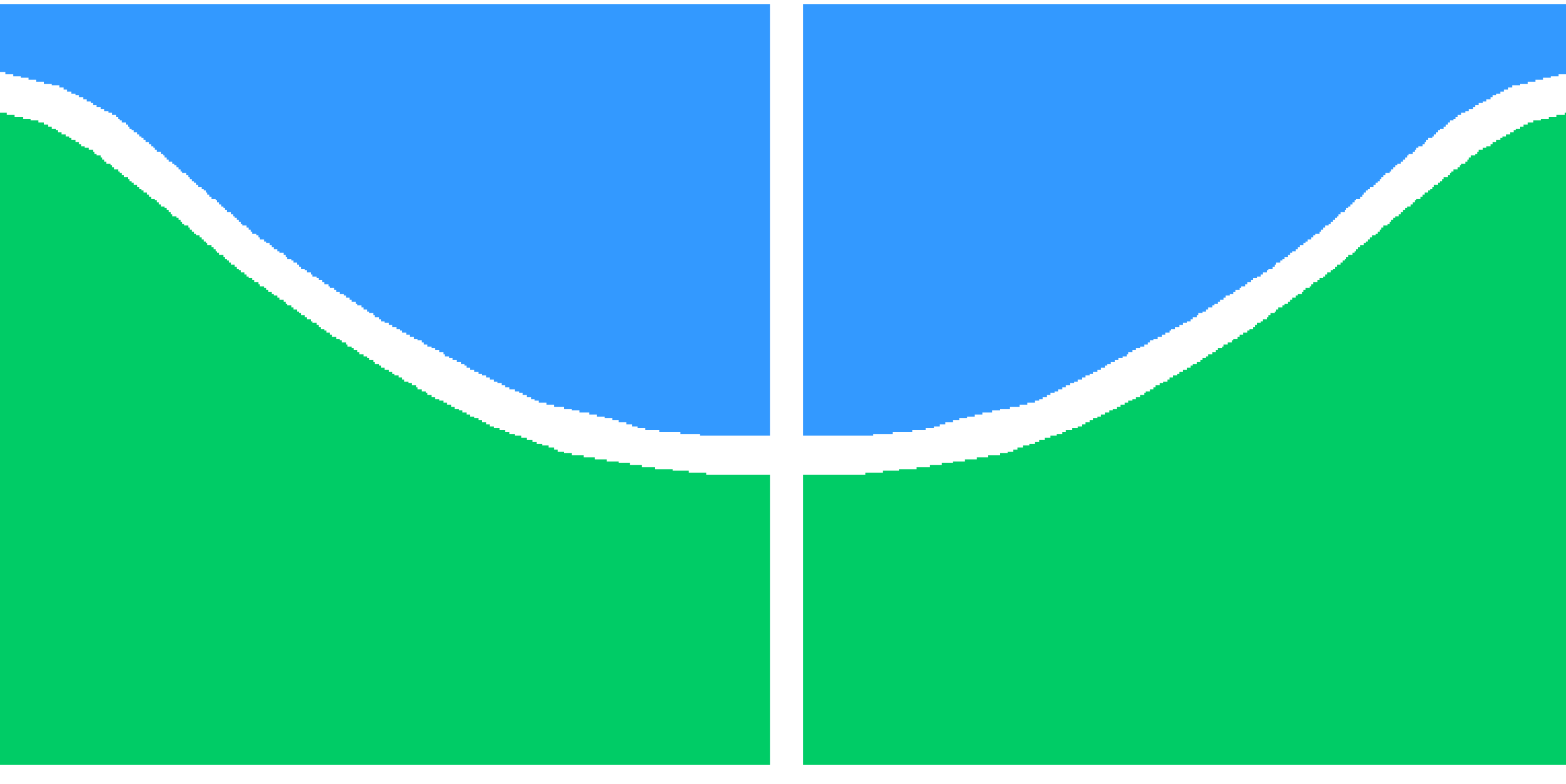

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Engenharia de Software

Cenário de Decisões Baseado em Métricas de Software: Definição e Implementação de Cenários a partir de Métricas de *Design* e de Vulnerabilidade para Tomada de Decisão

Autores: Arthur de Moura Del Esposte

Carlos Filipe Lima Bezerra

Orientador: Professor Doutor Paulo Roberto Miranda Meirelles

Coorientador: Prof. Msc. Hilmer Rodrigues Neri

Brasília, DF

2014

Arthur de Moura Del Esposte
Carlos Filipe Lima Bezerra

Cenário de Decisões Baseado em Métricas de Software: Definição e Implementação de Cenários a partir de Métricas de *Design* e de Vulnerabilidade para Tomada de Decisão

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Orientador: Professor Doutor Paulo Roberto Miranda Meirelles
Coorientador: Prof. Msc. Hilmer Rodrigues Neri

Brasília, DF
2014

Arthur de Moura Del Esposte
Carlos Filipe Lima Bezerra

Cenário de Decisões Baseado em Métricas de Software: Definição e Implementação de Cenários a partir de Métricas de *Design* e de Vulnerabilidade para Tomada de Decisão/ Arthur de Moura Del Esposte & Carlos Filipe Lima Bezerra. – Brasília, DF, 2014-

147 p. : il. (algumas color.) ; 30 cm.

Orientador: Professor Doutor Paulo Roberto Miranda Meirelles

Coorientador: Prof. Msc. Hilmer Rodrigues Neri

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Faculdade UnB Gama - FGA , 2014.

1. Métricas. 2. Design. 3. Segurança. 4. Monitoramento. 5. Data Warehousing. 6. Cenários de Decisões. I. Professor Doutor Paulo Roberto Miranda Meirelles. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Cenário de Decisões Baseado em Métricas de Software: Definição e Implementação de Cenários a partir de Métricas de *Design* e de Vulnerabilidade para Tomada de Decisão

CDU 02:141:005.6

Arthur de Moura Del Esposte
Carlos Filipe Lima Bezerra

Cenário de Decisões Baseado em Métricas de Software: Definição e Implementação de Cenários a partir de Métricas de *Design* e de Vulnerabilidade para Tomada de Decisão

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Trabalho aprovado. Brasília, DF, 24 de junho de 2014:

Professor Doutor Paulo Roberto Miranda Meirelles
Orientador

Prof. Msc. Hilmer Rodrigues Neri
Coorientador

Professor Doutor Fabricio Ataides Braz
Convidado 1

Professor Doutor Edson Alves da Costa Junior
Convidado 2

Brasília, DF
2014

Agradecimentos

de Arthur de Moura Del Esposte

Agradeço ao grupo de professores de Engenharia de Software da Faculdade UnB Gama e à Universidade de Brasília por todo conhecimento fornecido ao longo da minha graduação. Em especial, sou grato a alguns professores que citarei devido a importantes contribuições pessoais e profissionais.

Agradeço ao Prof. André Barros de Sales pelas oportunidades e projetos desenvolvidos durante os primeiros semestres do curso. Também agradeço em especial ao Prof. Edson Alves da Costa Junior por tantas oportunidades de aprendizado técnico; por sempre fortalecer e acreditar no potencial dos alunos de Engenharia de Software; por promover e apoiar competições de programação; e por ser um grande exemplo e referência como profissional e ser-humano;

Ao Prof. Hilmer Rodrigues Neri, coorientador deste trabalho, agradeço por sempre buscar a melhoria do curso de Engenharia de Software; por sempre contribuir e ampliar minha visão sobre essa profissão; pelos ensinamentos sobre métodos ágeis e por suas contribuições para o desenvolvimento de habilidades gerenciais; e pela suas orientações nesta monografia.

Agradeço ao orientador Prof. Paulo Roberto Miranda Meirelles por todas as oportunidades acadêmicas e profissionais compartilhadas; por todos os ensinamentos técnicos, metodológicos e organizacionais fornecidos que são fundamentais para minha formação profissional; por me ensinar a importância e como contribuir com softwares livres; por atuar intensivamente na minha formação; por me mostrar a importância da contribuição e trabalho em equipe em minha vida profissional e pessoal; e pela orientação ao longo deste trabalho.

Agradeço à todos os membros do Laboratório Avançado de Produção, Pequisa e Inovação em Software - LAPPIS pelas experiências compartilhadas, o crescimento conjunto e o trabalho em equipe.

Aos meus pais, Antônio César Del Esposte e Lucimar de Moura Del Esposte, sou profundamente grato pelo dom da vida e por toda a dedicação; amor e confiança; pelo imensurável apoio oferecido em todos os momentos; pelos seus ensinamentos e conhecimentos; por serem o meu lar, minha luz, minha inspiração, meu espelho e minha base; e por sempre acreditarem em mim. Dedico esta conquista aos dois.

Ao meu irmão e amigo Heitor de Moura Del Esposte, fonte de admiração e inspiração, sou eternamente grato pelo companherismo, amizade, amor e tudo mais que sua presença agrega em minha vida. Sou a ele grato por toda admiração, fé, respeito e confiança em mim; por compartilharmos tudo, principalmente nossas conquistas; por estar sempre ao meu lado; e por ser a principal parte da minha força. Dedico este trabalho à ele também.

Agradeço à Ana Paula Vieira Araujo por ter me acompanhado em todos os passos da minha formação; por me ajudar a construir um grande futuro; por sua dedicação, amor e paciência; por ser fonte de inspiração; por compartilhar meus sonhos; e por todas as modificações positivas em minha vida provindas de sua presença;

Por fim, agradeço aos meus familiares e aos meus grandes amigos pelo apoio, amizade, dedicação; por comporem minha base; por participarem ativamente da minha vida pessoal e por todas alegrias compartilhadas.

Agradecimentos

de Carlos Filipe Lima Bezerra

Agradeço primeiramente a Deus, pelo dom da vida e todos os outros dons que me destes, além da força e sabedoria durante essa caminhada.

Agradeço a minha família, pelo o apoio e carinho, principalmente aos meus pais, Antônio da Silva Bezerra e Joana Darte Lima Bezerra, que nunca mediram esforços em me dar todo o suporte necessário para que chegasse a essa etapa da minha vida. Foram vocês que me ensinaram grandes valores da vida e continuam a me ensinar sempre. Vocês são verdadeiros exemplos a serem seguidos.

Agradeço ao Prof. Hilmer Neri, que iniciou este trabalho como meu orientador e com muita atenção abriu horizontes que eu não havia explorado ainda, estendendo ainda mais minha aprendizagem em diferentes aspectos da Engenharia de Software.

Agradeço ao Prof. Paulo Meirelles pela dedicação de seu papel como orientador e entusiasmo com o trabalho desenvolvido. Na disciplina de manutenção e evolução, sua abordagem de ensino permitiu o amadurecimento em vários aspectos aprendidos durante o curso e a aprendizagem de trabalho em equipe, além de permitir a contribuição com softwares em produção, experiências fundamentais para melhoria de minha formação.

Agradeço a Andrezza Santos de Oliveira, pelo apoio, amor e paciência ao longo da minha caminhada. Você é minha fonte de inspiração e dedicação para construção do meu futuro e realização de sonhos que compartilhamos.

Agradeço aos meus companheiros de graduação, em especial ao Pedro Potiguara e Marcos Ronaldo, que compartilharam comigo muitos conhecimentos ao longo dessa formação.

Agradeço a equipe do SEINT do Tribunal de Contas da União, em especial ao Marcus Vinicius Borela e ao Edmilson Rodrigues, que durante meu período de estágio me propiciaram vários desafios e experiências fundamentais para meu amadurecimento e formação.

Resumo

A qualidade interna é um dos fatores de sucesso de projetos de software, pois corresponde a aspectos primordiais do software tais como manutenibilidade e segurança. Softwares com boa qualidade interna proporcionam maior produtividade uma vez que possibilitam a criação de mais testes automatizados, são mais compreensíveis, reduzem o risco de *bugs* e facilitam as modificações e evoluções no código. Portanto, o Engenheiro de Software é um dos responsáveis por esse sucesso, uma vez que deve reunir um conjunto de habilidades e conhecimentos que o permitam aplicar práticas, técnicas e ferramentas para a criação de softwares seguros e com bom *design*. Diante disso, este trabalho aborda as principais ideias e conceitos relacionados à melhoria contínua do código-fonte. Nesse sentido, nesta monografia é destacada a importância da realização de atividades contínuas relacionadas ao *design* e segurança ao longo de todo o projeto de software, além de discutir a importância da utilização de métricas estáticas de código-fonte para suportar a tomada de decisões, tanto em nível técnico quanto gerencial. Nesse sentido, é apresentado o conceito de Cenários de Decisões que definem uma abstração para escolha e interpretação de métricas, além da proposta de exemplos de utilização destes Cenários para medição da segurança de software. Para suportar a utilização de cenários e métricas no desenvolvimento de software, este trabalho ainda contempla a colaboração na evolução da plataforma livre de monitoramento de código-fonte chamada Mezuro e a construção de uma solução de DataWarehousing.

Palavras-chaves: Métricas; Design; Segurança; Monitoramento; DataWarehousing; Cenários de Decisões; Código-Fonte;

Abstract

The internal quality is a success factor of software projects because it corresponds to the main aspects of the software such as maintainability and security. Software with good internal quality provides more productivity since it supports the creation of more automated tests, as well as it is more understandable, reduces the risk of bugs, and makes the code changes and developments easier to be done. Therefore, the Software Engineer is a major contributor for this success since he should gather a set of skills and knowledge to apply practices, techniques, and tools for creating secure and well design software. Thus, this research covers the main ideas and concepts related to continuous improvement of source code. In this context, in this degree monograph, we highlight the importance of conducting ongoing activities related to design and security throughout the software project, as well as discuss the importance of using static source code metrics to support decision making at managerial level and technical as well. In this regard, we present the concept of Decisions Scenarios that define an abstraction for metrics choice and interpretation, as well as proposals of examples to use scenarios for measuring software security. To support the use of scenarios and metrics in software development, this work also includes a collaboration for the evolution of a source code monitoring platform called Mezuro and building a datawarehousing solution.

Key-words: Metrics; Design; Security; Monitoring; DataWarehousing; Decisions Scenarios; Source Code;

Lista de ilustrações

Figura 1 – Práticas do Design Ágil Utilizando Métricas de Código-Fonte 51
Figura 2 – Método para execução dos estudos de casos. 73
Figura 3 – Reading Group criado para Cenários. Disponível no Mezuro.org <http://mezuro.org/reading_groups/7> 76
Figura 4 – Configuração criada para Cenários de Design Seguro. Disponível no Mezuro.org <http://mezuro.org/mezuro_configurations/24> 77
Figura 5 – Visualização da Métrica NOC no Mezuro. Disponível no Mezuro.org <http://mezuro.org/mezuro_configurations/24/metric_configurations/69> 78
Figura 6 – Cenário Completo representado no Mezuro. Disponível no Mezuro.org <http://mezuro.org/mezuro_configurations/24/compound_metric_configurations/67> 81
Figura 7 – Modelo dimensional do fato *f_qtd_scenarios* que irá armazenar os cenários identificados no projeto. 85
Figura 8 – Modelo dimensional do fato *f_design_metrics_values* que irá armazenar os valores das métricas de cada classe do projeto. 85
Figura 9 – Modelo dimensional do fato *f_security_metrics_values* que irá armazenar os valores das métricas de segurança de cada arquivo do projeto. 86
Figura 10 – Ferramentas utilizadas para o desenvolvimento do ambiente de *DWing*. 88
Figura 11 – Processo de ETL para identificação dos cenários de decisão feito no PDI 90
Figura 12 – Apresentação do processo de coleta de métricas no Mezuro 95
Figura 13 – Apresentação da árvore de módulos do Mezuro 96
Figura 14 – Apresentação das medições do Mezuro 97
Figura 15 – Resultado da classe MessageBox do projeto Athom Shell 98
Figura 16 – Workspace do plugin *Saiku Analytics* no BI server 98
Figura 17 – Atributos que forma as hierarquias para fazer agregações de dados. . . 99
Figura 18 – Gráfico gerado pelo *Saiku Analytics* que permite ver a quantidade de cenários de decisões ao logo de cada release 100
Figura 19 – Relatório com a quantidade de vulnerabilidades por release e por tipo de vulnerabilidade específica encontradas no Octave 100
Figura 20 – Relatório que verifica quais as vulnerabilidades foram encontradas no arquivo *CSparse.cc*, indicando também a linha de ocorrência 101
Figura 21 – Práticas do Design Ágil, adaptado de Scoot W. Ambler 112
Figura 22 – Componentes de um *DWing*. Adaptação de (KIMBALL; ROSS, 2002) . 123
Figura 23 – Exemplo de cubo de dados Fonte: (GUIMARAES, 2012) 126
Figura 24 – Esquema estrela Fonte: (WAGNER, 2012) 127

Figura 25 – Esquema floco de neve Fonte: (WAGNER, 2012) 128
Figura 26 – Demonstração da operação de *Drill Down* Fonte: (TUTORIALSPOINT, 2014) 129
Figura 27 – Demonstração da operação de *Slice* Fonte: (TUTORIALSPOINT, 2014) 130
Figura 28 – Demonstração da operação de *Dice* Fonte: (TUTORIALSPOINT, 2014) 131
Figura 29 – Demonstração da operação de *Pivoting* Fonte: (TUTORIALSPOINT, 2014) 131
Figura 30 – Ciclo de vida de um Projeto de *DWing* (KIMBALL; ROSS, 2002) . . . 132
Figura 31 – Tela principal do Mezuro. Disponível em <http://mezuro.org/> 136
Figura 32 – Arquitetura Atual do Mezuro. Extraído de (MANZO et al., 2014) . . . 138
Figura 33 – Arquitetura Futura do Mezuro. Extraído de (MANZO et al., 2014) . . 138

Lista de tabelas

Tabela 1 – Parte I - Resumo de todos os cenários 69
Tabela 2 – Parte II - Resumo de todos os cenários 70
Tabela 3 – Parte III - Resumo de todos os cenários 71

Lista de abreviaturas e siglas

AGPL	GNU Affero General Public License
BI	Business Inteligence
CVE	Common Vulnerabilities and Exposures
CWE	Common Weakness Enumeration
DPA	Data Presentation Area
DRY	Don’t Repeat Yourself
DSA	Data Staging Area
DW	Data Warehouse
DWing	Data Warehousing
ETL	Extract Transform Load
FGA	Faculdade UnB Gama
HTTP	Hipertext Transfer
ISO	International Organization for Stardardization
JaBUTi	Java Bytecode Understanding and Testing
LOC	Lines of Code
MVC	Model-View-Controller
NIST	National Institute of Technology
OLAP	On-line Analytical Processing
OSS	Operational Source Systems
Rails	Ruby on Rails
RPC	Remote Procedure Call
SGBD	Sistema de Gerenciamento de Banco de Dados
SOAP	Simple Object Access Protocol

TCC	Trabalho de Conclusão de Curso
UnB	Universidade de Brasília
USP	Universidade de São Paulo
XML	Extensible Markup Language

Sumário

1 INTRODUÇÃO . . . 23
1.1 Problema . . . 26
1.2 Objetivos . . . 26
1.3 Contribuições . . . 26
1.4 Metodologia e Pesquisa . . . 27
1.5 Organização do Trabalho . . . 27

2 DESIGN E SEGURANÇA DE SOFTWARE . . . 29
2.1 Design de Software . . . 29
2.1.1 O Design e seus Princípios . . . 30
2.2 Segurança de Software . . . 33
2.2.1 Classificações e taxonomias de vulnerabilidades . . . 36
2.2.2 Princípios de segurança . . . 40

3 MÉTRICAS DE CÓDIGO-FONTE . . . 43
3.1 Métricas Estáticas de *Design* de Software . . . 45
3.2 Métricas Estáticas de Segurança . . . 47
3.3 Proposta de Utilização de Métricas de Código-fonte . . . 51

4 CENÁRIOS DE DECISÕES . . . 53
4.1 Definição de Cenários de Decisão para Segurança de Software . . . 56
4.1.1 Cenários de Decisão para Caracterização da Qualidade do Código . . . 57
4.1.1.1 Alta Superfície de Ataque a Atributos Internos . . . 57
4.1.1.2 Alta Superfície de Ataque Operacional . . . 58
4.1.1.3 Ponto Crítico de Falha . . . 59
4.1.1.4 Risco Elevado de Segurança . . . 60
4.1.2 Cenários de Decisão para Caracterização de Vulnerabilidades Específicas de Código . . . 62
4.1.2.1 Uso de variáveis não inicializadas . . . 62
4.1.2.2 Alta possibilidade de Falha por mau uso de ponteiros . . . 64
4.1.2.3 Buffer Overflow . . . 65
4.1.2.4 Confidencialidade Ameaçada . . . 66
4.1.2.5 Operações lógicas (somas, divisões) com integridade ameaçada . . . 67

5 ESTUDO DE CASO . . . 73
5.1 Cenários de Decisões no Mezuro . . . 74

5.1.1 Criação do *Reading Group* para os Cenários . . . 76
5.1.2 Criação de uma Configuração para uma Categoria de Cenários . . . 77
5.1.3 Criação de Cenários . . . 79
5.1.4 Associando o *Reading Group* ao Cenário . . . 80
5.2 Cenários de Decisão e *Data Warehousing* . . . 83
5.2.1 Modelagem Dimensional . . . 84
5.2.2 Criação do ambiente de *DWing* . . . 87
5.3 Projetos analisados . . . 91
5.3.1 GNU Octave . . . 92
5.3.2 Athom Shell . . . 92
5.3.3 Synergy . . . 93
5.4 Coleta de Dados . . . 93
5.5 Análise dos Cenários nas Ferramentas . . . 95
5.5.1 Visualização no Mezuro . . . 95
5.5.2 Visualização no DWing . . . 97
5.5.3 Análise e Discussão . . . 101

6 CONCLUSÕES . . . 105
6.1 Limitações . . . 107
6.2 Trabalhos Futuros . . . 108

Appendices . . . 109

A DESIGN E SEGURANÇA DE SOFTWARE . . . 111
A.1 Design de Software . . . 111
A.1.1 O Design e suas Práticas . . . 111
A.1.2 *Code Smells* - Cheiros de Código . . . 113
A.1.3 Código Limpo . . . 115
A.2 Unindo Conceitos de Segurança e *Design* . . . 116

B *DATA WAREHOUSE* . . . 121
B.1 *Data Warehousing* . . . 122
B.2 Modelagem dimensional . . . 125
B.3 OLAP . . . 128
B.4 Ciclo de vida de um ambiente de *Data Warehousing* . . . 130
B.5 Construção de ambiente de DW para o monitoramento de métricas de software. . . . 132

C MEZURO: UMA PLATAFORMA DE MONITORAMENTO DE CÓDIGO-FONTE . . . 135
C.0.1 Arquitetura do Mezuro . . . 137

Referências **141**

1 Introdução

A Engenharia de Software tem evoluído seus métodos e técnicas para prover melhorias no desenvolvimento de software com objetivos baseados em cumprimento de prazos e orçamentos assim como a implementação de produtos que atendem parâmetros de qualidades desejados. Essas melhorias são observáveis em diferentes pontos, desde o processo ao produto, cujos objetivos e prioridades podem variar de acordo com a metodologia de desenvolvimento. Apesar de suas diferenças conceituais e de valores, a maior parte dos métodos preveem processos e técnicas referentes ao *design*, testes e medição, que visam garantir a qualidade do software em desenvolvimento.

No contexto de projetos que adotam metodologias ágeis, observa-se que tanto a qualidade interna quanto a qualidade externa do software são preponderantes, pois são fatores fundamentais para suportar a simplicidade, o *feedback* contínuo e adaptação à mudanças, valores que solidificam o desenvolvimento ágil. A qualidade interna do software é observada a partir de atributos de qualidades na perspectiva de desenvolvimento que, segundo Berander (2005), se resumem em corretude, testabilidade, flexibilidade, portabilidade, reusabilidade, interoperabilidade, analisabilidade, adaptatividade e estabilidade. As práticas ressaltadas pela metologia *Extreme Programming* (BECK; ANDRES, 2000) visam realçar os valores dos atributos destacados, atributos esses que definem um bom *design*.

O *design* simples pode ser obtido através de técnicas como o Desenvolvimento Orientado à Testes (BECK, 2002) e a Refatoração (FOWLER et al., 1999), que por sua vez influenciam diretamente os atributos testabilidade, reusabilidade e adaptatividade. Ambas as técnicas se baseiam fortemente em testes unitários que provêem a segurança necessária para realização de mudanças assim como o feedback automatizado da manutenção do software. A Programação em Pares, dentre outras práticas, também contribui para a garantia da qualidade interna, uma vez que exercita a programação e revisão ao mesmo tempo, reduzindo a ocorrência de não-conformidades técnicas e inserção de *bugs*. Por outro lado, a qualidade externa do software pode ser alcançada a partir do envolvimento do "cliente" ao longo das atividades de desenvolvimento e, principalmente, a partir de entregas contínuas de software com valor de negócio.

Valores semelhantes podem ser observados nas comunidades de desenvolvimento de softwares livres refletindo diretamente na alta qualidade do código produzido em diversos projetos livres (SCHMIDT; PORTER, 2001); (HALLORAN; SCHERLIS, 2002); (MICHLMAYR; HILL, 2003). Essas comunidades adotam a padronização de código e testes automatizados para manter a qualidade interna do código e incentivar a contribuição

de diversos desenvolvedores.

A melhoria da qualidade interna do código apoia a melhoria contínua do processo oferecendo subsídios para que a equipe de desenvolvimento aumente sua produtividade e implemente novas funcionalidades com maior facilidade. Beck (2007) corrobora essa afirmação ao destacar que a maior parte do tempo utilizado por um Programador ao inserir novas funcionalidades é destinado ao entendimento do código em manutenção. Diretamente relacionado à qualidade de código está a sua segurança (TSIPENYUK; CHESS; MCGRAW, 2005). A segurança de software está relacionada a confidencialidade, disponibilidade e integridade dos diversos componentes que compõe o software.

Dados do ICAT/NIST [1] de 2005 já apontavam que 80% das vulnerabilidades remotamente exploráveis estavam ligadas a má codificação do programa (DUARTE; BARBATO; MONTES, 2005). Embora a segurança de uma aplicação também estejam relacionadas a aspectos externos ao software como a redes e componentes de hardware, o elo mais fraco continua sendo o próprio software. Dessa forma, cabe aos projetistas e desenvolvedores a responsabilidade do desenvolvimento de software seguro, sem prejuízos aos seus usuários.

À medida que o tempo vai passando, novas vulnerabilidades vão sendo descobertas pela comunidade. O projeto CVE (*Common Vulnerabilities and Exposures List*), que tem como objetivo enumerar vulnerabilidades de software existentes, tinha uma lista de 321 vulnerabilidades diferentes no ano de sua concepção, em 1999 (MARTIN; CHRISTEY; BAKER, 2002). No ano de 2002, a lista já havia aumentado para 2032 vulnerabilidades e atualmente o número já chega a 61 mil vulnerabilidades específicas encontradas por empresas de todo o mundo.

Visto esse cenário de inúmeras vulnerabilidades é fundamental que todos aqueles envolvidos no processo de produção do software tenham conhecimento das implicações relativas a segurança. O conhecimento de vulnerabilidades e meios de detectá-las são habilidades necessárias para garantia de software seguro. Esse conhecimento deve estar alinhado às habilidades de concepção de um bom *design* para prover a segurança necessária no desenvolvimento do software e se alcançar os valores e objetivos anteriormente destacados.

Neste sentido, a medição pode ser utilizada como um processo de apoio ao acompanhamento da segurança e qualidade, através do estabelecimento de metas e indicadores que indiquem oportunidades de melhorias observáveis do produto. Em um cenário otimista, os próprios Engenheiros de Software podem adotar como prática a medição do

[1] ICAT foi um motor de busca de vulnerabilidades, desenvolvido pelo NIST(*National Institue of Standards and Technology*), catalogadas no padrão CVE. O ICAT foi substituido pelo NVD (*National Vulnerability Database*), que além de possuir o mesmo mecanismo de busca, é um repositório governamental dos Estados Unidos que armazena diversas informações sobre vulnerabilidades de software (nomenclaturas, métricas, checklists, etc).

código-fonte para auxiliar as tomadas de decisões, ou até mesmo para avaliação do código inserido ou da aplicação de refatorações. Entretanto, uma grande quantidade de métricas, coletas manuais e poucos recursos de visualização são fatores que acabam por desmotivar o uso dessas para o monitoramento do código. Além disso, a compreensão do significado de valores obtidos através de métricas não é uma tarefa trivial, demandando um grande esforço de interpretação necessárias para a tomada de decisão efetiva sobre o projeto de software.

Assim, destaca-se a utilidade de ferramentas que auxiliem o processo de medição, compreensão e visualização do software. Atualmente existem algumas ferramentas que automatizam a extração de métricas do código-fonte com objetivo de coletar as informações sobre o produto a partir da análise estática do código, as quais definimos como Extratores. Outras ferramentas denominadas Plataformas de Monitoramento de código-fonte procuram oferecer melhores formas de monitoramento e visualização do software a partir da personalização de métricas e mecanismos que facilitem a interpretação dos resultados obtidos. Alternativamente, um ambiente de *Data Warehousing* (DWing) é uma solução que tem se destacado no ramo de *Business Intelligence* - BI e tem ênfase em fornecer uma ambiente de fácil acesso a informação para a tomada de decisão. O *Data Warehouse* constitui-se de uma base de dados que procura de maneira eficiente e flexível tratar de grande volume de dados e obter informações que auxiliem no processo de tomada de decisão (LOPES; OLIVEIRA, 2007). Alguns trabalhos já utilizaram um DWing no contexto de monitoramento de métricas e apresentaram bons resultados (CASTELLANOS et al., 2005) (FOLLECO et al., 2007) (SILVEIRA; BECKER; RUIZ, 2010)(MAZUCO, 2011) (RêGO, 2014).

Portanto, neste trabalho foram exploradas a utilização de métricas para o monitoramento de código-fonte para compreender e estabelecer possíveis relações existentes entre as mesmas no que diz respeito a vulnerabilidades e qualidade de software. A partir do estabelecimento de cenários buscou-se identificar oportunidades de utilização de métricas na melhoria contínua do desenvolvimento e, consequentemente, na qualidade interna do produto. Com o objetivo de facilitar a interpretação e evitar possíveis equívocos, que são baseados em análises errôneas sobre métricas isoladas, correlações inexistentes ou até mesmo a escolha de métricas inadequadas cujo problemas são discutidos em (CHIDAMBER; KEREMER, 1994), estes cenários foram compostos a partir da análise de relação entre métricas. Tal relação buscou evidenciar boas e más características de bom *design* de um projeto que impactam em vulnerabilidades no sistema. Para auxiliar no monitoramento e na tomada de decisão, foi explorado o uso de plataforma de monitoramento de código-fonte Mezuro e um ambiente de *DWing*.

1.1 Problema

Como utilizar métricas de código-fonte de maneira mais eficiente para melhor apoiar práticas de melhoria contínua da qualidade interna do software.

1.2 Objetivos

O objetivo deste trabalho consiste na proposta e implementação da técnica de construção de Cenários de Decisões para monitoramento do código-fonte que consiste na caracterização de determinado componente do sistema baseado na sua observação através de métricas, para orientar a aplicação de práticas e tomada de decisões sobre o software. Além disso, tem-se como objetivo do trabalho a proposta de alguns cenários através do uso desta técnica sobre características de segurança de software, a partir do estudo teórico sobre métricas de monitoramento de código-fonte. Por fim, o último objetivo consiste na realização de dois estudos de casos que visam reproduzir a técnica em duas ferramentas distintas de tomada de decisão, o Mezuro e um ambiente DWing.

Para alcançar os objetivos descritos, tem-se como objetivos específicos a realização das seguintes contribuições:

1.3 Contribuições

Contribuições Tecnológicas

1. **CT1** - Evolução da plataforma livre Mezuro de monitoramento de código-fonte:
   a) Evolução da arquitetura do Mezuro para melhorar sua modularização e flexibilidade.
   b) Evolução da configuração de métricas do Mezuro
   c) Evolução de mecanismos de visualização de software do Mezuro
2. **CT2** - Criação de um ambiente de *Data Warehousing* para monitoramento dos cenários de decisão no contexto de vulnerabilidade de software:
   a) Criação de modelo dimensional que, diferente de um modelo relacional utilizado para modelagem de banco de dados, permitiu melhor performance para processamento analítico dos dados.
   b) Implementação de mecanismos de extração, transformação e carregamento da base de dados a partir dos resultados das ferramentas de análise estática.
   c) Geração Cubo de dados para definição das agregações para a manipulação dos dados do *Data Warehouse*.

d) Configuração dos mecanismos de visualização do cubo e geração de relatórios.

Contribuições Científicas

1. **CC1** - Catalogar definições teóricas a respeito dos principais conceitos relacionados à vulnerabilidades de software.
2. **CC2** - Estudo teórico sobre a relação de vulnerabilidades de software com o *design*
3. **CC3** - Definição de cenários a partir de estudos teóricos para melhorar a interpretação e tomada de decisão sobre métricas estáticas de código-fonte.

1.4 Metodologia e Pesquisa

Dado o objetivo principal desta monografia, foram realizados estudos teóricos através de revisão bibliográfica para compreensão e definição dos conceitos básicos sobre características de *design* e de vulnerabilidades de software, referindo-se às contribuições científicas CC1 e CC2. Este estudo é o insumo principal para a concepção da técnica Cenário de Decisão.

Por fim, a partir da definição e discussão a respeito da técnica Cenário de Decisão, baseado principalmente na revisão bibliográfica sobre *design*, segurança e métricas estáticas de software, serão apresentados dois estudos de casos da evolução de ferramentas de monitoramento de código-fonte para suportar Cenários de Decisões. Os estudos foram realizados sobre as duas soluções propostas, Mezuro e ambiente DWing. Essa atividade está relacionado às contribuições tecnológicas deste trabalho de conclusão de curso, onde técnicas, métodos e padrões de Engenharia de Software puderam ser aplicados.

1.5 Organização do Trabalho

Esta monografia está dividida em mais outros 4 capítulos. A seguir, o leitor encontrará o Capítulo 2 onde são estruturados e discutidos os principais conceitos relacionados à *design* e segurança, principalmente relacionado à vulnerabilidades de software. No Capítulo 3 são introduzidos os principais conceitos de métricas de software, principalmente métricas estáticas de código-fonte, além de serem listadas e explicadas métricas de *design* e métricas de vulnerabilidades. Posteriormente, no Capítulo 4 serão apresentados o conceito de Cenários de Decisões e algumas propostas que podem ser utilizadas no monitoramento de projetos. No Capítulo 5 iremos apresentar dois estudos de casos relacionados à evolução de plataformas de monitoramento de código-fonte para a adoção de Cenários de Decisões. Neste mesmo Capítulo, serão definidas e descritas características, evoluções e configurações dos dois ambientes de tomada de decisão citados na Introdução:

a plataforma de monitoramento de código-fonte Mezuro e o *Data Warehousing*. Por fim, no Capítulo 6 serão realizadas as principais considerações e ponderações a respeito deste trabalho, além de apresentar possíveis passos futuros para a evolução da técnica Cenário de Decisão e das ferramentas envolvidas.

No fim desta monografia existem três apêndices que complementam e detalham os aspectos do presente trabalho. No Apêndice A são aprofundados em mais detalhes todos os conceitos estudos sobre *design* e segurança de Software. O Apêndice B apresenta detalhes técnicos sobre DWing, assim como Apêndice C apresenta os detalhes técnicos sobre o Mezuro.

2 Design e Segurança de Software

2.1 Design de Software

O desenvolvimento de software envolve diversas etapas que visam (i) o entendimento dos problemas a serem solucionados; (ii) o projeto de uma solução; (iii) a implementação desta solução; (iv) os testes sobre o produto, dentre outros passos. As metodologias mais conhecidas de desenvolvimento de software tais como o Processo Unificado e o *Extreme Programming* perpassam por essas etapas com diferentes focos e técnicas, ambos buscando a entrega de soluções em software que atendam aos problemas de seus clientes. Com isso, tanto do ponto de vista processo quanto do produto de software, prazos e custos não devem ser os únicos fatores considerados para reger um projeto de software. Em complementação, a qualidade deve ser preponderante para o sucesso do produto que, além de resolver o problema, deve fazê-lo adequadamente segundo os atributos e critérios de qualidade estabelecidos para o projeto.

O *design* ganha um grau de importância maior à medida que a complexidade dos softwares aumentam durante o desenvolvimento, pois suas consequências são diretas sobre os atributos de qualidade do software tais como flexibilidade, testabilidade, manutenibilidade, desempenho e segurança. Em projetos de software livre, por exemplo, esses atributos de qualidade são fatores fundamentais na atratividade de colaboradores para os projetos, onde observa-se uma correlação entre métricas de qualidade de código-fonte com a atratividade desses projetos (MEIRELLES et al., 2010).

Visando amenizar os riscos de se construir um sistema que não alcance seus objetivos, a arquitetura do software tem recebida uma maior atenção através dos métodos, práticas de desenvolvimento e estudos acadêmicos. O conjunto de decisões sobre as estruturas estáticas do sistema, hierarquia de módulos, descrição de dados, seleção de algoritmos, agrupamento e interface entre módulos podem ser previamente pensados a partir de modelos e documentação como também podem emergir a partir da aplicação de padrões de implementação e *refactorings* sobre o código. O mais importante é que a medida que novas funcionalidades são incorporadas, o *design* do código continue mantendo suas características, aplicando bons princípios e não sendo um impendimento para manutenção e evolução do software, comprometendo assim sua existência e utilização. Nesse sentido, o Engenheiro de Software tem a responsabilidade de desenvolver o software sem degradar sua arquitetura, respeitando padrões estabelecidos, evoluindo seu *design* à medida que implementa novas linhas de código e manter o código limpo e seguro para possibilitar que outros Engenheiros também compreendam e evoluam o software.

