

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



**RECUPERAÇÃO DE VISÃO ESTRUTURAL DE SOFTWARE
COMBINANDO ANÁLISE DE SEMÂNTICA ESTÁTICA E
LATENTE**

FELIPE CÉZAR DE CASTRO ANTUNES

Uberlândia - Minas Gerais

2009

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



FELIPE CÉZAR DE CASTRO ANTUNES

**RECUPERAÇÃO DE VISÃO ESTRUTURAL DE SOFTWARE
COMBINANDO ANÁLISE DE SEMÂNTICA ESTÁTICA E
LATENTE**

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

Área de concentração: Engenharia de Software.

Orientador:

Prof. Dr. Marcelo de Almeida Maia

Co-orientador:

Prof. Dr. Ilmério Reis da Silva

Uberlândia, Minas Gerais

2009

Dados Internacionais de Catalogação na Publicação (CIP)

- A636r Antunes, Felipe César de Castro, 1983-
 Recuperação de visão estrutural de software combinando análise de
 semântica estática e latente [manuscrito] / Felipe César de Castro Antunes.
 - 2009.
 71 f. : il.
- Orientador: Marcelo de Almeida Maia.
 Co-orientador: Ilmério Reis da Silva.
- Dissertação (mestrado) – Universidade Federal de Uberlândia, Progra-
 ma de Pós-Graduação em Ciência da Computação.
 Inclui bibliografia.
1. Engenharia de software - Teses. I. Maia, Marcelo de
 Almeida. II. Silva, Ilmério Reis da, 1961- III. Universidade Federal de
 Uberlândia. Pro-grama de Pós-Graduação em Ciência da Computação. III.
 Título.

CDU: 681.3.06

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

Os abaixo assinados, por meio deste, certificam que leram e recomendam para a Faculdade de Ciência da Computação a aceitação da dissertação intitulada “**Recuperação de Visão Estrutural de Software Combinando Análise de Semântica Estática e Latente**” por **Felipe César de Castro Antunes** como parte dos requisitos exigidos para a obtenção do título de **Mestre em Ciência da Computação**.

Uberlândia, 31 de Agosto de 2009

Orientador:

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

Co-orientador:

Prof. Dr. Ilmério Reis da Silva
Universidade Federal de Uberlândia

Banca Examinadora:

Prof. Dr. Stéphane Julia
Universidade Federal de Uberlândia

Prof. Dr. Mark Alan Junho Song
Pontifícia Universidade Católica de Minas Gerais

Dedicatória

Aos meus pais, obrigado!

Agradecimentos

Agradeço...

Ao Grande Arquiteto do Universo, por minha vida.

A meus pais José Gilberto Antune e Fátima Inês de Castro Antunes pela dedicação, confiança, apoio, carinho e amor incondicional em todos os momentos.

Aos meus irmãos Jose Henrique, João Paulo e Pedro Ivo, pelas palavras de carinho, amor e amizade infindos. Eles são essenciais para que eu siga adiante!

A meus amigos da UFU que se mostraram companheiros e, diretamente ou indiretamente, contribuíram para a realização deste trabalho, André Alves de Resende, Crícia Zilda Felício, Éverton Hipólito de Freitas, Gutierrez Soares Caixeta, Italo Tiago da Cunha, Jean Carlo de Souza Santos, Klérissom Vinícius Ribeiro da Paixão, Marcos Roberto Ribeiro Mirella Silva Junqueira, Núbia Rosa da Silva, Rodrigo Reis Pereira, Robson Silva Lopes, Stéfano Schwenck Borges Vale Vita, Tauller Augusto de Araújo Matos, Valquíria Aparecida Rosa Duarte, Vinícius Borges Pires e Willians Ribeiro Mendes pelos conselhos e apoio.

Aos colegas e amigos do Instituto Federal de Educação, Ciência e Tecnologia do Norte de Minas Gerais, Alisson Magalhães Castro, Ana Alves Neta, Anderson de Oliveira Vallejo, Cláudio Alexandre Gusmão, Cleiane Gonçalves Oliveira, Danilo Magalhães Nunes, Edinei Canuto Paiva, Egídio Rodrigues Martins, Felipe Lisboa Guedes, Giuliano Viana de Alkmim, Joabe Jobson de Oliveira Pimentel, José Ricardo Martins da Silva, Júlio César Pereira Braga, Maria Aparecida Colares Mendes, Rodrigo Martins, Ronaldo Maurício Sampaio, Valkennedy de Moura Castro pelo apoio e compreensão durante minha trajetória.

Ao professor, e agora meu amigo, Marcelo de Almeida Maia, pelos ensinamentos transmitidos, paciência em vários momentos difíceis, palavras de incentivo e de crítica, quando necessário. A convivência com a Marcelo, a sua amizade e a sua sabedoria certamente me fazem hoje uma pessoa melhor.

"A morte do homem começa no instante em que ele desiste de aprender."
(Albino Teixeira)

Resumo

A compreensão de software ainda é um desafio para a manutenção de sistemas desconhecidos. Isto ocorre porque grande quantidade das informações não está disponível de forma explícita. Este trabalho propõe apoiar a compreensão de software através do uso de uma ferramenta visual e interativa, criando uma visão estrutural do sistema alvo. Para tal se faz uso da técnica de recuperação de informação *Latent Semantic Indexing* - LSI combinada com a análise semântica referencia explícita do código fonte. Neste trabalho o conhecimento informal oculto contido no vocabulário dos artefatos é combinado com a estrutura estática do código fonte na construção de uma visão estrutural de um software desconhecido. O método proposto e a ferramenta são validados com a análise da ferramenta *Prevayler* e do compilador do *OpenJDK*. Especificamente, a abordagem proposta mostrou que uma visão estrutural de sistema desconhecido pode ser encontrada de forma simples, provendo uma ajuda valiosa para compreensão de tais sistemas.

Palavras chave: visão estrutural, análise semântica estática, análise semântica latente, compreensão de programas

Abstract

Software Comprehension is still a challenge for the maintenance of unknown systems. This situation occurs because most of the information is not explicitly available. This work aims to support Software Comprehension through the use of a visual interactive tool, creating a structural vision of a target system. For this purpose, the technique of information retrieval *Latent Semantic Indexing* - LSI combined with the static structure of the source code is used. In this work, the informal hidden knowledge contained in the vocabulary of artifacts is combined with the static structure of source code for the construction of a structural vision of the unknown system. The method and its respective tool are validated with the analysis of the Prevayler system and the OpenJDK compiler. Specifically, the proposed method has shown that a structural vision of unknown system can be found through a simple way, providing a valuable aid to understanding such systems.

Keywords: structural vision, static semantic analysis, latent semantic analysis

Sumário

Lista de Figuras	xi
Lista de Tabelas	xii
Lista de Abreviaturas e Siglas	xiii
1 Introdução	14
1.1 Exposição do Problema	16
1.2 Hipóteses e Proposta	16
1.3 Objetivos	17
1.4 Justificativas	17
1.5 Visão Geral da Dissertação	18
2 Referencial Teórico	19
2.1 Compreensão de Software	19
2.2 Análise Referencial Explícita e Análise Dinâmica	20
2.3 Recuperação de Informação	21
2.4 Indexação Semântica Latente - LSI	21
2.4.1 Decomposição do Valor Singular	22
2.4.2 Execução do LSI	22
2.4.3 Exemplo de Aplicação do Método LSI	25
2.5 Conclusão	29
3 Abordagem Proposta	30
3.1 Objetivos da Abordagem	30
3.2 Descrição da Abordagem	31
3.3 Extração da Semântica Referencial Explícita	32
3.4 Extração da Semântica Latente	33
3.5 Semântica Referencial Explícita versus Semântica Latente	35
3.6 Unificando as Abordagens	36
3.7 Implementação da Abordagem	39
3.8 Conclusão	41

4	Avaliação da Abordagem Proposta	42
4.1	Metodologia de avaliação	42
4.1.1	<i>Goal Question Metric</i> (GQM)	43
4.2	Definição do Estudo	45
4.3	Execução do Estudo Experimental	46
4.3.1	<i>Prevayler</i>	47
4.3.2	Compilador do <i>OpenJDK</i>	50
4.4	Análise dos Resultados Obtidos	56
4.4.1	Precisão dos Agrupamentos Recuperados	56
4.4.2	Precisão das Relações Recuperadas entre Agrupamentos	57
4.4.3	Apoio da Visão Recuperada na Compreensão	58
4.4.4	Análise do Desempenho das Ferramentas	59
4.5	Conclusão	60
5	Trabalhos Relacionados	61
6	Conclusões e Perspectivas	64
6.1	Contribuições	65
6.2	Limitações Identificadas	65
6.3	Trabalhos Futuros	66
	Referências Bibliográficas	67

Lista de Figuras

2.1	Decomposição do valor singular	24
2.2	Visualização dos resultados da consulta no plano cartesiano	29
3.1	Visão geral da abordagem proposta	31
3.2	Semântica referencial explícita versus semântica latente	35
3.3	Agrupamento de elementos	37
3.4	Agrupamento visual dos elementos	39
3.5	Diagrama de componentes da ferramenta	40
4.1	Estratégia usada na verificação da viabilidade da abordagem proposta. . .	43
4.2	Estrutura hierárquica do GQM.	43
4.3	Exemplo modelo GQM [DdSK03].	45
4.4	Grafo do <i>Prevayler</i> com fator de calibragem 0.75	48
4.5	Grafo do <i>Prevayler</i> com fator de calibragem 0.6	49
4.6	Grafo do <i>Prevayler</i> com fator de calibragem 0.5	50
4.7	Grafo do <i>Prevayler</i> com fator de calibragem 0.4	51
4.8	Grafo do <i>Prevayler</i> com fator de calibragem 0.3	52
4.9	Grafo do <i>Prevayler</i> com fator de calibragem 0.2	53
4.10	Agrupamentos identificados no <i>Prevayler</i> com fator de calibragem 0.4 . . .	53
4.11	Definição do fator de calibragem para o compilador do <i>OpenJDK</i>	54
4.12	Grafo do compilador do <i>OpenJDK</i> com fator de calibragem 0.54	55
4.13	Arquitetura em camadas do compilador do <i>OpenJDK</i> [Mic09a]	57
4.14	Relações de semântica referencial explícita do compilador OpenJDK	59

Lista de Tabelas

2.1	Ocorrências termo-documento	25
2.2	Ranking de documentos	28
4.1	Sumário do modelo GQM	44
4.2	Métricas extraídas do código fonte do <i>Prevayler</i> e do compilador do <i>Open-JDK</i> com o <i>plugin</i> do <i>Eclipse Metrics</i>	47
4.3	Análise do desempenho da ferramenta	60

Lista de Abreviaturas e Siglas

GQM	<i>Goal Question Metric</i>
LSI	<i>Latent Semantic Indexing</i>
JDK	<i>Java Development Kit</i>
JVM	<i>Java Virtual Machine</i>
RAM	<i>Random Access Memory</i>
RI	Recuperação de Informação
SVD	<i>Singular Value Decomposition</i>
UML	<i>Unified Modeling Language</i>

Capítulo 1

Introdução

O processo de obtenção de conhecimento sobre um sistema de software representa uma das principais e mais onerosas atividades da engenharia de software. Estima-se que até 60% do custo total de um sistema são referentes a esta atividade [GT03].

A compreensão de software pode ser caracterizada como um processo através do qual um desenvolvedor constrói um modelo mental do programa. O conhecimento é obtido através do reconhecimento de estruturas abstratas, estrutura de controle do programa, funcionalidade e arquitetura global. Por meio da compreensão, o desenvolvedor pode decidir como o programa deve ser modificado para satisfazer os novos requisitos, diminuindo assim o esforço gasto no processo.

Importa ressaltar que a manutenção de software faz parte do ciclo de vida do software, e já começa a ser aplicada logo após o software entrar em produção. Esta fase é responsável pelo processo de melhoria e otimização de um software já desenvolvido, bem como o reparo de defeitos deste. Ou seja, a manutenção de software envolve tarefas de mudanças no software para corrigir defeitos e deficiências que foram encontrados durante a utilização pelo usuário, ou mesmo a adição de novas funcionalidades para melhorar a aplicabilidade e usabilidade do software.

A importância da tarefa de compreensão de software fica evidente por ser uma parte essencial do processo de manutenção de software. O desafio de se compreender um software desconhecido se torna ainda mais crítico devido às grandes organizações apresentarem, em geral, uma significativa base de sistemas legados.

Outro fator agravante é que atualmente, os sistemas de software estão cada vez mais complexos e, ao mesmo tempo, existe uma demanda crescente por software de qualidade e uma necessidade que sejam entregues em menor tempo. Por consequência, se faz necessário uma maior efetividade no uso da compreensão de software durante o desenvolvimento.

O alto esforço investido no desenvolvimento e o grande conhecimento sobre o negócio contido nos sistemas legados, e que muitas vezes são indisponíveis explicitamente nas organizações, aumentam a preocupação constante dos engenheiros de software em com-

preender estes sistemas e a sua organização estrutural.

Diante deste contexto, as atividades que envolvem a inspeção do código fonte do sistema, como por exemplo: localizar as partes do programa que implementam uma funcionalidade [ZZL⁺04]; prever impactos de uma mudança [RT01]; e recuperar aspectos não documentados da arquitetura do sistema [PDP⁺07], se tornam cruciais no cotidiano dos engenheiros do software.

A arquitetura de software ganha espaço nesse cenário de desenvolvimento, visto que enfatiza a organização global do sistema, definindo a sua topologia e permitindo que desenvolvedores se concentrem nos requisitos funcionais e não-funcionais que precisam ser atendidos, antes de se aterem ao projeto das estruturas de dados e algoritmos. Deste modo, projetar a estrutura global de um sistema emerge como um novo desafio: o desenvolvimento de software orientado para arquitetura.

Os conceitos de arquitetura de software foram formalizados [BMR⁺96], porém, os sistemas de software já aplicam determinados padrões de organização estrutural ao longo de décadas. Estilos arquiteturais, como o da divisão hierárquica de sistemas em programa principal e sub-rotinas, vêm sendo empregados há muitos anos pelos desenvolvedores de software, representando uma das organizações estruturais possíveis de ser encontrada em sistemas legados.

Entretanto, os sistemas legados raramente dispõe de uma documentação de apoio que facilite a sua compreensão e esta, quando existe, se encontra em geral defasada em relação a atual implementação do software. Neste contexto, o código fonte torna-se a principal fonte de informação disponível para a compreensão dos sistemas. Devido ao fato do código se apresentar em um nível de abstração baixo, por ser direcionado a ferramentas do gênero dos compiladores e das dimensões de um software serem, geralmente, grandes (considerando a quantidade de linhas de código), o entendimento destes sistemas acaba se tornando uma tarefa não trivial.

Sendo assim, torna-se necessário, que os aspectos da arquitetura de um sistema sejam extraídos a partir de artefatos criados durante seu desenvolvimento. Tal processo é denominado recuperação de arquitetura.

Diversos trabalhos propõem o uso de algoritmos de agrupamento (clustering) a fim de recuperar automaticamente uma decomposição em módulos das entidades do código-fonte (classes, métodos e atributos) [MMR⁺98, And05, MB07]. Tal estratégia se mostra promissora, porém uma grande quantidade das informações sobre os sistemas não estão disponíveis de forma explícita, e isso implica na diminuição da precisão e qualidade dos resultados.

Como visto, compreender um software desconhecido ainda representa um desafio para engenharia de software. Sendo assim, a próxima seção irá apresentar alguns dos problemas enfrentados pela tarefa de compreensão de software.

1.1 Exposição do Problema

O tema principal da pesquisa está na recuperação de uma visão estrutural de um software desconhecido, que possa ser utilizada como ponto de partida para a tarefa de compreensão de software. Entretanto tal tarefa enfrenta diversos desafios:

- Existe uma enorme diversidade de linguagens, padrões e metodologias utilizadas no desenvolvimento de software, fator que dificulta a portabilidade e precisão dos processos de compreensão de software.
- Os softwares legados encontrados atualmente possuem uma grande dimensão (quantidade de linhas de código e número de artefatos), deste modo as abordagens de compreensão de software devem ser capazes de processar enormes volumes de informações.
- Atualmente, por consequência da dimensão dos softwares, é exigido um alto esforço manual para se compreender um software desconhecido, quando este é feito sem o auxílio de um ferramental adequado.
- Há a falta de uma padronização na representação do conhecimento recuperado sobre um software. Existem diversas metodologias para descrição do conhecimento sobre um software, porém não há uma linguagem de referência. Deste modo cada abordagem de compreensão cria ou define como o conhecimento sobre o software será representado;
- As abordagens de compreensão de software, em geral, não produzem resultados intermediários do conhecimento recuperado. Temos então todo o conhecimento representado em um único resultado, porém devido as dimensões dos sistemas atuais, este resultado pode ser de difícil interpretação.
- Os processos automáticos para compreensão de software, normalmente não utilizam o conhecimento prévio já adquirido sobre o software. Apenas são analisadas informações explícitas encontradas no artefatos que estão disponíveis.

1.2 Hipóteses e Proposta

A compreensão de software é uma tarefa de natureza complexa. Isto ocorre porque grande quantidade das informações não está disponível de forma explícita. Diversas abordagens de compreensão de software buscam obter as informações do sistema através de análises estruturais ou de documentação externa. No entanto, tais estratégias subestimam o vocabulário utilizado nos diversos artefatos produzidos durante todo o ciclo de vida do software. Esta informação é relevante e fundamental, justamente por ser inserida pela própria equipe de desenvolvimento.

A transmissão de informação é o princípio básico de toda linguagem. Tal fato também pode ser observado nas linguagens de programação. O código fonte representa a comunicação entre programador e máquina. Existe também comunicação informal entre os desenvolvedores, a qual pode ser feita por meio dos identificadores e dos comentários inseridos no código.

A nossa hipótese é que o vocabulário contido nos artefatos pode ser utilizado na recuperação de uma visão estrutural de um software desconhecido. Essa visão da estrutura do sistema é importante por indicar relações consistentes entre os artefatos de código fonte.

Logo, a união da técnica de recuperação de informação de Indexação Semântica Latente - LSI com a análise semântica referencia explícita do código fonte poderia produzir uma visão estrutural do software melhor do que se aplicadas isoladamente e pode ser utilizada como ponto de partida para se compreender um software desconhecido.

A principal contribuição desta pesquisa é a definição de uma abordagem capaz de agilizar e apoiar a tarefa de compreensão de software, através do uso de uma ferramenta visual e interativa que facilite a interpretação dos resultados. Tal abordagem irá combinar o conhecimento informal oculto contido no vocabulário dos artefatos com a estrutura estática do código fonte e poderá ser aplicada a software de grandes dimensões independente da linguagem ou técnica utilizada no seu desenvolvimento.

1.3 Objetivos

O objetivo deste trabalho é investigar a efetividade do uso da técnica de recuperação de informação LSI combinada com a análise semântica referencia explícita do código fonte na recuperação de uma visão estrutural sobre um sistema alvo, podendo esta ser utilizada como ponto de partida na compreensão de um software desconhecido.

Como objetivos específicos, pode-se salientar:

- Comparar as relações encontradas na análise semântica referencia explícita do código fonte com as relações de semântica latente geradas pela técnica de LSI.
- Propor um modelo que unifique as relações de semântica referencial explícita e relações de semântica latente.
- Analisar os resultados da aplicação do modelo proposto como ponto de partida para compreensão de um software desconhecido.

