MESTRADO EM SISTEMAS E COMPUTAÇÃO

THIAGO DE LIMA MARIANO

UM AMBIENTE DE VISUALIZAÇÃO PARA APOIO À COMPREENSÃO DE PROGRAMAS MATLAB E OCTAVE BASEADO NO KNOWLEDGE DISCOVERY METAMODEL (KDM)

Salvador
2017

THIAGO DE LIMA MARIANO

UM AMBIENTE DE VISUALIZAÇÃO PARA APOIO À COMPREENSÃO DE PROGRAMAS MATLAB E OCTAVE BASEADO NO KNOWLEDGE DISCOVERY METAMODEL (KDM)

Dissertação apresentada ao Programa de Pós-Graduação, Mestrado Acadêmico em Sistemas e Computação da UNIFACS Universidade Salvador, Laureate International Universities, como requisito parcial à obtenção do título de Mestre.
Orientador: Prof. Dr. Glauco de Figueiredo Carneiro.

Salvador
2017

FICHA CATALOGRÁFICA
(Elaborada pelo Sistema de Bibliotecas da UNIFACS Universidade Salvador, Laureate International Universities)

Mariano, Thiago de Lima

Um ambiente de visualização para apoio à compreensão de programas MATLAB e Octave baseado no Knowledge Discovery Metamodel (KDM)./ Thiago d Lima Mariano. – Salvador: UNIFACS, 2017.

121 f. : il.

Dissertação apresentada ao Programa de Pós- Graduação em Ciência da Computação da UNIFACS Universidade Salvador, Laureate International Universities como parte dos requisitos para a obtenção do título de Mestre.

Orientador: Prof. Dr. Glauco de Figueiredo Carneiro.

1. MATLAB. 2. Visualização de software. I. Carneiro, Glauco de Figueiredo, orient. II. Título.

CDD: 005.3

TERMO DE APROVAÇÃO

THIAGO DE LIMA MARIANO

UM AMBIENTE DE VISUALIZAÇÃO PARA APOIO À COMPREENSÃO DE PROGRAMAS MATLAB E OCTAVE BASEADO NO KNOWLEDGE DISCOVERY METAMODEL (KDM)

Dissertação aprovada como requisito parcial para obtenção do grau de Mestre em Ciência da Computação, UNIFACS Universidade Salvador, Laureate International Universities, apresentada à seguinte banca examinadora:

Glauco de Figueiredo Carneiro – Orientador _____
Pós Doutor pela University of Wisconsin-Milwaukee (EUA)
UNIFACS Universidade Salvador, Laureate International Universities

Marcelo de Almeida Maia _____
Doutor em Ciência da Computação pela Universidade Federal de Minas Gerais
Universidade Federal de Uberlândia - UFU

Artur Henrique Kronbauer _____
Doutor em Ciência da Computação, pela Universidade Federal da Bahia (UFBA)
UNIFACS Universidade Salvador, Laureate International Universities

Salvador, 5 de maio de 2017.

Dedico este trabalho às minhas filhas Maria Luiza Mariano e Maria Fernanda Mariano e à minha esposa Luana Mariano, meus grandes tesouros.

AGRADECIMENTOS

Agradeço em primeiro lugar a Jesus Cristo, minha fonte de inspiração e modelo a ser seguido. À minha esposa, por todo apoio, carinho e dedicação. À minha filha por me fazer sorrir nos momentos mais difíceis, me incentivando a sempre ir para frente. Aos meus pais, minhas irmãs e minha avó materna por terem contribuído significativamente na minha formação de caráter e personalidade. Sem isso, eu não teria chegado até aqui.

À minha irmã postiça Rosane Coutinho, pelo incentivo e carinho. Ainda sou grato ao meu sogro e minha sogra por me incentivarem nos estudos me impulsionando a seguir em frente.

Faço um agradecimento especial ao meu orientador, Profº. Drº. Glauco de Figueiredo Carneiro, que não desistiu de mim e me cedeu muitas horas do seu tempo para que este trabalho fosse realizado. Ele foi de extrema importância para a conclusão desta dissertação, me apoiando nos momentos mais difíceis.

Agradeço aos meus amigos Tainnan e Rogério, por estarem ao meu lado e me ajudarem a desestressar nos momentos difíceis. Ao meu colega de mestrado e amigo Ivan Lessa, pelo incentivo e apoio. Ao meu chefe Serge Normando Rehem pelo seu apoio e aos meus colegas de trabalho e amigos Robson, Marlon, Cleverson e Wilson.

Agradeço ainda às minhas colegas de trabalho e amigas Regina, Izabel e Márcia, que aliviaram minha barra muitas vezes no Serpro, me concedendo mais tempo para finalizar este trabalho. E um agradecimento ao Serpro por financiar parte do meu mestrado e por ceder horas do meu trabalho para dedicação a este projeto.

“ There are only two ways to live your life. One is as though nothing is a miracle. The other is as though everything is a miracle.”

Albert Einstein.

RESUMO

Os Ambientes de Visualização são ferramentas que utilizam recursos visuais para apoiar a compreensão de um conjunto de dados de um determinado domínio. Esses recursos visuais são agrupados em representações denominadas Metáforas Visuais. A área da computação responsável por estudar essas técnicas é chamada de Visualização da Informação. É comum encontrar na literatura relatos sobre o uso desse tipo de ferramenta e técnica para apoiar a compreensão dos códigos fonte dos programas de computadores. Essa atividade é uma especialização da Visualização de Informação que recebe o nome de Visualização de Software. Já existem muitos estudos sobre a Visualização de Software como uma ferramenta de apoio para análise e compreensão de programas desenvolvidos em Java e outras linguagens mais conhecidas e utilizadas no mundo da computação. Porém, existe uma carência de estudos nessa área em cima de programas desenvolvidos nas linguagens MATLAB e Octave. Essas linguagens, apesar de serem menos conhecidas, têm papel fundamental nas áreas de engenharia e matemática, visto que essas linguagens fornecem um ambiente rico e acessível para a criação de exibições de dados usando objetos gráficos. Esta dissertação propõe uma ferramenta de visualização web para apoiar a compreensão de programas MATLAB e Octave, através do uso de um metamodelo proposto para a área de reengenharia de software chamado Knowledge Discovery Metamodel (KDM). Esse metamodelo é capaz de armazenar todos os elementos contidos nos códigos fonte de aplicações e suas respectivas relações.

Palavras-chaves: Knowledge Discovery Metamodel (KDM). MATLAB. Octave. Visualização de Software.

ABSTRACT

Visualization Environments are tools that use visual resources to support the under-standing of a set of data in a particular domain. These visual resources are grouped into representations called Visual Metaphors. The area of computing responsible for studying these techniques is called Information Visualization. It is common to find in the literature reports on the use of this type of tool and technique to support the understanding of the source codes of computer programs. This activity is a specialization of Information Visualization called Software Visualization. There are already many studies on Software Visualization as a tool to support analysis and understanding of programs developed in Java and other languages better known and used in the world of computing. However there is'nt many studies in this area over programs developed in the languages MATLAB and Octave. Although these languages are less well-known, they are very important in engineer-ing and mathematics areas, as these languages provide a rich and accessible environment for creating data views using graph objects. This dissertation proposes a web visualization tool to support the understanding of MATLAB and Octave programs through the use of a metamodel proposed for the software reengineering area, called Knowledge Discovery Metamodel (KDM). This metamodel is capable of storing all the elements contained in the source codes of applications and their respective relationships.

Keywords: Knowledge Discovery Metamodel (KDM). MATLAB. Octave. Software Visualization.

LISTA DE FIGURAS

Figura 1 - Exemplo de Conversão de uma Expressão Matemática para uma Expressão em Árvore 20
Figura 2 - (a) Código (b) AST Correspondente Exemplo de Código Fonte e sua Respectiva AST 21
Figura 3 - Parte do Código da Biblioteca Parsetree do GNU Octave 22
Figura 4 - Estrutura de Pacotes do KDM 33
Figura 5 - Domínios de Conhecimento e Níveis de Compliance do KDM 36
Figura 6 - Representação Visual Treemap 39
Figura 7 - Metodologia de Pesquisa 42
Figura 8 - Parser from Source Code to respective KDM 48
Figura 9 - Diagrama que Exibe a Interação entre as Cinco Bibliotecas da Solução Proposta 50
Figura 10 - Fluxo para Geração da Estrutura KDM 52
Figura 11 - Fluxo Utilizado pela Biblioteca ASTOctave para Geração da AST 65
Figura 12 - Exemplo de Conversão de um Código Fonte Octave em sua Respectiva AST 69
Figura 13 - Estrutura Hierárquica dos Objetos KDM 70
Figura 14 – Diagrama de classes de Biblioteca OctaveKDMStructure 73
Figura 15 - (a) Fluxo para geração do arquivo XMI (b) Fluxo para geração do arquivo Json 74
Figura 16 - Saída do Arquivo XMI 76
Figura 17 - Saída do Arquivo JSON 76
Figura 18 - Tecnologias Utilizadas na Arquitetura do Ambiente de Visualização 80
Figura 19 - Arquitetura do Ambiente de Visualização 81
Figura 20 - Exemplo de um Documento Json que Representa o Objeto KDMSegment 82
Figura 21 - Estrutura de Dados de um Arquivo JSON que Representa o Elemento CallableUnit do KDM Referenciado na Parte B da Figura 19 84
Figura 22 - Estrutura de Dados de um Arquivo JSON que Representa o Elemento BlockUnit do KDM Referenciado na Parte B da Figura 19 85
Figura 23 - Estrutura de Dados de um Arquivo JSON que Representa o Elemento CompIlationUnit do KDM Referenciado na Parte B da Figura 19 85
Figura 24 - Estrutura de Dados de um Arquivo JSON que Representa o Elemento Code-Model do KDM Referenciado na Parte B da Figura 19 85
Figura 25 - Exemplo de uma View no CouchDB 86
Figura 26 - Resultado da Execução da View Exemplificada na Figura 25 86
Figura 27 - Exemplo de um Arquivo couchdb.properties 89
Figura 28 - Exemplo de um Arquivo Json Contendo os Dados para Alimentar a Visão Treemap 90
Figura 29 - Exemplo de um Arquivo Json Contendo os Dados para Alimentar a Visão Chord .. 90

Figura 30 - Tela Treemap do Ambiente de Visualização Fazendo o Agrupamento pelo Tamanho dos Arquivos de Código Fonte 92

Figura 31 - Tela Treemap do Ambiente de Visualização Fazendo o Agrupamento pela Quantidade de Funções dos Arquivos de Código Fonte 93

Figura 32 - Tela Chord do Ambiente de Visualização 94

Figura 33 - Tela Treemap do Ambiente de Visualização Fazendo o Agrupamento pela Quantidade de Relações Eferentes dos Arquivos de Código Fonte 95

Figura 34 - Tela Treemap do Ambiente de Visualização Fazendo o Agrupamento pela Quantidade de Relações Aferentes dos Arquivos de Código Fonte 96

Figura 35 - Tela Inicial do Ambiente de Visualização 96

Figura 36 - Tela de Importação dos projetos do Ambiente de Visualização 97

Figura 37 - Etapas do Estudo Exploratório 99

Figura 38 - Estratégia Proposta para a Análise de Programas MATLAB/Octave 105

Figura 39 - Visão Treemap do Projeto MATLAB2Tikz que Representa o Tamanho de suas Funções e Arquivos 106

Figura 40 - Visão Treemap do Projeto MATLAB2Tikz que Representa a Quantidade de Relações dos Arquivos 107

Figura 41 - Visão Treemap do Projeto MATLAB2Tikz que Representa a Quantidade de Dependências dos Arquivos 107

Figura 42 - Visão Chord do Projeto MATLAB2Tikz que Representa as Relações Entre os Arquivos 108

Figura 43 - Visão Chord do Projeto MATLAB2Tikz que Representa as Dependências dos Arquivos 110

Figura 44 - (a) Resultado do Estudo no Projeto (b) Resultado do Estudo no Projeto DeepLearnToolbox 111

export_fig-master 111

Figura 45 - (a) Resultado do Estudo no Projeto (b) Resultadodo Estudo no Projeto fieldtrip- master HDR_Toolbox-master 111

Figura 46 - (a) Resultado do Estudo no Projeto (b) Resultado do Estudo no Projeto gpstuff-master 111

Figura 47 - (a) Resultado do Estudo no Projeto (b) Resultado do Estudo no Projeto 112

Figura 48 - Resultado do Estudo no Projeto octave-networks-toolbox-master 112

Figura 49 - Resultado do Estudo no Projeto pmtk3-master 112

LISTA DE ALGORITMOS

Algoritmo 1 - Estrutura de um Teste Condicional MATLAB/Octave 27
Algoritmo 2 - Estrutura de um SWITCH/CASE MATLAB/Octave 28
Algoritmo 3 - Estrutura WHILE das Linguagens MATLAB e Octave 28
Algoritmo 4 - Estrutura FOR das Linguagens MATLAB e Octave 29
Algoritmo 5 - Estrutura DO das Linguagens MATLAB e Octave 29
Algoritmo 6 - Estrutura TRY das Linguagens MATLAB e Octave 30
Algoritmo 7 - Estrutura de uma Função nas Linguagens MATLAB e Octave 30
Algoritmo 8 - Estrutura de uma Classe das Linguagens MATLAB e Octave 31

LISTA DE TABELAS

Tabela 1 - Operadores Aritméticos das Linguagens MATLAB/Octave 26
Tabela 2 - Operadores Relacionais das Linguagens MATLAB/Octave 26
Tabela 3 - Operadores Lógicos das Linguagens MATLAB e Octave 27
Tabela 4 - Descrição dos Pacotes KDM 33
Tabela 5 - Objetos KDM e suas Rrespectivas Saídas XMI 75
Tabela 6 Objetos KDM e suas Rrespectivas Saídas JSON 75
Tabela 7 - Lista dos Requisitos Funcionais e não Funcionais do Ambiente de Visualização 79
Tabela 8 - Critérios Utilizados para Seleção dos Projetos 101
Tabela 9 - Resultado dos arquivos candidatos à God File dos 10 projetos analisados 113

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
AST	Árvore Sintática Abstrata
IDE	Integrated Development Environment
KDM	Knowledge Discovery Metamodel
OMG	Object Management Group
ADM	Architecture Driven Mmodernization
XMI	XML Metadata Interchange
SOA	Service-Oriented Architecture
MEG	Magnetoencefalografia
EGG	Eletroencefalografia
REST	Representational State Transfer
SGBD	Sistema de Gerenciamento de Banco de Dados
HDR	High Dynamic Range

SUMÁRIO

1 INTRODUÇÃO 17
1.1 MOTIVAÇÃO DO TRABALHO 17
1.2 PROBLEMA ABORDADO 18
1.3 PROPOSTA PARA RESOLUÇÃO DO PROBLEMA 19
1.4 ESTRUTURA DA DISSERTAÇÃO 22
1.5 CONCLUSÃO DO CAPÍTULO 23
2 FUNDAMENTAÇÃO TEÓRICA 24
2.1 AS LINGUAGENS MATLAB E GNU/OCTAVE 24
2.2 KNOWLEDGE DISCOVERY METAMODEL (KDM) 31
2.3 COMPREENSÃO DE PROGRAMAS 36
2.4 VISUALIZAÇÃO DE SOFTWARE 38
2.5 TRABALHOS RELACIONADOS 40
2.6 CONCLUSÃO DO CAPÍTULO 41
3 METODOLOGIA DE PESQUISA 42
3.1 ETAPAS ADOTADAS NESTA DISSERTAÇÃO 42
3.2 DEFINIÇÃO DAS QUESTÕES DE PESQUISA 43
3.3 PROPOSTA DA SOLUÇÃO PARA RESPONDER AS QUESTÕES DE PESQUISA 44
3.4 IMPLEMENTAÇÃO DO PARSER 44
3.5 IMPLEMENTAÇÃO DO AMBIENTE DE VISUALIZAÇÃO 45
3.6 ESTUDO EXPLORATÓRIO 45
3.7 ANÁLISE E RESPOSTA DAS QUESTÕES DE PESQUISA 46
3.8 CONCLUSÃO DO CAPÍTULO 46
4 UMA PROPOSTA DE UM PARSER DE APLICAÇÕES MATLAB E OCTAVE 47
4.1 PASSOS NECESSÁRIOS AO DESENVOLVIMENTO DO PARSER 47
4.2 SOLUÇÃO ADOTADA 47
4.3 BIBLIOTECA ASTOCTAVETOKDM 51
4.4 BIBLIOTECA ASTOCTAVE 58
4.5 BIBLIOTECA OCTAVEKDMSTRUCTURE 69
4.6 BIBLIOTECA OCTAVEKDMTOXMI E OCTAVEKDMTOJSON 74
4.7 CONCLUSÃO DO CAPÍTULO 77
5 UM AMBIENTE DE VISUALIZAÇÃO BA-SEADO EM MULTIPLAS VISÕES PARA A COMPREENSÃO DE PROGRAMAS MA-TLAB E OCTAVE 78
5.1 INFRAESTRUTURA DO AMBIENTE DE VISUALIZAÇÃO 78
5.2 DETALHES DE IMPLEMENTAÇÃO DO AMBIENTE DE VISUALIZAÇÃO 89
5.3 CONCLUSÃO DO CAPÍTULO 98

6 ESTUDO EXPLORATÓRIO DO AMBIENTE DE VISUALIZAÇÃO PARA ATIVIDADES DE COMPREENSÃO 99
6.1 QUESTÕES DE PESQUISA PARA O ESTUDO EXPLORATÓRIO 99
6.2 O PROTOCOLO DE ESTUDO 100
6.3 SELEÇÃO DE PROJETOS 101
6.4 ESTRATÉGIA SUGERIDA PARA USO DO AMBIENTE 103
6.5 COLETA E ANÁLISE DOS DADOS 104
6.6 RESPOSTAS DAS QUESTÕES DE PESQUISA DA DISSERTAÇÃO 113
6.7 CONCLUSÃO DO CAPÍTULO 115
7 CONCLUSÃO E PERSPECTIVAS FUTURAS 116
7.1 CONSIDERAÇÕES FINAIS 116
7.2 CONTRIBUIÇÕES 116
7.3 LIÇÕES APRENDIDAS 117
7.4 LIMITAÇÕES 117
7.5 TRABALHOS EM ANDAMENTO 118
REFERÊNCIAS 119

1 INTRODUÇÃO

Este capítulo apresenta a motivação desta dissertação de mestrado, o problema a ser resolvido, a proposta para resolução do problema (com uma breve descrição da ferramenta desenvolvida), a estrutura da dissertação e a conclusão do capítulo.

1.1 MOTIVAÇÃO DO TRABALHO

A quantidade de dados disponíveis no mundo virtual é muito grande e o número de publicações vêm crescendo de forma exponencial a partir do início dos anos 70 (EVELIEN; RONALD, 2002) e (BORGATTI S.P.; FOSTER, 2003). Transformar essa grande quantidade de dados em informações que possam ser visualizadas e compreendidas com facilidade pelo ser humano é uma tarefa de grande valor e que está presente na computação há muito tempo, através da visualização da informação.

A visualização da informação pode ser entendida como uma área da computação que procura representar visualmente informações, permitindo uma melhor compreensão de um grande volume de dados (IEPSEN E. F.; LUZZARDI, 2007). Essas representações visuais são chamadas de metáforas visuais.

Uma metáfora pode ser considerada como o uso de conceitos e conhecimentos a partir de uma área da experiência humana para entender melhor a estrutura, os fenômenos e conceitos de outra que, a princípio, é mais abstrata (AVERBUKH et al., 2007); é a ideia principal que determina o mapeamento do domínio de aplicação para o mundo visual (AVERBUKH, 2001).

Essa grande quantidade de dados disponíveis para visualização, associada a diversidades de informações, criam dificuldades na exploração, interpretação e compreensão desses dados na busca de conhecimento. É por isso que existe uma grande preocupação, por parte de pesquisadores, em projetar e desenvolver ferramentas que possibilitem um acesso fácil e rápido a essa informação.

Uma das especialidades da visualização da informação é a visualização de software, que consiste em utilizar técnicas de visualização para apoiar o processo de compreensão dos códigos-fonte de um programa por parte do ser humano.

Como os programas desenvolvidos tem se tornado cada vez mais complexos e a quantidade de informações que podem ser analisadas em seus códigos-fonte é cada vez maior, é importante e se faz necessário o uso de ferramentas de visualização que auxiliem os

engenheiros de software a compreender melhor os códigos-fonte desenvolvidos, suas complexidades, relacionamentos, entre outras características.

1.2 PROBLEMA ABORDADO

Atualmente, há um conjunto significativo de recursos que podem ser analisados dentro de um software MATLAB/Octave para garantir sua qualidade. O grau de acoplamento, a modularidade (tangling e espalhamento), o nível de granularidade e a complexidade ciclomática são algumas dessas características que podem ser utilizadas para avaliar softwares desenvolvidos com essas linguagens de programação.

Embora exista a necessidade de compreensão de softwares desenvolvidos nas linguagens MATLAB/Octave, foi constatado, através de pesquisas em repositórios científicos, que há uma carência muito grande de ferramentas de representação visual que apoiem a análise de arquiteturas e das estruturas desse tipo de aplicação.

Isso não significa dizer que não haja ferramentas que suportem o desenvolvimento de aplicações MATLAB e Octave. Ao contrário, é possível mencionar MATLAB Online, Octave Online, JDoodle e Coding Ground. MATLAB online é uma versão web da ferramenta de desenvolvimento de software Mathworks. Nessa versão, os arquivos de código fonte são armazenados em MATLAB Drive. Octave online é uma ferramenta web utilizada no desenvolvimento de aplicações Octave. JDoodle e Coding Ground são ferramentas da web para desenvolver aplicações em várias linguagens de programação, incluindo MATLAB e Octave. O problema é que nenhuma delas parece disponibilizar alguma funcionalidade para analisar visualmente códigos fonte MATLAB/Octave.

Além da escassez de programas para análises de código fonte MATLAB/Octave um outro problema significativo é o de escalabilidade: muitos programas compreendem dezenas, centenas ou até mesmo milhares de arquivos, com cada arquivo contendo dezenas, centenas ou milhares de linhas de programação. São necessárias abordagens, técnicas ou ferramentas para lidar com esses grandes volumes de dados e extraí-los dos códigos fonte para analisá-los.

Uma solução é organizar todas as informações contidas no código-fonte em uma estrutura que facilite a busca por informações úteis para posterior análise. Através de um processo de revisão da literatura, foram encontradas muitas referências a histórias de sucesso sobre o uso do Knowledge Discovery Metamodel (KDM) (OMG, 2016) [1] no que se refere ao seu apoio aos processos de modernização dos sistemas legados. O KDM faz parte de um

conjunto de metamodelos propostos pelo OMG (Object Management Group) que suporta a descrição de sistemas de software em diferentes representações arquitetônicas (PÉREZ-CASTILLO; GUZMAN; PIATTINI, 2011) (ULRICH, 2004) (CANOVAS; MOLINA, 2010) (BOUSSAIDI et al., 2012) (DELTOMBE; GOAER; BARBIER, 2012).

O objetivo principal do KDM é descrever um conjunto de classes distribuídas em doze pacotes que são usadas para representar os sistemas de software, bem como suas partes constituintes e suas relações (OMG, 2016). O KDM representa os elementos físicos e lógicos de um sistema, bem como suas relações em vários níveis de abstração (PÉREZ-CASTILLO; GUZMAN; PIATTINI, 2011). Embora sua função primária esteja relacionada à reengenharia, o metamodelo KDM também pode ser usado para extrair informações anteriormente indisponíveis sobre o sistema, suportando processos de análise de dados. Usar o KDM implica em ler o código fonte e armazená-lo em uma estrutura de acordo com o metamodelo.

Como o desenvolvimento de software é uma atividade que está em constante evolução, é possível que existam características de um software, ou na forma como ele foi desenvolvido, não previstas e contempladas pelo KDM. Nesse caso, devido à flexibilidade desse metamodelo, é perfeitamente viável que pontos de extensão possam ser propostos ao KDM tornando-o ainda mais abrangente. Como exemplo, é possível citar o paradigma de orientação à aspectos que não está contemplado no metamodelo, mas que já existem pontos de extensão propostos com objetivo de incluí-lo.

Até o momento, não foi encontrado nenhum relato na literatura que valide a utilização do KDM, sem pontos de extensão, para representar softwares desenvolvidos nas linguagens MATLAB e Octave. Também não existe nenhum relato na literatura mostrando que é possível utilizar esse metamodelo em conjunto com técnicas e/ou ferramentas de visualização de software.

Essa junção de tecnologias pode trazer inúmeras possibilidades para a engenharia de software, principalmente para o processo de compreensão de programas MATLAB/Octave, que atualmente é uma área muito carente de ferramentas de apoio. Mas para isso, é preciso mostrar, a um baixo custo, a viabilidade dessa forma de trabalho.

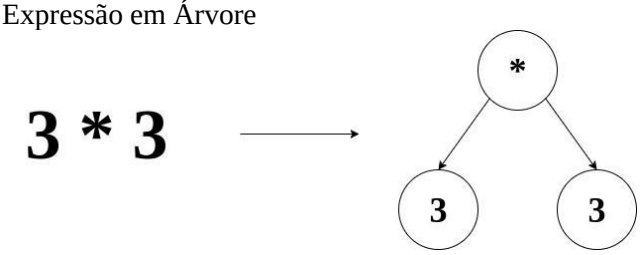
1.3 PROPOSTA PARA RESOLUÇÃO DO PROBLEMA

Esta dissertação propõe uma ferramenta para suportar tarefas de análise e compreensão dos programas MATLAB e Octave. A ferramenta incorpora um parser que converte os

códigos fonte MATLAB/Octave em uma estrutura de dados de acordo com o KDM, a partir da qual métricas e outras informações podem ser derivadas e sujeitas a vários tipos de análise.

No contexto dos compiladores de linguagens de programação, um parser corresponde a uma fase que analisa as sequências de tokens e constrói uma estrutura de dados hierárquica de acordo com regras sintáticas preestabelecidas. Na maioria dos casos, a saída dessa transformação é uma estrutura em árvore, cujo processo de construção é executado de maneira top-down ou bottom-up. No primeiro caso, a árvore é construída a partir da raiz para os nós folha, enquanto no modo bottom-up o caminho é percorrido da forma contrária (SEIDL; WILHELM; HACK, 2012) (AHO et al., 2006). A figura 1 mostra o resultado da análise de uma expressão aritmética simples. Em um contexto mais amplo, um parser é responsável por aplicar regras a um dado conteúdo, gerando, em última análise, uma saída diferente da entrada original. Existem muitos tipos diferentes de parsers, por exemplo, para transformar arquivos XML em JSON, para transformar expressões matemáticas em expressões em formato de notação polonesa e para extrair árvores de sintaxe abstratas (AST) de códigos-fonte.

Figura 1 - Exemplo de Conversão de uma Expressão Matemática para uma Expressão em Árvore

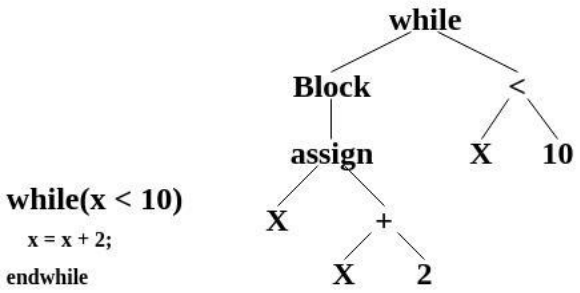


O objetivo principal do parser desenvolvido é a geração de representações KDM de programas MATLAB/Octave, que serão posteriormente utilizadas para suportar processos de análise e compreensão dos sistemas alvo. No entanto, a maneira como o algoritmo é organizado dentro dos códigos fonte torna muito difícil gerar as instâncias do KDM. Por esta razão, foi decidido procurar as informações a partir da AST dos códigos fonte.

Uma AST normalmente representa, de forma hierárquica, a estrutura sintática de um artefato de código-fonte. Cada nó na árvore corresponde a um elemento encontrado no código. É abstract porque não exibe todos os detalhes encontrados na sintaxe real (SEIDL; WILHELM; HACK, 2012) (AHO et al., 2006). As figuras 2(a) e 2(b) mostram, respectivamente, um pedaço de código escrito na linguagem Octave e sua AST correspondente.

Já existem alguns parsers desenvolvidos para as linguagens de programação MATLAB e Octave, mas nenhum deles produz representações KDM. À exemplo é possível enumerar os seguintes parsers desenvolvidos para as linguagens MATLAB/Octave e que estão disponíveis para uso: Octclipse, GNU Octave e ANTLRv4.

Figura 2 - (a) Código (b) AST Correspondente Exemplo de Código Fonte e sua Respectiva AST



Fonte: Octave.

O Octclipse é um plugin do ambiente de desenvolvimento integrado (IDE) Eclipse que suporta o desenvolvimento de aplicativos Octave. No entanto, Octclipse não tem nenhuma atividade de desenvolvimento desde 2013 e, de acordo com sua documentação, é compatível até a versão 3.6.x de Octave. Por esta razão, alguns recursos encontrados em versões mais recentes desta linguagem não são cobertos por seu analisador. O mesmo acontece com ANTLRv4, um analisador capaz de gerar o AST de programas MATLAB. A manutenção deste parser também parece ter parado desde 2013 e não suporta os recursos mais recentes da linguagem. Além disso, o analisador não parece cobrir certos recursos, como instruções de switch, structures e cell arrays.

O código fonte do GNU Octave é interpretado diretamente e dentro de seu interpretador existe uma biblioteca parse-tree que gera a AST do código. Essa biblioteca foi desenvolvida em C ++. Apesar de abranger todos os recursos do Octave, ele não inclui documentação sobre seu código-fonte. A implementação do analisador é muito complexa com nomes e métodos de atributos difíceis de entender. Para ilustrar, alguns métodos encontrados no código-fonte do analisador incluem: yypull_parse, yypstate_new, yyparse, yyalloc e yytnamerr. A figura 3 mostra um snippet de exemplo de código encontrado no analisador. Naturalmente, o interpretador MATLAB também tem um analisador, mas não é possível verificar o tipo de saída gerada, uma vez que esta linguagem é proprietária e, portanto, não fornece seu código fonte.

A primeira proposta para o desenvolvimento do parser foi usar uma das três bibliotecas mencionadas acima para gerar a AST, e, a partir dela, gerar a estrutura KDM dos programas MATLAB / Octave. Essa solução não foi possível devido às limitações encontradas em algumas bibliotecas e à dificuldade em entender o código fonte do parse-tree. Como existem pequenas diferenças entre as linguagens MATLAB e Octave, seria necessário fazer alterações no código-fonte do parse-tree para contemplar os programas desenvolvidos na linguagem MATLAB. Alterar o código-fonte desse analisador faria mais trabalho do que implementá-lo a partir do zero devido à dificuldade de entender seu código-fonte. Portanto, foi necessário desenvolver um analisador intermediário capaz de gerar o AST de programas MATLAB/Octave a partir de seu código-fonte e outro para gerar instâncias de KDM a partir de AST.

Figura 3 - Parte do Código da Biblioteca Parsetree do GNU Octave

```
static void
yy_symbol_value_print (FILE *yyoutput, int yytype, YYSTYPE const * const yyvaluep, octave::base_parser& parser)
{
  FILE *yyo = yyoutput;
  YYUSE (yyo);
  YYUSE (parser);
  if (!yyvaluep)
    return;
# ifdef YYPRINT
  if (yytype < YYNTOKENS)
    YYPRINT (yyoutput, yytoknum[yytype], *yyvaluep);
# endif
  YYUSE (yytype);
}
```

1.4 ESTRUTURA DA DISSERTAÇÃO

Esta dissertação está organizada em mais seis capítulos distribuídos da seguinte forma: o capítulo 2 discorre sobre a fundamentação teórica necessária para o desenvolvimento da proposta apontada. O capítulo 3 apresenta a metodologia de pesquisa adotada. O capítulo 4 explica sobre os passos necessários para a implementação do parser desenvolvido, detalhes de sua implementação e todas as suas características. O capítulo 5 mostra a ferramenta de visualização desenvolvida. O capítulo 6 demonstra o uso da ferramenta para análises e compreensão de programas MATLAB/Octave e o capítulo 7 apresenta a conclusão, as limitações da proposta apresentada, suas contribuições e futuros trabalhos relacionados.

1.5 CONCLUSÃO DO CAPÍTULO

Este capítulo apresentou a motivação desta dissertação de mestrado, o problema a ser resolvido, a proposta para resolução do problema com uma breve descrição da ferramenta desenvolvida e a estrutura da dissertação. No próximo capítulo verifica-se a fundamentação teórica necessária para o entendimento deste trabalho. Serão aprofundados os conceitos de compreensão de programas, visualização de Software, metamodelo KDM e as linguagens de programação MATLAB e Octave.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta os principais conceitos utilizados nesta dissertação. Serão discutidos mais profundamente os aspectos sobre o metamodelo KDM, as linguagens de programação MATLAB e Octave, a compreensão de programas e a visualização da informação.

2.1 AS LINGUAGENS MATLAB E GNU/OCTAVE

MATLAB é uma linguagem de programação interpretada que fornece um ambiente rico e acessível para a criação de exibições de dados usando objetos gráficos. Cada objeto gráfico tem um conjunto de características que podem ser facilmente manipuladas através de suas configurações de propriedade (MAJUMDAR, 2012). Essa linguagem é otimizada para o processamento numérico e sua aplicabilidade é grande na área educacional, em especial, no ensino da álgebra linear e análise numérica, e se tornou popular entre os cientistas envolvidos no processamento de imagem.

Apesar de ser muito utilizada, esta linguagem é proprietária e, portanto, não disponibiliza acesso ao seu código fonte. Além disso, o preço para utilizá-la não é barato. Essa característica restringe muito a criação de ferramentas para apoio ao MATLAB. Por esse motivo, a comunidade de software livre resolveu desenvolver uma linguagem similar ao MATLAB que também tivesse a capacidade de construir objetos gráficos e resolver problemas matemáticos com tanta eficiência. Foi então que surgiu a linguagem de programação GNU Octave. O Octave é uma linguagem open source e free.

