha = \begin{cases} \alpha - \beta & \text{se o sinal da posição do eixo Y de } \vec{QA} \text{ for negativo} \\ \alpha + \beta & \text{caso contrário,} \end{cases} \tag{5.7}$$

Centralização dos nós filhos

A centralização dos nós é feita após o alinhamento dos mesmos. Para realizar essa atividade, é calculado o ângulo (λ) que representa a abertura da árvore. Esse ângulo é a soma de todos os ângulos α aplicados aos nós filhos. A Equação 5.8 retrata o referido cálculo e a Figura 5.17 ilustra o conceito aplicado. A importância da centralização no modelo é para fazer com que o crescimento dos nós filhos seja em direções diferentes. Caso não houvesse tal definição, a probabilidade de ocorrer sobreposição entre nós filhos de diversas árvores seria maior. Por fim, vale ressaltar que o ângulo α recebe um acréscimo do ângulo λ (conforme Equação 5.8b) para que o posicionamento dos nós filhos seja centralizado.

$$\lambda = \frac{\alpha \cdot (n-1)}{2} \tag{5.8a}$$

$$\alpha = \alpha + \lambda, \tag{5.8b}$$

onde n é a quantidade de nós filhos.

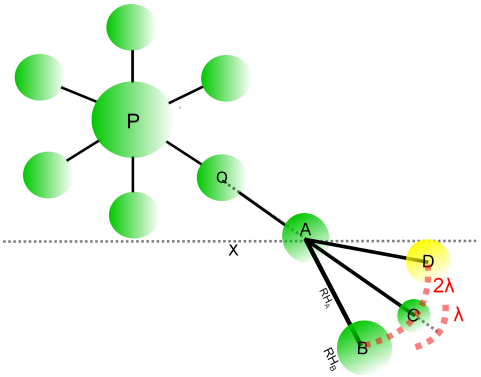


Figura 5.17: Centralização dos nós filhos na visualização do tipo árvore.

5.3.6 Analisando a interseção entre nós

Com a posição do nó gerada, é preciso que ela seja validada antes do nó ser incluído. Para que a validação ocorra, não pode haver sobreposição entre os objetos que já estão no plano e o objeto que será incluído. A Figura 5.18 mostra os possíveis casos de sobreposição entre nós. Os nós K e L estão alocados em posições inválidas e, portanto, devem receber uma nova posição válida. Este processo se repete até que uma posição válida seja encontrada. Já o nó M está em uma posição válida. A área dos nó pai com os nós filhos é chamada de AH (área hierárquica). Se o objeto que será inserido estiver posicionado dentro dessa área, sua posição é inválida, e, portanto, será gerada uma nova posição.

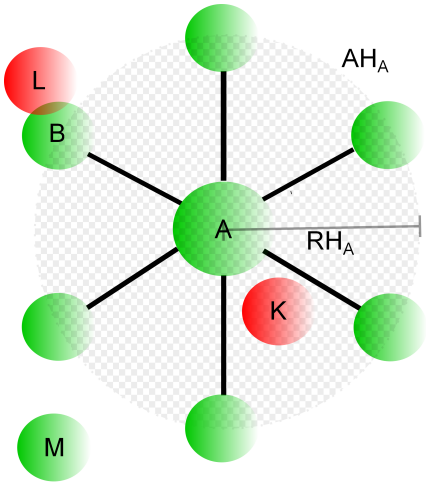


Figura 5.18: Exemplo da sobreposição de nós. Os nós K e L estão em posições inválidas. Já o nó M está em uma posição válida. A área sombreada, chamada de área hierárquica, é baseada no raio hierárquico de A.

Para verificar se ocorrerá sobreposição de nós, todos os nós existentes no plano são lidos. Então, para cada nó, é verificado se ocorre interseção entre as regiões ocupadas pelos nós considerando suas áreas hierárquicas. Caso ocorra sobreposição, outra posição deve ser gerada. Se, após efetuar a leitura de toda a lista, não for encontrado sobreposição, então essa nova posição é considerada como válida. As Equações 5.9a e 5.9b mostram

como cálculo da sobreposição é realizado.

$$D = ||P(i) - P(j)|| \tag{5.9a}$$

$$\text{Posição Válida} = \begin{cases} falso & \text{se } (D - (RH(i) + RH(j)) \leq 0 \\ verdadeiro & \text{caso contrário,} \end{cases} \tag{5.9b}$$

onde $P(i)$ é a posição do nó que será inserido, $P(j)$ é a posição do nó já inserido que será comparado à $P(i)$, D é a distância entre o centro dos pontos $P(i)$ e $P(j)$, $RH(i)$ é o raio hierárquico do ponto que está na posição $P(i)$ e $RH(j)$ é o raio hierárquico do ponto que está na posição $P(j)$. Se a diferença entre D e a soma dos raios hierárquicos for menor ou igual a zero, os dois pontos estão se sobrepondo, senão, a posição é válida. Assim que posição é considerada válida, o nó pode ser incluído no espaço.

5.4 Software para visualização

O aplicativo que implementa o modelo proposto nesta dissertação – denominado como SUVsoft – é descrito a seguir. As atividades de leitura de bibliotecas, importação, visualização e exportação de dados serão tratadas. Esse sistema possui 6 namespaces, 58 classes e 2546 linhas de código e está em produção desde Janeiro de 2012.

5.4.1 Arquitetura do SUVsoft

A arquitetura do SUVsoft é composta pelos módulos de interface, métricas, visualização, importação e exportação. Esses módulos e a comunicação entre eles estão ilustrados no fluxograma da Figura 5.19 e serão descritos a seguir.

O módulo de **Métricas** é responsável pela leitura dos arquivos do tipo *dll*, a transformação do arquivos em objetos das classes *FNamespaces*, *FClasses*, *FMethods* e *FAttributes* (conforme Seção 5.4.2), o cálculo das métricas CK e a caracterização dos valores das métricas em baixo, alto e anomalia.

O módulo **Visualização** realiza a transformação das classes em objetos gráficos de acordo com a metáfora do universo proposta no modelo. As classes são transformadas em nós e recebem as características de posição, cor e raio de acordo com os valores das métricas RFC, WMC e CBO, respectivamente. Além disso, nesse módulo é aplicado o conceito de ligação entre os nós por meio de arcos conectores e o posicionamento dos nós baseado na quantidade de métodos invocados de outras classes.

O módulo de **importação** é responsável pela leitura de um arquivo XML e a transformação desse arquivo em objetos gráficos, de acordo com as propriedades armazenadas no arquivo.

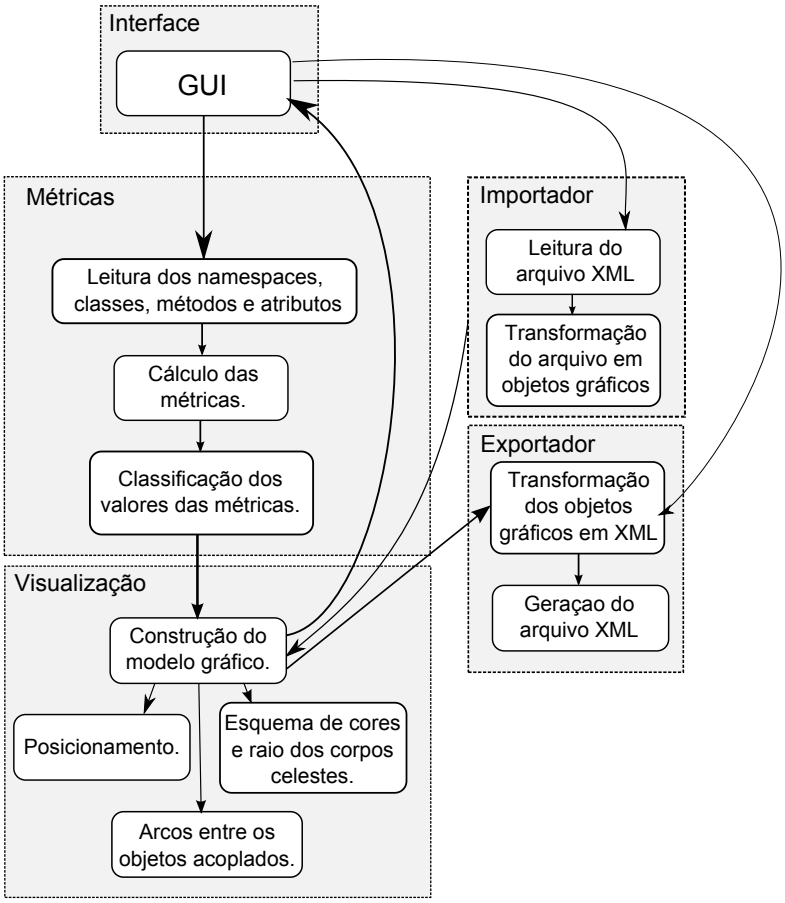


Figura 5.19: Arquitetura do SUVsoft.

E, por fim, o módulo de **exportação** faz a leitura dos objetos gráficos e exporta esses objetos para um arquivo XML, que armazena posição, cor, raio e nome dos nós.

5.4.2 Extração dos dados para a visualização

A geração do modelo pode acontecer de duas formas distintas: realizando a leitura de arquivos compilados no Framework.NET[2] ou importando informações via arquivo XML.

Quando a geração ocorre por meio da leitura de bibliotecas, uma série de características como cor, raio e posicionamento são aplicadas ao modelo com base nas informações extraídas do conjunto de métricas CK (Seção 2.3), que são calculadas pelo software.

Na importação via arquivo XML, os valores de cor e raio devem ser informados ao sistema. Já o posicionamento é facultativo – se não for informado, é gerado aleatoriamente.

Engenharia Reversa

Engenharia reversa é a atividade de analisar sistemas com o objetivo de (1) identificar componentes do sistema e suas interrelações e (2) criar a representação do sistema em uma forma diferente ou em um nível mais alto de abstração [Chikofsky e Cross 1990].

[2]http://msdn.microsoft.com/en-US/vstudio/aa496123

Assim que a biblioteca na extensão dll[3] é lida, os pacotes, classes, métodos e atributos extraídos servem como fonte de informação para a criação de objetos das classes *FNamespaces*, *FClasses*, *FMethods* e *FAttributes*, respectivamente. A Figura 5.20 ilustra o diagrama dessas classes.

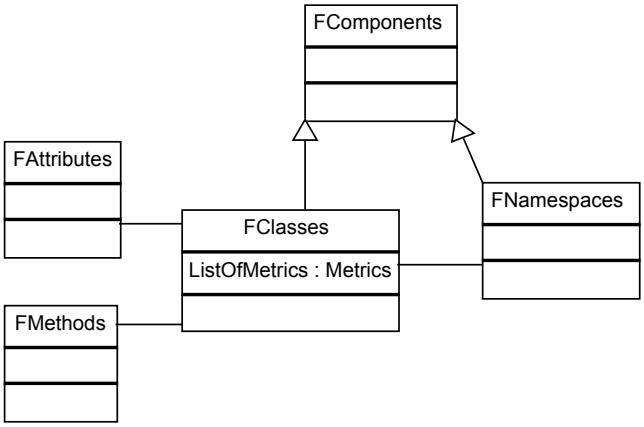


Figura 5.20: Diagrama de classes dos objetos extraídos da biblioteca.

