Universidade Federal do Pampa

Renan Marcel Uchôa

Um Modelo para Análise da Árvore de Sintaxe Abstrata como apoio à Extração de Fatos e a Compreensão de Software

Alegrete

2013

Renan Marcel Uchôa

Um Modelo para Análise da Árvore de Sintaxe Abstrata como apoio à Extração de Fatos e a Compreensão de Software

Trabalho de Conclusão de Curso , apresentado ao Curso de Graduação em Engenharia de Software da Universidade Federal do Pampa como requisito parcial para a obtenção do título de Bacharel em Engenharia de Software.

Orientador: Me. João Pablo Silva da Silva

Alegrete

2013

Renan Marcel Uchôa

Um Modelo para Análise da Árvore de Sintaxe Abstrata como apoio à Extração de Fatos e a Compreensão de Software

Trabalho de Conclusão de Curso , apresentado ao Curso de Graduação em Engenharia de Software da Universidade Federal do Pampa como requisito parcial para a obtenção do título de Bacharel em Engenharia de Software.

Trabalho de Conclusão de Curso , defendido e aprovado em 28 de Março de 2014

Banca examinadora:

Me. João Pablo Silva da Silva
Orientador

Dr. Márcia Cristina Cera
(Unipampa)

Ma. Andréa Sabreda Bordin
(Unipampa)

Este trabalho é dedicado ao meu paizinho querido, El Shaddai,
o criador dos céus e da terra, que me formou, e me chamou,
e me sustenta, que cuida de cada uma das minhas necessidades,
e hoje está me dando a graça de realizar mais este sonho.

Agradecimentos

Agradeço a Deus por estar comigo, e me dar esperança a cada dia, para enfrentar as minhas próprias limitações e avançar em direção a projetos ainda mais nobres e maiores que estes. Agradeço aos meus pais que investiram na minha formação acadêmica, e não me deixaram desistir, mesmo ante os inúmeros desafios que se apresentaram. Agradeço à Fernanda, minha noiva, que me suportou durante esses tempos de intenso trabalho, e que não perdeu as esperanças em todos os planos que fizemos juntos. Agradeço aos meus irmãos em Cristo, com quem tenho aprendido a ser alguém melhor, e que existem sonhos maiores para se sonhar, sonhos eternos duradouros.

Agradeço aos meus colegas e amigos, que dividiram experiências e aprendizados durante todo esse período de graduação. Agradeço ao meu orientador que me suportou durante todo esse ano, ajudando-me a administrar os prazos e ensinando-me os valores da dedicação e do empenho. Agradeço ao colega Fábio Petrillo, com quem dividi as experiências, o aprendizado, e todo o trabalho de quase um ano de pesquisa e resolução. Obrigado por participar de maneira tão ativa, dividindo as cargas e colaborando para os meus resultados. Agradeço ao colega Guilherme Lacerda, que gentilmente colaborou propondo alternativas metodológicas muito relevantes para encontrar caminhos dentro da Compreensão de Software. Agradeço ao colega Eduardo Bohrer, que também colaborou trazendo algumas sugestões, discutindo sobre a solução implementada, e ajudando na resolução de alguns problemas técnicos.

"Os que semeiam com lágrimas, com alegria colherão
Aquele que leva a preciosa semente, andando e chorando,
voltará com alegria trazendo os seus feixes."
(Bíblia Sagrada, Salmos 126:5-6)

Resumo

Dar manutenção a sistemas de softwares é uma tarefa bastante complexa, principalmente com o passar do tempo, em que equipes formadas por desenvolvedores de diferentes níveis de experiência contribuem para uma constante evolução do software. A medida em que novas funcionalidades são desenvolvidas, diversos defeitos surgem e são corrigidos. Porém nesse processo evolutivo, vários problemas podem ser gerados no projeto e no código-fonte, que dificultam a compreensão da arquitetura implementada. A *Visualização de Software* (VS) se apresenta como uma abordagem para a *Compreensão de Software* (CS), utilizando princípios de engenharia reversa para analisar artefatos que forneçam informações relevantes ao estudo da sua arquitetura. O objetivo da VS é representar o software através de elementos gráficos que estimulem o sentido visual e cognitivo dos desenvolvedores, proporcionando um melhor entendimento das características implícitas no código-fonte. Para isso são utilizadas diversas técnicas de extração de fatos através da análise da *Árvore de Sintaxe Abstrata* (ASA) do código. Entretanto existe um déficit muito grande de ferramentas de VS que proporcionem um processo de desenvolvimento de software mais dinâmico e interativo, sem abandonar a escalabilidade com relação às complexas análises combinadas e a extração de milhares de linhas de código. O grande problema é o alto custo envolvido no processamento dessas análises complexas, que dependendo da quantidade de código analisado, chegam a carregar gigabytes de informação em memória. Com base neste problema, propõe-se a construção de um modelo de extração de fatos chamado *PF Factfinder*, que realize a análise do código-fonte para armazenar os dados em um banco de dados relacional. O modelo aproveita princípios dos diversos modelos de extração consolidados na área, apontando o uso de *Structured Query Language* (SQL) como uma abordagem que viabilize todo tipo de consultas para a elaboração das análises de software. Separar as responsabilidades entre os processamentos de extração, análise dos dados e VS pode potencializar os processamentos realizados sobre o código-fonte, aumentando e viabilizando com isso a amplitude das análises combinadas de software com um mínimo impacto na escalabilidade da ferramenta.

Palavras-chave: Extração de Fatos. Análise da Árvore de Sintaxe Abstrata. Compreensão de Software. Engenharia Reversa. Visualização de Software.

Abstract

Maintain software systems is a complex task, particularly over time, in which teams of developers of varying levels of experience contribute to a constant evolution of software. When new features are developed, many issues appear and are fixed. But this evolutionary process various problems can be generated in the design and source code, which hinder the understanding of the implemented architecture. The *Software Visualization* (SV) is presented as an approach to *Software Comprehension* (SC), using reverse engineering principles to analyze artifacts that provide relevant information to the study of architecture. The goal of SV is to represent software through graphics that stimulate the visual and cognitive sense of the developers, providing a better understanding of the characteristics implicit in the source code. For this, several techniques are used to extract facts by analyzing the code *Abstract Syntax Tree* (AST). However, there is a very large deficit of SV tools that provide a more dynamic and interactive development process, without abandoning the scalability of the complex combined analysis and extraction of thousands of lines of code. The biggest problem is the high cost involved in the process of such complex analysis, which depending on the amount of analyzing code can load even gigabytes of information in memory. Based on this problem, we propose the construction of a fact extraction model called *PF FactFinder*, which performs the analysis of the source code to store data in a relational database. The model should reuse principles of many consolidated extraction models in the area, indicating the use of *Structured Query Language* (SQL) as an approach that allows all kinds of queries to the development of software analysis. Separating responsibilities among processes of extraction, data analysis and SV can enhance the processing performed on the source code, enabling and enhancing it with the amplitude of the combined analysis software with minimal impact on the scalability of the tool.

Key-words: Fact Extraction. Abstract Syntax Tree Analysis. Software Comprehension. Reverse Engineering. Software Visualization

Lista de ilustrações

Lista de tabelas

Lista de siglas

AHF *Attribute Hiding Factor*

ASA *Árvore de Sintaxe Abstrata*

AST *Abstract Syntax Tree*

CLM *Lines of Comment per Method*

COM *Lines of Comment*

CS *Compreensão de Software*

DIT *Depth of Inheritance Tree*

FAMIX *FAMOOS Information Exchange Model*

GQM *Goal-Question-Metrics*

GXL *Graph Exchange Language*

IDE *Integrated Development Environment*

JDT *Java Developer Tools*

JPA *Java Persistence API*

LOC *Lines of Code*

MHF *Method Hiding Factor*

MVC *Model-View-Controller*

NCM *Number of Class Methods in a Class*

NCV *Number of Class Variables in a Class*

NI *Number of Invocations*

NIM *Number of Instance Methods in a Class*

NIV *Number of Instance Variables in a Class*

NMA *Number of Methods Added*

NMAA *Number of Accesses on Attributes*

NME *Number of Methods Extended*

NMI *Number of Methods Inherited*

NMO *Number of Methods Overridden*

NOA *Number of Ancestors*

NOC *Number of Children*

NOM *Number of Methods*

NOP *Number of Parameters*

NPA *Number of Public Attributes*

NPM *Number of Parameters per Method*

OWL *Web Ontology Language*

PCM *Percentage of Commented Methods*

RDF *Resource Description Framework*

RHDB *Release History Database*

SAVE *Software Architecture Visualization and Evaluation*

SC *Software Comprehension*

SCM *Software Configuration Management*

SEON *Software Evolution Ontologies*

SOM *Software Ontology Model*

SQL *Structured Query Language*

SV *Software Visualization*

UML *Unified Modeling Language*

VS *Visualização de Software*

VTML *Visual Trace Modeling Language*

XMI *XML Metadata Interchange*

XML *Extensible Markup Language*

XSTL *Extensible Stylesheet Language Transformations*

Sumário

1 Introdução

O processo de construir software é uma atividade complexa, e interagir com a sua arquitetura, comportamentos e domínio da informação, mesmo para desenvolvedores experientes, pode ser um desafio. A Engenharia de Software permite ao desenvolvedor criar soluções muito criativas para os problemas que lhe são apresentados, empregando toda a sua capacidade de abstrair os conceitos e questões envolvidas na construção de sistemas. Porém a indústria de software vem se mostrando carente de ferramentas eficazes e práticas para estruturar o processo da Manutenção de Software (PARNIN et al., 2008).

Segundo Löwe e Panas (2005) e Telea e Voinea (2011) entre 50% a 80% dos custos de um sistema de software estão relacionados com atividades de manutenção, removendo problemas de projeto e implementação, bem como acrescentando novas funcionalidades, sendo assim uma atividade essencial em qualquer projeto de software. Contudo, esse trabalho pode se tornar bastante improdutivo, caso o desenvolvedor tenha dificuldades de interagir com os artefatos apresentados, especialmente a arquitetura definida no código-fonte.

Uma arquitetura complexa e uma implementação problemática podem criar grandes obstáculos ao processo de constante evolução do software. Entendendo a necessidade de compreender e descrever a arquitetura da solução de software, Kruchten (1995) apresentou o conceito de visões arquiteturais, cujo objetivo é oferecer, através de diferentes modelos e representações visuais, instrumentos para identificar e compreender as diversas necessidades e perspectivas de um sistema de software.

Compreensão de Software (CS) é a tarefa de elaborar e responder perguntas específicas sobre um determinado sistema de software (REISS, 1998). Estas questões surgem durante o desenvolvimento e manutenção de um sistema de software quando o programador quer entender o comportamento ou determinar as consequências de uma modificação no código. Isso engloba compreender os diferentes conceitos envolvidos na construção e estruturação das classes, pacotes e responsabilidades dentro do sistema, bem como o fluxo, a troca de mensagens entre os componentes do sistema e a própria estrutura física da aplicação, plataforma e suas complexidades.

Por depender tanto do raciocínio, intelecto e experiência do desenvolvedor com a tecnologia e o domínio da aplicação, o processo de compreender um projeto de software exige recursos específicos, como modelos e representações, que facilitem a percepção e a compreensão de seus processos de desenvolvimento (OLIVEIRA, 2011). Porém, em muitos casos, a produção de novos documentos e modelos de representação não satisfazem as reais expectativas das organizações, sendo geralmente mais interessadas em prazos e entregas

do que em uma documentação abrangente. Além disso, devido a falta de uma manutenção mais adequada nos artefatos de especificação do sistema, o código-fonte acaba se tornando a principal e muitas vezes única fonte confiável de informações sobre a aplicação (LÖWE; PANAS, 2005).

Numerosos estudos indicam que a análise semântica é realizada mais rapidamente pelo cérebro para imagens do que para textos, visto que a informação gráfica é mais fácil e eficiente de ser lembrada do que a informação textual (UMPHRESS et al., 2002). Nesse contexto, a *Visualização de Software* (VS) propõe a construção de mecanismos de extração e visualização de informações aplicados ao software e seus artefatos, que promovam um aumento da eficiência na CS. A VS tem como objetivo expor características e comportamentos identificados na implementação do software, por meio de representações gráficas e modelos visuais. Os modelos gerados durante os processos de VS permitem ao desenvolvedor detectar padrões e estruturas implícitas no código-fonte, estimulando a percepção visual através de metáforas e analogias que evocam abstrações mentais e contribuem para a compreensão e memorização do software, suas estruturas e funções (OLIVEIRA, 2011).

1.1 Motivação e Problemas Encontrados

Telea et al. (2010) identificaram que existe uma exigência de ferramentas de VS mais flexíveis e rápidas, que produzam visualizações mais simples e eficazes. Eles mesmos apontaram que as ferramentas atuais de visualização da arquitetura ainda são bem limitadas e pouco escaláveis, especialmente quanto à aquisição e extração dos dados e quanto à rapidez no processamento de grandes quantidades de linhas de código. Isso ocorre porque manter esses dados em memória é caro e consome uma quantidade muito grande de recursos. Segundo Anslow et al. (2004), dependendo do programa que se deseja mapear e a quantidade de informações extraídas, esses custos podem chegar a gigabytes em uso de memória.

Paralelamente, Löwe e Panas (2005) diagnosticaram que o grande número de informações, perspectivas, objetivos, análises e abstrações, pode implicar em um número muito elevado e variável para as possíveis visões de um determinado software. Isso tornaria muito difícil predefinir uma combinação exata de informações apropriadas para a extração e a análise dos dados. O problema disso, segundo Telea et al. (2010) é a complexidade envolvida na comparação entre os diversos estados encontrados do software e as diferentes visões arquiteturais construídas a partir do software ao longo do tempo. Para diluir os custos envolvidos no processamento dessas análises especializadas, seria necessário armazenar essas visões em uma base comum e de fácil acesso, que ficaria disponível para todo tipo de consultas.

Segundo Löwe e Panas (2005), definir e separar a responsabilidade de extração dos dados da camada de visão, permitiria modelar uma base incremental de propriedades, mapeadas de acordo com a demanda de visualização. Esses dados poderiam ser reutilizados também para mapear informações ainda mais profundas do software, sem perder os dados analisados e mapeados anteriormente. Esses dados confirmariam a existência de inúmeras relações existentes entre as propriedades do software, permitindo evidenciar padrões e conceitos normalmente sugeridos apenas em análises teóricas e com experimentos controlados (SAMADZADEH; NANDAKUMAR, 1991).

Destacou-se a necessidade de realizar análises de software mais complexas, em projetos de software com grandes quantidades de linhas de código, priorizando atributos de desempenho e escalabilidade. Para atender esta demanda, optou-se pela criação de um modelo de extração de fatos, denominado *PF Factfinder*, apoiando-se na arquitetura relacional para armazenar informações relevantes à CS, e disponibilizando-as para todo tipo de consultas *Structured Query Language* (SQL).

1.2 Objetivos do Trabalho

Para implementar uma solução que atenda ao problema destacado, decidiu-se implementar uma ferramenta extratora, que obtenha informações sobre o código-fonte através da análise da *Abstract Syntax Tree* (AST), persistindo as informações em um banco de dados incremental como apoio à CS e evolução de software. O objetivo da solução implementada é disponibilizar uma arquitetura de extração de código que seja ao mesmo tempo robusta e escalável, com um desempenho que viabilize análises de software mais complexas, através de consultas aos dados mais dinâmicas e otimizadas. Foram enumerados alguns objetivos específicos para a solução implementada:

a) O modelo de extração deve mapear entidades da AST, de maneira a permitir extrair informações de diferentes projetos de software de orientados a objetos.

b) O modelo proposto deve reunir e agregar características comuns aos principais modelos de extração de linguagens Orientadas a Objetos, propostos pela comunidade atual de Engenharia Reversa.

c) O modelo deve mapear conceitos que viabilizem a construção de uma base de dados incremental e reutilizável de dados históricos que apóiem a evolução do software.

d) O extrator deve persistir as informações obtidas em um banco de dados relacional disponível para todo tipo de consultas SQL.

1.3 Principais Contribuições

A implementação de uma arquitetura de extração apoiada por uma base de dados relacional, bem como o uso de recursos como a linguagem de consultas SQL, pode contribuir para a elaboração de ferramentas de CS e VS mais ágeis e versáteis. A melhoria no desempenho das consultas, pode viabilizar um aprofundamento nas análises de software realizadas para compreender software, sem abrir mão da performance mesmo para projetos de maior porte. A otimização do consumo de processamento em memória pode abrir espaço para atingir novos níveis de complexidade tanto nas análises de evolução do software, quanto na qualidade de código, permitindo analisar grandes quantidades de linhas de código, sem impactar nos custos computacionais envolvidos.

O modelo proposto pretende agregar os principais conceitos da Orientação a Objetos encontrados nos modelos de extração propostos pela comunidade. A implementação de um modelo convergente pode ampliar a abrangência da arquitetura, permitindo utilizar as estruturas de extração para compreender diferentes projetos de software, com suas características específicas. Com um modelo genérico baseado nos princípios da Orientação a Objetos, seria possível atender mesmo às diferentes linguagens de programação orientadas a objetos utilizadas no mercado.

1.4 Organização do Trabalho

O restante do trabalho está organizado da seguinte maneira. O Capítulo 2 traz um aprofundamento para os conceitos de CS, engenharia reversa, extração de fatos, arquitetura de software e VS, incluindo definições obtidas de um aprofundado estudo realizado sobre os temas. O Capítulo 3 detalha os procedimentos de uma revisão sistemática realizada com o intuito de identificar trabalhos fortemente relacionados com a presente abordagem de extração de fatos e o armazenamento de dados relacionados à CS. O Capítulo 4 apresenta a elaboração de uma solução baseada na implementação de um modelo de extração convergente que atenda às necessidades de visualização apontadas pelos diferentes modelos de extração estudados. Foi implementado um extrator de fatos capaz de transformar elementos de código-fonte em informações de projetos de software orientados a objetos e persistindo-as em um banco de dados. O Capítulo 5 apresenta alguns dados gerados e extraídos de projetos de software existentes, através de um processo de medição realizado sobre a ferramenta, utilizando algumas medidas identificadas com o intuito de validar os resultados obtidos. Por fim, o Capítulo 6 apresenta algumas considerações finais sobre os resultados obtidos pela ferramenta, analisando-os e comparando-os com os objetivos propostos para o trabalho. São apresentadas algumas perspectivas de evolução da solução apresentada, apontando trabalhos futuros que complementem os resultados obtidos.

2 Compreensão de Software

Neste capítulo é apresentado um estudo sobre a *Compreensão de Software* (CS), buscando definições técnicas e teóricas que fundamentem o presente trabalho. Na seção 2.1 é apresentada uma introdução ao estudo da CS, justificando para tal o uso da Engenharia Reversa e da *Visualização de Software* (VS). Na seção 2.2 são apresentados conceitos da Engenharia Reversa relacionados com a CS, trazendo definições aos conceitos de Extração de Fatos e introduzindo uma tecnologia de análise da Árvore de Sintaxe Abstrata do Java chamada *Java Developer Tools* (JDT), que pode ser utilizada na implementação do extrator proposto. Na seção 2.3 são abordados diferentes níveis de análise de software realizadas em cima do código-fonte, apresentando os conceitos de indicadores e medidas de código e a análise de cheiros de código. Na seção 2.4 é apresentada a VS, com seus diferentes enfoques de CS e algumas técnicas de visualização que contribuam para a construção de ferramentas de VS interativas e efetivas. Por último, na seção 2.5 são realizadas algumas considerações finais com relação à CS, fundamentando a pesquisa e justificando a necessidade de uma ferramenta de extração de dados históricos de software.

2.1 Introdução

Uma das atividades mais presentes na vida de um desenvolvedor é a Manutenção de Software. Segundo Löwe e Panas (2005) e Telea e Voinea (2011) 50% a 80% dos gastos com software provêm de atividades de correção de *bugs* e implementação de novas funcionalidades. Sistemas de software reais estão sempre mudando para satisfazer as exigências do mercado e as expectativas de seus usuários. De outra maneira não seria possível adaptar o negócio da aplicação às tendências, legislações e inovações tecnológicas emergentes. Para isso as equipes de desenvolvimento de software precisam estar constantemente preocupadas com a melhoria contínua e a qualidade do produto ou serviços oferecidos (CANFORA et al., 2011). Contudo, existem ainda muitos desafios envolvidos na manutenção de software, entre eles a necessidade de compreender software e todas as complexidades envolvidas no processo de desenvolvimento.

> Compreensão de Software é um processo humano intensivo, onde desenvolvedores adquirem conhecimento suficiente sobre um artefato de software ou um sistema completo, de modo que sejam capazes de realizar, através de análise, suposições e levantamento de hipóteses, uma determinada tarefa com sucesso (CANFORA et al., 2011).

Técnicas de inspeção são bastante comuns na Engenharia de Software, pois proporcionam ao desenvolvedor conhecimento sobre o estado e a evolução dos artefatos contruí-

dos ao longo do desenvolvimento de software. Elas se baseiam na intuição e na experiência do desenvolvedor ao deparar-se com problemas recorrentes de *design* e qualidade do código. Entretanto, segundo Parnin et al. (2008) técnicas de inspeção são adotadas muitas vezes em troca de altos custos com recursos humanos. Isso acontece porque atividades como essas demandam um intenso trabalho de revisão em cima dos artefatos construídos. No entanto, a falta de percepção e compreensão dos desvios arquiteturais e da evolução do software podem comprometer, tanto o processo de desenvolvimento, quanto o próprio produto desenvolvido (OLIVEIRA, 2011).

Paralelamente, técnicas de refatoração de código podem se tornar uma grande engrenagem de melhoria contínua para o produto de software, principalmente se a equipe é formada por desenvolvedores preocupados com a qualidade do código implementado. Segundo Parnin et al. (2008) o problema é que os benefícios da qualidade de software obtidos através dessas práticas são muitas vezes diluídos pelos altos custos e a baixa prioridade, quando comparada com a urgência da correção de *bugs* e da implementação de novas funcionalidades. Esse problema é ainda mais agravado quando se percebe que 40% do tempo investido em manutenção de software é gasto para compreender o software e a arquitetura que serão evoluídos (TELEA; VOINEA, 2011).

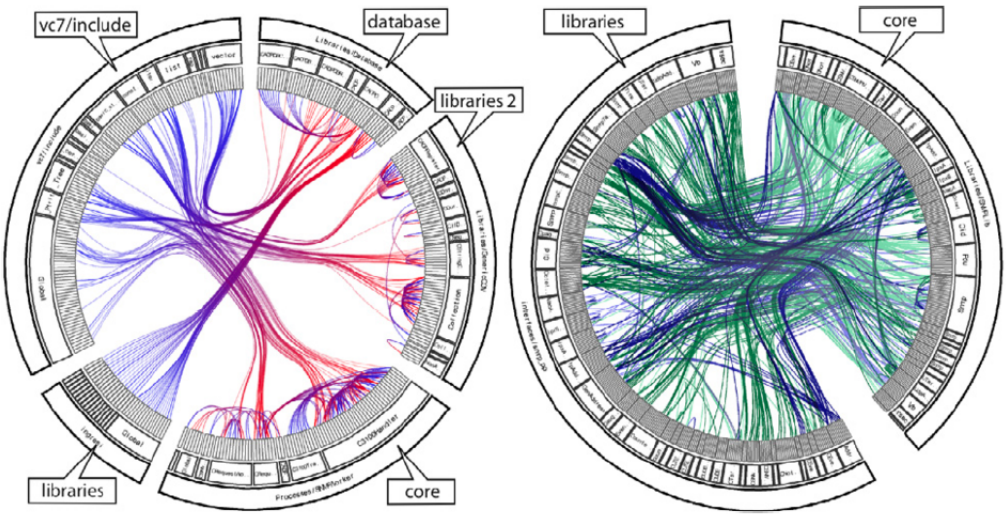
Nesse contexto, a Engenharia Reversa aliada às técnicas de VS tem sido intensamente abordada como uma alternativa à CS em ambientes de desenvolvimento corporativo. Segundo Storey et al. (2008) existem várias ferramentas de manutenção do código que usam da visualização para revelar ao desenvolvedor, informações que não são obviamente percebidas durante o exame dos artefatos, como é o caso da Figura 1, um grafo de chamadas que evidência o grau de dependência entre diferentes módulos de um sistema. Exemplos mais triviais desse conceito podem ser encontrados em ferramentas de depuração de código, cobertura de testes e marcação de erros e *warnings*, que apresentam ao desenvolvedor formas visuais alternativas, para a interação com o software durante o desenvolvimento.

