PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS

Programa de Pós-Graduação em Informática

UMA ABORDAGEM PARA RECUPERAÇÃO DA ARQUITETURA DINÂMICA DE SISTEMAS DE SOFTWARE

Hugo de Brito Valadares Rodrigues Alves

Belo Horizonte

2011

Hugo de Brito Valadares Rodrigues Alves

UMA ABORDAGEM PARA RECUPERAÇÃO DA ARQUITETURA DINÂMICA DE SISTEMAS DE SOFTWARE

Dissertação apresentada ao Programa de Pós-Graduação em Informática como requisito parcial para obtenção do Grau de Mestre em Informática pela Pontifícia Universidade Católica de Minas Gerais

Orientador: Marco Túlio de Oliveira Valente

Co-Orientador: Humberto Torres Marques Neto

Belo Horizonte

2011

FICHA CATALOGRÁFICA
Elaborada pela Biblioteca da Pontifícia Universidade Católica de Minas Gerais

A474a

Alves, Hugo de Brito Valadares Rodrigues
Uma abordagem para recuperação da arquitetura dinâmica de sistemas de software. / Hugo de Brito Valadares Rodrigues Alves. Belo Horizonte, 2011.
85f. : il.

Orientador: Marco Túlio de Oliveira Valente
Co-orientador: Humberto Torres Marques Neto

Dissertação (mestrado) – Pontifícia Universidade Católica de Minas Gerais. Programa de Pós-Graduação em Informática.

1. Engenharia de software. 2. Métodos orientados a objetos (Computação). 3. UML (Computação). 4. Arquitetura de software I.Valente, Marco Túlio de Oliveira. II. Marques Neto, Humberto Torres. III. Pontifícia Universidade Católica de Minas Gerais. Programa de Pós-Graduação em Informática. III. Título.

CDU: 681.3.066

PUC Minas
Programa de Pós-graduação em Informática

FOLHA DE APROVAÇÃO

Uma Abordagem para Recuperação da Arquitetura Dinâmica de Sistemas de Software

HUGO DE BRITO VALADARES RODRIGUES ALVES

Dissertação defendida e aprovada pela seguinte banca examinadora:

Prof. Marco Túlio de Oliveira Valente - Orientador (UFMG)
Doutor em Ciência da Computação - UFMG

Prof. Humberto Torres Marques Neto - Co-orientador (PPGINF/PUC Minas)
Doutor em Ciência da Computação - UFMG

Prof. Carlos Alberto Marques Pietrobon (PPGEE/PUC Minas)
Doutor em Informática - PUC Rio

Prof. Mark Alan Junho Song (PPGINF/PUC Minas)
Doutor em Ciência da Computação - UFMG

Belo Horizonte, 28 de junho de 2011.

Anel Rodoviário – KM 23,5, Rua Walter Ianni, 255 – São Gabriel / Belo Horizonte – MG / CEP: 31980-110
Fone: (31) 3439-5204 – Fax: (31) 3439-5204 – e-mail: mestradoinfsec@pucminas.br

A minha família,

por sempre me apoiarem

e estarem ao meu lado.

AGRADECIMENTOS

Agradeço especialmente, aos meus pais, José Carlos Rodrigues Alves e Jane Maria de Brito, irmãos Bernardo de Brito e Douglas de Brito e à minha noiva Gabriella Campos Dutra pela compreensão, incentivo e apoio.

À Deus pela vida e por iluminar o meu caminho.

Aos professores Mark Song, Silvio Jamil, Luis Zárate, Zenilton, Carlos Pietrobon e Ana. Não poderia deixar de agradecer a Giovana Silva por sua atenção e colaboração.

Ao grupo que formamos: Ricardo Terra, Henrique Rocha e Humberto Torre.

Não tenho palavras suficientes para agradecer ao meu orientador, professor e amigo Marco Túlio de Oliveira Valente. Obrigado pelos conhecimentos transmitidos, paciência, dedicação e confiança.

À PUC Minas e a CAPES, pelo apoio a realização desta dissertação.

Muito obrigado a todos que contribuíram de alguma forma para a concretização deste trabalho.

RESUMO

Técnicas de engenharia reversa são frequentemente utilizadas para extrair uma representação da arquitetura implementada de um sistema. No entanto, essas técnicas normalmente permitem a extração apenas de diagramas que revelam a arquitetura estática de um sistema, tais como diagramas de classe. Por outro lado, o estudo da arquitetura dinâmica de um sistema pode ser particularmente útil quando se deseja formar um primeiro entendimento do funcionamento de um sistema ou avaliar o impacto de uma possível manutenção no mesmo. Sendo assim, descreve-se nesta dissertação uma abordagem que permite a recuperação de um grafo de objetos a partir da execução de um sistema existente. O grafo proposto possui diversas características que – quando combinadas – o distinguem de outras abordagens semelhantes, incluindo: (a) suporte a grupos de objetos de maior granularidade, chamados domínios; (b) suporte aos diversos tipos de relacionamentos e entidades que podem existir em sistemas orientados por objetos; (c) suporte a sistemas *multi-thread*; (d) suporte a uma linguagem para definição de alertas associados a relacionamentos que são esperados (ou que não são esperados) em um sistema. Adicionalmente, descreve-se o projeto e a implementação de uma ferramenta para visualização dos grafos de objetos propostos. Por fim, apresenta-se estudos de casos que visa demonstrar como essa abordagem pode auxiliar na compreensão da arquitetura dinâmica de três sistemas.

Palavras-chave: Engenharia Reversa. Análise Dinâmica. Recuperação Arquitetural. Compreensão de Sistemas.

ABSTRACT

Reverse engineering techniques are usually applied to extract concrete architecture models. However, these techniques usually extract diagrams that just reveal static architectures, such as class diagrams. On the other hand, the extraction of dynamic architecture models can be particularly useful for an initial understanding of how a system works or to evaluate the impact of a possible maintenance task. This dissertation describes an approach to extract hierarchical object graphs from running systems. The proposed graphs have several features that – when combined – make them valuable when compared to similar approaches, including: (a) support to the summarization of objects in domains, (b) support to the complete spectrum of relations and entities that are common in object-oriented systems, (c) support to multithreading systems, (d) support to a language to alert about expected (or unexpected) relations between the extracted objects. We also describe the design and implementation of a tool for visualizing the proposed object graphs. Finally, we show how our approach can contribute for understanding the running architecture of three systems.

Keywords: Reverse Engineering. Dynamic Analysis. Recovery and Understanding Architectural Systems.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

API - *Application Programming Interface*

DAO - *Data Access Object*

DCL - *Dependency Constraint Language*

HSQLDB - *HyperSQL DataBase*

MVC - *Model-View-Controller*

OGAL - *Object Graph Alert Language*

OGT - *Object Graph Trace*

OGV - *Object Graph Vizualization*

OMG - *Object Management Group*

OMT - *Object Modeling Technique*

OOSE - *Object-Oriented Software Engineering*

RM - *Reflection Mode*

SAVE - *Software Architecture Visualization and Evaluation*

SCHOLIA - *Static Conformance Checking Of Object-Based Structural Views of Architecture*

SGBD - *Sistema de Gerenciamento de Banco de Dados*

SQL - *Structured Query Language*

UML - *Unified Modeling Language*

SUMÁRIO

1 INTRODUÇÃO

1.1 Motivação

A Arquitetura de Software se preocupa em descrever os principais componentes de um sistema, bem como em especificar os possíveis relacionamentos entre esses componentes (SHAW; GARLAN, 1996; BASS; CLEMENTS; KAZMAN, 2003; PERRY; WOLF, 1992). Em um sistema, a arquitetura envolve um conjunto de decisões que são críticas para seu sucesso e que não poderão ser facilmente revertidas nas fases seguintes de seu desenvolvimento.

Apesar de sua importância, componentes e abstrações com valores arquiteturais não existem para a maioria dos sistemas ou, quando existem, eles não refletem a implementação atual desses sistemas (KNODEL *et al.*, 2006; CLEMENTS; SHAW, 2009; PASSOS *et al.*, 2010). Alguns motivos contribuem para a falta ou não atualização da documentação como, por exemplo, estimativa de prazo incorreta resultando em um prazo curto para análise do sistema ou a própria cultura da empresa que na fase de análise reduz o tempo por considerar que é uma fase que não necessita despender muito tempo, o que implica em uma documentação superficial do sistema.

O problema relacionado a documentação fica evidente quando estes sistemas necessitam de manutenções. Neste caso, os desenvolvedores de sistemas recorrem a três possíveis soluções: (a) Ajuda de especialistas: entrar em contato com um ou mais especialistas que tenham conhecimento do sistema para que possam detalhar o sistema. Tem como vantagem o rápido levantamento das informações sobre o sistema, porém as possibilidades de contar com ajuda de especialistas são muito reduzidas para sistemas legados e/ou sistemas de código aberto. (b) Engenharia reversa: buscar ferramentas que recuperam informações para auxiliar no entendimento do sistema. (c) Análise manual: quando as duas opções acima não estão disponíveis e/ou caso a documentação seja insuficiente, o desenvolvedor é obrigado a compreender o sistema tendo somente o código fonte como o principal artefato a ser consultado.

O desenvolvedor pode não ter tempo suficiente para aguardar um especialista (caso exista) ou fazer uma análise detalhada do código. Uma solução para este problema é a engenharia reversa. A engenharia reversa permite ao desenvolvedor recuperar informações providas do artefato mais confiável de um sistema, o código fonte. Com a utilização da engenharia reversa, é possível extrair informações arquiteturais de um sistema (DUCASSE; POLLET, 2009; TONELLA, 2005). Por meio da conformação arquitetural é possível verificar se a arquitetura planejada está de acordo com o código fonte. Entretanto, mesmo com a utilização de técnicas de engenharia reversa, um desafio para compreensão de sistemas é determinar o nível de detalhes que o desenvolvedor necessita. Isto é, o que não é relevante (devendo ser ocultado) ou que é relevante (devendo ser exibido) para auxiliar no entendimento. Várias abordagens geram uma única visão do sistema, mas uma única visão pode não ser o suficiente. Então, para aumentar o nível de compreensão podem ser necessárias diversas visões ou perspectivas arquiteturais diferentes (KRUCHTEN, 1995; HOFMEISTER; NORD; SONI, 1999; SARTIPI; KONTOGIANNIS; MAVADDAT, 2000; RIVA; RODRIGUEZ, 2002; BOOCH; RUMBAUGH; JACOBSON, 2005).

Por exemplo, um desenvolvedor pode ter conhecimento de parte do sistema, então prefere ocultar ou sumarizar esta parte e deixar as partes que tem um menor conhecimento com uma granularidade fina, isto é, com mais detalhes. Por outro lado, um desenvolvedor pode não ter conhecimento nenhum do sistema e prefere mais detalhes. À medida que vai explorando o sistema e tomando conhecimento então começa a ocultar ou sumarizar partes que não considera importantes para o seu entendimento. Isto se deve ao conhecimento prévio que se tem do sistema (muito ou nenhum) e como cada pessoa trata a informação. O importante a ser ressaltado nestes dois exemplos é que sistemas podem gerar um grande volume de informação – mesmo para uma funcionalidade dependendo do porte do sistema. A manipulação dessas informações tende a variar de pessoa para pessoa. Este desafio também se aplica aos diagramas da UML. Portanto, a sumarização e ocultamento de informação necessitam ser simples e práticos para que cada desenvolvedor possa manipular as informações de acordo com suas necessidades.

A engenharia reversa pode ser realizada utilizando duas técnicas: Análise Estática e Análise Dinâmica. A engenharia reversa utilizando análise estática permite extrair diagramas que revelam a arquitetura estática de um sistema, tais como diagramas de classes (FOWLER, 2003), diagramas de pacotes (FOWLER, 2003) e matrizes de dependência estrutural (SANGAL *et al.*, 2005). Esses diagramas têm como principal vantagem o fato de serem recuperados diretamente do código fonte do sistema (isto é, sem a necessidade de execução). No entanto, esses diagramas apresentam uma visão parcial dos relacionamentos

que são estabelecidos durante a execução de um sistema (ABI-ANTOUN; ALDRICH, 2009a; BRIAND; LABICHE; LEDUC, 2006; JACKSON; WAINGOLD, 2001).

A engenharia reversa utilizando análise dinâmica permite extrair diagramas que revelam a arquitetura dinâmica de um sistema, tais como diagramas de objetos e de sequência (FOWLER, 2003). Como principal vantagem, essas alternativas permitem expressar o fluxo de execução de um sistema, capturando inclusive relacionamentos decorrentes de chamadas dinâmicas e de reflexão computacional (SCHMERL *et al.*, 2006; YAN *et al.*, 2004). No entanto, esses diagramas normalmente não são escaláveis, apresentando milhares de objetos mesmo para sistemas de pequeno porte (ABI-ANTOUN; ALDRICH, 2009a, 2009b). Um diagrama escalável fornece meios de sumarização para torná-lo mais legível, ao contrário de um diagrama que não é escalável, o qual apresenta muitas informações que o torna ilegível. Além disso, modelos dinâmicos normalmente não fazem distinção entre objeto de baixo nível (exemplo: um objeto da classe `java.util.Date`) e objeto de maior valor arquitetural (exemplo: uma coleção de objetos do tipo `Customer`). As soluções propostas para melhorar a escalabilidade de diagramas dinâmicos se baseiam em um mesmo princípio: agrupar objetos em unidades de maior granularidade (normalmente denominadas domínios (ABI-ANTOUN; ALDRICH, 2009a), componentes (MEDVIDOVIC; TAYLOR, 2000), clusters (ANQUETIL; FOURRIER; LETHBRIDGE, 1999), etc), de forma a permitir uma visão hierarquizada do diagrama.

1.2 Objetivos

Esta dissertação de mestrado tem como objetivo principal apresentar uma abordagem que por meio de engenharia reversa forneça meios para compreensão de sistemas. Os objetivos específicos do trabalho são descritos a seguir:

a) Criar uma abordagem que suporte visões arquiteturais em diferentes níveis de granularidade. Dessa maneira, a abordagem permite ao desenvolvedor possuir uma visão do sistema em um momento com mais detalhes e em outro momento com menos detalhes;

b) Criar uma linguagem para a definição dos domínios para possibilitar ao desenvolvedor o controle dos objetos por meio da sumarização. Com os domínios será possível o desenvolvedor criar diferentes perspectivas arquiteturais do sistema;

c) Criar uma linguagem de alerta para destacar os relacionamentos entre os objetos. Com isto, o desenvolvedor poderá conhecer mais detalhes da arquitetura,

já que com a linguagem é possível associar alertas a relacionamentos que são esperados (ou que não são esperados) em um sistema;

d) Criar uma ferramenta em modo *on-the-fly* que permita ao desenvolvedor visualizar e manipular o resultado de forma rápida e prática. A expressão *on-the-fly* quer dizer dinamicamente, isto é, em tempo de execução;

e) Avaliar a abordagem proposta nesta dissertação. Para avaliação, será feito uma comparação com os diagramas da UML para distinguir os pontos fortes e fracos. Outra avaliação em sistemas reais para demonstrar se a abordagem consiste em uma solução viável.

1.3 Visão Geral da Solução Proposta

Nesta dissertação, apresentaremos uma abordagem que utiliza engenharia reversa por meio da análise dinâmica que permite a recuperação de um grafo de objetos a partir da execução de um sistema existente. O grafo proposto possui diversas características que o distinguem de outras abordagens semelhantes, incluindo:

a) Suporte a grupos de objetos de maior granularidade, chamados domínios, os quais são definidos de forma não-invasiva, por meio de uma linguagem de expressões regulares;

b) Suporte aos diversos tipos de relacionamentos e entidades que podem existir em um sistema orientado por objetos, incluindo relacionamentos devido a chamadas dinâmicas, reflexão computacional e relacionamentos entre objetos e campos estáticos de classes;

c) Suporte a sistemas *multi-thread* por meio do uso de cores para diferenciar objetos criados por *threads* diferentes;

d) Suporte a uma linguagem para definição de alertas associados a relacionamentos que são esperados (ou que não são esperados) em um sistema.

Adicionalmente, descreve-se o projeto e a implementação de uma ferramenta para visualização dos grafos de objetos. Essa ferramenta foi projetada de forma que ela pode ser "acoplada" a um sistema existente, permitindo assim a visualização do grafo de objetos à medida que o programa base está sendo executado (ou seja, diferentemente de outras

abordagens, não é gerado um arquivo com o rastro da execução, que somente então será exibido no final da execução do sistema). A ferramenta permite também gerar mais de uma visão do sistema. Por fim, apresenta-se um estudo de caso que demonstra como a ferramenta pode auxiliar na compreensão da arquitetura dinâmica de três sistemas.

1.4 Estrutura da Dissertação

A dissertação está organizada conforme descrito a seguir:

a) No Capítulo 2, apresenta-se uma revisão da literatura e discutem-se os trabalhos relacionados;

b) No Capítulo 3, descrevem-se o grafo de objetos proposto, incluindo uma descrição de seus principais elementos e descreve-se a linguagem de alertas, usada para notificar os desenvolvedores de sistemas sobre o estabelecimento de relacionamentos que são desejados (ou que não são desejados) em um sistema. Por fim, descreve a ferramenta para visualização e manipulação do grafo de objetos proposto nesta dissertação de mestrado;

c) No Capítulo 4, são comparados diagramas da UML com a abordagem proposta nesta dissertação;

d) No Capítulo 5, são apresentados os estudos de casos, incluindo os sistemas *JHotDraw*, *myAppointments* e a própria ferramenta proposta nesta dissertação;

e) No Capítulo 6, conclui a dissertação, resumindo as principais contribuições dos grafos de objetos propostos e delineando possíveis linhas de trabalhos futuros.

2 REVISÃO DA LITERATURA

Neste capítulo é apresentado a revisão da literatura sobre UML (*Unified Modeling Language*), Arquitetura de Software e Compreensão de Programas. A *UML* possui diagramas para representação da arquitetura e diagramas para compreensão de funcionalidade. Por esta razão, utilizamos diagramas da UML para compararmos (conforme a Seção 4) com a abordagem apresentada nesta dissertação. Na seção a seguir, apresenta-se uma breve história da UML e seus principais diagramas.

2.1 UML

A UML foi concebida por James Rumbaugh e Grady Booch unificando o método OMT (*Object Modeling Technique*), método do *Booch* e outros métodos. Depois de criado o primeiro esboço, Ivar Jacobson juntou-se ao grupo e foi incorporado o método OOSE (*Object-Oriented Software Engineering*). A primeira versão com o nome UML foi a versão *0.9* e no final de 1997 a OMG (*Object Management Group*) aprovou a UML. Com a aprovação a OMG ficou responsável pelo desenvolvimento e padronização da linguagem. Atualmente a UML encontra-se na versão *2.3*.

A Linguagem de Modelagem Unificada (UML) é um conjunto de notações gráficas que auxiliam a visualizar, especificar, construir e documentar sistemas de software (BOOCH; RUMBAUGH; JACOBSON, 2005; FOWLER, 2003). Para novos sistemas a UML fornece diversos diagramas para o usuário projetar permitindo verificar se o seu projeto esta completo e correto. Para sistemas legados, que não possuem documentação ou possuem documentação defasada pode-se recuperar a documentação utilizando técnicas de engenharia reversa sendo gerados alguns diagramas da UML como, por exemplo, diagrama de classe, pacote ou sequência.

No total a UML possui treze diagramas que são classificados em estrutural e comportamental (conforme a Figura 1):

a) *Diagramas Estruturais:* tem como objetivo visualizar e documentar aspectos estáticos do sistema (BOOCH; RUMBAUGH; JACOBSON, 2005). Os diagramas que pertencem à classificação estrutural são os diagramas de classe, componentes, objetos, implantação, pacotes e estruturas compostas;

b) *Diagramas Comportamentais:* tem como objetivo visualizar e documentar aspectos dinâmicos do sistema (BOOCH; RUMBAUGH; JACOBSON, 2005). Os diagramas que pertencem a classificação comportamental são os diagramas de sequência, comunicação, caso de uso, estado, atividade, temporização e visão geral da interação.

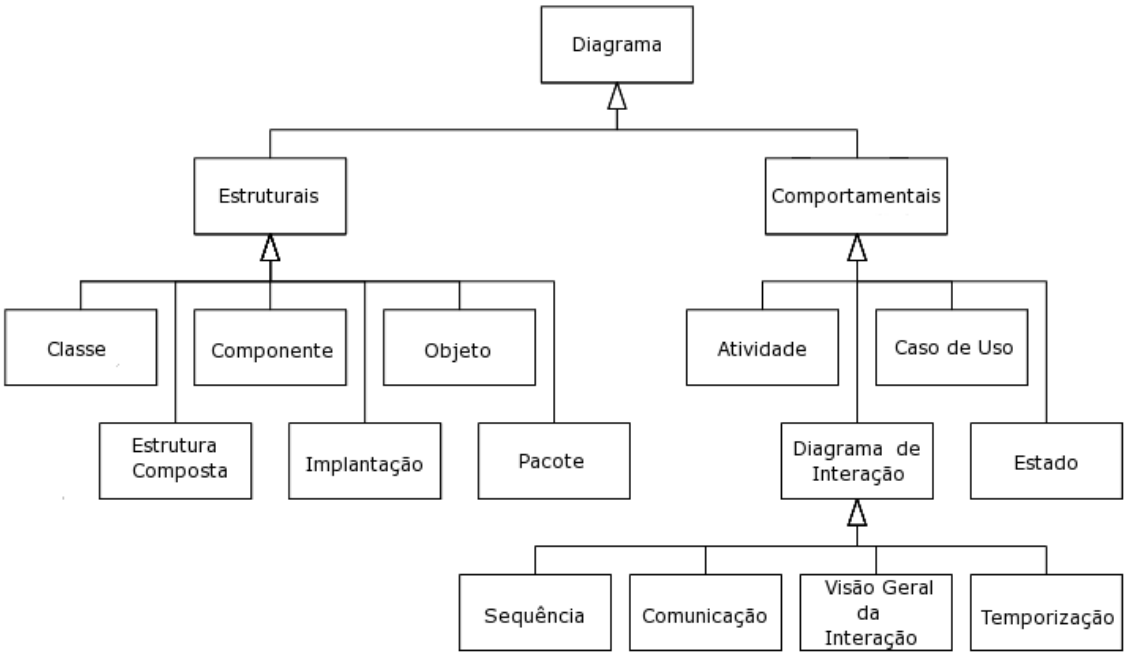

Figura 1: Classificação dos diagramas da UML
Fonte: FOWLER, 2003

A seguir, cada um desses diagramas é discutido de forma resumida.