Embora existam pequenas diferenças na syntax dessas linguagens, a compatibilidade entre elas é de mais de 90%. A principal diferença entre essas duas linguagens está nos nomes das funções disponibilizadas por ambas. Além disso, na linguagem Octave é possível encerrar um bloco de código através das palavras reservadas END, ENDFUNCTION, END_TRY_CATCH, END_UNWIND_PROTECT, ENDFOR, ENDGRENT, ENDIF, ENDPARFOR, ENDPWENT, ENDSWITCH e ENDWHILE, enquanto no MATLAB só é possível utilizar a palavra reservada END. Outra diferença está na forma de fazer comentário. Enquanto que no MATLAB os comentários são feitos através do caractere %, no Octave é possível utilizar o mesmo caractere mais o # (SHARMA; GOBBERT, 2010).

Existem ainda outras pequenas diferenças entre essas duas linguagens, mas que não afetam a compatibilidade entre elas.

Assim como a maioria das linguagens, o MATLAB e o Octave disponibilizam um conjunto de recursos que podem ser utilizados no desenvolvimento de algoritmos computacionais (CHAPMAN, 2003). Esses recursos são os operadores lógicos, operadores aritméticos, operadores de incremento e decremento (disponível apenas no Octave), operadores relacionais, sentenças, laços de repetição, estrutura condicional, blocos de código, tratamento de exceções, além da possibilidade de construir funções.

A tabela 1 identifica os operadores aritméticos que podem ser utilizados por essas linguagens de programação. Através deles é possível fazer cálculos numéricos como soma, subtração, divisão, multiplicação, potenciação e funções de transpose. Os operadores de incremento e decremento só são aceitos na linguagem Octave.

A tabela 2 identifica os operadores relacionais das linguagens MATLAB e Octave. Através desses operadores é possível fazer comparações entre dois elementos. Eles podem ser utilizados dentro da estrutura de outros elementos das linguagens como laços de repetição ou estrutura condicional. O operador relacional “!=” só é aceito na linguagem Octave.

A tabela 3 apresenta os operadores lógicos das linguagens MATLAB e Octave. Esses operadores são utilizados para efetuar as operações lógicas AND, OR e NOT entre dois elementos ou entre conjuntos de elementos. Esses operadores também podem ser utilizados em conjunto com outras operações das linguagens como estruturas de repetição e condicional. O operador lógico “!” só está disponível na linguagem Octave.

Tabela 1 - Operadores Aritméticos das Linguagens MATLAB/Octave

Operador Aritmético	Descrição	Operador Aritmético	Descrição
+	Soma	\	Resto da divisão
.+	Soma elemento por elemento	.\	Resto da divisão elemento por elemento
-	Subtrai	^	Potenciação
.-	Subtrai elemento por elemento	.^	Potenciação elemento por elemento
*	Multiplica	**	Potenciação
.*	Multiplica elemento por elemento	.**	Potenciação elemento por elemento
/	Divide	’	Transpose
./	Divide elemento por elemento	.’	Transpose elemento por elemento
++	Auto incremento	–	Auto decremento

Tabela 2 - Operadores Relacionais das Linguagens MATLAB/Octave

Operador Relacional	Descrição
<	Menor que
<=	Menor ou igual que
==	Igual a
>	Maior que
>=	Maior ou igual que
!=	Diferente de
=	Diferente de

Tabela 3 - Operadores Lógicos das Linguagens MATLAB e Octave

Operador Lógico	**Descrição**
&	AND
|	OR
!	NOT
~	NOT

Além dos operadores citados, as linguagens MATLAB e Octave disponibilizam um conjunto de statements que podem ser utilizados para as mais variadas funções. O statement IF pode ser utilizado para criar uma estrutura de teste condicional que verifica se a condição presente na sentença é verdadeira ou não. Caso a sentença seja verdadeira, o interpretador executará as linhas de código que estão dentro do bloco IF. Caso negativo, serão executadas as linhas de código dentro do bloco ELSE. Nessa estrutura de repetição ainda é possível adicionar uma sequência de comandos denominados ELSEIF. Esse tipo de comando é utilizado para fazer um novo teste caso o primeiro não tenha sido satisfeito.

Algoritmo 1 - Estrutura de um Teste Condicional MATLAB/Octave

```
1: if x < 0 then
2:     return "número negativo";
3: else
4:     return "número positivo";
```

O statement SWITCH pode ser utilizado para fazer com que o interpretador siga por diferentes caminhos, a depender do valor armazenado dentro de uma variável.

Algoritmo 2 - Estrutura de um SWITCH/CASE MATLAB/Octave

```
1: switch (x)
2:       case 1
3:            do_something();
4:       case 2
5:            do_something2();
6:       otherwise
7:            do_something_different();
8:       endswitch
```

Essas linguagens também disponibilizam estruturas de repetição. Ao todo são três estruturas que podem ser utilizadas. Os statements WHILE, DO e FOR. Todos os três statements possuem um bloco de código que será executado, caso uma determinada condição seja satisfeita.

O statement WHILE testa a condição e caso ela seja verdadeira, o bloco de código é executado e continua executando até que a condição não seja mais verdadeira. No código, exemplo abaixo, a sentença x = x + 2 será executada cinco vezes. Na quinta execução, o valor atribuído à variável x corresponderá à constante 10. Nesse caso, a condição x < 10 deixa de ser satisfeita e o bloco não é mais executado.

Algoritmo 3 - Estrutura WHILE das Linguagens MATLAB e Octave

```
1: x = 0;
2: while (x < 10)
3:       x = x + 2;
4: endwhile
```

O statement FOR executa um bloco de código n vezes, sendo que esse valor n já é configurado no início da estrutura. No exemplo abaixo, a sentença x = x + 2 será executada nove vezes, pois, no início da estrutura de repetição FOR, é criada uma variável X que terá seu valor iniciado em 2 e variará de uma em uma unidade até chegar no valor 10.

Algoritmo 4 - Estrutura FOR das Linguagens MATLAB e Octave

```
1: for (x : 2 : 10)
2:       x = x + 2;
3: endfor
```

O statement DO executa um bloco de código repetidas vezes até que uma determinada condição seja verdadeira. Porém, existe uma grande diferença entre essa estrutura de repetição e as demais. O bloco de código será executado, pelo menos, uma vez independente da condição já ser verdadeira. Isso acontece porque a condição só é testada depois da execução do bloco de código. No exemplo a seguir, a sentença x = x + 2 será executada uma vez mesmo que o valor de x já satisfaça a condição de ser maior do que a constante 0.

Algoritmo 5 - Estrutura DO das Linguagens MATLAB e Octave

```
1: x = 2;
2: do
3:       x = x + 2;
4: until (x > 0)
```

Um outro statement que vale a pena ser mencionado é o statement utilizado para tratamento de exceção TRY. Esse statement tenta executar um bloco de código. Se, por algum motivo, acontecer algum erro dentro desse bloco, o interpretador pula a execução para a primeira linha do bloco de código catch. No exemplo a seguir o interpretador tentará executar a sentença x = x / 0. Essa tentativa ocasionará em um erro de execução, pois não é possível dividir um número por zero. Nesse momento, o interpretador não executará a sentença x = x + 3. Ao invés disso, ele executará a chamada da função cleanup que corresponde à primeira linha do bloco catch.

Algoritmo 6 - Estrutura TRY das Linguagens MATLAB e Octave

```
1: x = 2;
2: try
3:       x = x=0;
4:       x = x + 3;
5: catch err
6:       cleanup
7: end_try_catch
```

Um recurso disponível nas linguagens MATLAB e Octave e que não pode deixar de ser comentado é a possibilidade de criar funções dentro dos arquivos de código fonte. Essas funções correspondem à blocos de código autocontidos que podem ser invocados dentro do próprio arquivo em que ele está inserido ou a partir de outros arquivos MATLAB/Octave. Essas funções possuem um nome e podem possuir algum retorno e uma lista de parâmetros. Mas também é possível criar funções sem retorno e sem nenhum tipo de parâmetro. No exemplo abaixo, foi desenvolvida uma função chamada soma, que recebe como parâmetro dois valores e possui um retorno que corresponde à soma dos valores passados como parâmetro. A sentença soma(3,4) é um exemplo de como essa função pode ser invocada. Nesse caso, a função soma retornará o valor 7 para quem a invocou.

Algoritmo 7 - Estrutura de uma Função nas Linguagens MATLAB e Octave

```
1: function result = soma(a,b)
2:       result = a + b;
3: endfunction
```

Existem muitos outros recursos disponíveis nessas linguagens, mas aqui foram comentados apenas os principais deles restando apenas um que será explicado agora. Atualmente as duas linguagens permitem o desenvolvimento seguindo o paradigma de orientação a objetos. Isso significa dizer que é possível criar classes com atributos e métodos nas linguagens MATLAB e Octave.

Algoritmo 8 - Estrutura de uma Classe das Linguagens MATLAB e Octave

```
 1: classdef test
 2:     properties (Hidden, SetAccess = protected)
 3:         hfoo = [];
 4:     end
 5:     methods
 6:         function self = method_function_handle_test ()
 7:             self.hfoo = @foo;
 8:         end
 9:         function bar (self)
10:             self.hfoo (self);
11:         end
12:     end
13: end
```

Para definir uma classe, basta inserir no arquivo a palavra reservada classdef seguida pelo nome da classe. No final do arquivo, é preciso utilizar o finalizador de bloco end. Tudo que estiver dentro desse bloco faz parte da classe criada. Dentro da classe é possível criar mais dois blocos: um corresponde ao bloco de atributos e o outro corresponde ao bloco das funções.

O bloco de atributos é criado a partir da palavra reservada properties e concluído com o finalizador de bloco end. Na definição desse bloco é permitido passar parâmetros que correspondem às características que os atributos podem ter, como, por exemplo seu modificador de acesso.

O bloco de funções é criado a partir da palavra reservada methods e finalizado através do finalizador de bloco end. O código abaixo é um exemplo de definição de classe com seus atributos e métodos, sendo que os atributos criados são ocultos e possuem o modificador de acesso protected.

2.2 KNOWLEDGE DISCOVERY METAMODEL (KDM)

O Knowledge Discovery Metamodel (KDM) é um metamodelo proposto pela OMG que tem como principal objetivo representar um software com todos os seus elementos, relações entre cada elemento e seu ambiente operacional. Ele faz parte de um conjunto de metamodelos

relacionados à arquitetura orientada à modernização (ADM). O KDM foi criado para ser utilizado em reengenharia de software permitindo que informações de um determinado sistema possam ser trocadas entre ferramentas de diferentes fornecedores.

Uma característica comum deste tipo de ferramenta é que elas analisam os artefatos de um software (por exemplo, código fonte, descrição de banco de dados, script de build, etc) para obter conhecimento (OMG, 2016). Esse metamodelo tem um papel de grande importância na evolução de sistemas legados, pois ele permite o compartilhamento de informações de um software durante seu processo de modernização (PÉREZ-CASTILLO; GUZMAN; PIATTINI, 2011).

Nem todas as operações permitidas pelas linguagens de programação são contempladas pelo KDM, embora ele englobe a maior parte delas. Entretanto é possível propor extensões a este metamodelo. A exemplo é possível citar uma extensão proposta para permitir a inclusão de orientação a aspecto da ferramenta AspectJ ao KDM (SANTOS et al., 2014).

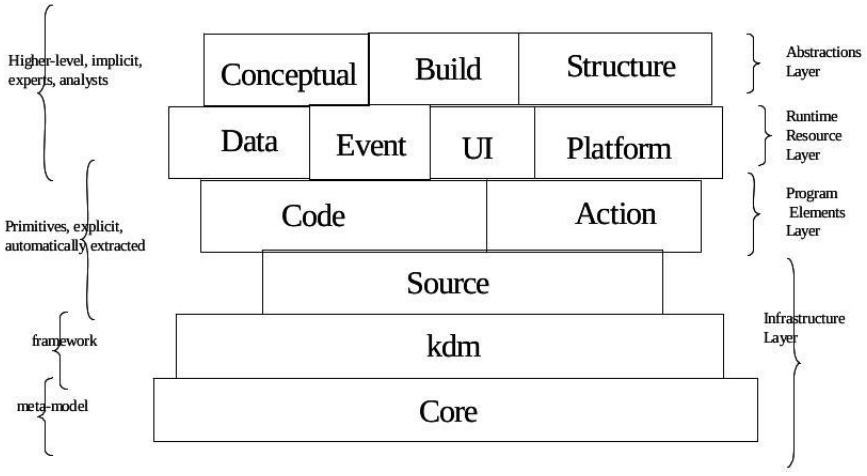
Embora sua principal função esteja relacionada à modernização de sistemas legados através da reengenharia de software, é perfeitamente aceitável que este metamodelo possa ser utilizado para diversos fins relacionados à descoberta de conhecimento, visto que ele é capaz de conter praticamente todas as informações sobre um determinado software. Inclusive já existe na literatura relato sobre o uso deste metamodelo para extração e cálculo de métricas a partir de códigos fonte de aplicações (CANOVAS; MOLINA, 2010).

O KDM determina o XML Metadata Interchange (XMI) como sendo o formato de arquivo a ser utilizado para o compartilhamento de informações de um software. Esse formato de troca de informações é chamado de schema XMI do KDM.

No metamodelo KDM são definidos doze pacotes com um ou mais diagramas de classes que descrevem conjuntos de elementos que podem ser utilizados para representar alguma parte de um software. Cada um desses pacotes possui um namespace URIs para identificá-lo dentro do arquivo XMI. A figura 4 representa a estrutura de pacotes do KDM. Essa estrutura de pacotes está disponível on-line no site do metamodelo.

Os pacotes estão definidos em camadas uma sobre a outra conforme consta na figura 4. Os pacotes definidos nas camadas superiores dependem direta ou indiretamente dos pacotes definidos nas camadas inferiores. Sendo assim, todos os pacotes dependem do Core. A tabela 4 identifica e define cada um dos doze pacotes estabelecidos no metamodelo.

Figura 4 - Estrutura de Pacotes do KDM



Fonte: OMG (2016).

Tabela 4 - Descrição dos Pacotes KDM

Pacote KDM	Descrição
Core	Define os elementos básicos que podem ser usados por todos os outros pacotes, incluindo as entidades, seus relacionamentos e os tipos de dados básicos como String, Boolean e Integer. Portanto, todos os outros pacotes dependem dele, mas o pacote Core não depende de nenhum outro. Esses tipos de dados básicos são usados para representar os atributos do KDM, suas operações e seus parâmetros.
KDM	Define os principais elementos de infraestrutura que configuram os padrões para a construção dos pontos de vista arquiteturais do software no KDM. Isso significa dizer que é neste pacote que estão descritos os elementos que definem os domínios de conhecimento do KDM, como as classes InventoryModel ou CodeModel. Os elementos descritos neste pacote, juntamente com os elementos descritos em Core, compreendem o framework KDM.
Source	Define os elementos que são usados para

	representar os artefatos físicos de um software como imagens, códigos fonte, arquivos de configuração, entre outros. Isso significa dizer que se o software tiver dez arquivos compondo sua estrutura, cada um desses arquivos será representado por um elemento descrito no pacote Source.
Code	Possui um amplo conjunto de elementos utilizados para representar os componentes de implementação do software. Cada instância de um elemento Code corresponde a um elemento do artefato de código fonte do software.
Action	Define um conjunto de elementos que servem para representar determinados comportamentos das linguagens de programação como declarações, operadores e condicionais.
Platform	Utiliza seus elementos para representar os ambientes operacionais em que os softwares são executados. Estes são os fatores que determinam o contexto no qual a execução do programa ocorreu.
UI	Representa informações relacionadas à interface de usuário do software.
Event	Representa em alto nível o comportamento de um software, que corresponde aos possíveis estados da aplicação e as transições entre esses estados.
Data	Representa a organização de dados dentro de um software. Estes dados podem ser armazenados, por exemplo, em bases de dados relacionais, arquivos de log e esquemas XML.
Structure	Representa os elementos arquiteturais de um

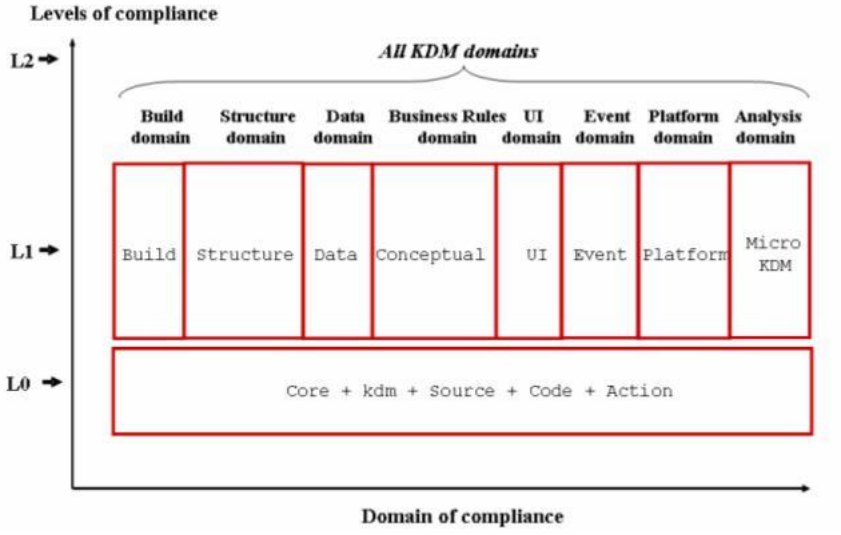
	software como subsistemas, camadas, pacotes etc. Através dos elementos definidos no pacote Structure é possível rastrear esses elementos para outros elementos dos pacotes KDM.
Conceptual	Define elementos usados na fase de análise da descoberta de conhecimento para criar um modelo conceitual do software.
Build	Representa os fatos que envolvem a fase de Build de um software.

O KDM segue o princípio de separação de interesses para permitir que apenas partes do metamodelo possam ser utilizadas de acordo com o interesse da ferramenta que deseja fazer uso do KDM. A separação desses interesses na concepção do KDM está incorporada ao conceito de domínios KDM (CANOVAS; MOLINA, 2010) (PÉREZ-CASTILLO; GUZMAN; PIATTINI, 2011) (OMG, 2016).

Cada domínio do KDM define um ponto de vista arquitetural sobre o software. Cada pacote estabelece um conjunto de elementos que podem ser utilizados para representar partes de um software. Os elementos contidos em cada um desses pacotes, ou ainda em mais de um pacote, definem o ponto de vista para um domínio específico. Por exemplo, o pacote code e action definem os elementos que caracterizam o domínio code. Esse domínio representa elementos de código individuais do software em estudo como variáveis, procedimentos e instruções (OMG, 2016).

Os seguintes domínios de conhecimento foram identificados como a base para a definição de compliance no KDM: Inventory, Code, Build, Structure, Data, Business Rules, UI, Event, Platform, e micro KDM. Os domínios de conhecimento são agrupados em três níveis de compliance, como é mostrado na figura 5.

Figura 5 - Domínios de Conhecimento e Níveis de Compliance do KDM



Fonte: OMG (2016).

O nível 0 ou L0, como é chamado, aborda os domínios de inventory e code. O L0 é determinado pelos pacotes Core, kdm, Source, Code e Action.

O nível L1 aborda todos os domínios restantes do KDM e estende as capacidades providas pelo nível L0. Esse nível é determinado pelos pacotes Build, Structure, Data, Conceptual, UI, Event, Platform, bem como o conjunto de restrições para o Micro KDM

O nível L2 é a união do nível L1 para todos os domínios do KDM.

Antes de utilizar o metamodelo KDM, é de fundamental importância identificar os domínios que se deseja alcançar com essa ferramenta para não perder tempo implementando classes que não serão úteis.

2.3 COMPREENSÃO DE PROGRAMAS

A compreensão de programas é o processo de entender o significado e o comportamento de um software. É uma atividade do desenvolvimento de software de extrema importância (STOREY, 2005). Embora a compreensão de programas não seja uma fase independente, é um componente de nove diferentes fases de desenvolvimento de software. A atividade de compreender o software é fundamental para as modificações do código e para a análise de impacto das mudanças (HEVNER et al., 2005).

Compreender um software consiste em obter o conhecimento necessário sobre um programa para realizar com êxito uma determinada atividade (BIGGERSTAFF; MITBANDER; WEBSTER, 1994).

Os modernos ambientes de desenvolvimento integrado (IDEs) contêm ferramentas de compreensão de programas que facilitam o trabalho do programador. No entanto, o programador ainda gasta um tempo considerável realizando manualmente a compreensão do programa. A compreensão manual do programa envolve a construção de um modelo mental do comportamento do programa (KOSAR; MERNIK; CARVER, 2012).

Ao construir este modelo, além de ter que conhecer conceitos específicos do domínio do problema, os programadores também têm que processar e entender sua implementação usando conceitos de linguagem de programação (KOSAR; MERNIK; CARVER, 2012).

Um estudo realizado em uma empresa de software mostrou que mais de 28% do tempo dos desenvolvedores é dedicado a aprender e entender seu código ou o código de seus colegas (HEVNER et al., 2005).

Esta fração de tempo é uma média aproximada do esforço de compreensão do programa exigido em várias atividades de desenvolvimento, incluindo especificação do sistema, arquitetura do sistema, design de componentes, avaliação / seleção de componentes, implementação de componentes, avaliação de correção de componentes, integração de sistemas e evolução (HEVNER et al., 2005).

A fase de manutenção e evolução apresentou um dos maiores percentuais de esforço dos programadores na compreensão do comportamento do sistema (40%) (HEVNER et al., 2005).

O conhecimento dos conceitos arquiteturais de grandes softwares é a chave para a compreensão dos sistemas legados e para o desenvolvimento de novos softwares. Esses conceitos incluem a arquitetura em camadas, agregações, especializações, generalizações, o conceito de herança, estratégias de manipulação de eventos, separação das interfaces de usuário, arquiteturas no modelo cliente servidor, níveis de acoplamento, entre outros (MÜLLER; TILLEY; WONG, 1993).

Uma das abordagens mais promissoras para o problema da evolução do software é a tecnologia de compreensão de programas. Estima-se que cinquenta a noventa por cento do trabalho de evolução é dedicado à compreensão de programas. Um caminho para melhorar a compreensão de programas é através da reengenharia de software (MÜLLER; TILLEY; WONG, 1993).

Os relatos na literatura têm mostrado a importância da compreensão de programas na evolução do software, como a engenharia reversa pode ajudar esse processo de entender os

códigos fontes das aplicações e como é importante utilizar essas técnicas ou ferramentas para apoiar a compreensão de programas, de modo a minimizar o tempo gasto nesse processo.

Uma das ferramentas mais importantes que vêm sendo usadas no processo de análise e compreensão de programas é a visualização de software (STOREY; MULLER, 1995). Essas ferramentas podem economizar dinheiro e tempo à medida que auxiliam os desenvolvedores no entendimento do software, aumentando, assim, sua produtividade (KOSCHKE, 2002).

2.4 VISUALIZAÇÃO DE SOFTWARE

A visualização é um importante recurso para a atividade de compreensão. Através dela é possível construir um modelo mental ou uma imagem mental a respeito de uma determinada situação ou realidade (SPENCER, 2007). A principal função da visualização consiste em aprimorar a percepção e o raciocínio visual do ser humano de modo que ele possa, através de técnicas ou ferramentas, entender eventos que não são facilmente compreendidos (BABAR; CHEN; SHULL, 2010).

Visualização da informação é uma área da computação que utiliza técnicas de computação gráfica para auxiliar o processo de análise e compreensão de um conjunto de dados (FREITAS et al., 2001).

Quando esse conjunto de dados é oriundo de um software, seja através do seu código fonte ou de qualquer outro artefato relacionado ao programa, essa importante atividade de compreensão recebe o nome de visualização de software.

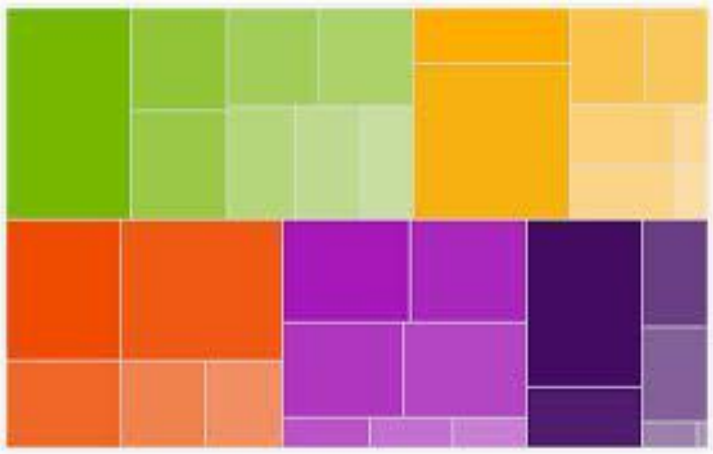
O processo de visualização de software consiste em extrair os dados de uma aplicação, processá-los e mapeá-los em uma representação visual que apoie o processo de compreensão do software. Ela se utiliza de atributos visuais como tamanho e cor para representar características do software com o intuito de tornar visíveis padrões e comportamentos que estejam imperceptíveis (PETRE, 2010).

O primeiro passo para utilizar a visualização de software de maneira eficiente é identificar qual ou quais informações podem ser extraídas da aplicação. Em seguida é preciso escolher os atributos que podem ser utilizados para exibir essas informações. Uma vez selecionados, é preciso escolher qual a metáfora visual que melhor se adequa ao contexto para mapear os atributos do software em atributos visuais.

Como exemplo, será exibida uma representação visual conhecida como treemap. Ela é utilizada para demonstrar uma visão hierárquica de um conjunto de dados. Na figura 6 é

possível identificar três atributos visuais que podem ser utilizados para representar alguma característica de um software: cor, intensidade da cor e tamanho. Imaginando o código fonte de uma aplicação desenvolvida na linguagem Java, é possível utilizar essa representação visual para exibir todas as classes do projeto. Nesse caso, seria possível fazer o seguinte mapeamento: cada retângulo do treemap seria uma classe do projeto.

Figura 6 - Representação Visual Treemap



A cor utilizada para pintar o retângulo poderia estar diretamente relacionada ao pacote que a classe pertence. Isso significa dizer que todas as classes do mesmo pacote seriam pintadas da mesma cor. O tamanho do retângulo poderia ser mapeado para representar o tamanho da classe em número de linhas. Quanto mais linhas de código à classe possuir, maior será o tamanho do retângulo que a representa. E, por fim, a intensidade da cor poderia ser mapeada para representar a complexidade ciclomática da classe. Quanto mais complexa for a classe, mais escuro será o retângulo e quanto mais simples, mais claro será o retângulo.

Existem dois passos dessa atividade que são extremamente complexos. Extrair os dados para montar a visão desejada e mapear os atributos. Essa complexidade existe devido ao grande número de arquivos existentes dentro de um software e à quantidade de informações existentes dentro de cada arquivo.

Para apoiar esse processo se faz necessário o uso de ferramentas de visualização de software. Existe um conjunto de ferramentas que podem ser utilizadas para visualização de software. Normalmente essas ferramentas disponibilizam mais que uma representação visual.

O NodeXL é um exemplo deste tipo de ferramenta. Ela é capaz de exibir visualmente dados de redes armazenados numa planilha do Microsoft Excel. O Action Science Explorer é uma ferramenta utilizada para representação bibliográfica de artigos científicos. O Noodles é

uma ferramenta projetada para visualizar de forma interativa um conjunto de dados de eventos climáticos. O SourceMiner é uma ferramenta capaz de apoiar a compreensão de programas desenvolvidos na linguagem Java através de metáforas visuais. O SeeGH é uma ferramenta para visualização de dados genoma. Essas são apenas algumas ferramentas relatadas na literatura. Existem muitas outras.

Além de disponibilizar as metáforas visuais, uma ferramenta de visualização deve permitir a inclusão de filtros na busca das informações. Essa funcionalidade é muito importante visto que, muitas vezes, o volume de dados que precisam ser visualizados é muito grande e, se não houver, uma forma de filtrar as informações de modo a reduzir esse volume, pode se tornar inviável visualizar todos esses dados.

Uma outra funcionalidade também importante para melhorar a visualização de um grande volume de dados é a possibilidade de combinar mais de uma metáfora visual para a mesma visualização.

Um outro requisito importante para esse tipo de ferramenta é o tempo de resposta na obtenção dos resultados que deverão ser visualizados, visto que é muito comum utilizar as ferramentas de visualização em cima de um volume de dados que pode chegar na casa de milhares de informações em uma única visualização.

Nos tempos atuais em que as pessoas estão cada vez mais acostumadas a terem todas as informações literalmente na palma da mão, através do uso de smartphones e tablets, é desejável que as ferramentas de visualização também sejam compatíveis com esses dispositivos.

2.5 TRABALHOS RELACIONADOS

A visualização da Informação já vem sendo utilizada há algum tempo para analisar a evolução de softwares. Já existem muitos trabalhos em diferentes linhas de pesquisa que utilizam essa técnica com objetivo de apoiar a compreensão de códigos-fonte. Existem trabalhos cujo o principal objetivo é propor uma nova metáfora visual que possa ser utilizada na análise de códigos-fonte, como é o caso da TimeLine Matrix, uma visão que permite analisar a evolução de elementos de um determinado software sob demanda (NOVAIS; JUNIOR; MENDONÇA, 2012). Também é possível encontrar muitos trabalhos que têm como principal objetivo propor ferramentas que utilizam a visualização da informação para apoiar o processo de compreensão de programas.

O SourceMiner, por exemplo, é uma ferramenta de visualização desenvolvida como um plugin do Eclipse. Seu principal objetivo é fornecer um conjunto de metáforas visuais que podem ser utilizadas separadamente, ou em conjunto, para analisar códigos-fonte desenvolvidos na linguagem Java. Através dessa ferramenta é possível fazer, entre outras coisas, uma análise do grau de acoplamento e de modularidade do software analisado. (CARNEIRO, 2013).

Apesar de existirem muitos trabalhos na área de visualização de software, foi encontrado apenas um que propõe uma ferramenta para apoiar o processo de compreensão de programas MATLAB e Octave. O OctMiner é um plugin do Eclipse que se propõe a ser um ambiente interativo de múltiplas visões para análises de códigos fonte MATLAB e Octave (LESSA et al., 2015). O grande problema desta ferramenta é a dependência da IDE Eclipse. Outra possível limitação da ferramenta é que ela não possibilita exportar as estruturas de dados visuais dos programas analisados para outras plataformas, fato que pode ser feito com a solução proposta nesta dissertação por intermédio do formato JSON.

Não foi encontrado nenhum relato na literatura que demonstrasse a utilização do KDM, como um metamodelo de apoio no processo de análise e compreensão de códigos-fonte de programas. Também não foi encontrado nenhum relato de trabalhos que utilizem o KDM para extrair informações que possam ser utilizadas para carregar metáforas visuais.

2.6 CONCLUSÃO DO CAPÍTULO

Este capítulo apresentou os principais conceitos utilizados nesta dissertação, fazendo uma breve narrativa sobre as linguagens de programação MATLAB e Octave, sobre o metamodelo KDM, sobre a compreensão de programas e a visualização de software.

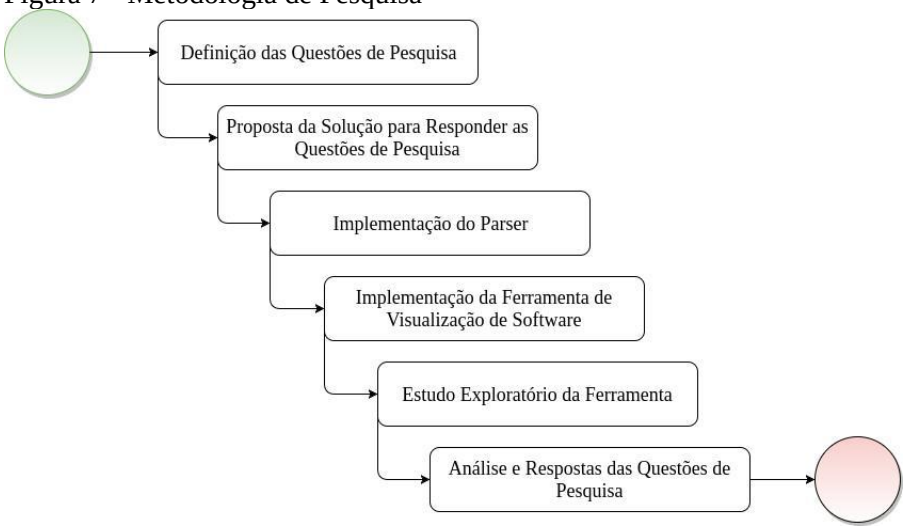
3 METODOLOGIA DE PESQUISA

Este capítulo descreve a metodologia de pesquisa utilizada nesta dissertação. As seções que se seguem apresentam como a pesquisa foi conduzida para atingir os objetivos propostos.

3.1 ETAPAS ADOTADAS NESTA DISSERTAÇÃO

A figura 7 ilustra as etapas que constituem a metodologia adotada nesta dissertação, que corresponde a um estudo exploratório. Nesse tipo de metodologia é preciso inicialmente levantar as questões de pesquisa baseadas em um problema que precisa ser resolvido. Em seguida, é imprescindível fazer um planejamento das ações necessárias à obtenção de respostas para as questões elaboradas de pesquisa. Depois, é preciso executar essas ações e analisar se houve efetividade em suas resoluções.

Figura 7 - Metodologia de Pesquisa



A primeira etapa desta dissertação de mestrado consistiu em elaborar as questões de pesquisa que serviram para nortear a execução deste trabalho. Elas foram elaboradas baseadas no problema que foi abordado. Na etapa seguinte, foi apresentada uma proposta que tem por objetivo tentar responder as questões de pesquisa. Baseado na proposta apresentada nessa segunda etapa, foi identificada a necessidade de criar um parser e um ambiente de visualização que foram descritos respectivamente na terceira e quarta etapa deste projeto. A quinta etapa mostrou como as ferramentas desenvolvidas neste trabalho podem ser utilizadas

para responder as questões de pesquisa. A sexta e última etapa consistiu na análise e resposta das questões de pesquisa.