Nesta Seção apresentaremos os princípios reconhecidos de "bom *design*" e algumas formas como esses princípios podem ser aplicados no código-fonte. No Anexo A, identificamos como as decisões de design são observáveis em termos de *Bad Smells* e Código Limpo. Nesse contexto, pretendemos estudar e revisar como essas características observáveis possuem relação com métricas de código-fonte com objetivo de prover mecanismos que permitam ao Engenheiro de Software e gestores destinarem seus esforços na remoção de não-conformidades e evolução do software.

2.1.1 O Design e seus Princípios

O *design* do software consiste no conjunto de decisões importantes tomadas sobre a organização de um sistema de software que podem ser observadas e mapeadas em diferentes níveis de abstração no código-fonte e em outros produtos. Katki (1991) completa a definição explicando que *design* é tanto o processo de definição da arquitetura, módulos, interfaces e outras características de um sistema quanto o resultado deste processo. Para comunicação e documentação pode-se ter modelos que representem o *design* conceitual demonstrando, por exemplo, o estilo arquitetural adotado na aplicação. Por outro lado, esse modelo também é observável no código-fonte, assim como violação de restrições estabelecidas. Um nível mais detalhado do *design* consiste nas decisões de implementações existentes no código de uma classe do software que influenciam, por exemplo, na testabilidade desta classe. Assim, as decisões de *design* tratam problemas em diferentes níveis, desde da escolha do paradigma adequado para desenvolvimento do sistema até os padrões de nomenclatura a serem utilizadas no código.

Neste trabalho estamos essencialmente interessados nas decisões de projetos em nível de código e suas consequências no desenvolvimento do software. Assim, estamos tratando principalmente do trabalho realizado pelo Engenheiro de Software, de quais formas esse trabalho é realizado e como podemos apoiar e evoluir a atuação desse profissional para obtenção de bons resultados para projetos de software. Para o desenvolvimento de códigos com bom *design*, assim como defendido por Spinellis (2006), é importante que o Engenheiro de Software conheça os principais problemas de implementação conhecidos, características e princípios que compõem um paradigma e técnicas que permitem a aplicação desses princípios.

Neste trabalho estamos argumentando que os problemas do software podem ser definidos com as características indesejáveis que dificultam a manutenção e evolução do sistema ou até mesmo comprometem sua segurança. Um dos mais conhecidos e facilmente observável é a complexidade do software que depende das estruturas de dados, tamanho do sistema, algoritmos utilizados, complexidade das estruturas de controle e do fluxo de dados do software (BASILI; HUTCHENS, 1983). A complexidade afeta diretamente o quão compreensível um programa é, dificultando sua legibilidade e o encontro de *bugs* e

vulnerabilidades. Além disso, a complexidade afeta a testabilidade dos módulos, dificultando uma boa cobertura de testes que exercitem as estruturas do software. De modo geral, entendemos a complexidade como um fator de risco do software uma vez que a evolução do mesmo é comprometida por falta de legibilidade, flexibilidade e, muito provavelmente, pela falta de testes automatizados que suportem operações de *refactorings* no código, aumentando os riscos do projeto em termos de qualidade, custos e prazos.

Outras características que argumentamos como indesejadas para o software surgem a partir da atribuição inadequada de responsabilidades entre os módulos que o compõe. A baixa coesão surge quando um módulo é responsável pela realização de mais tarefas, concentrando mais dados e operações do que deveria. A baixa coesão gera problemas de falta de reusabilidade, aumenta a complexidade e dificulta a manutenção uma vez que não apoia a modularização adequadamente. O alto acoplamento também pode ser consequência da baixa coesão e ocorre quando há forte dependência entre os componentes do software. As consequências do alto acoplamento estão principalmente nas dificuldades de se manter e evoluir o software já que as modificações em um componente podem afetar outros. Além disso, algumas mudanças tendem a deixar de serem simples, pois ocorrem em mais de um lugar, o que pode dificultar, por exemplo, a criação de testes unitários, uma vez que os componentes tendem a não poder serem testados isoladamente (MARTENSSON; GRAHN; MATTSSON, 2005).

Os problemas elencados acima, assim como outras características indesejadas para o código-fonte são evitadas a partir da adoção de princípios reconhecidos e propostos através de diversos trabalhos para concepção de bons *designs* (MARTIN, 2002) (LAKOS, 1996) (DEMEYER; DUCASSE; O, 2002) (LIEBERHERR, 1996). Esses princípios podem ser genéricos, como características desejáveis em qualquer código-fonte, ou podem estar relacionados com um paradigma específico, acoplado as estruturas introduzidas pelo próprio paradigma. A seguir apresentamos os princípios gerais que indicamos ser levados em consideração pelos Engenheiros de Software no desenvolvimento de qualquer aplicação, pois são relacionados com a manutenibilidade, extensibilidade e testabilidade do software:

- **Reusabilidade** - O princípio de reusabilidade de código visa a implementação de componentes reutilizáveis, apoiados pela alta coesão e baixo acoplamento. A reusabilidade pode existir em diferentes níveis do *design*, podendo ser aplicada desde a introdução de funções reutilizáveis até serviços completos que oferecem operações autocontidas e reutilizáveis, sendo que a complexidade de implementação e as habilidades necessárias para conseguir a reusabilidade crescem proporcionalmente com o nível de abstração como discutido em (ALMEIDA et al., 2007).

  Este princípio possui impactos positivos na qualidade do software, assim como no custo e produtividade do projeto uma vez que menos código deve ser produzido e mantido, menor o esforço de teste, dentre outros benefícios (SAMETINGER, 1997).

O princípio de reusabilidade é estendido e aplicável na definição do conhecido princípio *DRY - Don't Repeat Yourself*[1] que é bastante enfatizado em *frameworks* modernos de desenvolvimento web como Rails[2] e Django[3].

- **Modularização** - O princípio de modularização visa a decomposição do sistema em estruturas lógicas bem definidas conceitualmente e fisicamente. A modularização é muito importante para os atributos de qualidade do software, principalmente manutenibilidade e testabilidade, uma vez que apoia a alta coesão, baixo acoplamento e a reusabilidade. Baldwin & Clark (2000) argumentam que um sistema bem modularizado permite o trabalho paralelo em diversas partes do produto, ameniza as dificuldades com a complexidade e esconde as incertezas e detalhes não necessários dentro dos módulos.

- **Abstração** - O princípio de abstração recomenda que um elemento que compõe o *design* seja representado apenas por suas características essenciais, provendo apenas as informações relevantes para sua utilização, além de permitir sua distinção com outros elementos por parte do observador (BARBOSA, 2009). A abstração é importante para a comunicação, compreensão e reutilização dos componentes.

- **Baixo acoplamento** - O acoplamento é uma característica natural do software sendo o grau de qualquer interação existente entre dois ou mais módulos. Essa característica é fundamental para a composição lógica do sistema, sendo que as interações dos componentes são necessárias para a implementação de funcionalidades que se complementam e juntas fornecem serviços complexos e completos. Beck & Diehl (2011) apresentam e discutem os diferentes tipos de acoplamentos e suas consequências na modularização do *design*. O princípio de baixo acoplamento ressalva a importância que esse acoplamento não seja forte ao ponto de dificultar a evolução de componentes que dependem de outros. O grau de acoplamento determina como é difícil fazer alterações em uma aplicação, assim como quão difícil é compreendê-la e testá-la, principalmente se os componentes acoplados são instáveis e sofrem constantes mudanças. Portanto, este princípio consiste na composição e modularização de serviços pouco acoplados com outros módulos a partir da melhor definição e distribuição de dados, de interfaces e responsabilidades.

- **Alta coesão** - De acordo com a Terminologia Padrão da IEEE[4] (IEEE, 1990), coesão é o grau com que cada tarefa realizada por um módulo está relacionado funcionalmente com o mesmo. Um módulo pode ser definido como coeso se todos as suas operações estão relacionadas com uma única abstração. O princípio da alta

[1] <http://c2.com/cgi/wiki?DontRepeatYourself>
[2] <http://rubyonrails.org/>
[3] <https://www.djangoproject.com/>
[4] Institute of Electric and Electronic Engineers

coesão sugere manter o maior nível de coesão possível no *design* de componentes do software. A alta coesão apoia a redução de complexidade do sistema, pois melhora seu entendimento conceitual, diminui as dependências de seus módulos e apoia a modularidade e reusabilidade, além de possuir uma relação direta com o baixo acoplamento, conforme estudado por Baig (2004).

- **Simplicidade** - Um dos maiores desafios no desenvolvimento de software é manter o *design* o mais simples possível, sendo esse o principal objetivo do princípio da simplicidade. A simplicidade dentro do software é obtida a partir da redução da complexidade de seus módulos, desde a escolha de algoritmos até suas interfaces de comunicação. Os benefícios deste princípio consiste na solução dos problemas inerentes a complexidade do software, discutidos anteriormente nesta Seção. A simplicidade do software é também conhecida através do princípio *KISS - Keep It Simple, Stupid* que enfatiza que a principal característica do *design* deve ser a simplicidade.

No Apêndice A tem-se uma discussão mais aprofundada sobre as diferentes práticas e técnicas que podem ser utilizadas para a aplicação dos princípios de bom *design* apresentados.

2.2 Segurança de Software

A segurança de software está relacionada com o contínuo processo de manter a confidencialidade, integridade e disponibilidade nas diversas camadas que o compõe, sendo considerado parte dos requisitos não-funcionais do sistema. Independentemente da criticidade do sistema, a segurança em software deve ser tratada com prioridade dentro do ciclo de vida de desenvolvimento do software. Aggarwal e colaboradores (2002) citam que o custo e esforço gastos na segurança do software são bem altos, podendo chegar a 70% do esforço total de desenvolvimento e suporte do software.

Problemas de segurança são recorrentes em diversos tipos de sistemas podendo gerar perdas materiais e humanas em diferentes proporções. Vulnerabilidades em softwares são as maiores causas de infecção de computadores das corporações e perda de dados importantes segundo a pesquisa Global Corporate IT Security Risks 2013 conduzido por B2B International em colaboração com Kaspersky Lab (LAB; INTERNATIONAL, 2013). Esse estudo aponta que aproximadamente 85% das empresas reportaram incidentes internos de segurança de TI. Mesmo com o grande esforço destinado a aspectos de segurança, tais problemas são difíceis de solucionar, pois a Engenharia de Segurança de Sistemas está em fase intermediária de desenvolvimento (PáSCOA, 2002). Gandhi e colaboradores (2013) realçam as dificuldades de se detectar vulnerabilidades no estágio operacional do software, pois os problemas de segurança não são endereçados ou suficientemente conhecidos nas fases iniciais do desenvolvimento de software.

Formalmente, uma vulnerabilidade pode ser definida como uma instância de uma falha na especificação, desenvolvimento ou configuração do software de tal forma que a sua execução pode violar políticas de segurança, implícita ou explícita (KRSUL, 1998). Vulnerabilidades podem ser maliciosamente exploradas para permitir acesso não autorizado, modificações de privilégios e negação de serviço. A exploração maliciosa de vulnerabilidades em grade parte são realizadas através de *Exploits*, ferramentas ou scripts desenvolvidos para este propósito, que se baseiam extensivamente nas vulnerabilidades mais comuns tal como *buffer-overflow*.

Vulnerabilidades podem existir em diferentes níveis de um sistema, podendo, portanto, gerar problemas com diferentes proporções. Os níveis mais comuns suscetíveis a existência de vulnerabilidades são:

- **Hardware** - Vulnerabilidades relacionadas ao hardware de sistemas que estão expostos a umidade, poeira, calor, locais inseguros, dentre outros fatores físicos relacionados ao local onde se encontra a infraestrutura de TI.
- **Software** - Vulnerabilidades relacionadas às estruturas internas do software assim como aos dados que são acessados e processados. No geral, podem ser exercitados a partir de interações com o usuário não esperadas ou não validadas.
- **Rede** - Vulnerabilidades relacionadas aos componentes da rede, tanto físicos (cabos, *switches*) quanto em software (protocolos, dados). Este tipo de vulnerabilidade também está relacionada à falhas na comunicação como linhas de comunicação não protegidas, compartilhamento de informações com não interessados.
- **Humana** - Vulnerabilidades relacionadas à processos que envolvem pessoas e níveis de acesso.
- **Organizacional** - Vulnerabilidades relacionadas problemas em nível organizacional, principalmente relacionado à falta de políticas, auditorias e planos adequados.

No presente trabalho, estamos interessados essencialmente em vulnerabilidades de software. Mais especificamente, procuramos uma abordagem que facilite o tratamento destas vulnerabilidades dada sua importância e suas consequências. Nesse sentido, faz-se necessário compreender quais são as ocorrências conhecidas de falhas de segurança em software e com quais vulnerabilidades estas falhas se relacionam.

Vulnerabilidades de software são, na maior parte das vezes, causadas pela falta ou imprópria validação das entradas realizadas pelo usuário. Essas condições indesejáveis são usadas por usuários maliciosos para injetar falhas e códigos no sistema que os permitam executar seus próprios códigos e aplicações (JIMENEZ; MAMMAR; CAVALLI, 2009). McGraw e colaboradores (2004) afirmam que 50% dos problemas de segurança surgem

no nível de *design*. Poucas ações específicas são tomadas por Engenheiros de Software para manter a segurança no desenvolvimento de novas funcionalidades ou até mesmo na realização de *refactorings*. Em outras palavras, muitas vezes o desenvolvedor de software pode estar inserindo vulnerabilidades no código que podem ser exploradas por usuários maliciosos ou, acidentalmente, por usuários comuns. Mesmo os Engenheiros de Software que realizam testes unitários automatizados tendem a não exercitar estas vulnerabilidades, pois no geral testam principalmente as condições de uso padrão do software, enquanto deveriam explorar melhor o comportamento do software à interações indesejadas (VRIES, 2006).

Algumas das vulnerabilidades mais comuns estão listada abaixo:

- ***Buffer overflow***: caso comum de violação de segurança da memória que ocorre normalmente quando dados são escritos em *buffers* de tamanhos fixos e ultrapassam os limites de memória definidos para eles. Como consequência, pode gerar mal funcionamento do sistema, já que o dado escrito pode corromper os dados de outros *buffers* ou até mesmo de outros processos, erros de acesso à memória, resultados incorretos e até mesmo interromper a execução do software. Esta vulnerabilidade também pode ser explorada para injetar códigos maliciosos, alterando a ordem de execução do programa para que o código malicioso tome controle do sistema. Algumas linguagens de programação oferecem mecanismos de proteção contra acesso ou sobrescrita da dados em qualquer parte da memória indesejada. Contudo, *buffer ovewflows* ocorrem principalmente com programas escritos em C e C++ que não realizam a verificação automática se o dado a ser escrito em um *array* cabe dentro dos limites de memória do mesmo.
- ***Dangling pointer***: vulnerabilidade de violação de segurança da memória que ocorre quando um ponteiro não aponta para um objeto ou destino válido. Esta vulnerabilidade acontece ao se deletar um objeto ou desalocar a memória de um ponteiro sem modificar, entretanto, o valor deste ponteiro. Como resultado, o ponteiro ainda aponta para a mesma posição de memória que, por sua vez, já não está mais alocada para esse processo. Como consequência, o sistema operacional pode realocar essa posição de memória para outro processo que, se acessado novamente pelo primeiro processo, irá conter dados inconsistentes com o esperado. Em C e C++ esta vulnerabilidade existe também quando o ponteiro de um endereço de memória é declarado somente no escopo de um função e retornado por esta função. Muito provavelmente esse endereço de memória será sobrescrito na pilha de alocação do processo pela chamada de funções posteriores. Além de inconsistência de dados, esta vulnerabilidade pode ainda ser a causa de quebras de programas, como falhas de segmentação e pode ser explorada por ataques de injeção de código (AFEK; SHARABANI, 2007). Algumas linguagens de programação como Java, Python e

Ruby possuem um mecanismo de gerenciamento de destruição de objetos chamado *Garbage Collector*[5].

- ***Strings* formatadas não-controladas**: vulnerabilidade decorrida do tratamento inadequado das entradas do usuário sobre o software que, quando explorada, o dado submetido por uma *string* de entrada é avaliado como um comando pela aplicação. Uma *string* formatada pode conter dois tipos de dados: caracteres imprimíveis e diretivas de formatação de caracteres. Na linguagem C, funções de *strings* formatadas tal como o *printf* recebem um número variável de argumentos, dos quais uma *string* formatada é obrigatória. Para acessar o restante dos parâmetros que a chamada da função colocou na pilha, a função de *string* formatada analiza a sequência de formatação e interpreta as diretrizes do formato a medida que realiza sua leitura (LHEE; CHAPIN, 2002).

- ***SQL Injection***: vulnerabilidade presente em aplicações que aceitam dados de uma fonte não confiável, não os validando adequadamente e os usando posteriormente para construção de *queries* dinâmicas de SQL para comunicação com o banco de dados da aplicação. Todos os tipos de sistemas que incorporam SQL estão sujeitos a esta vulnerabilidade, apesar de serem mais comuns em aplicações WEB. Como consequência da exploração desta vulnerabilidade tem-se a perda de confiabilidade e quebra de integridade dos dados de uma base de dados. Em alguns casos, a exploração de *SQL Injection* pode permitir ao atacante levar vantagens através da persistência de informações e geração de conteúdos dinâmicos em páginas web (DOUGHERTY, 2012).

2.2.1 Classificações e taxonomias de vulnerabilidades

O primeiro passo para que o desenvolvedor consiga cuidar de vulnerabilidades no código fonte é conhecer quais são os problemas mais comuns existentes em softwares e como os atacantes utilizam estas vulnerabilidades para falhar o sistema. Existem muitos tipos de vulnerabilidades, e com o passar do tempo, novas vulnerabilidades são descobertas e novos *exploits* são criados por atacantes. Em meio a essa diversidade, a classificação de vulnerabilidades representa um grande desafio, porém, já houve vários avanços na área com o objetivo de enumerar e catalogar estas vulnerabilidades.

O CVE (*Common Vunerabilities and Exposures*) é um projeto criado pelo MITRE[6] com o objetivo de enumerar as vulnerabilidades descobertas pela comunidade para facilitar o compartilhamento de informações. Como é mostrado em (MARTIN, 2001), antigamente cada organização nomeava de sua maneira uma vulnerabilidade, de forma que uma mesma vulnerabilidade era referenciada de maneira completamente distinta entre as organizações.

[5] <http://www.informit.com/articles/article.aspx?p=30309&seqNum=6>
[6] <http://www.mitre.org/>

A enumeração realizada pelo CVE é feita por uma junta de especialistas de segurança. Essa equipe nomeia, descreve e referencia cada nova ameaça. As novas ameaças são reportadas pela comunidade. Após estudo e aprovação, essa ameaça passa a incluir a lista da CVE, cujo o identificador é representado da seguinte forma: CVE-2014-0002. O valor 2014 significa que a CVE foi criada em 2014, e o numero 0002 se refere ao numero sequencial atribuído a essa CVE dentre todas que foram aprovadas nesse ano.

Surgiram propostas além do CVE para a classificação e agrupamento de vulnerabilidades, que são chamadas de taxonomias. O trabalho de Malerba (2010) descreve as as primeiras propostas taxonômicas, como os trabalhos de Landwher em 1992(*A toxonomy of Computer Security Flaws*) e Aslam em 1997 (*Use of a taxonomy of Security Faults*). Porém, consideramos mais relevante abordar mais detalhadamente sobre as taxonomias mais recentes, que são utilizadas no projeto CWE, iniciativa semelhante a CVE que será abordada mais adiante. As principais taxonomias são:

- PLOVER (*Preliminary List Of Vulnerability Examples for Researchers*)
- CLASP (*Comprehensive, Lightweight Application Security Process*)
- *Seven Pernicious Kingdom*

O PLOVER foi criado em 2005 pelo MITRE em parceria com o DHS[7] e o NIST[8]. É um documento que lista mais de 1400 exemplos reais de vulnerabilidades identificados pelo CVE. Trata-se de um framework conceitual que descreve as vulnerabilidades em diversos níveis de detalhe. O PLOVER também fornece uma série de conceitos que podem ajudar na comunicação e discussões à respeitos de vulnerabilidades.

Foram definidas 28 categorias de mais alto nível para a categorização de vulnerabilidades. Algumas dessas são:

- BUFF: inclui vulnerabilidades de Buffer Overflow, formatação de strings, etc;
- CRIPTO: inclui vulnerabilidades relacionadas a criptografia;
- SPECTS: inclui vulnerabilidades que ocorrem e tecnologias específicas, como a injeção de SQL e XSS

A lista completa das categorias e mais detalhes sobre o PLOVER podem ser encontrados em (CHRISTEY, 2006).

[7] US. Departament of Homeland Secutity
[8] National Institute of Technology

O CLASP não é apenas uma taxonomia de categorização de vulnerabilidades, mas sim um processo que busca melhorar a segurança de softwares. No de diz respeito a classificação, o CLASP define 6 categorias alto nível que incluem 104 tipos de vulnerabilidades, que são:

- Erros de tipo e de Range
- Problemas no ambiente
- Erros de sincronização e de temporização
- Erros de protocolo
- Erros de lógica
- *Malware*

Exemplificando, as vulnerabilidades do tipo *Buffer Overflow* se encaixariam na categoria de erro de tipo e de range, pois nesse tipo de vulnerabilidade é permitido a escrita de uma informação além o limite do buffer. A vulnerabilidade do tipo injeção de SQL também se encaixaria nessa categoria, pois esse tipo de vulnerabilidade ocorre quando não há validação no tipo de informação fornecida pelo usuário. Mais detalhes podem ser vistos no site do CWE[9].

A taxonomia Seven Pernicious Kingdoms é a que possibilita melhor entendimento ao desenvolvedor, e é até utilizada como base em uma das visões das CWEs denominada *Development View*[10]. Utiliza os conceitos da biologia de Reino e Filo. Nessa taxonomia, o Reino é a classificação mais abrangente e o Filo é uma subdivisão do Reino. Embora o nome sugere sete, possui oito reinos sendo que sete reinos são dedicados a vulnerabilidades de código fonte e um reino referente a aspectos de configuração e ambiente. São eles:

- **Validação e representação de entrada**: inclui erros de *Buffer Overflow*, injeção de SQL, XSS, etc;
- **Abuso de API**: Ocorre quando uma função que chama outra função assume certas condições que não estão garantidas pela rotina chamada.
- ***Features* de segurança**: está relacionado ao uso correto de peças chaves em segurança de código como criptografia, autenticação, gerenciamento de privilégios, etc;

[9] <http://cwe.mitre.org/about/sources.html>
[10] <http://cwe.mitre.org/data/graphs/699.html>

- **Tempo e estado**: está relacionado a erros de paralelismo, sincronização e uso de informação.

- **Erros**: está relacionado erros oriundo da falta de tratamento de erros da aplicação.

- **Qualidade de código**: são erros originados pela falta de qualidade no código fonte. Geralmente acontecem quando são utilizadas más praticas de programação que podem gerar colapso no sistema, como por exemplo, não desalocar recursos não utilizados pode gerar *Memory Leak*[11];

- **Encapsulamento**: são erros relacionados ao não estabelecimento de limite de acesso aos componentes do sistema

- **Ambiente**: são erros que estão relacionados a fatores externos do software, que não estão no escopo deste trabalho.

A ideia dessa taxonomia foi criar reinos bem amplos para que novos filos fossem inseridos em seu lugar correto, porém a taxonomia proposta está aberta para a inserção de novos reinos, caso necessário. Mais informações e detalhes a cerca dos filos que incluem cada reino pode ser encontrado em (TSIPENYUK; CHESS; MCGRAW, 2005)

Com esse cenário apresentado acima, em que temos a CVE como projeto de enumerar as vulnerabilidades encontradas e as taxonomias elaboradas por diversos trabalhos com o objetivo de classificar e dar mais informações sobre a vulnerabilidade, ainda haviam a necessidade das empresas e organizações de utilizarem o uma terminologia padrão para listar e classificar vulnerabilidades de software, gerando uma linguagem unificada e uma base para ferramentas e serviços de medição dessas vulnerabilidades. Para essa necessidade foi elaborado o projeto CWE.

O CWE é uma lista formal de vulnerabilidades comuns de software (*Common Weakness Enumeration*). Tem como objetivo estabelecer uma linguagem comum para descrever vulnerabilidades de software no *design*, arquitetura ou no código; servir de base para ferramentas de análise de cobertura de segurança de código, dessa forma é possível saber quais vulnerabilidades as ferramentas conseguem capturar; e prover uma base de informações padrão a respeito de como identificar, mitigar e prevenir uma certa vulnerabilidade. Dessa forma, diferente da enumeração fornecida pela CVE, O CWE também inclui detalhes sobre a vulnerabilidade e cria uma classificação baseadas nos trabalhos desenvolvidos sobre taxonomias apresentados neste trabalho e outros que não foram apresentados. Além disso, observou-se que o CVE busca enumerar casos de vulnerabilidades reais na comunidade, especificando a maneira de como a vulnerabilidade foi explorada. Já o projeto CWE, o termo "vulnerabilidade"está mais relacionado a fraqueza de software,

[11] Falta de memória livre para uso no sistema

que se refere a códigos e práticas de programação que podem oferecer risco a segurança da aplicação, não indicando como a vulnerabilidade pode ser explorada e sim indicando quais características podem tornar o software vulnerável.

2.2.2 Princípios de segurança

Como visto no estudo sobre a classificação e taxonomias, existem muitas vulnerabilidades de software que são passíveis de exploração por atacantes. Nesse sentido, ganha-se importância para o desenvolvimento de softwares mais seguros que os Engenheiros de Software construam códigos com qualidade suficiente que os permitam identificar, corrigir e evitar a inserção de vulnerabilidades. Tendo-se o conhecimento de quais são as principais vulnerabilidades existentes, os Engenheiros podem tratar essas vulnerabilidades desde as primeiras fases de *design* e desenvolvimento código-fonte, seguindo até o fim do ciclo de vida do desenvolvimento do software. Assim como os desenvolvedores programam aplicando ao código princípios de *design*, devem evoluir o código aplicando princípios de *design* seguro tais quais os apresentados por outros trabalhos (SALTZER; SCHROEDER, 1975b) (BISHOP, 2003) (VIEGA; MCGRAW, 2002) (ALSHAMMARI; FIDGE; CORNEY, 2009).

Dentre os princípios de segurança que os Engenheiros de Software podem aplicar no *design* de seus programas, introduziremos aqui os seguintes princípios:

- ***Least privilege***: este princípio sugere que o usuário deve ter somente os direitos necessários para completar suas tarefas (BISHOP, 2003). Em termos de *design* de classes, significa que o *design* mais seguro é aquele cujos métodos realizam o menor número de ações possíveis (ALSHAMMARI; FIDGE; CORNEY, 2009).

- ***Reduce attack surface***: este princípio tem como objetivo limitar o acesso a dados não permitidos. Pode-se aplicar este princípio reduzindo-se a quantidade de código executável, com *design* prezando por menor números métodos de acesso (públicos) e menor número de parâmetros possíveis que possam afetar atributos privados para realização de uma tarefa. Pode-se também buscar eliminar serviços que são usados somente por poucos clientes.

- ***Defend in depth***: este princípio sugere que os mecanismos de defesas devem ser aplicados na maior extensibilidade possível, mesmo que isso gere redundância. O princípio *defend in depth* busca defender o sistema contra qualquer possível ataque através de implementação de métodos ou mecanismos diferentes de tratamento destes ataques. O *design* em camadas facilita sua implementação, pois permite dividir os métodos de defesa de acordo com as responsabilidades de cada camada. Como ponto negativo, a implementação de vários mecanismos de defesa pode acrescentar

complexidade ao software, aumentando riscos de inserção de outras vulnerabilidades e a dificuldade de encontrá-las.

- ***Fail securely***: este princípio está relacionado ao controle das falhas que possam ocorrer na aplicação. As possíveis falhas existentes em um software devem ser exploradas e tratadas para que o software esteja preparado para responder a estas falhas adequadamente, sem gerar alarmes, quebrar a aplicação e principalmente abrir espaços para mais ataques maliciosos. Com a aplicação do princípio *fail securely* tem-se a identificação e tratamento de erros, a inserção de mecanismos de respostas que facilitam a utilização correta do software e que permita que estes comportamentos sejam testados pelos desenvolvedores. Outra consequência positiva da aplicação deste princípio é que o princípio *defend in depth* é apoiado uma vez que a identificação de possíveis erros reforça a modularização e separação dos métodos de tratamento dos mesmos.

- ***Economy of mechanism***: princípio que se refere a manter o código que implementa mecanismos seguros menor e o mais simples possível. Este princípio é de suma importância para o tratamento de vulnerabilidades no desenvolvimento do software, pois a simplicidade é fundamental para que os Engenheiros de Software possam encontrar erros e corrigi-los. A medida que a complexidade aumenta, os módulos inseguros do software tendem a ficarem ocultos e mais difíceis de serem testados. A aplicação de técnicas de programação do XP tais como *refactoring*, *test-driven development* e programação em pares são fundamentais para alcançar os objetivos deste princípio. Este princípio está extremamente ligado ao princípio de design *KISS - Keep It Simple, Stupid!*, pois ambos enfatizam que evitar a complexidade significa evitar problemas (VIEGA; MCGRAW, 2002)

- ***Mediate completely***: princípio que defende que todos os acessos a quaisquer objetos devem ser verificados para garantir se há permissões para realizar tal ação. Se em algum momento for solicitado a leitura de um objeto, o sistema deve verificar se o sujeito tem permissão de leitura. Caso tenha, deve prover somente os recursos necessários para realização das tarefas que interessa a este sujeito. Essa operação deve se repetir todas as vezes que a requisição ao objeto for feita, não somente na primeira vez (BISHOP, 2003).

- ***Separation of duties***: princípio relacionado a separação de interesses dentro dos métodos e mecanismos de segurança do software. A OWASP[12] sugere a separação entre as entidades que aprovam a ação, entidades que realizam a ação e entidades que monitoram a ação [13]. Este princípio está diretamente relacionado com os princípios de *design* orientado à objetos tais como coesão e separação de interesses. Por

[12] Open Web Application Security Project
[13] <https://www.owasp.org/index.php/Separation_of_duties>

outro lado, também mantém forte relação com o princípio de *design* seguro *Economy of mechanism*, pois proporciona código mais limpo e que podem ser mantidos separadamente.

Um estudo mais aprofundado sobre a relação existente entre os princípios de segurança e de *design* pode ser encontrado no Apêndice A, na Seção A.2.

3 Métricas de código-fonte

Tanto a existência de um bom *design* quanto a de testes automatizados que exercitem as funcionalidades são características desejáveis em projetos de software. Boa parte das técnicas modernas da Engenharia de Software são voltadas para desenvolvimento com *design* que proporcione simplicidade, manutenibilidade e testabilidade. Mesmo que a maior parte dessas técnicas tenham sido disseminadas a partir do advento dos Métodos Ágeis e do Software Livre, cujo foco central está em atividades relacionadas ao código-fonte, elas são aplicáveis independentemente da metodologia de desenvolvimento utilizada (MEIRELLES, 2013). A valorização por softwares que atendam esses parâmetros de qualidade deve-se ao fato de sempre que o Engenheiro de Software está escrevendo novas linhas de código, um tempo significativo é gasto por ele na leitura e entendimento do código existente, muitas vezes desenvolvidos por outros Engenheiros. Martin (2008) destaca que o código-fonte deve ser escrito para ser entendido principalmente por pessoas, e não pela máquina.

Nesse sentido, o monitoramento da qualidade de código-fonte pode ser utilizado para apoiar a utilização de técnicas de desenvolvimento que visam a melhoria contínua do código. Além disso, as métricas de código-fonte são muito importantes para projetos de software, pois estas podem ser utilizadas tanto como ferramenta para gestão do projeto quanto como referência técnica para tomada de decisões sobre o código-fonte.

Uma métrica, no âmbito da Engenharia de Software, provê uma forma de medir quantitativamente atributos relacionados as entidades do software e do processo de desenvolvimento. Assim, métricas são importantes ferramentas para avaliação da qualidade do código-fonte produzido e acompanhamento do projeto. Meirelles (2013) destaca que, com métricas de software, propõe-se uma melhoria de processo de gestão com identificação, medição e controle dos parâmetros essenciais do software.

Métricas de monitoramento de código-fonte possuem natureza objetiva e foram inicialmente concebidas para medir o tamanho e a complexidade do software (HENRY; KAFURA, 1984)(TROY; ZWEBEN, 1981)(YAU; COLLOFELLO, 1985). Outras métricas surgiram para avaliar softwares que utilizam paradigmas específicos, não sendo aplicáveis a qualquer tipo de software. Por exemplo, métricas Orientada a Objetos (OO) são usadas para avaliar sistemas orientados a objetos (SÿSTA, 2000). Métricas OO são destinadas, portanto, para avaliar a coesão de classes, as hierarquias de classes existentes, nível de acoplamento entre classes, reuso de código, dentre outras características.

Algumas características importantes ajudam a classificar as métricas de código-fonte. Assim, podemos classificá-las como estáticas e dinâmicas. Como o próprio nome diz,

métricas estáticas capturam propriedades estáticas dos componentes de software e não necessita que o software seja executado para que seus valores sejam coletados. Por outro lado, métricas dinâmicas refletem características chaves tais como dependência dinâmica entre os componentes em tempo de execução do software.

No contexto desta monografia estamos interessados principalmente na análise estática de códigos-fontes. Nesse sentido, a análise estática é definida como o processo de avaliar um sistema ou seus componentes baseados em suas formas, estruturas, conteúdo ou documentação que podem gerar insumos para compreensão da qualidade de *design* do código assim como endereçar suas principais vulnerabilidades, podendo ser realizada sobre módulos e até mesmo sobre códigos ainda não finalizados (BLACK, 2009). Esta análise pode ser realizada manualmente, tal como é feita em inspeções de código e com *pair programming* ou de maneira automatizada através de ferramentas desenvolvidas para tal fim.

As métricas de software também podem ser classificadas quanto ao método de obtenção. Métricas primitivas podem ser diretamente coletadas refletindo um valor observável de um atributo, sendo raramente interpretadas independentemente. Por outro lado, métricas compostas são obtidas a partir da relação de uma ou mais métricas, derivada, por exemplo, a partir de uma expressão matemática.

Entretanto, as definições de métricas adequadas para o acompanhamento do projeto, dimensionamento do software e principalmente para a aferimento da qualidade do código-fonte são tarefas que aumentam a complexidade de adoção de métricas em projetos de software, assim como destacado por Rakić e Budimac (2011). Isso se deve a diversos fatores: à grande quantidade de métrica existentes; pouca aderência de algumas métricas com a realidade; diversas formas de interpretação de dados; dificuldades de definir parâmetros para comparação; poucos recursos de visualização de dados; coleta de dados não automatizados ou difíceis. Fenton e Pfleeger (1998) definem características desejáveis de métricas que orientam a escolha das mesmas enquanto outros autores estudam formas de viabilizar a utilização de métricas pelos desenvolvedores em geral(MEIRELLES, 2013; ALMEIDA; MIRANDA, 2010).

O presente trabalho visa relacionar métricas de desenvolvimento de software com objetivo de definir configurações para estabelecer cenários que representem o estado da qualidade do software. Dessa forma, espera-se reduzir as dificuldades de utilização de métricas de código fonte, tanto para o acompanhamento gerencial quanto para a tomada de decisões de *design* por desenvolvedores baseada em evidências. Para tanto, nas próximas Seções serão apresentados estudos realizados sobre métricas de monitoramento de código-fonte para sistemas orientados à objetos e métricas para avaliação de vulnerabilidades do software.

3.1 Métricas Estáticas de *Design* de Software

Nesta Seção iremos apresentar um conjunto de métricas de código-fonte que estão diretamente relacionadas ao *design* de software. Neste conjunto de métricas estamos incluindo métricas que medem atributos do software tais como tamanho, complexidade assim como características específicas relacionadas à orientação à objetos. Portanto, métricas de *design* de software devem ser compreendidas como um conjunto de métricas que medem atributos do código-fonte que permitam a avaliação de produtos de software.

A escolha de métricas para mensurar os atributos de *design* se torna complexa, pois existem inúmeras propostas de métricas diferentes destinadas a medir os mesmo atributos, principalmente sobre tamanho e complexidade. Li & Cheung (1987), por exemplo, referencia e compara 31 métricas diferentes de complexidade. Entretanto, não está no escopo deste trabalho realizar uma comparação detalhada sobre métricas semelhantes. As métricas que iremos apresentar a seguir foram selecionadas devido a sua vasta utilização em estudos científicos referenciados neste trabalho e devido a sua existência em boa parte das ferramentas extratoras de métricas de código-fonte. Primeiramente serão apresentadas métricas relacionados a tamanho.

- **LOC (*Lines of Codes* - Linhas de Código)**: LOC calcula o número de linhas executáveis, desconsiderando linhas em branco e comentários. Esta é a métrica de tamanho mais comum. Entretanto, deve ser cuidadosamente usada e composta, pois os parâmetros de comparação não devem ser os mesmos quando se varia a linguagem e estilo de programação.

- **(*Total Number of Modules or Classes* - Número Total de Módulos ou Classes)**: Esta métrica mensura o tamanho do software baseados na quantidade de módulos e classes, sendo menos sensível por linguagens de programação, nível de desenvolvedores e estilo de codificação (MEIRELLES, 2013).

- **AMLOC (*Average Method LOC* - Média de Número de Linhas de Código por Método)**: Esta métrica avalia a distribuição do código entre os métodos, sendo uma importante indicador de coesão, reutilização e outras características importantes. Entretanto, assim como a LOC, deve ser avaliada considerando-se a linguagem e estilo de programação.

As métricas de tamanho são muito importantes para a composição de novas métricas que permitam avaliar outras características do código-fonte. A seguir são apresentadas algumas métricas que avaliam atributos estruturais, sendo muito importantes para a compreensão do nível da qualidade do *design*, principalmente manutenibilidade, flexibilidade e testabilidade.

- **NOA (*Number of Attributes* - Número de Atributos)**: NOA calcula o número de atributos de uma classe, sendo bastante importante para avaliar a coesão de uma classe.