A extração acontece por meio da biblioteca *Mono.Cecil*[4]. Essa biblioteca tem o objetivo de ler e inspecionar programas e bibliotecas escritas no formato ECMA (***E**uropean **C**omputer **M**anufacturers **A**ssociation*) CIL (***C**ommon **I**ntermediate **L**anguage*) [Mono.Cecil 2013]. A CIL, também conhecida como MSIL (***M**icro**s**oft **I**ntermediate **L**anguage*), é uma linguagem intermediária entre Framework e as linguagens de alto nível disponíveis pela Microsoft (C#, J# e Visual Basic.NET) [Sharp 2011]. Esta linguagem é intermediaria entre as linguagens suportadas pelo Framework.NET e pela CLR (***C**ommon **L**anguage **R**untime*). Graças a esse conceito, é possível interpretar bibliotecas de diferentes linguagens de programação, aumentando a abrangência do aplicativo produzido neste trabalho.

Enquanto as classes estão sendo processadas na atividade de engenharia reversa, são calculadas também as suas métricas CK (conforme Seção 2.3). As métricas de cada classe são armazenadas em uma lista de métricas, chamada de *ListOfMetrics*. Esses valores produzidos pelas métricas servirão como referência para diversos itens da visualização, tais como cor, raio e posicionamento dos corpos celestes que representam as classes.

[3]**D**ynamic-**L**ink **L**ibrary. Implementação da Microsoft para o conceito de bibliotecas compartilhadas em sistemas operacionais *Microsoft Windows* [Hart 2005].

[4]Disponível em: http://www.mono-project.com/Cecil

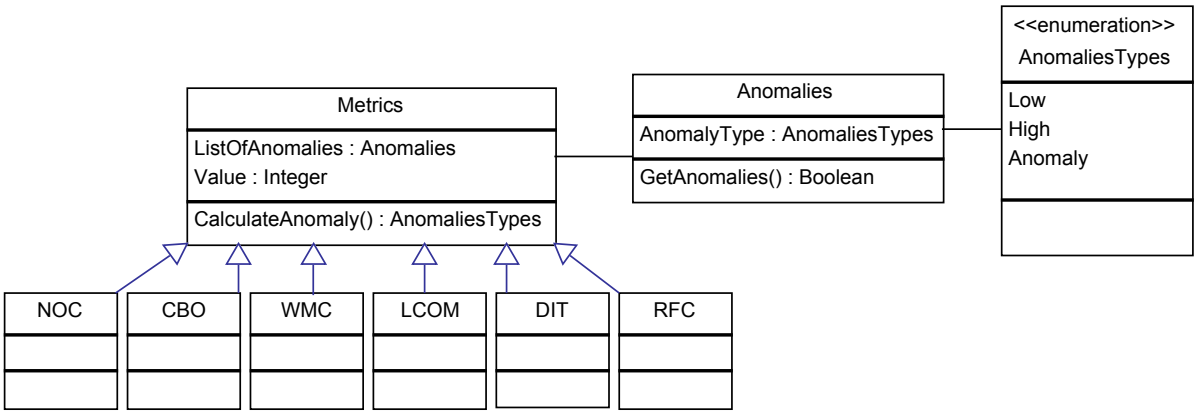


Figura 5.21: Diagrama de classe das métricas.

Importação de arquivos XML

Uma outra possibilidade para a geração do modelo gráfico é por meio da importação de arquivos XML. A estrutura do arquivo XML deve conter os seguintes critérios:

1. **Vértices:** Lista de objetos gráficos que serão representados pelos corpos celestes. Cada um dos itens do conjunto de vértices possui os atributos *Name*, *FullName*, *Color*, *Radius* e *Position* (Nome, Nome Completo, Cor, Raio e Posição).
2. **Mapas:** Ligação entre os nós, criando a relação de nó pai/nó filho. Possui as propriedades *nodeFather* e *nodeChild* (nó pai e nó filho).
3. **Acoplamento:** Ligação entre os nós, realizada por meio de arcos. Possui as propriedades *nodeFrom* e *nodeTo* (nó origem, nó destino).

A Figura 5.22 exibe um pequeno trecho do arquivo XML usado para a importação da biblioteca *AjaxControlToolkit*. Os elementos *Vertices, Maps* e *Coupling* representam os critérios vértices, mapas e acoplamento, respectivamente.

5.4.3 Construção dos objetos gráficos

Na construção dos itens de visualização, quatro componentes são frequentemente utilizados: esferas (*vtkSphereSource*), linhas (*vtkLineSource*), textos (*vtkVectorText*) e arcos (*vtkTube*).

O ambiente gráfico que realiza a metáfora com o universo é um objeto da classe *ObjectGraphicsEnvironment*, que, por sua vez, possui uma lista de objetos gráficos. A classe abstrata *ObjectGraphicsBase* é herdada pelos itens citados anteriormente, em que as esferas, linhas, textos e arcos são representadas por objetos das classes *ObjectGraphics*, *Lines*, *ObjectGraphicsTextName* e *Arcs*, respectivamente. A Figura 5.23 ilustra o diagrama de classes que compõe os objetos da visualização.

```xml
<?xml version="1.0"?>
<ListOfNodes>
  <vertices>
    <item name="Accordion" fullName="AjaxControlToolkit.Accordion" radius="10" color="0, 255, 0" />
    <item name="AccordionCommandEventArgs" fullName="AjaxControlToolkit.AccordionCommandEventArgs" radius="10" color="0, 255, 0" />
    <item name="AccordionContentPanel" fullName="AjaxControlToolkit.AccordionContentPanel" radius="10" color="0, 255, 0" />
    <item name="AccordionDesigner" fullName="AjaxControlToolkit.AccordionDesigner" radius="10" color="0, 255, 0" />
    <item name="IControlResolver" fullName="AjaxControlToolkit.IControlResolver" radius="10" color="0, 255, 0" />
    <item name="ExtenderControlBase" fullName="AjaxControlToolkit.ExtenderControlBase" radius="10" color="0, 255, 0" />
    <item name="AccordionExtender" fullName="AjaxControlToolkit.AccordionExtender" radius="10" color="0, 255, 0" />
    <item name="AccordionExtenderDesigner" fullName="AjaxControlToolkit.AccordionExtenderDesigner" radius="10" color="0, 255, 0" />
    <item name="AccordionItemEventArgs" fullName="AjaxControlToolkit.AccordionItemEventArgs" radius="10" color="0, 255, 0" />
    <item name="AccordionItemType" fullName="AjaxControlToolkit.AccordionItemType" radius="10" color="0, 255, 0" />
    <item name="AccordionPane" fullName="AjaxControlToolkit.AccordionPane" radius="10" color="0, 255, 0" />
    <item name="AccordionPaneCollection" fullName="AjaxControlToolkit.AccordionPaneCollection" radius="10" color="0, 255, 0" />
  </vertices>
  <maps>
    <item nodeFather="AjaxControlToolkit.ExtenderControlBase" nodeChild="AjaxControlToolkit.AccordionExtender" />
  </maps>
  <coupling>
    <item nodeFrom="AjaxControlToolkit.Accordion" nodeTo="AjaxControlToolkit.AccordionExtender" />
    <item nodeFrom="AjaxControlToolkit.Accordion" nodeTo="AjaxControlToolkit.AccordionPaneCollection" />
    <item nodeFrom="AjaxControlToolkit.Accordion" nodeTo="AjaxControlToolkit.AccordionPane" />
    <item nodeFrom="AjaxControlToolkit.Accordion" nodeTo="AjaxControlToolkit.AccordionContentPanel" />
    <item nodeFrom="AjaxControlToolkit.Accordion" nodeTo="AjaxControlToolkit.AccordionItemEventArgs" />
  </coupling>
</ListOfNodes>
```

Figura 5.22: Exemplo do arquivo XML de importação.

Os objetos da classe *ObjectGraphics* possuem características que auxiliam na construção do modelo. As propriedades de raio, cor, raio hierárquico e posição são armazenados nos atributos *Radius*, *Color*, *RadiusHierarquical* e *Position*, respectivamente. Vale ressaltar que a propriedade *HasFewChildren* informa que o nó possui poucos filhos, de acordo com a constante K.

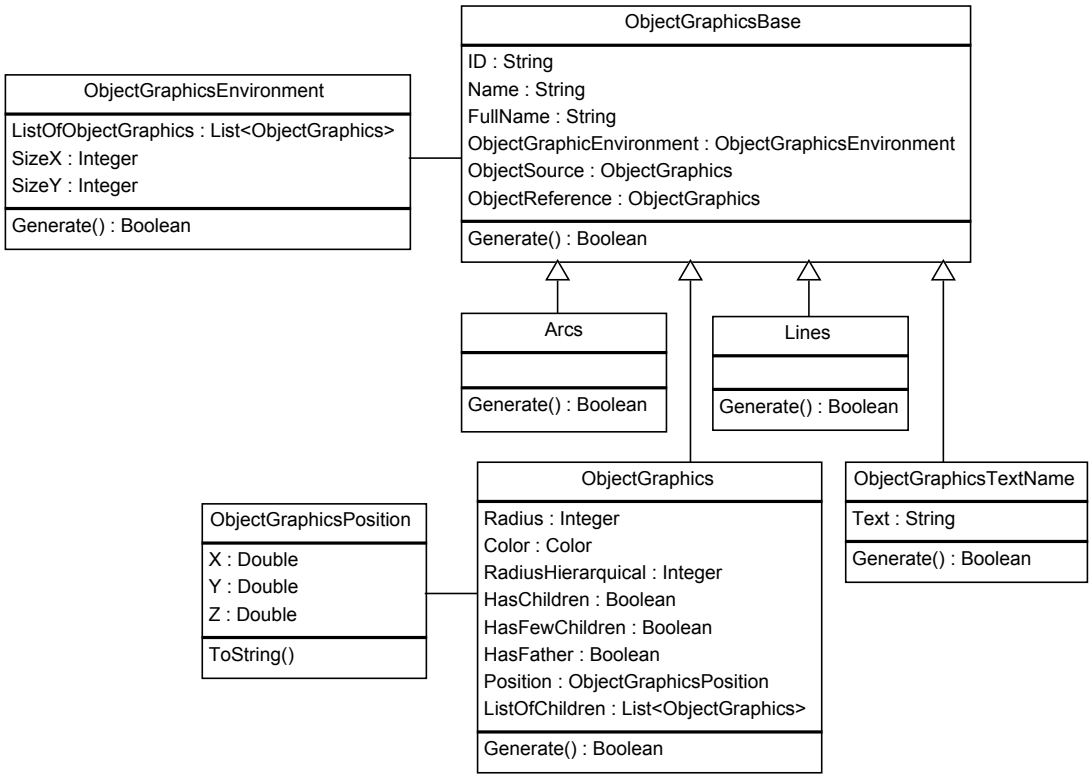
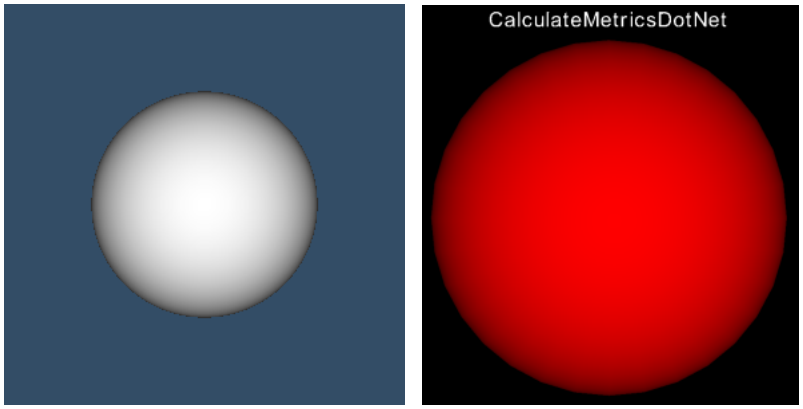


Figura 5.23: Diagrama de classe simplificado dos itens da visualização.