Diversas abordagens têm sido levantadas para as diferentes propostas de CS e Engenharia Reversa. Alguns autores identificam níveis de profundidade para a análise de CS, outros apontam visões de software baseadas em diferentes perspectivas do desenvolvedor. Em seu livro, Diehl (2007) apresentou uma divisão em etapas para o processo de tratamento dos dados, com a finalidade de compreensão do software:

- Primeira etapa: A aquisição dos dados através de procedimentos de extração em cima de artefatos como o código-fonte;

- Segunda etapa: A análise dos dados extraídos, através de filtragens e agrupamentos para limitar a consulta aos dados mais relevantes;

Figura 1 – Visualização de um grafo de chamadas



Sistema modular (esquerda) *versus* "código spaghetti" (direita) (TELEA et al., 2009)

- Terceira etapa: A visualização das informações mapeadas e convertidas em símbolos gráficos e representações visuais para serem apresentados ao usuário;

2.2 Engenharia Reversa

À medida que um software é modificado, ele vai se tornando mais complexo, de maneira que ao longo do tempo fica mais difícil de dar manutenção ao código-fonte. Para viabilizar uma melhor compreensão do software mantido, é necessário que sua arquitetura seja bem definida, documentada e atualizada (OLIVEIRA, 2011). No entanto, um dos grandes desafios para a CS, é a ausência de documentações e projetos que especifiquem adequadamente o produto de software. Isso acontece porque as organizações estão geralmente mais interessadas em entregáveis que agreguem um real valor ao cliente, do que em uma documentação técnica abrangente (FOWLER; HIGHSMITH, 2001).

Entretanto o conhecimento adequado sobre o produto de software, dá aos desenvolvedores maior segurança na hora de dar manutenção ao código-fonte. Sem isso, a equipe de desenvolvimento pode perder o controle da arquitetura implementada, e cometer muitos erros no *design*. Segundo Löwe e Panas (2005) a falta de preocupação com a manutenção dos artefatos de especificação, frequentemente transforma o código na única fonte confiável de informações sobre o sistema. Segundo Oliveira (2011) isso prejudica o equilíbrio entre o custo, tempo e a qualidade, gerando insatisfação em todos os envolvidos, visto que a única forma de identificar o problema é vasculhar o código manualmente. Nesses casos, o padrão IEEE-1219 recomenda a Engenharia Reversa como um apoio fun-

damental para obter conhecimento sobre a arquitetura e padrões no projeto de software (EDELSTEIN, 1993).

> Engenharia reversa é o processo de análise de um sistema, responsável por identificar componentes e artefatos do sistema e suas inter-relações, a fim de criar outras formas de representação do sistema em um nível mais alto de abstração (CHIKOFSKY et al., 1990).

Nesse sentido além de permitir a compreensão da arquitetura e do código-fonte, a Engenharia Reversa pode oferecer ao desenvolvedor recursos como a redocumentação do sistema e do banco de dados, rastreabilidade entre artefatos de software, bem como a identificação de impactos de mudança nas práticas de Evolução de Software (CANFORA et al., 2011). Os impactos mapeados podem ajudar tanto na remodularização de sistemas, quanto na migração de código para novas arquiteturas e plataformas ou renovação de interfaces de usuários. Como é o caso da Figura 2, que através de comparações visuais realizadas com relação à quantidade de linhas de código e comentários, permitiu identificar um problema de clonagem e duplicação de código em um determinado conjunto de classes de um sistema.

Existem diversas ferramentas que utilizam métodos de Engenharia Reversa, entre elas *parsers* e extratores, que analisam o código-fonte identificando e recuperando informações do sistema, e disponibilizando-as para a CS.

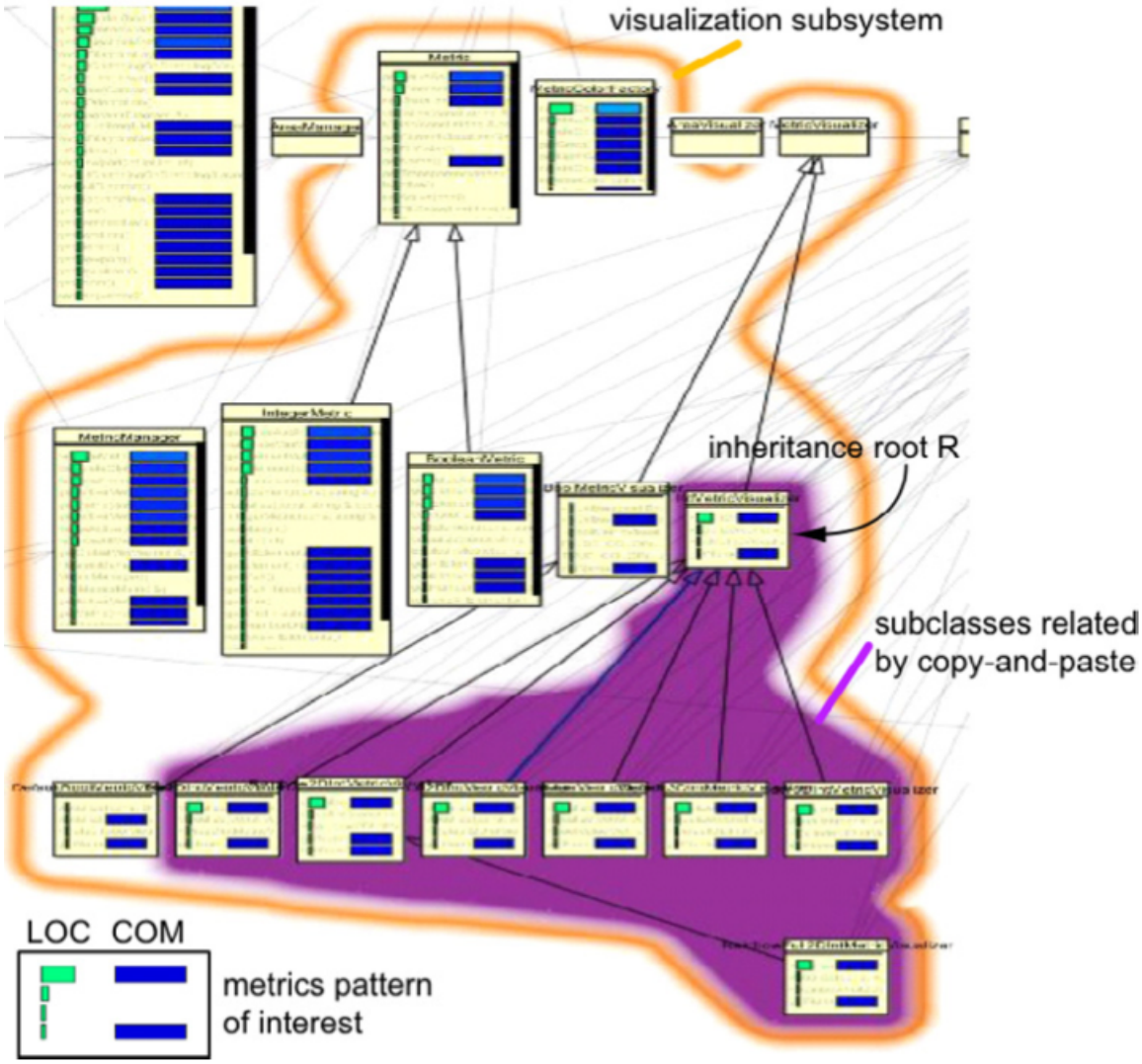
2.2.1 Extração de Fatos

O Software é uma entidade composta por vários atributos e propriedades que o descrevem. Segundo Siket e Ferenc (2004), qualquer informação que ajude de alguma maneira a descrever o código-fonte e suas diferentes características pode ser denominada Fato. Essas propriedades podem ser medidas e quantificadas para caracterizar o software (MATHIAS et al., 1999), ou mesmo referenciar conceitos que possuem suas próprias características, como por exemplo classes, métodos e pacotes. Essas medidas, quando mapeadas por um modelo de Extração de Fatos, definem um conjunto inicial de informações sobre o software.

Um dos modelos que mais se tem difundido no meio acadêmico é o modelo *FAMOOS Information Exchange Model* (FAMIX) proposto por Ducasse et al. (2011), sendo reaproveitado em diversos trabalhos como em (GHEZZI; GALL, 2013) e (D'AMBROS; LANZA, 2010), e resumidamente apresentado através da Figura 3, que representa o núcleo do modelo construído:

> Extração de fatos é um processo que define diferentes etapas que descrevem a maneira como os fatos sobre o código-fonte podem ser obtidos. Essas etapas incluem a aquisição de informações sobre projeto e configuração, a análise dos arquivos fontes com ferramentas de análise de

Figura 2 – Identificação de Clonagem de Código

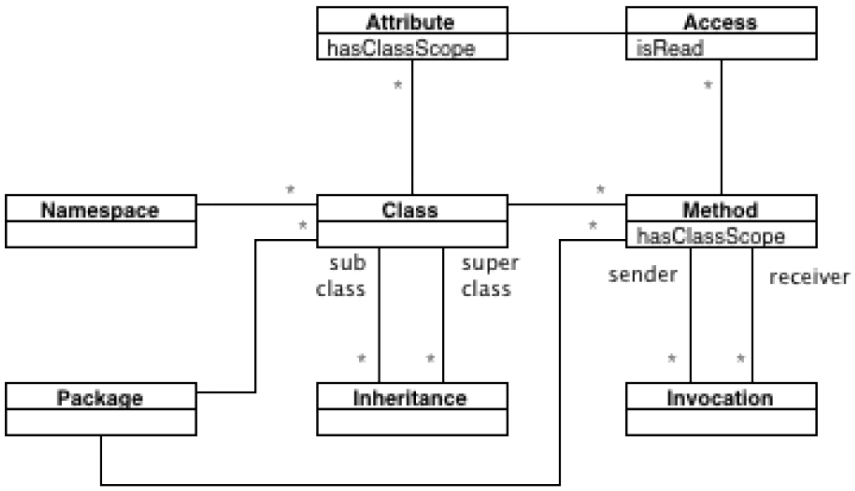


Os índices de LOC e COM apontam um padrão de cópia e cola (TELEA et al., 2009)

projeto, a criação de representações desses fatos extraídos, a fusão dessas representações e os diferentes processamentos realizados na fusão dessas representações para permitir o uso das informações coletadas (SIKET; FERENC, 2004).

Extração de fatos se preocupa em recuperar, modelar e fornecer informações detalhadas sobre o software, em um nível mais baixo de abstração, obtendo diretamente do código-fonte as informações necessárias para a análise da *Abstract Syntax Tree* (AST) (TELEA et al., 2009). O objetivo de um extrator de fatos é permitir construções de análises de compreensão do software mais elaboradas, com base nas perspectivas desejadas. Para isso é necessário adotar um modelo de dados adequado ao nível de detalhamento esperado pelas análises, o que implica em manipular um conjunto de dados que seja ao

Figura 3 – FAMIX *Core* - Principais Entidades



Principais entidades do modelo de extração FAMIX

mesmo tempo suficiente e completo.

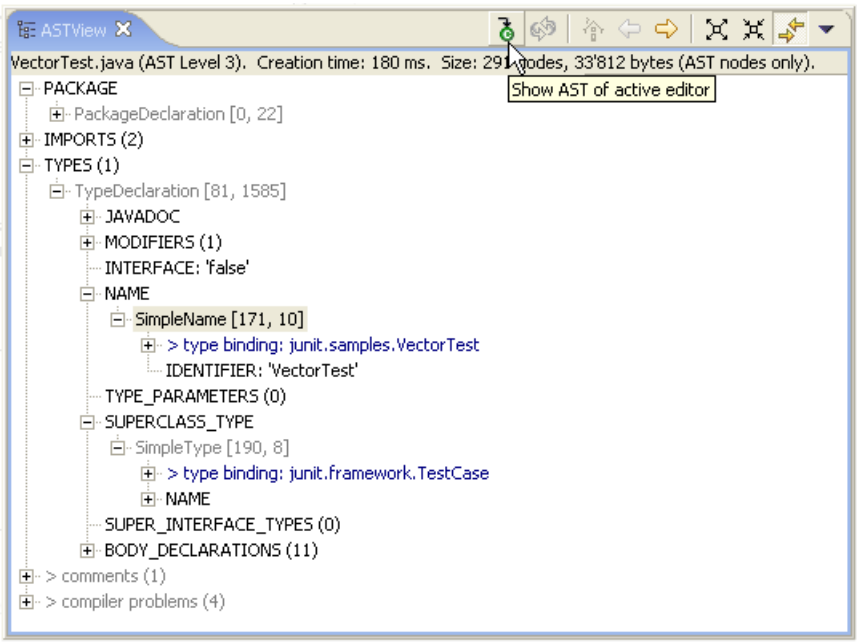
Definidas as entidades que serão mapeadas (TELEA; VOINEA, 2011), é preciso estabelecer vínculos entre elas. Isso sugere mapear as associações, dependências, hierarquias, acessos, invocações, tipos de dados e o que for necessário para manter o modelo conciso (TELEA et al., 2010). É preciso também mapear a sintaxe apresentada de maneira coerente. Pode-se escolher entre construir uma gramática específica para uma determinada linguagem de programação, utilizar alguma gramática já existente para a linguagem dada (PENTA et al., 2008), ou então adaptar os conceitos sintáticos e linguísticos de maneira que o extrator possa ser compatível com um maior número de linguagens de programação possíveis (SANTOS; BARROS, 2009).

Existem muitas abordagens para a extração de fatos no meio acadêmico atual. Em sua grande maioria, as propostas procuram se apropriar da característica extensível, reusável e adaptável do *Extensible Markup Language* (XML), para permitir serviços de análises integradas e combinadas dos fatos de software (HOLT et al., 2000). O desafio da extração de fatos, entretanto, ainda está relacionado com a escalabilidade nas análises realizadas a partir dos dados obtidos, principalmente no processamento de grandes quantidades de linhas de código (TELEA et al., 2010). Isso aponta para possíveis revisões nas propostas de persistência dos dados extraídos, buscando alternativas que se apresentem mais robustas para o nível de complexidade envolvidas nos processamentos dos diferentes tipos de massas de dados.

2.2.2 Java Development Tools

JDT é uma ferramenta de análise de AST, mantido pela Eclipse, que tem como objetivo compilar e analisar o código-fonte, organizando-os em uma estrutura de árvore hierárquica (BAXTER et al., 1998), que representa as entidades do código-fonte através de nós e associações, para extrair informações relevantes dos fragmentos do código, com o objetivo de CS. Segundo Sager et al. (2006), cada pedaço de código-fonte pode ser representado como uma AST, e através do componente *ASTParser*, presente no núcleo do JDT, também pode ser convertido em representações que permitam análises e comparações mais detalhadas.

Figura 4 – *AST View* - Componente JDT



Componente responsável por apresentar uma visão da AST compilada do código (ECLIPSE, 2014)

O JDT oferece um modelo que mapeia um conjunto básico de entidades de código, fornecendo rotinas de *parse* que transformam linhas de código textuais em objetos com suas propriedades específicas. Usando os componentes do JDT, é possível processar arquivos de código-fonte Java, com o intuito de identificar classes, métodos, atributos, parâmetros e todo tipo de notação sintática da linguagem. Um exemplo de visualização obtido pelo processamento da AST do código através do JDT pode ser obtido através da ferramenta *AST View* apresentada na Figura 4. Segundo Murphy et al. (2006), o JDT está presente em representações para diversas perspectivas do Eclipse *Integrated Development Environment* (IDE), como o *Package Explorer view*, *Problems view*, *Outline view* e mesmo as *views* relacionadas à depuração de código. Entretanto, com o JDT *Core*, é possível se apropriar das estruturas primárias da API para realizar análises de código sem dependências com a IDE do Eclipse.

2.3 Análise de Software

A *Unified Modeling Language* (UML) tem sido largamente usada para a descrição de arquiteturas de software, tornando-se o padrão de escrita e modelagem para a indústria de software. Entretanto, segundo Oliveira (2011), algumas deficiências têm sido identificadas, levantando certos questionamentos com relação a algumas abordagens do uso da linguagem. Segundo ele, não existem mecanismos suficientemente robustos de análise e verificação de consistência entre as informações expressas nos modelos UML e o codigo-fonte implementado.

Para Oliveira (2011), o principal limitador técnico seria a divergência muito comum entre a arquitetura conceitual planejada e a arquitetura implementada. Isso acontece porque na prática, os sistemas costumam ser modificados diretamente no código-fonte, criando incompatibilidades entre os modelos e a implementação, gerando com isso, instabilidades na manutenção do sistema. Segundo ele a negligência dos desenvolvedores, prazos curtos, técnicas inadequadas, necessidade de otimizações eventuais e a recorrente falta de documentação da arquitetura, geralmente direcionam os desenvolvedores a não dar manutenção aos modelos conceituais projetados.

Sem uma representação fiel da arquitetura do software, a qualidade da manutenção do código pode ficar comprometida. Para solucionar esse problema, muitas abordagens de Engenharia Reversa têm sido implementadas para construir análises mais aprofundadas sobre a arquitetura do software de maneira dinâmica (D'AMBROS; LANZA, 2010). Isso pode viabilizar a CS sem a necessidade de uma documentação prescritiva abrangente, como por exemplo em projetos de software onde a manutenção dos artefatos conceituais não é tão prioritária (OLIVEIRA, 2011).

Análise de software visa obter uma quantidade de informação suficiente sobre o software para preparar o desenvolvimento no processo de manutenção. Segundo Oliveira (2011), durante as etapas de análise em um processo de evolução de software, os dados relevantes são filtrados, combinados, agrupados, de modo a estruturar e focar em informações que precisam trazer ao desenvolvedor algum conhecimento ou percepção específica. Juntamente com D'Ambros e Lanza (2010) ele identificou alguns níveis de aprofundamento da análise de software presentes nos processos de CS: São eles a seleção de indicadores do software e a descoberta de padrões e problemas no *design* e evolução da arquitetura de software.

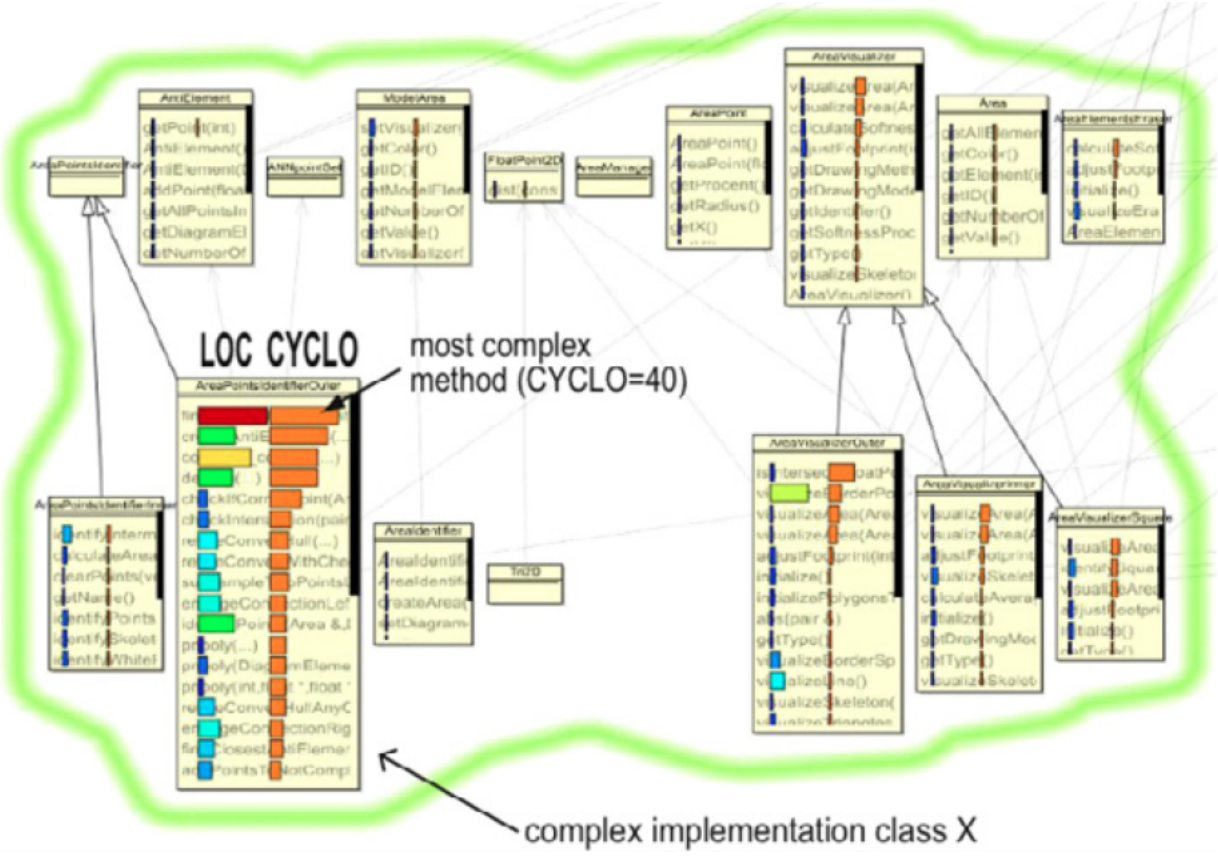
2.3.1 Análise de Indicadores de Software

Indicadores de software tem um papel interessante na Engenharia de Software, pois utilizam de conceitos de Engenharia Reversa para avaliar a qualidade e a complexidade de sistemas de software. Segundo Lanza (2003), a extração de medidas e fatos

do código-fonte, muitas vezes produz tabelas enormes, difíceis de interpretar, e cheias de dados irrelevantes para determinadas necessidades de análise. Nesse sentido, medidas e indicadores de software podem auxiliar na filtragem e agrupamento das informações, analisando determinados valores para encontrar informações que melhor contribuam com a compreensão do software (SIKET; FERENC, 2004).

Na Figura 5 foi possível visualizar, através da análise dos indicadores e medidas do código, o método mais complexo identificado no projeto de software. Curiosamente o método mais complexo é também o maior método de todo o código e pertence à maior classe de todo o projeto.

Figura 5 – Análise de Medidas de Código



Identificado o método maior e mais complexo (TELEA et al., 2009)

Formalmente, indicadores medem certas propriedades de um sistema de software, mapeadas através de números ou outros símbolos, de acordo com regras bem definidas para os objetivos de medição. Os resultados dessa medição podem ser usados para descrever, julgar ou mesmo predizer características do sistema de software com respeito às propriedades que têm sido medidas (LANZA, 2003).