2.1.1 Diagrama de Classe

Considerado o diagrama mais utilizado na UML, o Diagrama de Classe tem como objetivo representar propriedades, operações e relacionamentos das classes (BOOCH; RUMBAUGH; JACOBSON, 2005; FOWLER, 2003). Pode ser recuperado estaticamente e possui a vantagem de gerar uma visão estrutural completa do sistema, isto é, captura todas as classes e relacionamentos estáticos entre elas resultando em uma visão estrutural do sistema.

2.1.2 Diagrama de Pacotes

Sistemas de médio a grande porte envolvem centenas ou milhares de classes. O diagrama de pacote auxilia a organização e o controle da visibilidade das classes (BOOCH; RUMBAUGH; JACOBSON, 2005; FOWLER, 2003). O diagrama de pacote pode ser recuperado por meio da análise estática ou por meio da análise dinâmica. Na Seção 5.3.1, apresenta-se um diagrama de pacote recuperado por meio de análise dinâmica.

2.1.3 Diagrama de Objetos

O diagrama de objetos representa um conjunto de objetos (com os valores de seus atributos) e seus relacionamentos em um ponto da execução do sistema – é como registrar uma foto dos objetos do sistema em um determinado momento (BOOCH; RUMBAUGH; JACOBSON, 2005; FOWLER, 2003). Para exemplificar um diagrama de objeto, a Figura 2 representa objetos *c* do tipo *Company* que relacionam com dois departamentos (objetos *d1* e *d2*). Percebe-se que os departamentos possuem um atributo chamado *name*, com valores diferentes para os objetos *d1* e *d2*. A diferença dos dois objetos *d1* e *d2* não é somente os valores dos atributos, mas também o fato de o objeto *d1* se relacionar com outro objeto *d3* do tipo *Department*.

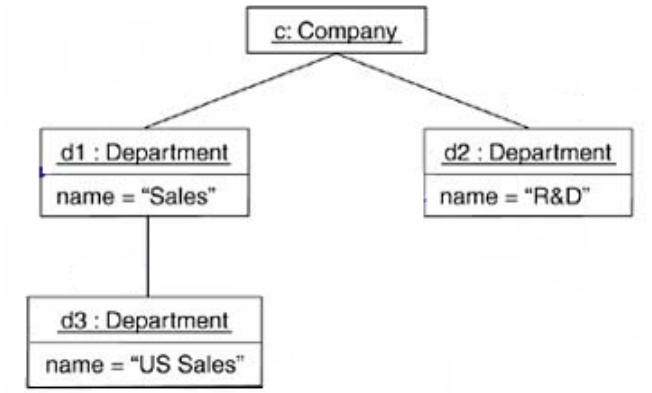

Figura 2: Diagrama de Objetos
Fonte: Adaptado de BOOCH; RUMBAUGH; JACOBSON, 2005

2.1.4 Diagrama de Sequência

O Diagrama de Sequência é classificado como diagrama de interação e tem como objetivo demonstrar uma sequência de mensagens entre as classes – enfatiza a ordem temporal (BOOCH; RUMBAUGH; JACOBSON, 2005; FOWLER, 2003). Pode ser recuperado

dinamicamente e possui como vantagem a variedade de representações possíveis como: condição, chamada assíncrona, estrutura de repetição (*loop*), etc.

2.1.5 Diagrama de Comunicação

O Diagrama de Comunicação é classificado como diagrama de interação. Possui conceitos semelhantes ao do Diagrama de Sequência, devido a essas semelhanças algumas ferramentas *case* transformam um diagrama para outro. A diferença está no foco de cada diagrama. O Diagrama de Comunicação enfatiza organização estrutural dos objetos (BOOCH; RUMBAUGH; JACOBSON, 2005). O diagrama de comunicação era denominado diagrama de colaboração nas versões anteriores a 2.0 da UML. A Figura 3 apresenta um diagrama de comunicação que possui três objetos sendo representados pelas classes *Client*, *Transaction* e *ODBDProxy*. Sobre a comunicação pode-se observar que o primeiro método *create* é invocado pela classe *Client* e referência a classe *Transaction*. No segundo momento a classe *Client* invoca o método *setActions* da classe *Transaction* e por último a classe *Transaction* invoca o método *setValues* da classe *ODBDProxy*.

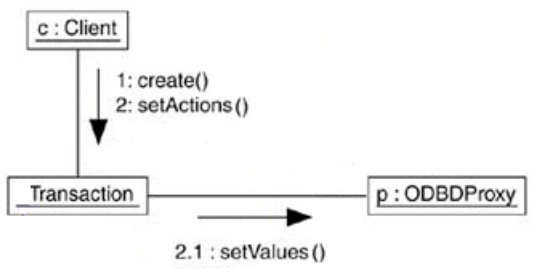

Figura 3: Diagrama de Comunicação
Fonte: BOOCH; RUMBAUGH; JACOBSON, 2005

2.1.6 Considerações Finais

A UML, com seus treze diagramas, possibilita ao usuário modelar vários aspectos de um sistema de software, isto é, o usuário pode escolher qual aspecto especificar de um sistema (estrutural e/ou comportamental) e o nível de detalhes que pretende adotar para cada diagrama escolhido. A UML possui outras vantagens como ser independente de processo de software (adapta-se melhor para processo que é orientado a caso de uso, iterativo, incremental e com arquitetura centralizada) e de linguagem de programação (mesmo com o propósito principal sendo para linguagem de programação orientada a objetos) (BOOCH; RUMBAUGH; JACOBSON, 2005). Por estas razões, a UML se tornou

a linguagem mais conhecida e utilizada para especificação e construção de sistemas de software.

2.2 Arquitetura de Software

Os sistemas de software estão cada vez mais presentes tornando-se mais importantes em nossas vidas. Por isto, a demanda por manutenções nestes sistemas cresce na mesma proporção. Para fazer manutenção de um sistema é muito importante compreender como o sistema funciona e como é a sua arquitetura.

Arquitetura de Software atualmente é uma área que se destaca na engenharia de software. Existem vários conceitos sobre arquitetura de software e utilizamos nesta dissertação o seguinte: Arquitetura de software se preocupa em descrever os principais componentes de um sistema, bem como em especificar os relacionamentos possíveis entre esses componentes (SHAW; GARLAN, 1996; BASS; CLEMENTS; KAZMAN, 2003; PERRY; WOLF, 1992). Possuir um projeto arquitetural apresenta diversos benefícios para um sistema como, por exemplo, possibilitar a reutilização dos componentes do sistema, facilitar a interoperabilidade, tornar o sistema escalável, possibilitar que as equipes trabalhem em módulos diferentes e facilitar a manutenção do sistema.

Para compreender a arquitetura de um sistema, o desenvolvedor pode precisar de várias visões ou perspectivas arquiteturais diferentes e ao mesmo tempo complementares (KRUCHTEN, 1995; HOFMEISTER; NORD; SONI, 1999; SARTIPI; KONTOGIANNIS; MAVADDAT, 2000; RIVA; RODRIGUEZ, 2002). Com várias perspectivas, o desenvolvedor pode obter mais informações aumentando a chance de compreender o sistema ou uma determinada funcionalidade. Inclusive, estas diversas perspectivas (ou visões) é uma das vantagens da UML. Para classificar os tipos de visões arquiteturais, podemos destacar dois modelos: modelo de Kruchten (KRUCHTEN, 1995) e modelo de Hofmeister (HOFMEISTER; NORD; SONI, 1999).

O modelo de Kruchten apresenta cinco visões arquiteturais, conforme apresentado na Figura 4. As visões propostas pelo modelo são:

a) *Visão Lógica:* trata o mapeamento dos requisitos funcionais e abstrações do domínio do problema. Pode ser descrito pela UML por meio de diagramas de classe e pacote, já que o objetivo dessa visão é descrever uma visão estática da arquitetura;

b) *Visão de Processo:* foca em requisitos não-funcionais como desempenho, tolerância a falhas e como as principais abstrações da visão lógica estão se relacionando. Pode ser descrito pela UML por meio do diagrama de caso de uso (para capturar os requisitos não-funcionais) e o diagrama de classe ou componentes para apresentar uma visão dos relacionamentos entre as principais abstrações do modelo;

c) *Visão de Desenvolvimento:* descreve a organização do software em módulos ou subsistemas. Esta visão auxilia na alocação de equipes para trabalhar em módulos específicos, na portabilidade e reutilização. Pode ser descrito pela UML por meio do diagrama de classe com nível de abstração maior, diagrama de pacote ou diagrama de componentes;

d) *Visão Física:* descreve os requisitos não-funcionais como disponibilidade, desempenho e escalabilidade mapeando o software para o hardware. Pode ser representado pela UML por meio do diagrama de implantação;

e) *Visão de Cenário:* descreve o comportamento do sistema (em caso de uso) utilizando as quatro visões arquiteturais apresentadas acima. Tem como objetivo descobrir elementos arquiteturais na fase de concepção. Quando o projeto encontra-se finalizado, o foco passar a ser validar e demonstrar o projeto da arquitetura. Pode ser representado pela UML por meio do diagrama de caso de uso.

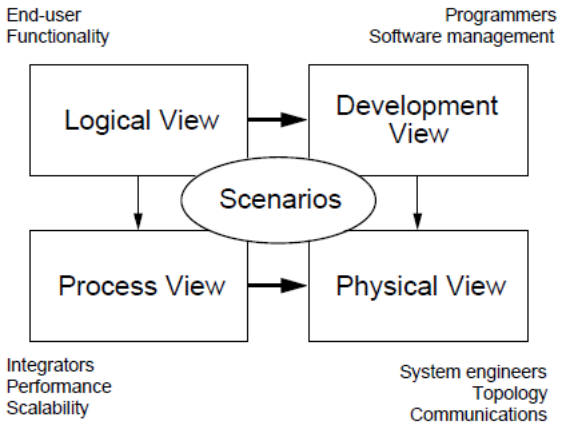

Figura 4: Tipos de Visões Arquiteturais
Fonte: KRUCHTEN, 1995

O modelo do Hofmeister (HOFMEISTER; NORD; SONI, 1999) propõem quatro visões que são:

a) *Visão Conceitual:* descreve a arquitetura em elementos dos domínios e características funcionais do sistema. Trata-se de uma visão importante para a evolução e reutilização. Pode ser representado pela UML por meio do diagrama de classe para representação estática e com o diagrama de sequência para representação comportamental do sistema;

b) *Visão de Módulo:* descreve os módulos do software e sua organização em camadas. Trata-se de uma visão importante para diminuir o impacto de uma mudança no software ou hardware. Pode ser representado pela UML por meio do diagrama de classe com nível de abstração maior representando módulos ou diagrama de implantação;

c) *Visão de Execução:* descreve a visão do sistema em tempo de execução. Pode ser representado pela UML por meio do diagrama de objetos;

d) *Visão de Código:* descreve como os módulos são mapeados para o código fonte e como as tarefas ou processos são mapeados para os executáveis. Pode ser representado pela UML por meio do diagrama de classe, pacotes e/ou diagrama de implantação.

Sobre os modelos apresentados anteriormente (Kruchten e Hofmeister) podemos destacar dois aspectos comuns. O primeiro aspecto é que as visões arquiteturais que foram propostas podem ser representadas por meio dos diagramas da UML. O segundo aspecto, e o mais importante, é a preocupação em representar dois tipos de arquiteturas: Arquitetura Estática e Arquitetura Dinâmica que são descritas a seguir.

2.2.1 Arquitetura Estática

A representação de uma arquitetura estática tem como objetivo demonstrar componentes e/ou relacionamentos estáticos. Esta arquitetura pode ser recuperada por meio da análise estática ou com a utilização dos diagramas estruturais da UML como, por exemplo, diagrama de classe ou de pacotes.

Os diagramas estruturais têm como principal vantagem o fato de serem recuperados diretamente do código fonte do sistema (isto é, sem a necessidade de execução). Podemos destacar também que atualmente diversas ferramentas suportam a extração de uma arquitetura estática por meio da análise do código fonte de um sistema.

No entanto, uma arquitetura estática apresenta uma versão parcial dos relacionamentos que são estabelecidos durante a execução de um sistema (ABI-ANTOUN; ALDRICH, 2009a; BRIAND; LABICHE; LEDUC, 2006; JACKSON; WAINGOLD, 2001). Por exemplo, diagramas estáticos não revelam relacionamentos estabelecidos por meio de polimorfismo e de chamada dinâmica de métodos, nem relacionamentos por meio do uso de reflexão computacional. Outro problema é a ordem da comunicação entre as classes (ou objetos) e informações sobre *threads*. Devido a essas limitações não é possível afirmar, por exemplo, como as *threads* influenciam o sistema e quando ocorre a comunicação entre as classes no sistema.

2.2.2 Arquitetura Dinâmica

A representação de uma arquitetura dinâmica tem como objetivo demonstrar componentes e/ou relacionamentos recuperados por meio da análise dinâmica. A UML permite representar aspectos dinâmicos de uma arquitetura por meio de seus diagramas comportamentais, por exemplo, diagrama de comunicação, sequência e de objetos.

A recuperação de uma arquitetura dinâmica tem como vantagens o acompanhamento do fluxo de execução, a captura de relacionamentos decorrentes de chamadas dinâmicas, a identificação de objetos definidos por meio de reflexão e objetos que possuem vinculação tardia (*late-binding*) em um sistema (SCHMERL *et al.*, 2006; YAN *et al.*, 2004). Também é possível verificar a existência de *threads* e como influenciam o comportamento do sistema. Em resumo, com essas informações o desenvolvedor consegue acompanhar passo a passo o comportamento do sistema. O problema dessa arquitetura consiste na escalabilidade, isto é, pode apresentar milhares de objetos mesmo para sistemas de pequeno porte (ABI-ANTOUN; ALDRICH, 2009a, 2009b). Outro problema, é que modelos dinâmicos normalmente não fazem distinção entre objeto de mais baixo nível (exemplo: `java.util.Date`) e objeto de maior valor arquitetural (exemplo: uma coleção de objetos do tipo `Customer`).

As soluções propostas para melhorar a escalabilidade de diagramas dinâmicos se baseiam em um mesmo princípio: agrupar objetos em unidades de maior granularidade (normalmente denominadas domínios (ABI-ANTOUN; ALDRICH, 2009a), componentes (MEDVIDOVIC; TAYLOR, 2000), clusters (ANQUETIL; FOURRIER; LETHBRIDGE, 1999) etc), de forma a permitir uma visão hierarquizada do diagrama. Basicamente, existem duas propostas para agrupar objetos em unidades de maior granularidade:

a) *Soluções Automáticas:* tem como objetivo agrupar classes ou objetos automaticamente, por exemplo, usando algoritmos de clusterização (ANQUETIL; LETHBRIDGE, 2009; ANQUETIL; FOURRIER; LETHBRIDGE, 1999) ou com a utilização de *data mining* (SAFYALLAH; SARTIPI, 2006). As soluções automáticas têm a desvantagem de não gerar grupos de objetos semelhantes àqueles que seriam definidos pelos desenvolvedores de sistemas;

b) *Soluções Manuais:* tem como objetivo agrupar classes ou objetos manualmente usando, por exemplo, anotações no código (ABI-ANTOUN; ALDRICH, 2009a, 2009b) ou definição de fatos em Prolog (RIVA; RODRIGUEZ, 2002). As soluções manuais têm dois problemas em comuns. O primeiro problema é a necessidade de possuir conhecimento sobre o sistema ou caso não tenha conhecimento, a necessidade de especialistas para auxiliá-lo. O segundo problema consiste da necessidade de conhecer a linguagem de especificação para definir regras de agrupamento dos elementos.

A seguir são apresentados os trabalhos relacionados a arquitetura de software e compreensão de sistemas.

2.3 Trabalhos Relacionados

Esta seção foi organizada em três subseções: Ferramentas e Abordagens baseadas em analise estática (Subseção 2.3.1), ferramentas e abordagens baseadas em análise dinâmica (Subseção 2.3.2) e linguagem para descrição e análise arquitetural (Subseção 2.3.3).

2.3.1 Análise Estática

Análise estática consiste em uma técnica que, a partir da análise do código fonte permite recuperar todas as comunicações entre as classes e apresentar uma visão do sistema como todo.

A abordagem SCHOLIA (*Static Conformance Checking Of Object-Based Structural Views of Architecture*) recupera a arquitetura dinâmica (*run-time architecture*) (ABI-ANTOUN; ALDRICH, 2009a, 2009b). No entanto, existem duas diferenças principais entre SCHOLIA e a proposta desta dissertação. Primeiro, SCHOLIA baseia-se exclusivamente nas técnicas de análise estática para recuperar as relações dinâmicas orientadas a objeto.

Portanto, as relações recuperadas por SCHOLIA representam uma aproximação nas relações concretas estabelecidas em uma execução do sistema alvo. Por exemplo, SCHOLIA não pode capturar informações sobre a cardinalidade de uma relação (por exemplo, a abordagem pode indicar que uma coleção é composta por elementos do tipo `A`, mas não se pode inferir quantos objetos existem de fato). Como uma segunda diferença, SCHOLIA depende de anotações explícita no código para definir a hierarquia que deve ser seguida para mostrar a arquitetura dinâmica. Por exemplo, a Figura 5 apresenta três classes (*Main*, *UI* e *Model*), a definição da arquitetura por meio de anotações (por meio do uso *@Domains*) e o gráfico gerado pela abordagem.

A exigência de anotar código dificulta a aplicação de SCHOLIA em cenários reais de desenvolvimento de software, devido ao esforço necessário para aplicar anotações em um sistema grande e complexo. Em um estudo de caso foram necessárias 30 horas para adicionar anotações em um sistema com 30 mil linhas de código com o auxílio de especialistas que conheciam o sistema (ABI-ANTOUN; ALDRICH, 2008). Além disso, desenvolvedores são reticentes à inserção de anotações no código, o que além de ser uma tarefa custosa, implica em um acoplamento do sistema a uma ferramenta específica. Por outro lado, SCHOLIA fornece suporte a conformação arquitetural, ou seja, é possível verificar e comparar os diagramas recuperados com um modelo da arquitetura pretendida.

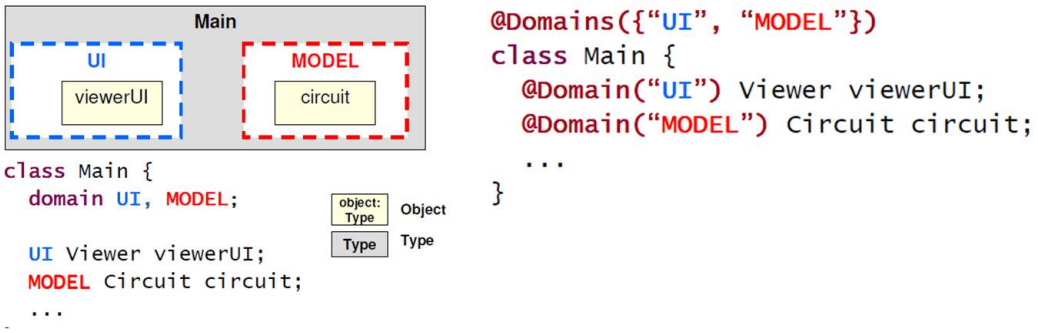

Figura 5: Ferramenta SCHOLIA
Fonte: ABI-ANTOUN; ALDRICH, 2009a

Os modelos de reflexão (RM) (*Reflection Mode*) é outra abordagem para a conformidade arquitetural estática (MURPHY; NOTKIN; SULLIVAN, 1995, 2001). A técnica RM exige que os desenvolvedores forneçam uma arquitetura planejada de alto nível e um mapeamento entre o modelo declarativo e o modelo do código-fonte. Uma ferramenta baseada em RM – como o SAVE (*Software Architecture Visualization and Evaluation*) (KNODEL; POPESCU, 2007) – destaca relações convergentes, divergentes e ausentes entre o modelo de

alto nível e o modelo do código fonte. Normalmente, os desenvolvedores fornecem modelos de alto nível não hierárquicos e que representam somente relacionamentos estáticos entre os elementos de um programa (classes, pacotes, etc).