1.4 Justificativas

As mudanças são inerentes ao ciclo de vida do desenvolvimento de software. Tais adaptações podem ser referentes à correção de erros, melhoria de desempenho, adição de novas funcionalidades, alteração do modelo ou adaptação a novas tecnologias.

Este contexto, aliando ao aumento significativo do tamanho dos sistemas atuais, faz com que a compreensão um sistema represente custos que variam de 40% a 60% do custo total de um sistema [GT03].

Os sistemas mantidos há algum tempo pelas empresas, em geral, apresentam sua documentação incompleta, desatualizada ou mesmo inexistente. Sendo assim, o código fonte surge como a principal fonte de informação disponível para a manutenção de tais sistemas.

Neste contexto, a compreensão de software a partir dos artefatos de código fonte representa uma atividade relevante. Uma preocupação é definir os componentes do sistema e suas interrelações.

A compreensão de software se mostra como um processo bastante oneroso e, portanto, ferramentas que automatizem esta tarefa são cada vez mais valiosas.

Por fim, o aumento significativo do volume de sistemas desenvolvidos segundo o paradigma de orientação a objetos justifica a necessidade do estudo de abordagens para compreensão de sistemas desenvolvidos sobre o mesmo.

1.5 Visão Geral da Dissertação

Este capítulo apresentou a motivação, proposta e hipóteses de nossa abordagem. O restante da dissertação está organizada como se segue.

No Capítulo 2 são apresentados os conceitos básicos de compreensão de software e recuperação de informação que compõem a fundamentação teórica da proposta. O Capítulo 3 define de forma detalhada a arquitetura da abordagem proposta nesta dissertação.

O Capítulo 4 descreve dois estudos de caso para mostrar a viabilidade da abordagem em sistemas reais. No Capítulo 5 serão apresentados alguns trabalhos relacionados a tarefa de compreensão de software e ao uso da recuperação de informação no contexto da engenharia de software.

Finalmente, no Capítulo 6 são discutidos os resultados encontrados e apresentadas as conclusões e perspectivas para trabalhos futuros.

Capítulo 2

Referencial Teórico

Neste capítulo será apresentada uma revisão na literatura sobre os principais temas envolvidos na pesquisa, dentre eles, a Compreensão de Software (Seção 2.1), a Recuperação de Informação (Seção 2.3) e a técnica de Indexação Semântica Latente (Seção 2.4).

2.1 Compreensão de Software

O termo compreensão expressa a capacidade de explicar um dado conceito, definindo seu significado e/ou funcionamento. Já a compreensão de software está relacionada ao entendimento da funcionalidade, estrutura e comportamento de um sistema de software [vMV95]. Ou seja, compreender um software significa ser capaz de explicar seu funcionamento, sua estrutura, seu comportamento, seu relacionamento com o domínio de aplicação, sendo portanto fundamental para a atividade de manutenção de software.

Importa ressaltar que a compreensão de software não se limita à uma fase específica do ciclo desenvolvimento, mas abrange toda a vida útil do software, destacando-se no período de pós entrega do sistema. Temos como exemplos de atividades que dependem da compreensão de software: evolução, reestruturação ou refatoração, engenharia reversa, reuso, documentação, migração entre outras tantas.

Entretanto, a compreensão de software enfrenta desafios que abrangem aspectos como a complexidade do domínio de aplicação, as limitações cognitivas dos programadores, as práticas de desenvolvimento e características da linguagem de programação.

Uma forma de classificar as técnicas de compreensão de software é segundo o fluxo de compreensão [DT98]:

- **Bottom-up:** Nesta técnica, à medida que vai se compreendendo pequenos trechos de código estes são agrupados a fim de produzir uma visão de mais alto nível. Comumente usada quando o código é totalmente novo para o programador.
- **Top-Down:** É usada quando se trabalha com um código familiar. Inicialmente se reconstrói o conhecimento sobre o domínio da aplicação e depois o código fonte é

mapeado em relação a esse conhecimento. Na prática é definida uma hipótese global e à medida que se conhece a aplicação a hipótese é verificada, até que se chegue no nível onde se pode comprovar ou não a hipótese.

É fato que os atuais sistemas podem conter milhares de linhas de código, tornando-os difíceis e complexos de entender. Além disso, o software é intangível, não tendo portanto uma estrutura física, deste modo após escrito seu conhecimento, este desaparece em arquivos mantidos em discos.

Em geral, existe também a limitação humana em assimilar uma grande quantidade de informação simultaneamente. Tal limitação é um dos principais desafios enfrentados pelos desenvolvedores na tarefa de compreender e manipular sistemas complexos, pois inibe a capacidade de processar simultaneamente as diferentes partes do sistema que interagem entre si [Cam97].

Neste contexto, abordagens que apóiam a compreensão de software são de grande importância, pois ajudam a reduzir a complexidade inerente ao processo. Tais abordagens auxiliam o desenvolvedor na construção de um modelo mental do programa, utilizando para isso mecanismos de análise, exploração e visualização da informação em diferentes níveis de abstração.

2.2 Análise Referencial Explícita e Análise Dinâmica

A compreensão de software pode ser feita por meio da análise referencial explícita e/ou dinâmicas. A análise referencial explícita é aquela que examina apenas o código fonte do sistema, sendo adequada para entender relacionamentos entre classes e identificar padrões arquiteturais. Já a técnica dinâmica extrai informação a partir da execução do sistema. Essa técnica é adequada principalmente para ilustrar interações entre objetos e fluxos de controles entre componentes [PRW04].

Sendo assim, a análise referencial explícita busca extrair informações básicas do software, como dependência de acesso a variáveis e grafos de chamadas de procedimentos. Contudo, a análise dinâmica se concentra nas propriedades de um programa em execução, informando, por exemplo, as seqüências de chamadas de funções, os valores de variáveis, etc.

Tanto a análise referencial explícita, quanto a dinâmica possuem potencialidades e limitações. Se o objetivo é encontrar a dependência de chamadas de funções/métodos a análise referencial explícita é suficiente. Porém, para saber quais objetos de código são instanciados, é fundamental analisar o programa em execução, necessitando assim de uma análise dinâmica.

A análise dinâmica é menos geral do que a análise referencial explícita, no sentido de que ela fornece informações sobre a chamada do programa com um conjunto de dados de

entrada particular. Um mesmo programa com outros diferentes dados de entrada pode fornecer outras informações.

Neste trabalho, somente a análise referencial explícita foi explorada por permitir a identificação de relações evidentes entre os artefatos de código fonte de um software desconhecido. Tais relações são importantes na descoberta de agrupamentos entre as entidades do sistema alvo.

2.3 Recuperação de Informação

Recuperação de Informação (do inglês, *Information Retrieval*) é a área da Ciência da Computação que estuda a representação, armazenamento, organização e acesso a informação a partir de coleções de documentos. O problema de recuperar informações relevantes e oportunas existe desde antes da invenção dos computadores. Este problema é de interesse desde a década de 1940 [BYRN99].

A utilização de computadores praticamente resolveu o problema de pesquisar grandes volumes de documentos (pelo menos em formato texto), porém os algoritmos dos computadores ainda não conseguem interpretar eficientemente as informações contidas nestes documentos.

Um sistema de Recuperação de Informação não tem a responsabilidade de responder precisamente a uma consulta e sim apresentar documentos e oferecer condições para que o usuário satisfaça sua necessidade de informação.

Existem diferentes metodologias para se avaliar os resultados de um sistema de Recuperação de Informação. Uma das mais usadas é através das medidas de precisão e revocação [BYRN99]. A precisão (*precision*) é a porcentagem de itens recuperados que são relevantes. A revocação (*recall*) é a porcentagem de itens relevantes que foram recuperados.

Para aumentar a precisão de uma busca, o sistema de recuperação de informações deve ser capaz de descartar itens recuperados que são irrelevantes. Para isso, deve saber contextualizar os dados recuperados, de forma a identificar falsos relacionamentos que levaram à inclusão inicial de um documento no conjunto de itens recuperados.

Já para melhorar a revocação, o algoritmo de busca utilizado deve perceber a relevância de documentos de forma não explícita. Em ambos os casos, o conhecimento do domínio da aplicação é útil.

2.4 Indexação Semântica Latente - LSI

O método de recuperação de informação *Latent Semantic Indexing* - LSI, tem como princípio a existência de uma estrutura semântica oculta (latente) no uso dos termos nos documentos. Tal método não permite que esta estrutura semântica seja obscurecida

pela aleatoriedade da escolha de termos, fato que ocorre nos métodos tradicionais de recuperação de informação, visto que utilizam palavras individuais para serem recuperadas e indexadas [DDL⁺90, Dum91].

O LSI faz uso de uma técnica estatística conhecida como decomposição do valor singular (do inglês, *Singular Value Decomposition* - SVD) [GR70] para obter esta estrutura latente. O SVD é responsável por diminuir o "ruído" constituído pela polissemia e pela sinonímia existente nos documentos.

A estrutura semântica recuperada pelo LSI refere-se a estrutura que reflete a correlação entre as palavras individuais que aparecem nos documentos de uma coleção. Esta estrutura é denominada como semântica devido aos termos serem considerados dentro do contexto em que estão inseridos, ou seja, o método captura a semântica do termo, relacionando a este, os termos que o circundam.

2.4.1 Decomposição do Valor Singular

O modelo matemático de decomposição do valor singular SVD é utilizado na criação da estrutura semântica do LSI. Este modelo após realizar operações matriciais, obtém uma matriz de menor dimensão aproximada à matriz original. A matriz original representa uma relação (termo-documento, termo-termo ou documento-termo). Já o resultado das operações matriciais pode ser interpretado como uma configuração espacial na qual o produto cosseno entre vetores representa a similaridade estimada entre dois documentos.

Sendo assim, o SVD pode ser definido como uma técnica para reduzir o tamanho da matriz original. Cabe salientar, que o SVD se apóia no fato que, em geral, os dados da matriz original são altamente esparsos, visto que nem todos os termos aparecem em todos os documentos.

2.4.2 Execução do LSI

O primeiro passo da execução do LSI consiste na construção da matriz de incidência A , ou seja, é feita a contagem de termos para cada documento da coleção. Nesta matriz os T termos da coleção representam as linhas e os N documentos as colunas. Cada coluna corresponde ao vetor de um documento representado no modelo vetorial, onde a posição $A_{(t,d)}$ contém o peso do termo t no documento d .

O peso do termo caracteriza a importância do termo para definir o conteúdo do documento e pode ser obtido com base no número de ocorrências do termo (frequência). Esta frequência é calculada usando a medida *tf.idf* (*Term Frequency Times Inverse Document Frequency*)¹, que define a importância do termo buscando balancear características em

¹A medida *Term Frequency Inverse Document Frequency* (*tf.idf*) corresponde a uma medida estatística utilizada para avaliar o quanto uma palavra é importante para um documento em relação a uma coleção (*corpus*).

comum nos documentos (*intra-document*) e características para fazer a distinção entre os documentos (*inter-document*). O *tf.idf* atribui um peso ao termo para cada documento da coleção e indica o número de ocorrências do termo no documento ($tf_{t,d}$), modificada por uma escala de importância do termo (idf_t) na coleção, chamada de frequência inversa do documento [Dum91].

O próximo passo na execução do LSI é decompor a matriz A em um produto de três outras matrizes. Tal decomposição é feita com o uso do modelo matemático SVD:

$$\underset{T \times N}{A} = \underset{T \times R}{U_o} \cdot \underset{R \times R}{\Sigma_o} \cdot \underset{R \times N}{V_o^T}$$

A decomposição da matriz A é representada na Figura 2.1, onde as primeiras R colunas das matrizes ortogonais U e V definem os autovetores ortogonais associados aos R autovalores não zeros AA^T e $A^T A$, respectivamente. As colunas de U e V são referenciadas como os vetores singulares esquerdo e direito respectivamente, e os valores singulares de A são definidos pelos elementos da diagonal de Σ , que são as raízes quadradas dos T autovetores não nulos de AA^T .

Se somente os K maiores valores singulares de Σ_0 são mantidos com as suas correspondentes colunas em U_0 e V_0 e o restante removido, a matriz resultante A_k é a matriz de posto K que mais se aproxima da matriz original A :

$$\underset{T \times N}{A} \approx \underset{T \times N}{A_K} = \underset{T \times K}{U} \cdot \underset{K \times K}{\Sigma} \cdot \underset{K \times N}{V^T}$$

A quantidade K de valores singulares determina os número de dimensões do espaço conceitual em que os termos e os documentos serão representados. Em aplicações de recuperação de informação a escolha deste valor tem sido feita empiricamente [Dum91]. Neste caso, K é determinado pelo valor que fornece os melhores resultados de precisão na base de dados. A redução de dimensão permite remover muitos dos "ruídos" da coleção, mas um número reduzido pode implicar em perda de informação importante.

A Figura 2.1 mostra uma representação matemática para a decomposição do valor singular. As matrizes U_K e V_K^T resultantes da decomposição do valor singular, fornecem as coordenadas no espaço conceitual dos termos e dos documentos, respectivamente.

A recuperação de documentos com LSI é feita transformando-se as consultas para este espaço de conceitos através da equação $q = q^T U_K \Sigma_K^{-1}$, seguida da aplicação do modelo vetorial. Os vetores dos documentos são representados no espaço de conceitos através da matriz V_K^T , onde a j -ésima coluna ($1 \leq j \leq N$) representa um vetor que descreve o j -ésimo documento da coleção no espaço de conceitos.

Esta transformação da consulta é um dos principais problemas práticos do modelo LSI. Quando o modelo vetorial está sendo utilizado diretamente sobre a matriz A , o principal custo envolvido no processamento da consulta consiste na leitura das listas invertidas relacionadas aos termos fornecidos na consulta. Como as consultas submetidas para a

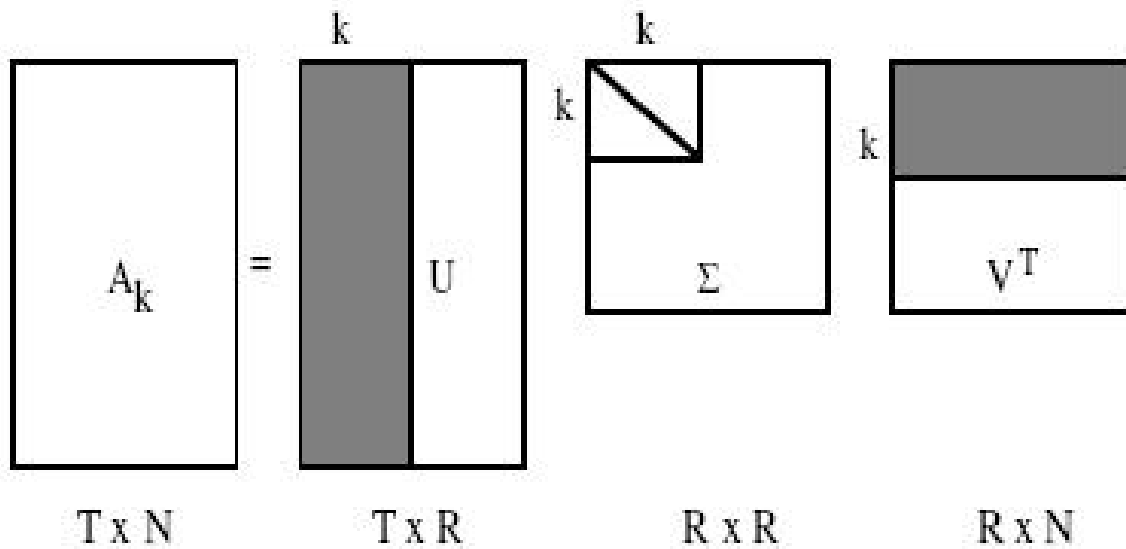


Figura 2.1: Decomposição do valor singular

maioria dos sistemas de recuperação de informação modernos são normalmente compostas de poucos termos, apenas algumas listas invertidas são lidas do disco.

Já no processamento do modelo de espaço vetorial aplicado sobre a matriz V_K^T , a consulta é transformada do espaço de termos para o espaço de conceitos, resultando em uma nova consulta, a qual normalmente possui um número de conceitos que é muito maior do que o número de termos da consulta original (tipicamente centenas de vezes maior). As listas invertidas de cada conceito desta nova consulta devem então ser lidas para que se descubra os documentos mais relacionados a mesma. Assim, o custo no processamento das consultas com o modelo LSI torna-se extremamente alto.

Outra desvantagem do modelo LSI é que muitas vezes a transformação da consulta para o espaço de conceitos descaracteriza a consulta original do usuário, pois a consulta não é feita a partir dos termos indicados pelo usuário. Entretanto como nosso foco é encontrar as relações entre os documentos e não sendo necessário a inserção de pesquisas pelo usuário, tal fator não representa impacto em neste trabalho.

Cada uma das associações que existem para construir a matriz A possui uma equação. A seguir se descreve cada uma dessas equações:

Comparação termo-termo. O produto co-seno entre dois vetores-linha de A_k mostra a extensão na qual dois termos têm um padrão semelhante de ocorrências, em um conjunto de documentos. A matriz resultante é uma matriz quadrada simétrica:

$$A = U \Sigma^2 U^T$$

Comparação documento-documento. O produto cosseno entre duas colunas mostra a extensão na qual dois documentos têm um perfil similar entre eles:

$$A = U \Sigma^2 V^T$$

Comparação termo-documento. O produto das matrizes resultantes de uma ma-

triz que é aproximadamente igual à matriz A , sendo esta matriz de nível k . Assim o produto resultante é a matriz A_k :

$$A = U \Sigma V^T$$

2.4.3 Exemplo de Aplicação do Método LSI

Nesta seção é apresentado um exemplo de aplicação do método LSI a uma coleção de documentos. Será desenvolvido o processo de decomposição e, após serem geradas as matrizes de nível k , será efetuado o *matching* entre uma consulta e a estrutura semântica encontrada. Note-se que esta coleção serve somente como exemplo para entender este método.

A "coleção" é composta pelos seguintes "documentos".

d1: Shipment of gold damaged in a fire.

d2: Delivery of silver arrived in a silver truck.

d3: Shipment of gold arrived in a truck.

Problema: Utilizar a indexação semântica latente (LSI) para recuperar os documentos segundo a consulta $q = \text{gold silver truck}$.

