
UNIVERSIDADE DO ESTADO DO RIO GRANDE DO NORTE
UNIVERSIDADE FEDERAL RURAL DO SEMI-ÁRIDO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO


**Robson Locatelli Macedo**

# Priorização de Anomalias Arquiteturais *versus* Refatoração de Artefatos de Código: Um Estudo envolvendo Sistemas de *Software* em Evolução

**Mossoró-RN**

**2018**

**Robson Locatelli Macedo**

# Priorização de Anomalias Arquiteturais *versus* Refatoração de Artefatos de Código: Um Estudo envolvendo Sistemas de *Software* em Evolução

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação - associação ampla entre a Universidade do Estado do Rio Grande do Norte e a Universidade Federal Rural do Semi-Árido, para a obtenção do título de Mestre em Ciência da Computação.

Orientador: Profº Dr. Francisco Dantas de Medeiros Neto

**Mossoró-RN**

**2018**

**Catalogação da Publicação na Fonte.**
**Universidade do Estado do Rio Grande do Norte.**

M141p Macedo, Robson Locatelli

Priorização de Anomalias Arquiteturais versus Refatoração de Artefatos de Código: Um Estudo envolvendo Sistemas de Software em Evolução. / Robson Locatelli Macedo. - Mossoró, 2018.

96p.

Orientador(a): Prof. Dr. Francisco Dantas de Medeiros Neto.

Dissertação (Mestrado em Programa de Pós-Graduação em Ciência da Computação). Universidade do Estado do Rio Grande do Norte.

1. Anomalias Arquiteturais. 2. Sistemas em Evolução. 3. Refatoração. 4. Escopo. 5. Priorização. I. de Medeiros Neto, Francisco Dantas. II. Universidade do Estado do Rio Grande do Norte. III. Título.

O serviço de Geração Automática de Ficha Catalográfica para Trabalhos de Conclusão de Curso (TCC´s) foi desenvolvido pela Diretoria de Informatização (DINF), sob orientação dos bibliotecários do SIB-UERN, para ser adaptado às necessidades da comunidade acadêmica UERN.

ROBSON LOCATELLI MACEDO

Priorização de Anomalias Arquiteturais versus Refatoração de Artefatos de Código: Um Estudo envolvendo Sistemas de Software em Evolução.

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação para a obtenção do título de Mestre em Ciência da Computação.

APROVADA EM: 28 / 03 / 2018

Prof. Dr. Francisco Dantas de Medeiros Neto
Orientador e Presidente

Prof. Dr. Daniel Sabino Amorim de Araujo
Universidade Federal do Rio Grande do Norte - UFRN

Prof. Dr. Carlos Andre Guerra Fonseca
Universidade do Estado do Rio Grande do Norte

Profa. Dra. Marcia Jacyntha Nunes Rodrigues Lucena
Universidade Federal do Rio Grande do Norte - UFRN

Dedido este trabalho a minha família que nunca mediram esforços para me ajudar, que sempre me apoiaram, sempre estiveram presentes nos momentos mais importantes e que torceram para que este dia chegasse. A todos os momentos de angústia que a disciplina de Projeto e Análise de Algoritmo (PAA), me fez passar (risos).

# Agradecimentos

Deus é bom o tempo todo.

**A**gradeço à Deus, por me proporcionar este momento e por me dar forças para continuar, por cada choro permitido, cada medo enfrentado, cada surpresa e cada conquista. Aos meus pais, senhor Job e dona Maria José, que mesmo estando à mais de 2000 km de distância sempre se fizeram presentes. Seja nas ligações, nas videoconferências ou em pensamento. Todo meu Carinho à eles que durante todo esse tempo, tiveram a paciência e sempre continuaram na torcida para que eu conseguisse alcançar o objetivo que me propus quando deixei meu lugarzinho pacato no interior do Espírito Santo e, fui enfrentar todo o desconhecido no estado do Rio Grande do Norte.

**M**eu sincero obrigado, ao meu orientador Prof. Dr. Francisco Dantas, por todo o suporte durante esses dois anos de caminhada. Principalmente, por todo apoio quando me mudei de Mossoró-RN para Natal-RN. Todo meu carinho, de verdade, por não só ser um orientador, mas também ter se tornado um amigo. Por aconselhar, esclarecer minhas dúvidas e, principalmente, socorrer nos momentos críticos. Obrigado pelo ótimo padrão de pesquisa que me impôs a realizar.

**E**xtremamente grato aos meus grandes parceiros não só de Mestrado, mas agora também de vida - Jefferson, Morais Neto, Paul e Weliton. Obrigado pelas conversas na mesa do "Espetinho do Ivan", pelas noites passadas dentro do laboratório enquanto Mosssoró caia em festa e por toda a ajuda e recepção desde que cheguei na cidade que tem a água mais quente que um cara do Suldeste (eu) conheceu até hoje (Mossoró, RN). Obrigado por serem amigos de verdade. Espero que ainda possamos conservar essa amizade por muitos e muitos anos.

**M**uito obrigado aos meus outros amigos que o Mestrado me proporcionou fazer. Em especial, agradeço a todos que convivi durante esses dois anos, de segunda à sábado (e muitas vezes, até no domingo), no Laboratório de Computação (LABCOMP) - Ademar, Alexandre, Arthur, Felipe Fernandes, Ramiro e Salatiel. Sou imensamente grato àqueles que se tornaram minha família durante esses anos e, espero que todos nós consigamos alcançar nossos objetivos. Sou grato por toda ajuda, conselho e, até mesmo as madrugadas de estudo no laboratório. Aos meus amigos fora do Mestrado, a "galera do curtiço"(Carlos, Lívia e Renata), bem como, Pedro, Vânio, Uriel, Yasmin, Joyce e Jordânia, que me acolheram com total satisfação e me permitiu me sentir em casa. Com certeza, sozinho ninguém chega a lugar algum.

A todo Corpo Docente do Programa de Pós-Graduação em Ciência da Computação, pelos ensinamentos passados. À Professora Ma. Camila de Araújo Sena, por me conceder

a disciplina de Engenharia de Software e permitir que eu lecionasse para os alunos do 7º período do curso superior em Ciência da Computação da UERN - Campus de Natal. **Agradeço** a CAPES, CNPq, Universidade do Estado do Rio Grande do Norte (UERN) e Universidade Federal Rural do Semi-Árido (UFERSA), por me proporcionar essa oportunidade de fazer o Mestrado e por todo o suporte.

**A todos** àqueles que direta ou indiretamente colaboraram para que tudo isso acontecesse da forma que precisava acontecer. Como disse o Prof. Dr. Paulo Grabriel Gadelha durante o meu Mestrado: *"O caminho mais curto é a desistência"*. Posso dizer que lutei e ainda vou lutar, mas com certeza um dia a gente chega lá.

Por último e não menos importante, à mim. **Por acreditar** que seria possível e pela coragem de sair sozinho da zona de conforto e enfrentar o desconhecido em outro estado tão distante. Espero sempre continuar com essa vontade de arriscar, **sempre**.

“A tarefa não é tanto ver aquilo que ninguém viu, mas pensar o que ninguém ainda pensou sobre aquilo que todo mundo vê.”

*Arthur Schopenhauer*

# Resumo

Projetos arquiteturais de *software* tendem a evoluir priorizando o reuso de seus componentes, de modo a promover a evolução dos artefatos de código livre de refatorações não esperadas. No entanto, este reuso nem sempre acontece como esperado, devido principalmente a problemas estruturais dos projetos, oriundos de falta de planejamento e constantes refatorações. Problemas estruturais, em sua maioria, estão relacionados à existência de anomalias arquiteturais e ao escopo crescente de seus componentes. O escopo de um componente arquitetural refere-se ao percentual de dependência que ele possui em relação aos demais componentes do sistema. Infelizmente, existe uma carência de estudos que abordam a priorização de anomalias arquiteturais como forma de minimizar operações indesejadas de refatoração de código. Sabe-se que a ordem de tratamento das anomalias interfere diretamente no número de operações de refatoração. Neste contexto, há uma demanda crescente por estratégias capazes de priorizar o tratamento de anomalias arquiteturais de modo a minimizar o número de operações de refatoração. Este trabalho de pesquisa de mestrado analisou três sistemas de *software* em evolução, com o objetivo de prover suporte a priorização do tratamento de anomalias de modo a interferir na refatoração de elementos de código de sistemas de *software* em evolução. Neste contexto, emergem as contribuições: (i) Definição de mecanismos capazes de relacionar a ocorrência de anomalias arquiteturais com o número de operações de refatoração que acontecem no código, (ii) Desenvolvimento de uma solução computacional e (iii) resultados empíricos. Entendemos que os mecanismos para relação das anomalias arquiteturais com as operações de refatoração no código em (i) e o desenvolvimento de uma solução computacional em (ii), serão úteis para apoiar desenvolvedores e arquitetos na tomada de decisão acerca da priorização no tratamento de anomalias arquiteturais. Para as três aplicações avaliadas, o escopo mostrou ser um forte indicador para priorização no tratamento de anomalias para projetos arquiteturais menos refatorados. Ao priorizar adequadamente o tratamento de anomalias arquiteturais espera-se contribuir para a construção de projetos arquiteturais com menos problemas estruturais.

**Palavras-chave**: Anomalias Arquiteturais, Sistemas em Evolução, Refatoração, Escopo, Priorização.

# Abstract

Software architectural projects tend to evolve prioritizing the reuse of their components in order to avoid code refactoring. However, this reuse does not always happen as expected, mainly due to structural problems of the projects, lack of planning and constant refactorings. In general, structural problems are related to the existence of architectural anomalies and to the increasing scope of its components. The scope of an architectural component refers to the percentage of dependence that it has in relation to the other components of the system. Unfortunately, there is a lack of studies that address the prioritization of architectural anomalies as a way to minimize unwanted code refactoring operations. It is known that anomalies treatment order impacts directly on the number of refactoring operations. In this context, there is a growing demand for strategies capable of prioritizing the treatment of architectural anomalies in order to minimize the number of refactoring operations. This master's research work analyzed three evolving software systems, with the objective of providing support for the prioritization of the handling of anomalies in order to interfere in the refactoring of code elements of evolving software systems. In this context, the contributions emerge: (i) Definition of mechanisms capable of relating the occurrence of architectural anomalies with the number of refactoring operations that occur in the code, (ii) development of a computational solution, and (iii) empirical results. We understand that the mechanisms for relation of architectural anomalies with code refactoring operations in (i) and the development of a computational solution in (ii), will be useful to support developers and architects in the decision making about prioritization in the treatment of anomalies architecture. For the three applications evaluated, the scope proved to be a strong indicator for prioritization in the treatment of anomalies for less refactored architectural projects. When properly prioritizing the treatment of architectural anomalies, it is expected to contribute to the construction of architectural projects with fewer structural problems.

**Keywords**: Architectural Bad Smells, Evolving Systems, Refactoring, Scope, Prioritization.

# Lista de ilustrações

# Lista de tabelas

# Lista de abreviaturas e siglas

| | |
|---|---|
| AI | *Ambiguous Interface* |
| ALP | Arquitetura de Linha de Produto |
| SOA | *Service Oriented Architecture* |
| CCO | *Component Concern Overload* |
| CE | *Connector Envy* |
| EAC | *Extraneous Adjacent Connector* |
| ESBC | Engenharia de *Software* Baseado em Componentes |
| FC | *Feature Concentration* |
| GUI | *Graphical User Interface* |
| HA | Heurística de Aglomeração |
| HE | Heurística de Escopo |
| HV | Heurística de Volatilidade |
| LPS | Linha de Produto de *Software* |
| MVC | *Model-View-Controller* |
| PA | Pacote de Atividades |
| POA | Programação Orientada à Aspecto |
| QGM | *Goal-Question-Metric* |
| QP | Questão de Pesquisa |
| SFP | *Scattered Parasitic Functionality* |
| SGC | Sistema de Gestão de Componentes |
| UML | *Unified Modeling Language* |

# Sumário

# 1 Introdução

Projetos arquiteturais de sistemas de *software* tendem a apresentar problemas estruturais quando evoluem sem o devido planejamento (SHAH, 2008). Boa parte destes problemas estão associados a realização de mudanças, que se concretizam sem o real entendimento da estrutura do projeto e, consequentemente, sem nenhum conhecimento sobre o impacto que essas mudanças podem causar na evolução dos componentes arquiteturais e posteriormente nos elementos de código (SOMMERVILLE *et al.*, 2011). Idealmente falando, as decisões arquiteturais antecedem a evolução do código e devem ser fielmente implementadas. Logo, quanto maior for o planejamento arquitetural menor será o risco de o código evoluir guiado por correções indesejadas.

Porém, a medida que os sistemas de *software* evoluem surge um fator complicador para um bom planejamento arquitetural: o aumento da complexidade da implementação dos requisitos nos diferentes níveis de abstração dos sistemas (LEHMAN, 1996). Quanto mais complexo o requisito se torna, maior é a demanda direta por mudanças, com severas restrições de prazo e custo, levando os projetistas a tomarem decisões indesejadas e/ou precipitadas (SILVA; BALASUBRAMANIAM, 2012). No caso dos projetos arquiteturais, essa complexidade tende a ser agravada a medida que o escopo dos seus componentes aumenta. O escopo de um componente arquitetural refere-se ao percentual de dependência entre ele e os demais componentes da arquitetura. Quanto maior o escopo de um componente, maior a probabilidade desse componente desencadear mudanças indesejadas nos demais componentes. Espera-se que para cada nova versão de um sistema, uma nova versão do projeto arquitetural tenha sido previamente gerada, priorizando o reuso dos componentes já existentes.

É importante também destacar que a falta de planejamento durante a evolução dos sistemas, contribui para geração de projetos arquiteturais infectados por anomalias (HIRAMA, 2012) e artefatos de código constantemente refatorados (SAMARTHYAM; SURYANARAYANA; SHARMA, 2016). Refatoração refere-se à reestruturação dos artefatos existentes, sem necessariamente promover a modificação do seu comportamento externo (FOWLER, 1999). Anomalias arquiteturais refere-se aos sintomas ocasionados por um conjunto de más decisões que impactam negativamente no ciclo de vida dos projetos arquiteturais (MACIA *et al.*, 2012; GARCIA *et al.*, 2009b).

Um projeto arquitetural anômalo prejudica todo o processo de evolução do *software* e tende a impactar negativamente na evolução dos artefatos do código, demandando recorrentes operações de refatoração. Estudos preliminares têm analisado e proposto técnicas para detecção (ARCOVERDE *et al.*, 2012; MACIA *et al.*, 2013; PADILHA,

2013) e priorização de anomalias de código (ARCOVERDE *et al.*, 2013; GUIMARÃES, 2014). Andrade (2013a), Andrade (2013b), Andrade, Almeida e Crnkovic (2014), Kaur e Kumar (2014), Nascimento, Fonseca e Dantas (2017) têm investigado as anomalias arquiteturais em sistemas de *software* em evolução e seus impactos no reuso. No trabalho de Nascimento, Fonseca e Dantas (2017), foram analisados artefatos arquiteturais de dois sistemas de *software* em evolução, a fim de detectar anomalias arquiteturais por meio de uma ferramenta especialista denominada *ArchiDES* e realiza-se uma análise do impacto de tais anomalias detectadas, na reutilização de componentes arquiteturais dos sistemas analisados. No entanto, nenhum trabalho investiga a relação existente entre anomalias arquiteturais e operações de refatoração do código. Também não há estratégias que possam ser utilizadas no sentido de orientar, dentro de um conjunto de anomalias, quais devem ser tratadas de forma prioritária como forma de minimizar as operações de refatoração do código em evolução, levando em consideração, por exemplo, o escopo de seus componentes.

Neste contexto, temos que lidar com dois problemas inerentes à evolução de sistemas de *software*: projetos arquiteturais anômalos e código constantemente refatorado. Acredita-se que os projetos arquiteturais evoluem infectados por anomalias devido ao fato de que na comunidade de engenharia de *software*, a priorização para tratamento de tais anomalias tem sido pouco explorada (ARCOVERDE *et al.*, 2013; GUIMARAES *et al.*, 2013). Tal problema tem levado os desenvolvedores a realizar inúmeras refatorações de forma desnecessária (ARCOVERDE *et al.*, 2013). Resultando assim, em um tempo maior para a manutenção do projeto. Idealmente falando, priorizar o tratamento de anomalias em níveis mais altos de abstração (*i.e.*, projeto arquitetural) tende a impactar positivamente na redução do número de operações de refatoração nos artefatos de código.

No contexto de um curto espaço de tempo para finalização dos projetos e diante das necessidades de reestruturações da arquitetura do software, uma vez detectadas, os projetistas precisam de apoio para priorizar o tratamento das anomalias de modo a minimizar operações de refatoração no código dos sistemas de *software*. Contudo, apesar de os projetistas fazerem uso de técnicas e ferramentas de detecção de anomalias arquiteturais, somente usando essas estratégias não é possível saber quais anomalias precisam ser tratadas primeiro, com o objetivo de combater as operações de refatorações indesejáveis, visto que, uma vez detectadas, a sequência de tratamento das anomalias, um que muitas vezes demanda refatorações, fica a cargo do projetista arquitetural.

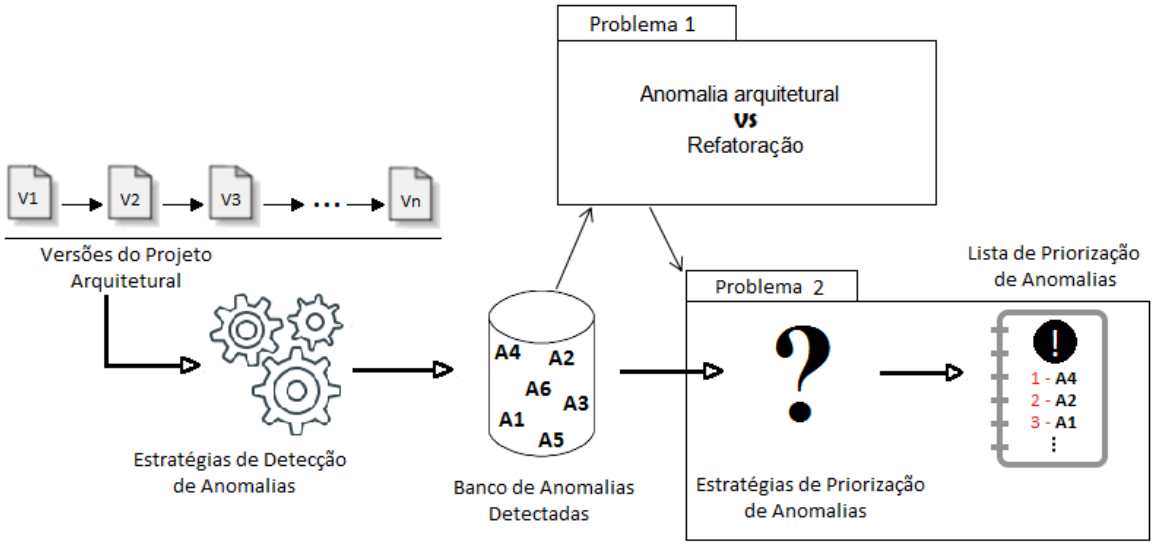
## 1.1 Problemática

Nesse contexto, emerge uma problemática associada à falta de estudos voltados para a priorização de anomalias arquiteturais, tendo como fio condutor promover a redução de operações de refatoração no código dos sistemas. Estudos anteriores (GARCIA *et*

*al.*, 2009a; VALE *et al.*, 2014), indicam que diferentes anomalias promovem diferentes impactos na evolução de sistemas. Idealmente falando, uma vez detectadas, a ordem de tratamento das anomalias deve ser guiada pelo impacto negativo que tais anomalias podem causar na evolução dos artefatos de código, com base no poder de espalhamento de tais anomalias na arquitetura. Quanto maior for o potencial de uma anomalia para demandar mudanças no código de um sistema de *software* em evolução, maior será a prioridade de tratamento que esta anomalia terá. Porém, com base na literatura, as abordagens de priorização de anomalias existentes, concentram-se na priorização de anomalias no nível de código (ARCOVERDE *et al.*, 2013; GUIMARAES *et al.*, 2013), tirando do desenvolvedor a possibilidade de antecipar o tratamento desta anomalia ainda no nível arquitetural. Atualmente, a ordem de tratamento das anomalias arquiteturais detectadas é realizada com base no expertise dos desenvolvedores, uma vez que não há nenhuma estratégia que indique a prioridade de tratamento de anomalias.

Podemos dividir a problemática de priorizar o tratamento de anomalias em dois problemas, ambos dependentes de um banco de projetos arquiteturais anômalos. A Figura 1 apresenta uma visão geral de tal problemática. Como ilustrado, estratégias de detecção de anomalias já existentes, apoiadas pelos estudos de Andrade (2013b), Andrade, Almeida e Crnkovic (2014), Kaur e Kumar (2014), Nascimento, Fonseca e Dantas (2017), são alimentadas por várias versões de projetos arquiteturais de *software* em evolução, criando um banco contendo os componentes arquiteturais anômalos. Como podemos observar, a detecção de anomalias levanta dois problemas. O primeiro problema (ver Figura 1 - box Problema 1) consiste em identificar se existe relação entre anomalias arquiteturais e refatoração dos elementos de código. Este primeiro problema aborda, entre outros pontos, a investigação da relação do escopo dos componentes arquiteturais com as operações de refatoração. O segundo problema (ver Figura 1 - box Problema 2) consiste em preencher a lacuna da priorização de anomalias arquiteturais, fornecendo meios para que os projetistas possam priorizar o tratamento de anomalias adequadamente, minimizando a refatoração de elementos de código ao longo de sua evolução.

Figura 1 – Visão Geral da Problemática




**Fonte**: Autor.

Resolver os problemas anteriormente mencionados significa atender um demanda dos desenvolvedores de sistemas de software em evolução, no que diz respeito ao gerenciamento de operações indesejadas de refatoração. Considerando que os clientes de *software*, que estão cada dia mais exigentes, aliados a velocidade assustadora com que as operações de refatoração no código acontecem, podem fazer com que a equipe se perca em um mar de solicitações de mudança, novas funcionalidades e trechos de código a serem corrigidos. Muitas dessas correções são necessárias devido a ausência de um tratamento prioritário das anomalias arquiteturais.

## 1.2 Objetivos e Questão de Pesquisa

O objetivo deste trabalho é prover suporte aos projetistas arquiteturais e desenvolvedores de sistemas, quanto à priorização do tratamento de anomalias arquiteturais, durante o processo de evolução da arquitetura.

### 1.2.1 Objetivos Específicos

Os objetivos específicos descrevem as metas a serem alcançadas no decorrer da condução da pesquisa. Para realização da pesquisa são previstos os seguintes objetivos específicos:

1. **Investigar por meio de estudos empíricos o relacionamento entre os elementos de código e seus artefatos arquiteturais correspondentes.** Tem-se por objetivo a rastreabilidade entre os artefatos de *software* de cada versão do projeto arquitetural.

2. **Investigar empiricamente como as anomalias arquiteturais podem afetar o processo de refatoração.** Tem-se por objetivo analisar se determinadas categorias de anomalia arquitetural tendem a causar uma demanda maior no número de refatorações em elementos de código. Em particular, será realizado um estudo sobre o impacto do escopo dos elementos arquiteturais na ocorrência de anomalias.

3. **Investigar técnicas para modelagem de heurísticas de priorização.** Tal investigação tem por objetivo traçar um modelo de desenvolvimento para heurísticas de priorização.

4. **Analisar empiricamente uma ou mais heurísticas de priorização investigadas no ponto 3**. Tal estudo objetiva-se analisar a eficácia da aplicação das heurísticas de priorização desenvolvidas neste trabalho.

5. **Divulgar os resultados obtidos.** Tal divulgação tem por objetivo contribuir para com a comunidade acadêmica, publicando e ou apresentando artigos científicos em conferências, revistas e *journals* na área de Ciência da Computação.

Ao vencer os objetivos descritos, teremos condições de responder à questão principal de pesquisa de nosso trabalho que é: "a priorização de anomalias arquiteturais contribui para minimizar o número de operações de refatoração dos elementos de código de sistemas de *software* em evolução?"

## 1.3 Contribuições do Trabalho

O trabalho entrega três contribuições principais, que incluem: (i) definição de mecanismos capazes de relacionar a ocorrência de anomalias arquiteturais com o número de operações de refatoração que acontecem no código, (ii) desenvolvimento de uma solução computacional para capaz de fornecer a correta priorização das anomalias arquiteturais, tendo como foco a redução de operações de refatoração no nível de código e (iii) o resultado de estudos empíricos que fornecem dados para estabelecer como anomalias arquiteturais e operações de código se relacionam.

- **Definição de mecanismos capazes de relacionar a ocorrência de anomalias arquiteturais com o número de operações de refatoração que acontecem no código**. Foi proposta uma heurística com base no escopo dos componentes arquiteturais infectados como forma de associar as anomalias arquiteturais com o processo de refatoração dos elementos de código. Tais heurísticas respondem parcialmente à questão de pesquisa deste trabalho levantadas na Seção 1.2.

- **Desenvolvimento de uma solução computacional**. Para otimizar o processo de investigar a relação entre anomalias arquiteturais e refatoração dos elementos de

código foi desenvolvido um programa capaz que, com base nos mecanismos entregues pela contribuição 1, fornece uma lista de prioridades para o tratamento de anomalias.

- **Resultado empírico**. Foi realizado um estudo exploratório com o objetivo de avaliar a correlação existente entre anomalias arquiteturais e refatoração de código. A investigação foi conduzida usando três sistemas diferentes, o *MobileMedia, Notepad* e *HealthWatcher*. Este resultado responde a nossa questão de pesquisa apresentada na Seção 1.2.

## 1.4 Estrutura do Documento

Este documento está dividido em 4 capítulos além dessa introdução. No Capítulo 2, são apresentados o referencial teórico que serviu de base para a condução deste trabalho. No Capítulo 3, apresentamos estudos preliminares que relacionam a ocorrência de anomalias com operações de refatoração dos elementos de código. No Capítulo 4, apresentamos a solução computacional e a discussão dos resultados. Por fim, apresenta-se a conclusão deste estudo no Capítulo 5.

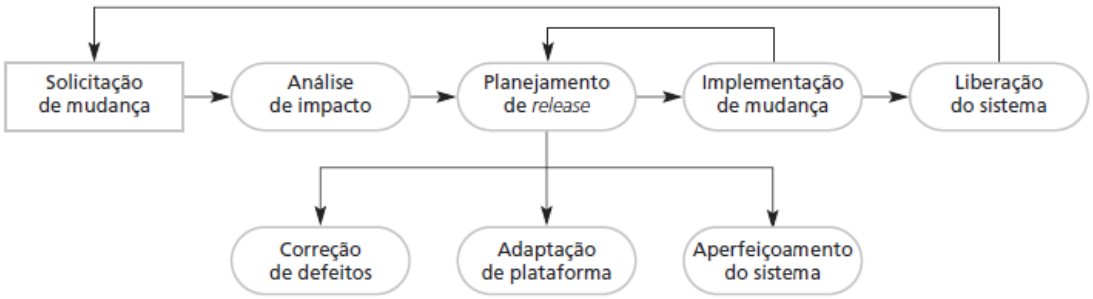
# 2 Referencial Teórico e Trabalhos Correlatos

Nesse capítulo será apresentada a teoria base para o desenvolvimento deste trabalho de pesquisa, permeada pela discussão dos trabalho relacionados. Na Seção 2.1 daremos uma visão geral sobre evolução de sistemas. Os conceitos e os trabalhos relacionados a anomalias arquiteturais serão apresentados na Seção 2.2. Na sequência, falaremos sobre detecção de anomalias arquiteturais na Seção 2.3. Em seguida, na Seção 2.4 falaremos sobre o estado da arte na priorização de anomalias detectadas. Posteriormente, na Seção 2.5 trataremos da refatoração arquitetural. Na Seção 2.6 falaremos sobre métricas de *software*, mecanismos essenciais para mensurar os resultados da pesquisa. Em seguida, daremos uma visão geral dos estudos relacionados a acomplamento arquitetural na Seção 2.7. Na Seção 2.8 apresentaremos os estudos relacionados a técnica de rastreamento (chamado aqui de *treacebility*). Finalmente, na Seção 2.9 resumimos a discussão do capítulo.