Uma abordagem simples que se propõe a recuperar diagramas de objetos por meio da análise estática chama-se Womble (JACKSON; WAINGOLD, 2001). Portanto, Womble compartilha as mesmas vantagens e desvantagens de SCHOLIA quanto a precisão das relações recuperadas. No entanto, ao contrário de SCHOLIA, Womble não fornece meios de sumarizar objetos. Esta limitação resulta em um gráfico extraído com milhares de objetos, mesmo para sistemas pequenos.

Para o uso comercial existem diversas ferramentas que utilizam engenharia reversa para recuperar diagramas da UML como, por exemplo, *Enterprise Architect*, *Argo UML*, *Borland Together* e *Rational Rose*.

2.3.2 Análise Dinâmica

Análise Dinâmica consiste em uma técnica que se vale do rastro de execução para recuperar o comportamento de um sistema. Com a utilização da análise dinâmica é possível verificar a ordem em que objetos foram instanciados, a ordem em que ocorreu a comunicação entre objetos e outras informações como polimorfismo e a existência de *threads*. A análise dinâmica possui uma limitação que consiste no fato de o rastro de execução para uma mesma funcionalidade pode ser diferente dependendo da entrada de dados. Porém, como um dos objetivos do desenvolvedor é compreender a funcionalidade, esta diferença pode não ser uma limitação tão forte, pois é a representação real do comportamento do sistema.

A ferramenta Discotect tem como objetivo recuperar a arquitetura dinâmica de um sistema (YAN *et al.*, 2004). No entanto, em vez de diagramas de objetos, Discotect extrai uma visão arquitetural baseada em conectores e componentes. Para isso, Discotect instrumenta o sistema alvo, de forma a permitir a captura de eventos que ocorrem durante a sua execução (por exemplo, chamadas de métodos). O sistema requer que desenvolvedores definam um mapeamento entre os eventos monitorados e componentes/conectores. Portanto, esse mapeamento requer um conhecimento mais detalhado do sistema alvo.

A abordagem proposta por Briand, Labiche e Leduc consiste em uma extração de diagramas de sequência a partir da execução de um sistema alvo (BRIAND; LABICHE; LEDUC, 2006). De forma similar à ferramenta a ser descrita nesta dissertação, a ins-

trumentação do sistema base é feita por meio de orientação por aspectos. No entanto, a instrumentação proposta gera eventos que são armazenados em um repositório. A ferramenta de visualização, recupera as informações do repositório e gera diagramas de sequência de forma *off-line*. Por fim, a abordagem proposta não permite a definição de unidades de maior granularidade do que objetos.

A abordagem proposta por Safyallah e Sartipi é uma análise a partir de frequências de padrões verificados por meio dos rastros de execução (SAFYALLAH; SARTIPI, 2006). A abordagem se resume em executar funcionalidades com características semelhantes e posteriormente aplica-se um algoritmo de mineração de dados que busca padrões de frequência. Com o resultado da mineração de dados é possível separar as funções com menor relevância das funções comuns, isto é, funções que ocorrem com alguma frequência nas diferentes funcionalidades executadas. A limitação desta abordagem consiste em não fornecer meios do desenvolvedor controlar a granularidade da informação que deseja, pois informações somente das funções que são comuns entre funcionalidades semelhantes possivelmente não são suficiente para compreender adequadamente um sistema mais complexo.

A abordagem proposta por Riva e Rodriguez utiliza uma técnica que combina modelos estáticos e dinâmicos (RIVA; RODRIGUEZ, 2002). A recuperação das informações estáticas ocorre a partir da análise do código fonte e mantidas como fatos em *Prolog*. Para recuperar informações dinâmicas, os autores instrumentam o código com o auxílio da ferramenta *ThirdEye* (LENCEVICIUS; RAN; YAIRI, 2000). A sumarização (agrupamento) ocorre por meio da definição de regras sobre os fatos em *Prolog*. A Figura 6 no primeiro momento *1*, apresenta um diagrama de sequência contendo três classes (A,B e C). No segundo momento *2*, com a sumarização das classes e métodos (abstração vertical e horizontal) o número de classes foram reduzidas para duas (AB e C) e houve uma redução no número de comunicação entre as classes. A limitação dessa abordagem é a complexidade de definir regras sobre os fatos em prolog para sistemas maiores.

O software *JProfiler* (TECHNOLOGIES EJ, 2011) é uma ferramenta que possui diversos recursos, mas podemos destacar o recurso chamado *Heap Walker*. Este recurso permite o desenvolvedor visualizar os objetos do sistema (de forma semelhante a um diagrama de objetos). A Figura 7 apresenta a funcionalidade *Heap Walker* e como os objetos estão se relacionando. Nessa Figura, por exemplo, três objetos (*WLLogger*, *Logging* e *LogManager*) comunicam com o objeto *LogManager*. A limitação desta funcionalidade é representar este gráfico de forma legível para sistemas de médio/grande porte, onde

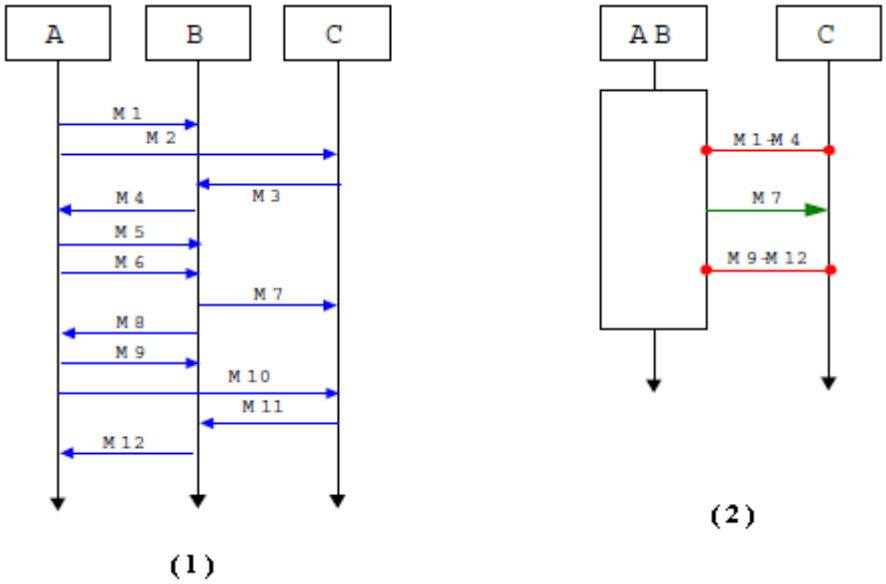

Figura 6: Diagrama de sequência
Fonte: RIVA; RODRIGUEZ, 2002

podem existir milhares de objetos.

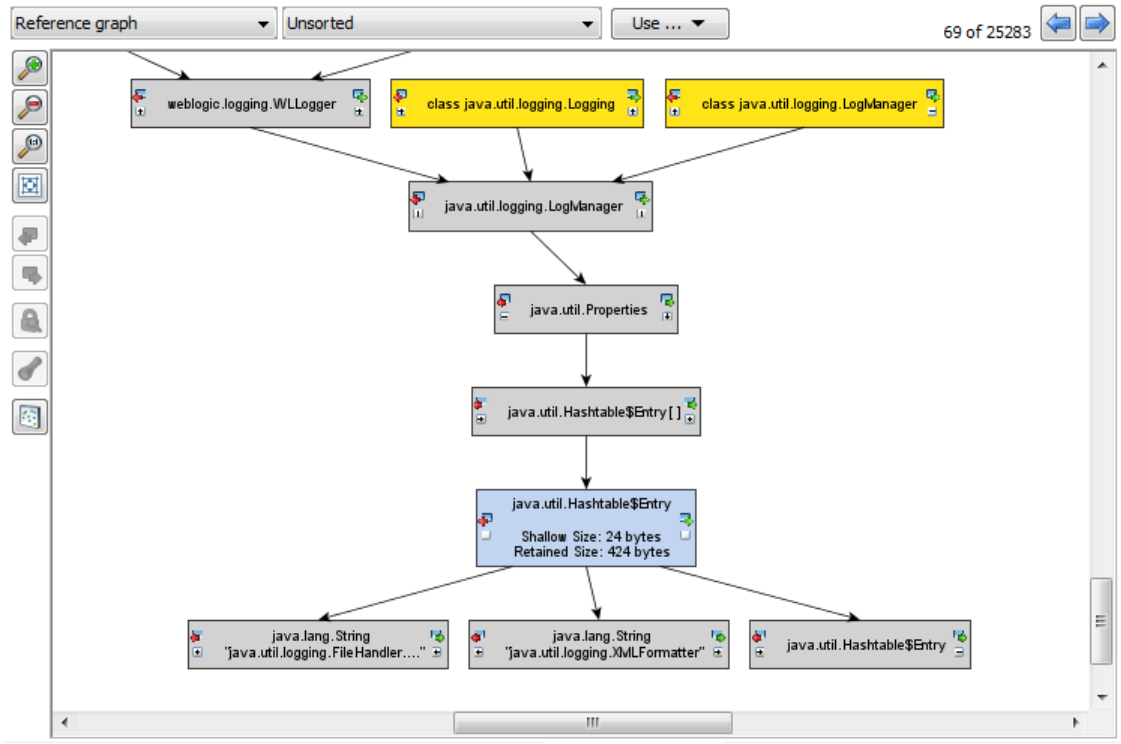

Figura 7: Ferramenta *JProfiler*
Fonte: Elaborado pelo autor

2.3.3 Linguagens

As linguagens de descrição arquitetural (ADLs) têm como objetivo especificar uma arquitetura de software e/ou hardware por meio de uma linguagem formal e/ou através de gráficos (ALDRICH; CHAMBERS; NOTKIN, 2002). Em geral, os benefícios das ADLs são a possibilidade de testar uma arquitetura como, por exemplo, em coerência, conformação e desempenho antes de implementá-la. A dificuldade comum nas linguagens de descrição arquitetural é o tempo necessário para aprender uma determinada linguagem – tanto para escrita como para leitura. Existem diversas ADLs como *ACME*, *byADL*, *Wright*, *ArchJava*, etc.

A linguagem ArchJava tem como objetivo definir uma arquitetura, que se estende Java com arquitetura de abstrações, como componentes e conectores (ALDRICH; CHAMBERS; NOTKIN, 2002). A limitação é que o ArchJava exige que os desenvolvedores migrem seus sistemas para uma nova linguagem.

A linguagem para conformação arquitetural DCL (*Dependency Constraint Language*) (TERRA; VALENTE, 2008, 2009) permite definir dependências aceitáveis e inaceitáveis de acordo com a arquitetura planejada de um sistema. Uma vez definidas, tais restrições são verificadas por uma ferramenta de conformação integrada à plataforma Eclipse. Portanto, DCL é uma linguagem voltada para conformação arquitetural, usando para isso análise estática. O modo de definir as regras arquiteturais do *DCL* foram adaptados para a abordagem apresentada nesta dissertação.

3 OBJECT GRAPH

Um *OG* (*Object Graph*) consiste é um grafo direcionado que representa o comportamento dinâmico dos objetos de um sistema. O *OG* permite ao desenvolvedor visualizar diversas perspectivas do sistema (granularidade grossa e/ou fina) com objetivo de auxiliá-lo a compreender a arquitetura ou uma determinada funcionalidade.

Este capítulo está organizado conforme descrito a seguir. Uma seção com os conceitos básicos e os exemplos (Seção 3.1) para demonstrar as utilidades de um *OG*. Uma seção com os conceitos avançados que discute outros aspectos de um *OG* (Seção 3.2). Uma seção com a linguagem de alertas com os seus conceitos e exemplos (Seção 3.3). Por fim, uma seção com a ferramenta para visualização e manipulação de grafo de objetos (Seção 3.4).

3.1 Conceitos Básicos

Nesta seção são apresentados os conceitos básicos de um *OG* como: vértices e arestas. Os vértices de um *OG* representam todos os objetos e algumas das classes de um sistema. As arestas representam os possíveis relacionamentos entre os vértices do grafo. Detalhes sobre os vértices e arestas de um *OG* são fornecidos a seguir.

3.1.1 Vértices

Um *OG* admite dois tipos de vértices. Vértices na forma de círculo são usados para representar objetos. Vértices na forma de um quadrado denotam classes. Vértices na forma de um quadrado são criados para representar classes com membros estáticos que referenciam ou fazem uso de serviços providos por objetos. Ou seja, nem todas as classes de um programa são representadas em um *OG*, mas apenas aquelas com atributos estáticos que referenciam objetos ou que possuem métodos estáticos que acessam serviços providos por objetos. Vértices na forma de um círculo têm o mesmo tempo de vida de

objetos, isto é, são destruídos quando o objeto que representam é alvo do coletor de lixo de Java.

Um vértice de um *OG* – seja ele objeto ou classe – consiste em uma estrutura composta por:

```
[Ordem] Nome (cor_vértice)
```

Os membros da estrutura de um vértice são apresentados a seguir:

a) *Ordem:* representa um inteiro não-negativo e sequencial que indica a ordem de criação dos vértices no grafo. Por convenção, o primeiro vértice criado – normalmente, representando a classe que contém o método *main* do sistema – recebe o número zero. O objetivo desse inteiro é viabilizar uma leitura sequencial do grafo, começando pelo ponto de entrada da aplicação, passando então para os objetos criados ou referenciados a partir dessa classe e assim por diante;

b) *Nome:* representa o nome da classe do objeto representado (se vértice circular) ou o nome da classe com membros estáticos que referenciam ou acessam serviços de objetos (se vértice quadrangular);

c) *Cor_ Vértice:* no *OG*, usam-se cores para diferenciar vértices criados ou referenciados pelas *threads* de um sistema. Todos os vértices criados pela *thread* principal recebem a cor preta. Caso o programa crie outra *thread*, uma nova cor é automaticamente utilizada para representar os objetos criados pela mesma. Em resumo, os vértices de um *OG* possuem múltiplas cores, as quais denotam as *threads* em que foram referenciados.

Para ilustrar os elementos básicos de um *OG* mostram-se a seguir exemplos de grafos extraídos de trechos de código hipotéticos.

Exemplo (Vértices): Suponha o código mostrado na Listagem 3.1. Neste código, a classe *Main* instancia um objeto do tipo *Invoice* e chama o método *load* do mesmo (linhas 4-5). A execução do método *load* cria um *ArrayList* e insere um *Product* no mesmo (linhas 12-14).

```
1  public class Main {
2     private static Invoice invoice;
3     public static void main(String[] args) {
4        invoice = new Invoice();
5        invoice.load();
6     }
7  }
8
9  public class Invoice {
10    private Collection<Product> col;
11    public void load(){
12       col = new ArrayList<Product>();
13       Product p = new Product();
14       col.add(p);
15    }
16 }
```

Listagem 3.1: Exemplo 1
Fonte: Elaborado pelo autor

A Figura 8 apresenta o *OG* gerado pela execução do código mostrado na Listagem 3.1. Esse *OG* possui um vértice quadrangular (representando a classe que contém o *main* da aplicação) e três vértices circulares, representando os objetos criados durante a execução do programa.

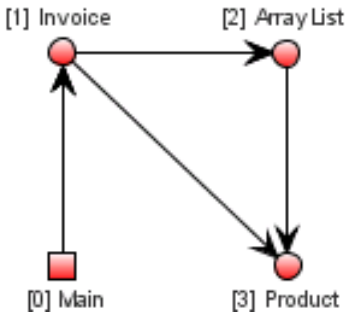

Figura 8: *OG* do Exemplo 1
Fonte: Elaborado pelo autor

O *OG* apresentado modela de modo compacto a execução do programa. Acompanhando o inteiro sequencial associado a cada vértice, pode-se verificar que a classe *Main* (vértice 0) acessou um objeto do tipo *Invoice* (vértice 1), que por sua vez acessou um

objeto do tipo *ArrayList* (vértice 2). Por fim, pode-se verificar que o último objeto criado foi do tipo *Product* (vértice 3). Em algum ponto da execução do programa, esse objeto foi acessado pelos objetos *Invoice* (responsável pela sua criação) e *ArrayList* (responsável pelo seu armazenamento).

Exemplo (Threads): Suponha o código mostrado na Listagem 3.2. Nesse código, a classe *Main* pode instanciar vários objetos do tipo *Box* dependendo da variável *length* (linhas 3-4). Após a instanciação de um objeto do tipo *Box* (linha 5), ativa-se uma *thread*. A *thread* criada simplesmente instancia um objeto do tipo *Product* (linha 11).

```
1  public class Main{
2      public static void main(String[] args) {
3          int length = Integer.parseInt(args[0]);
4          for (int i = 0; i < length; i++) {
5              new Box().start();
6          }
7      }
8  }
9  public class Box extends Thread {
10     public void run() {
11         new Product();
12     }
13 }
```

Listagem 3.2: Exemplo 2
Fonte: Elaborado pelo autor

A Figura 9 apresenta o *OG* gerado pela execução do código da Listagem 3.2. É importante ressaltar que nesse exemplo o valor passado por parâmetro e atribuído para a variável *length* (linha 3) foi igual a dois – percebe-se isso pelo número de instâncias da classe *Product*.

Sobre o *OG* da Figura 9, pode-se verificar que a classe *Main* (vértice 0) acessou dois objetos do tipo *Box* (vértices 1-2). Pode-se verificar também que cada objeto *Box* acessou um objeto do tipo *Product* (vértices 3-4). Mais importante, no *OG* mostrado, os vértices que representam *Product* possuem cores diferentes, visto que eles foram criados por *threads* diferentes. Por existir concorrência, a ordem de criação dos objetos impactados pelas *threads* pode não ser a mesma caso a funcionalidade seja executada novamente. Porém,

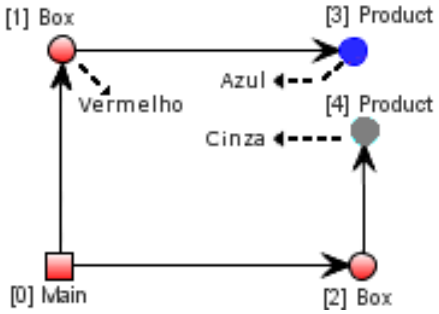

Figura 9: *OG* do Exemplo 2
Fonte: Elaborado pelo autor

o desenvolvedor consegue observar que existem *threads* impactando os objetos porque o *OG* deixa evidenciado esses objetos por meio das cores dos vértices.

3.1.2 Arestas

Em um *OG* arestas representam relacionamentos entre os objetos e classes do grafo. Suponha que o_1 e o_2 são vértices circulares (representando objetos) e que c_1 e c_2 são vértices quadrangulares (representando classes). Uma aresta direcionada (o_1,o_2) indica que o_1 – em algum momento do seu tempo de vida – adquiriu uma referência para o_2. Essa referência pode ter sido adquirida por meio de um campo, de uma variável local ou de um parâmetro formal de método. Já uma aresta direcionada (o_1,c_1) indica que o_1 – em algum momento do seu tempo de vida – invocou um método estático implementado por c_1.

Por outro lado, uma aresta direcionada (c_1,o_1) indica que c_1 – em algum momento da execução do programa – adquiriu uma referência para o_1. Essa referência foi adquirida por meio de um campo estático da classe c_1. Por fim, uma aresta direcionada (c_1,c_2) indica que c_1 – em algum momento da execução do programa – invocou um método estático implementado por c_2.

As arestas descritas anteriormente são inseridas no grafo tão logo o relacionamento representado seja estabelecido durante a execução do programa. Quando um vértice é removido do grafo, todas as arestas que chegam e saem do mesmo são também removidas. Para aumentar a legibilidade do grafo, não são mostradas arestas representando auto-relacionamentos (isto é, arestas que saem e chegam ao mesmo vértice).

Detalhes sobre arestas: É possível exibir informações detalhadas sobre as relações orientadas a objeto modeladas pelas arestas de um OG's. Suponha que o_1 e o_2 são vértices de um *OG* e que existe uma aresta que conecta esses dois vértices. O nome da aresta consiste em uma estrutura no seguinte formato:

```
Ordem - [O1_Ordem] O1_Localização [Sufixo] > [O2_Ordem] O2_Serviço [Sufixo]
```

Os membros da estrutura de uma aresta são apresentados a seguir:

a) *Ordem:* número sequencial não negativo que indica a ordem que cada aresta foi criada no grafo. O objetivo da numeração é possibilitar uma leitura sequencial das arestas do grafo;

b) *O1_ Ordem:* número sequencial não negativo que indica a ordem que o vértice o_1 foi criado no grafo;

c) *O1_ Localização:* representa o método de o_1 onde a relação com o_2 foi estabelecida;

d) *O2_ Ordem:* número sequencial não negativo que indica a ordem que o vértice o_2 foi criado no grafo;

e) *O2_ Serviço:* representa o serviço provido por o_2 que foi acessado por o_1.

f) *Sufixo:* fornece informação sobre *Localização* e *Serviço*. Os possíveis valores são:

- *():* acesso a método;
- *(MS):* acesso a método estático;
- *(C):* acesso a construtor da classe;
- *(A):* acesso a atributo;
- *(AS):* acesso a atributo estático;
- *<new>* instanciação de objeto.