Segundo Mathias et al. (1999) as medidas podem ser geradas combinando diferentes informações extraídas do software, com a finalidade de orientar e qualificar o software analisado e o próprio código-fonte. Estudos complementares entre os trabalhos de Chidamber e Kemerer (1994) Siket e Ferenc (2004) e Demeyer et al. (1999) identificaram, organizaram e agruparam medidas de código para diferentes necessidades e perspectivas de desenvolvimento. Baseado nos estudos realizados, é possível identificar quatro grupos bem distintos de medidas:

a) Medidas de Sistema:
   - NCL: Número de classes do sistema
   - TLOC: Número total de linhas de código não vazias no sistema
   - TNM: Número total de métodos no sistema
   - TNA: Número total de atributos no sistema
   - WMC: Média da complexidade de método por classe

b) Medidas de Classe:
   - HNL: Nível de aninhamento hierárquico, quantidade de superclasses
   - DIT: Nível de profundidade na árvore hierárquica
   - NAM: Número de métodos abstratos
   - NCV: Número de variáveis de classe
   - NIA: Número de atributos herdados
   - NIV: Número de variáveis de instância
   - NMA: Número de métodos adicionados, não definidos na superclasse
   - NME: Número de métodos estendidos na subclasse, chamando o mesmo método também na superclasse
   - NMI: Número de métodos herdados da superclasse e inalterados
   - NMO: Número de métodos sobrescritos da superclasse
   - NOC: Número de filhos imediatos de uma superclasse
   - NOM: Número de métodos da classe
   - CBO: Número de classes ao qual está acoplada
   - RFC: Número de métodos de outras classes que podem ser chamados em trocas de mensagens
   - LCOM: Índice de falta de coesão nos métodos da classe
   - WLOC: Número de linhas de código em todos os corpos de método da classe
   - WMSG: Número de mensagens enviadas em todos método da classe
   - WNAA: Número de vezes que todos os atributos definidos são acessados
   - WNI: Número de invocações de todos os métodos declarados na classe
   - WNMAA: Número de acessos nos atributos da classe

- WNOC: Número de todas as classes descendentes
- WNOS: Número de instruções em todos os corpos de métodos da classe

c) Medidas de Métodos:

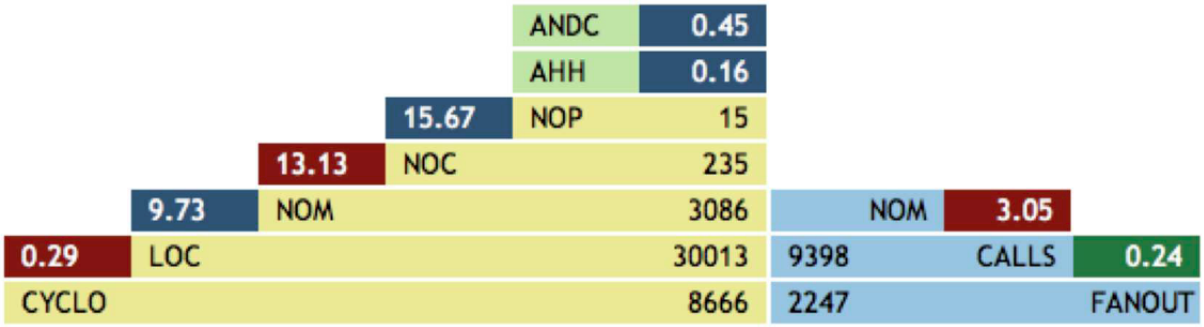
- LOC: Linhas de código no corpo do método
- MHNL: Nível de aninhamento hierárquico da classe que implementa o método
- MSG: Número de mensagens enviadas no corpo do método
- NI: Número de invocações/chamadas a outros métodos
- NMAA: Número de acessos do método a atributos da classe
- NOP: Número de parâmetros do método
- NOS: Número de instruções no corpo do método

d) Medidas de Atributos:

- AHNL: Nível de aninhamento hierárquico das classes que define o atributo
- NAA: Número de acessos ao atributo

Existem visualizações específicas voltadas para representar indicadores e medidas de projetos de software, como as apresentadas na Figura 6, onde um projeto é mapeado através de uma pirâmide de medidas que realiza comparativos entre alguns indicadores para identificar padrões relacionados a quantidade de parâmetros por métodos, número de métodos, quantidade de linhas de código e complexidade ciclomática.

Figura 6 – Pirâmide de Medidas de Software



Apresenta medidas obtidas de um determinado software (GHEZZI; GALL, 2013)

As medidas por si só não oferecem uma compreensão muito aprofundada do software, nem sugerem alternativas imediatas aos desafios do código-fonte. Entretanto, mesmo sem apontar precisamente problemas ou soluções de implementação, as medidas podem servir como um ponto de partida para análises mais apuradas dos fatos extraídos a fim de elevar o nível de compreensão do software (PARNIN et al., 2008).

2.3.2 Análise de Arquitetura do Software

Arquitetura de software é a estrutura ou organização de um sistema expressa através dos seus componentes, as relações entre eles, propriedades externamente visíveis e princípios que guiam o projeto e seu processo evolutivo (OLIVEIRA, 2011). A necessidade de organizar e descrever arquiteturas de software instigou a criação de conceitos como visões arquiteturais, cujo objetivo é representar arquiteturas de sistemas de software de acordo com os interesses de compreensão apresentados (KRUCHTEN, 1995).

No contexto de CS, indicadores de software contribuíram muito para diminuir os custos em análise e inspeção de código, mas segundo Parnin et al. (2008), também criaram um novo problema. Dependendo do nível de abrangência da quantidade de análises realizadas em cima da aplicação, as técnicas adotadas para apontar medidas de código ainda podem gerar uma grande massa de dados, difícil de comparar e com inúmeros falsos-positivos, dificultando o processo de compreensão do software.

A análise de cheiros de código foi proposta por Fowler (1999) em seu livro, e visa aplicar análises mais aprofundadas sobre o código-fonte, a fim de identificar sintomas claros de problemas de implementação e arquitetura. O autor propôs um catálogo de refatorações baseado na identificação de maus cheiros de código, apontando problemas de implementação comuns ao desenvolvimento, e respectivas alternativas de melhoria perfectiva. Baseado nessa proposta, Parnin et al. (2008) construiu um catálogo de visualizações para cheiros de código (*code smells*) que permitiria oferecer maior precisão às análises de código. Os cheiros catalogados foram agrupados em quatro categorias distintas para diferentes perspectivas do código: *statement code smells*, *class code smells*, *method code smells* e *collaboration code smells*.

> Cheiros de código são indicações estruturais de problemas de *design* muito maiores escondidos no código. Desenvolvedores experientes percebem que os cheiros surgem quando eles estão fazendo modificações no código-fonte e encontram muita resistência, demandando esforço indevido para implementá-las (PARNIN et al., 2008).

Alguns dos cheiros de código abordados por Parnin et al. (2008) são listados a seguir:

a) *Statement Code Smells*

- *Message Chain*: Uma instrução que tem uma longa sequência de invocações de métodos ou acessos de atributos.
- *Data Clumps*: Um aglomerado de variáveis relacionadas que é passado por parâmetros durante uma troca de mensagens.
- *Primitive Obsession*: Um dado que é costumeiramente representado através de tipos primitivos ao invés de objetos especializados.

b) *Class Code Smells*

- *Data Class*: Classes que possuem muitos atributos públicos e poucos comportamentos ou funcionalidades.
- *Large Class*: Classes que são muito grandes, com muitas funcionalidades e responsabilidades.
- *Refused Bequest*: Classes que não utilizam ou sobrescrevem quase nenhum comportamento herdado de suas superclasses.

c) *Method Code Smells*

- *Long Method*: Método que é muito longo, difícil de entender e com muito código duplicado.
- *Long Parameter List*: Métodos com uma lista de parâmetros muito longa e difícil de trabalhar.

d) *Collaboration Code Smells*

- *Middle Man*: Classe que delega a maioria do seu trabalho e responsabilidades para outras classes.
- *Feature Envy*: Um método que faz muitas chamadas para outras classes em vez de chamar métodos de sua própria classe.

Parnin et al. (2008) apresentou uma proposta inicial à visualização de cheiros de código, representada através da Figura 7, deixando porém ainda a desejar com relação a interatividade das representações propostas. Ghezzi e Gall (2013) também propôs uma alternativa de análise da evolução da arquitetura do software, que aborda três níveis de análise de software, integrando a extração de fatos com a identificação de medidas de código e problemas de *design* sinalizados através dos *code smells*. Entretanto o grande desafio atualmente é encontrar propostas visuais que melhor representem os problemas de código identificados.

Segundo Umphress et al. (2002), as construções gráficas devem complementar o código-fonte sem afetar sua aparência familiar, e uma visão do software deve se apresentar como um prolongamento natural do código fonte, assim como o código fonte deve também parecer ser um prolongamento natural de suas representações. Para Diehl (2007), a visualização deve ser efetiva em seu objetivo, e para isso é importante que as representações e metáforas visuais se adaptem às habilidades de percepção dos interessados, e não o contrário, sendo este um grande campo de pesquisa ainda pouco explorado pela Engenharia de Software.

Figura 7 – Visualização de Cheiros de Código

Alguns cheiros de código representados por (PARNIN et al., 2008)

2.4 Visualização de Software

O ser humano é dotado de capacidades cognitivas muito grandes. Isso dá a ele a habilidade de aprender com a realidade ao seu redor e se adaptar ao meio ambiente através de seus sentidos e percepções. O raciocínio lógico e dedutivo oferece horizontes inimagináveis ao aprendizado humano, mas essa capacidade aliada ao emprego dos cinco sentidos pode acelerar esse processo. Segundo Umphress et al. (2002) o cérebro humano consegue realizar análises semânticas muito mais rapidamente quando processa informações obtidas de imagens do que informações textuais. Assim como na célebre frase "Uma imagem vale mais do que mil palavras" (MCLUHAN; PARÉ, 1968), isso ocorre porque a visão humana, desde a infância, é estimulada com o ambiente visual a sua volta. Instintivamente as formas visuais, imagens e fotos são naturalmente usadas para representar as informações e até mesmo para a comunicação.

> Visualização de Software tem as suas raízes nas práticas de desenvolvimento de software mais antigas, quando os programadores viam as luzes no painel de controle do computador e ouviam os sons de acesso ao disco para tentar entender o que o programa estava fazendo na ausência de outros sinais perceptíveis(PETRE, 2010).

A VS está preocupada com o uso de técnicas gráficas e visuais que forneçam sinais perceptíveis aos desenvolvedores, que os ajudem a esclarecer e compreender aspectos do

sistema e características inerentes ao código-fonte, implícitas durante o processo de desenvolvimento (PETRE, 2010). Cada tarefa de desenvolvimento de software pode apresentar uma necessidade específica de compreensão, o que implica em diferentes perspectivas do software e suas respectivas propriedades. Nesse sentido, quanto maior a necessidade de compreender uma arquitetura de software, mais sugestivas devem ser as representações construídas pelas ferramentas de visualização (UMPHRESS et al., 2002). O desafio então é identificar quais as visualizações mais adequadas para orientar a CS em uma determinada tarefa.

2.4.1 Estratégias de Visualização de Software

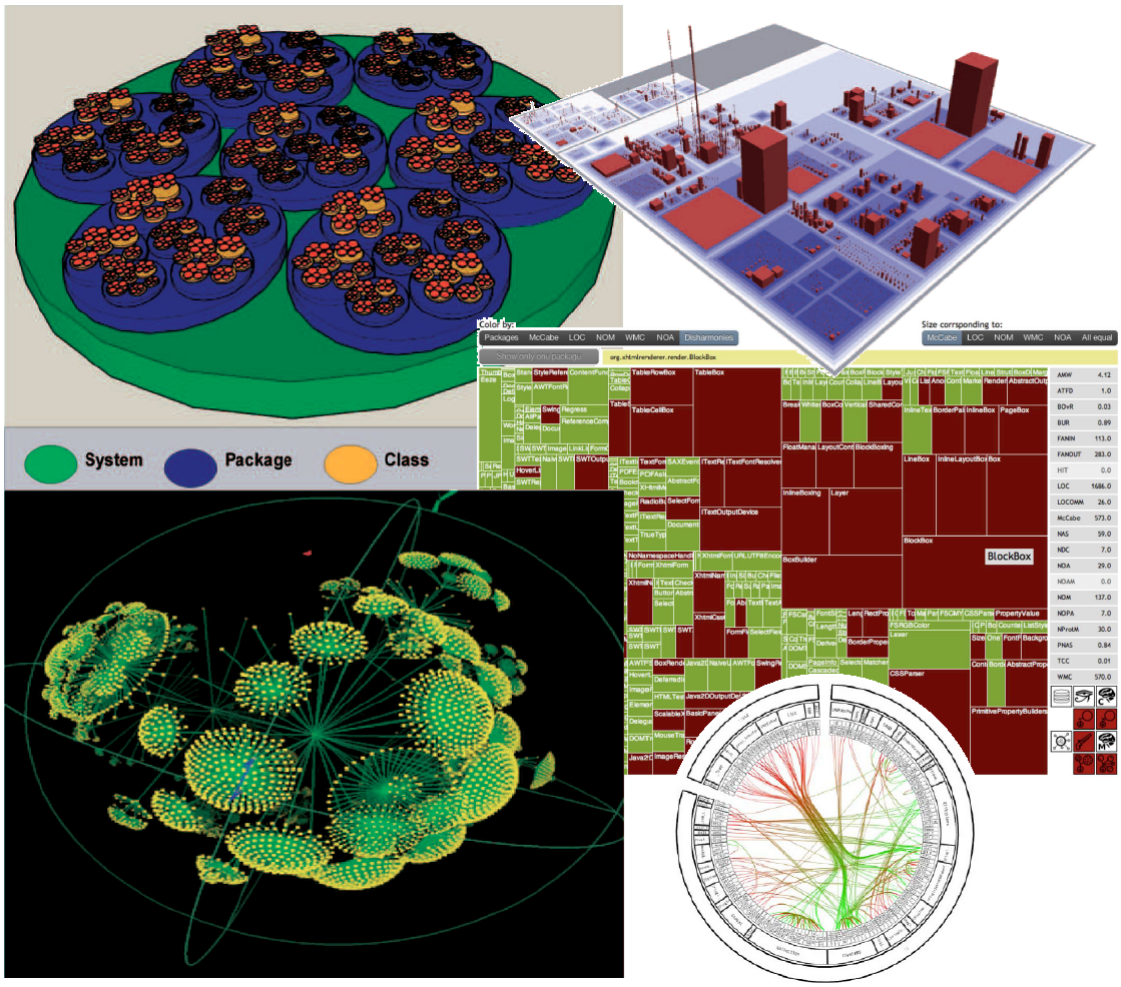
Diferentes abordagens podem ser obtidas para a VS através da Engenharia Reversa, como demonstrado na Figura 8, onde formas visuais aliadas aos padrões de cores são utilizados para sinalizar padrões e comportamentos da arquitetura do software. Canfora et al. (2011) apontou três abordagens diferentes para a análise de software, a análise estática, a análise dinâmica e a análise histórica da evolução do software, podendo ser elas combinadas em uma quarta abordagem denominada análise híbrida. O objetivo de cada análise apontada por ele, é produzir visões específicas sobre o software, através de ferramentas de Engenharia Reversa, que proporcionem ao desenvolvedor diferentes perspectivas sobre o sistema e o código-fonte.

2.4.1.1 Análise Estática de Software

O propósito da análise estática de software, é construir uma abordagem de compreensão que permita entender como se apresenta a arquitetura de software e como ela está no momento (TELEA et al., 2010). Entender como ela está significa identificar padrões de comportamento e problemas de implementação, percebidos e apontados através do código-fonte em uma determinada instância do código, que sinalizem a qualidade da arquitetura do software, bem como a estrutura adotada na construção da aplicação, dos módulos e camadas. Identificar o estado da arquitetura auxilia na elaboração de comparativos entre a arquitetura encontrada, e a arquitetura desejada (ideal), de forma a avaliar a necessidade de implementar possíveis refatorações (TELEA; VOINEA, 2011).

A análise estática do software dá o enfoque para um determinado fato extraído do código-fonte, e procura abrir espaço para diferentes visualizações, de maneira a enxergar o software através de perspectivas que revelem diferentes características (TELEA et al., 2010). Para isso, ferramentas de análise estática buscam adotar técnicas de representação visual que melhor evidenciem as características encontradas no software, apontando grandezas e medidas que sinalizem características da arquitetura através de componentes em 2D e 3D (TELEA; VOINEA, 2011). Alguns exemplos disso foram apresentados na Figura 8, como por exemplo, a representação cityview, que mostra pacotes como vizinhan-

Figura 8 – Diferentes abordagens de Visualização de Software



Diferentes representações gráficas de projetos de software através de árvores de nodos, visualizações em forma de cidades (TEYSEYRE; CAMPO, 2009), quadros de classes e pacotes (GHEZZI; GALL, 2013) e grafos de chamadas (TELEA; VOINEA, 2008)

ças e classes como edifícios de uma cidade, utilizando as medidas visuais para representar medidas de software.

Existem muitos desafios para que ferramentas de análise de código sejam adotadas como artefatos de desenvolvimento nas equipes de desenvolvedores. Pensando em ferramentas de análise estáticas, subentende-se que a visualização seja de fácil acesso durante o desenvolvimento, para que possa ser consultada sempre que for necessário. Nesse contexto, Telea et al. (2010) identificou uma profunda necessidade de ferramentas fortemente integradas às IDE's de desenvolvimento utilizadas, e que sejam escaláveis e navegáveis o suficiente para permitir manipulação dos recursos visuais durante a implementação do código, sem altos custos em processamento.

Esse desafio não é trivial, pois integrar a VS com ambientes de desenvolvimento,

requer ao mesmo tempo análises que façam sentido às diversas necessidades de compreensão dos desenvolvedores, em uma velocidade que contribua com as dinâmicas de desenvolvimento consolidadas. Segundo Umphress et al. (2002) as construções gráficas devem complementar o código-fonte sem afetar o desenvolvimento, servindo como um prolongamento natural do código. Comparativamente, a interação com as construções gráficas propostas pela visualização devem ser um prolongamento natural da interação com o código-fonte.

2.4.1.2 Análise Dinâmica de Software

Análise dinâmica de software, segundo Cornelissen et al. (2009) é a análise das propriedades de um software durante a sua execução e sobre o controle do desenvolvedor. A análise dinâmica expõe entidades e atributos do código-fonte manipulado durante a execução do software, e com acesso aos objetos em memória, também permite expor comportamentos reais do software em tempo de execução, para levantar uma maior compreensão sobre o seu comportamento (LIEBER, 2013). Existem diversas vantagens e desvantagens para a análise dinâmica sobre a análise estática, visto que deixa o processo de compreensão mais versátil, dinâmico e interativo, porém limitado apenas ao domínio do conhecimento processado nos comportamentos executados.

A ideia do dinamismo é permitir que o software em execução seja analisado passo a passo, apontando características que dificilmente seriam identificadas estaticamente. Segundo Cornelissen et al. (2009) inicialmente a análise dinâmica do software era usada apenas para testes, depuração de código e análise de desempenho. Isso significa visualizar a estrutura de um objeto em memória, quantas vezes um método foi chamado, custos de processamento de um determinado comportamento do software, bem como enxergar o fluxo sendo percorrido pelo software durante o funcionamento. Isso dá ao desenvolvedor um certo nível de controle sobre a aplicação, dando liberdade para que ele possa investigar o software, como no caso de ferramentas de depuração (LIEBER, 2013). Entretanto há um campo muito grande para pesquisas relacionadas à construção de interatividade com representações visuais em tempo de execução.

2.4.1.3 Análise Histórica da Evolução de Software

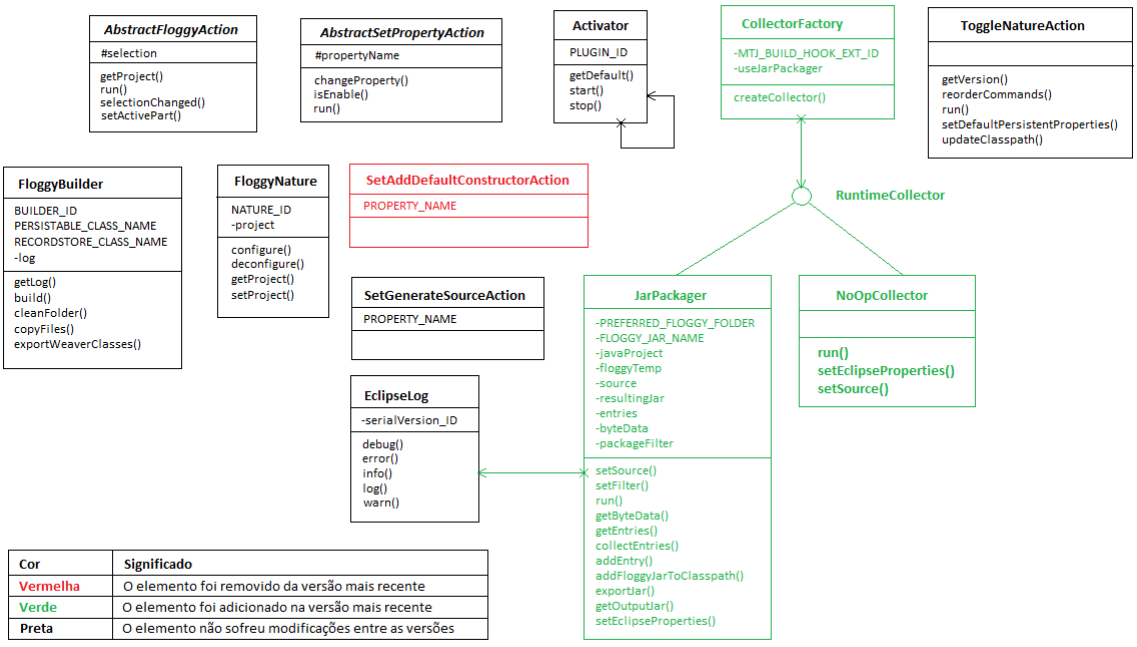
> Análise da evolução de software se preocupa com as mudanças de software, suas causas e seus efeitos. Ela usa todas as fontes de um sistema de software para realizar uma análise histórica e retrospectiva. Esses dados compreendem os históricos de *releases*, com todo o código-fonte, as informações de alterações e os relatórios de erros (D'AMBROS et al., 2008).

À medida que o software vai sendo desenvolvido, invariavelmente ele vai passando por diversos estados evolutivos, que podem ser mapeados através de ferramentas de con-

trole de versão como SVN, CVS, Git e Mercurial. Muitos desenvolvedores podem dar manutenção ao mesmo código, implementando novas funcionalidades, alterando ou removendo entidades, registrando com isso, históricos que demarcam a maneira como o software foi desenvolvido e quais direções foram tomadas pela equipe de desenvolvimento (D'AMBROS et al., 2008).

Para compreender a evolução histórica do software, é preciso entender cada versão registrada nos repositórios, e as mudanças realizadas ao longo do tempo, bem como decisões tomadas. Compreender as diferentes versões de um software pode permitir a construção de análises mais elaboradas sobre a evolução do software que, como na Figura 9, comparem os diferentes estados do software a fim de identificar padrões e características implementadas no projeto ao longo do tempo. Comparar uma versão do software com outra pode resultar em levantamentos históricos sobre clonagem de código, complexidade ciclomática e cobertura dos testes, mapeados a cada modificação.

Figura 9 – Visualização da Evolução de Software



Comparação entre duas versões de um determinado projeto de software (OLIVEIRA, 2011)

2.4.2 Técnicas de Visualização de Software

Algumas técnicas de visualização podem contribuir durante o processo de construção da CS. Segundo Oliveira (2011), o poder da visualização só passa a ser plenamente compreendido quando a visualização se torna parte integrante das atividades do usuário.

Para isso a proposta de visualização precisa ser suficientemente útil a ponto de agregar relevância ao desenvolvimento de software. As informações apresentadas na visualização devem atender a um determinado propósito, e produzir uma experiência visual que auxilie nesse propósito. Cada técnica segue princípios de *design* que podem orientar, durante a construção da visualização, a escolha das técnicas que melhor representam os conceitos que precisam ser visualizados e compreendidos.

a) *Zoom*: permite que o tamanho dos elementos visuais possa ser ampliado ou reduzido, de maneira a permitir uma melhor exploração de detalhes, permitindo em alguns casos representações semanticamente mais detalhadas à medida que o *zoom* vai sendo ampliado.

b) Filtragem: permite exibir apenas aquilo que é relevante para a construção de um determinado raciocínio ou compreensão, realçando aquilo que é importante, e suavizando ou ocultando aquilo que não é relevante ou atrapalha a CS.

c) Agrupamento: permite agrupar e dividir um grande conjunto de dados em subconjuntos, mediante determinadas características de similaridade, podendo ajudar a identificar partes do software fortemente interconectadas.

d) Hierarquias: permite exibir hierarquias de arquivos, módulos e classes, por meio de representações de árvores hierárquicas divididas recursivamente em áreas e subáreas.

e) Detalhamento: permite navegar através de hierarquias, e revelar mais detalhes visuais conforme a necessidade cognitiva do usuário, expandindo camadas do software ao selecionar nós e subnós em uma árvore hierárquica.

f) Foco + Contexto: permite realçar um determinado foco de informações sobre o software, mantendo um contexto que auxilie na construção de uma determinada compreensão do software.

g) Visão Geral + Detalhe: permite a exibição simultânea de uma visão geral e uma visão mais detalhada de um mesmo conjunto de informações, que permita interagir separadamente com cada uma delas.

2.5 Fechamento

Durante este capítulo foi possível identificar alguns problemas relacionados à necessidade de compreender software. Quando o código é a única fonte confiável de informações sobre o software, a engenharia reversa aliada a VS podem ser uma alternativa bastante apropriada para apoiar os processos de CS. Porém os diversos desafios relacionados à análise de código e arquitetura de software apontam para uma necessidade de construir um modelo de extração que seja ao mesmo tempo dinâmico e escalável. A experiência de

visualizar software deve ser intuitiva e apresentar uma interatividade tal, que se torne um procedimento comum ao desenvolvimento de software, com propostas visuais que induzam a CS. Nesse sentido as ferramentas de visualização devem ser simples, fáceis de usar, integradas ao ambiente de desenvolvimento de software, versáteis e dinâmicas o suficiente para acompanhar as diferentes necessidades de desenvolvimento de software. Para isso identificar abordagens que melhor respondam a essas necessidades pode ser crucial para alcançar um bom resultado.