Passo 1: Montar a matriz termo-documento com as ocorrências respectivas. Esta matriz é visualizada na Tabela 2.1:

<i>Termos</i>	<i>d1</i>	<i>d2</i>	<i>d3</i>	<i>q</i>
a	1	1	1	0
arrived	0	1	1	0
damaged	1	0	0	0
delivery	0	1	0	0
fire	1	0	0	0
gold	1	0	1	1
in	1	1	1	0
of	1	1	1	0
shipment	1	0	1	0
silver	0	2	0	1
truck	0	1	1	1

Tabela 2.1: Ocorrências termo-documento

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 2 & 0 \\ 0 & 1 & 1 \end{bmatrix} \quad q = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

Passo 2: Aplicar a decomposição SVD à matriz A (matriz anterior). Encontram-se as matrizes U , Σ e V :

$$U = \begin{bmatrix} -0.4201 & 0.0748 & -0.0460 \\ -0.2995 & -0.2001 & 0.4078 \\ -0.1206 & 0.2749 & -0.4538 \\ -0.1576 & -0.3046 & -0.2006 \\ -0.1206 & 0.2749 & -0.4538 \\ -0.2626 & 0.3794 & 0.1547 \\ -0.4201 & 0.0748 & -0.0460 \\ -0.4201 & 0.0748 & -0.0460 \\ -0.2626 & 0.3794 & 0.1547 \\ -0.3151 & -0.6093 & -0.4013 \\ -0.2995 & -0.2001 & 0.44078 \end{bmatrix}$$

$$\Sigma = \begin{bmatrix} 4.0989 & 0.0000 & 0.0000 \\ 0.0000 & 2.3616 & 0.0000 \\ 0.0000 & 0.0000 & 1.2737 \end{bmatrix}$$

$$V = \begin{bmatrix} -0.4945 & 0.6492 & -0.5780 \\ -0.6458 & -0.7794 & -0.2556 \\ -0.5817 & 0.2469 & 0.7750 \end{bmatrix}$$

$$V^T = \begin{bmatrix} -0.4945 & -0.6458 & -0.5817 \\ 0.6492 & -0.71954 & 0.2469 \\ -0.5780 & -0.2556 & 0.7750 \end{bmatrix}$$

Passo 3: Neste exemplo iremos trabalhar com uma matriz A_k de nível igual a 2. Para isso, são utilizadas as primeiras k linhas e colunas das matrizes U , Σ e V para formar as

matrizes U_k , Σ_k e V_k com nível igual a 2.

$$U \approx U_K = \begin{bmatrix} -0.4201 & 0.0748 \\ -0.2995 & -0.2001 \\ -0.1206 & 0.2749 \\ -0.1576 & -0.3046 \\ -0.1206 & 0.2749 \\ -0.2626 & 0.3794 \\ -0.4201 & 0.0748 \\ -0.4201 & 0.0748 \\ -0.2626 & 0.3794 \\ -0.3151 & -0.6093 \\ -0.2995 & -0.2001 \end{bmatrix}$$

$$\Sigma \approx \Sigma_K = \begin{bmatrix} 4.0989 & 0.0000 \\ 0.0000 & 2.3616 \end{bmatrix}$$

$$V \approx V_K = \begin{bmatrix} -0.4945 & 0.6492 \\ -0.6458 & -0.7194 \\ -0.5817 & 0.2469 \end{bmatrix}$$

$$V^T \approx V_K^T = \begin{bmatrix} -0.4945 & -0.6458 & -0.5817 \\ 0.6492 & -0.7194 & 0.2469 \end{bmatrix}$$

Passo 4: Gerar a matriz $A_{k=2}$ multiplicando as matrizes anteriores utilizando a equação $A = U \Sigma V^T$. A matriz A de nível 2 resultante é:

$$A_{k=2} = \begin{bmatrix} -0.4945 & -0.6458 & -0.5817 \\ 0.6492 & -0.7194 & 0.2469 \end{bmatrix}$$

Passo 5: Efetuar o *matching* entre a consulta q fornecida e a coleção de documentos. Através da equação:

$$q = q^T U_K \Sigma_k^{-1}$$

$$q = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} -0.4201 & 0.0748 \\ -0.2995 & -0.2001 \\ -0.1206 & 0.2749 \\ -0.1576 & -0.3046 \\ -0.1206 & 0.2749 \\ -0.2626 & 0.3794 \\ -0.4201 & 0.0748 \\ -0.4201 & 0.0748 \\ -0.2626 & -0.3794 \\ -0.3151 & -0.6093 \\ -0.2995 & -0.2001 \end{bmatrix} \begin{bmatrix} \frac{1}{4.0989} & 0.0000 \\ 0.0000 & \frac{1}{2.3616} \end{bmatrix}$$

$$q = \begin{bmatrix} -0.2140 & -0.1821 \end{bmatrix}$$

Passo 6: Construir o *ranking* decrescente a partir do calculo da similaridade, $sim(q, d) = \frac{q \bullet d}{|q||d|}$, entre o documento d , coluna de A_k , e a consulta q .

$$sim(q, d_1) = \frac{(-0.2140)(-0.4945) + (-0.1821)(0.6492)}{\sqrt{(-0.2140)^2 + (-0.1821)^2} \sqrt{(-0.4945)^2 + (0.6492)^2}} = -0.0541$$

$$sim(q, d_2) = \frac{(-0.2140)(-0.6458) + (-0.1821)(0.7194)}{\sqrt{(-0.2140)^2 + (-0.1821)^2} \sqrt{(-0.6458)^2 + (-0.7194)^2}} = 0.9910$$

$$sim(q, d_3) = \frac{(-0.2140)(-0.5817) + (-0.1821)(0.2469)}{\sqrt{(-0.2140)^2 + (-0.1821)^2} \sqrt{(-0.5817)^2 + (-0.2469)^2}} = 0.4478$$

A Tabela 2.2 mostra-se o *ranking* para todos os documentos da coleção.

<i>Rank</i>	<i>Documento</i>	<i>Sim()</i>
1º	d_2	0.9910
2º	d_3	0.0541
3º	d_1	- 0.4478

Tabela 2.2: Ranking de documentos

A Figura 2.2 apresenta os resultados da consulta por meio da visualização no plano cartesiano.

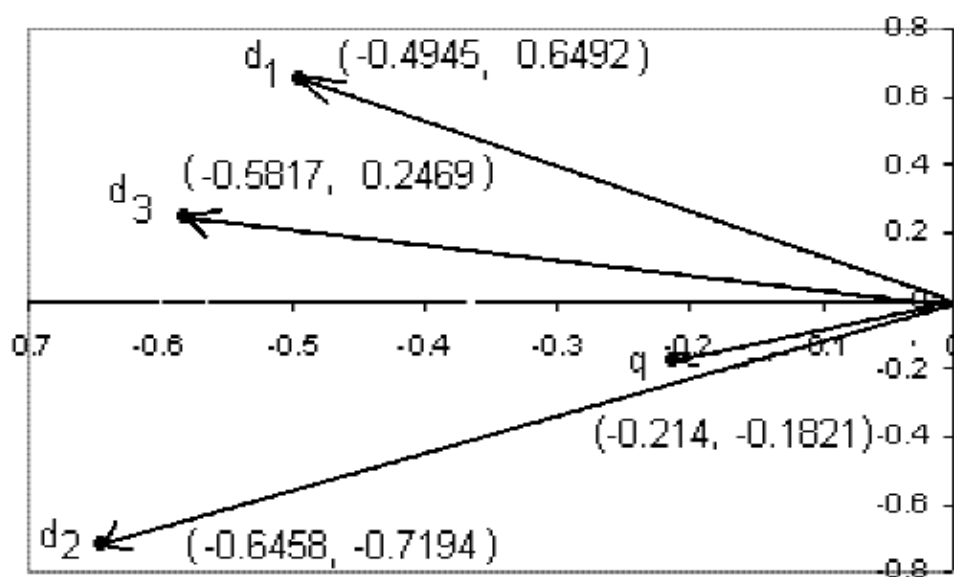


Figura 2.2: Visualização dos resultados da consulta no plano cartesiano

2.5 Conclusão

Neste capítulo apresentou-se os conceitos aplicados neste trabalho, ressaltando a descrição da técnica de recuperação de informação LSI que será utilizada no decorrer do trabalho. No próximo capítulo é apresentada a abordagem proposta para recuperação da visão inicial do software desconhecido.

Capítulo 3

Abordagem Proposta

Neste capítulo, apresenta-se uma abordagem que apóia à atividade de compreensão de um software desconhecido. Ela recupera uma visão inicial do software orientado a objetos e serve como ponto de partida para a compreensão do mesmo. Na Seção 3.1, apresenta-se os objetivos da ferramenta proposta. Na Seção 3.3, é detalhado o processo de recuperação de relações referenciais explícitas. Já a Seção 3.4 descreve o processo de recuperação de relações de semântica latente. A Seção 3.5 compara as relação de semântica latente e referencial explícita. Em seguida na Seção 3.6 é apresentado o modelo que unifica as duas abordagens. Por fim, a Seção 3.7 descreve como a abordagem foi implementada.

3.1 Objetivos da Abordagem

A hipótese apresentada na Seção 1.2 desta dissertação, sugere que as relações de semântica latente descobertas pela análise do vocabulário contido nos artefatos de código fonte, podem ser combinadas com as relações geradas pela análise referencial explícita do código fonte, resultando assim em um modelo unificado. Tal modelo pode ser utilizado como ponto de partida na compreensão de um software desconhecido. Sendo assim, propomos agilizar e apoiar a tarefa compreensão de software através do uso de uma ferramenta visual, que una o conhecimento informal oculto contido no vocabulário do código fonte com as relações semânticas referencial explícita do mesmo.

A compreensão de software ocorrerá através da identificação de agrupamentos que representem uma decomposição em módulos dos artefatos de código fonte. Tais agrupamentos serão construídos utilizando o modelo unificado proposto neste trabalho.

Portanto, faz parte do escopo do trabalho diminuir o grande esforço manual requerido para a realização da tarefa de compreensão de um software desconhecido, através da definição da abordagem e do desenvolvimento de um ferramental de suporte à mesma.

O foco desta pesquisa são sistemas orientados a objetos, pois uma das hipóteses estabelecidas é a de que os sistemas orientados a objetos serão os sistemas legados de um

futuro próximo dado o volume de desenvolvimento de software que vem ocorrendo neste paradigma.

A abordagem proposta busca combinar as relações entre artefatos de código fonte produzidas por duas técnicas distintas e independentes na construção de agrupamento semânticos, esses agrupamentos serão a base da compreensão do sistema alvo.

Concluindo, pode-se dizer que a abordagem proposta pretende construir uma visão inicial de software que possa a ser utilizada como ponto de partida na compreensão de sistemas legados e defina o desenvolvimento de um ferramental que suporte a abordagem proposta.

3.2 Descrição da Abordagem

A visão geral da estratégia proposta por esta abordagem é apresentada na Figura 3.1:

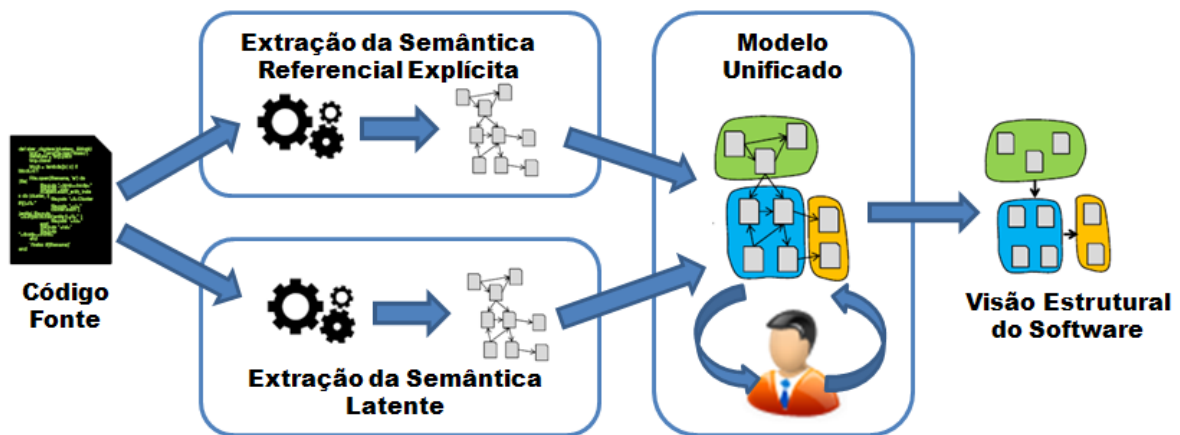


Figura 3.1: Visão geral da abordagem proposta

Nossa abordagem tem como entrada os artefatos de código fonte do sistema alvo. Estes artefatos são submetidos a dois processos de extração das relações semânticas, a extração da semântica referencial explícita e a extração da semântica latente. Os processos de extração podem ser executados de forma independente e ambos recuperam os relacionamentos entre os artefatos de código fonte.

Em seguida é feita a união dos dois grupos de relações encontrados. O resultado desta união é o modelo unificado proposto nesta pesquisa. Até esse momento, todas as atividades são executadas de modo automatizado, não exigindo assim uma interação com o usuário.

O resultado produzido pela abordagem é a visão inicial do software que é obtida por meio da análise dos agrupamentos apresentados pelo modelo unificado. Tal análise ocorre de forma interativa e incremental, exigindo assim a intervenção humana no processo.

Nas próximas Seções serão descritos de forma detalhada os elementos usados em nossa abordagem: a extração da semântica referencial explícita, a extração da semântica latente,

uma análise comparativa entre os resultados obtidos pelas duas técnicas de extração de relações semânticas e o modelo unificado de relações que é o utilizado para a construção da visão inicial do software.

3.3 Extração da Semântica Referencial Explícita

A extração da semântica referencial explícita visa capturar os relacionamentos entre os artefatos de código fonte de um programa, podendo estes serem de herança, associação, dependência e realização (ou implementação).

O processo de extração da semântica referencial explícita resulta em um grafo que descreve a inter-conexão entre os artefatos do código fonte, i.e., classes e interfaces do software implementadas em uma determinada linguagem de programação. Tal grafo reflete a implementação do software e se encontra em um baixo nível de abstração, fato que possibilita um posterior agrupamento das entidades em elementos arquiteturais.

As relações entre os artefatos do código fonte ocorrem quando elementos de uma classe atuam sobre elementos de outra classe. Por atuar, entenda-se que métodos de uma classe referenciam métodos ou variáveis da outra classe, além de relações de generalizações, associações ou de agregações em um software orientado a objetos.

A extração da semântica referencial explícita tem como entrada os artefatos de código fonte a partir do qual é gerado o grafo de relacionamento entre tais artefatos. Todo o processo de extração ocorre de forma automatizada, e se baseia apenas na análise semântica do código fonte, encontrando somente relações explícitas do mesmo.

O sistema alvo é composto por artefatos de código fonte, que por sua vez são formados por várias classes se o mesmo for desenvolvido utilizando o paradigma de orientação a objetos. Na linguagem Java, que é a utilizada pelos estudos de caso desta dissertação, normalmente cada classe é armazenada em um arquivo separado. Portanto, para se produzir o grafo de relações entre artefatos, é necessário submeter todas os artefatos, e conseqüentemente todos os arquivos, ao extrator da semântica referencial explícita.

Inicialmente é necessário determinar o universo para análise. Neste momento são descartados módulos com pouco significado semântico para o domínio do sistema alvo. Algumas classes internas também podem ser descartadas, porém tal decisão deve avaliar o significado semântico do módulo descartado para o processo de extração semântica referencial explícita. Exemplos de classes que podem ser descartadas são classes relacionadas a testes, mas esta é uma decisão do usuário.

Posteriormente, ocorre a análise dos artefatos do sistema alvo. Esta análise serve para prover o conhecimento sobre as classes declaradas no programa *Java*. Tais informações possibilitam realizar o mapeamento da árvore sintática destas classes. Neste momento é gerada uma estrutura de dados, a qual representa as relações de hierarquia (interfaces e heranças) juntamente com os elementos (atributos e métodos) declarados em cada classe.

A estrutura de dados gerada pela extração da semântica referencial explícita pode ser apresentada sobre a forma de um grafo, onde cada nó representa um artefato do código fonte e as arestas indicam o relacionamento entre os artefatos.

O objetivo da extração semântica referencial explícita é obter informações da especificação de um software para posterior análise no intuito de identificar os componentes da aplicação e seus inter-relacionamentos.

Outro aspecto, é que a extração semântica referencial explícita possibilita revelar as dependências entre as classes, detectando assim o grau de acoplamento entre elas.

Em suma, a extração da semântica referencial explícita deverá encontrar relações entre os artefatos de código fonte, que são classificados pela UML (*Unified Modeling Language*) [BRJ05] como:

- **Associação:** relacionamento estrutural entre instâncias que especifica que objetos de uma classe estão ligados a objetos de outras classes. Podemos ter associação unária, binária, etc.
- **Dependência:** relacionamento de utilização no qual uma mudança na especificação de um elemento pode alterar a especificação do elemento dependente. A dependência entre classes indica que os objetos de uma classe usam serviços dos objetos de outra classe.
- **Generalização:** relacionamento entre um elemento mais geral e um mais específico. O elemento mais específico herda as propriedades e métodos do elemento mais geral. A relação de generalização é suportada através do mecanismo de herança no modelo a objetos e linguagens associadas. Como a relação de dependência, ela existe só entre as classes.
- **Agregação:** relacionamento com características todo-parte, onde existe um grau de coesão entre o todo e as partes menos intenso, podendo haver certo grau de independência entre os mesmos (todo e as partes).
- **Composição:** relacionamento entre um elemento (o todo) e outros elementos (as partes) onde as partes só podem pertencer ao todo e são criadas e destruídas com ele.

Em nossa abordagem, não será necessário classificar as relações encontradas pela extração semântica referencial explícita, pois o intuito da abordagem é identificar os agrupamentos semânticos dos artefatos do código fonte.

3.4 Extração da Semântica Latente

As relações de semântica latente entre os artefatos de código fonte, serão obtidas com o uso da técnica de recuperação de informação LSI. Tal técnica é normalmente utilizada

em grandes coleções de documentos da internet, porém se consideramos os artefatos de código fonte como documentos textuais e ignorarmos a informação presente na sintaxe das instruções da linguagem, o LSI pode ser utilizado perfeitamente neste artefatos.

Antes do processo de descoberta em si, é necessário um pré-processamento dos artefatos de código fonte. Este pré-processamento busca preparar os artefatos a fim de aumentar a qualidade das relações descobertas.

Em geral, os artefatos são descrições textuais, porém podem existir artefatos em diferentes formatos, textuais ou não. Todos estes diferentes formatos devem ser convertidos em texto puro, extraindo assim apenas os dados úteis para a descoberta de relações semânticas latentes.

A atividade de pré-processamento ocorre de forma automática tendo como entrada os artefatos do sistema alvo. Cabe a essa atividade eliminar informações que sejam irrelevantes ao processo de descoberta de relações. São consideradas informações irrelevantes e que devem ser eliminadas: os caracteres especiais e as *stop words*¹. Tais informações são eliminadas por conterem pouco valor semântico para o processo. Quando lidamos com artefatos de código fonte são excluídos também as palavras-chaves e os elementos da sintaxe da linguagem. Nesse caso especial de artefato, também é feita uma separação de termos, quando estes forem identificadores que usem o padrão camelCase.

Alguns artefatos são compostos por componentes que juntos formam grandes blocos de informações. É também função da atividade de pré-processamento dividir os blocos em documentos menores e mais concisos, com isso ajudando na precisão da indexação. Por exemplo: os pacotes de códigos-fonte da linguagem Java podem ser divididos em classes; um manual de usuário pode de dividido em capítulos.

O resultado da atividade de pré-processamento é um corpus que contém o vocabulário de cada artefato. Este será fornecido como entrada para o processo de descoberta de relações.

A descoberta de relações é responsável por encontrar as ligações entre os artefatos de código fonte. Esta atividade faz uso de técnicas de recuperação de informação. A similaridade entre os artefatos é determinada pela técnica de LSI. Cada ligação contém um score que indica o grau de similaridade. Esta similaridade representa a correlação conceitual entre artefatos baseada nos termos contidos nos mesmos. O score pode assumir valores entre 0.0 (nada similar) e 1.0 (completamente similar).

Tanto o pré-processamento, quanto a descoberta das relações ocorrem sem a interação com o usuário, ou seja, são executadas automaticamente.