## 2.1 Evolução de Sistemas de Software

Segundo Samarthyam, Suryanarayana e Sharma (2016) a evolução de sistemas pode ser comparada a evolução de uma cidade. Com base nesta metáfora, no momento em que uma cidade não consegue atender às necessidades da população que nela vive, cria-se uma atmosfera de crise e, por fim, o seu abandono. Nesse contexto de evolução, para que o sistema possa evoluir de forma adequada, existem modelos de processos de *software* que auxiliam desde a solicitação e proposta de mudança até a implantação e comunicação da mudança do *software* e então liberação do sistema, tornando o processo de evolução menos passivo de erros. A Figura 2 extraída de Sommerville *et al.* (2011), demonstra um popular modelo de evolução. A etapa de solicitação de mudança, ocorre por parte dos clientes ou desenvolvedores. Na etapa de análise de impacto, avalia-se os impactos que podem causar na estrutura do *software*. Na etapa de planejamento de *release*, compreende-se as mudanças que foram solicitadas pelo cliente ou desenvolvedores. Para isso, é levado em consideração todas as mudanças propostas (correção de erros, adaptação da plataforma e novas funcionalidades). Na etapa de implementação de mudanças, os novos requisitos do sistema são propostos, analisados e então validados, para que sejam implementas as mudanças solicitadas. Finalmente na etapa de liberação do sistema, uma nova versão do sistema será liberada para uso no ambiente.

Figura 2 – Processo de Evolução de Software




**Fonte**:(SOMMERVILLE *et al.*, 2011).

Sistemas de *software* tendem a evoluir, a partir do momento que são postos em utilização. Independente do modelo, a evolução de seus artefatos caracteriza-se por sofrer mudanças sucessivas para satisfazer um determinado conjunto de requisitos (RAJLICH; SILVA, 1996). Por meio do processo evolutivo, o software se mantém útil e continua operando no ambiente. Idealmente, a arquitetura de sistemas *software* devem evoluir alinhadas com a evolução dos demais artefatos. Arquitetura de *software* refere-se a estruturas de alto nível, que descrevem como um sistema é organizado e como os componentes interoperam, modeladas no geral, por meio de diagramas. Infelizmente, há alguns fatores complicadores que podem, a princípio, dificultar a evolução arquitetural. Dentre estes fatores podemos destacar a dificuldade no devido planejamento arquitetural. Tal dificuldade pode ser resultado da necessidade de mercado para manter-se útil ou a necessidade de modificações no *software* para a correção de *bugs* detectados, bem como, para melhoraria de seu desempenho (SOMMERVILLE *et al.*, 2011). Assim, sistemas de *software* que entram em contínuo crescimento, possuem uma arquitetura cada vez mais complexa (LEHMAN, 1996), exigindo então, um aumento no custo de manutenção, em particular no que diz respeito as operações de refatoração (JAZAYERI, 2002).

Para Chaki *et al.* (2009), planejar a evolução da arquitetura é fator fundamental para orientar e planejar a evolução do *software*. As propriedades de um sistema, como disponibilidade e volatilidade (*i.e.*, resistência a mudanças), são relacionadas com a arquitetura adotada (SOMMERVILLE *et al.*, 2011). Entretanto, os requisitos que uma arquitetura deve cumprir mudam bastante ao longo do tempo (KÄKÖLÄ; DUEÑAS, 2006). Ao mesmo tempo, organizações buscam na arquitetura de software uma maneira de controlar os custos de desenvolvimento e principalmente evolução de *software*.

A evolução de sistemas de *software* tem sido estudada em diferentes níveis de abstração. Alguns estudos tem analisado o processo de evolução de sistemas (THOMAS *et al.*, 2014; NOVAIS; SANTOS; MENDONÇA, 2017). No trabalho de Thomas *et al.* (2014) são avaliados os modelos de tópicos[1] estatísticos na análise da evolução do *software*,

[1] **Modelos de Tópicos:** No contexto do trabalho dos autores, os tópicos são conjuntos de palavras que

onde é feito uma análise manual sobre o histórico do código-fonte de dois sistemas bem documentados. Os autores sugerem que usar modelos de tópicos para avaliar a evolução do código-fonte de um projeto de software pode ser útil para os desenvolvedores e demais *stakeholders* do projeto, proporcionando uma compreensão do histórico do sistema.

No trabalho de Novais, Santos e Mendonça (2017) foi realizado um estudo experimental para validar os benefícios da combinação de estratégias diferenciais[2] e temporais[3], objetivando encontrar evidências empíricas sobre o uso das múltiplas estratégias combinadas para a compreensão da evolução do software. Os autores enfatizam a importância de diferentes estratégias visuais durante a visualização da evolução do software, onde estando o usuário com a tarefa em mãos, poderá decidir qual usar.

De forma mais específica, alguns estudos tem analisado a evolução arquitetural com diferentes objetivos (MACIA *et al.*, 2012; BARNES, 2013; JAZAYERI, 2002; SILVA; KULESZA, 2016). No trabalho de Silva e Kulesza (2016), os autores aplicam técnicas de Visualização de *Software* para acompanhar a evolução arquitetural de um *software* com foco no atributo de qualidade desempenho. Visualização de *Software* é o uso de representações visuais para melhorar o entendimento e compreensão dos diferentes aspectos de um sistema de *software* (CARPENDALE; GHANAM, 2008). Diante disso, os autores propuseram um framework desenvolvido em Java para realizar análises estáticas e dinâmicas baseadas em cenários de caso de uso.

No trabalho de Jazayeri (2002), o autor propõe uma análise retrospectiva com o objetivo de avaliar a estabilidade arquitetural, examinando a quantidade de mudança aplicada em sucessivos lançamentos de um produto de *software*, sugerindo estabilidade ou resiliência como um critério primário para avaliar uma arquitetura. Em outro contexto de evolução, alguns estudos tem analisado a evolução arquitetural em sistemas com várias versões (MACIA *et al.*, 2011; MACIA *et al.*, 2012) onde uma nova funcionalidade vai sendo adicionada a cada nova versão, e no contexto de Linha de Produto de *Software* (LPS) (ANDRADE, 2013a; ANDRADE, 2013b; VALE *et al.*, 2014) onde a evolução tende acontecer naturalmente. Linha de produto de *Software* caracteriza-se por um trabalho sobre um grupo de sistemas compartilhando funcionalidades em comum, que satisfazem as necessidades específicas de um dado segmento, e que são desenvolvidos a partir de um aglomerado comum de artefatos base e de forma previamente planejada (SILVA *et al.*, 2011).

No trabalho de Macia *et al.* (2012), os autores apresentam um *plug-in* como

ocorrem com frequência nos documentos e geralmente têm uma clara relação semântica (THOMAS *et al.*, 2014).

[2] **Estratégias Diferenciais:** Levam em consideração, para análise, apenas duas versões do sistema em um dado momento. Na representação visual, mostra os módulos de uma versão em comparação com outro (NOVAIS; SANTOS; MENDONÇA, 2017).

[3] **Estratégias Temporais:** Representam a evolução considerando, muitas vezes, todas as versões do *software* disponíveis para análise (NOVAIS; SANTOS; MENDONÇA, 2017).

ferramenta para detecção Anomalias de Código arquiteturalmente relevantes. Para tal, faz-se necessário a informação de arquitetura, em vez de análise estática somente, e análise de grupos de anomalias de código para identificar padrões de ocorrências. Anomalia de código são estruturas de código recorrentes que possivelmente indicam um problema de manutenção mais profundo (ARCOVERDE *et al.*, 2012), ou seja, possuem a mesma definição aplicada a anomalias arquiteturais, entretanto, estão relacionadas a más decisões ao nível do código (GARCIA *et al.*, 2009a)

No trabalho de Vale *et al.* (2014) é apresentado uma revisão sistemática para encontrar e classificar trabalhos publicados sobre anomalias em LPS e seus respectivos métodos de refatoração. Nesse estudo, foram identificadas 70 anomalias e 95 estratégias de refatoração. A Seção 2.2 apresenta exemplos de algumas anomalias arquiteturais já catalogadas.

## 2.2 Anomalias Arquiteturais

O termo 'anomalia arquitetural' surgiu pela primeira vez no trabalho de LIPPERT M.; ROOCK (2006), e refere-se aos sintomas que emergem no projeto arquitetural de um sistema de *software*, geralmente indicando um problema estrutural profundo no projeto (GARCIA *et al.*, 2009a). Segundo LIPPERT M.; ROOCK (2006) anomalias arquiteturais podem ser encontradas em vários e/ou diferentes níveis de um projeto. Dentre eles:

- Em usos e relações de herança entre classes;
- Entre pacotes;
- Entre subsistemas (Módulos);
- Entre Camadas;

No trabalho de Garcia *et al.* (2009b) são destacados os fenômenos que podem contribuir para o surgimento de anomalias. Nossa proposta de pesquisa baseia-se no segundo e terceiros fenômenos. São eles:

1. Aplicação de uma solução de projeto em um contexto inadequado;
2. Mistura de combinações de abstrações de projeto que têm comportamentos indesejáveis;
3. Aplicação de abstrações de projeto no nível errado de granularidade;

A definição de anomalias arquiteturais é complementada no trabalho de Macia *et al.* (2011), relacionando-as com problemas de modularidades que podem apresentar

decomposição arquitetural por meio de decisões de *design* inapropriadas. Vendo a partir desta perspectiva, as anomalias arquiteturais alteram a estrutura interna e o comportamento dos elementos do sistema, tais como componentes, conectores, interfaces e configurações, porém não alteram o comportamento externo do mesmo (GARCIA *et al.*, 2009b).

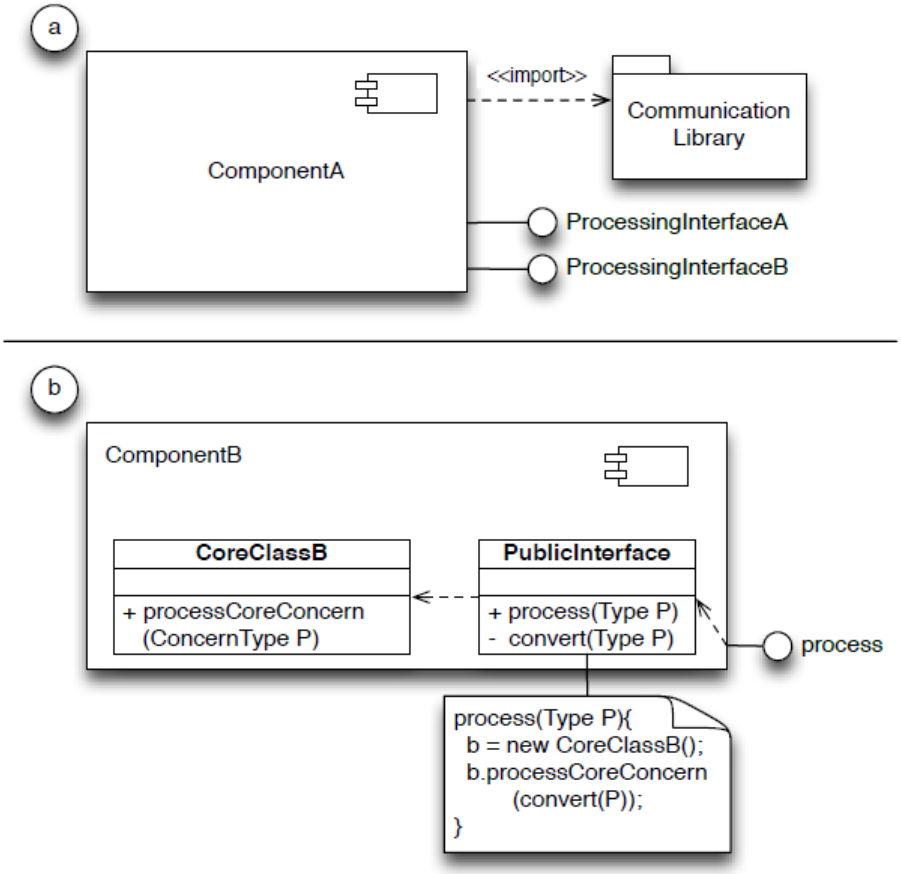
O trabalho de Garcia *et al.* (2009b) identificou e propôs um catálogo de anomalias arquiteturais, que inclui: *Connector Envy* (CE), *Scattered Parasitic Functionality* (SFP), *Ambiguous Interface* (AI), *Extraneous Adjacent Connector* (EAC), *Component Concern Overload* (CCO). As anomalias arquiteturais mencionadas acima, podem ocorrer em sistemas de *software* convencionais e em LPS. Baseado nesse estudo, Andrade (2013a) identificou a anomalia *Feature Concentration* (FC) porém com uma diferença das outras, pois é específica de LPS. Tais anomalias são descritas a seguir.

*Connector Envy:* ocorre quando um componente faz uso excessivo de algumas das quatro categorias de conectores: comunicação, coordenação, conversão e facilitação. Na Figura 3 extraída de Garcia *et al.* (2009a) é apresentado dois exemplos distintos dessa anomalia. O componente `ComponentA` implementa serviços de comunicação e facilitação. `ComponentA` importa uma biblioteca de comunicação, o que implica que ele gerencia as facilidades de rede de baixo nível usadas para implementar a comunicação remota. Os serviços de nomeação, entrega e roteamento manipulados por comunicação remota são um tipo de serviço de facilitação. O componente `ComponentB` executa uma conversão como parte do processamento do componente. A interface do `ComponentB` chamado *process* é implementada pela classe *PublicInterface* de `ComponentB`. *PublicInterface* implementa seu método de processo chamando um método de conversão que transforma um parâmetro do tipo *Type* em um *ConcernType* (GARCIA *et al.*, 2009b). Esta anomalia gera dependência entre os serviços de aplicação e interação e essa interdependência do conector com a funcionalidade do componente, faz com que seja reduzido a reusabilidade, compreensibilidade e testabilidade do projeto arquitetural.

*Scattered Parasitic Functionality:* Essa anomalia se caracteriza por vários componentes serem utilizados para realizar uma mesma interesse de alto nível, que implica na adição de interesses ortogonais. Em outras palavras, ocorre quando interesses são espalhados entre vários componentes, infectando um componente com outra preocupação ortogonal, semelhante a um parasita (GARCIA *et al.*, 2009b). Esta anomalia viola o princípio de separação de interesses, porque o interesse é espalhado através de múltiplos componentes.

Na Figura 4 extraída de Garcia *et al.* (2009a) é apresentado três componentes que são responsáveis pela mesmo interesse de alto nível chamado *SharedConcern*, enquanto `ComponentB` e `ComponentC` são responsáveis por interesses ortogonais, compartilhando um interesse em comum, o interesse *SharedConcern*. `ComponentB` e `ComponentC` violam o princípio de separação de interesse, uma vez que ambos são responsáveis por interesses ortogonais junto aos interesses específicos (GARCIA *et al.*, 2009b), impactando na

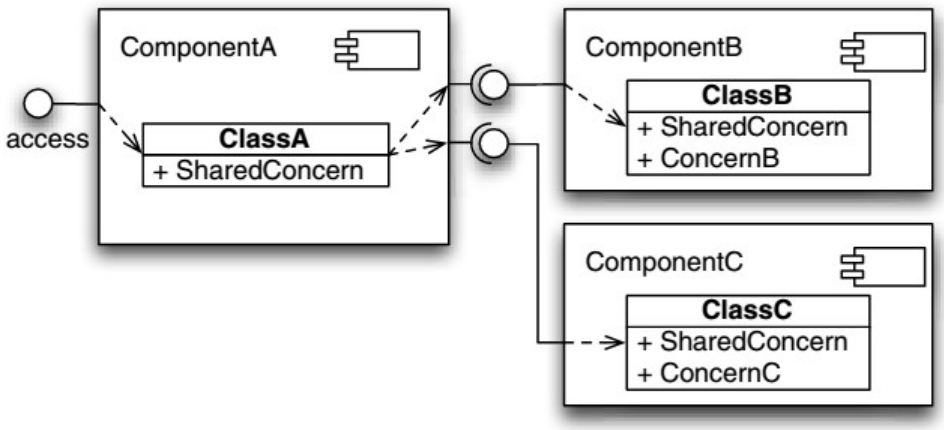
Figura 3 – Anomalia Arquitetural *Connector Envy*




**Fonte**:(GARCIA *et al.*, 2009a).

modificabilidade, compreensibilidade, testabilidade e reusabilidade.

Figura 4 – Anomalia Arquitetural *Scattered Parasitic Functionality*



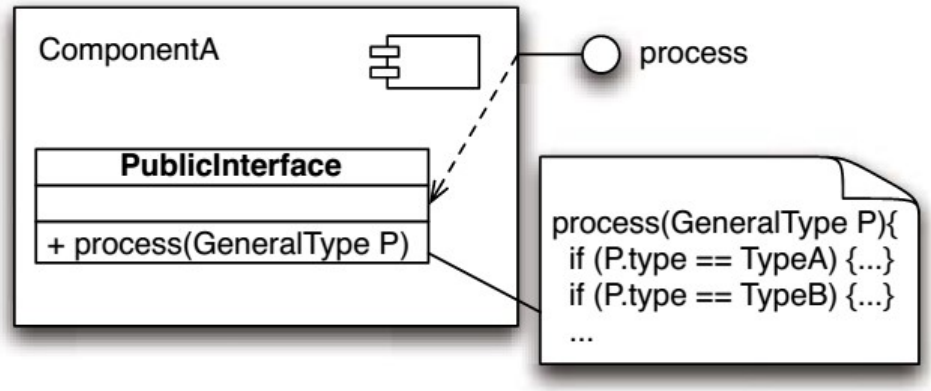

**Fonte**:(GARCIA *et al.*, 2009a).

*Ambiguous Interfaces:* ocorre quando o interfaces são usadas como único ponto genérico de entrada para o componente, por onde oferece e processa vários serviços. Por

consequência, o parâmetro de entrada de dados é genérico, sendo interpretado internamente pelo componente para escolher o serviço a ser realizado.

Na Figura 5 extraída de Garcia *et al.* (2009a), a interface `process` oferece apenas um serviço ou método público, por onde oferece e processa múltiplos serviços. O componente aceita todos os dados através do único ponto de entrada e envia internamente para outros serviços. Pelo parâmetro de entrada ser genérico, o componente que implementa esta interface tratando mais tipos de parâmetros do que irá processar (GARCIA *et al.*, 2009b). Essa anomalias impacta em no sentido de reduzir a capacidade de análise e a compreensibilidade do sistema.

Figura 5 – Anomalia Arquitetural *Ambiguous Interfaces*



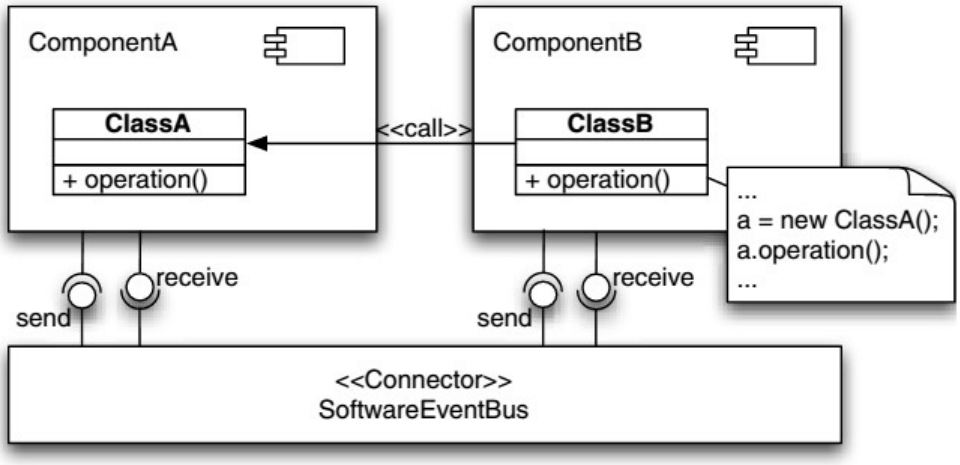

**Fonte**:(GARCIA *et al.*, 2009a).

*Extraneous Adjacent Connector:* caracteriza-se por conectores de tipos distintos serem utilizados para unir um par de componentes. Na Figura 6 extraída de Garcia *et al.* (2009a), apresenta a anomalia onde, dois componentes (`ComponentA, ComponentB`) que se comunicam usando uma combinação onde, o conector baseado em evento (`SotwareEventBus`) é acompanhado por um conector de chamada de procedimento. Esta anomalia pode afetar propriedades do ciclo de vida como compreensibilidade (GARCIA *et al.*, 2009b).

*Component Concern Overload:* ocorre quando componentes são sobrecarregados com dois ou mais interesses arquiteturais. A Figura 7 extraída de Macia *et al.* (2011), ilustra um exemplo onde podemos notar que `Concurrency_Control`, `Persistence_Manager` e `Distribution_Manager` são responsáveis pela realização de múltiplos interesses arquiteturais, onde, `Concurrency_Control` está sobrecarregado por 4 interesses distintos (persistência, transação, tratamento de exceção e concorrência) e `Persistence_Manager` e `Distribution_Manager` aparecem sobrecarregados com 2 interesses distintos (persistência e distribuição) e (persistência e tratamento de exceção), respectivamente. Por consequência, violam o princípio da separação de interesses no projeto arquitetural.

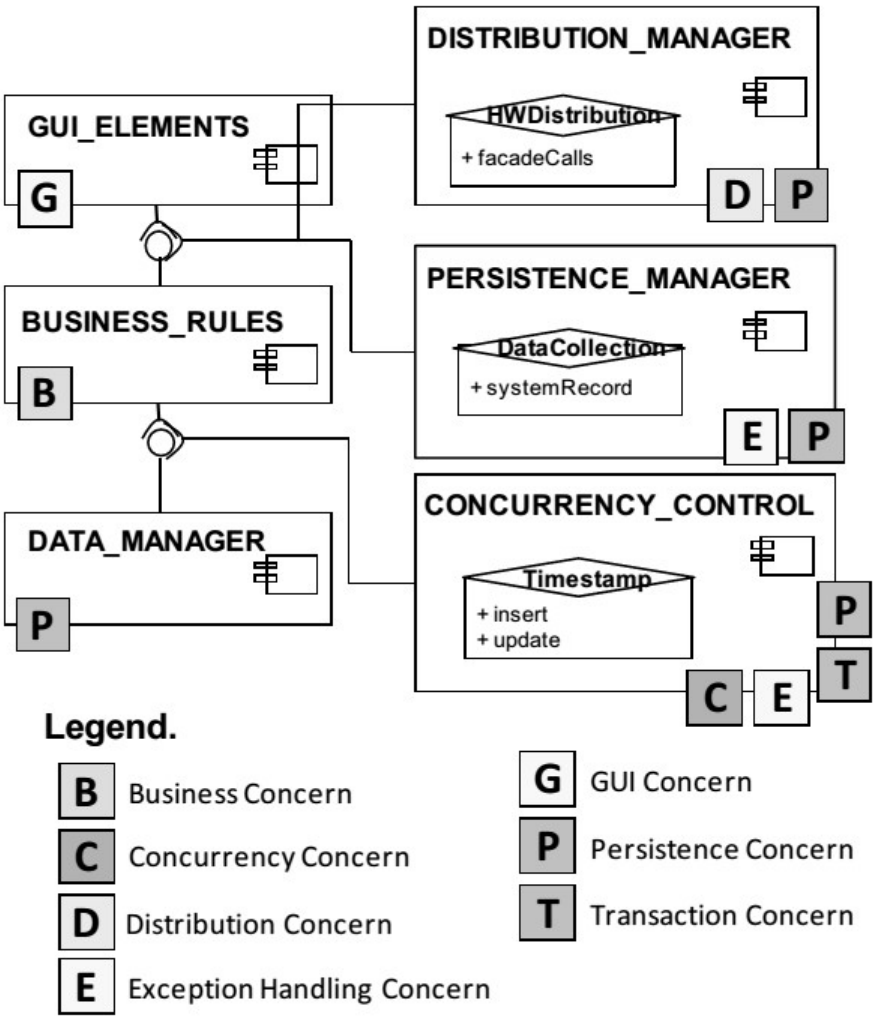
*Feature Concentration:* essa anomalia se destaca por ser específica de Linha de Produto de *Software* (LPS), a mesma se caracteriza pela concentração de *features* em um

Figura 6 – Anomalia Arquitetural *Extraneous Adjacent Connector*




**Fonte**:(GARCIA *et al.*, 2009a).

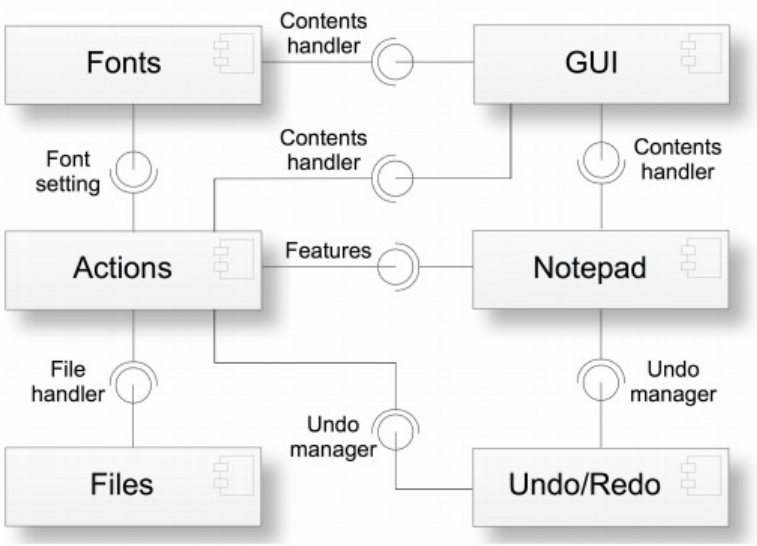
Figura 7 – Anomalia Arquitetural *Component Concern Overload*




**Fonte**:(MACIA *et al.*, 2011).

componente arquitetural. Entende-se por *feature* uma unidade de funcionalidade de um sistema de software que satisfaz um requisito e representa uma decisão de design (APEL; KÄSTNER, 2009). Pode está relacionada com a anomalia *Ambiguous Interface*, porém, ela se opõe da *Scarated Parasitic Functionality* (ANDRADE, 2013b). Na Figura 8 extraída de Andrade, Almeida e Crnkovic (2014), é apresentado o modelo de componente de um bloco de notas denominado Notepad LPS. Como pode ser observado, os recursos em componente são concentrados por meio de um único ponto de entrada genérico no componente *Actions*. Segundo Andrade, Almeida e Crnkovic (2014), concentração de várias *features* podem impactar na compreensibilidade e manutenibilidade da Arquitetura de Linha de Produtos (ALP) ao longo de seu ciclo de vida. Entende-se por ALP como base da arquitetura de cada produto gerado em uma LPS (DURSCKI *et al.*, 2004).