- **NOM (*Number of Methods* - Número de Métodos)**: Esta métrica se refere ao tamanho de uma classe medindo a quantidade de operações de uma classe. Sua interpretação pode ser complicada. Um número excessivo de métodos pode representar falta de coesão e de potencial de reusabilidade da classe. Por outro lado, pode representar uma classe bem estruturada com operações bem definidas. Entretanto, a avaliação isolada desta métrica não permite este tipo de afirmação.

- **NPA (*Number of Public Attributes* - Número de Atributos Públicos)**: NPA mede basicamente o encapsulamento de uma classe. Independente da linguagem, é desejado que este valor seja o mais baixo possível, pois é recomendado que a manipulação de atributos de uma classe sejam realizados por métodos de acesso e operacionais.

- **NPM (*Number of Public Methods* - Número de Métodos Públicos)**: Esta métrica é muito importante para a compreensão da abstração da classe, pois mede diretamente o tamanho da interface de acesso à mesma. NPM pode ser melhor utilizada para compreender o potencial de reusabilidade e coesão de uma classe do que a métrica NOM isoladamente.

- **ANPM (*Average Number of Parameters per Method* - Média de Parâmetros por Método)**: Essa métrica calcula a média de parâmetros dos métodos da classe, onde não se deseja um valor alto.

- **MNPM (*Maximum Number of Parameters per Method* - Número Máximo de Parâmetros por Método)**: Essa métrica corresponde à maior ocorrência de número de parâmetros dos métodos de uma classe.

- **DIT (*Depth of Inheritance Tree* - Profundidade da Árvore de Herança)**: Está métrica consiste no número de classes ancestrais da classe em análise, sem considerar heranças provindas de *frameworks* ou bibliotecas. Para linguagens com herança múltipla, o valor desta métrica é o DIT da maior hierarquia.

- **NOC (*Number of Children* - Número de Filhos)**: NOC consiste no número de filhos direto de uma classe.

- **ACCM (*Average Cyclomatic Complexity per Method* - Média da Complexidade Ciclomática por Método)**: Esta métrica mede a complexidade média dos métodos de uma classe, baseando-se na complexidade dos fluxos de controle existente no método.

As últimas métricas que serão apresentadas nesta Seção buscam medir características discutidas anteriormente tais como coesão e acoplamento.

- **RFC (*Response For a Class* - Resposta de uma Classe)**: Esta métrica mede a complexidade de uma classe contando o número de métodos que um objeto de uma classe pode invocar, tanto métodos internos quanto de outras classes. É, portanto, o número de métodos que podem ser executados em resposta a uma mensagem recebida por um objeto da classe, sendo a quantidade de métodos da classe somado à quantidade de métodos invocados por cada método da classe. Em resumo, apresenta por quantos caminhos uma classe está conectada à outras classes.

- **ACC (*Afferent Connections per Class* - Conexões Aferentes por Classe)**: Esta métrica mede a conectividade de uma classe a partir da contagem de quantas classes do sistema acessam um atributo ou método da classe em análise. Caso o valor de ACC de uma classe seja alto, modificações em sua estrutura podem afetar mais classes.

- **CBO (*Coupling Between Objects* - Acoplamento Entre Objetos)**: Esta métrica calcula de quantas classes a classe em análise depende, sendo a recíproca da ACC.

- **COF (*Coupling Factor* - Fator de Acoplamento)**: Esta métrica é a razão entre o número acoplamento existente que não sejam provindos de herança e do número total de possíveis acoplamentos. O máximo de acoplamento possível acontece quando todas as classes estão e são acopladas com as outras classes do projeto.

- **LCOM4 (*Lack of Cohesion in Methods* - Ausência de Coesões em Métodos)**: Esta métrica calcula o número de componentes conectados em uma classe. Um componente conectado consiste em um conjunto de métodos relacionados. Dois métodos são relacionados se ambos acessam as mesmas variáveis da classe ou um método invoca ao outro.

3.2 Métricas Estáticas de Segurança

Atualmente existem várias ferramentas de análise estática que detectam vulnerabilidades no código fonte. Essas ferramentas utilizam de diversas técnicas de detecção e buscam encontrar tipos específicos de vulnerabilidades. Como visto na Seção 2.2, o projeto CWE busca definir e classificar vulnerabilidades descobertas pela comunidade levando em consideração os detalhes de como essa vulnerabilidade ocorre no código para a geração de um erro. Considerando como referência o projeto CWE, vamos tomar como exemplo o erro de *Buffer Overflow*. Existem várias CWEs que especificam uma maneira

diferente de se ter o erro de *Buffer Overflow*, dessa forma, cada maneira diferente compõe uma vulnerabilidade específica. As ferramentas de análise estática buscam encontrar essas vulnerabilidades específicas e quantificar o seu número de ocorrências.

O uso de ferramentas de análise estática de código para a identificação de vulnerabilidades, como bem explanado em (CHESS; WEST, 2007), é uma alternativa muito interessante para o desenvolvedor, visto que é muito difícil para alguém que não tem muito conhecimento a respeito de vulnerabilidades de código fonte saber se está inserindo ou não uma vulnerabilidade no software. O trabalho de Aranha (2012) mostra claramente a importância do uso de ferramentas de análise estática para a identificação de vulnerabilidades, visto que uma vulnerabilidade identificada no software da urna eletrônica utilizadas em votações no Brasil podia ter sido facilmente identificada e tratada se fossem utilizadas ferramentas de análise estática de código durante o desenvolvimento.

Nesse contexto, em termos de métricas, ferramentas de análise estáticas podem fornecer as seguintes métricas relacionadas a vulnerabilidades:

- Número total de vulnerabilidades no projeto
- Número de vulnerabilidades por arquivo
- Número de vulnerabilidades por função
- Quantidade de uma vulnerabilidade específica no projeto
- Quantidade de uma vulnerabilidade específica por arquivo
- Quantidade de uma vulnerabilidade específica por função

Abaixo seguem algumas vulnerabilidades que podem ser encontradas e quantificadas por ferramentas de análise estática de código para linguagem C e C++, linguagens que oferecem uma grande flexibilidade ao programador, favorecendo a introdução de vulnerabilidades.

- **UAV(*Uninitialized Argument Value*- Varíavel não inicializada)**: Esta métrica conta as variáveis não inicializadas no código. As linguagens C e C++ não são inicializadas com valores *default* quando são declaradas (recurso que é disponível em algumas linguagens) fazendo com que essas contenham lixo em seu conteúdo caso não sejam inicializadas. Dessa forma, a aplicação pode ter um comportamento inesperado quando utilizar essa variável não inicializada.
- **RSVA (*Return of stack variable address* - Retorno de endereço de uma variável de pilha)**: Esta vulnerabilidade acontece quando uma função retorna um endereço para uma variável que está alocada na pilha (stack). A pilha é o local

onde variáveis temporárias são armazenadas, como por exemplo variáveis declaradas dentro de funções. Se uma função declara uma variável dentro de seu escopo e usa esta mesma para seu retorno, temos o retorno de uma variável alocada na pilha. Ao termino da execução da função, a área de memória utilizada por ela fica disponível. Dessa forma, a próxima função chamada pode utilizar esse espaço de memória, sobrescrevendo o conteúdo que anteriormente foi retornado pela função anterior. Logo, o ponteiro retornado pela primeira função pode ter o valor alterado, podendo gerar comportamento inesperado no sistema ou até a quebra da aplicação. Este tipo de vulnerabilidade é difícil de se identificar, sendo aconselhável o uso de ferramentas de análise estática de código.

- **PITFC (*Potential insecure temporary file in call "mktemp"* - Arquivo temporário potencialmente inseguro pelo uso da chamada "mktemp")**: Esta vulnerabilidade ocorre quando um arquivo temporário inseguro é criado e usado pela aplicação, tornando a aplicação e o sistema de dados vulneráveis a ataques. Um arquivo criado pela aplicação é considerado inseguro quando ele é criado por mecanismos (funções específicas de APIs) que não geram arquivos com nomes únicos ou com nomes de randomização fraca. A função "mktemp"é um exemplo de mecanismo de geração de arquivos temporários que gera um nome único para um arquivo com base em um prefixo definido no código fonte. Porém, essa randomização gerada pelo mktemp é fraca, de maneira que outra aplicação maliciosa pode usar o mktemp passando o mesmo prefixo e conseguir gerar um arquivo com o mesmo nome que pode conter código malicioso, ou mesmo utilizar deste arquivo para obter as informações que seriam salvas pela aplicação original.

- **FGBO (*Potential buffer overflow in call to "gets"* - Possível *Buffer Overflow* ao chamar a função "gets")**: Esta vulnerabilidade está relacionada ao uso da função "gets"da linguagem C que copia toda informação passada pela entrada do programa para um *buffer* sem checar se o tamanho da entrada é equivalente ao tamanho do *buffer*. Dessa forma, caso a entrada seja maior que o *buffer*, haverá a sobrescrita da memória adjacente. Isso pode resultar em comportamento errado do programa, incluindo erros de acesso à memória, resultados incorretos, parada total do sistema, ou uma brecha num sistema de segurança.

- **ASOM (*Allocator sizeof operand mismatch* - Operador de alocação de *sizeof* não correspondente)**: Esta vulnerabilidade consiste em passar o operador inadequado para o tamanho de uma alocação de memória. Por exemplo, a vulnerabilidade ocorre quando temos um ponteiro para *int* e no momento de alocarmos a memória passarmos no *sizeof* um tipo *char*. Esse tipo de situação pode gerar um *buffer overflow* no momento da atribuição da variável.

- **DUPV (*Dereference of undefined pointer value* - Acessar o valor de um ponteiro não definido)**: Esta vulnerabilidade ocorre quando é feito o acesso ao valor de um ponteiro cujo estado é indefinido. Isso ocorre, por exemplo, quando este ponteiro aponta para um ponteiro que não foi inicializado. Esta vulnerabilidade está relacionada a CWE 457 (Use Unitialized of variable) pois o ponteiro está indefinido, uma vez que não foi inicializado. Portanto, as consequências podem ser desde a leitura de lixo de memória até mesmo a falha da aplicação.

- **DBZ (*Divisions by zero* - Divisão por zero)**: Esta vulnerabilidade acontece quando existe uma divisão de um valor por zero. Quando temos essa situação, o programa para de funcionar. Essa vulnerabilidade geralmente acontece quando um valor inesperado é passado para divisor do cálculo ou ocorre algum erro que gere este valor. O melhor jeito de prevenir é a realizar uma verificação que cheque se o divisor não é zero, e caso seja, deve-se implementar um tratamento, como por exemplo, o lançamento de exceções.

- **MLK (*Memory leak* - Estouro de memória)**: O software não gerencia o uso de memória, provocando o consumo excessivo desta, podendo haver a falta de memória para a aplicação. Sem memória, a aplicação consegue funcionar corretamente, podendo gerar resultados inesperados como também a falha da aplicação.

- **OBAA (*Out-of-bound array access* - Acesso de posição de um *array* fora do *range*)**: Esta vulnerabilidade acontece quando a aplicação tenta acessar um índice de *array* que está fora de seu *range*. O acesso de uma posição fora do *array* pode causar falha na execução (por exemplo, na linguagem C, falha de segmentação) como também a execução de código, pois a região de memória acessada pode conter código a ser executado ou até outras informações, impactando na confidencialidade de dados.

- **DF (*Double free* - Liberar memória duas vezes)**: Esta vulnerabilidade ocorre na linguagem C, quando o programa realiza a chamada free() duas vezes para o mesmo ponteiro. Chamar duas vezes o free() para o mesmo ponteiro pode corromper a estrutura de dados do programa que gerencia a memória. Esse erro ocorre normalmente em encadeamentos de estruturas condicionais má construídas.

- **AUV (*Assigned value is garbage or undefined* - Valor atribuído é lixo ou indefinido)**: Esta vulnerabilidade ocorre quando não temos certeza que o valor atribuído é válido. Por exemplo, uma variável recebe outra que não foi inicializada. Ou também quando uma variável recebe um valor composto por outras duas, como por exemplo uma soma, porém, uma das variáveis envolvidas na soma também não foi inicializado. Ou seja, essa métrica está relacionada a atribuições com valores inválidos ou indefinidos.

3.3 Proposta de Utilização de Métricas de Código-fonte

Métricas estáticas de código podem ser utilizadas para compreender e analisar características do código. Apresentamos as principais métricas relacionadas ao *design* de software e também algumas métricas de identificação de vulnerabilidades de código.

Acreditamos que o monitoramento do código através de métricas deva ser utilizado para melhoria contínua do desenvolvimento, independentemente da metodologia utilizada. Além disso, acreditamos que o Engenheiro de Software deveria utilizar métricas como uma prática constante de desenvolvimento tal qual a criação de testes automatizados e a realização de *refactorings*. A Figura 1 apresenta uma adaptação da Figura 21, onde propomos a inserção de dois novos componentes para o conjunto de práticas de *design* ágil.

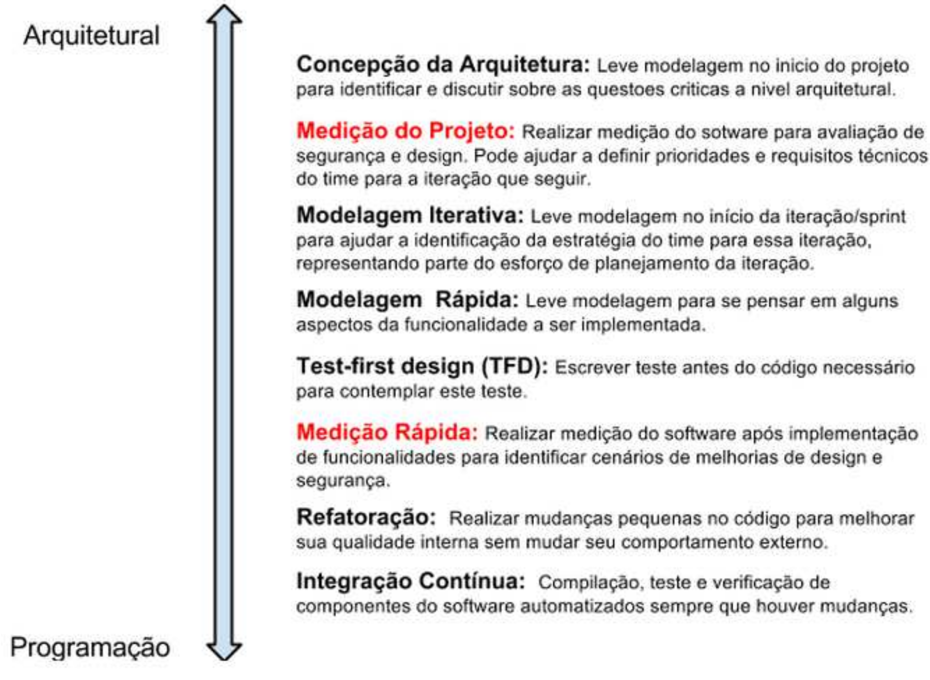

Figura 1 – Práticas do Design Ágil Utilizando Métricas de Código-Fonte

A prática Medição do Projeto, apresentada na Figura 1, propõe o monitoramento detalhado do código a cada iteração de desenvolvimento. Através desse monitoramento a equipe de desenvolvimento e gestores podem definir e priorizar um conjunto de requisitos técnicos para a próxima iteração que visam atacar os mau cheiros de código identificados, problemas de segurança ou violações do *design*.

Na Figura 1 ainda propomos a prática Medição Rápida como uma prática a ser utilizada constantemente pelo Engenheiro de Software ao evoluir e estender o código. As modificações realizadas durante o desenvolvimento podem gerar inconformidades técnicas indesejadas para o bem estar do código, que por sua vez podem ser identificadas através de métricas. Assim, o Engenheiro de Software teria um melhor suporte para escolher quais

são os seus próximos passos, como por exemplo, analisando qual seria a melhor refatoração a ser aplicada. Portanto, a prática de Medição Rápida poderia ser utilizada várias vezes ao longo do desenvolvimento, proporcionando a melhoria contínua do código.

Para que as práticas destacadas na Figura 1 possam de fato ser utilizadas na melhoria contínua do código, é necessário que os parâmetros de qualidade de um projeto sejam bem definidos e conhecidos pelos membros da equipe técnica. Mais do que isso, é importante que as medições sejam utilizadas para caracterizar o estado do software adequadamente, não baseado somente em interpretações de métricas isoladas, de tal forma que possibilite a tomada de decisão segura.

Em busca de mecanismos que facilitem a adoção da medição como prática constante no desenvolvimento de software, no Capítulo 4 é proposto a técnica Cenários de Decisões para utilização de métricas de código-fonte que visa abstrair as métricas para cenários que caracterizem o estado de um código, suportando a tomada de decisões. Ainda neste Capítulo fazemos algumas propostas de cenários que utilizam tanto métricas de *design* quanto de segurança para caracterizar estados indesejáveis para a segurança do código.

Por fim, o estudo e levantamento de métricas apresentadas no presente Capítulo serão a base para a proposta de exemplos de Cenários de Decisões do Capítulo 4. Além disso, buscamos compreender e relacionar as métricas de *design* e vulnerabilidades apresentadas para melhorar o monitoramento e evolução de softwares, dada as motivações apresentadas no Capítulo 2.

4 Cenários de Decisões

Neste Capítulo será apresentado o conceito de Cenários de Decisões, que visa contemplar um dos objetivos principais da presente monografia: Definição de cenários a partir de estudos teóricos para melhorar a interpretação e tomada de decisão sobre métricas estáticas de código-fonte. No Capítulo 2 foi apresentada a importância do *design* de software. Além disso, discutimos o papel do Engenheiro de Software no desenvolvimento de códigos com *design* robusto, limpo e seguro, explorando os principais conceitos, problemas, princípios e práticas que este profissional deve conhecer para alcançar esse objetivo.

Na Seção 2.2 ainda foram introduzidos temas relacionados ao monitoramento de código-fonte, métricas de *design* de software e métricas de vulnerabilidades. Aferimos que apesar de vários estudos relacionados à utilização de métricas de código-fonte ainda existem muitas dificuldades da adoção prática de métricas de código-fonte em projetos reais de Software. Deve-se enfatizar que métricas não resolvem problemas e sim as pessoas (WESTFALL, 2005). Métricas de software atuam como indicadores para prover informação, entendimento, avaliação, controle e predição para que as pessoas possam fazer escolhas e ações. Nesse sentido, no presente Capítulo buscamos reduzir a distância entre a medição do código-fonte e tomada de decisões por Engenheiros de Software através da proposta de Cenários de Decisões.

A ideia de definição de cenários para tomada de decisões é advinda do estudo realizado por Almeira & Miranda (2010) sobre mapeamento de métricas de código-fonte com os conceitos de Código Limpo. Neste Capítulo, apresentamos os primeiros passos para a extensão desse estudo e propomos o conceito de Cenários de Decisões como uma técnica que pode ser utilizada na abstração de métricas para facilitar o monitoramento de código-fonte. Cenários de Decisões nomeiam e mapeiam estados observáveis através de métricas de código-fonte que indicam a existência de determinada característica dentro do software, classe ou método. Um Cenário de Decisão é composto por:

- **Nome**: Identificação única do cenário. Deve ser significativo para prover a compreensão do estado que o cenário representa.
- **Métricas Envolvidas**: Identifica as métricas necessárias para a caracterização do cenário, sem definir relações entre essas métricas.
- **Nível**: Define o nível de abstração de software que pode ser caracterizada pelo cenário, por exemplo: Projeto; Estrutura de Herança; Classe; Método;
- **Descrição**: Discute os problemas, princípios envolvidos e a caracterização

- **Caracterização com Métricas**: Define matematicamente como caracterizar o cenário com as métricas envolvidas. Pode definir a composição destas métricas ou a interpretação conjunta necessária.

- **Ações sugeridas**: Propõe um conjunto de ações específicas tais como uma refatoração, a utilização de um padrão de projeto, prática e aplicação de princípios.

Cenários de Decisões ainda podem ser classificados de três formas diferentes baseados nas diferentes formas de utilização de métricas para a caracterização do mesmo:

- **Monométrico**[1]: Classificação de cenário que pode ser caracterizado por apenas uma métrica, não havendo a necessidade de observação de outras variáveis para que o cenário seja identificado.

- **Polimétrico**[2]: Classificação de cenário que precisa da interpretação de duas ou mais métricas para ser caracterizado. Cenários polimétricos são menos acoplados a escolha de métricas uma vez que sua identificação depende de mais de uma métrica, podendo ser uma caracterização mais completa de um cenário monométrico.

- **Composto**[3]: Classificação de cenário que é caracterizado a partir de uma métrica composta por outras métricas a partir de fórmulas matemáticas.

O objetivo da definição de Cenários é minimizar as principais dificuldades existentes na medição do código-fonte:

- **Escolha de Métricas**: Cada cenário é composto por um conjunto de métricas que devem ser utilizadas para aferir a ocorrência do mesmo. Assim, caso se queira observar se um software possui um determinado cenário de vulnerabilidade específica, por exemplo, o Engenheiro de Software ou Gerente devem se preocupar apenas sobre a escolha correta do cenário, abstraindo a escolha de métricas específicas.

- **Interpretação de Valores**: A ocorrência de um cenário em algum trecho específico do código deve ser, por si só, o suficiente para o entendimento e avaliação do estado de *design* deste trecho, não havendo a necessidade de ter que se interpretar os valores obtidos. A existência de um cenário ruim específico em um método deve prover o entendimento necessário para que o Engenheiro de Software realize ações para remoção deste cenário.

[1] Monométrico: de uma só medida
[2] Polimétrico: que apresenta ou emprega uma variedade de medidas
[3] Composto: formado de diversas partes

- **Redundância de Métricas**: Existem muitas métricas na Engenharia de Software que podem ser utilizadas para medir a mesma característica do software tais como tamanho, complexidade e coesão. Compreender cada uma delas e suas intersecções é uma tarefa dispendiosa, dificultando a escolha adequada de métricas que avaliem bem os elementos do software sem redundância de informação. A redundância de métricas não é algo desejado em projetos de software uma vez que a medição é um processo complexo e caro. Cenários idealmente devem ser estabelecidos a partir de estudos e experimentos. Assim, métricas consideradas redundantes podem ser eliminadas a partir da definição de cenários ou podem ser necessárias para a identificação de cenários diferentes para os quais essas métricas agregam alguma informação.

- **Interpretações Isoladas**: Podem existir métricas que possam ser utilizadas para a identificação de um cenário específico sem a necessidade de uso de outras métricas. Entretanto, na maioria dos casos uma métrica não provê informação suficiente a ponto de poder ser interpretada isoladamente. Assim, Cenários de Decisões diminuem o risco de interpretações isoladas inadequadas, pois reúnem um conjunto de métricas necessários para sua caracterização. Mesmo quando um cenário é definido por uma métrica específica, ele oferece um nível de abstração e interpretação que diminui as possibilidades de erros de interpretação.

- **Parâmetros de Comparação**: Os cenários definem uma interpretação a partir de um conjunto de métricas cujos valores podem ser especificados a partir do contexto envolvido, como por exemplo, variar de acordo com a linguagem de programação. Assim, um cenário deve ser adaptável para diferentes contextos devido a importância de se flexibilizar as interpretações de métricas, tema discutido e defendido por Meirelles (2013). Sugere-se que a escolha dos parâmetros adequados para caracterização do cenário deve ser feita por especialistas que compreendam as necessidades de seus projetos e as limitações e recursos da linguagem e paradigma de programação utilizados no software. Portanto, a escolha dos valores para caracterização de um cenário é a instanciação deste cenário para um contexto específico. Nesta monografia, além da proposta de cenários, também iremos propor instâncias destes cenários para projetos em C++.

Com os Cenários de Decisões introduzimos um novo conceito a ser utilizado na medição de software. Espera-se que o esforço destinado a medição de software em um projeto seja concentrado sobre a instanciação desses cenários, diminuindo-se o esforço necessário para coleta, interpretação e visualização de dados. Entretanto, os benefícios dos Cenários de Decisões são passíveis de experimentação, experimentos esses que estão fora do escopo deste trabalho.

4.1 Definição de Cenários de Decisão para Segurança de Software

A revisão teórica feita nesta monografia encontrada no Capítulo 2 a respeito de características de bom *design* e vulnerabilidades de software nos permite afirmar que a qualidade do código está diretamente relacionada a vulnerabilidades, visto que um código que possui alta complexidade, baixa modularização, alto acoplamento, entre outras características apresentadas na Seção 2.1.1 são mais fáceis de inserir vulnerabilidades e dificultam a descoberta de vulnerabilidades já existentes. Na Seção (2.2.2) foi visto também que muitos princípios de segurança de software estão relacionados ao *design* do código. Visto isso, a aplicação de boas práticas de *design* de código se torna essencial para o desenvolvimento de softwares seguros.

Porém, vulnerabilidades de software não ocorrem somente em códigos com mal *design*. Na Seção 2.2.1 foi visto que existe uma gama de vulnerabilidades específicas catalogadas pela comunidade, e muitas destas vulnerabilidades foram descobertas e reportadas por grandes empresas de softwares renomados no mercado, que são bem maduras em relação a qualidade de seus produtos. Foi visto também que muitas das vulnerabilidades de software que são identificadas por ferramentas de análise estática Seção 3.3 são vulnerabilidades específicas de uso de códigos, funções ou práticas consideradas perigosas para segurança da aplicação. Tais vulnerabilidades podem ser encontradas no código-fonte independente da aplicação de boas práticas de *design*, pois são difíceis de identificar e necessitam do conhecimento mais aprofundado do Engenheiro de Software para que este saiba se está inserindo ou não uma vulnerabilidade no código.

Dessa forma, para o contexto de segurança do software, podemos definir dois tipos de cenários de decisão:

- **Cenários de Decisão para Caracterização da Qualidade de Código**: Estes cenários buscam identificar características de software relacionadas a qualidade e *design* de código que podem influenciar em sua segurança, como complexidade, acoplamento, entre outros.
- **Cenários de Decisão para Caracterização de Vulnerabilidades Específicas de Código**: Estes cenários buscam identificar vulnerabilidades de software que podem ser encontradas em código-fonte independente que este esteja em um bom nível de qualidade e *design*. São cenários relacionados mais a erros que podem ser cometidos pelos desenvolvedores no momento da implementação;

O estudo sobre vulnerabilidades de software nos mostrou que grande parte das vulnerabilidades são encontradas na linguagem de programação C/C++. Além disso, existem uma quantidade expressiva de projetos livres desenvolvidos em C++, que incluem grandes projetos renomados como Chrome, Firefox, MySQL e OpenOffice. Vale ressaltar

também que ambas as ferramentas que utilizaremos para demonstrar o uso dos cenários já suportam a coleta da maior parte das métricas introduzidas no Capítulo 2 para C++.

Dessa forma, decidimos por usar a linguagem C++ como base para instanciação dos cenários propostos a seguir. Como se trata de uma linguagem orientada a objetos, permitirá aplicarmos tanto os conceitos estudados a respeito de *design* de código quanto os conceitos estudados a respeito de vulnerabilidades específicas.

4.1.1 Cenários de Decisão para Caracterização da Qualidade do Código

A seguir serão apresentados instâncias de cenários de decisões que visam identificar características de qualidade de código que impactam na segurança. Os cenários a seguir podem ser aplicados a programas que utilizam o paradigma OO, porém, os valores utilizados para as métricas que compõe os cenários foram definidos para projetos C++. Tais valores foram retirados do estudo de Meirelles (2013). Nesta Seção propomos quatro Cenários de Decisões.

4.1.1.1 Alta Superfície de Ataque a Atributos Internos

Este cenário busca identificar a violação do princípio de *design* de segurança Redução da Superfície de Ataque (Seção 2.2.2) em relação aos atributos de um objeto. Este princípio se baseia fundamentalmente na redução da exposição das estruturas do sistema em relação a interações externas. Em termos de *design*, diminuição do acesso as informações internas da classe podem ser obtidos a partir de um maior grau de encapsulamento das estruturas que compõem esta classe. Em termos de métricas de código-fonte, este cenário busca medir o tamanho da superfície de ataque aos atributos de uma classe. Estão inclusos nesse cenário classes ou módulos que tenham um alto valor de quantidade de atributos públicos.

Quadro resumo:

- **Nome**: Alta Superfície de Ataque a Atributos Internos
- **Classificação**: Composta;
- **Métricas Envolvidas**: NPA (Número de Atributos Públicos), NOA (Número de Atributos);
- **Nível**: Classe;
- **Descrição**: Esse cenário busca identificar classes que possuem muita exposição devido ao alto número de atributos públicos, violando o princípio de *design* seguro Redução de Superfície de Ataque.

- **Caracterização com Métricas**: NPA/NOA > value
  - Caracteriza o tamanho da superfície de ataque a atributos em percentagem. O cenário existe quando o valor desta métrica composta está acima do valor (referência) estipulado. Para projetos em C++, value pode ser 0.2
- **Ações sugeridas**:
  - **Refatorações aplicáveis:** - *Encapsulate Field*[4].
  - **Princípios aplicáveis à classe** - Princípio e Redução de Superfície de Ataque; Princípio de Encapsulamento;

4.1.1.2 Alta Superfície de Ataque Operacional

Este cenário busca identificar a violação do princípio de *design* de segurança Redução da Superfície de Ataque (Seção 2.2.2) em relação aos métodos de uma classe. Este princípio se baseia fundamentalmente na redução da exposição das estruturas do sistema em relação as manipulações e operações realizadas pelos métodos da classe. Em termos de *design*, quanto maior a quantidade de métodos públicos, maior a quantidade de interações possíveis com um objeto o que dificulta a aplicação do princípio *Mediate Completely* (Seção 2.2.2) e aumenta a vulnerabilidade das operações. Além disso, a dependência de parâmetros externos para realização de operações internas pode expor maiores detalhes dos mecanismos e algoritmos das classes de um projeto, sendo resultado do baixo grau de encapsulamento das operações, além de aumentar riscos de ataques e dificuldades de correção de problemas de segurança.

Em termos de métricas de código-fonte, este cenário busca medir o tamanho da superfície de ataque através de operações acessíveis de uma classe e os parâmetros necessários para sua realização. Estão inclusos nesse cenário classes ou módulos que tenham um alto valor de quantidade de métodos públicos e que tenham grande quantidade de parâmetros.

Quadro resumo:

- **Nome**: Alta Superfície de Ataque Operacional
- **Classificação**: Polimétrico;
- **Métricas Envolvidas**: NPM (Número de Métodos Públicos), ANPM (Média de Parâmetros por Método);
- **Nível**: Classe;

[4] <http://refactoring.com/catalog/encapsulateField.html>

- **Descrição**: Esse cenário busca identificar classes que devem receber maior atenção em relação à aplicação do princípio *Mediate Completely* devido à grande exposição da classe em termos operacionais. Este cenário indica que a aplicação do princípio de *design* seguro Redução de Superfície de Ataque pode melhorar o *design* da classe uma vez que as definições e abstrações de seus métodos são reestruturadas.

- **Caracterização com Métricas**: NPM > npm_value || ANPM > anpm_value

  - **NPM** - Caracteriza a quantidade de interações possíveis para as quais o princípio *Mediate Completely* deve ser aplicado. Para projetos em C++, o valor de npm_value pode ser 10;

  - **ANPM** - Caracteriza a quantidade de informações que poderiam ser verificadas para evitar ataques externos. Para projetos em C++, o valor de anpm_value pode ser 3;

- **Ações sugeridas**:

  - **Refatorações aplicáveis aos métodos da classe** - *Hide Method*[5]; *Remove Parameter*[6];

  - **Princípios aplicáveis à classe** - Princípio e Redução de Superfície de Ataque; Princípio de Encapsulamento; Princípios de Distribuição de Responsabilidades GRASP (LARMAN, 2007);

  - **Padrões aplicáveis no projeto para reduzir a ocorrência deste cenário** - Padrão *Facade*[7];

4.1.1.3 Ponto Crítico de Falha

Este cenário busca identificar a ocorrência de classes ou módulos que concentram muitas responsabilidades das quais muitas outras classes dependem. Este cenário é problemático uma vez que a classe das quais muitas outras classes dependem, em caso de falhas e exploração de vulnerabilidades tendem a afetar muitas outras estruturas do projeto. Além disso, a alta concentração de dependências em uma classe pode ser devido a inadequada distribuição de responsabilidades entre os módulos que compõem o projeto.

Em termos de métricas de código-fonte, este cenário é caracterizado a partir da medição da quantidade de classes que dependem da classe em análise. Esta dependência deve ser considerada em termos de acesso à métodos, acesso à atributos e herança.

Quadro resumo:

---

[5] <http://refactoring.com/catalog/hideMethod.html>
[6] <http://refactoring.com/catalog/removeParameter.html>
[7] <http://sourcemaking.com/design_patterns/facade>

- **Nome**: Ponto Crítico de Falha
- **Classificação**: Composto
- **Métricas Envolvidas**: ACC (Conexões Aferentes por Classe), NOC (Número de Filhos)
- **Nível**: Classe;
- **Descrição**: Esse cenário busca identificar classes que potencialmente são pontos críticos do projeto de software. Classes que são caracterizadas com este cenário devem ser cuidadosamente repensadas em termos de responsabilidades para não ser permanecerem sendo potenciais pontos críticos do projetos, uma vez que falhas e exploração de vulnerabilidades podem comprometer toda a estrutura acoplada a ela.
- **Caracterização com Métricas**: ACC + NOC > value
  - As duas métricas são utilizadas para Caracterizar a quantidade de classes acopladas à classe em análise, tanto em termos de acesso à métodos e atributos quanto em termos de herança. Este cenário existe quando o valor calculado através desta composição é maior do que estipulado. Para projetos em C++, o value pode ser 5.
- **Ações sugeridas**:
  - **Refatorações aplicáveis aos métodos da classe** - *Extract Class*[8], *Move Method*[9], *Push Down Method*[10];
  - **Princípios aplicáveis à classe** - Princípios de bom *design*: Modularização, Baixo Acoplamento; Princípio de Encapsulamento; Princípios de Distribuição de Responsabilidades GRASP (LARMAN, 2007);

4.1.1.4 Risco Elevado de Segurança

Este cenário busca identificar métodos que, devido sua estrutura interna, aumentam os riscos relacionados à segurança, principalmente por dificultarem a realização de testes, a busca de *bugs* existentes e a compreensão do software. Em outras palavras, quanto maior a complexidade de um método, mais complexas são suas interações, verificações e mecanismos de segurança. Além disso, o aumento da complexidade tem como consequência a dificuldade de se entender o módulo em análise. Assim, quanto mais complexo um método ou classe, menos pessoas vão compreendê-los completamente, reduzindo as chances dos envolvidos no projeto encontrarem inconformidades e vulnerabilidades.

[8] <http://refactoring.com/catalog/extractClass.html>
[9] <http://refactoring.com/catalog/moveMethod.html>
[10] <http://refactoring.com/catalog/pushDownMethod.html>

O problema com a complexidade de módulos do software está diretamente relacionado ao princípio de *design* seguro *Economy of mechanism* e ao princípio Simplicidade de bom *design*. Saltzer & Schroeder (1975a) afirmam que esses dois princípios são muito importantes pois, erros ou falhas que resultam em vulnerabilidades e acessos não desejados normalmente não são notados durante o uso normal. Portanto, técnicas como inspeção de código e outras são necessárias para assegurar os mecanismos de proteção necessários. Assim, para que tais técnicas possam ser utilizadas, um *design* simples e pequeno é essencial.

Em termos de métricas de código-fonte, este cenário busca medir a complexidade dos métodos das classes do projeto de software. Idealmente, este cenário poderia ser caracterizado à nível de método, porém, optou-se por manter a nível de classe. Estão inclusos nesse cenário classes com métodos com alta complexidade.

Quadro resumo:

- **Nome**: Risco Elevado de Segurança

- **Classificação**: Polimétrico;

- **Métricas Envolvidas**: ACCM (Média de Complexidade Ciclomática por Método), MLOC (Média de Número de Linhas de Código por Método);

- **Nível**: Classe;

- **Descrição**: Esse cenário busca identificar métodos que aumentam o risco de problemas de segurança e *bugs* no código-fonte devido sua complexidade e difícil manutenção.

- **Caracterização com Métricas**: ACCM > accm_value || MLOC > mloc_value

  - **ACCM** - Caracteriza a complexidade lógica do método, além de medir o mínimo de esforço de testes necessários para o método e o nível de dificuldade para compreensão completa do método. Para projetos em C++, accm_value pode ser 4;

  - **MLOC** - Caracteriza a quantidade de linhas de código dentro de um método, sendo que quanto maior a quantidade de linhas em um método, mais difícil é sua compreensão e inspeção. Para projetos em C++, mloc_value pode ser 10;

- **Ações sugeridas**:

- **Refatorações aplicáveis:** - *Extract Method*[11], *Extract Surrounding Method*[12], *Replace Conditional with Polymorphism*[13], *Replace Nested Conditional with Guard Clauses*[14], *Substitute Algorithm*[15].
- **Princípios aplicáveis ao método** - Princípio de Economia de Mecanismos; Princípio da Simplicidade ou KISS;

4.1.2 Cenários de Decisão para Caracterização de Vulnerabilidades Específicas de Código

A seguir serão apresentados instâncias de cenários de decisão que buscam identificar vulnerabilidades específicas de código. Observamos que instâncias desse tipo de cenário são particulares para cada linguagem de programação ou contexto, pois cada tecnologia envolvida na criação de software possui suas particularidades e com elas, vulnerabilidades específicas. Nesse sentido, como dito no início desse Capítulo, iremos abordar apenas vulnerabilidades presentes na linguagem C++, tornando os cenários a seguir aplicáveis apenas nesse contexto.

Com o estudo sobre as *CWE's*, percebemos que vulnerabilidades específicas já descrevem um cenário específico pois cada vulnerabilidade caracteriza um problema e um estado do código fonte. Porém, vimos com o estudo das taxonomias que algumas vulnerabilidades estão relacionadas à um problema mais geral. Dessa forma, os cenários propostos a seguir buscam agrupar algumas vulnerabilidades para descrever uma estado de código de maneira mais alto nível.