A saída produzida pelo processo de descoberta das relações de semântica latente é um grafo, onde os nós são artefatos de código fonte e as arestas indicam as relações semânticas

¹*Stop words*, ou palavras de parada (tradução livre), são artigos, proposições, pronomes, palavras curtas e comuns que os motores de busca ignoram nas pesquisas. Exemplos: as, e, os, de, para, com, sem, foi.

entes estes artefatos.

3.5 Semântica Referencial Explícita versus Semântica Latente

Os dois processos de extração de relações semânticas utilizados em nossa abordagem buscam identificar a similaridade conceitual entre os elementos de um programa, porém utilizam estratégias diferentes. A extração da semântica referencial explícita se concentra nas definições explícitas do código fonte, já a extração de semântica latente busca informações implícitas contidas no vocabulário utilizado pelo desenvolvedor nos identificadores e comentários.

Entretanto, apesar de compartilhar o objetivo, a extração da semântica latente e a extração da semântica referencial explícita produzem resultados distintos. Sendo assim, uma forma de comparar as duas abordagens é analisar as possibilidades de resultados que as duas abordagens podem produzir ao serem submetidas ao mesmo sistema alvo.

A Figura 3.2 apresenta o universo de situações encontradas:

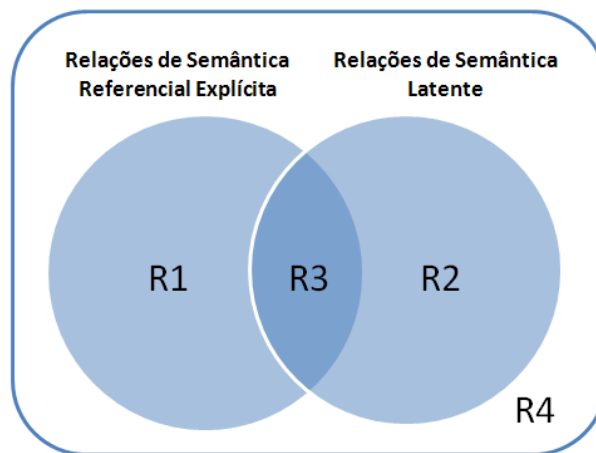


Figura 3.2: Semântica referencial explícita versus semântica latente

- **R1:** A extração semântica referencial explícita encontrou uma relação entre os artefatos, porém o mesmo não ocorre na extração semântica latente. Fica evidente a existência da relação entre as entidades. Entretanto, tal relação por estar indicada apenas de forma explícita no código fonte, não apresentando deste modo, indícios de coesão semântica entre os identificadores e/ou comentários do código fonte.
- **R2:** A relação é recuperada apenas pela extração semântica latente. Neste caso, há uma relação implícita entre os artefatos de código fonte. O fato da relação não estar declarada diretamente no código, abre margem para uma análise de coesão e se necessário e oportuno, uma reorganização do código.

- **R3:** Ambas as abordagens encontraram a relação entre os artefatos. Aqui não fica dúvida sobre a relação semântica entre as entidades, as relações de semântica latente são validadas pelas relações de semântica referencial explícita.
- **R4:** A relação entre os artefatos não é recuperado por nenhuma das abordagens, sendo assim, concluímos que não existe conexão semântica entre as entidades.

O fato das abordagens adotarem focos distintos abre a possibilidade de união das mesmas. Os resultados produzidos pelos processos de extração semântica podem se completar, estimulando a criação de um modelo que combine os dois processos. Tal união pode resultar no enriquecimento da abordagem, e consequentemente na qualidade das relações entre artefatos de código recuperadas.

3.6 Unificando as Abordagens

Nossa abordagem busca combinar os resultados, relações entre artefatos, produzidos pela extração da semântica latente com os encontrados pela extração da semântica referencial explícita do código fonte.

A linha mestra da abordagem são relações encontradas através da extração da semântica latente, ou seja, é a análise destas relações que guiarão todo processo. Sendo assim, esta análise é a interface de interação pela qual o usuário conduzirá a abordagem.

Já as relações da semântica referencial explícita, serão utilizadas como forma de indentificar os agrupamentos entre os artefatos de código fonte. Ou seja, os relacionamentos implícitos entre artefatos encontrados pela técnica de recuperação de informação LSI são validados pelas conexões explícitas presentes no código fonte recuperadas através do extrator de semântica referencial explícita. Sendo assim, as duas técnicas de extração semântica serão executadas separadamente e seus resultados unificados.

As relações encontradas apenas pelo extrator de semântica referencial explícita não serão priorizadas, pois não trazem informações que complementem o processo, visto que nossa abordagem busca valorizar o conhecimento oculto contido no vocabulário do código fonte. Deste modo, utilizaremos apenas as relações de semântica latente e as relações presentes em ambos os processos de extração.

Conforme mencionado, a compreensão do sistema alvo ocorre de forma empírica, dependendo assim, da análise de um especialista humano. As relações são exibidas na forma de um grafo, onde os nós são os artefatos e as arestas as relações entre os artefatos.

Os grafos são uma ótima maneira de representar informações técnicas, pois são adequados para exibir relações entre elementos, facilitando assim, sua leitura e entendimento. Por isso, diversos problemas de apresentação de dados envolvem a visualização de grafos [Nor92]. Máquinas de estado finito, redes, grafos de chamadas de programas, e

vários outros tipos de dependência entre objetos são exemplos de informações que podem ser compreendidas mais facilmente quando apresentadas na forma de diagramas.

Contudo, é importante possuir um meio de construir, visualizar e interagir com os grafos. Em nossa abordagem, a interação ocorre por meio do fator de calibragem que é usado de forma gradativa e decrescente, onde se busca encontrar uma melhor visualização das relações entre os artefatos do sistema alvo. Este fator de calibragem incide sobre as relações de semântica latente, exibindo apenas aquelas com *score* de similaridade superior ao fator de calibragem.

As relações de semântica referencial explícita são usadas na identificação dos agrupamentos de artefatos de código fonte. Vale lembrar, que somente são exibidas as relações entre artefatos que são encontradas também pela extração de semântica latente. Sendo assim, o aumento da quantidade de relações de semântica latente, aumentará também a possibilidade de ocorrência de agrupamentos entre os artefatos.

Como dito anteriormente, o grafo que é utilizado na criação da visão estrutural do sistema alvo contém dois tipos de relações entre os nós:

- relações produzidas apenas pela extração da semântica latente.
- relações produzidas por ambos os processos de extração.

As relações produzidas por ambos os processos tem maior relevância, visto que representam tanto uma relação explícita quanto uma relação implícita. Isto sugere que os artefatos ligados por esse tipo de relação tem uma forte conexão semântica, podendo assim serem agrupados.

Já o relacionamento produzido apenas pela semântica latente indica uma relação entre artefatos de agrupamentos distintos, indicando assim uma interface de ligação entre estes elementos do sistema.

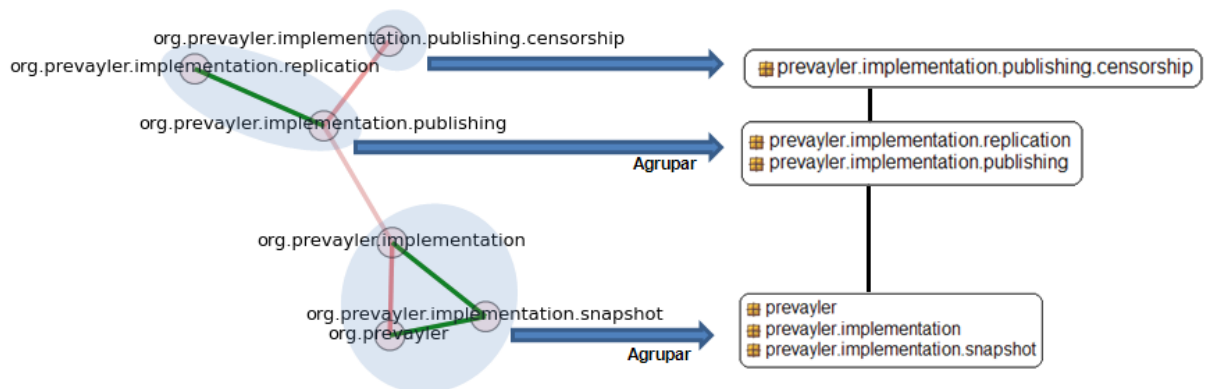


Figura 3.3: Agrupamento de elementos

A Figura 3.3 exemplifica como é feito o agrupamento dos elementos em nossa abordagem. Inicialmente temos um grafo onde os elementos, neste exemplo pacotes do sistema

Prevayler [Wue09], são interconectados por arestas das cores verde e vermelho. Os elementos ligados pela cor vermelha indicam que existe uma relação de semântica latente, já quando a aresta é da cor verde, ambas as relações foram identificadas, ou seja, existe uma relação entre os elementos tanto de semântica latente quanto de semântica referencial explícita.

Por conterem uma ligação mais forte, os elementos conectados por arestas verdes são agrupados. Neste caso, observamos a união dos seguintes pacotes em agrupamentos:

- K_1

prevayler.implementation.publishing.censorship.

- K_2

prevayler.implementation.replication

prevayler.implementation.publishing.

- K_3

prevayler

prevayler.implementation

prevayler.implementation.snapshot

No caso dos pacotes *prevayler* e *prevayler.implementation* pertencentes ao agrupamento K_3 , eles não são ligados por uma aresta de cor verde, porém graças a propriedade de transitividade eles foram caracterizados como pertencentes ao mesmo agrupamento.

Entretanto, existem grafos onde a identificação do agrupamentos não pode ser feita apenas unindo os elementos ligados por uma aresta de cor verde. A Figura 3.4 mostra um grafo que representa o *OpenJDK* [Mic09b]. Este grafo exige que desenvolvedor identifique visualmente os agrupamentos, sendo então sua responsabilidade identificar os limites que definem os agrupamentos.

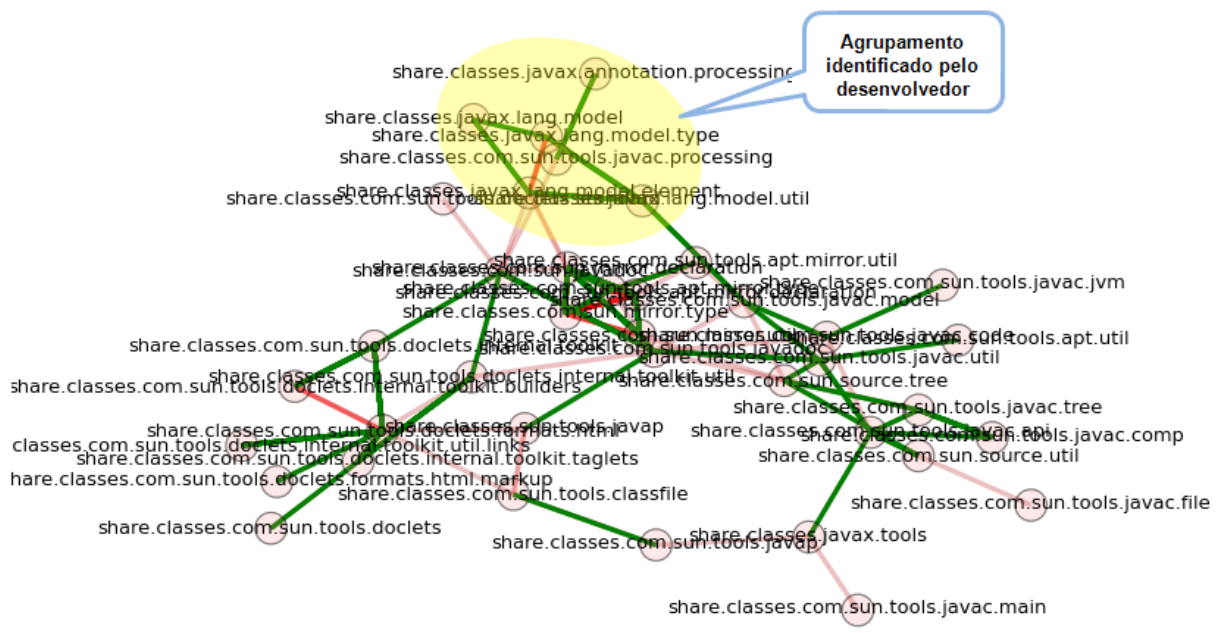


Figura 3.4: Agrupamento visual dos elementos

3.7 Implementação da Abordagem

Nesta seção a ferramenta desenvolvida é descrita incluindo as características implementadas, a arquitetura e alguns detalhes da implementação. A ferramenta foi desenvolvida com o intuito de apoiar a tarefa de compreensão de software, pela construção de uma visão estrutural do software. Neste sentido, a ferramenta tenta reduzir o esforço para entender as relações entre os elementos do código fonte através da geração de um grafo. Dentre os benefícios esperados pelo uso da ferramenta estão: baixo esforço na recuperação de uma visão estrutural de um software e a valorização do conhecimento já adquirido pela equipe de desenvolvimento.

A Figura 3.5 apresenta o diagrama de componentes da ferramenta de acordo com a visão geral da abordagem mostrada na Figura 3.1. Os principais componentes da ferramenta são: extrator de semântica latente, extrator de semântica referencial explícita e o gerador e visualizador de grafos.

A ferramenta foi implementada utilizando as linguagens *Java* e *Python*. O componente extrator de semântica latente e o componente gerador e visualizador de grafos foram desenvolvidos em *Python*. Já extrator de semântica referencial explícita foi contruido em *Java*.

O extrator de semântica latente encontra as relações ocultas entre os artefatos de código fonte através da classe *ExtratorLatente.py* que está associada as classes *LSI.py* e *VetorSpace.py*. A saída deste componente são as relações de semântica latente.

A classe *VetorSpace.py* é responsável por construir a representação da coleção de documentos, ou seja, transformar os artefatos de código fontes em vetores. Para isso é utilizado um *paser* implementado na classe *Paser.py*. O *paser* faz uma análise morfológica

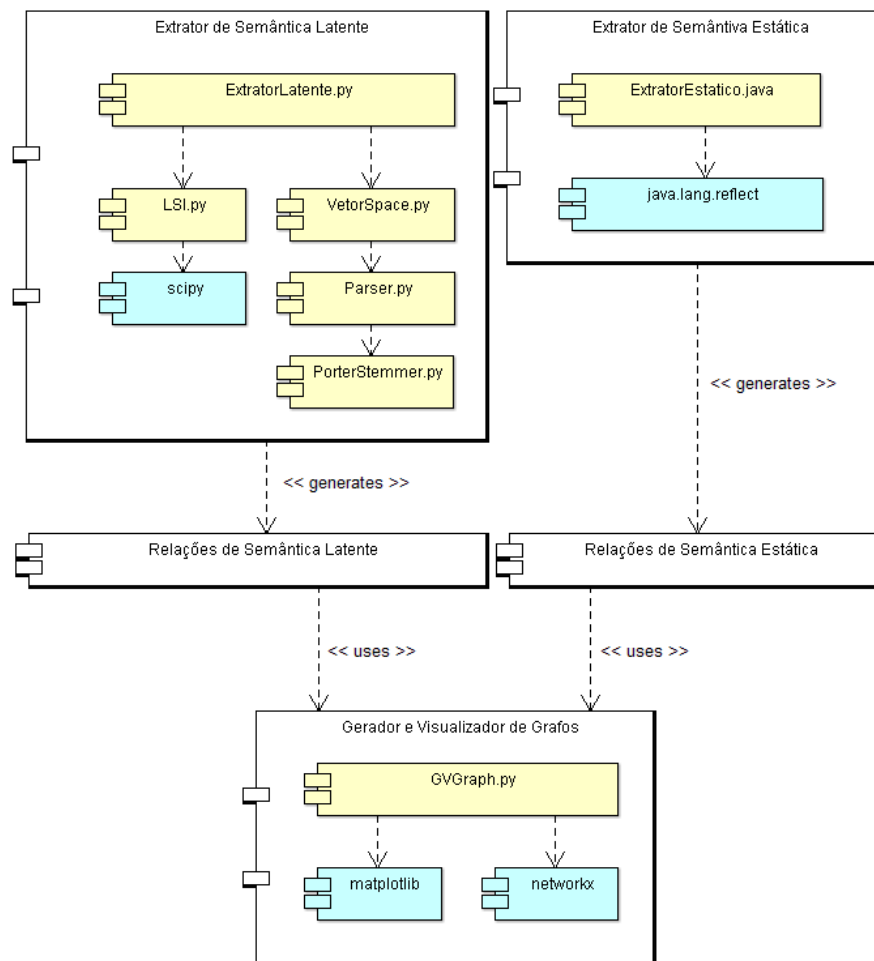


Figura 3.5: Diagrama de componentes da ferramenta

preliminar com o uso do algoritmo *Porter Stemmer* [Por09] implementado na classe *PorterStemmer.py*.

A indexação semântica latente é feita pela classe *LSI.py* que utiliza o pacote básico *SciPy* [JOP⁺01] da linguagem *Python*, este pacote implementa diversas técnicas de computação científica, e nossa ferramenta o utiliza no cálculo do SVD.

O extrator de semântica referencial explícita tem como entrada os artefatos de código fonte do sistema alvo e constroi as relações de semântica referencial explícita usando *Java Reflection API* [Mic09c].

A ferramenta é manipulada através do componente gerador e visualizador de grafos. O grafo é construído pela classe *GVGraph.py* utilizando os pacotes *Matplotlib* [Mat09] e *NetworkX* [HSS08]. O *Matplotlib* é usada na criação de gráficos e plotagens de diversas formas. Já o pacote *NetworkX* é usado na criação, manipulação e estudo da estrutura de redes complexas.

Como visto, a implementação da ferramenta é feita utilizando algumas bibliotecas externas. Observe que houve uma divisão funcional da ferramenta em componentes, isto

facilita futuras adaptações.

3.8 Conclusão

Este capítulo descreveu a metodologia utilizada para se recuperar da visão estrutural de um software desconhecido. Esta visão é obtida a partir da união das relações de semântica latente e referencial explícita. Por fim, foi descrito alguns detalhes arquiteturais da ferramenta que implementa nossa abordagem.

Capítulo 4

Avaliação da Abordagem Proposta

4.1 Metodologia de avaliação

Os estudos experimentais que serão apresentados neste capítulo se enquadram na categoria de estudos de caso, estratégia de estudo experimental selecionada devido a uma série de fatores, como:

- escassez de tempo e recursos (ex: participantes) para o planejamento e execução dos estudos;
- tipo de avaliação que se deseja realizar (ou seja, a viabilidade e eficácia da abordagem de recuperação de uma visão estrutural de um software desconhecido);
- impossibilidade de controle total sobre as variáveis dos estudos (ex: perfil dos participantes e nível de compreensão esperado, a qual é avaliada por especialistas das aplicações).

Em relação aos seus objetivos, os estudos realizados buscam verificar a viabilidade da abordagem proposta em um sistema real. Apresenta-se neste capítulo dois estudos de caso: um sobre a ferramenta *Prevayler* e outro sobre o compilador do *OpenJDK* com o intuito de comprovar as diferentes aplicações da abordagem.

A Figura 4.1 apresenta a estratégia usada na verificação da viabilidade da abordagem proposta.

A equipe de desenvolvimento tem uma visão inicial do software a ser desenvolvido, visão que é expressa através da documentação do software. Sendo assim, a verificação da viabilidade ocorrerá submetendo os artefatos de código do sistema alvo à abordagem proposta, e em seguida comparando a visão inicial do software definida pela equipe de desenvolvimento com a obtida por meio da abordagem definida nesta pesquisa.