Exemplos: Para exemplificar a utilidade das informações providas sobre arestas, são apresentados duas listagens de arestas extraídos dos Exemplos 8 e 9.
A Listagem 3.3 apresenta uma amostra da lista de arestas do *OG* representado pela Figura 3.1. Essa Listagem mostra que no momento *01* a classe *Main* por meio do seu método

estático *main* atribuiu o objeto *Invoice* ao seu atributo estático *invoice*. No momento *03*, o objeto *Invoice* por meio do seu método *load* instanciou um *ArrayList*. No momento *04*, o objeto *Invoice* por meio do seu método *load* atribuiu o objeto *ArrayList* ao seu atributo *listProducts*. No momento *06*, o objeto *Invoice* por meio do seu método *load* invocou o método *add* do objeto *ArrayList*.

```
1 ...
2 01-[0]Main.main(MS) > [1]Invoice.invoice(AS)
3 ...
4 03-[1]Invoice.load() > [2]ArrayList.<new>
5 04-[1]Invoice.load() > [1]Invoice.listProducts(A)
6 ...
7 06-[1]Invoice.load() > [2]ArrayList.add()
8 ...
```

Listagem 3.3: Lista de arestas da Figura 8
Fonte: Elaborado pelo autor

A Listagem 3.4 apresenta uma amostra da lista de arestas do *OG* apresentado pela Figura 3.2. A Listagem 3.4 indica que no momento *02* a classe *Main* por meio do seu método estático *main* instanciou um segundo objeto *Box*. No momento *03*, a classe *Main* por meio do seu método estático *main* invocou o método *start* do objeto *Box*. No momento *04*, o primeiro objeto *Box* por meio do seu método *run* instanciou um objeto da classe *Product*.

```
1 ...
2 02-[0] Main.main(MS) > [2] Box.<new>
3 03-[0] Main.main(MS) > [2] Box.start()
4 04-[1] Box.run() > [3] Product.<new>
5 ...
```

Listagem 3.4: Lista de Aresta da Figura 9
Fonte: Elaborado pelo autor

Portanto, é possível fazer uma leitura sequencial e precisa de funcionalidades do sistema com as informações fornecidas pelas arestas, como mostrado nas Listagens 3.3 e 3.4.

3.1.3 Pacotes e Domínios

Como em qualquer representação visual da execução de um programa, o tamanho e a quantidade de informações representadas em um *OG* tendem a crescer rapidamente, podendo comprometer a legibilidade do grafo mesmo em aplicações pequenas. Para garantir que *OG* são escaláveis até sistemas de grande porte, dispõe-se de um recurso para sumarização de vértices. Basicamente, ao visualizar um *OG* pode-se optar por sumarizar alguns vértices em um vértice único. Conforme descrito a seguir, existem dois critérios para sumarização de vértices.

No primeiro critério, a sumarização consiste em agrupar os vértices de um *OG* de acordo com conjuntos definidos pelos desenvolvedores, denominados domínios – o qual são representados por meio de um hexágono. O objetivo é fornecer um mecanismo mais flexível para sumarização de vértices do que aquele baseado exclusivamente na divisão estática de um sistema em pacotes. Basicamente, um domínio consiste em uma coleção de objetos de classes especificadas pelo desenvolvedor, por meio da seguinte sintaxe:

```
domain name: [!] classes [ + | .* | .** ]
```

onde *name* é o nome do domínio e *classes* é uma lista de classes separadas por vírgulas. Colchetes são usados para delimitar termos opcionais (tal como em *[!]*, indicando que o operador *!* é opcional). Para fins de sumarização, todos os objetos das classes listadas são considerados como pertencentes ao domínio *name*. Para facilitar a definição das classes de um domínio podem ser utilizadas expressões regulares da seguinte forma: operador *+* (exemplo: *A+* denota a classe *A* e suas subclasses), operador *.** (exemplo: *package.** denota todas as classes pertencentes ao pacote *package*) ou o operador *.*** (exemplo: *package.*** denota todas as classes de pacotes com o prefixo especificado). O operador *!* significa negação (exemplo: *!A* significa que o domínio será composto por todas as classes com o nome diferente de *A*).
É importante ressaltar que caso existam classes que pertençam a mais um domínio, elas serão alocadas no primeiro domínio definido. Ou seja, a ordem dos domínios neste caso prevalece, sendo que o primeiro domínio tem prioridade sobre o segundo domínio e assim por diante.

Exemplos de Domínios: Suponha um sistema hipotético, que utilize o padrão arquitetural MVC (*Model-View-Controller*). Para prover uma visão arquitetural (isto é, de alto nível) e dinâmica (isto é, que expresse os relacionamentos entre os objetos desse sistema),

suponha que foram definidos os seguintes domínios:

```
1 domain View:      myapp.view.IView+
2 domain Controller: myapp.controller.*
3 domain Model:     "myapp.model.[a-zA-Z0-9/.]*DAO"
4 domain Swing:     javax.swing.**
5 domain Hibernate: org.hibernate.**
```

Listagem 3.5: Exemplo de Definição dos Domínios
Fonte: Elaborado pelo autor

Nessa definição, o domínio *View* denota quaisquer objetos que implementam a interface *myapp.view.IView* e suas subclasses. O domínio *Controller* denota objetos de quaisquer classes do pacote *myapp.controller*. Já o domínio *Model* denota objetos de classes cujo nome inicia-se com *myapp.model* e termina com DAO (*Data Access Object*). O operador ** seleciona todas as classes de pacotes com o prefixo especificado. Portanto, os domínios *Swing* e *Hibernate* denotam, respectivamente, objetos de classes dos pacotes *javax.swing* e *org.hibernate*, assim como objetos de quaisquer classes de pacotes internos aos pacotes mencionados.

A Figura 10 apresenta um possível *OG* gerado para o sistema MVC considerado nesse exemplo. Primeiro, veja que cinco vértices possuem uma forma de hexágono, representando cada um dos domínios definidos anteriormente. Existe ainda um único vértice em forma de círculo, representando um objeto do tipo *Util*, o qual não pertence a nenhum dos domínios especificados. Ou seja, objetos pertencentes a um dos domínios definidos são automaticamente sumarizados em um vértice em forma de hexágono; objetos que não são capturados por nenhum dos domínios definidos continuam sendo representados por meio de vértices circulares (no caso de objetos) ou quadrangulares (no caso de classes).

No *OG* da Figura 10, é possível observar que a divisão de domínios segue o padrão MVC. Por exemplo, existe uma comunicação bidirecional entres os domínios *View* e *Controller* e entre os domínios *Controller* e *Model*. Em outras palavras, o *OG* mostrado revela que o domínio *Controller* desempenha o papel de um mediador entre os domínios *View* e *Model*, conforme previsto em arquiteturas MVC. Pode-se observar ainda que somente o domínio *View* está acoplado ao *framework Swing* e que somente o domínio *Model* está acoplado ao *framework Hibernate*.

Exemplos de Pacotes e Domínios: O segundo critério de sumarização consiste em representar apenas o pacote ao qual pertencem as classes dos objetos representados no grafo.

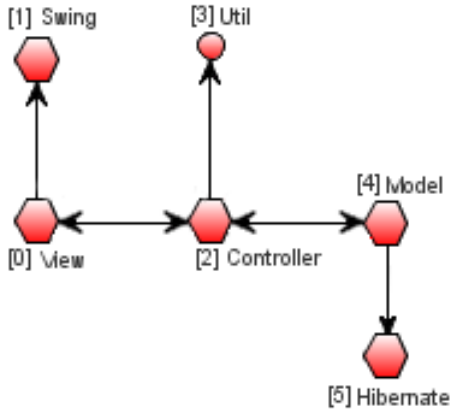

Figura 10: *OG* do Exemplo 3
Fonte: Elaborado pelo autor

Ou seja, todos os objetos que pertencem a classes de um mesmo pacote são agrupados em um vértice único. Suponha dois vértices representando os pacotes p_1 e p_2. Existirá uma aresta direcionada (p_1,p_2) se existir pelo menos um elemento de p_1 conectado a um elemento de p_2 na versão não-sumarizada do *OG*.

Existe ainda a possibilidade de unir a visão de pacotes com a visão de domínios. Suponha um sistema hipotético, em que o desenvolvedor conheça somente dois componentes: *Util* e *View*. As classes que não pertencem aos domínios definidos pelo desenvolvedor serão sumarizadas nos seus respectivos pacotes. Suponha que o desenvolvedor deseja analisar como estes dois componentes (*Util* e *View*) se relacionam. Além disso, deseja verificar se os dois componentes estão de acordo com as suas responsabilidades. Isto é, o domínio *Util* como provedor de serviços e o domínio *View* deve ser referenciado (se referenciado) somente por pacotes pertencentes à interface gráfica como *java.awt* ou *javax.swing*.
A definição dos domínios segue a divisão dos pacotes conforme demonstrado na Listagem 3.6.

```
1 domain Util: com.Util.**
2 domain View: com.View.**
```

Listagem 3.6: Definição de Domínios (Exemplo 4)
Fonte: Elaborado pelo autor

A Figura 11 mostra o possível *OG* do sistema hipotético. Como pode ser observado, foram criados cinco vértices, sendo três referentes a pacotes (*com.action*, *java.awt* e *com.io*) e dois referentes a domínios (*Util* e *View*). Como o desenvolvedor deseja verificar as responsabilidades dos componentes, percebe-se que o domínio *Util* deveria apresentar

somente arestas de entradas, mas o mesmo relaciona com o pacote *com.io*. Este relacionamento não deveria existir porque diminui a capacidade de reutilização do componente *Util* (já que depende de outra classe). Outro componente *View*, relaciona com o componente *Util* e com os pacotes *com.action* e *java.awt* não apresentando nenhum problema. Esta visão do *OG* é uma nova perspectiva do sistema, que auxilia na compreensão dos relacionamentos entre os componentes e outros pacotes. Isto possibilita ao desenvolvedor ter uma visão macro do sistema permitindo detectar problemas e compreender a arquitetura do sistema.

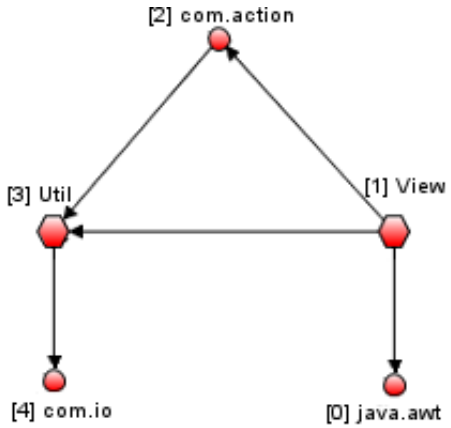

Figura 11: *OG* do Exemplo 4
Fonte: Elaborado pelo autor

3.2 Conceitos Avançados

Nesta seção são apresentados detalhes sobre a remoção e criação dos vértices de um *OG*.

Remoção dos Vértices: Ocorre quando o coletor de lixo começa a coletar objetos que não estão sendo mais referenciados em um sistema. No contexto do *OG*, quando ocorre a destruição de um objeto o vértice que o representa será removido do grafo juntamente com as suas arestas. A seguir são apresentados dois exemplos onde este recurso seria útil ao desenvolvedor.

No primeiro exemplo, o desenvolvedor deseja analisar o comportamento real de um determinado sistema, isto é, o desenvolvedor deseja analisar o momento em que são criados os objetos e o momento em que os objetos são destruídos.

No segundo exemplo, o desenvolvedor deseja identificar quais são os objetos que sempre estão sendo criados e que ficam mais tempo na memória antes de serem destruídos pelo coletor de lixo. Uma vez identificados estes objetos, pretende-se realizar uma avaliação sobre viabilidade de aplicar o padrão de projeto *Singleton*, isto é, ao invés de criar vários objetos poderá ser criado um único objeto (por meio do *Singleton*).

O recurso de remoção dos vértices tem utilidade, mas caso exista uma necessidade de uma leitura sequencial, então este recurso contribuirá pouco porque os vértices são removidos do *OG* algumas vezes de forma muito rápida impossibilitando a leitura sequencial. Por esta razão, existe a possibilidade de ativar ou não a remoção dos vértices.

Criação dos vértices: Em um *OG* a criação dos vértices pode ocorrer de duas maneiras. A Listagem 3.7 descreve um código hipotético demonstrando a criação de vértices no *OG*. A primeira maneira consiste na utilização do operador *new* criando uma nova instância da classe *Figure* (linha 2). A segunda maneira ocorre quando um objeto recebe a referência de um objeto já instanciado. Por exemplo, quando captura-se o rastro de execução de uma funcionalidade do sistema, alguns objetos podem ter sido instanciados anteriormente à captura. Quando o desenvolvedor aciona a funcionalidade, esses objetos (que estão instanciados) podem vir a ser utilizados conforme apresentado nas linhas 3 e 4. Neste caso, se os vértices referentes a esses objetos não foram criados então serão criados dois vértices (*View* e *Figure*) e uma aresta ligando os mesmos.

```
1 public void addFigure() {
2     Figure fig = new Figure();
3     View view = this.getView();
4     view.setFigure(fig);
5 }
```

Listagem 3.7: Exemplo de código
Fonte: Elaborado pelo autor

3.3 Linguagem de Alertas

Grafos de objetos foram projetados para capturar de forma não-invasiva a arquitetura dinâmica de sistemas. Sendo assim, são ferramentas úteis a um desenvolvedor de sistema que esteja interessado em compreender o comportamento dinâmico dos sistemas

sob sua responsabilidade. A fim de auxiliar os desenvolvedores em tais tarefas, foi definida uma linguagem para exibição de alertas em um *OG*. A ideia básica é associar alertas a relacionamentos que são esperados (ou que não são esperados) em um sistema. Quando tais relacionamentos forem estabelecidos no *OG*, um alerta é exibido para o desenvolvedor.

Suponha um sistema onde o acesso a dados é realizado por meio de objetos que implementam o padrão DAO (FOWLER, 2002). Um desenvolvedor interessado em estudar e entender melhor os módulos de persistência desse sistema pode, por exemplo, definir um alerta que gere uma mensagem toda vez que um determinado domínio usar os serviços de objetos *DAO*. Como um segundo exemplo, dessa vez de relacionamento que não deveria existir, um desenvolvedor poderia definir um alerta para gerar uma mensagem toda vez que uma classe utilitária usar serviços de outros módulos do sistema. Esse relacionamento não deveria ocorrer no sistema em questão, visto que classes utilitárias não devem estabelecer dependências com sistemas clientes, de forma a preservar o potencial de reúso das mesmas.

Os alertas definidos pelos desenvolvedores de um *OG* são exibidos de duas formas: (a) mudando a cor das arestas relativas aos relacionamentos responsáveis pelo alerta; (b) gerando uma mensagem em uma janela de alertas, onde são informados detalhes sobre os alertas gerados durante a execução do sistema.

Sintaxe: Na solução proposta, alertas são definidos por meio da seguinte gramática:

```
1 <alert_clause> ::= alert <domain> {, <domain>}
2     <relation> <domain> {, <domain>}
3
4 <domain> ::= [!] <string> | *
5 <relation> ::= access | create | depend
```

Listagem 3.8: Definição de Alertas Arquiteturais
Fonte: Elaborado pelo autor

Nessa gramática, símbolos não-terminais são escritos entre < e > (tal como em *<domain>*). As chaves *{* e *}* (tal como em *{, <domain>}*) indicam que o elemento pode ter zero ou mais repetições. Símbolos terminais são escritos sem nenhum delimitador especial (tal como em *alert*, *access*, etc). Colchetes são usados para delimitar símbolo opcional (tal como em *[!]*, indicando que o terminal *!* é opcional). O não-terminal *<string>* denota uma *string* (isto é, uma sequência de caracteres, sem aspas delimitadoras).De acordo com essa gramática, uma cláusula de alerta define um relacionamento entre dois domínios. Isso

é, o alerta será ativado quando for detectado um relacionamento do tipo especificado entre os domínios definidos na cláusula. Na definição de um domínio, pode-se usar o operador de negação *!* para denotar todos os objetos não pertencentes ao domínio especificado. Por exemplo, *!A* é o domínio que contém qualquer objeto não pertencente ao domínio *A*. Por fim, o símbolo * denota qualquer objeto.

Para ilustrar a sintaxe de regras de alertas, suponha os objetos *a1* do domínio *A*, *b1* do domínio *B* e *c1* do domínio *C*. As regras de alertas seriam especificadas conforme descrito a seguir:

a) *alert A* x[1] *B*: essa restrição destaca dependências do tipo x dos objetos do domínio *A* com objetos do domínio *B*. Logo, a aresta entre *a1* e *b1* será destacada se, e somente se, *a1* estabelecer uma dependência do tipo x com *b1*;

b) *alert A x !B*: essa restrição destaca dependências do tipo x objetos do domínio *A* e objetos não pertencentes ao domínio *B*. Logo, a aresta entre *a1* e *c1* será destacada se, e somente se, *a1* estabelecer uma dependência do tipo x com *c1* (supondo que *c1* não pertence ao domínio *B*);

c) *alert !A x B*: essa restrição destaca dependências do tipo x entre objetos não pertencentes ao domínio *A* com objetos do domínio *B*. Logo, a aresta entre *c1* e *b1* será destacada se, e somente se, *c1* estabelecer uma dependência do tipo x com *b1* (supondo que *c1* não pertence ao domínio *A*).

Na definição de uma cláusula de alerta, podem ser especificados três tipos de relacionamentos entre os domínios que se deseja monitorar:

a) *depend*: Qualquer tipo de relacionamento, dentre aqueles que podem ser representados em um grafo de objetos. Em outras palavras, um alerta desse tipo será ativado quando durante a execução do programa, um objeto do tipo de origem referenciar um objeto do domínio de destino. Essa referência pode ter sido armazenada em uma variável local, em um parâmetro formal ou em um campo do objeto do domínio de origem;

b) *access*: Um alerta desse tipo será ativado quando durante a execução do programa, um objeto do tipo de origem não apenas referenciar um objeto do domínio de destino, como também acessar um campo ou chamar um método desse

[1] O literal x se refere ao tipo da dependência, que pode ser mais abrangente (*depend*) ou mais específico (*access* e *create*).

objeto. Ou seja, *access* é um caso específico de um relacionamento do tipo *depend*. Por exemplo, um objeto pode referenciar um objeto de outro domínio (*depend*), mas não usar nenhum dos serviços providos por ele (*access*). Um exemplo são objetos de fachada, que simplesmente repassam referências recebidas como parâmetro para métodos escondidos por trás da fachada;

c) *create*: Um alerta desse tipo indica que, durante a execução do programa, um objeto do domínio de origem criou um objeto do domínio de destino. Mais especificamente, essa criação ocorreu durante a execução de um método de um objeto do domínio de origem.

A linguagem para definição de alertas em um *OG* foi inspirada na linguagem para conformação arquitetural *DCL* (TERRA; VALENTE, 2008, 2009). DCL é uma linguagem voltada para conformação arquitetural. Já a linguagem de alertas é voltada para ajudar desenvolvedor a detectar relações importantes em um grafo de objetos. Por relações importantes, entendemos tanto relações esperadas no sistema, como relações que representam alertas arquiteturais. Em resumo, a linguagem de alertas proposta no trabalho é um instrumento extra, do qual podem dispor um desenvolvedor interessado em entender melhor detalhes de um sistema. A seguir apresenta-se um exemplo da utilização da linguagem de alerta.

3.3.1 Exemplo

A Listagem 3.9 demonstra três exemplos de alertas (usando os domínios definidos anteriormente na Listagem 10). Nessa listagem, são definidos alertas para quando um objeto acessar um objeto do domínio *Hibernate* (linha 1). Define-se também um alerta para quando um objeto da classe *myapp.Util* acessar um objeto que não seja dessa mesma classe (linha 2). Com esse segundo alerta, deseja-se verificar se classes utilitárias são auto-contidas, funcionando meramente como provedoras de serviços para módulos externos. Por fim, define-se um alerta destinado a monitorar se objetos do tipo *DAOImpl* são criados apenas pela Fábrica dos mesmos (linha 3). Assim, quando um objeto que não seja do tipo *DAOFactory* tentar instanciar uma implementação de DAO (*DAOImpl*), um alerta será ativado.

```
1 alert * access Hibernate
2 alert myapp.Util access !myapp.Util
3 alert !DAOFactory create DAOImpl
```

Listagem 3.9: Exemplo da *Linguagem de Alerta*
Fonte: Elaborado pelo autor

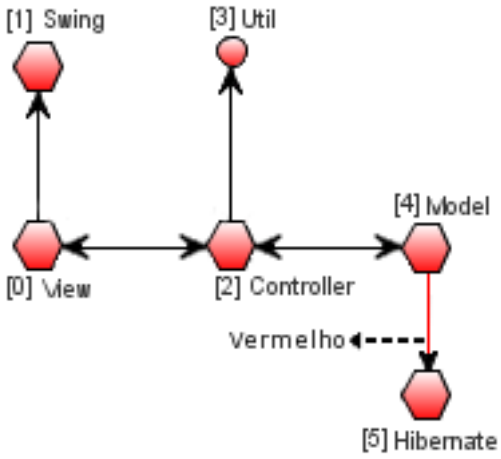

Figura 12: Análise arquitetural realizada pela *Linguagem de Alertas*
Fonte: Elaborado pelo autor

Pode-se notar que o *OG* destaca em vermelho a aresta que representa uma dependência entre o domínio *Model* e o domínio *Hibernate*, conforme especificado na definição do primeiro alerta da Listagem 3.9. Adicionalmente, em uma janela própria, esse alerta é detalhado, com mais informações sobre o objeto de origem e de destino do acesso responsável pela sua ativação.