Esferas – Corpos celestes

As esferas, construídas com objetos da classe vtkSphereSource, representam os corpos celestes do universo. Possuem os seguintes atributos: cor, raio, posição e resolução. Cor e raio são definidas pelas métricas WMC e CBO, respectivamente. A posição é baseada na herança entre classes ou na quantidade de métodos que uma classe invoca das demais. Já a resolução é informada pelo usuário. Quanto maior for seu valor nas esferas, mais custosa fica a visualização, e consequentemente, mais lenta. A Figura 5.24 exemplifica a visualização de uma esfera gerada pelo componente em questão, onde 5.25a é o objeto vtkSphereSource com as propriedades padrão e 5.25b é o objeto vtkSphereSource com as propriedades de cor, raio, posição e resolução definidas.



(a) (b)

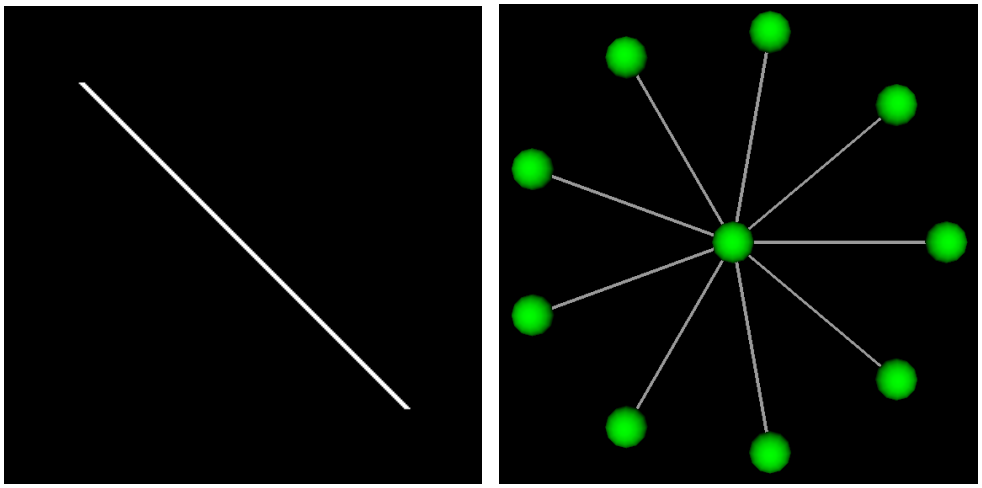
Figura 5.24: (a) vtkSphereSource padrão com COR[0.5, 0.5, 0.5] e RAIO[5]. (b) vtkSphereSource com propriedades de COR [1, 0, 0], RAIO[15] definidas por métricas.

Linhas – Herança entre classes

Para que os corpos celestes possuam alguma forma de conexão, com base na herança das classes, é utilizada a classe vtkLineSource. A Figura 5.25 exemplifica a visualização de uma linha gerada por esse componente.

Textos – Nomenclatura das classes

É possível visualizar o nome de cada uma das classes no modelo. Esta forma de visualização é opcional dentro do sistema. Para adicionar texto acima dos nós, foi utilizada a classe vtkVectorText. Assim que a posição do nó é definida, o objeto vtkVectorText é posicionado acima do nó, de forma centralizada. A Figura 5.26 mostra o resultado final da visualização com nome da classe sobre o objeto gráfico.



(a) (b)

Figura 5.25: (a) vtkLineSource padrão. (b) vtkLineSource conectando o nó pai aos nós filhos.

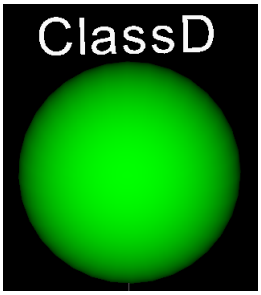


Figura 5.26: Nome das classes.

Arcos – Acoplamento entre classes

Quando uma classe utiliza métodos ou atributos, um arco é criado entre esses nós, representando a ligação que eles possuem, conforme Figura 5.27. Foi utilizado o componente vtkTube na criação do objeto que representa o acoplamento entre classes.

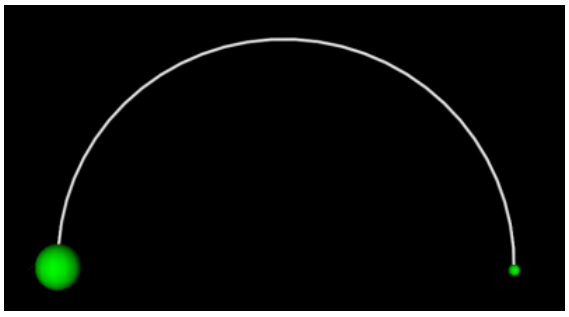


Figura 5.27: Corpos celestes ligados por um arco conector.

Para a construção do arco, é feita uma amostragem dos pontos do arco e esses pontos são conectados para formar o arco.

5.4.4 Sistema de visualização de Software

Um dos resultados desta dissertação foi a construção do software desenvolvido para implementar o modelo proposto anteriormente e o cálculo das métricas. A Figura 5.4.4 ilustra o software em funcionamento, em que as opções *Open Library*, *Import* e *Visualize* estão disponibilizadas no menu superior. No canto superior esquerdo existe uma janela com informações sobre a classe selecionada pelo usuário, por meio de um clique sobre o objeto gráfico (corpo celeste) desejado.

Para a implementação do modelo foi utilizada a biblioteca gráfica *Visualization ToolKit* (VTK)[5]. O VTK é baseado na biblioteca OpenGL[6] e é focado na visualização científica e visualização de informação. Possui diversos algoritmos de visualização em 2D e 3D, recursos de computação gráfica e processamento de imagens.

A escalabilidade do sistema é tratada por meio de aproximações/distanciações realizadas pelo usuário. Não foi analisado, nesta dissertação, um número limite que o sistema comporta. Entretanto, algumas melhorias podem ser realizadas para aprimorar a interação do usuário, conforme descrito na Seção 6.1.

O desenvolvimento do aplicativo utilizou, além da biblioteca VTK, as seguintes tecnologias. Todas as ferramentas utilizadas são *Open Source* ou gratuitas.

- *Visual Studio 2013 Express Edition* como ferramenta de desenvolvimento;
- *C# 5.0* como linguagem de programação;
- *Activiz 5.8* para geração dos objetos gráficos disponíveis na biblioteca VTK;
- *Mono.Cecil 0.9.1* para a leitura do arquivo .dll;
- *Mono.Gendarme 2.1* para o cálculo das métricas WMC e LCOM.

Manipulando características da visualização

Para que a aplicabilidade do modelo fique maleável, algumas opções são oferecidas ao usuário para que este altere opções da visualização de acordo com a necessidade (Figura 5.30). A seguir, são exibidas algumas das configurações disponíveis.

- O tamanho do espaço pode ser modificado de acordo com os parâmetros especificados nas variáveis *axis X* e *axis Y*. Quanto maior for o valor das variáveis citadas, mais disperso ficarão os objetos da visualização;
- A opção *Coupling Between Objects* habilita a visualização de arcos conectando os objetos que estão acoplados;

[5]Disponível em: http://www.kitware.com/opensource/vtk.html
[6]Disponível em: http://www.opengl.org/

- A opção *RFC Based Visualization* aplica no modelo o conceito das forças gravitacionais, aproximando as classes que estão acopladas;
- As opções do painel *Filter Classes* fornecem filtros para a visualização, exibindo apenas as classes que possuem relação de herança ou não;
- As opções do painel **Metrics** possibilita a alteração das métricas que serão visualizadas no modelo. É possível escolher entre as seis métricas propostas por [Chidamber e Kemerer 1994] para representar as características de raio e cor;
- Na aba **Metrics** (Fig. 5.29), é possível escolher o tipo de regra para anomalia. Estão disponíveis dois tipos de classificação: baseada no estudo realizado no capítulo 4 ou utilizando a regra proposta por [Lanza e Marinescu 2006] com a variação estipulada pelo usuário, com base no valor passado ao campo *Anomaly Value*.

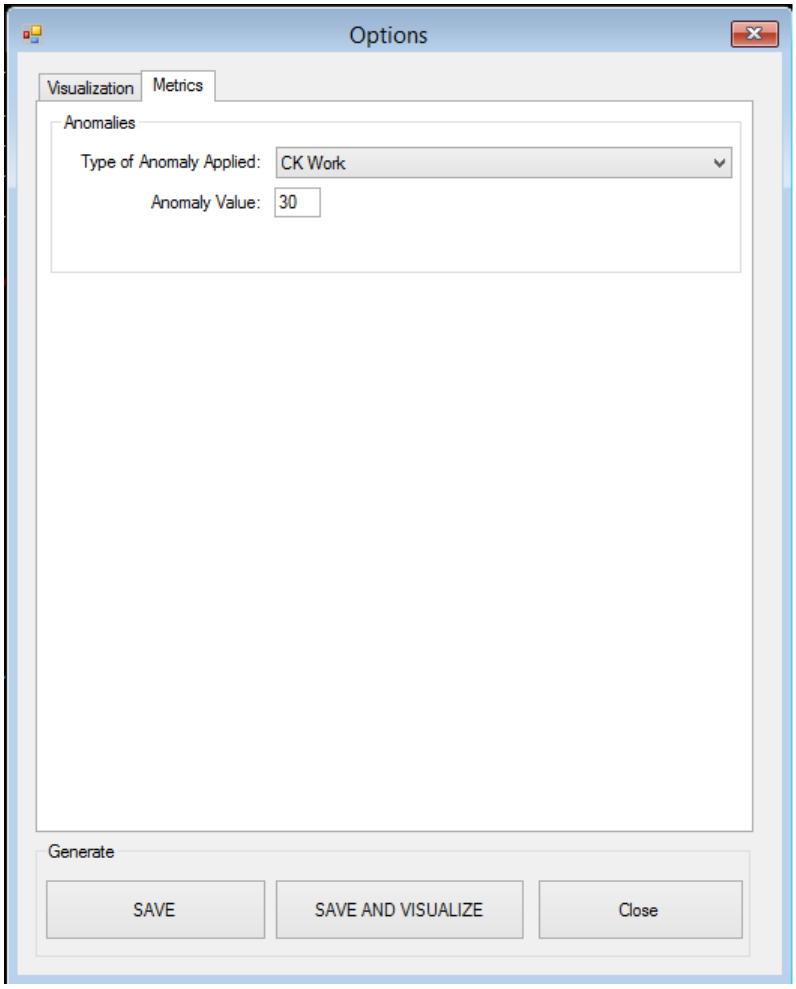


Figura 5.29: Opções das métricas do SUVsoft.

