Pontifícia Universidade Católica
do Rio de Janeiro

Manuele dos Reis Ferreira

Detecção de anomalias de código de relevância arquitetural em sistemas multilinguagem

Dissertação de Mestrado

Dissertação apresentada ao Programa de Pós–graduação em Informática da PUC–Rio como requisito parcial para obtenção do título de Mestre em Informática.

Orientador: Prof. Alessandro Fabrício Garcia

Rio de Janeiro
Abril de 2014

Pontifícia Universidade Católica
do Rio de Janeiro

Manuele dos Reis Ferreira

Detecção de anomalias de código de relevância arquitetural em sistemas multilinguagem

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós–graduação em Informática da PUC–Rio.Aprovada pela Comissão Examinadora abaixo assinada.

Prof. Alessandro Fabrício Garcia
Orientador
Departamento de Informática — PUC–Rio

Prof. Cláudio Nogueira Sant'Anna
Universidade Federal da Bahia

Prof. Carlos José Pereira de Lucena
Pontifícia Universidade Católica do Rio de Janeiro

Rio de Janeiro, 11 de Abril de 2014

Manuele dos Reis Ferreira

Bacharel em Ciência da Computação pela Universidade Federal da Bahia (2010).

Ficha Catalográfica

Ferreira, Manuele dos Reis

Detecção de anomalias de código de relevância arquitetural em sistemas multilinguagem / Manuele dos Reis Ferreira; orientador: Alessandro Fabrício Garcia. — Rio de Janeiro : PUC–Rio, Departamento de Informática, 2014.

v., 101 f: il. ; 29,7 cm

1. Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui referências bibliográficas.

1. Informática – Tese. 2. Arquitetura de Software. 3. Degradação Arquitetural. 4. Anomalia de Código. 5. Anomalia de Arquiteturalmente Relevante. 6. Sistema Multilinguagem. I. Garcia, Alessandro Fabrício . II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Agradecimentos

Para a construção de uma dissertação não são necessárias apenas palavras, mas também de um suporte de pessoas e instituições. Gostaria de agradecer à minha Família (Tone, Maria, Jones, Quele, Uli e Lulu) por todo o apoio durante essa jornada. Sem as suas palavras, amor e atenção não seria possível ultrapassar todos os obstáculos que surgiram no caminho.

Gostaria de agradecer aos meus amigos (Soraya, Andreson, Monique, Felipe, Bianca e Alexandre) por todo apoio durante os momentos de tristeza e companhia nos momentos de alegria. Queria agradecer a Ju pela ajuda, compreensão, apoio e dedicação em me auxiliar em todos os momentos.

Agradecimento a Minds por todo o apoio durante o desenvolvimento dos trabalhos e pelas oportunidades oferecidas. Um agradecimento especial ao grupo de pesquisa OPUS e a todos os amigos que fiz lá que tanto me deram apoio e me ajudaram a tornar menos complicado o caminho que trilhei. Aos meus amigos da PUC por toda a força! Além disso, ao meu professor e orientador Alessandro Garcia. O seu empenho, dedicação e profissionalismo é singular e extremamente motivante. Por fim, um agradecimento importante a PUC e a CAPES pelo apoio e investimento nesse trabalho e em tantos outros que trouxeram grandes frutos ao país.

Resumo

Ferreira, Manuele dos Reis; Garcia, Alessandro Fabrício . **Detecção de anomalias de código de relevância arquitetural em sistemas multilinguagem**. Rio de Janeiro, 2014. 101p. Dissertação de Mestrado — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Estudos recentes mostram que os sistemas são desenvolvidos por pelo menos quatro linguagens. Ao utilizar estas linguagens, boas práticas de desenvolvimento também são diferentes. Estes aspectos de heterogeneidade dificultam a concepção de soluções que apoiem desenvolvedores na construção de sistema multilinguagem com qualidade. Em particular, diversas abordagens têm surgido nos últimos anos com o objetivo de auxiliar os analistas nas tarefas de compreensão e manutenção desses sistemas. Porém, ainda existe uma carência de abordagens com foco na detecção de anomalias de código em sistemas multilinguagem. Dessa forma, o objetivo desse trabalho é oferecer suporte a identificação de sintomas de degradação arquitetural através do uso de estratégias baseadas em métricas em sistemas multilinguagem.