As próximas seções deste capítulo descreverão o detalhamento de cada uma dessas etapas.

3.2 DEFINIÇÃO DAS QUESTÕES DE PESQUISA

Em um estudo exploratório é extremamente importante definir as questões de pesquisa. Além de direcionarem a execução do trabalho, elas servirão posteriormente para a análise dos resultados obtidos após a execução.

O principal objetivo deste trabalho foi propor um ambiente baseado em KDM que permita utilizar a visualização de software para apoiar a compreensão de programas desenvolvidos nas linguagens MATLAB e Octave e avaliar sua efetividade. Esse objetivo ajudou na definição das seguintes questões de pesquisa desta dissertação:

- QPD1. É viável a representação visual de programas MATLAB/Octave usando o Knowledge Discovery Metamodel (KDM), sem necessidade de criar pontos de extensão ao metamodelo?

- QPD2. Qual a efetividade do uso de um ambiente de visualização de programas MATLAB/Octave baseado no Knowledge Discovery Metamodel (KDM) para apoiar atividades de compreensão de programas?

- QPD3. Até que ponto um ambiente de visualização de programas MATLAB/Octave baseado no Knowledge Discovery Metamodel (KDM) pode ser extensível para viabilizar a inclusão de novas metáforas visuais?

- QPD4. Até que ponto um ambiente de visualização de programas MATLAB/Octave baseado no Knowledge Discovery Metamodel (KDM) pode ser flexível para importar estruturas de dados no formato KDM de outras linguagens de programação?

- QPD5. Até que ponto um ambiente de visualização de programas MATLAB/Octave baseado no Knowledge Discovery Metamodel (KDM) pode ser flexível o bastante para

exportar a estrutura de dados KDM no formato Json para ser utilizada por outros ambientes de apoio à compreensão de programas?

A partir dessas questões foi possível fazer um planejamento do que seria necessário para desenvolver um ambiente de visualização baseado em KDM que apoie o processo de compreensão de programas MATLAB e Octave.

3.3 PROPOSTA DA SOLUÇÃO PARA RESPONDER AS QUESTÕES DE PESQUISA

Nesta etapa foi definida a arquitetura necessária para a construção do ambiente de visualização que atenda ao objetivo especificado sem deixar de responder as questões elaboradas de pesquisa.

Foi constatado que, para utilizar o KDM, seria necessário extrair os dados dos códigos fonte MATLAB e Octave a partir de suas ASTs e que esses dados deveriam ser armazenados em uma estrutura que representasse o metamodelo KDM de aplicações MATLAB e Octave.

Após algumas análises sobre o que existe disponível no mercado que pudesse ser utilizado nesse projeto, foi decidido desenvolver uma ferramenta capaz de ler as linhas de código de arquivos MATLAB e Octave convertendo-as em suas respectivas ASTs. Em seguida foi elaborada uma nova ferramenta capaz de percorrer as ASTs criadas e converter os dados contidos nos nós das ASTs no padrão descrito pelo KDM.

Também nesta etapa foi tomada uma decisão de que para obter uma boa performance nas consultas das informações seria interessante que os dados dos códigos fonte fossem armazenados em um banco de dados NoSQL.

3.4 IMPLEMENTAÇÃO DO PARSER

Para implementar a ferramenta capaz de ler os dados armazenados nos códigos fonte MATLAB e Octave foi necessário, inicialmente, entender todos os recursos disponíveis nessas linguagens, e mapeá-los nos elementos disponíveis no metamodelo KDM. Dessa forma seria possível identificar se o KDM consegue atender plenamente essas linguagens sem a necessidade de adaptação. Além disso, nesse momento foi possível descobrir que o nível de compliance do KDM a ser utilizado seria o nível L0 e que os domínios utilizados seriam o code e o inventory. Portanto só seria necessário utilizar os elementos descritos nos pacotes core, model, code, source e inventory.

O KDM atendeu perfeitamente às necessidades de representação de aplicações desenvolvidas nessas linguagens sem necessidade de adaptação e as ferramentas foram desenvolvidas como bibliotecas na linguagem de programação Java.

3.5 IMPLEMENTAÇÃO DO AMBIENTE DE VISUALIZAÇÃO

Nessa etapa, foi desenvolvida uma ferramenta de visualização web utilizando a arquitetura proposta baseada em serviços fazendo uso da tecnologia Representational State Transfer (REST).

A ferramenta desenvolvida disponibiliza duas representações visuais que podem interagir entre si. Para visualizar grandes volumes de informações, a ferramenta disponibiliza também o uso de filtros.

Ela foi desenvolvida utilizando na camada de visão páginas escritas em HTML5 com CSS3 e JavaScript. Para montar as visões, foi utilizado uma biblioteca JavaScript chamada D3js.

O modelo cliente/servidor foi utilizado no desenvolvimento da ferramenta, em que as páginas HTML representam o lado cliente. O cliente se comunica com o servidor através de chamadas REST. O lado servidor foi desenvolvido na linguagem de programação Java.

3.6 ESTUDO EXPLORATÓRIO

Para utilizar o ambiente de visualização desenvolvido, foram selecionados alguns projetos MATLAB/Octave disponíveis em um repositório público. Algumas informações que se pretendia descobrir foram estabelecidas a partir desses projetos e, em seguida, o ambiente foi utilizado na tentativa de exibir essas informações.

O que se pretendia descobrir nos projetos era o nível de acoplamento entre os arquivos e a concentração de muitas funcionalidades em um único arquivo/função.

Para buscar esse tipo de informação foi necessário utilizar uma visão de relacionamentos e uma outra que pudesse representar hierarquicamente a concentração de volume de dados. As visões utilizadas foram chord e treemap.

O volume de dados nos projetos selecionados chegou perto da casa de milhões de registros no banco de dados, visto que tiveram projetos com mais de 4000 arquivos de código fonte.

3.7 ANÁLISE E RESPOSTA DAS QUESTÕES DE PESQUISA

As questões de pesquisa da dissertação QPD1 e QPD2 foram respondidas com sucesso através da utilização do ambiente de visualização desenvolvido nos projetos selecionados.

É preciso ressaltar que o KDM foi utilizado conforme proposto pela OMG, sem pontos de extensão.

Para efeito de teste, o ambiente de visualização foi utilizado em computadores e tablets. O tempo de resposta para milhares de informações foi na casa de segundos.

As questões QPD3, QPD4 e QPD5 foram respondidas através da arquitetura adotada no desenvolvimento do ambiente de visualização.

3.8 CONCLUSÃO DO CAPÍTULO

Este capítulo descreveu a metodologia adotada na condução desta dissertação de mestrado e todas as etapas percorridas para atingir os objetivos propostos.

4 UMA PROPOSTA DE UM PARSER DE APLICAÇÕES MATLAB E OCTAVE

Este capítulo abordará os passos necessários para o desenvolvimento do parser responsável por gerar o KDM de projetos MATLAB/Octave e descreverá com detalhes as implementações necessárias.

4.1 PASSOS NECESSÁRIOS AO DESENVOLVIMENTO DO PARSER

Antes de implementar o parser responsável por gerar o KDM de projetos MATLAB/Octave, foi necessário entender a gramática das linguagens de modo a conhecer todas as funcionalidades disponíveis e entender o funcionamento do KDM com seus respectivos pacotes e classes. Esse processo foi necessário para saber se o KDM é capaz de representar todas as funcionalidades das linguagens MATLAB e Octave sem a necessidade de fazer algum tipo de adaptação. Além disso, o estudo sobre essas tecnologias foi o que permitiu fazer o mapeamento de cada funcionalidade das linguagens em um elemento do KDM. A figura 4 exibe os pacotes disponíveis no metamodelo KDM.

Durante a fase de análise das linguagens, foram identificadas todas as operações relevantes, como elas funcionam e quais os operadores que podem ser utilizados. Isso inclui todos os operadores aritméticos, operadores relacionais, expressões booleanas, chamadas de função, expressões de atribuição, operadores de incremento e decremento e definições de função.

Também foram abordadas as várias estruturas de controle das linguagens (estrutura condicional, estrutura de repetição, estrutura de switch etc.) e instruções tais como continue, break, unwind protect, try, entre outras. Para identificar tais elementos foi utilizada a documentação da versão 4.2.0 do GNU Octave disponível através da url: <www.gnu.org/software/octave/doc/v4.2.0/index.html#SEC_Contents>.

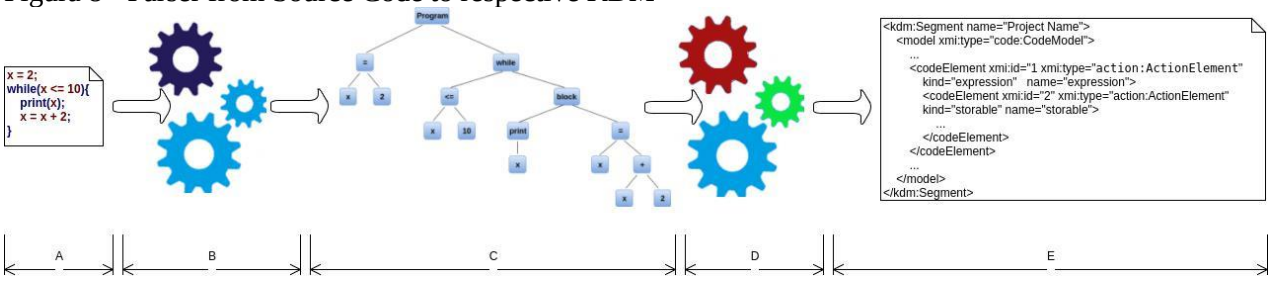
4.2 SOLUÇÃO ADOTADA

Conforme pode ser visto na figura 8, para gerar o KDM das aplicações MATLAB/Octave, o parser desenvolvido foi dividido em duas partes. Na primeira parte foi elaborado um parser intermediário capaz de ler os dados armazenados nos arquivos de código-fonte de um determinado projeto e gerar sua AST correspondente. Na segunda parte

foi desenvolvido um parser capaz de ler os objetos contidos nos nós da AST gerada e criar uma estrutura de dados contendo as informações das aplicações seguindo o KDM.

A implementação do parser foi concebida como um conjunto de cinco bibliotecas desenvolvidas na linguagem Java. Nesta seção serão descritos os processos de desenvolvimento das bibliotecas ASTOctave, OctaveKDMStructure, ASTOctaveToKDM, OctaveKDMToXMI, OctaveKDMToJSON e as principais características de suas implementações. A figura 9 mostra como ocorre a interação entre essas cinco bibliotecas.

Figura 8 - Parser from Source Code to respective KDM



A biblioteca ASTOctave inclui um parser capaz de ler o conteúdo de um arquivo de código fonte MATLAB ou Octave e gerar a AST correspondente. A biblioteca OctaveKDMStructure representa a estrutura KDM de aplicações MATLAB ou Octave. A biblioteca ASTOctaveToKDM é responsável por varrer todos os arquivos de código fonte de um programa MATLAB ou Octave, gerar a AST de cada um desses arquivos usando a biblioteca ASTOctave, converter os objetos contidos nos nós das ASTs criadas em elementos da estrutura KDM e gerar os arquivos Json e XMI dessa estrutura fazendo uso respectivamente das bibliotecas OctaveKDMToJSON e OctaveKDMToXMI.

Embora a documentação do KDM proponha que a representação da estrutura seja descrita em um arquivo XMI, foi decidido também criar uma representação idêntica no formato de arquivo JSON. Esta decisão foi tomada para permitir a inclusão destes dados no banco de dados NoSQL. Esse processo facilitará a extração de informações dos dados armazenados para possível análise. Esta foi a motivação para o desenvolvimento da biblioteca OctaveKDMToJSON.

A biblioteca ASTOctaveToKDM é a grande mediadora do conjunto de bibliotecas propostas. Ela é responsável por instanciar o KDM correspondente ao projeto que terá sua estrutura criada. Também é de sua responsabilidade controlar quando e como as outras bibliotecas serão instanciadas e utilizadas. O ciclo de execução da solução proposta é iniciado e finalizado na biblioteca ASTOctaveToKDM. A figura 9 mostra o diagrama de sequência

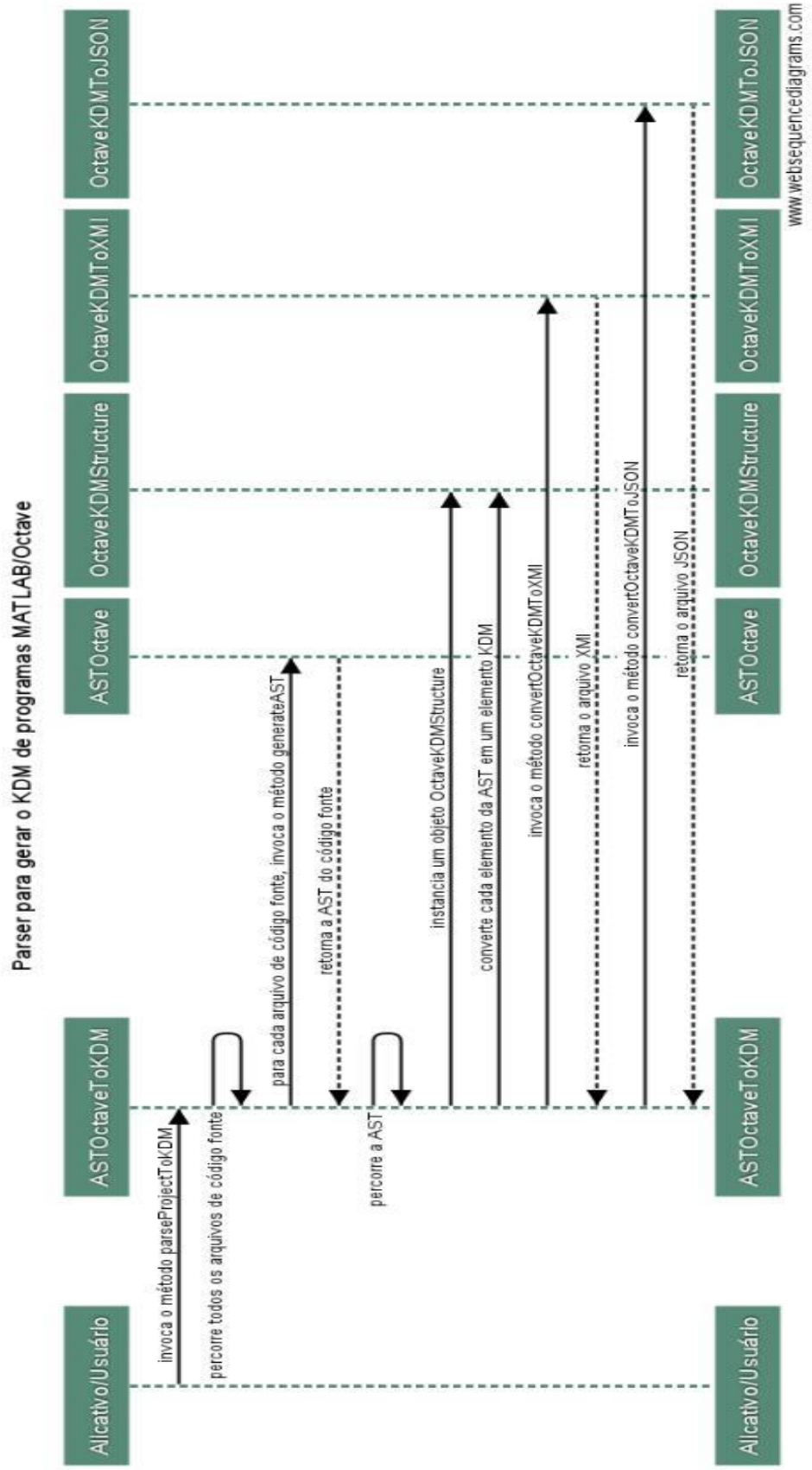
que representa a interação entre as cinco bibliotecas implementadas e a ordem cronológica em que essas bibliotecas são utilizadas na solução proposta.

A partir da figura 9, é possível constatar que para executar o parser proposto é preciso fazer uso inicialmente da biblioteca ASTOctaveToKDM. É a partir dela que o usuário ou uma outra aplicação qualquer poderá iniciar o processo de transformação, através do método parseProjectToKDM. A esse método, será passado como parâmetro o caminho do projeto que deverá ser gerado o KDM correspondente.

Depois que o processo é iniciado, a biblioteca ASTOctaveToKDM percorre todos os arquivos de código fonte do projeto e, para cada arquivo encontrado, ela faz uma chamada à biblioteca ASTOctave através do método generateAST, passando como parâmetro o próprio arquivo. A biblioteca ASTOctave ler o conteúdo do arquivo e gera a AST correspondente e retorna essa AST para a biblioteca ASTOctaveToKDM. De posse da AST, a biblioteca ASTOctaveToKDM varre todos os nós e, para cada um deles, ela converte o elemento encapsulado no nó em um elemento KDM através do uso da estrutura disponibilizada na biblioteca OctaveKDMStructure.

Quando todos os elementos contidos nos nós da AST são convertidos, a biblioteca ASTOctaveToKDM passa a instancia KDM criada, que representa o projeto, para as bibliotecas OctaveKDMToXMI e OctaveKDMToJSON que são responsáveis por ler todos os elementos KDM e gerar respectivamente os arquivos XMI e JSON correspondentes.

Figura 9 - Diagrama que Exibe a Interação entre as Cinco Bibliotecas da Solução Proposta



4.3 BIBLIOTECA ASTOCTAVETOKDM

Nesta seção será mostrado o algoritmo responsável por converter os objetos armazenados nos nós da AST em objetos que representem o metamodelo KDM do software existente. O algoritmo de conversão pode ser visualizado através da figura 10.

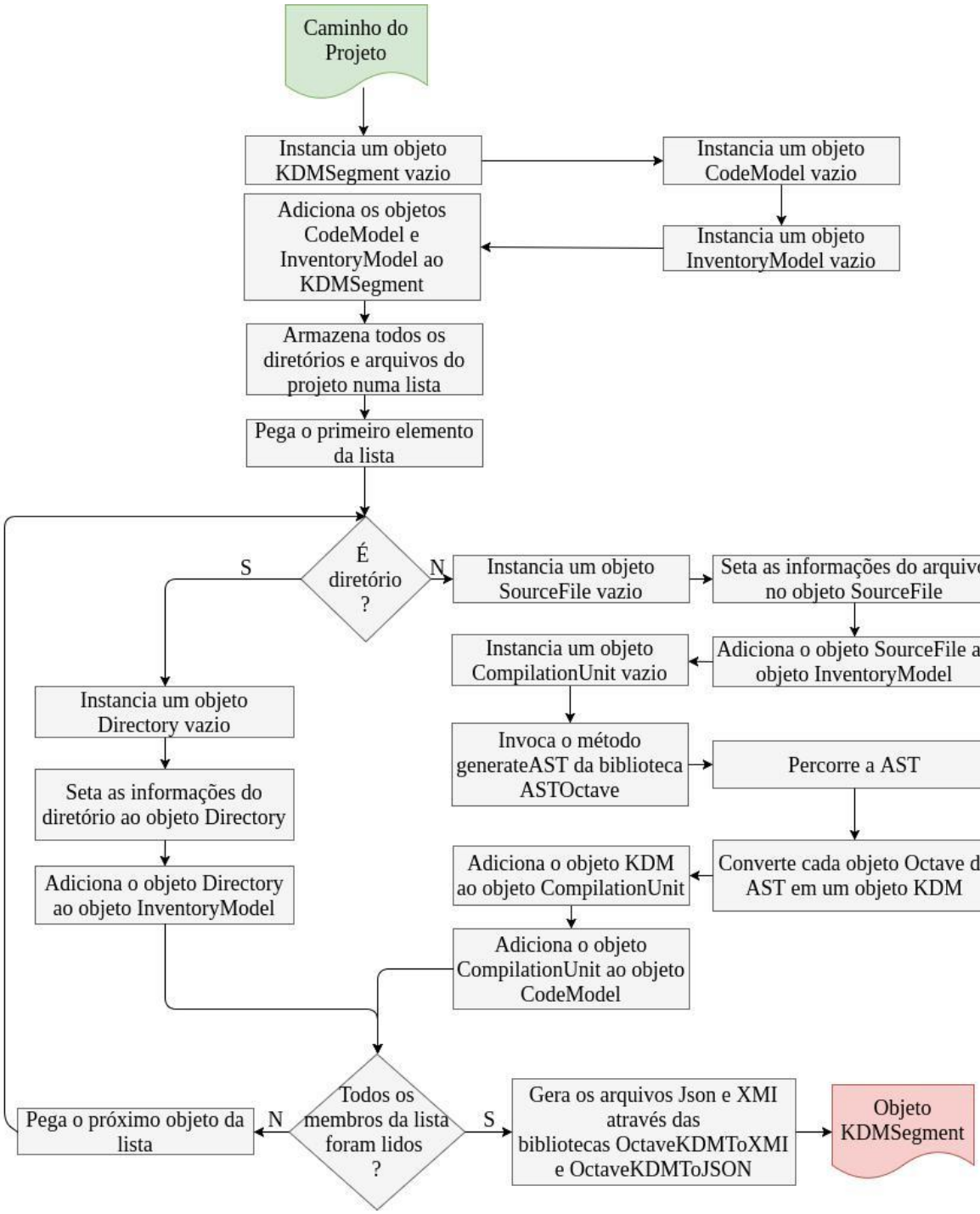
Foi disponibilizado nessa biblioteca o método público parseProjectToKDM da classe OctaveToAST. Ele recebe como parâmetro um objeto do tipo String, que corresponde ao caminho do diretório físico do projeto Octave ou Matlab e uma outra String contendo o nome do projeto que será convertido. Caso o diretório não exista ou o objeto passado como parâmetro seja um arquivo ao invés de um diretório, uma exceção é retornada. Caso contrário, um objeto da classe KDMSegment é instanciado, passando como parâmetro o nome do projeto que será carregado na estrutura. Uma lista vazia do tipo KDMModel é criada em seu construtor.

O método instancia em seguida um objeto do tipo InventoryModel setando seu atributo name com a String ‘source references’. Nesse momento, é criada, dentro desse objeto, uma lista vazia do tipo AbstractInventoryElement. Em seguida, é feita uma chamada ao método privado loadFile passando como parâmetro o diretório do projeto. Ele percorre todos os arquivos e subdiretórios da pasta do projeto. Para cada arquivo encontrado, é instanciado um objeto do tipo SourceFile contendo o nome do arquivo e o caminho físico onde ele está armazenado. Para cada pasta encontrada, é criada uma instância do objeto Directory contendo a localização física do diretório. Todos esses objetos são adicionados à lista de AbstractInventoryElement. Ao final do método, a instância de InventoryModel é adicionada à lista de KDMModel do objeto KDMSegment.

No momento em que faz a leitura dos arquivos, o método loadFile também instancia um objeto do tipo CodeModel e adiciona-o à lista de KDMModel do objeto KDMSegment. O objeto CodeModel criado possui uma lista vazia capaz de armazenar instâncias de objetos do tipo CompilationUnit e o atributo name preenchido com a String ‘code’.

Para cada arquivo de código fonte encontrado no projeto, o método loadFile, fazendo uso da biblioteca ASTOctave, cria uma AST e uma instância da classe CompilationUnit, colocando o nome do arquivo de código fonte no seu atributo name. Para cada AST gerada, feita uma chamada ao método sweepAST, passando como parâmetro o primeiro nó da árvore e a instância de CompilationUnit criada. Após a chamada, a instância de CompilationUnit é adicionada à lista de CompilationUnit do objeto CodeModel.

Figura 10 - Fluxo para Geração da Estrutura KDM



O método sweepAST percorre a lista de filhos do nó passado como parâmetro e, para cada filho, é criada uma instância do elemento KDM correspondente. A instância criada é adicionada à lista de abstractCodeElement do elemento passado como parâmetro. Em seguida, é feita uma nova chamada ao método sweepAST, passando como parâmetro o nó filho que foi lido e a instância de AbstractCodeElement criada. Dessa forma recursiva, o método é capaz de

percorrer todos os nós da AST, convertê-los no elemento KDM correspondente e armazená-lo de forma correta na estrutura criada e disponibilizada através da biblioteca Java OctaveKDMStructure.

O primeiro nó de um arquivo Octave possui o objeto Block, que é convertido em uma instância da classe BlockUnit e possui o atributo kind preenchido com a string ’principal block’. Esse objeto é adicionado à lista de abstractCodeElement do objeto CompilationUnit correspondente. Os nós filhos podem conter objetos das seguintes classes: OctaveFunction, OctaveFor, OctaveParfor, OctaveWhile, OctaveDo, OctaveIf, OctaveExpression, OctaveSwitch, OctaveGlobal, OctavePersistent, OctaveVariable, OctaveFunctionCall, OctaveTry, OctaveVariableStructure, OctaveStructure, OctaveWarning, OctaveString, OctaveConstant, OctaveMatrix, OctaveImport, OctaveUnwindProtect, OctaveArray, OctaveAnonymousFunction e OctaveHandleFunction. Para facilitar o entendimento do algoritmo, todas as classes citadas, com exceção da classe OctaveFunction, serão classificadas como OctaveUniversalElements. Foi criado um método para cada uma delas, responsável por convertê-las no elemento KDM correspondente.

O objeto OctaveFunction, que corresponde à uma função ou procedure da linguagem Octave, é convertido no objeto CallableUnit da estrutura KDM. O atributo kind é preenchido com a String ‘function’ ou ‘procedure’.

O nó que contém o objeto OctaveFunction pode ter n filhos. Eles podem conter objetos de qualquer uma das classes citadas anteriormente. No objeto OctaveFunction é possível encontrar a declaração da função ou procedure com seus argumentos, caso existam.

É, portanto, convertido no objeto Signature e adicionado ao atributo correspondente do objeto CallableUnit criado. Para cada parâmetro existente, é criado um objeto do tipo ParameterUnit, que é adicionado à lista de ParameterUnit do objeto Signature. Os objetos contidos nos nós filhos são convertidos no elemento KDM correspondente e adicionados à lista de AbstractCodeElement do objeto CallableUnit.

Os objetos OctaveFor, OctaveParfor, OctaveWhile e OctaveDo que representam as estruturas de repetição ‘for’, ‘while’ e ‘do’, respectivamente, são convertidos no objeto ActionElement da estrutura KDM. O primeiro possui a String ‘for’ no atributo kind, o segundo a String ‘parfor’, o terceiro a String ‘while’ e o quarto a String ‘do’.

Os objetos da classe OctaveFor e OctaveParfor possuem um Objeto do tipo OctaveExpression, que corresponde à identificação de uma variável e um objeto também da classe OctaveExpression que equivale ao critério de parada da estrutura de repetição. Os dois

objetos são convertidos em objetos do tipo ActionElement. O atributo kind do primeiro preenchido com a String ‘assignment’ e do segundo com a String ‘infix expression’. Os objetos das classes OctaveWhile e OctaveDo contém o atributo expression, que corresponde ao critério de parada da estrutura de repetição. Ele é convertido em um ActionElement contendo a String ‘infix expression’ no atributo kind.

É criado um objeto do tipo BlockUnit, que possui a String ‘block for’, ‘block parfor’, ‘block while’ ou ‘block do’ no atributo kind. O ActionElement que representa o critério de parada das estruturas de repetição e o objeto BlockUnit são adicionados à lista de AbstractCodeElement do ActionElement que representa o OctaveForStatement, OctaveWhileStatement ou o OctaveDoStatement. No caso do OctaveForStatement, o ActionElement que representa a identificação da variável também é adicionado à sua lista de AbstractCodeElement. Os demais filhos podem conter qualquer um dos objetos classificados como OctaveUniversalElements. Eles também são convertidos em objetos do tipo ActionElement e adicionados à lista de AbstractCodeElement do objeto BlockUnit.

O objeto OctaveIf que representa o teste condicional da linguagem Octave é convertido no objeto ActionElement da estrutura KDM. A String ‘if’ é armazenada no seu atributo kind.

O nó da AST que possui o objeto OctaveIf pode ter n filhos com objetos de qualquer uma das doze classes classificadas como OctaveUniversalElements. O objeto da classe OctaveIf contém um atributo que é uma instância da classe OctaveBinaryExpression, que corresponde ao critério do teste. Ele é convertido no objeto ActionElement da estrutura do KDM e a String ‘infix expression’ é armazenada no seu atributo kind. O elemento convertido é adicionado à lista de AbstractCodeElement do elemento que representa o objeto OctaveIf.

Além das classes citadas, também é possível encontrar nós filhos que possuam objetos das classes OctaveElseIf e OctaveElse. Esses objetos são convertidos em objeto ActionElement da estrutura KDM com a String ‘elseif’ e ‘else’ respectivamente. Todos os nós filhos encontrados e convertidos são adicionados à lista de AbstractCodeElement do ActionElement que representa o objeto OctaveIfStatement. Os nós que contém objetos da classe OctaveElseIf e OctaveElse também podem conter nós filhos com objetos de qualquer uma das doze classes classificadas como OctaveUniversalElements. Esses objetos são convertidos no elemento KDM correspondente e adicionados às suas listas de AbstractCodeElement.

O objeto OctaveExpression utilizado para representar uma expressão da linguagem Octave pode ser formado pela junção de várias expressões. Esse objeto é convertido no objeto ActionElement da estrutura KDM e possui a String ‘expression statement’ no atributo kind. Caso ele seja formado por outras expressões, cada uma delas também é convertida em objetos do tipo ActionElement e adicionadas à sua lista de AbstractCodeElement e assim por diante. Se na expressão houver uma atribuição de valor a alguma variável, cria-se um objeto do tipo StorableUnit que possui a String ‘local’ no atributo kind. Esse objeto também é adicionado ‘a sua lista de AbstractCodeElement.

O objeto OctaveGlobal utilizado para representar o uso de variáveis globais na linguagem Octave e o objeto OctavePersistent utilizado para representar variáveis que não possuem seu valor alterado entre as chamadas de funções são convertidos no objeto ActionElement da estrutura KDM. O primeiro possui a String ‘global’ no atributo kind e o segundo possui a String ‘persistent’. A expressão encontrada em cada um desses objetos é convertida no elemento KDM correspondente e adicionado à lista de AbstractCodeElement do elemento KDM que representa o elemento Global ou Persistent.

O objeto OctaveTry que representa uma estrutura de tratamento de exceção da linguagem Octave é convertido no Objeto ActionElement da estrutura KDM e possui a String ‘try’ no atributo kind. O nó que possui o objeto da classe OctaveTry pode conter n filhos com objetos de qualquer uma das classes classificadas como OctaveUniversalElements. Cada um desses objetos é convertido no elemento KDM correspondente e adicionado à lista de AbstractCodeElement do elemento KDM que representa o objeto OctaveTry. Além disso, esses nós também contém um nó filho com objeto da classe OctaveCatch.

Os objetos da classe OctaveCatch são convertidos no objeto ActionElement da estrutura KDM com a String ‘catch’ no atributo kind. Os nós que contém o objeto da classe OcatveCatch também podem ter n filhos com objetos de qualquer uma das classes classificadas como OctaveUniversalElements. Cada objeto presente nesses nós filhos são convertidos no elemento KDM correspondente e adicionados à lista de AbstractCodeElement do elemento KDM que representa o objeto OctaveCatch.

O objeto OctaveUnwindProtect é convertido no Objeto ActionElement da estrutura KDM e possui a String ‘unwindprotect’ no atributo kind. O nó que possui o objeto da classe OctaveUnwindProtect pode conter n filhos com objetos de qualquer uma das classes classificadas como OctaveUniversalElements. Cada um desses objetos é convertido no elemento KDM correspondente e adicionado à lista de AbstractCodeElement do elemento

KDM que representa o objeto OctaveUnwindProtect. Além disso, esses nós também contém um nó filho com objeto da classe OctaveUnwindProtectCleanup.

Os objetos da classe OctaveUnwindProtectCleanup são convertidos no objeto ActionElement da estrutura KDM com a String ‘unwindprotectcleanup’ no atributo kind. Os nós que contém o objeto da classe OctaveUnwindProtectCleanup também podem ter n filhos com objetos de qualquer uma das classes classificadas como OctaveUniversalElements. Cada objeto presente nesses nós filhos é convertido no elemento KDM correspondente e adicionado à lista de AbstractCodeElement do elemento KDM que representa o objeto OctaveUnwindProtectCleanup.

O objeto OctaveSwitch que corresponde à estrutura Switch Case da linguagem Octave é convertido no objeto ActionElement da estrutura KDM e possui a String ‘switch’ no atributo kind. O objeto OctaveSwitch possui um atributo do tipo OctaveExpression, que também é convertido no elemento KDM e adicionado à sua lista de AbstractCodeElement. Além disso, o nó que contém o objeto OctaveSwitch pode ter n filhos com objetos da classe OctaveCaseSwitch e um filho com um objeto da classe OctaveOtherwiseSwitch. Esses objetos são convertidos no elemento ActionElement contendo as Strings ‘case’ e ‘otherwise’ respectivamente no atributo kind e adicionados à lista de AbstractCodeElement do elemento KDM que representa o objeto OctaveSwitch.

O objeto OctaveCaseSwitch também possui um atributo que corresponde à uma instância da classe OctaveExpression. Esse objeto é convertido no elemento KDM correspondente e adicionado à lista de AbstractCodeElement do elemento KDM que representa o objeto da classe OctaveCaseSwitch.

Os nós que possuem objetos da classe OctaveCaseSwitch e OctaveOtherwiseSwitch podem ter n filhos com objetos de qualquer uma das classes classificadas como OctaveUniversalElements. Cada um desses objetos encontrados é convertido no elemento KDM correspondente e adicionado à lista de AbstractCodeElment dos elementos KDM que representam os objetos das classes OctaveCaseSwitch e OctaveOtherwiseSwitch.