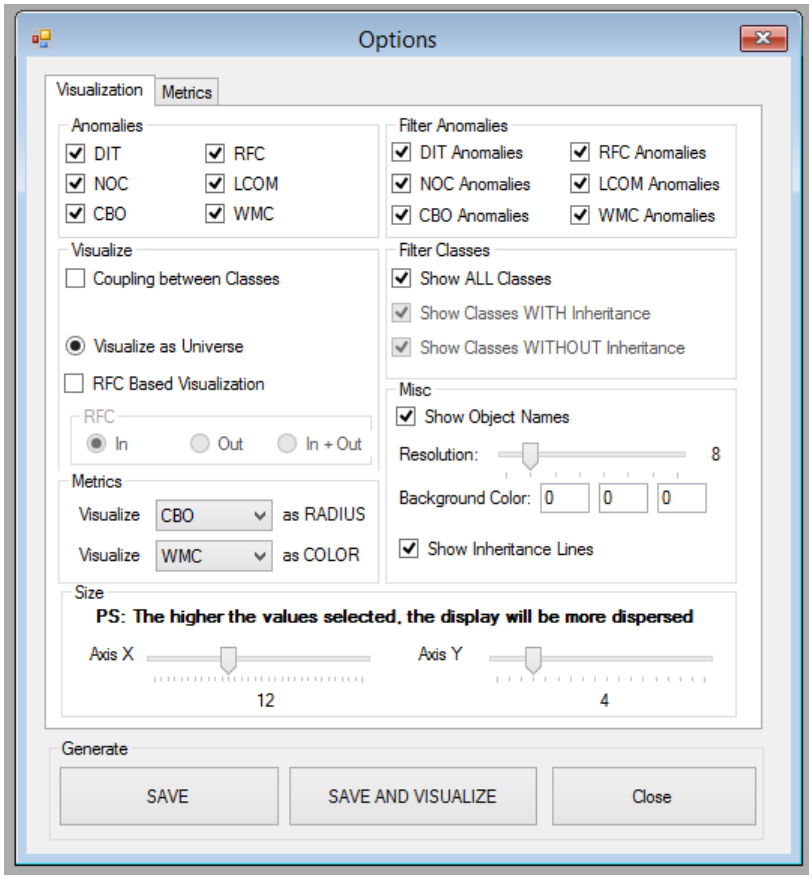


Figura 5.30: Opções do SUVsoft.

5.5 Exemplos de visualização pelo SUVsoft

Nesta seção será mostrado o resultado da visualização de alguns softwares pelo aplicativo SUVsoft. Foram escolhidos os seguintes softwares:

- **Entity Framework:**[7] 78 namespaces e 1725 classes, visualizado na Figura 5.31;
- **AjaxControlToolkit:**[8] 9 namespaces e 423 classes, visualizado na Figura 5.32, utilizando o conceito de força gravitacional;
- **SUVsoft:** 7 namespaces e 56 classes, visualizado na Figura 5.33 utilizando a visualização do acoplamento entre classes.
- **Software comercial:** 6 namespaces e 243 classes, visualizado na Figura 5.34 utilizando o conceito de força gravitacional.
- **Activiz:**[9] 1 namespace e 1763 classes, visualizado na Figura 5.35.

Esses softwares foram selecionados para mostrar algumas características do modelo de visualização.

[7]Disponível em http://entityframework.codeplex.com.
[8]Disponível em: http://ajaxcontroltoolkit.codeplex.com.
[9]Disponível em: http://www.kitware.com/opensource/avdownload.php.

No primeiro deles, a biblioteca do aplicativo Entity Framework é visualizada. Esse software é utilizado para mapeamento objeto-relacional de banco de dados e foi desenvolvido pela Microsoft. Já recebeu mais de três milhões de downloads no repositório *http://www.nuget.com*. Esse aplicativo, se comparado aos demais visualizados, é grande. Utilizando recursos de aproximação, é possível analisar partes específicas da visualização.

O segundo aplicativo visualizado é a biblioteca *AjaxControlToolkit*. O software é um conjunto de componentes que implementam algumas características de interface para aplicativos web desenvolvidos em ASP.NET e foi baixado cerca de 500.000 vezes no repositório *http://www.nuget.com*. É possível observar que existe, no centro da visualização, um grande aglomerado de classes, gerado pela força gravitacional aplicada sobre as classes que possuem relacionamento.

A terceira visualização é da biblioteca do SUVsoft, de tamanho pequeno, com uma quantidade considerável de classes que possuem herança.

Na quarta visualização, um software comercial, de tamanho médio, é visualizado. Assim como na visualização da biblioteca *AjaxControlToolkit*, é possível identificar uma área de aglomeração dos objetos, mostrando que ali existe uma força gravitacional atuando. Nesse exemplo, os arcos foram visualizados e mostram uma informação complementar sobre o acoplamento entre as classes. Além da proximidade, é possível identificar a quantidade de métodos que as classes invocam de outras classes, reforçando a informação extraída do posicionamento dos objetos gráficos.

O quinto aplicativo visualizado é o Activiz. Este sistema possui uma relação de herança entre as classes alta. O sistema tem 1763 classes e apenas três não possuem relação de herança. Quando o software possui uma grande quantidade de classes SUPER e SUB, como é o caso do Activiz, o modelo proposto neste trabalho não consegue fornecer uma visão clara de todas as métricas do software. É possível identificar que algumas classes possuem as métricas NOC e DIT com valores elevados, mas para identificar as métricas CBO e WMC, por exemplo, é necessário ampliar diversas vezes a visualização. Outra consideração é que o posicionamento dos objetos não é afetado pela força gravitacional, uma vez que a grande maioria dos objetos possui ligação direta entre si (herança).

Todos os exemplos foram analisados utilizando as métricas CBO como medida para definir o tamanho do raio e WMC para definir a cor. Esses exemplos servem somente para ilustrar algumas funcionalidades do modelo. Várias pesquisas são necessárias para tornar o modelo mais abrangente e funcional. Alguns itens, citados a seguir, devem ser estudados e melhorados para que o modelo seja consolidado.

- O algoritmo que calcula o tamanho do universo precisa ser aprimorado. Quando grande parte dos objetos possui um raio grande, a geração do modelo fica lenta e, em algumas vezes, pode travar. Isso ocorre porque o algoritmo de verificação de sobreposição de objetos não consegue encontrar posições livres no espaço.

- Por gerar posições aleatórias, até o presente momento, não foi possível realizar a visualização de um mesmo software de forma igual, exceto pela exportação da visualização para um arquivo XML. Embora as características dos objetos continuem iguais, o posicionamento é modificado a cada visualização.
- A visualização dos arcos que representam o acoplamento entre classes dificulta a visualização quando existem várias ligações entre as classes. Essa característica pode ser incrementada utilizando o conceito de HEB, criada por [Holten 2006].
- A visualização de softwares que possuem uma grande quantidade de ligação de herança entre as classes fica prejudicada. É necessário que o usuário faça diversas aproximações na visualização para conseguir extrair informações. Uma possível solução seria a diminuição da distância entre os nós pai e filhos.
- O tamanho mínimo do raio dos nós (R_{min}) precisa ser aprimorado. Quando a visualização possui poucos objetos gráficos, o raio se torna mais perceptível do que em visualização com muitos objetos gráficos. A proposta para trabalhos futuros é gerar o raio mínimo de acordo com a quantidade de objetos gráficos e do tamanho do universo.

5.6 Relacionando as métricas CK com a visualização de software

As métricas de CK foram escolhidas por dois motivos: sua aceitação diante da comunidade acadêmica [Radjenovi et al. 2013] e por ser uma medida escalável. Alguns conceitos de *Bad Smells* definem condições para que uma classe seja considerada ou não como tal. Para que a visualização fosse gradativa, era indispensável que a métrica escolhida pudesse ser quantificada escalavelmente.

As métricas de CK norteiam a visualização da seguinte forma: as métricas DIT, NOC e RFC interferem diretamente no posicionamento dos corpos celestes no universo. As duas primeiras de forma direta, alterando o tipo de visualização (árvore ou circular) e a terceira aproxima os objetos de acordo com o grau de acoplamento entre eles. As métricas CBO e WMC são ilustradas pelo raio e pela cor dos corpos celestes, respectivamente. E, por fim, o acoplamento entre as classes é representado por conexões em forma de arco no eixo Z do espaço da visualização.