Palavras–chave

Arquitetura de Software; Degradação Arquitetural; Anomalia de Código; Anomalia de Arquiteturalmente Relevante; Sistema Multilinguagem;

Abstract

Ferreira, Manuele dos Reis; Garcia, Alessandro Fabrício (Advisor). **Detecting Architecturally-Relevant Code Anomalies on Multi-Language Systems**. Rio de Janeiro, 2014. 101p. MsC Dissertation — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Recent studies show that the systems are designed with at least four languages. Using these languages, best practices to development are also different. These aspects of heterogeneity make it difficult to design solutions that support developers activities on developing of multi-language system with quality. In particular, several approaches have emerged with the aim to assist analysts in comprehension and maintaining systems. However, there is still a lack of approaches focused on detection of code anomaly on multi-language systems. Thus, the aim of this work is to support the identification of symptoms of architectural degradation through the use of metrics-based strategies on multi-language systems.

Keywords

Software Architecture; Architectural Degradation; Code Anomaly; Architecturally-Relevant Code Anomaly; MultiLanguage System;

Sumário

Lista de figuras

Lista de tabelas

Na falta do que fazer, inventei a minha liberdade.

Engenheiros do Hawaii, *Surfando Karmas e DNA.*

1
Introdução

Ainda no fim da década de 90, Jones reportou que um terço dos sistemas nos EUA eram sistemas multilinguagem, isto é, sistemas desenvolvidos utilizando pelo menos duas linguagens de programação [Jones, 1998]. Já nos últimos anos, esse número tem crescido de forma significativa [Mayer and Schroeder, 2012]. Karus reportou, em um recente estudo com softwares, que desenvolvedores utilizam, na média, 4 diferentes linguagens durante o desenvolvimento de um sistema [Karus and Gall, 2011]. Estas linguagens utilizadas em um mesmo projeto, usualmente suportam paradigmas diferentes e são usadas para apoiar a resolução de problemas diferentes. Consequentemente, ao utilizar estas linguagens, boas práticas de desenvolvimento também são significativamente diferentes.

Todos estes aspectos de heterogeneidade dificultam a concepção de soluções que apoiem os desenvolvedores na construção de sistema multilinguagem com qualidade. Em particular, diversas abordagens têm surgido nos últimos anos com o objetivo de auxiliar os analistas nas tarefas de compreensão e manutenção desses sistemas [Kontogiannis et al., 2006] [Mayer and Schroeder, 2012] [Alves et al., 2011]. No entanto, quando essas tarefas não são realizadas de maneira adequada, surgem anomalias de código. Anomalia de código (ou *code smell*) é qualquer sintoma identificado na estrutura do código fonte que pode dificultar a compreensão e manutenção de sistemas [Fowler, 1999]. Por exemplo, um método é diagnosticado como portador da anomalia método longo quando possui uma grande quantidade de linhas de código. Classes também podem ser diagnosticadas como portadoras, tais como *divergent change*. Esta anomalia é observada quando uma classe é modificada de diversas maneiras por diferentes razões.

Anomalias de código são particularmente nocivas a um projeto de software quando elas representam problemas arquiteturais na implementação. Uma anomalia de código é relevante arquiteturalmente quando está associada a um problema arquitetural [Macia et al., 2012a]. Um problema arquitetural é uma decisão arquitetural que tem impacto negativo na qualidade do sistema [Garcia et al., 2009]. Por exemplo, o problema arquitetural *Ambiguous Interface* indica

que uma interface oferece serviços através de um ponto de entrada genérico e único do componente [Garcia et al., 2009]. Este ponto trata e redistribui internamente todas as requisições feitas ao componente. Esse problema cria a necessidade do conhecimento interno do funcionamento da interface pelos seus clientes. Devido a falta usual de documentação da arquitetura, tais problemas devem ser revelados através de anomalias de código-fonte [Binkley, 2007] [Harris et al., 1996].

Infelizmente, poucos esforços existem na realização de estudos e na concepção de técnicas e ferramentas que apoiem a detecção de anomalias de código que sejam relevantes à arquitetura de um sistema. Em sistemas multilinguagem esses desafios estão relacionados à necessidade de uma análise homogênea de elementos do programa implementados em diferentes linguagens. Sem esta análise, torna-se difícil ou impeditivo diagnosticar como anomalias nas diferentes linguagens estão relacionadas com um problema arquitetural.