O objeto OctaveVariable utilizado para representar a declaração de variáveis dentro da linguagem Octave é convertido em um ActionElement com a String ’variable declaration’ no atributo kind e, no atributo name, o valor contido no objeto identifier da instância de OctaveVariable.

O objeto OctaveStructure utilizado para representar a declaração de variáveis que representam uma estrutura de dados dentro da linguagem Octave é convertido em um

ActionElement com a String ’variable structure declaration’ no atributo kind. O atributo name é preenchido com o valor contido no objeto value da instância de OctaveStructure. O objeto OctaveVariableStructure utilizado para representar uma chamada à uma variável dentro de uma estrutura de dados da linguagem Octave. Esse objeto é convertido em um ActionElement com a String ’variable structure call’ no atributo kind. O atributo name preenchido com o valor contido no objeto value da instância de OctaveVariableStructure. O objeto OctaveMatrix utilizado para representar uma matriz da linguagem Octave é convertido em um ActionElement com a String ’matrix’ no atributo kind. O atributo name é preenchido com o valor contido no objeto identifier da instância de OctaveMatrix.

O objeto OctaveArray utilizado para representar um vetor da linguagem Octave é convertido em um ActionElement com a String ’array’ no atributo kind. O atributo name preenchido com o valor contido no objeto identifier da instância de OctaveArray.

O objeto OctaveWarning utilizado para representar a função Warning da linguagem Octave é convertido em um ActionElement com a String ’warning’ no atributo kind. O atributo name é preenchido com o valor contido no objeto expression da instância de OctaveArray.

Os objetos das classes OctaveString, OctaveConstant, OctaveImport, OctaveAnonymousFunction e OctaveHandleFunction são convertidos em objetos da classe ActionElement com as Strings ’string’, ’constant’, ’import’, ’anonymousfunction’ e ’handlefunction’ respectivamente no atributo kind. O ActionElement que representa o objeto da classe OctaveString possui o atributo name preenchido com o valor da String Octave. O ActionElement que representa o objeto da classe OctaveConstant possui o atributo name preenchido com o valor da constante Octave. O ActionElement que representa o objeto da classe OctaveImport possui o atributo name preenchido com o valor do import que está sendo feito no objeto OctaveImport. O ActionElement que representa o objeto da classe OctaveHandleFunction possui o atributo name preenchido com o nome da função handle.

O objeto OctaveAnonymousFunction possui uma lista de parâmetros. Cada parâmetro é convertido no elemento KDM correspondente e adicionado à lista de AbstractCodeElement do elemento KDM que representa o objeto da classe OctaveAnonymousFunction. O mesmo acontece com o seu atributo expression.

O objeto OctaveFunctionCall, que corresponde à uma chamada de função da linguagem Octave, é convertido em um ActionElement com a String ’call’ no atributo kind. Caso a chamada de função possua parâmetros, para cada um deles, será criado um

ActionElement correspondente. Todos eles são adicionados à lista de AbstractCodeElement do elemento que representa a chamada de função.

Os objetos OctaveClassDef, OctaveProperties, OctaveEvents, OctaveMethods, OctaveError, OctaveClear, OctaveIdentifier, OctaveBreak, OctaveContinue e OctaveReturn são convertidos em objetos do tipo ActionElement. Os atributos kind dos objetos OctaveError, OctaveIdentifier e OctaveClear são preenchidos com as Strings ’error’, ’identifier’ e ’clear’ respectivamente. O atributo name é preenchido com o valor contido nos seus objetos. Os atributos kind e name dos objetos OctaveClassDef, OctaveProperties, OctaveEvents, OctaveMethods, OctaveBreak, OctaveContinue e OctaveReturn são preenchidos com as Strings ’class-def’, ’properties’, ’events’, ’methods’, ’break’, ’continue’ e ’return’ respectivamente.

Os nós da AST que contém objetos do tipo OctaveMethods podem ter filhos com objetos do tipo OctaveFunction. Os nós da AST, que contém objetos do tipo OctaveClassDef, podem ter filhos com objetos dos tipos OctaveProperties, OctaveEvents e OctaveMethods. E os nós da AST, que contém objetos dos tipos OctaveProperties e OctaveEvents, podem ter filhos com objetos do tipo OctavExpression ou OctaveFunctionCall.

4.4 BIBLIOTECA ASTOCTAVE

Para representar todas as operações suportadas pelas linguagens MATLAB e Octave e todos os elementos disponíveis, foram desenvolvidas cinquenta e uma classes estruturais, cada uma das quais representa uma dessas opções ou elemento.

A classe abstrata OctaveElement representa qualquer elemento da linguagem MATLAB ou Octave, ou seja, variáveis e instruções. Ela não tem atributos e é herdada por todas as outras classes.

A Class Block também não tem atributo e é usada para representar um bloco de código dentro de uma função, uma estrutura de repetição ou uma estrutura de tomada de decisão.

A classe OctaveAnonymousFunction representa as funções anônimas das linguagens e possui os atributos parameters e expression. O primeiro corresponde a uma lista de objetos do tipo OctaveElement usado para armazenar todos os parâmetros de uma função anônima - estas podem ser variáveis, chamadas de função, expressões e constantes. O atributo

expression é um objeto do tipo OctaveExpression utilizado para armazenar a expressão da função anônima.

A classe OctaveArray representa uma estrutura de array dos programas MATLAB/Octave e possui dois atributos: value e identifier. O primeiro é uma instância da classe String e serve para armazenar o valor do array. O atributo identifier é uma instância de classe OctaveIdentifier e armazena o nome da variável que corresponde a um array em programas MATLAB/Octave.

A classe OctaveBreak representa a instrução break dos programas MATLAB/Octave usada para parar os cálculos dentro de uma estrutura de repetição.

A classe OctaveSwitch representa a instrução switch e possui o atributo expression correspondente a uma instância da classe OctaveExpression, que representa as instruções do comando switch.

A classe OctaveCaseSwitch também tem o atributo expression, uma instância da classe OctaveExpression e que representa os elementos case do comando switch dos programas MATLAB/Octave.

A classe OctaveOtherwiseSwitch não tem atributos e representa o elemento Otherwise da instrução switch dos programas MATLAB/Octave.

As classes OctaveTry e OctaveCatch não possuem atributos e representam as instruções try e catch dentro dos sistemas MATLAB e Octave, respectivamente.

A classe OctaveFunctionCall representa as chamadas de função dentro dos sistemas MATLAB/Octave. Seus dois atributos identificam o nome da função invocada dentro do programa, bem como os parâmetros passados na invocação. O primeiro atributo é uma instância da classe OctaveIdentifier denominada callIdentifier e o segundo é uma lista de instâncias OctaveElement denominadas parameters.

A classe OctaveConstant representa constantes dentro dos sistemas MATLAB/Octave e possui o atributo value da classe String que armazena o valor da constante.

A classe OctaveContinue não possui atributos e representa a instrução continue dos programas MATLAB/Octave, que são usados dentro das estruturas de repetição para fazer com que a execução salte para a próxima iteração.

A classe OctaveDo representa o statement do dos programas MATLAB/Octave. O statement do é uma das opções de estrutura de repetição fornecida pelas linguagens MATLAB e Octave. Essa classe possui o atributo expression como uma instância da classe OctaveExpression utilizado para representar a condição de parada do statement.

As classes OctaveIf, OctaveElseIf e OctaveElse representam as declarações if juntamente com suas partes elseif e else. As classes OctaveIf e OctaveElseIf possuem o atributo expression que corresponde a condição do teste da estrutura condicional. A classe OctaveElse não possui atributos.

A class OctaveExpression representa uma expressão dentro de um programa MATLAB/Octave. Ela tem o atributo expression como uma instância da classe OctaveElement.

As classes OctaveAssigmentExpression, OctaveDecrementExpression, OctaveDecrementExpression, OctaveLoopingExpression, OctaveTransposeExpression e OctaveUnaryExpression são variações mais específicas de expressões dentro dos programas MATLAB e Octave. Essas classes não herdam diretamente da classe OctaveElement. Em vez disso, elas herdam da classe OctaveExpression, que, por sua vez, herda de OcataveElement.

A classe OctaveAssigmentExpression representa atribuições em programas MATLAB e Octave e tem os atributos variable e value, ambas instâncias da classe OctaveElement. O primeiro representa a variável atribuída e o segundo o valor atribuído. A classe OctaveAssigmentExpression herda da classe OctaveExpression.

As classes OctaveIncrementExpression e OctaveDecrementExpression herdam da classe OctaveExpression e respectivamente representam incrementos e decrementos suportados pela linguagem Octave. Ambas possuem o atributo variable como uma instância da classe OctaveVariable, que representa a variável que está sendo incrementada ou diminuída.

A classe OctaveBinaryExpression herda da classe OctaveExpression e representa as operações binárias das linguagens MATLAB e Octave, tais como soma, subtração, divisão, multiplicação, exponencial etc. Possui os atributos firstOperand, secondOperand e operator, cujos primeiros dois são instâncias da classe OctaveElement e representam os dois operandos. O terceiro atributo é uma String e representa o operador de expressão.

A classe OctaveLoopingExpression herda da classe OctaveExpression e representa uma operação de loop suportada pelas linguagens MATLAB e Octave, fazendo com que uma determinada variável varie de um valor para outro. Por exemplo, x = 1: 3 faz com que a variável x varie seu valor de 1 a 3, iterando de uma em uma unidade. Ela possui os atributos from e to, sendo ambas instâncias da classe OctaveElement e representando respectivamente o valor inicial e final da variável.

A classe OctaveTransposeExpression herda da classe OctaveExpression e representa a operação aritmética de transposição como suportada nas linguagens MATLAB e Octave. Ela

possui o atributo element - instância da classe OctaveElement - que representa o elemento que estará sujeito à operação e operator - instância da classe String - representando o operador usado na expressão, que pode ser («.») ou («.'»).

As classes OctaveFor e OctaveParFor representam respectivamente os loops for e parfor. Ambas as classes possuem dois atributos que são instâncias da classe OctaveExpression. O primeiro é chamado de variable e representa a variável que sofre uma alteração no seu valor dentro da condição do for. O segundo atributo é chamado de expression e representa a expressão cujo resultado produz o valor atribuído à variável.

A classe OctaveUnaryExpression herda da classe OctaveExpression e representa uma expressão unária. Ela possui o atributo element - instância da classe OctaveElement que representa o elemento sujeito à operação - e o atributo operator - instância da classe String e representa um operador que pode ser o sinal de mais («+») ou o sinal de menos («-»).

A classe OctaveFunction representa uma definição de função. Ela possui os atributos return - instância da classe String e que armazena o valor retornado pela função - identifier que é um objeto da classe OctaveIdentifier e armazena o nome da função que está sendo definida e parameters, que corresponde a uma lista de objetos da classe OctaveElement e serve para conter os parâmetros que são passados na invocação da função.

A classe OctaveGlobal representa as variáveis globais suportadas pelas linguagens MATLAB e Octave. Ela possui o atributo expression que é uma instância da classe OctaveExpression e armazena a expressão classificada como global.

A classe OctavePersistent representa as variáveis persistentes suportadas pelas linguagens MATLAB e Octave. Ela possui o atributo expression que é uma instância da classe OctaveExpression e armazena a expressão que está sendo classificada como persistente.

A classe OctaveHandleFunction representa as funções Handle suportadas pelas linguagens MATLAB e Ocatve. Ela possui o atributo identifier - instância da classe OctaveIdentifier que armazena o identificador da função Handle.

A classe OctaveIdentifier representa um identificador nos programas MATLAB/Octave. Esses artefatos identificam variáveis, funções, chamadas de função e assim por diante. Ela possui o atributo name - instância da classe String que armazena o identificador em questão.

A classe OctaveImport representa cláusulas de importação nos sistemas MATLAB/Octave. Ela possui o atributo value - instância da classe String - que armazena o valor de importação.

A classe OctaveMatrix representa o tipo de dados de matriz suportado pelas linguagens MATLAB e Octave. Ela possui os atributos value - instância da classe String que armazena o conteúdo da matriz - e o atributo identifier - instância da classe OctaveIdentifier que armazena o identificador da matriz.

A classe OctaveComparisonOperators herda da classe OctaveExpression e representa operações de comparação. Ela possui os atributos firstOperating e secondOperating, que são instâncias da classe OctaveElement. Eles armazenam os elementos que serão comparados. Ela também possui o atributo operator, que é uma instância da classe String e identifica a operação de comparação específica que está sendo utilizada.

A classe OctaveLogicalOperators herda da classe OctaveExpression e representa as operações booleanas suportadas pelas linguagens MATLAB e Octave. Ela possui os atributos firstOperating e secondOperating que são instâncias da classe OctaveElement e armazena os elementos envolvidos na operação lógica. Ela também possui o atributo operator - instância da classe String - que identifica a operação lógica em questão.

A classe OctaveReturn representa a ação de retorno dentro de uma função.

A classe OctaveString representa uma sequência de caracteres dos programas MATLAB e Octave. Ela possui o atributo value - instância da classe String - que armazena esta sequência de caracteres.

A classe OctaveStructure representa estruturas de dados como suportado pelas linguagens MATLAB e Octave. Ela possui o atributo value - instância da classe String - que armazena a estrutura de dados.

As classes OctaveUnwindProtect e OctaveUnwindProtectCleanup representam as instruções OctaveUnwindProtect e OctaveUnwindProtectCleanup respectivamente. Elas não possuem atributos.

A classe OcatveVariable representa uma variável dentro de um sistema MATLAB/Octave. Ela possui o atributo identifier - instância da classe OctaveIdentifier - mantendo a identificação da variável.

A classe OctaveVariableStructure representa um acesso a uma variável pertencente a uma estrutura de dados. Ela possui o atributo identifier - instância da classe OctaveIdentifier - mantendo a identificação da variável.

A class OctaveWarning representa a função de aviso como suportado pelas linguagens MATLAB e Octave. Ela possui o atributo expression - instância da classe OctaveExpression - que armazena a expressão a ser usada pela função Warning.

A classe OctaveWhile representa a estrutura de repetição while, disponível nas linguagens MATLAB e Octave. Ela possui o atributo expression - instância da classe OctaveExpression - que armazena a expressão correspondente ao critério de parada dentro do loop.

A classe OctaveError representa uma função especial chamada ERROR das linguagens MATLAB e Octave que é usada para lançar mensagens de erro durante a execução do programa. A classe OctaveError tem um atributo do tipo OctaveElement que armazena a mensagem de erro que será lançada pelo programa.

A classe OctaveClear também é usada para representar uma função especial chamada CLEAR que é usada para limpar variáveis. A classe OctaveClear tem um atributo - instância da classe OctaveElement - que armazena a variável que deverá ter seu conteúdo limpo.

A versão 4.0 do Octave fornece um novo recurso importante que merece uma atenção: o suporte ao desenvolvimento baseado em classes através do uso do framework ClassDef. Através desta estrutura, é possível desenvolver classes, com atributos, eventos e métodos. Para representar essa estrutura no AST, a classe OctaveClassDef foi criada. Possui um atributo - instância da classe OctaveElement - que armazena uma expressão usada na criação de um objeto Octave.

Além da classe OctaveClassDef, algumas classes adicionais foram criadas para representar os atributos ou propriedades do Octave: OctaveProperties representa o bloco de código properties utilizado para definir as variáveis da classe, OctaveEvents representa os eventos que podem ser adicionados às classes e OctaveMethods que representa o bloco de código responsável por armazenar os métodos ou funções das classes. A classe OctaveMethods não tem nenhum atributo. As classes OctaveProperties e OctaveEvents possuem um atributo - instância da classe OctaveElement - para representar as expressões que podem ser usadas na definição dos atributos e eventos da classe. Deve-se notar que estas quatro classes foram desenvolvidas após testar a biblioteca, pois somente nessa fase se

percebeu que esses tipos de construções existiam, uma vez que nenhuma referência na documentação GNU foi encontrada sobre esse tópico.

Além das classes estruturais citadas acima, foram criadas três outras classes: duas para representar a AST e outra para representar o analisador responsável pela leitura do código-fonte MATLAB/Octave, identificando os elementos relevantes e criando os objetos correspondentes, bem como adicionando-os aos nós da AST.

As classes que representam a AST são OctaveTree e Node. A primeira possui apenas um apontador para o primeiro elemento da árvore, que corresponde a um objeto da classe Node. A segunda possui um objeto da classe OctaveElement utilizado para armazenar o elemento no nó e uma lista de objetos da classe Node, que servem para guardar os nós filhos.

A classe OctaveParserAST é responsável por fazer o parser. Ela possui o método de acesso público generateAST que recebe como parâmetro um objeto do tipo File. Esse método dispara uma chamada ao método de acesso privado parse que, por fim, retorna um objeto da classe OctaveTree.

Quando um objeto File é passado para o método generateAST, o conteúdo deste arquivo é armazenado em um objeto que corresponde a um vetor de char - atributo da classe. O método generateAST também cria um objeto Block que representa o bloco principal de instruções do arquivo e é armazenado como o primeiro nó da AST. O nó que contém este objeto é inserido em uma lista de elementos pendentes - listPendingNode - a serem fechados, que corresponde a um atributo privado da classe OctaveParserAST.

Essa lista armazena as estruturas com blocos de código. Assim, todos os nós criados são armazenados na AST como filhos do nó que foi armazenado na lista mais recentemente. A Figura 11 mostra, de uma forma geral, como esse fluxo ocorre.

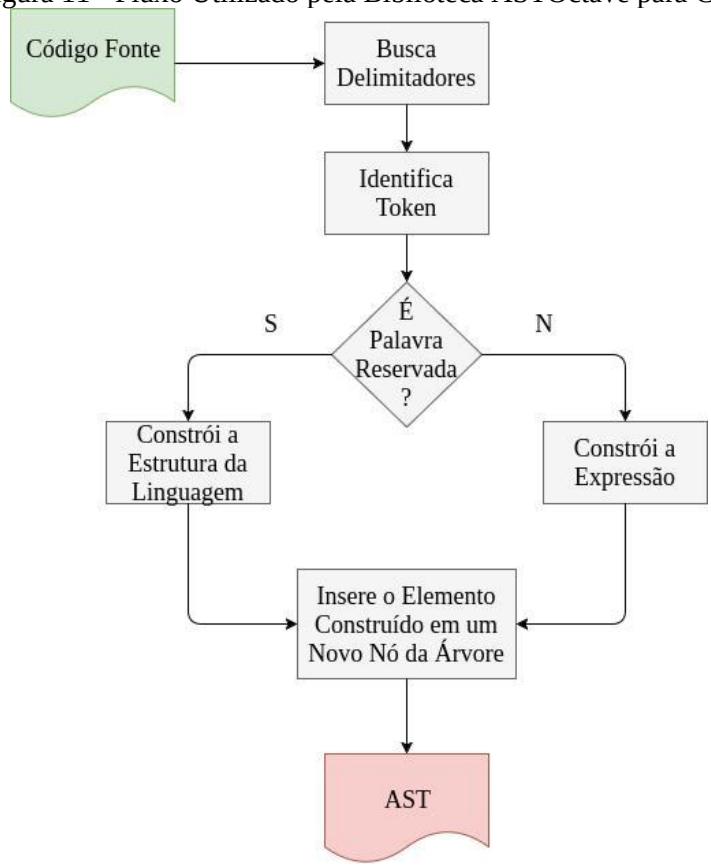
Para efetuar o parse foi necessário identificar as palavras reservadas das linguagens, os caracteres que são usados como delimitadores entre os tokens e os operadores aritméticos, lógicos e relacional.

As palavras reservadas encontradas são: FUNCTION, FOR, PARFOR, IMPORT, GLOBAL, PERSISTENT, IF, ELSEIF, ELSE, SWITCH, CASE, OTHERWISE, WHILE, DO, UNTIL, TRY, CATCH, UNWIND_PROTECT, UNWINDPROTECTCLEANUP, BREAK, CONTINUE, RETURN, CLASSDEF, PROPERTIES, METHODS, EVENTS, WARNING, ERROR, CLEAR, END, ENDFUNCTION, ENDTRYCATCH, ENDUNWIND-PROTECT, ENDFOR, ENDGRENT, ENDIF, ENDPARFOR, ENDPWENT, ENDSWITCH e

ENDWHILE. As últimas onze palavras iniciadas pelo prefixo ’END’ representam finalizadores de blocos.

Os operadores suportados pelas linguagens são: («=»), («==»), («>»), («>=»), («<»), («<=»), («&»), («&&»), («j»), («j j»), («!»), («!=»), (« »), (« »), (« =»), («^»), («.^»), («+»), («++»), («.+»), («-»), («–»), («.-»), («*»), («.*»), («.**»), («/»), («./»), («n»), («.n»). Todos esses operadores também foram tratados como caracteres delimitadores. Além desses, os seguintes caracteres delimitadores são suportados: («{»), («[»), («(»), («)»), («]»), («}»), («:»), («nn»), («nr»), («nt»), («;») e («,»).

Figura 11 - Fluxo Utilizado pela Biblioteca ASTOctave para Geração da AST



O método parse faz uma leitura, caractere a caractere, do código fonte MATLAB/Octave e vai concatenando esses caracteres até encontrar um delimitador. Quando isso acontece, os caracteres concatenados formam um lexema, que, logo em seguida, é convertido em um token da linguagem. O método parse verifica se o token corresponde a uma palavra reservada da linguagem. Caso positivo, o token é passado como parâmetro ao método buildOctaveStructure, que de acordo com o token encontrado, verificará qual método invocar.

Quando o conteúdo do token não corresponde a uma palavra reservada da linguagem, o método mantém a leitura dos caracteres restantes e vai concatenando-os em uma outra variável. Essa leitura acontece até que uma quebra de linha seja encontrada ou o caractere («;»), que é o delimitador indicando a conclusão de uma expressão. O lexema formado é convertido em um token que é passado como parâmetro ao método buildOctaveExpression, que decodifica a expressão correspondente.

Em geral, cada palavra reservada corresponde a uma estrutura da linguagem e para cada uma dessas estruturas há um método responsável por construí-la. Assim, se a palavra reservada encontrada corresponde à String ’FUNCTION’, o método buildFunction é chamado; se a string tiver o valor ’FOR’, o método buildFor é chamado; e assim por diante. No caso da palavra reservada for um finalizador de bloco, significa dizer que o fim do bloco foi atingido e o método closeBlock é chamado. Esse método remove o último nó da lista de elementos pendentes. Os nós subsequentes são armazenados na AST como filhos do nó anterior, que se torna o novo nó mais recente.

Os métodos buildOctaveDo, buildUnwindProtect e buildTry criam objetos respectivamente dos tipos OctaveMethods, OctaveDo, OctaveUnwindProtect e OctaveTry, que são encapsulados em instâncias de Node e armazenados na AST como filhos do nó na última posição na lista de elementos pendentes. Em seguida, os nós são inseridos nessa lista também.

Os métodos buildError, buildClassDef, buildProperties, buildEvents, buildWarning, buildGlobal, buildPersistent, buildWhile, buildUntil, buildIf, buildElseIf e buildSwitch fazem uma chamada ao método getNextExpression, que localiza o texto da próxima expressão, que pode ser composto por um ou mais tokens. Ele percorre os caracteres e os concatena, até que um delimitador de expressão seja encontrado, que pode ser («;») ou uma quebra de linha. O texto da expressão é retornado aos métodos que invocaram getNextExpression e é passado para o método buildOctaveExpression, que o converte em um objeto OctaveExpression.

Para criar o objeto do tipo OctaveExpression, o método buildOctaveExpression precisa resolver a expressão que foi passada como parâmetro. Para tanto, cada operador, seja aritmético, lógico ou de comparação recebeu um identificador de acordo com a prioridade desses elementos dentro da linguagem. Isso quer dizer que um operador de multiplicação possui uma prioridade maior do que um operador de soma e assim por diante. Esses identificadores variam do número 0 ao 9. Em seguida, foi aplicado o algoritmo de shunting yard que é responsável por transformar as expressões de uma notação usual para a notação

polonesa inversa. Após essa conversão, foi utilizado o recurso de pilha para montar as expressões e armazená-las nos objetos correspondentes.

Foram desenvolvidas dez classes para representar as expressões das linguagens MATLAB e Octave. A primeira classe chamada de OctaveExpression foi desenvolvida com objetivo de ser utilizada genericamente dentro da biblioteca. Todas as demais classes utilizam o conceito de herança do paradigma de orientação à objetos para herdar características e comportamentos da classe OctaveExpression.

O método buildError instancia um objeto do tipo OctaveError, passando como parâmetro do seu construtor o objeto OctaveExpression criado. Em seguida, o objeto instanciado é encapsulado em um objeto do tipo Node que é armazenado na AST como filho do objeto Node que se encontra na última posição da lista de elementos pendentes listPendingNode. Um procedimento semelhante foi feito nos métodos buildClassDef, build-Properties, buildEvents, buildWarning, buildGlobal, buildPersistent, buildWhile, buildUntil, buildIf, buildElseIf e buildSwitch.

Porém ao invés de instanciar um objeto do tipo OctaveError, eles instanciam objetos do tipo OctaveClassDef, OctaveProperties, OctaveEvents, OctaveWarning, Octave-Global, OctavePersistent, OctaveWhile, OctaveUntil, OctaveIf, OctaveElseIf e OctaveSwitch respectivamente. Os nós que contém objetos do tipo OctaveClassDef, OctaveProperties, OctaveEvents, OctaveWhile, OctaveIf, OctaveElseIf e OctaveSwitch além de serem armazenados na AST, são também inseridos na lista de elementos pendentes.

Quando um objeto do tipo OctaveUntil é instanciado, significa dizer que o último nó da lista de elementos pendentes possui um objeto do tipo OctaveDo. Esse nó é removido da lista de elementos pendentes e o objeto OctaveUntil é atribuído ao objeto OctaveElement do OctaveDo removido da lista.

Os métodos buildOctaveContinue, buildOctaveBreak e buildOctaveReturn instanciam objetos do tipo OctaveContinue, OctaveBreak e OctaveReturn respectivamente, encapsula esses objetos em nós que são adicionados à AST como filhos do nó armazenado na última posição da lista de elementos pendentes.

Os métodos buildClear e buildImport possuem um loop responsável por concatenar os próximos caracteres do array de char até encontrar um delimitador que finaliza a expressão, que pode ser um («;») ou uma quebra de linha. Em seguida, os métodos buildClear e buildImport instanciam objetos do tipo Octaveclear e OctaveImport respectivamente passando

como parâmetro em seus construtores a String concatenada. Os objetos instanciados são encapsulados em um nó que é armazenado na AST como filho do último nó da lista de elementos pendentes.

Os métodos buildUnwindProtectCleanup, buildCatch e buildElse instanciam objetos do tipo OctaveUnwindProtectCleanup, OctaveCatch e OctaveElse respectivamente. Em seguida eles encapsulam esses objetos em nós que são armazenados na AST como filho do último nó da lista de elementos pendentes. Após a inserção na AST, esses métodos removem o último nó da lista de elementos pendentes e adicionam os novos nós instanciados.

O método buildCaseSwitch faz uma chamada ao método getNextExpression para receber o texto da próxima expressão. Em seguida, ele passa o texto recebido como parâmetro para o método buildOctaveExpression para criar o elemento OctaveExpression. Após a montagem da expressão, um objeto do tipo OctaveCaseSwitch é instanciado passando como parâmetro do construtor o OctaveExpression criado. Esse objeto é encapsulado em um nó que é armazenado na AST. Porém antes disso acontecer, é preciso saber quem será o nó pai deste elemento na AST. Para tanto, é preciso verificar se o último nó da lista de elementos pendentes possui um objeto do tipo OctaveCaseSwitch ou um objeto do tipo OctaveSwitch. Se o objeto for do tipo OctaveSwitch, ele será armazenado como filho do último nó da lista. Se o último nó da lista possuir um objeto do tipo OctaveCaseSwitch, o algoritmo removerá esse nó da lista. Após a remoção, o último nó da lista passará a ser o anterior, que com certeza possuirá um objeto do tipo CaseSwitch encapsulado. Depois dessa remoção, o nó instanciado será adicionado à AST como filho do último nó da lista de elementos pendentes. Em seguida ele é adicionado à lista de elementos pendentes.

O método buildOtherwiseSwitch é muito parecido com o buildCaseSwitch. A diferença é que não existe expressão e o objeto criado e encapsulado no nó corresponde à uma instância da classe OctaveOtherwiseSwitch. Como não existe expressão, ele não faz chamadas aos métodos getNextExpression e buildOctaveExpression.

Os métodos buildParfor e buildFor possuem um loop com objetivo de pegar o próximo token, que corresponde à variável que será incrementada na estrutura de repetição. Em seguida, o algoritmo utiliza mais um loop para pegar a expressão que vem depois do símbolo («=»). Após capturar a variável de atribuição e a expressão utilizada, os métodos buildParFor e buildFor instanciam objetos do tipo OctaveParfor e Octavefor respectivamente, passando como parâmetros em seus construtores dois objetos do tipo OctaveExpression. O primeiro corresponde à variável de atribuição e o segundo à expressão. O objeto instanciado é

encapsulado em um nó e armazenado como filho do último nó da lista de elementos pendentes. Em seguida, o nó é inserido na lista.

O método buildFunction instancia um objeto do tipo OctaveFunction. Em seguida ele utiliza um loop para pegar a assinatura da função. Nesse loop o método é capaz de descobrir o retorno da função (que é convertido em uma instância da classe String), caso exista, o nome da função e os parâmetros, caso existam. O nome da função é convertido no objeto OctaveIdentifier. O retorno é convertido num objeto do tipo String e os parâmetros são convertidos em objetos do tipo OctaveElement e armazenados numa lista. Os três objetos são adicionados ao OctaveFunction, que por sua vez é encapsulado em um nó e armazenado na AST como filho do último nó da lista de elementos pendentes. Em seguida ele também é adicionado à lista de elementos pendentes.

Para ilustrar o resultado do processo de geração da AST dos programas MATLAB/Octave, foi desenvolvido um script contendo uma função simples, como pode ser visto na figura 12(a). Este script foi submetido à biblioteca ASTOctave que por sua vez gerou a AST mostrada na figura 12(b).

Figura 12 - Exemplo de Conversão de um Código Fonte Octave em sua Respectiva AST

```
function x = soma(a,b)
    x = a + b;
    return;
end
```

(a) Código Fonte Octave

```
BlockUnit
    OctaveFunction
        OctaveAssigmentExpression
            OctaveBinaryExpression
                PlusOperator
                    OctaveVariable
                        OctaveIdentifier - a
                    OctaveVariable
                        OctaveIdentifier - b
            OctaveVariable
                OctaveIdentifier - x
        OctaveExpression
            OctaveReturn
```

(b) AST de um Programa Octave

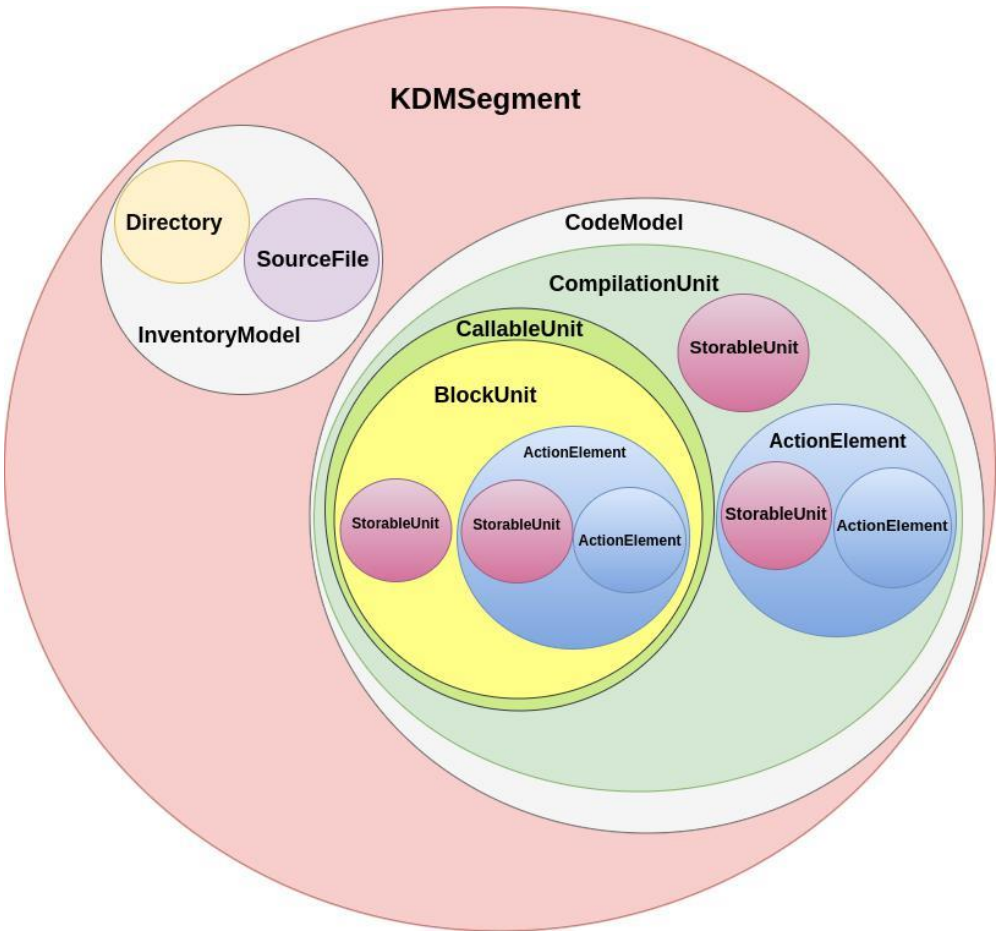
4.5 BIBLIOTECA OCTAVEKDMSTRUCTURE

A representação XMI não pode ser produzida através de uma simples passagem linear da AST, uma vez que existem múltiplas dependências cruzadas entre os elementos. A solução adotada usa uma estrutura de classes que armazena no formato KDM todos os elementos dos nós da AST, que são posteriormente gravados no arquivo XMI. A figura 14 mostra o diagrama de classes da biblioteca OctaveKDMStructure.