A comparação será realizada com o auxílio do paradigma “*Goal Question Metric*” (*GQM*) [BCR94]. Este paradigma foi aplicado buscando caracterizar e fornecer um melhor

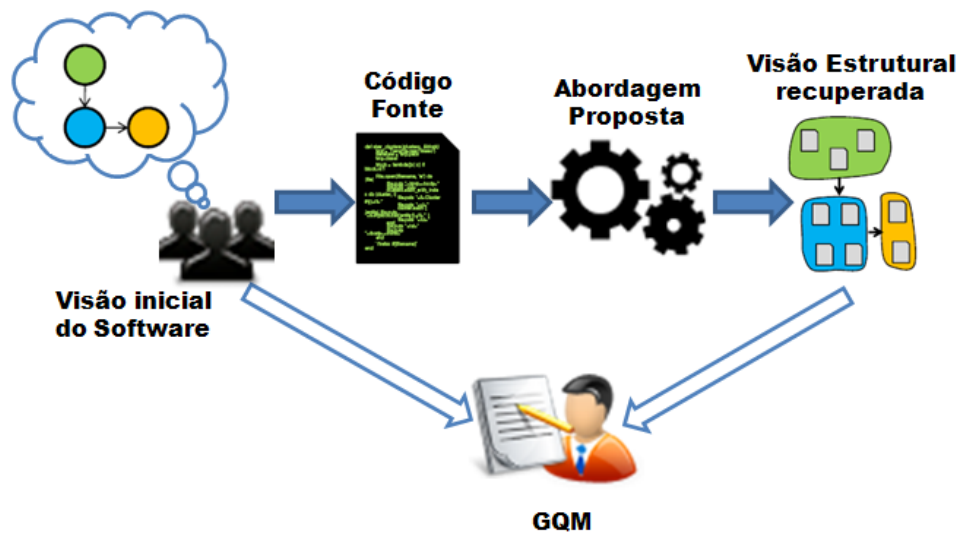


Figura 4.1: Estratégia usada na verificação da viabilidade da abordagem proposta.

entendimento do processo e, assim, estabelecer bases para comparação com trabalhos relacionados. Uma breve explicação desse paradigma é apresentado na Seção seguinte.

4.1.1 Goal Question Metric (GQM)

O GQM baseia-se na definição de objetivos a serem mensurados para que estes sirvam de rota para o estabelecimento de questões que orientarão a definição de métricas em um contexto particular [MB97]. Importa destacar que o sucesso da aplicação do GQM está diretamente ligado à definição dos objetivos, uma vez que as escolhas das métricas e posterior avaliação dos dados dependem dos objetivos como mostrado na Figura 4.2.

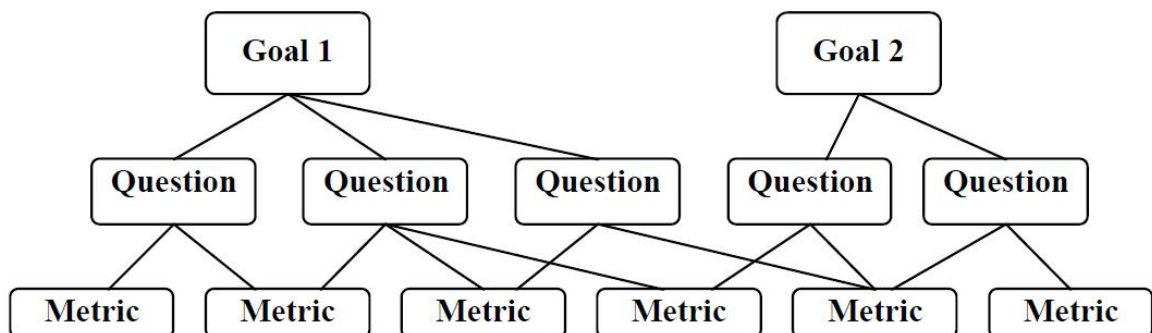


Figura 4.2: Estrutura hierárquica do GQM.

Briand et al. [BDR96] definiram uma estrutura modelo para GQM baseado em cinco aspectos:

- O **objetivo** define o objeto, o alvo, que será estudado, isto é, o processo ou produto que será analisado.
- O **propósito** do estudo expressa a intensão que se tem ao analisar o objeto em questão. Propósitos comuns, em ordem de dificuldade, são:

Dimensão	Definição	Exemplos
Objetivo do estudo	O que será analisado	teste de sistema, processo de desenvolvimento, produto final, etc
Propósito	Para que será analisado o objeto	caracterização, avaliação, predição, etc
Foco	Qual propriedade/atributo do objeto será analisado	custo, mudanças, corretude, etc
Ponto de vista	Quem usará os dados coletados	desenvolvedores, líderes de projeto, usuários, etc
Contexto	Em qual ambiente	projeto X, na corporação A, etc

Tabela 4.1: Sumário do modelo GQM

Caracterização: Retratar o atual estado ou desempenho de um processo de desenvolvimento de software ou produto.

Monitoração: Investigar tendências/evoluções de um estado/desempenho de um processo ou produto.

Avaliação: Comparar e medir a qualidade de um produto e a eficiência/eficácia de um processo.

Predição: Identificar relações entre vários fatores de processos ou produtos e através dessas relações prever atributos relevantes do processo ou produto.

Controle ou mudança: Identificar relacionamentos que influenciam o estado/desempenho de um processo ou produto.

- O **foco** define um atributo em particular do objeto em estudo que estará sujeito a um propósito. Exemplos de focos são custos, mudanças, manutenibilidade, etc.
- O **ponto de vista** identifica os papéis ou posições do pessoal que usará os resultados obtidos. Por exemplo, quem interpretará os dados coletados e usará para prever modelos.
- O **contexto** do estudo especifica o ambiente no qual o paradigma será aplicado.

Um sumário do modelo é apresentado na Tabela 4.1. Um exemplo paradigma é apresentado na Figura 4.3.

Objetivo 1 Verificar a aderência ao Processo

Propósito: Verificar
Questão: a aplicação
Objeto: do processo de manutenção de software
Ponto de Vista: analisado pelo ponto de vista dos gerentes de projetos

Questão 1: Qual o grau de aplicabilidade do processo de manutenção?

Métrica 1: Densidade de execução das Tarefas

$$X = \frac{\text{N}^\circ \text{ de Tarefas Realizadas nos Ciclos}}{\text{N}^\circ \text{ de Tarefas Previstas nos Ciclos}}$$

Métrica 2: Porcentagem da Precisão dos Prazos

$$X = \frac{\text{N}^\circ \text{ de Solicitações Atendidas no Prazo} * 100}{\text{N}^\circ \text{ de Solicitações Feitas Durante o Período de Avaliação}}$$

Métrica 3: Precisão do Esforço

$$X = \frac{\text{Total de homens-hora realizado em cada tarefa da Implementação}}{\text{Total de homens-hora dimensionado em cada tarefa da Implementação}}$$

Figura 4.3: Exemplo modelo GQM [DdSK03].

4.2 Definição do Estudo

Objetivo do Estudo

Analisar a visão estrutural de um software desconhecido recuperada por nossa abordagem

com o propósito de caracterizar a viabilidade do uso dessa visão,

tem como foco a tarefa de compreensão de software

sob o ponto de vista de equipe de desenvolvimento

no contexto manutenção de aplicações de grande porte.

Questões

As questões e métricas utilizadas para análise dos resultados são apresentadas abaixo:

Q1: Os agrupamentos recuperados são precisos, i.e. estão corretos?

M1: Análise qualitativa do especialista.

Q2: As relações entre os agrupamentos são precisas?

M1: Análise qualitativa do especialista.

Q3: A visão estrutural recuperada apoia a compreensão da aplicação?

M3: Análise qualitativa do especialista.

4.3 Execução do Estudo Experimental

Inicialmente é preciso determinar quais artefatos serão submetidos à abordagem. Para os estudos de caso apresentados nesta dissertação foram excluídos os pacotes referentes a exemplos e demonstrações. Outro ponto fundamental é definir qual o grau de granularidade utilizada na análise dos agrupamentos. Devido ao grande volume de informações encontrado nos estudos de caso escolhidos, será utilizada a granularidade de pacotes, ou seja os elementos apresentados representam pacotes do sistema alvo. Tal escolha visa facilitar a análise, pois diminui a quantidade de elementos e ocorreu após avaliações em estudos preliminares.

A análise da visão estrutural recuperada depende do fator de calibragem, que é diferente para cada software, e tem a função de determinar o limiar de corte para a exibição de ambas as relações de semântica latente e de semântica referencial explícita. Este fator também é definido de forma empírica, ou seja, através de experimentos que são submetidos à avaliação de um especialista humano.

Diante disso, o fator de calibragem é definido por um especialista para cada estudo de caso. Essa definição é feita comparando os resultados encontrados por fatores de calibragem candidatos. Inicialmente são testados valores próximos a 1 e gradativamente são testados valores menores, até que se encontre um fator que represente os agrupamentos, ou seja, que se encontre agrupamentos onde o especialista reconheça em sua organização uma visão do software.

As Seções 4.3.1 e 4.3.2 apresentam os estudos de caso das aplicações *Prevayler* e do compilador do *OpenJDK* respectivamente. Por fim, na Seção 4.4 a visão estrutural do software é avaliada pelo especialista utilizando as questões e métricas definidas segundo o paradigma GQM.

A Tabela 4.2 mostra algumas métricas extraídas dos softwares *Prevayler* e *OpenSDK* com o *plugin Metrics* [Sau09] para o ambiente de desenvolvimento integrado *Eclipse*. As métricas extraídas são:

1. NOP: número de pacotes (com classes ou interdaces)
2. NOC: número de classes
3. NOI: número de interfaces
4. NOM: número de métodos
5. NOC+NOI: número de classes + interfaces

6. TLOC: número total de linhas de código fonte

<i>SistemaAlvo</i>	<i>NOP</i>	<i>NOC</i>	<i>NOI</i>	<i>NOM</i>	<i>NOC + NOI</i>	<i>TLOC</i>
Prevayler	13	61	23	357	84	2.974
OpenJDK	49	752	214	7.337	966	86.106

Tabela 4.2: Métricas extraídas do código fonte do *Prevayler* e do compilador do *OpenJDK* com o *plugin* do *Eclipse Metrics*

4.3.1 *Prevayler*

O primeiro sistema alvo escolhido para o estudo de caso foi o *Prevayler* [Wue09]. Tal escolha foi motivada porque este é um projeto código de fonte aberto, sendo assim seus artefatos estão facilmente disponíveis. Outro fator importante é a grande comunidade de usuários do projeto.

O *Prevayler* implementa o conceito de Prevalência em *Java*, sendo recomendado para sistemas que tenham memória RAM suficiente para conter todos os seus dados. Como os dados são mantidos na memória, e as consultas são executadas recuperando objetos nesta memória, o acesso torna-se mais rápido do que em sistemas que requerem acesso a disco, deste modo seu desempenho é superior a sistemas de persistência tradicionais.

Encontrando o Fator de Calibragem para o *Prevayler*

Como descrito na seção 3.6, a descoberta do fator de calibragem se faz de forma gradativa e incremental. Para este estudo de caso iniciamos escolhendo o valor 0.75 como mostra o grafo de rede apresentado na Figura 4.4. Observamos a presença de 8 pacotes do *Prevayler*, de um total de 13 pacotes. Apenas uma relação entre pacotes foi encontrada, logo é necessário diminuir o fator de calibragem para obter um grafo que expresse melhor o sistema alvo.

Ao decrementarmos o fator de calibragem para 0.6, apresentado na Figura 4.5, vemos um ganho de abrangência em relação à quantidade de pacotes e de interconexões encontradas. Porém, ao analisar o grafo gerado constatamos que ainda não é possível abstrair a visão estrutural.

Continuando a sucessão de decrementos, com o fator de calibragem em 0.5 é possível notar, como mostra a Figura 4.6, a presença de um agrupamento central para o *Prevayler*, possivelmente seu núcleo. Entretanto existem ainda diversos pacotes que estão desconectados deste agrupamento.

O grafo gerado com o fator de calibragem igual a 0.4, visto na Figura 4.7 traz uma melhor visualização do sistema alvo, visto que é possível identificar alguns pacotes sendo centros de relações e outros permanecendo em regiões periféricas. Logo, este é um forte

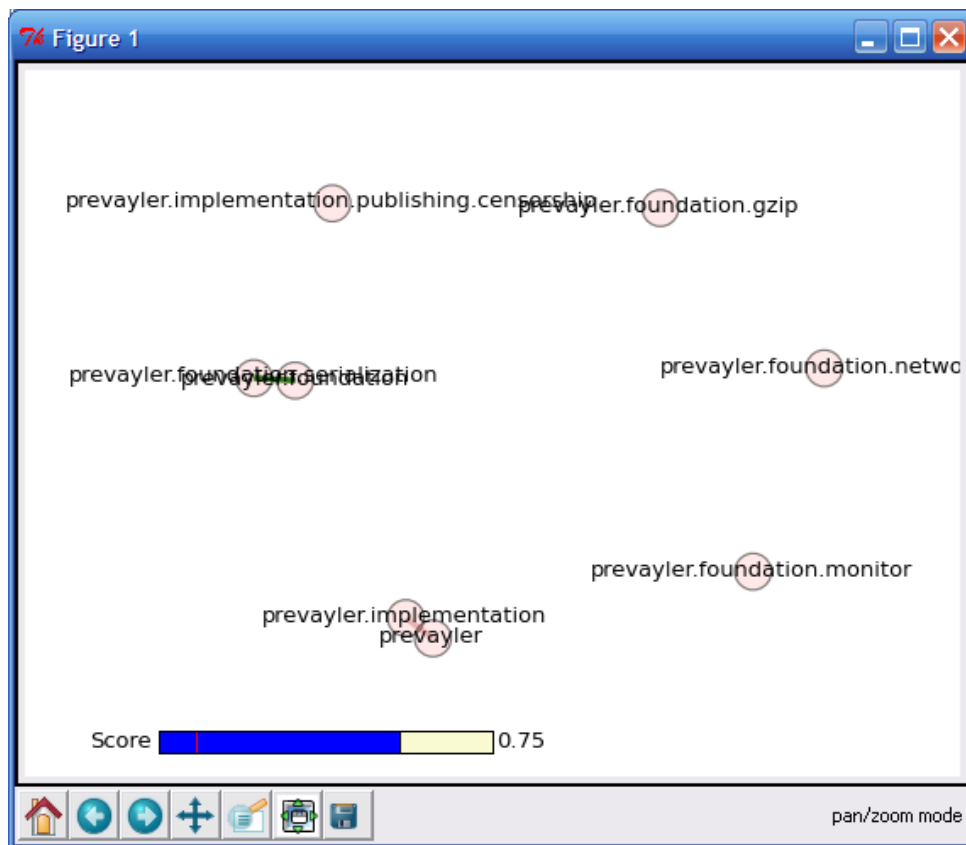


Figura 4.4: Grafo do *Prevayler* com fator de calibragem 0.75

candidato a fator de calibragem, porém devemos continuar os decrementos a fim de confirmar essa hipótese.

Os fatores 0.3 e 0.2, como observamos nos respectivas grafos das Figuras 4.8 e 4.9, apresentam uma grande quantidade de relações entre os pacotes, dificultando assim sua análise. Deste modo, optamos por utilizar o fator de calibragem 0.4.

Como visto, a definição do fator de calibragem depende fundamentalmente de uma avaliação qualitativa dos grafos gerados a partir de fatores candidatos. Como o processo é conduzido por um especialista humano, o fator de calibragem pode variar dependendo do conhecimento que o mesmo tenha sobre o domínio da aplicação.

Identificando os Agrupamentos

A seção anterior decreveu o processo no qual se definiu o valor 0.4 como fator de calibragem para o estudo de caso do *Prevayler*. O próximo passo é identificar os agrupamentos a partir do grafo gerado por esse fator de calibragem e que pode ser visto na Figura 4.7. Neste grafo podemos notar que alguns elementos são conectados por arestas de cor verde, isso indica que eles possuem uma forte ligação semântica, e portanto podem ser agrupados.

A Figura 4.10 apresenta o diagrama resultante do proceso de união dos pacote do *Prevayler* onde a aresta que os conecta tem a cor verde, ou seja representa uma relação

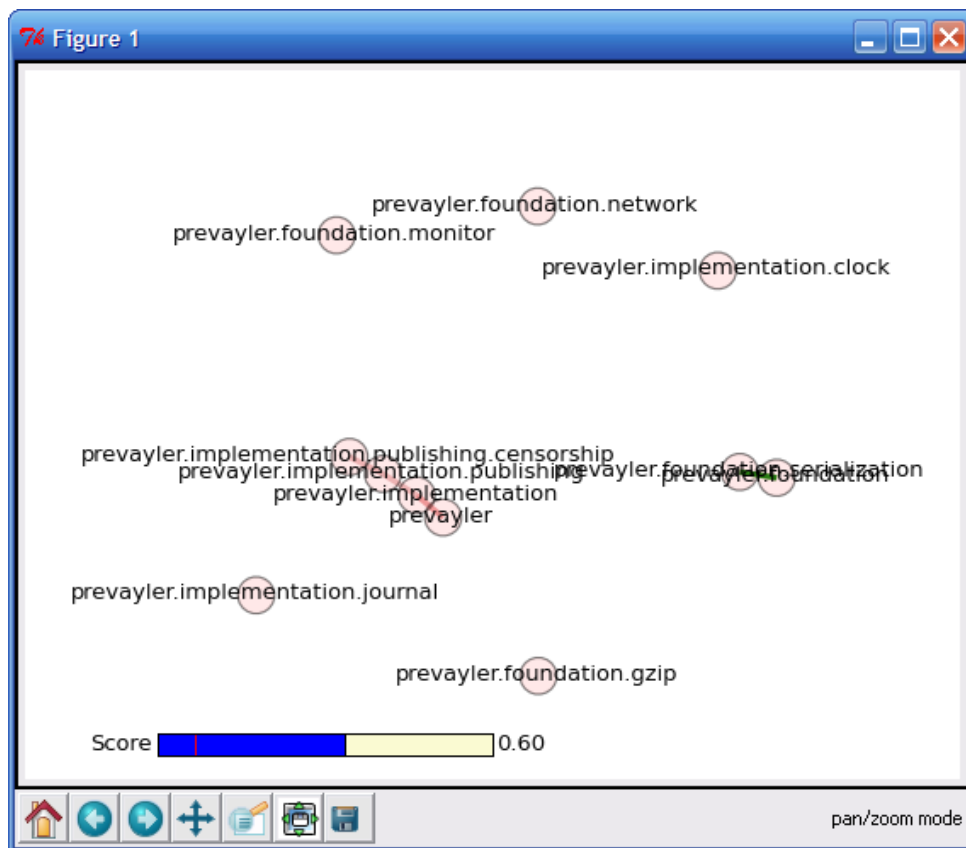


Figura 4.5: Grafo do *Prevayler* com fator de calibragem 0.6

de semântica latente e de semântica referencial explícita. São 7 (sete) os agrupamentos identificados, entretanto apenas 3 (três) deles foram formados pela união de elementos:

C_1 :

prevayler
prevayler.implementation
prevayler.implementation.snapshot

C_2 :

prevayler.implementation.replication
prevayler.implementation.publishing
prevayler.foundation.network