3 Trabalhos Relacionados

Neste capítulo são apresentados os procedimentos e resultados de um trabalho de revisão sistemática, que resultou em um conjunto de trabalhos relacionados que compõem o estado da arte sobre ferramentas e modelos de extração de fatos para a compreensão e visualização de software. Na seção 3.1 é detalhado o processo metodológico implementado para pesquisar, catalogar, identificar e selecionar trabalhos relacionados com a abordagem proposta neste trabalho. Na seção 3.2 são resumidamente descritos cada um dos trabalhos selecionados durante a revisão, expondo seus objetivos e resultados obtidos. Na seção 3.3 é realizada uma análise sobre os trabalhos selecionados, comparando propostas e resultados entre si, para justificar decisões e abordagens tomadas durante a implementação do trabalho. Por último, na seção 3.4 é realizada uma análise das contribuições da revisão sistemática realizada para os objetivos definidos para o trabalho.

3.1 Metodologia de Pesquisa

Para levantar o estado da arte sobre ferramentas e modelos de extração de fatos para a compreensão e visualização de software, foi realizada uma revisão sistemática do assunto, baseada no estudo sobre revisão sistemática e evidência científica, realizado por Sampaio e Mancini (2007). O portal da Capes foi definido como base para as consultas, e três *strings* de busca foram selecionadas para uma pesquisa inicial. A primeira *string* foi definida como (‘Software Visualization’ ‘database’ ‘extraction’), a segunda como (‘Software Comprehension’ ‘database’) e a terceira (‘fact extractor’). Alguns critérios iniciais foram estabelecidos, de maneira que os trabalhos encontrados deveriam ser artigos acessíveis, publicados em periódicos da computação há no máximo 5 anos e que respeitassem um número mínimo de 6 páginas.

A primeira *string* de busca resultou em 76 artigos, de diversos assuntos envolvendo software e visualização, porém apenas 43 destes resultados abordavam problemas relacionados à computação. A segunda *string* resultou em mais 41 trabalhos publicados em periódicos da computação. Por último, a terceira *string* resultou em 77 artigos. De todos os 194 artigos encontrados na primeira triagem, 55 foram selecionados para uma segunda triagem, respeitando também para cada um deles, o número de páginas e o tempo de publicação estipulados. Nessa etapa foram estipulados 3 novos critérios de seleção, para orientar a escolha dos trabalhos relacionados. Para isso foram analisados cada um dos artigos selecionados, de maneira a identificar qual o contexto e o objetivo de cada um deles. O primeiro critério de seleção definido era identificar se o artigo analisado se tratava de um trabalho de CS ou VS. O segundo critério pretendia garantir que o trabalho propunha

alguma ferramenta específica de CS ou VS. O terceiro e último critério, propunha-se a verificar se o trabalho oferecia alguma modelagem para a extração de dados do código-fonte. Os dois últimos critérios se tornaram decisivos para identificar e apontar se um trabalho contribuiria ou não com a abordagem deste trabalho.

Dos 55 trabalhos selecionados para a segunda triagem, 23 foram identificados como trabalhos relacionados com CS e VS. Destes, apenas 10 propunham alguma ferramenta específica ou apresentavam alguma modelagem para solucionar o problema da extração de dados. Por último, ao analisar algumas referências encontradas nos artigos selecionados, foi identificada a necessidade de apontar também algumas abordagens clássicas da VS e da Engenharia Reversa, que apresentaram propostas e modelos que são adotados em muitos trabalhos atuais.

3.2 Resultados da Pesquisa

Os resultados da pesquisa foram organizados em três categorias de trabalhos relacionados distintas. A primeira categoria é formada por trabalhos que apresentaram alguma ferramenta, modelo ou solução na área da extração de fatos para a análise estática do código-fonte, que visa obter retratos estáticos do estado de um determinado projeto de software. A segunda categoria envolve trabalhos relacionados com a análise histórica do código, apresentando ferramentas ou modelos de extração preocupados com a evolução do software ao longo do tempo. Por último, também foram identificados alguns trabalhos seminais, que serviram como base para a elaboração de vários dos trabalhos relacionados anteriores, e ajudaram a formar uma terceira categoria de trabalhos clássicos para a CS e VS. Todos os trabalhos foram resumidos e registrados a seguir, relacionando cada um deles com as suas respectivas categorias.

3.2.1 Análise Estática de Código

A proposta de Santos e Barros (2009) tem como objetivo oferecer uma representação de código-fonte em formato XML, denominada CodeMI, para descrever elementos estruturais de projetos orientados a objetos, independente da linguagem de programação. A representação proposta utiliza o formato *XML Metadata Interchange* (XMI) para representar conceitos como classes, atributos e métodos, de maneira a permitir um dinamismo na extração de medidas do sistema, sem evidenciar questões pontuais de lógica da programação. Sendo o XMI um formato de intercâmbio de meta dados também usado na integração entre modelos UML, garante à representação um alto grau de reuso e extensibilidade. Para a extração das medidas foram utilizadas transformações *Extensible Stylesheet Language Transformations* (XSTL), que descrevem regras de transformação de uma árvore XML em outra a partir da associação de padrões a modelos.

Em seu trabalho, Mäder e Cleland-Huang (2012) apresentaram um projeto que aborda e define estratégias de rastreabilidade de sistemas, utilizando diagramas de classe e conceitos UML para a compreensão e elaboração de consulta *Structured Query Language* (SQL). Os modelos são utilizados como uma alternativa para orientar a construção de consultas de rastreabilidade, a fim de definir cláusulas e restrições de consulta de maneira mais intuitiva. A *Visual Trace Modeling Language* (VTML) foi projetada para aliviar as dificuldades de escrever consultas de rastreio e para tornar os seus benefícios mais acessíveis aos desenvolvedores e *stakeholders* do projeto. Eles descobriram que os usuários lêem e constroem complexas consultas de rastreabilidade consideravelmente mais rápido e corretamente usando VTML do que usando SQL.

SolidFX apresentado por Telea et al. (2009) é um ambiente de apoio à qualidade e sustentabilidade de grandes bases de código C/C++. A ferramenta SolidFX foi especificada como uma ferramenta de análise de código, extração de fatos e cálculo de medidas através de engenharia reversa integrada. A solução segundo eles, oferece suporte à análise visual e interativa dos resultados, permitindo lidar de maneira escalável com bases de milhões de linhas de código. Ela integra as várias funcionalidades de visão das medidas, através de uma interface única e completa que apresenta tanto visões do código, quanto gráficos, relatórios, UML e exportação para vários formatos. A maior necessidade porém, ainda é de um *parser* mais eficiente que ofereça um acesso mais detalhado aos dados da base de fatos.

Em sua Tese de Mestrado Tshering (2010) propôs uma plataforma de Evolução e Visualização de Arquitetura de Software denominada Fraunhofer *Software Architecture Visualization and Evaluation* (SAVE), que funciona como um analisador de código-fonte Ruby. A ferramenta foi desenvolvida como um conjunto de *plug-ins* integrados ao Eclipse, uma ferramenta de extração de fatos, que extrai informações arquiteturais baseadas no processamento da AST obtida pela análise do código-fonte. Uma prova de conceito foi desenvolvida para aplicar a solução e permitir uma avaliação mais profunda em torno da abordagem. A ferramenta contribuiu como um primeiro extrator de fatos para código-fonte desenvolvido em Ruby on Rails, um framework estruturado no padrão *Model-View-Controller* (MVC), criado para aumentar a produtividade e minimizar barreiras da programação.

3.2.2 Análise Histórica de Código

Em sua abordagem, D'Ambros et al. (2008) propuseram um modelo de dados que descreve sistemas com base na identificação e interpretação de arquivos de *log* de versionamento e relatórios de *bugs* obtidos durante os processos de evolução de software. Os fatos são extraídos e armazenados em uma base histórica de releases ou *Release History Database* (RHDB), para serem analisados e gerarem um determinado nível de compreensão

do software. Ao se referir à história de um artefato de software, está implícito entender a forma como foi desenvolvido, como cresceu ou diminuiu ao longo do tempo, e quais desenvolvedores foram responsáveis pelo seu processo evolutivo. Mas, mais do que isso, a análise da evolução do software espera compreender a arquitetura do sistema, bem como as dependências entre os seus componentes, a fim de detectar áreas críticas do software que precisam receber alguma atenção especial.

Mohan et al. (2008) desenvolveram um modelo de rastreabilidade que representa elementos de conhecimento essenciais para gerir de maneira compreensível alterações da gestão de mudança, com suporte à práticas integradas de rastreabilidade e gerenciamento de configuração de software ou *Software Configuration Management* (SCM). Enquanto o SCM ajuda na evolução do sistema, bem como na gestão, controle e execução das mudanças, a rastreabilidade ajuda no gerenciamento de dependências entre artefatos relacionados em todo e em diferentes fases do ciclo de vida do desenvolvimento. O modelo apresentado visa qualificar o gerenciamento de configuração, através de recursos de rastreabilidade para melhorar a gestão de mudanças. Por exemplo, ao rastrear mudanças em um sistema de controle de versão, o conhecimento adquirido pode ajudar a equipe a entender as decisões tomadas ao longo do tempo, e o impacto das mudanças nos artefatos de software.

A ferramenta Churrasco proposta por D'Ambros e Lanza (2010), é voltada para apoiar sistemas de software de grande porte, com análise e evolução de software colaborativa. O objetivo da ferramenta é realizar uma análise retrospectiva que apoie operações de manutenção e oriente previsões sobre a evolução do software no futuro. Essas operações envolvem 1) recuperação de dados de repositórios de código, 2) análise dos dados brutos para extrair fatos relevantes e 3) população dos modelos como base para análises reais sobre a evolução do software. A ferramenta proposta deve apresentar um modelo flexível para englobar diversas fontes de informação, acessibilidade, armazenamento incremental dos fatos extraídos para permitir comparação entre as análises, e suporte à colaboração com outras ferramentas.

Em seu trabalho, Tappolet et al. (2010) introduziram um conjunto de ontologias denominado EvoOnto, com o objetivo de representar informações semânticas de projetos de software para cobrir os problemas cotidianos relacionados com a CS. Eles propõem o uso de tecnologias para a web semântica, como *Web Ontology Language* (OWL), *Resource Description Framework* (RDF) e SPARQL, como uma forma de interligar ferramentas, componentes, *frameworks* e projetos de software. A estratégia do EvoOnto é utilizar a web semântica para estruturar e publicar conhecimento sobre software, suas funcionalidades, versões e relatórios de erros. Isso permite que análises mais dinâmicas sejam realizadas entre os diferentes projetos de software com alguma dependência entre si. Eles conseguiram, através da abordagem proposta, reduzir mais de 75% das tarefas realizadas

durante o processo de análise da evolução de software.

Software Evolution Ontologies (SEON) é um conjunto de ontologias de evolução de software, desenvolvidas por Würsch et al. (2012), através de meta modelos extensíveis baseados no modelo FAMIX, para descrever explicitamente as relações entre as estruturas de código, solicitações de mudança, *bugs* e basicamente todas as alterações feitas em um sistema ao longo do tempo. O propósito é descobrir como descrever adequadamente o conhecimento de evolução do software, fornecendo uma ferramenta eficaz de Engenharia de Software que possa apoiar à gestão de sistemas de software e todo o seu ciclo de vida. Isso inclui o conhecimento sobre as partes interessadas, atividades, artefatos e as relações entre todos eles. O caráter extensível dos meta modelos facilitou a reutilização durante a construção de ferramentas com aplicações bem específicas voltadas à web semântica e CS.

Em seu trabalho, Ghezzi e Gall (2013) propuseram uma abordagem para o problema de integração e compartilhamento dos dados de diferentes análises, de maneira a permitir uma compreensão histórica mais aprofundada do software. Eles criaram uma arquitetura RESTfull denominada SOFAS (GHEZZI; GALL, 2011), que introduz o conceito de Análise de Software como um Serviço (*Software Analysis as Service*) com o objetivo de integrar análises de um sistema de software, permitindo uma interoperabilidade leve entre as análises sem preocupação com os limites geográficos. A arquitetura proposta utiliza um conjunto de ontologias SEON, com uma estrutura que permite criar, acessar e integrar serviços de análise de software em três níveis de interesse, abrangendo 1) Coleta, extração e importação de dados, 2) Análises básicas de medidas da evolução do software e 3) Compreensão aprofundadas da evolução do software através de análises combinadas.

3.2.3 Abordagens Clássicas

A abordagem de Tichelaar et al. (2000) apresentou um mecanismo de suporte à refatoração de código independente de linguagens de programação, com o objetivo de integrar diferentes refatorações em um ambiente de reengenharia. Foi construído um modelo de representação de código-fonte orientado a objetos, denominado FAMIX (DUCASSE et al., 2011), pertencente à família de ferramentas MOOSE (DUCASSE et al., 2000) e FAMOOS (BÄR; DUCASSE, 1999), para armazenar informações que suportem a maioria das refatorações existentes.

Os autores Müller e Klashinsky (1988) descreveram em seu trabalho, um modelo e uma ferramenta gráfica chamada Rigi, com mecanismos de abstração, com a finalidade de estruturar e representar através de grafos, informações acumuladas sobre sistemas de grande porte, seus componentes e suas dependências. Em seu trabalho, Ferenc et al. (2002) apresentou um *framework* de engenharia reversa denominado Columbus, capaz de analisar grandes projetos C++, através de uma arquitetura flexível formada por um

conjunto de ferramentas versáteis e extensíveis. O objetivo é suprir, através de uma ferramenta de análise de sistemas de software, a necessidade de compreender as diversas partes de um grande sistema de software e sua complexidade, de maneira a contribuir para a manutenção e o desenvolvimento de software.

Graph Exchange Language (GXL) é um formato XML baseado em grafos proposto por Holt et al. (2000), que tem como objetivo apoiar a interação entre ferramentas de extração, manipulação e análise de software, definindo um formato padrão para integração e troca de informação de software. Já Badros (2000) apresenta JavaML, uma aplicação destinada a descrever programas com código Java, através de uma estrutura XML, com elementos que podem ser combinados para representar entidades da linguagem e do código-fonte. Collard et al. (2003) por sua vez apresenta srcML uma ferramenta leve de extração de fatos baseada em XML, com o objetivo de extrair informações estáticas de código C++ e permitir todo tipo de consulta usando XML.

3.3 Análise dos Trabalhos Relacionados

Ao analisar os trabalhos selecionados, foram identificadas abordagens comuns entre os trabalhos estudados. Essas abordagens foram enumeradas em tópicos, que permitiram delinear características, semelhanças e diferenças entre os trabalhos, comparando e relacionado-os entre si. As abordagens usadas foram listadas da seguinte maneira:

a) XML como estrutura de persistência e consulta aos dados

b) SQL e entidade-relacionamento como estrutura de persistência e consulta aos dados

c) SPARQL e OWL como estrutura de persistência e consulta aos dados

d) Uso de recursos para obtenção de rastreabilidade de software

e) Ferramentas de extração interlinguística de fatos de software

f) Ferramentas de extração de fatos para linguagens de programação específicas

g) Modelo de extração documentado

Nessa revisão sistemática foi possível identificar uma variedade grande de usos e aplicações para a extração de fatos, que foram catalogadas através da Tabela 1. Um exemplo é a obtenção de medidas de rastreabilidade apresentadas tanto na VTML de Mäder e Cleland-Huang (2012) quanto na abordagem de Mohan et al. (2008). Outras abordagens para a extração foram apresentadas por Telea et al. (2009) em SolidFX e por Tshering (2010) na plataforma SAVE, úteis para a compreensão e identificação de questões arquiteturais dos projetos, e ainda algumas abordagens destinadas a identificar relações históricas da evolução do código-fonte, como apresentado por Ghezzi e Gall (2013) na arquitetura SOFAS e na ferramenta Churrasco proposta por D'Ambros e Lanza (2010).

Outras relações foram encontradas durante o estudo, relacionadas com as diferentes propostas de modelagem apresentadas, dedicando-se a uma abordagem específica, como nas ontologias apresentadas por Tappolet et al. (2010) em EvoOnto, e por Würsch et al. (2012) em SEON, ou dedicando-se a alguma proposta interlinguística usando XML, como é o caso das propostas GXL e CodeMI idealizadas por Holt et al. (2000) e Santos e Barros (2009). Algumas abordagens que propuseram modelos restritos a uma linguagem específica, foram também identificadas como válidas, visto que podem atender algumas demandas específicas de linguagem de programação. São elas as linguagens de marcação propostas por Badros (2000) JavaML e Collard et al. (2003) srcML, que se propõem a construir modelos mais fiéis ao código-fonte original.

Tabela 1 – Comparativo entre os Trabalhos Relacionados

Trabalhos Selecionados	**a)**	**b)**	**c)**	**d)**	**e)**	**f)**	**g)**
Famix (TICHELAAR et al., 2000)					x		x
Rigi (MÜLLER; KLASHINSKY, 1988)							
Columbus (FERENC et al., 2002)	x					x	x
GXL (HOLT et al., 2000)	x				x		
JavaML (BADROS, 2000)	x					x	
srcML (COLLARD et al., 2003)	x					x	
CodeMI (SANTOS; BARROS, 2009)	x	x			x		
VTML (MÄDER; CLELAND-HUANG, 2012)				x			
SolidFX (TELEA et al., 2009)	x					x	
SAVE (TSHERING, 2010)	x					x	x
RHDB (D'AMBROS et al., 2008)		x					
SCM (MOHAN et al., 2008)				x			
Churrasco (D'AMBROS; LANZA, 2010)		x			x		
SOM (TAPPOLET et al., 2010)	x		x			x	x
SEON (WÜRSCH et al., 2012)	x		x		x		x
SOFAS (GHEZZI; GALL, 2013)	x		x		x		

Comparativo realizado entre as abordagens adotadas pelos trabalhos relacionados

Assim como se pode perceber através da Tabela 1, a maioria dos trabalhos relacionados propuseram abordagens utilizando formatos baseados em XML para modelar os fatos extraídos do código-fonte, por motivos que variam entre atribuir flexibilidade aos modelos construídos, extensibilidade e interoperabilidade com outras ferramentas. Mas uma avaliação inicial indicou que uma abordagem utilizando bases de dados relacionais, traria à ferramenta de CS um desempenho aprimorado, principalmente quanto às análises históricas complexas da evolução do software. Um aumento considerável no ganho de processamento permitiria, por exemplo, dinamizar o processo de construção e execução de consultas no banco de dados, e permitir análises de VS mais rápidas e interativas.

Com base nos estudos realizados foi decidido aplicar uma solução com uma abordagem fundamentada na arquitetura de bancos de dados relacionais, e na linguagem de consulta SQL, que ainda não tiveram todos os seus recursos e características de performance, explorados pela comunidade de CS e VS. Foi escolhido utilizar a modelagem

entidade-relacional, e os diversos modelos encontrados nos trabalhos, para ajudar a implementar um modelo convergente que atenda às diferentes necessidades de compreensão e visualização apresentadas pelos modelos estudados. O modelo elaborado deve atender necessidades de compreensão apresentadas em diferentes linguagens de programação orientadas a objetos disponíveis no mercado, assemelhando-se a algumas das abordagens interlinguísticas encontradas.

3.4 Fechamento

Neste capítulo foi apresentada a revisão sistemática realizada para estabelecer os trabalhos relacionados à extração de fatos e a análise da evolução de software. Através do estudo realizado foi possível identificar abordagens recentes e clássicas para a CS e VS, com modelos de extração idealizados para diversas finalidades. A revisão sistemática, além de aprofundar os conhecimentos relacionados com a temática envolvida, permitiu confirmar nos trabalhos, opiniões em comum sobre os diversos problemas percebidos no campo da CS e VS atual. Esse entendimento contribuiu para consolidar os objetivos inicialmente propostos, mostrando através de outros trabalhos a necessidade de solucionar alguns dos problemas encontrados nas áreas das análises estática e evolutiva do software.

4 PF Factfinder

Neste capítulo é apresentada a solução implementada para atender as necessidades de extração de fatos apontadas anteriormente. Na seção 4.1 é apresentada uma visão geral da solução proposta, reapresentando o problema e apontando algumas decisões de projeto e tecnológicas. Na seção 4.2 são estudados e catalogados alguns modelos de extração encontrados durante os estudos realizados, de maneira a propor um modelo comum que atenda à maioria das necessidades apontadas pelos modelos analisados, tomando decisões quanto à escolha de cada uma das entidades modeladas. Na seção 4.3 é apresentado uma proposta arquitetural para o problema proposto no trabalho, apontando algumas ferramentas que foram utilizadas para a implementação da extração e persistência dos dados, bem como propondo um mapeamento para a árvore de sintaxe abstrata. Na seção 4.4 é proposto um plano de medição com o objetivo de verificar se a ferramenta atende as necessidades de visualização apontadas pelos modelos de extração estudados. O processo de medição visa apontar se os dados gerados durante a extração implementada podem ser realmente usados em análises de projetos orientados a objetos. Na seção 4.5 foi realizado um estudo de cada uma das medidas de código obtidas durante a elaboração do processo de medição, com o intuito de orientar a implementação dos serviços responsáveis por calcular e obter as medidas de código. Por fim, na seção 4.6 é realizado um fechamento com uma análise rápida sobre os resultados do estudo realizado e a implementação da ferramenta.

4.1 Visão Geral

Assim como visto nos capítulos anteriores, um dos maiores problemas relacionados com a CS e VS, é o alto custo envolvido na complexidade do processamento das análises de software, combinadas para aprofundar o nível de compreensão durante a visualização de grandes projetos de software. Segundo Anslow et al. (2004), dependendo da quantidade de informações extraídas e análises realizadas, esse custo pode chegar a gigabytes de uso em memória, fato que indica algumas limitações para o alcance e a versatilidade das análises de CS. Análises mais complexas, que extraem informações de milhares de linhas de código, podem se tornar tão extensas e demoradas a ponto de dificultar o seu reaproveitamento dentro dos processos cotidianos de desenvolvimento de software e na interação com o desenvolvedor.

Para atender esta necessidade, foi decidido adotar uma abordagem de persistência em banco de dados relacional, que ofereça maior agilidade e versatilidade no processamento das consultas necessárias para a análise dos dados. Para a arquitetura de extração,

se decidiu separar as responsabilidades entre a extração, compreensão, análise e posteriormente até mesmo a visualização apresentada. A estratégia escolhida foi distribuir rotinas específicas para processar a AST nos diferentes níveis de informação, complexidade e etapas de processamento, evitando guardar grandes quantidades de informação em memória e persistindo os dados obtidos em cada uma das etapas.

A estratégia de persistência adotada foi elaborada com o apoio da arquitetura *Java Persistence API* (JPA), que permitiu implementar de maneira simples o mapeamento objeto relacional, entre as entidades do modelo e as tabelas do banco de dados. O banco de dados escolhido inicialmente foi o banco de dados MySQL, por ser um banco de dados leve e de fácil manutenção, mas através da arquitetura JPA, seria possível adaptar a ferramenta facilmente para a utilização de outros bancos de dados existentes. Os recursos da SQL foram utilizados tanto para o armazenamento quanto para a consulta das informações extraídas do código-fonte, implementando serviços que contribuam para a compreensão e evolução do software. Através dos recursos SQL utilizados, algumas medidas de código foram processadas, elaborando consultas aos dados armazenados, e disponibilizando os resultados dos cálculos para análises mais profundas sobre os projetos de software orientado a objetos.

4.2 Modelo de Extração de Fatos

Para construir uma base de dados estável e pronta para armazenar dados de diferentes projetos de software, foi necessário estudar modelos de extração apresentados nos diferentes trabalhos relacionados encontrados no Capítulo 3. Os trabalhos relacionados apresentaram diferentes estratégias para a modelagem de dados mapeados para a extração de fatos de projetos de software orientados a objetos. O estudo sobre eles permitiu identificar características comuns que ajudariam a levantar um conjunto bem definido de entidades de código que represente de maneira simples e completa as necessidades de compreensão apontadas pela maioria dos trabalhos. Juntamente com as necessidades de extração e modelagem identificadas, foram apontados por vários trabalhos estudados, problemas relacionados com os altos custos computacionais em memória e processamento, necessários para as complexas análises de código realizadas durante o processo de compreensão de software.