O metamodelo utilizado foi concebido para representar uma gama muito vasta de aplicações quanto possível, incluindo as desenvolvidas com as características orientadas a objetos. Portanto, alguns elementos e pacotes não são estritamente necessários para representar arquivos MATLAB/Octave. Por esse motivo, alguns elementos foram ignorados e não fazem parte das estruturas criadas. A figura 13 mostra os elementos KDM utilizados para representar as aplicações MATLAB e Octave com suas respectivas hierarquias.

A classe KDMSegment representa o elemento Segment descrito no pacote KDM, que contém um conjunto de fatos sobre um software MATLAB/Octave. Essa classe pode conter uma ou mais instâncias do modelo KDM. Isso significa dizer que é possível armazenar nesse elemento uma ou mais visões de arquitetura para um software existente. Ela possui os atributos name - para guardar o nome do projeto MATLAB/Octave - e listKDMModel, que consiste em uma lista de instâncias da classe KDMModel.

Figura 13 - Estrutura Hierárquica dos Objetos KDM



A classe KDMModel reflete o elemento Model descrito no pacote Core, que serve para representar alguns aspectos do software existente. Ela é abstrata, pois não existem instâncias desse elemento definidas no KDM. Ela serve para permitir que elementos de outros pacotes possam herdar suas características. Possui apenas o atributo name, que foi utilizado para armazenar o nome da visão de arquitetura que o modelo está descrevendo.

Foram criadas duas classes de modelos: CodeModel e InventoryModel. A primeira representa um modelo simples do KDM que pode conter elementos dos pacotes Code e Action. A segunda reflete um modelo do pacote Source que serve para armazenar um conjunto de fatos relacionados aos artefatos físicos do software existente.

A classe InventoryModel herda características da classe KDMModel através do conceito de herança descrito pelo paradigma de orientação a objetos. Portanto, além de possuir o atributo name já presente na classe KDMModel, possui também o atributo listAbstractInventoryElement, que foi utilizado para armazenar uma coleção de objetos do tipo AbstractInventoryElement. Porém, essa classe é abstrata e não pode ser instanciada. Por esse motivo, foram criadas duas classes concretas que herdam suas características: SourceFile e Directory, ambas descritas no pacote Source.

A classe SourceFile é utilizada para representar os arquivos de código fonte do software existente. Nesta classe estão presentes os atributos name, que armazena o nome físico do arquivo, e path, que guarda o caminho onde o arquivo está armazenado fisicamente. A classe Directory representa os diretórios do software existentes e possui o atributo path para indicar o caminho físico no sistema de arquivos.

A classe CodeModel descrita no KDM possui o atributo name, herdado de KDM-Model, e o atributo listModule para armazenar uma lista de objetos do tipo Module, que corresponde a uma classe abstrata descrita no pacote Code que serve para representar um módulo ou um componente do software, como é determinado pela linguagem de programação.

Existem cinco módulos descritos na documentação do KDM. Cada um deles é representado por uma classe que herda da classe Module. Na estrutura criada neste projeto foi utilizado apenas um módulo. Isto porque os outros quatro módulos não são necessários para representar aplicações Octave ou Matlab. O módulo utilizado é o CompilationUnit, que corresponde a parte lógica contida nos arquivos fontes do software existente. Por esse motivo, foi criada uma classe chamada CompilationUnit, mas que, ao invés de herdar da classe Module, herda características da classe AbstractCodeElement. A classe Module não foi

implementada e a classe CodeModel descrita anteriormente, ao invés de possuir um objeto que representasse uma lista de objetos do tipo Module, possui uma lista de CompilationUnit.

Foi necessário então, criar uma classe chamada AbstractCodeElement que é descrita como um elemento do pacote Code e representa algo genérico determinado pela linguagem de programação. Ela possui um atributo chamado codeElements, que corresponde a uma lista de objetos do tipo AbstractCodeElement. Isso acontece porque dentro de um elemento de código, é possível existir outros elementos de código.

Além da classe Module já mencionada, existem mais duas outras classes que também herdam características de AbstractCodeElement: ActionElement e ComputationalObject. A classe ActionElement está descrita como elemento do pacote Action e é utilizada para representar unidades de comportamento dentro do código fonte do software existente. Esses comportamentos podem ser um teste condicional, uma estrutura de repetição, uma chamada de função etc. Possui os atributos kind, usado para representar o significado das operações e listAbstractCodeElement, que é uma lista de objetos da classe AbstractCodeElement. Um tipo especial de ActionElement precisou ser criado: BlockUnit. É um elemento do pacote Action que representa blocos de código dentro de um programa MATLAB/Octave.

A classe ComputationalObject é abstrata e representa um elemento descrito no pacote Code que descreve objetos computacionais em tempo de execução como variáveis e procedimentos. Foram criadas três classes concretas que herdam de ComputationalObject e estão descritas no pacote Code: CallableUnit, StorableUnit e ParameterUnit. CallableUnit representa um elemento que pode ser invocado através de um mecanismo de chamada-retorno, como uma procedure ou uma função. Possui os atributos kind, que indica o tipo de chamada, signature que é um objeto do tipo Signature descrito no KDM e listAbstractCodeElement que corresponde a uma lista de objetos do tipo AbstractCodeElement. Signature é uma classe também descrita no pacote code utilizada para representar a assinatura de procedimentos ou funções e possui uma lista de objetos do tipo ParameterUnit.

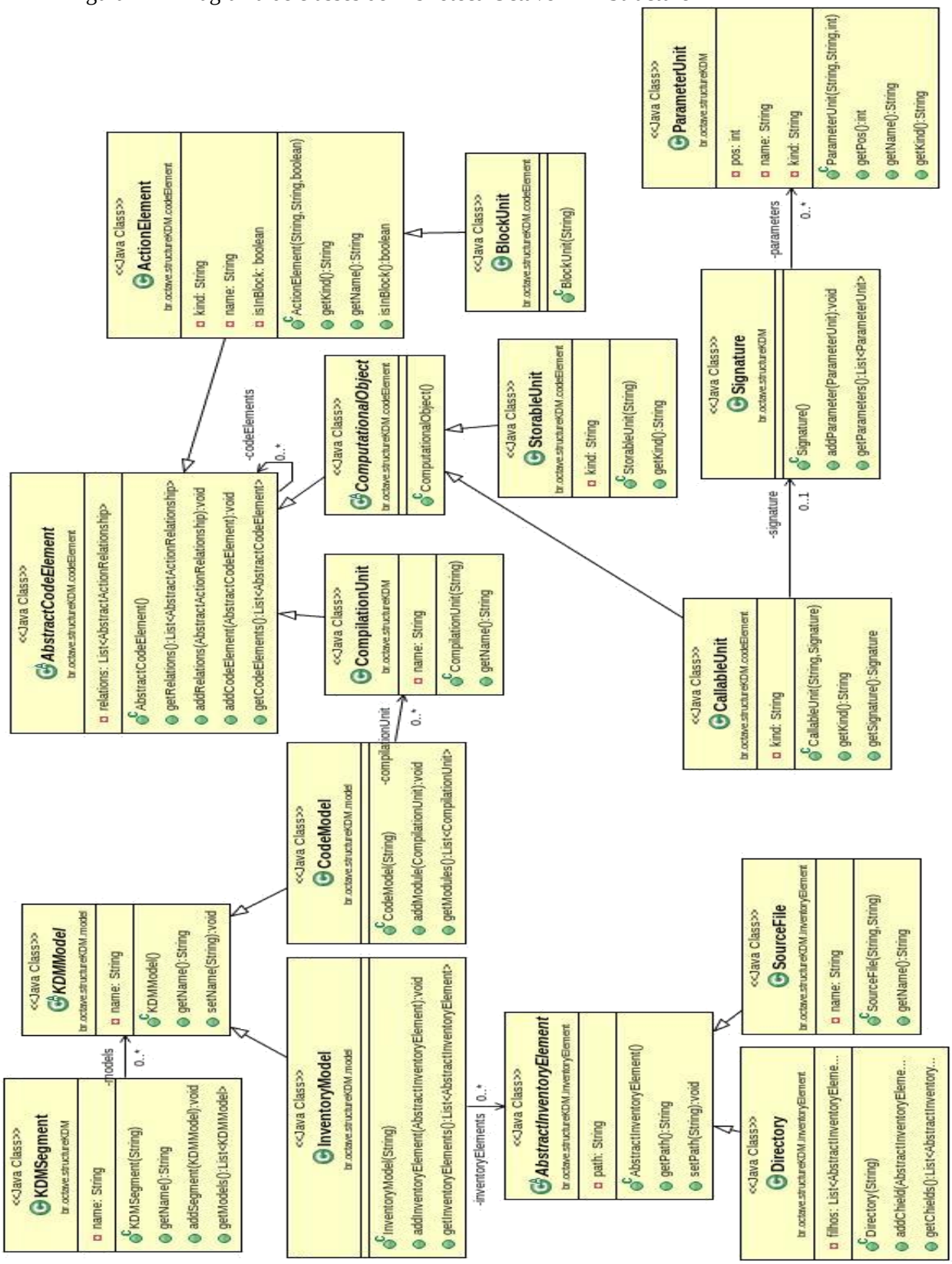
StorableUnit é utilizada para representar as variáveis do software existente e possui o atributo listAbstractCodeElement.

A classe ParameterUnit reflete um parâmetro formal de um procedimento ou função do software existente. Compõe a classe Signature e possui o atributo pos, que indica sua posição na assinatura do procedimento ou da função.

Não foram utilizados os elementos que representam os tipos de dados disponibilizados no KDM. Essa decisão foi tomada porque não existe tipagem de dados na linguagem Octave.

Todas as classes criadas foram empacotadas em uma biblioteca Java que recebeu o nome de OctaveKDMStructure.

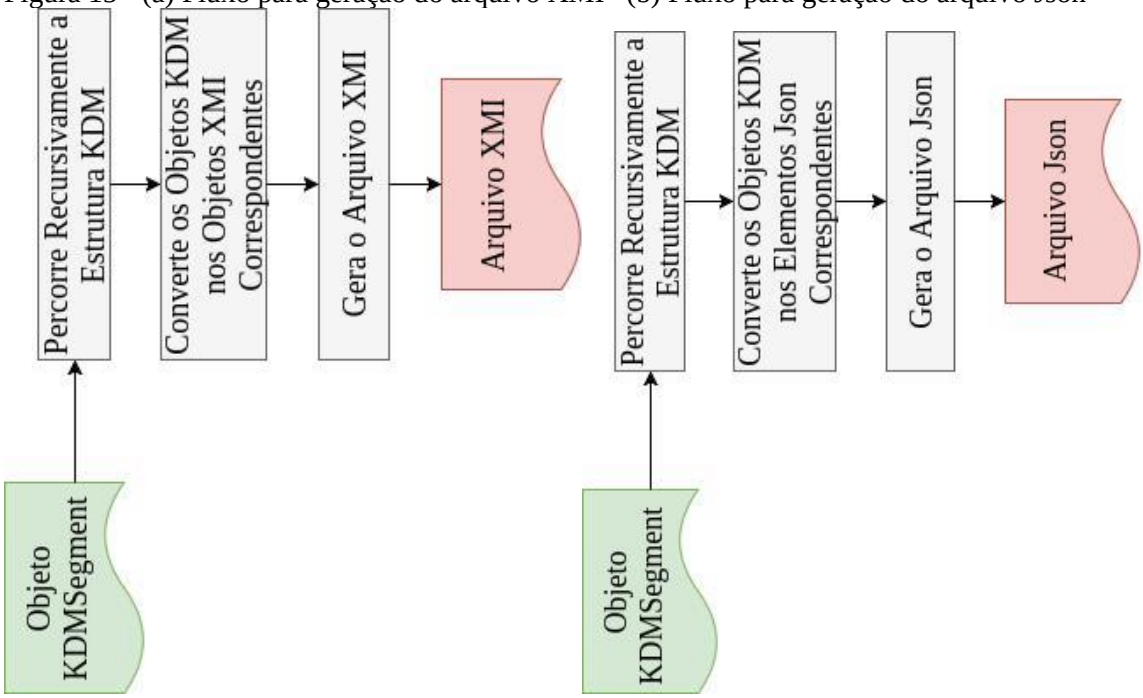
Figura 14 – Diagrama de classes de Biblioteca OctaveKDMStructure



4.6 BIBLIOTECA OCTAVEKDMTOXMI E OCTAVEKDMTOJSON

Depois de criar as bibliotecas ASTOctave, OctaveKDMStructure e ASTOctaveToKDM, foram desenvolvidas duas novas bibliotecas com objetivo de escrever os dados contidos na estrutura KDM em arquivos nos formatos XMI e JSON. As duas bibliotecas fazem exatamente a mesma coisa, porém com saídas diferentes. As bibliotecas OctaveKDMToXMI e OctaveKDMToJSON percorrem os dados armazenados na estrutura KDM, interpretam o tipo de dado encontrado e geram uma saída correspondente nos formatos XMI e JSON respectivamente. O algoritmo utilizado neste processo pode ser visualizado de forma macro através das figuras 15(a) e 15(b) e os elementos de saída para cada objeto KDM encontrado pode ser visto na tabela 5 e 6.

Figura 15 - (a) Fluxo para geração do arquivo XMI (b) Fluxo para geração do arquivo Json



Nas duas bibliotecas foram disponibilizados métodos de acesso público responsáveis por percorrer todos os objetos armazenados na estrutura KDM e convertê-los nos elementos dos arquivos XMI e JSON. Cada objeto deve ser convertido em um elemento específico, segundo o KDM. A hierarquia dos elementos nos arquivos XMI e JSON devem seguir a mesma hierarquia dos objetos na estrutura KDM. Para que os métodos possam varrer a estrutura KDM criada, é preciso passar como parâmetro um objeto do tipo KDMSegment para os construtores das classes

XMIParse e JSONParse disponibilizadas nas bibliotecas OctaveKDMToXMI e OctaveKDMToJSON respectivamente.

Tabela 5 - Objetos KDM e suas Rrespectivas Saídas XMI

KDM Object	Output XMI
KDM Segment	
CodeModel	<model type="code:CodeModel"name="ProjectName">
InventoryModel	<model type="source:InventoryModel"name="source references">
SourceFile	<inventoryElement type="source:SourceFile"name="FileName"path="FilePath">
Directory	<inventoryElement type="source:Directory"name="DirectoryName">
CompilationUnit	<codeElement type="code:CompilationUnit"name="SourceFileName">
CallableUnit	<codeElement type="code:CallableUnit"name="MethodName">
Signature	<codeElement type="code:Signature"name="MethodName">
ParameterUnit	<parameterUnit name="ParameterName"kind="ParameterType"pos="Position">
StorableUnit	<codeElement type="code:StorableUnit"kind="VariableType">
BlockUnit	<codeElement type="action:BlockUnit"kind="BlockFrom">
ActionElement	<codeElement type="action:ActionElement"kind="ActionType">

Tabela 6 Objetos KDM e suas Rrespectivas Saídas JSON

KDM Object	Output JSON
KDM Segment	{"type":"kdm:segment","name":"ProjectName"}
CodeModel	{"type":"code:CodeModel","name":"ProjectName"}
InventoryModel	{"type":"source:InventoryModel", "name":"source references"}
SourceFile	{"type":"source:sourceFile", "name":"FileName", "path":"FilePath"}
Directory	{"type":"source:Directory","name":"DirectoryName"}
CompilationUnit	{"type":"code:CompilationUnit", "name":"SourceFileName"}
CallableUnit	{"type":"code:CallableUnit", "name":"MethodName"}
Signature	{"type":"code:Signature", "name":"MethodName"}
ParameterUnit	{"type":"parameterUnit", "kind":"ParameterType", "name":"ParameterName", "pos":"Position"}
StorableUnit	{"type":"code:StorableUnit", "kind":"VariableType"}
BlockUnit	{"type":"action:BlockUnit", "kind":"BlockFrom"}
ActionElement	{"type":"action:ActionElement", "kind":"ActionType"}

Para exemplificar as saídas XMI e Json, o script descrito na figura 12(a) foi aplicado às bibliotecas OctaveKDMToXMI e OctaveKDMToJSON e os resultados obtidos podem ser visualizados nas figuras 16 e 17, respectivamente.

Figura 16 - Saída do Arquivo XMI

```
<kdm:Segment name="calculadora">
    <model name="calculadora" xmi:type="code:CodeModel">
        <codeElement name="OperacoesBasicas.m" xmi:type="code:CompilationUnit">
            <codeElement kind="Bloco Principal" name="Bloco Principal" xmi:type="action:BlockUnit">
                <codeElement name="soma" xmi:type="code:CallableUnit">
                    <codeElement name="soma" xmi:type="code:Signature">
                        <parameterUnit kind="return" name="x" pos="0"/>
                        <parameterUnit kind="parameter" name="a" pos="1"/>
                        <parameterUnit kind="parameter" name="b" pos="2"/>
                    </codeElement>
                    <codeElement kind="block function" name="block function" xmi:type="action:BlockUnit">
                        <codeElement kind="assignment" name="assignment" xmi:type="action:ActionElement">
                            <codeElement kind="local" xmi:type="code:StorableUnit">
                                <codeElement kind="variable" name="x" xmi:type="action:ActionElement"/>
                            </codeElement>
                            <codeElement kind="aritimetic operator" name="plus" xmi:type="action:ActionElement">
                                <codeElement kind="variable" name="a" xmi:type="action:ActionElement"/>
                                <codeElement kind="variable" name="b" xmi:type="action:ActionElement"/>
                            </codeElement>
                        </codeElement>
                        <codeElement kind="return" name="return" xmi:type="action:ActionElement"/>
                    </codeElement>
                </codeElement>
            </codeElement>
        </codeElement>
    </model>
    <model xmi:type="source:InventoryModel">
        <inventoryElement language="octave" path="calculadora" xmi:type="source:Directory"/>
    </model>
</kdm:Segment>
```

Figura 17 - Saída do Arquivo JSON


```
{"type":"kdm:Segment","name":"calculadora","model":[
    {"type":"code:CodeModel","name":"calculadora",
      "codeElement":[
          {"type":"code:CompilationUnit","name":"OperacoesBasicas.m","codeElement":[
                {"type":"action:BlockUnit","kind":"Bloco Principal","name":"Bloco Principal",
                 "codeElement":[
                     {  "type":"code:CallableUnit","name":"soma",
                        "codeElement":[
                           {  "type":"code:Signature","name":"soma",
                              "parameterUnit":[{"kind":"return","name":"x","pos":"0"},
                                               {"kind":"parameter","name":"a","pos":"1"},
                                               {"kind":"parameter","name":"b","pos":"2"}]
                           },
                           {"type":"action:BlockUnit","kind":"block function","name":"block function",
                              "codeElement":[
                                 {"type":"action:ActionElement","kind":"assignment","name":"assignment",
                                    "codeElement":[
                                       {"type":"code:StorableUnit","kind":"local","codeElement":
                                          [
                                             {"type":"action:ActionElement","kind":"variable","name":"x"}
                                          ]
                                       },
                                       {"type":"action:ActionElement","kind":"aritimetic operator","name":"plus",
                                           "codeElement":[{"type":"action:ActionElement","kind":"variable","name":"a"},
                                                          {"type":"action:ActionElement","kind":"variable","name":"b"}
                                                         ]
                                       }
                                    ]
                                 },
                                 {"type":"action:ActionElement","kind":"return","name":"return"}
                              ]
                           }
                        ]
                     }
                 ]
                }
          ]
      },
      {"type":"source:InventoryModel","inventoryElement":[
            {"type":"source:Directory","path":"calculadora","language":"octave"
            }
      ]
      }
  ]
}
```

4.7 CONCLUSÃO DO CAPÍTULO

Este capítulo explorou os detalhes de implementação do parser responsável por gerar o KDM de projetos MATLAB e Octave, e descreveu os passos necessários para sua implementação.

5 UM AMBIENTE DE VISUALIZAÇÃO BA-SEADO EM MULTIPLAS VISÕES PARA A COMPREENSÃO DE PROGRAMAS MA-TLAB E OCTAVE

Este capítulo descreverá o ambiente de visualização proposto para suportar a análise visual de códigos fonte de programas MATLAB/Octave, exibindo e detalhando a arquitetura utilizada com sua respectiva implementação.

5.1 INFRAESTRUTURA DO AMBIENTE DE VISUALIZAÇÃO

O principal objetivo deste trabalho é propor um ambiente de visualização que possa ser utilizado para apoiar a compreensão de programas desenvolvidos nas linguagens MATLAB e Octave, através do uso do metamodelo KDM.

Para desenvolvê-lo foi necessário estabelecer alguns requisitos funcionais e não funcionais. A tabela 7 contém os requisitos estabelecidos para o desenvolvimento do ambiente de visualização.

Visando atender alguns dos requisitos funcionais e não funcionais estabelecidos, a arquitetura escolhida se baseia no modelo cliente/servidor e recebe o nome de Service-Oriented Architecture (SOA), ou simplesmente arquitetura orientada a serviços. A arquitetura principal da infraestrutura do ambiente de visualização proposto pode ser melhor entendida através da figura 19.

O principal motivador da escolha por essa arquitetura está diretamente relacionado à sua flexibilidade. A arquitetura SOA foi criada com propósito de integrar os diversos sistemas dentro das empresas com baixo custo e pouco tempo gasto. Essa arquitetura permite que novos produtos sejam desenvolvidos dentro de uma aplicação sem que haja necessidade de mudanças no que já foi implementado.

Tabela 7 - Lista dos Requisitos Funcionais e não Funcionais do Ambiente de Visualização

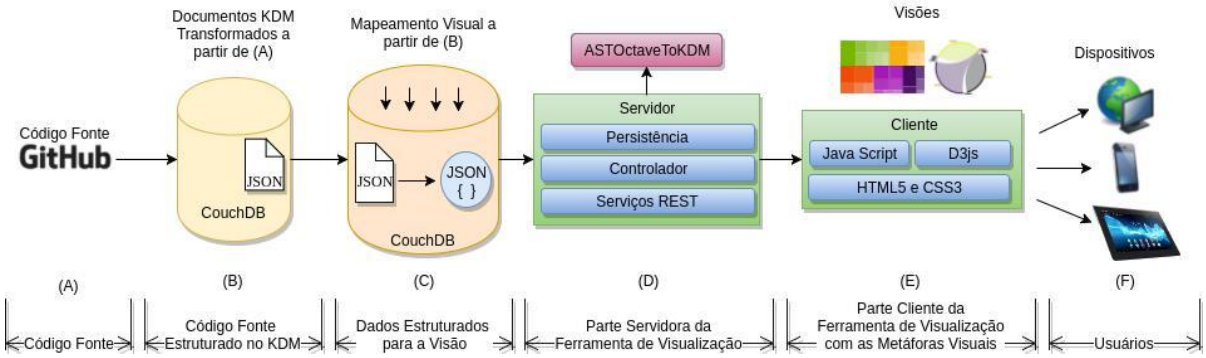
Requisitos Funcionais	Requisitos não Funcionais
Disponibilizar uma metáfora visual para representação hierárquica do código fonte	Permitir o acesso através da WEB
Disponibilizar uma metáfora visual para representar o acoplamento entre os módulos do código fonte	Permitir o acesso através dos dispositivos móveis
Permitir que o usuário navegue de uma determinada entidade selecionada da metáfora visual que representa a estrutura hierárquica do código fonte para a metáfora visual que representa o acoplamento entre os módulos do código fonte	A ferramenta deve fornecer tempo de resposta abaixo de 1 minuto independente do volume de dados
Permitir ao usuário filtrar na metáfora visual, que representa o acoplamento entre módulos do código fonte, a representação de entidades limitadas à uma determinada faixa de número de acoplamento	Utilizar tecnologias open source
Permitir ao usuário selecionar projetos MATLAB/Octave do repositório público Github para que seus respectivos dados de código fonte possam ser importados para o ambiente de visualização e representados visualmente através das metáforas visuais disponíveis	
Permitir ao usuário exportar as estruturas de da dos visuais dos programas analisados em formato Json para que possam ser utilizadas em outros ambientes de apoio à compreensão de programas MATLAB/Octave	
Permitir ao usuário/programador incluir novas metáforas visuais para a representação dos programas analisados	

As tecnologias utilizadas na infraestrutura proposta podem ser visualizadas na figura 18. Essa infraestrutura é composta por um container WEB chamado Apache Tomcat e um sistema de gerenciamento de banco de dados NoSQL chamado CouchDB. Além disso, foram utilizadas nessa infraestrutura a API REST Java para criar os serviços, o jQuery Ajax e o D3.js como uma biblioteca Java Script para manipular objetos DOM com base em dados no formato Json. Essas tecnologias são utilizadas para implementar as partes B, C, D e E do pipeline de visualização representado na Figura 19.

A razão para selecionar um banco de dados NoSQL está diretamente relacionado à um dos requisitos não funcionais do ambiente de visualização proposto, que diz respeito performance da ferramenta. Para tanto, é de fundamental importância que o SGBD utilizado para guardar os dados também sejam eficientes, independente do volume de dados armazenados ou do número de usuários que utilizem a ferramenta. A flexibilidade associada à alta performance dos bancos de dados NoSQL foram os principais pontos levados em consideração para sua adoção nessa arquitetura (CATTELL, 2011) (LI; MANOHARAN, 2013). A flexibilidade dos bancos de dados NoSQL está no fato desse tipo de banco não precisar de schemas predefinidos para salvar os dados (BRITO, 2010).

Figura 18 - Tecnologias Utilizadas na Arquitetura do Ambiente de Visualização

Figura 19 - Arquitetura do Ambiente de Visualização



Existem basicamente quatro modelos de banco de dados NoQL que podem ser utilizados. Os sistemas baseados em armazenamento chave-valor, os bancos orientados a documentos, os sistemas orientados a coluna e os sistemas baseados em grafo (BRITO, 2010). O modelo que melhor se adequa à forma como os dados estão armazenados dentro dos códigos fonte das aplicações é o orientado a documentos.

O banco de dados NoSQL CouchDB, um acrônimo para Cluster of Unreliable Commodity Hardware 1, foi utilizado nessa arquitetura por ele ser um banco de dados distribuído de esquema que fornece uma API ResTful JSON (Java Script Object Notation) para gerenciar os documentos armazenados através de chamadas HTTP. Além disso, o mecanismo HTTP baseado no CouchDB é facilmente escalável para lidar com cargas pesadas de solicitações, respondendo às chamadas HTTP usando os métodos GET, POST, PUT ou DELETE http para uma URL CouchDB específica (ANDERSON; LEHNARDT; SLATER, 2010). O CouchDB é um banco de dados orientado a documentos de alta performance ideal para ser utilizado quando se deseja ter um bom desempenho em grandes volumes de dados. No CouchDB, os documentos armazenados devem estar no formato Json.

Ao salvar os documentos no banco, o CouchDB cria versões desse documento através da atribuição de um campo no documento chamado _rev. Quando esse documento é alterado ou removido, ele não é excluído fisicamente do banco. Um novo documento é criado com um novo valor no atributo _rev.

A parte A da infraestrutura descrita na figura 19 corresponde ao repositório de código fonte utilizado. O repositório escolhido foi o Github. Essa escolha se deu principalmente pela quantidade de projetos MATLAB e Octave disponíveis. Além disso, o Github é um repositório público mundialmente conhecido, utilizado e muito respeitado pelos desenvolvedores de software livre.

Na parte B da infraestrutura ocorre a transformação de dados brutos relacionados ao código fonte dos programas MATLAB/Octave disponíveis no repositório Github (parte A da mesma figura) para estruturas de dados baseadas no KDM para serem armazenadas no CouchDB. A transformação dos dados originais (código fonte) para a estrutura KDM já foi descrita na seção 4.2.

Para armazenar os elementos KDM utilizados na representação de programas MATLAB/Octave, foi necessário fazer um mapeamento desses elementos em documentos no formato Json. Os arquivos Jsons resultantes são ilustrados na parte B da figura 19. O primeiro elemento encontrado na estrutura KDM é o objeto KDMSegment. Esse objeto foi convertido em um documento Json com os seguintes atributos: _id, type e name. Nos atributos _id e name foram inseridos o nome do projeto e no atributo type foi inserida a String "kdm:Segment". O atributo _id de todos os demais documentos é gerado automaticamente pelo banco de dados. A figura 20 ilustra o resultado desse documento.

Figura 20 - Exemplo de um Documento Json que Representa o Objeto KDMSegment

O próximo elemento encontrado é o objeto CodeModel. Além dos atributos criados no KDMSegment, são criados mais dois atributos: idParent e idProject. No caso desse objeto, esses dois atributos terão o mesmo valor, que corresponde ao atributo _id do objeto KDMSegment. O mesmo acontece com o objeto InventoryModel. O atributo type inserido no documento que representa o CodeModel recebe como valor a String "code:CodeModel". No caso do objeto InventoryModel, a String "source:InventoryModel" é atribuída ao atributo type. Todos os demais documentos criados possuem o atributo idProject e o id do documento que representa o KDMSegment é adicionado a esse atributo.

Nos objetos do tipo InventoryModel podem ser encontrados objetos dos tipos Directory e SourceFile. Além dos atributos já citados, os documentos que representam esses objetos possuem também os atributos path e language. A String "MATLAB/Octave" é

atribuída ao atributo language, enquanto o atributo path recebe o caminho do diretório no qual esses arquivos ou pastas estão localizados dentro do projeto. O atributo type do documento que representa o objeto SourceFile recebe a String "source:SourceFile". No caso do objeto Directory, o atributo type recebe a String "source:Directory". O atributo idParent desses documentos recebe o id do documento que representa o objeto InventoryModel.

Os objetos do tipo CompilationUnit encontrados no objeto CodeModel são convertidos em documentos Json com os atributos _id, name, type, idProject e idParent. O atributo name recebe o nome do arquivo fonte que esse elemento representa. O atributo type recebe a String "code:CompilationUnit" e o idParent recebe o id do documento que representa o elemento CodeModel.

O elemento CallableUnit é convertido em um documento com os atributos _id, type, name, idParent, idProject, idCompilation e compilation. O atributo type recebe a String "code:CallableUnit". O atributo name recebe o nome da função representada por esse elemento. O atributo idParent recebe o id do documento que representa o CompilationUnit ao qual essa função faz parte. O atributo idCompiation recebe o id do documento que representa o elemento CompilationUnit ao qual essa função faz parte e o atributo compilation recebe o nome do arquivo propriamente dito.

Os documentos que representam os elementos BlockUnit, CodeElement e StorableUnit possuem os atributos _id, kind, name, idParent, idProject, idCompilation e compilation. Esses documentos podem receber também os atributos idCallable e nameCallable, caso os elementos ao qual eles representem estejam dentro de um elemento que representa uma função. O atributo type dos documentos que representam os elementos BlockUnit, CodeElement e StorableUnit recebem respectivamente os valores "action:BlockUnit", "action:ActionElement" e "code:StorableUnit". O atributo kind é preenchido com o valor armazenado na variável kind do elemento KDM. O mesmo acontece com o atributo name. A regra para o preenchimento dos demais atributos é exatamente a mesma utilizada no documento que representa o elemento CallableUnit.

A figura 21 mostra um exemplo de um arquivo JSON que representa o elemento CallableUnit do KDM de um programa MATLAB/Octave. Os atributos _id e _rev correspondem ao identificador exclusivo deste documento no banco de dados. Eles são gerados automaticamente pelo CouchDB. O atributo _rev indica a versão do documento. O mesmo documento pode ter várias versões dentro do CouchDB. O valor atribuído ao atributo type identifica que esse elemento corresponde à função dentro de um programa

MATLAB/Octave. O atributo name possui o nome dessa função. O atributo idParent indica o _id do documento pai desse elemento que representa uma função. Nesse caso, o documento pai exibido na figura 22 representa o elemento BlockUnit do KDM. Esse BlockUnit corresponde ao bloco de código de um arquivo de código-fonte. O atributo idProject tem o _id do documento que representa o elemento KDMSegment do KDM, representado na figura 20. Esse elemento identifica o projeto em análise. Os atributos idCompilation e compilation identificam o arquivo de código-fonte que contém esta função, representado na figura 23. O elemento KDM que representa um arquivo de código-fonte é o CompilationUnit. O idCompilation tem o id do documento que representa o CompilationUnit e o atributo compilation armazena o nome do arquivo de código-fonte. O atributo signature armazena a assinatura de uma função MATLAB/Octave com seus respectivos parâmetros.

Através das figuras 20, 21, 22, 23 e 24 é possível verificar que o elemento CallableUnit representado pela figura 21 é um elemento que está hierarquicamente logo abaixo do elemento BlockUnit representado pela figura 22 na estrutura KDM. Esse elemento por sua vez está hierarquicamente abaixo do elemento CompilationUnit representado pela figura 23. O CompilationUnit está hierarquicamente abaixo do elemento CodeModel representado pela figura 24 que está hierarquicamente abaixo do elemento KDMSegment representado pela figura 20. Essa verificação pode ser feita através da ligação entre os atributos idParent e id de cada um desses elementos.