Em relação aos valores de métricas destes cenários, como cada métrica conta quantas ocorrências daquela vulnerabilidade ocorre no código, basta então apenas uma ocorrência para que o código se torne vulnerável. Porém, sabemos que até mesmo grandes software estão em produção e possuem suas vulnerabilidades, deixando a ideia de que possa existir uma quantidade "aceitável"de determinada vulnerabilidade. Mas isso cabe a um estudo a parte definir esse valor. Nesta monografia, iremos considerar que vulnerabilidades não podem existir, e devem ser combatidas a todo momento.

Feitas essas observações, definimos 4 cenários nessa Seção.

4.1.2.1 Uso de variáveis não inicializadas

Este cenário busca identificar o uso de variáveis não inicializadas. Este cenário também é definido pela CWE-457 [16]. A declaração de variáveis em algumas linguagens,

[11] <http://refactoring.com/catalog/extractMethod.html>
[12] <http://refactoring.com/catalog/extractSurroundingMethod.html>
[13] <http://refactoring.com/catalog/eplaceConditionalWithPolymorphism.html>
[14] <http://refactoring.com/catalog/replaceNestedConditionalWithGuardClauses.html.html>
[15] <http://refactoring.com/catalog/substituteAlgorithm.html>
[16] http://cwe.mitre.org/data/definitions/457.html

como C, Perl e PHP, não as pré-inicializam com valores padrão. Com isso, variáveis que são criadas na pilha de memória podem conter em seus valores lixo ou até comandos, bem como valores de outras funções e variáveis que também utilizaram da mesma área de memória em outro momento de execução da aplicação. Essa situação é perigosa, pois tanto a aplicação pode apresentar resultados inesperados como também um atacante pode visualizar informações contidas nessa variável que ele não estaria autorizado a ver.

Muitas vezes podem ocorrer de existir uma estrutura de controle de fluxo de execução de código (como *if else* e *switch case*) que definem a situação em que a variável irá ser inicializada e, ao sair dessa estrutura de controle de fluxo, a variável é finalmente utilizada. Nessa situação, pode ocorrer da aplicação não passar pelo fluxo que inicializa a variável. Dessa forma, ao sair do fluxo, o programa irá usar a variável que não foi inicializada.

Em termos de métricas de código-fonte, este cenário busca medir ocorrências do uso de variáveis não inicializadas. A métrica UAV (*Uninitialized Argument Value*) busca medir argumentos passado como parâmetros de chamada de funções que não foram inicializados. A métrica ROGU (*Result of operation is garbage or undefined*) busca verificar situações onde usamos variáveis não inicializadas em operações lógicas. A métrica AUV (*Assinged Value is garbage or undefined*) busca verificar situações que uma variável recebe o valor de outra variável que pode não ter sido inicializada. Essas três métricas podem ser utilizadas em conjunto, e a ocorrência de qualquer uma delas podem indicar este cenário no código fonte.

Quadro resumo:

- **Nome**: Uso de Variáveis não Inicializadas
- **Classificação**: Composta;
- **Métricas Envolvidas**: UAV (*Uninitialized Argument Value*), ROGU (*Result of operation is garbage or undefined*), AUV (*Assinged Value is garbage or undefined*);
- **Nível**: Arquivo;
- **Descrição**: Este cenário busca identificar o uso de variáveis não inicializadas, pois isso pode causar desde falhas na aplicação quanto acesso a informação não autorizado.
- **Caracterização com Métricas**: AUV + UAV + ROGU >= 1
  - Este cenário ocorre quando qualquer uma dessas métricas é maior ou igual a um, caracterizando a ocorrência de pelo menos uma vulnerabilidade relacionada ao conceito do cenário.
- **Ações sugeridas**:

- **Princípios aplicáveis** - *Fail Securery* criando verificações para que não haja erros nem quebra da aplicação por utilizar uma variável não inicializada .
- **Refatorações**: Criar mecanismos para garantir que a variável será inicializada antes de ser usada.

4.1.2.2 Alta possibilidade de Falha por mau uso de ponteiros

Este cenário busca identificar possíveis locais no código fonte em que é feito o mau uso de ponteiros, e que consequentemente podem gerar falhas na aplicação.

Nos fóruns de dúvidas relacionados a computação e programação, é muito comum identificar relatos de erros de várias pessoas relacionado a falhas de ponteiros que geram o famoso *"segmentation fault"*. Ponteiros, se não utilizados corretamente, podem gerar falhas e são muito difíceis de se identificar. Além disso, existem diferentes situações em que o desenvolvedor pode estar gerando uma falha por mau uso de ponteiros.

Em termos de métricas de código-fonte, este cenário é composto pelas métricas:

- **DNP (*Dereference of null pointer*)**
- **DF (*Double free*)**
- **AUF (*Use After free*)**
- **DUPV (*Dereference of undefined pointer value*)**
- **BD (*Bad deallocator*)**

Todas essas métricas podem identificar possíveis falhas na aplicação. Este cenário pode ser muito útil para solucionar falhas na aplicação cujo não se sabe a causa.

Quadro resumo:

- **Nome**: Alta possibilidade de Falha por mau uso de ponteiros
- **Classificação**: Composta;
- **Métricas Envolvidas**: DNP (*Dereference of null pointer*); DF (*Double free*); AUF (*Use After free*); DUPV (*Dereference of undefined pointer value*); BD (*Bad deallocator*);
- **Nível**: Arquivo;
- **Descrição**: Este cenário busca identificar o uso de ponteiros de maneira não segura, que podem gerar falha na aplicação.
- **Caracterização com Métricas**: NP + DF + AUF + DUPV + BD >=1

  - Este cenário ocorre quando há a ocorrência de pelo menos uma dessas vulnerabilidade.

- **Ações sugeridas**:

  - **Refatorações**: Para cada métrica, deve se tomar uma ação específica para resolvê-la. Então, caso ocorra este cenário, devemos observar qual das vulnerabilidades ocorreu trabalhar para reduzir o valor dessa métrica. Por exemplo, se a métrica encontrada for a DNP, sabemos que existe um ponteiro nulo que está sendo desreferenciado, então deve-se tomar providências, de acordo com cada lógica de programa para que isso não aconteça.

    * DNP

4.1.2.3 Buffer Overflow

Este cenário busca no código fonte situações que podem gerar a vulnerabilidade de *Buffer Overflow*. Como já mencionado na sessão 2.2, o *Buffer Overflow* é um caso comum de violação da segurança a memória, e ocorre normalmente quando escrevemos dados cujo tamanho ultrapasse o tamanho do *buffer* definido para ele. Essa situação pode gerar mau funcionamento do sistema, pois a informação que estourou o *buffer* pode corromper outras informações ou processos que tiveram sua região de memória invadida; como também pode ser usado para injetar códigos malicioso e alterar o fluxo de execução da aplicação.

Em termos de métricas de código fonte, este cenário é composto pelas métricas FGBO (*Potential buffer overflow in call to "gets"*) e pela métrica ASOM (*Allocator sizeof operand mismatch*). A métrica FGBO identifica a utilização da função *"gets"* na linguagem C/C++. Essa função não garante garante que o tamanho da variável de entrada será do tamanho da variável a ser atribuída, se tornando um risco a ser explorado pelos atacantes. A métrica ASOM identifica situações no código em que é passado o operador inadequando no momento que se aloca memória para uma variável. Isso acontece, por exemplo na linguagem C, quando declaramos um ponteiro do tipo inteiro e no momento do alocação de memória é passado um tamanho refente a uma variável do tipo *char*. Logo, o espaço alocado não corresponde ao tamanho do dado que é esperado receber, podendo causar o estouro de *buffer*.

Quadro resumo:

- **Nome**: *Buffer Overflow*

- **Classificação**: Composta;

- **Métricas Envolvidas**: FGBO (*Potential buffer overflow in call to ёgets¨*) e ASOM (*Allocator sizeof operand mismatch*)

- **Nível**: Arquivo;
- **Descrição**: Este cenário busca identificar situações no código fonte que podem gerar a vulnerabilidade de *Buffer Overflow* (Estouro de memória);
- **Caracterização com Métricas**: ASOM + FGBO >=1
  - Este cenários ocorre quando pelo menos uma dessas vulnerabilidades ocorrem no código.
- **Ações sugeridas**:
  - **Substituir chamadas para a função *"gets"*** - a função *"fgets()"* pode ser utilizada no lugar da função *"gets"*, pois aquela possui um parâmetro em sua chamada que indica o tamanho máximo do *buffer*.
  - **Fazer *match* dos operadores de alocação de memória** - Refatorar os trechos de código atribuindo o operador correto para a alocação de memória;

4.1.2.4 Confidencialidade Ameaçada

Este cenário busca identificar situações no código fonte onde informações podem ser expostas sem a devida permissão, impactando diretamente em um dos principais aspectos quando falamos de segurança de software: a confidencialidade dos dados.

Um código seguro não pode dar brecha para que o atacante ou qualquer outro usuário possa ver informações que não eram para serem vistas. Alguns erros na codificação, como por exemplo, acessar uma posição de uma *array* fora de seu intervalo, podem acabar mostrando informações de outras regiões de memória, podendo ser essa informação crítica ou não para a segurança do sistema.

Em termos de métricas de código fonte, as métricas OBAA (*Out-of-bound array access*) e DUPV (*Dereference of undefined pointer value*) buscam identificar situações de código fonte em que podem ser expostas informações não autorizadas pelo sistema. A métrica OBAA identifica justamente o acesso de indices que não pertencem ao *array*, podendo assim vazar a informação contida na região de memória adjacente a região do *array*. A métrica DUPV identifica situações onde um ponteiro indefinido é desreferenciado, ou seja, quando é lido o valor de um ponteiro que está indefinido. Dessa forma, ele pode estar apontando para uma posição de memória a qual não sabemos, e , consequentemente, vazando alguma informação não autorizada.

Quadro resumo:

- **Nome**: *Confidencialidade Ameaçada*
- **Classificação**: Composta;

- **Métricas Envolvidas**: OBAA (*Out-of-bound array access*) e DUPV (*Dereference of undefined pointer value*)

- **Nível**: Arquivo;

- **Descrição**: Este cenário busca identificar trechos de código fonte que podem fornecer informações não autorizadas, impactando na confidencialidade do sistema. O acesso a informação não autorizada se dá pela leitura de regiões de memória que não estão no contexto da variável em questão.

- **Caracterização com Métricas**: OBAA + DUPV >=1

  – Este cenário ocorre quando pelo menos uma dessas vulnerabilidades é encontrada.

- **Ações sugeridas**:

  – **Especificar e verificar o range o *array* antes de acessa-lo** - identificar o trecho de código que métrica OBAA aponta e criar mecanismos que garantem o limite do índice de acesso do array, dessa forma iremos garantir que somente a região de memória pertencente ao *array* é acessada e lida.

  – **Garantir que o ponteiro não seja indefinido antes de desreferencia-lo** - Identificar locais em que ponteiros indefinidos são desreferenciados e garantir que eles tenham um valor no momento da leitura.

4.1.2.5 Operações lógicas (somas, divisões) com integridade ameaçada

Este cenário busca identificar no código fonte ocorrência de operações lógicas que podem estar comprometidas por utilizarem variáveis indefinidas ou não inicializadas, comprometendo então a integridade da operação.

Dessa forma, o software pode ter um comportamento diferente do esperado, pois o resultado da operação é indefinido, podendo levar o fluxo da aplicação para outro rumo ou fornecendo resultados errados ao usuário.

Em termos de métricas, as métricas DBZ (*Division by zero*) e ROGU (*Result of operation is garbage or Undefined*) buscam identificar ocorrências desse cenário no código fonte.

As divisões por zero, na linguagem C/C++, podem causar falha na aplicação, afetando sua disponibilidade. Esta vulnerabilidade é até definida pela CWE 396 [17]. Sugere-se verificar se o divisor é zero antes de se realizar a operação.

[17] http://cwe.mitre.org/data/definitions/369.html

A métrica ROGU busca identificar operações em que o resultado pode ser lixo ou indefinido. Por exemplo, se utilizamos uma variável não inicializada em uma equação, o resultado fica indefinido, pois certamente essa variável não inicializada terá lixo de memória em seu conteúdo. Essa métrica busca identificar esse tipo de situação. Como solução, sugere-se então inicializar as variáveis que irão participar da operação, garantindo assim que haverá o mínimo de integridade para a operação.

Quadro resumo:

- **Nome**: *Operações lógicas (somas, divisões) com integridade ameaçada*
- **Classificação**: Composta;
- **Métricas Envolvidas**: DBZ (*Division by zero*) e ROGU (*Result of operation is garbage or Undefined*)
- **Nível**: Arquivo;
- **Descrição**: Este cenário busca identificar no código fonte ocorrência de operações lógicas que podem estar comprometidas por utilizarem variáveis indefinidas ou não inicializadas, comprometendo então a integridade da operação.
- **Caracterização com Métricas**: DBZ + ROGU >=1
  - Este cenário ocorre quando é encontrado pelo menos uma ocorrência de uma dessas vulnerabilidades.
- **Ações sugeridas**:
  - **Verificar divisor antes da operação** - Verificar se o divisor é zero antes de realizar a operação;
  - **Inicializar variáveis antes de operações** - Garantir que todas as variáveis estão pelo menos pré-inicializadas antes de participarem de uma operação

A seguir, foi montado uma tabela resumo com todos o cenários propostos até o momento, a fim de sintetizar suas principais características, métricas envolvidas e ações sugeridas.

Cenário	Tipo	Nível	Descrição	Fórmula	Ações Sugeridas
Alta Superfície de Ataque a Atributos Internos	Qualidade de código	Classe	Esse cenário busca identificar classes que possuem muita exposição devido ao alto número de atributos públicos, violando o princípio de design seguro Redução de Superfície de Ataque	NPA/NOA >0.2	Refatorações aplicáveis: Encapsulate Field; Princípios: Princípio e Redução de Superfície de Ataque; Princípio de Encapsulamento;
Alta Superfície de Ataque Operacional	Qualidade de código	Classe	Esse cenário busca identificar classes que devem receber maior atenção em relação à aplicação do princípio Mediate Completely devido à grande exposição da classe em termos operacionais.	NPM >10 || ANPM >3	Refatorações: Hide Method; Remove Parameter;Princípios: Princípio e Redução de Superfície de Ataque; Princípio de Encapsulamento; Princípios de Distribuição de Responsabilidades GRASP; Padrões: Facade;
Ponto Crítico de Falha	Qualidade de código	Classe	Esse cenário busca identificar classes que potencialmente são pontos críticos do projeto de software.	ACC + NOC >5	Refatorações: Extract Class; Move Method; Push Down Method;Princípios: Modularização, Baixo Acoplamento; Princípio de Encapsulamento; Princípios de Distribuição de Responsabilidades GRASP

Tabela 1 – Parte I - Resumo de todos os cenários

Cenário	Tipo	Nível	Descrição	Fórmula	Ações Sugeridas
Risco Elevado de Segurança	Qualidade de código	Classe	Esse cenário busca identificar métodos que aumentam o risco de problemas de segurança e bugs no código-fonte devido sua complexidade e difícil manutenção.	ACCM >4 || MLOC >10	Refatorações aplicáveis: Extract Method, Extract Surrounding Method, Replace Conditional with Polymorphism, Replace Nested Conditional with Guard Clauses , Substitute Algorithm; Princípios:Princípio de Economia de Mecanismos; Princípio da Simplicidade ou KISS;
Uso de variáveis não inicializadas	Vulnerabilidade específica	Arquivo	Este cenário busca identificar o uso de variáveis não inicializadas, pois isso pode causar desde falhas na aplicação quanto acesso a informação não autorizado.	AUV + UAV + ROGU >= 1	Refatorações: Criar mecanismos para garantir que a variável será inicializada antes de ser usada; Princípios: Fail Securery
Alta possibilidade de Falha por mau uso de ponteiros	Vulnerabilidade específica	Arquivo	Este cenário busca identificar o uso de ponteiros de maneira não segura, que podem gerar falha na aplicação.	NP + DF + AUF + DUPV + BD >=1	Refatorações: Extract Class; Move Method; Push Down Method;Princípios: Modularização, Baixo Acoplamento; Princípio de Encapsulamento; Princípios de Distribuição de Responsabilidades GRASP

Tabela 2 – Parte II - Resumo de todos os cenários

Cenário	Tipo	Nível	Descrição	Fórmula	Ações Sugeridas
Buffer Overflow	Vulnerabilidade específica	Arquivo	Este cenário busca identificar situações no código fonte que podem gerar a vulnerabilidade de Buffer Overflow (Estouro de memória);	ASOM + FGBO >=1	Refatorações: Substituir chamadas para a função "gets"; Fazer match dos operadores de alocação de memória;
Confidencialidade Ameaçada	Vulnerabilidade específica	Arquivo	Este cenário busca identificar trechos de código fonte que podem fornecer informações não autorizadas, impactando na confidencialidade do sistema.	OBAA + DUPV >=1	Refatorações: Especificar e verificar o range o array antes de acessá-lo; Garantir que o ponteiro não seja indefinido antes de desreferencia-lo
Operações lógicas (somas, divisões) com integridade ameaçada	Vulnerabilidade específica	Arquivo	Este cenário busca identificar no código fonte ocorrência de operações lógicas que podem estar comprometidas por utilizarem variáveis indefinidas ou não inicializadas, comprometendo então a integridade da operação.	DBZ + ROGU >=1	Para cada métrica, deve se tomar uma ação específica para resolvê-la. Então, caso ocorra este cenário, devemos observar qual das vulnerabilidades ocorreu trabalhar para reduzir o valor dessa métrica.

Tabela 3 – Parte III - Resumo de todos os cenários

5 Estudo de Caso

Neste Capítulo será apresentada a utilização dos Cenários de Decisões em projetos reais, com o principal objetivo de demonstrar a reprodução de cenários em ambientes de tomada de decisão. Assim, este Capítulo é destinado ao Estudo de Caso de utilização da ferramenta Mezuro e de um ambiente *DWing* para observação de Cenários de Decisões em projetos de software.

Para tanto, será explorada a avaliação de três projetos de software livre em C++. Além disso, iremos explicar os passos necessários para reproduzir a estrutura de cenários nos dois ambientes de tomadas de decisões abordados nesta monografia. Assim, serão apresentadas as principais evoluções e adaptação de cada ferramenta e os detalhes específicos da observação de cada cenário.

O método de execução desses estudos de casos é apresentado na Figura 2. Nesta Figura são descritos os passos necessários para reproduzir cenários em ferramentas que apoiam a tomada de decisões baseados em métricas de software que, no contexto desta monografia, consiste no Mezuro e *DWing*. Esta mesma metodologia pode ser usada para o desenvolvimento de estudos de casos semelhantes com outras ferramentas ou até mesmo reproduzir os estudos de casos que serão apresentados neste Capítulo.

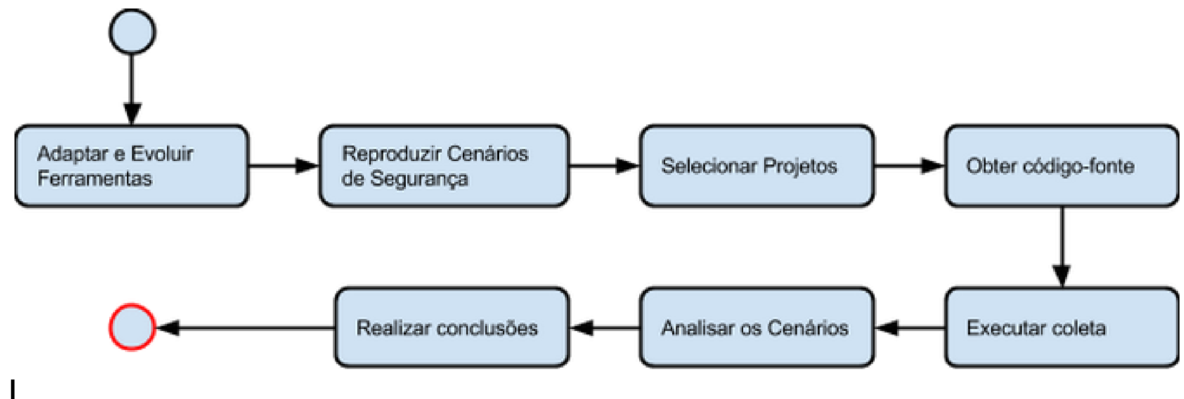

Figura 2 – Método para execução dos estudos de casos.

Os passos da Figura 2 são melhor descritos a seguir:

1. **Adaptar e Evoluir Ferramentas** - Este passo consiste em evoluir as estruturas, modelos, componentes e camada de apresentação das ferramentas que serão utilizadas com o objetivo de melhor suportar a utilização de Cenários de Decisão.

2. **Reproduzir Cenários** - Uma vez que as ferramentas utilizadas já suportam a reprodução de Cenários, deve-se utilizar dessa estrutura para definir cenários reais para avaliação de projetos de software. No contexto dessa monografia, esse passo consiste em reproduzir os Cenários de Segurança criados no Mezuro e no *DWing*.

3. **Selecionar Projetos** - Esta etapa consiste em definir quais projetos serão avaliados a partir dos cenários definidos e pode ser executada independente das ferramentas de tomada de decisão, variando de acordo com os objetivos de utilização dos Cenários.

4. **Obter Código-Fonte** - Uma vez selecionados os projetos, o próximo passo é obter o código-fonte referente aos projetos escolhidos. Algumas ferramentas. como o Mezuro, já possuem bom suporte para automatizar esse passo.

5. **Executar Coleta** - Esta etapa consiste na obtenção dos valores de métricas dos códigos-fontes dos projetos selecionados que deve ser automatizado. A saída desta etapa deverá ser processada pelas ferramentas trabalhadas para proporcionar a análise dos projetos. O esforço nessa etapa deve-se reduzir a coletar somente as métricas necessárias para composição dos Cenários de Decisão utilizados.

6. **Analisar Cenários** - A partir dos resultados obtidos, deve-se utilizar os Cenários de Decisão para observar as características do estado atual do projeto analisado. No contexto desse trabalho, esta etapa consiste em analisar quais os principais módulos oferecem riscos de segurança ao software. Porém, como o principal interesse é apresentar a utilização de cenários em ferramentas de tomada de decisões, esta etapa não será explorada detalhadamente.

7. **Realizar Conclusões** - Esta etapa final consiste em realizar ações a partir da compreensão dos Cenários observados, seja para fins de estudos ou para fins de desenvolvimento do software.

Estes passos metodológicos devem ser reproduzidos para cada ferramenta utilizada, sendo que uma vez que uma ferramenta já suporta adequadamente a observação de Cenários de Decisão, os passos 1 e 2 não precisam ser repetidos para análise de novos projetos.

5.1 Cenários de Decisões no Mezuro

O Mezuro é uma plataforma livre para monitoramento de código-fonte que busca auxiliar em vários problemas relacionados à utilização de métrica, visando ser uma interface que permita, de forma flexível, a extração e análise de métricas estáticas de código-fonte, licenciado como AGPLv3[1] (MANZO et al., 2014). O Mezuro é uma plataforma

[1] <http://www.gnu.org/licenses/agpl-3.0.html>

concebida através do amadurecimento de diversas ferramentas, inicializada através do projeto Qualipso[2]. Dentre estas ferramentas, destaca-se o Analizo[3], uma das ferramentas utilizadas pelo Mezuro para extração de métricas de código-fonte em C/C++ e Java.

A arquitetura do Mezuro atual é composta por vários serviços. Essa arquitetura está evoluindo para uma arquitetura de composição de quatro serviços principais que se resumem em:

- **Prezento**: Camada de apresentação da plataforma, desenvolvida em Ruby on Rails.
- **Kalibro Gatekeeper**: Serviço que faz a intermediação e orquestração das outras camadas com o Prezento.
- **Kalibro Processor**: Serviço que centraliza o processamento de métricas de projetos.
- **Kalibro Configuration**: Serviço responsável pelo processamento de configurações.

Uma explicação mais detalhada desta arquitetura e de cada serviço pode ser obtida no Apêndice C.

Dois destes serviços agregam conceitos fundamentais dentro da plataforma, que também serão muito importantes para a representação de cenários.

O Kalibro Configuration contempla o conceito de configuração que é um conjunto de métricas e parâmetros que podem ser utilizadas para a avaliação de um projeto. Uma configuração consiste, portanto, na composição de métricas cujos valores de referência são flexíveis e podem ser determinados separadamente para cada configuração criada. Associado ao conceito de configuração, está a criação de intervalos qualitativos associado a valores de métricas. Este módulo ainda é complementado por *Reading Gropus* que são grupos de leituras definidos por usuários que associam nomes qualitativos à cores que podem ser utilizados em configurações a partir na definição de intervalos de valores. Esta característica é muito importante para a utilização das métricas, uma vez que abstraem a interpretação direta dos valores obtidos para definições mais simples como bom, regular e ruim. Assim, tem-se a flexibilidade de ter parâmetros que variam de acordo com a linguagem, natureza do software, dentre outras coisas, apenas pela criação de diferentes configurações.

O Kalibro Processor contempla o módulo de extração e processamento de métricas de código-fonte. Para que esse processamento seja realizado, o Kalibro Processor realiza o download de projetos que estão em repositórios GIT ou SVN e utiliza uma ou mais ferramentas de extração de métricas, como mencionado anteriormente. Assim, obtem-se a

[2] Quality Platform for Open Source: <http://qualipso.icmc.usp.br/>
[3] <http://analizo.org/>

partir da análise estática do código-fonte um conjunto de métricas nativas que podem ser compostas para formular métricas mais complexas e de maior valor interpretativo. Dentro do Mezuro, esta abordagem é realizada através da criação de métricas compostas as quais serão muito importantes para a definição de Cenários de Decisões.

Atualmente, o Mezuro pode monitorar projetos em C, C++ e Java uma vez que utiliza o Analizo como seu principal extrator de métricas. Entretanto, com a evolução da plataforma, pretende-se acoplar novos extratores na ferramenta para outras linguagens de programação.

Descrevemos a seguir os passos necessários para a utilização de Cenários de Decisões no Mezuro referente a realização das etapas Adaptar e Evoluir Ferramentas e Reproduzir Cenários do estudo de caso. Todas as etapas exigem que haja um usuário autenticado no sistema.

5.1.1 Criação do *Reading Group* para os Cenários

A criação de um *Reading Group* para abstrair a interpretação do cenário é o primeiro passo para adaptação de cenários na ferramenta. Portanto, foi criado um grupo[4] que define duas opções para a leitura de resultados binários, a fim de mostrar se existe ou não um cenário, usando a cor verde para a resposta falsa e vermelho para a resposta verdadeira. Este grupo é criado independente de configurações ou projetos dentro do Mezuro, de tal forma que pode ser utilizado por qualquer configuração que venha a reproduzir cenários. O grupo criado pode ser visto na Figura 3.

Figura 3 – Reading Group criado para Cenários. Disponível no Mezuro.org <http://mezuro.org/reading_groups/7>

[4] <http://mezuro.org/reading_groups>

5.1.2 Criação de uma Configuração para uma Categoria de Cenários

Nesta etapa deve-se criar uma configuração onde serão definidos todos os cenários que queremos avaliar em conjunto. No contexto do Mezuro, os cenários serão definidos a partir do recurso de criação de Métricas Compostas que será explicado adiante.

Uma configuração, portanto, vai conter um conjunto de métricas e cenários caracterizados a partir dessas métricas. Como alternativas de criação de configurações como agregadores de cenários, teríamos a opção de criar uma única configuração que contemple todos os Cenários de Segurança que usam métricas de *design* (4.1.1) e os Cenários de Segurança que usam métricas de vulnerabilidade (4.1.2), da mesma forma que poderíamos criar uma configuração separada para cada tipo de cenário. O que mudaria é que, no primeiro caso o projeto seria monitorado uma única vez, enquanto no segundo o projeto deveria ser monitorado duas vezes, uma para cada configuração.

Sugere-se que a segunda abordagem seja utilizada, pois flexibiliza mais o monitoramento a partir de cenários, uma vez que existem diversos projetos com diversos interesses diferentes. Além disso, vale ressaltar que uma configuração deve contemplar apenas uma linguagem de programação e domínio, uma vez que os parâmetros de avaliação dos Cenários irão variar de acordo com o contexto.

A criação de uma configuração é bem simples, pois deve-se informar apenas o nome e a descrição da configuração[5]. Assim, criou-se a configuração intitulada "Cenários de Decisões - Design Seguro para C++"para a categoria de cenários descrita em (4.1.1) como apresentado na Figura 4.

Figura 4 – Configuração criada para Cenários de Design Seguro. Disponível no Mezuro.org <http://mezuro.org/mezuro_configurations/24>

Na visualização de uma configuração pode-se também ver a lista de métricas que compõem a configuração. Caso o usuário logado no Mezuro tenha a permissão necessária

[5] <http://mezuro.org/mezuro_configurations>

de edição de uma determinada configuração, na tela de apresentação ainda pode ser vista a opção de Adicionar Métrica (*Add Metric*), assim como apresentada na Figura 4.

A opção de Adicionar Métrica é fundamental para a segunda etapa da criação da configuração. Portanto, após criada uma configuração, deve-se adicionar todas as métricas básicas que são utilizadas nas caracterizações dos cenários que irão compor essa configuração. Para isso, basta ir em *Add Metric* na página da configuração, onde será aberta a lista de ferramentas coletoras utilizadas pelo Mezuro e as respectivas métricas suportadas por cada uma dessas ferramentas.

No contexto da configuração criada para esta monografia foram adicionadas todas as métricas envolvidas nos Cenários de Decisões listados em (4.1.1). Todas as métricas escolhidas utilizam o Analizo como ferramenta extratora. A própria definição da métrica escolhida a partir de uma ferramenta também já define o escopo da métrica e as linguagens de programação para as quais a métrica pode ser calculada, conforme demonstrado na Figura 5.

Figura 5 – Visualização da Métrica NOC no Mezuro. Disponível no Mezuro.org <http://mezuro.org/mezuro_configurations/24/metric_configurations/69>

Como pode ser visto na Figura 5, para cada métrica selecionada deve-se ainda definir o peso da métrica dentro da configuração (campo *Weight*) e a forma de agregação da métrica (campo *Aggregation Form*), que são informações importantes para a visualização da métrica em níveis de abstração maiores, principalmente a nível de pacotes e módulos. Para o contexto deste trabalho, mantivemos os valores 1.0 e Média para essas informações, tendo em vista que não são tão importantes neste caso, pois queremos observar nosso software principalmente a partir dos cenários, sendo os valores das métricas secundários, a serem utilizados somente por aqueles que desejam se aprofundar na caracterização dos cenários. Pelo mesmo motivo, não é necessário definir os intervalos para avaliação qualitativa dessas métricas básicas.

5.1.3 Criação de Cenários

Uma vez criada a configuração que irá conter os cenários e já adicionadas todas as métricas necessárias, o próximo passo é a criação dos Cenários de Decisões. Nesta etapa foi preciso estudos e evoluções sobre o Mezuro para determinar qual a melhor forma de representação dos cenários. Poderia-se adotar a estratégia de desenvolver uma estrutura de cenários novas, mas a partir da interação com a comunidade de desenvolvedores do Mezuro, optou-se por utilizar e evoluir o recurso de **métricas compostas**.

Dentro da ferramenta, métricas compostas são métricas que podem ser criadas a partir da composição das métricas bases das ferramentas extratoras adicionadas à uma configuração. Esta funcionalidade é muito interessante uma vez que flexibiliza a extensão da utilização do Mezuro para análise mais complexas que envolvem a composição matemática de outras métricas que não são diretamente extraídas dos projetos. Muitos estudos científicos apresentados ao longo desta monografia propõem novas métricas a partir da composição de métricas básicas conhecidas.

Para a definição de como a métrica composta deve ser calculada, o usuário deve escrever um *script* simples em Javascript que retorne o valor desejado. Como é definido um identificador para cada métrica básica adicionada na configuração (campo *Code* visível na Figura 5), nos *scripts* para métricas compostas pode-se acessar os valores das métricas extraídas a partir de uma chamada de método cujo nome é o mesmo do código da métrica. Para exemplo, dentro do *script* poderia-se acessar o valor da métrica *Number of Children* através da chamada *noc()*, onde *noc* é o código dessa métrica.

Além da utilização de *scripts*, assim como as métricas básicas, as métricas compostas ainda possuem informações de nome e descrição que são essenciais para a definição do quadro resumo do cenário. A utilização de heranças e polimorfismo relacionados às métricas no *design* do Mezuro são muito importantes para a extensão de outros tipos de métricas. Neste sentido, algumas contribuições foram realizadas ao serviço Prezento relacionadas à refatorações que apoiam a extensão de outros tipos de métricas.

No futuro, com a adoção de novos extratores na plataforma e novas métricas, será necessário o estabelecimento de um novo tipo de métrica (*hot spot*) cuja natureza não está na quantificação e sim na existência ou não de determinada característica no código-fonte. Exemplos desse novo tipo de métrica são métricas de vulnerabilidade e até mesmo os Cenários de Decisões. Entretanto, como dito anteriormente, optou-se por utilização de métricas compostas para a definição dos cenários e não na criação de um novo tipo de métrica.

Na maior parte dos casos de utilização de métricas compostas, os *scripts* são simples e retornam apenas o resultado de uma operação matemática simples entre métricas e números. Entretanto, devido as características de definição dos cenários, principalmente

dos cenários Polimétricos, deve-se utilizar outras estruturas nesses *scripts* tal como estruturas condicionais.

Como mencionado anteriormente, os Cenários de Decisões possuem características de métricas *hot spots* cujo valor medido é booleano baseado na existência ou não do cenário. Como ainda não há suporte a valores do tipo verdadeiro ou falso no Mezuro, os *scripts* de Cenários devem retornar 1 no caso de existência do cenário e 0 no caso de não existência.

Em resumo, a criação da uma métrica composta para um Cenário de Decisão deve conter um script que siga uma estrutura semelhante à listada a seguir:

Listing 5.1 – Estrutura de script básica para Cenários de Decisões

```
if(cenario_existente)
{
  return 1;
}
else
{
  return 0;
};
```

5.1.4 Associando o *Reading Group* ao Cenário

Após criada a métrica composta para um cenário, a última etapa que resta é associar os valores que podem ser obtidos pela métrica composta às interpretações qualitativas do *Reading Group* criado na primeira etapa descrita na presente Seção. Portanto, associaremos os valor obtido 1 ao grupo *Existint Scenario* (em vermelho) e o valor 0 ao grupo *Inexistint Scenario* (em verde) que podem ser vistos na Figura 3.

Para realizar esta etapa basta ir na visualização pública da métrica composta criada e adicionar dois intervalos, como mencionado. Como pode ser observado na Figura 6, para o valor qualitativo *Inexistint Scenario* definiu-se o valor -0.9 à 0.1, enquanto para o valor *Existint Scenario* definiu-se o intervalo de 0.9 à 1.1. Essa definição de intervalos é necessária uma vez o Mezuro não suporta valores de métricas booleanos, conforme discutido anteriormente. A Figura 6 apresenta a visualização pública de um Cenário de Decisão criado através de uma métrica composta no Mezuro.

A seguir será apresentado os scripts utilizadas na criação de cada Cenário de Decisão de Design Seguro no Mezuro. Os códigos utilizados em chamadas de métodos representam as métricas básicas envolvidas no cenário, conforme explorado na explicação de cada cenário.

High Surface Operational Attack

Description: Identify classes that should receive greater attention regarding the application of the principle Mediate Completely due to the greater exposure of the class in operational terms

Script: if(npm() > 10 || anpm() > 3) { return 1; } else { return 0; };

Scope: CLASS

Code: hsoa

Weight: 1.0

Aggregation Form: AVERAGE

Reading Group Name: Hotspot

Ranges

Add Range

Label	Beginning	End	
Inexistent Scenario	-0.9	0.1	Edit Destroy
Existing Scenario	0.9	1.1	Edit Destroy

Figura 6 – Cenário Completo representado no Mezuro. Disponível no Mezuro.org <http://mezuro.org/mezuro_configurations/24/compound_metric_configurations/67>

Alta Superfície de Ataque a Atributos Internos

Listing 5.2 – *Script* do Cenário *High Surface Attack to Internal Attributes*

```
if(noa() == 0)
{
  return 0;
}
else
{
  var result = npa()/noa();
  var value = (result < 0.2) ? 0 : 1;
  return value;
};
```

- **npa** - Número de Atributos Públicos
- **noa** - Número de Atributos

Alta Superfície de Ataque Operacional

Listing 5.3 – *Script* do Cenário *High Surface Operational Attack*

```
if(npm() > 10 || anpm() > 3)
{
  return 1;
}
else
{
  return 0;
};
```

- **npm** - Número de Métodos Públicos
- **anpm** - Média de Parâmetros por Método

Ponto Crítico de Falha

Listing 5.4 – *Script* do Cenário *Critical Point of Failure*

```
if(acc() + noc() > 5)
{
  return 1;
}
else
{
  return 0;
};
```

- **acc** - Número de Classes Aferentes
- **noc** - Número de Filhos

Risco Elevado de Segurança

Listing 5.5 – *Script* do Cenário *High Risk Security*

```
if(accm() > 4 || amloc() > 10)
{
  return 1;
}
else
{
```

```
  return 0;
};
```

- **accm** - Média de Complexidade Ciclomática por Método
- **amloc** - Média de Número de Linhas por Método

5.2 Cenários de Decisão e *Data Warehousing*

Data Warehousing (*DWing*) é um ambiente constituído pelo *Data Warehouse* (*DW*) e as demais ferramentas relacionadas a manipulação de dados, desde a extração até a visualização. Um ambiente de *DWing* não diz respeito apenas das tecnologias envolvidas e sim uma arquitetura que requer o suporte de diferentes tipos de tecnologias (INMON, 2002). As principais tecnologias envolvidas em um ambiente de *DWing* são SGBDs, Sistemas de conversão e transformação de dados (ferramentas de ETL), tecnologias cliente e servidor para dar acesso aos dados a múltiplos clientes e ferramentas de análise e geração de relatórios.