C_3 :

prevayler.foundation
prevayler.foundation.serialization

Além dos agrupamentos, o diagrama mostrado na Figura 4.10 também é composto por 8 (oito) relacionamentos. Vale lembrar que os relacionamentos exibidos no diagrama foram

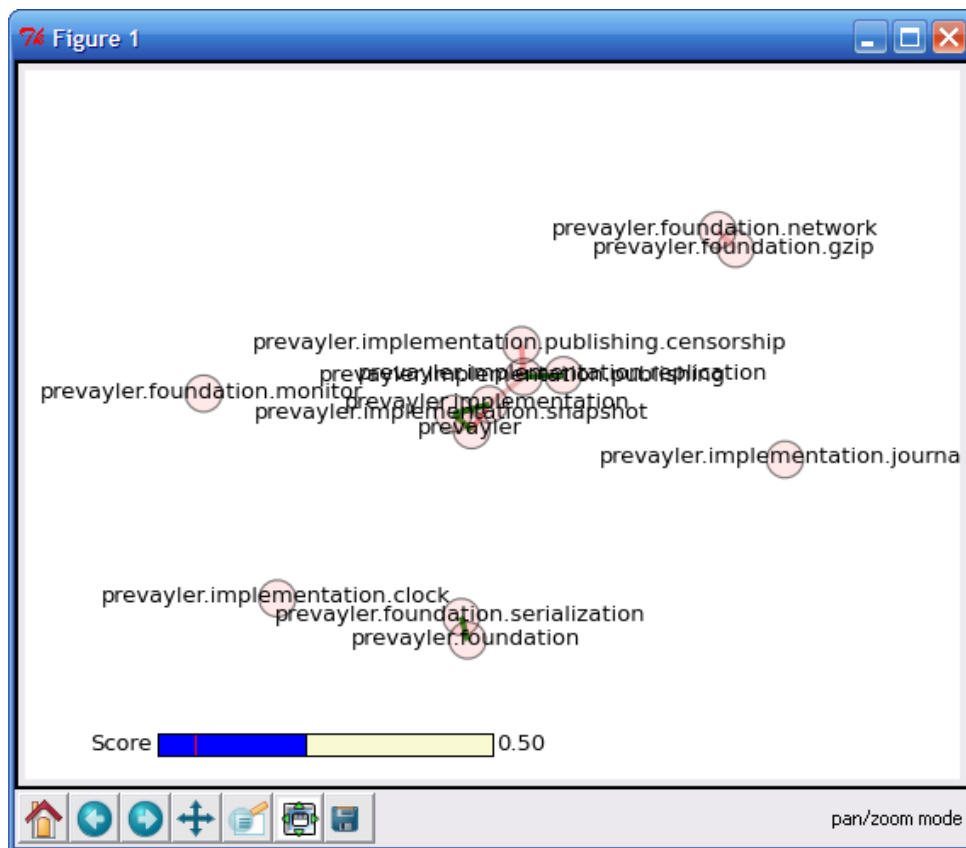


Figura 4.6: Grafo do *Prevayler* com fator de calibragem 0.5

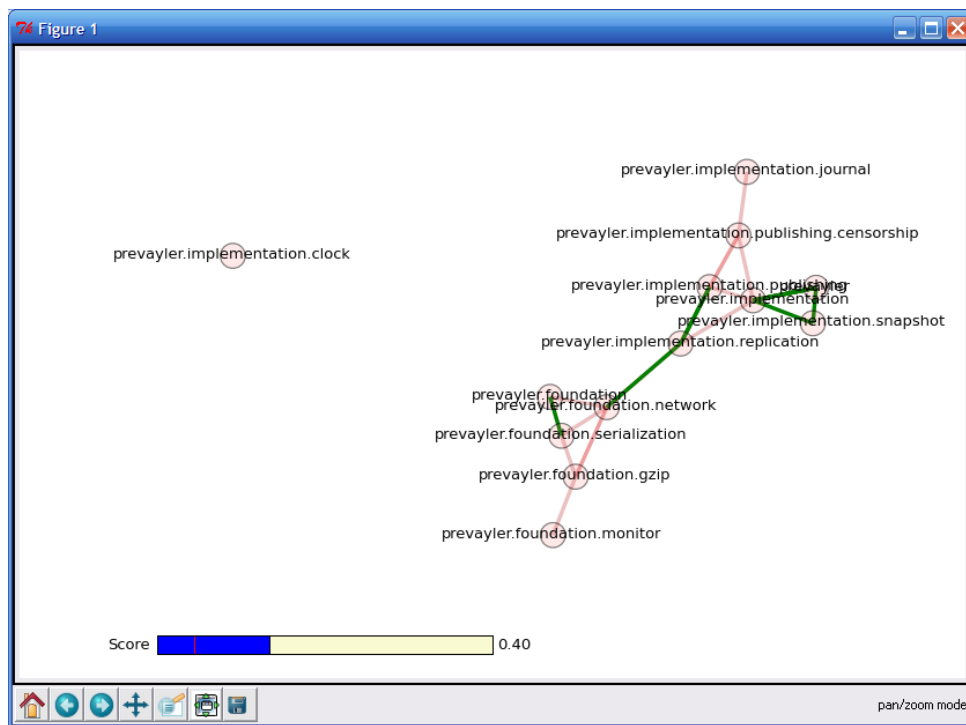
obtidos pela extração da semântica latente processada a partir do fator de calibragem 0.4 definido pelo especialista no início do estudo de caso.

Logo, a identificação dos agrupamentos consiste na união dos elementos conectados por arestas de cor verde, ou seja, pelo agrupamentos dos pacotes nos quais há tanto uma relação de semântica referencial explícita quanto uma relação de semântica latente.

4.3.2 Compilador do *OpenJDK*

O *OpenJDK* [Mic09b] foi o sistema alvo escolhido para ser o segundo estudo de caso. Como o processo de escolha do fator de calibragem já foi apresentado de forma detalhada na seção anterior, serão suprimidos a descrição de alguns passos do processo.

Em meados de 2007 a *Sun Microsystems* abriu todo o código de sua JVM (*Java Virtual Machine*) e JDK (*Java Development Kit*). Esta iniciativa deu início ao projeto *OpenJDK*, que tem a maior parte de seu código fonte disponível sob licença GNU GPLv2 (*General Public License Version 2*). Neste estudo de caso analisaremos apenas o compilador (*javac*) com pacote *OpenJDK*.

Figura 4.7: Grafo do *Prevayler* com fator de calibragem 0.4

Encontrando o Fator de Calibragem para o compilador do OpenJDK

O fator de calibragem para o estudo de caso do compilador do *OpenJDK* exigiu um maior esforço do especialista. Na Figura 4.11 comparamos 3 (três) grafos produzidos por fatores de calibragem distintos.

Após o estudo da documentação do compilador do *OpenJDK*, identificamos que uma importante funcionalidade deste sistema é o *parser* do compilador. Esta informação possibilitou concluir que o grafo produzido pelo fator de calibragem 0.65 não representa corretamente o compilador do *OpenJDK*, pois neste grafo não encontramos nenhum pacote que implemente o *parser*.

O grafo produzido pelo fator de calibragem 0.54 contém o pacote *tools.javac.parser*. Observamos também o surgimento de novas ligações entre alguns elementos, entretanto nenhuma grande alteração da topologia do grafo ocorreu. Contudo, com o fator de calibragem assumindo o valor 0.45, a quantidade de relações entre os pacotes passa a dificultar a compreensão do sistema alvo, pois se torna inviável a identificação de agrupamentos semanticamente interessantes.

Diante da análise feita pelo especialista através da comparação dos grafos produzidos com fatores de calibragem candidatos e utilizando informações obtidas a partir da documentação. O fator que viabiliza a compreensão do compilador do *OpenJDK* é 0.54. Cabe ressaltar que este fator pode não recuperar todas relações necessárias para o entendimento do sistema alvo.

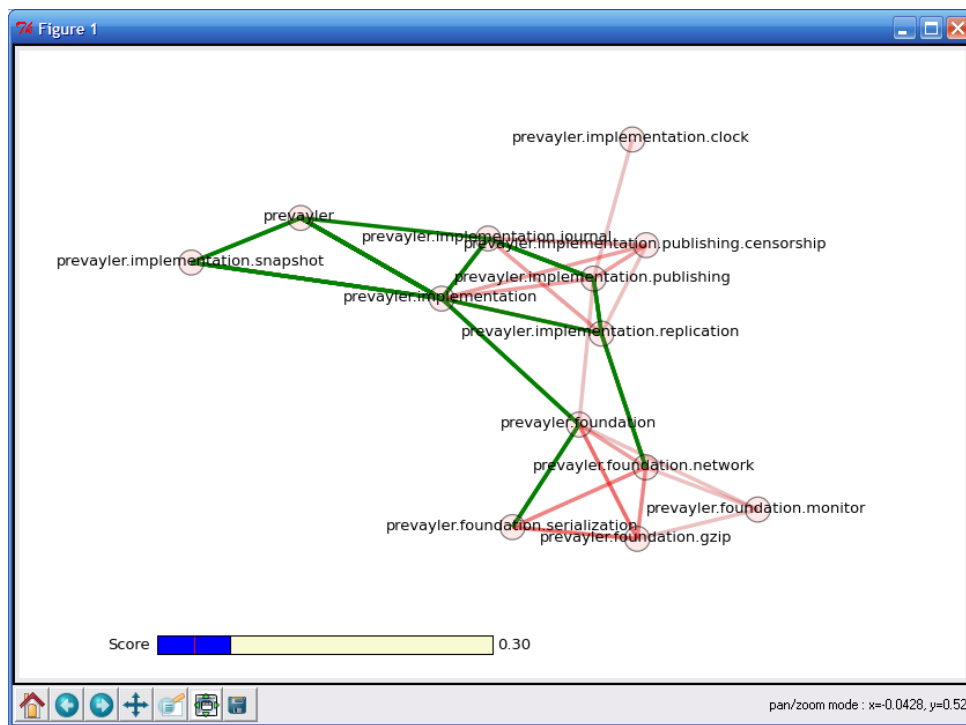


Figura 4.8: Grafo do *Prevayler* com fator de calibragem 0.3

Identificação dos Agrupamentos

A Figura 4.12 mostra como foram gerados os agrupamentos entre os pacotes do compilador do *OpenJDK*. O fator calibragem usado para produzir foi 0.54.

O grafo gerado contém um grande volume de pacotes conectados por arestas de cor verde. Sendo assim, a definição dos agrupamentos foi feita através de uma análise visual do grafo.

Como pode ser observado na Figura 4.12, foram identificados 7 agrupamentos pelo especialista. Após a análise dos pacotes que compõem cada agrupamento eles receberam os seguintes rótulos: 1. *model*; 2. *generate*; 3. *doclets*; 4. *jvm*; 5. *comp*; 6. *parser*; 7. *processing*.

Devido ao grande volume de informações deste estudo de caso, foi exigido do especialista um maior conhecimento sobre o domínio do problema. Esse conhecimento é fundamental para a escolha do fator de calibragem. Tal fator influencia diretamente nos agrupamentos encontrados.

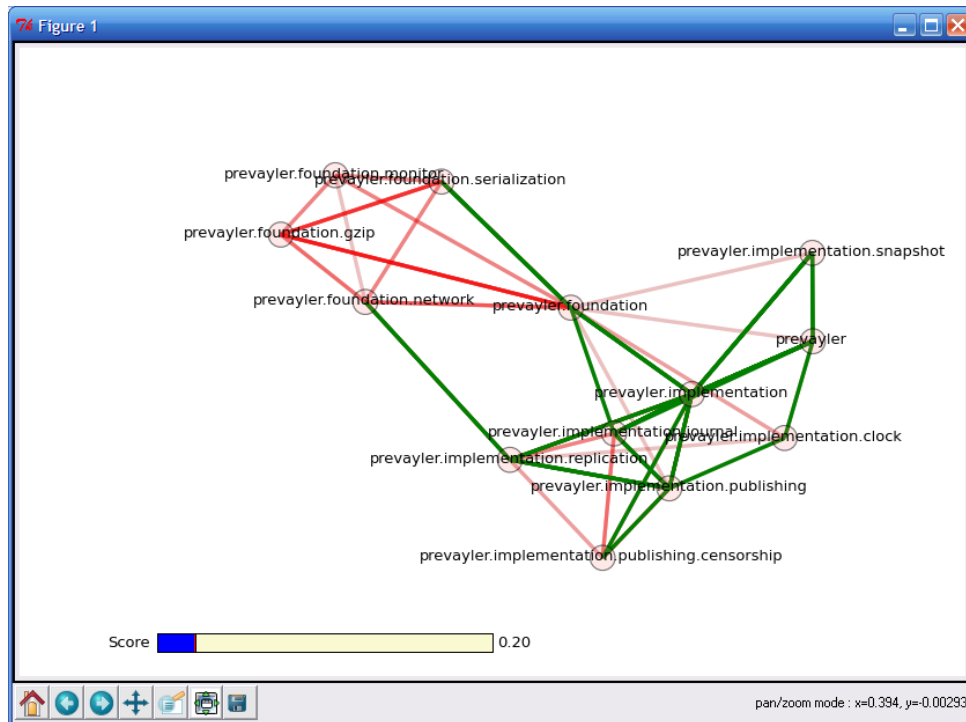


Figura 4.9: Grafo do *Prevayler* com fator de calibragem 0.2

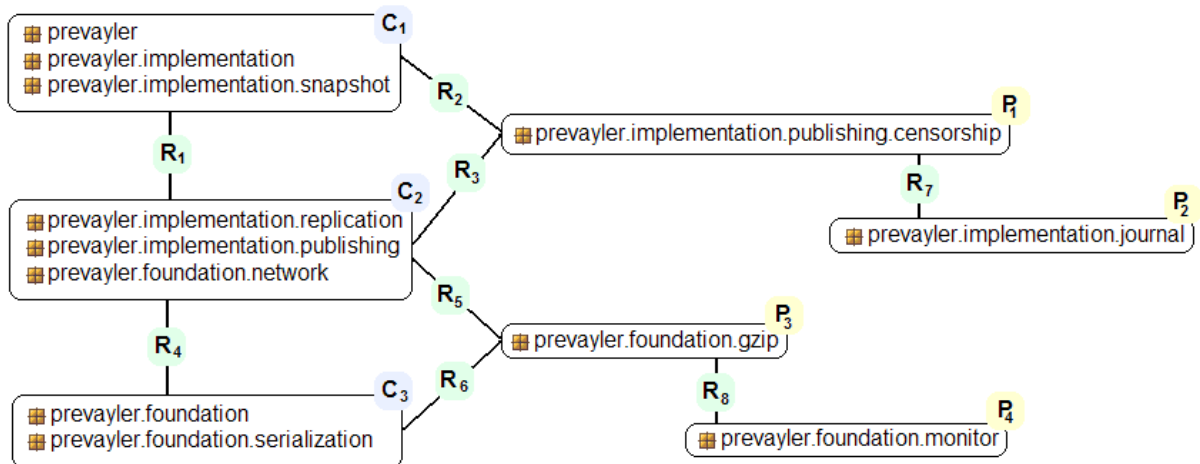
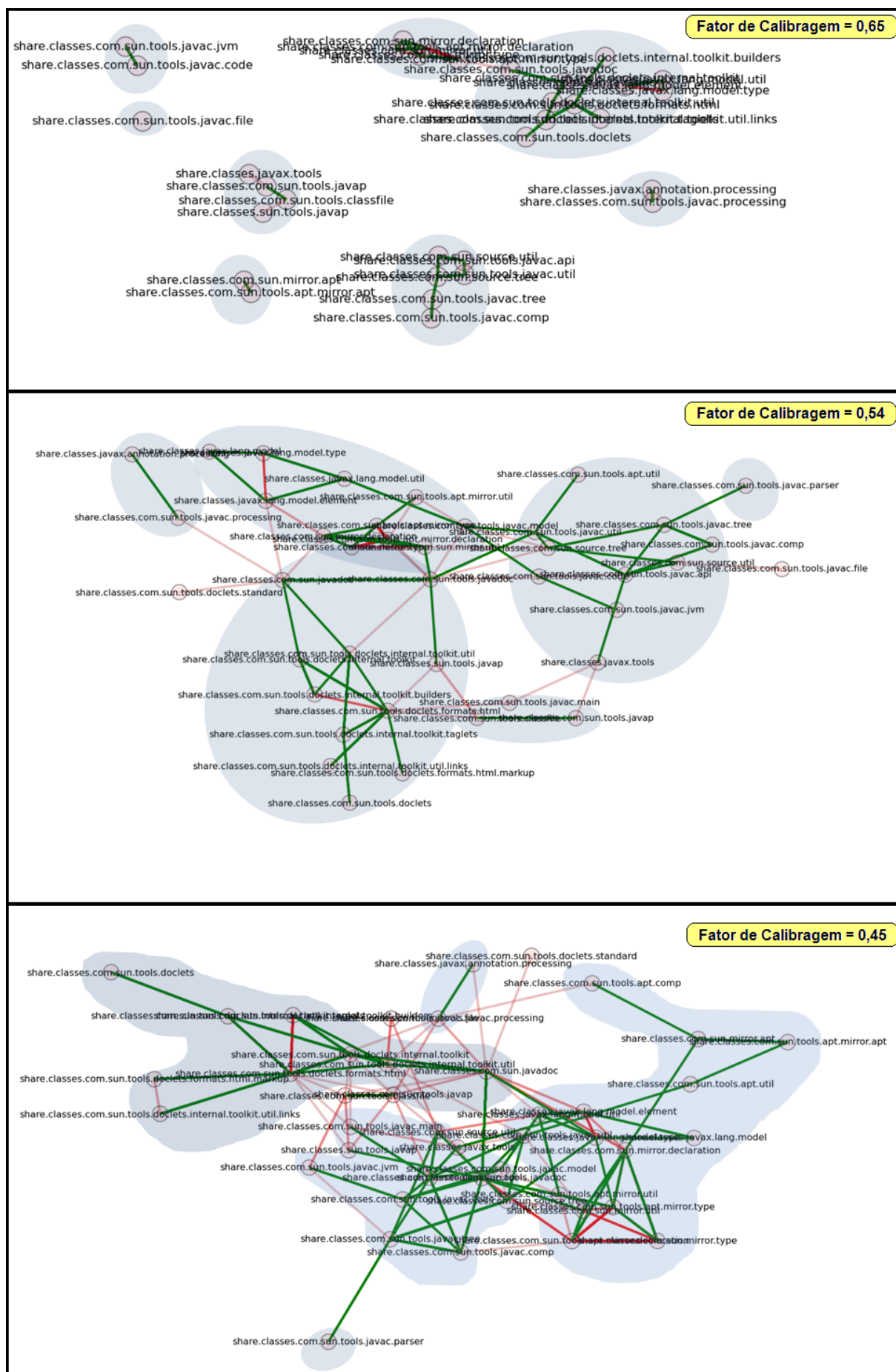


Figura 4.10: Agrupamentos identificados no *Prevayler* com fator de calibragem 0.4