Figura 21 - Estrutura de Dados de um Arquivo JSON que Representa o Elemento CallableUnit do KDM Referenciado na Parte B da Figura 19


```
{
  "_id": "c91436ca676f464d9abfbdecabc947d2",
  "_rev": "1-3af9e81243b7ceca927aa6f36e7d135d",
  "type": "code:CallableUnit",
  "name": "caebbp",
  "idParent": "d5b706860a3e49949f12fb330b515cf6",
  "idProject": "DeepLearnToolbox",
  "idCompilation": "1dacc19bfe304f5fbc663ca0c8ef5198",
  "compilation": "caebbp.m",
  "signature": {
      "type": "code:Signature",
      "name": "caebbp",
      "parameterUnit": [
          {
              "kind": "return",
              "name": "cae",
              "pos": "0"
          },
          {
              "kind": "parameter",
              "name": "cae",
              "pos": "1"
          }
      ]
  }
}
```

Figura 22 - Estrutura de Dados de um Arquivo JSON que Representa o Elemento BlockUnit do KDM Referenciado na Parte B da Figura 19


```
{
  "_id": "d5b706860a3e49949f12fb330b515cf6",
  "_rev": "1-f73e11d3769e809b0e28f011f44a5fa7",
  "type": "action:BlockUnit",
  "kind": "Bloco Principal",
  "name": "Bloco Principal",
  "idParent": "1dacc19bfe304f5fbc663ca0c8ef5198",
  "idProject": "DeepLearnToolbox",
  "idCompilation": "1dacc19bfe304f5fbc663ca0c8ef5198",
  "compilation": "caebbp.m"
}
```


Figura 23 - Estrutura de Dados de um Arquivo JSON que Representa o Elemento CompIlationUnit do KDM Referenciado na Parte B da Figura 19


```
{
  "_id": "1dacc19bfe304f5fbc663ca0c8ef5198",
  "_rev": "1-abebd4fc8327ccd4a21a93f9dba53b4f",
  "type": "code:CompilationUnit",
  "name": "caebbp.m",
  "idParent": "cc84aca571ab42119299a78e5ea39116",
  "idProject": "DeepLearnToolbox"
}
```


.

Figura 24 - Estrutura de Dados de um Arquivo JSON que Representa o Elemento Code-Model do KDM Referenciado na Parte B da Figura 19


```
{
   "_id": "cc84aca571ab42119299a78e5ea39116",
   "_rev": "1-a45e15113e2daa82616847c555cb5b8c",
   "type": "code:CodeModel",
   "name": "DeepLearnToolbox",
   "idParent": "DeepLearnToolbox",
   "idProject": "DeepLearnToolbox"
}
```


Para buscar as informações sobre os projetos MATLAB/Octave armazenadas dentro do SGBD, o CouchDB disponibiliza uma funcionalidade para acesso aos dados chamada view. As views são criadas dentro do próprio CouchDB como um documento Json e elas são definidas através de funções Java Script. A figura 25 representa uma view criada dentro do CouchDB. Essa view é capaz de retornar todos os documentos armazenados no banco, cujo

atributo type possua o valor "kdm:Segment". A figura 26 mostra o resultado da execução dessa view.

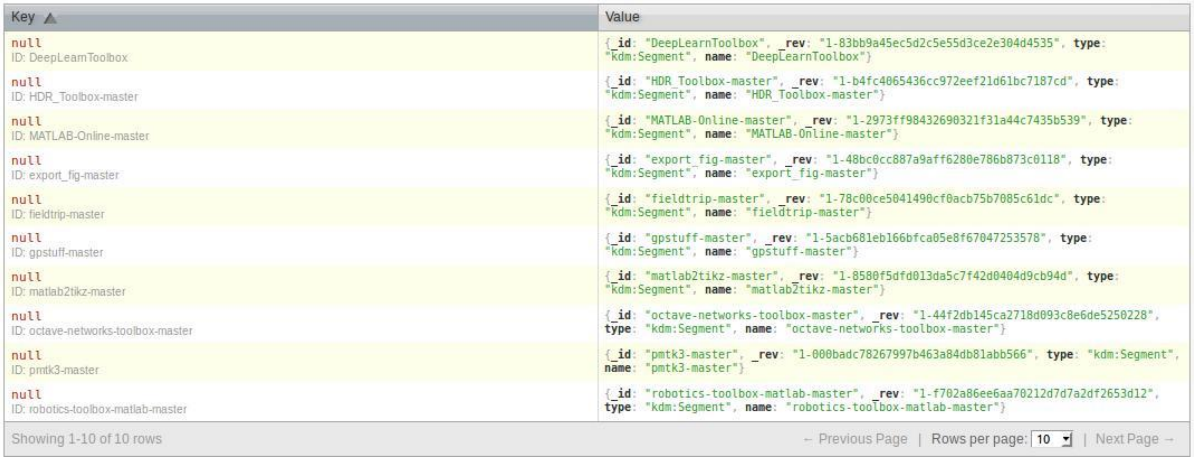
A view exibida na figura 26 foi executada através de uma ferramenta chamada Futon, que corresponde a uma aplicação web utilizada para o gerenciamento do CouchDB.

Figura 25 - Exemplo de uma View no CouchDB

```
function(doc) {
  if(doc.type == "kdm:Segment"){
    emit(doc.id, doc);
  }
}
```

Essa ferramenta permite criar bancos, documentos e views. Além da funcionalidade de criar os documentos, é possível alterá-los ou ainda removê-los. É possível também gerenciar as configurações do CouchDB através do Futon.

Figura 26 - Resultado da Execução da View Exemplificada na Figura 25



Key	Value
null ID: DeepLearnToolbox	{_id: "DeepLearnToolbox", _rev: "1-83bb9a45ec5d2c5e55d3ce2e304d4535", type: "kdm:Segment", name: "DeepLearnToolbox"}
null ID: HDR_Toolbox-master	{_id: "HDR_Toolbox-master", _rev: "1-b4fc4065436cc972eef21d61bc7187cd", type: "kdm:Segment", name: "HDR_Toolbox-master"}
null ID: MATLAB-Online-master	{_id: "MATLAB-Online-master", _rev: "1-2973ff98432690321f31a44c7435b539", type: "kdm:Segment", name: "MATLAB-Online-master"}
null ID: export_fig-master	{_id: "export_fig-master", _rev: "1-48bc0cc887a9aff6280e786b873c0118", type: "kdm:Segment", name: "export_fig-master"}
null ID: fieldtrip-master	{_id: "fieldtrip-master", _rev: "1-78c00ce5041490cf0acb75b7085c61dc", type: "kdm:Segment", name: "fieldtrip-master"}
null ID: gpstuff-master	{_id: "gpstuff-master", _rev: "1-5acb681eb166bfca05e8f67047253578", type: "kdm:Segment", name: "gpstuff-master"}
null ID: matlab2tikz-master	{_id: "matlab2tikz-master", _rev: "1-8580f5dfd013da5c7f42d0404d9cb94d", type: "kdm:Segment", name: "matlab2tikz-master"}
null ID: octave-networks-toolbox-master	{_id: "octave-networks-toolbox-master", _rev: "1-44f2db145ca2718d093c8e6de5250228", type: "kdm:Segment", name: "octave-networks-toolbox-master"}
null ID: pmtk3-master	{_id: "pmtk3-master", _rev: "1-000badc78267997b463a84db81abb566", type: "kdm:Segment", name: "pmtk3-master"}
null ID: robotics-toolbox-matlab-master	{_id: "robotics-toolbox-matlab-master", _rev: "1-f702a86ee6aa70212d7d7a2df2653d12", type: "kdm:Segment", name: "robotics-toolbox-matlab-master"}

Showing 1-10 of 10 rows ← Previous Page | Rows per page: 10 | Next Page →

Essa funcionalidade de view do CouchDB foi utilizada para percorrer o conjunto de documentos que formam a estrutura KDM de um determinado projeto e montar um arquivo Json com as informações necessárias para visualização. Esse processo de transformar os dados originais em um novo arquivo no formato necessário para visualização é ilustrado na parte C da infraestrutura descrita na figura 19.

As partes D e E da figura 19 representam a ferramenta de visualização desenvolvida na linguagem Java através de uma arquitetura baseada no modelo cliente/servidor. Essa aplicação disponibiliza suas funcionalidades através de páginas html ilustradas na parte E da figura 19, que podem ser facilmente acessadas através de navegadores web, seja por

computadores ou por dispositivos móveis ilustrados na parte F da figura 19. Para definir o layout e design das páginas, foram utilizadas folhas de estilos do CSS3.

O lado cliente do ambiente de visualização representado na parte E da figura 19 corresponde justamente a junção das páginas html com suas funções Java Script e folhas de estilo. É nessa etapa que são exibidas as metáforas visuais com os dados dos programas MATLAB/Octave. As metáforas visuais disponíveis na infraestrutura usam a biblioteca D3.js. D3.js é uma biblioteca Java Script de código fonte aberto que fornece recursos para manipular documentos HTML, usando Java Script como a linguagem para implementar o mapeamento de dados para os documentos (HEYDT, 2015).

As visualizações D3 são implementadas em puro HTML e CSS, melhorando a compatibilidade do navegador (BOSTOCK; OGIEVETSKY; HEER, 2011). Um dos benefícios do D3 é como ele expõe o designer diretamente para a página da Web sem ter que usar ou criar uma nova linguagem de plotagem (AMR; STAMBOLIYSKA, 2016). A biblioteca D3 permite que os dados sejam ligados à um Document Object Model (DOM) que são originalmente convertidos na extremidade do navegador a partir de arquivos HTML enviados por HTTP. DOM é, de fato, uma estrutura em árvore, representada por nós hierárquicos onde o nó superior é a página principal ou o documento que pode ser adaptado pelo Java Script. A biblioteca D3 visa permitir a manipulação de documentos baseados em dados, expondo os recursos de padrões da web como HTML, SVG e CSS (AMR; STAMBOLIYSKA, 2016).

A primeira metáfora visual indicada para ser usada no ambiente proposto é a Treemap. Essa metáfora visual é capaz de representar grandes volumes de dados hierárquicos através de retângulos recursivamente aninhados representando entidades relevantes do domínio também conhecidas como atributos reais. Exemplos desses atributos são o número de linhas de código fonte, os arquivos de código-fonte, suas respectivas funções, entre outros atributos reais que podem ser visualmente representados por atributos visuais como retângulo, cor e tamanho (SHNEIDERMAN, 1992).

A segunda metáfora visual indicada para ser usada no ambiente proposto é a chord. A visão chord é capaz de exibir um conjunto de arcos formando um círculo e um conjunto de linhas conectando cada um desses arcos para representar suas relações. O tamanho dos arcos no círculo varia de acordo com o número de conexões que tem com os outros arcos da circunferência. Enquanto uma pequena quantidade de dados poderia ser representada em um diagrama circular usando linhas retas para mostrar as interconexões, um diagrama com

inúmeras linhas rapidamente se tornaria ilegível. Para reduzir a complexidade visual, os diagramas chord empregam uma técnica chamada agrupamento hierárquico de bordas (HOLTEN, 2006).

A visão chord foi mapeada para conter em cada arco da circunferência um arquivo de código-fonte de um projeto MATLAB/Octave e as linhas que conectam os arcos representam os acoplamentos aferentes e eferentes entre cada um desses arquivos.

O lado servidor da ferramenta, representado na parte D da figura 19, é constituído por classes Java que são responsáveis por conter as regras de negócio da aplicação, o acesso ao banco de dados CouchDB e a disponibilização dos serviços de acesso aos dados. Também é responsabilidade da aplicação servidora utilizar a biblioteca ASTOctaveToKDM para gerar o metamodelo KDM dos programas MATLAB/Octave e armazená-lo no banco de dados.

No controlador da aplicação servidora foram desenvolvidos métodos responsáveis por percorrer todos os elementos contidos na estrutura KDM criada e convertê-los em documentos Json, que posteriormente foram salvos no CouchDB.

Depois de fazer o mapeamento dos elementos KDM no documento Json correspondente, foi desenvolvida uma funcionalidade para armazená-lo no banco através do uso de uma biblioteca Java chamada Lightcouch. Essa biblioteca corresponde a uma Application Programming Interface (API) Java para comunicação com o banco de dados CouchDB.

Para configurar as informações do banco que a biblioteca vai se comunicar, basta criar um arquivo chamado couchdb.properties dentro da pasta do projeto. Esse arquivo permite a inclusão de configurações do banco através de um modelo chave - valor. Nesse arquivo é possível configurar o nome do banco, o protocolo de comunicação utilizado, o endereço IP da máquina em que o banco está instalado, a porta de comunicação com o servidor, o usuário de autenticação e a senha do usuário correspondente. A figura 27 possui um exemplo de configuração desse arquivo.

Além de converter os elementos, que representam o código fonte de aplicações MATLAB/Octave, em documentos Json e salvá-los no CouchDB, foi necessário criar mecanismos para recuperar os dados armazenados. Esses dados devem ser recuperados no formato necessário para alimentar as representações visuais representadas na parte E da figura 19.

Figura 27 - Exemplo de um Arquivo couchdb.properties

```
couchdb.name=kdmoctave
couchdb.createdb.if-not-exist=true
couchdb.protocol=http
couchdb.host=127.0.0.1
couchdb.port=5984
couchdb.username=admin
couchdb.password=admin
```

A comunicação entre o lado cliente da aplicação e o lado servidor é feita através de chamadas por parte das funções Java Script a serviços disponibilizados no lado servidor através da tecnologia REST. Rest é um estilo arquitetural que permite o desenvolvimento de serviços que podem ser acessados através de chamadas via url.

A utilização da arquitetura SOA vai permitir que futuramente novas aplicações clientes sejam criadas sem a necessidade de modificar a aplicação servidora. Dessa forma, será possível desenvolver representações visuais utilizando recursos nativos de dispositivos móveis ou ainda fazendo uso de recursos de aplicações Desktop.

5.2 DETALHES DE IMPLEMENTAÇÃO DO AMBIENTE DE VISUALIZAÇÃO

A implementação do ambiente de visualização proposto foi dividida em duas partes. Primeiro, foi implementado o lado servidor da ferramenta e depois o lado cliente. Para tanto, foi necessário apenas que os estudos de compreensão de aplicações MATLAB/Octave estivessem bem definidos.

Conforme mencionado anteriormente, o objetivo de compreensão de programas MATLAB/Octave deste trabalho, consiste em descobrir arquivos que possuam um número elevado de responsabilidades, caracterizando uma anormalidade que na orientação à objetos é conhecida como God Class. Para tanto, é preciso descobrir funções e arquivos que possuam muitas linhas de código e alto grau de acoplamento aferente e eferente. Para apoiar esse estudo, foram selecionadas duas representações visuais: treemap e chord.

A visão treemap foi utilizada para representar visualmente os arquivos de código fonte com suas respectivas funções e tamanho de cada um deles. A visão chord foi utilizada para representar os arquivos de código fonte com os relacionamentos entre cada um deles.

As duas visões foram selecionadas a partir das visões disponibilizadas pela biblioteca D3js. Para montar as visões na página HTML, a biblioteca D3js precisa que os dados que vão carregar as visões estejam armazenados no formato Json. As figuras 28 e 29 mostram como

devem estar organizados os dados dentro do Json para que as visões treemap e chord sejam carregadas corretamente.

Figura 28 - Exemplo de um Arquivo Json Contendo os Dados para Alimentar a Visão Treemap

```
{"name":"calculadora","children":
  [
    {"name":"calculos.m","children":[{"name":"soma","size":3}]},
    {"name":"calculos2.m","children":[{"name":"soma","size":2}]},
    {"name":"calculos3.m","children":[{"name":"soma","size":1},{"name":"subtracao","size":3}]}
  ]
}
```

Figura 29 - Exemplo de um Arquivo Json Contendo os Dados para Alimentar a Visão Chord

```
[
  {"name":"arquivo1.m","size":2,"imports":["arquivo2.m","arquivo3.m"]},
  {"name":"arquivo2.m","size":1,"imports":["arquivo1.m"]},
  {"name":"arquivo3.m","size":0,"imports":[]}
]
```

O arquivo Json utilizado para alimentar a visão treemap possui um atributo chamado name responsável por armazenar o nome do projeto que será analisado e um outro chamado children responsável por conter uma lista de objetos Json que são hierarquicamente filhos do objeto que armazena o nome do projeto. Esses objetos filhos correspondem aos arquivos de código fonte do projeto. Cada um desses objetos também possuem um atributo name com o nome do arquivo de código fonte e um atributo children que contém uma lista de objetos Json. Os objetos Json armazenados nessa lista correspondem as funções existentes dentro de cada arquivo de código fonte. Cada um desses objetos possue o atributo name contendo o nome da função e o atributo size contendo a quantidade de linhas de código fonte da função.

Os nomes dos arquivos de código fonte, os nomes das funções dentro de cada arquivo e o tamanho de cada função são os atributos reais que serão mapeados em atributos visuais. As funções serão desenhadas como retângulos. Todos os retângulos que representam funções dentro de um mesmo arquivo serão pintados com a mesma cor. A cor utilizada é definida aleatoriamente pela biblioteca D3. Os tamanhos dos retângulos variam de acordo com a quantidade de linhas de código fonte de cada função. Quanto mais linhas uma função possuir, maior será o retângulo que a representa. Todos os retângulos que representam funções de um mesmo arquivo são agrupados de modo a formar um retângulo maior responsável por representar o próprio arquivo de código fonte.

O arquivo json utilizado para alimentar a visão chord possui um atributo chamado name responsável por armazenar o nome de um arquivo de código fonte, o atributo size que

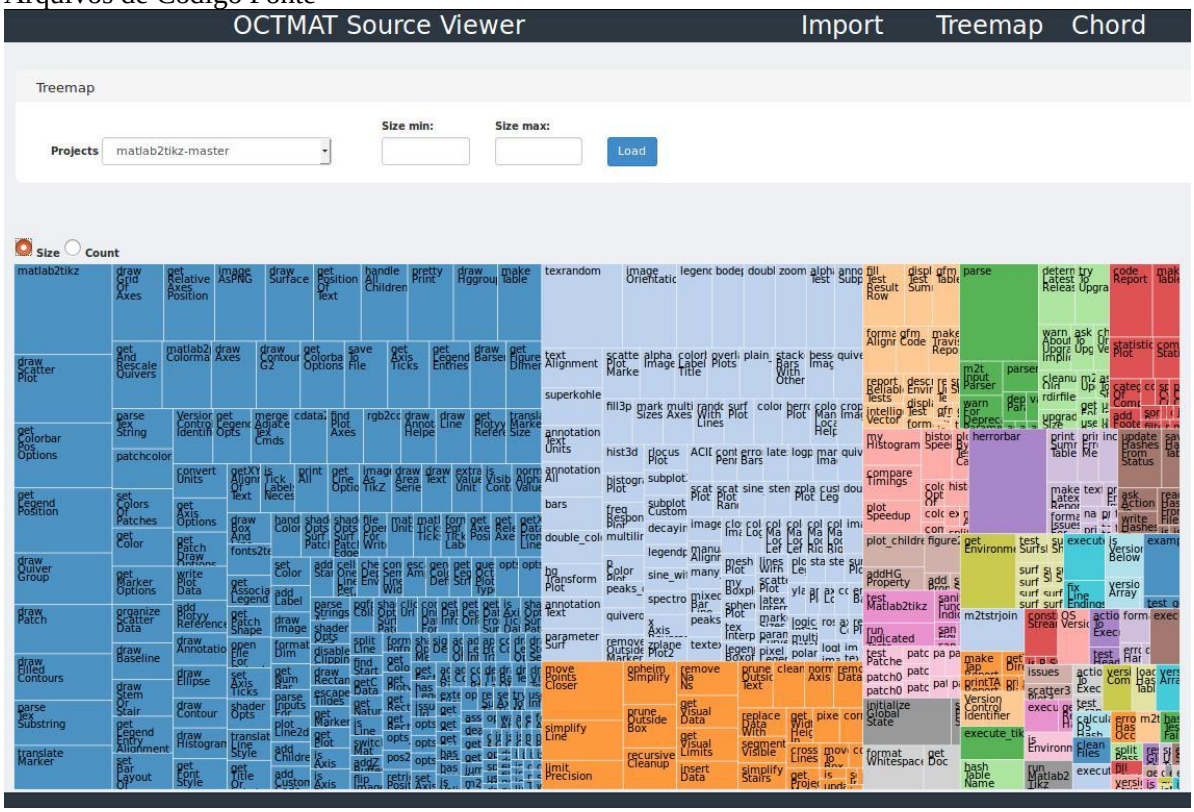
armazena a quantidade de arquivos com quem esse arquivo possui ligações aferentes ou eferentes e o atributo imports que corresponde à uma lista de nomes. Esses nomes correspondem aos nomes dos arquivos com que esse arquivo possuem ligações aferentes ou eferentes.

Esses três atributos reais são convertidos em atributos visuais. Os atributos name e imports são utilizados para montar os arcos da visão chord e as ligações entre cada um desses arcos. O atributo size é utilizado para definir o tamanho do arco dentro da circunferência. Quanto mais ligações o arquivo possuir, maior será o seu arco dentro da circunferência. As cores utilizadas para pintar os arcos e as ligações entre eles é definida aleatoriamente pela biblioteca D3.

Foram desenvolvidas seis páginas HTML contendo as representações visuais treemap e chord. A primeira contém a representação visual treemap. Nessa página foi incluído um filtro para permitir a redução do volume de dados a ser exibido graficamente. Esse filtro é formado por dois campos texto. No primeiro campo deve ser digitado um valor mínimo de linhas de código e no segundo um valor máximo. Só serão exibidos os nomes das funções e dos arquivos cujo arquivo de código fonte possua mais linhas de código do que o valor mínimo digitado e menos linhas de código do que o valor máximo digitado. O usuário do ambiente de visualização poderá informar apenas o valor mínimo de linhas de código, apenas o valor máximo, os dois valores ou nenhum dos dois valores. Além desses campos de texto, a tela possui uma lista no formato Combobox para que o usuário possa selecionar o projeto a ser analisado. Essa tela possui também um campo do tipo Radio Box do HTML utilizado para exibir o treemap pelo tamanho ou pelo número de arquivos. Se for escolhida a opção de tamanho, o treemap enfatizará os arquivos que possuem maior quantidade de linhas de código fonte. Se for escolhida a opção de quantidade, a visão treemap dará ênfase aos arquivos que possuem maior quantidade de funções. As figuras 30 e 31 ilustram o funcionamento dessa tela através da escolha dos campos tamanho e quantidade respectivamente.

Foram desenvolvidas duas telas para representação visual do Chord. Ambas as telas possuem os mesmos filtros da tela do treemap. Foram necessárias duas telas porque cada uma delas representa as relações entre os arquivos em um sentido específico. Uma tela exibe os arquivos de código fonte com as relações entre cada um deles e os demais arquivos dos quais eles dependem. Essas ligações são chamadas de eferentes.

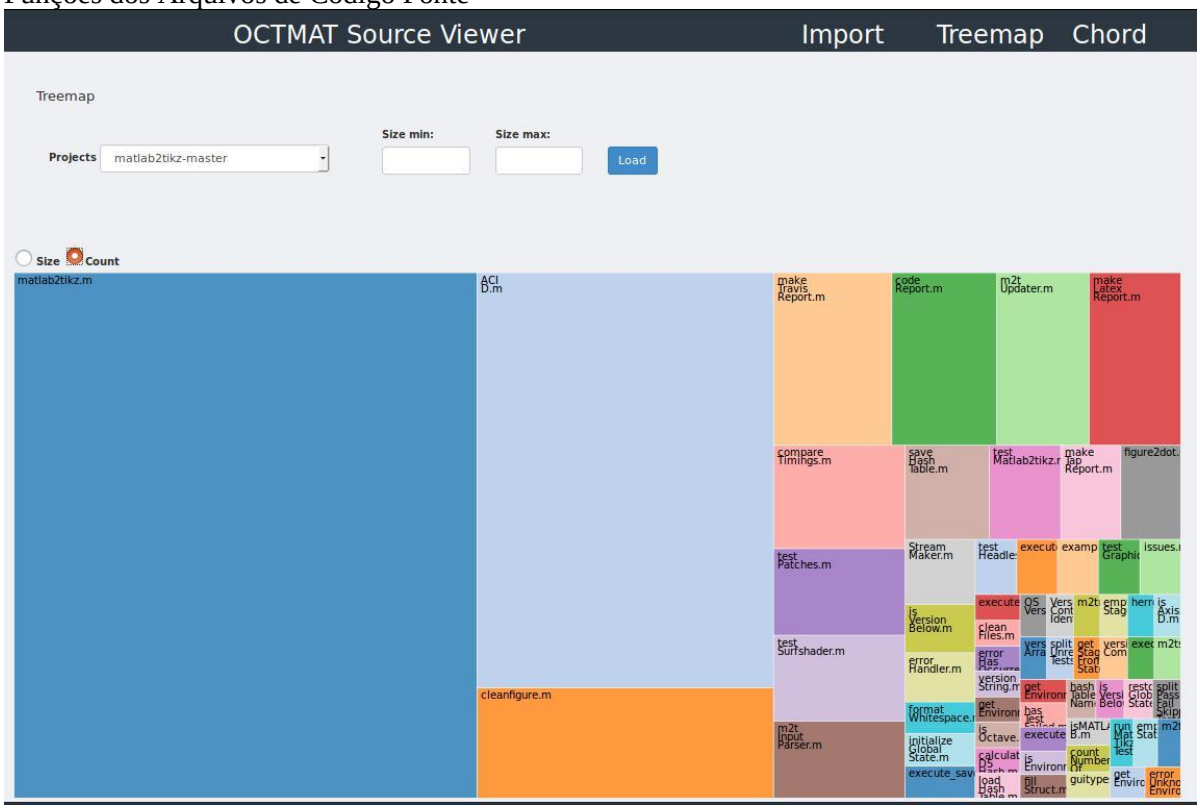
Figura 30 - Tela Treemap do Ambiente de Visualização Fazendo o Agrupamento pelo Tamanho dos Arquivos de Código Fonte



A outra tela exibe os arquivos de código fonte com as relações entre cada um deles e os demais arquivos que dependem dele. Essas relações são chamadas de aferentes. Enquanto no treemap os campos de texto foram utilizados para informar os tamanhos mínimo e máximo em número de linhas de código fonte, na visão chord, esses campos servem para identificar as quantidades mínima e máxima de relacionamentos entre cada arquivo de código fonte. A figura 32 ilustra o funcionamento dessas duas telas.

Um outro caminho utilizado para minimizar o volume de dados apresentado no chord, foi a inclusão de uma nova tela de representação treemap. Nessa tela, os tamanhos dos retângulos que formam a visão, ao invés de variarem de acordo com a quantidade de linhas de cada arquivo, variam de acordo com a quantidade de relações existentes em cada arquivo. Nessa implementação, foi feita uma alteração no treemap para incluir uma ação de clique do mouse na região que representa um arquivo de código fonte. Ao clicar em uma dessas regiões, o usuário é redirecionado para a última tela construída, que representa o chord, porém exibindo apenas o arquivo contido na região clicada com seus respectivos relacionamentos.

Figura 31 - Tela Treemap do Ambiente de Visualização Fazendo o Agrupamento pela Quantidade de Funções dos Arquivos de Código Fonte



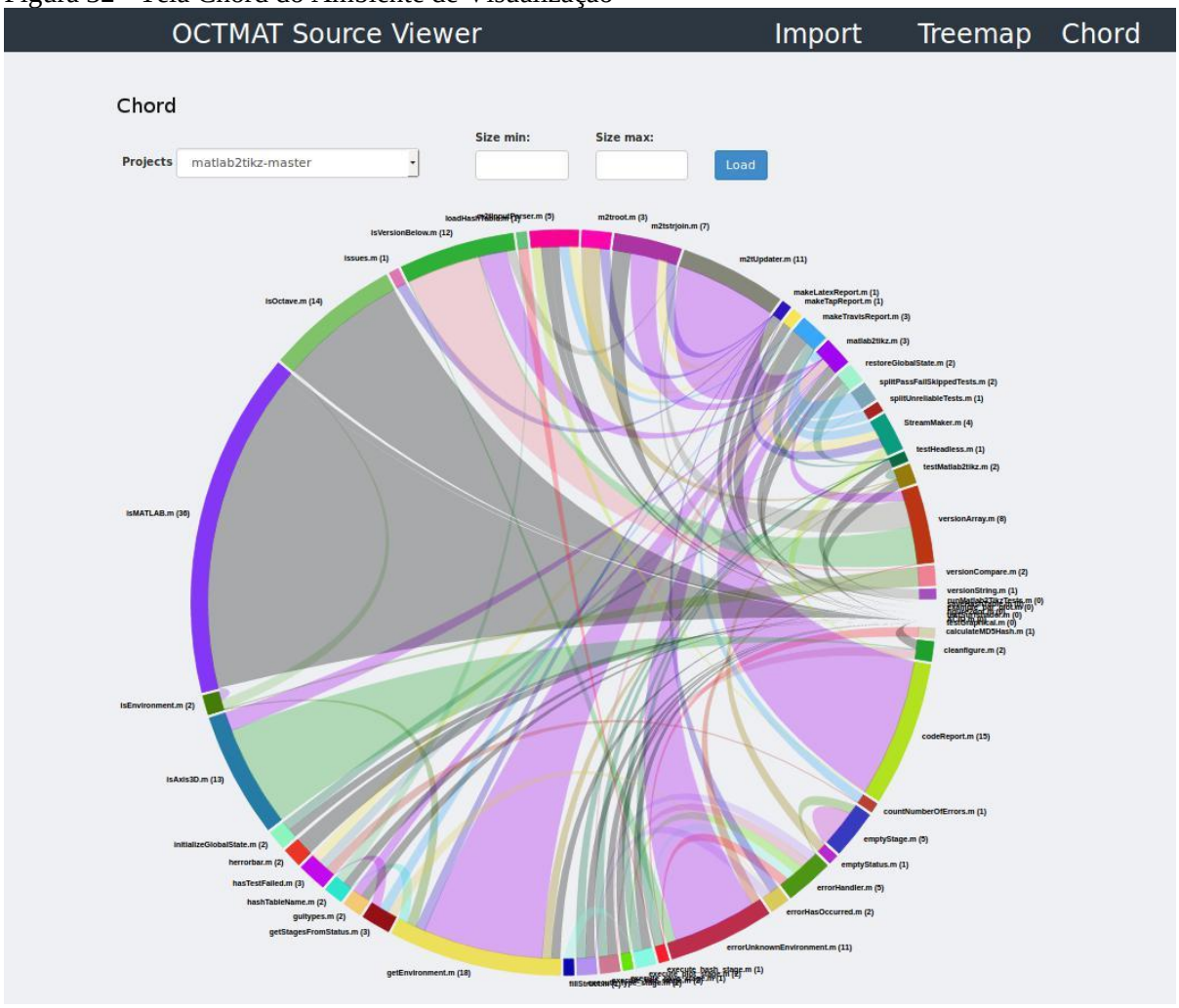
Existem duas telas com essa visão treemap. Uma representa os arquivos de código fonte, com a quantidade de relações eferentes, enquanto a outra representa os arquivos de código fonte com a quantidade de relações aferentes. As figuras 33 e 34 ilustram essas duas telas.

Além dessas seis, foram desenvolvidas mais duas outras telas. Uma corresponde a página inicial do sistema e a outra corresponde a uma tela utilizada para importar um determinado projeto para dentro do ambiente de visualização. A tela inicial possui uma lista em forma de tabela, que exibe todos os projetos importados, com as quantidades de arquivos de código fonte de cada projeto mais a quantidade de funções existentes. A figura 35 ilustra a tela inicial da ferramenta de visualização.

A tela que contempla a funcionalidade de importação de projetos possui apenas dois campos texto. O primeiro deve ser utilizado para informar o nome do projeto, enquanto

que o segundo deve ser utilizado para informar a url do projeto no Github. Ambos os campos possuem preenchimento obrigatório para que a importação seja realizada. A figura 36 ilustra a tela de importação dos projetos. Todas as oito telas possuem um menu através do qual é possível navegar pelo sistema.

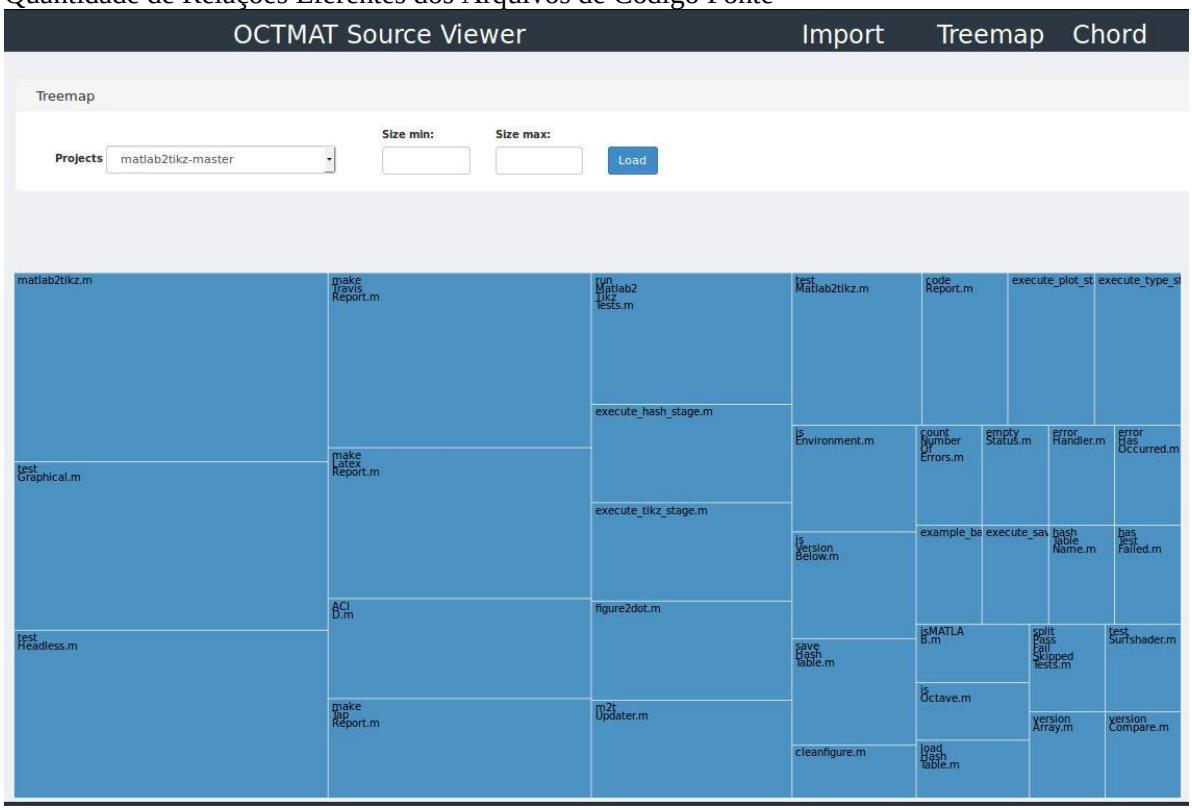
Figura 32 - Tela Chord do Ambiente de Visualização



Todas as telas se conectam a serviços disponíveis na parte servidora da aplicação, para executar as devidas funcionalidades. Essa comunicação ocorre através de chamadas por url. Isso é possível pelo fato de ter sido usada a tecnologia REST na construção dos serviços.