Nestes sistemas, existe também a dificuldade na avaliação de cada componente em separado em razão da variedade sintática e semântica das diferentes linguagens. Além disso, existe a variedade na forma com que esses componentes são internamente estruturados. Ou seja, tipos de decisões de *design* que foram tomadas nas implementações de cada componente. Tendo em vista essas dificuldades, existem poucos trabalhos que envolvem o desenvolvimento de técnicas e ferramentas que auxiliem analistas na detecção de anomalias de código de relevância arquitetural em sistemas multilinguagem, como será visto na próxima Seção.

1.1 Motivação

O *design* de um sistema deve ser aderente à sua organização fundamental, denominada arquitetura de software. A arquitetura de um sistema de software é composta por seus componentes, as relações entre eles, bem como o seu ambiente [Shaw and Garlan, 1996]. O ambiente determina a configuração do desenvolvimento sobre o sistema, como por exemplo, as linguagens que são utilizadas [Shaw and Garlan, 1996]. As linguagens utilizadas exercem um impacto chave na arquitetura de software de um sistema, uma vez que são estas que determinam como os componentes, suas interfaces e seus relacionamentos serão estruturados. Elas também determinam como seus componentes serão evoluídos.

Arquiteturas de sistemas multilinguagem possuem componentes desenvolvidos com diferentes linguagens de programação. A arquitetura de software de um sistema multilinguagem também precisa ser aprimorada com o tempo

em função de novos requisitos do sistema. Esse aprimoramento depende diretamente de tarefas relacionadas à compreensão e manutenção de código. Caso contrário, a inclusão de novos requisitos pode se tornar difícil ou inviável [Hochstein and Lindvall, 2005]. Em especial, este aprimoramento depende da detecção e remoção de anomalias de código relevantes à arquitetura do sistema.

Porém, ainda pouco se sabe sobre a dificuldade de se usar técnicas de detecção mesmo para sistemas implementados em uma única linguagem. Adicionalmente, pouco se tem investigado sobre as dificuldades de usar técnicas de detecção de anomalias de código em projetos reais de software. Alguns estudos empíricos recentes [Kullbach et al., 1998] [Linos et al., 2003] [Kontogiannis et al., 2006] [Mayer and Schroeder, 2012] [Alves et al., 2011] investigam a eficácia de tais técnicas em detecção de anomalias de código com relevância arquitetural. Porém, eles foram realizados de forma retrospectiva e os desenvolvedores destes sistemas não são envolvidos diretamente na avaliação da eficácia. Além disso, não existe o conhecimento a respeito do esforço do uso dessas técnicas. Esta falta de conhecimento é ainda maior se considerados sistemas multilinguagem, já que usualmente não é o foco até mesmo de estudos recentes. A dificuldade na detecção de anomalias de código de relevância arquitetural é influenciada por diversos desafios. Em sistemas multilinguagem, dois grandes desafios podem ser destacados:

1. Considerar isoladamente componentes implementados em cada linguagem não permite a análise homogênea do sistema. A análise isolada de cada componente oculta os relacionamentos entre componentes da arquitetura desenvolvidos em diferentes linguagens. Por exemplo, não é possível obter informações suficientes para detectar múltiplas anomalias que podem afetar uma interface escrita com uma linguagem A que seja requisitada por componentes escritos em outras linguagens, B e C.

2. A análise dos componentes implementados em diferentes linguagens não é trivial em razão da: (i) variedade sintática e semântica das diferentes linguagens; (ii) variedade de *design* nos componentes desenvolvidos em diferentes linguagens; e (iii) cada linguagem usualmente segue paradigmas distintos de programação, tais como orientação a objetos, funcional, dentre outros.

A título de exemplo, considere uma interface I que disponibiliza serviços úteis à geração de relatórios em um sistema e foi escrita na linguagem A. Esta interface I é o ponto de entrada de um componente e é acessada por outros componentes escritos em uma linguagem B. Cada um dos componentes implementa um tipo diferente de relatório. A interface I possui uma estrutura

que necessita, para cada novo relatório, a criação de um novo componente, mesmo que a diferença entre as suas implementações seja apenas o nome. Dessa forma, para realizar a identificação de que a interface I tem anomalias seria necessário saber que possui uma grande dependência em relação a componentes na linguagem B como definido em 2 (i). Além disso, a avaliação de que existe uma anomalia de código de relevância arquitetural na interface I só será possível através da avaliação dos componentes na linguagem B como definido em 2 (ii). Como A e B são usadas para fins distintos no sistema, utilizam paradigmas diferentes de programação visando obter o melhor desempenho, elucidando o desafio 2 (iii).

