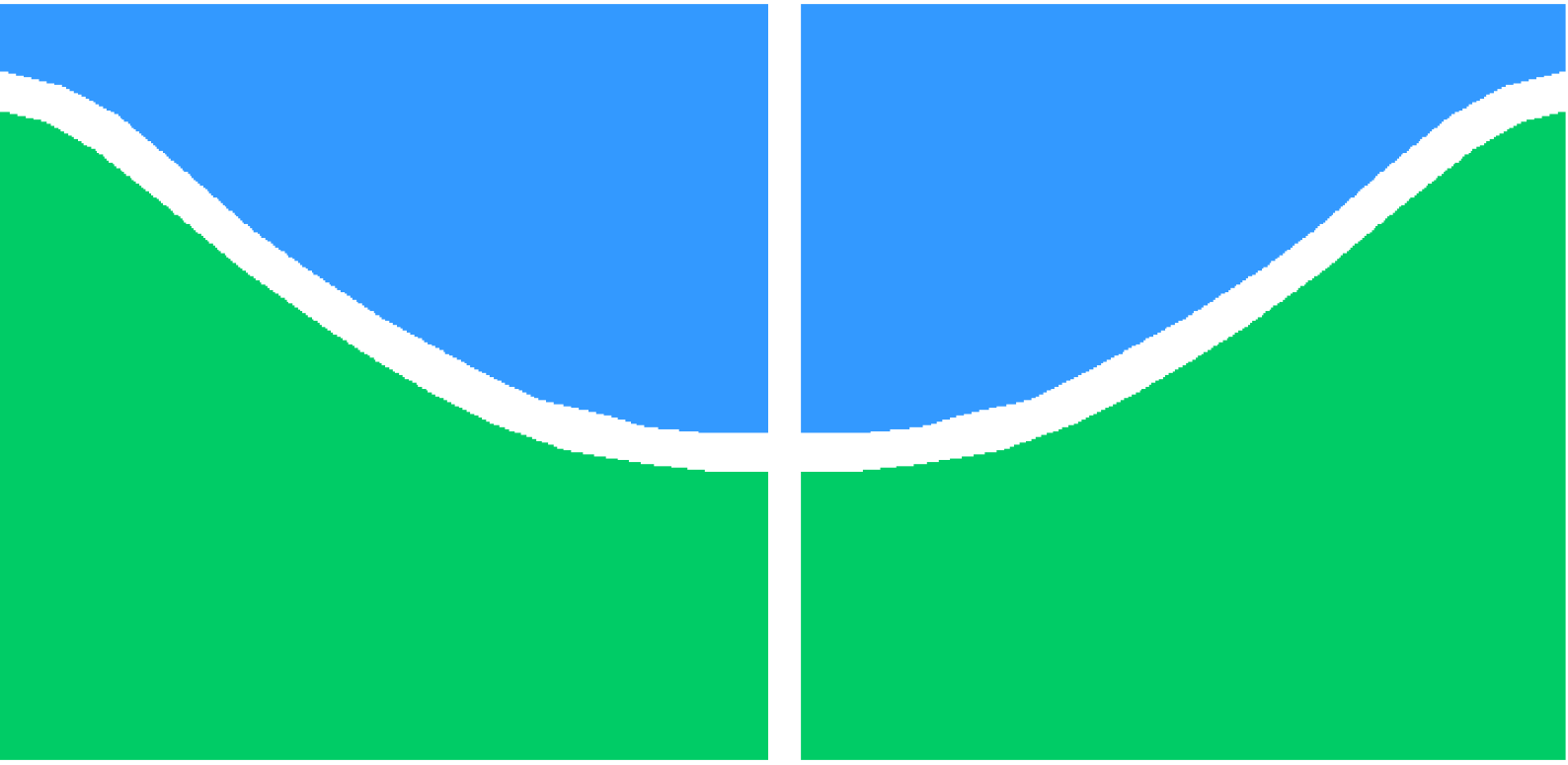

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Engenharia de Software

Tomada de decisões orientadas a métricas de software: observações de métricas de produto e vulnerabilidades de software via DW e Plataforma de monitoramente de código-fonte

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

Tomada de decisões orientadas a métricas de software: observações de métricas de produto e vulnerabilidades de software via DW e Plataforma de monitoramente de código-fonte

Monografia submetida ao curso de graduação em Engenharia de Softwareda 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

Tomada de decisões orientadas a métricas de software: observações de métricas de produto e vulnerabilidades de software via DW e Plataforma de monitoramente de código-fonte/ Arthur de Moura Del Esposte & Carlos Filipe Lima Bezerra. – Brasília, DF, 2014-

100 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. Tomada de decisões orientadas a métricas de software: observações de métricas de produto e vulnerabilidades de software via DW e Plataforma de monitoramente de código-fonte

CDU 02:141:005.6

Arthur de Moura Del Esposte
Carlos Filipe Lima Bezerra

Tomada de decisões orientadas a métricas de software: observações de métricas de produto e vulnerabilidades de software via DW e Plataforma de monitoramente de código-fonte

Monografia submetida ao curso de graduação em Engenharia de Softwareda 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 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. Agradeço também pelo imensurável apoio e inspiração em tudo que fiz. Ao meu irmão e amigo Heitor de Moura Del Esposte, fonte de admiração e inspiração, sou eternamente grato pelo companherismo, amizade, amor, respeito e tudo mais que sua presença agrega

em minha vida. Dedico este trabalho aos três.

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 demais amigos e familiares pelo apoio, amizade e por participarem ativamente da minha vida pessoal.

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 o principal fator 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 principais responsáveis por este 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 de conclusão de curso aborda as principais ideias e conceitos relacionados à melhoria contínua do código-fonte. Neste 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 a nível técnico quanto gerencial. Neste 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 evolução da plataforma livre de monitoramento de código-fonte chamada Mezuro e a construção de uma solução de DataWarehousing. Por fim, serão definidos protocolos para realização de estudos de casos que permitam avaliar estas ferramentas e compreender a correlação entre a qualidade interna e vulnerabilidades de software, através de métricas.

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

Abstract

The internal quality is the key 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, are more understandable, reduce the risk of bugs and make the code changes and developments easier to be done. Therefore, the Software Engineer is a major contributor for this success since he must gather a set of skills and knowledge in order to apply practices, techniques and tools for creating secure and well design software. Thus, this completing of course work covers the main ideas and concepts related to continuous improvement of source code. In this sense, in this monograph is highlighted the importance of conducting ongoing activities related to design and security throughout the software project, and discuss the importance of using static source code metrics to support decision making at managerial level and technical as well. In this sense, we present the concept of Decisions Scenarios that define an abstraction for metrics' choice and interpretation, as well as proposals of examples in order to use scenarios for measuring software security. To support the use of scenarios and metrics in software development, this work also includes the evolution of a source code monitoring free software called Mezuro and building a datawarehousing solution. Finally, will be defined protocols for conducting case studies to evaluate these tools and understand the correlation between internal quality and software vulnerabilities through metrics.

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

Lista de ilustrações

Figura 1 – Práticas do Design Ágil 34
Figura 2 – Tela principal do Mezuro. Disponível em <http://mezuro.org/> 64
Figura 3 – Arquitetura Atual do Mezuro. Extraído de (MANZO et al., 2014) . . . 66
Figura 4 – Arquitetura Futura do Mezuro. Extraído de (MANZO et al., 2014) . . 67
Figura 5 – Componentes de um DWing (KIMBALL; ROSS, 2002) 70
Figura 6 – Exemplo de visualização de dados através de dashboards 72
Figura 7 – Exemplo de cubo de dados 73
Figura 8 – Esquema estrela 74
Figura 9 – Esquema floco de neve 75
Figura 10 – Operações OLAP 77
Figura 11 – Ciclo de vida de um Projeto de DWing (KIMBALL; ROSS, 2002) . . . 78

Lista de tabelas

Tabela 1 – Avaliação de Indicadores da Métrica AV 60
Tabela 2 – Avaliação de Indicadores da Métrica AC 60
Tabela 3 – Avaliação de Indicadores da Métrica C 61
Tabela 4 – Avaliação de Indicadores da Métrica I 61
Tabela 5 – Avaliação de Indicadores da Métrica A 62
Tabela 6 – Parte I - Resumo de todos os cenários propostos para o monitoramento da segurança de software 89
Tabela 7 – Parte II - Resumo de todos os cenários propostos para o monitoramento da segurança de software 90
Tabela 8 – Cronograma para o TCC 2 93

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 21
1.1 Objetivos 23
1.1.1 Questões de Pesquisa 24
1.2 Metodologia e Pesquisa 25
1.3 Organização do Trabalho 26

2 DESIGN, SEGURANÇA E MÉTRICAS DE SOFTWARE 29
2.1 Design de Software 29
2.1.1 O Design, Princípios e Práticas 30
2.1.2 *Code Smells* - Cheiros de Código 35
2.1.3 Código Limpo 37
2.2 Segurança de Software 38
2.2.1 Classificações e taxonomias de vulnerabilidades 42
2.2.2 Princípios de segurança 45
2.3 Métricas em Engenharia de Software 50
2.3.1 Métricas Estáticas de Design de Software 52
2.3.2 Métricas Estáticas de Segurança 55
2.3.2.1 CVSS 58

3 MEZURO: UMA PLATAFORMA DE MONITORAMENTO DE CÓDIGO FONTE 63
3.1 Arquitetura do Mezuro 66

4 *DATA WAREHOUSE* 69
4.1 *Data Warehousing* 69
4.1.1 Sistemas de Fonte de Dados Operacionais - OSS 70
4.1.2 Área de Preparação dos Dados - DSA 70
4.1.3 Área de Apresentação - DPA 71
4.1.4 Ferramentas de Acesso de dados – Visualização de dados 72
4.2 Modelage dimensional 72
4.3 OLAP 75
4.4 Ciclo de vida de um ambiente de *Data Warehousing* 76

5 CENÁRIOS DE DECISÕES 79
5.1 Definição de Cenários de Decisão para Vulnerabilidades de Software 82
5.1.0.1 Cenários de Decisão para Caracterização da Qualidade do Código 83

5.1.0.2 Cenários de Decisão para Caracterização de Vulnerabilidades Específicas de Código . . . 86

6 CONSIDERAÇÕES FINAIS 91
6.1 Evolução do Trabalho 92

Referências 95

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. Estas 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 preveêm 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. 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 proveem a segurança necessária para realização de mudanças assim como o feedback automatizado da manutenção do software. A Programação em Pares, dentre outras técnicas, também possui papel fundamental na 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 esta 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 a qualidade de código está a sua segurança(TSIPENYUK; CHESS; MCGRAW, 2005). A segurança de software está relacionada a confiabilidade, disponibilidade e integridade dos diversos componentes que compõe o software. Dados do ICAT/NIST 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.