Os relacionamentos especificados pelos dois últimos alertas da Listagem 3.9 não foram detectados em nenhum ponto da execução do programa. Assim, pode-se concluir que – pelo menos nessa execução – não foram estabelecidas dependências entre classes utilitárias e outras classes do programa e não ocorreram instanciações de DAO fora das fábricas especificamente implementadas com tal finalidade.

3.4 Ferramenta *OGV*

Nesta seção, apresenta-se a ferramenta para visualização e manipulação de *OG's* denominada *OGV Object Graph Visualization*. O *OGV* consiste em uma ferramenta não-invasiva que pode ser conectada a um sistema existente para visualizar os grafos de objetos propostos neste trabalho. A Figura 13 apresenta a tela principal da ferramenta.

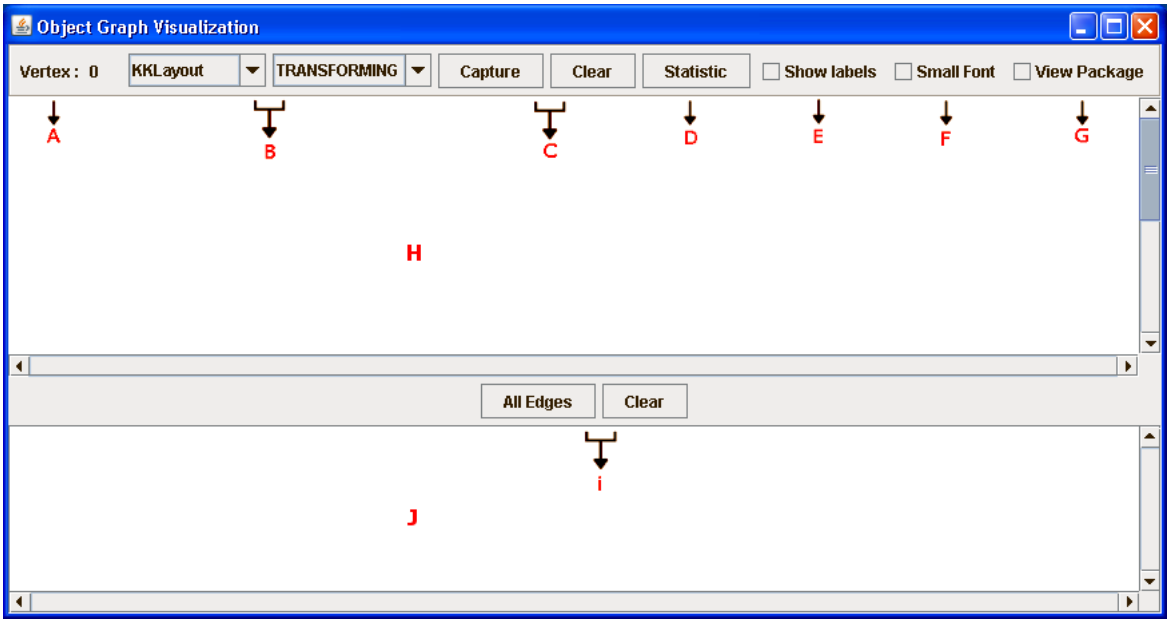

Figura 13: Tela Principal da Ferramenta *OGV*
Fonte: Elaborado pelo autor

Para descrever essa tela foram utilizadas letras (de *A* a *J*), as quais são descritas a seguir:

A: representa o número de vértices do grafo;

B: representa dois recursos oferecidos pela ferramenta para manipulação do grafo. O primeiro recurso permite ao desenvolvedor escolher um leiaute para o grafo. As opções de leiaute são: *KKLayout*, *FRLayout*, *CircleLayout*, *SpringLayout* e *ISOMLayout*. Ao selecionar um desses leiautes, a ferramenta *OGV* organiza o grafo automaticamente. O segundo recurso é usado para transformar (*transforming*) e selecionar (*picking*) o grafo. Quando o desenvolvedor seleciona a opção *transforming* pode-se fazer um *zoom in/out* e rotacionar o grafo. Quando o desenvolvedor seleciona o recurso *Picking* é possível organizar o leiaute do grafo "manualmente";

C: representa dois recursos da ferramenta – *Capture* e *Clear*. O primeiro recurso quando habilitado permite a exibição do *OG* a partir do estado atual do sistema alvo. Isto permite ao desenvolvedor começar e parar a captura no momento em que desejar. O recurso *Clear* limpa o *OG* capturado;

D: representa uma função para exibir informações como número de vértices, número de arestas, número de ocorrências das classes e o número de classes pertencentes a um pacote;

E, *F*, e *G:* respectivamente, representam opções que permitem ao desenvolvedor visualizar o nome dos vértices, reduzir o tamanho da fonte do texto e sumarizar automaticamente o *OG* de acordo com a estrutura dos pacotes;

H: painel onde é exibido o *OG* capturado;

I: representa dois recursos da ferramenta – *All Edges* e *Clear*. O primeiro recurso permite a visualização de todos os nomes das arestas de um *OG*. O segundo recurso limpa os nomes das arestas;

J: painel onde são exibidos informações sobre os nomes das arestas de um *OG*.

Para exemplificar a função *Statistic* (letra *D*), a Listagem 3.10 apresenta o uso dessa função para uma funcionalidade do sistema *JHotDraw*.

```
1 -- Statistic
2 - Class
3 Class: net.n3.nanoxml.XMLAttribute [10]
4 Class: net.n3.nanoxml.XMLElement [3]
5 Class: org.jhotdraw.samples.svg.figures.SVGRectFigure [1]
6 Class: net.n3.nanoxml.XMLWriter [1]
7
8 - Package
9 Package: net.n3.nanoxml [14]
10 Package: org.jhotdraw.samples.svg.figures [1]
11
12 - Results
13 Number of Vertex: 19
14 Number of Edges: 27
```

Listagem 3.10: Função *Statistic* da ferramenta *OGV*
Fonte: Elaborado pelo autor

A Listagem 3.10 descreve classes e pacotes pertencentes a um *OG*. Por exemplo, na linha *3* é descrito: *Class: net.n3.nanoxml.XMLAttribute [10]* que significa que existem 10 objetos da classe *net.n3.nanoxml.XMLAttribute* no *OG* capturado. Esta descrição possui a seguinte estrutura:

```
Class or Package: name [number]
```

a) *Class or Package:* determina se a informação é sobre uma classe ou pacote;

b) *name:* nome que identifica a classe ou pacote;

c) *number:* número de objetos pertencente à classe ou pacote.

O objetivo da função *Statistic* é fornecer ao desenvolvedor detalhes de um *OG*. Pode ser utilizada para determinar como sumarizar vértices de um *OG* quando não se tem informações sobre o sistema. No exemplo da Listagem 3.10, a classe *net.n3.nanoxml.XMLAttribute* é uma forte candidata a ser sumarizada, isto porque a classe possui 10 objetos no *OG*. Outra possibilidade para sumarização poderia ser o pacote *net.n3.nanoxml*, o qual possui 14 objetos. No final da estatística é apresentado o número total de vértices e arestas, conforme demonstrado nas linhas 13 e 14 da Listagem 3.10. Na próxima seção, apresenta-se a arquitetura da ferramenta *OG*.

3.5 Arquitetura

Como apresentado na Figura 14, a ferramenta *OG* possui quatro módulos principais: *OGT*, *OGV*, *Linguagem de Alertas* e o *Core*. É importante ressaltar que as setas no diagrama de componentes são no sentido de comunicação entre os componentes.

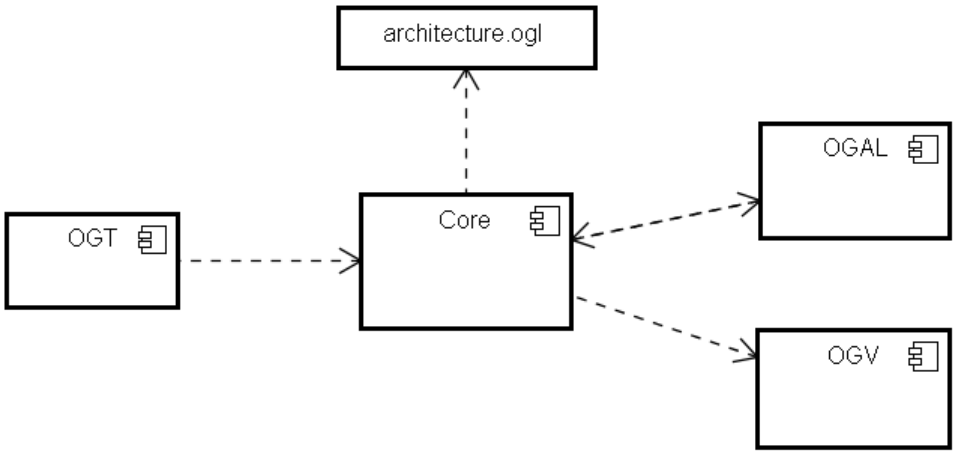

Figura 14: Módulos do *OG*
Fonte: Elaborado pelo autor

O módulo OGT (*Object Graph Tracer*) é o responsável por instrumentar o programa alvo com objetivo de capturar informações sobre chamadas de métodos, acesso a atributos, instanciação de objetos, coleções e *threads*. Para isso, utiliza a linguagem AspectJ (ORG, ECLIPSE, 2011) – uma extensão orientada a aspectos da linguagem Java que permite a injeção de código em um sistema de uma forma não intrusiva. Consequentemente, a abordagem proposta nesta dissertação de mestrado depende do compilador

`ajc`. O módulo OGV é responsável por exibir o grafo extraído e outras configurações relacionadas a exibição e manipulação do OG. Mais especificamente, o OGV depende da biblioteca *JUNG* (JUNG, 2011) para a renderização do grafo extraído. O OGAL (*Object Graph Alert Language*) é responsável pelo gerenciamento dos alertas arquiteturais. Este módulo avalia as regras de alertas definidas pelo desenvolvedor e monitora o sistema por meio das informações recebidas do *Core*. É importante ressaltar que, como se trata de análise em tempo de execução, não é possível mapear todas as dependências do sistema *a priori*. Assim, cada dependência é verificada no momento em que é estabelecida. O módulo *Core* é responsável por analisar o arquivo *architecture.ogl* onde se encontram especificados os filtros (classes ou pacotes que não serão analisados), domínios (classes ou pacotes que o desenvolvedor definiu para sumarização) e alertas arquiteturais. Outra função do módulo *Core* é o tratamento dos dados recebidos do módulo OGT e a distribuição desta informação para os diversos módulos do sistema.

O diagrama de classe que representa a ferramenta *OG* é apresentado em um estudo de caso na subseção 5.2.3.

3.6 Funcionamento

Nesta seção descreve-se o funcionamento da ferramenta *OG*. A Figura 15 apresenta a ideia básica da abordagem e da ferramenta *OG*. No primeiro momento *(1)*, o sistema é iniciado e a ferramenta *OG* é ativada automaticamente. No segundo momento *(2)*, a ferramenta *OG* fica analisando e manipulando o rastro de execução do sistema alvo. No terceiro momento *(3)*, o resultado dessa análise é exibido em formato de grafo de objetos.

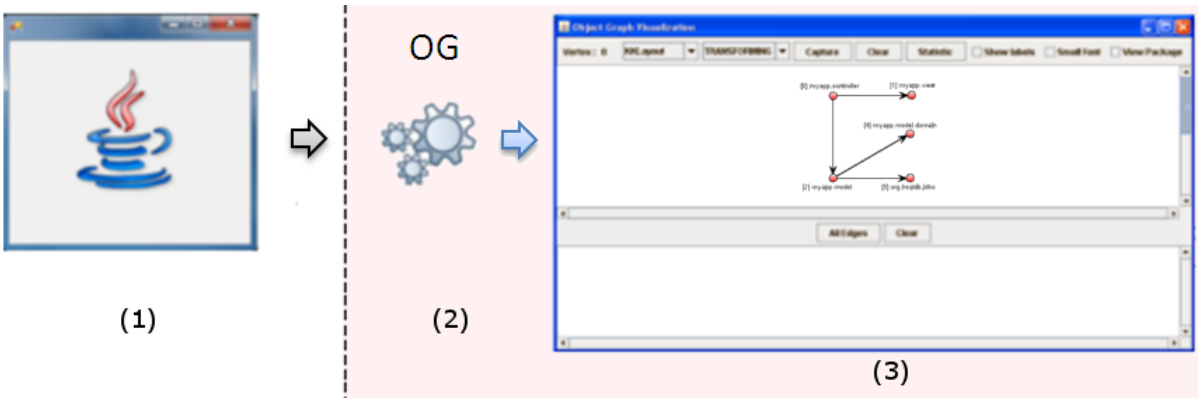

Figura 15: Processamento da Ferramenta *OG*
Fonte: Elaborado pelo autor

A ferramenta *OG* por suportar a visualização de *OG's* em modo *on-the-fly* é fácil explorar e gerar diversas perspectivas do sistema. Por exemplo, o desenvolvedor deseja ter duas perspectivas de uma funcionalidade do sistema. Deve-se realizar os seguintes passos:

a) Iniciar o sistema alvo (sistema em JAVA). A ferramenta *OG* é ativada automaticamente (conforme ilustrada na Figura 16);

b) O grafo de objetos é exibido somente quando o desenvolvedor aciona o botão *capture* e executa uma determinada funcionalidade do sistema alvo (momento *(2)*);

c) Com a exibição do grafo de objetos, caso necessário o desenvolvedor pode utilizar o recurso *Statistic* para conhecer melhor o grafo;

d) Para visualizar a mesma funcionalidade, mas em outra perspectiva, o desenvolvedor deve limpar o grafo (acionando o botão *clear*);

e) O desenvolvedor pode alterar filtros e/ou domínios especificados no arquivo *architecture.ogl*;

f) Para gerar outra perspectiva deve-se repetir o passo 2, por exemplo, seria gerado um novo grafo de objetos (momento *(3)*).

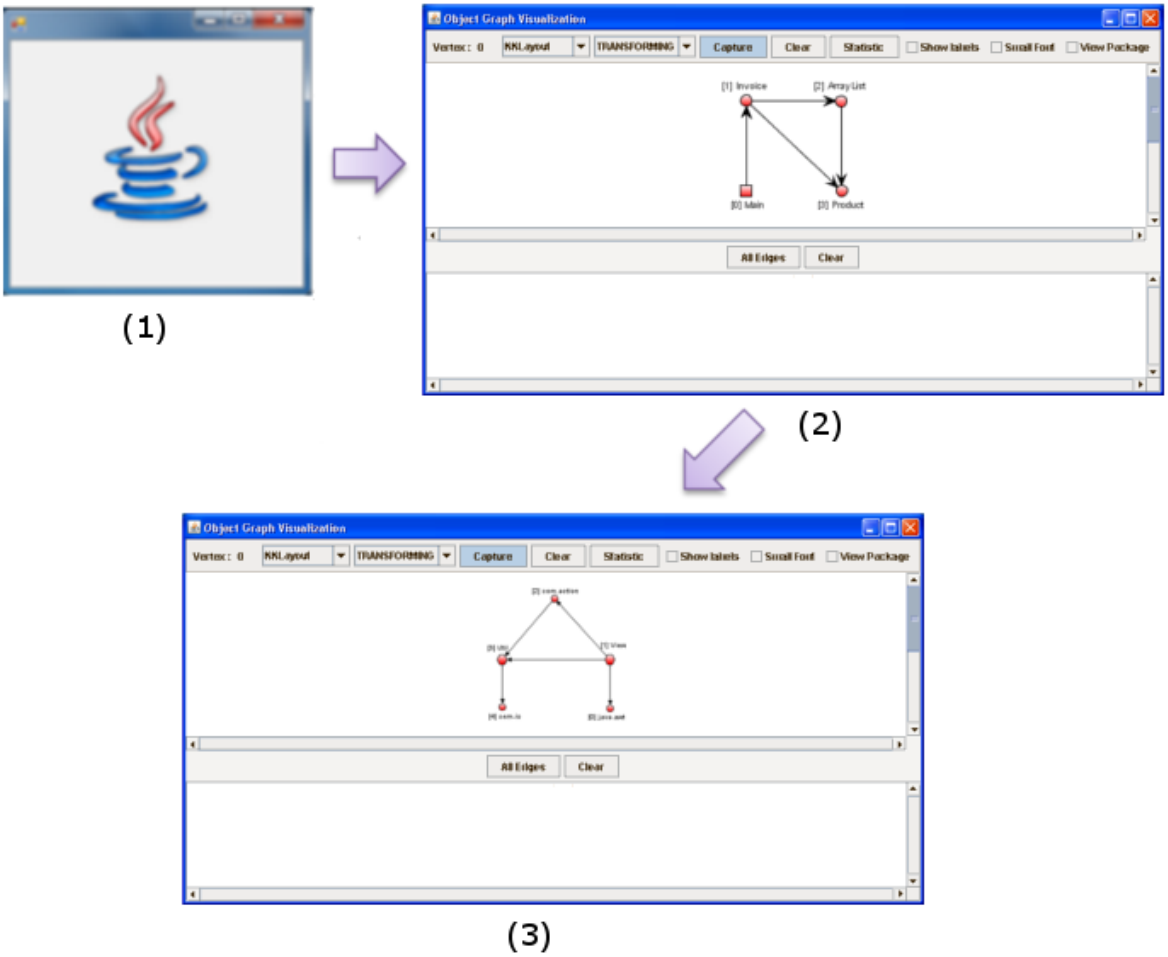

Figura 16: Funcionamento da Ferramenta *OG*
Fonte: Elaborado pelo autor

4 COMPARAÇÃO COM DIAGRAMAS UML

Neste capítulo é realizado comparações dos diagramas da UML com o *OG* para destacar pontos fracos e fortes da abordagem apresentada nesta dissertação. É importante ressaltar que a abordagem apresentada nesta dissertação não gera um diagrama da UML (por exemplo, diagrama de sequência ou comunicação) e sim outra notação gráfica chamada *OG*. Para exemplificar o motivo, a Figura 17 representa um diagrama de comunicação do Exemplo 3, com 5 domínios e 1 objeto. Comparando a Figura 10 com a Figura 17 percebe-se que na Figura 10 é possível identificar facilmente quais são os objetos e os domínios devido a representação diferenciada que o *OG* adota. Já no diagrama de comunicação, a representação de objetos, classes com membros estáticos e domínios é a mesma (todos são representados por um retângulo). A diferença são os estereótipos que podem identificar cada um desses elementos. Por exemplo, o domínio *View* pode possuir um estereótipo identificando que é um domínio. No entanto, é necessário a leitura desses estereótipos (quando eles existirem) para identificar o que representa cada retângulo. Por outro lado, o *OG* foi idealizado para ser um grafo simples onde desenvolvedores de sistemas possam identificar rapidamente quais são os objetos, domínios e classes com membros estáticos, utilizando para isso uma notação gráfica diferenciada, ao contrário do diagrama de comunicação, onde essa anotação é textual.

Os diagramas da UML apresentados para a comparação nesta seção, foram criados "manualmente", isto é, sem a utilização de ferramentas de engenharia reversa. Nos estudos de casos do *myAppointments* na subseção 5.3.1, os diagramas de classe e sequência foram criados "automaticamente", isto é, por uma ferramenta utilizando engenharia reversa. Como os dois diagramas da UML foram criados de formas diferentes é possível comparálos e verificar se a avaliação feita na subseção 4.6 possui o mesmo resultado para o estudo de caso do *myAppointments*.

Para comparação, foi escolhido o código do Exemplo 2 (conforme a Listagem 3.2) por apresentar a possibilidade de criar várias instâncias de um objeto e pela utilização de

threads, os quais são recursos utilizados na maioria dos sistemas. O objetivo é analisar qual notação gráfica fornece informações mais precisa sobre o código Exemplo 2.

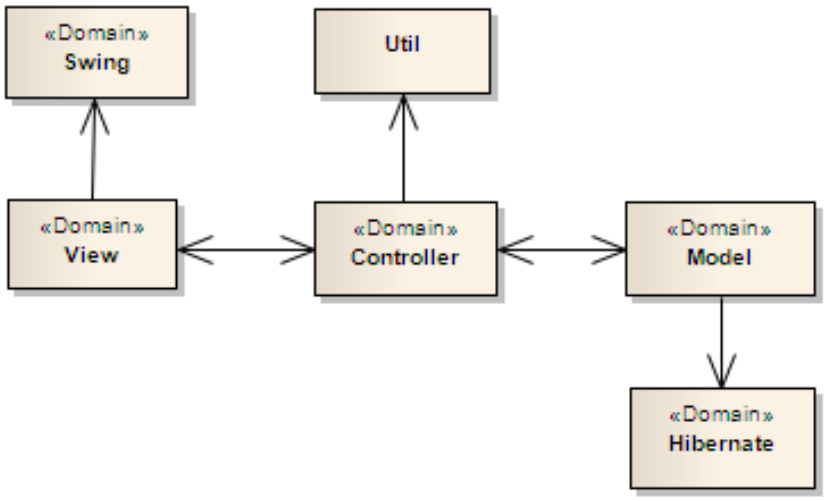

Figura 17: Diagrama de Comunicação do Exemplo 3
Fonte: Elaborado pelo autor

4.1 Diagrama de Classe

A Figura 18 representa o diagrama de classe do código do Exemplo 2 da subseção 3.1.1. A Figura 18 apresenta três classes *Main*, *Box* e *Product*. Pode-se destacar os relacionamentos entre a classe *Main* com *Box* e *Box* com *Product*, os quais demonstram a possibilidade de criar vários objetos do tipo *Box* e do tipo *Product*.