Dessa forma, a avaliação do que é uma anomalia de código de relevância arquitetural é dificultada em razão dos desafios apresentados. Estes desafios são inerentes a sistemas implementados em mais de uma linguagem. Uma vez que as abordagens existentes são destinadas a projetos implementados com uma única linguagem [Macia et al., 2012b] [Moha et al., 2010] [Marinescu, 2004] [Schumacher et al., 2010] [Wedyan et al., 2009], desenvolvedores tendem a prevalecer inconscientes de várias anomalias de código relevantes à arquitetura de software. Desta forma, ações preventivas não são aplicadas ou somente aplicadas tardiamente.

Porém, quando a tarefa de detecção não é acompanhada de ações preventivas, ocorre um acúmulo de anomalias de código nocivas à arquitetura de software. Esse fenômeno é denominado degradação arquitetural [Hochstein and Lindvall, 2005]. Problemas arquiteturais são sintomas de degradação arquitetural e são consequências de decisões arquiteturais que exercem um impacto negativo em atributos de qualidade do sistema, tais como dificuldade de manutenção e evolução [Hochstein and Lindvall, 2005]. Na implementação dos sistemas, estes problemas arquiteturais podem ser observados através de anomalias de código [Fowler, 1999] [Hochstein and Lindvall, 2005] [Macia et al., 2012b] [Macia et al., 2012a]. De fato, estudos recentes na literatura revelaram uma relação direta e frequente entre tais anomalias de código na implementação e problemas arquiteturais [Hochstein and Lindvall, 2005] [Macia et al., 2012b] [Macia et al., 2012a].

Portanto, desenvolvedores devem ser capazes de detectar anomalias de código que sejam relevantes à arquitetura de software em sistemas. Estudos afirmam que o uso de estratégias baseadas em métricas pode ser um caminho eficaz para detectar tais anomalias [Macia et al., 2012a] [Marinescu, 2004]. Essas estratégias permitem raciocinar sobre a estrutura do código de forma mais abstrata e independente da linguagem subjacente. Portanto, esta é uma característica particularmente interessante para sistemas multilinguagem.

Estas estratégias consistem de heurísticas que capturam desvios quantificáveis de princípios de bom *design* da arquitetura [Marinescu, 2004]. Desta forma, as heurísticas podem ser adequadas às características peculiares do *design* de cada componente e das linguagens subjacentes. No entanto, essas estratégias de detecção somente podem ser usadas se a eficácia destas é alta e o esforço reduzido. Caso contrário, não são pragmáticas e não podem ser aplicadas no contexto de sistemas reais, principalmente aqueles de natureza multilinguagem.

1.2 Caracterização do Problema

A identificação de anomalias de código de relevância arquitetural em sistemas, em especial, multilinguagem é atualmente ineficaz. Mesmo que o uso de estratégias baseadas em métricas seja uma abordagem promissora [Macia, 2013] e espera-se que o uso dessas estratégias seja eficaz para detectar anomalias código arquiteturalmente relevante, pouco se sabe sobre a sua eficácia e esforço em sistemas com uma única linguagem ou multilinguagem. Os estudos empíricos, citados anteriormente [Macia et al., 2012b] [Moha et al., 2010] [Marinescu, 2004] [Schumacher et al., 2010] [Wedyan et al., 2009] não são executados como um estudo de caso de campo, ou seja, no ambiente real de desenvolvimento.

Como consequência, não existe conhecimento em relação ao esforço necessário no processo de identificação. E, além disso, os desenvolvedores atuais também não são envolvidos no processo de comparação da eficácia e esforço das estratégias por eles usadas versus as estratégias baseadas em métricas. Um estudo mais amplo em relação à eficácia e esforço pode, inicialmente, avaliar quais os aspectos impactam nos números de falsos positivos e negativos no diagnóstico das anomalias de código de relevância arquitetural. Posteriormente, vai permitir a realização de uma avaliação mais detalhada em relação ao esforço requerido nesse processo.