A medida que o tempo vai passando, novas vunerabilidades vão sendo descobertas pela comunidade. O projeto CVE, que tem como objetivo enumerar vunerabilidades de software existentes, tinha uma lista de 321 vunerabilidades diferentes no ano de sua concepção, 1999 (MARTIN; CHRISTEY; BAKER, 2002). No ano de 2002, a lista já havia aumentado para 2032 vunerabilidades e atualmente o número já chega a 61 mil vunerabilidades 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 vunerabilidades e meios de detectá-las são habilidades necessárias para garantia de software seguro. Este 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 código-fonte para auxiliar as tomadas de decisões, ou até mesmo para avaliação do código inserido ou da aplicação de refatorações.

Porém, a grande quantidade de métricas, coletas manuais e poucos recursos de visualização são fatores que acabam por desmotivar o uso destas para o monitoramento do código. Além disso, a compreenção 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 importância 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 faciltem a interpretação dos resultados obitidos. Alternativamente, o ambiente de *Data Warehousing* (DWing) é uma solução que tem se destacado no ramo de BI e tem enfase 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 têm utilizados o DWing no contexto de monitoramento de métricas apresentando bons resultados (CASTELLANOS et al., 2005) (FOLLECO et al., 2007) (SILVEIRA; BECKER; RUIZ, 2010)(MAZUCO, 2011).

Portanto, neste trabalho serão 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 vunerabilidades e qualidade de software. Assim, espera-se identificar as oportunidades de utilização de métricas na melhoria contínua do processo e, consequentemente, na qualidade interna do produto a partir do estabelecimento de cenários, compostos a partir da análise de correlação de métricas, que evidenciem as boas e más características do *design* de um projeto que impactam na vunerabilidade do sistema com o objetivo de facilitar a interpretação e evitar possíveis equívocos que são baseados em análises errôneas sobre métricas isoladas, sobre correlações inexistentes ou até mesmo a escolha de métricas inadequadas cujos problemas são discutidos em (CHIDAMBER; KEREMER, 1994). Para auxiliar no monitoramento e na tomada de decisão, será explorado o uso de plataforma de monitoramento de código-fonte e um ambiente de DWing e será observado o uso dessas duas soluções e suas contribuições para a melhoria do processo e qualidade do produto.

1.1 Objetivos

O objetivo deste trabalho consite no estudo teórico sobre conceitos relacionados a métricas de monitoramento de código-fonte buscando relacionar características de bom *design*, como Código Limpo e Bad Smells no contexto de vunerabilidades de software para o estabelecimento de cenários que caracterizam determinado componente do sistema e auxiliem na tomada de decisão no que diz respeito à práticas de *design* de código para melhoria da segurança da aplicação.

Além disso, tem-se como objetivo nesta monografia oferecer duas propostas de ambientes de visualização e monitoramento do código com base nos cenários definidos:

- **Plataforma livre de monitoramento de métricas**: Evolução da plataforma Mezuro para suportar a criação de configurações que caracterizem os cenários estabelecidos;
- **Ambiente de DWing**: Criação de um ambiente de DWing para extração e análise das métricas baseada nos cenários definidos para a ajuda na tomada de decisão;

Assim, pretende-se evidenciar como estes cenários podem ser utilizados através da ferramenta de análise estática de código automatizada incorporados as boas práticas de desenvolvimento das Engenharia de Software. Uma comparação entre as duas propostas também cabem ao escopo deste trabalho.

1.1.1 Questões de Pesquisa

Este trabalho busca responder as seguintes questões de pesquisa:

- **QP1** - Um ambiente de *Data Warehousing*(DWing) é adequado para auxiliar na tomada de decisão tanto do ponto de vista estratégico gerencial quanto no ponto de vista de modificações de *design* por desenvolvedores/Engenheiro de software?
- **QP2** - O Mezuro é adequado para suportar a tomada de decisões gerenciais de projetos de software e para auxiliar o Engenheiro de Software na melhoria contínua do produto?
- **QP3** - O Mezuro e o DWing podem ser utilizados em conjunto para auxiliar a tomada de decisões em vários níveis de projetos de software?
- **QP4** - Métricas de *design* de código possuem correlação com métricas de vulnerabilidades?
- **QP5** - O monitoramento de vulnerabilidades é viável em projetos de softwares não-críticos?
- **QP6** - Métricas estáticas podem ser compostas em cenários para a definição de indicadores mais informativos e completos?

Para responder estas questões de pesquisa, busca-se atingir os seguintes objetivos específicos:

Objetivos Técnológicos

- **OT1** - Evoluir a plataforma livre Mezuro de monitoramento de código-fonte:
  - Evoluir a arquitetura do Mezuro para melhorar sua modularização e flexibilidade.
  - Evoluir a configuração de métricas do Mezuro
  - Implementar mecanismos de definição de celos/cenários no Mezuro
  - Melhorar mecanismos de visualização de software do Mezuro
- **OT2** - Criar ambiente de Data Warehousing para monitoramento dos cenários de decisão no contexto de vulnerabilidade de software.
  - Implementar extração, transformação e carregamento da base de dados a partir de reports de ferramentas de análise estática na ferramenta Pentaho.
  - Criar modelo dimensional
  - Gerar Cubo de dados
  - Configurar mecanismos de visualização do cubo de dados na plataforma BI server.
- **OT3** - Evoluir extração de métricas de vulnerabilidade para a ferramenta Analizo

Objetivos Científicos

- **OC1** - Catálogar definições teóricas a respeito dos principais conceitos relacionados à vulnerabilidades de software.
- **OC2** - Estudo teórico sobre a relação de vulnerabilidades de software com o *design*
- **OC3** - Análise estatísticas de dados de softwares livres a fim de comprovar correlação entre vunerabilidades e *design* de software.
- **OC4** - Propor um modelo baseado em sequência de passos para o Engenheiro de Software no desenvolvimento de softwares robustos.
- **OC5** - Definição de cenários a partir de estudos teóricos para melhorar a interpretação e tomada de decisão sobre métricas estáticas de código-fonte.

1.2 Metodologia e Pesquisa

Os objetivos científicos e tecnológicos deste trabalho contemplam passos para a alcançar o seu Objetivo Geral e responder às perguntas de pesquisa. Portanto, serão realizados estudos teóricos através de revisão bibliográfica para compreensão e definição

dos conceitos básicos sobre características de *design* e de vulnerabilidades de software, contemplando os objetivos científicos OC1 e OC2.

Além das discussões obtidas a partir da revisão bibliográfica, para se alcançar o OC3 será projetado um expertimento para se avaliar a relação entre características de *design* e vulnerabilidades de software. Este experimento consistirá em uma coleta de métricas em diversos projetos de software livre e a análise estatística de correlação entre essas métricas, onde pretende-se analisar softwares críticos e não críticos. Neste sentido, espera-se respoder as questões de pesquisa QP4 e QP5. Para tanto, será utilizado o modelo de regressão múltipla para verificar a correlação entre os dados. Iremos então determinar o impacto de métricas de *design* (variáveis independentes) com a ocorrência de vulnerabilidades específicas (varíaveis dependentes). Este experimento será projetado através de um protocolo de estudo experimental, contemplando a definição de hipóteses, seleção da amostra, desenho do estudo, estratégia de condução e análise de dados conforme proposto por Luna (1998).

Para responder às questões de pesquisa Q1, Q2 e Q3, ou seja, verificar a adequação do ambiente de DWing e da plataforma de monitoramento Mezuro para auxílio na tomada de decisão orientada a métricas de software, será elaborado um protocolo de estudo de caso. Este protocolo será usado para guiar a condução do estudo de caso e deverá conter os procedimentos e regras que governam a condução da pesquisa (MILES; HUBERMAN, 1994). O objetivo principal deste estudo de caso é avaliar aspectos e características das soluções frente às necessidades de projetos de software. Com o resultado da aplicação desse protocolo, também será possível comparar as duas soluções e verificar se ambas podem se complementar, fornecendo um serviço mais completa e com mais benefícios.

Por fim, baseado principalmente na revisão bibliográfica sobre *design*, segurança e métricas estáticas de software, as duas ferramentas propostas, Mezuro e ambiente DWing, serão construídas e evoluídas. Esta atividade está relacionado aos objetivos tecnológicos deste trabalho de conclusão de curso, onde técnicas, métodos e padrões de Engenharia de Software poderão se aplicados.