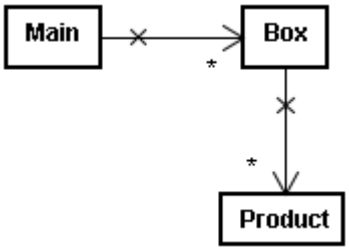

Figura 18: Diagrama de Classe do Exemplo 2
Fonte: Elaborado pelo autor

4.2 Diagrama de Sequência

A Figura 19 representa uma classe *Main* e dois objetos *Box* e *Product*. É importante ressaltar que existe uma notação chamada *frame* do tipo *loop* que engloba o ato de

instanciar um objeto *Box* e o ato de invocar o método *run*. Isto significa que as ações que pertencem ao *loop* podem ser repetidas um determinado número de vezes. Sobre a ordem das comunicações, a classe *Main* instancia o objeto *Box*. No segundo momento, a classe *Main* invoca o método *run* do objeto *Box*. Por último, o objeto *Box* em uma chamada assíncrona instancia um objeto do tipo *Product*. É importante ressaltar que quando ocorre uma chamada assíncrona o sistema não precisa esperar pela resposta. Em sistemas *multi-thread* e em sistemas que usam plataformas de *middleware* orientadas a mensagens, chamadas assíncronas são mais frequentes (FOWLER, 2003).

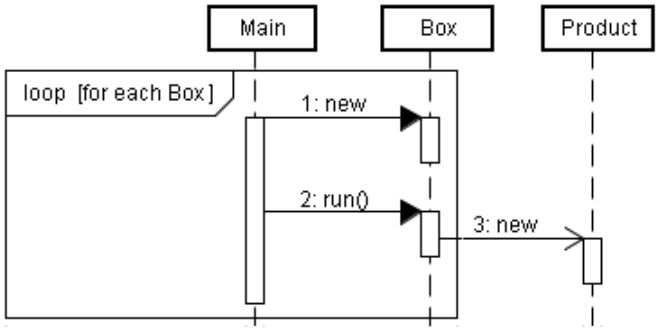

Figura 19: Diagrama de Sequência do Exemplo 2
Fonte: Elaborado pelo autor

4.3 Diagrama de Comunicação

A Figura 20 representa uma classe *Main* e dois objetos *Box* e *Product*. Sobre a ordem dos relacionamentos, primeiro a classe *Main* instancia o objeto *Box* (passo 1). Depois o objeto *Main* invocou o método *run* do objeto *Box* (passo 2). Por último, em uma chamada assíncrona o objeto *Box* instância um objeto do tipo *Product* (passo 3).

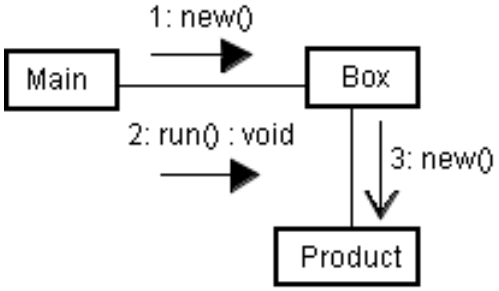

Figura 20: Diagrama de Comunicação do Exemplo 2
Fonte: Elaborado pelo autor

4.4 Diagrama de Objetos

A Figura 21 representa uma classe *Main* e quatro objetos, dois objetos *Box* e dois objetos *Product*. Sobre os relacionamentos podemos observar as dependências entre os objetos. Por exemplo, o objeto *Box* referencia um objeto *Product* e o *Main* referencia *Box*, mas não podemos afirmar nada sobre a ordem de criação dos objetos.

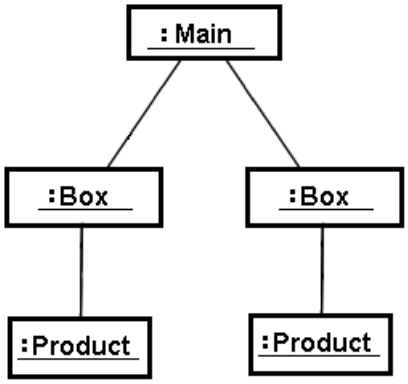

Figura 21: Diagrama de Objetos do Exemplo 2
Fonte: Elaborado pelo autor

4.5 Diagrama de Pacotes

O diagrama de pacotes não será apresentado e comparado porque o *OG* permite visualização por pacotes.

4.6 Conclusão

A descrição do *OG* representado pela Figura 9 encontra-se na subseção 3.1.1. Realiza-se a seguir algumas comparações entre esse *OG* e os diagramas da UML mencionados anteriormente:

a) *Diagrama de classe:* representado na Figura 18 percebe-se que o diagrama não possui informação sobre *threads*, ordem da comunicação e sobre o número de instâncias;

b) *Diagrama de objetos:* representado na Figura 21 percebe-se que o diagrama não possui informação sobre *threads*;

c) *Diagrama de interação:* o diagrama de sequência mostrado na Figura 19 e o diagrama de comunicação mostrado na Figura 20 percebe-se que possuem informações sobre mensagens assíncronas, mas não é possível afirmar quantos objetos serão impactados.

A Tabela 1 representa uma comparação dos diagramas UML – versão 2.0 – com o *OG*. Para comparação dos diagramas UML foram avaliadas as seguintes propriedades:

a) *Escalabilidade:* significa que o diagrama é escalável para sistemas de grande porte;

b) *Threads:* significa que os diagramas representam de forma adequada sistemas com múltiplas *threads*;

c) *Expressividade:* significa que o diagrama apresenta grande expressividade nas diversas representações *OO*;

d) *Usabilidade:* significa a facilidade do uso do diagrama;

e) *Granularidade:* significa que o diagrama representa melhor uma granularidade fina, média ou grossa.

Para avaliação das propriedades foram utilizados os seguintes critérios:

a) *Não:* significa que o diagrama não atende bem o critério em questão;

b) *Sim:* significa que o diagrama atende bem o critério em questão;

c) *Parcial:* significa que o diagrama atende parcialmente o critério em questão.

Sobre a classificação apresentada na Tabela 1, podemos realizar os seguintes comentários

a) *Escalabilidade:* o diagrama de objetos não é escalável para sistemas maiores. O *OG* e o diagrama de classe são escaláveis porque permitem representações de mais alto nível;

b) *Threads:* o *OG* apresenta como os objetos estão sendo manipulados por *threads* diferentes conforme mostrado na Figura 9. Para o diagrama de classe, considerando o Exemplo 2 (Listagem 3.2), não seria possível raciocinar as *threads* que ocorrem em tempo de execução;

Tabela 1: Comparação entre diagramas UML com *OG*

	Classe	Sequência	Comunicação	Objetos	*OG*
Escalabilidade	Sim	Parcial	Parcial	Não	Sim
Threads	Não	Parcial	Parcial	Não	Sim
Expressividade	Sim	Sim	Sim	Parcial	Parcial
Usabilidade	Parcial	Sim	Sim	Sim	Sim
Granularidade	Média; Grossa	Fina	Fina	Fina	Fina; Média; Grossa

Fonte: Elaborado pelo autor

c) *Expressividade:* os diagramas de classe, sequência e de comunicação possuem representações que permitem modelar diversas situações. O *OG* atende parcialmente porque possui várias representações para vértices e arestas, mas por exemplo, não possui notação para modelar uma condição *If-Else*. O *OG* e o diagrama de objetos comparado com os outros diagramas representam melhor as instâncias dos objetos conforme mostrado na Figura 9 e 21;

d) *Usabilidade:* o diagrama de classe devido ao poder de expressividade, isto é, expressa diversas situações implica no aumento da dificuldade do seu uso (FOWLER, 2003). O diagrama de comunicação comparado com o diagrama de sequência é considerado mais fácil de manipular (por sua organização não ser horizontal) (FOWLER, 2003);

e) *Granularidade:* o diagrama de classe permite uma representação com uma granularidade média ou grossa. O diagrama de sequência e de comunicação poderia representar granularidade grossa, mas eles perderiam a sua essência. O *OG* pode representar granularidade fina, média ou grossa por meio das visualizações por objetos, pacotes, domínios ou com as combinações das visões.

Todos os diagramas da UML são importantes e amplamente utilizados, mas alguns diagramas, até pelo seu objetivo, atendem melhor ou não aos critérios que definimos. O diagrama de classe pode fornecer uma boa representação estrutural, mas somente com o diagrama de classe é difícil compreender um sistema (FOWLER, 2003). O diagrama de sequência segundo (FOWLER, 2003) não representa uma definição precisa do comportamento. Outro problema do diagrama de sequência é quando se torna grande, pois sabe-se que existe um limite para compreensão humana de grandes estruturas aninhadas (BOOCH; RUMBAUGH; JACOBSON, 2005). Para compreender um sistema, podem ser necessárias di-

versas visões ou perspectivas arquiteturais diferentes (KRUCHTEN, 1995; HOFMEISTER; NORD; SONI, 1999; SARTIPI; KONTOGIANNIS; MAVADDAT, 2000; RIVA; RODRIGUEZ, 2002; BOOCH; RUMBAUGH; JACOBSON, 2005). O *OG* permite ao desenvolvedor visualizar diferentes visões, além disso, é mais preciso para representação do comportamento do sistema, possibilita a manipulação do nível de detalhes (granularidades fina, média ou grossa) e possui uma ferramenta que suporta a visualização de *OG's* em modo *on-the-fly*.

5 AVALIAÇÃO

Para ilustrar o uso da ferramenta *OG* tanto no quesito de visualização quanto no uso da *Linguagem de Alertas*, neste capítulo são apresentados estudos de casos. O objetivo central é deixar claro os benefícios que a ferramenta *OG* pode proporcionar.

5.1 Sistemas Alvos

Para avaliação, foram utilizados três sistemas. O sistema *myAppointments* é um sistema de gerenciamento de informações pessoais. De forma sucinta, esse sistema segue o padrão arquitetural MVC e possui 1.215 linhas de código, 16 classes e três interfaces. A Figura 22 apresenta a tela principal do sistema. Basicamente, *myAppointments* permite aos desenvolvedores criar, pesquisar, atualizar e remover compromissos. Esse sistema foi originalmente projetado para ilustrar o emprego de soluções para conformação estática de arquiteturas de software (PASSOS *et al.*, 2010).

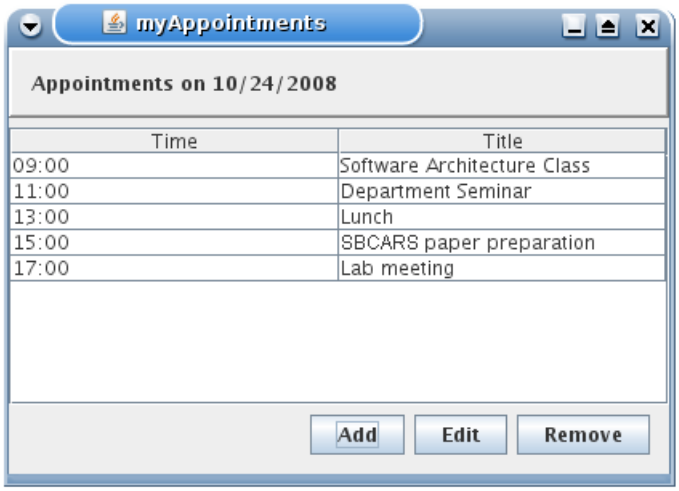

Figura 22: Sistema *myAppointments*
Fonte: Elaborado pelo autor

O segundo sistema, *JHotDraw*, é um *framework* bem conhecido que auxilia no desenvolvimento de aplicativos gráficos. Esse *framework* possui 15 KLOC e 200 classes.

Do ponto de vista arquitetural, ele faz o uso de diversos padrões de projeto e segue o padrão arquitetural MVC. A Figura 23 apresenta a tela principal do exemplo que foi utilizado.

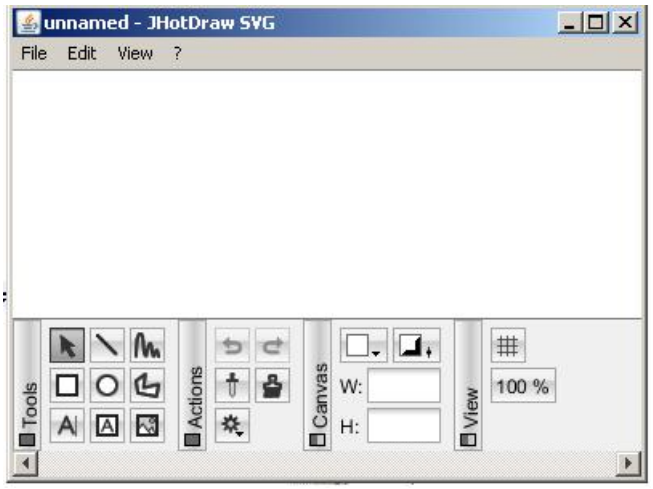

Figura 23: Exemplo do *JHotDraw*
Fonte: Elaborado pelo autor

O terceiro sistema é a própria abordagem apresentada nesta dissertação, isto é, será apresentado um *OG* da ferramenta *OG*.

5.2 Recuperação Arquitetural

A ferramenta *OG* permite ao desenvolvedor representar por meio dos domínios os principais componentes de um sistema. Uma vez modelados os domínios o desenvolvedor poderá analisar o comportamento do sistema executando diversas funcionalidades. Deste modo, é possível mapear e raciocinar sobre os relacionamentos entre os principais componentes.

5.2.1 Arquitetura do JHotDraw

Para entendimento da arquitetura do sistema *JHotDraw*, foi primeiro gerado o grafo a partir da inicialização do sistema (Figura 23) sem qualquer tipo de filtro ou recurso de sumarização. Dessa maneira, foi gerado um grafo com milhares de vértices. Conforme pode ser observado na Figura 24, esse grafo é ilegível.

Para tornar o grafo mostrado legível e útil, ele foi novamente capturado com uso de domínios. A definição de domínios tomou como base a divisão de classes proposta por Abi-Antoun e Aldrich (ABI-ANTOUN; ALDRICH, 2009b). Nessa definição, que segue o

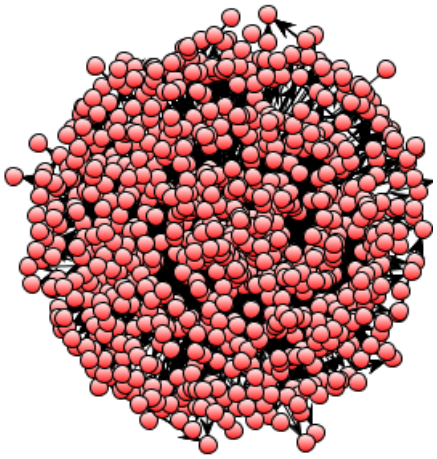

Figura 24: *OG* do *JHotDraw* sem filtro ou sumarização
Fonte: Elaborado pelo autor

padrão arquitetural MVC, objetos de apresentação, como instâncias de `DrawingEditor` e `DrawingView`, foram definidos em domínios com prefixo `View` (linhas 6 a 9). Os objetos responsáveis pela lógica de apresentação, como instâncias de `Tool`, `Command` e `Undoable`, foram definidos em um domínio chamado `Controller` (linha 4). Por fim, objetos de modelo, como instâncias de `Drawing`, `Figure` e `Handle`, foram definidos em um domínio chamado `Model` (linhas 1 e 2). A Listagem 5.1 apresenta uma amostra da definição dos domínios:

```
1  domain Model:          org.jhotdraw.draw.Figure+,
2                         org.jhotdraw.draw.DrawingEditor+ ...
3
4  domain Controller:     org.jhotdraw.draw.tool.Tool+ ...
5
6  domain View(Draw):     org.jhotdraw.draw.DrawingView+ ...
7
8  domain View:           org.jhotdraw.app.Application+,
9                         org.jhotdraw.app.View+ ...
10
11 domain Util:           org.jhotdraw.util.** ...
```

Listagem 5.1: Domínios do *JHotDraw*
Fonte: Elaborado pelo autor

Assim, foram definidos cinco domínios – dois relacionados à visão, um ao controle, um ao modelo e um de classes utilitárias – conforme pode ser observado na Figura 25. A

fim de deixar o grafo mais legível, foi utilizado um recurso da ferramenta que faz com que vértices que se comunicam em ambos os sentidos tenham suas arestas unificadas em uma única aresta bidirecional. Ao contrário da Figura 24, a Figura em questão traz diversas informações úteis ao desenvolvedor. Ela permite, por exemplo, visualizar as três camadas de objetos existentes em sistemas construídos segundo o padrão arquitetural MVC.

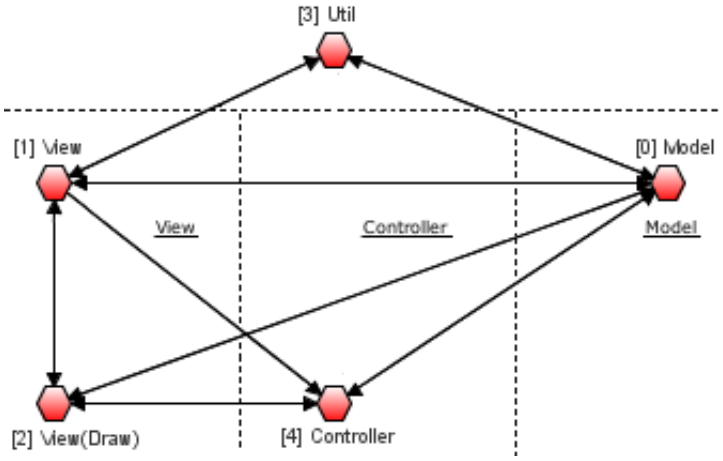

Figura 25: *OG* do *JHotDraw* com definição de domínios
Fonte: Elaborado pelo autor

Portanto, esse cenário indica que, para a tarefa de compreensão da arquitetura de um sistema, é importante que se obtenha inicialmente uma visão macro (de granularidade grossa) do sistema e, à medida que se compreende a arquitetura, pode-se refinar a granularidade do grafo. Por exemplo, caso o desenvolvedor pretenda obter mais detalhes de um domínio, ele pode expandir o mesmo, isto é, desagrupar o domínio de forma que seja exibida toda a comunicação interna entre seus objetos e classes (como demonstrado na próxima Subseção).

5.2.2 Explorando funcionalidades do JHotDraw

Nesta subseção, apresenta-se quatro funcionalidades do sistema *JHotDraw*. O objetivo é explorar como os componentes definidos pelos domínios da Listagem 5.1 se comportam. Foram realizadas duas mudanças: a primeira foi a unificação dos domínios *View* e *View(Draw)* resultando em somente um domínio *View* e a segunda foi a retirada da definição *org.jhotdraw.draw.Figure+* do domínio *Model*.

A Figura 26 apresenta essas quatros funcionalidades (foram adicionados letras nos quadrantes para facilitar a identificação). As funcionalidades representadas pelas letras são:

Quadrante A: representa a funcionalidade de adicionar um retângulo. Pelo *OG*, percebemos que o objeto *SVGRectFigure* é o responsável pela manipulação da figura;

Quadrante B: representa a funcionalidade de adicionar um círculo. Pelo *OG* percebemos que o objeto *SVGEllipseFigure* é o responsável pela manipulação da figura. Comparando com a letra *A* percebe-se que o comportamento é muito semelhante;

Quadrante C: representa a funcionalidade de preencher com uma cor uma figura (no caso um quadrado). Pelo *OG* percebemos que o objeto *SVGRectFigure* é manipulado. Comparando com as letras anteriores (*A* e *B*) o comportamento é um pouco diferente (na ordem da criação dos vértices);

Quadrante D: representa a funcionalidade de adicionar uma linha. Pelo *OG* percebemos que o objeto *SVGBezierFigure* é o responsável pela manipulação da figura. Comparando com as letras *A* e *B* (funções similares que adicionam uma figura) percebe-se que os dois objetos *SVGBezierFigure* não relacionam com os domínios *Controller* e *View*. Outra diferença ocorre na ordem de criação dos vértices.

Na Figura 26, percebe-se que em todos os casos (independentemente da ação), os quatro domínios estão presentes: *Controller*, *View*, *Model* e *Util*. Dependendo da ação, a comunicação pode variar entre os domínios conforme mostrado nos quadrantes *C* e *D*.

Essencialmente este estudo de caso destaca três vantagens de *OG's*. A primeira é que uma vez definidos os domínios para uma determinada necessidade pode utilizá-los para explorar o sistema e verificar o seu comportamento. A segunda vantagem é a fácil manipulação dos domínios para atender a necessidade do desenvolvedor. No caso, foram feitas duas modificações simples e foi possível explorar funcionalidades do sistema gerando novas visões. Por último, o fato da ferramenta suportar a visualização de *OG's* em modo *on-the-fly*. Esta característica da ferramenta viabiliza a geração de várias visões diferentes do sistema de maneira rápida e prática.