Como o nome sugere, um *Data Warehouse* , em português, significa armazém de dados. Segundo Inmon (2002) um *Data Warehouse* é uma coleção de dados de uma corporação que tem como objetivo dar suporte a tomada de decisão. O DW possibilita a análise de grandes volumes de dados, fazendo com que ele seja o núcleo de muitas soluções de *Business intelligence* (BI) [6]. Uma explicação mais detalhada sobre os conceitos relacionados a um ambiente de *DWing* pode ser obtida no apêndice B.

Neste trabalho procuramos desenvolver mecanismos que nos permitam monitorar e analisar o código fonte através de métricas de forma automatizada e que nos auxilie na tomada de decisão de refatorar ou não refatorar determinadas partes do código. Nesse sentido, foram encontrados alguns trabalhos na literatura que utilizaram um *DWing* para monitorar métricas de processos e produtos de software (FOLLECO et al., 2007) (SILVEIRA; BECKER; RUIZ, 2010)(MAZUCO, 2011) (RêGO, 2014). Dentre estes, o trabalho que se destaca e que serviu como ponto de partida para a criação do modelo dimensional do DW desta monografia foi o trabalho de Rego (2014). Nele, o autor propõe um *DWing* que além de monitorar métricas de um projeto, identifica e monitora cenários de limpeza de código fonte compostos por métricas relacionados a qualidade de software.

O fato de o DW oferecer um alto poder de análise e permitir o tratamento de uma grande quantidade de dados nos faz ter uma expectativa de que esse ambiente pode ser uma solução mais completa em termos de monitoramento e auxílio na tomada de decisão.

[6] *Business intelligence* é o processo de coleta, organização, análise, compartilhamento e monitoramento de informações, oferecendo suporte a gestão de negócios

Nesta Seção, apresentamos a execução das etapas Adaptar e Evoluir Ferramentas e Reproduzir Cenários de Segurança para o estudo de caso sobre o ambiente *DWing*.

5.2.1 Modelagem Dimensional

A modelagem dimensional é uma técnica de modelagem de banco de dados para auxílio a consultas do DW. Ela é de grande importância, se não a principal etapa de desenvolvimento de um ambiente de *DWing*, pois é nessa etapa que os requisitos de negócio são traduzidos em um modelo de dados que permitirá realizar as consultas desejadas. Uma boa modelagem irá permitir um bom desempenho, intuitividade e escalabilidade de um DW.

Antes partirmos para modelagem de nosso DW, precisamos ter definidas as necessidades do negócio. No caso dessa monografia, queremos conseguir monitorar e visualizar quais os cenários estão ocorrendo no projeto e também monitorar o valor de suas métricas que estão envolvidas nos cenários, de modo que, ao identificar que um determinado cenário está acontecendo, seja possível também identificar as métricas que estão relacionadas com a ocorrência do cenário, ajudando assim a identificar quais ações devemos tomar para solucionar essa ocorrência. Também queremos observar esses cenários ao longo do tempo, permitindo a realização de análises por *sprint*, mês, etc. Dessa forma, temos dois fatos claros: (1) um fato para registrar a quantidade de cenários e (2) outro para registrar o valor das métricas.

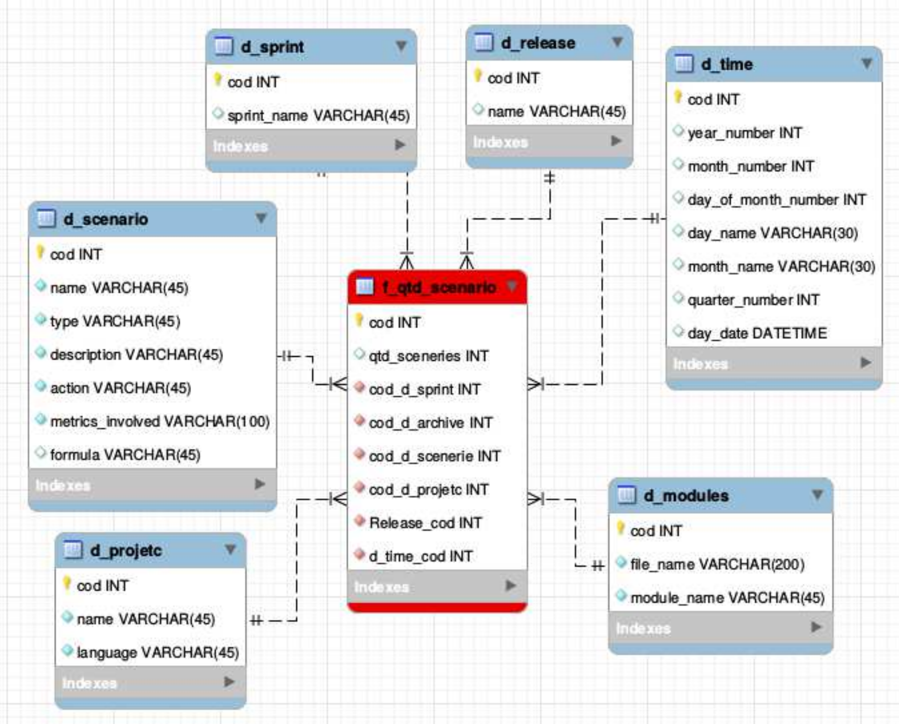

Figura 7 – Modelo dimensional do fato *f_qtd_scenarios* que irá armazenar os cenários identificados no projeto.

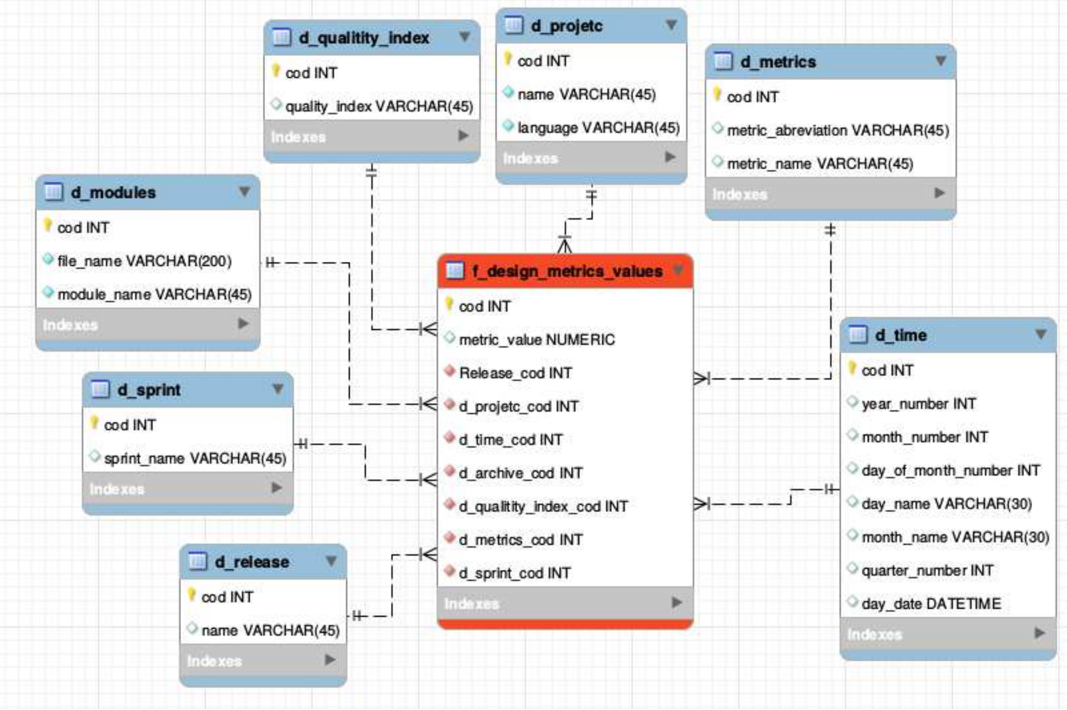

Figura 8 – Modelo dimensional do fato *f_design_metrics_values* que irá armazenar os valores das métricas de cada classe do projeto.

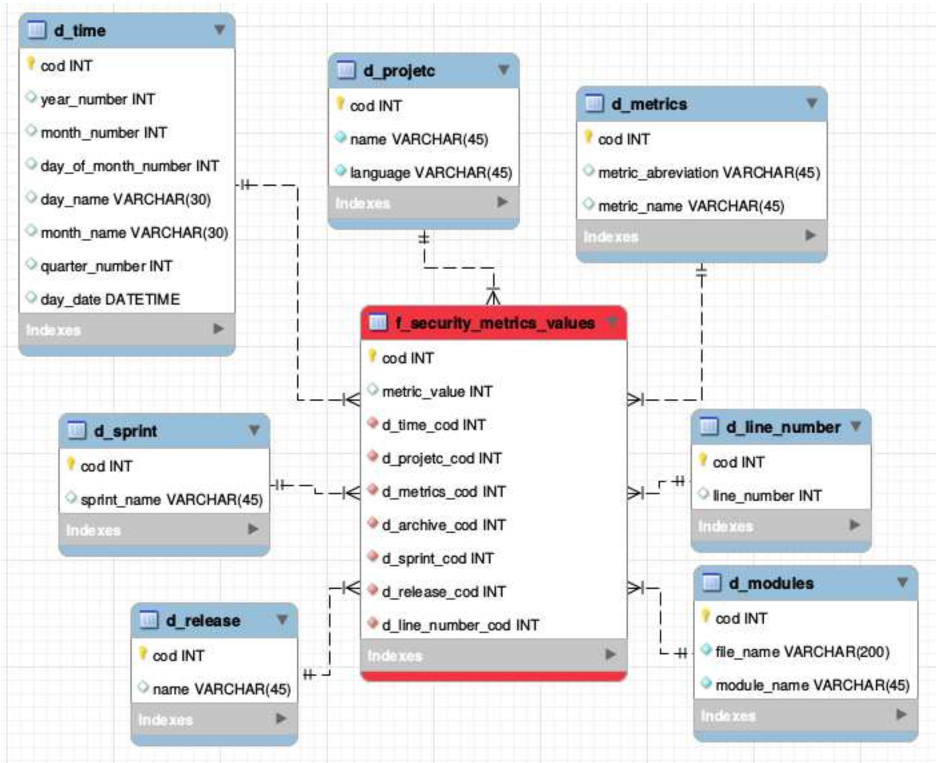

Figura 9 – Modelo dimensional do fato *f_security_metrics_values* que irá armazenar os valores das métricas de segurança de cada arquivo do projeto.

No trabalho de Rego 2014 observamos que a modelagem proposta pelo autor está próxima de atender as necessidades dessa monografia. A modelagem de Rego 2014 permite que o usuário possa identificar cenários de limpeza ao nível de classe de código fonte e também monitorar os percentis dos valores de métricas *design* do projeto. Baseado nisso, chegamos na modelagem mostrada nas Figuras (7) (8) e (9) para o DW deste trabalho.

Alguns aspectos da modelagem do trabalho de Rego (2014) foram modificados e evoluídos para atender melhor a definição dos cenários de decisões proposta nessa monografia. A tabela fato de cenários (*f_qtd_scenarios*) que pode ser vista na Figura 7 foi modelada de maneira muito semelhante. Porém, foram adicionadas a dimensão tempo e de *sprints*. Uma das principais características de um ambiente de *DWing* é permitir a análise temporal dos dados, logo viu-se interessante a inclusão da dimensão tempo. Da mesma forma, a dimensão *sprint* permitirá o usuário a fazer análise da quantidade de cenários por *sprint*.

Outra modificação importante foi feita na dimensão *d_scenarios*. Novos atributos foram incluídos para contemplar a estrutura de cenários de decisões proposta nessa monografia. Foram inclusos os atributos tipo do cenário, métricas envolvidas, fórmula e uma descrição. Uma limitação do trabalho de Rego (2014) foi que os cenários só podiam ser compostos por no máximo duas métricas. Isso se dá ao fato de que Rego utilizou me-

tadados para definir quais métricas estavam envolvidas a cada cenário e automatizar de maneira mais eficiente a identificação de novos cenários. Neste trabalho, levamos em conta que metadados não são disponibilizados para o usuário, preferimos criar o atributo *involved_metrics* e *formula* na dimensão *d_scenario*, permitindo assim que o usuário possa saber quais métricas compõe cada cenário e como elas definem a existência ou não do cenário. Como se trata de um atributo de natureza descritiva, podemos ter cenários que possuem várias métricas envolvidas.

Rego também criou outro fato para armazenar os valores percentis das métricas de um determinado projeto. Para este trabalho, achamos que seria melhor o fato relacionado aos valores das métricas ser a nível de módulo. Dessa forma, o usuário tem mais insumo para identificar a causa de um cenário em uma classe específica. Por exemplo, suponhamos que o usuário tenha identificado a ocorrência do cenário *Ponto crítico de falha* em uma determinada classe. Posteriormente, o usuário pode verificar qual foi o valor das métricas ACC e NOC nesse módulo e tomar uma decisão de refatoração baseado nos valores dessas métricas. No trabalho de Rego (2014), como os valores das métricas estariam disponíveis ao nível de projeto apenas, esse tipo de análise não poderia ser feito.

Em nossa modelagem, preferimos separar em duas tabelas fatos as métricas de *design* e as métricas de segurança. A motivação foi que esses dois tipos de métricas possuem natureza diferente. Métricas de vulnerabilidade contam o número de ocorrência de uma determinada vulnerabilidade no código. Logo, podemos realizar agregações e descobrir a quantidade de vulnerabilidade do mesmo tipo por arquivo, por projeto, etc. Já com métricas de *design*, nem sempre é possível fazer esse tipo de agregação de somatório com seus valores. Outro aspecto desses dois fatos é que, como discutido no Capítulo 4, não temos um índice de qualidade para métricas de vulnerabilidade, tornando essa dimensão inutilizada na fato *f_security_metrics_values*. Por outro lado, é possível chegar ao nível de linha para caracterizar uma vulnerabilidade, incluindo então a dimensão *d_line_number* a esse fato.

5.2.2 Criação do ambiente de *DWing*

O ambiente de *DWing* foi criado conforme a Figura 10.

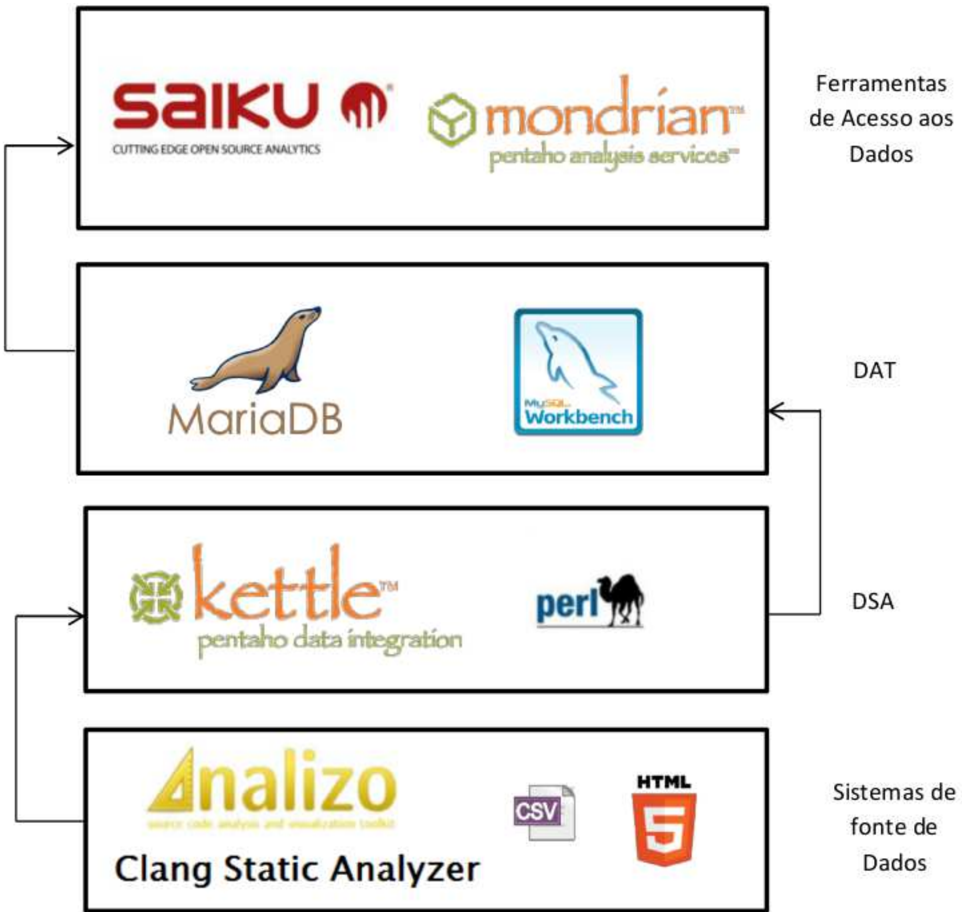

Figura 10 – Ferramentas utilizadas para o desenvolvimento do ambiente de *DWing*.

Como podemos ver na Figura 10, as métricas foram coletadas do Analizo e do *Clang static Analyzer*. O uso do Analizo por si só deveria ser o suficiente, pois este oferece valores de métricas tando de design quanto de vulnerabilidade. Porém, foi identificado um problema com o Analizo que não consegue gerar métricas de vulnerabilidade para projetos maiores. Esse problema será melhor explicado mais adiante na Seção (5.4). Dessa forma, foi utilizado o *Clang-Static-Analyzer*[7] para coletar apenas as métricas de vulnerabilidades.

Os dados são disponibilizados na forma de arquivos CSV, por parte do Analizo, e HTML, por parte do clang. Porém, a fim de facilitar a extração desses dados, foi criado um parser para transformar esse relatório HTML para um arquivo CSV de estrutura parecida com o relatório do Analizo.

Para o processo de ETL foi utilizado uma ferramenta da suíte de ferramentas Pentaho, chamada de *Pentaho Data Integration*[8] (PDI). O PDI aceita diversos formatos de entrada de Dados, além de fazer integração com diversos banco de dados, incluindo

[7] <http://clang-analyzer.llvm.org/>
[8] <http://community.pentaho.com/projects/data-integration/>

o MariaDB [9], que foi o SGBD utilizado para a implementação do DW. É no processo de ETL que são identificados os cenários de decisão. Diferente do Mezuro, que são feitas configurações na própria ferramenta para que ela possa identificar os cenários de decisão, no *DWing* o processo de ETL já identifica o cenário e já passa a informação pronta para ser a armazenada do *Data Warehouse* (qual a classe, release, projeto, entre outros atributos que estão envolvidos com a identificação daquele cenário).

9 <https://mariadb.org/pt-br/>

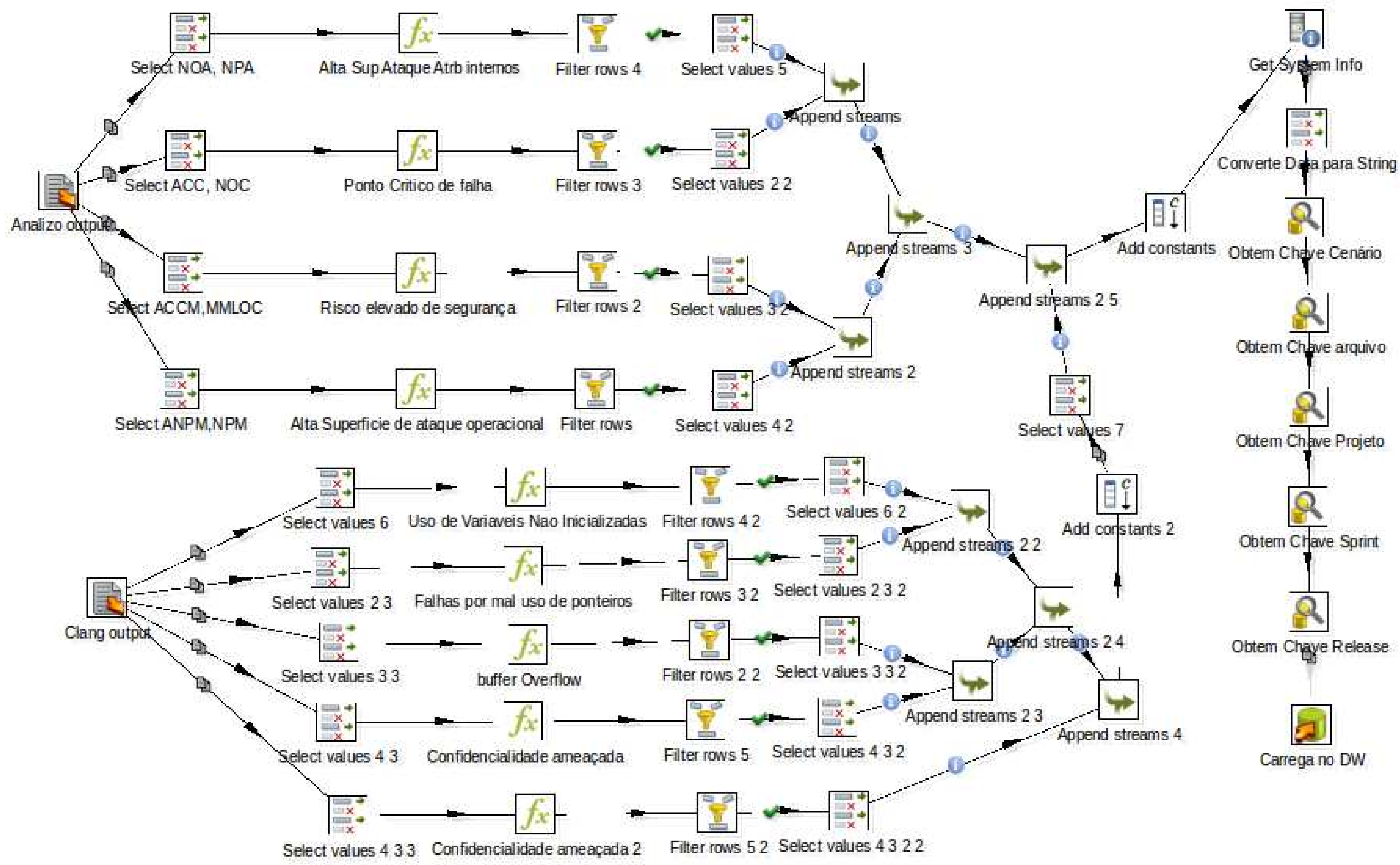

Figura 11 – Processo de ETL para identificação dos cenários de decisão feito no PDI

Na Figura 11 é demonstrado o processo de ETL, elaborado na ferramenta PDI através de uma *transformation*, responsável por extrair os dados, identificar os cenários e carregar a informação no DW. De maneira resumida, os dados são extraídos de arquivos CSV. Esses dados contém informações a respeito das classes e arquivos, e o valor das diversas métricas suportadas pelas ferramentas para cada classe. Para cada cenário, são selecionadas apenas as métricas de interesse para então verificar se o cenário existe o não em determinada classe. É no step *formula* (steps caracterizados pelo *Fx*) onde colocamos a regra de identificação do cenário. Dessa forma, quando o cenário é identificado, este é atribuído a aquela classe. Nos passo posteriores os dados são limpos e preparados até formarem uma tupla que será carregada na tabela de fatos.

Os processos de ETL para carregar os fatos relacionados aos valores das métricas foram mais simples, pois consistiu basicamente em armazenar o relatório fornecido pelas ferramentas. Apenas as métricas que compõe os cenários criados foram selecionadas para serem armazenadas do DW.

Os dados são armazenados no *Data Warehouse*, implementado em um banco de dados MariaDB. Foi utilizado o *MySql Workbench*[10] para criação dos modelos e das tabelas.

Para visualizar os dados, realizar consultas OLAP e gerar relatórios, foi utilizado a ferramenta *Pentaho Business Inteligence Server*[11] (conhecida como BI server). Essa ferramenta ainda inclui a tecnologia de cliente e servidor, possibilitando o acesso de diversos usuários a um sistema de consultas e análise em rede. Ela conta com diversos plugins que realizam diversas operações, tais como, geração de gráficos, visualização de dados em tabelas, etc. O Plugin utilizado para a visualização dos dados e geração de consultas OLAP foi o *Saiku Analytics*[12].

Entretanto, antes de visualizar os dados, é necessário primeiramente criar cubos de dados. Para criação e publicação do cubos de dados, foi utilizado a ferramenta Mondrian [13]. Esta gera um arquivo XML com todas as configurações definidas do cubo, que é usado pelo BI *server* para gerar os relatórios e fazer as conultas OLAP à base de dados. É nessa etapa onde definimos quais são as hierarquias e agregações que poderão ser feitas em cima dos dados no momento das análise.

5.3 Projetos analisados

Nesta monografia definimos a técnica de Cenários de Decisão e propomos um conjunto de cenários para tomada de decisão sobre a segurança do software, seja a partir

[10] <http://www.mysql.com/products/workbench/>
[11] <http://community.pentaho.com/>
[12] <http://www.meteorite.bi/>
[13] <http://sourceforge.net/projects/mondrian/files/schema%20workbench/>

da utilização de métricas de *design* ou através de métricas de vulnerabilidades. Para apresentar como estes cenários podem ser utilizados, foram utilizados três softwares livres cujos resultados do monitoramento irão ser utilizados para demonstrar a visualização dos cenários.

Como discutido no Seção 4.1 do Capítulo de Cenários, nos restringimos a apenas projetos escritos em C++. A seguir é feita uma pequena introdução à cada um dos projetos escolhidos.

5.3.1 GNU Octave

O GNU Octave é um projeto desenvolvido em C++ que consiste em uma linguagem interpretada de alto nível destinada à computação matemática distribuída através da licença GNU General Public License [14]. Ele possui uma interface baseada em linha de comando para a solução de problemas matemáticas lineares e não lineares, assim como possibilita a execução de experimentos numéricos. Além disso, também provê um conjunto extensivo de ferramentas para geração de gráficos, visualização e manipulação de dados.

Outras informações específicas sobre o projeto GNU Octave podem ser obtidas através da documentação oficial do projeto, disponível na página <https://www.gnu.org/software/octave/>.

O código fonte do projeto está disponível na página <ftp://ftp.gnu.org/gnu/octave>. Lá, além da versão atual, podemos ter acesso ao código de releases anteriores.

Este projeto será analisado tanto através do Mezuro quanto através do DWing.

5.3.2 Athom Shell

O Athom Shell é um framework desenvolvido em C++ que permite escrever aplicações *desktop* multi-plataforma através de JavaScript, HTML e CSS. Esse framework é baseado em node.js[15] e no Chromium e é usada no editor de texto Athom[16]. Esse projeto é distribuído através da licença MIT[17].

A documentação oficial do projeto pode ser encontrada no próprio repositório principal[18] através da url: <https://github.com/atom/atom-shell/tree/master/docs>.

Avaliação através do Mezuro pode ser vista em: <http://mezuro.org/projects/33/repositories/58>

[14] <http://www.gnu.org/copyleft/gpl.html>
[15] <http://nodejs.org/>
[16] <https://atom.io/>
[17] <http://opensource.org/licenses/MIT>
[18] <https://github.com/atom/atom-shell>

5.3.3 Synergy

O Synergy é um programa escrito em C++ que permite compartilhamento de mouse e teclado entre computadores que não necessariamente utilizam o mesmo sistema operacional. Além disso permite copiar e colar conteúdos de um sistema operacional para outro. Esse projeto é distribuído através da licença GNU General Public License [19] e possui uma wiki bem completa, indicando até padrões de codificação seguidos no projeto.

A documentação oficial pode ser encontrada em sua wiki através da url: <http://synergy-project.org/wiki/Main_Page>

Este projeto será analisado somente no *DWing*.

5.4 Coleta de Dados

O Mezuro já trabalha em conjunto com o Analizo[20]. O Analizo é uma ferramenta de análise estática de código e dá suporte a diversas métricas, inclusive as métricas discutidas no capitulo 2, tanto de *design* quanto de vulnerabilidade. Um *DWing* não está atrelado a apenas uma fonte de dados, podendo utilizar diversas fontes. Dessa forma, utilizamos o Analizo para coletar as métricas e gerar o insumo para a análise dos cenários em ambas as soluções.

Porém, foi identificado um problema com o Analizo em relação as coleta de métricas de vulnerabilidade. Para projetos pequenos e simples as métricas de vulnerabilidade podem ser coletadas. Contudo, para projetos maiores e mais complexos, com arquitetura bem definida, o Analizo não consegue gerar tais métricas. Isso se deve ao fato do Analizo utilizar o *Clang Static Analizer*[21], que é outra ferramenta livre que faz análise estática para identificação de bugs em projetos C/C++, para a geração das métricas de vulnerabilidade. A utilização do *Clang* consiste em chamar o comando *scan-build* e o comando de compilação do programa, por exemplo "*scan-build gcc myprog.c* ". O Analizo chama o *clang* em seu código para cada arquivo, usando apenas o comando de compilação *gcc* em cada um deles. Isso pode funcionar para projetos simples, porém projetos mais complexos possuem linhas de compilação mais complexas, geralmente gerenciadas em um *Makefile*, com diversas linhas de compilação. Não conseguindo compilar o código, o *Clang* não consegue executar, fazendo com que o Analizo, por sua vez, não consiga extrair as métricas de vulnerabilidade.

Grandes alterações deveriam ser feitas para que o Analizo pudesse gerar os valores dessas métricas para projetos grandes, não cabendo ao escopo e tempo desta monografia. Esse problema afeta diretamente ao Mezuro, pois, dependente do relatório gerado pelo

[19] <http://www.gnu.org/copyleft/gpl.html>
[20] <http://www.analizo.org/>
[21] <http://clang-analyzer.llvm.org/>

Analizo, não consegue os valores de métricas de vulnerabilidade para projetos maiores. Como o ambiente de *DWing* pode ter diversas fontes de dados, podemos recorrer a outra fonte de dados que ofereça apenas as métricas de vulnerabilidade.

Dessa forma, decidimos usar o próprio *clang* para extrair as métricas de vulnerabilidade. O *clang* gera um relatório no formato HTML. Foi feito um parser para transformar o HTML em um arquivo CSV, com a mesma estrutura fornecida pelo Analizo. Assim, o ambiente de *DWing* pode utilizar desse arquivo CSV gerado pelo parser para se alimentar das métricas de vulnerabilidade da mesma maneira que trata o CSV gerado pelo próprio Analizo. O parser foi desenvolvido na linguagem perl justamente para aproveitarmos das lógicas de programação e códigos do próprio Analizo responsáveis pela geração do CSV.

No Mezuro, para se analisar um projeto, deve-se criar um projeto informando o seu nome e descrição[22]. Para cada projeto pode-se fazer diferentes análises, utilizando diferentes configurações, através da criação de Repositórios dentro do Mezuro onde devem ser informados o nome, descrição, a licença, tecnologia de versionamento utilizada, o endereço para o repositório e a configuração desejada. Além disso, o Mezuro possui um recurso interessante para acompanhamento de projetos: a definição da periodicidade para coletas periódicas que varia desde periodicidade diária à mensal, sendo este um recurso opcional.

Uma vez salvo um repositório com a devida configuração e endereço para o repositório, o Mezuro realiza o download do projeto e, através do Kalibro Processor, executa os extratores envolvidos na configuração utilizada sobre o projeto. Assim, o relatório gerado pelas ferramentas são tratados e enviados de volta para o Prezento, onde são apresentados os resultados das medições. Caso a opção de coleta periódica seja utilizada, o Mezuro irá sempre fazer o download da versão mais recente.

No *DWing*, uma vez criado o mecanismo de ETL, utilizando a ferramenta PDI, basta selecionarmos os arquivos CSV quem contém os relatórios da versão do código que se deseja analisar. A seleção dos arquivos CSV são feitas nos steps *Analizo output* e *Clang output*, que podem ser visualizados na Figura 11. O ambiente de *DWing* não permitiu, como no Mezuro, a automatização completa, desde a coleta até a geração das análises. No *DWing* temos que ter os relatórios das ferramentas (Analizo e Clang) em mãos, para então carregar os dados e gerar as análises. Entretanto, é possível automatizar todo esse processo com o auxílio de shellscripts e utilização de Jobs[23] no PDI, mas tais atividades não couberam nos escopo deste trabalho.

[22] <http://mezuro.org/projects>

[23] Assim como as *Transformations*, os *Jobs* são um tipo de componente do PDI que possibilita executar tarefas de mais alto nível, como mandar e-mails, baixar arquivos, executar transformações, etc

5.5 Análise dos Cenários nas Ferramentas

Nesta Seção será apresentada a visualização dos cenários nas duas ferramentas apresentadas através do monitoramento dos projetos apresentados na Seção 5.3.

O projeto GNU Octave foi monitorado através do Mezuro e do ambiente DWing. O projeto Athom Shell foi analisado somente através do Mezuro, enquanto o projeto Synergy foi monitorado somente através do ambiente DWing.

5.5.1 Visualização no Mezuro

A visualização de qualquer monitoramento realizado no Mezuro segue uma apresentação semelhante. Primeiro são apresentadas as informações do projeto, seguido das informações sobre o *status* do monitoramento e data de realização (Figura 12). Logo após é apresentada uma árvore contendo os módulos e classes do projeto e o resultado agregado das métricas medidas (Figura 13). A medida que se realiza uma navegação sobre a árvore de módulos, diminui-se a granularidade dos resultados e tem-se os resultados das medições para cada unidade de software analisada (Figura 14).

Caso algum erro ocorre durante o processo de medição, o erro será informado na aba *Processing information*, apresentada na Figura 12.

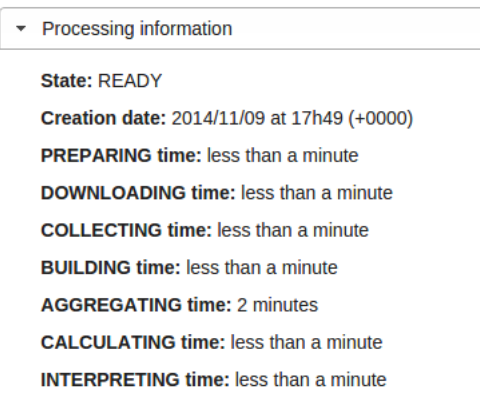

Figura 12 – Apresentação do processo de coleta de métricas no Mezuro

A segunda aba *Modules Tree*, destacada na Figura 13, apresenta todos os módulos encontrados no projeto e seus respectivos níveis de granularidade. Cada item apresentado é um link para o respectivo módulo, onde são apresentados somente os sub-módulos e as medições daquele módulo. Portanto, a navegação entre os módulos e os resultados individuais de cada módulo é bem simples dentro da plataforma, permitindo uma análise detalhada e aprofundada.

A última aba *Metrics Results*, apresentada na Figura 14, lista as métricas básicas medidas e as métricas compostas, com seus resultados agregados, no caso de visualização

Modules Tree

Name: ROOT

Granularity: SOFTWARE

Grade: 0.00

../

Name	Granularity	Grade
atom_main	CLASS	0.00
atom_api_app	CLASS	0.00
atom_api_protocol	CLASS	0.00
atom_browser_client	CLASS	0.00
AtomApplication	CLASS	0.00
AtomApplicationDelegate	CLASS	0.00
atom_url_request_job_factory	CLASS	0.00
resource	CLASS	0.00

Figura 13 – Apresentação da árvore de módulos do Mezuro

de pacotes, ou com seus resultados medidos, no caso de visualização de unidades ou classes.

No Mezuro, o conceito de agregação é aplicado nas métricas básicas, mas não nas métricas compostas. Na visualização de um pacote (root, da Figura 13), os valores das métricas básicas são agregadas de acordo com a escolha feita na criação de cada uma (média, mediana, desvio padrão, máximo e mínimo). Entretanto, as métricas compostas, como não possuem agregação, são calculadas sobre os valores agregados das métricas bases que elas utilizam. Para os Cenários de Decisões esta característica não é muito interessante uma vez que pode apresentar falsos negativos. Como exemplo, se o valor agregado das métricas básicas utilizadas por um cenário não corresponderem à existência do cenário, na visualização geral será informado que o cenário não existe, enquanto na verdade ele pode existir em alguns poucos módulos do software.

Por outro lado, como pode ser observado na Figura 14, o cenário Alta Superfície de Ataque a Atributos Internos é o único que é apresentado como existente no projeto, pois as métricas de Atributos Públicos tiveram uma média alta. Nesse sentido, esse cenário deve ser recorrente no projeto e deve ser o foco de melhorias futuras, uma vez que deve se repetir em várias classes. Entretanto, para esse tipo de análise, seria mais interessante ter-se utilizado a agregação por mediana nas métricas básicas, identificando mais adequadamente se os cenários ocorrem na maior parte das classes.

O mais adequado seria o Mezuro apresentar quantas ocorrências do cenário existem para um pacote, ou seja, as métricas compostas terem suas próprias agregações do tipo SOMA. Alguns passos foram dados no contexto do presente trabalho, onde imple-

Metric Results

Metric	Value	Weight	Threshold
Afferent Connections per Class (used to calculate COF - Coupling Factor)	1.94	1.0	Missing range
Average Cyclomatic Complexity per Method	1.39	1.0	Missing range
Average Method Lines of Code	7.92	1.0	Missing range
Average Number of Parameters per Method	1.07	1.0	Missing range
Number of Attributes	2.41	1.0	Missing range
Number of Children	0.10	1.0	Missing range
Number of Public Attributes	1.38	1.0	Missing range
Number of Public Methods	3.71	1.0	Missing range
Critical Point of Failure	0.00	1.0	Inexistent Scenario
High Risk Security	0.00	1.0	Inexistent Scenario
High Surface Attack to Internal Attributes	1.00	1.0	Existing Scenario
High Surface Operational Attack	0.00	1.0	Inexistent Scenario

Figura 14 – Apresentação das medições do Mezuro

mentamos o suporte à esse tipo de agregação dentro do Kalibro Processor.