1.3 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 a *design*, vulnerabilidade e métricas de software. Posteriormente no Capítulo 3 será apresentada a plataforma de monitoramento de código-fonte Mezuro, uma ferramenta livre que será evoluída de acordo com os Objetivos Tecnlógicos deste trabalho. O Capítulo 4 apresenta os principais conceitos relacionados a Data Warehousing que também será explorada no contexto deste trabalho como solução para utilização de métricas de código-

fonte. Por fim, no Capítulo 5 serão apresentados o conceito de Cenários de Decisões e algumas propostas que podem ser utilizadas no monitoramento de projetos.

2 Design, Segurança e Métricas de Software

2.1 Design de Software

O desenvolvimento de software é um processo complexo, pois envolve diversas etapas que visam o entendimento dos problemas a serem solucionados, o projeto de uma solução, a implementação desta solução, testes sobre o produto, dentre outros passos importantes e não menos complexos. 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. Devida esta complexidade, tanto do 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* se torna mais importante a 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, estes atributos de qualidade são fatores fundamentais na atratividade de colaboradores para os projetos, onde se observa uma correlação entre métricas de qualidade de código-fonte com a atratividade desses projetos (MEIRELLES, 2013).

Visando amenizar os riscos de se construir um sistema que não alcance seus objetivos, a arquitetura do software tem recebida atenção especial 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. Neste 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 estes princípios podem ser aplicados no código-fonte. Posteriormente, identificamos como as decisões de design são observáveis em termos de *Bad Smells* e Código Limpo. Neste sentido, pretendemos estudar e revisar como estas características observáveis possuem relação com métricas de monitoramento 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, Princípios e Práticas

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, este 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.

Nesta monografia estamos essencialmente interessados nas decisões de projetos a nível de código e suas consequências no desenvolvimento do software. Assim, estamos falando principalmente do trabalho realizado pelo Engenheiro de Software, de quais formas este trabalho é realizado e como podemos apoiar e evoluir a atuação deste profissional para obtenção de bons resultados para projetos de software. Para o desenvolvimento de códigos com bom *design*, 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 destes princípios.

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, a complexidade se torna o principal 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 indesejadas para o software surgem a partir da atribuição inadequada de responsabilidades entre os módulos que o copõe. 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, mudanças simples se tornam complexas, pois ocorrem em mais de um lugar e se torna cada vez mais difícil a criação de testes unitários para o software, pois os componentes tendem a não poder serem testados isoladamente (MARTENSSON; GRAHN; MATTSSON, 2005).

Os problemas citados acima, assim como outras características indesajadas 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). Estes princípios podem ser genéricos, como características desejáveis em qualquer código-fonte, ou podem estar relacionados com um paradigma específico, acoplado as estruturas introduzidas pelo paradigma. A seguir são apresentados princípios gerais que devem ser levados em consideração pelos Engenheiros de Software no desenvolvimento de qualquer aplicação, pois são fundamentais para 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 auto-contidas 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 extendido e aplicável na definição do conhecido princípio *DRY - Don't Repeat*

Yourself[1] que é bastante enfatizado em *frameworks* modernos de desenvolvimeno 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 muito 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. Esta 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 este 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, 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 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

[1] <http://c2.com/cgi/wiki?DontRepeatYourself>
[2] <http://rubyonrails.org/>
[3] <https://www.djangoproject.com/>

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 este 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.

A concepção do *design* para endereçar os problemas de software consiste na aplicação destes e outros princípios a partir de técnicas e práticas no decorrer do desenvolvimento do software. Esta concepção se inicia na escolha do paradigma que irá reger o desenvolvimento. Uma 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 daí, 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. Por outro lado, estamos mais 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 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 1[4].

O conjunto de práticas destacadas na Figura 1 podem ser utilizadas para a aplicação e exercicío dos princípios que são fundamentais para o desenvolvimento de um software que atenda a requisitos de qualidade 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, sendo de suma importância para se conseguir os objetivos de nível arquitetural. Ainda a respeito de práticas como *refactoring* e integração contínua vale ressaltar que estas são aplicadas constantemente pelos Engenheiros de Software, várias vezes por iteração até se atingir os objetivos funcionais e não-funcionais estabelecidos.

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

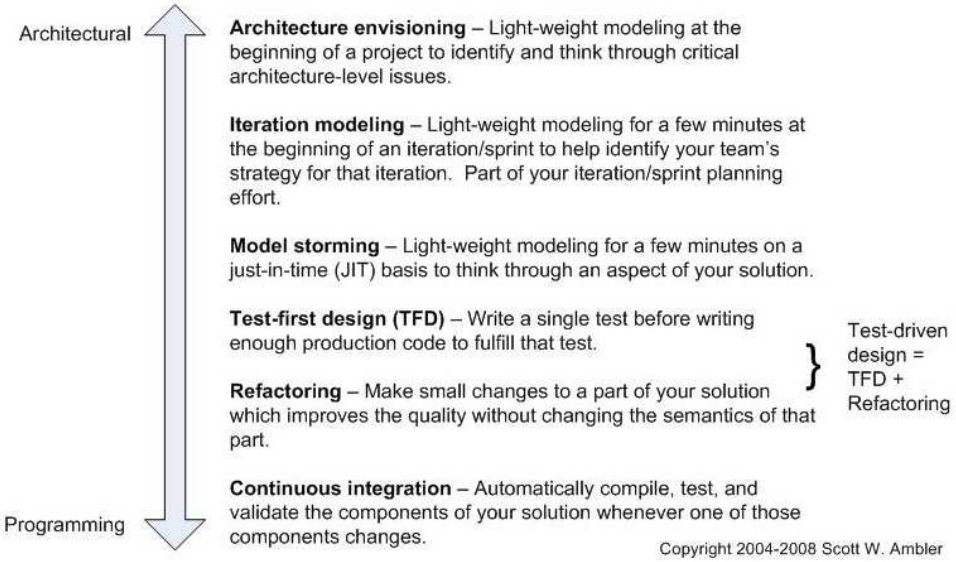

Figura 1 – Práticas do Design Ágil

Durante as iterações e a aplicação das práticas da Figura 1, outros elementos e técnicas de *design* são importantes e devem ser considerados para se aplicar os princípios de *design*. Paradigmas de programação, padrões de projetos, estilo de código e utilização de *frameworks* são 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. 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*[5]. 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 extendida para o conceito de Código Limpo, explorado posteriormente nesta seção.

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

[5] <http://checkstyle.sourceforge.net/>

livro *Design Patterns: Elements of Reusable Object-Oriented Software*, Erich Gamma e 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, extensível, 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, extensibilidade 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).

2.1.2 *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 semânticas 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 a outro objeto uma sequência de objetos para realizar alguma operação. Indica um forte acoplamento entre estas 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 fragilizadas, pois dependem 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. Neste 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.

2.1.3 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. Este 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. Esta 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 a 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 1. 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 estes 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*. Neste 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.

2.2 Segurança de Software

A segurança de software está relacionada com o contínuo processo de manter a confiabilidade, 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) cita que o custo e esforço gastos na segurança do software são bem altos, podendo chegar a 70% to 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 coorporações 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). Este 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 humidade, poeira, calor, locais inseguros, dentre outros fatores físicos relacionados ao local onde se encontra a infra-estrutura 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. Neste 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 este processo. Como consequência, o sistema operacional pode realocar esta posição de memória para outro processo que, se acessado novamente pelo primeiro processo, irá conter dados incosistentes 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 este endereço de memória será sobrescrito na pilha de alocação do processo pela chamada de funções posteriores. Além de incosistência 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*[6].

- ***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 estam 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 eploração de *SQL Injection* pode permitir ao atacante levar vantagens através da persistência de informações e geração de conteúdos dinâmicos em páginas web (DOUGHERTY, 2012).

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

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 atancantes utilizam estas vulnerabilidadespara 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[7] 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 entra as organizações.

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 (US. Departament of Homeland Secutity) e o NIST (National Institute of Technology). 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

[7] <http://www.mitre.org/>

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 nivel 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).

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[8].

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*[9]. Utiliza os conceitos da biologia de Reino e Filo. Nessa taxonomia, o

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

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;
- **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*[10];
- **Encapsulamento**: são erros relacionados ao não estebelecimento 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 idéia desta 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 este cenário apresentado acima, em que temos a CVE como projeto de enumerar as vulnerabilidades encontradas e as taxonomias elaboradas por diversos trabralhos com o objetivo de classificar e dar mais informações sobre a vulnerabilidade, ainda haviam a necessidade das empresas e organizações de utilizarem 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.

[10] Falta de memória livre para uso no sistema

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 possivel saber quais vulnerabilidades as ferramentas conseguem capturar; e prover uma base de informações padrão a respeito de como identificar, mitigar e previnir 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, 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 que tal característica pode 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. Neste sentido, é de suma 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 devem 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, 1975) (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 corrigí-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. Esta 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 sugere a separação entre as entidades que aprovação a ação, entidades que realizam a ação e entidades que monitoram a ação [11]. Este princípio está diretamente relacionado com os princípios de design orientado à objetos tais como coesão e separação de interesses. Por 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.

Alguns módulos do software requerem mais atenção do que outros quanto riscos de vulnerabilidades. Neste 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. Isto acontece devido a 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.

[11] <https://www.owasp.org/index.php/Separation_of_duties>

- **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 o principal responsável por manter a segurança de seus projetos. Este 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 (FAZER CITAÇÃO), é de suma importância 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.

2.3 Métricas em Engenharia de Software

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 (CITAR TÉCNICAS) 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 estes 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.

Neste sentido, o monitoramento da qualidade de código-fonte é fundamental e pode 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 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. Neste 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 enderaçar 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 (PRECISA CITAR FERRAMENTAS???).