Figura 8 – Anomalia Arquitetural *Feature Concentration*




**Fonte**:(ANDRADE; ALMEIDA; CRNKOVIC, 2014).

Alguns estudos tem analisado o impacto das anomalias na evolução de sistemas de *software* (ARCOVERDE; GARCIA; FIGUEIREDO, 2011; MACIA *et al.*, 2011; MACIA *et al.*, 2012). Enquanto Arcoverde, Garcia e Figueiredo (2011) investigam os efeitos da presença de anomalias sobre o código fonte, Macia *et al.* (2011) investiga o impacto que anomalias tendem a causar na modularidade arquitetural. Já no trabalho de Macia *et al.* (2012), é investigado qual a correlação entre as anomalias de código detectadas automaticamente e os problemas que ocorrem com a arquitetura de um sistema em evolução.

Segundo Garcia *et al* (GARCIA *et al.*, 2009a) e Vale *et al* (VALE *et al.*, 2014), diferentes anomalias arquiteturais podem impactar de diferentes maneiras em projetos arquiteturais. As anomalias arquiteturais tem um impacto negativo no projeto arquitetural pois podem afetar propriedades que acarretam danos no ciclo de vida de um sistema de

*software*, como testabilidade e reusabilidade. Esses danos prejudicam a evolução do *software* e impedem que projetos arquiteturais possam dar continuidade no reuso, fazendo com que o desempenho e confiabilidade do software seja afetado negativamente, (ANDRADE, 2013a).

## 2.3 Detecção de Anomalias Arquiteturais

A literatura apresenta abordagens para identificação de algumas anomalias, dentre elas, anomalias arquiteturais e anomalias de código arquiteturalmente relevantes. Contudo, Ferreira (2014) salienta que algumas técnicas de identificação são limitadas no que diz respeito aos auxílios para arquitetos de *software* na identificação de problemas na arquitetura do *software*. No decorrer desta seção, serão descritas as abordagens atuais para identificação de anomalias arquiteturais e anomalias de código com relevância arquitetural.

No estado da arte existem diferentes abordagens que permitem que anomalias sejam identificadas: (i) manualmente (ANDRADE, 2013b), fazendo inspeções de artefatos arquiteturais e (ii) automaticamente (NASCIMENTO; FONSECA; DANTAS, 2017), utilizando-se de ferramentas como, sistemas especialistas e entre outros. Em relação as abordagens manuais, segundo Melo (2009), desenvolvedores enfrentam um problema em que, apesar de, eles escreverem documentos de requisitos, código e projeto, eles não aprendem a fazer uma leitura adequada de tais documentos. Apesar da inspeção de *software* ser uma abordagem comumente usada para detecção de anomalias, a aplicação dessa abordagem um nível de esforço elevado (FERREIRA, 2014). Por outro lado, as abordagens automatizadas tendem a apoiar os arquitetos a compreenderem melhor o processo e diminuir o esforço, além de aumentar sua eficácia na atividade de detecção.

Buscando a comparação desses dois tipos de abordagens para detecção de anomalias, Ferreira *et al.* (2014) conduziu um estudo de caso, onde comparou a eficácia e o esforço de estratégias baseada em métricas e uso de inspeções de código ad-hoc[4], para detecção de anomalias de código relevantes para a arquitetura. Enquanto Arcoverde *et al.* (2012) apresentam em seu trabalho um sistema de recomendação e classificação de anomalias de código relevantes para a arquitetura. Os autores salientam que o sistema tem como foco a detecção de padrões de anomalias de código com maior precisão apoiando a priorização para tratamento das anomalias detectadas.

Outros estudos (MACIA *et al.*, 2012; NASCIMENTO; FONSECA; DANTAS, 2017) têm proposto ferramentas em diferentes cenários, porém, com o mesmo propósito final, a automatização na detecção de anomalias. No trabalho de Macia *et al.* (2012) é apresentado a ferramenta *SCOOP*, que é um *plugin* para o Eclipse que suporta a definição de estratégias

[4] **Ad-Hoc:** Técnica de revisão baseada na leitura e entendimento de algum documento, tendo como objetivo encontrar defeitos.

baseadas em métricas. Por outro lado, no trabalho de Nascimento, Fonseca e Dantas (2017) é apresentado a ferramenta *ArchiDES*, que é um *plugin* para a IDE Palladio Bench (KIT; FZI;UNIVERSITY, 2016) uma ferramenta para modelagem arquitetural, onde a *ArchiDES* suporta a deteção de anomalias arquiteturais analisando diretamente artefatos arquiteturais. A seguir, essas duas ferramentas são apresentadas mais detalhadamente:

- **SCOOP** (MACIA *et al.*, 2012). Uma ferramenta que apoia o processo de detecção de anomalias de código arquiteturalmente relevantes. Segundo Macia *et al.* (2012), o processo de detecção de anomalia do SCOOP baseia-se em compreender dois tipos de informações: (i) análise da relação entre elementos de arquitetura e seu código fonte correspondente (chamado de traços de código da arquitetura) e (ii) análise de diferentes tipos de relações entre anomalias de código para que seja possível identificar quais anomalias tendem a afetar mais a arquitetura do *software*. O processo de detecção de anomalia do SCOOP é baseado em duas etapas: (i) utilizando métricas comuns como o número de linhas de código (LOC) para identificação de anomalias individuais e (ii) fazendo uso de métricas baseadas em traços arquiteturais (FERREIRA, 2014) para a detecção das anomalias inter-relacionadas. Um interesse arquitetural é uma responsabilidade ou propriedade de um sistema de *software* que é realizado por elementos arquiteturais (MACIA *et al.*, 2012).

- **ArchiDES** (NASCIMENTO; FONSECA; DANTAS, 2017). Uma ferramenta que apoia o processo de detecção de anomalias arquiteturais. A ferramenta ArchiDES é um Sistema Especialista (SE) composto por: (i) Interface de Aquisição (IA) para obtenção de conhecimento sobre o domínio da aplicação, (ii) Base de Conhecimento (BC) onde ficam armazenados o conhecimento formalizado e (iii) Máquina de Inferência (MI) que é responsável por verificar se os fatos fornecidos pelo usuário possuem relação com os conhecimento da BC. Para cada projeto arquitetural, ocorre o mapeamento dos elementos arquiteturais que compõem a arquitetura do *software*. Estes elementos arquiteturais são estruturados em fatos e comparados com os fatos contidos na base de conhecimento através da Máquina de inferência que procura identificar a presença de anomalias arquiteturais (NASCIMENTO; FONSECA; DANTAS, 2017).

No estado da arte, as abordagens automatizadas têm se tornado de grande importância. Como discutido nessa seção, a adoção de processos automatizados tem como um dos principais objetivos a eficácia na execução e diminuição no custo/esforço durante o processo de detecção de anomalias no projeto de *software*. Porém, somente a detecção de anomalias não é possível apoiar os desenvolvedores na classificação (priorização) de quais anomalias devem ser tratadas primeiro. A seguir, na Seção 2.4 discutiremos sobre priorização de anomalias.

## 2.4 Priorização de Anomalias

O contexto de deteção de anomalias arquiteturais traz consigo diferentes técnicas (GARCIA *et al.*, 2009b; MACIA *et al.*, 2012; ANDRADE, 2013a; NASCIMENTO; FONSECA; DANTAS, 2017). Porém, nenhuma delas ajuda desenvolvedores na priorização de anomalias arquiteturais. Isso se dá, pelo fato que as técnicas de detecção usam métricas estáticas e não analisam o impacto de uma determinada anomalia que fora detectada no projeto arquitetural.

Nos trabalhos de Garcia *et al.* (2009a), Vale *et al.* (2014) é possível concluir que diferentes anomalias promovem diferentes impactos nos projetos arquieturais. Diante disso, priorizar as anomalias detectadas no projeto arquitetural, torna-se uma tarefa ardua pois há uma carência de estudos que apoiem os desenvolvedores nesse cenário. Alguns estudos têm tratado de priorização de anomalias, porém no contexto de anomalias de código (ARCOVERDE *et al.*, 2013; GUIMARAES *et al.*, 2013; VIDAL *et al.*, 2016). Entretando, trabalhado muito próximo ao estudo desempenhado nesta pesquisa, pensando no impacto no nível arquitetural, ou seja, bucando priorizar as anomalias de código detectadas com base no impacto causado na arquitetura do *software*.

Em seu trabalho, Arcoverde *et al.* (2013) apresenta uma abordagem composta por quatro heurísticas (*Change Density Heuristic, Error-Density Heuristic, Anomaly DensityHeuristic, Architecture Role Heuristic*) para apoiar desenvolvedores na priorização de anomalias de código, baseando-se no potencial de degradação da arquitetura de *software*. Ainda segundo Arcoverde *et al.* (2013), tais heurísticas tem como foco explorar diferentes características de um mesmo projeto de software, com objetivo de classificar de forma automática elementos de código que devem ser priorizados para refatoração, baseando-se em sua relevância arquitetural.

No trabalho de Guimaraes *et al.* (2013) é investigado se a priorização de três tipos clássicos de anomalias de código pode ser melhorada quando suportada por modelos arquiteturais. Enquanto Vidal *et al.* (2016) apresenta e avalia um conjunto de critérios suportados de forma automática, para priorização de um aglomerado de anomalias de código como forma de indicadores de problemas arquiteturais em sistemas de *software* em evolução.

O uso de heurísticas de priorização tem como um dos objetivos, colaborar na redução de custos durante o ciclo de vida do *software*. Tais custos podem está relacionados, por exemplo, ao tempo extra para que o projeto seja refatorado, devido a presença de anomalias. A seguir, na Seção 2.5 discutiremos sobre refatoração arquitetural e apresentaremos os trabalhos relacionados.

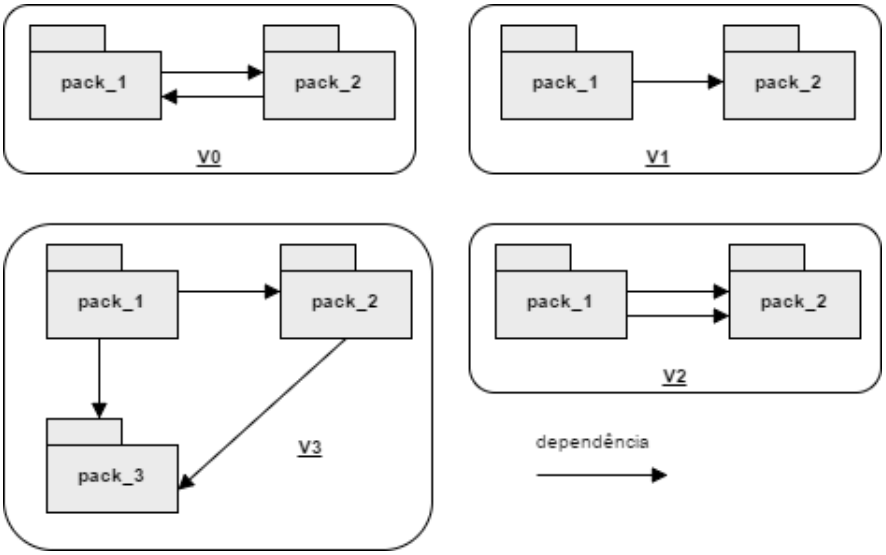
## 2.5 Refatoração Arquitetural

Segundo Hanenberg, Oberschulte e Unland (2003), o processo de refatoração arquitetural refere-se a reestruturação de elementos arquiteturais, que tem por objetivo melhorar a compreensão do projeto sem que o comportamento externo do sistema seja afetado. Esta melhoria na compreensão tendem a facilitar a manutenção e evitar falhas no projeto arquitetural. Segundo Mens e Tourwé (2004) o ciclo do processo de refatoração consiste em uma sequência de seis atividades:

1. Identificar onde deve ser realizado a refatoração.
2. Determinar qual refatoração deverá ser aplicada no local que foi identificado.
3. Assegurar que a refatoração aplicada preserve o comportamento externo.
4. Aplicar a refatoração.
5. Avaliar o efeito da refatoração no custo, esforço e/ou nos atributos de qualidade do software.
6. Garantir a consistência entre os artefatos de *software* e elemento refatorado.

Abaixo, a Figura 9 exemplifica a aplicação de operação de refatoração por meio diferentes exemplos. Seja $V_0$ a versão inicial de um sistema, é possível derivar três cenários diferentes de refatoração. Como pode ser observado, em $V_1$ é realizado a remoção de uma das dependências entre os pacotes `pack_1` e `pack_2` . Em $V_2$, é realizado uma refatoração onde a direção de uma das dependências é alterada. Já em $V_3$, é realizado um outro exemplo de refatoração em que uma das dependências é movida para um novo pacote `pack_3` e uma nova dependência é criada entre `pack_1` e `pack_3`.

Figura 9 – Exemplos de refatorações




**Fonte**: Autor.

Alguns estudos analisam o processo de refatoração em diferentes contextos. Critchlow *et al.* (2003) introduz o conceito de refatoração para degradações em arquitetura de linha de produto (ALP). Segundo os autores, conjuntos de mudanças individuais na arquitetura não resulta, necessariamente, na melhor estrutura para um ALP base. Com isso, propõem um conjunto de refatorações arquiteturais que podem ser usadas para resolver tais problemas. Enquanto Zimmermann (2015) aborda como refatorações arquiteturais podem servir como técnica de evolução para investigar decisões arquiteturais e identificar tarefas relacionadas de *design*, implementação e documentação.

Bisztray, Heckel e Ehrig (2008) propõem uma abordagem para verificar transformações de modelos arquiteturais baseado em UML, apoiado por mapeamento semântico. Enquanto Hanenberg, Oberschulte e Unland (2003), após analisarem a relação entre refatoração orientada à objetos e orientação à aspectos, propõem soluções para elementos orientados à aspectos que são conflitantes com demais refatorações e, apresentam um conjunto de refatorações orientadas à aspectos que podem ajudar na migração do *software* orientado à objetos para orientado à aspectos.

Diante do cenário de refatoração arquitetural e analisando a carência de estudos que o permeia, alguns trabalhos têm motivado a realização de estudos a respeito de refatoração da arquitetura (BOURQUIN; KELLER, 2007; SAMARTHYAM; SURYANARAYANA; SHARMA, 2016). No trabalho de Bourquin e Keller (2007) propõe a refatoração em nível arquitetural, levando em consideração ao alto impacto que tende a causar em relação ao ganho de qualidade do sistema. Os autores fazem considerações importantes sobre a refatoração de anomalias, tais como: ter conhecimento prévio do *software*; as anomalias não são fáceis de quantificar e difíceis de priorizar. Os autores ressaltam que solucionar más decisões arquiteturais pode facilitar a manutenção. Concluem salientando que muitas aplicações industriais não possuem arquiteturas bem definidas, o que reforça a necessidade de revisão da arquitetura para melhorar sua qualidade e a manutenção. Enquanto Samarthyam, Suryanarayana e Sharma (2016), nos motivam com estudo a respeito da refatoração da arquitetura, discutindo as pesquisas atuais e apresentando algumas potenciais pesquisa a respeito de refatoração da arquitetura.

Impactos causados por operações de refatoração e ocorrência de anomalias só podem ser mensurados por meio da aplicação de métricas. Uma visão geral sobre métricas de *software* é apresentada na Seção 2.6

## 2.6 Métricas de *Software*

Métricas de *software* são mecanismos usadas para medir propriedades de uma parte do sistema ou suas especificações. Para Conte *et al.* (1987), métricas são peças fundamentais para controlar os processos de entendimento, desenvolvimento e manutenção

de *software*. Segundo Sommerville *et al.* (2011), as métricas de *software* podem ser divididas em dois grupos distintos: (a) Métricas de Controle, que são em grande parte associadas aos processos de *software*, tendo por objetivo apoiar os processos de gerenciamento de *software*, e (b) Métricas de Previsão, que são por vezes conhecidas como "Métricas de Produto" e tem por objetivo apoiar na previsão de características do *software*.

Segundo Pressman (2010) para medir artefatos de *software* por meio de métricas, as medições devem ser definidas de acordo com objetivos específicos. Caracterizar e validar métricas para que seu valor seja comprovado, é um processo importante para que as mesmas tornem-se úteis (PRESSMAN, 2010). Porém, a principal dificuldade da medição é saber o que medir. No trabalho de Basili e Weiss (1984) é proposto um paradigma denominado Objetivo-Questão-Métrica (GQM do inglês *Goal-Question-Metric*) para identificar métricas relevantes para qualquer parte do *software*. Os níveis do paradigma são descritos a seguir (BASILI, 1992):

1 **Objetivo:** É definido um conjunto de objetivos com vários pontos de vista relativo ao processo de medição.

2 **Questão:** Um conjunto de perguntas é usado para caracterizar a forma de como a avaliação/realização de um objetivo específico será realizada.

3 **Métrica:** Um conjunto de dados é associado a cada pergunta para que seja possível respondê-la de forma quantitativa.

Após estudos preliminares, tem-se observado o aumento no número de trabalhos que avaliam métricas de *software* de forma empírica. No trabalho de Bengtsson (1998), é realizado uma analise da eficácia de adaptações de métricas. Em seu estudo, é demonstrado que é possível realizar adaptações de conjunto de métricas orientado a objetos para ser usado em projeto de arquitetura de *software*. Enquanto Tonu, Ashkan e Tahvildari (2006), avalia por meio de uma abordagem baseada em métricas, a estabilidade arquitetural em diferentes versões de dois sistemas de planilhas eletrônicas, combinando técnicas de avaliação retrospectiva e preditiva[5]

Ademais, o uso de métricas de *software* também possibilita analisar atributos de qualidade de *software* (PADILHA, 2013), avaliando a eficácia dos métodos, ferramentas e processos por meio de métodos quantitativos (SOMMERVILLE *et al.*, 2011). Diante disso, diversos pesquisadores na área de engenharia de *software* empírica buscam propor métricas para medição de atributos de qualidade como, manutenibilidade (COLEMAN

[5] **Avaliação retrospectiva:** Tem por objetivo examinar as sucessivas versões de um sistema de software para analisar como ocorreu a evolução. Enquanto **Avaliação preditiva** tem objetivo examinar um conjunto de possíveis mudanças e mostrar que a arquitetura pode suportar tais mudanças (TONU; ASHKAN; TAHVILDARI, 2006).

*et al.*, 1994; LAND, 2002) e adaptabilidade (SUBRAMANIAN; CHUNG, 2001; CHUNG; SUBRAMANIAN, 2001) em diferentes níveis de abstrações.

No trabalho de Chung e Subramanian (2001) são propostas métricas orientadas a processos para adaptação de arquitetura de *software*, que tem por objetivo fornecer pontuações representando a adaptabilidade de uma arquitetura de *software* analisada. Enquanto que no trabalho de Land (2002) é realizado uma investigação de como a manutenibilidade de um artefato de software pode ser medida e estimada, comparando assim, as medidas de nível de código e arquitetural com o objetivo de encontrar alguma correlação entre elas.

Entretanto, métricas de *software* mostram-se úteis dentre outros contexto da engenharia de *software*, na detecção de anomalias (código ou arquitetura). Neste contexto, alguns estudos tem utilizado métricas como apoio para identificação de anomalias em diferentes contextos (SRIVISUT; MUENCHAISRI, 2007; PADILHA, 2013). No trabalho de Srivisut e Muenchaisri (2007) são propostas métricas para Programação Orientada à Aspectos (POA), como um meio para detectar previamente anomalias baseando-se em suas características. POA é considerado um novo paradigma de programação, que tem por objetivo melhorar a separação de interesses por meio de módulos de aspecto (SRIVISUT; MUENCHAISRI, 2007). Enquanto que no trabalho de Padilha (2013), a eficácia do uso de métricas como indicadores para a detecção de cinco anomalias é investigada por meio de dois experimentos.

Em outro contexto, alguns estudos têm analisado e proposto métricas para refatoração em diferentes cenários (SIMON; STEINBRUCKNER; LEWERENTZ, 2001; MEANANEATRA; RONGVIRIYAPANISH; APIWATTANAPONG, 2011). No trabalho de Simon, Steinbruckner e Lewerentz (2001), é avaliado se métricas podem ser usadas como uma forma eficaz de apoiar na tomada decisão sobre onde aplicar uma determinada refatoração. Nesse contexto, Meananeatra, Rongviriyapanish e Apiwattanapong (2011) propõem um método que introduz condições para selecionar a refatoração com base em métricas de software.

Como discutido nessa seção, métricas de *software* têm sido utilizadas para quantificar atributos internos de *software*. Dentre os atributos está o acoplamento de componentes arquiteturais. Na Seção 2.7 é apresentada uma visão geral sobre acoplamento arquitetural.

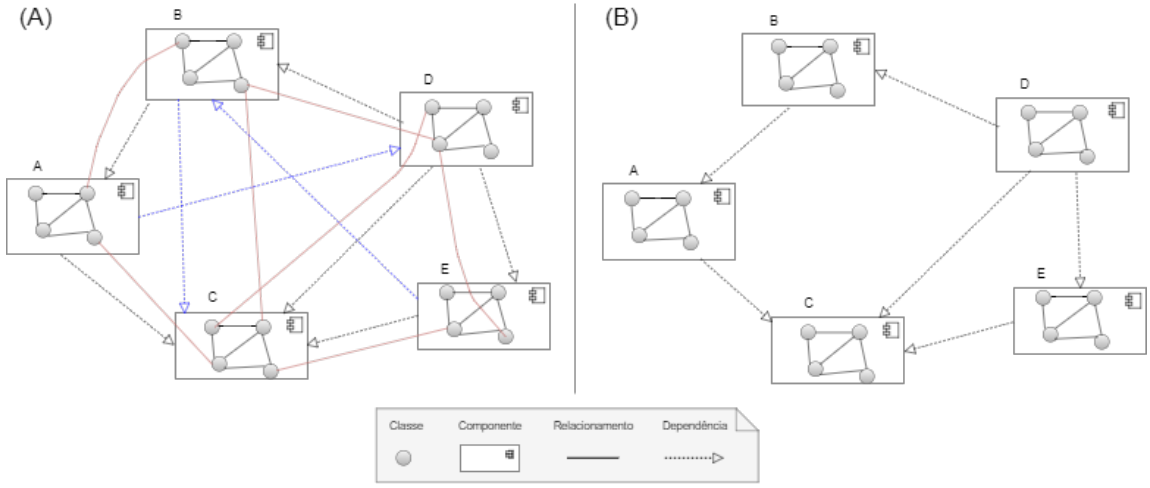
## 2.7 Acoplamento Arquitetural

Para Hitz e Montazeri (1995), acoplamento entre componentes é uma medida da interconexão entre cada componente do sistema. Em outras palavras, é o grau de dependência entre um componente arquitetural para os demais (SOMMERVILLE *et al.*, 2011), onde quanto maior o grau de dependência entre ambos os componentes, mais

altamente acoplados estarão. Para Sousa *et al.* (2017), o alto acoplamento traz consigo muitos problemas como, as alterações realizada em um componente terão que se propagar por todo o sistema, e o sistema será potencialmente mais difícil de entender, afetando assim princípios de qualidade *software* como, compreensibilidade e manutenibilidade.

Quando projetos arquiteturais são classificados com alto grau de acoplamento, as mudanças no *software* tendem a ser mais custosas, acarretando assim, em baixa qualidade interna (PINTO; COSTA, 2014). Idealmente falando, sistemas de *software* devem ter baixo acoplamento, para que assim, sejam menos custosos quando necessário realizar manutenções (LEIGH; WERMELINGER; ZISMAN, 2016). A Figura 10 ilustra dois exemplos distintos, onde (A) apresenta um projeto arquitetural com alto nível de dependência, enquanto (B) apresenta um projeto arquitetural ideal com baixo nível de dependência.

Figura 10 – Grafo ilustrativo com diferentes níveis de acoplamento




**Fonte**: Autor.

Após estudos preliminares, tem-se observado que alguns autores têm abordado sobre acoplamento com diferentes perspectivas (GALL; HAJEK; JAZAYERI, 1998; GOSEVA-POPSTOJANOVA *et al.*, 2003; ALSHARIF; BOND; AL-OTAIBY, 2004; WANG, 2009; BRONDUM; ZHU, 2012; STARON *et al.*, 2013; PINTO; COSTA, 2014; LEIGH; WERMELINGER; ZISMAN, 2016). No trabalho de Gall, Hajek e Jazayeri (1998), é apresentado uma abordagem que usa informações do histórico de um sistema de *software* para identificar o acoplamento lógico entre os módulos de tal forma que possíveis falhas estruturais podem ser identificadas e examinadas. Enquanto Goseva-Popstojanova *et al.* (2003) é proposto uma metodologia de avaliação de risco para nível arquitetural utilizando-se de métricas de acoplamento e complexidade dinâmica obtida a partir de especificações de UML (do termo em inglês - *Unified Modeling Language*).

No trabalho de AlSharif, Bond e Al-Otaiby (2004) apresentam uma abordagem para avaliar a complexidade geral da arquitetura de software, considerando o acoplamento como um dos fatores causadores de alta complexidade. Nesse contexto, Brondum e Zhu (2012)

propõem uma abordagem para visualizar relacionamentos de dependência architetural, onde em seu modelo os autores buscam identificar três dimensões de *design*: (i) tipo de dependência, (ii) a fonte da dependência e (iii) grau de efeito no elemento arquitetural dependente.

No trabalho de Wang (2009) são definidas métricas centradas em torno dos princípios de design para avaliar o acoplamento do serviço e a adequação da granularidade do serviço em soluções de Arquitetura Orientada a Serviço (SOA - do inglês *Service Oriented Architecture*). Os autores ressaltam que tais métricas se relacionam com princípios de design de serviço que: (i) enfatizam fortemente a desvinculação de clientes dos provedores de serviços. Em outras palavras, priorizam a redução as dependências entre os provedores e os clientes e (ii) que a granularidade de serviço possa ser medida pelo "número de operações"cujo é definido em um serviço.

No trablho de Staron *et al.* (2013) é apresentado um método para descoberta automatizada de dependências entre componentes de *software* com base na análise do histórico de revisões de repositórios do sistema. Os autores ressaltam que existe o risco de que os modelos de arquitetura prescritiva na forma de diagramas sejam desatualizados e as dependências arquiteturais implícitas possam se tornar mais freqüentes do que as explícitas. Com isso, o método proposto traça um mapa de dependências implícitas para apoiar os arquitetos nas decisões sobre a evolução da arquitetura (STARON *et al.*, 2013).

No trabalho de Leigh, Wermelinger e Zisman (2016), os autores utlizaram-se de um estudo de caso com regras de *design* de modelo arquitetural, para que pudessem determinar se o acoplamento arquitetetural resulta em dificuldades na implementação e entender quais fatores, além do acoplamento arquitetural, também é importantes ser observado. Entretanto, no trabalho de Mo *et al.* (2016) foi proposto uma nova métrica de manutenção de arquitetura para medir o nível de desacoplamento, ou seja, em vez de medir o nível de acoplamento, a métrica visa medir até que ponto uma arquitetura é desacoplada em pequenos módulos independentes que podem ser alterados separadamente.

Ademais, devido a carência de informações a respeito da arquitetura ou má gerência do processo evolutivo dos sistema de *software*, medir o acoplamento arquitetural tende a ser um fator complicador, nesse cenário. Com isso, torna-se necessário realizar técnicas de rastreamento entre seus componentes e o código-fonte que o representa. Na Seção 2.8 apresenta uma visão sobre técnicas de rastreamento.