Especificamente em relação a sistemas multilinguagem, existem diversos aspectos que devem ser avaliados como: (2.1) O que é uma anomalia de código para cada linguagem. Pelo fato de que cada linguagem possui as suas próprias particularidades, caracterizar um trecho de código como uma anomalia pode ser diferente para cada linguagem; (2.2) Qual a relação entre linguagens de programação e tipos recorrentes de problemas arquiteturais. Assim como existem particularidades de cada linguagem em relação à definição do que é uma anomalia, caracterizar problemas arquiteturais pode ser diferente também; (2.3) Como diagnosticar problemas na comunicação entre as interdependências dos componentes. A estrutura do código de cada linguagem, ou seja, componente,

é diferente.

Tendo em vista estas dificuldades atuais, pouco se sabe sobre as nuances na detecção de anomalias de código relevantes a arquitetura em sistemas multilinguagem. Estudos empíricos na literatura não abordam explicitamente esta questão, em grande parte pela falta de suporte ferramental adequado ao processo de análise. Consequentemente, é limitado o conhecimento sobre as características peculiares de anomalias de código relevantes à arquitetura quando componentes foram implementados em diferentes linguagens. Existem evidências iniciais de que problemas arquiteturais são frequentemente associados com múltiplas anomalias no código de um sistema [Macia et al., 2012a]. Estudos, em especial, conduzido por Macia et al. mostram que a associação de múltiplas anomalias é uma característica eficaz na identificação de anomalias de código de relevância arquitetural [Macia et al., 2012a]. Sem um suporte a análise de anomalias de código inter-relacionadas, a detecção de problemas arquiteturais pode ser ainda mais difícil. Dessa forma, seria importante viabilizar e estudar como e quais anomalias inter-relacionadas em sistemas multilinguagem estão associadas com a degradação arquitetural nestes sistemas.

1.3 Limitações dos Trabalhos Relacionados

As abordagens existentes são categorizadas em dois grupos. Primeiramente, técnicas para **reengenharia de sistemas** que focalizam na geração de estruturas como grafos, que permitem a identificação das relações entre os elementos e a construção de consultas para análise desses relacionamentos [Kullbach et al., 1998] [Linos et al., 2003] [Kontogiannis et al., 2006]. Essa abordagem permite analisar as dependências entre componentes escritos em diferentes linguagens, permitindo uma análise homogênea dos sistemas e componentes que o constituem. Em contrapartida, não permitem a avaliação das propriedades estruturais do código, fundamentais para a detecção de anomalias de código. Exemplos dessas propriedades estruturais é o número de linhas de código de uma classe e a complexidade ciclomática.

Por outro lado, o segundo grupo de abordagens provê suporte a **avaliação da estrutura do código** por meio de métricas que contabilizam propriedades estruturais do código [Nicolay et al., 2013] [Terceiro et al., 2010] em uma linguagem específica. Por exemplo, o trabalho de [Terceiro et al., 2010] permite contabilizar métricas para diversas linguagens tais como, Java e C. No entanto, esse grupo de abordagens não oferece um suporte à avaliação de dependências entre componentes escritos em diferentes linguagens. Outros trabalhos desenvolveram técnicas para contabilização única de métricas para

algumas linguagens. No entanto, possuem a limitação de avaliar linguagens específicas que geram linguagens intermediárias, o que permite contornar a falta de um suporte homogêneo em sistemas desenvolvidos por essas linguagens específicas [Linos et al., 2007] [Nguyen et al., 2012].

Dessa forma, não existem trabalhos no estudo da arte baseados em abordagens híbridas que combinam características dos dois grupos de abordagens em sistemas multilinguagem. Por exemplo, a visualização de software utiliza das boas técnicas de apresentação de informações de maneira que o interlocutor compreenda-as melhor. O uso dessas técnicas facilita a compreensão das pessoas e utiliza gráficos e imagens em geral para ilustrar sistemas [Ghanam and Carpendale, 2008]. As abordagens atuais de visualização de software permitem fazer a avaliação de sistemas e utilizam como base informações de métricas e dependências entre componentes [de F Carneiro et al., 2009] [Ghanam and Carpendale, 2008]. No entanto, essas abordagens são limitadas a avaliar sistemas escritos em apenas uma linguagem.