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). Isto 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 (MEIRELLES, 2013)(ALMEIDA; MIRANDA, 2010) estudam formas de viabilizar a utilização de métri-

cas pelos desenvolvedores em geral. O presente trabalho visa correlacionar métricas de desenvolvimento de software com objetivo de definir configurações para estabelecer cenários que representem o estado da qualidade do software. Desta 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.

2.3.1 Métricas Estáticas de Design de Software

Nesta sub-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 mensuram 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.

- **(*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).

- **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 sub-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.
- **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 cálcula 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.

2.3.2 Métricas Estáticas de Segurança

Atualmente existem várias ferramentas de análise estática que detectam vulnerabilidades no código fonte. Essas ferramentas utilizam de divesas 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 é uma alternativa muito interessante ao 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 importancia do uso de ferramentas de análise estática para a identificação de vulnerabilidades, visto que uma vulnerabilidade indentificada 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.

Então, 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 um vulnerabilidade especifica no projeto
- Quantidade de um vulnerabilidade especifica por arquivo
- Quantidade de um vulnerabilidade especifica 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 itrodução de vulnerabilidades.

- **UAV(*Uninitialized Argument Value*- Varíavel não inicializada)**: Esta vulnerabilidade identifica 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 anteriomente 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 dificil de se identificar, sendo aconselhável o uso de ferramentas de análise estática de código.

- **PITFC (*Potential insecure temporary file in call "mktemp"* - Arquivo temporário pontencialmente 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 vuneráveis 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 um ponteiro que não foi inicializado é acessado. Esta vulnerabilidade está relacionada a CWE 457 (Use Unitialized of variable) pois o ponteiro está indefindo pois não foi inicializado. Portanto, as consequencias 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 previnir 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 excessoes.

- **MLK (*Memory leak* - Estouro de memória)**: O software não gerencia o uso de memória, provocando o consumo exessivo 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 indice 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 condigo 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 construidas.

- **AUV (*Assigned value is garbage or undefined* - Valor atribuído é lixo ou indefinido**:Esta vunerabilidade ocorre quando não temos certeza que o valor atribuido existe. Esta vulnerabilidade ocorre quando uma variável recebe um valor cujo a origem vem de entrada do usuário no sistema. Caso não seja feito a verificação desse parâmetro de entrada, a variável poderá receber lixo ou um valor indefinido e sua posterior utilização pode causar desde quebra da aplicação como também resultados inesperados de comportamento de software.

3 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 ferramentas que podem ser utilizadas 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. Na Seção 2.3, apresentamos um conjunto de métricas estáticas cujo propósito é mensurar 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, é de extrema importância 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 3.1).

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 2 apresenta a tela principal da nova versão do Mezuro.

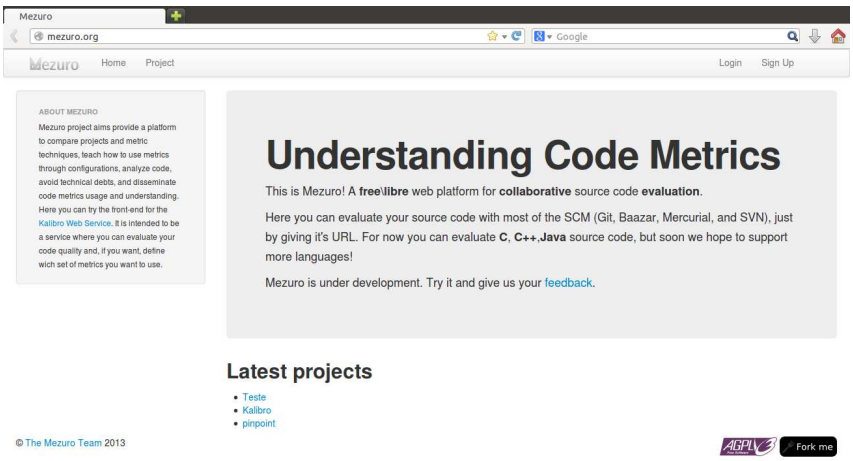

Figura 2 – Tela principal do Mezuro. Disponível em <http://mezuro.org/>

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

- **SonarQube**[12] - Plataforma livre de gerenciamento de qualidade de código que

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/>
12 <http://www.sonarqube.org/>

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.

- **SQO-OSS**[14] (*Software Quality Assessment of Open Source Software*) - Conjunto de ferramentas de análise e *benchmarking* de projetos de software livre.

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.

[13] <https://codeclimate.com/>
[14] <http://www.sqo-oss.org/>

3.1 Arquitetura do Mezuro

Como mencionado, o Mezuro está evoluindo para uma aplicação independente. Neste sentido, a arquitetura da plataforma como um todo está sendo modificada, visando principalmente maior modularização em diversos serviços independentes. A Figura 3 apresenta a arquitetura atual da plataforma.

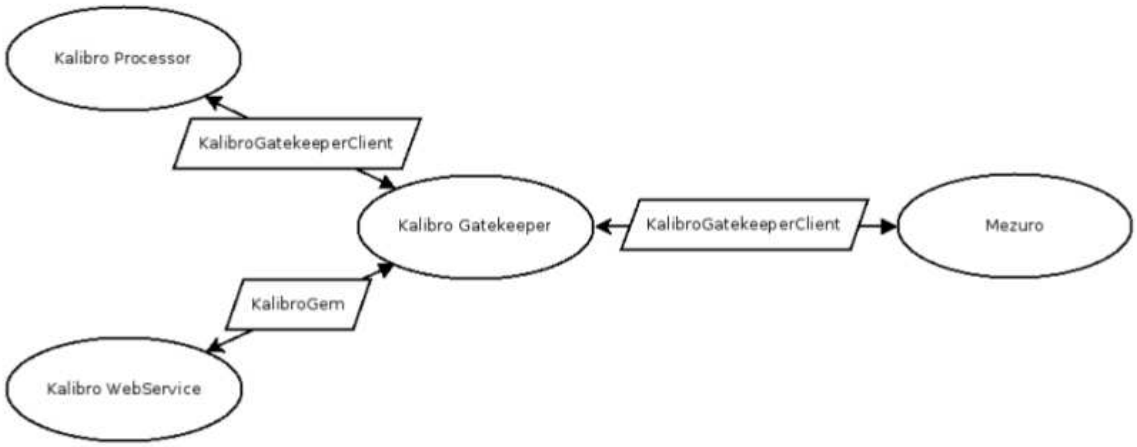

Figura 3 – Arquitetura Atual do Mezuro. Extraído de (MANZO et al., 2014)

O Mezuro utiliza o WebService Kalibro Metrics [15] para fornecer a funcionalidade de análise e avaliação de métricas de código-fonte. O Kalibro é um WebService SOAP[16], baseado em mensagens XML[17] 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 estes módulos principais é intermediada pelo módulo correspondente à elipse Kalibro Gatekeeper.

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 4. O primeiro módulo já está em uso, sendo também represetado na Figura 3, é o módulo Kalibro Processor. Neste sentido, 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. Esta 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).

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

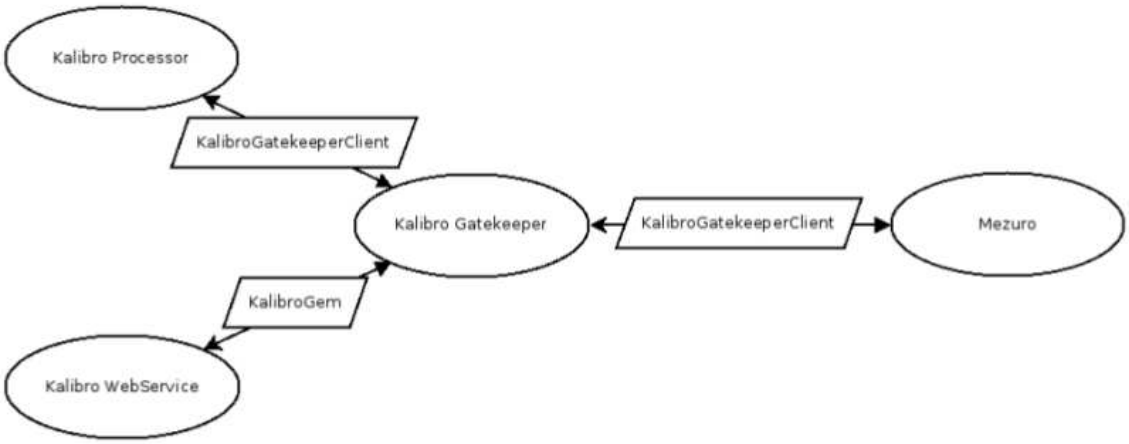

Figura 4 – Arquitetura Futura do Mezuro. Extraído de (MANZO et al., 2014)

4 *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 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 sistema de vendas de uma loja, podemos analisar as informações sobre venda, como também podemos analisar informações sobre frequência de funcionários ou também sobre o estoque. 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 a atualização 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.

4.1 *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.

Kimball (2002) define os componentes básicos de um ambiente de DWing conforme a Figura 5:

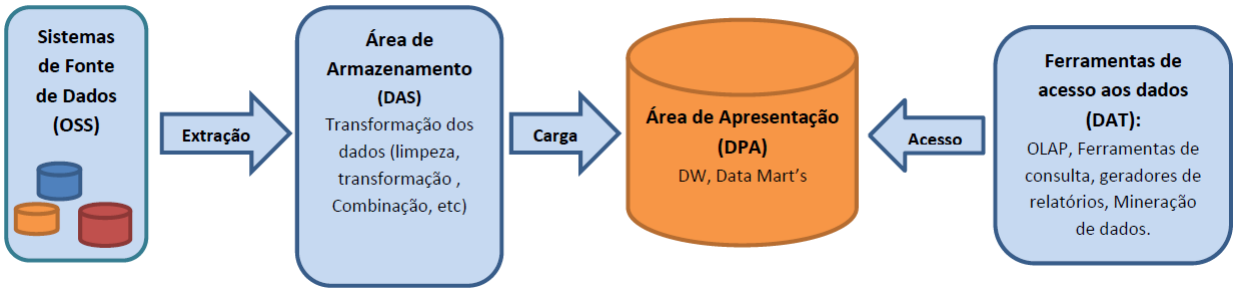

Figura 5 – Componentes de um DWing (KIMBALL; ROSS, 2002)

.