Diante disso, optou-se por reaproveitar os recursos disponibilizados nos bancos de dados relacionais e pela linguagem SQL, para otimizar o processamento em memória das análises, fornecendo uma alternativa versátil e consolidada para o acesso aos dados extraídos do código-fonte. Foi decidido convergir os diferentes modelos de dados encontrados durante a revisão sistemática, em um modelo entidade-relacionamento, que agregue os principais conceitos abordados nos trabalhos, mapeando em uma arquitetura relacional,

elementos de código-fonte de projetos de software orientados a objetos. Dado que os elementos do código-fonte seriam processados através de objetos, a arquitetura JPA demonstrou ser uma alternativa bastante apropriada para os mapeamentos objeto-relacional necessários na persistência dos dados.

4.2.1 Diferentes Modelos de Extração

Para a construção do modelo, foram identificadas e selecionadas nos trabalhos estudados, propostas de modelagem que foram explicitamente apresentadas e representadas com algum tipo de modelagem conceitual. Dado que o autor apresentava uma modelagem conceitual compreensível para representar o conjunto de dados manipulados durante a extração de fatos pela ferramenta proposta, esse modelo foi selecionado e catalogado para compará-lo com os demais modelos encontrados. Alguns desses modelos foram identificados como exercendo grande influência sobre outros, como é o caso do modelo FAMIX, que foi identificado como fonte e referência para outros trabalhos como SEON e *Software Ontology Model* (SOM). Os modelos encontrados que foram selecionados para o estudo estão listados a seguir, com as respectivas referências:

a) FAMIX: (DUCASSE et al., 2011)

b) SEON: (WÜRSCH et al., 2012) e (GHEZZI; GALL, 2011)

c) Columbus AST: (FERENC et al., 2001)

d) SAVE: (TSHERING, 2010)

e) SOM: (TAPPOLET et al., 2010)

Durante este trabalho, foram analisados diversos fatores relacionados com cada elemento da AST representado nos modelos, catalogando através da Tabela 2 cada entidade da AST identificada em cada um dos modelos estudados. As nomenclaturas usadas na representação das entidades mapeadas pelos modelos também foram registradas, bem como algumas marcações caso os conceitos relacionados com a entidade não tenham sido declarados no modelo ou não tenham sido apresentados de maneira clara.

4.2.2 Um Modelo Convergente

Após catalogar os 5 modelos, foi realizado um estudo comparativo entre eles, analisando os conceitos e entidades utilizados por cada um para representar o código-fonte. Isso foi feito com o objetivo de identificar semelhanças e divergências entre os modelos, e apontar um conjunto de características da Orientação a Objetos comuns entre os modelos. Paralelamente, foram analisadas todas as nomenclaturas adotadas na representação das entidades, com o objetivo de convergir os modelos em um modelo comum, que atenda a maioria das necessidades de CS e VS apontadas em cada um deles.

Tabela 2 – Catálogo de modelos de extração encontrados

Entidades de Código	**FAMIX**	**SEON**	**Columbus**	**SAVE**	**SOM**
Projeto	N/D	N/D	N/D	SaveProject	Project
Versão	N/D	Release	N/D	N/D	Release
Revisão	N/D	Revision	N/D	N/D	Version
Pacote	Package	Directory	N/D	FSFolder	N/D
Arquivo	N/D	File	N/D	N/D	Path
Contexto	Namespace	N/D	Namespace	SavePackage	Namespace
Classe	Class	ComplexType	Type	FSType	Class
Método	Method	Method	N/D	FSOORoutine	Method
Atributo	Attribute	Attribute	Variable	FSVariable	Attribute
Parâmetro	Parameter	FormalParameter	Parameter	N/D	FormalParameter
Acesso	Access	accesses	N/D	N/D	accesses
Invocação	Invocation	invokes	N/D	N/D	invokes
Herança	Inheritance	isSubClassOf	N/D	N/D	isSubClassOf
Referência	Reference	N/D	refersToType	FSRelation	N/D
Variável Local	LocalVariable	LocalVariable	Variable	FSVariable	LocalVariable
Variável Global	GlobalVariable	N/D	PrimSpec	FSVariable	N/D
Tipo Primitivo	PrimitiveType	ComplexType	PrimSpec	FSType	ComplexType
Anotação	Annotation Type	N/D	N/D	N/D	N/D
Enum	Enum	ComplexType	N/D	FSType	ComplexType
Comentário	Comment	N/D	N/D	N/D	N/D
Função	Function	Function	Function	FSProcedural Routine	Function
Retorno	N/D	declaredReturn	Return	N/D	isDeclaredReturn ClassOf
Bloco de Código	N/D	N/D	BlockScope	N/D	N/D
Expressão	N/D	N/D	N/R	N/D	N/D

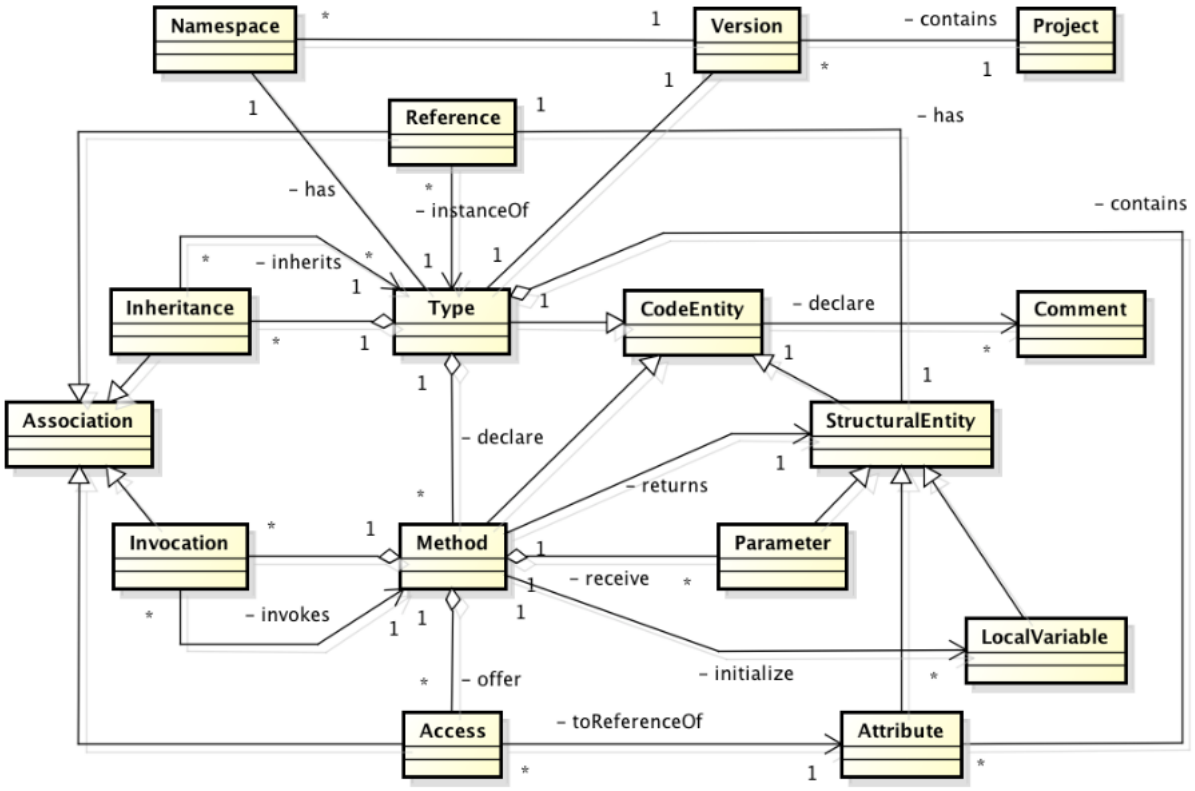
N/D: não declarado - N/R: não resolvido

Para definir as terminologias que seriam adotadas para a convergência entre os modelos catalogados, foram analisados os termos mais utilizados pelos modelos estudados, para representar cada uma das entidades listadas. Para as entidades que não possuem uma terminologia predominante entre os modelos, foi selecionada uma terminologia específica dentre os termos identificados, escolhendo de forma empírica o termo que melhor se adequasse ao conceito. O conceito de classes, por exemplo, não demonstrou uma nomemclatura única para a representação, sendo chamado por diferentes modelos como *Class*, *Type*, *ComplexType* ou *FSType*. O termo *Type* foi selecionado, porque permitiu representar, através de uma mesma nomenclatura, tanto classes, tipos primitivos de dados, como enumerações, sendo as nomenclaturas catalogadas para o conceito de classes derivadas de *Type*.

Alguns elementos como *functions* não foram utilizados, visto que se aplicam ao contexto de códigos procedurais e não à grande maioria atual de linguagens de programação Orientadas a Objetos. Outra decisão tomada foi adotar os conceitos de projeto e versão, empregados apenas por alguns dos modelos estudados, haja vista a necessidade de

mapear e compreender diferentes projetos de software em diferentes versões de produto. Não foram adotados todos os conceitos de versionamento de código e a complexidade de um repositório de versões, *commits* e revisões de código. O conceito de versão foi utilizado para representar diferentes extrações de um mesmo projeto, ou as diferentes versões de projeto fechadas ao longo do tempo.

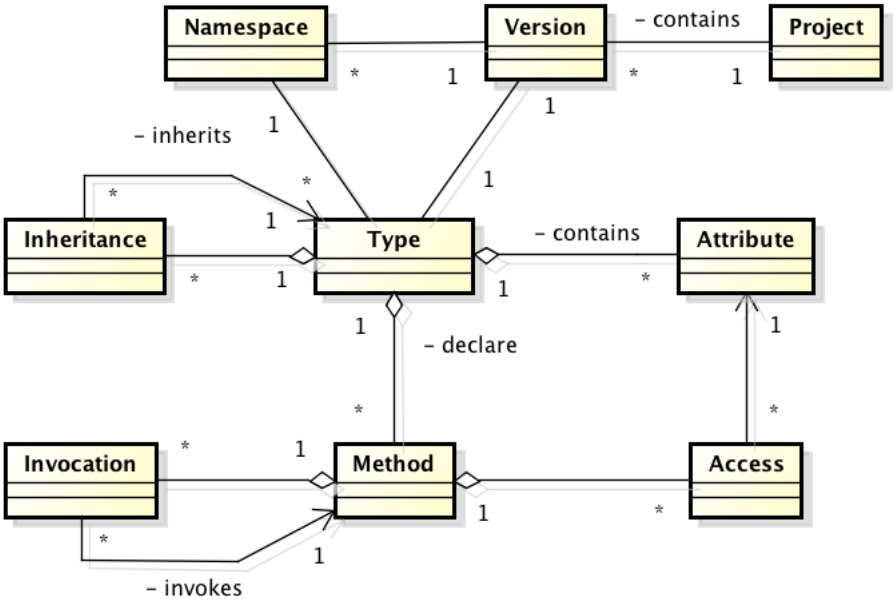
Figura 10 – PF Factfinder - Modelo de extração comum aos modelos estudados



O estudo realizado permitiu a implementação de um modelo de extração convergente, denominado *PF Factfinder*, assim como na modelagem da Figura 10, que atendesse a maioria das necessidades apresentadas pelos modelos estudados. Esse modelo orientou a construção de uma ferramenta extratora que fizesse a análise da AST, para armazenar os dados do código-fonte em uma base de dados relacional incremental.

Posteriormente o modelo convergente foi simplificado para remover algumas complexidades desnecessárias, e deixá-lo mais aderente ao conjunto de dados principal que se desejava extrair da ferramenta. Algumas hierarquias foram eliminadas, para evitar complexidades desnecessárias à modelagem do banco de dados. Algumas entidades menos relevantes foram reduzidas a campos de outras entidades, como é o caso de *Parameter* e *Comment*. Outras entidades foram simplesmente removidas, porque aumentariam consideravelmente o nível de complexidade na implementação e no tratamento dos dados

Figura 11 – PF Factfinder - Modelo de extração compactado



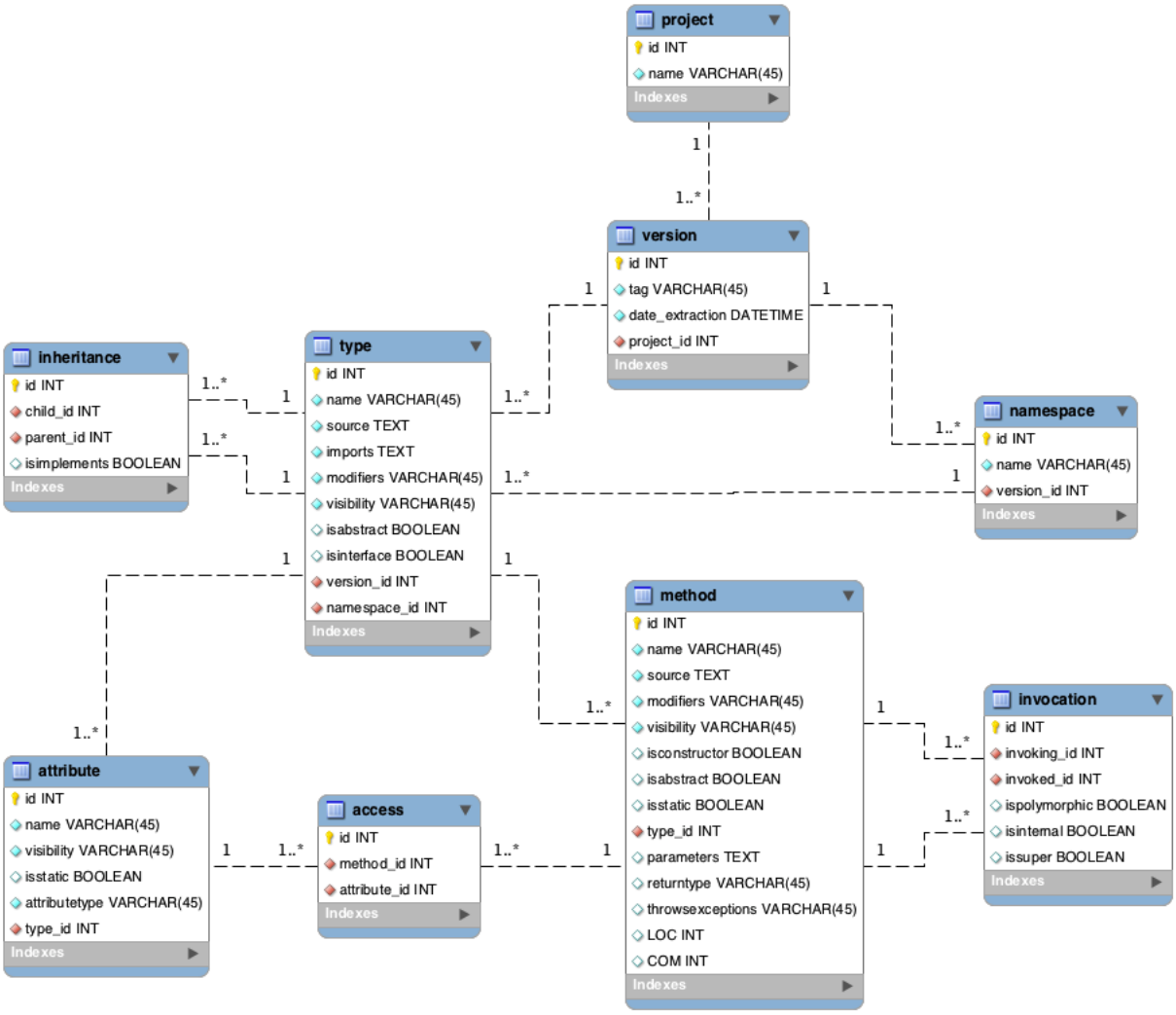
extraídos, sem agregar inicialmente grande valor ao conjunto de dados extraídos, como é o caso de *LocalVariable* e *Reference*.

4.3 Arquitetura de Extração

Apoiado em todo o estudo realizado para a construção do modelo *PF Factfinder*, iniciou-se a implementação de uma ferramenta de extração e persistência de dados para a compreensão de projetos de software orientados a objetos. Coube à ferramenta processar e mapear a AST do código-fonte, persistindo as informações obtidas em um banco de dados relacional com todo o suporte da linguagem SQL. Para isso o modelo conceitual obtido foi transformado em um modelo entidade-relacionamento apresentado através da Figura 12, que integrado a uma arquitetura JPA, permitiu definir a estrutura de dados a ser persistida.

Para ajudar no processamento da AST do Java foi utilizada uma biblioteca JDT denominada *JDT Core*, um componente do Eclipse IDE responsável pela construção e navegação da AST para códigos Java. Essa biblioteca foi utilizada nos processos de compilação do código, e nas navegações dentro da AST, buscando informações do código-fonte relevantes ao modelo, contribuindo para a implementação de toda uma arquitetura de extração de fatos. Uma visão geral dessa arquitetura pode ser vista através da Figura 13, que contém a estrutura dos componentes implementados, por exemplo, para a extração de classes de um projeto.

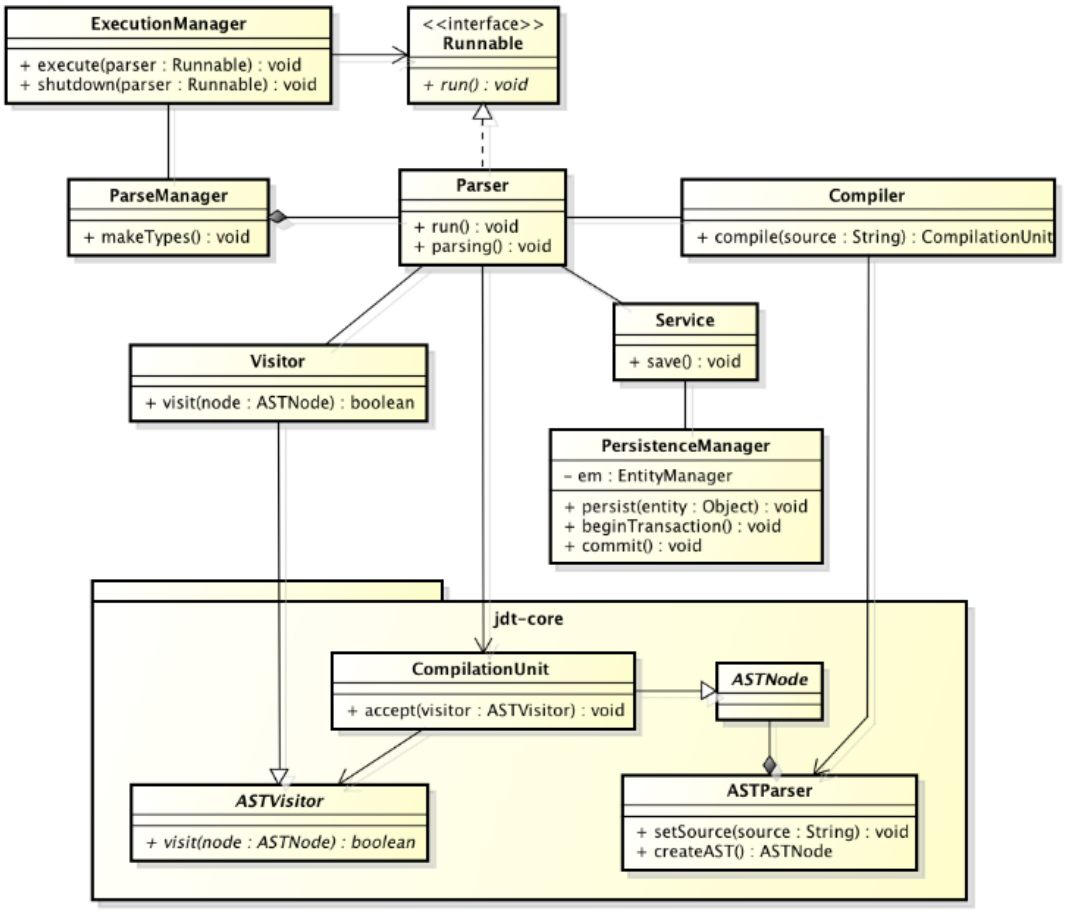
Figura 12 – Modelo Entidade-Relacionamento para o *PF Factfinder*



Durante a extração do código-fonte, os componentes de compilação do *JDT Core*, foram utilizados para transformar o código-fonte textual, em uma árvore hierárquica de nodos e objetos interconectados, representando a estrutura sintática da linguagem. Foram implementados *Parsers* responsáveis por mapearem os diferentes componentes da AST gerada pelo compilador do JDT, reorganizando as informações de acordo com o modelo de extração previamente definido. Cada *Parser* foi implementado com responsabilidades distintas, navegando pela árvore em busca de um conjunto específico de dados, transformando as informações em objetos do modelo e persistindo-as no banco de dados com a ajuda de uma implementação da JPA.

Para mapear os nodos mais relevantes da AST do código e efetuar as devidas transformações, foi adotado o padrão *Visitor* disponibilizado pela própria *JDT Core* para implementação. O *Visitor* é aceito pela árvore compilada via parâmetro de invocação, que

Figura 13 – Arquitetura de Extração



Recorte da arquitetura da ferramenta de extração implementada

utiliza suas implementações de visita, percorrendo nodo por nodo em busca de elementos específicos que tragam informações relevantes para o mapeamento do código. Cada *Visitor* implementa o comportamento *visit(ASTNode node)*, sobrecarregado por diferentes implementações para diferentes tipos de nós da AST. Ao sobrescrever os comportamentos necessários, as informações do código compilado são visitadas, processadas e reorganizadas. A Figura 14, apresenta alguns métodos das classes *JavaTypeParser*, *JavaCompiler* e *JavaTypeVisitor*, e exemplifica o processo de compilação, navegação, transformação e persistência de elementos da entidade *Type* ao longo da AST.

No início do processo de extração, a ferramenta recebe um caminho que servirá como ponto de partida para a extração de todo o projeto de software. Esse projeto e as diversas versões que podem ser extraídas em diferentes momentos, são gerenciados pela ferramenta junto ao banco de dados através de uma etiqueta (*tag*) que é incrementada a cada nova versão extraída. Todos os arquivos Java são obtidos a partir do caminho informado, com os quais são extraídos também todos os pacotes, classes, métodos, atributos,

Figura 14 – Implementação do *Visitor*

1) *Parser* envia o código-fonte para o compilador e recebe uma estrutura de dados compilada; 2) implementação do *Visitor* é recebida pela estrutura compilada, que opera a navegação através de uma árvore de nodos; 3) serviço de persistência em banco de dados; 4) Compilador transforma o código-fonte em uma AST compilada; 5) comportamento implementado da classe abstrata *ASTVisitor*, é chamado sempre que a unidade de compilação encontra na árvore uma declaração de tipo.

invocações, acessos e heranças do projeto. Para gerenciar a extração, decidiu-se adotar uma arquitetura *multithreading*, para obter maior desempenho e aproveitando os recursos de máquina disponíveis. Cada *Parser* implementa a interface *Runnable*, sendo gerenciado por um *ExecutorService*, que dispara a execução de cada processo, realizando de maneira otimizada a devida compilação do código e a transformação em entidades mapeadas pelo modelo, através do componente *Visitor*.

Métodos e Atributos são identificados, extraídos, vinculados e persistidos juntamente com o processo de extração de classes, de acordo com o modelo entidade-relacionamento da Figura 12. Em seguida são executados os *parsers* responsáveis por identificar dentro do código-fonte, heranças entre as classes, invocações a métodos internos e externos, com todas as questões polimórficas envolvidas, e finalmente os acessos a atributos via métodos. Por último, com todo o projeto extraído, também são calculadas algumas medidas de código, com enfoque em questões da orientação a objetos. A saída da ferramenta é um conjunto de dados, impressos no console através da saída padrão do Java, organizando todas as classes por pacotes, e informando para elas os atributos e métodos vinculados, bem como algumas medidas de código. Futuramente essa saída poderá ser revitalizada em uma aplicação web que apresente todos esses dados através de páginas de formulários, tabelas, listagens e gráficos que façam comparativos entre os dados e as medidas geradas.