O objetivo é que o usuário consiga, por meio da visualização, identificar classes que possuam valores de suas métricas discrepantes das demais, fazendo com que o entendimento e a compreensão do programa torne-se mais fácil. Futuramente, espera-se que, por meio da visualização, seja possível identificar classes que se destacam e possam estar

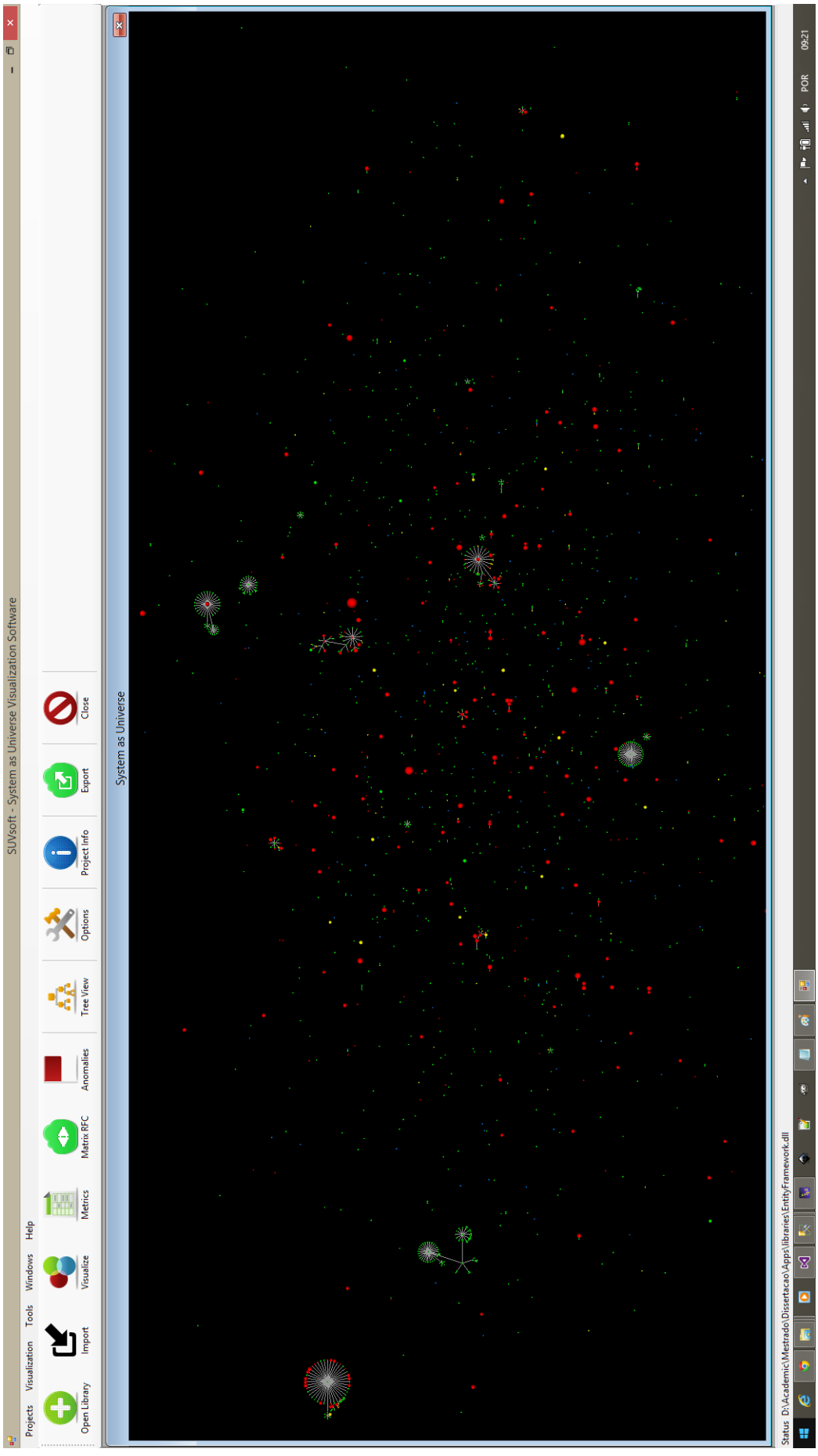


Figura 5.31: Visualização da biblioteca *Entity Framework*.

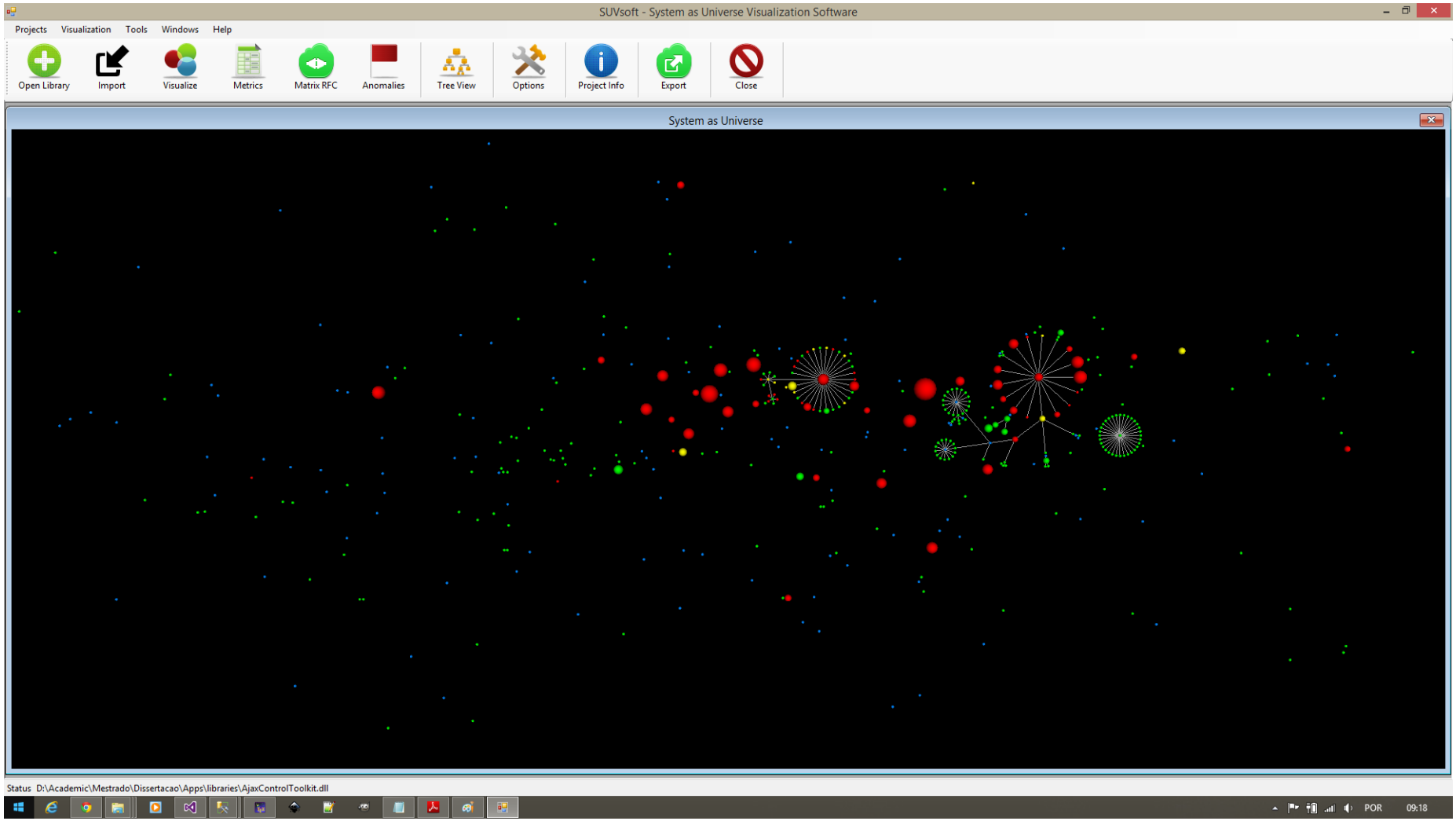


Figura 5.32: Visualização da biblioteca *AjaxControlToolkit*.

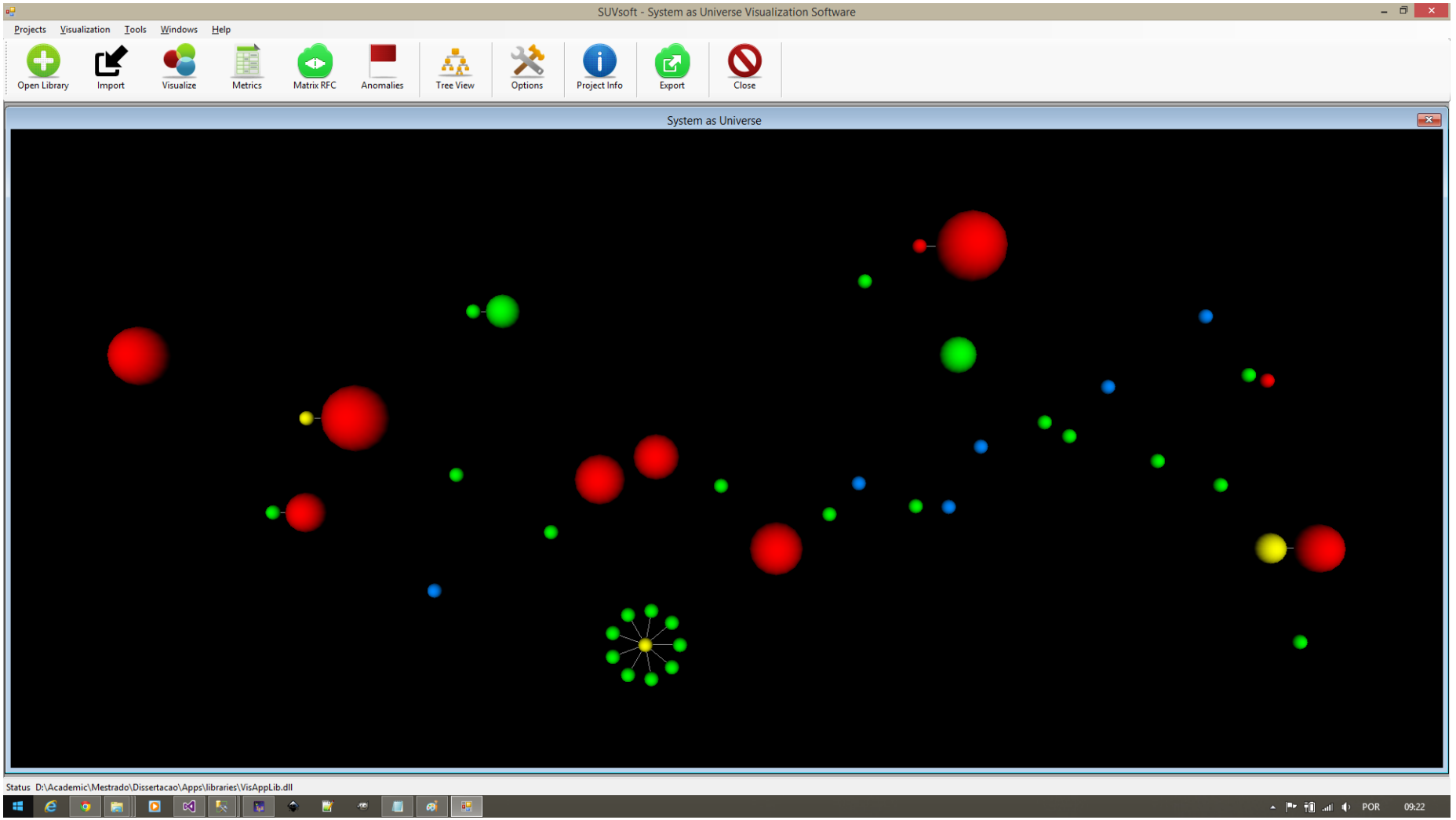


Figura 5.33: Visualização da biblioteca *SUVsoft*.