4.1.1 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. Essas bases de dados não precisam ser 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. As fontes 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).

4.1.2 Área de Preparação dos Dados - DSA

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. 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.

4.1.3 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 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 envolve 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 visto na Seção (4.2).

4.1.4 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*. Um *dashboard* é um tipo de aplicação analítica constituída por uma tela com informação resumida que pode conter diferentes elementos de exploração, como tabelas, gráficos, manometros, entre outros, tal como se mostra na Figura (6).

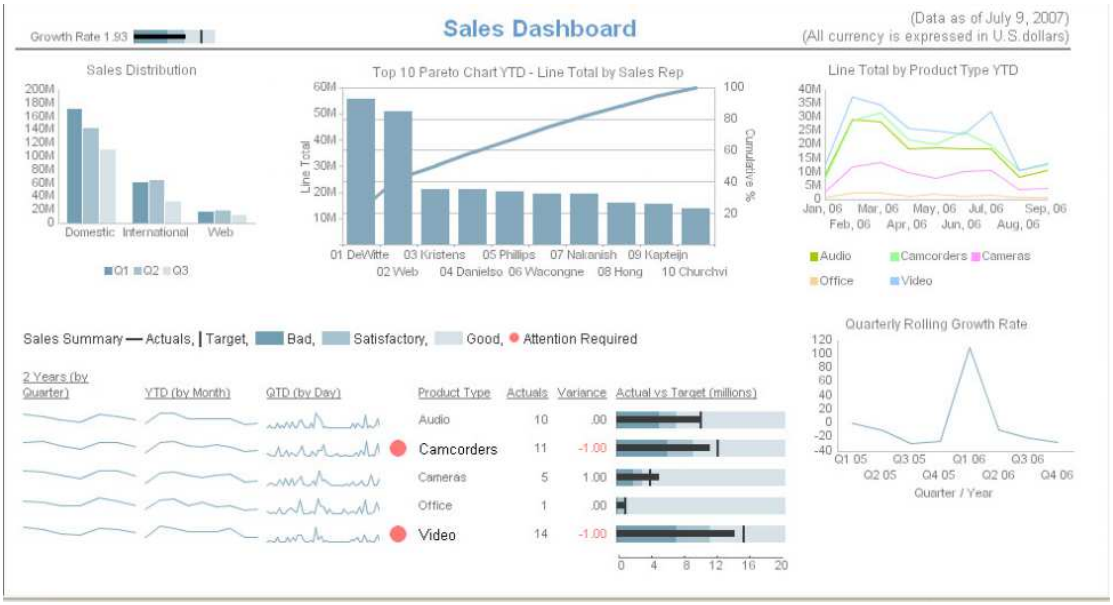

Figura 6 – Exemplo de visualização de dados através de dashboards

.

4.2 Modelage dimensional

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.

A idéia 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 7 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.

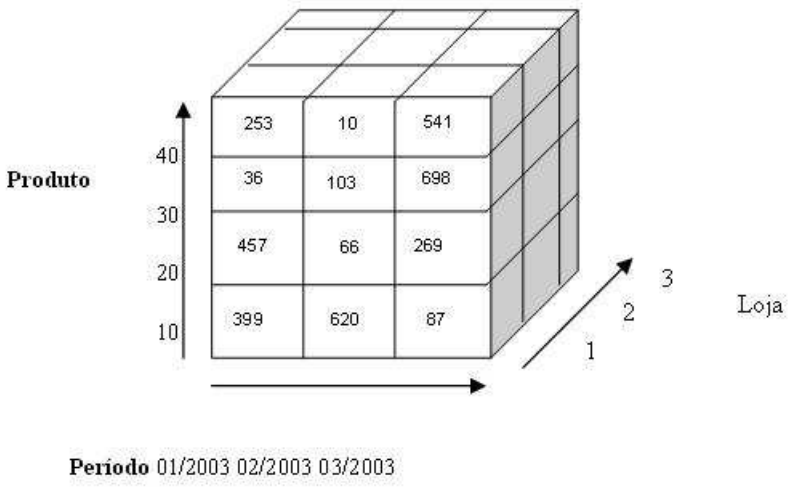

Figura 7 – Exemplo de cubo de dados

.

O modelo dimensional proposto por Kimball (2002) é chamado de modelo estrela (*star scheme*, Figura 8). 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 primarias de diferentes dimensões. Para juntar as informações basta fazer o *Join* entre elas.

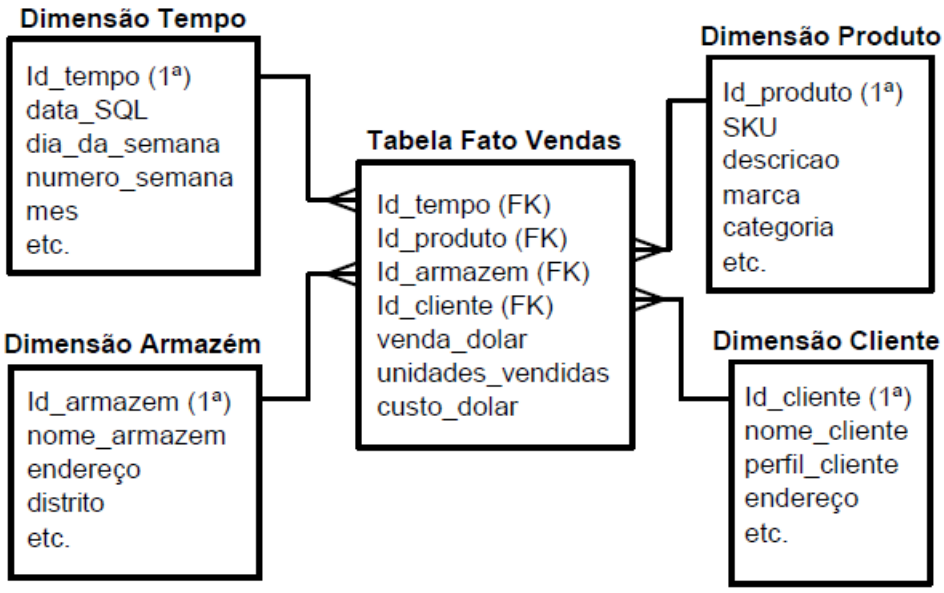

Figura 8 – Esquema estrela

.

Uma Tabela de Dimensão deve ser construída de maneira a incluir atributos que podem ser agregados, fornecendo ao usuário maneiras alternativas de visualizar as informações. Como por exemplo, uma tabela de produto tem o atributo “nome do produto” e o atributo “categoria”. Dessa forma, o usuário poderá agrupar os produtos por categoria para ter uma diferente visão sobre a informação. Diferente do modelo relacional, o fato da tabela dimensão não ser normalizada implica na melhoria da performance, pois nesse exemplo citado, “categoria” do produto 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 9. 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 4.3.

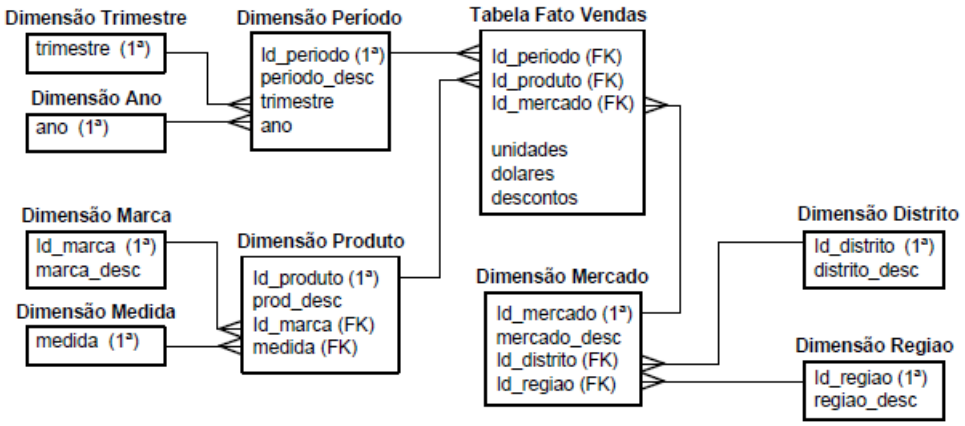

Figura 9 – Esquema floco de neve

.

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 na 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.

- **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 "o que estamos medindo?". Nessa estapa é definido os fatos numéricos que irão popular as tabelas fatos.

4.3 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 seguintes operações:

- ***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 ano. Uma operação de *Drill Down* consistiria em trazer essa mesma informação por mês.

- ***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, em um cubo com informações sobre a venda, fatiar este cubo pelo ano 2012 consiste em eliminar a dimensão tempo e apenas considerar as vendas para o ano de 2012. 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* em um cubo de vendas consiste em selecionar as vendas entre o ano de 2010 a 2013, nas localizações “Rio de janeiro”, “São Paulo”, “Distrito Federal” e produtos de categoria “Alimentos” e “Eletronicos”.

- ***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).