4.4 Plano de Medição

Uma vez com a ferramenta de extração implementada, percebeu-se a necessidade de comparar os resultados encontrados pelo extrator, com dados que validem o funcionamento e a corretude dos processos de extração implementados. Para estabelecer uma base de comparação, decidiu-se comparar os dados extraídos pela ferramenta com dados reais identificados e coletados de maneira empírica, a partir de projetos de software existentes. Para isso foi decidido adotar o processo *Goal-Question-Metrics* (GQM)[1], como uma forma obter valores e medidas que atendessem diretamente às necessidades de comparação idealizadas para a ferramenta (WANGENHEIM, 2000).

O processo GQM orientou a elaboração de um plano de medição, baseado em alguns objetivos definidos, que permitiram levantar algumas perguntas de medição bem específicas, com o intuito de apontar medidas de código que ajudem a responder as questões elaboradas e atingir os objetivos definidos. Os objetivos de medição foram definidos de acordo com um dos objetivos específicos propostos para a solução implementada, focando na necessidade de extrair informações de projetos orientados a objetos. Primeiramente foi definido um objetivo geral de medição, para orientar a elaboração do plano de medição: *Identificar se o modelo de extração construído apresenta características que suportem a visualização de projetos orientados a objetos.* A partir desse objetivo inicial, foram derivados outros três objetivos, que apontam para necessidades mais específicas de visualização dentro da orientação a objetos:

a) Identificar se o modelo construído permite encontrar informações sobre classe que contribuam com a visualização de classe por parte de um desenvolvedor que necessite analisar um código implementado;

b) Identificar se o modelo construído permite encontrar informações sobre método que constribuam com a visualização dos métodos de uma classe por parte de um desenvolvedor que necessite analisar um código implementado;

c) Identificar se o modelo construído permite encontrar informações sobre classes e heranças que contribuam para a visualização de hierarquias de classes por parte de um desenvolvedor que necessite analisar um código implementado;

Com os objetivos de medição bem delineados, foram elaborados alguns questionamentos sobre classes, métodos e herança, que se respondidas juntamente ao modelo de extração desenvolvido, contribuiriam para atingir os objetivos específicos e gerais estabelecidos inicialmente. Foram enumeradas as questões da seguinte maneira:

[1] Segundo Wangenheim (2000) o GQM é uma abordagem orientada a metas para a mensuração de produtos e processos de software, suportando a definição *top-down* de um programa de mensuração e a análise e interpretação *bottom-up* dos dados de mensuração.

a) Identificar se o modelo construído permite encontrar informações sobre classe que contribuam com a visualização da classe;
   - Qual a relação entre o número de atributos privados e públicos de uma classe?
   - Qual a relação entre o número de variáveis de classe e de instância de uma classe?
   - Qual a relação entre o número de métodos de classe e de instância de uma classe?
   - Qual a relação entre o número de parâmetros por método e o número de métodos de uma classe?
   - Qual a relação entre atributos e métodos invisíveis e o encapsulamento de atributos e métodos nas classes?
   - Quantas linhas de comentários por método tem uma classe?

b) Identificar se o modelo construído permite encontrar informações sobre método que constribuam com a visualização dos métodos da classe;
   - Quantos parâmetros são recebidos na declaração do método?
   - Quantos métodos são invocados por um determinado método?
   - Quantos atributos são acessados por um determinado método?
   - Quantas linhas de código e comentário tem um determinado método?

c) Identificar se o modelo construído permite encontrar informações sobre classes e heranças que constribuam para a visualização de hierarquias de classes;
   - Quantos ancestrais tem uma classe?
   - Quantos filhos tem uma classe?
   - Qual o número de métodos herdados pela subclasse?
   - Qual o número de métodos adicionados pela subclasse?
   - Qual o número de métodos estendidos pela subclasse?
   - Qual o número de métodos sobrescritos pela subclasse?
   - Qual o nível de profundidade da classe na hierarquia?

Para responder cada uma dessas perguntas, foi realizado um estudo sobre medidas de código orientadas a objetos, sendo identificado, levantado e enumerado um conjunto de medidas de código, divididas em três grupos.

Medidas de Classe:

a) Qual a relação entre o número de atributos privados e públicos de uma classe?
   - *Number of Public Attributes* (NPA)

b) Qual a relação entre o número de variáveis de classe e de instância de uma classe?

  - *Number of Class Variables in a Class* (NCV)
  - *Number of Instance Variables in a Class* (NIV)

c) Qual a relação entre o número de métodos de classe e de instância de uma classe?

  - *Number of Class Methods in a Class* (NCM)
  - *Number of Instance Methods in a Class* (NIM)

d) Qual a relação entre o número de parâmetros por método e o número de métodos de uma classe?

  - *Number of Parameters per Method* (NPM)
  - *Number of Methods* (NOM)

e) Qual a relação entre atributos e métodos invisíveis e o encapsulamento de atributos e métodos nas classes?

  - *Attribute Hiding Factor* (AHF)
  - *Method Hiding Factor* (MHF)

f) Qual a relação entre o número de linhas de comentário por método e a quantidade de métodos comentados?

  - *Lines of Comment per Method* (CLM)
  - *Percentage of Commented Methods* (PCM)

Medidas de Método:

a) Quantos parâmetros são recebidos na declaração do método?

  - *Number of Parameters* (NOP)

b) Quantos métodos são invocados por um determinado método?

  - *Number of Invocations* (NI)

c) Quantos atributos são acessados por um determinado método?

  - *Number of Accesses on Attributes* (NMAA)

d) Quantas linhas de código e comentário tem um determinado método?

  - *Lines of Code* (LOC)
  - *Lines of Comment* (COM)

Medidas de Herança:

a) Quantos ancestrais tem uma classe?

  - *Number of Ancestors* (NOA)

b) Quantos filhos tem uma classe?

- *Number of Children* (NOC)

c) Qual o número de métodos herdados pela subclasse?

- *Number of Methods Inherited* (NMI)

d) Qual o número de métodos estendidos pela subclasse?

- *Number of Methods Extended* (NME)

e) Qual o número de métodos sobrescritos pela subclasse?

- *Number of Methods Overridden* (NMO)

f) Qual o número de métodos adicionados pela subclasse?

- *Number of Methods Added* (NMA)

g) Qual o nível de profundidade da classe na hierarquia?

- *Depth of Inheritance Tree* (DIT)

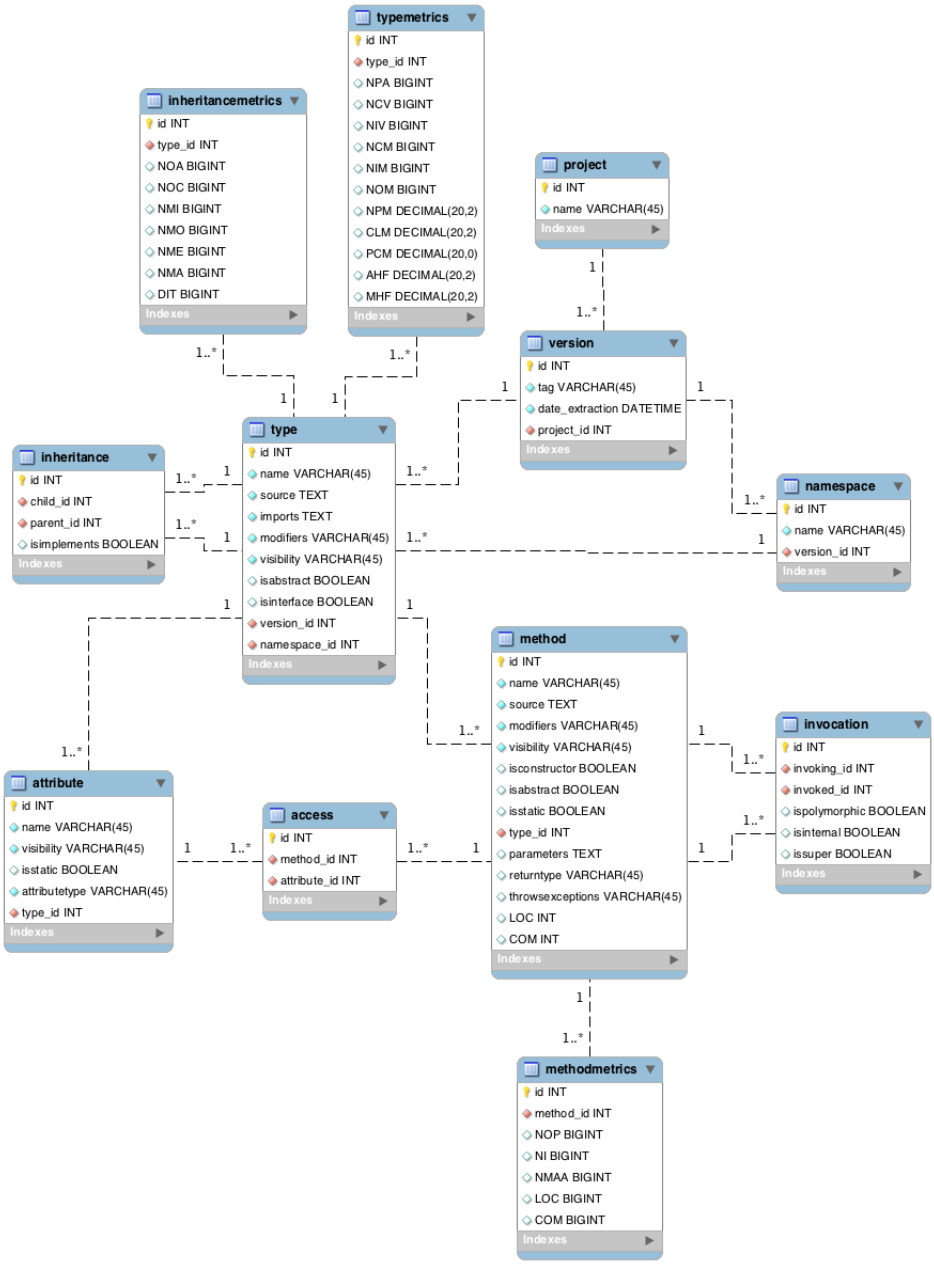
Tendo alcançado um conjunto bem definido de medidas de código, estabeleceu-se um plano de medição responsável por orientar o processo de validação da ferramenta. Coletar, registrar e analisar os resultados de cada uma das medidas propostas pelo plano de medição, permitiria responder a cada uma das questões elaboradas anteriormente, indicando se o extrator está preparado para extrair informações úteis para a visualização e compreensão de projetos de software orientados a objetos.

4.5 Processamento das Medidas

Para extrair medidas de código com base nos dados extraídos da AST, foram implementados serviços para processar e persistir no banco de dados as medidas obtidas. Após realizar a extração do código-fonte, a ferramenta passa a calcular medidas de código para cada uma das classes e métodos do projeto. Para isso foi criada uma camada de modelagem e processamento de medidas, seguindo o conjunto de medidas apontadas pelo plano de medição elaborado durante o processo GQM. Foram implementadas medidas de classe e herança vinculadas em relações de 1x1 com a respectiva entidade *Type*, e medidas de métodos também vinculadas à entidade *Method* em uma relação 1x1, como pode ser visto através da Figura 15. Outras medidas de projeto, pacote e atributos podem ser levantadas futuramente para acrescentar ao conjunto de medidas mapeadas pela ferramenta.

Cada uma das medidas foi calculada com base em trabalhos referentes encontrados. Essas medidas foram estudadas e interpretadas com base nos conceitos e definições apresentados pelos diferentes autores estudados. Os conceitos envolvidos em cada medida foram aplicados ao extrator implementado, gerando fórmulas que foram usadas para calculá-las. Ao final de todo o processo de extração e processamento de medidas por

Figura 15 – Modelo ER com as medidas mapeadas e suas relações



parte da ferramenta, é deixado ao usuário, um legado histórico de informações referentes às versão extraídas do projeto. Essa base histórica pode ser usada para fazer outras comparações ainda mais profundas e complexas com relação às análises da evolução do software ao longo do tempo.

4.5.1 Medidas de Classe

Algumas medidas de classe foram apresentadas no trabalho de Xenos et al. (2000), como o número de atributos públicos ou (NPA), o número de variáveis de classe (NCV), o

número de variáveis de instância (NIV), o número de métodos de classe (NCM), o número de métodos de instância (NIM) e o número de parâmetros por método (NPM). Segundo Xenos et al. (2000) a medida NPA pode ser calculada com um simples *count* na tabela *attribute* filtrando por atributos que tenham visibilidade *public*. Já as medidas NCV e NIV podem ser calculadas com um *count* em atributos que tenham a propriedade *static* ou não, respectivamente. As medidas NCM e NIM por sua vez são calculadas através de um *count* na tabela *method* filtrando por métodos que tenham também a propriedade *static* ou não, respectivamente. Por último a medida NPM calcula uma média entre o número de parâmetros para todos os métodos da classe, dividido pelo número total de métodos da classe.

Outros trabalhos como (LI; HENRY, 1993), (LAJIOS et al., 2008) e (HARRISON et al., 1998), trouxeram definições mais claras e precisas sobre o número de métodos (NOM), o número de linhas de comentário por método (CLM), a porcentagem de métodos comentados (PCM), o fator de invisibilidade dos atributos (AHF) e o fator de invisibilidade dos métodos (MHF). O NOM apontado por Li e Henry (1993) pode ser obtido com um simples *count* dos métodos para uma determinada classe. O CLM de Lajios et al. (2008) calcula uma média entre o número de linhas de comentário em todos os métodos da classe, dividido pelo número de métodos da classe. O PCM por sua vez é uma porcentagem obtida sobre o número de métodos que possuem algum comentário, e o número total de métodos da classe. Já as medidas apontadas por Harrison et al. (1998), possuem uma fórmula de cálculo bem mais complexa. Para o MHF é dado a seguinte expressão:

$$\frac{\sum_{i=1}^{TC}\sum_{m=1}^{M_d(C_1)}\left(1-V\left(M_m i\right)\right)}{\sum_{i=1}^{TC}M_d\left(C_i\right)}$$

Sendo que $M_d\left(C_i\right)$ é o número de métodos declarados na classe e

$$V\left(M_m i\right)=\frac{\sum_{j=1}^{TC} isVisible\left(M_m i, C_1\right)}{TC-1}$$

Sendo TC o número total de classes do projeto e

$$isVisible\left(M_m i, C_1\right)=\begin{cases}1 & if\left(j\neq i\wedge C_j\right) canInvoke M_m i\\ 0 & otherwise\end{cases}$$

Dessa maneira chegou-se à conclusão que MHF pode ser calculado processando

$$mhf=\frac{1-\frac{allMethodInvisibility}{allProjectTypes-1}}{typeMethods}$$

A mesma fórmula pode ser adaptada para encontrar a medida AHF, substituindo o cálculo dos métodos para os atributos. Para os dados chamados por *allMethodInvisibility*,

foi implementada uma consulta mais elaborada ao banco de dados, com o objetivo de calcular para cada método da classe apontada, quantas classes do projeto extraído, não podem invocar o método quando possuir uma restrição de baixa visibilidade. O mesmo foi implementado para atributos, considerando para isso visibilidades como *private*, *protected* e *package* que no Java é representado pela ausência do modificador de visibilidade. Cada nível de 'invisibilidade' restringe a um determinado grupo de classes, que é contabilizado pela consulta, assim como na Figura 16 apresentada.

Figura 16 – Implementação da Consulta elaborada para a Medida MHF

```
StringBuilder hql = new StringBuilder(1024);
hql.append("SELECT SUM(blinded_types) ");
hql.append("FROM ( ");
hql.append("    SELECT COUNT(type.ID) AS blinded_types ");
hql.append("    FROM METHOD method, TYPE type ");
hql.append("        WHERE method.VISIBILITY = 'private' ");
hql.append("        AND method.TYPE_ID != type.ID ");
hql.append("        AND method.TYPE_ID = ? ");
hql.append("        AND type.VERSION_ID = ? ");
hql.append("    UNION ");
hql.append("    SELECT COUNT(type.ID) AS blinded_types ");
hql.append("    FROM METHOD method, TYPE type ");
hql.append("        WHERE method.VISIBILITY = 'protected' ");
hql.append("        AND method.TYPE_ID = ? ");
hql.append("        AND method.TYPE_ID != type.ID ");
hql.append("        AND type.VERSION_ID = ? ");
hql.append("        AND method.TYPE_ID NOT IN ( ");
hql.append("            SELECT inheritance.PARENT_ID ");
hql.append("            FROM INHERITANCE inheritance, Type t ");
hql.append("                WHERE inheritance.CHILD_ID = t.ID ");
hql.append("                AND t.ID = type.ID ");
hql.append("                AND t.VERSION_ID = ? ");
hql.append("        ) ");
hql.append("    UNION ");
hql.append("    SELECT COUNT(type.ID) AS blinded_types ");
hql.append("    FROM METHOD method, TYPE type, TYPE t ");
hql.append("        WHERE method.VISIBILITY = 'package' ");
hql.append("        AND method.TYPE_ID = ? ");
hql.append("        AND method.TYPE_ID = t.ID ");
hql.append("        AND t.NAMESPACE_ID != type.NAMESPACE_ID ");
hql.append("        AND type.VERSION_ID = ? ");
hql.append("        AND t.VERSION_ID = ? ");
hql.append(") AS hiddingMethodTypes ");
```

A consulta busca todos as classes que não podem enchergar algum método da classe

4.5.2 Medidas de Métodos

Algumas medidas de método foram apresentadas por (DEMEYER et al., 1999), e separadas para serem implementados neste trabalho, como é o caso do número de parâmetros (NOP), o número de invocações (NI), o número de acessos aos atributos (NMAA). A medida NOP pode ser obtida tomando diretamente a referência da lista

de parâmetros declarados para o método e verificando o tamanho da lista. A medida NI já precisa de um *count* na tabela *invocation* buscando todas as invocações para um determinado método. O mesmo acontece com a medida NMAA, um simples *count* na tabela *access* filtrando pelo método retorna o número de acessos realizados pelo método.

Outros autores apresentaram algumas medidas de métodos, como é o caso de (GYIMOTHY et al., 2005) e (CHAWLA; CHHABRA, 2012), que apontaram respectivamente definições para o número de linha de código (LOC) e o número de linhas de comentário (COM). As medidas LOC e COM são obtidas previamente, e são calculadas durante a extração do código. A medida LOC é obtida extraindo o código-fonte do método, separando o conteúdo através de cada quebra de linha, e identificando quantas quebras de linhas são encontradas dentro do código. Da mesma maneira o conteúdo dos comentários é separado pelas quebras de linhas existentes dentro do comentário, facilitando para a identificação do número de linhas.

Existem alguns limitadores para as medidas de invocação, acesso e comentários de código. O *parser* foi implementado de maneira que só contabiliza invocações e acessos a atributos e métodos de classes que pertençam ao projeto extraído. Métodos ou atributos de classes do próprio Java, ou de bibliotecas importadas no projeto não serão levados em conta. Quanto aos comentários, o extrator considera inicialmente, apenas comentários de javadoc, ignorando por enquanto comentários *inline* ou comentários simples, o que pode afetar a exatidão dos resultados.

4.5.3 Medidas de Herança

Por último foram implementadas algumas medidas de herança entre as classes, apresentadas paralelamente por autores como (AGGARWAL et al., 2009) e (DEMEYER et al., 1999), que trouxeram definições bem claras sobre o número de ancestrais (NOA), o número de filhos (NOC), o número de métodos herdados (NMI), o número de métodos sobrescritos (NMO), o número de métodos estendidos (NME), o número de métodos adicionados (NMA) e a profundidade na árvore hierárquica (DIT). Para calcular essas medidas foi necessário implementar comportamentos recursivos que percorressem todos os níveis hierárquicos e trouxessem informações específicas de cada um deles.

Para a medida NOA, foi implementada uma rotina que navega pela árvore hierárquica de herança em herança, até encontrar todos os ancestrais da classe requisitada. Já a medida NOC foi calculada para identificar apenas as classes filhas que estiverem diretamente associadas à uma superclasse. Para a medida NMI é preciso contar todos os métodos herdados por uma subclasse, que não tenham sido sobrescritos por nenhum método declarado. Já para a medida NMA são contados todos os métodos que foram declarados pela subclasse, e não sobrescrevem nenhum método das superclasses. No caso da NME são calculados todos os métodos da superclasse sobrescritos pela subclasse, mas

invocados através do comando *super*. A medida NMO por sua vez é calculada contando todos os métodos da superclasse que foram sobrescritos pela subclasse, e rejeitados não sendo reaproveitados por ela. Por último, a medida DIT calcula o nível da profundidade de uma subclasse dentro da árvore hierárquica, começando por zero caso não tenha nenhuma superclasse, e incrementando até encontrar o último ancestral.

Para implementar o processamento das medidas de herança entre classes, foi necessário adotar estratégias recursivas que permitam à ferramenta navegar através das árvores hierárquicas de classes representadas pelos registros persistidos no banco de dados. Entretanto escolheu-se evitar as estruturas recursivas disponibilizadas pelos bancos de dados relacionais, adotando rotinas recursivas em tempo de programação, visto que não se tem uma padronização para as consultas recursivas implementadas pelos bancos de dados atuais. O motivo se deve à necessidade de manter uma arquitetura independente de banco de dados, associada à especificação JPA e compatível com as principais soluções de banco de dados existentes.

4.6 Fechamento

Durante este capítulo foi apresentada uma solução para a extração de fatos de projetos de software orientados a objetos. Foi apresentada também, toda a arquitetura de extração implementada, representando as decisões arquiteturais e as ferramentas utilizadas através de modelos conceituais e arquiteturais elaborados. Foram realizados vários estudos para orientar as decisões de projeto, que apontaram para a elaboração de um modelo de extração convergente, que atenda a maioria das necessidades de extração apresentadas pelos modelos estudados, mapeando um conjunto de entidades de código obtidos da AST. Foram utilizadas ferramentas de apoio para a implementação do extrator, como o *JDT Core*, e a especificação JPA, com o intuito de reutilizar e acelerar o processo de implementação das estruturas de compilação, navegação e processamento da AST do código, bem como o processo de persistência dos dados em banco de dados. Foi também elaborado um plano de medição de software GQM, com o objetivo de apontar a melhor forma de medir os resultados obtidos pela ferramenta. Foi levantado um conjunto de indicadores e medidas para ajudarem a a obter as informações de medição necessárias, e foram implementados serviços de processamento para essas medidas. Algumas consultas foram elaboradas para calcular e obter os indicadores desejados, utilizando os recursos da linguagem SQL para processar essas informações de forma otimizada.

5 Validação do Modelo PF Factfinder

Neste capítulo é apresentado o processo adotado para validar a ferramenta implementada. Na seção 5.1 são descritas as decisões, as ações e as tecnologias adotadas para a validação da ferramenta, bem como os testes realizados para verificar o seu funcionamento. Na seção 5.2 são apresentados os resultados das extrações realizadas através da ferramenta, estruturando os dados coletados em tabelas que permitam estudá-los e compará-los. Na seção 5.3 é feita uma análise dos resultados obtidos durante a validação da ferramenta, apontando fatores positivos e negativos identificados durante o estudo. Na seção 5.4 é realizado um fechamento com uma revisão dos procedimentos adotados e uma comparação dos resultados com os objetivos de medição propostos anteriormente.

5.1 Processo de Validação

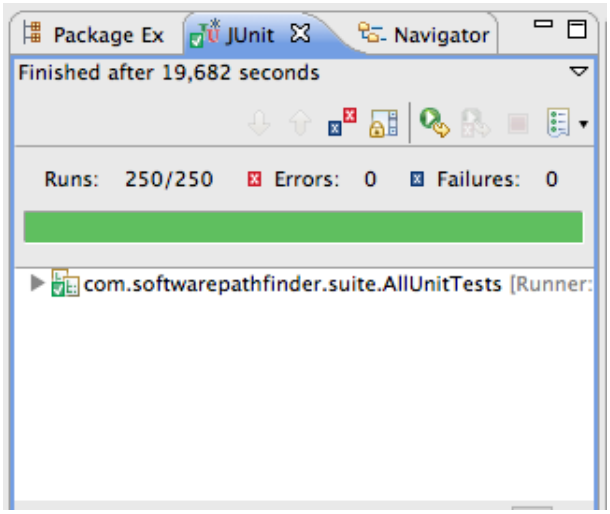
Tendo em vista a necessidade de medir os resultados obtidos pelo extrator implementado, escolheu-se o processo GQM como uma estratégia para auxiliar na identificação e no levantamento de medidas que atendam às necessidades de medição da ferramenta. Com um conjunto bem definido de medidas em mãos, deu-se início a um processo de preparação e estudo, com o intuito de permitir que a coleta e a análise das medidas seja feita corretamente.