A visualização da ocorrência dos cenários dentro de cada classe se torna bem interessante. A Figura 15 apresenta os resultados da classe MessageBox do projeto Athom Shell, avaliado a partir da configuração de cenários criados. O principal foco dos resultados está nos cenários, onde é possível observar claramente que essa classe possui os cenários Risco Elevado de Segurança e Alta Superfície de Ataque Operacional. Além disso, pode-se ainda observar os valores reais obtidos por cada métrica básica para as quais não são providas nenhuma interpretação, uma vez que o foco interpretativo deve estar na natureza do cenário. Pode-se observar ainda que outros dois cenários não caracterizam essa classe.

5.5.2 Visualização no DWing

A visualização dos cenários no *DWing* é feita com o uso do plugin *Saiku Analytics* disponível na ferramenta *BI server*. Para utilizar o *Saiku Analytics*, depois de logado no *BI server*, deve-se clicar na opção *Create New* -> *New Saiku Analytics*. O *Saiku Analytics* permite o acesso a um cubo de dados publicado e a realização de consultas OLAP em cima dos dados de maneira bem intuitiva, ou , pela criação de *querys* MDX [24] .

[24] O MDX (*Multidimensional Expression*) é uma linguagem que foi criada com o propósito de manipular dados Multidemensionals. Possui uma sintaxe semelhante ao SQL

▾ Metric Results

Metric	Value	Weight	Threshold
Afferent Connections per Class (used to calculate COF - Coupling Factor)	0.00	1.0	Missing range
Average Cyclomatic Complexity per Method	1.33	1.0	Missing range
Average Method Lines of Code	18.00	1.0	Missing range
Average Number of Parameters per Method	5.00	1.0	Missing range
Number of Attributes	0.00	1.0	Missing range
Number of Children	0.00	1.0	Missing range
Number of Public Attributes	0.00	1.0	Missing range
Number of Public Methods	3.00	1.0	Missing range
Critical Point of Failure	0.00	1.0	Inexistent Scenario
High Risk Security	1.00	1.0	Existing Scenario
High Surface Attack to Internal Attributes	0.00	1.0	Inexistent Scenario
High Surface Operational Attack	1.00	1.0	Existing Scenario

Figura 15 – Resultado da classe MessageBox do projeto Athom Shell

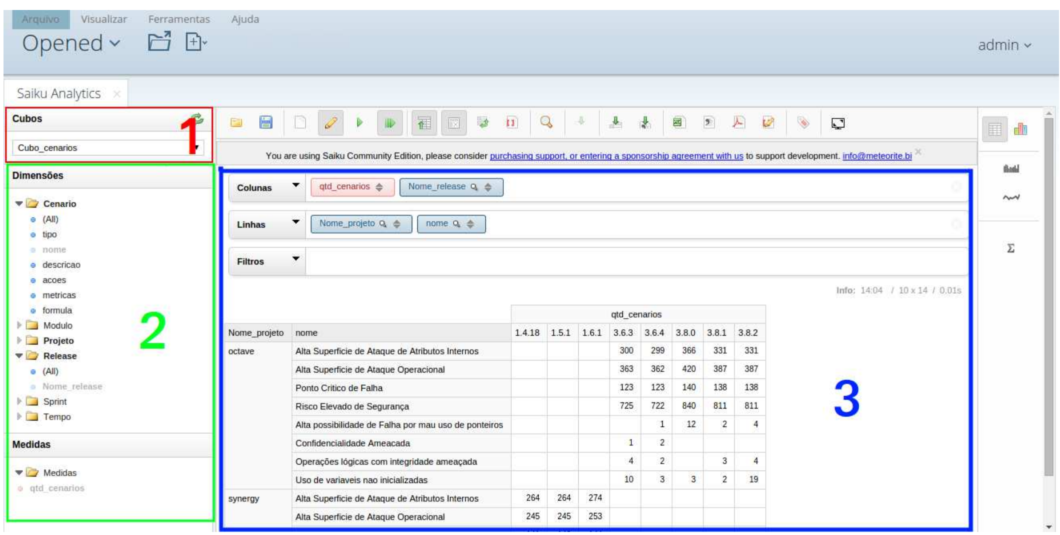

Figura 16 – Workspace do plugin *Saiku Analytics* no BI server

Na Figura 16 podemos ver a área de trabalho do *Saiku Analytics*. Na área demarcada como 1, podemos escolher os cubos que serão analisados. Na área demarcada como 2, é onde podemos ver as dimensões e medidas do cubo escolhido. Ali podemos ver todos os atributos de cada dimensão. É nessa área que selecionamos quais atributos deverão ver no relatório disposto na área 3. Na área 3 é apresentado o relatório, que é totalmente customizado de acordo com a maneira que os atributos são colocados nos campos "Colunas",

"Linhas"e "Filtros".

Para gerar um relatório deve haver pelo menos uma medida envolvida. A partir disso, basta o usuário escolher quais atributos ele quer visualizar, selecionando-os na área 2 e arrastando para os campos "Colunas", "Linhas"e "Filtros"na área 3.

Figura 17 – Atributos que forma as hierarquias para fazer agregações de dados.

Os atributos das dimensões são mostrados de maneira hierárquica em relação a granularidade, sendo que o primeiro atributo mostrado detém menor nível de detalhe e o ultimo atributo mostrado detém o maior nível de detalhe. Na Figura 17 podemos ver a hierarquia da dimensão tempo, que possui vários níveis de detalhe. Porém, existem também dimensões que possuem relações hierárquicas, como a dimensão projeto e a dimensão modulo. Isso permite que eu possa analisar os cenários a nível de projeto (mais alto nível) ou a nível de classe , ou arquivo (mais baixo nível), trazendo mais nível de detalhe. As hierarquias são importantes pois os dados podem ser mostrados de forma agregada dependendo do nível de detalhe desejado.

A primeira vista, podemos na Figura 16 que foram encontrados muitos cenários nos projetos. Como os projetos possuem muitas classes e arquivos e que também existem muitas bibliotecas utilizadas e também um framework por parte do *synegy*, gerar um relatório a nível de classe não seria muito interessante, pois este ficaria muito grande. Nesse sentido, as operações OLAP de *Slice and Dice* podem ser feitas através dos filtros. Isso permite que sejam selecionados os dados de apenas uma release específica, como também classes específicas, arquivos específicos. Permite também analisar apenas os dados relativos a um tipo de cenário. Enfim, com os filtros, diferentes inúmeras consultas podem

ser feitos e relatórios de diversas naturezas podem ser gerados.

Figura 18 – Gráfico gerado pelo *Saiku Analytics* que permite ver a quantidade de cenários de decisões ao logo de cada release

Outra funcionalidade interessante é a possibilidade de geração de gráficos a partir do relatório criado. Na Figura 18 podemos ver o comportamento da quantidade de cenários de decisões encontrados no projeto Octave ao longo de suas releases.

		valor_metrica				
Nome_projeto	Sigla	3.6.3	3.6.4	3.8.0	3.8.1	3.8.2
octave	an	2	2	2	2	2
	asom	1	1	1	1	1
	auv	10	10	10	10	10
	da	19	19	15	15	14
	dnp	9	9	9	9	9
	mlk	2	2	3	3	3
	obaa	1	1	1	1	1
	rogu	12	12	13	13	15
	uaf	1	1	1	1	1
	uav	2	2	1	1	1

Figura 19 – Relatório com a quantidade de vulnerabilidades por release e por tipo de vulnerabilidade específica encontradas no Octave

				valor_metrica
Arquivo	Sigla	nome metrica	linha	3.8.2
liboctave/array/CSparse.cc	da	Dead assignment	1119	1
	rogu	Result of operation is garbage or undefined	4534	1
			5101	1

Figura 20 – Relatório que verifica quais as vulnerabilidades foram encontradas no arquivo *CSparse.cc*, indicando também a linha de ocorrência

As mesmas operações e manipulações dos dados podem ser feitas no cubo que armazena as métricas de segurança. Como a natureza das métricas de vulnerabilidades é contar a quantidade de ocorrências de uma vulnerabilidades específica, esse tipo de métrica pode ser agregada com a operação de soma. Dessa forma, podemos ver o total de vulnerabilidades existente no projeto. A Figura 19 mostra o relatório com o total de vulnerabilidades de cada tipo encontradas nas releases do Octave observadas. Realizando operações de *Drill down*, podemos chegar até na linha que foi encontrada uma vulnerabilidade específica, como mostrado na Figura 20. Isso é bom pois é apontado o local que se encontra a vulnerabilidade e que deve ser refatorado. Não foram encontradas vulnerabilidades específicas no Synergy, apenas no Octave.

5.5.3 Análise e Discussão

Os dois estudos de casos desenvolvidos no contexto deste trabalho tiverem como objetivo representar os Cenários de Decisões em ferramentas de tomada de decisão. O primeiro estudo foi baseado no projeto Mezuro, enquanto o segundo foi baseado em um ambiente DWing para os quais a mesma metodologia de desenvolvimento de estudo de caso foi aplicada.

O Mezuro, sendo uma plataforma já utilizada para monitoramento de projetos livres, foi adaptada para suportar os cenários a partir da evolução das métricas compostas, funcionalidade já existente no Mezuro. Esta evolução se apresentou como uma boa opção dado o tempo e complexidade de se implementar uma estrutura completa de Cenário de Decisão na ferramenta. Além disso, as métricas compostas contém informações fundamentais para a identificação e descrição dos cenários, além de oferecer o recurso de criação de *scripts* para definição dos cenários.

Algumas contribuições técnicas puderam ser feitas à plataforma Mezuro. É importante enfatizar que as evoluções propostas à plataforma sempre foram discutidas e adaptadas com a comunidade de desenvolvedores. Neste sentido, alinhou-se as necessi-

dade deste trabalho às expectativas de melhorias na comunidade.

No Mezuro, a criação de configurações e o monitoramento de projetos pode ser realizado com passos simples, comparado aos passos necessários para reprodução no DW. Da mesma forma, o Mezuro visa oferecer uma interface simples para apresentação dos resultados de medições, oferecendo navegabilidade entre os módulos do software, agregação de valores de métricas e interpretações qualitativas para as métricas.

Entretanto, o fato de não existir agregação para métricas compostas limita a visualização de cenários a nível de pacotes e projetos, uma vez que pode dar margem à erros interpretativos e dificulta um planejamento a partir de uma visão geral sobre o projeto. Por outro lado, a visualização da cenários dentro das classes é simples, completa e informativa, apoiando a tomada de decisões, principalmente para os desenvolvedores, uma vez que o nível de granularidade é menor.

Foi possível representar todos os Cenários de Design Seguro para C++ dentro do Mezuro através de uma configuração que pode ser utilizada por qualquer projeto em C++ que venha a ser avaliado através do Mezuro. Essa configuração é uma contribuição técnica para a plataforma, provida dos estudos realizados nesta monografia. Por outro lado, devido aos problemas já relatados sobre o Analizo, não foi possível utilizar métricas de vulnerabilidades no Mezuro para a construção de novos cenários.

Por fim, vale ressaltar que o recurso de análise periódica oferecida pela plataforma complementa a interpretação através de Cenários de Decisões, podendo apoiar o desenvolvimento e evolução do software diariamente.

Embora o *DWing* seja uma solução de construção mais complexa em relação Mezuro, ele mostrou-se uma abordagem interessante e eficiente para monitorar a ocorrência de cenários de decisões em um projeto de software. O principal destaque está no caráter temporal de análise dos dados, ou seja, no *DWing* é possível armazenar os dados coletados ao longo do tempo, permitindo uma perspectiva sobre o projeto no que diz repeito a saber se está havendo melhoria ou não da qualidade desse software, que no caso deste trabalho o foco foi em características de *design* seguro. Com esse tipo de análise, alinhado com as atividades de desenvolvimento, o Engenheiro de Software pode identificar quais práticas de desenvolvimento estão dando resultado em relação a melhoria da segurança do código, ou quais não estão. Além disso, a solução ajuda na identificação do local onde deve ser feita melhorias e também auxilia na identificação de quais práticas precisam ser melhoradas na equipe por conta do perfil dos cenários mais encontrados.

Diferente do Mezuro, foi possível realizar agregações e também reproduzir todos os cenários propostos. Isso se deu por conta de que não se tratou da evolução de uma ferramenta que já possui suas limitações, mas sim da construção de um ambiente completo que atendesse as necessidades desejadas.

Porém, como já discutido, o *DWing* é uma solução complexa de se construir, principalmente no que se diz respeito aos mecanismos de ETL dos dados. Outro aspecto negativo foi que, nesta monografia, não conseguimos automatizar completamente a carga de dados no ambiente, fazendo com que seja necessário ter os relatórios das ferramentas em mão para então poder realizar a coleta de dados, tarefa que é feita de forma automática pelo Mezuro graças ao Kalibro Processor. A total automatização do ambiente só se daria com a criação de shellscripts e utilização de Jobs no PDI.

Por fim, nota-se que as duas soluções realizam os mesmos passos na coleta e tratamento dos dados medidos, tendo-se uma redundância entre elas. Uma opção seria utilizar o Mezuro como a principal fonte de dados para o DWing, dado que a coleta de dados do Mezuro já é estável e mantida por uma comunidade de desenvolvedores.

6 Conclusões

Métricas são ferramentas importantes que podem ser utilizadas em projetos de software, sendo também muito estudadas na academia e utilizadas na Engenharia de Software Experimental. Entretanto, ainda existem muitas dificuldades inerentes a adoção de métricas nesses projetos. Neste trabalho buscamos apresentar a técnica de Cenários de Decisões como alternativa para monitoramento através de métricas de código-fonte e utilizá-la a partir da evolução e adaptação de duas ferramentas.

A técnica de medição Cenário de Decisão, proposta neste trabalho, se apresenta como uma forma alternativa de utilização de métricas em projetos de software cujos benefícios não foram avaliados experimentalmente no contexto desta monografia. Tomar uma decisão em cima do resultado de uma métrica específica é perigoso, pois uma métrica pode representar várias situações, que podem ser melhor identificadas com a análise de outras métricas. Por isso, a criação de cenários busca identificar situações específicas de código fonte baseado, em sua maioria, em um conjunto de métricas, permitindo assim uma tomada de decisão mais segura.

Observou-se que outros trabalhos já utilizaram conceitos e abordagens semelhantes aos Cenários de Decisões. Sendo assim, outros trabalhos que utilizam ou propõem métricas de software podem vir a utilizar a estrutura proposta de Cenários de Decisões para diminuir os problemas inerentes ao processo de medição. Entretanto, como discutido durante os Capítulos 3 e 4, os Cenários de Decisões possuem potencial para serem utilizados principalmente no desenvolvimento de projetos de software para apoiar o monitoramento do código-fonte e o contínuo processo de melhoria da qualidade interna do produto.

Outro objetivo deste trabalho era aplicar a técnica de medição proposta para melhorar o monitoramento de aspectos de segurança do código-fonte. Para isso, a primeira etapa deste trabalho apresentou uma revisão bibliográfica sobre qualidade interna de software, abrangendo aspectos relacionados ao *design* e segurança.

Com esse estudo, averigou-se que métricas de *design* de código podem ter relações com vulnerabilidades de software, pois foi observado que um código com baixa qualidade e alta complexidade facilitam a inserção de erros e dificultam a identificação de vulnerabilidades já existentes. Além disso, discutiu-se que a aplicação dos princípios de segurança em software envolve decisões de *design*. Dessa forma, algumas métricas de *design* em conjunto podem determinar vulnerabilidades de software, dado que a qualidade interna do código-fonte influencia na inserção de vulnerabilidades e na identificação e remoção destas vulnerabilidades e falhas.

Neste sentido, quatro cenários foram propostos para monitorar a segurança de

projetos de software a partir de métricas de *design* e outros cinco cenários foram criados para monitorar a segurança do código-fonte a partir de métricas específicas de vulnerabilidades, para os quais foram definidos os valores de referencia para softwares desenvolvidos em C++. O primeiro conjunto de cenários pode ser utilizado em diversos contextos, uma vez que as métricas envolvidas são extraídas por vários extratores e não dependem da linguagem de programação do projeto. Por outro lado, o segundo conjunto de cenários são específicos para projetos em C/C++ e tratam de problemas de segurança diretamente ligados a essa linguagem. Nesse sentido, pode ser criado um catálogo de cenários de decisões envolvendo outros aspectos como a qualidade, manutenibilidade, entre outros, que podem ser aplicados para todas as linguagens, como também nesse catálogo pode conter seções para cenários específicos para cada linguagem ou paradigma de programação. A ideia é que uma instância de cenários de decisões possa ser utilizado em diferentes contextos e projetos, desde que a natureza dos projetos (linguagem, paradigma) sejam as mesmas.

Observamos ao longo desta monografia que, para o contexto de vulnerabilidade de software, métricas relacionadas ao *design* de código podem ser mais facilmente agrupadas em Cenários de Decisão do que métricas de vulnerabilidades específicas. Isto acontece pois as métricas de vulnerabilidades identificam ocorrências específicas das mesmas, sendo difícil relacionar uma métrica com outra. Além disso, cada uma dessas vulnerabilidades são documentadas por CWE's específicas e, portanto, suas ocorrências e soluções são bem definidas. Entretanto, o estudo sobre taxonomias de vulnerabilidades sugeriu uma maneira de agrupar algumas vulnerabilidades específicas em vulnerabilidades mais alto nível. Um exemplo disso é a vulnerabilidade *Buffer Overflow*, que pode ocorrer de diferentes formas, como uso de funções como *gets()*, ou pela má manipulação de *arrays*, etc. Baseado nisso, buscou aplicar esse tipo de conceito na criação dos cenários de decisão para vulnerabilidades específicas.

Por fim, a última contribuição do trabalho consistiu na realização de dois estudos de caso que visaram a utilização de Cenários de Decisões em ferramentas de tomada de decisão.

O primeiro estudo consistiu na evolução e configuração da plataforma livre de monitoramento de código-fonte Mezuro para suportar a técnica de cenários. Observou-se através desse estudo que a técnica pode ser utilizada com sucesso dentro da plataforma, onde já existem dois projetos, avaliados no contexto desta monografia, que são monitorados a partir dos Cenários de Decisões de Design Seguro. Neste primeiro estudo de caso, observou-se que algumas limitações favorecem a utilização do Mezuro para acompanhar o desenvolvimento e desfavorecem a utilização da plataforma para práticas gerenciais, no contexto de utilização dos cenários.

O segundo estudo consistiu na evolução de um modelo e ambiente de DW para

observar projetos de software a partir de Cenários de Decisões. Neste estudo, verificou-se que o modelo apresentado suporta adequadamente o monitoramento através de cenários. Além disso, verificou-se que os diversos recursos do ambiente DWing proporcionam um excelente ecossistema para gerenciamento de projetos, fornecendo diferentes formas de informações visuais, temporais e estruturais sobre a qualidade do software.

Tanto as evoluções e cenários criados no Mezuro podem ser utilizados pela comunidade para o monitoramento de outros projetos. Da mesma forma, o modelo e ambiente DWing criado é reprodutível e pode ser adaptado à outros contextos para monitoramento de projetos reais.

Apesar do objetivo da realização dos estudos de caso não ser realizar comparações entre as ferramentas, observou-se que as duas podem ser utilizadas em conjunto para apoiar o monitoramento de código-fonte. Isso se deve ao fato do Mezuro prover mecanismos de rápido e fácil monitoramento de projetos e ser excelente para análises individuais dos módulos do projeto e do ambiente DWing prover um conjunto de mecanismos que facilitam o planejamento e tomada de decisões gerenciais. Em outras palavras, a utilização de Cenários de Decisões no DWing poderia ser utilizado para a realização da prática Medição do Projeto, enquanto o Mezuro poderia apoiar a prática Medição Rápida, ambas descritas na Figura 1.

6.1 Limitações

O Analizo foi utilizado como extrator de métricas, tanto para o Mezuro quanto para o ambiente *DWing*. Porém, o Analizo não estava preparado para realizar a coleta de métricas de vulnerabilidades, não permitindo que o Mezuro tivesse êxito na análise dos Cenários de Decisões de vulnerabilidades específicas.

Além disso, algumas melhorias previamente pensadas para o Mezuro não puderam ser implementadas no contexto desta monografia. Destaca-se a não criação de uma estrutura separada para os Cenários de Decisões para a utilização de métricas compostas. Assim, apesar da flexibilidade e outros ganhos relacionados à esta adaptação, também foram herdados as limitações inerentes à natura das métricas compostas.

Outra limitação a ser destacada foi a não automatização da coleta de dados com o *DWing*. Dessa forma, o ambiente fica dependente do relatório da ferramenta para poder então fazer a análise dos dados. Outro aspecto está na utilização dos metadados, fazendo com que todas as informações e configurações das instâncias dos cenários de decisões fiquem no meio do processo de ETL, e não registrado em uma base de metadados.

Quanto à definição de valores de métricas, limitou-se à instanciação de cenários para a linguagem C++. Para as métricas de vulnerabilidades, diferentemente das de

design, não foram encontrados valores de referências que pudessem ser aproveitados. Além disso, não foi possível realizar um estudo estatístico para comprovar a relação de causa e efeito das métricas de *design* sobre as de vulnerabilidade.

Por fim, vale ressaltar que, apesar dos estudos de casos realizados sobre duas ferramentas de tomada de decisão, não foi escopo do trabalho compará-las e discutir a eficiência das mesmas em relação à observação de cenários. Limitamos à suportar tecnologicamente a técnica de Cenários de Decisões e demonstrar as alternativas geradas por cada ferramenta evoluída.

6.2 Trabalhos Futuros

Quanto aos trabalhos futuros, espera-se realizar um estudo e validar a correlação existente entre métricas de *design* com métricas de vulnerabilidades específicas. A ideia seria verificar através de estudos de correlação estatística a partir de análise de diversos softwares livres para verificar se de fato métricas de *design* tem relação direta com vulnerabilidades específicas. Além disso, com esse estudo também seria possível averiguar se para projetos não críticos, o cuidado com o *design* e a aplicação de princípios de *design* seguros são o suficiente para termos softwares seguros.

Outra possibilidade de evolução deste trabalho seria também a validação da utilização da técnica Cenário de Decisão. Esse estudo poderia se basear em um estudo de caso real, onde seria verificado os resultados significativos em relação a qualidade de código a partir do monitoramento baseado em Cenários de Decisões. Este estudo poderia buscar integrar as duas abordagens de monitoramento apresentado nessa monografia (Mezuro e *DWing*) e utilizar das propostas de Medição Rápida e Medição de Projetos apresentado na Figura 1. Pode-se também verificar se para os cenários propostos o conjunto de ações relacionadas realmente são efetivas na remoção das inconformidades identificadas.

Além disso, muito se tem para contribuir com a busca de valores de referências para métricas, principalmente para métricas vulnerabilidades e o desenvolvimento de ferramentas livres que apoiam a utilização de métricas de código-fonte no desenvolvimento de software.

Appendices

A Design e Segurança de Software

Neste anexo, contém o estudo e conceituação teórica mais aprofundada sobre *design* e segurança de software.

A.1 Design de Software

A.1.1 O Design e suas Práticas

A concepção do *design* para endereçar os problemas de software consiste na aplicação dos princípios de *design* destacados na Seção 2.1.1 e outros princípios a partir de técnicas e práticas no decorrer do desenvolvimento do software. Essa concepção inicia-se na escolha do paradigma que irá reger o desenvolvimento. Um paradigma é constituído basicamente dos princípios gerais que são utilizados para a composição de um software, caracterizando a maneira de se pensar sobre os problemas e suas soluções. A partir desse ponto, um conjunto de práticas são aplicadas com o foco na construção do *design* do software que, no contexto do Processo Unificado, são realizadas principalmente na fase de Elaboração (LARMAN, 2007). Por outro lado, estamos interessados no modelo gradual de desenvolvimento da arquitetura do sistema, proposto pelos métodos ágeis, pois favorece a aplicação de práticas constantes de *design* e pressupõe o desenvolvimento de testes como parte integrada do processo de construção do software. Scoot W. Ambler[1] apresenta um conjunto de práticas que são realizadas durante o desenvolvimento de software nos diversos níveis de abstração para a concepção de um *design* que aplicam princípios ágeis, representadas através da Figura 21.

O conjunto de práticas destacadas na Figura 21 podem ser utilizadas para a aplicação e exercicío dos princípios que influenciam o desenvolvimento de um software que atenda aos requisitos do cliente, qualidade interna e proporcionem o sucesso em termos de custo e prazo. Quanto mais próximo do nível de abstração arquitetural, mais atenção é dada para os elementos e decisões genéricas do sistema. Por outro lado, quanto mais próximo do nível de programação, as práticas destacadas são aplicadas em elementos menores do software, o que influencia mais nos objetivos de nível arquitetural.

A respeito de práticas como *refactoring* e integração contínua, vale ressaltar que essas são aplicadas constantemente pelos Engenheiros de Software, várias vezes por iteração até se atingir os objetivos funcionais e não-funcionais estabelecidos. Durante as iterações e a aplicação das práticas da Figura 21, outros elementos e técnicas de *design* devem ser considerados para se aplicar os princípios de *design*. Além dos paradigmas de

[1] <http://www.agilemodeling.com/essays/agileDesign.htm>

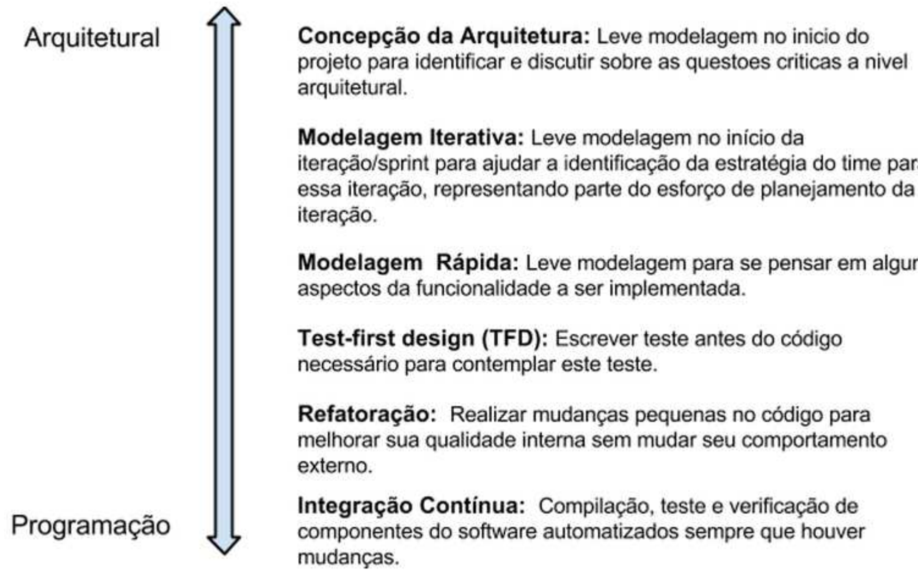

Figura 21 – Práticas do Design Ágil, adaptado de Scoot W. Ambler

programação, padrões de projetos, estilo de código e utilização de *frameworks* são outros exemplos de elementos importantes para a concepção do *design* de um sistema.

A definição do estilo ou padrão de programação é a escolha de um conjunto de regras e diretrizes para a escrita de um software, sendo fundamental para a propriedade coletiva do código[2]. Um padrão de codificação implica que não há pessoalidade no código, facilitando a leitura e evolução do mesmo. Muitas vezes, a definição do padrão para um projeto se baseia em sugestões oferecidas pela comunidade de desenvolvimento de uma linguagem específica e em regras que a equipe de desenvolvimento consideram importantes para suportar a legibilidade, manutiblidade e impessoalidade do software. Estas regras devem estar documentadas e acessíveis para os desenvolvedores do projeto ou até mesmo estarem configuradas em ferramentas especializadas para suportar a aplicação de estilos de programação tal como o *Checkstyle*[3]. Kernighan & Plauger (1978), em seu livro *The Elements of Programming Style*, apresentam e avaliam elementos do estilo de programação de softwares reais, destacando lições aprendidas na análise dos códigos. A noção de padrões de código é estendida para o conceito de Código Limpo, explorado na Seção A.1.3.

Tão importante quanto conhecer padrões e estilo de códigos é conhecer padrões de projetos para aplicação dos princípios de *design*. Um padrão de projeto descreve uma solução geral reutilizável para um problema recorrente no desenvolvimento de sistemas de software que, geralmente, estão relacionados à algum paradigma específico. No conhecido livro *Design Patterns: Elements of Reusable Object-Oriented Software*, Erich Gamma e

[2] Propriedade Coletiva do Código: convenção explícita de que qualquer membro da equipe de desenvolvimento pode modificar e evoluir qualquer pedaço do código, pois qualquer pedaço de código é de todos e não de um só programador.
[3] <http://checkstyle.sourceforge.net/>

seus colegas (1994) definem que um padrão de projeto nomeia, abstrai e identifica os principais aspectos de uma estrutura comum de projeto útil para a criação de software reutilizável. A partir de um estudo empírico, Hegedüs e colaboradores (2012) avaliaram os resultados obtidos a partir da aplição de alguns padrões de projetos em relação aos atributos de qualidade do ISO/IEC 9126 (1998), onde foi observados impactos positivos sobre a manutenibilidade do software. Portanto, os padrões de projetos são ferramentas importantes para o desenvolvimento de softwares com aplicação dos bons princípios de *design* e estabelecimento de uma arquitetura que seja reutilizável, estensível, manutenível, simples e modularizada. Os padrões de projeto devem ser considerados nas decisões em nível arquitetural e aplicados durante a implementação do software, por exemplo, a partir de refatorações como proposto em (KERIEVSKY, 2008).

O apoio de *frameworks* no desenvolvimento do sistema possui impacto direto na qualidade do software, seja esta desenvolvido para o projeto ou por terceiros. Os benefícios da utilização de *frameworks* estão na melhoria da modularidade do sistema, reusabilidade, estensibilidade e inversão de controle provida para os desenvolvedores (FAYAD; SCHMIDT, 1997). Além disso, corroborando os benefícios da utilização de *frameworks*, vale ressaltar que o desenvolvimento de *frameworks* consiste na aplicação extrema de alguns padrões de projetos para provimento de alguns serviços e estabelecimento de controle de operações de uma aplicação, se tornando uma estrutura fundamental que vem sendo extremamente utilizado no desenvolvimento de sistemas (FAYAD; SCHMIDT, 1997).

A.1.2 *Code Smells* - Cheiros de Código

Um software pode ter sintomas (popularmente conhecido como *Smells*) que podem indicar problemas relacionados ao uso de más práticas e a aplicação inadequada de princípios de *design*. *Code smells* não são a causa direta de falhas na aplicação, mas podem influenciar indiretamente para a inserção de erros responsáveis por futuras falhas (FOWLER et al., 1999). Em geral, eles são responsáveis pelas dificuldades de manutenção e evolução do sistema, realização de testes e propicia a inserção de *bugs* (MANSOOR et al., 2014). Por isso, é muito importante saber como identificá-los para se aplicar os mecanismos necessários para sua remoção e, consequentemente, melhorar o *design* do código existente.

O tratamento de mals cheiros de códigos pode ser realizado preventivamente a partir do desenvolvimento da solução com pouca inserção de anomlias e características indesejáveis, a partir de aplicações de práticas de desenvolvimento e de princípios de *design*. Por outro lado, também deve ser tratada constantemente a medida que o código é desenvolvido, através por exemplo da aplicação de *refactorings* como proposto por (FOWLER et al., 1999). Para isso é necessária a identificação de suas ocorrências no código-fonte, que consiste na detecção de fragmentos de código que violam a estrutura ou propriedades se-

mânticas desejadas, provocando acoplamento e complexidade, por exemplo (MANSOOR et al., 2014).

Martin Fowler em seu livro (1999), expõe que a identificação de mals cheiros de código (ou *Bad Smells*) é o primeiro passo para realização de *refactorings* controladas e em pequenos passos. Para tanto, ele explora quais são as principais ocorrências conhecidas de cheiros de códigos que devem ser tratados das quais iremos introduzir algumas:

- **Código Duplicado** - Mesma estrutura de código em mais de um lugar, indicando falta de reusabilidade.
- **Método Longo** - Métodos longos com muitas linhas de código, indicando falta de modularidade, reusabilidade e baixa coesão, dificultando o entendimento do código.
- **Classes Grande** - Classes que possuem muitas linhas de código, atributos e operações. Este mal cheiro indica falta de coesão na classe uma vez que as responsabilidades não são bem atribuídas.
- **Lista Grande de Parâmetros** - Número grande de parâmetros passados para um método. Este mal cheiro dificulta o entendimento do código e do objetivo do método, podendo indicar também falta de coesão, uma vez que o objeto precisa de muitas informações externas para realizar suas operações internamente.
- **Mudança Divergente** - Este mal cheiro acontece quando uma classe é constantemente modificada de diferentes formas por diferentes motivos. Mudanças Divergentes indicam que não há variações protegidas, demonstrando um alto acoplamento entre uma classe e a implementação de classes com quem ela se relaciona.
- **Sirurgia de Espingarda** - Existe quando uma mudança realizada em uma classe afeta o funcionamento de outras estruturas. Este mau cheiro é bem semelhante à Mudanças Divergentes e ambos indicam os mesmos problemas.
- **Dados Aglomerados** - Conjunto de atributos sempre são utilizados em conjunto seja em lista de parâmetros ou em operações em métodos. Este mal cheiro pode indicar uma falta de coesão relacionados a estes atributos, uma vez que seria mais interessante se estes atributos fossem compostos em um objeto mais apropriado para sua manipulação.
- **Estruturas com *Switch*** - Utilização da estrutura de seleção *switch* em algumas linguagens. Este mal cheiro indica duplicação e falta de reutilização de código, uma vez que esta estrutura geralmente é utilizada repetidamente no código para realizar o mesmo controle de fluxo.

- **Classes Preguiçosas** - Classes que não fazem o bastante para justificar sua existência. Este mal cheiro indica falta de coesão e pode indicar a existência de acoplamento desnecessário.

- **Cadeias de Mensagens** - Existe quando um objeto solicita o outro objeto uma sequência de objetos para realizar alguma operação. Indica um forte acoplamento entre essas classes e aplicação inadequada do princípio de abstração.

- **Heranças Recusadas** - Classes que recebem atributos e operações de suas classes mães, mas não gostariam de recebe-los. Esse mal cheiro indica a falta de encapsulamento e muito provavelmente a utilização inadequada de herança.

O entendimento e reconhecimento dos *code smells* são muito importantes para que sua remoção seja feita o mais cedo possível no desenvolvimento. Entretanto, a identificação deles pode ser fragilizada, pois depende diretamente da interpretação e habilidade de identifação do desenvolvedor. Outro problema é não conhecer quais os mecanismos podem ser aplicados para a remoção destes mals cheiros de código. Nesse sentido, o presente trabalho visa contribuir para o desenvolvimento de habilidades e ferramentas que possam ser utilizadas pelo Engenheiro de Software para encontrar as principais falhas de seus softwares e atuar de maneira a melhorar a qualidade interna dos mesmos.

A.1.3 Código Limpo

A definição de um bom código pode ser dada a partir de algumas características desejáveis para um código. No livro *Clean Code* (MARTIN, 2008), o autor aborda sobre um conjunto de características importantes que contribuem principalmente para um bom *design* do código, sintetizando-os em um estilo de programação chamado Código Limpo. Esse estilo de programação pode ser complementado pelas diretrizes propostas no livro *Implementation Patterns* (BECK, 2007), conforme estudado por Almeida & Miranda (2010), pois ambos buscam a aplicação de três características:

- **Expressividade** - um código expressivo pode ser facilmente lido e deixa claro as intenções do autor através de operações e abstrações bem escolhidas. Essa expressividade permite a outros desenvolvedores compreender o código, modificar e utilizá-lo.

- **Simplicidade** - simplicidade é também um dos principais princípios de *design*, e diz respeito à redução de quantidade de informações que o leitor deve compreender para realizar alterações.

- **Flexibilidade** - um código flexível permite que o software seja estendido sem que muitas alterações na estrutura deva ser feita.

Martin ressalva que para se conseguir um código expressivo, simples e flexível deve-se trabalhar em vários aspectos constantemente, desde o nome de métodos à estrutura da solução, pois a construção de um Código Limpo é iterativa e incremental. Esta ideia condiz com o modelo apresentado na Figura 21. Portanto, um Código Limpo é resultado da constante evolução do código com cuidados sobre o seu *design* com o exercício de práticas que aplicam os princípios de *design*.

Alguns aspectos importantes que devem ser sempre considerados para prover um código limpo são resumidas a seguir:

- **Nomes Significativos** - os desenvolvedores são responsáveis pela escolha de nomes de variáveis, classes e operações e, portanto, é de extrema importância que esses nomes revelem bem a intenção, diferenciando bem os elementos que compõe o software. Além disso, a escolha dos nomes são importantes para a abstração e compreensão dos diferentes módulos do software.

- **Métodos Coesos** - métodos são fundamentais para o Código Limpo, pois encapsulam trechos de código, definem escopo de variáveis e são essenciais para a aplicação de princípios de *design*. Nesse sentido, há grande preocupação quanto ao tamanho dos métodos, reduzindo-se não só o número de linhas, mas principalmente a complexidade e número de responsabilidades.

- **Argumentos Reduzidos** - é muito importante um número reduzido de argumentos dos métodos para facilitar sua compreensão, reduzir o esforço de testes e o acoplamento da classe com elementos externos.

- **Classes Coesas** - as classes encapsulam dados e operações, sendo a principal estrutura que compõe o *design* do software. É muito importante que as responsabilidades estejam bem definidas e distribuídas para cada classe de tal forma que haja menos dependências entre elas.

A composição de um Código Limpo é consequência da aplicação de princípios de *design* e de boas práticas de programação. Um Código Limpo é, portanto, o estado desejável para que um software atenda os principais requisitos de qualidade interna.