1.4 Objetivos e Questões de Pesquisa

O objetivo principal do trabalho é aperfeiçoar o suporte à identificação de problemas arquiteturais através do uso de estratégias baseadas em métricas em sistemas multilinguagem. A concretização desse objetivo só é possível através de: (i) Avaliação da eficácia e esforço associado no uso das estratégias atuais no suporte a identificação do ponto de vista de eficácia e esforço; (ii) Suporte a análise de anomalias de código inter-relacionadas de forma que o desenvolvedor possa detectar problemas arquiteturais de forma mais eficaz em software multilinguagem.

Neste contexto, o objetivo será alcançado através das respostas das seguintes perguntas:

- RQ1 Qual é a eficácia e o esforço associado ao aplicar estratégias atuais na identificação de anomalias de código de relevância arquitetural?
- RQ2 Como poderão ser definidas estratégias para identificação de problemas arquiteturais em componentes dependentes em sistemas multilinguagem?
- RQ3 Quais as características das anomalias de código podem indicar problemas arquiteturais em um sistema multilinguagem?

A resposta à pergunta RQ1 vai permitir identificar oportunidades de melhorias nas técnicas atuais visando aumentar a eficácia e diminuir o esforço associado. Além disso, os resultados dessa avaliação vão permitir adquirir

insumos para a definição de uma abordagem para identificação de problemas arquiteturais em sistemas multilinguagem como definido na RQ2. Por exemplo, ao responder RQ1 estaremos revelando limitações específicas de uso de técnicas convencionais para software multilinguagem. A RQ2 vai permitir definir uma abordagem que identifique problemas arquiteturais através das estratégias em sistemas multilinguagem. A resposta para a pergunta RQ3 será importante para a análise da eficácia da abordagem proposta em resposta a RQ2.

1.5 Estrutura do Documento

A dissertação está estruturada em seis capítulos, sendo o primeiro a presente Introdução. No Capítulo 2, são apresentados os principais conceitos que fundamentam a pesquisa, tais como sistemas multilinguagem, degradação arquitetural, problemas arquiteturais e anomalias de código de relevância arquitetural. Além disso, são discutidos os trabalhos relacionados existentes, tanto na academia quanto no mercado, que permitem a identificação de anomalias de código de relevância arquitetural. No Capítulo 3, é respondida a RQ1, através um estudo de caso realizado para avaliar a eficácia e o esforço associado ao uso de estratégias convencionais. Estas estratégias foram usadas na identificação de anomalias de código em um sistema multilinguagem do mercado.

No Capítulo 4, é respondida a RQ2 através da elaboração de procedimentos para o desenvolvimento de uma abordagem que permite o suporte a análise de anomalias inter-relacionadas. No Capítulo 5, é respondida a RQ3, através da construção e execução de um estudo realizado em três aplicações comerciais. O foco principal deste estudo foi avaliar a eficácia da abordagem proposta na identificação de anomalias de código de relevância arquitetural em sistemas multilinguagem. Por fim, o Capítulo 6 resume o trabalho, discute conclusões, contribuições e limitações desta dissertação e apresenta perspectivas futuras de pesquisa.

2
Principais Conceitos e Trabalhos Relacionados

Neste Capítulo, serão descritos conceitos relevantes para o contexto desta dissertação. Inicialmente, serão apresentados os conceitos associados com sistemas multilinguagem, arquitetura de software e degradação arquitetural. Posteriormente, serão descritos o que são os problemas arquiteturais, a sua relação com anomalias de código, bem como a formulação de estratégias para a sua detecção. Por fim, são apresentadas as abordagens atuais de identificação de anomalias de código de relevância arquitetural, com uma Subseção específica para o SCOOP, uma ferramenta de detecção de anomalias de código utilizada e estendida neste trabalho.