## 2.8 Técnicas de Rastreamento (*Traceability*)

A arquitetura de *software* define em alto nível a estrutura de um sistema, onde são representados os componentes e seus relacionamentos (NGUYEN; MUNSON; THAO, 2005). Segundo Asuncion, Asuncion e Taylor (2010) projetos de grande porte geralmente

são compostos por grandes quantidade de artefatos de *software*, como documentos de requisitos, projeto de *software*, códigos, relatórios, entre outros. Com isso, identificar relacionamento entre esses artefatos nem sempre é uma tarefa fácil de desempenhar. Diante desse problema, surge o que na comunidade de engenharia de *software* é denominado *traceability*.

*Traceability* é uma técnica de rastreamento que tem por objetivo identificar os relacionamentos entre artefatos de *software* (ASUNCION; ASUNCION; TAYLOR, 2010), em busca da compreensão de informações relevantes para os desenvolvedores e analistas (GALVAO; GOKNIL, 2007; BRCINA; RIEBISCH, 2008), para possíveis análises do impacto de mudanças ocorridas durante a manutenção (GHABI; EGYED, 2012). O problema de rastreabilidade pode ser abordado de duas formas (ASUNCION; ASUNCION; TAYLOR, 2010): (i) retrospectiva, onde as relações entre artefatos são inferidas, como exemplo as abordagens automatizadas e (ii) prospectiva que gera links de rastreamento à medida que os artefatos vão sendo criados e modificados durante o processo de desenvolvimento.

No estado da arte, alguns estudos propõem técnicas de rastreamento entres os artefatos arquiteturais em diferentes contextos (EGYED, 2003; MURTA; HOEK; WERNER, 2006; GALVAO; GOKNIL, 2007; TEKINERDOGAN; HOFMANN; AKSIT, 2007; MIRAKHORLI; CLELAND-HUANG, 2011). No trabalho de Tekinerdogan, Hofmann e Aksit (2007) é proposto um metamodelo de rastreabilidade de interesse, que permite modelar os interesses, os elementos arquiteturais e os links de rastreamento entre os elementos nos pontos de vista arquitetural. Com isso, os autores puderam aplicar o metamodelo não só para rastrear interesses, mas também, puderam realizar uma análise do impacto que determinadas mudanças nesses interesses podem causar no projeto.

No trabalho de Mirakhorli e Cleland-Huang (2011) é apresentada uma solução por meio de um modelo reutilizável de links de rastreabilidade focada em tática arquitetural, a fim de reduzir o risco de degradação arquitetural quando realizado modificações, preservando as qualidades como confiabilidade, desempenho e segurança. Os atores ressaltam que o uso de infraestruturas reutilizáveis tem como um dos objetivos a redução do esforço necessário para criação e manutenção de links de rastreabilidade arquitetural.

A rastreabilidade entre artefatos arquiteturais e o código-fonte que os realiza são dificilmente gerenciados (NGUYEN; MUNSON; THAO, 2005) e muitas vezes, tais links tendem a tornar-se inconsistentes e incompletos à medida em que são atualizados (CLELAND-HUANG *et al.*, 2007). Isso se dá pelo fato de que, segundo Ghabi e Egyed (2012), a rastreabilidade normalmente é capturada de forma manual, referenciando elementos de modelo e partes de código, armazenando os dados em arquivos.

Segundo Galvao e Goknil (2007), os modelos de *software* são utilizados com finalidade de representar a informação do projeto em diferentes níveis de abstração, de acordo com notações específicas e pontos de vista. Diante disso, Egyed (2003) em seu

trabalho, apresenta uma abordagem automatizada para gerar e validar dependências de rastreamento. Nesse trabalho, o autor aborda o grave problema da ausência de informações de rastreamento ou a incerteza de sua correção, o que torna um fator complicador que tende a limitar a utilidade dos modelos de *software* durante o desenvolvimento. Diante disso, o autor propôs automatizar o que ele considera normalmente como uma atividade demorada devido às possíveis dependências de traços entre artefatos de desenvolvimento e o código.

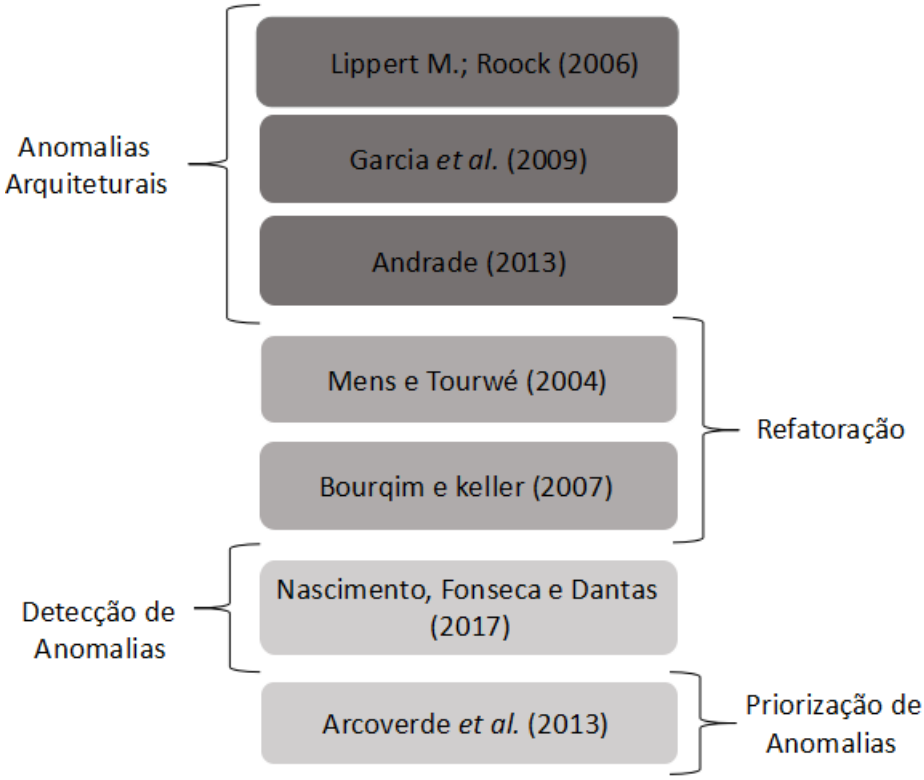
No trabalho de Murta, Hoek e Werner (2006), os autores apresentam uma ferramenta denominada ArchTrace, com uma abordagem que pode ser adaptada às diferentes práticas dos usuários para garantir que os links de rastreabilidade sejam mantidos entre a arquitetura conceitual e seu código-fonte correspondente, durante o processo de evolução. Os autores ressaltam que: (i) a ferramenta atualiza continuamente os links de rastreabilidade em resposta a todas as alterações realizadas e (ii) a atualização específica a ser feita é determinada por um conjunto de políticas de gerenciamento de rastreabilidade.

Pode-se identificar que estudos a respeito de *traceability* tem se tornado um tópico consideravelmente estudado, buscando atacar problemas como a inconsistência de links de rastreabilidade entre artefatos em projeto de *software*.

## 2.9 Resumo

Este capítulo apresentou uma revisão dos conceitos e trabalhos existentes, que amparam a condução desta pesquisa de mestrado. Discutimos a literatura sobre arquiteturas de sistema de *software* em evolução, anomalias arquiteturais e estudos relacionados a sua detecção e priorização, refatoração arquitetural, métricas de *software*, acoplamento arquitetural e por fim, técnicas de rastreamento dos artefatos arquiteturais e suas classes correspondente no nível de código. A Figura 11 ilustra, de forma resumida, as áreas e os estudos que guiaram nossa pesquisa de mestrado.

Figura 11 – Estudos guia




**Fonte**: Autor.

Embora anomalias arquiteturais tenham sido largamente estudadas, sua influência nas operações de *refactoring* dos componentes arquiteturais ainda necessita de investigação. O catálogo de anomalias arquiteturais apresentado na Seção 2.2 serviu de base para a construção do conhecimento utilizado no desenvolvimento de uma solução computação para o tratamento priorizado de anomalias, apresentado no Capítulo 4.

# 3 Anomalias Arquiteturais vs. Refatoração

Neste capítulo será apresentado e discutido os resultados obtidos em estudos preliminares com o objetivo de identificar a relação existente entre refatoração de código e anomalias arquiteturais. Na Seção 3.1 são descritos, brevemente, as informações referentes aos modelos arquiteturais das aplicações utilizadas para condução deste trabalho. Na Seção 3.2, será apresentada e discutida a relação entre anomalias arquiteturais e operações de refatoração do código. Em Seguida, na Seção 3.3 é apresentada a correlação das anomalias arquiteturais e o escopo de seus elementos. Por fim, o resumo do capítulo é apresentado na Seção 3.4.

## 3.1 Aplicações Alvo

O estudo foi conduzido por meio da análise de três aplicações de *software* em evolução, *MobileMedia* (FIGUEIREDO *et al.*, 2008), *Notepad* (ANDRADE, 2013b) e *HealthWatcher* (UFPE, 2017), com propósito de analisar a relação das anomalias arquiteturais presentes em seus projetos e a demanda de operações de *refactoring*. Na Tabela 1 apresentamos brevemente as aplicações:

Tabela 1 – Aplicações Alvo

| ***Software*** | **Versões** | **Objetivo** |
|---|---|---|
| *MobileMedia* | 8 | É uma família de programas que oferece suporte para gerenciar (criar, excluir, visualizar, reproduzir, enviar) diferentes tipos de mídia (foto, música e vídeo) em dispositivos móveis. Foram analisadas 8 versões do MobileMedia. |
| *Notepad SPL* | 7 | É uma LPS de editor de texto, desenvolvido na Universidade do Texas, com objetivo de prover suporte em edições de textos, onde cada versão contém um conjunto de funcionalidades (recortar, copiar, colar, procurar). Foram analisadas 7 versões do Notepad. |
| *HealthWatcher* | 10 | Aplicação Web para permitir que os cidadãos registram queixas relativas a questões de saúde em instituições públicas. Foram analisadas 10 versões do HealthWatcher. |

Os sistemas indicados na Tabela 1 foram objeto de nossa análise por conter implementações na linguagem de programação Java e devido seus projetos terem sido

concebidos seguindo boas práticas de desenvolvimento. Priorizamos o estudo de aplicações Java considerando que (i) Java é uma linguagem bastante disseminada no mercado de desenvolvimento e (ii) há relatos públicos de uma adoção bem sucedida de Java em projetos de desenvolvimento de software industrial. Os pontos que podemos destacar que nos ajudaram na escolha das aplicações envolvidas no projeto, são:

**A.** Todas elas evoluíram e sofreram mudanças por um longo período de tempo;

**B.** Todas possui um conjunto de diferentes funcionalidades e foram devolvidas por diferentes desenvolvedores;

**C.** Cada aplicação pertence a um domínio diferente;

**D.** Todas elas foram utilizadas com sucesso em diferentes estudos;

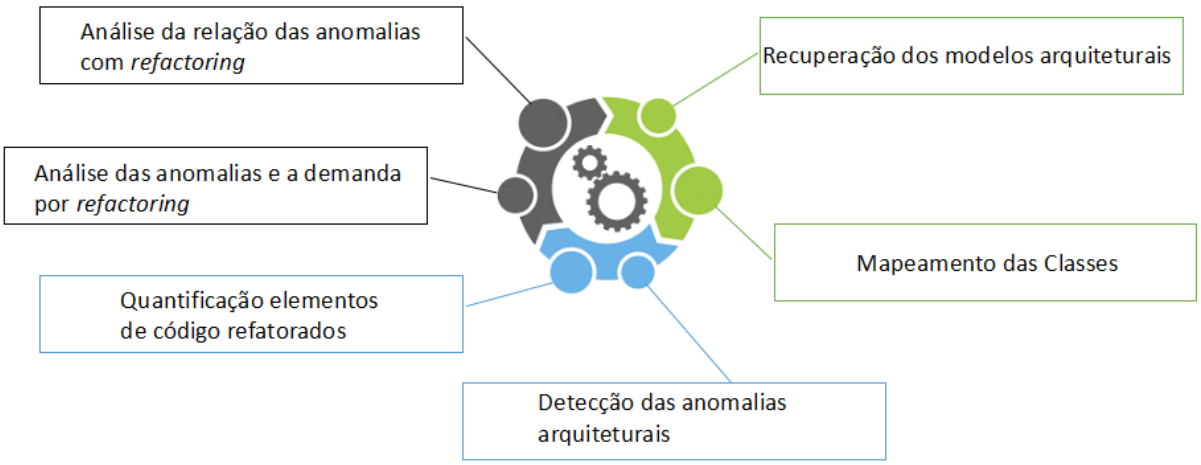
## 3.2 Anomalias Arquiteturais *versus* Refatoração

Em seu trabalho, (GARCIA *et al.*, 2009b) ressalta que os problemas estruturais tendem a estar relacionados a existência de anomalias arquiteturais (Capítulo 2). Segundo (GARCIA *et al.*, 2009b) para tratar anomalias arquiteturais é necessário realizar mudanças corretivas na estrutura dos projetos arquiteturais. Sabe-se que o tratamento de anomalias minimiza a ocorrência de operações corretivas nos projetos (ARCOVERDE; GARCIA; FIGUEIREDO, 2011; MACIA *et al.*, 2011). Porém, nem todas as operações de refatoração acontecem em função do tratamento das anomalias e consequentemente pouco se sabe sobre a relação existente entre a ocorrência simultânea de anomalias arquiteturais e operações de refatoração.

Para a fase de análise de relação da coexistência de anomalias arquiteturais e as operações de refatoração, as atividades seguintes foram divididas em 6 etapas: (i) recuperação dos modelos arquiteturais dos sistemas de *software MobileMedia, Notepad SPL* e *HealthWatcher* para obter os modelos de componentes arquiteturais dos projetos; (ii) mapeamento das classes que compõem cada componente dos projetos; (iii) detecção das anomalias arquiteturais, analisando os modelos de componentes arquiteturais com o sistema especialista ArchiDES (NASCIMENTO; FONSECA; DANTAS, 2017); (iv) quantificação de elementos refatorados; (v) classificação das anomalias e o número de operações de *refactoring* e (vi) identificação das operações de *refactoring* associados às anomalias detectadas nos projetos. O objetivo da divisão por etapas foi facilitar o entendimento e acompanhar da melhor forma a condução do trabalho para que ao fim das análises pudéssemos verificar se a quantidade de operações de refatoração é influenciada pela existência de anomalias, ou seja, se anomalias arquiteturais demandam um maior número de mudanças nos elementos de código. A Figura 12 ilustra as etapas para a fase

de análise direta da relação da coexistência de anomalias arquiteturais e as operações de refatoração no código.

Figura 12 – Etapas do estudo de análise




**Fonte**: Autor.

**Recuperação dos modelos arquiteturais**. Para os procedimentos de recuperação dos modelos arquiteturais, a ferramenta SoMOX (KROGMANN, 2012) foi utilizada. Foram recuperados 8 modelos arquiteturais do *MobileMedia*, bem como, dos 7 modelos arquiteturais do *Notepad SPL* e dos 10 modelos arquiteturais do *HealthWatcher*. O processo de recuperação foi orientado pelos modelos arquiteturais previamente encontrados nos trabalhos de Figueiredo *et al.* (2008) e Andrade (2013b), bem como, os projetos disponíveis pela UFPE (2017), respectivamente. Para que as arquiteturas recuperadas refletissem a realidade dos sistemas, foi necessário realizar alguns ajustes das métricas na ferramenta SoMoX. As métricas foram ajustadas com base nos valores guiados pelo trabalho de Nascimento, Fonseca e Dantas (2017). Para mais detalhes sobre a recuperação dos modelos arquiteturais, as métricas utilizadas são apresentadas no Apêndice A e os modelos arquiteturais apresentados no Apêndice B.

**Mapeamento de classes**. Para a identificação das classes que compõem cada componente arquitetural, uma inspeção manual foi realizada. Esta inspeção foi guiada pelos modelos arquiteturais gerados pela ferramenta SoMoX. Utilizando-se de técnica *Traceability*, foi possível a identificar os relacionamentos entre os elementos arquiteturais e seus respectivos elementos de código. Devido a falta de informações a respeito dos projetos de cada aplicação alvo de estudo, o mapeamento das classes que compunham os componentes arquiteturais anômalos foi realizado de forma manual, a partir dos modelos arquiteturais gerados pela ferramenta SoMoX. Com base nas informações como, nome dos componentes, seus relacionamentos e suas interfaces provedoras de serviços, foi possível identificar os elementos de código correspondentes aos componentes arquiteturais. As Tabelas 2, 3 e 4 detalham as versões dos projetos que tiveram o maior número de operações

de refatoração, apresentando as respectivas classes que compõem os componentes anômalos da versão 7 do *MobileMedia*, da versão 3 do *Notepad SPL* e da versão 10 do *HealthWatcher*, respectivamente.

Tabela 2 – Componentes e suas classes - Versão 7 do *MobileMedia*

| **Componentes Anômalos** | **Classe(s) que o compõe (.java)** |
|---|---|
| *BasicComponent Nº 1* | BaseThread; |
| *BasicComponent Nº 19* | MediaListScreen; |
| *BasicComponent Nº 21* | SelectTypeOfMedia; |
| *BasicComponent Nº 23* | CaptureVideoScreen; |
| *BasicComponent Nº 25* | MediaUtil; |
| *BasicComponent Nº 26* | SmsMessaging, BaseMessaging; |
| *BasicComponent Nº 27* | SmsReceiverController, ControllerInterface; |
| *BasicComponent Nº 28* | SmsSenderController, ControllerInterface; |
| *CompositeComponent Nº 1* | AbstractController, ControllerInterface, MediaController, AlbumController, BaseController, SelectMediaController, MusicPlayController, VideoCaptureController, PhotoViewController, PlayVideoController, MediaListController; |
| *CompositeComponent Nº 2* | VideoMediaAccessor, AlbumData, MediaData, ImageMediaAccessor, MediaAccessor, MusicMediaAccessor; |
| *CompositeComponent Nº 3* | PlayVideoScreen, PlayMediaScreen; CaptureVideoScreen, AlbumListScreen; |
| *CompositeComponent Nº 4* | SmsSenderThread, ReceiverThread; |

Tabela 3 – Componentes e suas classes - Versão 3 do *Notepad SPL*

| **Componentes Anômalos** | **Classe(s) que o compõe (.java)** |
|---|---|
| *BasicComponent Nº 2* | Notepad; |
| *BasicComponent Nº 5* | Fonts; |
| *CompositeComponent Nº 1* | Center, ExampleFileFilter, Actions, Print; |
| *CompositeComponent Nº 2* | UndoAction, RedoAction; |

Tabela 4 – Componentes e suas classes - Versão 10 do *HealthWatcher*

| **Componentes Anômalos** | **Classe(s) que o compõe (.java)** |
|---|---|
| *BasicComponent Nº 83* | RMIFacadeAdapter, HWServer, ComplaintRecord, DiseaseRecord, SymptomRecord, EmployeeRecord, RMIFacadeFactory, AbstractFacadeFactory, MedicalSpecialityRecord, HealthUnitRecord, ArrayRepositoryFactory, RDBRepositoryFactory, AbstractRepositoryFactory, EmployeeRepositoryArray, DiseaseTypeRepositoryArray, HealthUnitRepositoryArray, SymptomRepositoryArray, ComplaintRepositoryArray, SpecialityRepositoryArray, DiseaseTypeRepositoryRDB, EmployeeRepositoryRDB, SpecialityRepositoryRDB, AddressRepositoryRDB, HealthUnitRepositoryRDB, ComplaintRepositoryRDB, SymptomRepositoryRDB, DiseaseType, ComplaintState, IComplaintRepository, IAddressRepository, IDiseaseRepository, IEmployeeRepository; IFacadeRMITargetAdapter, IHealthUnitRepository, ISpecialityRepository, ISymptomRepository; |
| *BasicComponent Nº 84* | Complaint, Symptom, Employee, MedicalSpeciality, HealthUnit, RMIServletAdapter, IFacade, Observer; |

Continue na próxima página

Tabela 4 – Continuação da página anterior

| **Componentes Anômalos** | **Classe(s) que o compõe (.java)** |
|---|---|
| *CompositeComponent Nº 1* | ServletRequestAdapter, Command, CommandRequest, UpdateHealthUnitList, UpdateSymptomList, UpdateEmployeeSearch, UpdateMedicalSpecialityData, InsertFoodComplaint, InsertAnimalComplaint, InsertSymptom, InsertDiseaseType, GetDataForSearchBySpeciality, GetDataForSearchByHealthUnit, UpdateSymptomSearch, LoginMenu, UpdateSymptomData, UpdateMedicalSpecialitySearch, UpdateHealthUnitData, SearchDiseaseData, InsertEmployee, SearchComplaintData, GetDataForSearchByDiseaseType, ConfigRMI, UpdateComplaintSearch, UpdateHealthUnitSearch, InsertSpecialComplaint, UpdateEmployeeData, Login, UpdateComplaintList, InsertHealthUnit, InsertMedicalSpeciality, SearchSpecialtiesByHealthUnit, UpdateMedicalSpecialityList, SearchHealthUnitsBySpecialty, UpdateComplaintData, HWServlet, ServletWebServer; |
| *CompositeComponent Nº 2* | IteratorRMITargetAdapter, ConcurrencyManager, LogMechanism, ThreadLogging, PersistenceMechanism, Functions, ConcreteIterator, Date, Library, HTMLCode, Schedule, IIteratorRMITargetAdapter, IPersistenceMechanism, IteratorDsk, LocalIterator; |
| *CompositeComponent Nº 3* | IteratorRMITargetAdapter, ConcurrencyManager, LogMechanism, ThreadLogging, PersistenceMechanism, Functions, ConcreteIterator, Date, Library, HTMLCode, Schedule, IIteratorRMITargetAdapter, IPersistenceMechanism, IteratorDsk, LocalIterator; |
| *CompositeComponent Nº 4* | HealthWatcherFacade, IteratorRMITargetAdapter, ConcurrencyManager, LogMechanism, ThreadLogging, PersistenceMechanism, Functions, ConcreteIterator, Date, Library, HTMLCode, Schedule, IIteratorRMITargetAdapter, IPersistenceMechanism, IteratorDsk, LocalIterator, IFacade, Observer; |

**Detecção das anomalias arquiteturais**. Os modelos arquiteturais recuperados

serviram de entrada para a ferramenta ArchiDES (NASCIMENTO; FONSECA; DANTAS, 2017) para detecção das anomalias. A ferramenta trabalha da seguinte forma: Segundo (NASCIMENTO; FONSECA; DANTAS, 2017), a partir de cada modelo arquitetural a ferramenta realiza uma análise com objetivo de detectar as anomalias, baseando em regras de produção construídas para esse fim. Após a análise, um relatório é gerado como diagnóstico do modelo arquitetural fornecido. Após a detecção das anomalias, foi realizado a inspeção do relatório com objetivo de identificar possíveis detecções repetidas e então remove-las da lista de detecção. A Tabela 5 apresenta o total de anomalias arquiteturais detectadas em cada versão das três aplicações alvo deste estudo, após análise e inspeção do relatório. O total de anomalias de cada versão das aplicações de objeto de estudo, está dividido entre as anomalias arquiteturais destacadas por Garcia *et al.* (2009b) e apresentadas no Capítulo 2 deste trabalho.

Tabela 5 – Total de anomalias arquiteturais em cada versão das aplicações alvo

| ***Software*** | **V1** | **V2** | **V3** | **V4** | **V5** | **V6** | **V7** | **V8** | **V9** | **V10** |
|---|---|---|---|---|---|---|---|---|---|---|
| *HealtWhatcher* | 27 | 32 | 34 | 31 | 34 | 45 | 70 | 58 | 94 | 44 |
| *MobileMedia* | 20 | 25 | 26 | 29 | 43 | 41 | 45 | 49 | * | * |
| *Notepad SPL* | 37 | 37 | 35 | 37 | 37 | 31 | 37 | * | * | * |

**Quantificação dos elementos de código refatorados**. Nessa etapa foi possível analisar quantitativamente as mudanças nos elementos de código. Essa etapa analisamos um conjunto de medidas como número de componentes e suas classes adicionadas ou alteradas. Contabilizamos as mudanças relacionadas às linhas de código adicionadas ou modificadas (LOC) como, alteração do nome da classe, inserção de condições condições, alterações no tratamento de exceções, alterações na assinatura do método. Para a quantificação dos elementos refatorados, utilizou-se de contagem manual. A análise das versões dos sistemas foi conduzida através do mapeamento de todas as operações de refatoração realizadas nos elementos de código que compõe cada componente arquitetural. O mapeamento foi conduzido através da inspeção manual dos elementos de código de cada versão das aplicações. Usando a ferramenta *Diff* (SOURCEGEAR, 2016), os elementos de código da versão $V_{i+1}$ foram comparados com a sua versão antecessora $V_i$, onde $i$ corresponde ao número da versão das aplicações. Em cada comparação foi contabilizado o total de mudanças realizadas nos elementos de código para cada versão das aplicações. No fim, foi gerado um relatório contendo as refatorações nos elementos de código de cada versão dos sistemas. A Tabela 6 apresenta, após análise dos componentes arquiteturais e seus elementos de código (classes), o total de mudanças em cada versão das aplicações alvo deste estudo. Totalizando quinhentos e cinquenta (550) refatorações no *HealthWatcher*,

cento e trinta e nove (139) no *MobileMedia* e noventa e seis (96) no *Notepad SPL*. As versões iniciais (V1) de cada projeto foram tomadas como base para análise e mapeamento dos *refactoring* em suas versões sucessoras. Por esse motivo, a contagem de refatorações para as versões bases é igual à Zero (0).