A.2 Unindo Conceitos de Segurança e *Design*

Alguns módulos do software requerem mais atenção do que outros quanto riscos de vulnerabilidades. Nesse sentido, principalmente em atividades de manutenção e evolução de um software existente, pode ser necessária a priorização do esforço para evolução da segurança do código-fonte voltados para módulos com maior risco. Pode-se, por exemplo,

priorizar a redução da superfície de ataque em módulos mais expostos. Howard (2006) propôs, dentre outras, as seguintes heurísticas para priorização da revisão de segurança de códigos:

- **Códigos antigos**: códigos mais antigos podem ter mais vulnerabilidades do que códigos produzidos recentementes. Isso acontece devido à evolução do entendimento da equipe de desenvolvimento quanto aos possíveis problemas de segurança. Além disso, Howard ainda enfatiza que qualquer código legado deve ser profundamente investigado.

- **Códigos anonimamente acessíveis**: códigos que podem ser acessados por qualquer usuário, mesmo não autenticado, devem ser cuidadosamente revisados.

- **Códigos que escutam em interfaces de rede globalmente acessíveis**: códigos que escutam as interfaces acessíveis de redes por padrão, principalmente de redes desconhecidas como a Internet, devem ser cuidadosamente revisadas e ter monitoramento de vulnerabilidades.

- **Códigos escritos nas linguagens C, C++ e Assembly**: essas linguagens de programação possuem mecanismos de acesso direto à memória e devem ser periodicamente revisadas quanto a vulnerabilidades de *buffer overflow* e de ponteiros inválidos ou inapropriadamente desalocados.

- **Códigos com histórico de vulnerabilidades**: códigos que já apresentaram problemas de vulnerabilidade devem sempre ser foco de novas revisões, a não ser que possa ser demonstrado que as vulnerabilidades apresentadas já foram realmente removidas.

- **Códigos que processam dados sensíveis**: códigos que manipulam dados sensíveis devem ser revisados para garantir que existam vulnerabilidades que permitam o acesso indevido aos mesmos por usuários não confiáveis.

- **Códigos complexos**: códigos que estrutura complexa devem ser periódicamente revisados para investigar possíveis melhorias que diminuam a complexidade. Como já destacado anteriormente nesta monografia, a complexidade é uma das principais inimigas da segurança e pode ocultar vulnerabilidades perigosas.

- **Códigos que mudam frequentemente**: códigos instáveis que são passíveis a mudanças frequentes devem ser revisados a cada grande mudança, pois mudanças podem trazer a inserção de novos *bugs* e vulnerabilidades.

O estudo sobre conceituação de vulnerabilidades conhecidas, de princípios de *design* seguro e da revisão bibliográfica nos permite afirmar que o Engenheiro de Software é

um dos principais responsáveis por manter a segurança de seus projetos. Esse profissional deve se preocupar com os problemas de segurança desde os primeiros passos da concepção do código-fonte e *design* até o desenvolvimento dos últimos testes automatizados. Verifica-se uma forte relação entre princípios de *design* de software com os princípios de *design* seguro, onde a aplicação de ambos podem prover softwares mais robustos, extensíveis e seguros.

Como já mencionado, as decisões de *design* são fundamentais para a concepção de um software seguro. Khan & Khan (2010) enfatizam que a complexidade é o maior desafio para desenvolvedores de software ao projetarem um produto de qualidade que cubra ao máximo aspectos de segurança. Os mesmos autores ainda definem que a complexidade de software orientados a objetos está relacionada principalmente a quantidade de parâmetros de *design* de um objeto e as relações estabelecidas entre os objetos do projeto. Da mesma forma, a complexidade é um dos principais problemas que afetam a qualidade interna do software, dificultando principalmente a manutenção e evolução do software. Mesmo a complexidade sendo uma propriedade da essência do software e não acidental, conforme afirmado por Brook (1986), é importante que os desenvolvedores cuidem da complexidade de seus códigos, pois estes esforços reduzem os impactos negativos diretos sobre a estrutura interna do software assim como na segurança do mesmo. Para tanto, faz-se necessário a aplicação dos princípios de bom *design* e de princípios de *design* seguro através, por exemplo, da prática de *refactorings* e aplicação de padrões de projeto. A seguir são listadas algumas características observáveis no *design* do software que podem indicar complexidade (KHAN; KHAN, 2010):

- Grande número de métodos específicos da aplicação de um objeto afeta a reusabilidade.
- Árvores de herança profundas.
- A grande quantidade de números de filhos de uma classe.
- Alto acoplamento entre objetos.
- Grande número de métodos públicos de um objeto.
- Baixa coesão de classes.

O encapsulamento é uma das principais características de projetos orientados a objetos, sendo fundamental para estabelecer critérios de relação entre as classes de um projeto. Em termos de segurança, esta é outra característica fundamental que deve ser cuidadosamente pensada no *design* de de códigos seguros, pois se relaciona diretamente com os princípios *reduce attack surface* e *mediate completely.*

O nível de encapsulamento também está extritamente relacionado com o princípio *mediate completely*, uma vez que um baixo grau de encapsulamento provê diferentes formas de interações com o objeto que, segundo este princípio, devem ser verificadas sempre. Quando se tem diversos pontos de interação com um objeto as verificações necessárias para prover uma interação segura devem ser mais complexas para explorar os perigos inerentes a cada um desses tipos de interação. Portanto, restringir acessos à alguns métodos e atributos públicos podem beneficiar a aplicação do princípio de *design* seguro *mediate completely*.

Os cuidados com o nível de encapsulamento das estruturas do software tem outros impactos sobre o *design* seguro. A maior parte das decisões de design afetam a complexidade do código-fonte, o que também é verdade para decisões relacionadas a encapsulamentos de classes. A redução dos métodos de acesso diminui as opções de interação com um objeto, tornando a API do objeto mais simples, sendo que o mesmo pode ser dito para a diminuição de parâmetros. Complementarmente, o encapsulamento auxilia na redução do acoplamento entre classes, promovendo maior independência entre os módulos do projeto, aumentando a extensibilidade, manutenibilidade e testabilidade do código. A redução do acoplamento entre classes apoia o *design* seguro, principalmente na detecção e tratamento de vulnerabilidades através de testes e aplicações dos princípios de *design* seguro cujos impactos são mais facilmente gerenciáveis.

Os cuidados com o bom design do código-fonte são fundamentais para o desenvolvimento de códigos seguros. Entretanto, ações específicas devem ser realizadas com objetivos de tratar especificamente das vulnerabilidades inerentes ao código produzido. Em um cenário ideal, essas ações deveriam ser realizadas por Engenheiros de Software ao longo do desenvolvimento, como inspeções de código-fonte para busca de vulnerabilidades. Felizmente, existem estudos e padrões que buscam compreender e definir a existência de vulnerabilidades no software e de que maneiras podemos tratá-las. Tais estudos permitem a criação de ferramentas que automatizam a identificação de possíveis vulnerabilidades no software através de métricas obtidas com a análise estática do código-fonte que podem e devem ser utilizadas por Engenheiros de Software para apoiar a produção de softwares seguros, independentemente da criticidade do sistema.

B *Data Warehouse*

Como o nome sugere, *Data Warehouse* (DW), em português, significa armazém de dados. Segundo Inmon (2002) um *Data Warehouse* é uma coleção de dados de uma coorporação que tem como objetivo dar suporte a tomada de decisão. O DW possibilita a análise de grandes volumes de dados, fazendo com que ele seja o núcleo de muitas soluções de *Business intelligence* (BI) [1].

As informações tem sido cada vez mais valiosas nas organizações. Hoje em dia, as empresas detêm um volume enorme de dados e estes estão espalhados em diversos sistemas diferentes. Diante disso, o processo de gestão desses dados torna-se difícil, dificultando a geração de relatórios com esses dados para a obtenção de informação e para a tomada de decisão. Dessa forma, o DW surgiu para organizar esses dados de tal forma que melhorasse na extração de informação e conhecimento sobre esses dados. Muitas empresas de venda e varejo utilizaram de DW para identificar tendências e obter informações para tomar decisões de marketing, elaborar estratégias de compras e merchandising. Isso se dá ao forte poder de análise oferecido. Informações sobre a organização, quantidade de vendas e de produtos em estoque são mais básicas, entretanto, informações consistentes e precisas sobre o comportamento de seus clientes e histórico dos ultimos anos, por exemplo, são informações que só são possíveis com o uso de DW's.

Neste trabalho procuramos desenvolver mecanismos que nos permitam monitorar e analisar o código fonte através de métricas de forma automatizada e que nos auxilie na tomada de decisão de refatorar ou não refatorar determinadas partes do código. Nesse sentido, foram encontrados alguns trabalhos na literatura que utilizaram o DW para monitorar métricas de processos e produtos de software (FOLLECO et al., 2007) (SILVEIRA; BECKER; RUIZ, 2010)(MAZUCO, 2011). Dentre estes, o que se destaca é o trabalho de (SILVEIRA; BECKER; RUIZ, 2010) que propõe um processo automatizado de extração e carga em um repositório central de métricas de qualidade de software. O autor também oferece suporte a monitoração dos projetos utilizando a técnica EVA e a análise da qualidade interna do software para o auxilio a tomada de decisões sobre os projetos analisados.

O fato de o DW oferecer um alto poder de análise e poder tratar uma grande quantidade de dados nos faz ter uma espectativa de que esse ambiente pode ser uma solução mais completa em termos de monitoramento e auxílio na tomada de decisão. Entretanto, temos a hipótese de que o DW seja uma solução mais adequada para o uso por gerentes ou líderes de projeto, diferente do Mezuro que poderia ser melhor utilizado

[1] *Business intelligence* é o processo de coleta, organização, análise, compartilhamento e monitoramento de informações, oferencendo suporte a gestão de negócios

no uso cotidiano dos desenvolvedores. Mas essas são apenas hipóteses que serão validadas com a real comparação entre os dois ambientes implementados.

Para a construir um DW que auxilie no monitoramento de métricas e na tomada de decisão precisamos entender melhor algumas características e recursos que este ambiente pode oferecer. Dessa forma, iremos entrar em alguns detalhes de suas características e recusos.

O DW tem como característica ser:

- **Integrado**: capaz de integrar dados de diversas fontes de formatos.
- **Orientado a assunto**: um sistema corporativo pode fornecer diversas informações sobre determinados aspectos da corporação. O DW é construído focado em alguns aspectos específicos, como por exemplo, em um processo de desenvolvimento de software podemos analisar vários aspectos como a segurança, qualidade, custo, recursos, etc. Cada aspecto sugere a seleção de apenas dados específicos do sistema que serão uteis para a análise desse aspecto. Os demais dados não são de interesse.
- **Não volátil**: os dados de um DW representam a informação capturada em um determinado momento da aplicação. Em aplicações, os dados estão sempre sujeitos a modificações. Dessa forma, o DW irá capturar novamente, em outro momento, essa informação e irá acrescentá-la ao DW, e não atualiza-la, permitindo visualizar os registros dessa determinada informação em momentos diferentes. Ou seja, os dados de um DW não são modificados, salvo raras excessões.
- **Temporais**: consiste em armazenar a data referente à informação ou à coleta da informação. Dessa forma, o DW consegue fornecer várias visões da informação agrupadas por medidas de tempo.

B.1 *Data Warehousing*

O conjunto de ferramentas de manipulação dos dados, desde sua extração até a sua visualização para o apoio a consultas e tomada de decisão é denominado de *Data Warehousing* (DWing). Portanto, *DWing* não são as tecnologias em si envolvidas e sim uma arquitetura que requer o suporte de diferentes tipos de tecnologias (INMON, 2002). As principais tecnologias envolvidas em um ambiente de *DWing* são:

- SGBDS – Gerenciadores de bases de dados
- Sistemas de conversão e transformação de dados (ferramentas de ETL)
- Tecnologias cliente e servidor para dar acesso aos dados a múltiplos clientes

- Ferramentas de análise e geração de relatórios.

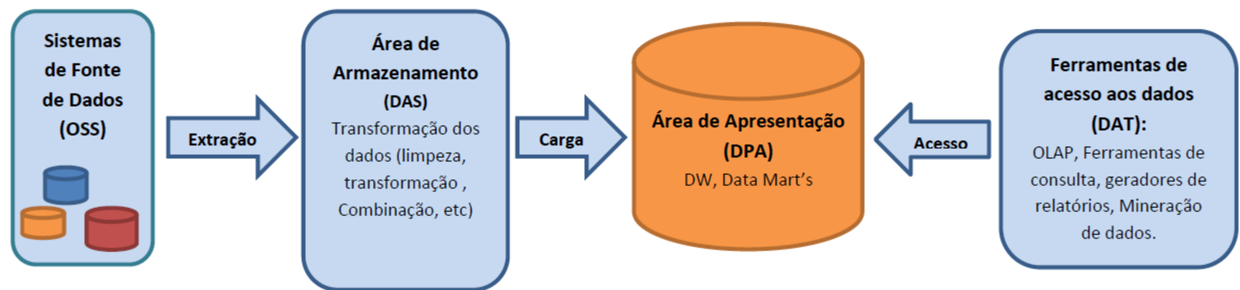

Figura 22 – Componentes de um *DWing*. Adaptação de (KIMBALL; ROSS, 2002)

.

Kimball (2002) define os componentes básicos de um ambiente de *DWing* conforme a Figura 22. A extração dos dados que irão compor o DW é feita de fonte de dados de sistemas, podendo ser de um ou vários sistemas relacionados. Esses dados são tratados na DSA, onde podem ser combinados, limpados ou transformados, tornado-os mais consistentes para que possam então ser amazenados na DPA, que seriam o DW ou *Data Mart's* [2]. As ferramentas de acesso são as responsáveis pela análise dos dados, podendo realizar consultas, gerar relatórios entre outros mecanismos de visualização. Cada um dos componentes serão um pouco mais detalhados a seguir.

Sistemas de Fonte de Dados Operacionais - OSS

O Sistemas de Fonte de Dados Operacionais (OSS - *Operational Source Systems*) são as fontes dos dados de negócio que irão compor o ambiente de DW. Pode ser de origem de várias aplicações que compõe o sistema corporativo de uma instituição, podendo ser unificada com uma única ou várias bases de dados. Não podem ser consideradas dentro do escopo do *DWing*, pois não se tem nenhum controle sobre o conteúdo ou sobre o formato de dados provenientes da fonte (KIMBALL; ROSS, 2002). Essas bases de dados não precisam utilizar necessariamente da mesma tecnologia, podendo ser um banco de dados transacionais, arquivos de texto, planilhas, arquivos XML ou qualquer outra forma de se armazenar e representar informações de negócio.

Área de Preparação dos Dados - DSA

[2] *Data Mart* é um subconjunto de dados de um DW. Foca em uma ou mais áreas específicas (KIMBALL; ROSS, 2002)

A Área de Preparação dos Dados (DSA - *Data Staging Area*) é o ambiente no qual é feita a extração, transformação e carga dos dados operacionais, processo comumente chamado de ETL (*Extract, Transform, Load*). Essa área de preparação, denominada por Kimball (2002), utiliza arquivos simples ou tabelas relacionais temporários, não acessíveis aos usuários, para armazenamento e manipulação das informações durante o processo de ETL. Assim que os dados estiverem prontos é feito a carga na base de dados dimensional, base essa acessível ao usuário. A etapa de extração dos dados consiste em ler e entender as fontes de dados e extrair apenas os dados necessários para o DW. Os dados extraidos na camada OSS não são integrados, e, segundo Inmon (2002), esses dados, dessa forma, não podem ser utilizados para dar suporte a uma visão corporativa, que é uma das essências do ambiente de *DWing*. Dessa forma, uma série de transformações podem ser feitas buscando a limpeza de dados (resolução de conflitos, tratamento de informações não existente, conversão de dados para um formato padronizado), combinação de dados de diversas fontes, remoção de dados duplicados e atribuições de chaves que serão utilizadas no DW (KIMBALL; ROSS, 2002). Ao final da transformação, as informações necessárias são selecionadas e incluídas na base de dados multidimensional, encontrada na área de apresentação que será tratada em seguida.

Área de Apresentação - DPA

A Área de Apresentação dos dados (DPA - *Data Presentation Area*) é onde os dados são organizados, armazenados e disponibilizados para consultas á usuários ou ferramentas de geração de relatórios ou análises. Esse ambiente pode ser materializado no DW em si (KIMBALL; ROSS, 2002).

Um dos principais propósitos de um DW é ter uma navegação intuitiva e de alta performance (KIMBALL; ROSS, 2002). A visualização das informações de um DW é resultado de agregações de dados, ou seja, diferentes formas de agrupamentos de dados que geram diferentes tipos de informações. Essas agregações são caracterizadas pelas consultas OLAP. Dessa forma, a modelagem relacional, normalmente utilizadas em bancos de dados transacionais, não dão suporte esses propósitos, pois a base de dados é normalizada e a agregação de grande número de dados torna-se custosa em termos de performance. O processo de normalização foi inicialmente proposto por Byce Codd que consiste em esquematizar as relações de uma base de dados relacional, minimizando redundância de informações e anomalias de inserção, exclusão e alteração, no que diz respeito a manter a consistência de dados caso haja a duplicidade deste. Dessa forma, uma base de dados relacional normalizada oferece mais segurança em transações de acesso e alteração/inclusão/deleção de dados pois possui esquemas menores e consistentes (ELMASRI; NAVATHE, 2006). O problema de esquemas menores é a performance, pois a realização consultas exige a navegação entre várias tabelas, e se tratando de uma consulta que en-

volve grande quantidades de dados, como em um *DWing*, a performance se torna um fator muito importante para a qualidade do ambiente.

Kimball (2002) então propõe o uso da modelagem dimensional, que possui as mesmas informações que as bases normalizadas, porém em um formato diferente, atendendo a facilidade na navegação e na performance das consultas. Mais detalhamento sobre a modelagem dimensional será vista na Seção (B.2).

Ferramentas de Acesso de dados – Visualização de dados:

As Ferramentas de Acesso a Dados são ferramentas que tem a capacidade de realizar consulta aos dados da Área de Apresentação. Elas podem variar desde simples ferramentas de consulta *ad-hoc* até ferramentas de análises complexas e de mineração de dados (KIMBALL; ROSS, 2002).

Ferramentas de visualização são muito importantes em um ambiente de *DWing*, pois são essas ferramentas que irão facilitar o acesso a informação coletada, que é o primeiro requisito de um abiente de *DWing* listado por Kimball (2002). Um dos modos de se apresentar os dados é através de relatórios, que são vistas agregadas da informação que sejam relevantes para a tomada de decisão. Muitas ferramentas permitem a realização de consultas OLAP *ad-hoc* para a criação de relatórios customizados ou até mesmo a definição de relatórios definidos que são utilizados com frequência. A partir desses relatórios podem ser extraídos gráficos, sendo este mais um meio de facilitar a visualização da informação.

Outra forma de visualização de dados é dada através de *dashboards*, que são muito úteis para o monitoramento de indicadores por ser basicamente uma tela com informações resumidas com diferentes elementos de exploração como tabelas, graficos, manometros, entre outros.

B.2 Modelagem dimensional

A modelagem dimensional é uma tecnica que define bem um ambiente de DW. Segundo Kimball 2002 a modelagem dimensional é a única técnica viável para banco de dados que devem responder a consultas em um DW.

A modelagem dimensional é mais simples, mais expressiva e mais fácil de compreender do que a modelagem relacional (BALLARD et al., 1998). A modelagem dimensional busca obter um modelo que representa um conjunto de medidas que são descritas por aspectos comuns de negócio. Os conceitos básicos da modelagem dimensional são:

- **fatos**: são dados que contém medidas e seu contexto. Cada fato pode representar uma transação do negócio ou um evento que pode ser usado para análise do próprio

negócio. São instâncias da realidade que podem ser mensuradas de maneira quantitativa (KIMBALL; ROSS, 2002). Em um DW os fatos são armazenadas em tabelas de fatos, que consomem cerca de 90% do espaço de uma base de dados dimensional. Tabela de fatos é a principal tabela de um modelo dimensional (KIMBALL; ROSS, 2002; BALLARD et al., 1998).

- **dimensões**: determinam os detalhes do contexto em que foi obtido um fato. São tabelas que contem as descrições textuais de negócio e ajudam na identificação de um componente da respectiva dimensão. Cada fato se relaciona com várias dimensões, associado a apenas a um dado em cada uma dessas dimensões.

- **medidas**: é um atributo numérico do fato. A medida determina a performance ou o comportamento de aspectos do negócio. As medidas são determinadas como combinações de membros de dimensões e alocados na tabela de fatos.

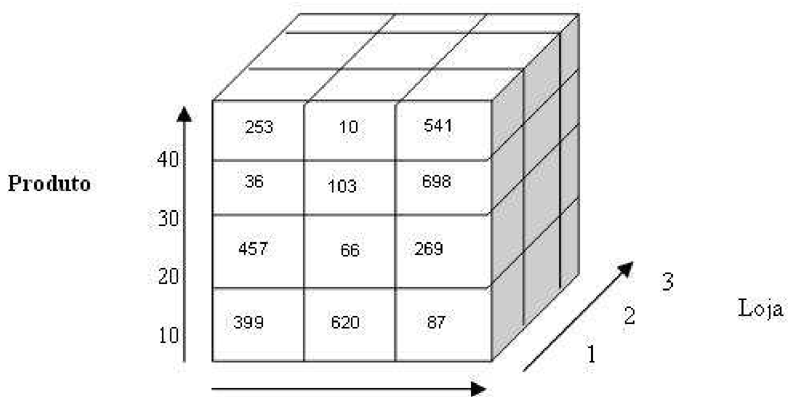

Figura 23 – Exemplo de cubo de dados Fonte: (GUIMARAES, 2012)

.

A ideia da modelagem dimensional é representar os tipos de dados de negócio em estruturas de cubo de dados. As células desse cubo contém os valores medidos e os lados definem as dimensões. Na Figura 23 é exemplificado um cubo de dados para o contexto de uma loja. As células desse cubo pode representar a quantidade de vendas, sendo possível a realização de diferentes análises.

O modelo dimensional proposto por Kimball (2002) é chamado de modelo estrela (*star scheme*, Figura 24). Nele, temos a tabela fato no centro e varias tabelas dimensões se relacionando com essa tabela fato. As tabelas fatos devem possuir duas ou mais chaves estrangeiras para as chaves primárias de diferentes dimensões. Para juntar as informações basta realizar um *Join* entre elas. Uma Tabela de Dimensão deve ser construída de maneira a incluir atributos que podem ser agregados, fornecendo ao usuário maneiras alternativas

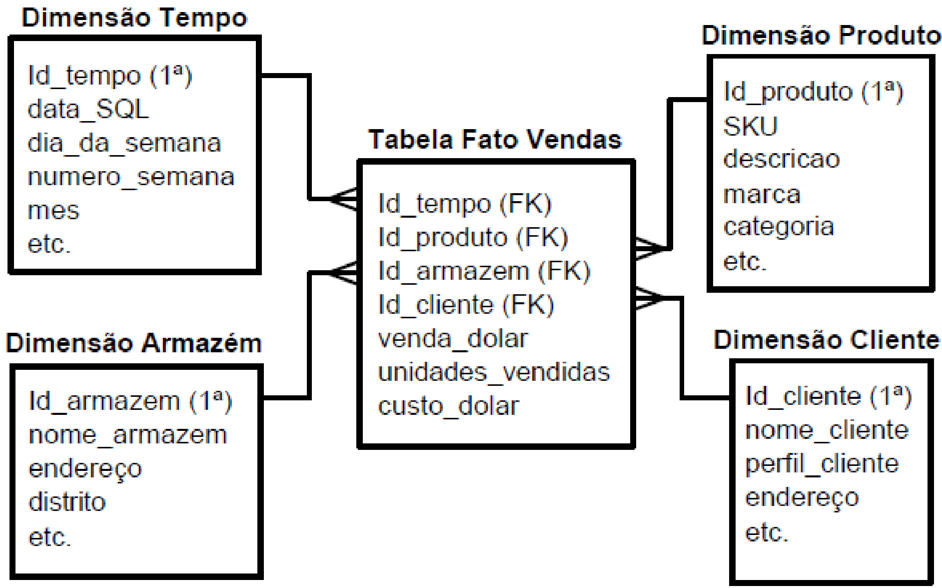

Figura 24 – Esquema estrela Fonte: (WAGNER, 2012)

.

de visualizar as informações. No contexto desse trabalho, essa característica nos permite analisar, por exemplo, quais os métodos com mais vulnerabilidades e subir o nível de hieraquia, agregando os dados para visualizar as classes com mais vulnerabilidades, os pacotes, projetos, e assim por diante. Diferente do modelo relacional, o fato da tabela dimensão não ser normalizada implica na melhoria da performance, pois nesse exemplo citado, "Classe" do método normalmente seria outra tabela, e para a referida análise seria necessário a realização de um *Join*, que foi substituído apenas por uma clausula *Group by*.

Porém, tabelas dimensões podem ser normalizadas com o intuito de diminuir o uso do espaço de armazenamento de informações redundantes (KIMBALL; ROSS, 2002). Nesse caso, quando uma dimensão é normalizada passa-se a ter o esquema floco de neve, como pode ser visto na Figura 25. Esse esquema torna mais fácil a manutenção de dimensões, porém é aconselhado o seu uso apenas em situações que realmente seja necessário abrir mão da performance que o esquema estrela oferece. A modelagem dimensional facilita o processamento analítico dos dados (OLAP), aspecto que será tratado na Seção B.3. Kimbal (2002) define quatro passos que guiam o processo da modelagem dimensional, que são:

- **Selecionar o processo de negócio a ser modelado**: consiste em definir qual o assunto no qual o *DWing* será orientado. Como ja foi explicado, o DW é orientado a assunto, e tomando como exemplo um sistema de vendas de uma loja, pode ser feita a análise sobre as vendas, sobre o estoque, etc. Selecionar o processo de negócio é selecionar qual desses assuntos serão analizados e modelados.

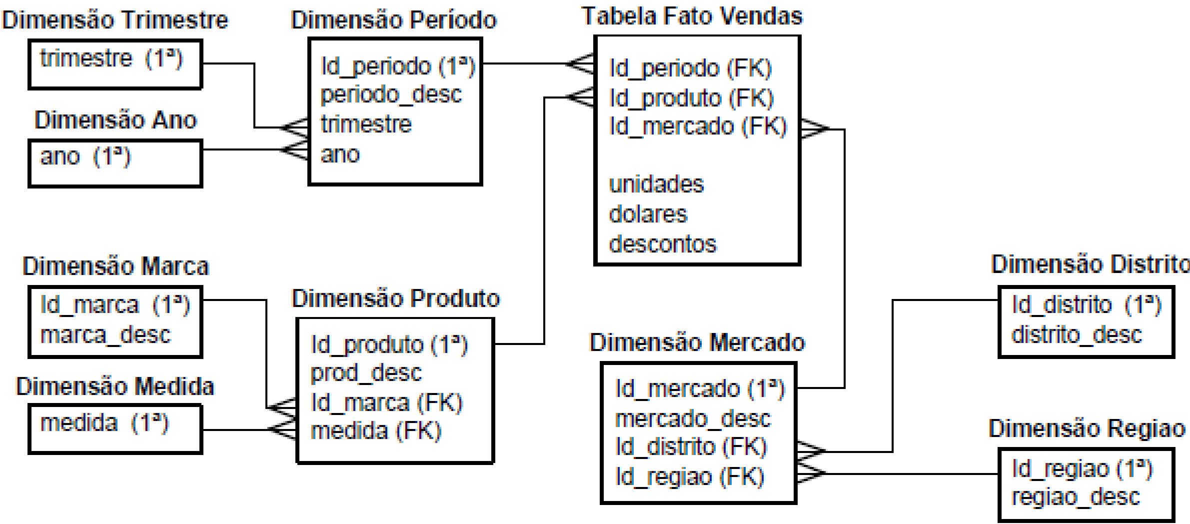

Figura 25 – Esquema floco de neve Fonte: (WAGNER, 2012)

.

- **Declarar o *"grão"* do processo de negócio**: significa especificar exatamente o que uma linha da tabela de fatos representa. Nessa etapa é definido o nível de glanuralidade da informação. Por exemplo, visualizar as informações por dia ou por mês. A granularidade se refere ao nível de detalhe que o fato deve ter.

- **Escolher as dimensões**: consiste em definir as dimensões que se aplicam acada linha de fato que foi definido. Se o nível granularidade foi bem definido, é fácil definir as dimensões.

- **Identificar fatos**: consite em responder a pergunta ö que estamos medindo?: Nessa estapa é definido os fatos numéricos que irão popular as tabelas fatos.

B.3 OLAP

O Processamento Analítico *On-Line* (OLAP – *On-line Analytc Processing*) é toda atividade de consulta que busca trazer ao usuário uma visão analítica dos dados através de comparações, visões personalizadas, análises históricas, diferentes cenários e entre outras opções (KIMBALL; ROSS, 2002). Pode-se definir OLAP como sistemas ou ferramentas que realizam consultas ao DW. Tais sistemas permitem aumentar ou diminuir o nível de detalhes da informação através das operações descritas abaixo.

- ***Drill down***: Consiste em navegar em uma informação de menor nível de detalhe para uma informação de maior nível de detalhe. Por exemplo, uma análise utilizando a dimensão tempo fornece o tempo por bimestre. Uma operação de *Drill Down*

consistiria em trazer essa mesma informação por mês. Este exemplo pode ser visto na Figura 26.

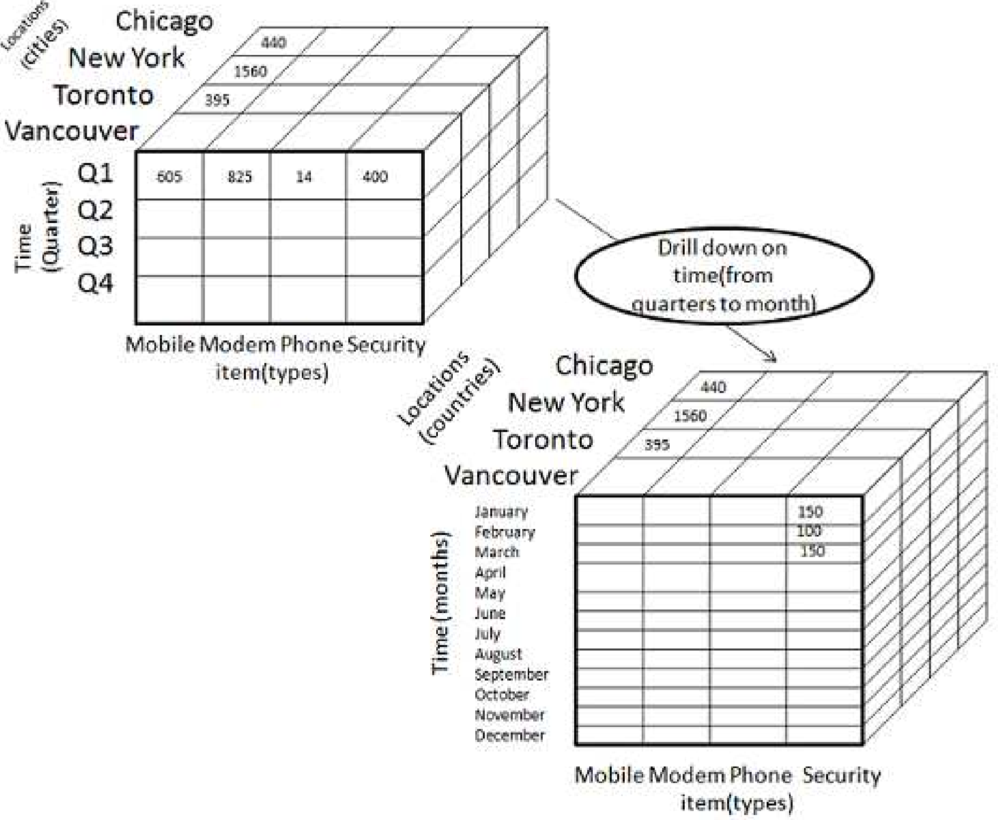

Figura 26 – Demonstração da operação de *Drill Down* Fonte: (TUTORIALSPOINT, 2014)

- ***Roll up***: Consiste em navegar em uma informação de maior nível de detalhe para um menor nível de detalhes. É exatamente o inverso do *Drill down*.

- ***Slice and dice***: A operação de *slice* consiste em fatiar o cubo, que consiste em selecionar um atributo de uma dimensão específica e olhar apenas as informações das outras dimensões sobre esse atributo, eliminando a dimensão fatiada. Por exemplo, dado o cubo demosntrado na Figura 27, a operação *slice* foi feita ao selecionar apenas as informações do primeiro bimestre (Q1), eliminando qualquer outra informação da dimensão de tempo. A operação *dice*, como o nome sugere, consiste em fatiar em formato de cubo. Nessa caso, não será eliminado nenhuma dimensão, mas será selecionado alguns subgrupos em duas ou mais dimensões, resultando em um subcubo. Por exemplo, a operação de *dice* no cubo representado na Figura 28 consistiu em selecionar apenas as informações de Toronto ou Vancouver, no primeiro e segundo bimestre e de apenas itens Mobile ou Modem, gerando um subcubo menor.

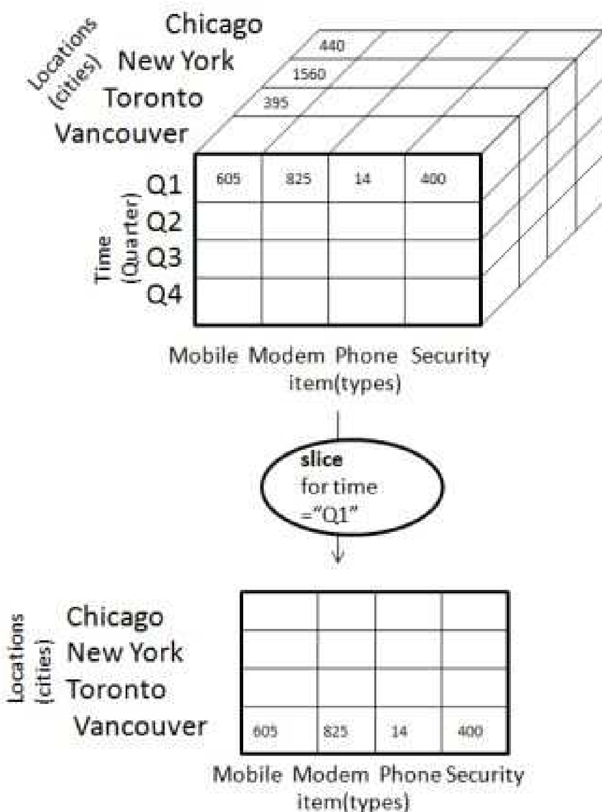

Figura 27 – Demonstração da operação de *Slice* Fonte: (TUTORIALSPOINT, 2014)

- ***Pivoting***: Também conhecida como *rotate*, é uma operação que realiza uma rotação nos eixos de um cubo, gerando uma visualização alternativa da informação (CAVALCANTI, 2012). O resultado de uma operação de *pivoting* pode ser visto na Figura 29.

B.4 Ciclo de vida de um ambiente de *Data Warehousing*

Kimball (2002) define um ciclo de vida para o processo de construção de um ambiente de *DWing*. Neste ciclo, a primeira atividade a ser realizada é o planejamento do projeto. Essa atividade consiste em avaliar a iniciativa de construção do *DWing*, estabelecendo um escopo inicial e a justificativa, como também contempla a obtenção de recursos e o lançamento do projeto.

A próxima atividade é a definição dos requisitos de negócio. O alinhamento do *DWing* com o s requisitos dos usuários é absolutamente crucial. Não adianta construir o ambiente com as melhores ferramentas do mercado se o este não fornece a informação que o usuário precisa ver. Dessa forma, mais de 50% das iniciativas de *DWing* não tem sucesso (SEN, 2011). Pelo levantamento feito na pesquisa de Kimpel (2013), a principal causa é o não entendimento do problema que o usuário de negócio quer solucionar. E é na etapa de requisitos que o problema e as necessidades devem ser entendidos e transformados em

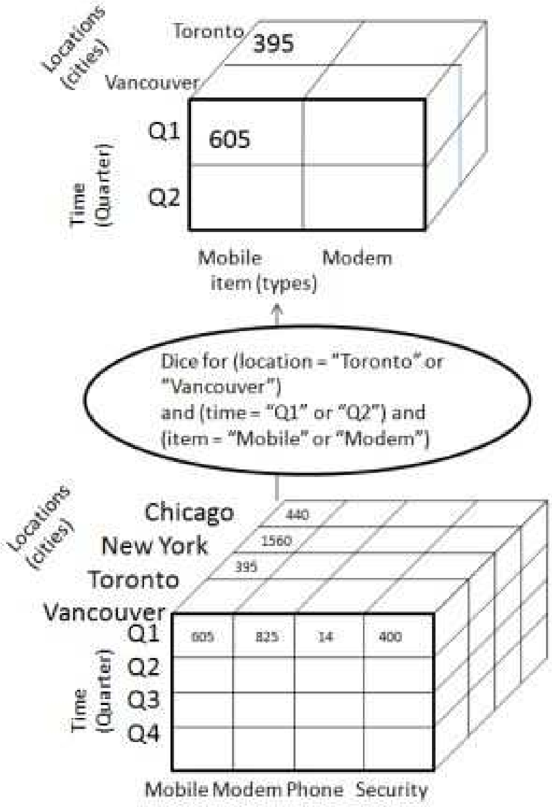

Figura 28 – Demonstração da operação de *Dice* Fonte: (TUTORIALSPOINT, 2014)

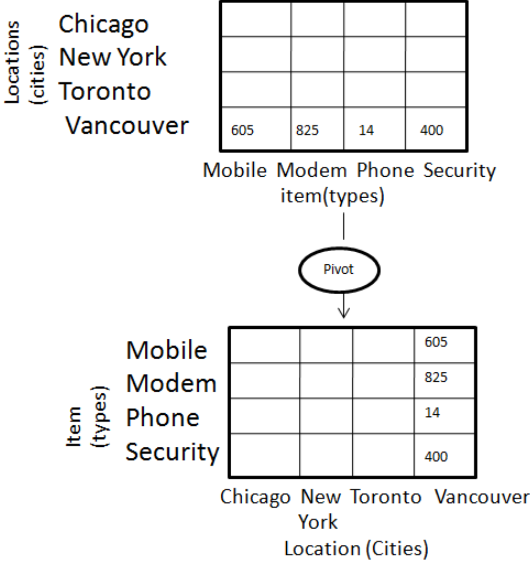

Figura 29 – Demonstração da operação de *Pivoting* Fonte: (TUTORIALSPOINT, 2014)

requisitos de negócio para as etapas seguintes, de modelagem e construção do ambiente.