A Figura (10) exemplifica cada uma das operações OLAP que podem ser aplicadas em um cubo de dados.

4.4 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

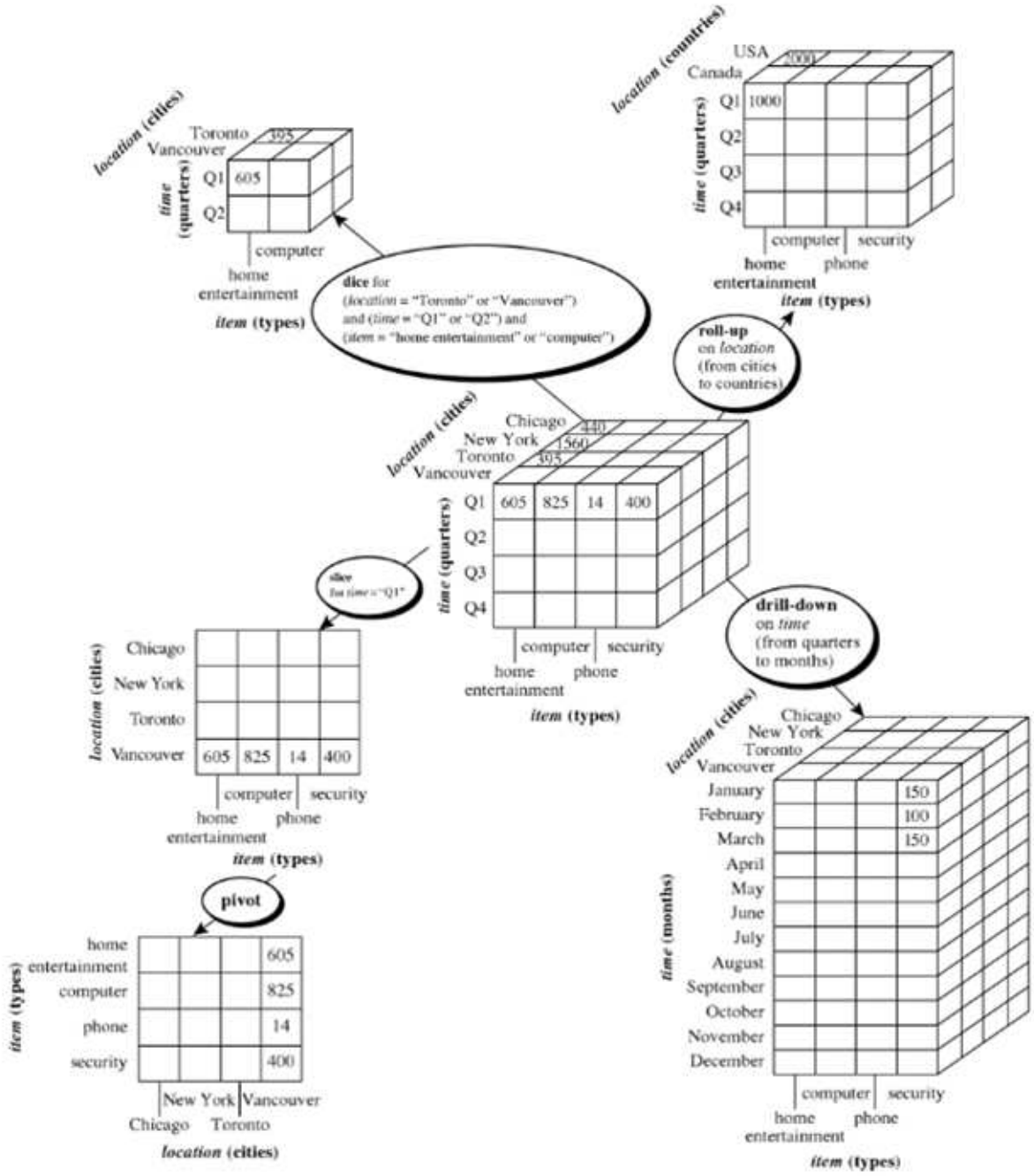

Figura 10 – Operações OLAP

.

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 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-

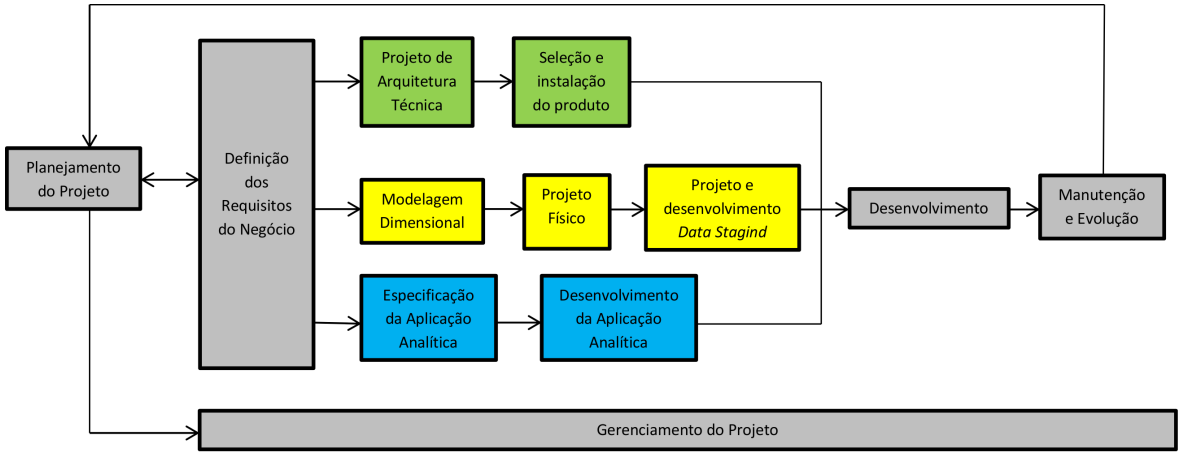

Figura 11 – Ciclo de vida de um Projeto de DWing (KIMBALL; ROSS, 2002)

.

res (na cor verde) da Figura 11 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.

5 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 este objetivo.

Na seção 2.3 do Capítulo 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. Neste 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 cápitulo, extendemos este 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 abstração de software que pode ser caracterizada pelo cenário, por exemplo: Projeto; Estrutura de Herança; Classe; Método;
- **Descrição**: Discuti os problemas, princípios envolvidos e a caracterização

- **Caracterização com Métricas**: Define e discuti como as métricas envolvidas devem ser utilizadas para identificar o cenário. 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. Neste sentido, 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 reunem um conjunto de métricas necessários para sua caracterização. Mesmo quando um cenário é composto 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, mas não especificando valores de intervalos necessários para caracterizar este cenário. 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 determinados contextos.

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 destes cenários, diminuindo-se o esforço necessário para coleta, interpretação e viusalização de dados. Entretanto, os benefícios dos Cenários de Decisões são passíveis de experimentação, experimentos esses que estão fora do escopo deste trabalho.

5.1 Definição de Cenários de Decisão para Vulnerabilidades 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 caracteristicas apresentdas 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 muito provavelmente 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 2.3.2) são vulnerabilidaddes específicas de uso de códigos, funções ou práticas consideradas perigosas para segurança da aplicação. Tais vulnerabilidades podem ser encontradas no código fonte indepentente da aplicação de boas práticas de *design*, pois tais vulnerabilidades são difíceis de identificar pois necessitam do conhecimento mais aprofundado do Engenheiro de Software para que este saiba que está inserindo uma vulnerabilidade no código.

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

- **Cenários de Decisão para Caracterização da Qualidade de Código**: Estes cenários buscam identificar caracteristicas de software relacionadas a qualidade e design de código que podem influenciar em sua segurança, como complexidade, acoplamento, etc.

- **Cenários de Decisão para Caracterização de Vulnerabilidades espefífica de código**: Estes cenários buscam identificar vulnerabilidades de software que podem ser encontradas em código-fonte indepente 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;

5.1.0.1 Cenários de Decisão para Caracterização da Qualidade do Código

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
- **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**:
  - **Composição NPA/NOA** - Caracteriza o tamanho da superfície de ataque a atributos em percentagem. O cenário existe quando o valor desta métrica composta está acima do valor estipulado.
- **Ações sugeridas**:
  - **Refatorações aplicáveis:** - *Encapsulate Field*[4].
  - **Princípios aplicáveis à classe** - Princípio e Redução de Superfície de Ataque; Princípio de Encapsulamento;

Alta Superfície de Ataque Operacional

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

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), o que 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), MNPM (Número Máximo de Parâmetros por Método);
- **Nível**: Classe;
- **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** - Caracteriza a quantidade de interações possíveis para as quais o princípio *Mediate Completely* deve ser aplicado. O cenário pode existir caso os valores desta métrica esteja acima do estipulado;
  - **MNPM** - Caracteriza a quantidade de informações que poderiam ser verificadas para evitar ataques externos. O cenário pode existir caso os valores desta métrica esteja acima do estipulado;

- **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];

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 quatindade 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:

- **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.

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

- **Caracterização com Métricas**:
  - **Composição ACC + NOC** - Caracteriza 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, ou seja, o número de classes acopladas a classe em análise é superior ao valor definido para o projeto.
- **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);

5.1.0.2 Cenários de Decisão para Caracterização de Vulnerabilidades Específicas de Código

Entradas não validadas

Este cenário busca identificar situações no código fonte onde valor de uma entrada de usuário é passado para uma variável sem que tenha sido feito, antes da atribuição e uso, uma verificação desse dado de entrada. Essa vulnerabilidade é catalogada no padrão CWE como CWE-20 [11], que é classificada como uma classe que inclui diversas vulnerabilidades com mais nível de detalhe relacionadas a não validação. A não validação de atributos de entrada de usuário pode implicar em diversos riscos para a aplicação. Com essa vulnerabilidade podem ser exploradas vulnerabilidades mais específica como *Buffer Overflow*, *SQL injection* e *XSS*, permitindo ao atacante inserir valores inesperados, podendo causar a quebra da aplicação; ler arquivos confidenciais por manipulação do do programa, como por exemplo o uso de *SQL injection*; como também pode modificar dados ou mudar o fluxo de programa explorando o *Buffer Overflow*.