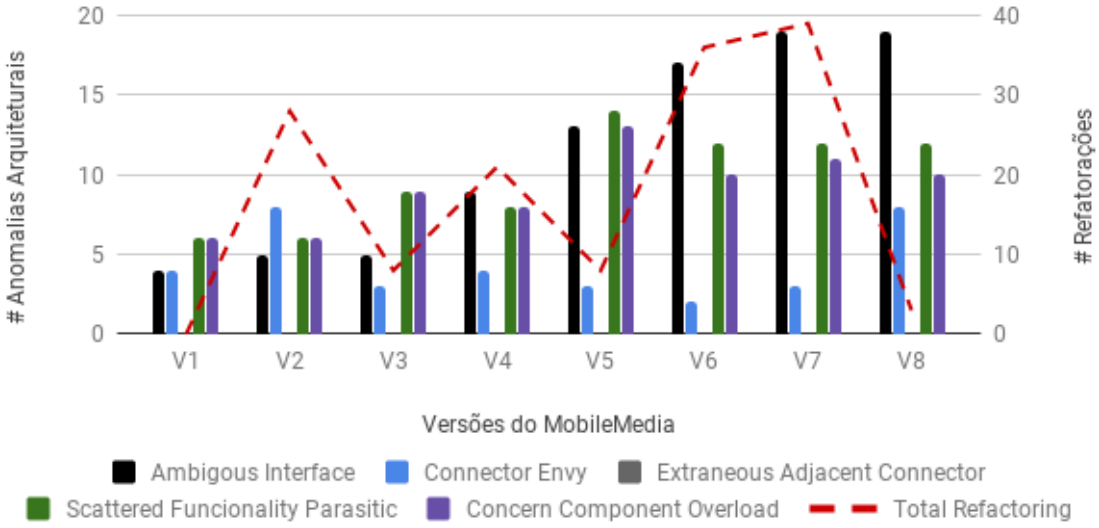
Tabela 6 – Total de refatorações em cada versão das aplicações alvo

| ***Software*** | **V1** | **V2** | **V3** | **V4** | **V5** | **V6** | **V7** | **V8** | **V9** | **V10** | **Total** |
|---|---|---|---|---|---|---|---|---|---|---|---|
| *HealtWhatcher* | 0 | 106 | 57 | 19 | 8 | 5 | 108 | 1 | 26 | 220 | 550 |
| *MobileMedia* | 0 | 27 | 8 | 21 | 8 | 36 | 36 | 3 | * | * | 139 |
| *Notepad SPL* | 0 | 14 | 29 | 16 | 7 | 27 | 3 | * | * | * | 96 |

**Anomalias e as demandas por operações de *refactoring***. Idealmente falando, uma anomalia demanda maior operação de refatoração quando para corrigi-la é necessário realizar alterações em vários elementos de código, resultando assim em um número muito grande de refatorações. Essa fase foi realizada de forma manual. Em cada comparação foi contabilizada o total de refatorações, bem como, o total de determinada anomalia presente em cada versão das aplicações alvo. Os estudos foram conduzidos por meio da análise de 8 versões do *MobileMedia* (FIGUEIREDO *et al.*, 2008), 7 versões do *Notepad SPL* (ANDRADE, 2013b) e 10 versões do *HealthWatcher* (UFPE, 2017) (vide Tabela 1). O objetivo desse estudo foi verificar se a quantidade de operações de refatoração é influenciada pela existência de anomalias arquiteturais, inicialmente catalogadas por Garcia *et al.* (2009a). A análise das versões dos três sistemas foi conduzida por meio do mapeamento de todas as operações de refatoração realizadas. O mapeamento foi conduzido por meio da inspeção manual dos elementos de códigos dos respectivos componentes arquiteturais. As análises da influência das anomalias no número de mudanças realizadas são apresentadas a seguir.

A Figura 13 ilustra a relação existente entre a ocorrência de anomalias e as operações de refatoração para as 8 versões ($V_1$-$V_8$) do *MobileMedia*. Os eixos $y$ primário e $y$ secundário representam o número de anomalias e operações de refatoração, respectivamente.

Figura 13 – Operação de *refactoring* vs. Anomalia - *MobileMedia*

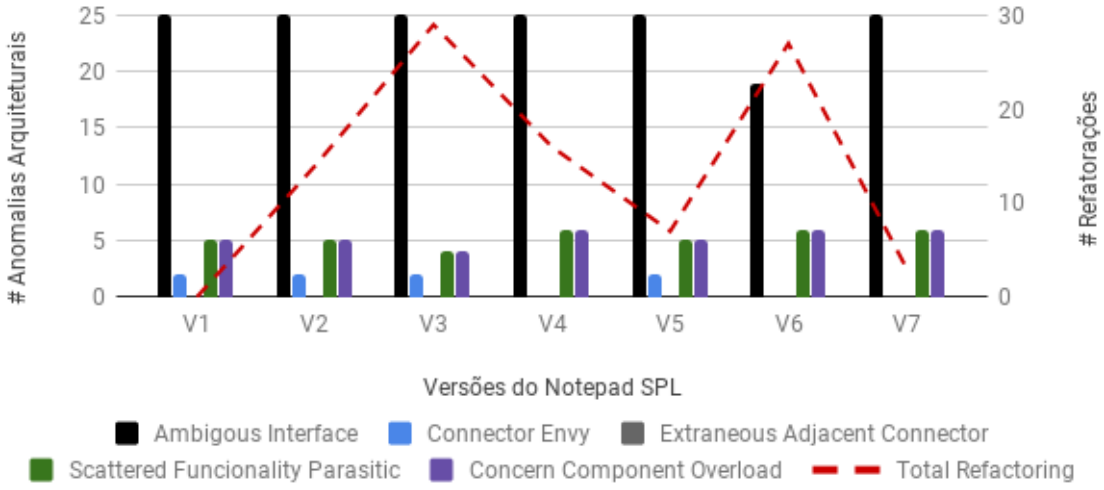



**Fonte**: Autor.

Após análise, foi possível observar que o número de refatorações no sistema *MobileMedia* atingiu seu pico em $V_7$. Coincidentemente, $V_7$ apresentou um acréscimo considerável no número de anomalias instaladas, quando comparado com suas versões antecessoras. Em particular, $V_7$ apresenta um crescimento significativo da anomalia *Ambiguous Interface* (AI), o que significa que interfaces estão sendo usadas como único ponto genérico de entrada para um mesmo componente, ao mesmo tempo que operações de refatoração se tornam mais frequente. Foi possível também identificar um aumento de operações de refatoração já em $V_2$. Nesta versão, surgiram mais anomalias do tipo *Connector Envy* (CE). No que diz respeito a CE, o crescimento foi de 60%. Enquanto $V_1$ possuía 5 anomalias do tipo CE, $V_2$ apresentou 8.

Analisando a influência das anomalias em outro cenário, a Figura 14 ilustra a relação existente entre a ocorrência de anomalias e as operações de refatoração para as 7 versões ($V_1$-$V_7$) do *Notepad SPL*. Os eixos $y$ primário e $y$ secundário representam o número de anomalias e operações de refatoração, respectivamente.

Figura 14 – Operação de *refactoring* vs. Anomalia - *Notepad SPL*

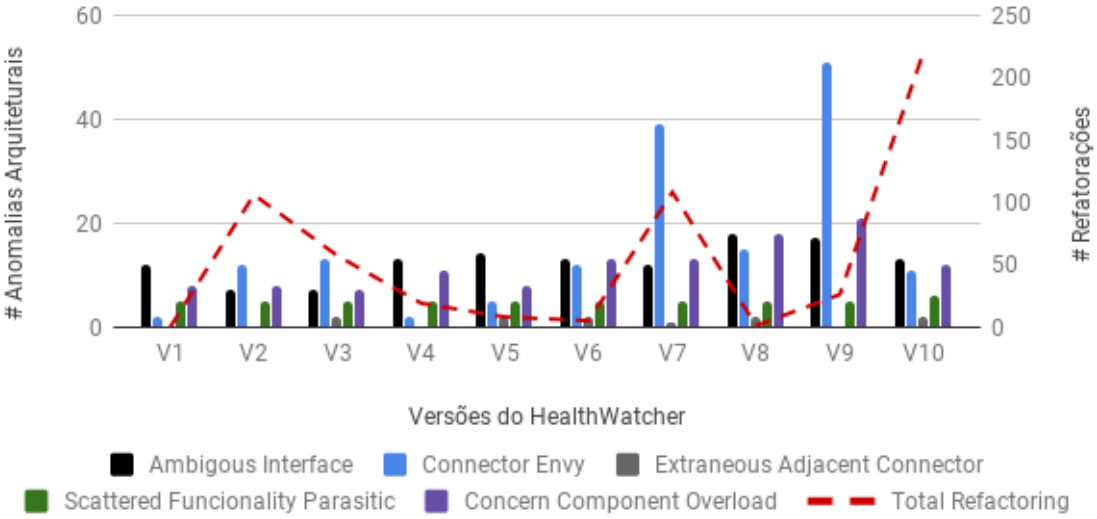



**Fonte**: Autor.

Analisando o segundo cenário de aplicação, foi possível observar que o número de refatorações no sistema *Notepad SPL* cresceu consideravelmente nas versões $V_3$ e $V_6$. Porém, atingiu seu pico somente em $V_3$. Durante suas primeiras 5 versões e sua última versão $V_7$, a presença da anomalia *Ambiguous Interface* (AI) se faz constante, apresentando uma queda somente em sua penúltima versão $V_6$. Foi possível também identificar uma queda significativa nas operações de *refactoring* em $V_5$. No que diz respeito as refatorações, houve um decrescimento de 75,87%, comparado com o pico de refatoração em $V_3$. Enquanto $V_3$ possuía 29 operações de *refactoring*, $V_5$ apresentou somente 7.

Analisando a influência das anomalias em um terceiro cenário, a Figura 15 ilustra a relação existente entre a ocorrência de anomalias e as operações de refatoração para as 10 versões ($V_1$-$V_{10}$) do *HealthWatcher*. Os eixos $y$ primário e $y$ secundário representam o número de anomalias e operações de refatoração, respectivamente.

Figura 15 – Operação de *refactoring* vs. Anomalia - *HealthWatcher*




**Fonte**: Autor.

Analisando o terceiro cenário de aplicação, foi possível observar que o número de refatorações no sistema *HealthWatcher* cresceu consideravelmente nas versões $V_2$, $V_7$ e $V_{10}$. Porém, atingiu seu pico na última versão, $V_{10}$. Um crescimento de 103,7% em relação ao segundo maior pico de mudanças, ocorrido na $V_7$. É importante salientar que $V_9$ apresentou um crescimento significativo da anomalia *Connector Envy* (CE). No que diz respeito a CE, houve um crescimento de 170%. Enquanto $V_9$ possuía 51 anomalias do tipo CE, $V_8$ apresentou 15.

Analisando os três cenários de aplicações, não foi possível extrair indicadores da relação existente entre a ocorrência simultânea de anomalias arquiteturais e operações de refatoração. Como cenário ideal, somente a aplicação *MobileMedia* foi capaz de traçar o indicativo dessa relação. Nessa aplicação, em especial, a ocorrência de anomalias do tipo AI e CE variaram alinhadas com a quantidade de operações de refatoração.

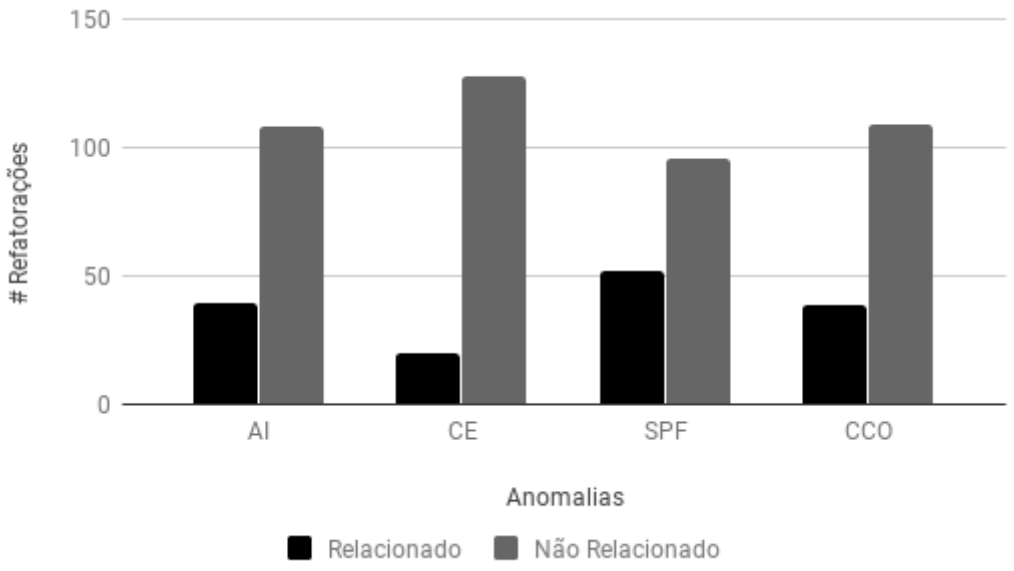
Com base nos resultados obtidos com as três aplicações alvo, não é possível concluir que tal alinhamento acontece para todos os sistemas. Visto que, em nossos estudos o alinhamento das ocorrências de anomalias arquiteturais e as ocorrências de operações de *refactoring* ocorreu somente nas versões do *MobileMedia*. Apesar da anomalia do tipo AI se fazer constante nas cinco primeiras versões do *Notepad SPL*, o número de refatorações não variou alinhado a tal anomalia. Apesar de a anomalia do tipo CE apresentar um crescimento bastante significativo em $V_7$ e $V_9$ da aplicação *HealthWatcher*, esse crescimento também não pode ser considerado um alinhamento ao número de mudanças ocorridas. No que diz respeito às mudanças no *HealthWatcher*, $V_9$ obteve um decrescimento de 75,92% em relação a $V_7$. Enquanto $V_7$ apresentou um total de 108 refatorações, $V_9$ apresentou 26. Com base nesses resultados, conclui-se que nem sempre as versões mais infectadas por um

determinado tipo de anomalia tendem a indicar relações com a ocorrência simultânea de refatorações no código.

**Anomalias arquiteturais e a relação com operações de *refactoring*.** Após análise das refatorações e as anomalias presentes nas aplicações, realizou-se a análise com objetivo de identificar quais operações de *refactoring* estariam relacionadas com as anomalias presentes nas aplicações, de forma isolada. Essa análise foi feita de modo manual. Nessa fase, foi necessário analisar o total de mudanças realizadas nos elementos de códigos que compõe cada componente arquitetural, ao longo da evolução do sistema. Assim, verificou se o total de refatoração nos elemento de código das aplicações alvo está relacionada com uma determinada anomalia presente no componente arquitetural. Nessa etapa, as mudanças foram separadas em dois grupos: (i) grupo contendo o total de mudanças relacionadas a uma determinada anomalia arquitetural e (ii) grupo contendo o total de mudanças não relacionadas a uma determinada anomalia arquitetural. As Figuras 16 17 e 18 apresentam a relação de refactoring relacionados e não relacionado às anomalias arquiteturais detectadas nas versões das três aplicações alvo deste trabalho.

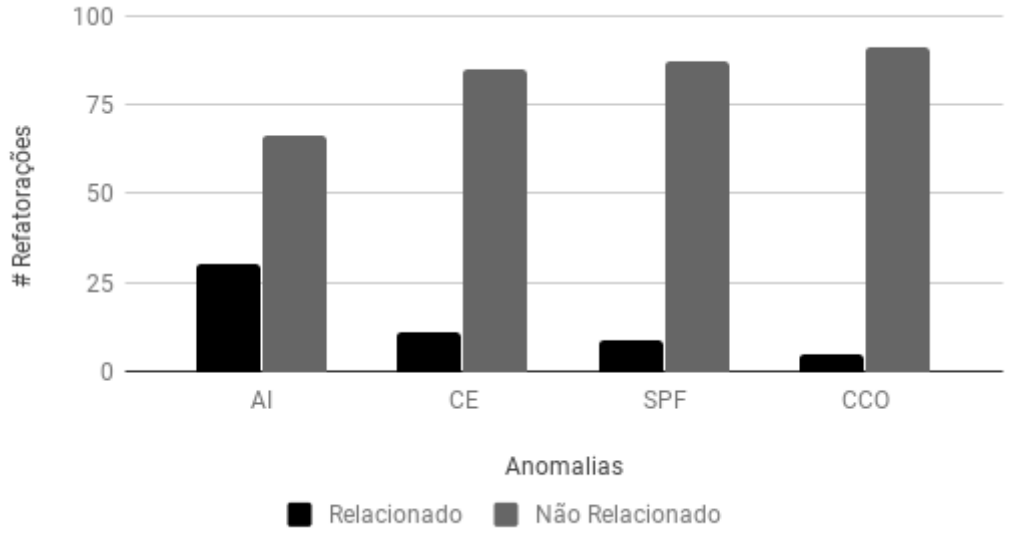
Durante a fase de detecção das anomalias arquiteturais, a ferramenta ArchiDES não detectou a anomalia arquitetural *Extraneous Adjacent Connector* (EAC) nas aplicações *MobileMedia* e *Notepad SPL*. Porém, após a detecão de anomalias nas versões do sistema *HealthWatcher*, pudemos perceber que a anomalia EAC foi detectada. Diante disso, acreditamos que isso pode indicar que os dois primeiros sistemas estejam livres de tal anomalia ou, no pior caso, os elementos arquiteturais infectados com essa anomalia não foram mapeados pela ferramenta de recuperação arquitetural. Portanto, a análise e quantificação das operações de *refactoring* relacionadas a essa anomalia não foi contabilizada para os sistemas *MobileMedia* e *Notepad SPL*.

Figura 16 – *Refactoring* relacionados e não relacionados às Anomalias - *MobileMedia*



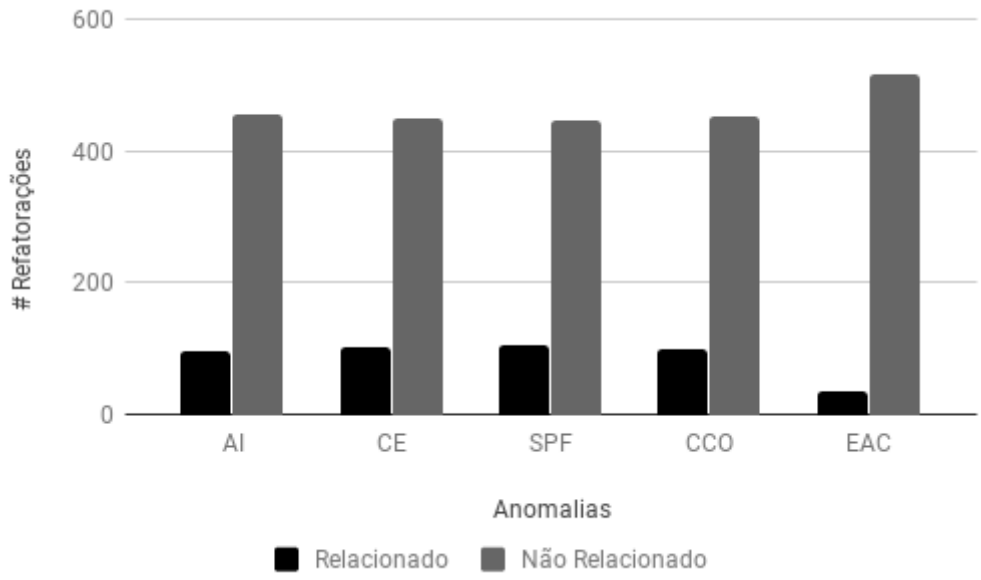

**Fonte**: Autor.

Figura 17 – *Refactoring* relacionados e não relacionados às Anomalias - *Notepad SPL*




**Fonte**: Autor.

Figura 18 – *Refactoring* relacionados e não relacionados às Anomalias - *HealthWatcher*



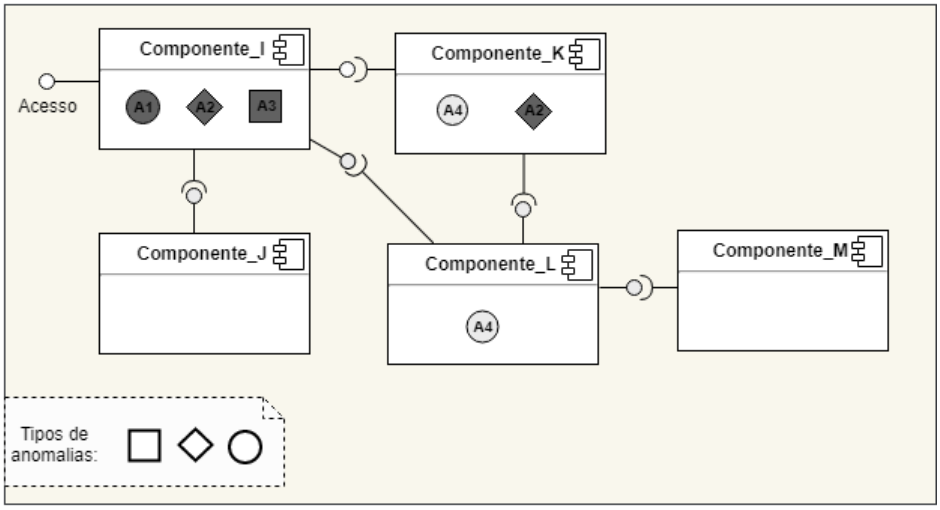

**Fonte**: Autor.

Em análise da relação das refatorações ocorridas nos elementos de código e a coexistência das anomalias arquiteturais, pode-se observar que nos três sistemas exite um número bastante significativo de operações de *refactoring* não relacionados com as anomalias. Podendo chegar em alguns casos, próximo a 100%, se as anomalias persistirem.

## 3.3 Escopo dos Componentes Arquiteturais vs. Anomalias

O escopo de um componente arquitetural refere-se ao percentual de componentes da aplicação que estão a ele associados. O escopo de um componente nos revela a quantidade de componentes possivelmente afetados a partir dele. Nesta direção, o escopo dos componentes arquiteturais surge como um forte indicador para operações de refatoração dos elementos de código. Com base neste entendimento, analisando a Figura 19 é possível observar que os componentes de maior escopo são `Componente_I` e `Componente_L`, pois os mesmos alcançam dois outros componentes por meio de suas interfaces provedoras (`Componente_K` e `Componente_L`) e (`Componente_K` e `Componente_M`), respectivamente. Logo, o escopo dos componentes `Componente_I` e `Componente_L` são iguais a 40% (número de componentes alcançados dividido pelo número total de elementos).

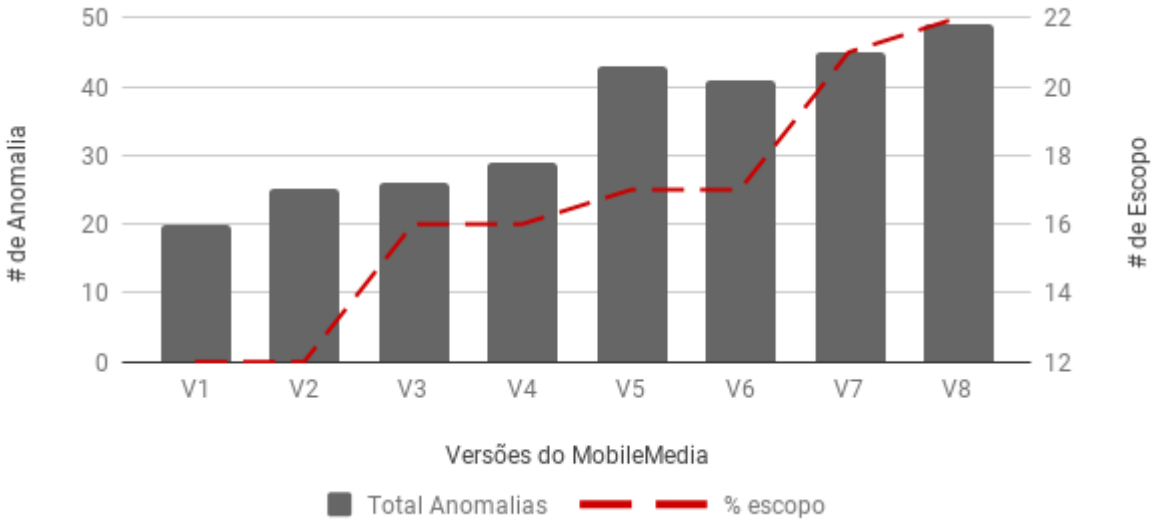
Figura 19 – Componentes arquiteturais




**Fonte**: Autor.

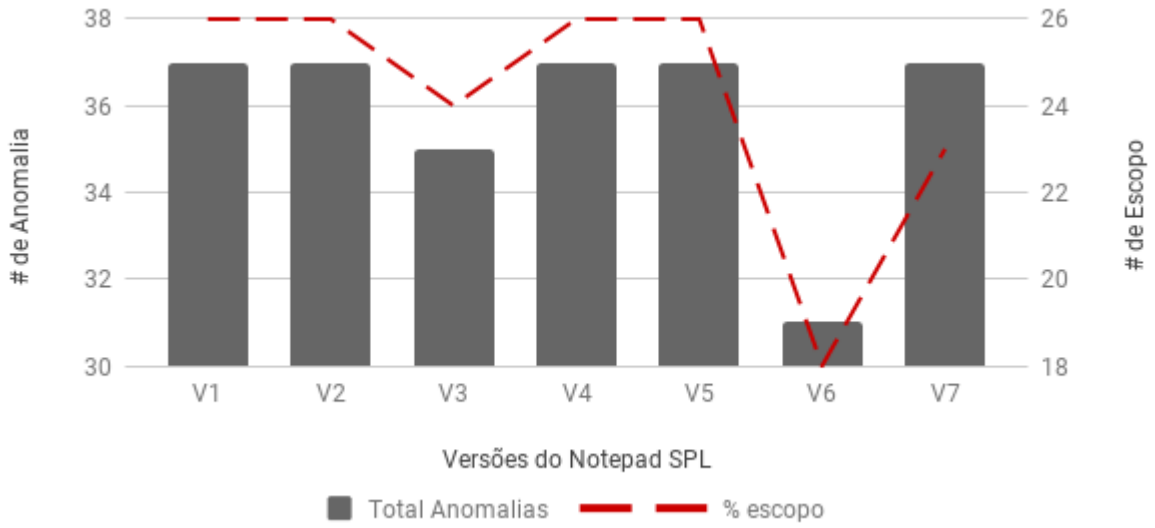
As Figuras 20, 21 e 22, ilustram, respectivamente, para todas as versões do *Notepad, MobileMedia* e *HealthWatcher*, o número de anomalias arquiteturais detectadas e os valores médios dos escopos dos componentes arquiteturais anômalos. Os eixos y primário e y secundário representam o valor total de anomalias e o valores médios dos escopos dos componentes arquiteturais, respectivamente. Como podemos observar, o valor do escopo dos elementos infectados varia de acordo com o número de anomalias. Em geral, a medida que o número de anomalias aumenta, o escopo de seus elementos também aumenta. Este comportamento está associado ao efeito cascata provocado pelas refatorações nos componentes infectados e que possuem um escopo elevado quando comparado com elementos não infectados. Operações de refatoração realizadas em um elemento a fim de tratar uma anomalia tende a se propagar por todos os elementos pertencentes ao seu escopo. Essas mudanças em cascata podem comprometer o sistema, parcial ou completamente, devido a um possível custo e trabalho excessivo. Logo, faz-se interessante ter o total controle do escopo de cada componente para que se possa moldá-lo da melhor maneira possível.