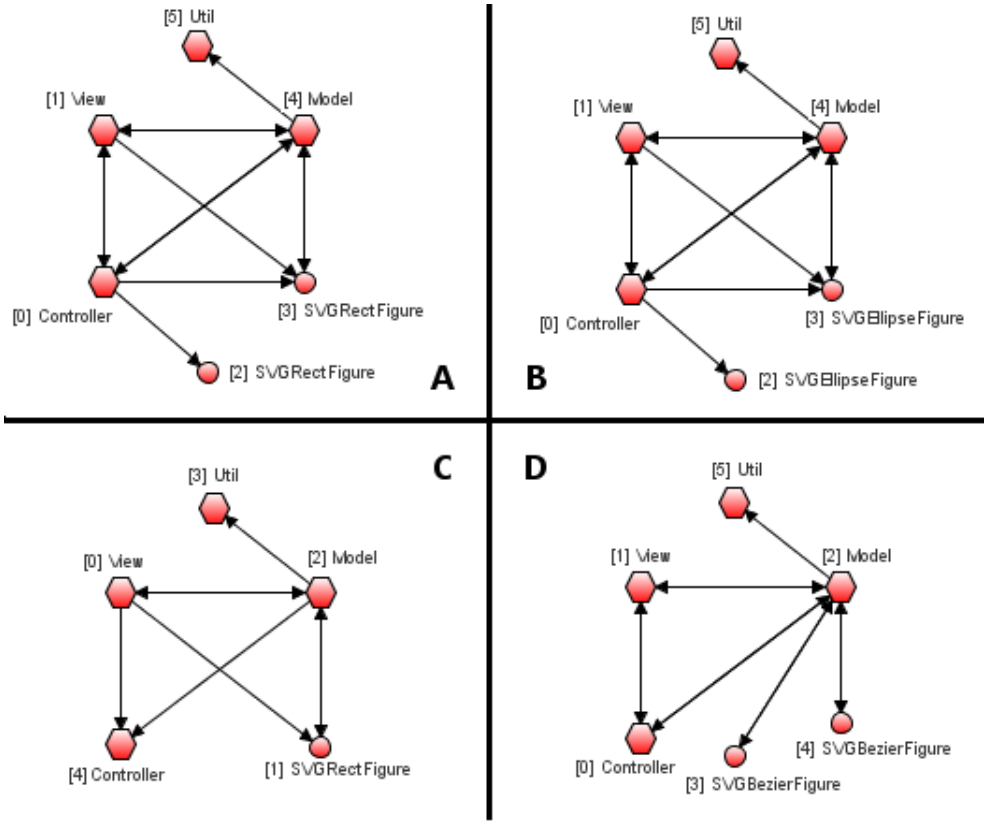

Figura 26: *OG* das funcionalidades do *JHotDraw*
Fonte: Elaborado pelo autor

5.2.3 Ferramenta OG

Nesta seção apresenta-se um *OG* da ferramenta *OG*. Para uma comparação, foi extraído o diagrama de classe com as principais classes da ferramenta *OG* conforme mostrado na Figura 27.

A Figura 27 apresenta 7 classes e o módulo *OGT*. No entanto, não será feito o detalhamento do diagrama de classe mostrado nessa figura porque o detalhamento do *OG* equivale para o diagrama de classe. Para extrair o *OG* da ferramenta *OG*, foi necessário duas ferramentas OG. A primeira foi plugada em um sistema alvo e a segunda ferramenta foi plugada na primeira ferramenta OG. O *OG* extraído é apresentado na Figura 28. Primeiro, o *OG* mostrado nessa figura apresenta quatro domínios:

a) *ObjectId*: armazena dados sobre os objetos extraídos, como código *hash*, nome da classe, *thread*, etc. Os dados recebidos pela classe *OglCore* são transformados em *ObjectId* e enviados para a classe *ObjectGraph*;

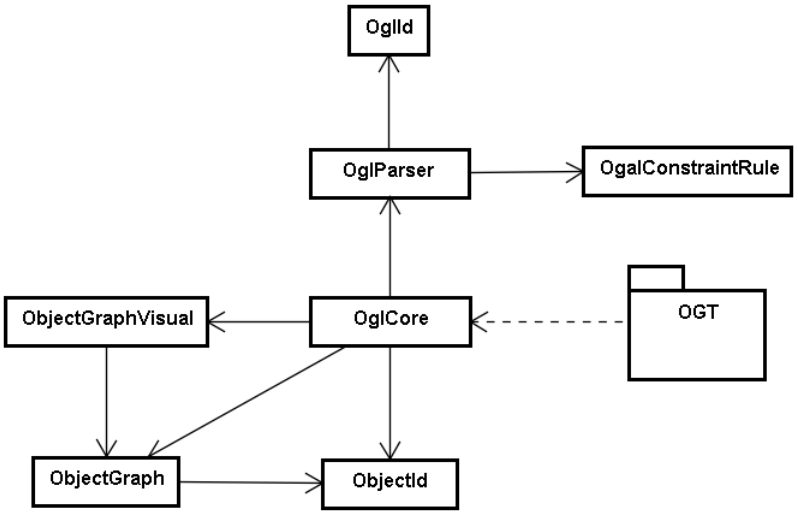

Figura 27: Diagrama de Classe do *OG*
Fonte: Elaborado pelo autor

b) *ObjId*: representa filtros, domínios ou alertas arquiteturais. Como em um sistema pode possuir diversos filtros, domínios ou alertas arquiteturais então as classes *ObjId*, *ObjIdClass* e *ObjIdClassType* foram representadas em domínios.

Como pode ser observado na Figura 28, a execução da ferramenta começa pelo objeto `ObjCore` (vértice 0), que recebe informações do módulo OGT, analisa as informações recebidas verificando se dever ser aplicados filtros ou domínios. No segundo momento, o `OglCore` comunica-se com um membro estático da classe `OglParser` (vértice 1) e depois acessa um serviço do objeto *OglParser* (vértice 2). Para análise de filtros e domínios, o objeto *OglParser* acessa os serviços do `OglId`, `OglIdClass` e `OglIdClassType` (vértices 3, 4 e 5, respectivamente). Em seguida, o `OglCore` verifica os possíveis alertas, acessando serviços prestados pelo objeto *OglParser* (vértice 2) que acessa serviços do objeto `OglConstraintRule` (vértice 6). Finalmente, o `OglCore` (vértice 0) atualiza o módulo OGV, representada por objetos das classes `ObjectId` (vértice 7), `ObjectGraph` (vértice 8) e `ObjectGraphVisual` (vértice 9).

Portanto, se comparamos o diagrama de classe (Figura 27) e diagrama de componentes (Figura 14) com o *OG* (Figura 28) podemos perceber que o *OG* fornece mais detalhes do comportamento do sistema, portanto, propicia uma visão diferente ajudando no entendimento da ferramenta.

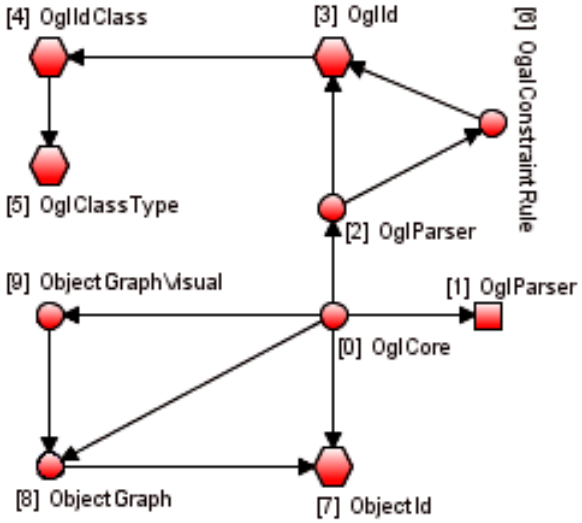

Figura 28: *OG* da ferramenta *OG*
Fonte: Elaborado pelo autor

5.3 Compreensão de funcionalidades

A ferramenta *OG* pode ser utilizada também para ajudar o desenvolvedor compreender funcionalidades de um sistema. Mesmo quando existe um diagrama de classe do sistema, localizar quais classes estão envolvidas, como as classes se interagem e por onde começa uma determinada funcionalidade é uma tarefa árdua para o desenvolvedor. É importante mencionar que, conforme abordado na Seção 3, existem diversos recursos que permitem modificar o *OG* de forma a facilitar o entendimento por parte do desenvolvedor.

5.3.1 Manutenção myAppointments

Dado o sistema *myAppointments*, suponha que o desenvolvedor esteja planejando uma modificação na remoção de um compromisso. Contudo, ele desconhece o comportamento do sistema no processo de remoção de registros. Para isso, o desenvolvedor pode utilizar a ferramenta *OG* para capturar o rastro de execução no ato da ação. Por exemplo, em uma primeira visualização, pode-se optar pela visualização em pacotes. Ainda, caso pretenda obter mais detalhes, pode-se optar pela visualização de objetos.

Concretizando discussão acima, na Figura 29 apresenta-se o grafo de pacotes somente do processo de remoção de um compromisso. Como pode se observar, foram criados cinco vértices (representando pacotes): `myapp.controller` (controle), `myapp.view` (visão), `myapp.model` (modelo), `org.hsqldb.jdbc` (persistência) e `myapp.model.domain`

(objetos de domínio). As arestas representam as comunicações e, nesse caso, percebe-se que o pacote de controle estabelece comunicação com pacotes de visão e modelo, e que o modelo se comunica com o módulo de persistência e com objetos de domínio.

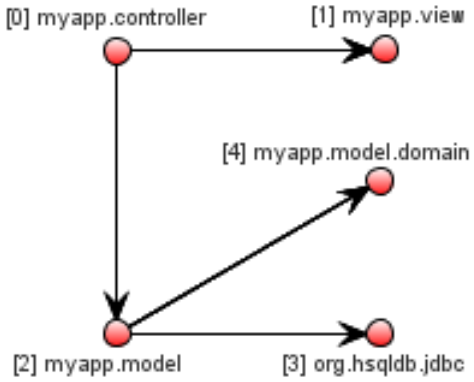

Figura 29: *OG* com sumarização por pacotes
Fonte: Elaborado pelo autor

No entanto, o desenvolvedor pode estar interessado em uma visualização mais detalhada do sistema. Para isso, o mesmo grafo mostrado na Figura 29 pode ser visualizado em granularidade mais fina – em nível de objetos – como mostrado na Figura 30. Sabe-se que um *OG* em granularidade fina não é escalável. No entanto, um grafo detalhado de apenas uma determinada funcionalidade é mais viável. Como pode se observar, foram criados agora sete vértices: objeto `AgendaController` (ponto de entrada da aplicação), objetos `AgendaView` e `AgendaDAO`, classe estática `DB` e objetos `JDBCConnection`, `DAOCommand` e `App`. Veja que esse grafo oferece mais informações sobre o comportamento do sistema, mostrando, por exemplo, que objetos DAO são usados para acesso a dados, que a comunicação com o banco de dados é feita por meio de JDBC etc.

O desenvolvedor pode ainda definir domínios para melhor representar o papel arquitetural de grupos de objetos, para sumarizar parte do sistema que já compreendeu ou para sumarizar alguma parte que não considere de muita relevância. Por exemplo, suponha que o desenvolvedor defina o domínio `Model` como sendo composto por objetos de quaisquer classes do pacote `myapp.model`, conforme mostrado na Listagem 5.2.

```
1 domain Model: myapp.model.**
```

Listagem 5.2: Definição de domínio no `myAppointments`
Fonte: Elaborado pelo autor

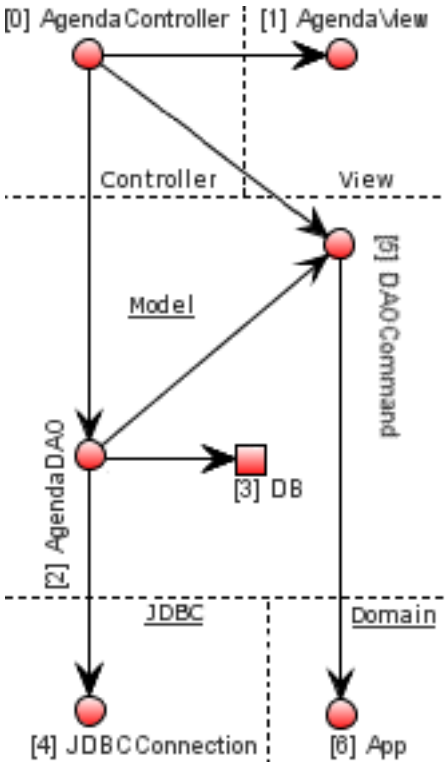

Figura 30: *OG* em granularidade fina
Fonte: Elaborado pelo autor

Após a definição desse domínio, o mesmo grafo das Figuras 29 e 30 é visualizado na Figura 31. Nesse grafo, todos os vértices que representavam objetos ou classes da camada de modelo – objetos `AgendaDAO`, `DAOCommand`, `App` e a classe estática `DB` – foram sumarizados em um único vértice, `Model`. Assim, foram criados somente quatro vértices, o que simplifica e facilita o entendimento do grafo pelo desenvolvedor.

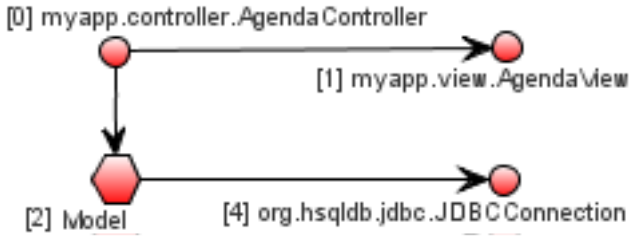

Figura 31: *OG* com definição do domínio `Model`
Fonte: Elaborado pelo autor

Em resumo, quando do interesse do desenvolvedor, a abordagem permite recuperar diagramas mais detalhados que aqueles sumarizados por pacotes (como o diagrama mostrado na Figura 30). Por outro lado, se necessário, ela é flexível a ponto de tam-

bém permitir a recuperação de diagramas de mais alto nível que aqueles sumarizados por pacotes (como o diagrama mostrado na Figura 31).

Para finalizar a avaliação descrita na presente seção, foram utilizados outros dois diagramas. A Figura 32 mostra o diagrama de classe do sistema *myAppointments*. Esse diagrama de classe possui 18 classes e 4 interfaces. Já na Figura 33 mostra o diagrama de sequência do sistema *myAppointments*. Por causa do tamanho do diagrama de sequência, foram removidas algumas classes do diagrama. Mesmo assim, percebe-se que o diagrama de sequência possui um tamanho razoável. Esse diagrama possui 7 classes.

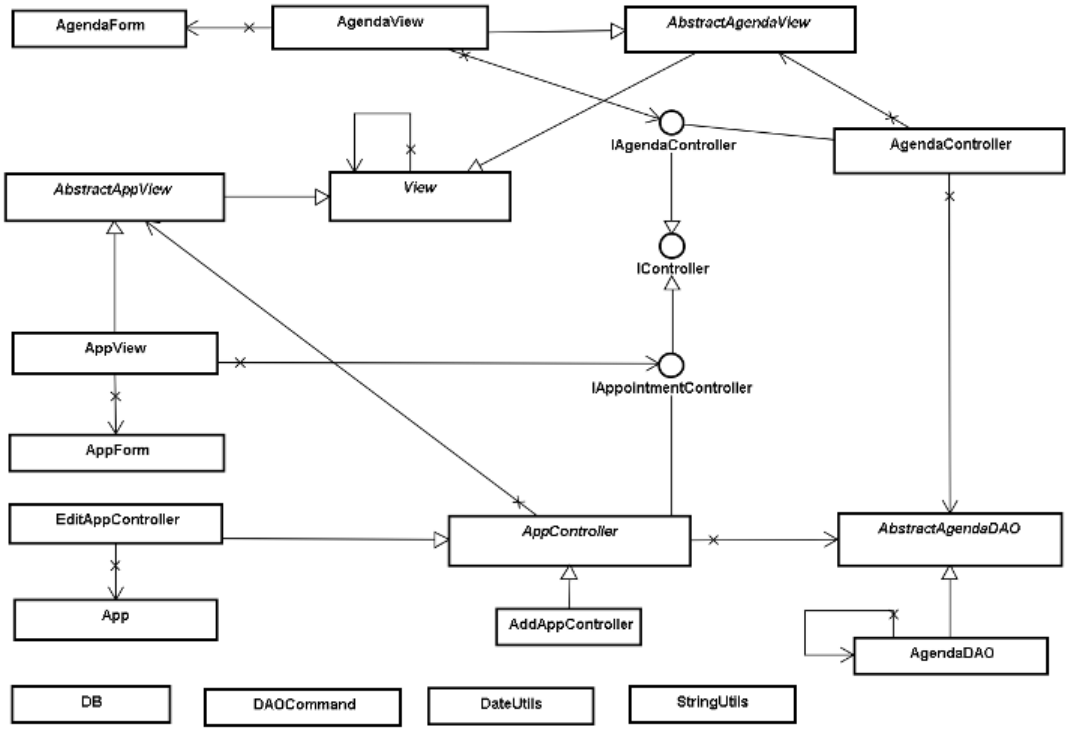

Figura 32: Diagrama de Classe do sistema *myAppointments*
Fonte: Elaborado pelo autor

Fazendo uma comparação das Figuras 32 e 33 com as Figuras 29, 30 percebe-se que existe um objeto *JDBCConnection* no *OG* que não é mostrado nas Figuras 32 e 33 (os filtros aplicados no diagrama de sequência não impactam nesta análise). Outro detalhe é o número de instâncias do objeto *App*. No *OG* esta informação é precisa.

Em todos os exemplos analisados neste cenário de aplicação, foram filtrados os objetos de classes do pacote *Java AWT*, da biblioteca padrão de Java (*java.lang*) e utilitários do sistema, uma vez que eles não são relevantes para o entendimento de um sistema.

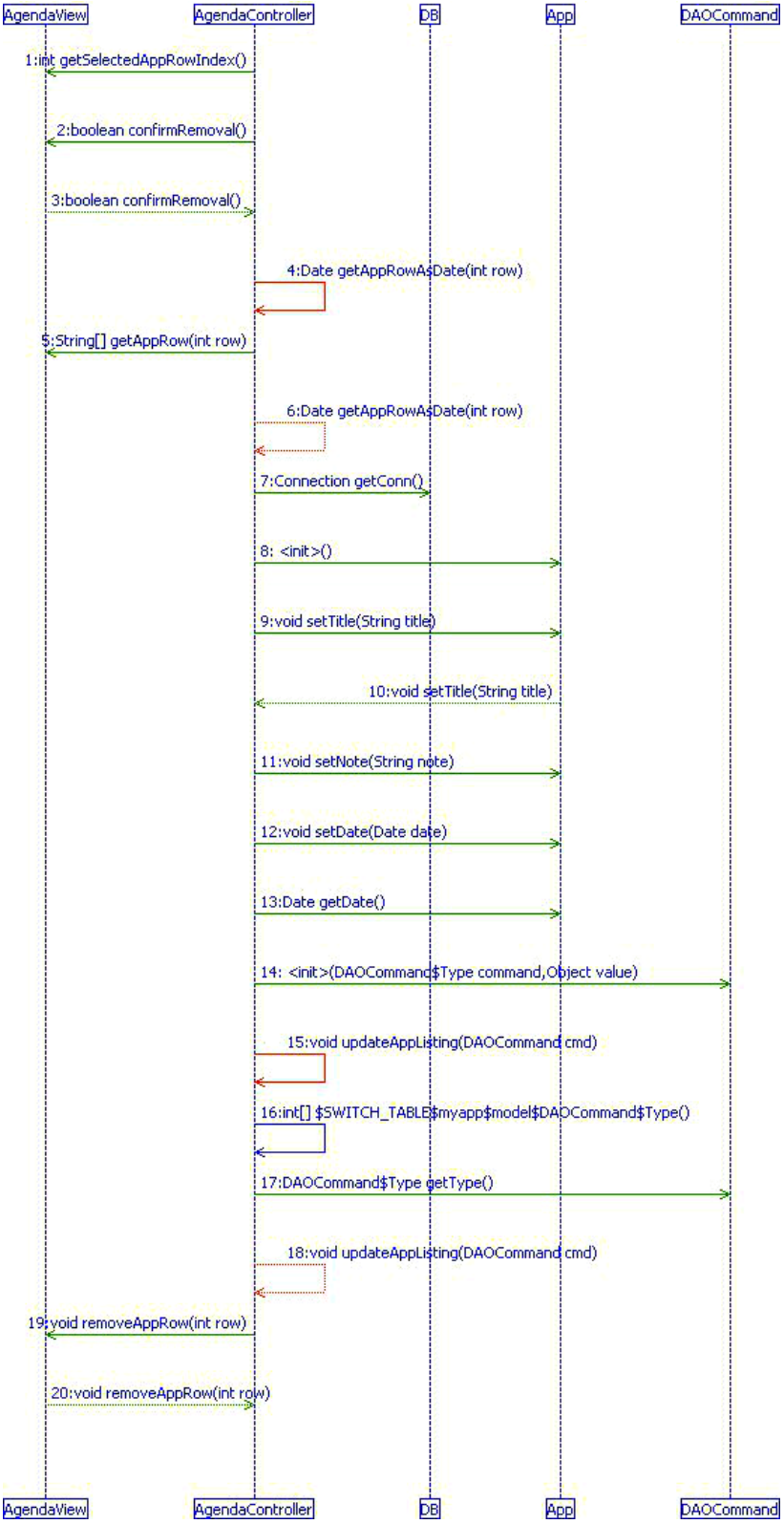

Figura 33: Diagrama de Sequência do sistema *myAppointments*
Fonte: Elaborado pelo autor

5.4 Linguagem de Alertas

Quando se faz o uso da API de reflexão de Java, técnicas de análise dinâmica conseguem detectar violações que não são detectadas por meio de análise estática. Com

análise estática não é possível obter a informação de qual objeto foi criado dinamicamente e, nem mesmo, quais atributos ou métodos de um objeto foram acessados por reflexão. Esse é um ponto em que a análise dinâmica se sobressai em relação à análise estática.