Em termos de métricas de código fonte, este cenário busca medir a quantidade de vulnerabilidades que identificam a ocorrência da não verificação dos parâmetros de entrada. As métricas AUV (*Assigned value is garbage or undefined*) e FGBO (*Potential Buffer Overflow in call to "gets"*), descritas na Seção (2.3.2) medem a quantidade de vulnerabilidades relacionadas a não validação de entradas na linguagem C/C++.

[8] <http://refactoring.com/catalog/extractClass.html>
[9] <http://refactoring.com/catalog/moveMethod.html>
[10] <http://refactoring.com/catalog/pushDownMethod.html>
[11] <http://cwe.mitre.org/data/definitions/20.html>

Para tratar este cenário deve ser feita a verificação e tratamento do dado de entrada antes de seu uso para se certificar que a informação está realmente condizente ao contexto. Então pode ser interessante a criação de uma lista de valores válidos ou determinação de regras para a verificação desses dados de entrada. Também vale a criação de mecanismos que limitem o tamanho do *buffer* de entrada para que não haja estouro do mesmo.

Quadro resumo:

- **Nome**: Entradas não validadas
- **Classificação**: Composta
- **Métricas Envolvidas**: AUV (*Assigned value is garbage or undefined*) e FGBO (*Potential Buffer Overflow in call to "gets"*)
- **Nível**: Método;
- **Descrição**: Esse cenário busca identificar parâmetros de entrada de usuário que não são validados ou verificados, e posteriormente são utilizados pela aplicação. Nessa situação, não temos certeza o real valor que foi passado como entrada, podendo causar desde comportamento inesperado da aplicação até quebra e leitura de dados confidenciais.
- **Caracterização com Métricas**:
  - **Composição AUV+FGBO** - Aponta linhas do código em que se encontram a variáveis que podem ter o valor indefinido ou com lixo obtidos na entrada do programa e também o uso da função *gets*. Este cenário existe quando o valor calculado desta métrica é diferente de zero, ou seja, esse cenário existe se tiver pelo menos uma ocorrencia dessa métrica no código fonte
- **Ações sugeridas**:
  - **Implementar mecanismos de validação** - A implementação de mecanismos de verificação da integridade e consistência do dado de entrada ajuda a garantir que esta entrada não seja indefinida ou lixo.
  - **Substituir chamadas para a função *"gets"*** - Caso não seja possível utilizar de recurso de interface gráfica para a limitação do tamanho de entrada do usuário, a função *"fgets()"* pode ser utilizada no lugar da função *"gets"*, pois aquela possui um parâmetro em sua chamada que indica o tamanho maximo do buffer.

Variáveis não inicializadas

Este cenário busca identificar o uso de variáveis não inicializadas. Este cenário também é definido pela CWE-457 [12]. A declaração de variáveis em algumas linguagens, como C, Perl e PHP, não inicializam estas variáveis com valores default. Com isso, variáveis que são criadas na pilha de memória podem conter em seus valores lixo ou até comandos e valores de outras funções e variáveis que também utilizaram da mesma área de memória em outro instante. 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 a quantidade de variáveis não inicializadas. A métrica UAV (*Uninitialized Argument Value*) busca medir exatamente essa vulnerabilidade.

Quadro resumo:

- **Nome**: Variáveis não inicializadas
- **Classificação**: Monométrica;
- **Métricas Envolvidas**: UAV (*Uninitialized Argument Value*);
- **Nível**: Método;
- **Descrição**: Este cenário busca identificar o uso de variáveis não inicializadas, pois isso pode causar desde falhas na aplicação quanto acesso a informaçãõ não autorizado.
- **Caracterização com Métricas**:
  - **UAV** - Este cenário ocorre quando existe ao menos uma ocorrência dessa vulnerabilidade.
- **Ações sugeridas**:
  - **Princípios aplicáveis** - Redução da superfície de ataque, inicializando variáveis no momento em que são instanciadas.

[12] http://cwe.mitre.org/data/definitions/457.html

Cenário	**Caracterização do cenário**	**Nível do cenário**	**Descrição**	**Caracterização com Métricas**	**Ações Sugeridas**
Alta Superfície de Ataque a Atributos Internos	Qualidade de código	Classe	Busca diminuir a exposição da classes em relação aos seus atributos públicos	Valor alto de NPA/NOA	**Refatorações**: Encapsulate Field; **Aplicar Princípios**:Redução da 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.	Valor alto de número NPM e MNPM	**Refatorações**: Hide Method; Remove Parameter; **Aplicar Princípios**:Princípio e Redução de Superfície de Ataque; Princípio de Encapsulamento; Princípios de Distribuição de Responsabilidades GRASP; **Aplicar padões**: Padrão facade

Tabela 1 – Parte I - Resumo de todos os cenários propostos para o monitoramento da segurança de software

Cenário	**Caracterização do cenário**	**Nível do cenário**	**Descrição**	**Caracterização com Métricas**	**Ações Sugeridas**
Ponto Crítico de Falha	Qualidade de código	Classe	Este cenário busca identificar classes muito acopladas, que representam pontos críticos da aplicação	Alto valor de ACC+NOC	**Refatorações**: Extract Class, Move Method, Push Down Method; **Aplicar Princípios**:Modularização, Baixo Acoplamento; Princípio de Encapsulamento; Princípios de Distribuição de Responsabilidades GRASP
Entradas não validadas	Vulnerabilidade específica	Método	Busca indentificar a não ocorrencia de validação de variáveis de entrada por outros mecanismos e também pelo uso da função *gets*	Ao menos uma ocorrência da métrica FGBO ou AUV	Implementar mecanismos validação de entrada de usuário ; substituir função *gets* pela *fgets*
Variáveis não inicializadas	Vulnerabilidade específica	Método	Variáveis não inicializadas podem causar falha da aplicação ou acesso não autorizado de informações caso sejam utilizadas	Ao menos uma ocorrência da métrica UAV	Inicializar variáveis sempre que criar uma

Tabela 2 – Parte II - Resumo de todos os cenários propostos para o monitoramento da segurança de software

6 Considerações Finais

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 discutir e compreender como métricas de código-fonte podem ser utilizadas dentro do processo de desenvolvimento de software. Também estudamos ferramentas que podem apoiar a utilização de métricas em projetos de software, além de propor uma técnica para o uso de métricas de código-fonte que possa diminuir as principais dificuldades inerentes à medição de software, suportanto a tomada de decisões técnicas e gerenciais.

A primeira etapa deste trabalho apresentou uma revisão bibliográfica sobre qualidade interna de software, abrangendo aspectos relacionados ao *design* e segurança. Além disso, foi apresentado os principais conceitos de métricas de software onde explorou-se as métricas que podem ser utilizadas para mensurar alguns atributos de qualidade do software e até identificar possíveis vulnerabilidades existentes.

Nesta etapa do trabalho ainda foram estudadas ferramentas que poderão ser utilizadas para apoiar a utilização de métricas em projetos de software. Portanto, foi explorado a plataforma livre de monitoramento de código-fonte Mezuro e a utilização de DWing, onde foram apresentadas as principais características e a arquitetura de cada solução. Estes estudos são fundamentais para a compreensão, construção e evolução dessas ferramentas para responder as questões de pesquisa QP1, QP2 e QP3. Este estudo também é fundamental para os próximos passos do trabalho que visam responder à essas questões de pesquisa, consistindo na definição do protocolo de estudo de caso que será utilizado para avaliar essas duas soluções e compará-las.

Como discutido ao longo deste trabalho, métricas de *design* de código podem ter relações com vulnerabilidades de software, pois vimos 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. Assim, apresentamos o conceito de Cenários de Decisões, uma proposta de estrutura para ser utilizado em projetos de software para minimizar as dificuldades existentes na adoção de métricas nesses projetos.

Foi visto também nas seções 2.2 e 2.3.2 que existem vulnerabilidades específicas e

catalogadas, que, porém, não existem no contexto de todas as linguagens de programação. Essas vulnerabilidades específicas estão sujeitas a ataques, pois são pontos falhos do software que estão sujeito a exploração por atacantes maliciosos.

Baseado no estudo teórico, podemos responder parcialmente a questão de pesquisa QP4 definida neste trabalho, que diz respeito a saber se métricas de *design* possui correlação com métricas de vulnerabilidade. Métricas de *design* podem especificar vulnerabilidades de software, porém não necessariamente estão diretamente relacionadas a vulnerabilidades específicas. Mas podemos afirmar que um código que segue bons princípios de *design* pode reduzir vulnerabilidades, pois em um código limpo e de baixa complexidade a manutenção e evolução é mais segura devido sua legibilidade, flexibilidade e simples. Para responder essa pergunta completamente, será criado um protocolo de estudo experimental para verificar a correlação entre esses dois grupos de métricas através da análise de softwares livres.

Neste trabalho também foi questionado se métricas isoladas poderiam se relacionar e compor cenários para definição de indicadores mais informativos e completos (QP6). O fato de que a análise de métricas isoladas podem gerar indicadores não tão confiáveis é um assunto discutido em vários trabalhos. 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 nesta 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.