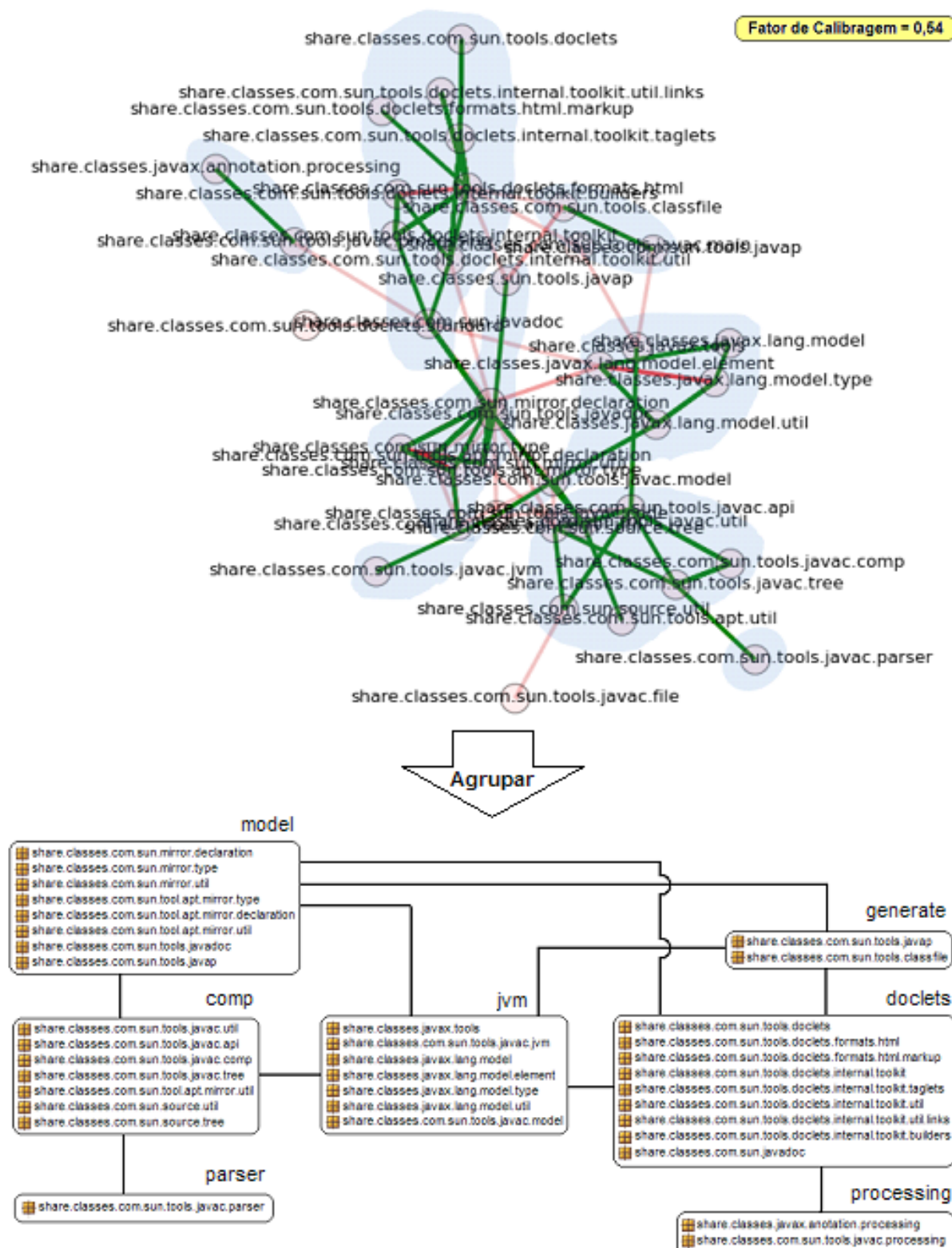


Figura 4.12: Grafo do compilador do *OpenJDK* com fator de calibragem 0.54

4.4 Análise dos Resultados Obtidos

O objetivo principal desses estudos de caso, definido na Seção 4.2, é avaliar a viabilidade da abordagem de recuperação de uma visão estrutural de um software desconhecido. Esta viabilidade é constatada respondendo aos questionamentos estabelecidos no estudo.

As respostas aos questionamentos foram fundamentadas na análise da documentação e do código fonte de cada sistema alvo. A visão estrutural recuperada foi avaliada sob a ótica de elementos arquiteturais que representem um conjunto de serviços coesos ou conceitos do domínio.

4.4.1 Precisão dos Agrupamentos Recuperados

Inicialmente vamos analisar o estudo de caso do *Prevayler*. Em sua documentação foram identificadas as seguintes características:

- Baseada-se na serialização de Transações;
- Primeiro serializa depois executa;
- Permite utilização de qualquer collection da *Java API*;
- Cada transação deve ser uma classe com um método execute;
- Não padroniza ou impõe qualquer constraint (restrição) do modelo de classes.

Deste modo, para fornecer uma camada de persistência de objetos em memória, o *Prevayler* mantém uma cópia primária dos objetos permanentemente em memória, sendo que em um sistema tradicional, tal cópia permanece em disco e blocos são temporariamente carregados para memória.

O *Prevayler* utiliza dois mecanismos para garantir a persistência dos objetos: um log de transação e um *backup*, ou *snapshot*, do estado dos objetos.

O *log* de transação é implementado pela utilização de classes que representam alterações em objetos. Para essas alterações persistirem, as classes são serializadas e gravadas em disco. Já o *snapshot* é responsável por salvar o estado de todos os objetos da memória principal para o disco, tirando uma fotografia do estado dos objetos em um dado momento.

Sendo assim, a serialização ocorre no *Prevayler* em dois momentos: no log, onde uma transação é salva; e em um snapshot, onde toda a memória (gerenciada pelo *Prevayler*) é despejada em um arquivo. Quando a aplicação reinicia, o *Prevayler* carrega o último snapshot, se houver, e executa todas as transações depois dele.

O diagrama resultante do processo de identificação de agrupamentos para este estudo de caso é apresentado na Figura 4.10. Nela identificamos três agrupamentos, C_1 , C_2 e C_3 . Com base na informações encontradas na documentação do *Prevayler* foi possível reconhecer nos pacotes que compõe cada agrupamento da Figura 4.10 os seguintes conceitos ou serviços:

- C_1 : *Backup*, ou *snapshot*.
- C_2 : Gestão do *log* das transações.
- C_3 : Armazenamento das informações em disco.

Portanto, no estudo de caso do *Prevayler* os agrupamentos recuperados pela abordagem, refletem os elementos conceituais do sistema alvo.

Os agrupamentos encontrados no estudo de caso do compilador do *OpenJDK* foram avaliados a partir da Figura 4.13, onde é apresentada a arquitetura em camadas deste software. A Figura 4.12 apresenta os 7 agrupamentos encontrados por nossa abordagem, dos quais 5 possuem componentes equivalentes em 4.13, *generate* e *doclets* não estão explicitamente representados na arquitetura. Entretanto, a arquitetura do compilador do *OpenJDK* é composta por 16 componentes, sendo assim, diversos componentes desta arquitetura não foram encontrados por nossa abordagem.

Contudo, ao diminuir o fator de calibragem buscando recuperar uma maior quantidade de componentes arquiteturais do compilador do *OpenJDK*, seriam produzidos também um elevado número de relações entre os pacotes, comprometendo assim a compreensão do grafo pelo especialista.

Cabe ressaltar que todos os componentes da camada *Compilation* foram identificados por nossa abordagem, e sendo esta a principal camada de um compilador, concluímos que os agrupamentos recuperados são corretos.

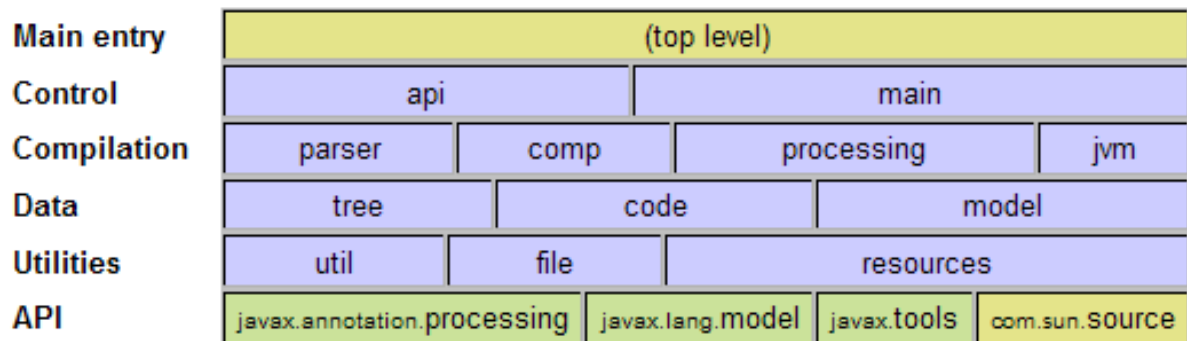


Figura 4.13: Arquitetura em camadas do compilador do *OpenJDK* [Mic09a]

Como visto, em ambos os estudos de caso foi possível relacionar os agrupamentos recuperados com conceitos ou serviços dos sistemas alvo.

4.4.2 Precisão das Relações Recuperadas entre Agrupamentos

Para responder a esta pergunta é necessário avaliar se o relacionamento entre agrupamentos é coerente com o relacionamento entre os conceitos identificados anteriormente. Por exemplo, no estudo de caso do *Prevayler* (Figura 4.10) a visão inicial recuperada indica a existência de uma relação R_1 entre os agrupamentos C_1 e C_2 , ou seja aponta

que existe uma relação entre os elementos arquiteturais de *Backup* e gestão do *log* das transações. Esta relação é coerente, visto que o Prevayler após carregar o último *snapshot*, executa todas as transações armazenadas no *log*.

No estudo de caso do compilador do *OpenJDK*, a arquitetura em camadas apresentada na Figura 4.13 não apresenta relações de dependências entre os componentes. Também não é possível extrair da visão recuperada na Figura 4.12 as relações entre camadas. Entretanto, as relações produzidas por nossa abordagem existem referencialmente em um contexto estático.

Sendo assim, observamos que as relações encontradas pela abordagem refletem relações presentes também no domínio do sistema alvo.

4.4.3 Apoio da Visão Recuperada na Compreensão

Para responder a esta pergunta, é necessário verificar se com o uso da abordagem proposta houve uma diminuição no esforço para se iniciar a compreensão do sistema alvo. Para isso, será comparada a quantidade de informação (relações e elementos) disponibilizada pela descoberta das relações de semântica referencial explícita, com a disponibilizada através de nossa abordagem.

Para verificar essa redução no esforço, o volume de informação apresentado no grafo gerado no estudo de caso do compilador do *OpenJDK* (Figura 4.12) é comparado com o volume de informação apresentado no grafo gerado apenas pela análise semântica referencial explícita (Figura 4.14).

A abordagem proposta se mostrou eficiente na diminuição do esforço, pois a extração da semântica referencial explícita apresentou uma quantidade de informação (49 elementos e 378 relações) superior a nossa abordagem (7 elementos e 10 relações). Sendo assim, há uma redução significativa do volume de informações a serem analisadas pelos desenvolvedores.

A redução ocorreu devido a seleção dos relacionamentos, foram podados os relacionamentos de acordo com o fator de calibragem ou que estão presentes apenas na semântica referencial explícita. O agrupamento de alguns elementos (pacotes do sistema alvo) também foi responsável por diminuir o volume de informação. Logo, a visão estrutural recuperada restringe a exibição das informações do sistema alvo, possibilitando que o desenvolvedor trabalhe um nível de abstração maior.

Importa ressaltar que a visão estrutural recuperada não tem o objetivo de automatizar a compreensão completa do software, mas sim possibilitar seu início. Portanto, a abordagem reduz o trabalho manual da compreensão de um software desconhecido.

Diante disso, constata-se que a visão inicial recuperada apóia a compreensão da aplicação visto que reduz o volume de informação a ser compreendido pelo desenvolvedor.

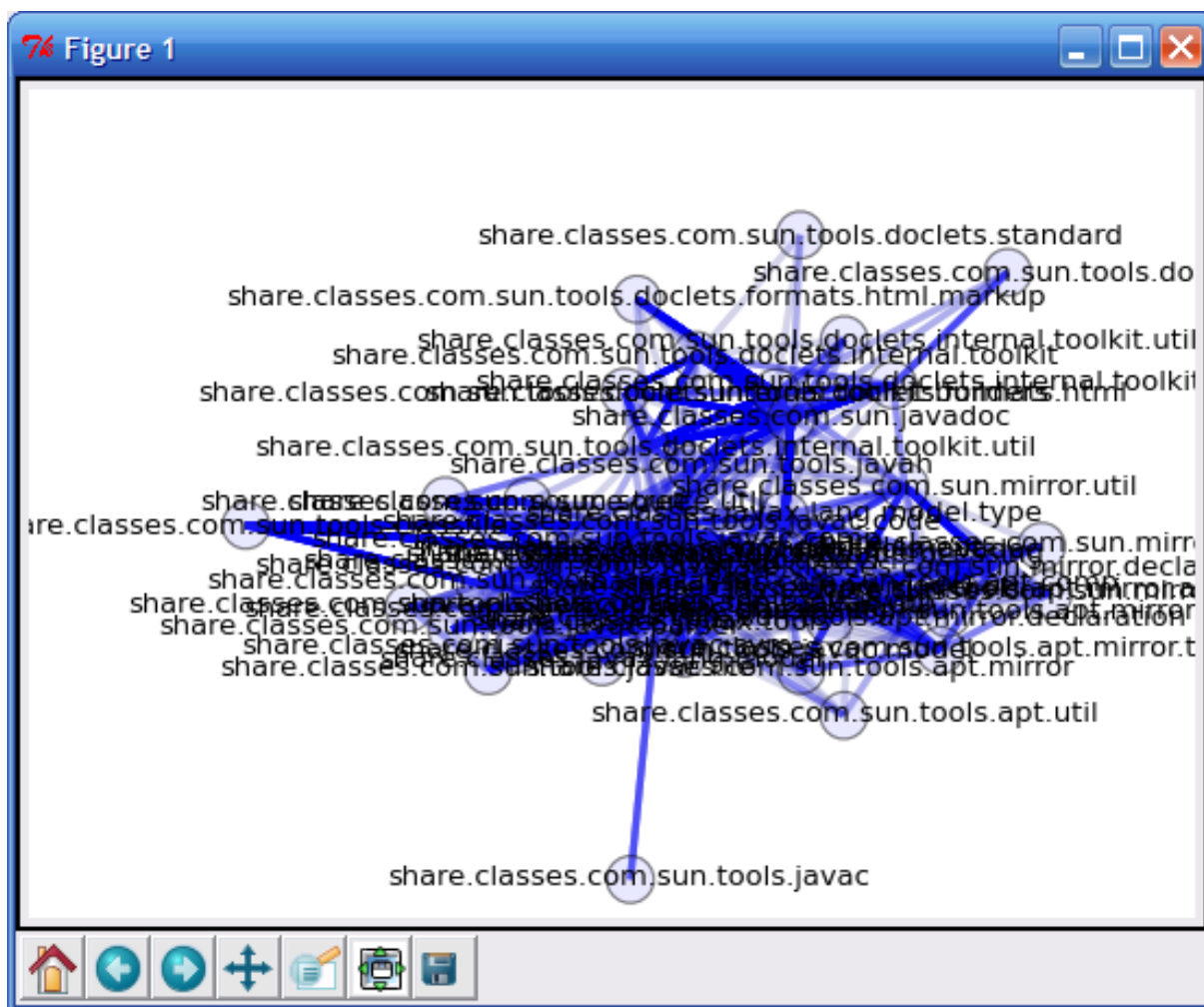


Figura 4.14: Relações de semântica referencial explícita do compilador OpenJDK

4.4.4 Análise do Desempenho das Ferramentas

A análise do desempenho da ferramenta foi feita em um computador com processador com 1.6 *GHz* e 896 *MegaBytes* de memória RAM. Os resultados são apresentados na Tabela 4.3, as métricas extraídas são:

1. TLOC: número total de linhas de código fonte,
2. TCLSI: tempo gasto (segundos) para realizar o cálculo do LSI e determinar as relações entre os artefatos,
3. TCG: tempo gasto (segundos) para gerar e visualizar um grafo.

Como vemos, a fase com maior custo operacional é a geração da relações entre os artefatos. Nossa ferramenta armazena o resultado desta fase em disco, de modo que este cálculo é realizado apenas uma vez para cada sistema alvo. A geração e visualização de grafos será repetida inúmeras vezes durante a definição do fator de calibragem, sendo assim, mesmo tendo um custo razoavelmente baixo esta fase deve ser melhorada nas próximas versões da ferramenta, pois é um fator limitante para um sistema interativo.

Observamos também que desempenho é proporcional ao total de linhas de código de cada sistema alvo.

<i>SistemaAlvo</i>	<i>TLOC</i>	<i>TCLSI</i>	<i>TCG</i>
Prevayler	2.974	5,22	0,53
Compilador do OpenJDK	86.106	449,27	72,14

Tabela 4.3: Análise do desempenho da ferramenta

4.5 Conclusão

Este capítulo apresentou dois estudos de caso onde uma visão estrutural de um sistema desconhecido pode ser encontrada de forma simples com o uso da abordagem proposta neste trabalho, a visão recuperada é uma ajuda valiosa para a compreensão de tais sistemas.

Capítulo 5

Trabalhos Relacionados

Muitos aspectos deste trabalho foram inspirados por outros trabalhos em compreensão de software. Destacando pesquisas que aplicavam técnicas de Recuperação de Informação - RI [Dum91] na Engenharia de Software.

Diversos trabalhos propostos na literatura [AL99, ACN02, Mit06, Sar03, SAG⁺06] buscam obter conhecimentos sobre um software por meio da recuperação de arquitetura de software. Estas abordagens tem o objetivo de representar visões arquiteturais, extraíndo diferentes elementos arquiteturais. Em sua maioria as abordagens recuperam uma visão estática da arquitetura. Nossa abordagem também utiliza a análise estática na recuperação de uma visão estrutural do software, porém não é nosso objetivo obter uma representação arquitetural completa do sistemas alvo.

Outro meio de axiliar a compreensão de um programa é apresentado por [Ant02]. Este trabalho avalia dois métodos baseados em modelos de RI, probabilístico e o modelo de espaço vetorial. O estudo confronta os dois métodos utilizando dois estudos de caso, os resultados mostram que o desempenho de ambos é similar, desde que seja feita uma análise morfológica preliminar (*stemming*) sobre termos contidos nos artefatos. Um dos pilares de nossa abordagem é o uso de técnicas de RI na compreensão de programas, deste modo, o tratamento preliminar dos termos se mostra fundamental, pois em geral não há uma grande diversidade de vocabulário nos artefatos. Logo, é necessário o uso de técnicas como o *stemming* para aumentar a abrangência do processo de recuperação.

As técnicas de RI podem ser empregadas em diferentes artefatos de software. [AC-CdL00] opta pelo espaço vetorial para recuperar relações entre as requisições de manutenção e os documentos de software impactados. Já [Hay04] utiliza desse mesmo modelo para encontrar a relações entre requisitos. Outra contribuição desse autor foi sugerir como forma de melhorar o desempenho o uso das palavras chaves dos documentos. Como em geral, os artefatos de software são compostos de documentos textuais, é possível aplicar as técnicas de RI na maioria dos artefatos. Entretanto, como buscamos compreender um software que não possua documentação, nossa abordagem irá analisar apenas os artefatos de código fonte.