Um cenário comum é o uso de reflexão para registrar um *driver* de um SGBD (Sistema de Gerenciamento de Banco de Dados) em uma aplicação Java. Isso é realizado por meio da instanciação de uma classe específica do SGBD, denominada *driver*. Normalmente, o nome qualificado dessa classe é armazenado em um arquivo texto ou mesmo inserido diretamente no código fonte, conforme ilustrado na Linha 1 da Listagem 5.3. Nesse exemplo, o *driver* do HSQLDB (*HyperSQL DataBase*) é registrado na aplicação por meio de reflexão.

```
1 Class.forName("org.hsqldb.JdbcDriver");
2 ...
3 Connection conn = DB.getConnection(...);
```

Listagem 5.3: Pseudo-código que registra o *driver* do HSQLDB
Fonte: Elaborado pelo autor

Suponha que somente o HSQLBD seja suportado pelo sistema. Logo, pode-se utilizar a ferramenta *OG* – completamente baseada em análise dinâmica – para garantir que, se o *driver* de conexão ao SGBD for alterado, um alerta será exibido ao desenvolvedor avisando sobre essa mudança. Isso é de extrema importância, uma vez que a mudança do SGBD sem uma prévia inspeção de um desenvolvedor pode resultar em diversos problemas na aplicação. Por exemplo, comandos SQL que possuem códigos específicos do HSQLDB deixarão de funcionar.

Um alerta para essa violação pode ser definido conforme mostrado na Listagem 5.4, que alerta se a classe `DB` – responsável pela conexão ao SGBD – criar qualquer objeto que não seja o *driver* do HSQLDB ou que não pertence ao domínio da API de Java SQL. Em outras palavras, qualquer tentativa de acessar outro SGBD é exibida no grafo de visualização como um alerta arquitetural.

```
1 domain JavaSql: java.sql.**
2 domain DB:      myapp.model.DB
3 alert DB create !org.hsqldb.JdbcDriver,
4                 !JavaSql
```

Listagem 5.4: Definições de alertas caso haja mudança do SGBD
Fonte: Elaborado pelo autor

5.4.1 Alerta nas camadas MVC

Na subseção 5.2.1 foi discutido a arquitetura do sistema *JHotDraw* conforme a Figura 25. Como o sistema *JHotDraw* segue o padrão arquitetural MVC é comum o desenvolvedor observar se as camadas *View*, *Controller* e *Model* estão sendo respeitadas. A Figura 25 mostra a comunicação entre os domínios e existe uma comunicação dos domínios *View*, *View(Draw)* com o domínio *Model* (desrespeitando assim a regra estabelecida pelo padrão MVC). Para descobrir o motivo dessa comunicação indevida pode-se utilizar a *Linguagem de Alertas* conforme a regra mostrada na Listagem 5.5.

```
1 alert View, View(Draw) depend Model
```

Listagem 5.5: Alerta Arquitetural
Fonte: Elaborado pelo autor

Por meio do alerta definido nessa listagem, foi possível descobrir que a comunicação ocorreu no método *setDrawing* da classe *DefaultDrawingView* (modelado como *View*) com a classe *QuadTreeDrawing* (modelado como *Model*).

A Figura 34 apresenta o diagrama de classe do sistema *JHotDraw*, com as classes envolvidas nesta avaliação. A Figura 34 possui 5 classes e 2 *interfaces*.

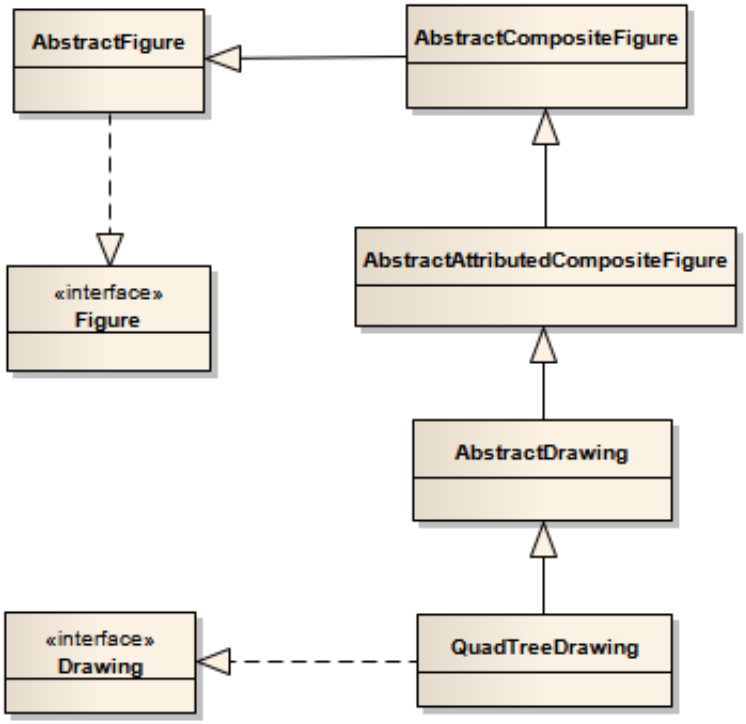

Figura 34: Diagrama de Classe *JHotDraw*
Fonte: Elaborado pelo autor

```
1  public class DefaultDrawingView extends JComponent implements
       DrawingView, EditableComponent {
2  ...
3      public void setDrawing(Drawing newValue) {
4          ...
5          this.drawing = newValue;
6          if (this.drawing != null) {
7              this.drawing.addCompositeFigureListener(eventHandler);
8              this.drawing.addFigureListener(eventHandler);
9          }
10         ...
11     }
12 ...
13 }
```

Listagem 5.6: Trechos de códigos da classe *DefaultDrawingView*
Fonte: Código Fonte do *JHotDraw*

Para compreender o motivo do alerta, a Listagem 5.6 mostra trechos de códigos do método *setDrawing* da classe *DefaultDrawingView*. No momento da execução desse trecho de código, o parâmetro *newValue* é uma instancia de *QuadTreeDrawing* (modelado como *Model*) que foi passado para a classe *DefaultDrawingView* (modelado como *View*). Na linha *5*, é atribuído para *drawing* a referência da variável *newValue*. Conforme as linhas *7* e *8* mostra, *drawing* invoca dois métodos *addCompositeFigureListener* e *addFigureListener*. Estes trechos de códigos (linhas *7* e *8*) significa que uma classe pertencente a camada *View* esta alterando o comportamento de uma classe pertencente a camada *Model*. Por esta razão, foi destacado como um alerta arquitetural para que o desenvolvedor possa analisar se esta correto ou não esta manipulação do ponto de vista do padrão arquitetural MVC.

Para localizar este tipo de alerta manualmente não seria trivial, percebe-se pela Figura 34 que existe diversas heranças e implementações de interfaces. Entretanto, com a utilização da linguagem de alertas foi possível em tempo de execução descobrir qual era o tipo do objeto e se poderia existir aquele relacionamento informando o ponto exato onde ocorreu o alerta.

6 CONCLUSÕES

Este capítulo apresenta uma visão geral do trabalho realizado nesta dissertação, incluindo um conjunto de considerações finais evidenciando os pontos principais da abordagem proposta e possíveis linhas de trabalhos futuros.

6.1 Visão Geral

Para o desenvolvedor compreender um sistema que não possui documentação é necessário auxílio de um especialista, análise do código fonte ou engenharia reversa. O desenvolvedor pode não ter tempo suficiente para aguardar um especialista (caso exista) ou fazer uma análise detalhada do código. Uma solução para este problema é a engenharia reversa. A engenharia reversa pode ser realizada utilizando duas técnicas: análise estática e análise dinâmica. A engenharia reversa utilizando análise estática apresenta uma visão parcial dos relacionamentos que são estabelecidos durante a execução de um sistema. Já a engenharia reversa utilizando análise dinâmica tem como problema principal a escalabilidade dos diagramas recuperados.

Nesta dissertação foi apresentada uma abordagem para compreensão de sistemas por meio da recuperação da arquitetura dinâmica de sistemas de software, isto é, a abordagem realiza engenharia reversa utilizando análise dinâmica por meio de instrumentação via aspectos. A abordagem proposta é particularmente interessante uma vez que: permite ao desenvolvedor uma flexibilidade ao controlar o nível de informação desejado, exibe com precisão o comportamento do sistema, possui uma linguagem de alertas arquiteturais para auxiliar na compreensão arquitetural e uma ferramenta que permite a visualização de grafos de objetos em modo *on-the-fly*.

Para demonstrar a aplicabilidade da solução proposta foram realizados avaliações em sistemas reais. As duas primeiras avaliações tiveram como objetivo avaliar a capacidade da abordagem representar a arquitetura de um sistema por meio de domínios. Como pode ser observado, os resultados (Subseções 5.2.1 e 5.2.3) foram satisfatórios. Além disso

a abordagem permite ao desenvolvedor alterar os domínios para que possa atender a outros objetivos, como no caso das funcionalidades do *JHotdraw* (Subseção 5.2.2). Foi realizada outra avaliação com objetivo de demonstrar como a abordagem auxilia na compreensão de funcionalidades de um sistema, no caso *myAppointments* (Subseção 5.3.1). Os resultados mostraram que a abordagem representa um comportamento mais preciso quando comparado com diagramas de classe e sequência. Além disso a abordagem permite ao desenvolvedor trabalhar com outras visões (como visão de pacotes e domínios) aumentando a chance do desenvolvedor compreender a funcionalidade do sistema. Finalmente, foram realizadas duas avaliações com o objetivo de demonstrar os benefícios dos alertas arquiteturais (Seção 5.4). Os resultados mostraram que relações definidas por meio de reflexão computacional e polimorfismo são casos não triviais de serem detectados com a utilização da análise estática. Com a utilização de alertas arquiteturais é possível detectar se existe um relacionamento que não era esperado e descobrir o motivo que produziu o alerta, auxiliando o desenvolvedor a compreender a arquitetura do sistema.

6.2 Contribuições

A abordagem proposta apresenta as seguintes contribuições principais:

a) Proposição de uma abordagem que permite visões arquiteturais em diferentes níveis de granularidade. Isto possibilita ao desenvolvedor controlar a granularidade da informação que deseja e fornece diferentes perspectivas arquiteturais, aumentando assim a chance de compreensão do sistema;

b) Definição e implementação de uma linguagem para sumarização de objetos em domínios. A linguagem permite ao desenvolvedor a sumarização de diversas maneiras como, por exemplo, em pacotes ou por meio de relacionamentos típicos de linguagem orientadas a objetos, como herança e implementação de interfaces;

c) Definição e implementação de uma linguagem para alertas arquiteturais. Essa linguagem possibilita ao desenvolvedor associar alertas a relacionamentos que são esperados (ou que não são esperados), auxiliando-o a compreender sistemas (como discutido na Seção 5.4);

d) Projeto e implementação da ferramenta *OGV* (*Object Graph Visualization*). O *OGV* consiste em uma ferramenta *on-the-fly* para visualização e manipulação de grafos de objetos.

Publicação: Hugo Alves; Henrique Rocha; Ricardo Terra; Marco Tulio Valente. Uma Abordagem para Recuperação da Arquitetura Dinâmica de Sistemas de Software. IV Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software (SBCARS). Salvador: UFBA, 2010. Classificado como um dos três melhores artigos do simpósio.

6.3 Trabalhos Futuros

Algumas possíveis linhas de trabalho futuro incluem:

a) Aprimorar a ferramenta *OGV*, incluindo as seguintes atividades:

- Integrar a ferramenta *OGV* ao ambiente Eclipse. Desse modo, será mais fácil conectar a ferramenta *OG* no sistema alvo;
- Permitir capturas em telas diferentes e possibilitar a união e/ou interseção de todas as visões capturadas. Por exemplo, suponha que tenham sido capturadas duas visões de funcionalidades diferentes. Nesse cenário, o desenvolvedor pode desejar unir os dois grafos e analisar o que é comum no comportamento das duas funcionalidades;
- Permitir ao desenvolvedor acrescentar anotações em vértices e arestas. Assim, quando gerado uma nova visão, caso o vértice ou aresta que possui alguma anotação for exibido será mostrado a anotação também;
- Permitir ao desenvolvedor maior interação com o grafo, por exemplo, excluir vértices e arestas ou aplicar uma sumarização manual. Ao realizar alguma dessas ações, a ordem dos vértices e arestas seria atualizada.

b) Criar novas formas de sumarização:

- Sumarização automática utilizando métricas de software, como por exemplo, grau de coesão e acoplamento entre classes;
- Criar uma extensão da linguagem de domínios, que permita a definição de domínios genéricos. Por exemplo, dado qualquer sistema, o domínio *View* incluiria as classes que acessam, herdam ou criam objetos do *framework Swing*.

c) Aprimorar o *OG*:

- Possibilitar a visualização de um subgrafo de um *OG*, isto é, o desenvolvedor poderá visualizar o grafo de um domínio;

- Permitir a realização de inferências arquiteturais, isto é, o *OG* analisar e sugerir melhorias arquiteturais para um sistema;
- Possibilitar ao desenvolvedor analisar a evolução do grafo, isto é, mostrar passo a passo a construção do grafo;
- Possibilitar a exportação dos dados de um *OG* para XMI.

d) Aprimorar os filtros implementados permitindo ao desenvolvedor filtrar, por exemplo, métodos de classes. Atualmente é possível filtrar apenas classes e pacotes;

e) Realizar novos estudos casos, incluindo um estudo de caso para verificar a aplicabilidade do *OG* em analisar o acoplamento entre as classes de um sistema. Outro estudo de caso pode ser realizado para verificar como informações sobre *threads* podem auxiliar no entendimento de uma funcionalidade do sistema.

REFERÊNCIAS

ABI-ANTOUN, Marwan; ALDRICH, Jonathan. A field study in static extraction of runtime architectures. In: **Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering**. New York, NY, USA: ACM, 2008. p. 22–28.

ABI-ANTOUN, Marwan; ALDRICH, Jonathan. Static extraction and conformance analysis of hierarchical runtime architectural structure using annotations. In: **Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications**. New York, NY, USA: ACM, 2009. p. 321–340.

ABI-ANTOUN, Marwan; ALDRICH, Jonathan. Static extraction of sound hierarchical runtime object graphs. In: **TLDI 09: Proceedings of the 4th international workshop on Types in language design and implementation**. New York, NY, USA: ACM, 2009. p. 51–64.

ALDRICH, Jonathan; CHAMBERS, Craig; NOTKIN, David. Archjava: connecting software architecture to implementation. In: **Proceedings of the 24th International Conference on Software Engineering**. New York, NY, USA: ACM, 2002. p. 187–197.

ANQUETIL, Nicolas; FOURRIER, Cédric; LETHBRIDGE, Timothy C. Experiments with clustering as a software remodularization method. In: **Proceedings of the Sixth Working Conference on Reverse Engineering**. Washington, DC, USA: IEEE Computer Society, 1999. p. 235.

ANQUETIL, Nicolas; LETHBRIDGE, Timothy C. Ten years later, experiments with clustering as a software remodularization method. In: **Proceedings of the 2009 16th Working Conference on Reverse Engineering**. Washington, DC, USA: IEEE Computer Society, 2009. p. 7.

BASS, Len; CLEMENTS, Paul; KAZMAN, Rick. **Software Architecture in Practice**. 2. ed. Boston: Addison-Wesley Professional, 2003.

BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar. **Unified Modeling Language User Guide**. 2. ed. Boston: Addison-Wesley Professional, 2005.

BRIAND, Lionel C.; LABICHE, Yvan; LEDUC, Johanne. Toward the reverse engineering of UML sequence diagrams for distributed Java software. **IEEE Transacions on Software Engineering**, v. 32, n. 9, p. 642–663, 2006.

CLEMENTS, Paul; SHAW, Mary. The golden age of software architecture revisited. **IEEE Software**, v. 26, n. 4, p. 70–72, 2009.

DUCASSE, Stéphane; POLLET, Damien. Software architecture reconstruction: A process-oriented taxonomy. **IEEE Transactions on Software Engineering**, v. 35, n. 4, p. 573–591, 2009.

FOWLER, Martin. **Patterns of Enterprise Application Architecture**. Boston: Addison-Wesley Professional, 2002.

FOWLER, Martin. **UML Distilled: A Brief Guide to the Standard Object Modeling Language**. 3. ed. Boston: Addison-Wesley Professional, 2003.

HOFMEISTER, Christine; NORD, Robert L.; SONI, Dilip. Describing software architecture with uml. In: **Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1)**. Deventer, The Netherlands: Kluwer, 1999. p. 145–160.

JACKSON, Daniel; WAINGOLD, Allison. Lightweight extraction of object models from bytecode. **IEEE Transactions on Software Engineering**, v. 27, n. 2, p. 156–169, 2001.

JUNG. **Java Universal Network/Graph Framework**. Disponível em: <http://jung.sourceforge.net>. Acesso em: 01 de mar. de 2011.

KNODEL, Jens *et al.* Static evaluation of software architectures. In: **Proceedings of the Conference on Software Maintenance and Reengineering**. Washington, DC, USA: IEEE Computer Society, 2006. p. 279–294.

KNODEL, Jens; POPESCU, Daniel. A comparison of static architecture compliance checking approaches. In: **Proceedings of the Sixth Working IEEE/IFIP Conference on Software Architecture**. Washington, DC, USA: IEEE Computer Society, 2007. p. 12.

KRUCHTEN, Philippe. The 4+1 view model of architecture. **IEEE Software**, v. 12, n. 6, p. 42–50, 1995.

LENCEVICIUS, Raimondas; RAN, Alexander; YAIRI, Rahav. Third eye. specification-based analysis of software execution traces (poster session). In: **Proceedings of the 22nd international conference on Software engineering**. New York, NY, USA: ACM, 2000. p. 772.

MEDVIDOVIC, Nenad; TAYLOR, Richard N. A classification and comparison framework for software architecture description languages. **IEEE Transactions on Software Engineering**, v. 26, n. 1, p. 70–93, 2000.

MURPHY, Gail; NOTKIN, David; SULLIVAN, Kevin. Software reflexion models. **IEEE Transactions on Software Engineering**, v. 27, n. 4, p. 364–380, 2001.

MURPHY, Gail C.; NOTKIN, David; SULLIVAN, Kevin. Software reflexion models: bridging the gap between source and high-level models. In: **Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering**. New York, NY, USA: ACM, 1995. p. 18–28.

ORG, Eclipse. **AspectJ**. Disponível em: <http://eclipse.org/aspectj/>. Acesso em: 01 de mar. de 2011.

PASSOS, Leonardo *et al.* Static architecture conformance checking – an illustrative overview. **IEEE Software**, v. 27, n. 5, p. 82–89, 2010.

PERRY, Dewayne E.; WOLF, Alexander L. Foundations for the study of software architecture. **Software Engineering Notes**, v. 17, n. 4, p. 40–52, 1992.

RIVA, Claudio; RODRIGUEZ, Jordi Vidal. Combining static and dynamic views for architecture reconstruction. In: **Proceedings of the 6th European Conference on Software Maintenance and Reengineering**. Washington, DC, USA: IEEE Computer Society, 2002. p. 47.

SAFYALLAH, Hossein; SARTIPI, Kamran. Dynamic analysis of software systems using execution pattern mining. In: **Proceedings of the 14th IEEE International Conference on Program Comprehension**. Washington, DC, USA: IEEE Computer Society, 2006. p. 84–88.

SANGAL, Neeraj *et al.* Using dependency models to manage complex software architecture. In: **Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications**. New York, NY, USA: ACM, 2005. p. 167–176.

SARTIPI, Kamran; KONTOGIANNIS, Kostas; MAVADDAT, Farhad. Architectural design recovery using data mining techniques. In: **Proceedings of the Conference on Software Maintenance and Reengineering**. Washington, DC, USA: IEEE Computer Society, 2000. p. 129.

SCHMERL, Bradley R. *et al.* Discovering architectures from running systems. **IEEE Transactions on Software Engineering**, v. 32, n. 7, p. 454–466, 2006.

SHAW, Mary; GARLAN, David. **Software architecture: perspectives on an emerging discipline**. 1. ed. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.

TECHNOLOGIES ej. **JProfiler**. Disponível em: `<http://www.ej-technologies.com/>`. Acesso em: 01 de mar. de 2011.

TERRA, Ricardo; VALENTE, Marco Tulio. A dependency constraint language to manage object-oriented software architectures. **Software: Practice and Experience**, v. 32, n. 12, p. 1073–1094, 2009.

TERRA, Ricardo; VALENTE, Marco Tulio Oliveira. Towards a dependency constraint language to manage software architectures. In: **Proceedings of the 2nd European conference on Software Architecture**. Berlin, Heidelberg: Springer-Verlag, 2008. p. 256–263.

TONELLA, Paolo. Reverse engineering of object oriented code. In: **Proceedings of the 27th international conference on Software engineering**. New York, NY, USA: ACM, 2005. p. 724–725.

YAN, Hong *et al.* Discotect: A system for discovering architectures from running systems. In: **Proceedings of the 26th International Conference on Software Engineering**. Washington, DC, USA: IEEE Computer Society, 2004. p. 470–479.