Figura 20 – Escopo de componentes vs Anomalias arquiteturais - *MobileMedia*



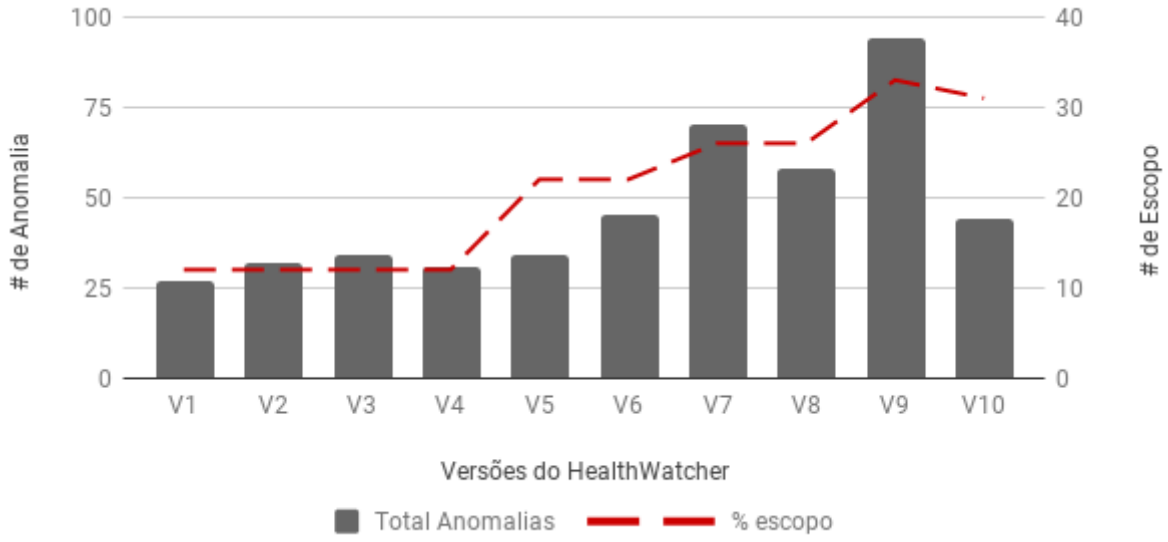

**Fonte**: Autor.

Figura 21 – Escopo de componentes vs Anomalias arquiteturais - *Notepad SPL*




**Fonte**: Autor.

Figura 22 – Escopo de componentes vs Anomalias arquiteturais - *HealthWatcher*




**Fonte**: Autor.

Analisando a relação da coexistência de anomalias e o escopo dos componentes anômalos, acreditamos que a justificativa da alta demanda de operações de refatoração no nível de código seja reflexo de componentes arquiteturais com escopo elevado. Esse comportamento pode ser explicado pois, quando determinado componente tem seu escopo elevado, as operações de refatoração realizadas em seus elementos de código a fim de tratar uma anomalia tende a se propagar. Essa propagação acontece, em geral, em cascata, por todos os elementos pertencentes ao seu escopo, devido ao reflexo do alto índice de interdependência entre os componentes, o que gera um grande número de refatorações no nível de código. Essas mudanças em cascata podem comprometer o sistema, parcial ou completamente, devido a um possível custo e trabalho excessivo. Com base nessa análise, podemos concluir que por meio do escopo podemos priorizar o tratamento de anomalias arquiteturais para que as refatorações no nível de código possam ser melhor controlados, conforme indicado no Capítulo 4.

## 3.4 Resumo

Este capítulo apresentou a relação das anomalias arquiteturais com operações de *refactoring* em sistemas em evolução. Buscou-se analisar se as anomalias poderiam influenciar no número de refatorações do projeto. Contudo, pela análise dos resultados, foi verificado que as anomalias arquiteturais não influenciam diretamente no número de refatorações dos elementos de código correspondentes aos componentes arquiteturais anômalos. Por outro lado, o escopo dos componentes arquiteturais emergiu como forte

indicador que justifica a alta demanda de refatorações no nível de código, conforme discutido no Capítulo 4.

# 4 Avaliação e Discussão

Este capítulo apresenta a aplicação da heurística de priorização de anomalias arquiteturais por meio da análise do projeto arquitetural de três sistemas de *software* em evolução (Capítulo 3). Os objetivos da avaliação e os procedimentos adotados para a realização da mesma são descritos nas Seção 4.1 . A solução computacional para referente a aplicação da heurística é descrita na Seção 4.2. A discussão dos resultados encontrados aparece na Seção 4.3. As ameaças a validação do estudo são apresentada na Seção 4.4. Finalmente, o capítulo é resumido na Seção 4.5.

## 4.1 Objetivos e Procedimentos

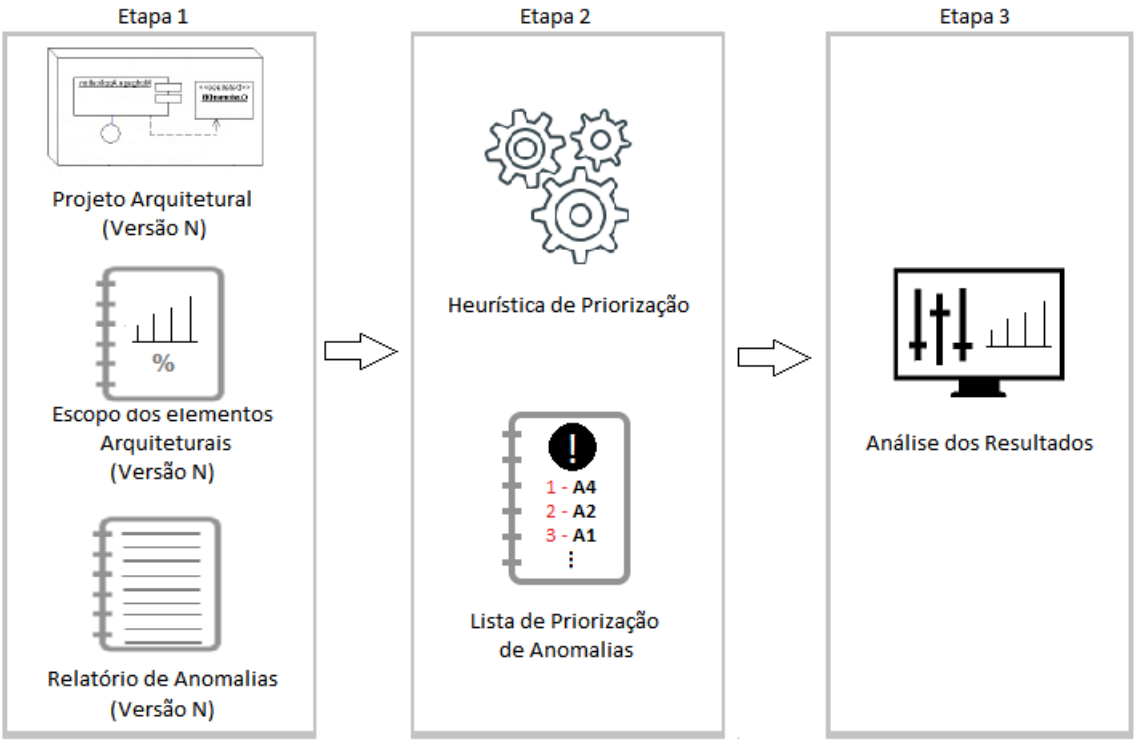
Este estudo tem como objetivo principal verificar a eficácia de aplicação de uma heurística de priorização (Seção 4.2) de anomalias arquiteturais que impactam em operações de refatoração no nível de código. Para tanto pretende-se compreender e analisar a sua aplicabilidade em três sistemas de *software* em evolução: *MobileMedia, Notepad* e *HeathWathcer*.

O propósito é fazer uso da heurística apresentada na Seção 4.2 e, a partir dos projetos arquiteturais das aplicações citadas, indicar a sequência prioritária de tratamento para as anomalias. Anomalias que em geral são responsáveis por demandar um elevado número de operação de refatoração, idealmente devem ser tratadas primeiro.

O estudo não tem como objetivo propor estratégias de tratamento das anomalias que compõem a lista de priorização. Estudos anteriores sugerem o uso de técnicas para o tratamento de anomalias (ABDEEN; SHATA; ERRADI, 2013; VALE *et al.*, 2014)

Este trabalho foi dividido em três fases: (1) medição do escopo de cada um dos elementos arquiteturais dos três sistemas de software em evolução; (2) aplicação da heurística de priorização; e (3) análise da relação entre escopo e a ocorrência de anomalias. A Figura 23 ilustra a relação entre as fases. Conforme pode ser observado na Etapa 1, os projetos arquiteturais juntamente com o relatório de anomalias e o escopo de cada elemento arquitetural são fornecidos à heurística de priorização (Etapa 2) que com base no escopo dos elementos arquiteturais, sugere uma lista de tratamento prioritário para as anomalias detectadas, segundo a ferramenta 4.2. Por fim, os resultados alcançados são analisados (Etapa 3).

Figura 23 – Procedimentos do estudo

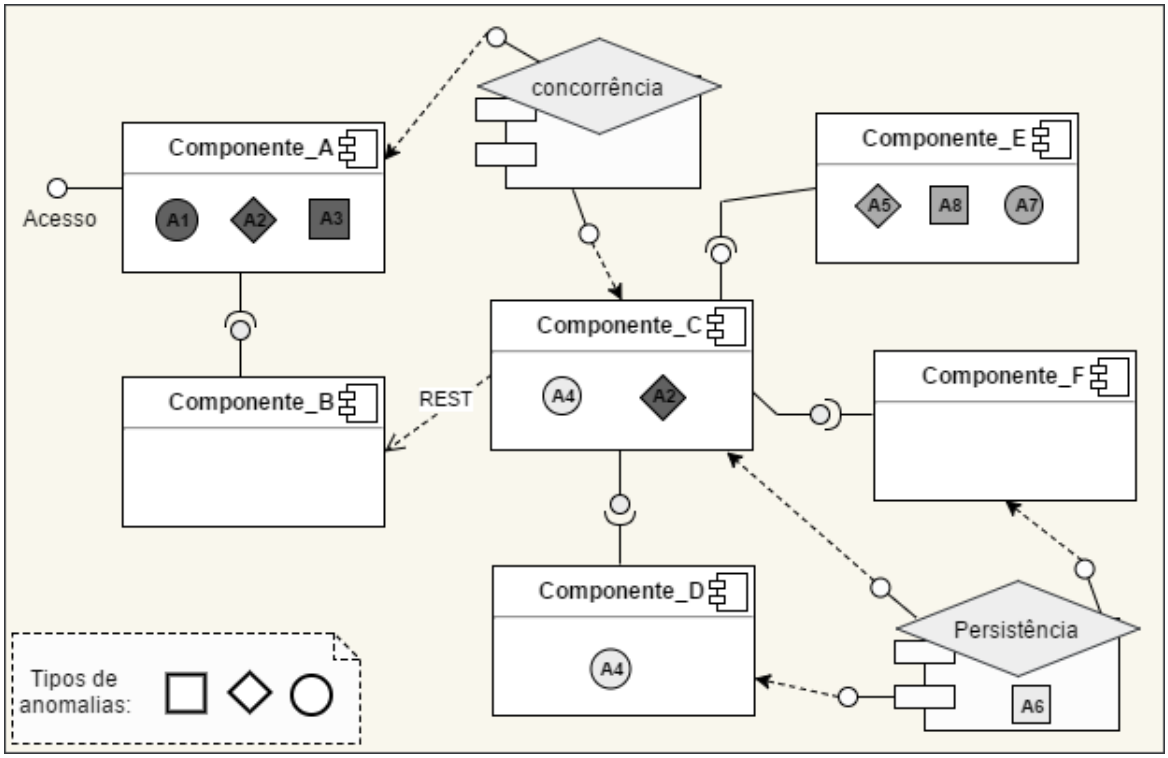



**Fonte**: Autor.

## 4.2 Solução Computacional

O escopo de um elemento refere-se ao conjunto de todos os elementos que dele dependem e é apresentado em termos percentuais. Considerando o exemplo apresentado na Figura 24, o `Componente_B` possui escopo igual a 12,5%. Este percentual equivale ao número de elementos alcançados a partir de `Componente_B` (1 elemento) dividido pelo número total de elementos do projeto. Dito isso, um cenário é dito centrado em escopo quando os elementos nele contido possuem um alto escopo quando comparado com o escopo dos demais elementos.

Figura 24 – Cenário de aplicação das Heurísticas de priorização




**Fonte**: Autor.

Caso o componente arquitetural não tenha dependência alguma, seu escopo será 0. Do contrário, será somado, de maneira recursiva, o escopo de cada um dos componentes que ele depende dividido pelo somatório de componentes do projeto arquitetural. Simplificando, poderíamos dizer que será somado 1 para cada componente que dele dependa diretamente e, em seguida, esse valor é divido pela quantidade total de componentes. A equação 4.1 ilustra o cálculo do escopo (E).

$$E_{c_i} = \begin{cases} 0\ ,\ se\ c_i = \emptyset \\ \frac{\sum_{k=1}^{d} E_{c_{ik}}}{\sum_{j=1}^{t} c_{ij}} \mid c_{in} \notin E_{c_{ik}},\ se\ c_{in} \neq \emptyset \end{cases} \tag{4.1}$$

Onde:

- **d** é o número de dependências diretas do componente arquitetural;
- **t** é o número total de componentes arquiteturais.

Lê-se:

O escopo de um componente $c$ na versão $i$ é igual a zero se o seu conjunto de dependências for vazio; e, caso não seja vazio, é igual ao somatório dos componentes incluídos no conjunto de dependências dividido pelo número total de componentes.

A solução computacional apresentada tem como base uma heurística de escopo baseada no escopo dos elementos arquiteturais, que implementa o cálculo do escopo conforme descrito na equação 4.1. A heurística em questão relaciona as anomalias a serem tratadas levando em consideração o escopo de seus componentes. Quanto maior o escopo de um componente arquitetural infectado, maior prioridade de tratamento ele terá. Analisando a Figura 24 é possível observar que o `Componente_C` alcança três outros componentes (`Componente_D`, `Componente_E` e `Componente_F`). Logo, o escopo de `Componente_C` é igual a 37,5% (número de componentes alcançados dividido pelo número total de elementos). Deste modo, ao aplicar a heurística, tal componente tende a ter um nível de prioridade maior quando comparado com os demais componentes e emerge como componente a ser prioritariamente tratado.

Computacionalmente falando, foi desenvolvido uma solução denominada PriAA (***P**riorization of **A**rchtectural **A**nomalies Tool*). Por meio do PriAA calculamos o escopo de cada componente arquitetural de uma determinada versão de software em evolução. A solução realiza a leitura de um ou mais diagramas arquiteturais para que possa comparar os valores do escopo em cada versão. O PriAA tem como entrada os projetos arquiteturais dos sistemas no formato xml e uma lista de componentes anômalos, detectados pela ferramenta ArchiDES (NASCIMENTO; FONSECA; DANTAS, 2017). Na etapa de processamento, para cada componente identifica-se suas ligações e sua versão correspondente, bem como, calcula-se os valores do escopo. Esse cálculo é realizado por meio de uma função recursiva, a qual percorre todas as possíveis ligações a partir de determinado componente e, a cada novo componente alcançado, o escopo correspondente a aquele componente é incrementado (ver equação 4.1).

A versão corrente do PriAA tem uma interface com o usuário baseado em linha de comando. Isto significa que o usuário interage com o PriAA fornecendo comandos para a aplicação na forma de linhas de texto. Para executar o PriAA a seguinte sequência de comandos é necessária:

```
PriAA /C<ArqProjects> [/B<DirBase>] /L<Alist>
```

Onde:

- **<ArqProjects>** é um parâmetro obrigatório. Ele refere-se aos arquivos contendo os projetos arquiteturais da aplicação de software em formato `xml`. A extensão padrão para o arquivo em questão é .PROJ. A atual verão do PriAA requer que cada versão do sistema de software em evolução seja passada individualmente.

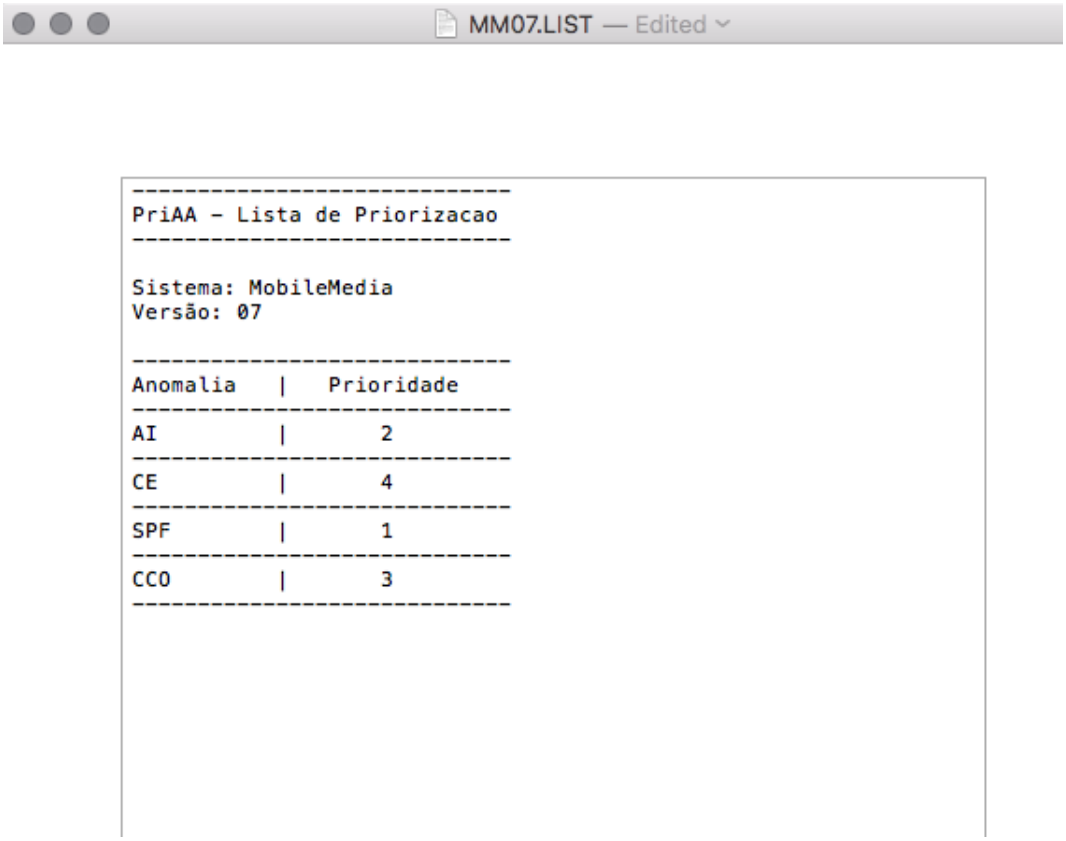
- **<DirBase>** é um parâmetro opcional. Ele refere-se ao nome do diretório de saída. Quando não informado `DirBase` passa a ser o diretório corrente.

- **<Alist>** É um parâmetro obrigatório. Ele refere-se ao nome do arquivo que contem

a lista de componentes anômalos infectados. A extensão padrão deste arquivo é .LIST.

O PriAA fornece como saída um documento de texto contendo os nomes das anomalias a serem tratadas prioritariamente com base nos valores do escopo de cada componente. Caso a consulta tenha sido a mais de uma versão da aplicação, os escopos dos componentes são apresentados separados por versões ordenadas na mesma sequência que foram inseridas na entrada. Um exemplo da saída é mostrado na Figura 25.

Figura 25 – Saída PriAA




**Fonte**: Autor.

## 4.3 Discussão

Esta seção apresenta a priorização de anomalias arquiteturais com base nas informações do escopo de seus componentes, por meio da quantificação apresentada na Seção 3.3. Os dados das métricas de escopo foram coletados manualmente. Porém, o resultado da priorização que tem por base a quantificação do escopo foi determinado de forma automática por meio da solução computacional apresentada na Seção 4.2. A priorização de anomalias arquiteturais contribui para reduzir de forma significativa o número de operações de refatoração dos elementos de código. Apresentamos essa relação na Seção 4.3.1, quando

comparamos nossa priorização do tratamento com o impacto no número de mudanças ao longo da evolução dos sistemas.

### 4.3.1 Priorização das Anomalias

Quanto mais operações de refatoração no código são necessárias para realizar uma evolução, mais instável o projeto se tornará.

Embora a ocorrência de refatoração não esteja diretamente relacionada com a ocorrência de anomalias arquiteturais (Seção 3.2), o escopo dos componentes arquiteturais tende a fornecer melhores indicadores (Seção 4.2). O escopo dos elementos arquiteturais surge consistentemente como uma das propriedades mais significativas para indicar a priorização de anomalias. A Tabela 3, 4 e 5 apresenta a lista de priorização de anomalias para as versões dos projetos *MobileMedia, Notepad* e *HealthWatcher*, com base nos escopos dos elementos arquiteturais. Anomalias com prioridade 1 devem ser tratadas primeiro, enquanto que anomalias com prioridade 5 devem ser deixadas por último no tratamento.

Tabela 7 – Lista de priorização de anomalias - Versão 7 do *MobileMedia*

| **Anomalias detectadas** | **Prioridade de tratamento** |
|---|---|
| Ambigous Interface | 2 |
| Connector Envy | 4 |
| Scattered Parasitic Functionality | 1 |
| Component Concern Overload | 3 |

Tabela 8 – Lista de priorização de anomalias - Versão 3 do *Notepad SPL*

| **Anomalias detectadas** | **Prioridade de tratamento** |
|---|---|
| Ambigous Interface | 1 |
| Connector Envy | 3 |
| Scattered Parasitic Functionality | 2 |
| Component Concern Overload | 4 |

Tabela 9 – Lista de priorização de anomalias - Versão 10 do *HealthWatcher*

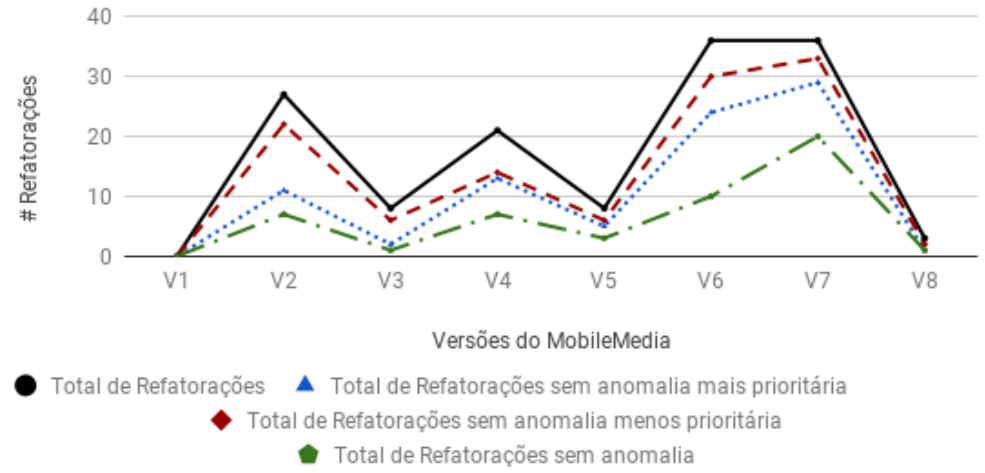
| **Anomalias detectadas** | **Prioridade de tratamento** |
|---|---|
| Ambigous Interface | 2 |
| Connector Envy | 1 |
| Extraneous Adjacent Connector | 4 |
| Scattered Parasitic Functionality | 3 |
| Component Concern Overload | 5 |

A heurística de escopo apresentada na Seção 4.2 está associada a facilidade de mudança de componentes arquiteturais. Tais mudanças, em grande maioria, são reflexo da inclusão de novos requisitos ou adequações de funcionalidades dos sistemas em evolução. Na Figura 24, o `Componente_A` é responsável por fornecer uma interface de acesso e naturalmente tende a sofrer mudanças durante a evolução do sistema, devido as adequações de novas funcionalidades. Dito isso, tal componente pode ser classificado como um componente mais fácil de mudar, comportamento que também é refletido no código.

### 4.3.2 Queda na operação de Refatoração

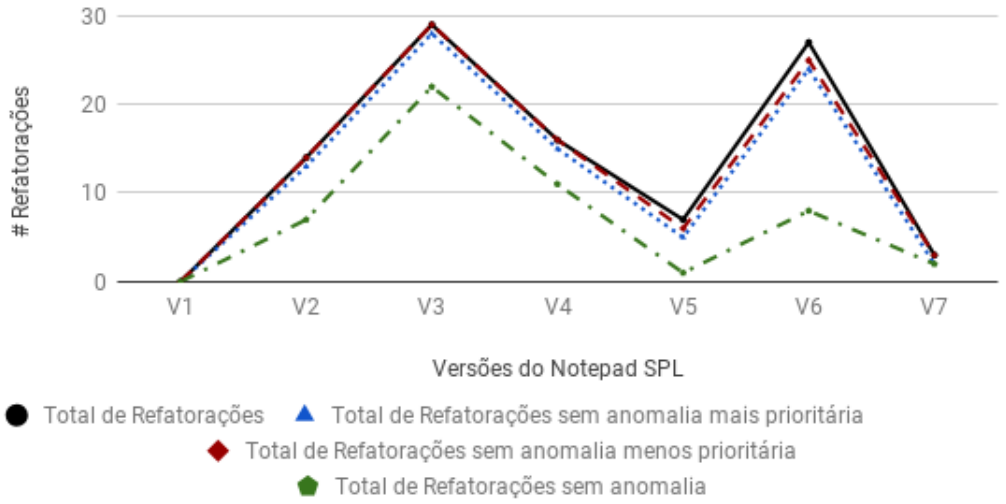
As Figuras 26, 27 e 28 ilustram a variação nas refatorações de código ao longo do processo de evolução das aplicações, considerando o impacto nas operações de refatoração quando somente a anomalia de maior prioridade e menor prioridade são tratadas. Como podemos observar, a queda no número de operações de refatoração varia de acordo com a ordem de tratamento. Para cada uma das aplicações é ilustrado a variação das operações de refatoração, considerando quatro cenários: (i) total de refatorações, (ii) total de refatorações sem considerar a existência de anomalias, (iii) total de refatorações considerando o tratamento da anomalia indicada como prioritária e (iv) total de refatorações considerando o tratamento da anomalia menos prioritária.

Figura 26 – Refatorações de código vs. Tratamento de anomalias - *MobileMedia*



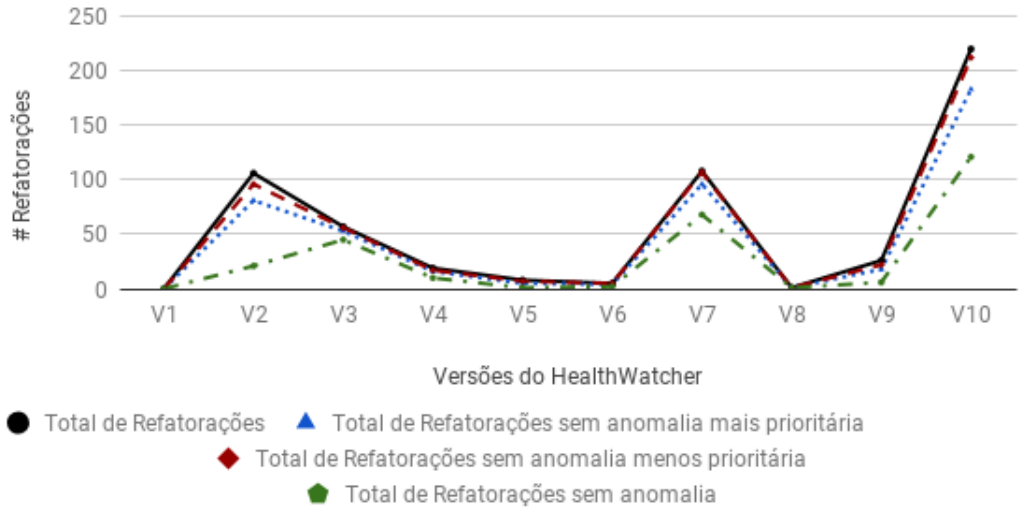

**Fonte**: Autor.

Figura 27 – Refatorações de código vs. Tratamento de anomalias - *Notepad SPL*




**Fonte**: Autor.

Figura 28 – Refatorações de código vs. Tratamento de anomalias - *HealthWatcher*




**Fonte**: Autor.

No caso do MobileMedia (ver Figura 24), versão 7, a anomalia SPF foi indicada como prioridade máxima de tratamento, enquanto a a anomalia CE teve a menor prioridade de tratamento. Analisando a Figura 24 percebe que o tratamento de SPF impacta em um redução de 12% nas operações de refatoração quando comparado com CE. Isto significa que a ordem de tratamento das anomalias possui impactos diferentes nas operações de refatoração do código. Esta variação esta diretamente associado ao impacto de escopo de SPF, que é superior ao escopo de CE.