Com os requisitos definidos, existem três conjuntos de tarefas. As tarefas superio-

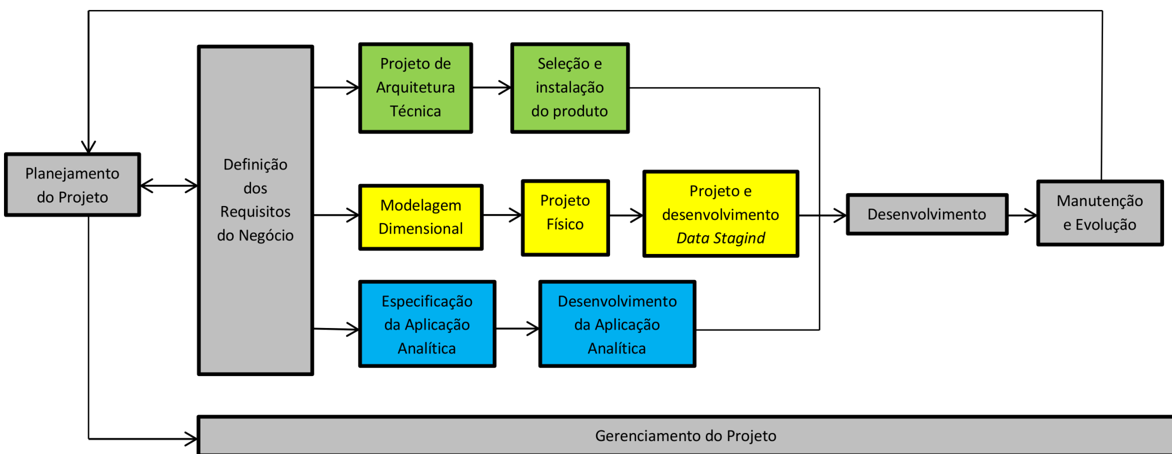

Figura 30 – Ciclo de vida de um Projeto de *DWing* (KIMBALL; ROSS, 2002)

.

res (na cor verde) da Figura 30 são responsáveis pela concepção tecnológica do ambiente. Consiste na definição da arquitetura técnica e seleção das tecnologias envolvidas na solução de *DWing*. O conjunto de tarefas que se encontram no meio do ciclo (na cor amarela) são responsáveis pelo desenho do modelo dimensional e físico e a definição e desenvolvimento do processo de ETL dos dados. O conjunto de tarefas inferiores (na cor azul) são responáveis pelo desenho e desenvolvimento das aplicações analíticas, que irão fornecer a visualização da informação gerada pela *DWing* para o usuário. Juntando esses três conjuntos de tarefas temos a implantação e disponibilização do ambiente de *DWing* para o usuário, então pode-se dizer que a solução de *DWing* foi implantada e pode começar a ser utilizada.

No fim do ciclo ainda existe uma atividade de manutenção e crescimento do ambiente, dado que esse é um tipo de ambiente que deve evoluir dinamicamente de acordo com o negócio e suas necessidades de tomada de decisão.

B.5 Construção de ambiente de DW para o monitoramento de métricas de software.

Nas seções anteriores foi feito uma revisão teórica sobre as principais caracteríticas de um ambiente de *DWing*. Neste trabalho, buscamos desenvolver um ambiente de *DWing* que auxilie no monitoramento e na tomada de decisão, baseado em métricas de software. Dessa forma, a proposta é desenvolver um ambiente que nos permita coletar métricas de ferramentas de análise estática de código, identificar o cenários de decisão a partir da correlação dessas métricas (assunto que será abordado no Capítulo 4) e sugerir refatorações para solucionar esse cenário. A suíte de ferramentas Analizo, como fornecedor de métricas, irá compor o componente OSS da arquitetura de um *DWing* discutida no início

desta seção. O Pentaho é uma suíte de ferramentas de BI que oferece desde mecanismos de ETL, *reporting*, OLAP e mineração de dados. Essa ferramenta nos permitirá extraír, tratar e analisar os dados de relatórios do Analizo a fim de identificar os cenários de decisão. Os recursos OLAP irão permitir diferentes visões que permitirá identificar diversas características que poderão auxiliar na tomada de decisão, como qual a classe que possui mais cenários, qual cenário é o mais recorrente, além de poder extrair indicadores a partir dos dados e analisar o histórico das informações, verificando se houve melhoria ou não na situação do projeto.

Percebe-se que vários recursos de um ambiente de *DWing* podem ser explorados a fim de dar mais informações a respeito de um projeto de software a partir de métricas. Entretanto, tais aspéctos e configurações da solução serão ainda definidos e refinados para a composição da solução final.

C Mezuro: Uma plataforma de Monitoramento de Código-Fonte

No Capítulo 2 discutimos as principais questões de *design*, segurança e métricas no contexto da Engenharia de Software. As métricas estáticas de código-fonte são recursos que podem ser utilizados para a compreensão da qualidade e identificação de vulnerabilidades do código-fonte. Entretanto, como discutido no mesmo capítulo, avaliar a qualidade de um software envolve um processo dispendioso e não trivial. No Capítulo 3, apresentamos um conjunto de métricas estáticas cujo propósito é medir alguns atributos do software tais como complexidade, acoplamento e tamanho.

Entretanto, o esforço necessário para extrair as métricas estáticas manualmente é imensurável, dificultando ainda mais a adoção dessas métricas em projetos de software. Portanto, faz-se necessário a utilização de ferramentas que auxiliem a utilização de métricas de código-fonte, automatizando a extração e visualização dessas métricas. Porém, existem poucas ferramentas disponíveis, e muitas delas nem sempre são adequadas para análise de determinados projetos de software (MEIRELLES et al., 2010). Isso também decorre devido a existência de várias ferramentas extratoras independentes, que seguem seus próprios padrões e oferecem um conjunto de métricas limitados que podem não se adequar para determinados contextos.

Em virtude do que foi mencionado foi criado o Mezuro[1], uma plataforma livre para monitoramento completo de código-fonte. O Mezuro busca auxiliar em vários problemas relacionados à utilização de métrica, visando ser uma interface que permita, de forma flexível, a extração e análise de métricas estáticas de código-fonte, licensiado como AGPLv3[2] (MANZO et al., 2014). O Mezuro é uma plataforma concebida através do amadurecimento de diversas ferramentas, inicializada através do projeto Qualipso[3]. Dentre estas ferramentas, destaca-se o Analizo[4], uma das ferramentas utilizadas pelo Mezuro para extração de métricas de código-fonte em C/C++ e Java.

A primeira versão do Mezuro foi criada a partir da plataforma web chamada Noosfero[5]. O Noosfero é uma plataforma de criação de redes sociais livre, disponível sob licença AGPL[6] V3, que facilita a criação de redes sociais personalizadas e geração de conteúdo

[1] <http://mezuro.org/>
[2] <http://www.gnu.org/licenses/agpl-3.0.html>
[3] Quality Platform for Open Source: <http://qualipso.icmc.usp.br/>
[4] <http://analizo.org/>
[5] <http://noosfero.org/>
[6] Licença de software GNU Affero General Public License

colaborativo. O Participa BR [7], o Stoa[8] e o Portal da FGA[9] são exemplos de portais que utilizam o Noosfero.

O Noosfero é construído em Ruby, implementando a arquitetura MVC através do *framework* Rails. A arquitetura do Noosfero ainda permite a implementação de novas funcionalidades através da criação de *plugins* que podem ser habilitados no ambiente instanciado. Assim, na primeira versão do Mezuro, foi implementado um *plugin* que adicionava as funcionalidades específicas para o monitoramento de código-fonte e realizava a conexão com os outros componentes necessários (explicados a seguir na Seção C.0.1).

Entretanto, o Mezuro tem sido evoluído como uma plataforma independente. Isto ocorreu devido à algumas dificuldades existentes inerentes ao acoplamento com o Noosfero, principalmente relacionados às versões das tecnologias ainda utilizadas pelo Noosfero. Além disso, atualmente o Mezuro já possui uma pequena comunidade de colaboradores o que favorece o desacoplamento do código do Mezuro, uma vez que os *plugins* do Noosfero são mantidos junto com o código principal. Com isto, o Mezuro está sendo desenvolvido atualmente com as versões estáveis do Ruby 2[10] e Rails 4[11]. A Figura 31 apresenta a tela principal da nova versão do Mezuro.

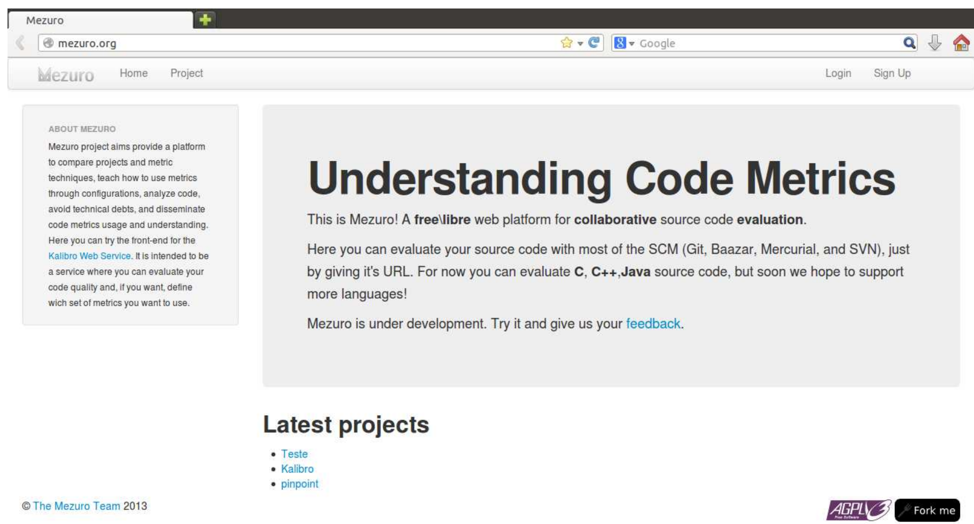

Figura 31 – Tela principal do Mezuro. Disponível em <http://mezuro.org/>

Listamos a seguir duas plataformas semelhantes ao Mezuro, ferramentas identificadas e detalhadas através de outros trabalhos (MEIRELLES et al., 2010)(MENESES; MEIRELLES, 2013)(MANZO et al., 2014):

7 <https://www.participa.br/>
8 <http://stoa.usp.br/>
9 <http://fga.unb.br/>
10 <https://www.ruby-lang.org/>
11 <http://rubyonrails.org/>

- **SonarQube**[12] - Plataforma livre de gerenciamento de qualidade de código que classifica problemas encontrados e calcula métricas simples relacionados a testes e dívidas técnicas.

- **CodeClimate**[13] - Ferramenta que procura identificar *code smells* no software em análise e classificá-los a partir de notas que varia de A a F. Esta ferramenta fornece análise sobre códigos JavaScript e Ruby.

Destacamos as principais características do Mezuro, que também serão discutidas durante a apresentação da arquitetura da ferramenta:

- Coletar dados a partir de diversos extratores, possibilitando a escolha rica de diversas métricas. Além disso, o Mezuro é extensível para a inserção de extratores diferentes.

- Criação de configurações que são um conjunto de métricas e parâmetros que podem ser utilizadas para a avaliação de um projeto. Esta característica permite que especialistas definam os parâmetros e métricas adequadas para um determinado contexto, sendo que uma configuração também pode ser aplicado em outros projetos a depender da necessidade de quem utilizará a ferramenta.

- Criação de intervalos qualitativos associado a valores de métricas. Esta característica é muito importante para a utilização das métricas, uma vez que abstraem a interpretação direta dos valores obtidos para definições mais simples como bom, regular e ruim.

- Criação de métricas mais complexas a partir da combinação de métricas nativas, flexibilizando e estendendo a utilização da ferramenta.

- Monitoramento de projetos a partir de repositórios ou arquivos compactados. As opções de repositórios possíveis incluem o GIT, Subversion e o Bazaar. Vale ressaltar que os resultados dos monitoramentos são públicos a acessíveis à comunidade.

- Monitoramento de projetos com periodicidade definido pelo usuário.

- Escolha de qual configuração um determinado projeto irá utilizar.

C.0.1 Arquitetura do Mezuro

Como mencionado, o Mezuro está evoluindo para uma aplicação independente. Nesse sentido, a arquitetura da plataforma como um todo está sendo modificada, visando principalmente maior modularização em diversos serviços independentes. A Figura 32 apresenta a arquitetura atual da plataforma.

12 <http://www.sonarqube.org/>
13 <https://codeclimate.com/>

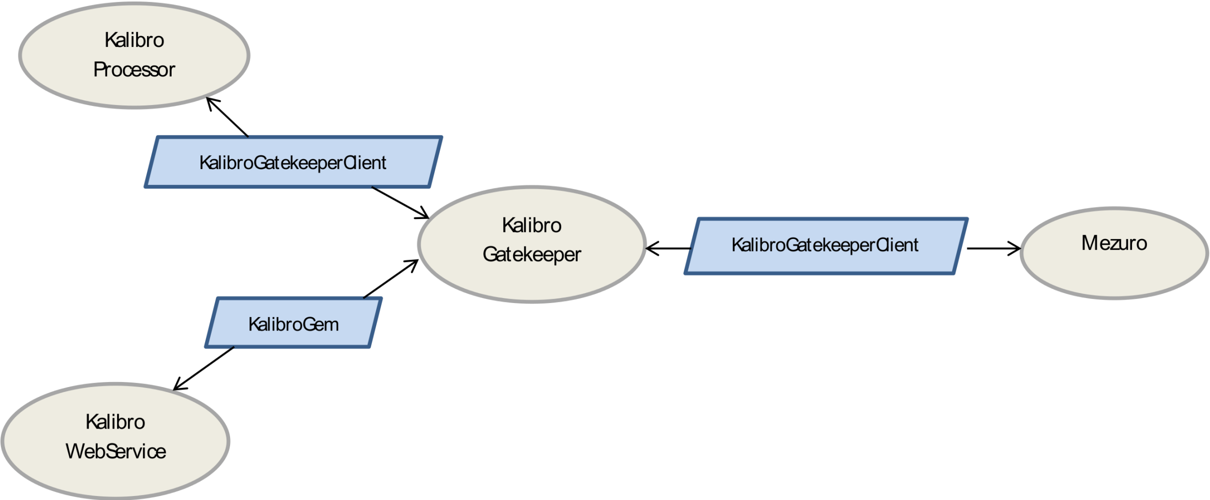

Figura 32 – Arquitetura Atual do Mezuro. Extraído de (MANZO et al., 2014)

O Mezuro utiliza o WebService Kalibro Metrics [14] para fornecer a funcionalidade de análise e avaliação de métricas de código-fonte. O Kalibro é um WebService SOAP[15], baseado em mensagens XML[16] cujos componentes correspondem Kalibro WebService. O principal objetivo do Kalibro é se conectar com ferramentas extratoras de métricas, como o Analizo, executando-as para realizar a coleta de dados. Outro módulo é representado pela elipse Kalibro Processor cujo objetivo é centralizar o processamento de métricas. Por último, a comunicação entre todos esses módulos principais é intermediada pelo módulo correspondente à elipse Kalibro Gatekeeper, conforme apresentado na Figura 32

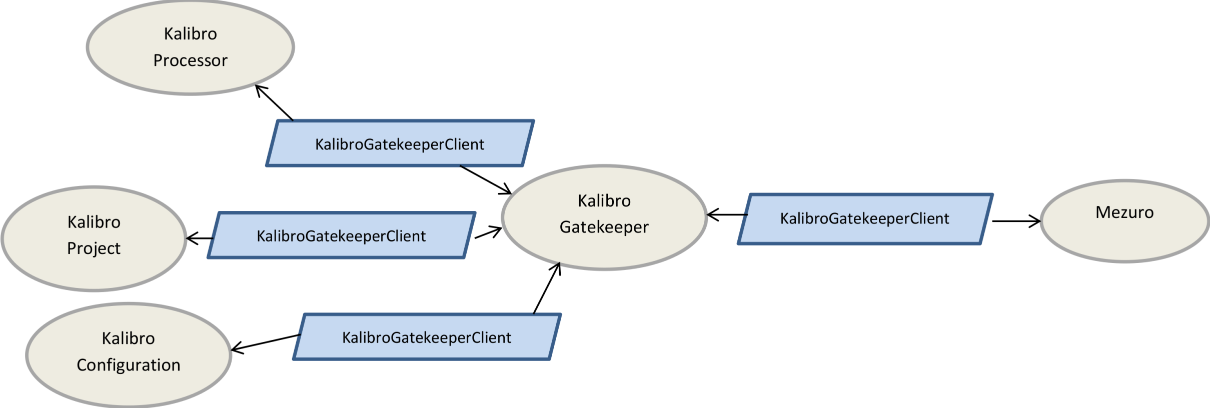

Figura 33 – Arquitetura Futura do Mezuro. Extraído de (MANZO et al., 2014)

Dentre os objetivos da evolução desta arquitetura, tem-se a reescrita do Kalibro e sua modularização em três serviços principais, que pode ser observada na Figura 33.

[14] <http://kalibro.org>
[15] Simple Object Access Protocol
[16] Linguagem de Marcação Extensível

O primeiro módulo já está em uso, sendo também represetado na Figura 32, é o módulo Kalibro Processor. Assim, a contemplação desta nova arquitetura consiste na reescrita de serviços do Kalibro, de Java para Ruby, para concepção de dois novos módulos: Kalibro Project e Kalibro Configuration, responsáveis pelo processamento de projetos e configuração respectivamente. Essa nova proposta apoia a modularização e o baixo acoplamento, utilizando-se principalmente o conceito de orquestração de serviços que podem ser encontrados em Arquiteturas Orientadas à Serviço - SOA (ERL, 2007). Esta nova arquitetura proporcionará que novos extratores possam ser mais facilmente acoplados à plataforma, além de proporcionar que as funcionalidades e processamento dos projetos e configurações evoluam independentemente.

Os objetivos tecnológicos deste trabalho estão diretamente relacionados com os objetivos de evolução arquitetural da comunidade do Mezuro. Portanto, já iniciamos a contribuição com esta evolução, buscando viabilizar as melhorias necessárias para que o Mezuro contemple os objetivos desde trabalho. Dentre as futuras contribuições, pretendemos trabalhar principalmente sobre as funcionalidades de configuração de métricas do Mezuro, pois desejamos melhorar os mecanismos de configuração e visualização de métricas, viabilizando a utilização da estrutura de Cenários de Decisões que será apresentada no próximo Capítulo.

O Mezuro já possui as métricas de *design* listadas no Capítulo 3. Atualmente, métricas de vulnerabilidades específicas tais quais as mencionadas no Capítulo 3 já estão sendo coletadas pelo Analizo. Portanto, nossas contribuições também serão direcionadas para que estas métricas sejam tratadas e processadas na plataforma Mezuro.

Com estas contribuições, o Mezuro será avaliado como proposta de plataforma de monitoramento de código-fonte para utilização de métricas na tomada de decisão de projetos de software.

Referências

AFEK, J.; SHARABANI, A. *Dangling Pointer - Smashing the Pointer for Fun and Profit*. 2007. A whitepaper from Watchfire. Citado na página 35.

AGGARWAL, K. K.; SINGH, Y.; CHHABRA, J. K. An integrated measure of software manitainability. In: ICIT. *Proceedings Annual Reliability and Security Symposium*. [S.l.], 2002. p. 235–241. Citado na página 33.

ALMEIDA, E. S. et al. *C.R.U.I.S.E - Component Reuse in Software Engineering*. [S.l.]: CESAR, 2007. Citado na página 31.

ALMEIDA, L. T.; MIRANDA, J. M. *Código Limpo e seu Mapeamento para Métricas de Código Fonte*. 2010. Citado 3 vezes nas páginas 44, 53 e 115.

ALSHAMMARI, B.; FIDGE, C.; CORNEY, D. Security metrics for object-oriented class designs. In: IEEE. *Ninth International Conference on Quality Software*. Jeju - Coréia, 2009. Citado na página 40.

ARANHA, D. F. et al. *Vulnerabilidades no software da urna eletrônica brasileira*. [S.l.], 2012. Citado na página 48.

BAIG, I. *Measuring Cohesion and Coupling of Object-Oriented Systems - Derivation and Mutual Study of Cohesion and Coupling*. 2004. Citado na página 33.

BALDWIN, C. Y.; CLARK, K. B. *Design Rules: The Power of Modularity*. [S.l.]: The MIT Press, 2000. Citado na página 32.

BALLARD, C. et al. *Data Modeling Techniques for Data Warehousing*. 1nd. ed. [S.l.: s.n.], 1998. Citado 2 vezes nas páginas 125 e 126.

BARBOSA, G. M. G. *Arquitetura de Software*. [S.l.: s.n.], 2009. Citado na página 32.

BASILI, V. R.; HUTCHENS, D. H. An empirical study of a syntactic complexity family. In: IEEE. *IEEE Transaction on Software Engineering*. [S.l.], 1983. v. 9, p. 664–672. Citado na página 30.

BECK, F.; DIEHL, S. On the congruence of modularity and code coupling. In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. New York - USA: [s.n.], 2011. p. 354–364. Citado na página 32.

BECK, K. *Test Driven Development: By Example*. [S.l.]: Addison-Wesley, 2002. Citado na página 23.

BECK, K. *Implementation Pattens*. [S.l.]: Addison-Wesley Longman Publishing Co., 2007. Citado 2 vezes nas páginas 24 e 115.

BECK, K.; ANDRES, C. *eXtreme Programming eXplained: Embrace Change*. [S.l.]: Addison-Wesley Longman Publishing Co., 2000. Citado na página 23.

BERANDER, P. et al. *Software quality attributes and trade-offs*. Tese (Doutorado) — Blekinge Institute of Technology, 2005. Citado na página 23.

BISHOP, M. *Computer Security: Art and Science*. [S.l.]: Addison-Wesley, 2003. Citado 2 vezes nas páginas 40 e 41.

BLACK, P. Static analyzers in software engineering. In: *CrossTalk, The Journal of Defense Software Engineering*. [S.l.: s.n.], 2009. p. 16–17. Citado na página 44.

BROOKS, F. P. No silver bullet - essence and accident in software engineering. In: *Proceedings of the IFIP Tenth World Computing Conference*. [S.l.: s.n.], 1986. p. 1069–1076. Citado na página 118.

CASTELLANOS, M. et al. ibom: A platform for intelligent business operation management. In: *Proceedings of the 21st International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 2005. (ICDE '05), p. 1084–1095. ISBN 0-7695-2285-8. Disponível em: <http://dx.doi.org/10.1109/ICDE.2005.73>. Citado na página 25.

CAVALCANTI, T. R. *Suporte a Decisão - 02 - Sobre as operações de OLAP*. 2012. Disponível em: <http://www.itnerante.com.br/profiles/blogs/artigo-suporte-a-decis-o-02-sobre-as-opera-es-de-olap>. Acesso em: 25 de Abril de 2014. Citado na página 130.

CHESS, B.; WEST, J. *Secure Programming with Static Analysis*. [S.l.]: Addison-Wesley, 2007. ISBN 0-321-42477-8. Citado na página 48.

CHIDAMBER, S. R.; KEREMER, C. F. A metrics suite for object oriented design. In: *IEEE Transactions on Software Engineering*. [S.l.: s.n.], 1994. v. 20, p. 476–493. Citado na página 25.

CHRISTEY, S. Plover - preliminary list of vulnerability examples for researchers. In: . [S.l.: s.n.], 2006. Citado na página 37.

DAVIS, N. et al. Processes for producing secure software: Summary of us national cybersecurity summit subgroup report. In: IEEE. *IEEE Security & Privacy*. [S.l.], 2004. p. 18–25. Citado na página 34.

DEMEYER, S.; DUCASSE, S.; O, N. *Object-Oriented Reengineering Patterns*. [S.l.]: Morgan Kaufmann, 2002. Citado na página 31.

DOUGHERTY, C. *Practical Identification of SQL Injection Vulnerabilities*. 2012. US-CERT - United States Computer Emergency Readiness Team. Citado na página 36.

DUARTE, L. O.; BARBATO, L. G. C.; MONTES, A. Vulnerabilidades de software e formas de minimizar suas explorações. 2005. Citado na página 24.

ELMASRI, R.; NAVATHE, S. *Sistemas de Banco de Dados*. 4nd. ed. [S.l.]: Pearson Education do Brasil Ltda, 2006. Citado na página 124.

ERL, T. *SOA: Principles of Service Design*. [S.l.]: Prentice Hall, 2007. Citado na página 139.

FAYAD, M. E.; SCHMIDT, D. C. Object-oriented application frameworks. In: *Communications of the ACM*. [S.l.: s.n.], 1997. v. 40, p. 32–38. Citado na página 113.

FENTON, N. E.; PFLEEGER, S. L. *Software Metrics: A Rigorous and Practical Approach*. [S.l.]: Course Technology, 1998. Citado na página 44.

FOLLECO, A. et al. Learning from software quality data with class imbalance and noise. In: *Proceedings of the Nineteenth International Conference on Software Engineering & Knowledge Engineering (SEKE 2007), Boston, Massachusetts, USA, July 9-11, 2007*. [S.l.]: Knowledge Systems Institute Graduate School, 2007. p. 487. ISBN 1-891706-20-9. Citado 3 vezes nas páginas 25, 83 e 121.

FOWLER, M. et al. *Refactoring - Improving the Design of Existing Code*. [S.l.]: Addison-Wesley, 1999. Citado 3 vezes nas páginas 23, 113 e 114.

GAMMA, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. [S.l.]: Addison-Wesley, 1994. Citado na página 113.

GANDHI, S. H. et al. Security metric for object oriented class design - result analysis. In: IJITEE. *International Journal of Inoovative Technology and Exploring Engineering*. [S.l.], 2013. v. 2. Citado na página 33.

GUIMARAES, S. R. G. *Modelagem multidimensional I*. 2012. Disponível em: <http://sandrorgguimaraes.blogspot.com.br/2012/04/modelagem-multidimensional.html>. Acesso em: 2 de Junho de 2014. Citado 2 vezes nas páginas 13 e 126.

HALLORAN, T. J.; SCHERLIS, W. L. High quality and open source software practices. In: *Paper presented at the Second Workshop on Open-Source Software Engineering*. [S.l.: s.n.], 2002. Citado na página 23.

HEGEDüS, P. et al. Myth or reality? analyzing the effect of design patterns on software maintainability. In: *Computer Applications for Software Engineering, Disaster Recovery, and Business Continuity Communications in Computer and Information Science*. [S.l.: s.n.], 2012. v. 340, p. 138–145. Citado na página 113.

HENRY, S.; KAFURA, D. The evaluation of software systems' structure using quantitative software metrics. In: *Software Practice and Experience*. [S.l.: s.n.], 1984. p. 561–573. Citado na página 43.

HOWARD, M. A process for performing security code reviews. In: IEEE. *IEEE Security & Privacyg*. [S.l.], 2006. p. 74–79. Citado na página 117.

IEEE. *610.12-1990 - IEEE Standard Glossary of Software Engineering Terminology*. [S.l.], 1990. Citado na página 32.

INMON, W. H. *Building the Data Warehouse*. 3nd. ed. New York, NY, USA: Jhon Wiley & Sons, Inc, 2002. Citado 4 vezes nas páginas 83, 121, 122 e 124.

ISO/IEC 9126 - International Standard. Information Technology - Software Product Quality. [S.l.], 1998. Citado na página 113.

JIMENEZ, W.; MAMMAR, A.; CAVALLI, A. R. Software vulnerabilities, prevention and detection methods: A review. In: *First International Workshop on Security in Model Driven Architecture*. Enschede - Holanda: [s.n.], 2009. Citado na página 34.

KERIEVSKY, J. *Refatoração para Padrões*. [S.l.]: Bookman, 2008. Citado na página 113.

KERNIGHAN, B. W.; PLAUGER, P. J. *The Elements of Programming Style*. [S.l.]: McGraw-Hill Book Company, 1978. Citado na página 112.

KHAN, S. A.; KHAN, R. A. Securing object oriented design: A complexity perspective. In: *International Journal of Computer Applications (0975 – 8887)*. [S.l.: s.n.], 2010. v. 8. Citado na página 118.

KIMBALL, R.; ROSS, M. *The Data Warehouse Toolkit - The Complete Guide to Dimensional Modeling*. 2nd. ed. New York, NY, USA: Jhon Wiley & Sons, Inc, 2002. Citado 10 vezes nas páginas 13, 14, 123, 124, 125, 126, 127, 128, 130 e 132.

KIMPEL, J. F. Critical success factors for data warehousing: A classic answer to a modern question. In: *Issues in Information Systems*. [S.l.: s.n.], 2013. v. 14, p. 376–384. Citado na página 130.

KRSUL, I. V. *Software Vulnerability Analysis*. Tese (Doutorado) — Purdue University, West Lafayette, 1998. Citado na página 34.

LAB, K.; INTERNATIONAL, B. *Global Corporate IT Security Risks: 2013*. 2013. Research. Citado na página 33.

LAKOS, J. *Large-scale C++ software design*. [S.l.]: Addison-Wesley, 1996. Citado na página 31.

LARMAN, C. *Utilizando UML e Padrões*. [S.l.]: Bookman, 2007. Citado 3 vezes nas páginas 59, 60 e 111.

LHEE, K.-S.; CHAPIN, S. J. Buffer overflow and format string overflow vulnerabilities. In: *SP&E - Software, Practice and Experience*. Syracuse - Nova York - Estados Unidos: [s.n.], 2002. Citado na página 36.

LI, H. F.; CHEUNG, W. K. An empirical study of software metrics. In: *IEEE Transactions Software Engineering*. [S.l.: s.n.], 1987. p. 697–708. Citado na página 45.

LIEBERHERR, K. J. *Lieberherr, Adaptative Object Oriented Software: The Demeter Method*. [S.l.]: PWS Publishing, 1996. Citado na página 31.

LOPES, M. C.; OLIVEIRA, P. A. de. Ferramenta de construção de data warehouse. 2007. Citado na página 25.

MALERBA, C. *Vulnerabilidades e Exploits: técnicas, detecção e prevenção*. Monografia (Graduação) — Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2010. Citado na página 37.

MANSOOR, U. et al. ode-smells detection using good and bad software design examples. In: *Computational Optimization and Innovation - COIN*. [S.l.: s.n.], 2014. Citado 2 vezes nas páginas 113 e 114.

MANZO, R. R. et al. Mezuro - coleta, interpretacão e exibição automatizadas de métricas estáticas de código-fonte. Enviado para o Congresso Brasileiro de Software 2014. 2014. Citado 5 vezes nas páginas 14, 74, 135, 136 e 138.

MARTENSSON, F.; GRAHN, H.; MATTSSON, M. Forming consensus on testability in software developing organizations. In: *Fifth Conference on Software Engineering Research and Practice in Sweden*. Västerås - Sweden: [s.n.], 2005. p. 31–38. Citado na página 31.

MARTIN, R.; CHRISTEY, S.; BAKER, D. *A Progress Report on the CVE Initiative*. 2002. Disponível em: <http://cve.mitre.org/docs/docs-2002/prog-rpt_06-02/>. Citado na página 24.

MARTIN, R. A. *The Vulnerabilities of Developing on the Net*. 2001. Disponível em: <http://cve.mitre.org/docs/docs-2001/DevelopingOnNet.html>. Acesso em: 08 maio 2007. Citado na página 36.

MARTIN, R. C. *Agile Software Development: Principles, Patterns and Practices*. [S.l.]: Prentice Hall, 2002. Citado na página 31.

MARTIN, R. C. *Clean Code - A Handbook of Agile Software Craftsmanship*. [S.l.]: Prentice Hall, 2008. Citado 2 vezes nas páginas 43 e 115.

MAZUCO, G. M. *Uma Abordagem de Data Warehouse para Gestão de Métricas de Software com Análise e Valor Agregado*. Monografia (Graduação) — Faculdade de Informática, Universidade Católica do Rio Grande do Sul, Porto Alegre, 2011. Citado 3 vezes nas páginas 25, 83 e 121.

MEIRELLES, P. et al. *Mezuro: A Source Code Tracking Platform*. Tese (Doutorado) — FLOSS Competence Center – University of São Paulo, Instituto de Matemática e Estatística, Maio 2010. Citado 2 vezes nas páginas 135 e 136.

MEIRELLES, P. R. M. *Monitoramento de métricas de código-fonte em projetos de software livre*. Tese (Doutorado) — Instituto de Matemática e Estátistica – Universidade de São Paulo (IME/USP), 2013. Citado 5 vezes nas páginas 43, 44, 45, 55 e 57.

MEIRELLES, P. R. M. et al. A study of the relationships between source code metrics and attractiveness in free software projects. In: SBC. *Simpósio Brasileiro de Engenharia de Software - SBES*. [S.l.], 2010. Citado na página 29.

MENESES, V. V.; MEIRELLES, P. R. M. *Evolução Plataforma Mezuro: De Plugin a Aplicação Independente*. [S.l.], 2013. Citado na página 136.

MICHLMAYR, M.; HILL, B. M. Quality and the reliance on individuals in free software projects. In: *Paper presented at the Third Workshop on Open-Source Software Engineering*. [S.l.: s.n.], 2003. Citado na página 23.

PáSCOA, J. E. P. *Fatores e Subfatores para Avaliação da Segurança em Software de Sistemas Críticos*. 2002. Citado na página 33.

RAKIC, G.; BUDIMAC, Z. Problems in systematic application of software metrics and possible solution. In: ICIT. *5th International Conference on Information Technology*. [S.l.], 2011. Citado na página 44.

RêGO, G. B. Monitoramento dea métricas de código-fonte com suporte de um ambiente de data warehousing: um estudo de caso em uma autarquia da administração pública federal. 2014. Disponível em: <http://bdm.unb.br/handle/10483/8069>. Citado 4 vezes nas páginas 25, 83, 86 e 87.

SALTZER, J. H.; SCHROEDER, M. D. Basic principles of information protection. In: IEEE. *Proceedings of the IEEE*. [S.l.], 1975. Citado na página 61.

SALTZER, J. H.; SCHROEDER, M. D. The protection of information in operating systems. In: IEEE. *Proceedings of the IEEE*. [S.l.], 1975. p. 1278–1308. Citado na página 40.

SAMETINGER, J. *Software Engineering with Reusable Components*. [S.l.]: Springer-Verlag, 1997. Citado na página 31.

SCHMIDT, D. C.; PORTER, A. Leveraging open-source communities to improve the quality performance of open-source software. In: *Paper presented at the First Workshop on Open-Source Software Engineering*. [S.l.: s.n.], 2001. Citado na página 23.

SEN, A. A model of data warehousing process maturity. In: *Software Engineering, IEEE Transactions on*. [S.l.: s.n.], 2011. v. 38, p. 336–353. Citado na página 130.

SILVEIRA, P. S.; BECKER, K.; RUIZ, D. D. Spdw+: a seamless approach for capturing quality metrics in software development environments. *Software Quality Control*, Kluwer Academic Publishers, Hingham, MA, USA, v. 18, n. 2, p. 227–268, jun. 2010. ISSN 0963-9314. Disponível em: <http://dx.doi.org/10.1007/s11219-009-9092-9>. Citado 3 vezes nas páginas 25, 83 e 121.

SOCIETY, I. C.; COMMITTEE, S. C. *IEEE Standard Computer Dictionary: Compilation of IEEE Standard Computer Glossaries*. [S.l.]: IEEE, 1991. Citado na página 30.

SPINELLIS, D. *Code Quality: The Open Source Perspective*. [S.l.]: Addison-Wesley Professional, 2006. ISBN 0-321-16607-8. Citado na página 30.

SÿSTA, T. *Static and Dynamic Reverse Engineering Techniques for Java Software Systems*. Tese (Doutorado) — University of Tampere, Department of Computer and Information Science. Finland, Maio 2000. Citado na página 43.

TROY, D. A.; ZWEBEN, S. H. Measuring the quality of structured designs. In: *J. Systems and Software*. [S.l.: s.n.], 1981. p. 113–120. Citado na página 43.

TSIPENYUK, K.; CHESS, B.; MCGRAW, G. Seven pernicious kingdoms: A taxonomy of software security errors. November/December 2005, p. 81–84, 2005. Citado 2 vezes nas páginas 24 e 39.

TUTORIALSPOINT. *Data Warehousing - OLAP*. 2014. Disponível em: <http://www.tutorialspoint.com/dwh/dwh_olap.htm>. Acesso em: 20 de Junho de 2014. Citado 4 vezes nas páginas 14, 129, 130 e 131.

VIEGA, J.; MCGRAW, G. *Building Secure Software: How To Avoid Security Problems The Right Way*. [S.l.]: Addison-Wesley, 2002. Citado 2 vezes nas páginas 40 e 41.

VRIES, S. d. Security testing web applications throughout automated software tests. In: OWASP. *The Open Web Application Security Projects Europe Conference*. Leuven - Bélgica, 2006. Citado na página 35.

WAGNER, C. *Data Warehouse (DW)*. 2012. Disponível em: <http://cacau-indicou.blogspot.com.br/2012/02/data-warehouse-dw.html>. Acesso em: 2 de Junho de 2014. Citado 4 vezes nas páginas 13, 14, 127 e 128.

WESTFALL, L. *12 Steps to Useful Software Metrics*. 2005. Citado na página 53.

YAU, S. S.; COLLOFELLO, J. S. Design stability measures for software maintenance. In: IEEE. *IEEE Transactions Software Engineering*. [S.l.], 1985. p. 849–856. Citado na página 43.