Existe apenas uma classe de serviços criada na parte servidora do sistema. Essa classe disponibiliza um conjunto de nove métodos. Desses nove, oito devem ser invocados utilizando uma requisição do tipo GET e apenas um deve ser invocado utilizando uma requisição do tipo POST. O método que utiliza a requisição POST tem a funcionalidade de importar um determinado projeto MATLAB/Octave para dentro do sistema.

Figura 33 - Tela Treemap do Ambiente de Visualização Fazendo o Agrupamento pela Quantidade de Relações Eferentes dos Arquivos de Código Fonte



Dos oito métodos restantes, seis são utilizados para retornar os dados que alimentarão cada uma das visões disponibilizadas no sistema. Um método retorna um documento Json contendo uma lista com os nomes dos projetos importados para alimentar o campo combobox das páginas HTML e o último método citado retorna um documento json contendo uma lista com os nomes dos projetos importados para o sistema, a quantidade de arquivos de código fonte existentes em cada projeto e a quantidade de funções. Esse arquivo é utilizado para montar a página inicial do ambiente de visualização.

De acordo com a especificação do REST, a classe de serviços recebeu anotação com o nome path. Essa anotação recebe como parâmetro a String "/kdm". Com essa anotação, todos os métodos da classe que disponibilizam algum tipo de serviço, deverá ter como parte de sua url o caminho "/kdm". Os métodos também receberam a mesma anotação, porém cada método teve uma String diferente passada como parâmetro. Por exemplo, a string passada como parâmetro à anotação path do método que retorna uma lista com os nomes dos projetos foi "/allProjects". Isso significa dizer que, para invocar esse método, basta fazer uma requisição http através da url: < http://endereço IP do servidor : porta do servidor / nome da aplicação / kdm / allProjects >.

Figura 34 - Tela Treemap do Ambiente de Visualização Fazendo o Agrupamento pela Quantidade de Relações Aferentes dos Arquivos de Código Fonte

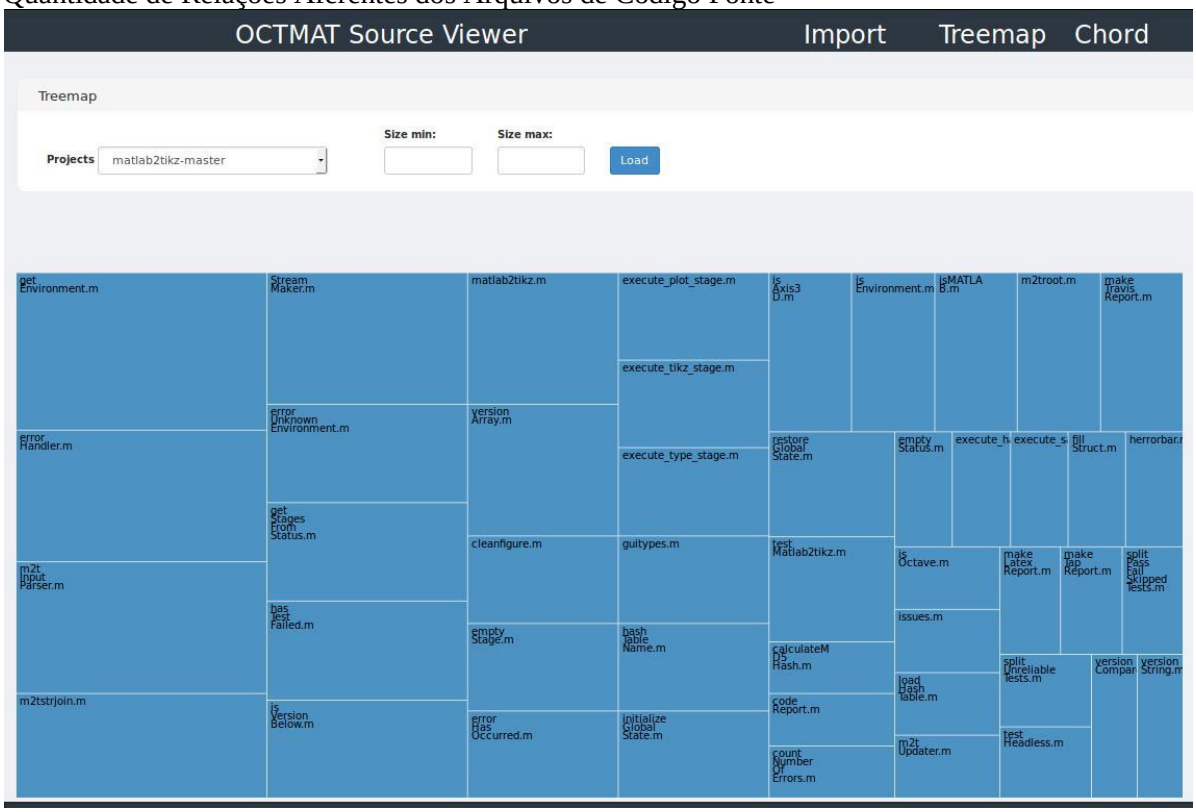
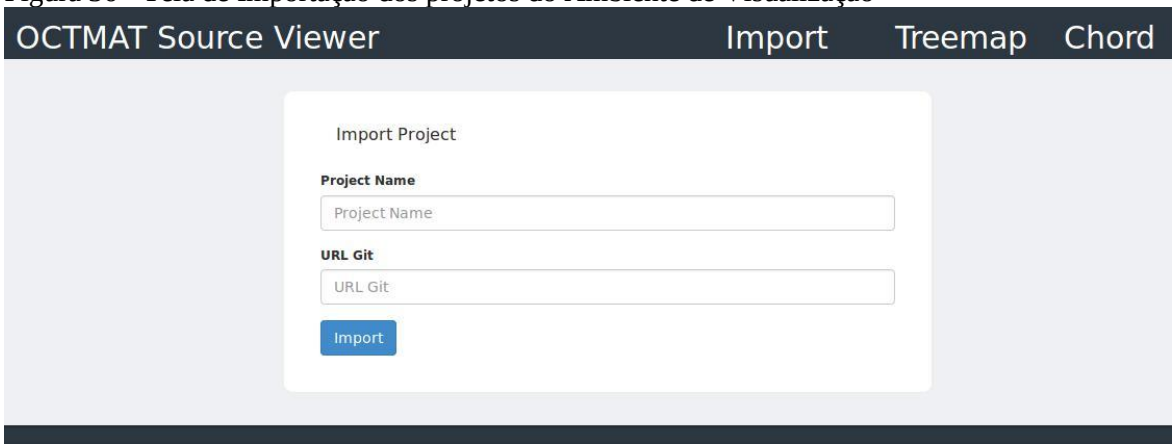


Figura 35 - Tela Inicial do Ambiente de Visualização

Foi implementada também uma classe de negócio responsável por executar cada funcionalidade do sistema. Cada método disponível na classe de serviços, faz uma chamada a um determinado método da classe de negócio, que implementará as ações necessárias para a execução da funcionalidade.

Figura 36 - Tela de Importação dos projetos do Ambiente de Visualização



A última classe implementada, mas não menos importante, foi a de persistência, que tem a responsabilidade de fazer a comunicação com o banco de dados através do uso da biblioteca Lightcouch. Essa classe possui métodos que são invocados pelos métodos disponíveis na classe de negócio.

Para retornar os dados solicitados pelos serviços disponíveis na aplicação, foi desenvolvido um conjunto de views dentro do CouchDB. A solicitação de execução dessas views é feita pela classe de persistência que, por sua vez, retorna esses dados à classe de negócio. Os métodos disponíveis na classe de negócio faz o devido tratamento desses dados e em seguida retorna-os à classe de serviços já no formato esperado pelas páginas HTML.

Uma funcionalidade que merece destaque é a de importação do projeto para dentro do sistema. Quando o usuário do sistema informa na tela de importação o nome do projeto e a sua respectiva url git, esses dados são passados como parâmetro ao serviço de importação, que por sua vez, repassa ao método importProject da classe de negócio. Para efetivar a importação, esse método faz um clone do projeto dentro de uma pasta no servidor. Para tanto, ele utiliza uma biblioteca java chamada Jgit. Depois que o projeto é clonado, o método importProject utiliza a biblioteca ASTOctaveToKDM para gerar a estrutura KDM do projeto. Em seguida, essa estrutura é percorrida e os elementos encontrados são convertidos nos respectivos documentos json e salvos no CouchDB. Esse processo pode demorar um tempo significativo

dependendo da velocidade de conexão com a internet e do tamanho do projeto. Após a importação, o diretório do projeto é apagado do servidor.

5.3 CONCLUSÃO DO CAPÍTULO

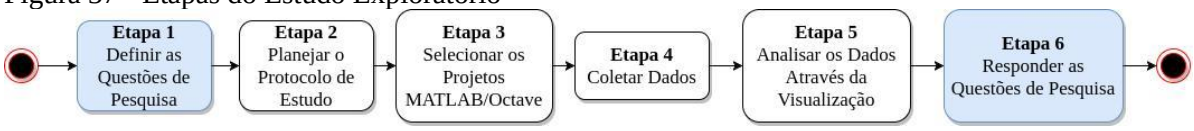
Este capítulo descreveu o ambiente de visualização proposto, exibiu e detalhou a arquitetura utilizada acompanhada de suas características de implementação.

6 ESTUDO EXPLORATÓRIO DO AMBIENTE DE VISUALIZAÇÃO PARA ATIVIDADES DE COMPREENSÃO

Neste capítulo será apresentado um estudo exploratório do ambiente de visualização proposto neste trabalho para constatar como ele pode apoiar a compreensão de programas MATLAB/Octave. Para tanto, foram elaboradas algumas questões de pesquisa do estudo e uma estratégia de como utilizar o ambiente de visualização para responder as questões de pesquisa elaboradas. Foi elaborado também um protocolo contendo as etapas necessárias à execução do estudo.

A escolha por este estudo exploratório se deu devido ao fato de a literatura relatar a ocorrência de módulos de uma aplicação que tendem a concentrar um grande número de funcionalidades e consequentemente um conjunto considerável de informações. Como resultado, verifica-se que este módulo tende a ser muito chamado e/ou referenciado pelos demais. Em orientação a objetos, esta anomalia é conhecida como God Class e traz grandes prejuízos na manutenibilidade da aplicação. Em programação procedural, esta anomalia também pode trazer problemas em relação à ameaça da modularidade da aplicação, do auto acoplamento entre os módulos que a compõem e consequentemente a necessidade de maior esforço para manutenção. A figura 37 mostra as etapas executadas neste estudo exploratório.

Figura 37 - Etapas do Estudo Exploratório



6.1 QUESTÕES DE PESQUISA PARA O ESTUDO EXPLORATÓRIO

Nesta seção pretende-se abordar a seguinte Questão de Pesquisa para o estudo exploratório: QEX Até que ponto a infraestrutura visual proposta suporta a identificação de arquivos de um determinado projeto MATLAB/Octave que tem características semelhantes às God Class do Paradigma Orientado a Objetos? Para viabilizar a análise desta pergunta no paradigma procedural, a pergunta foi derivada em outras quatro perguntas:

QEX1-Quais são os cinco arquivos que tem maior número de linhas de código fonte em um determinado projeto? As classes consideradas como God Class no paradigma orientado à objetos normalmente possuem um número exacerbado de linhas de código fonte. Isso ocorre porque essas classes acabam por assumir responsabilidades, através da construção

de métodos e funções, que não lhes pertencem. Acredita-se que o mesmo ocorre com os arquivos MATLAB/Octave.

QEX2-Quais os cinco arquivos com maior número de funções em um determinado projeto? Arquivos que têm um número elevado de funções pode sugerir que eles possuem também um número elevado de responsabilidades. Essa característica pode ser um indício de que o projeto não está bem modularizado, o que também pode ser um problema para a manutenibilidade do sistema.

QEX3-Quais os cinco arquivos mais referenciados em um determinado projeto? Uma God Class normalmente possui um número elevado de linhas de código e de sub-rotinas. Isso significa dizer que essas classes provavelmente possuem um nível de acoplamento também elevado, pois suas sub-rotinas tendem a ser utilizadas por vários outros arquivos do projeto.

QEX4-Quais os cinco arquivos que mais fazem referências a funções de outros arquivos em um determinado projeto? Ainda é possível que as funções desenvolvidas em uma God Class não sejam muito utilizadas pelo próprio projeto. Isso porque existem muitos casos de God Class criadas para serem utilizadas diretamente pelo usuário do sistema. Ao invés de modularizarem as funcionalidades disponibilizadas aos usuários do sistema em vários arquivos/funções, os desenvolvedores concentram tudo em um único arquivo. Nesse caso, uma outra análise que pode ser feita está relacionada ao nível de acoplamento desses arquivos no sentido inverso. Com a quantidade de arquivos do qual eles dependem.

6.2 O PROTOCOLO DE ESTUDO

Esse estudo foi realizado pelo próprio autor desta dissertação, que é formado em Ciência da Computação, possui experiência no desenvolvimento de programas através da linguagem Java, e está atualmente cursando o mestrado em Sistemas e Computação.

Para executá-lo, o autor estabeleceu os critérios para seleção dos projetos que seriam submetidos ao estudo, selecionou dez projetos e aplicou o parser proposto em cada um deles, salvando suas informações no banco de dados CouchDB. Em seguida, ele desenvolveu consultas no banco de dados, necessárias para responder as questões de pesquisa estabelecidas e utilizou o retorno dessas consultas para alimentar representações visuais disponibilizadas através da biblioteca Java Script D3js.

6.3 SELEÇÃO DE PROJETOS

Para executar o estudo, inicialmente foi preciso selecionar um repositório público que disponibilizasse aplicações desenvolvidas nas linguagens MATLAB e Octave. O repositório escolhido foi o Github. Essa escolha se deu principalmente pela quantidade de projetos MATLAB e Octave disponíveis. Além disso, o Github é um repositório público mundialmente conhecido, utilizado e muito respeitado pelos desenvolvedores de software livre.

Após a escolha pelo repositório Github, foi feita uma pesquisa para encontrar aplicações desenvolvidas nas linguagens MATLAB e Octave. Primeiro foi executada uma busca digitando a palavra Octave no campo de pesquisa da ferramenta. O resultado da busca foi ordenado pelos projetos que tivessem mais forks. Após a ordenação, foi aplicado um filtro para selecionar apenas os projetos que estivessem definidos como sendo da linguagem MATLAB. Em seguida foram selecionados os projetos que tivessem pelo menos 3 contribuidores, um mínimo de vinte forks e a partir de dez arquivos fonte. O mesmo procedimento foi feito buscando a palavra MATLAB. É importante lembrar que só foram selecionados os projetos desenvolvidos em uma das duas linguagens. No total, foram selecionados dez projetos que serão mostrados a seguir. As características de cada um desses dez projetos que permitiram que eles fossem aceitos pelos critérios estabelecidos podem ser melhor visualizadas através da tabela 8. A coluna que informa a linguagem de desenvolvimento possui o valor MATLAB/Octave para os projetos cuja documentação descreve explicitamente que eles são compatíveis com as duas linguagens.

Tabela 8 - Critérios Utilizados para Seleção dos Projetos

Projetos	Contribuidores	Forks	Arquivos	Linguagem
FieldTrip	49	176	4884	MATLAB
PMTK3	17	474	1753	MATLAB/Octave
robotics-toolbox-matlab	3	22	455	MATLAB
GPstuff	7	21	428	MATLAB/Octave
HDR Toolbox	6	22	348	MATLAB/Octave
octave-networks-toolbox	3	47	126	Octave
MATLAB PLOTLY API WRAPPER	9	80	109	MATLAB
DeepLearntoolbox	12	1668	67	MATLAB/Octave
MATLAB2Tikz	26	122	64	MATLAB/Octave
Export_fig	7	110	16	MATLAB

O projeto PMTK3 está disponível no Github através da url <https://github.com/probml/pmtk3> e corresponde a um conjunto de funções MATLAB/Octave desenvolvidas para fornecer um quadro conceitual e de software unificado que englobe a aprendizagem de máquinas, modelos gráficos e estatísticas bayesianas. Esse projeto está estável e nenhuma nova funcionalidade foi adicionada desde dezembro de 2011. Foram feitas apenas correções de bugs. Ele possui 17 contribuidores, 474 forks e 1753 scripts.

O projeto DeepLearntoolbox é uma caixa de ferramentas para Deep Learning. Deep Learning é um novo subcampo da aprendizagem de máquina que se concentra em aprender profundos modelos hierárquicos de dados. É inspirado na arquitetura profunda do cérebro humano. Este projeto pode ser acessado através da url <https://github.com/rasmusbergpalm/DeepLearnToolbox>. Ele possui 12 contribuidores, 1668 forks e 67 scripts.

Export_fig é um conjunto de funções MATLAB que exportam satisfatoriamente figuras que são visualizadas na tela através do MATLAB para um documento da forma esperada pelo usuário de modo que esse documento possa ser facilmente publicado e compartilhado. É possível visualizar esse projeto através da url <https://github.com/altmany/export_fig>. Ele possui 7 contribuidores, 110 forks e 16 scripts.

O projeto MATLAB2Tikz foi concebido para converter figuras nativas do MATLAB, incluindo gráficos 2D e 3D em figuras TikZ/Pgfplots que se interagem perfeitamente em documentos LaTex. É possível saber mais sobre esse projeto e ter acesso ao seu código fonte através da url <https://github.com/matlab2tikz/matlab2tikz>. Ele possui 26 contribuidores, 122 forks e 64 scripts.

FieldTrip é uma ferramenta que possui algoritmos para análises básicas e avançadas de dados MEG(Magnetoencefalografia) e EGG(Eletroencefalografia). MEG e EGG são técnicas não invasivas eletrofisiológicas para registro da atividade cerebral. Este projeto está disponível através da url <https://github.com/fieldtrip/fieldtrip>. Ele possui 49 contribuidores, 176 forks e 4884 scripts.

O projeto GPstuff pode ser acessado através da ur <https://github.com/gpstuff-dev/gpstuff> e corresponde a um conjunto de funções que possui métodos de inferência, aproximações esparsas e métodos de avaliação de modelos utilizados em modelos de processos gaussianos. Ele possui 7 contribuidores, 21 forks e 428 scripts.

HDR Toolbox é um projeto MATLAB/Octave utilizado para processamento de conteúdo HDR (High Dynamic Range). O código fonte deste projeto está disponível no

Github através da url <https://github.com/banterle/HDR_Toolbox>. Ele possui 6 contribuidores, 22 forks e 348 scripts.

O projeto octave-networks-toolbox possui um conjunto de funções Octave para análises de gráficos/redes organizadas por funcionalidade. Ele está disponível através da url <https://github.com/aeolianine/octave-networks-toolbox>. Este projeto possui 3 contribuidores, 47 forks e 126 scripts.

MATLAB PLOTLY API WRAPPER é um projeto MATLAB também conhecido por MATLAB Online. Seu objetivo consiste em converter figuras MATLAB em gráficos Plotly online de uma forma muito simples e fácil. Ele está disponível através da url <https://github.com/plotly/MATLAB-Online>, possui 9 contribuidores, 80 forks e 109 scripts.

O projeto robotics-toolbox-matlab fornece algoritmos que podem ser utilizados no desenvolvimento de aplicações robóticas autônomas mobile. Nos algoritmos estão incluídos representação de mapa, planejamento de caminho e seguimento de caminho para robôs de unidade diferencial. Seu código fonte pode ser acessado através da url <https: //github.com/petercorke/robotics-toolbox-matlab>. Este projeto possui 3 contribuidores, 32 forks 455 scripts.

Todas as informações sobre os dez projetos selecionados foram coletadas no dia dezesseis de novembro de 2016 e podem sofrer alterações.

6.4 ESTRATÉGIA SUGERIDA PARA USO DO AMBIENTE

O objetivo da infraestrutura proposta na figura 19 é suportar a análise de códigos fonte de programas MATLAB e Octave seguindo a estratégia em níveis definida na figura 38 que será abordada nesta seção.

A estratégia de análise proposta nesta seção tem por objetivo orientar os estudos de compreensão dos programas MATLAB/Octave, definindo o passo a passo a ser executado na tentativa de descobrir nesses projetos, arquivos de código fonte que assumem muitas responsabilidades dentro do programa. Esses arquivos configuram uma anormalidade, que na orientação a objetos é conhecida como God Class.

A estratégia de análise divide o estudo em três fases. A primeira fase tem por objetivo tentar descobrir nos projetos MATLAB/Octave os arquivos com maior tamanho em número de linhas de código fonte e os arquivos que possuem maior número de funções. Para que essa identificação aconteça, foi utilizada uma visão hierárquica chamada treemap. Através dela, é possível visualmente identificar todos os arquivos de um determinado projeto com suas

respectivas funções. As funções são representadas através de retângulos que variam de tamanho de acordo com a quantidade de linhas de código fonte existente na função. Os retângulos que representam as funções de um mesmo arquivo de código fonte são pintados com a mesma cor.

A segunda fase analisa as dependências entre os arquivos de código fonte de um mesmo projeto e consequentemente seu grau de acoplamento, seja ele aferente ou eferente. Para visualizar esses relacionamentos, foi utilizada uma visão conhecida como Chord. Através dela é possível identificar todos os arquivos de código fonte de um determinado projeto e suas relações com os demais arquivos do mesmo projeto.

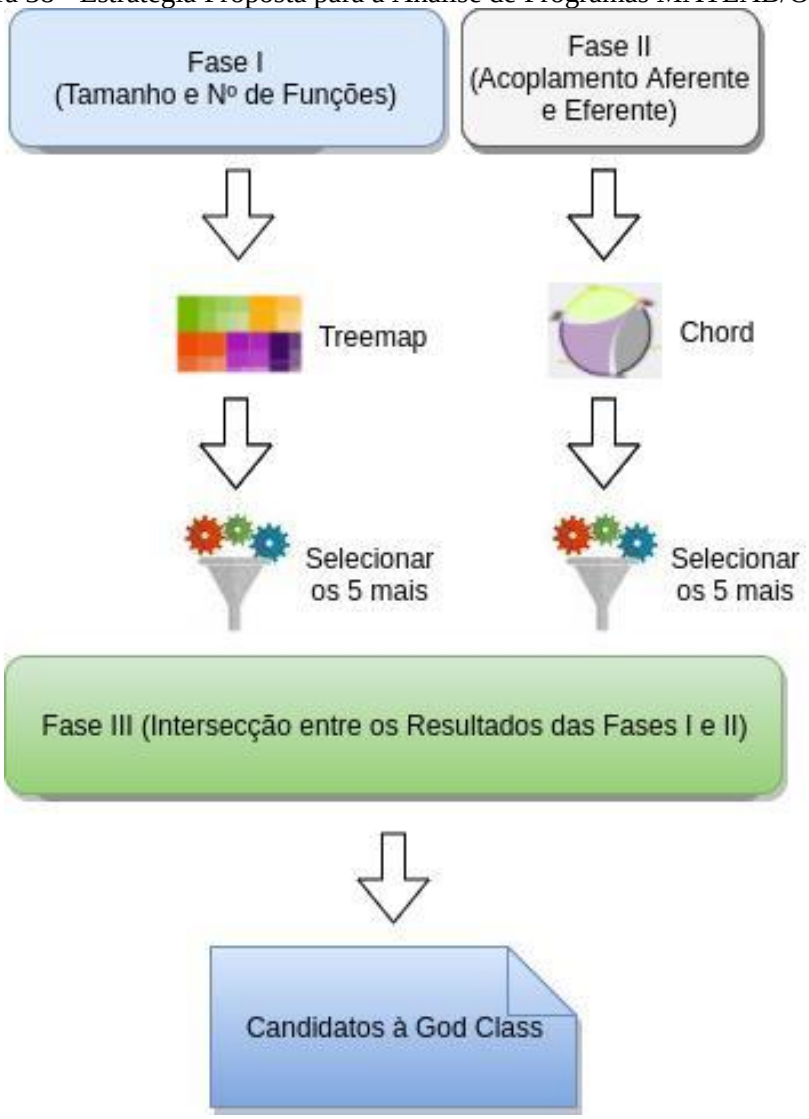
A terceira e última fase da estratégia proposta consiste em filtrar os cinco arquivos com maior número de funções obtidos a partir da fase 1, os cinco maiores arquivos do projeto em número de linhas de código fonte também obtidos a partir da fase 1, os cinco arquivos que possuem mais ligações aferentes obtidos a partir da fase 2 e os cinco arquivos que possuem mais ligações eferentes também obtidos a partir da fase 2. Em seguida é feita a intersecção dos dados obtidos nas fases I e II. Os arquivos obtidos a partir dessa intersecção são considerados pelo estudo como possíveis candidatos à God Class, e, portanto, merecem uma análise mais detalhada.

6.5 COLETA E ANÁLISE DOS DADOS

Para coletar as informações necessárias à obtenção das respostas para as quatro questões de pesquisa elaboradas no estudo exploratório, os dez projetos foram importados para o ambiente de visualização. Ao efetuar a importação, o ambiente de visualização utiliza a biblioteca ASTOctaveToKDM para gerar o KDM desses projetos, converter os elementos contidos na estrutura KDM em documentos Json e salvá-los no CouchDB.

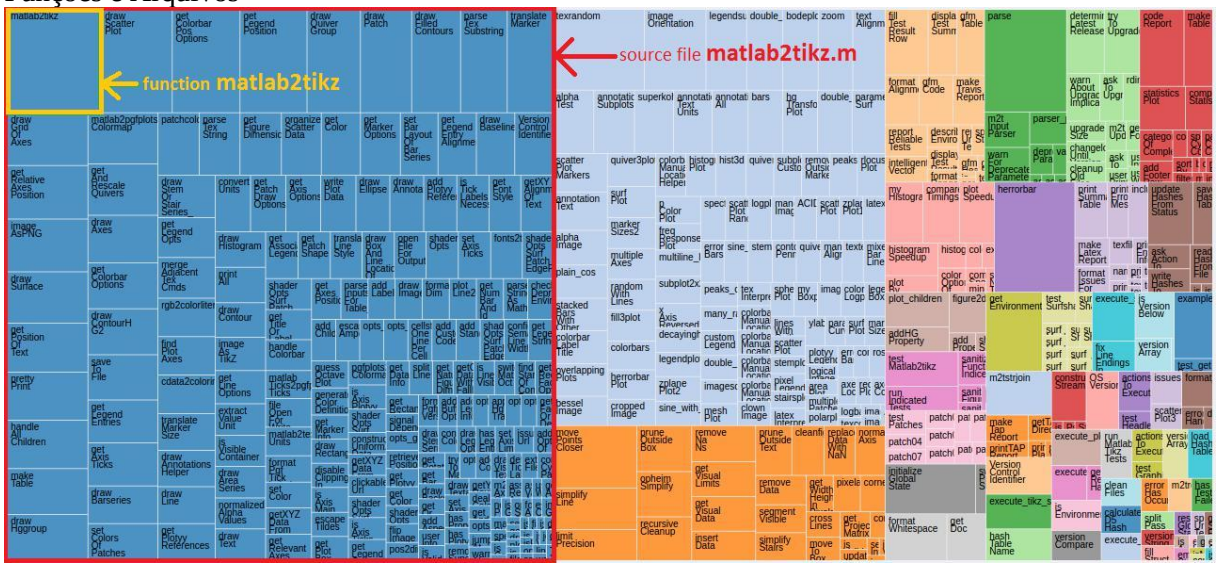
A partir dos dados armazenados no SGBD foram elaboradas algumas consultas de banco de dados. A primeira delas recebeu o nome de getAllFunctionsByFileProject e ela é capaz de retornar do CouchDB todos os arquivos de um determinado projeto com suas respectivas funções e o tamanho em número de linhas de código fonte de cada uma dessas funções. Através dessa consulta é possível responder as questões de pesquisa QEX1 e QEX2.

Figura 38 - Estratégia Proposta para a Análise de Programas MATLAB/Octave



Os dados retornados pela consulta getAllFunctionsByFileProject foram exibidos em uma visão Treemap através do uso de uma biblioteca JavaScript chamada D3js. Dois atributos de visualização foram utilizados nesta visão: o tamanho e a cor. Cada retângulo do treemap corresponde à uma função dentro do arquivo e o tamanho do retângulo varia de acordo com o tamanho da função. A cor utilizada para pintar os retângulos servem para identificar o arquivo do qual a função faz parte. Isso quer dizer que todas as funções de um mesmo arquivo serão pintadas no treemap com a mesma cor. Como pode ser constatado na figura 39, o matlab2tikz.m é o arquivo que possui o maior número de linhas de código fonte do projeto MATLAB2Tikz. São 3008 linhas de código fonte sendo que destas, 97 estão concentradas na função matlab2tikz, que corresponde à maior função em número de linhas do projeto.

Figura 39 - Visão Treemap do Projeto MATLAB2Tikz que Representa o Tamanho de suas Funções e Arquivos

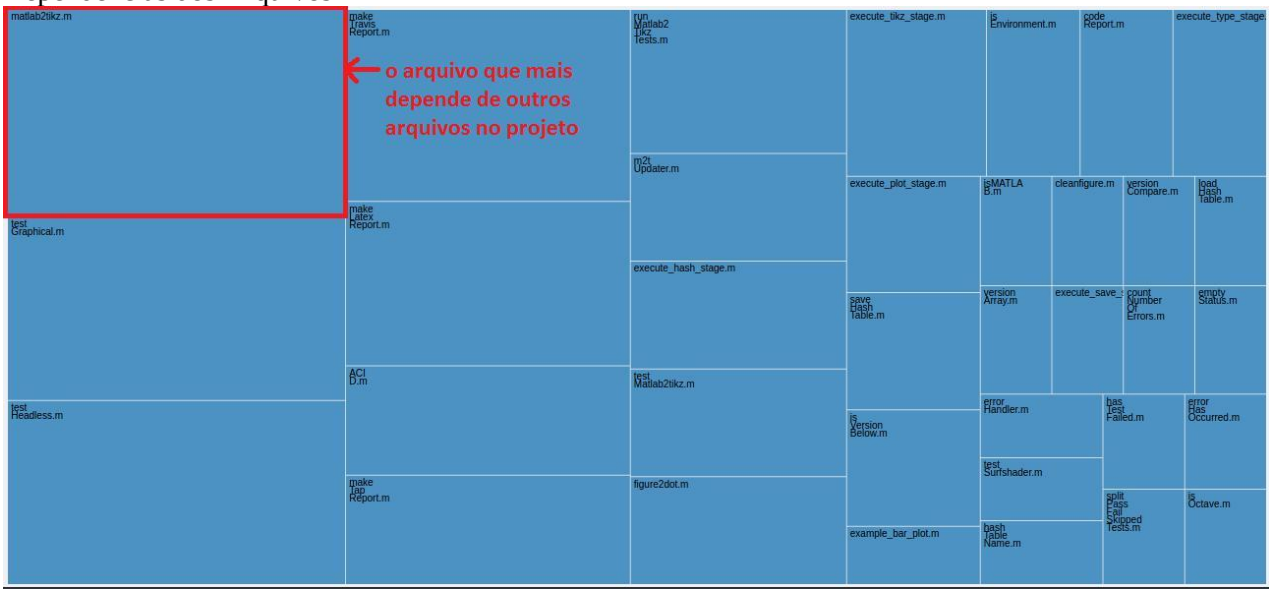


Foram desenvolvidas mais quatro consultas. Através delas é possível obter as respostas para as questões de pesquisa QEX3 e QEX4. A primeira é capaz de retornar do CouchDB todos os arquivos de um projeto e a quantidade de arquivos que dependem de cada um deles. Essa dependência acontece devido ao uso de uma de suas funções. A segunda consulta também traz do banco os nomes de todos os arquivos de um determinado projeto e a quantidade de arquivos dos quais eles dependem. Essa dependência se dá através do uso de funções localizadas em outros arquivos. Os dados retornados destas duas consultas também foram utilizados para alimentar visões treemap. Nesse caso, cada arquivo é representado no treemap por um retângulo que varia de tamanho de acordo com a quantidade de arquivos dependentes ou que eles dependem. Todos os retângulos são pintados com a mesma cor. Isso significa dizer que o atributo cor não tem relevância visual neste treemap.

A figura 40 mostra através de uma visão treemap que os arquivos mais utilizados dentro do projeto MATLAB2Tikz são getEnvironment.m e errorHandler.m. Cada um destes dois arquivos são utilizados por outros seis dentro do projeto MATLAB2Tikz. Na figura 41 é possível verificar que o arquivo que mais depende de outros arquivos dentro do projeto MATLAB2Tikz é o matlab2tikz.m. Ele utiliza funções localizadas em nove arquivos do projeto.