As Figuras 25 e 26, apresentam o mesmo cenário comparativo para o Notepad e HeathWatcher, respectivamente. Para o Notepad, não percebemos muita variação ao longo da evolução quando olhamos para as curvas de refatoração com o tratamento da anomalia mais e menos prioritária. Isso acontece porque no sistema em questão a anomalia AI se faz constante e as refatorações realizadas estão relacionadas a ela. Então, a anomalia menos prioritária não causa efeito de redução nas operações de refatoração. Já para o HealthWatcher (Figura 26) teve como anomalia de maior prioridade CE e a de menor prioridade SPF. De forma semelhante ao Notepad SPL, o tratamento deve ser realizado na ordem indicada em nossa lista de prioridade.

Desse modo, podemos concluir que a resposta para a nossa questão de pesquisa principal "*A priorização de anomalias arquiteturais contribui para minimizar o número de operações de refatoração dos elementos de código de sistemas de software em evolução?*"é positiva no que diz respeito a relação existente entre operações de refatoração no nível de código e priorização de anomalias arquiteturais.

## 4.4 Ameaça a Validade do Estudo

Estudos desta natureza demandam a discussões das potencias ameaças a validade dos resultados (MAGALHÃES, 2013). A princípio, identificamos em nosso trabalho ameaças à Validade de Construção, Validade de Conclusão, Validade Interna e Validade Externa. Segundo Travassos, Gurov e Amaral (2002), Validade de Construção está relacionada a capacidade de observar os aspectos relevantes para o experimento, Validade Interna, está focada no modo da aplicação dos tratamentos do estudo/experimento, Validade Externa, define as condições que limitam a habilidade de generalizar os resultados do estudo para outros contextos fora do ambiente avaliado e Validade de Conclusão está relacionada à capacidade de chegar a uma conclusão correta a respeito dos resultados obtidos. As ameaças à tais validades são apresentadas a seguir, bem como, os meios para minimiza-las.

**Validade de Construção**. Esta ameaça inclui as métricas que foram utilizadas para recuperar os modelos arquiteturais dos sistemas de *software*. Com a ausência dos modelos arquiteturais originais dos sistemas *MobileMedia, Notepad SPL* e *HealthWatcher*, a ferramenta utilizada podem não gerar os modelos arquiteturais com 100% fiéis, visto que tais ferramentas de recuperação não são 100% precisas. Como forma de minimizar as ameaças a construção, os modelos recuperados do *MobileMedia, Notepad SPL* e *HealthWatcher*, foram guiados pelos trabalhos de Figueiredo *et al.* (2008), Andrade (2013b), UFPE (2017), respectivamente.

**Validade Interna**. As ameaças à validade interna estão relacionadas à indisponibilidade dos desenvolvedores da aplicações objeto de estudo, bem como, a contagem manual das operações de *refactoring*. A ausência dos desenvolvedores originais dos sistemas e a não inclusão de terceiros na contagem manual das refatorações, pode prejudicar as análises e consequentemente, os resultados em relação a influência das anomalias arquiteturais diante das operações de refatoração. Para minimizar essa ameaça, utilizamos uma ferramenta para verificação de mudanças que destacam visualmente as diferenças entre uma versão e outra, dos sistemas. Assim, além de contar manualmente os *refactoring* e cada contagem contabilizar o total de mudanças, pudemos analisar visualmente onde elas aconteceram no projeto.

**Validade Externa**. As ameaças à validade externa estão relacionadas a generalização dos resultados obtidos à outras aplicações de *software*, em outros contextos. Para minimizar a ameaça a generalização dos resultados, escolhemos inicialmente três aplicações de *software* representativas de diferentes contextos. Vale destacar que as aplicações selecionadas foram desenvolvidas por profissionais da indústria de *software*. Foram exaustivamente investigados em diversos estudos acadêmicos. Todavia, os projetos que foram escolhidos permitiram-nos realizar avaliações consideradas úteis para que o trabalho pudesse ser conduzido.

**Validade de Conclusão**. As ameaças à validade a conclusão estão relacionadas a quantidade de aplicações utilizadas para estudo. Devido curto prazo de tempo e a carência de sistemas em evolução disponibilizados pela academia, este estudo envolveu somente 3 aplicações alvo. Portanto, há uma necessidade de realizar este estudo em outros sistemas de domínio e configuração diferente para que possamos analisar se, a não influência das anomalias arquiteturais no número de operações de *refactoring*, persiste.

## 4.5 Resumo

Este capítulo apresenta uma estratégia de solução para a priorização do tratamento de anomalias arquiteturais. O tema em tela foi investigado prioritariamente em nível de código. O desenvolvimento do trabalho conta com a proposição de um heurística de priorização que teve como base o escopo dos elementos arquiteturais, avaliada por meio de três sistemas de *software* em evolução. Este estudo nos leva as conclusões apresentadas no Capítulo 5.

# 5 Conclusão

A priorização do tratamento de anomalias arquiteturais é uma tarefa que deve ser realizada constantemente por uma equipe de desenvolvimento de softwares. Este trabalho relatou uma investigação exploratória sobre a relação existente entre o escopo de componentes arquiteturais infectados e as operações de refatoração de *software* em evolução. Através deste estudo, foi possível analisar a correlação entre anomalias arquiteturais e operações de refatoração no nível de código tendo por base o escopo de cada um. Torna-se evidente que os programadores devem ter ciência do impacto que a não priorização de anomalias podem causar nas operações de refatoração dos sistemas em evolução.

A investigação só foi possível graças à formalização de uma heurística para quantificação do escopo de componentes arquiteturais infectados. Uma vez formalizada, a heuristica serviu de base para a implementação do da solução computacional PriAA. Por meio do PriAA, os desenvolvedores podem fornecer uma lista de priorização para o tratamento de anomalias. Eles poderão gerenciar os valores obtidos de forma eficiente para evitar refatorações indesejadas no código durante a evolução do sistema, uma vez que existe uma correlação entre o escopo dos elementos arquiteturais e as operações de refatoração, ou seja, uma boa gestão do escopo dos componentes arquiteturais propicia uma maior chance de não ocorrer mudanças indesejadas nos elementos de código de sistemas em evolução.

## 5.1 Dificuldades encontradas

Durante o desenvolvimento do trabalho encontramos algumas limitações. Em particular, entendemos que o projeto arquitetural é essencial para que seja possível avaliar impacto de qualquer alteração solicitada e assim facilitar a manutenção do sistema em questão. No entanto, não encontramos aplicações em evolução com a documentação arquitetural disponíveis e adequadas

Supostamente, a documentação formal deve descrever o sistema e, assim, tornar sua compreensão mais fácil para as pessoas que fazem as mudanças. Porém, na prática, identificamos que a documentação formal, principalmente a relativa a projetos, nem sempre existe e quanto existe nem sempre é atualizada, e, portanto, não reflete exatamente o código do programa.

## 5.2 Trabalhos Futuros

Como trabalhos futuros, vislumbramos a inclusão de novas aplicações para validar os resultados aqui apresentados, bem como a sugestão de novas heurísticas que expressem de forma complementar o relacionamento existente entre anomalias arquiteturais e as operações de refatoração. Em particular, espera-se adicionar novas heurísticas que levem em consideração a aglomeração de anomalias em um determinado componente e também a volatilidade das mudanças causadas.

Ainda para trabalhos futuros, poderiam ser investigadas aplicações que não são Linhas de Produto de *Software*, pois essas novas aplicações possuem uma evolução mais fixa, isto é, não tem a flexibilidade de inserção e remoção de componentes durante a evolução do sistema como as LPS. Com uma análise detalhada sobre essas novas aplicações, poderia ser afirmado que o escopo de componentes e refatoração de código estão relacionados tanto em sistemas de LPS como em outros sistemas em evolução comuns.

Também cabe uma análise estatística dos dados coletados. Chegamos a aplicar um teste de correlação. No entanto, o tamanho da amostra limitou a generalização dos resultados. O estudo estatístico mais aprofundado poderia indicar a necessidade de ajustes nas métricas. Finalmente, a evolução da ferramenta tem como meta a inclusão de outros formatos como leitura além do `xml`.

# Referências

ABDEEN, H.; SHATA, O.; ERRADI, A. Software interfaces: On the impact of interface design anomalies. In: IEEE. *Computer Science and Information Technology (CSIT), 2013 5th International Conference on*. [S.l.], 2013. p. 184–193. Citado na página 61.

ALSHARIF, M.; BOND, W. P.; AL-OTAIBY, T. Assessing the complexity of software architecture. In: ACM. *Proceedings of the 42nd annual Southeast regional conference*. [S.l.], 2004. p. 98–103. Citado na página 38.

ANDRADE, H. Identifying architectural bad smells in software product lines. *IDT Workshop on Interesting Results in Computer Science and Engineering*, 2013. Citado 5 vezes nas páginas 17, 24, 26, 31 e 33.

ANDRADE, H. *Software Product Line Architectures: Reviewing the Literature and Identifying Bad Smells*. Dissertação (Mestrado) — *School of Innovation, Design and Engineering* - IDT, 2013. Citado 10 vezes nas páginas 17, 18, 24, 30, 31, 43, 45, 50, 70 e 87.

ANDRADE, H. S. de; ALMEIDA, E.; CRNKOVIC, I. Architectural bad smells in software product lines: An exploratory study. In: ACM. *Proceedings of the WICSA 2014 Companion Volume*. [S.l.], 2014. p. 12. Citado 3 vezes nas páginas 17, 18 e 30.

APEL, S.; KÄSTNER, C. An overview of feature-oriented software development. *Journal of Object Technology*, v. 8, n. 5, p. 49–84, 2009. Citado na página 30.

ARCOVERDE, R.; GARCIA, A.; FIGUEIREDO, E. Understanding the longevity of code smells: preliminary results of an explanatory survey. In: ACM. *Proceedings of the 4th Workshop on Refactoring Tools*. [S.l.], 2011. p. 33–36. Citado 2 vezes nas páginas 30 e 44.

ARCOVERDE, R. *et al.* Prioritization of code anomalies based on architecture sensitiveness. In: IEEE. *Software Engineering (SBES), 2013 27th Brazilian Symposium on*. [S.l.], 2013. p. 69–78. Citado 3 vezes nas páginas 17, 18 e 33.

ARCOVERDE, R. *et al. Automatically Detecting Architecturally-relevant Code Anomalies*. Piscataway, NJ, USA, 2012. 90–91 p. (RSSE '12). ISBN 978-1-4673-1759-7. Disponível em: <http://dl.acm.org/citation.cfm?id=2666719.2666740>. Citado 4 vezes nas páginas 16, 17, 25 e 31.

ASUNCION, H. U.; ASUNCION, A. U.; TAYLOR, R. N. Software traceability with topic modeling. In: *2010 ACM/IEEE 32nd International Conference on Software Engineering*. [S.l.: s.n.], 2010. v. 1, p. 95–104. ISSN 0270-5257. Citado 2 vezes nas páginas 39 e 40.

BARNES, J. M. *Software architecture evolution*. [S.l.], 2013. Citado na página 24.

BASILI, V. R. *Software Modeling and Measurement: The Goal/Question/Metric Paradigm*. [S.l.], 1992. Acessado: 20 de Abril de 2017. Disponível em: <http://hdl.handle.net/1903/7538>. Citado na página 36.

BASILI, V. R.; WEISS, D. M. A methodology for collecting valid software engineering data. *IEEE Transactions on Software Engineering*, SE-10, n. 6, p. 728–738, Nov 1984. ISSN 0098-5589. Citado na página 36.

BENGTSSON, P. Towards maintainability metrics on software architecture: An adaptation of object-oriented metrics. In: *First Nordic Workshop on Software Architecture, Ronneby*. [S.l.: s.n.], 1998. Citado na página 36.

BISZTRAY, D.; HECKEL, R.; EHRIG, H. *Verification of architectural refactoring rules*. [S.l.], 2008. Citado na página 35.

BOURQUIN, F.; KELLER, R. K. High-impact refactoring based on architecture violations. In: IEEE. *Software Maintenance and Reengineering, 2007. CSMR'07. 11th European Conference on*. [S.l.], 2007. p. 149–158. Citado na página 35.

BRCINA, R.; RIEBISCH, M. Architecting for evolvability by means of traceability and features. In: *2008 23rd IEEE/ACM International Conference on Automated Software Engineering - Workshops*. [S.l.: s.n.], 2008. p. 72–81. ISSN 2151-0830. Citado na página 40.

BRONDUM, J.; ZHU, L. Visualising architectural dependencies. In: IEEE PRESS. *Proceedings of the Third International Workshop on Managing Technical Debt*. [S.l.], 2012. p. 7–14. Citado na página 38.

CARPENDALE, S.; GHANAM, Y. *A survey paper on software architecture visualization*. [S.l.], 2008. Citado na página 24.

CHAKI, S. *et al.* Towards engineered architecture evolution. In: *Workshop on Modeling in Software Engineering 2009*. [S.l.: s.n.], 2009. Citado na página 23.

CHAVEZ, C. *et al.* Crosscutting interfaces for aspect-oriented modeling. *Journal of the Brazilian Computer Society*, SciELO Brasil, v. 12, n. 1, p. 43–58, 2006. Citado na página 87.

CHUNG, L.; SUBRAMANIAN, N. Process-oriented metrics for software architecture adaptability. In: IEEE. *Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on*. [S.l.], 2001. p. 310–311. Citado na página 37.

CLELAND-HUANG, J. *et al.* Best practices for automated traceability. *Computer*, v. 40, n. 6, p. 27–35, June 2007. ISSN 0018-9162. Citado na página 40.

COLEMAN, D. *et al.* Using metrics to evaluate software system maintainability. *Computer*, IEEE, v. 27, n. 8, p. 44–49, 1994. Citado na página 37.

CONTE, S. *et al.* A software metrics survey. 1987. Citado na página 35.

CRITCHLOW, M. *et al.* Refactoring product line architectures. *IWR: Achievements, Challenges, and Effects*, p. 23–26, 2003. Citado na página 35.

DURSCKI, R. C. *et al.* Linhas de produto de software: riscos e vantagens de sua implantação. *Simpósio Brasileiro de Processo de Software. S. Paulo*, 2004. Citado na página 30.

EGYED, A. A scenario-driven approach to trace dependency analysis. *IEEE Transactions on Software Engineering*, v. 29, n. 2, p. 116–132, Feb 2003. ISSN 0098-5589. Citado na página 40.

FERREIRA, M. *et al.* Detecting architecturally-relevant code anomalies: a case study of effectiveness and effort. In: ACM. *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. [S.l.], 2014. p. 1158–1163. Citado na página 31.

FERREIRA, M. d. R. *Detecçao de anomalias de código de relevância arquitetural em sistemas multilinguagem*. Dissertação (Mestrado) — PUC-Rio, 2014. Citado 2 vezes nas páginas 31 e 32.

FIGUEIREDO, E. *et al.* Evolving software product lines with aspects: an empirical study on design stability. In: ACM. *Proceedings of the 30th international conference on Software engineering*. [S.l.], 2008. p. 261–270. Citado 5 vezes nas páginas 43, 45, 50, 70 e 86.

FOWLER, M. *Refactoring: improving the design of existing code*. [S.l.]: Addison Wesley Longman, Inc, 1999. 464 p. ISBN 0-201-48567-2. Citado na página 16.

GALL, H.; HAJEK, K.; JAZAYERI, M. Detection of logical coupling based on product release history. In: IEEE. *Software Maintenance, 1998. Proceedings., International Conference on*. [S.l.], 1998. p. 190–198. Citado na página 38.

GALVAO, I.; GOKNIL, A. Survey of traceability approaches in model-driven engineering. In: *11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007)*. [S.l.: s.n.], 2007. p. 313–313. ISSN 1541-7719. Citado na página 40.

GARCIA, J. *et al.* Identifying architectural bad smells. In: IEEE COMPUTER SOCIETY. *Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on*. [S.l.], 2009. p. 255–258. Citado 9 vezes nas páginas 18, 25, 26, 27, 28, 29, 30, 33 e 50.

GARCIA, J. *et al.* Toward a catalogue of architectural bad smells. In: SPRINGER-VERLAG. *Proceedings of the 5th International Conference on the Quality of Software Architectures: Architectures for Adaptive Software*. [S.l.], 2009. p. 146–162. Citado 7 vezes nas páginas 16, 25, 26, 28, 33, 44 e 49.

GHABI, A.; EGYED, A. Exploiting traceability uncertainty between architectural models and code. In: *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*. [S.l.: s.n.], 2012. p. 171–180. Citado na página 40.

GOSEVA-POPSTOJANOVA, K. *et al.* Architectural-level risk analysis using uml. *IEEE transactions on software engineering*, IEEE, v. 29, n. 10, p. 946–960, 2003. Citado na página 38.

GUIMARAES, E. *et al.* Prioritizing software anomalies with software metrics and architecture blueprints. In: IEEE. *Modeling in Software Engineering (MiSE), 2013 5th International Workshop on*. [S.l.], 2013. p. 82–88. Citado 3 vezes nas páginas 17, 18 e 33.

GUIMARÃES, E. T. *A Blueprint-Based Approach for Prioritizing and Ranking Critical Code Anomalies*. Tese (Doutorado) — PUC-Rio, 2014. Citado na página 17.

HANENBERG, S.; OBERSCHULTE, C.; UNLAND, R. Refactoring of aspect-oriented software. In: *4th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World (Net. ObjectDays)*. [S.l.: s.n.], 2003. p. 19–35. Citado 2 vezes nas páginas 34 e 35.

HIRAMA, K. *Engenharia de software: Qualidade e produtividade com tecnologia*. [S.l.]: Elsevier Editora Ltda, 2012. Citado na página 16.

HITZ, M.; MONTAZERI, B. Measuring coupling and cohesion in object-oriented systems. na, 1995. Citado na página 37.

JAZAYERI, M. On architectural stability and evolution. In: _____. *Reliable Software Technologies — Ada-Europe 2002: 7th Ada-Europe International Conference on Reliable Software Technologies Vienna, Austria, June 17–21, 2002 Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002. p. 13–23. ISBN 978-3-540-48046-4. Disponível em: <http://dx.doi.org/10.1007/3-540-48046-3_2>. Citado 2 vezes nas páginas 23 e 24.

KÄKÖLÄ, T.; DUEÑAS, J. C. *Software product lines*. [S.l.]: Springer, 2006. Citado na página 23.

KAUR, M.; KUMAR, P. Spotting the phenomenon of bad smells in mobilemedia product line architecture. In: IEEE. *Contemporary Computing (IC3), 2014 Seventh International Conference on*. [S.l.], 2014. p. 357–363. Citado 2 vezes nas páginas 17 e 18.

KROGMANN, K. *Reconstruction of software component architectures and behaviour models using static and dynamic analysis*. Dissertação (Mestrado) — Karlsruhe Institute of Technology - KIT, 2012. Citado 2 vezes nas páginas 45 e 82.

LAND, R. Measurements of software maintainability. In: *Proceedings of the 4th ARTES Graduate Student Conference*. [S.l.: s.n.], 2002. Citado na página 37.

LEHMAN, M. M. Laws of software evolution revisited. In: SPRINGER. *European Workshop on Software Process Technology*. [S.l.], 1996. p. 108–124. Citado 2 vezes nas páginas 16 e 23.

LEIGH, A.; WERMELINGER, M.; ZISMAN, A. An evaluation of design rule spaces as risk containers. In: IEEE. *Software Architecture (WICSA), 2016 13th Working IEEE/IFIP Conference on*. [S.l.], 2016. p. 295–298. Citado 2 vezes nas páginas 38 e 39.

LIPPERT M.; ROOCK, S. *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*. [S.l.]: John Wiley, 2006. Citado na página 25.

MACIA, I. *et al.* Supporting the identification of architecturally-relevant code anomalies. In: IEEE. *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. [S.l.], 2012. p. 662–665. Citado 3 vezes nas páginas 24, 31 e 32.

MACIA, I. *et al.* Enhancing the detection of code anomalies with architecture-sensitive strategies. In: IEEE. *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*. [S.l.], 2013. p. 177–186. Citado 2 vezes nas páginas 16 e 17.

MACIA, I. *et al.* On the impact of aspect-oriented code smells on architecture modularity: An exploratory study. In: IEEE COMPUTER SOCIETY. *Software Components, Architectures and Reuse (SBCARS), 2011 Fifth Brazilian Symposium on*. [S.l.], 2011. p. 41–50. Citado 6 vezes nas páginas 24, 25, 28, 29, 30 e 44.

MACIA, I. *et al.* Are automatically-detected code anomalies relevant to architectural modularity?: an exploratory analysis of evolving systems. In: ACM. *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*. [S.l.], 2012. p. 167–178. Citado 5 vezes nas páginas 16, 24, 30, 32 e 33.

MAGALHÃES, A. Estudo potencial de colaboração. 2013. Citado na página 70.

MEANANEATRA, P.; RONGVIRIYAPANISH, S.; APIWATTANAPONG, T. Using software metrics to select refactoring for long method bad smell. In: IEEE. *Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON), 2011 8th International Conference on*. [S.l.], 2011. p. 492–495. Citado na página 37.

MELO, S. M. Inspeção de software. *University of São Paulo: São Carlos, SP*, 2009. Citado na página 31.

MENS, T.; TOURWÉ, T. A survey of software refactoring. *IEEE Transactions on software engineering*, IEEE, v. 30, n. 2, p. 126–139, 2004. Citado na página 34.

MIRAKHORLI, M.; CLELAND-HUANG, J. Using tactic traceability information models to reduce the risk of architectural degradation during system maintenance. In: IEEE. *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*. [S.l.], 2011. p. 123–132. Citado na página 40.

MO, R. *et al.* Decoupling level: A new metric for architectural maintenance complexity. In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. [S.l.: s.n.], 2016. p. 499–510. Citado na página 39.

MURTA, L. G. P.; HOEK, A. V. D.; WERNER, C. M. L. Archtrace: Policy-based support for managing evolving architecture-to-implementation traceability links. In: *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. [S.l.: s.n.], 2006. p. 135–144. ISSN 1938-4300. Citado 2 vezes nas páginas 40 e 41.

NASCIMENTO, R. J. O. d.; FONSECA, C. A. G.; DANTAS, F. Using expert systems for investigating the impact of architectural anomalies on software reuse. *IEEE Latin America Transactions*, IEEE, v. 15, n. 2, p. 374–379, 2017. Citado 10 vezes nas páginas 17, 18, 31, 32, 33, 44, 45, 49, 64 e 82.

NGUYEN, T. N.; MUNSON, E. V.; THAO, C. Object-oriented configuration management technology can improve software architectural traceability. In: *Third ACIS Int'l Conference on Software Engineering Research, Management and Applications (SERA'05)*. [S.l.: s.n.], 2005. p. 86–93. Citado 2 vezes nas páginas 39 e 40.

NOVAIS, R.; SANTOS, J. A.; MENDONÇA, M. Experimentally assessing the combination of multiple visualization strategies for software evolution analysis. *Journal of Systems and Software*, p. –, 2017. ISSN 0164-1212. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0164121217300572>. Citado 2 vezes nas páginas 23 e 24.

PADILHA, J. *Detecção de anomalias de código usando métricas de software*. Dissertação (Mestrado) — UNIVERSIDADE FEDERAL de Minas Gerais - UFMG, 2013. Citado 4 vezes nas páginas 16, 17, 36 e 37.

PINTO, F.; COSTA, H. Melhoria da qualidade da estrutura interna de sistemas de software por redução do nível de acoplamento entre pacotes. *Simpósio Brasileiro de Qualidade de Software*, p. 194–208, 2014. Citado na página 38.

PRESSMAN, R. S. *SOFTWARE ENGINEERING: A PRACTITIONER'S APPROACH, SEVENTH EDITION*. [S.l.]: McGraw-Hill, 2010. 895 p. ISBN 978–0–07–337597–7. Citado na página 36.

RAJLICH, V.; SILVA, J. H. Evolution and reuse of orthogonal architecture. *IEEE Transactions on Software Engineering*, IEEE Computer Society, v. 22, n. 2, p. 153–157, 1996. Citado na página 23.

SAMARTHYAM, G.; SURYANARAYANA, G.; SHARMA, T. Refactoring for software architecture smells. In: ACM. *Proceedings of the 1st International Workshop on Software Refactoring*. [S.l.], 2016. p. 1–4. Citado 3 vezes nas páginas 16, 22 e 35.

SHAH, S. M. A. Formalization of architectural refactorings. 2008. Citado na página 16.

SILVA, A. *et al.* Linhas de produto de software: Uma tendência da indústria. *V Encontro Regional de Informática Ceará-Piauí (ERCEMAPI 2011), Cap*, v. 1, 2011. Citado na página 24.

SILVA, L.; KULESZA, U. Um framework para visualização da evolução arquitetural de sistemas de software baseado em cenários de casos de uso. *ANAIS| PROCEEDINGS*, p. 77, 2016. Citado na página 24.

SILVA, L. d.; BALASUBRAMANIAM, D. Controlling software architecture erosion: A survey. *Journal of Systems and Software*, Elsevier, v. 85, n. 1, p. 132–151, 2012. Citado na página 16.

SIMON, F.; STEINBRUCKNER, F.; LEWERENTZ, C. Metrics based refactoring. In: IEEE. *Software Maintenance and Reengineering, 2001. Fifth European Conference on*. [S.l.], 2001. p. 30–38. Citado na página 37.

SOMMERVILLE, I. *et al. Engenharia de software. Traduzido por Kalinka Oliveira Ivan Bosnic*. [S.l.]: Addison Wesley, 2011. ISBN 978-85-7936-108-1. Citado 5 vezes nas páginas 16, 22, 23, 36 e 37.

SOURCEGEAR, L. *DiffMerge*. 2016. Acessado: 19 de Agosto de 2016. Disponível em: <https://sourcegear.com/diffmerge/>. Citado na página 49.

SOUSA, E. P. de *et al.* Arquitetura de aplicações spring mvc: Uma análise baseada no acoplamento lógico. *V Workshop on Software Visualization, Evolution and Maintenance*, 2017. Citado na página 38.

SRIVISUT, K.; MUENCHAISRI, P. Bad-smell metrics for aspect-oriented software. In: IEEE. *Computer and Information Science, 2007. ICIS 2007. 6th IEEE/ACIS International Conference on*. [S.l.], 2007. p. 1060–1065. Citado na página 37.

STARON, M. *et al.* Identifying implicit architectural dependencies using measures of source code change waves. In: *2013 39th Euromicro Conference on Software Engineering and Advanced Applications*. [S.l.: s.n.], 2013. p. 325–332. ISSN 1089-6503. Citado 2 vezes nas páginas 38 e 39.

SUBRAMANIAN, N.; CHUNG, L. Metrics for software adaptability. *Proc. Software Quality Management (SQM 2001)*, Citeseer, p. 18–20, 2001. Citado na página 37.

TEKINERDOGAN, B.; HOFMANN, C.; AKSIT, M. Modeling traceability of concerns for synchronizing architectural views. *journal of object technology*, ETH Swiss Federal Institute of Technology, v. 6, n. 7, p. 7–25, 2007. Citado na página 40.

THOMAS, S. W. *et al.* Studying software evolution using topic models. *Science of Computer Programming*, v. 80, Part B, p. 457 – 479, 2014. ISSN 0167-6423. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0167642312001621>. Citado 2 vezes nas páginas 23 e 24.

TONU, S. A.; ASHKAN, A.; TAHVILDARI, L. Evaluating architectural stability using a metric-based approach. In: IEEE. *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*. [S.l.], 2006. p. 10–pp. Citado na página 36.