O conhecimento semântico reunido pela equipe de desenvolvimento pode contribuir no processo de recuperação de informação sobre o sistema alvo. Fica mais claro esse ganho no trabalho de [CH05] onde se propôs melhorar o desempenho do processo incorporando três diferentes estratégias: modelagem hierárquica da nomenclatura; clusterização lógica dos artefatos; e poda semi-automática na rede probabilística. Todas elas agregam semântica ao modelo probabilístico. Assim como o trabalho de [CH05], nossa abordagem valoriza uso do conhecimento semântico reunido pela equipe de desenvolvimento, visto que este conhecimento é fundamental na definição do fator de calibragem utilizado para restringir a exibição das relações semânticas e também na identificação e interpretação dos agrupamentos.

Outra aplicação da semântica na recuperação de informação sobre os artefatos de software, é apresentada por [NR07], onde se propõe a utilização da descrição ontologia para especificação formal das relações entre as entidades de software, seu modelo é incorporado a um modelador UML.

O trabalho de [MM03] utiliza de estudos de casos similares a [Ant02] para comparar o desempenho do método de Indexação Semântica Latente - LSI [DDL⁺90] com métodos baseados no modelo de espaço vetorial e probabilístico. Este trabalho demonstrou que LSI tem um desempenho superior, porém é necessário o uso de técnicas de *stemming*.

A pesquisa de [LFOT04] também aplica LSI para recuperar relações entre diferentes tipos de artefatos. Posteriormente em [DL05] essa técnica é incorporada a ferramenta de manutenção de rastreabilidade ADAMS [LFFT04].

Assim como os trabalhos de [MM03,DL05], nossa abordagem faz uso do método de LSI para recuperar as relações de semântica latente entre os artefatos, entretanto em nossa abordagem, os resultados do LSI são combinados aos resultados da análise estática a fim de construir uma visão estrutural do software.

Apesar da aplicação de RI em Engenharia de Software ter alcançado resultados significativos, pesquisas como [LFOT06] argumentam que é impraticável encontrar todas as relações corretas. Neste mesmo trabalho se propõe uma abordagem incremental na recuperação da rastreabilidade. Ainda segundo [LFOT07] falhas nas descrições dos artefatos resultam em deficiências no processo de descoberta das relações. Entretanto, ao se combinar as relações obtidas pelo método LSI com as relações encontradas através da análise estática, nossa abordagem busca aumentar a quantidade de relações corretas recuperadas, porém as falhas nas descrições dos artefatos, discutidas em [LFOT07], ainda influenciam negativamente na qualidade dos resultados da abordagem.

A compreensão de programas também é o foco do trabalho de [SMADP06] que faz uso de uma análise dinâmica conduzida por roteiros de uso, e busca produzir visões em forma de grafos e métricas de similaridade entre os casos de uso. Por serem obtidos a partir da análise dinâmica, os grafos gerados se referem as relações em tempo de execução entre entidades em diferentes níveis de abstração (metodos, classes, módulos e casos de uso).

Já as métricas indicam o grau de similaridade dos casos de uso em termos de elementos compartilhados na sua implementação. Este trabalho também se preocupa com volume de informação que é produzido pela grande quantidade de elementos e relações presente nos sistemas de grande porte, para resolver tal problema é feita uma filtragem dos elementos.

Além de apoiar a compreensão de software, a representação das relações entre os elementos do software na forma de um grafo é outro aspecto comum entre o trabalho de [SMADP06] e o nosso. Entretanto, os trabalhos produzem visões distintas, pois nossa abordagem recupera uma visão estrutural, já [SMADP06] utiliza a análise dinâmica para recuperar a visão comportamental do software.

O trabalho de [RHRLR08] combina as técnicas de análise estática e dinâmica. Esta abordagem produz um rastro de execução (análise dinâmica) que é ordenado com o uso de um grafo de dependências (análise estática). A abordagem afirma que ao se medir o impacto da modificação de um componente no resto do sistema é possível identificar as características do sistema alvo. A combinação de técnicas distintas também é utilizada no trabalho de [EKS03]. Esta pesquisa utiliza a técnica de análise dinâmica combinada com a Análise Formal de Conceitos para localizar as características e explorar as relações entre conjuntos de características e conjunto de artefatos de código fonte.

Nossa abordagem assim como os trabalhos [RHRLR08] e [EKS03] utiliza do artifício da união de técnicas distintas. Tal estratégia visa melhorar os resultados da compreensão de software, de forma que uma técnica possa validar a outra.

Como visto, resultados significativos têm sido alcançados por autores que aplicam modelos baseados em RI, em especial a técnica LSI, na recuperação de conhecimento sobre artefatos de software. Além disso, observamos que bons resultados tem sido obtidos por abordagens que combinem técnicas distintas na análise dos artefatos de software.

Capítulo 6

Conclusões e Perspectivas

Este capítulo apresenta as considerações finais a respeito da pesquisa desenvolvida, bem como as frentes de trabalhos futuros.

A abordagem de apoio à compreensão de programas apresentada neste trabalho, mostrou como a técnica de recuperação de informação LSI pode ser combinada a semântica referencial explícita na construção de uma visão estrutural do software. Esta abordagem se mostrou útil ao restringir o volume de informação sobre o software apresentado ao desenvolvedor.

Sendo assim, nossa abordagem auxilia na tarefa de compreensão de software, pois apresenta uma visão do software em um nível maior de abstração. Logo, por trabalhar com um volume menor de informações, ela diminui o esforço para se compreender um software em comparação com a simples análise do código e documentação.

O trabalho definiu, ainda, uma metodologia para aplicação da abordagem, onde também foram discutidos alguns detalhes sobre sua arquitetura e implementação.

Com base na metodologia definida, dois estudos de caso foram conduzidos. Os estudos mostraram resultados positivos, visto que a organização dos elementos e relações apresentadas pela visão estrutural recuperada pode ser comprovada pelo especialista através de uma análise da documentação e do código fonte. Houve também uma diminuição no volume de informação e quantidade de elementos exibidos em relação a análise da semântica referencial explícita do código fonte.

Diante do desafio de se compreender um sistema legado, uma vez que normalmente estes sistemas costumam ser complexos, mal projetados e não possuem documentação de apoio atualizada, nossa abordagem apoia a tarefa de compreensão de software através de um processo semi-automático que reduz a quantidade de elementos e relações a serem analisadas, resultando assim na diminuição do esforço desta tarefa.

6.1 Contribuições

Diante dos pontos discutidos, é possível apontar que a principal contribuição do trabalho consiste na implementação de uma abordagem com baixo esforço de execução na recuperação de uma visão estrutural de um software.

Outras contribuições deste trabalho foram:

1. Apoio à tarefa de compreensão de software. A visão estrutural obtida pela abordagem apoia atividade de obtenção de conhecimento sobre um software desconhecido, visto que reduz o volume de informações a serem analisadas pelo desenvolvedor.
2. Utilização do conhecimento já adquirido pela equipe de desenvolvimento. O fato da visão estrutural ser gerada a partir de um fator de calibragem, possibilita que diferentes visões sejam produzidas dependendo do nível de conhecimento sobre o software já adquirido pela equipe de desenvolvimento. Tal conhecimento também é fundamental para a identificação dos agrupamentos entre artefatos de código fonte. Sendo assim, o uso do conhecimento prévio sobre o domínio do software caracteriza-se como contribuição deste trabalho.
3. Aplicação da abordagem em sistemas reais. Também foram conduzidos dois estudos de caso, onde se constatou uma conformidade entre a visão estrutural recuperada e a documentação do sistema alvo, tal fato indica a viabilidade do uso da abordagem com ponto de partida para o processo de compreensão de software.

6.2 Limitações Identificadas

Algumas limitações que identificou-se neste trabalho são:

1. Qualidade do vocabulário contido no código fonte. Este vocabulário tem influência direta nos agrupamentos gerados pela abordagem proposta. Portanto, a inexistência de comentários nos artefatos de códigos fonte, ou o uso de identificadores com pouco significado semântico, interferem negativamente na formação dos agrupamentos. Isto ocorre porque a linha mestra da abordagem são relações encontradas através da técnica LSI. Sendo assim, o vocabulário presente no código fonte é fundamental, visto que ele é a principal matéria prima para o processo de recuperação da visão estrutural.
2. Conhecimento sobre o domínio. A abordagem é dependente também do conhecimento prévio que o desenvolvedor detém sobre o domínio em que o sistema alvo esta inserido, visto que tal conhecimento será fundamental na determinação do fator de calibragem e nas análises dos agrupamentos e relações encontradas. Sendo que em sistemas maiores a atividade de identificação dos agrupamentos exige uma carga significativa de conhecimento prévio sobre o domínio.

3. Representação da visão estrutural. Nossa abordagem utiliza os grafos como meio de exibir os resultados, porém esta estratégia ainda não é ideal, visto que os grafos não transmitem o significado das relações e elementos. Entretanto, não há um padrão para representação da visão estrutural do software, tornando necessário que a equipe de desenvolvimento utilize recursos auxiliares para registrar o conhecimentos recuperado pela abordagem.
4. Anomalias no código fonte. Em um software legado, o código pode conter várias anomalias, como partes do programa que nunca podem ser executadas (código morto) e trechos de código que foram copiados e levemente modificados. Estas anomalias diminuem a qualidade da visão estrutural recuperada pela abordagem.

6.3 Trabalhos Futuros

Buscando-se melhorar e expandir a abordagem proposta, observa-se que algumas perguntas ainda permanecem para trabalhos futuros, podendo assim proporcionar importantes pesquisas:

1. Avaliar a escalabilidade da abordagem. Outros experimentos precisam ser executados com sistemas maiores e/ou utilizando diferentes tipos de artefatos, a fim de considerar a escalabilidade da abordagem e da ferramenta.
2. Alterar a forma de representar a visão estrutural recuperada. Vislumbramos também, a necessidade de definir uma metodologia para representação da visão recuperada pela abordagem, sendo que o uso de ontologias [MS01] pode resultar em uma evolução deste trabalho.
3. Identificar os agrupamentos de forma automática. É importante reduzir o esforço gasto para se identificar os agrupamentos entre os elementos do grafo produzido por nossa abordagem. Esta redução pode ser alcançada com a aplicação de técnicas automáticas de agrupamentos, tal uso representaria uma melhoria funcional na abordagem.
4. Aplicar a análise dinâmica em conjunto com nossa abordagem. A análise dinâmica iria apoiar nossa abordagem, pois permitiria relacionar variações de entrada, relativas a uma determinada funcionalidade, a visão estrutural recuperada.

Referências Bibliográficas

- [ACCdL00] G. Antoniol, G. Canfora, G. Casazza, and A. de Lucia. Identifying the starting impact set of a maintenance request: A case study. *CSMR '00: Proceedings of the Conference on Software Maintenance and Reengineering*, page 227, 2000.
- [ACN02] Jonathan Aldrich, Craig Chambers, and David Notkin. Archjava: connecting software architecture to implementation. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 187–197, New York, NY, USA, 2002. ACM.
- [AL99] Nicolas Anquetil and Timothy C. Lethbridge. Recovering software architecture from the names of source files. *Journal of Software Maintenance*, 11(3):201–221, 1999.
- [And05] Information-theoretic software clustering. *IEEE Trans. Softw. Eng.*, 31(2):150–165, 2005. Member-Andritsos, Periklis and Member-Tzerpos, Vasilios.
- [Ant02] G.; Casazza G.; De Lucia A.; Merlo E. Antoniol, G.; Canfora. Recovering traceability links between code and documentation. *Transactions on Software Engineering*, 28(10):970–983, 2002.
- [BCR94] V. Basili, G. Caldiera, and H.D. Rombach. Goal question metric approach. In *Encyclopedia of Software Engineering*, pages 528–532. John Wiley & Sons, Inc., 1994.
- [BDR96] L.C. Briand, C. Differding, and H.D. Rombach. Practical guidelines for measurement-based process improvement. ISCN, 1996.
- [BMR⁺96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons, August 1996.
- [BRJ05] Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide, The (2nd Edition) (The Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, May 2005.
- [BYRN99] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley, May 1999.

- [Cam97] Marcelo Ricardo Campos. *Compreensão Visual de Frameworks através da Introspecção de Exemplos*. Tese (doutorado em ciência da computação), Universidade Federal do Rio Grande do Sul, 1997.
- [CH05] R.; Chuan Duan; Xuchang Zou Cleland-Huang, J.; Settimi. Utilizing supporting evidence to improve dynamic requirements traceability. *Requirements Engineering, 2005. Proceedings. 13th IEEE International Conference on*, pages 135–144, 29 Aug.-2 Sept. 2005.
- [DDL⁺90] Scott C. Deerwester, Susan T. Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society of Information Science*, 41(6):391–407, 1990.
- [DdSK03] Anquetil N. Damian de Sousa K., Oliveira K.M. de. Aplicação de GQM para avaliação de processo de manutenção de software. In *Jornadas Iberoamericanas de Ingeniería del Software e Ingeniería del Conocimiento*, Valdivia, Chile, 2003.
- [DL05] F.; Oliveto R.; Tortora G. De Lucia, A.; Fasano. Adams re-trace: A traceability recovery tool. *Software Maintenance and Reengineering, 2005. CSMR 2005. Ninth European Conference on*, pages 32–41, 21-23 March 2005.
- [DT98] Alastair Dunsmore and Livingstone Tower. Comprehension and visualisation of object-oriented code for inspections. Technical Report EFoCS-33-98, University of Strathclyde, Glasgow, UK, 1998.
- [Dum91] S. T. Dumais. Improving the retrieval of information from external sources. *Behavior Research Methods, Instruments and Computers*, 23(2):229–236, 1991.
- [EKS03] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating features in source code. *IEEE Trans. Softw. Eng.*, 29(3):210–224, 2003.
- [GR70] G. Golub and C. Reinsch. Singular value decomposition and least squares solutions. *Numerische Mathematik*, 14(5):403–420, April 1970.
- [GT03] Penny Grubb and Armstrong A. Takang. *Software Maintenance: Concepts and Practice*. World Scientific Publishing Company, second edition, September 2003.
- [Hay04] A.; Sundaram S.K.; Howard S. Hayes, J.H.; Dekhtyar. Helping analysts trace requirements: an objective look. *Requirements Engineering Conference, 2004. Proceedings. 12th IEEE International*, pages 249–259, 6-11 Sept. 2004.
- [HSS08] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using NetworkX. pages 11–15, Pasadena, CA USA, August 2008.
- [JOP⁺01] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001. <http://www.scipy.org/>.

- [LFFT04] Andrea De Lucia, Fausto Fasano, Rita Francese, and Genoveffa Tortora. Adams: an artefact-based process support system. *SEKE*, pages 31–36, 2004.
- [LFOT04] Andrea De Lucia, Fausto Fasano, Rocco Oliveto, and Genoveffa Tortora. Enhancing an artefact management system with traceability recovery features. *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 306–315, 2004.
- [LFOT06] Andrea De Lucia, Fausto Fasano, Rocco Oliveto, and Genoveffa Tortora. Can information retrieval techniques effectively support traceability link recovery? *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 307–316, 2006.
- [LFOT07] Andrea De Lucia, Fausto Fasano, Rocco Oliveto, and Genoveffa Tortora. Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Trans. Softw. Eng. Methodol.*, 16(4):13, 2007.
- [Mat09] Matplotlib. *Matplotlib documentation, 0.98.6svn*, 2009.
- [MB97] Yasuhiro Mashiko and Victor R. Basili. Using the GQM paradigm to investigate influential factors for software process improvement. *J. Syst. Softw.*, 36(1):17–32, 1997.
- [MB07] Onaiza Maqbool and Haroon Babri. Hierarchical clustering for software architecture recovery. *IEEE Transactions on Software Engineering*, 33(11):759–780, 2007.
- [Mic09a] Sun Microsystems. The java programming-language compiler (javac) group, 2009. <http://openjdk.java.net/groups/compiler/> : Visitada em Agosto de 2009.
- [Mic09b] Sun Microsystems. Openjdk, 2009. <http://openjdk.java.net> : Visitada em Agosto de 2009.
- [Mic09c] Sun Microsystems. Trail: The reflection api, 2009. <http://java.sun.com/docs/books/tutorial/reflect/>.
- [Mit06] On the automatic modularization of software systems using the bunch tool. *IEEE Trans. Softw. Eng.*, 32(3):193–208, 2006. Member-Mitchell, Brian S. and Senior Member-Mancoridis, Spiros.
- [MM03] Andrian Marcus and Jonathan I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 125–135, 2003.
- [MMR⁺98] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *IWPC '98: Proceedings of the 6th International Workshop on Program Comprehension*, page 45, Washington, DC, USA, 1998. IEEE Computer Society.

- [MS01] Alexander Maedche and Stefen Staab. Learning ontologies for the semantic web. *Proceedings of the Second International Workshop on the Semantic Web*, 2001.
- [Nor92] Stephen C. North. NEATO user's guide. Technical Report 59113-921014-14TM, AT&T Bell Laboratories, Murray Hill, NJ, USA, October 1992.
- [NR07] Rodrigo Perozzo Noll and Marcelo Blois Ribeiro. Ontological traceability over the unified process. In *ECBS '07: Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, pages 249–255, Washington, DC, USA, 2007. IEEE Computer Society.
- [PDP⁺07] Damien Pollet, Stephane Ducasse, Loic Poyet, Ilham Alloui, Sorana Cimpan, and Herve Verjus. Towards a process-oriented software architecture reconstruction taxonomy. In *CSMR '07: Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, pages 137–148, Washington, DC, USA, 2007. IEEE Computer Society.
- [Por09] M. F. Porter. Java implementation of porter's algorithm, 2009. Porter Stemming Algorithm: <http://www.tartarus.org/>.
- [PRW04] Michael J. Pacione, Marc Roper, and Murray Wood. A novel software visualisation model to support software comprehension. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering*, pages 70–79, Washington, DC, USA, 2004. IEEE Computer Society.
- [RHLR08] Abhishek Rohatgi, Abdelwahab Hamou-Lhadj, and Juergen Rilling. An approach for mapping features to code based on static and dynamic analysis. In *ICPC '08: Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, pages 236–241, Washington, DC, USA, 2008. IEEE Computer Society.
- [RT01] Barbara G. Ryder and Frank Tip. Change impact analysis for object-oriented programs. In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 46–53, New York, NY, USA, 2001. ACM.
- [SAG⁺06] Bradley Schmerl, Jonathan Aldrich, David Garlan, Rick Kazman, and Hong Yan. Discovering architectures from running systems. *IEEE Trans. Softw. Eng.*, 32(7):454–466, 2006.
- [Sar03] Kamran Sartipi. *Software architecture recovery based on pattern matching*. PhD thesis, Waterloo, Ont., Canada, Canada, 2003. Adviser-Kontogiannis, Kostas.
- [Sau09] Frank Sauer. Metrics 1.3.6, 2009. <http://metrics.sourceforge.net/> : Visitada em Agosto de 2009.
- [SMADP06] Maher Salah, Spiros Mancoridis, Giuliano Antoniol, and Massimiliano Di Penta. Scenario-driven dynamic analysis for comprehending large software systems. pages 71–80, 2006.

-
- [vMV95] Anneliese von Mayrhauser and A. Marie Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.
- [Wue09] Klaus Wuestefeld. Prevayler - the open source prevalence layer, 2009. <http://www.prevayler.org> : Visitada em Agosto de 2009.
- [ZZL⁺04] Wei Zhao, Lu Zhang, Yin Liu, Jiasu Sun, and Fuqing Yang. Sniafl: Towards a static non-interactive approach to feature location. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 293–303, Washington, DC, USA, 2004. IEEE Computer Society.