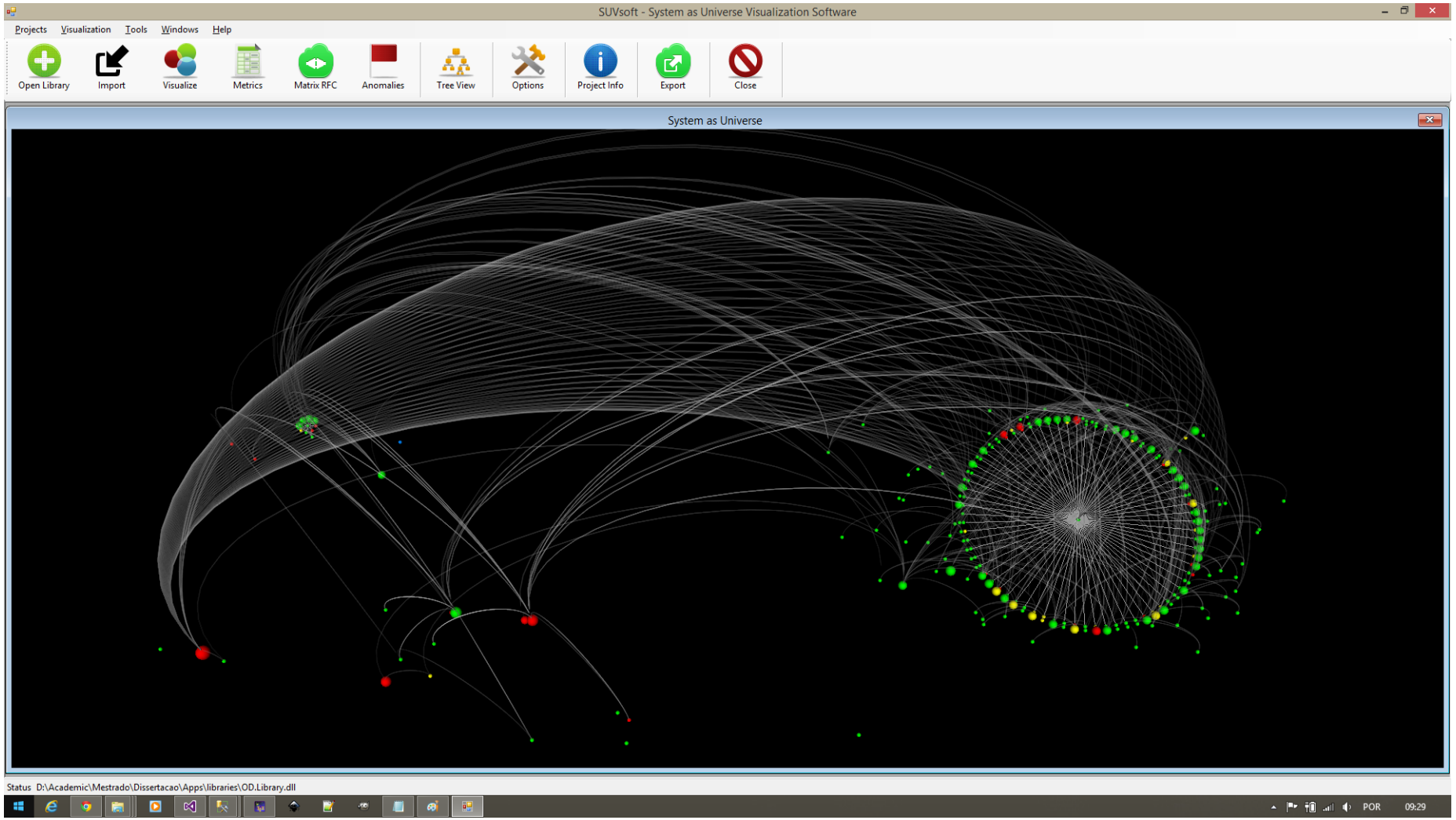


Figura 5.34: Visualização da biblioteca de um software comercial para *Business Intelligence* com o acoplamento representado pelos arcos.

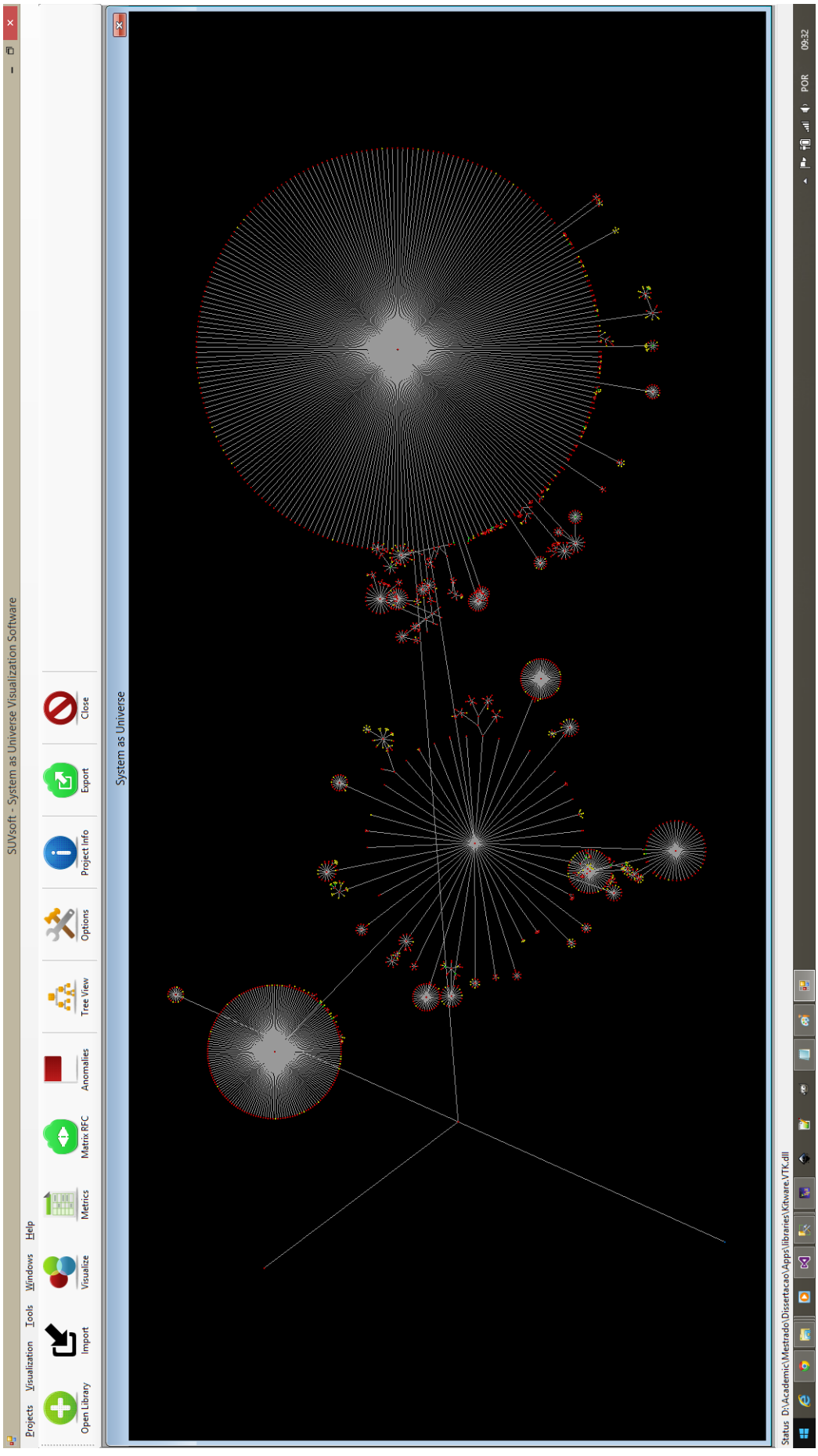


Figura 5.35: Visualização da biblioteca do Activiz, *wrapper* da biblioteca VTK para a linguagem C#.

relacionadas a algumas atividades de engenharia de software, como predição de propensão a erros, facilidade de reuso e índices de manutenibilidade e testabilidade.

Considerações Finais

Neste capítulo foi apresentado um novo modelo de visualização de software e um aplicativo – o SUVsoft – que utilizam uma metáfora do universo para a visualização de softwares. O presente modelo oferece uma alternativa para a visualização de software focada em métricas baseadas no paradigma orientado a objetos. As métricas propostas por Chidamber e Kemerer [Chidamber e Kemerer 1994] foram utilizadas para nortear o modelo de visualização. Além disso, a classificação dos valores das métricas, expostas no Capítulo 4, é utilizada para aplicar a escala de cores nos objetos gráficos.

O modelo apresenta algumas características novas como o posicionamento dos objetos gráficos com base na ligação que as classes possuem entre si, de acordo com os trabalhos correlatos apresentados no Capítulo 3. Com base no estudo apresentado no Capítulo 4, as métricas de acoplamento (CBO e RFC) possuem uma eficiência elevada se comparada as demais. No modelo apresentado neste capítulo, essas métricas são utilizadas para criar ligações entre as classes por meio de arcos e orientar o posicionamento dos objetos gráficos.

São necessárias melhorias no modelo para que este se torne mais abrangente e eficiente na atividade de compreensão de programas. Além disso, é necessária a validação do modelo, utilizando usuários de diversos níveis e em várias situações.

Capítulo 6

Conclusão

Neste trabalho foram apresentados dois estudos relacionados à métricas e visualização de software: (i) uma revisão sistemática sobre os valores das métricas CK em artigos científicos e uma análise da eficiência das métricas CK perante ao tema proposto pelos respectivos artigos e (ii) a criação de um modelo de visualização que utiliza uma metáfora simplista com o universo e orienta-se pelas métricas CK nas características de cor, tamanho (raio), posicionamento e estrutura.

No primeiro estudo foi mostrado que as métricas CBO, RFC e WMC foram eficientes, em todos os artigos científicos analisados, na realização de atividades pertinentes à engenharia de software como a predição da propensão a erros, impacto da refatoração nos valores das métricas e facilidade de reuso caixa-branca. A métrica LCOM foi eficiente em alguns casos e as métricas DIT e NOC foram eficientes em poucos casos. Esses resultados foram cruciais para a definição do modelo de visualização presente nesta dissertação.

Por fim, foi construído um modelo de visualização, baseado numa metáfora simplista do universo, para apresentar os valores das métricas de CK de diferentes formas por meio da cor, raio, conexão e posicionamento dos objetos gráficos. Um aplicativo, denominado SUVsoft, foi desenvolvido para implementar os conceitos definidos no modelo. Foram visualizados cinco aplicativos de tamanhos variados e foi possível observar os conceitos propostos pelo modelo, como a aplicação da força gravitacional, cor e raio.

Foi observado que as opções de arcos conectores e força gravitacional podem facilitar a identificação de classes que estão acoplada com as demais. Para confirmar tal afirmação é necessário expandir a validação deste modelo e utilizá-lo em diferentes tipos de software.