TRAVASSOS, G. H.; GUROV, D.; AMARAL, E. *Introdução à engenharia de software experimental*. [S.l.]: UFRJ, 2002. Citado na página 70.

UFPE, U. F. de P. *Health Watcher*. 2017. Acessado: 30 de abril de 2017. Disponível em: <http://www.cin.ufpe.br/~scbs/testbed/architecture/>. Citado 5 vezes nas páginas 43, 45, 50, 70 e 88.

VALE, G. *et al.* Bad smells in software product lines: A systematic review. In: IEEE COMPUTER SOCIETY. *Software Components, Architectures and Reuse (SBCARS), 2014 Eighth Brazilian Symposium on*. [S.l.], 2014. p. 84–94. Citado 6 vezes nas páginas 18, 24, 25, 30, 33 e 61.

VIDAL, S. *et al.* Identifying architectural problems through prioritization of code smells. In: IEEE. *Software Components, Architectures and Reuse (SBCARS), 2016 X Brazilian Symposium on*. [S.l.], 2016. p. 41–50. Citado na página 33.

WANG, X. j. Metrics for evaluating coupling and service granularity in service oriented architecture. In: *2009 International Conference on Information Engineering and Computer Science*. [S.l.: s.n.], 2009. p. 1–4. ISSN 2156-7379. Citado 2 vezes nas páginas 38 e 39.

ZIMMERMANN, O. Architectural refactoring: A task-centric view on software evolution. *IEEE Software*, IEEE, v. 32, n. 2, p. 26–29, 2015. Citado na página 35.

# Apêndices

# APÊNDICE A – Valores métricos utilizadas no SoMoX para recuperação arquitetural

Este apêndice descreve informações referentes as métricas que foram utilizadas para guiar os procedimentos de recuperação arquitetural das aplicações alvo deste trabalho: *MobileMedia*, *Notepad* e *HealthWatcher*.

A ferramenta SoMoX, é composta por métricas que são utilizadas com objetivo de recuperar modelos arquiteturais a partir do código-fonte. Esssas métricas são ajustadas a partir de valores numéricos correspondentes a pesos. Mais informações sobre as métricas podem ser encontradas no trabalho de Krogmann (2012).

Os ajustes das métricas para recuperação dos modelos arquiteturais foram guiados pelo trabalho de Nascimento, Fonseca e Dantas (2017). As Tabelas de 10 à 12 descrevem os valores para as métricas que foram ajustadas durante a recuperação arquitetural de 8 versões do *MobileMedia*. A Tabela 13 descreve os valores que foram utilizados para a recuperação dos modelos arquiteturais das 7 versões do *Notepad SPL*. As Tabelas 14 e 15, descrevem os valores que foram utilizados para a recuperação dos modelos arquiteturais das 10 versões do *HealthWatcher*.

Tabela 10 – Valores métricos aplicados nas versões 1, 2 e 3 do *MobileMedia*

| **Métricas** | **Pesos** |
|---|---|
| Clustering Merge Threshold Min | 99 |
| Clustering Merge Threshold Max | 100 |
| Clustering Merge Threshold Increment | 10 |
| Merge: Interface Violation | 25 |
| Clustering Composition Threshold Min | 70 |
| Clustering Composition Threshold Max | 100 |
| Clustering Composition Threshold Decrement | 10 |
| Composition: Interface Adherence | 100 |
| Package Mapping | 100 |
| Diretory Mapping | 90 |
| Hightest Name Resemblace | 90 |
| High Name Resemblace | 90 |
| Mid Name Resemblace | 50 |
| Low Name Resemblace | 35 |
| High SLAQ (Slice Layer Architecture Quality) | 5 |
| Low SLAQ | 5 |
| DMS (Distance from the Main Sequence) | 5 |
| High Coupling | 25 |
| Low Coupling | 0 |

Tabela 11 – Valores métricos aplicados nas versões 4, 5 e 6 do *MobileMedia*

| **Métricas** | **Pesos** |
|---|---|
| Clustering Merge Threshold Min | 99 |
| Clustering Merge Threshold Max | 100 |
| Clustering Merge Threshold Increment | 10 |
| Merge: Interface Violation | 20 |
| Clustering Composition Threshold Min | 70 |
| Clustering Composition Threshold Max | 100 |
| Clustering Composition Threshold Decrement | 10 |
| Composition: Interface Adherence | 100 |
| Package Mapping | 100 |
| Diretory Mapping | 90 |
| Hightest Name Resemblace | 90 |
| High Name Resemblace | 90 |
| Mid Name Resemblace | 50 |
| Low Name Resemblace | 35 |
| High SLAQ (Slice Layer Architecture Quality) | 5 |
| Low SLAQ | 5 |
| DMS (Distance from the Main Sequence) | 5 |
| High Coupling | 50 |
| Low Coupling | 10 |

Tabela 12 – Valores métricos aplicados nas versões 7 e 8 do *MobileMedia*

| **Métricas** | **Pesos** |
|---|---|
| Clustering Merge Threshold Min | 99 |
| Clustering Merge Threshold Max | 100 |
| Clustering Merge Threshold Increment | 10 |
| Merge: Interface Violation | 30 |
| Clustering Composition Threshold Min | 60 |
| Clustering Composition Threshold Max | 100 |
| Clustering Composition Threshold Decrement | 10 |
| Composition: Interface Adherence | 100 |
| Package Mapping | 100 |
| Diretory Mapping | 90 |
| Hightest Name Resemblace | 90 |
| High Name Resemblace | 90 |
| Mid Name Resemblace | 50 |
| Low Name Resemblace | 35 |
| High SLAQ (Slice Layer Architecture Quality) | 5 |
| Low SLAQ | 5 |
| DMS (Distance from the Main Sequence) | 5 |
| High Coupling | 25 |
| Low Coupling | 0 |

Tabela 13 – Valores métricos aplicados em todas versões do *Notepad SPL*

| **Métricas** | **Pesos** |
|---|---|
| Clustering Merge Threshold Min | 70 |
| Clustering Merge Threshold Max | 100 |
| Clustering Merge Threshold Increment | 45 |
| Merge: Interface Violation | 70 |
| Clustering Composition Threshold Min | 45 |
| Clustering Composition Threshold Max | 100 |
| Clustering Composition Threshold Decrement | 15 |
| Composition: Interface Adherence | 90 |
| Package Mapping | 60 |
| Diretory Mapping | 90 |
| Hightest Name Resemblace | 60 |
| High Name Resemblace | 30 |
| Mid Name Resemblace | 15 |
| Low Name Resemblace | 5 |
| High SLAQ (Slice Layer Architecture Quality) | 40 |
| Low SLAQ | 10 |
| DMS (Distance from the Main Sequence) | 10 |
| High Coupling | 50 |
| Low Coupling | 50 |

Tabela 14 – Valores métricos aplicados nas versões 1, 2, 3 e 4 do *HealthWatcher*

| **Métricas** | **Pesos** |
|---|---|
| Clustering Merge Threshold Min | 70 |
| Clustering Merge Threshold Max | 100 |
| Clustering Merge Threshold Increment | 45 |
| Merge: Interface Violation | 35 |
| Clustering Composition Threshold Min | 50 |
| Clustering Composition Threshold Max | 100 |
| Clustering Composition Threshold Decrement | 15 |
| Composition: Interface Adherence | 90 |
| Package Mapping | 60 |
| Diretory Mapping | 90 |
| Hightest Name Resemblace | 60 |
| High Name Resemblace | 30 |
| Mid Name Resemblace | 15 |
| Low Name Resemblace | 5 |
| High SLAQ (Slice Layer Architecture Quality) | 40 |
| Low SLAQ | 10 |
| DMS (Distance from the Main Sequence) | 10 |
| High Coupling | 30 |
| Low Coupling | 5 |

Tabela 15 – Valores métricos aplicados nas versões 5, 6, 7, 8, 9 e 10 do *HealthWatcher*

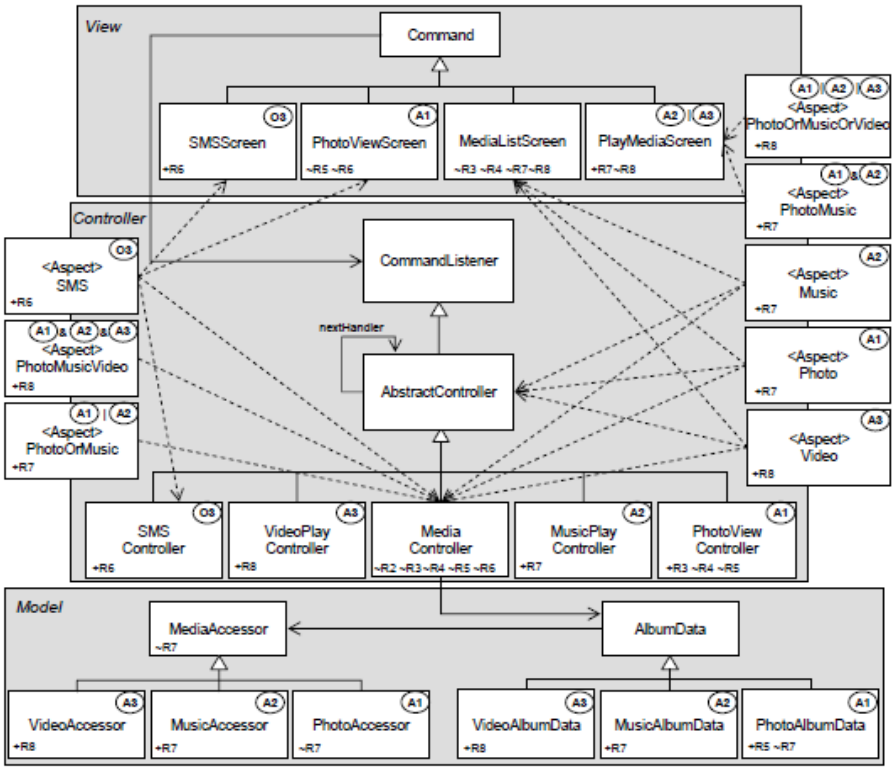
| **Métricas** | **Pesos** |
|---|---|
| Clustering Merge Threshold Min | 99 |
| Clustering Merge Threshold Max | 100 |
| Clustering Merge Threshold Increment | 10 |
| Merge: Interface Violation | 10 |
| Clustering Composition Threshold Min | 70 |
| Clustering Composition Threshold Max | 100 |
| Clustering Composition Threshold Decrement | 10 |
| Composition: Interface Adherence | 100 |
| Package Mapping | 100 |
| Diretory Mapping | 90 |
| Hightest Name Resemblace | 90 |
| High Name Resemblace | 90 |
| Mid Name Resemblace | 50 |
| Low Name Resemblace | 35 |
| High SLAQ (Slice Layer Architecture Quality) | 5 |
| Low SLAQ | 5 |
| DMS (Distance from the Main Sequence) | 5 |
| High Coupling | 45 |
| Low Coupling | 5 |

# APÊNDICE B – Modelos arquiteturais das aplicações alvo

Este apêndice descreve informações referentes aos modelos arquiteturais das aplicações alvo deste trabalho: *MobileMedia*, *Notepad* e *HealthWatcher*.

A Figura 29 ilustra o modelo arquitetural do *MobileMedia*, baseado no padrão MVC (*Model-View-Controller*), onde, a camada *View* é responsável pelas funcionalidades de interação entre usuário e o sistema; a camada *Controller* pelo o controle de fluxo de dados e a camada *Model* responsável pelo armazenamento de dados. Os módulos que estão orientados à aspectos não serão considerados neste estudo.

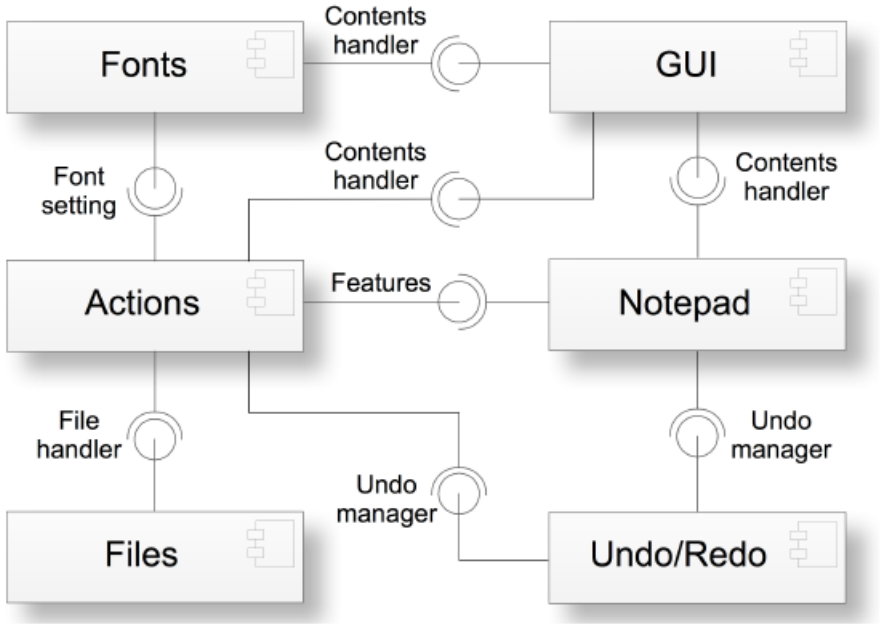
Figura 29 – Modelo Arquitetural do *MobileMedia*.




**Fonte**:(FIGUEIREDO *et al.*, 2008).

A Figura 30 ilustra o modelo arquitetural do *Notepad SPL*, baseado em componentes. Conforme pode ser observado, o modelo possui 6 componentes (*GUI, Fonts, Actions, Files, Undo/Redo, Notepad*). Observando os componentes do sistemas, é possível entender seus relacionamentos. Por exemplo, o componente *Action* é um componente que requisita serviços aos demais componentes, fornecendo serviços somente ao componente *Notepad*.

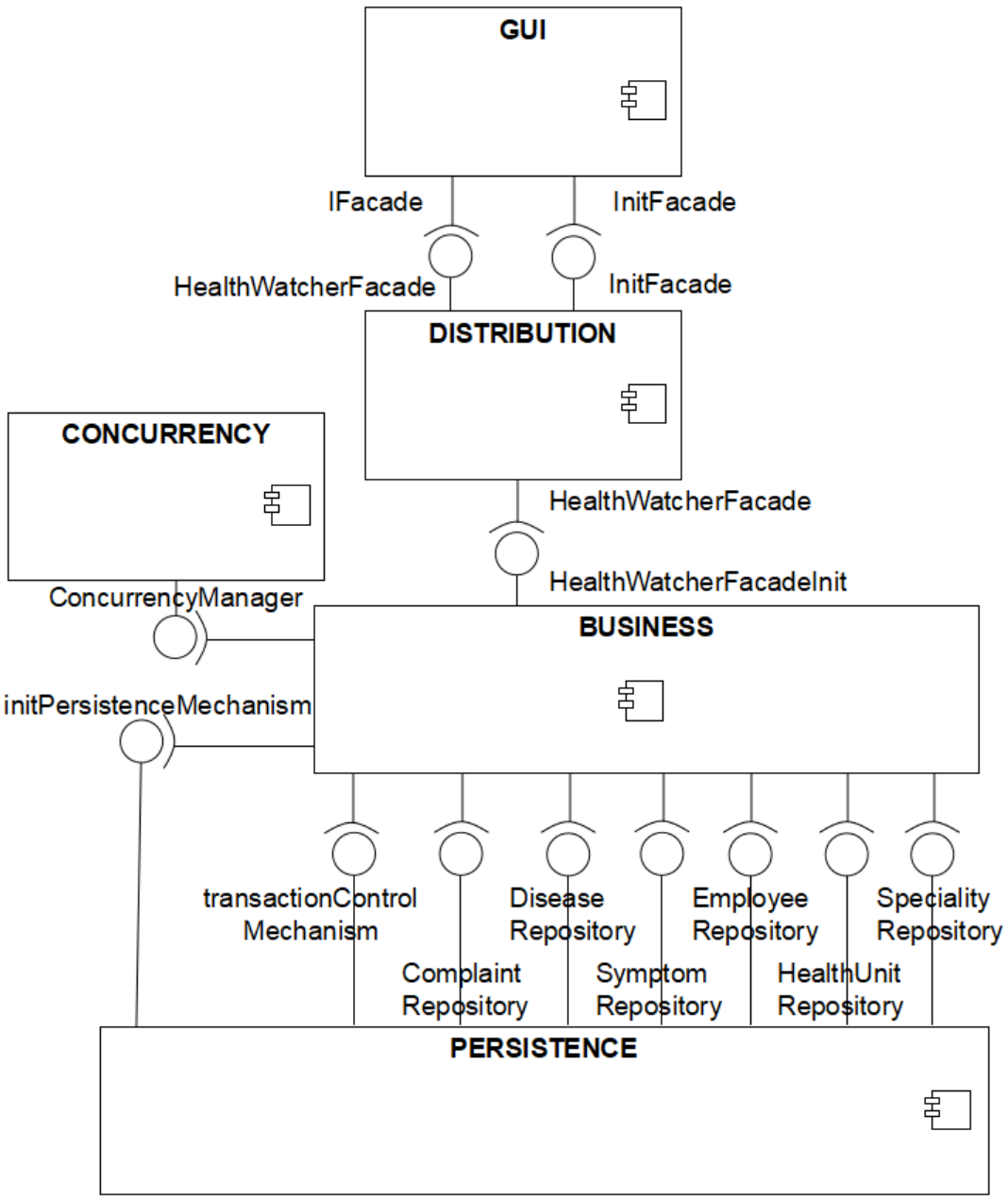
Figura 30 – Modelo Arquitetural Baseado em Componentes do *Notepad SPL*.




**Fonte**:(ANDRADE, 2013b).

A Figura 31 ilustra o modelo arquitetural do sistema *HealthWatcher*, baseado em componentes. Como pode ser observado, existem 5 componentes funcionais (CHAVEZ *et al.*, 2006): (i) *GUI*, responsável por receber os dados do usuário; (ii) *BUSINESS*, responsável por prover serviços de lógica de negócio, (iii) *DISTRIBUTION*, responsável por prover serviços para o componente *GUI*; (iv) *CONCORRENCY*, responsável por realizar diferentes estratégias de concorrência aos componente *BUSINESS* e (v) *PERSISTENCE*, responsável por mapear a implementação de vários problemas transversais como, gerenciamento de transações e atualização de dados para o componente *BUSINESS*.

Figura 31 – Modelo Arquitetural Baseado em Componentes do sistema *HealthWatcher*.




**Fonte**:(UFPE, 2017).

# APÊNDICE C – Anomalias arquiteturais e priorização de tratamento

Este apêndice apresenta informações referentes a lista de priorização de anomalias para cada versão das aplicações alvo deste trabalho: *MobileMedia*, *Notepad* e *HealthWatcher*.

As Tabelas de 16 à 23 apresenta a lista de priorização de anoamlias para as versões da aplicação *MobileMedia*. As Tabelas de 24 à 30 apresenta a lista de priorização de anoamlias para as versões da aplicação *Notepad SPL*. As Tabelas de 31 à 40 apresenta a lista de priorização de anoamlias para as versões da aplicação *HealthWatcher*.

Tabela 16 – Lista de priorização de anomalias - Versão 1 do *MobileMedia*

| **Anomalias detectadas** | **Prioridade de tratamento** |
|---|---|
| Ambigous Interface | 1 |
| Connector Envy | 3 |
| Scattered Parasitic Functionality | 2 |
| Component Concern Overload | 4 |

Tabela 17 – Lista de priorização de anomalias - Versão 2 do *MobileMedia*

| **Anomalias detectadas** | **Prioridade de tratamento** |
|---|---|
| Ambigous Interface | 2 |
| Connector Envy | 1 |
| Scattered Parasitic Functionality | 3 |
| Component Concern Overload | 4 |

Tabela 18 – Lista de priorização de anomalias - Versão 3 do *MobileMedia*

| **Anomalias detectadas** | **Prioridade de tratamento** |
|---|---|
| Ambigous Interface | 1 |
| Connector Envy | 2 |
| Scattered Parasitic Functionality | 3 |
| Component Concern Overload | 4 |

Tabela 19 – Lista de priorização de anomalias - Versão 4 do *MobileMedia*

| **Anomalias detectadas** | **Prioridade de tratamento** |
|---|---|
| Ambigous Interface | 4 |
| Connector Envy | 2 |
| Scattered Parasitic Functionality | 1 |
| Component Concern Overload | 3 |

Tabela 20 – Lista de priorização de anomalias - Versão 5 do *MobileMedia*

| **Anomalias detectadas** | **Prioridade de tratamento** |
|---|---|
| Ambigous Interface | 2 |
| Connector Envy | 1 |
| Scattered Parasitic Functionality | 3 |
| Component Concern Overload | 4 |

Tabela 21 – Lista de priorização de anomalias - Versão 6 do *MobileMedia*

| **Anomalias detectadas** | **Prioridade de tratamento** |
|---|---|
| Ambigous Interface | 2 |
| Connector Envy | 1 |
| Scattered Parasitic Functionality | 4 |
| Component Concern Overload | 3 |

Tabela 22 – Lista de priorização de anomalias - Versão 7 do *MobileMedia*

| **Anomalias detectadas** | **Prioridade de tratamento** |
|---|---|
| Ambigous Interface | 2 |
| Connector Envy | 4 |
| Scattered Parasitic Functionality | 1 |
| Component Concern Overload | 3 |

Tabela 23 – Lista de priorização de anomalias - Versão 8 do *MobileMedia*

| **Anomalias detectadas** | **Prioridade de tratamento** |
|---|---|
| Ambigous Interface | 2 |
| Connector Envy | 4 |
| Scattered Parasitic Functionality | 1 |
| Component Concern Overload | 2 |

Tabela 24 – Lista de priorização de anomalias - Versão 1 do *Notepad SPL*

| **Anomalias detectadas** | **Prioridade de tratamento** |
|---|---|
| Ambigous Interface | 1 |
| Connector Envy | 3 |
| Scattered Parasitic Functionality | 2 |
| Component Concern Overload | 4 |

Tabela 25 – Lista de priorização de anomalias - Versão 2 do *Notepad SPL*

| **Anomalias detectadas** | **Prioridade de tratamento** |
|---|---|
| Ambigous Interface | 1 |
| Connector Envy | 2 |
| Scattered Parasitic Functionality | 3 |
| Component Concern Overload | 4 |

Tabela 26 – Lista de priorização de anomalias - Versão 3 do *Notepad SPL*

| **Anomalias detectadas** | **Prioridade de tratamento** |
|---|---|
| Ambigous Interface | 1 |
| Connector Envy | 3 |
| Scattered Parasitic Functionality | 2 |
| Component Concern Overload | 4 |

Tabela 27 – Lista de priorização de anomalias - Versão 4 do *Notepad SPL*

| **Anomalias detectadas** | **Prioridade de tratamento** |
|---|---|
| Ambigous Interface | 1 |
| Scattered Parasitic Functionality | 2 |
| Component Concern Overload | 3 |

Tabela 28 – Lista de priorização de anomalias - Versão 5 do *Notepad SPL*

| **Anomalias detectadas** | **Prioridade de tratamento** |
|---|---|
| Ambigous Interface | 1 |
| Connector Envy | 2 |
| Scattered Parasitic Functionality | 3 |
| Component Concern Overload | 4 |

Tabela 29 – Lista de priorização de anomalias - Versão 6 do *Notepad SPL*

| **Anomalias detectadas** | **Prioridade de tratamento** |
|---|---|
| Ambigous Interface | 1 |
| Scattered Parasitic Functionality | 3 |
| Component Concern Overload | 2 |

Tabela 30 – Lista de priorização de anomalias - Versão 7 do *Notepad SPL*

| **Anomalias detectadas** | **Prioridade de tratamento** |
|---|---|
| Ambigous Interface | 1 |
| Scattered Parasitic Functionality | 2 |
| Component Concern Overload | 3 |

Tabela 31 – Lista de priorização de anomalias - Versão 1 do *HealthWatcher*

| **Anomalias detectadas** | **Prioridade de tratamento** |
|---|---|
| Ambigous Interface | 1 |
| Connector Envy | 3 |
| Scattered Parasitic Functionality | 2 |
| Component Concern Overload | 4 |

Tabela 32 – Lista de priorização de anomalias - Versão 2 do *HealthWatcher*

| **Anomalias detectadas** | **Prioridade de tratamento** |
|---|---|
| Ambigous Interface | 2 |
| Connector Envy | 3 |
| Scattered Parasitic Functionality | 1 |
| Component Concern Overload | 4 |

Tabela 33 – Lista de priorização de anomalias - Versão 3 do *HealthWatcher*

| **Anomalias detectadas** | **Prioridade de tratamento** |
|---|---|
| Ambigous Interface | 1 |
| Connector Envy | 2 |
| Extraneous Adjacent Connector | 4 |
| Scattered Parasitic Functionality | 3 |
| Component Concern Overload | 5 |

Tabela 34 – Lista de priorização de anomalias - Versão 4 do *HealthWatcher*

| **Anomalias detectadas** | **Prioridade de tratamento** |
|---|---|
| Ambigous Interface | 1 |
| Connector Envy | 2 |
| Scattered Parasitic Functionality | 4 |
| Component Concern Overload | 3 |

Tabela 35 – Lista de priorização de anomalias - Versão 5 do *HealthWatcher*

| **Anomalias detectadas** | **Prioridade de tratamento** |
|---|---|
| Ambigous Interface | 2 |
| Connector Envy | 1 |
| Extraneous Adjacent Connector | 4 |
| Scattered Parasitic Functionality | 3 |
| Component Concern Overload | 5 |

Tabela 36 – Lista de priorização de anomalias - Versão 6 do *HealthWatcher*

| **Anomalias detectadas** | **Prioridade de tratamento** |
|---|---|
| Ambigous Interface | 1 |
| Connector Envy | 2 |
| Extraneous Adjacent Connector | 3 |
| Scattered Parasitic Functionality | 4 |
| Component Concern Overload | 5 |

Tabela 37 – Lista de priorização de anomalias - Versão 7 do *HealthWatcher*

| **Anomalias detectadas** | **Prioridade de tratamento** |
|---|---|
| Ambigous Interface | 1 |
| Connector Envy | 2 |
| Extraneous Adjacent Connector | 3 |
| Scattered Parasitic Functionality | 4 |
| Component Concern Overload | 5 |

Tabela 38 – Lista de priorização de anomalias - Versão 8 do *HealthWatcher*

| **Anomalias detectadas** | **Prioridade de tratamento** |
|---|---|
| Ambigous Interface | 1 |
| Connector Envy | 2 |
| Extraneous Adjacent Connector | 3 |
| Scattered Parasitic Functionality | 4 |
| Component Concern Overload | 5 |

Tabela 39 – Lista de priorização de anomalias - Versão 9 do *HealthWatcher*

| **Anomalias detectadas** | **Prioridade de tratamento** |
|---|---|
| Ambigous Interface | 1 |
| Connector Envy | 2 |
| Scattered Parasitic Functionality | 4 |
| Component Concern Overload | 3 |

Tabela 40 – Lista de priorização de anomalias - Versão 10 do *HealthWatcher*

| **Anomalias detectadas** | **Prioridade de tratamento** |
|---|---|
| Ambigous Interface | 1 |
| Connector Envy | 2 |
| Extraneous Adjacent Connector | 4 |
| Scattered Parasitic Functionality | 3 |
| Component Concern Overload | 5 |