Para uma validação inicial da ferramenta foi codificado um projeto modelo simples, para a verificação de alguns casos mais básicos da extração de código. O projeto modelo foi elaborado de maneira a apresentar elementos relevantes da Orientação a Objetos como herança, polimorfismo, invocação de métodos e instanciação de objetos. Ele foi utilizado como prova de conceito, permitindo verificar o funcionamento da ferramenta nas situações mais corriqueiras do processo de extração.

Para coletar cada uma das medidas levantadas durante o processo GQM, foi implementada uma suíte de testes de integração, com o intuito de automatizar o processo de extração e conferência dos dados coletados. A suíte de testes foi projetada para realizar um processo completo de extração sobre o projeto modelo definido, persistindo todos os dados e processando as medidas de código implementadas. Para testar cada um dos serviços de processamento de medidas, foram criados casos de teste específicos, que verificam os resultados da extração, consultando os dados persistidos e comparando-os com dados conhecidos no código que foram calculados e processados empiricamente.

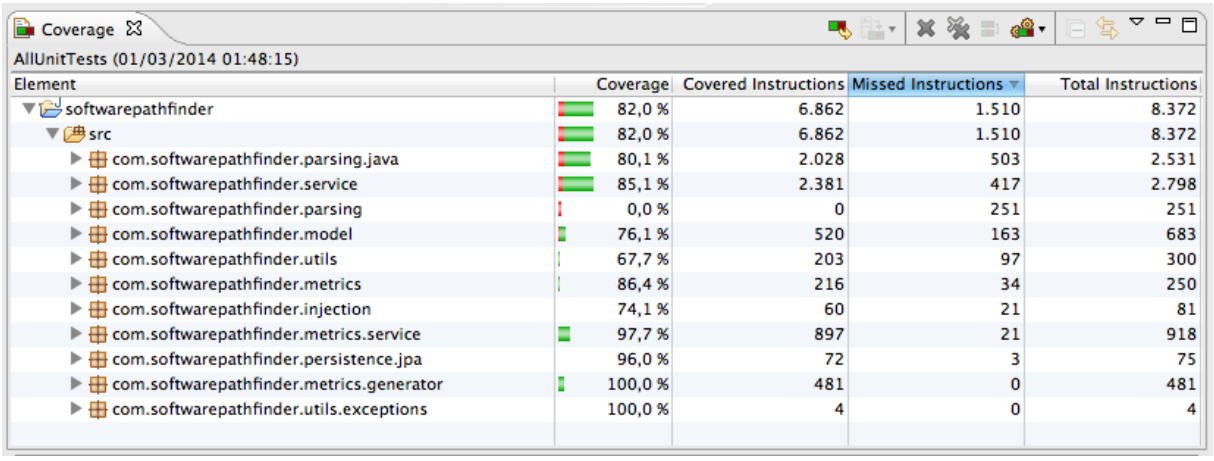
Foram construídos testes unitários para dar à ferramenta implementada maior cobertura de código. Os testes unitários foram úteis na hora de evoluir a arquitetura da

Figura 17 – Testes Unitários Implementados usando JUnit4



ferramenta, e para dar segurança ao realizar algumas refatorações e aplicar melhorias. Foram implementados 250 testes unitários, com uma cobertura de 82% do código implementado, assim como mostrado na Figura 17 e Figura 18. Toda essa estrutura de testes permitiu, em primeira instância, garantir o funcionamento adequado da ferramenta para as operações básicas de extração implementadas sobre projetos de software orientados a objetos.

Figura 18 – Cobertura dos testes unitários sobre o extrator



Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
softwarepathfinder	82,0 %	6.862	1.510	8.372
src	82,0 %	6.862	1.510	8.372
com.softwarepathfinder.parsing.java	80,1 %	2.028	503	2.531
com.softwarepathfinder.service	85,1 %	2.381	417	2.798
com.softwarepathfinder.parsing	0,0 %	0	251	251
com.softwarepathfinder.model	76,1 %	520	163	683
com.softwarepathfinder.utils	67,7 %	203	97	300
com.softwarepathfinder.metrics	86,4 %	216	34	250
com.softwarepathfinder.injection	74,1 %	60	21	81
com.softwarepathfinder.metrics.service	97,7 %	897	21	918
com.softwarepathfinder.persistence.jpa	96,0 %	72	3	75
com.softwarepathfinder.metrics.generator	100,0 %	481	0	481
com.softwarepathfinder.utils.exceptions	100,0 %	4	0	4

Em uma segunda etapa da coleta dos dados, foram estudados alguns projetos *Open Source* Java, que continham um número considerável de classes, métodos e linhas de código para extração. Experimentar a ferramenta em projetos mais complexos e grandes, permitiu consolidar a idéia, e verificar se a ferramenta suporta o uso das consultas SQL para análises em grande escala. Dentre os projetos de software estudados, foi selecionado

um projeto de médio porte chamado Apache Maven, para ser analisado e extraído, verificando com isso os resultados da ferramenta em um caso real de código. O Maven foi escolhido por ser um projeto Java amplamente conhecido, e por apresentar um conteúdo bastante desafiador e diversificado de dados para os algoritmos de extração implementados. São cerca de 880 classes e mais de 6500 métodos, com mais de 100000 linhas de código.

Para comparar os resultados obtidos pela ferramenta, decidiu-se levantar um conjunto inicial de dados sobre o projeto, que serviria como ponto de partida e parâmetro de comparação para os dados gerados durante a extração. Uma amostra inicial foi colhida para cada uma das medidas implementadas pela ferramenta, utilizando como objeto de estudo, classes e métodos encontrados no projeto Maven. Foram identificados e analisados trechos de código úteis para elaborar uma amostragem inicial, priorizando conjuntos com grandes quantidades de dados. As medidas foram colhidas e calculadas manualmente a partir do código-fonte analisado, registrando os resultados calculados para porteriormente compará-los com as saídas geradas pela ferramenta.

Após elaborar uma base de comparação clara, foram colhidas outras cinco amostras do projeto usando a ferramenta de extração para processar o código. As medidas foram igualmente processadas pela ferramenta e persistidas na base de dados sendo disponibilizadas para consulta. Os resultados obtidos pelas consultas de cada versão das amostras extraídas, foram tabulados e comparados entre si, com o intuito de verificar a exatidão[1] e a precisão[2] da ferramenta.

Por último, também foram aplicadas manualmente no banco de dados, algumas consultas utilizadas para calcular medidas, com o objetivo de analisar o tempo de processamento necessário. Foram selecionadas algumas medidas complexas e simples, dentre o conjunto de medidas abordadas pela ferramenta de extração. As consultas foram feitas sobre a massa de dados previamente gerada durante as extrações realizadas sobre o projeto Maven, coletando a medida de tempo necessária para calcular medidas de código no modelo de extração implementado. Esses valores foram também registrados para servirem como parâmetro para futuras comparações entre os resultados da ferramenta implementada com abordagens adotadas em outros trabalhos.

[1] Exatidão é a proximidade de uma medida, ou estimativa, do valor de uma variável medida ou do parâmetro que está sendo estudado (SABINO; VILLAÇA, 1999). Ou seja o quão próximo de um valor alvo definido uma ou mais medidas obtidas podem estar.

[2] A precisão é o grau de concordância entre um número de medidas, ou estimativas, para uma mesma população (SABINO; VILLAÇA, 1999). Ou seja o quão próximas as medidas obtidas podem estar umas das outras.

5.2 Resultados Gerados

Para a coleta dos dados, foram utilizadas as medidas levantadas durante a elaboração do plano de medição proposto na seção 4.5, sendo escolhidas algumas classes e métodos do projeto Maven para servirem como objeto de análise. Para coletar amostras para as medidas de classe, foram apontadas as classes *MavenProject* e *DefaultMavenExecutionRequest* que apresentaram visivelmente um conjunto maior e mais diversificado de dados para obter cada uma das medidas. As amostras de cada extração do código foram coletadas, organizando e estruturando as informações através da Tabela 3 e da Tabela 4, comparando através disso os resultados obtidos com as amostras iniciais obtidas.

Tabela 3 – Amostras de medidas coletadas para a classe MavenProject

Medidas	Amostra Inicial	Primeira Extração	Segunda Extração	Terceira Extração	Quarta Extração	Quinta Extração
NPA	3	3	3	3	3	3
NOM	180	180	180	180	180	180
NPM	0,59	0,59	0,59	0,59	0,59	0,59
NCV	4	4	4	4	4	4
NIV	44	44	44	44	44	44
NCM	2	2	2	2	2	2
NIM	178	178	178	178	178	178
AHF	-0,92	-0,92	-0,92	-0,92	-0,92	-0,92
MHF	-0,08	-0,08	-0,08	-0,08	-0,08	-0,08
CLM	1,03	0,75	0,75	0,75	0,75	0,75
PCM	22,77%	14%	14%	14%	14%	14%

Tabela 4 – Amostras de medidas coletadas para a classe DefaultMavenExecutionRequest

Medidas	Amostra Inicial	Primeira Extração	Segunda Extração	Terceira Extração	Quarta Extração	Quinta Extração
NPA	0	0	0	0	0	0
NOM	106	106	106	106	106	106
NPM	0,56	0,56	0,56	0,56	0,56	0,56
NCV	0	0	0	0	0	0
NIV	45	45	45	45	45	45
NCM	1	1	1	1	1	1
NIM	105	105	105	105	105	105
AHF	-0,98	-0,98	-0,98	-0,98	-0,98	-0,98
MHF	-0,01	-0,01	-0,01	-0,01	-0,01	-0,01
CLM	0,018	0,20	0,20	0,20	0,20	0,20
PCM	1,88%	0,94%	0,94%	0,94%	0,94%	0,94%

Os resultados inexatos para CLM e PCM atribuem-se a necessidade de evoluir o parser para suportar a extração de comentários de código simples, juntamente com a extração de comentários de javadoc. Também foram coletadas as medidas de método implementadas pela ferramenta. Foram escolhidos dois métodos distintos para realizar as operações de consultas às medidas, o método *loadPom(session, request, result)* da classe

DefaultArtifactDescriptorReader, e o método *recurse(result, node, resolvedArtifacts, managedVersions, request, source, filter, listeners, conflictResolvers)* da classe *DefaultLegacyArtifactCollector*. Os resultados coletados foram organizados e registrados através da Tabela 5 e da Tabela 6.

Tabela 5 – Amostras de medidas coletadas para o método loadPom() da classe DefaultArtifactDescriptorReader

Medidas	**Amostra Inicial**	**Primeira Extração**	**Segunda Extração**	**Terceira Extração**	**Quarta Extração**	**Quinta Extração**
NOP	3	3	3	3	3	3
NI	27	27	27	27	27	27
NMAA	6	4	4	4	4	4
LOC	114	86	86	86	86	86
COM	0	0	0	0	0	0

Tabela 6 – Amostras de medidas coletadas para o método recurse() da classe DefaultLegacyArtifactCollector

Medidas	**Amostra Inicial**	**Primeira Extração**	**Segunda Extração**	**Terceira Extração**	**Quarta Extração**	**Quinta Extração**
NOP	9	9	9	9	9	9
NI	127	96	75	96	75	96
NMAA	11	0	0	0	0	0
LOC	250	188	188	188	188	188
COM	50	0	0	0	0	0

As medidas de método coletadas apresentaram algumas limitações, que necessitam ser melhor investigadas e estudadas. A medida LOC, por exemplo, não teve um resultado compatível com as amostras iniciais coletadas. Inicialmente foi levantada a hipótese de que o processo manual e empírico realizado para calcular esssas medidas havia falhado. Isto porque os dois métodos analisados são muito extensos e complexos para realizar essas contagem manual dos elementos do código. Posteriormente foi verificado que o próprio componente de compilação reajusta o corpo do método, indentando e alterando algumas configurações da disposição das linhas de código empregadas para o método. Outros fatores precisam ser investigados para atribuir maior exatidão aos resultados da ferramenta sobre as medidas coletadas.

Por último, foram também coletadas amostras das medidas de herança sobre classes do projeto. Para isso foram escolhidas hierarquias que permitam comparar um conjunto grande de dados para a obtenção das medidas. Foram escolhidas as heranças entre as classes *AttachedArtifact* e *DefaultArtifact* e as classes *MavenModelMerger* e *ModelMerger*. As classes que serviram de base para o cálculo das medidas foram as subclasses, sendo necessário apenas obter a medida NOC da superclasse *DefaultArtifact*, visto que a classe filha não possuía nenhuma subclasse. Os dados obtidos para cada extração fo-

ram coletados e estruturados na Tabela 7 e na Tabela 8 para serem comparados com os resultados das amostras iniciais.

Tabela 7 – Amostras de medidas de herança para a classe AttachedArtifact subclasse de DefaultArtifact

Medidas	Amostra Inicial	Primeira Extração	Segunda Extração	Terceira Extração	Quarta Extração	Quinta Extração
NOA	1	1	1	1	1	1
NOC	3 (DefaultArtifact)	3	3	3	3	3
NMI	36	38	38	38	38	38
NME	1	0	0	0	0	0
NMO	21	20	20	20	20	20
NMA	1	3	3	3	3	3
DIT	1	1	1	1	1	1

As amostras coletadas para a herança entre a classe *AttachedArtifact* e *DefaultArtifact* também mostraram algumas limitações da ferramenta, desta vez quanto à identificação sobreescrita e extensão de métodos construtores para a subclasse e superclasse analisada.

Tabela 8 – Amostras de medidas de herança para a classe MavenModelMerger subclasse de ModelMerger

Medidas	Amostra Inicial	Primeira Extração	Segunda Extração	Terceira Extração	Quarta Extração	Quinta Extração
NOA	1	1	1	1	1	1
NOC	6	6	6	6	6	6
NMI	167	167	167	167	167	167
NME	3	3	3	3	3	3
NMO	35	35	35	35	35	35
NMA	3	3	3	3	3	3
DIT	1	1	1	1	1	1

Foram registrados também os tempos de processamento para as consultas utilizadas para calcular algumas das medidas catalogadas acima. Os tempos foram medidos usando a unidade de segundos, e calculados em um banco de dados MySQL com índices apenas para as chaves primárias e chaves estrangeiras. Os tempos obtidos para as medidas selecionadas estão disponíveis na Tabela 9 juntamente com o valor calculado para o tempo médio de processamento de cada uma delas.

5.3 Análise dos Resultados

Com base nos resultados obtidos pelas amostras coletadas foi possível concluir que a ferramenta possui um certo grau de precisão, visto que apresentou resultados idênticos para quase todas as amostras das extrações aplicadas sobre o projeto, com exceção das amostras da medida NI coletadas para o método *recurse()* da classe *DefaultLegacyArtifactCollector*. Quanto à exatidão dos resultados, entretanto, a ferramenta apresentou certas

Tabela 9 – Tempo de Processamento das Medidas

Medidas	Primeira Consulta	Segunda Consulta	Terceira Consulta	Quarta Consulta	Quinta Consulta	Média
NPA	0,051230	0,000885	0,000493	0,000700	0,000489	0,010759
NOM	0,027668	0,000364	0,000378	0,000362	0,000363	0,005827
NIV	0,032168	0,000476	0,000466	0,000564	0,000474	0,006829
NCM	0,052803	0,001218	0,001095	0,000969	0,000946	0,011406
AHF	0,212100	0,201600	0,214000	0,206300	0,202700	0,207340
MHF	0,235700	0,464900	0,152200	0,151500	0,166000	0,234060
NI	0,020232	0,000280	0,000284	0,000286	0,000287	0,004273
NMAA	0,362431	0,000361	0,000254	0,000238	0,000245	0,072705

Tempo de processamento de algumas medidas calculadas em uma base de dados MySQL. Unidade de medida: segundos(s)

oscilações, visto que algumas das medidas foram incompatíveis com os resultados esperados, distanciando-se consideravelmente dos valores empíricos coletados manualmente para as amostras iniciais no início do processo de validação.

Algumas das medidas apresentaram exatamente os resultados esperados, ou resultados bastante próximos dos resultados coletados na amostra inicial, como é o caso das medidas NOM, NPM, AHF e MHF encontradas na Tabela 3 e na Tabela 4. As medidas incompatíveis, entretanto, não obtiveram resultados muito claros, como é o caso das medidas LOC, NI e NMAA encotradas na Tabela 5 e na Tabela 6, que apresentaram valores bem diferentes dos valores esperados em comparação com as amostras iniciais coletadas manualmente. A grande diferença dos resultados da medida LOC, por exemplo deve-se a uma característica do JDT de reformatar o código processado, omitindo os comentários e reposicionando linhas de código não indentadas, como no caso de declaração ou invocação de métodos com muitos parâmetros, concatenação de variáveis que quebrem uma linha de código em várias.

Outras medidas apresentaram dados diferentes do esperado, devido a algumas decisões de implementação que acabaram refletindo nos resultados coletados. É o caso das medidas COM, CLM e PCM encontradas na Tabela 3, na Tabela 4 e na Tabela 6, que até o momento contabilizam apenas comentários de Javadoc, ignorando comentários simples de bloco e em linha. Algumas medidas de herança como NMO, NMI, NME e NMA encontradas na Tabela 7 e na Tabela 8, também apresentaram falhas pontuais, visto que o extrator não considera a sobrescrita dos métodos construtores da superclasse. Em contrapartida o extrator apresentou bom resultado identificando as 6 subclasses de *MavenModelMerger* Tabela 8, mesmo sendo várias delas classes privadas que não possuem arquivo de código próprio, o que tornou a coleta manual um grande desafio durante a manipulação do código-fonte.

Das 230 amostras coletadas, 225 apresentaram resultados precisos, representando

com isso 98% das amostras coletadas, enquanto apenas 5 delas mostraram uma certa imprecisão, como pode ser visto através da Figura 19 e da Figura 20. As mesmas amostras foram analisadas quanto à exatidão, permitindo identificar 160 amostras com resultados exatamente iguais, 40 amostras com resultados aproximados, e outras 30 amostras que obtiveram resultados mais distantes dos valores esperados para a coleta. Com isso, foi identificado que 87% das amostras são exatamente ou aproximadamente iguais aos valores esperados, restando outros 13% de amostras que obtiveram resultados muito diferentes dos valores esperados, como mostra a Figura 21 e a Figura 22.

Figura 19 – Amostras de precisão coletadas durante validação da ferramenta

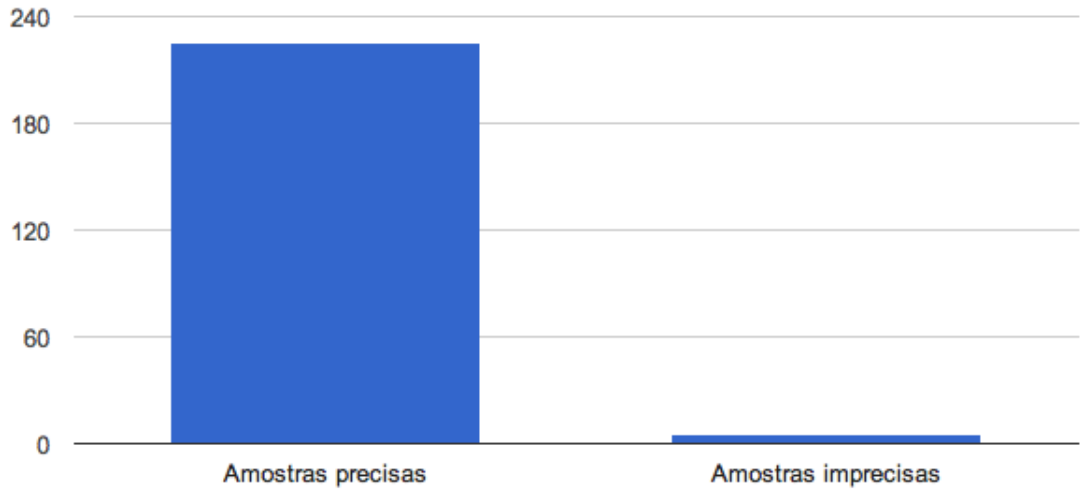
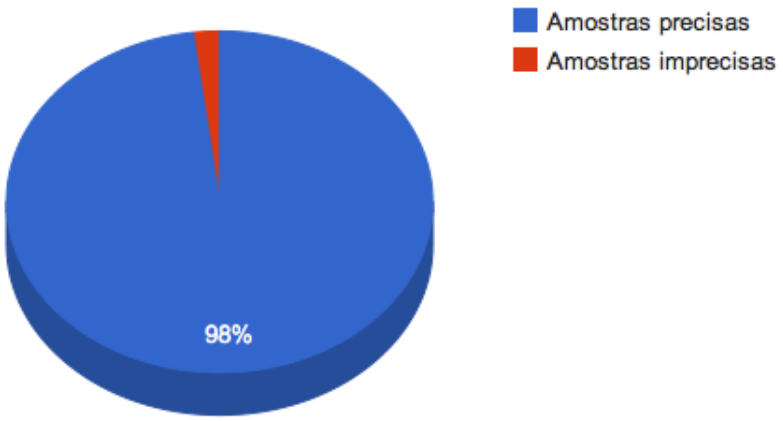


Figura 20 – Percentual de precisão para as amostras coletadas



Com isso, entende-se que, de um modo geral, o extrator consegue responder à maioria das questões elaboradas no plano de medição apresentado no Capítulo 4, permitindo obter dados precisos e exatos para as necessidades de compreensão apontadas pelo

Figura 21 – Amostras de exatidão coletadas durante validação da ferramenta

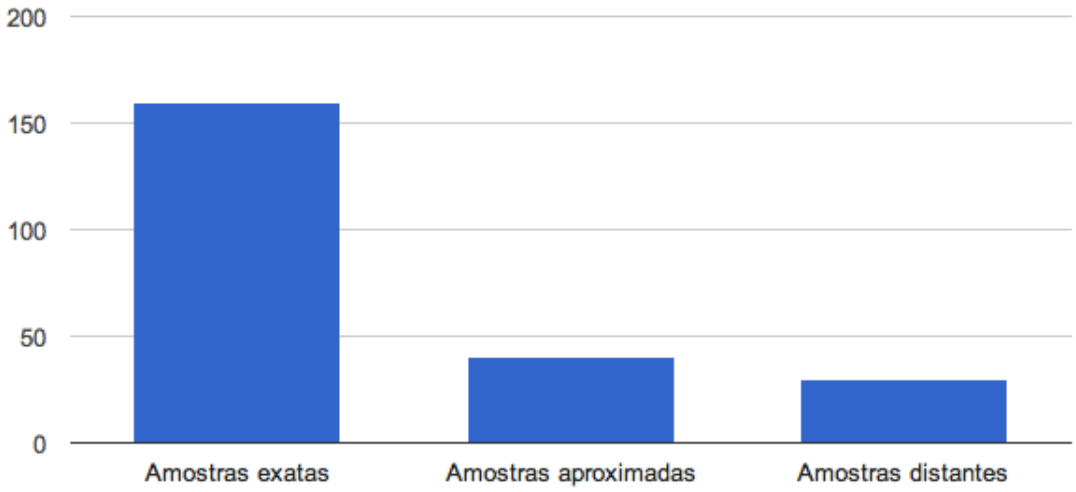
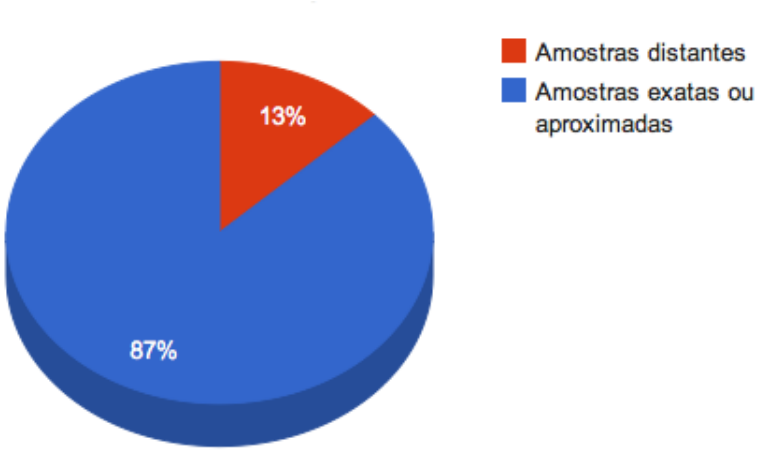


Figura 22 – Percentual de exatidao para as amostras coletadas



plano de medição. Os resultados obtidos através das medidas calculadas durante o processo de medição, permitiram verificar que a ferramenta está preparada para encontrar diversas informações úteis relacionadas à visualização de classes, métodos e heranças. Consequentemente, conclui-se que a ferramenta apresenta as características necessárias para suportar a compreensão de projetos de software orientados a objetos. Dado que os conceitos envolvidos na Orientação a Objetos não dependem de linguagem ou sintaxe específica, conclui-se que o modelo de extração proposto permite representar dados de diferentes linguagens de programação orientadas a objetos. Houve alguns pontos de atenção, por parte de medidas que não obtiveram uma exatidão apropriada, se comparando com os resultados coletados manualmente. Entretanto, esses resultados negativos não chegam a ofuscar os resultados positivos obtidos pela ferramenta, identificados na grande

quantidade de amostras exatas e precisas coletadas.