6.1 Trabalhos Futuros

Embora os resultados iniciais sejam promissores, são necessárias melhorias e correções, que estão descritas a seguir.

1. Validar o modelo gráfico em diferentes grupos de usuários e diferentes tipos de

software (jogos, interfaces, industriais);

2. Realizar a integração do sistema com outras ferramentas de desenvolvimento, como *Microsoft Visual Studio* ou *Mono-Project*;
3. Integrar a visualização com a atividade de compilar o código-fonte, deixando a visualização em tempo real. O objetivo é seguir a linha do trabalho de [Lanza et al. 2013], onde a visualização em tempo real mostrou-se eficiente e promissora;
4. Aplicar o conceito HEB (Seção 3.9) nas arestas que representam o acoplamento entre as classes, com o objetivo de suavizar a visualização do acoplamento;
5. Agrupar nós que estejam próximos, com o objetivo de aprimorar a escalabilidade do software. Quando um conjunto de nós, de acordo com a distância da câmera do usuário, for considerado próximo, este conjunto será substituído por um único nó, maior e mais visível;
6. Aprimorar o algoritmo de cálculo do raio para que este fique proporcional ao tamanho do universo;
7. Validar o modelo de visualização, analisando a correlação entre as classes que chamaram a atenção dos desenvolvedores com os seguintes itens:

   - A quantidade de erros que essas classes apresentaram;
   - O número de vezes que essas classes foram reutilizadas;
   - A quantidade de modificações realizada nas classes.

   Caso haja correlação entre as métricas e os itens descritos anteriormente, será possível auxiliar atividades de engenharia de software como predizer possíveis problemas no software ou indicar a dificuldade de manutenção.

6.1.1 Publicações e produções

Em termos de produtos, esta dissertação teve como resultados a publicação um artigo científico e a implementação de um software de visualização.

- "*Detection of Software Anomalies using Object-Oriented Metrics*", publicado na conferência *16th International Conference on Enterprise Information Systems - ICEIS 2014* [Juliano et al. 2014].
- Desenvolvimento do software SUVsoft (*System Universe Visualization Software*), que implementa o modelo de visualização gráfica proposta nesta dissertação e calcula as métricas DIT, NOC, CBO, RFC, WMC e LCOM, propostas por [Chidamber e Kemerer 1994].

Além dos trabalhos citados anteriormente, foi estudado o impacto que a similaridade e dissimilaridade de títulos de casos de uso podem ter no compartilhamento de linhas de código de classes que os implementam e em métricas de acoplamento, coesão e complexidade (CBO, LCOM e WMC). Descobriu-se que classes que são compartilhadas entre diferentes casos de uso tendem a ter uma baixa coesão. Além disso, classes que são compartilhadas entre casos de uso similares tendem a ser mais complexas que as demais. Não foi encontrada relação entre casos de uso similares e acoplamento ou compartilhamento de linhas de código. Essas descobertas podem guiar o desenvolvedor, seja nas fase iniciais do desenvolvimento ou nas atividades de manutenção, na construção de projetos com coesão mais alta e menor complexidade. Este estudo foi publicado na conferência *The 25th International Conference on Software Engineering and Knowledge Engineering - SEKE 2013*, com o título de "*Automated use case similarity computation can aid the assessment cohesion and method complexity of classes*" [Juliano et al. 2013].

Referências Bibliográficas

[Abreu e Carapua 1994] Abreu, F. B. e Carapua, R. (1994). Object–Oriented Software Engineering: Measuring and Controlling the Development Process. In *Object–Oriented Software Engineering: Measuring and Controlling the Development Process*, pp. 1–8, McLean, VA, USA.

[Abuasad e Alsmadi 2012] Abuasad, A. e Alsmadi, I. (2012). Evaluating the Correlation between Software Defect and Design Coupling Metrics. In *International Conference on Computer, Information and Telecommunication Systems (CITS)*, pp. 1–5.

[Ball e Eick 1996] Ball, T. e Eick, S. G. (1996). Software Visualization in the Large. *Computer*, 29(4):33–43.

[Bansiya e Davis 2002] Bansiya, J. e Davis, C. G. (2002). A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28(1):4–17.

[Bar-Yam 2003] Bar-Yam, Y. (2003). When Systems Engineering Fails - Toward Complex Systems Engineering. In *Proceedings of the International Conference on Systems, Man & Cybernetics*, volume 2, pp. 2012–2028.

[Basili et al. 1996] Basili, V., Briand, L., e Melo, W. (1996). A Validation of Object-Oriented Design Metrics as Quality Indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761.

[Benestad et al. 2006] Benestad, H., Anda, B., e Arisholm, E. (2006). Assessing Software Product Maintainability Based on Class-Level Structural Measures. In Münch, J. e Vierimaa, M. (editores), *Product-Focused Software Process Improvement*, volume 4034 de *Lecture Notes in Computer Science*, pp. 94–111. Springer Berlin - Heidelberg.

[Berry 2004] Berry, D. M. (2004). The Inevitable Pain of Software Development: Why There Is No Silver Bullet. In *Radical Innovations of Software and Systems Engineering in the Future*, Lecture Notes in Computer Science, pp. 50–74.

[Bishop e Nasrabadi 2006] Bishop, C. M. e Nasrabadi, N. M. (2006). *Pattern Recognition and Machine Learning*, volume 1. Springer New York.

[Boehm 2006] Boehm, B. W. (2006). A View of 20th and 21st Century Software Engineering. In *ICSE '06: Proceedings of the 28th International Conference on Software Engineering*, pp. 12–29.

[Booch et al. 2008] Booch, G., Maksimchuk, R. A., Engel, M. W., Young, B. J., Conallen, J., e Houston, K. A. (2008). *Object-Oriented Analysis and Design with Applications*, volume 3. Addison-Wesley, 3rd edition.

[Booch et al. 1994] Booch, G., Maksimchuk, R. A., Engle, M. W., Young, B. J., Conallen, J., e Houston, K. A. (1994). *Object-oriented Analysis and Design with Applications*. Addison-Wesley Professional, Redwood City, 3rd edition.

[Brooks 1995] Brooks, F. P. (1995). *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley Professional, Boston, USA.

[Brooks Jr. 1987] Brooks Jr., F. P. (1987). No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer*, 20(4):10–19.

[Caserta e Zendra 2011] Caserta, P. e Zendra, O. (2011). Visualization of the Static Aspects of Software: a Survey. *IEEE Transactions on Visualization and Computer Graphics*, 17(7):913–933.

[Caudwell 2010] Caudwell, A. H. (2010). Gource: Visualizing Software Version Control History. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pp. 73–74, Reno/Tahoe, Nevada, USA, New York, NY, USA. ACM.

[Charette 2005] Charette, R. N. (2005). Why Software Fails. *IEEE Spectrum*, 42(9):42–49.

[Chidamber et al. 1998] Chidamber, S. R., Darcy, D. P., e Kemerer, C. F. (1998). Managerial Use of Metrics for Object-Oriented Software: An ExploratoryAnalysis. *IEEE Transactions on Software Enggineering*, 24(8):629–639.

[Chidamber e Kemerer 1994] Chidamber, S. R. e Kemerer, C. F. (1994). A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493.

[Chikofsky e Cross 1990] Chikofsky, E. J. e Cross, J. H. (1990). Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17.

[Collins-Sussman et al. 2004] Collins-Sussman, B., Fitzpatrick, B. W., e Pilato, C. M. (2004). *Version Control with Subversion*. O'Reilly, Stanford.

[Dallal 2012] Dallal, J. A. (2012). Constructing Models for Predicting Extract Subclass Refactoring Opportunities using Object-Oriented Quality Metrics. *Information and Software Technology*, 54(10):1125–1141.

[DeMarco 1982] DeMarco, T. (1982). *Controlling Software Projects: Management, Measurement and Estimation*. Yourdon Press, New Jersey.

[Diehl 2007] Diehl, S. (2007). *Software visualization: visualizing the structure, behaviour, and evolution of software*. Springer.

[Eick et al. 1992] Eick, S. G., Steffen, J. L., e Summer Jr, E. E. (1992). Seesoft - A Tool For Visualizing Line Oriented Software Statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968.

[English et al. 2009] English, M., Exton, C., Rigon, I., Brendan, e Cleary (2009). Fault Detection and Prediction in an Open-source Software Project. In *Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, PROMISE '09, pp. 17:1–17:11, Vancouver, British Columbia, Canada, New York, NY, USA. ACM.

[Fenton e Pfleeger 1998] Fenton, N. E. e Pfleeger, S. L. (1998). *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., Boston, MA, USA, 2nd edition.

[Fry 2000] Fry, B. J. (2000). *Organic Information Design*. PhD thesis, Massachusetts Institute of Technology.

[Glass 1999] Glass, R. L. (1999). The Realities of Software Technology Payoffs. *Communications of the ACM*, 42(2):74–79.

[Graham et al. 2004] Graham, H., Yang, H. Y., e RebeccaBerrigan (2004). A Solar System Metaphor for 3D Visualisation of Object Oriented Software Metrics. In *Proceedings of the 2004 Australasian symposium on Information Visualisation-Volume 35*, pp. 53–59. Australian Computer Society, Inc.

[Greevy et al. 2006] Greevy, O., LanzaLanza, M., e Wysseier, C. (2006). Visualizing Live Software Systems in 3D. In *Proceedings of the 2006 ACM Symposium on Software Visualization*, SoftVis '06, pp. 47–56, New York, NY, USA. ACM.

[Gyimothy et al. 2005] Gyimothy, T., Ferenc, R., e Siket, I. (2005). Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction. *IEEE Transactions on Software Engineering*, 31(10):897–910.

[Harrison et al. 1998] Harrison, R., Counsell, S., e Nithi, R. V. (1998). An Investigation into the Applicability and Validity of Object-Oriented Design Metrics. *Empirical Software Engineering*, 3(3):255–273.

[Hart 2005] Hart, J. (2005). *Windows System Programming Third Edition*. Addison-Wesley.

[Holten 2006] Holten, D. H. R. (2006). Hierarchical Edge Bundles: Visualization of Adjacency Relations in Hierarchical Data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):741–748.

[Holten 2009] Holten, D. H. R. (2009). *Visualization of Graphs and Trees for Software Analysis*. PhD thesis, Eindhoven University of Technology.

[Holten et al. 2007] Holten, D. H. R., Zaidman, A., Cornelissen, B., Monnen, L., van Wijk, J. J., e van Deursen, A. (2007). Understanding Execution Traces Using Massive Sequence and Circular Bundle Views. *15th IEEE International Conference on Program Comprehension*, pp. 49–58.

[Huck 2012] Huck, S. W. (2012). *Reading Statistics and Research*. Pearson, Boston, MA, USA.