6.1 Evolução do Trabalho

Para contemplar os objetivos desta monografia apresentamos nesta seção um planejamento macro relacionado à segunda etapa deste trabalho a ser desenvolvido, que pode ser observado na Tabela 8.

Atividade	Julho	Agosto	Setembro	Outrubro	Novembro
Contribuições com Mezuro	x	x	x	x	
Criar Ambiente DW	x	x	x	x	
Contribuições com Analizo	x	x			
Criação e Refinamento de Cenários de Decisões	x	x	x	x	x
Criar protocolo de Estudo de Caso de análise das ferramentas	x	x			
Criar protocolo de estudo de correlação entre métricas de código-fonte e de vulnerabilidade	x	x			
Aplicar estudo estatístico de correlação			x	x	
Aplicar Estudo de Caso sobre as ferramentas			x	x	
Análise de dados e resultados				x	x
Relatar contribuições				x	x

Tabela 3 – Cronograma para o TCC 2

Durante os quatro próximos meses de trabalho serão realizadas as atividades técnicas de Engenharia de Software para desenvolvimento e evolução do Mezuro e construção do ambiente DWing e, a medida que for necessário, evoluir o Analizo. Algumas contribuições com o Analizo já foram feitas durante a disciplina de Manutenção e Evolução de Software, na qual, em conjunto com outros alunos, contribuimos com a inserção de algumas métricas de vulnerabilidade e com uma funcionalidade que permite identificar a linha em que a vulnerabilidade está ocorrendo, sendo esta última uma funcionalidade que pode ser muito útil para a criação dos ambientes e auxiliar na tomada de decisão para solucioná-la.

Atualmente, a evolução do Mezuro está no ponto de finalização do módulo Kalibro Processor. Portanto, pretendemos contribuir para com a evolução do Mezuro como plataforma independente em diferentes níveis. Dada os objetivos tecnológicos e de pequisa expostos no Capítulo 1, essas contribuições devem ser realizadas nos diferentes módulos que compõem a plataforma, contemplando a inserção de novos extratores e métricas sob o Kalibro, assim como melhorias relacionadas as formas de acompanhamento e configuração

de métricas. Essas contribuições serão de suma importância para esta monografia, assim como para a evolução do Mezuro, uma vez que as decisões serão compartilhadas com a comunidade de desenvolvedores e as contribuições em código terão impactos importantes e comuns quando se trata de software livre. Portanto, as contribuições irão ajudar para melhor compreensão da utilização do Mezuro e suas funcionalidades. Além disso, algumas das evoluções desejadas e destacadas no Capítulo 1 podem se tornar fundamentais para que o Mezuro possa se tornar não só uma ferramenta com viés acadêmico, mas que também contemple diferentes objetivos dentro de projetos de software. Pretende-se ainda adicionar novos extratores e métricas ao Mezuro, o torando ainda mais flexível e aumentando seu potencial de uso.

A construção de um ambiente de DWing também é um dos principais objetivos desta monografia. Assim como o Mezuro, o ambiente de DWing tem como objetivo ser construído para auxiliar no monitoramento de cenários de vulnerabilidade de software em que uma aplicação se encontra e dar suporte a tomada de decisão. Dessa forma, deverá ser desenvolvida todas as tarefas descritas no ciclo de de vida de desenvolvimento de um DWing (seção 4.4), desde a modelagem dimensional, processo de ETL e visualização das informações obtidas para a realização de análises. A fonte de entrada de dados será ferramentas de análise estática de código e a periodicidade de coleta será definida durante a modelagem, pois nesse processo que será definido a granularidade das informações.

A fim de comparar as duas soluções para responder a algumas das questões de pesquisa definidas neste trabalho, será elaborado um protocolo de estudo de caso. Este protocolo será projetado durante os dois primeiros meses de trabalho (Julho e Agosto) para que a sua aplicação seja realizada nos dois meses posteriores (Setembro e Outubro).

Durante todo o período de construção da segunda etapa deste trabalho queremos refinar e identificar outros Cenários de Decisões. O estudo de correlação entre métricas de *design* e vulnerabilidades de software, assim como mais estudos teóricos são fundamentais para evolução e criação desses cenários. Para tanto, projetaremos e realizaremos o experimento que avaliará a correlação estatística entre as métricas estudadas. Por fim, os dois últimos meses serão destinados a análise dos dados e relato de resultados obtidos.

Referências

AFEK, J.; SHARABANI, A. *Dangling Pointer - Smashing the Pointer for Fun and Profit*. 2007. A whitepaper from Watchfire. Citado na página 41.

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 39.

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 37, 51 e 79.

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 45.

ARANHA, D. F. et al. *Vulnerabilidades no software da urna eletrônica brasileira*. [S.l.], 2012. Citado na página 55.

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 72 e 73.

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 21.

BECK, K. *Implementation Pattens*. [S.l.]: Addison-Wesley Longman Publishing Co., 2007. Citado 2 vezes nas páginas 22 e 37.

BECK, K.; ANDRES, C. *eXtreme Programming eXplained: Embrace Change*. [S.l.]: Addison-Wesley Longman Publishing Co., 2000. Citado na página 21.

BERANDER, P. et al. *Software quality attributes and trade-offs*. Tese (Doutorado) — Blekinge Institute of Technology, 2005. Citado na página 21.

BISHOP, M. *Computer Security: Art and Science*. [S.l.]: Addison-Wesley, 2003. Citado 2 vezes nas páginas 45 e 47.

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 51.

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 23.

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 76.

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 23.

CHRISTEY, S. Plover - preliminary list of vulnerability examples for researchers. In: . [S.l.: s.n.], 2006. Citado na página 43.

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 40.

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 41.

DUARTE, L. O.; BARBATO, L. G. C.; MONTES, A. Vulnerabilidades de software e formas de minimizar suas explorações. 2005. Citado na página 22.

ELMASRI, R.; NAVATHE, S. *Sistemas de Banco de Dados*. 4nd. ed. [S.l.]: Pearson Education do Brasil Ltda, 2006. Citado na página 71.

ERL, T. *SOA: Principles of Service Design*. [S.l.]: Prentice Hall, 2007. Citado na página 66.

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 35.

FENTON, N. E.; PFLEEGER, S. L. *Software Metrics: A Rigorous and Practical Approach*. [S.l.]: Course Technology, 1998. Citado na página 51.

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 na página 23.

FOWLER, M. et al. *Refactoring - Improving the Design of Existing Code.* [S.l.]: Addison-Wesley, 1999. Citado 3 vezes nas páginas 21, 35 e 36.

GAMMA, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software.* [S.l.]: Addison-Wesley, 1994. Citado na página 35.

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 39.

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 21.

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 35.

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 51.

HOWARD, M. A process for performing security code reviews. In: IEEE. *IEEE Security & Privacyg.* [S.l.], 2006. p. 74–79. Citado na página 47.

INMON, W. H. *Building the Data Warehouse.* 3nd. ed. New York, NY, USA: Jhon Wiley & Sons, Inc, 2002. Citado 2 vezes nas páginas 69 e 71.

ISO/IEC 9126 - International Standard. Information Technology - Software Product Quality. [S.l.], 1998. Citado na página 35.

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 40.

KERIEVSKY, J. *Refatoração para Padrões.* [S.l.]: Bookman, 2008. Citado na página 35.

KERNIGHAN, B. W.; PLAUGER, P. J. *The Elements of Programming Style.* [S.l.]: McGraw-Hill Book Company, 1978. Citado na página 34.

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 2 vezes nas páginas 48 e 49.

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 9 vezes nas páginas 13, 70, 71, 72, 73, 74, 75, 76 e 78.

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 77.

KRSUL, I. V. *Software Vulnerability Analysis*. Tese (Doutorado) — Purdue University, West Lafayette, 1998. Citado na página 39.

LAB, K.; INTERNATIONAL, B. *Global Corporate IT Security Risks: 2013*. 2013. Research. Citado na página 39.

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 2 vezes nas páginas 85 e 86.

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 41.

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 52.

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 23.

LUNA, B. F. Sequência básica na elaboração de protocolos de pesquisa. In: *Arquivos Brasileiros de Cardiologia*. [S.l.: s.n.], 1998. v. 71. ISBN 0066-782X. Citado na página 26.

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 42.

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 35 e 36.

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 13, 63, 64, 66 e 67.

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 22.

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 42.

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 37 e 50.

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 na página 23.

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 63 e 64.

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 6 vezes nas páginas 29, 50, 51, 52, 53 e 81.

MELL, P.; SCARFONE, K.; ROMANOSKY, S. *A Complete Guide to the Common Vulnerability Scoring System*. [S.l.], 2007. Version 2.0. Citado na página 58.

MENESES, V. V.; MEIRELLES, P. R. M. *Evolução Plataforma Mezuro: De Plugin a Aplicação Independente*. [S.l.], 2013. Citado na página 64.

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 21.

MILES, M. B.; HUBERMAN, A. M. *Qualitative Data Analysis: An Expanded Source Book*. 2nd. ed. [S.l.]: Sage Publications Inc, 1994. Citado na página 26.

PáSCOA, J. E. P. *Fatores e Subfatores para Avaliação da Segurança em Software de Sistemas Críticos*. 2002. Citado na página 39.

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 51.

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 45.

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 21.

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 77.

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 na página 23.

SOCIETY, I. C.; COMMITTEE, S. C. *IEEE Standard Computer Dictionary: Compilation of IEEE Standard Computer Glossaries*. [S.l.]: IEEE, 1991. 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 51.

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 51.

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 22 e 44.

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 45 e 46.

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 40.

WESTFALL, L. *12 Steps to Useful Software Metrics*. 2005. Citado na página 79.

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 51.