Figura 40 - Visão Treemap do Projeto MATLAB2Tikz que Representa a Quantidade de Relações dos Arquivos

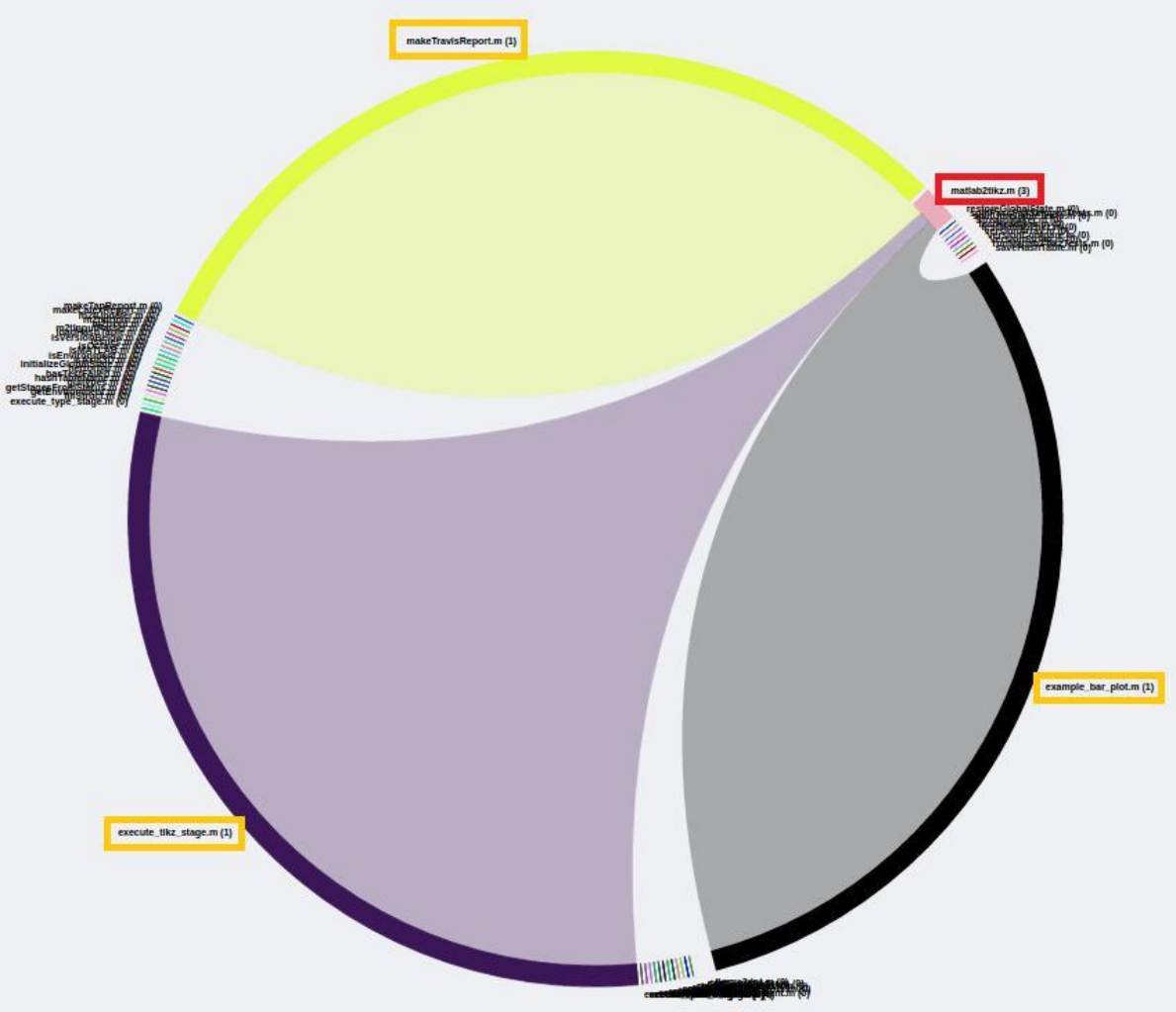
Figura 41 - Visão Treemap do Projeto MATLAB2Tikz que Representa a Quantidade de Dependências dos Arquivos



A terceira consulta desenvolvida busca no CouchDB todos os arquivos de um projeto e uma lista contendo os nomes dos arquivos que dependem dele. A quarta e última consulta criada traz do banco todos os arquivos de um projeto e uma lista contendo os nomes dos arquivos dos quais eles dependem. Os dados retornados do banco através dessas consultas serviram para alimentar uma visão de relacionamentos denominada Chord. Essa visão também foi desenvolvida através da biblioteca D3js. Através da figura 43 é possível verificar que o arquivo matlab2tikz.m, que está contornado em amarelo, depende de funções localizadas nos arquivos versionArray.m, m2tstrjoin.m, errorUnknownEnviron-ment.m, isAxis3d.m,

getEnvironment.m, guitypes.m, codeReport.m, isVersionBelow.m e m2tUpdater.m, que estão contornados em vermelho. Já na figura 42 pode-se constatar que os arquivos example_bar_plot.m, execute_tikz_stage.m e makeTravisReport.m, que estão contornados em amarelo, dependem das funções disponíveis no arquivo matlab2tikz.m, que está contornado em vermelho.

Figura 42 - Visão Chord do Projeto MATLAB2Tikz que Representa as Relações Entre os Arquivos

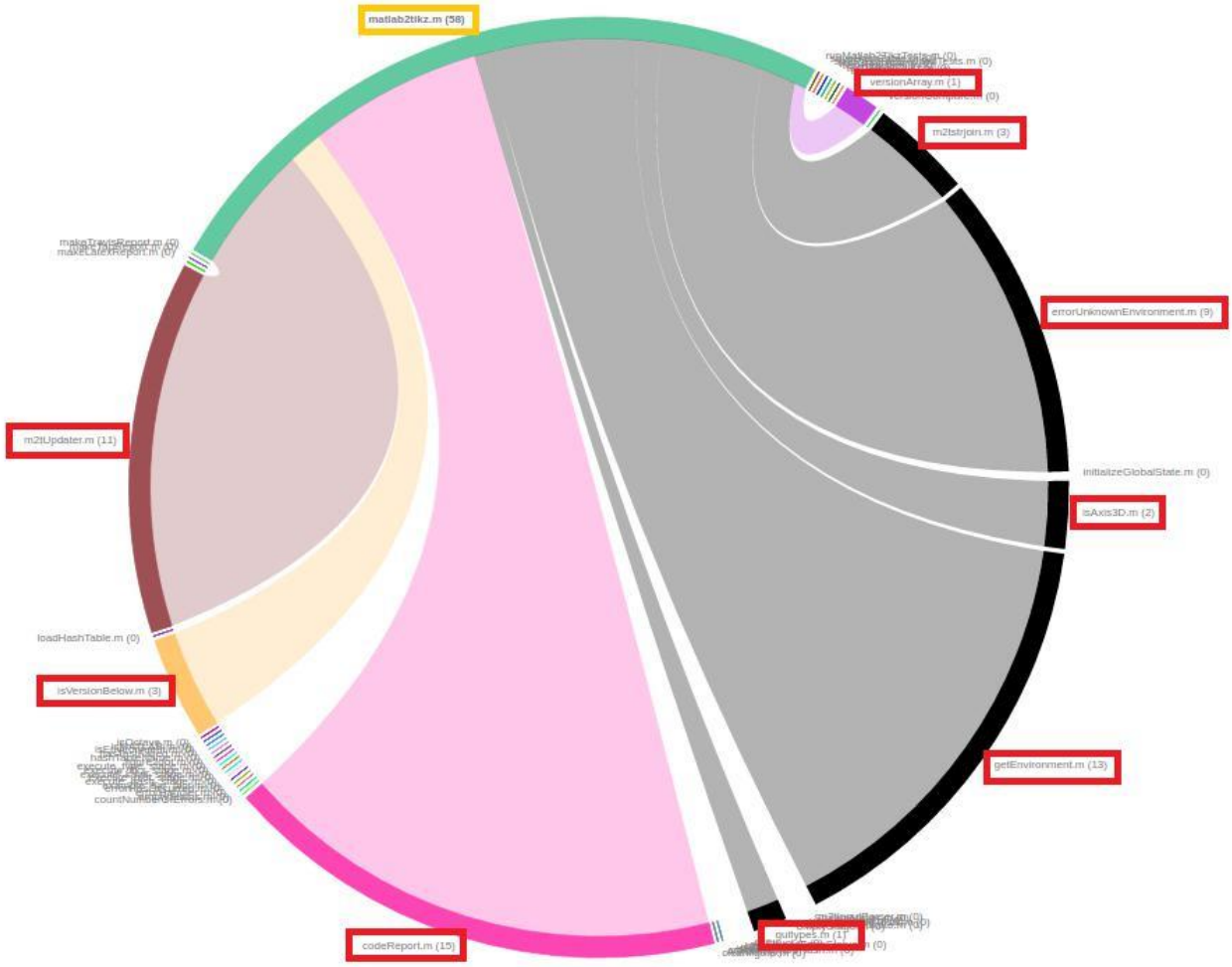


Essas consultas foram elaboradas através do desenvolvimento de views no CouchDB, conforme está descrito em sua documentação. Essas views nada mais são do que funções Java Script criadas para percorrer todos os documentos no banco de dados e retornar as informações necessárias de cada documento, caso ele faça parte da informação desejada.

O processo de utilizar estas cinco visões (3 treemaps e 2 chords) foi repetido para todos os dez projetos com o objetivo de encontrar os cinco maiores arquivos em números de linhas de código fonte de cada um dos dez projetos, os cinco arquivos com maior número de funções, os cinco arquivos que possuem ligações aferentes com a maior quantidade de arquivos e os cinco arquivos que possuem ligações eferentes com a maior quantidade de arquivos.

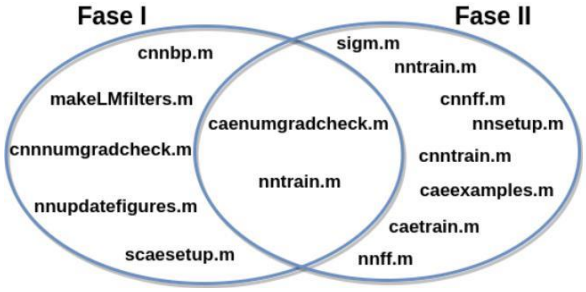
A intersecção entre esses vinte arquivos de cada projeto são os candidatos a possuírem características similares às God Class da orientação à objetos. Os resultados podem ser vistos nas figuras 44(a), 44(b), 45(a), 45(b), 46(a), 46(b), 47(a), 47(b), 48 e 49. A tabela 9 mostra o número de linhas de código-fonte, a quantidade de funções e o número de relações aferentes e eferentes de todos os arquivos que possuem características semelhantes às God Class resultantes do estudo exploratório.

Figura 43 - Visão Chord do Projeto MATLAB2Tikz que Representa as Dependências dos Arquivos



Dos dez projetos analisados, apenas três não tiveram arquivos como resultado da intersecção entre as fases I e II da estratégia de pesquisa definida: HDR_Toolbox-master, gpstuff-master e pmtk3-master. Isso não significa dizer que esses três projetos não possuam arquivos com características semelhantes às God Class da orientação à objetos. É possível que existam arquivos com tais características, mas que não foram identificados a partir da estratégia de estudo elaborada.

Figura 44 - (a) Resultado do Estudo no Projeto DeepLearnToolbox (b) Resultado do Estudo no export_fig-master



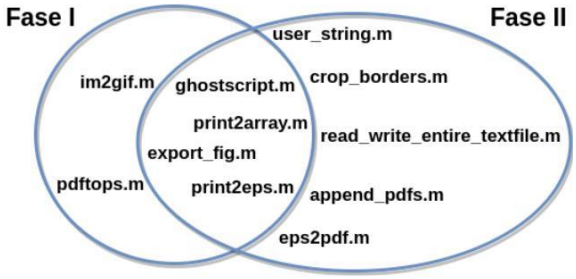
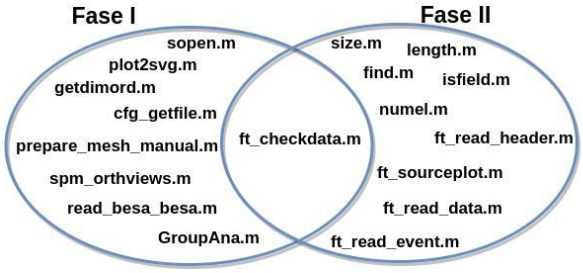


Figura 45 - (a) Resultado do Estudo no Projeto fieldtrip- master (b) Resultado do Estudo no Projeto HDR_Toolbox-master



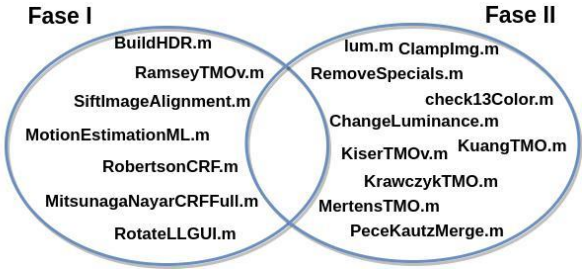
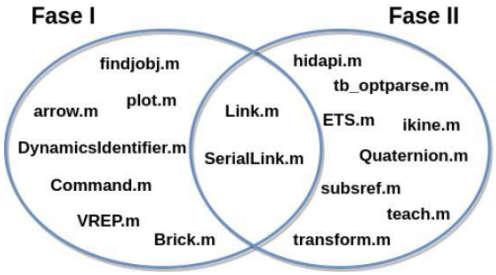


Figura 46 - (a) Resultado do Estudo no Projeto **Robotics-toolbox-matlab-master** (b) Resultado do Estudo no Projeto gpstuff-master



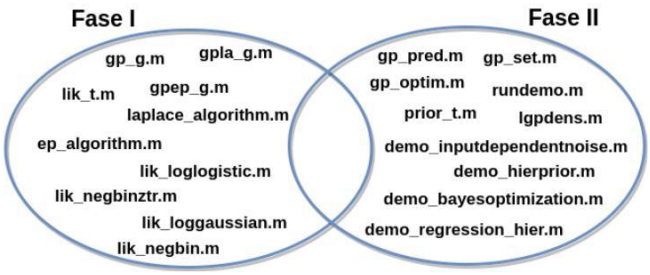


Figura 47 - (a) Resultado do Estudo no Projeto **MATLAB-Online-master** (b) Resultado do Estudo no Projeto **matlab2tikz-master**

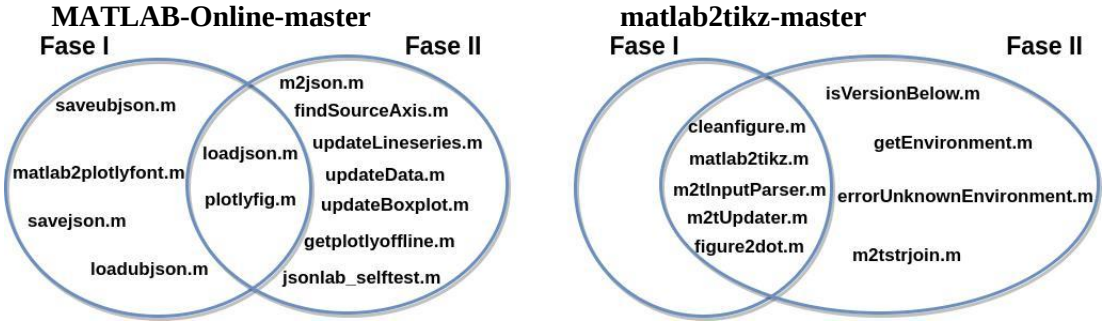


Figura 48 - Resultado do Estudo no Projeto octave-networks-toolbox-master

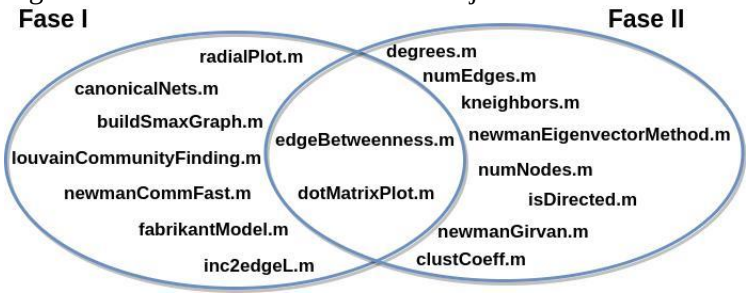


Figura 49 - Resultado do Estudo no Projeto pmtk3-master

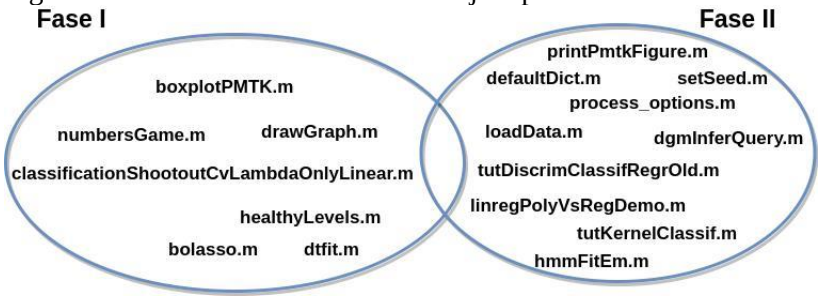


Tabela 9 - Resultado dos arquivos candidatos à God File dos 10 projetos analisados

Projetos	Qtde de Candidatos a God File	Candidatos a God File	Tamanho (Qtde de linhas)	Qtde de Funções	Qtde de ligações Aferentes	Qtde de ligações Eferentes
		matlab2tikz.m	3008	215	3	9
		cleanfigure.m	449	29	2	1
		m2tInputParser.m	142	9	5	0
		m2tUpdater.m	127	14	1	3
MATLAB2Tikz	5	figure2dot.m	60	5	0	3
FieldTrip	1	ft_checkdata.m	2244	32	176	60
		export_fig.m	621	13	0	10
		print2eps.m	245	2	2	5
		print2array.m	131	3	3	4
Export_fig	4	ghostscript.m	94	5	3	1
robotics-toolbox-matlab	2	Link.m	411	24	46	5
		SerialLink.m	293	28	43	6
MATLAB		loadJSON.m	354	16	12	3
PLOTLY API WRAPPER	2	plotlyfig.m	349	43	7	2
octave-networks-toolbox	2	edgeBetweenness.m	75	1	2	4
		dotMatrixPlot.m	70	1	0	7
		caenumgradcheck.m	62	2	0	3
DeepLearntoolbox	2	nntrain.m	43	1	4	5

6.6 RESPOSTAS DAS QUESTÕES DE PESQUISA DA DISSERTAÇÃO

As cinco questões de pesquisa desta dissertação, citadas no capítulo 3, foram utilizadas para apoiar o processo de elaboração da proposta e desenvolvimento do ambiente de visualização e suas respectivas respostas serão exibidas a seguir.

QPD1. É viável a representação visual de programas MATLAB/Octave usando o Knowledge Discovery Metamodel (KDM), sem necessidade de criar pontos de extensão ao metamodelo? As evidências apresentadas pelo ambiente de visualização proposto e sua respectiva implementação, seguindo fidedignamente o metamodelo, como proposto pela

OMG, sem pontos de extensão, permite-nos indicar a viabilidade do uso do KDM como parser para converter os dados do código fonte de programas MATLAB/Octave em documentos no formato Json que podem ser utilizados para alimentar metáforas visuais.

QPD2. Qual a efetividade do uso de um ambiente de visualização de programas MATLAB/Octave baseado no Knowledge Discovery Metamodel (KDM) para apoiar atividades de compreensão de programas? Baseado nos resultados obtidos no estudo exploratório desta dissertação, foram encontradas evidências de que o ambiente proposto e implementado se mostra efetivo para apoiar a compreensão de programas MATLAB/Octave.

QPD3. Até que ponto um ambiente de visualização de programas MATLAB/Octave baseado no Knowledge Discovery Metamodel (KDM) pode ser extensível para viabilizar a inclusão de novas metáforas visuais? Foi proposta uma arquitetura baseada em serviços justamente para viabilizar a inclusão de novas metáforas visuais que são carregadas a partir dos documentos Json que representam a estrutura KDM de um programa MATLAB/Octave. Para tanto, basta criar um serviço e uma view no CouchDB, que juntos, sejam capazes de montar o arquivo Json na estrutura exigida pela nova metáfora visual.

QPD4. Até que ponto um ambiente de visualização de programas MATLAB/Octave baseado no Knowledge Discovery Metamodel (KDM) pode ser flexível para importar estruturas de dados no formato KDM de outras linguagens de programação? A arquitetura utilizada no desenvolvimento do ambiente de visualização não depende de nenhum recurso específico de linguagem de programação. A dependência é única exclusivamente do metamodelo KDM. Sendo assim, basta que os dados dos códigos fonte, independente de linguagem de programação, estejam estruturados segundo o KDM, que eles poderão ser importados para o ambiente de visualização.

QPD5. Até que ponto um ambiente de visualização de programas MATLAB/Octave baseado no Knowledge Discovery Metamodel (KDM) pode ser flexível o bastante para exportar a estrutura de dados KDM no formato Json para ser utilizada por outros ambientes de apoio à compreensão de programas? Como os dados de código fonte utilizados pelo ambiente de visualização estão no padrão estabelecido pelo KDM, basta que esses dados sejam exportados do ambiente para que eles possam ser utilizados por qualquer outra ferramenta que também utilize esse padrão.

6.7 CONCLUSÃO DO CAPÍTULO

Este capítulo apresentou um estudo exploratório do ambiente de visualização proposto para apoiar a compreensão de programas MATLAB/Octave.

7 CONCLUSÃO E PERSPECTIVAS FUTURAS

Este capítulo apresenta as considerações finais, contribuições, e as perspectivas de trabalhos futuros.

7.1 CONSIDERAÇÕES FINAIS

Esta dissertação apresentou um ambiente de visualização da informação que pode ser utilizado para apoiar a compreensão de programas desenvolvidos nas linguagens MATLAB e Octave através do uso do metamodelo KDM. Para utilizar a ferramenta e refazer o estudo explorado nesta dissertação com os mesmos dados, basta acessar a url <https: //github.com/glaucocarneiro/MatLabVis> e seguir o roteiro de uso disponibilizado. O dump do banco e o ambiente de visualização também estão armazenados nesse repositório.

O capítulo 4 apresentou os detalhes de implementação de um parser capaz de gerar o metamodelo KDM de programas MATLAB e Octave. O capítulo 5 apresentou um ambiente de visualização que tem a capacidade de converter os elementos contidos na estrutura KDM em documentos Json e armazená-los em um banco de dados orientado a documentos conhecido como CouchDB.

A partir dos dados armazenados no banco de dados, o ambiente de visualização consegue extrair qualquer tipo de informação sobre o software que teve seu KDM gerado, carregando essas informações em documentos Json que são utilizados para alimentar um conjunto de representações visuais disponíveis em uma biblioteca Java Script (D3js).

O capítulo VI demonstra, através de um estudo exploratório sobre dez projetos MATLAB/Octave selecionados a partir de um repositório público, o uso do ambiente proposto para descobrir arquivos de código fonte com tamanho (números de linhas de código fonte) acima do normal. Também fez parte desse estudo a descoberta do nível de acoplamento dos arquivos de código fonte dos projetos analisados.

7.2 CONTRIBUIÇÕES

Esta dissertação apresentou as seguintes contribuições:

- Disponibilização de um parser capaz de gerar a AST de programas MATLAB/Octave;

- Disponibilização de um parser capaz de gerar o KDM de programas MATLAB/Octave;

- Uma proposta de infraestrutura, baseada em banco de dados NoSQL, para permitir a descoberta de conhecimento em códigos fonte a partir de grandes volumes de dados;

- Disponibilização de um ambiente de visualização WEB, que pode ser utilizado para visualizar informações sobre códigos fonte de aplicações;

Além das contribuições citadas acima, esta dissertação comprovou que é possível utilizar o KDM para representar softwares desenvolvidos nas linguagens MATLAB e Octave, sem a necessidade de criar pontos de extensão ao metamodelo. Também foi comprovado que, apesar do KDM ter sido proposto para atividades de engenharia reversa, é viável utilizá-lo para fornecer informações úteis no processo de análise e compreensão de programas, sendo que essas informações podem ser utilizadas para carregar metáforas visuais. Essa é uma contribuição de grande relevância para a engenharia de software, pois são inúmeras as possibilidades de utilização do modelo proposto.

7.3 LIÇÕES APRENDIDAS

Através deste trabalho foi possível constatar que o metamodelo KDM, proposto inicialmente para atividades de reengenharia de software, também é capaz de responder de forma eficiente informações sobre o software que permite uma melhor compreensão do código fonte desenvolvido.

O trabalho descrito nesta dissertação também serviu para comprovar a eficiência de um banco de dados NoSQL, orientado a documentos, na obtenção de informações em grandes volumes de dados.

Também foi possível comprovar que o metamodelo KDM é capaz de representar fidedignamente todas as informações existentes em programas MATLAB/Octave no domínio de código fonte, sem necessidade de criar pontos de extensão.

7.4 LIMITAÇÕES

Algumas limitações foram identificadas através do desenvolvimento do ambiente de visualização e a realização do estudo exploratório apresentado.

- Devido à necessidade de ter que importar o código fonte para dentro da estrutura do ambiente de visualização, qualquer alteração no programa MATLAB/Octave após a importação não será contemplada nas visões criadas. Para tanto, é necessário fazer nova importação;

- Só foram disponibilizadas duas representações visuais: Treemap e Chord;

- A ferramenta só é capaz de importar projetos que tenham seus códigos fonte armazenados no repositório Github;

- O ambiente de visualização não propaga a aplicação de um filtro em todas as visões. É preciso reaplicar o filtro toda vez que uma visão for selecionada.

7.5 TRABALHOS EM ANDAMENTO

Já estão sendo providenciadas algumas alterações no ambiente de visualização para reduzir as limitações da ferramenta. Mais especificamente, estão sendo incluídas novas representações visuais e a funcionalidade de propagação da aplicação de um determinado filtro. Também serão adicionados novos filtros à ferramenta.

Um estudo sobre o uso desse ambiente de visualização com sua infraestrutura para visualizar dados de um outro domínio de conhecimento está sendo realizado. A ideia é utilizar essa ferramenta para visualizar informações em cima de dados bibliométricos.

Está sendo implementado também uma visão treemap que leve em consideração a complexidade ciclomática das funções e consequentemente dos arquivos de código fonte para pintar os retângulos que fazem parte do treemap. Essa complexidade será representada através da intensidade das cores.

Também está sendo feito um trabalho para disponibilizar essa ferramenta em um ambiente de nuvem. Para tanto, será criada uma funcionalidade de autenticação dentro da ferramenta.

REFERÊNCIAS

AHO, A. V. et al. **Compilers, principles, techniques**. 2st. ed. [*S.l.*]: Pearson/Addison Wesley, 2006.

AMR, T.; STAMBOLIYSKA, R. Getting started with d3. *In:* PRACTICAL D3. js. [*S.l.*]: Springer, 2016. p. 75–90.

ANDERSON, J. C.; LEHNARDT, J.; SLATER, N. **CouchDB**: the definitive guide. [*S.l.*]: "O'Reilly Media, Inc.", 2010

AVERBUKH, V. et al. Interface and visualization metaphors. *In:* INTERNATIONAL CONFERENCE ON HUMAN-COMPUTER INTERACTION: INTERACTION PLATFORMS AND TECHNIQUES, 12., 2007. **Proceedings** […] 2007. p. 13–22.

AVERBUKH, V. L. Toward the conceptions of visualization language and visualization metaphor. *In:* SYMPOSIA ON HUMAN CENTRIC COMPUTING LANGUAGES AND ENVIRONMENTS (HCC'01), 2001. **Proceedings** […] 2001 p. 390.

BABAR, M. A.; CHEN, L.; SHULL, F. Managing variability in software product lines.

IEEE software, IEEE, v. 27, n. 3, p. 89–91, 2010.

BIGGERSTAFF, T. J.; MITBANDER, B. G.; WEBSTER, D. E. Program understanding and the concept assignment problem. **Communications of the ACM**, v. 37, n. 5, p. 72–82, 1994.

BORGATTI S.P.; FOSTER, P. The network paradigm in organizational research: a review and typology. **Journal of Management**, v. 29, n. 6, p. 991–1013, 2003.

BOSTOCK, M.; OGIEVETSKY, V.; HEER, J. D3 data-driven documents**. IEEE transactions on visualization and computer graphics,** IEEE, v. 17, n. 12, p. 2301–2309, 2011.

BOUSSAIDI, G. E. et al. Reconstructing architectural views from legacy systems. *In*: IEEE. REVERSE ENGINEERING (WCRE), 2012 WORKING CONFERENCE ON, 119., [*S.l.*], 2012. **Proceedings** […] 2012. p. 345–354.

BRITO, R. W. **Bancos de dados nosql x sgbds relacionai**s: análise comparativa. Fortaleza: Faculdade Farias Brito e Universidade de Fortaleza, 2010.

CANOVAS, J.; MOLINA, J. An architecture-driven modernization tool for calculating metrics. **IEEE software**, IEEE, v. 27, n. 4, p. 37–43, 2010.

CARNEIRO, G. d. F**. SourceMiner:** um ambiente integrado para visualização multi-perspectiva de software. *[S.l.]: [s.n.],* 2013.

CATTELL, R. Scalable sql and nosql data stores. **Acm Sigmod Record,** ACM, v. 39, n. 4, p. 12–27, 2011.

CHAPMAN, S. J**. Programação em MATLAB para engenheiros**. *[S.l.]: [s.n.],* 2003.

DELTOMBE, G.; GOAER, O. L.; BARBIER, F. Bridging kdm and astm for model-driven software modernization. *In*: **SEKE**. *[S.l.]: [s.n.]*, 2012. p. 517–524.

EVELIEN, O.; RONALD, R. Social network analysis: a powerful strategy, also for information sciences**. Journal of Information Science**, v. 28, n. 6, p. 441–453, 2002.

FREITAS, C. M. D. S. *et al*. Introdução à visualização de informações. **Revista de informática teórica e aplicada,** Porto Alegre, V. 8, n. 2, p. 143-158, out. 2001.

HEVNER, A. R. *et al*. The impact of function extraction technology on next-generation software engineering. *[S.l.]: [s.n.],* 2005.

HEYDT, M. D3. js By Example. *[S.l.]:* Packt Publishing Ltda, 2015.

HOLTEN, D. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. **IEEE Transactions on visualization and computer graphics**, IEEE, v. 12, n. 5, p. 741–748, 2006.

IEPSEN E. F.; LUZZARDI, P. R. G. L. S. Vismade: visualizing temporal information in databases structured as master-details. **Revista Eletrônica de Sistemas de Informação**, v. 6, n. 2, p. 1–10, 2007.

KOSAR, T.; MERNIK, M.; CARVER, J. C. Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments. **Empirical software engineering,** Springer, v. 17, n. 3, p. 276–304, 2012.

KOSCHKE, R. Software visualization for reverse engineering. *In:* SOFTWARE Visualization.*[S.l.]:* Springer, 2002. p. 138–150.

LESSA, I. d. M. et al. A multiple view interactive environment to support matlab and gnu/octave program comprehension. *In:* IEEE. INFORMATION TECHNOLOGY-NEW GENERATIONS (ITNG), INTERNATIONAL CONFERENCE ON, 12., 2015. **Proceedings** […] 2015. p. 552–557.

LI, Y.; MANOHARAN, S. A performance comparison of sql and nosql databases. *In:*

IEEE. COMMUNICATIONS, COMPUTERS AND SIGNAL PROCESSING (PACRIM), IEEE PACIFIC RIM CONFERENCE ON. 2013. **Proceedings** […] 2013. p. 15–19.

MAJUMDAR, N. **MATLAB Graphics and Data Visualization Cookbook**. [S.l.]: Packt Publishing Ltd, 2012.

MÜLLER, H. A.; TILLEY, S. R.; WONG, K. Understanding software systems using reverse engineering technology perspectives from the rigi project. *In:* IBM PRESS.

CONFERENCE OF THE CENTRE FOR ADVANCED STUDIES ON COLLABORATIVE RESEARCH: SOFTWARE ENGINEERING, 1., 1993. **Proceedings** […] 1993. p. 217–226.

NOVAIS, R. L.; JUNIOR, P. S.; MENDONÇA, M. Timeline matrix: an on demand view for software evolution analysis. *In*: SOFTWARE VISUALIZATION (WBVS), BRAZILIANWORKSHOP ON, 2., 2012. **Proceedings** […]. 2012. p. 1–8.

OMG, O. M. G. **Architecture-Driven Modernization**: Knowledge Discovery Meta-Model (KDM). 2016. Disponível em: http://www.omg.org/spec/KDM/1.4/.

PÉREZ-CASTILLO, R.; GUZMAN, I. G.-R. D.; PIATTINI, M. Knowledge discovery metamodel-iso/iec 19506: A standard to modernize legacy systems. **Computer Standards & Interfaces**, Elsevier, v. 33, n. 6, p. 519–532, 2011.

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.

SANTOS, B. M. et al. Kdm-ao: An aspect-oriented extension of the knowledge discovery metamodel. *In:* IEEE. SOFTWARE ENGINEERING (SBES), 2014 BRAZILIAN SYMPOSIUM On., 2014. **Proceedings** […] p. 61–70.

SEIDL, H.; WILHELM, R.; HACK, S. **Compiler Design**: Analysis and Transformation. *[S.l.]:* Springer Science & Business Media, 2012.

SHARMA, N.; GOBBERT, M. K. **A comparative evaluation of matlab, octave, freemat, and scilab for research and teaching**. [*S.l.*]: Department of Mathematics and Statistics, 2010.

SHNEIDERMAN, B. Tree visualization with tree-maps: 2-d space-filling approach. **ACM Transactions on graphics (TOG),** ACM, v. 11, n. 1, p. 92–99, 1992.

SPENCER, R. **Information visualization**: design for interaction. [*S.l.*]: Person Education, 2007.

STOREY, M.-A. Theories, methods and tools in program comprehension: Past, present and future. *In*: IEEE. PROGRAM COMPREHENSION, 2005. IWPC. INTERNATIONAL WORKSHOP ON, 13., 2005. **Proceedings […].** [S.l.], 2005. p. 181–191.

STOREY, M.-A.; MULLER, H. A. Manipulating and documenting software structures using shrimp views. *In*: IEEE. SOFTWARE MAINTENANCE, 1995. INTERNATIONAL WORKSHOP ON, **Proceedings […].** *[S.l.]*, 1995. p. 275–284.

ULRICH, W. A status on omg architecture-driven modernization task force. *In*: EDOC WORKSHOP ON MODEL-DRIVEN EVOLUTION OF LEGACY SYSTEMS (MELS). 2004. **Proceedings […].** *[S.l.]:* IEEE Computer Society Digital Library, 2004.