[Janes et al. 2006] Janes, A., Scotto, M., Pedrycz, W., Russo, B., Stefanovic, M., e Succi, G. (2006). Identification of Defect-Prone Classes in Telecommunication Software Systems Using Design Metrics. *Information Sciences*, 176(24"):3711–3734.

[Johari e Kaur 2012] Johari, K. e Kaur, A. (2012). Validation of Object Oriented Metrics using Open Source Software System: An Empirical Study. *SIGSOFT Software Engineering Notes*, 37(1):1–4.

[Jones et al. 2002] Jones, J. A., Harrold, M. J., e Stasko, J. (2002). Visualization of Test Information to Assist Fault Localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pp. 467–477, New York, NY, USA. ACM.

[Juliano et al. 2014] Juliano, R. C., Travençolo, B., e Soares, M. (2014). Detection of Software Anomalies using Object-Oriented Metrics. In *The 16th International Conference on Enterprise Information Systems*.

[Juliano et al. 2013] Juliano, R. C., Travençolo, B., Soares, M., e Maia, M. (2013). Automated Use Case Similarity Computation Can Aid the AssessmentCohesion and Method Complexity of Classes. In *The 25th International Conference on Software Engineering and Knowledge Engineering*, pp. 494–499.

[Kakarontzas et al. 2012] Kakarontzas, G., Constantinou, E., Ampatzoglou, A., e Stamelos, I. (2012). Layer assessment of object-oriented software: A metric facilitating white-box reuse. *Journal of Systems and Software*, 86:349–366.

[Kitchenham 2010] Kitchenham, B. (2010). Whats up with Software Metrics? A Preliminary Mapping Study. *Journal of Systems and Software*, 83(1):37–51.

[Kobayashi et al. 2013] Kobayashi, K., Kamimura, M., Yano, K., Kato, K., e Matsuo, A. (2013). SArF Map: Visualizing Software Architecture from Feature and Layer Viewpoints. *21st IEEE International Conference on Program Comprehension*, abs/1306.0958:43–52.

[Kocaguneli et al. 2010] Kocaguneli, E., Gay, G., Menzies, T., Yang, Y., e Keung, J. W. (2010). When to Use Data from Other Projects for Effort Estimation. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pp. 321–324, Antwerp, Belgium, New York, NY, USA. ACM.

[Koschke 2003] Koschke, R. (2003). Software Visualization in Software Maintenance, Reverse Engineering, and Re-engineering: A Research Survey. *Journal of Software Maintenance and Evolution: Research and Practice*, 15(2):87–109.

[Lanza 1999] Lanza, M. (1999). Combining Metrics and Graphs for Object-Oriented Reverse Engineering. Master's thesis, University of Berne, Switzerland.

[Lanza 2003] Lanza, M. (2003). CodeCrawler - Lessons Learned in Building a Software Visualization Tool. In *In Proceedings of CSMR 2003*, pp. 409–418. IEEE Press.

[Lanza et al. 2013] Lanza, M., D'Ambros, M., Bacchelli, A., Hattori, L., e Rigotti, F. (2013). Manhattan: Supporting Real-Time Visual Team Activity Awareness. In *21st International Conference on Program Comprehension (ICPC)*, pp. 207–210.

[Lanza e Marinescu 2006] Lanza, M. e Marinescu, R. (2006). *Object Oriented Metrics in Practice*. Springer, Berlin.

[Lehman 1980] Lehman, M. M. (1980). Programs, Life Cycles, and Laws of Software Evolution. *Proceedings of the IEEE*, 68(9):1060–1076.

[Li e Henry 1993] Li, W. e Henry, S. (1993). Object Oriented Metrics Which Predict Maintainability. *Journal of Systems and Software*, 23(2):111–122.

[Lorenz e Kidd 1994] Lorenz, M. e Kidd, J. (1994). *Object-Oriented Software Metrics: A Practical Guide*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[Marcus et al. 2003] Marcus, A., Feng, L., e Maletic, J. I. (2003). Comprehension of Software Analysis Data Using 3D Visualization. In *In Proceedings of the IEEE International Workshop on Program Comprehension IWPC) (2003), IEEE Computer*, pp. 105–114. Society Press.

[McCabe 1976] McCabe, T. (1976). A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320.

[McKee 1984] McKee, J. R. (1984). Maintenance as a Function of Design. In *Proceedings of the National Computer Conference and Exposition*, pp. 187–193. ACM.

[Menzies et al. 2011] Menzies, T., Butcher, A., Marcus, A., Zimmermann, T., e Cok, D. (2011). Local vs. Global Models for Effort Estimation and Defect Prediction. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pp. 343–351, Washington, DC, USA. IEEE Computer Society.

[Mono.Cecil 2013] Mono.Cecil (2013). Introducing Mono.Cecil.

[Moser et al. 2006] Moser, R., Sillitti, A., Abrahamsson, P., e Succi, G. (2006). Does Refactoring Improve Reusability? In Morisio, M. (editor), *Reuse of Off-the-Shelf Components*, volume 4039 de *Lecture Notes in Computer Science*, pp. 287–297. Springer Berlin Heidelberg.

[Nair e Selvarani 2011] Nair, T. G. e Selvarani, R. (2011). Defect Proneness Estimation and Feedback Approach for Software Design Quality Improvement. *Information and Software Technology*, 54(3):274–285.

[Ogawa e Ma 2009] Ogawa, M. e Ma, K.-L. (2009). code swarm: A Design Study in Organic Software Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1097–1104.

[Olague et al. 2007] Olague, H., Etzkorn, L., Gholston, S., e Quattlebaum, S. (2007). Empirical Validation of Three Software Metrics Suites to Predict Fault-Proneness of Object-Oriented Classes Developed Using Highly Iterative or Agile Software Development Processes. *IEEE Transactions on Software Engineering*, 33(6):402–419.

[Olbrich et al. 2009] Olbrich, S., Cruzes, D. S., Basili, V., e Zazworka, N. (2009). The Evolution and Impact of Code Smells: A Case Study of Two Open Source Systems. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM '09, pp. 390–400.

[Olbrich et al. 2010] Olbrich, S. M., Cruzes, D. S., e Sjoberg, D. I. (2010). Are all Code Smells Harmful? A Study of God Classes and Brain Classes in the Evolution of three Open Source Systems. *IEEE International Conference on Software Maintenance*, pp. 1–10.

[Pfleeger e Palmer 1990] Pfleeger, S. L. e Palmer, J. D. (1990). Software Estimation for Object-Oriented Systems. In *1990 Int. Function Point Users Group Fall Conf*, pp. 181–196.

[Pressman 2005] Pressman, S. R. (2005). *Engenharia de Software.* McGraw Hill, 6th edition.

[Pressman 2009] Pressman, S. R. (2009). *Software Engineering - A Practitioner's Approach.* McGraw Hill, 7th edition.

[Price et al. 1993] Price, B. A., Baecker, R. M., e Small, I. S. (1993). A Principled Taxonomy of Software Visualization. *Journal of Visual Languages & Computing*, 4(3):211–266.

[Radjenovi et al. 2013] Radjenovi, D., Heriko, M., Torkar, R., e Zivkovic, A. (2013). Software Fault Prediction Metrics: A Systematic Literature Review. *Information and Software Technology*, 55(8):1397–1418.

[Schiavon 2007] Schiavon, R. (2007). NGC 362. Disponível em: http://www.galex.caltech.edu/media/glx2007-03f_img01.html. Acesso em 04/10/2013.

[Sebesta 2009] Sebesta, R. W. (2009). *Concepts of Programming Languages.* Pearson, 9th edition.

[Sharp 2011] Sharp, J. (2011). *Microsoft Visual C# 2010: passo a passo.* Porto Alegre.

[Shatnawi 2010] Shatnawi, R. (2010). A Quantitative Investigation of the Acceptable Risk Levels of Object-Oriented Metrics in Open-Source Systems. *IEEE Transactions on Software Engineering*, 36(2):216–225.

[Shatnawi e Li 2008] Shatnawi, R. e Li, W. (2008). The Effectiveness of Software Metrics in Identifying Error-Prone Classes in Post-Release Software Evolution Process. *Journal of Systems and Software*, 81(11):1868–1882.

[Singh e Kahlon 2011] Singh, S. e Kahlon, K. S. (2011). Effectiveness of Encapsulation and Object-Oriented Metrics to Refactor Code and Identify Error Prone Classes using Bad Smells. *SIGSOFT Software Engineering Notes*, 36(5):1–10.

[Singh e Kahlon 2012] Singh, S. e Kahlon, K. S. (2012). Effectiveness of Refactoring Metrics Model to Identify Smelly and Error Prone Classes in Open Source Software. *SIGSOFT Software Engineering Notes*, 37(2):1–11.

[Sommerville 2009] Sommerville, I. (2009). *Software Engineering.* Addison Wesley, São Paulo, 9th edition.

[Storey 2001] Storey, M.-A. (2001). SHriMP Views: An Interactive Environment for Exploring Multiple Hierarchical Views of a Java Program. *9th International Workshop on Program Comprehension*, pp. 111–112.

[Storey et al. 1999] Storey, M.-A., Fracchia, F., e Muller, H. (1999). Cognitive Design Elements to Support the Construction of a Mental Model During Software Exploration. *Journal of Systems and Software*, 44(3):171 – 185.

[Stroggylos e Spinellis 2007] Stroggylos, K. e Spinellis, D. (2007). Refactoring – Does It Improve Software Quality? In *Fifth International Workshop on Software Quality, 2007. WoSQ'07.*

[Subramanyam e Krishnan 2003] Subramanyam, R. e Krishnan, M. S. (2003). Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects. *IEEE Transactions on Software Engineering*, 29(4):297–310.

[Tiobe 2013] Tiobe (2013). Ranking Programming Language. Disponível em: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html. Acesso em: 12/12/2013.

[Ware 2013] Ware, C. (2013). *Information Visualization: Perception for Design*. Elsevier, 3 edition.

[Wettel 2010] Wettel, R. (2010). *Software System as Cities*. PhD thesis, Università della Svizzera Italiana.

[Wettel e Lanza 2007] Wettel, R. e Lanza, M. (2007). Visualizing Software Systems as Cities. In *4th IEEE International Workshop on Visualizing Software for Understanding and Analysis, 2007.*, pp. 92–99.

[Wettel et al. 2011] Wettel, R., Lanza, M., e Robbes, R. (2011). Software systems as cities: a controlled experiment. In *Proceedings of the 33rd International Conference on Software Engineering*, pp. 551–560, Waikiki, Honolulu, HI, USA, New York, NY, USA. ACM.

[Wirth 2008] Wirth, N. (2008). A Brief History of Software Engineering. *IEEE Annals of the History of Computing*, 30(3):32–39.

[Zhou e Leung 2006] Zhou, Y. e Leung, H. (2006). Empirical Analysis of Object-Oriented Design Metrics for Predicting High and Low Severity Faults. *IEEE Transactions on Software Engineering*, 32(10):771–789.