Quanto ao tempo médio calculado para o processamento das medidas, constatou-se que o desempenho foi satisfatório mesmo para as consultas mais complexas como é o caso das medidas AHF e MHF, visto que para as coletas realizadas, o tempo de processamento foi sempre menor do que 1s, tendo um valor mínimo perto de 0,0002s. O tempo médio poderia ser ainda menor se fosse calculado levando em conta um número maior de amostras, visto que o banco de dados utiliza serviços de cache para evitar acesso desnecessário ao disco. Dessa maneira permite carregar uma única vez determinados recursos que serão utilizados repetidamente por consultas semelhantes ou idênticas, como é o caso de várias das consultas descritas na Tabela 9. Nos exemplos listados, o tempo demonstra ser um pouco maior na primeira consulta, mas reduz consideravelmente nas consultas seguintes, desde que o conjunto de dados consultados e o número de operações executadas no banco sejam semelhantes. Existe a possibilidade de que os tempos obtidos durante o experimento sofram alguma alteração, se a base de dados contiver uma quantidade muito grande de informações de software, lidando com muitos projetos extraídos, em suas diferentes versões de código. Acredita-se entretanto, que o desempenho não sofra grandes impactos, visto que a arquitetura de banco de dados relacional foi criada justamente para lidar com quantidades massivas de dados.

5.4 Fechamento

Durante este capítulo foi descrito um processo de validação realizado sobre a ferramenta implementada. Foram construídos diversos testes unitários e de integração com o objetivos de garantir o funcionamento das principais funcionalidades do extrator. As medidas levantadas durante o processo GQM foram adotadas com o objetivo de medir os resultados da ferramenta, com base em um projeto *Open Source* chamado Apache Maven. Foram coletadas algumas amostras das informações extraídas do projeto Maven através da ferramenta implementada. Essas amostras foram comparadas entre si, e também foram comparadas com uma amostra inicial obtida através de análises e cálculos realizados de maneira empírica e manual sobre as classes do projeto. De maneira geral a ferramenta apresentou um resultado satisfatório, atingindo os objetivos de medição definidos no plano de medição proposto anteriormente. Por último foram registrados também os tempos obtidos durante o processamento de algumas das medidas calculadas. A ferramenta apresentou um resultado bastante otimizado do cálculo das medidas utilizando SQL, principalmente em se tratando de um projeto, com muitas classes e uma vasta quantidade de código.

6 Considerações Finais

No início deste trabalho, foram apresentados alguns problemas relacionados com o desempenho das ferramentas de compreensão e visualização de software atuais, principalmente em se tratando de projetos grandes, com muitas linhas de código. Foi destacado que o custo em memória do processamento necessário para realizar algumas análises de software mais complexas, eram um fator determinante para limitar os resultados das ferramentas existentes, impedindo-as de garantir maior escalabilidade para as análises de CS realizadas. Com base nesses problemas, foi criado um modelo de extração de fatos, adotando as propriedades e recursos apresentados pelos bancos de dados relacionais existentes, bem como a linguagem de consulta SQL, como uma alternativa de desempenho para as análises de software realizadas sobre o código-fonte.

O objetivo do trabalho era implementar uma arquitetura de extração de fatos que obtenha informações relevantes sobre o código-fonte de projetos de software orientados a objetos, armazenando as informações em uma base de dados relacional que permita todo tipo de consultas usando SQL. Para atender a esse objetivo, fez-se a implementação de uma ferramenta de extração que mapeia as entidades da AST do código, disponibilizando as informações extraídas em uma base histórica com suporte à CS e evolução de software. Fez-se elaboração de um modelo de extração baseado nos principais modelos de extração encontrados, com o objetivo de atender as principais necessidades de visualização e compreensão apresentadas pelos modelos estudados. A solução implementada permitiu, com o uso da linguagem SQL, elaborar uma série de consultas sobre os dados extraídos, com o objetivo de aplicar análises mais complexas sobre as informações do código-fonte. Para isso foi levantado um conjunto de indicadores e medidas de código, através da elaboração de um plano de medição, que permitiu validar a coerência do modelo de extração quanto a requisitos relacionados aos conceitos da Orientação a Objetos, e verificar o desempenho das análises realizadas sobre os dados armazenados, usando consultas SQL elaboradas.

Os resultados mostraram que a arquitetura de extração implementada, está fundamentada em um modelo de extração coerente com os principais conceitos da Orientação a Objetos. As medidas utilizadas foram obtidas a partir de objetivos especialmente voltados para a medição de características de linguagens orientadas a objetos. Nesse contexto, as medidas obtidas através do processamento das informações de código persistidas, apresentaram uma precisão para 98% das amostras coletadas ao longo de várias extrações realizadas sobre um mesmo projeto. Essas medidas também apresentaram uma exatidão considerável, comparando com as amostras colhidas manualmente durante o processo de medição. 87% das amostras obtiveram uma medida exata ou aproximada dos valores esperados, enquanto apenas 13% apresentaram resultados distantes das amostras iniciais

coletadas.

Os resultados apontam que o modelo de extração implementado atende as necessidades de compreensão de diferentes projetos de software orientados a objetos. Ele permite mapear, armazenar e representar informações relacionadas com o código-fonte, para processar diferentes cálculos e análises de software relacionadas com a qualidade do código, evolução do software, garantia da qualidade, que podem ser obtidas através de consultas dinâmicas em SQL, elaboradas para este fim. Visto que o modelo atende as principais características da Orientação a Objetos em projetos de software, e que a os conceitos relacionados com a Orientação a Objetos independem de linguagem ou sintaxe, entende-se que o modelo também está preparado para suportar a representação de outras linguagens de programação orientadas a objetos, utilizando um mapeamento genérico e *Parsers* capazes de ler e interpretar cada sintaxe com as suas particularidades.

Os resultados colhidos também foram medidos em relação ao tempo necessário para processar as consultas, usando uma plataforma de banco de dados relacional o armazenamento e consulta dos dados. Os tempos foram registrado e analisados, chegando a conclusão de que a ferramenta atingiu os objetivos de desempenho estabelecidos para o trabalho. Os valores registrados foram bastante reduzidos, visto que nenhuma das consultas ultrapassou 1s de processamento, mesmo para as consultas mais complexas, e a maioria das amostras processou as consultas em menos de 0,001s.

Estes resultados mostram que a ferramenta implementada apresentou uma solução ágil e escalável para lidar com o processamento dos dados obtidos através da extração de fatos. O desempenho obtido pode abrir espaço para implementar análises de software ainda mais aprofundadas, mesmo sobre projetos de software com grandes quantidades de linhas de código. O custo envolvido, e os recursos disponíveis para a elaboração de consultas SQL, permitiria lidar com grandes massas de dados, sem se preocupar com os impactos no uso da memória.

A revisão sistemática realizada no início do trabalho foi uma peça fundamental para entender o problema abordado, e ajudar a tomar decisões durante a implementação do trabalho. A abordagem adotada para atribuir mais responsabilidade ao banco de dados durante o processamento das medidas e algumas análises realizadas durante a extração do código, permitiu diminuir consideravelmente o consumo de memória por parte do interpretador Java. Outra decisão importante para atingir esses resultados, foi simplificar o modelo elaborado para torná-lo mais aderente ao conjunto de dados que seria modelado em banco. Isso permitiu evitar granularidades complexas e desnecessárias que poderiam ter afetado o desempenho das consultas realizadas.

Durante o trabalho, foi possível aprender muito sobre as bases da construção sintática da linguagem Java, bem como identificar processos alternativos de compilação e análise de código, disponíveis para o estudo na comunidade. O trabalho foi imerso em

alguns pontos muito importantes da Engenharia de Software, que trouxeram novos conhecimentos com relação à Qualidade de Software e Evolução de Software. Para implementar algumas demandas, foi necessário desenvolver novas habilidades com relação à elaboração de consultas mais complexas no banco de dados, e a utilização de arquiteturas preocupadas com o consumo de processamento em memória.

Futuramente, espera-se evoluir o modelo de extração implementado, para que seja compatível com IDEs de desenvolvimento como o Eclipse, tornando a ferramenta mais *plugável* e reutilizável para outros projetos de software. O modelo pode sofrer ainda algumas alterações, visando atender no futuro, um maior número de necessidades de compreensão identificadas, como é o caso do cálculo de medidas para a identificação de coesão, acoplamento e complexidade de código, bem como para a identificação de padrões de projeto e cheiros de código.

Outro trabalho que dará continuidade aos resultados obtidos pela ferramenta, será a implementação de estruturas de extração que atendam às demais linguagens orientadas a objetos, disponíveis no mercado atual, como PHP, Ruby, Python, C# e C++. Também se espera utilizar a ferramenta futuramente, no desenvolvimento de propostas de VS, como uma estratégia de persistência e consulta, que proporcione maior agilidade e escalabilidade ao processamento das análises de código para a construção de visões mais dinâmicas e versáteis.

Referências

AGGARWAL, K. et al. Empirical analysis for investigating the effect of object-oriented metrics on fault proneness: a replicated case study. *Software Process: Improvement and Practice*, Wiley Online Library, v. 14, n. 1, p. 39–62, 2009. Citado na página 69.

ANSLOW, C. et al. Xml database support for program trace visualisation. In: AUSTRALIAN COMPUTER SOCIETY, INC. *Proceedings of the 2004 Australasian symposium on Information Visualisation-Volume 35*. [S.l.], 2004. p. 25–34. Citado 2 vezes nas páginas 22 e 53.

BADROS, G. J. Javaml: a markup language for java source code. *Computer Networks*, Elsevier, v. 33, n. 1, p. 159–177, 2000. Citado 2 vezes nas páginas 50 e 51.

BÄR, H.; DUCASSE, S. *The FAMOOS Object Oriented Reengineering Handbook*. [S.l.]: Forschungszentrum Informatik an der Univ., 1999. Citado na página 49.

BAXTER, I. D. et al. Clone detection using abstract syntax trees. In: IEEE. *Software Maintenance, 1998. Proceedings. International Conference on*. [S.l.], 1998. p. 368–377. Citado na página 31.

CANFORA, G.; PENTA, M. D.; CERULO, L. Achievements and challenges in software reverse engineering. *Communications of the ACM*, ACM, v. 54, n. 4, p. 142–151, 2011. Citado 3 vezes nas páginas 25, 28 e 39.

CHAWLA, M. K.; CHHABRA, I. Implementing source code metrics for software quality analysis. *International Journal of Engineering*, v. 1, n. 5, 2012. Citado na página 69.

CHIDAMBER, S. R.; KEMERER, C. F. A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, IEEE, v. 20, n. 6, p. 476–493, 1994. Citado na página 34.

CHIKOFSKY, E. J.; CROSS, J. H. et al. Reverse engineering and design recovery: A taxonomy. *Software, IEEE*, IEEE, v. 7, n. 1, p. 13–17, 1990. Citado na página 28.

COLLARD, M. L.; KAGDI, H. H.; MALETIC, J. I. An xml-based lightweight c++ fact extractor. In: IEEE. *Program Comprehension, 2003. 11th IEEE International Workshop on*. [S.l.], 2003. p. 134–143. Citado 2 vezes nas páginas 50 e 51.

CORNELISSEN, B. et al. A systematic survey of program comprehension through dynamic analysis. *Software Engineering, IEEE Transactions on*, IEEE, v. 35, n. 5, p. 684–702, 2009. Citado na página 41.

DEMEYER, S.; DUCASSE, S.; LANZA, M. A hybrid reverse engineering approach combining metrics and program visualisation. In: IEEE. *Reverse Engineering, 1999. Proceedings. Sixth Working Conference on*. [S.l.], 1999. p. 175–186. Citado 3 vezes nas páginas 34, 68 e 69.

DIEHL, S. *Software visualization: visualizing the structure, behaviour, and evolution of software*. [S.l.]: Springer, 2007. Citado 2 vezes nas páginas 26 e 37.

DUCASSE, S. et al. Mse and famix 3.0: an interexchange format and source code model family. 2011. Citado 3 vezes nas páginas 28, 49 e 55.

DUCASSE, S.; LANZA, M.; TICHELAAR, S. Moose: an extensible language-independent environment for reengineering object-oriented systems. In: CITESEER. *Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000)*. [S.l.], 2000. Citado na página 49.

D'AMBROS, M. et al. Analysing software repositories to understand software evolution. In: *Software Evolution*. [S.l.]: Springer, 2008. p. 37–67. Citado 4 vezes nas páginas 41, 42, 47 e 51.

D'AMBROS, M.; LANZA, M. Distributed and collaborative software evolution analysis with churrasco. *Science of Computer Programming*, Elsevier, v. 75, n. 4, p. 276–287, 2010. Citado 5 vezes nas páginas 28, 32, 48, 50 e 51.

ECLIPSE, T. E. F. *org.eclipse.jdt.astview - AST View*. 2014. Disponível em: <http://www.eclipse.org/jdt/ui/astview/index.php>. Citado na página 31.

EDELSTEIN, D. V. Report on the ieee std 1219–1993—standard for software maintenance. *ACM SIGSOFT Software Engineering Notes*, ACM, v. 18, n. 4, p. 94–95, 1993. Citado na página 28.

FERENC, R. et al. Columbus-reverse engineering tool and schema for c++. In: IEEE. *Software Maintenance, 2002. Proceedings. International Conference on*. [S.l.], 2002. p. 172–181. Citado 2 vezes nas páginas 49 e 51.

FERENC, R. et al. Towards a standard schema for c/c++. In: IEEE. *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*. [S.l.], 2001. p. 49–58. Citado na página 55.

FOWLER, M. *Refactoring: improving the design of existing code*. [S.l.]: Addison-Wesley Professional, 1999. Citado na página 36.

FOWLER, M.; HIGHSMITH, J. The agile manifesto. *Software Development*, [San Francisco, CA: Miller Freeman, Inc., 1993-, v. 9, n. 8, p. 28–35, 2001. Citado na página 27.

GHEZZI, G.; GALL, H. C. Sofas: A lightweight architecture for software analysis as a service. In: IEEE. *Software Architecture (WICSA), 2011 9th Working IEEE/IFIP Conference on*. [S.l.], 2011. p. 93–102. Citado 2 vezes nas páginas 49 e 55.

GHEZZI, G.; GALL, H. C. A framework for semi-automated software evolution analysis composition. *Automated Software Engineering*, Springer, p. 1–34, 2013. Citado 7 vezes nas páginas 28, 35, 37, 40, 49, 50 e 51.

GYIMOTHY, T.; FERENC, R.; SIKET, I. Empirical validation of object-oriented metrics on open source software for fault prediction. *Software Engineering, IEEE Transactions on*, IEEE, v. 31, n. 10, p. 897–910, 2005. Citado na página 69.

HARRISON, R.; COUNSELL, S. J.; NITHI, R. V. An evaluation of the mood set of object-oriented software metrics. *Software Engineering, IEEE Transactions on*, IEEE, v. 24, n. 6, p. 491–496, 1998. Citado na página 67.

HOLT, R. C.; WINTER, A.; SCHURR, A. Gxl: Toward a standard exchange format. In: IEEE. *Reverse Engineering, 2000. Proceedings. Seventh Working Conference on*. [S.l.], 2000. p. 162–171. Citado 3 vezes nas páginas 30, 50 e 51.

KRUCHTEN, P. B. The 4+ 1 view model of architecture. *Software, IEEE*, IEEE, v. 12, n. 6, p. 42–50, 1995. Citado 2 vezes nas páginas 21 e 36.

LAJIOS, G.; SCHMEDDING, D.; VOLMERING, F. Supporting language conversion by metric based reports. In: *CSMR*. [S.l.: s.n.], 2008. p. 314–316. Citado na página 67.

LANZA, M. *Object-Oriented Reverse Engineering Coarse-grained, Fine-grained, and Evolutionary Software Visualization*. Tese (Doutorado) — University of Bern, 2003. Citado 2 vezes nas páginas 32 e 33.

LI, W.; HENRY, S. Object-oriented metrics that predict maintainability. *Journal of systems and software*, Elsevier, v. 23, n. 2, p. 111–122, 1993. Citado na página 67.

LIEBER, T. *Understanding Asynchronous Code*. Dissertação (Mestrado) — Massachusetts Institute of Technology, 2013. Citado na página 41.

LÖWE, W.; PANAS, T. Rapid construction of software comprehension tools. *International Journal of Software Engineering and Knowledge Engineering*, World Scientific, v. 15, n. 06, p. 995–1025, 2005. Citado 5 vezes nas páginas 21, 22, 23, 25 e 27.

MÄDER, P.; CLELAND-HUANG, J. A visual language for modeling and executing traceability queries. *Software & Systems Modeling*, Springer, p. 1–17, 2012. Citado 3 vezes nas páginas 47, 50 e 51.

MATHIAS, K. S. et al. The role of software measures and metrics in studies of program comprehension. In: ACM. *Proceedings of the 37th annual Southeast regional conference (CD-ROM)*. [S.l.], 1999. p. 13. Citado 2 vezes nas páginas 28 e 34.

MCLUHAN, H. M.; PARÉ, J. *Pour comprendre les média: prolongements technologiques de l'homme*. [S.l.]: Mame, 1968. Citado na página 38.

MOHAN, K. et al. Improving change management in software development: Integrating traceability and software configuration management. *Decision Support Systems*, Elsevier, v. 45, n. 4, p. 922–936, 2008. Citado 3 vezes nas páginas 48, 50 e 51.

MÜLLER, H. A.; KLASHINSKY, K. Rigi-a system for programming-in-the-large. In: IEEE COMPUTER SOCIETY PRESS. *Proceedings of the 10th international conference on Software engineering*. [S.l.], 1988. p. 80–86. Citado 2 vezes nas páginas 49 e 51.

MURPHY, G. C.; KERSTEN, M.; FINDLATER, L. How are java software developers using the elipse ide? *Software, IEEE*, IEEE, v. 23, n. 4, p. 76–83, 2006. Citado na página 31.

OLIVEIRA, M. S. de. *Previa: Uma Abordagem para a Visualização da Evolução de Modelos de Software*. Dissertação (Mestrado) — Universidade Federal do Rio de Janeiro, 2011. Citado 7 vezes nas páginas 21, 22, 26, 27, 32, 36 e 42.

PARNIN, C.; GÖRG, C.; NNADI, O. A catalogue of lightweight visualizations to support code smell inspection. In: ACM. *Proceedings of the 4th ACM symposium on Software visualization*. [S.l.], 2008. p. 77–86. Citado 6 vezes nas páginas 21, 26, 35, 36, 37 e 38.

PENTA, M. D. et al. Search-based inference of dialect grammars. *Soft Computing*, Springer, v. 12, n. 1, p. 51–66, 2008. Citado na página 30.

PETRE, M. Mental imagery and software visualization in high-performance software development teams. *Journal of Visual Languages & Computing*, Elsevier, v. 21, n. 3, p. 171–183, 2010. Citado 2 vezes nas páginas 38 e 39.

REISS, S. P. Software visualization in the desert environment. In: ACM. *ACM SIGPLAN Notices*. [S.l.], 1998. v. 33, n. 7, p. 59–66. Citado na página 21.

SABINO, C.; VILLAÇA, R. Estudo comparativo de métodos de amostragem de comunidades de costão. *Revista Brasileira de Biologia*, SciELO Brasil, v. 59, p. 407–419, 1999. Citado na página 73.

SAGER, T. et al. Detecting similar java classes using tree algorithms. In: ACM. *Proceedings of the 2006 international workshop on Mining software repositories*. [S.l.], 2006. p. 65–71. Citado na página 31.

SAMADZADEH, M. H.; NANDAKUMAR, K. A study of software metrics. *Journal of Systems and Software*, Elsevier, v. 16, n. 3, p. 229–234, 1991. Citado na página 23.

SAMPAIO, R. F.; MANCINI, M. C. Estudos de revisão sistemática: um guia para síntese criteriosa da evidência científica. *Rev. bras. fisioter*, v. 11, n. 1, p. 83–89, 2007. Citado na página 45.

SANTOS, J. P. O. dos; BARROS, M. d. O. Codemi–source code as xml metadata interchange uma representação de código-fonte para coleta de métricas. 2009. Citado 3 vezes nas páginas 30, 46 e 51.

SIKET, I.; FERENC, R. Calculating metrics from large c++ programs. In: CITESEER. *6th International Conference on Applied Informatics Eger, Hungary*. [S.l.], 2004. p. 27–31. Citado 4 vezes nas páginas 28, 29, 33 e 34.

STOREY, M.-A. et al. Remixing visualization to support collaboration in software maintenance. In: IEEE. *Frontiers of Software Maintenance, 2008. FoSM 2008.* [S.l.], 2008. p. 139–148. Citado na página 26.

TAPPOLET, J.; KIEFER, C.; BERNSTEIN, A. Semantic web enabled software analysis. *Web Semantics: Science, Services and Agents on the World Wide Web*, Elsevier, v. 8, n. 2, p. 225–240, 2010. Citado 3 vezes nas páginas 48, 51 e 55.

TELEA, A.; BYELAS, H.; VOINEA, L. A framework for reverse engineering large c++ code bases. *Electronic Notes in Theoretical Computer Science*, Elsevier, v. 233, p. 143–159, 2009. Citado 6 vezes nas páginas 27, 29, 33, 47, 50 e 51.

TELEA, A.; VOINEA, L. An interactive reverse engineering environment for large-scale c++ code. In: ACM. *Proceedings of the 4th ACM symposium on Software visualization*. [S.l.], 2008. p. 67–76. Citado na página 40.

TELEA, A.; VOINEA, L. Visual software analytics for the build optimization of large-scale software systems. *Computational Statistics*, Springer, v. 26, n. 4, p. 635–654, 2011. Citado 5 vezes nas páginas 21, 25, 26, 30 e 39.

TELEA, A.; VOINEA, L.; SASSENBURG, H. Visual tools for software architecture understanding: A stakeholder perspective. *Software, IEEE*, IEEE, v. 27, n. 6, p. 46–53, 2010. Citado 4 vezes nas páginas 22, 30, 39 e 40.

TEYSEYRE, A. R.; CAMPO, M. R. An overview of 3d software visualization. *Visualization and Computer Graphics, IEEE Transactions on*, IEEE, v. 15, n. 1, p. 87–105, 2009. Citado na página 40.

TICHELAAR, S. et al. A meta-model for language-independent refactoring. In: IEEE. *Principles of Software Evolution, 2000. Proceedings. International Symposium on*. [S.l.], 2000. p. 154–164. Citado 2 vezes nas páginas 49 e 51.

TSHERING, N. *Fact Extraction for Ruby on Rails Platform*. Dissertação (Mestrado) — Blekinge Institute of Technology, 2010. Citado 4 vezes nas páginas 47, 50, 51 e 55.

UMPHRESS, D. A.; HENDRIX, T. D.; CROSS, J. H. Software process in the classroom: The capstone project experience. *Software, IEEE*, IEEE, v. 19, n. 5, p. 78–81, 2002. Citado 5 vezes nas páginas 22, 37, 38, 39 e 41.

WANGENHEIM, C. G. v. Utilização do gqm no desenvolvimento de software. *Laboratório de Qualidade de Software, Instituto de Informática, Universidade do Vale do Rio dos Sinos. São Leopoldo*, 2000. Citado na página 62.

WÜRSCH, M. et al. Seon: a pyramid of ontologies for software evolution and its applications. *Computing*, Springer, v. 94, n. 11, p. 857–885, 2012. Citado 3 vezes nas páginas 49, 51 e 55.

XENOS, M. et al. Object-oriented metrics-a survey. In: *Proceedings of the FESMA*. [S.l.: s.n.], 2000. p. 1–10. Citado 2 vezes nas páginas 66 e 67.