2.1
Sistemas Multilinguagem

Sistemas monolinguagem são sistemas desenvolvidos utilizando somente uma linguagem. Todos os componentes de tais sistemas são escritos em uma linguagem específica. Entretanto, existem muitos sistemas que são desenvolvidos utilizando diversas linguagens e, por esse motivo, são denominados sistemas multilinguagem. O uso de cada uma delas é justificado pelo ganho de eficiência no sistema em razão da sua melhor adequação a um domínio específico [Kullbach et al., 1998] [Jerraya and Ernst, 1999]. Por exemplo, algumas linguagens são melhores adaptadas para a especificação de estruturas de páginas web, outras para o gerenciamento de dados e outras para escrita de algoritmos que implementam regras de negócio. Assim, a implementação de cada um dos componentes é feita através do uso de diferentes linguagens. Consequentemente, tais sistemas são usualmente desenvolvidos por diferentes grupos de pessoas responsáveis por cada componente [Kullbach et al., 1998] [Jerraya and Ernst, 1999]. Isso também torna mais adequado o desenvolvimento de cada componente de acordo com os múltiplos domínios envolvendo uma única aplicação e a cultura organizacional de cada grupo de pessoas [Kullbach et al., 1998] [Jerraya and Ernst, 1999].

Um exemplo de arquitetura de um sistema multilinguagem pode ser visto na Figura 2.1. Essa arquitetura representa um típico sistema de informação

web. Esse sistema é estruturado em três camadas que ficam distribuídas em um modelo cliente-servidor, cada camada é representada com uma linha tracejada. A parte cliente é composta pela camada de apresentação na qual são utilizadas as linguagens HTML e JavaScript, cada linguagem é representada com uma linha contínua. A parte servidor é composta pelas camadas controle e modelo. Na camada de controle são usadas as linguagens JSON - JavaScript Object Notation, JSP - JavaServer Pages e Java e na camada modelo o SQL - Structured Query Language.

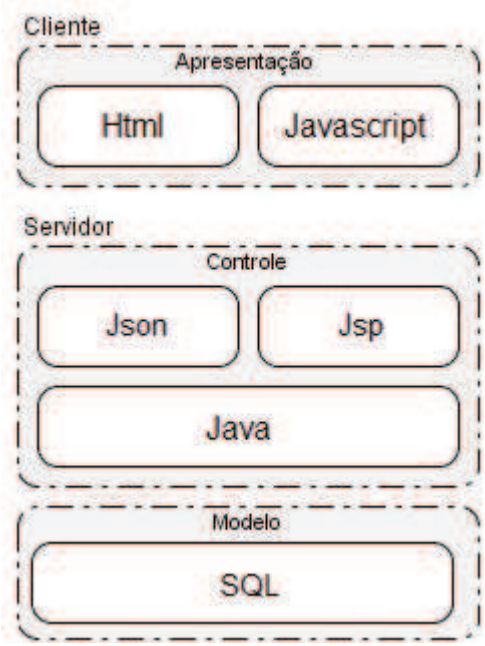

Figura 2.1: Exemplo da arquitetura de um sistema multilinguagem (Camada é representada com linha tracejada e linguagem com linha contínua)

Não somente sistemas web são desenvolvidos usando diversas linguagens. Jones documentou que um terço dos sistemas nos EUA está escrito em pelo menos duas linguagens [Jones, 1998]. Karus reportou em um recente estudo que desenvolvedores utilizam, na média, 4 diferentes linguagens [Karus and Gall, 2011]. Um sistema composto por diversos componentes, desenvolvidos por linguagens diferentes, precisa que estes componentes se interconectem. Com o tempo, são realizadas atividades de compreensão e manutenção desses sistemas. Dessa forma, a análise desses componentes é feita de maneira heterogênea, ou seja, exige que foco da análise de cada componente seja feito de maneira diferente.

As dificuldades associadas à análise heterogênea dos componentes foram divididas em dois pontos:

– Variedades sintática e semântica nas regras de uso das diferentes linguagens.

– Variedade de *designs* nos componentes desenvolvidos em diferentes linguagens. Por exemplo, um componente pode ter sido modelado seguindo os princípios de orientação a objetos (OO), enquanto o outro segue o paradigma estrutural [Kontogiannis et al., 2006].

Essa variedade evidencia a necessidade de diferentes formas de avaliação de componentes no sistema. Essa variedade se aplica na análise das soluções de implementação, uma vez que cada linguagem possui características estruturais diferentes. Ou seja, enquanto no *design* OO a organização em classes é um importante requisito na modularidade, na programação estruturada a decomposição se resume a estrutura de função. Consequentemente, a complexidade na compreensão e manutenção de sistemas multilinguagem aumenta, tornando esses processos custosos e suscetíveis a erros. Em especial, quando a tarefa de identificação de sintomas de degradação não é simplificada, o sistema pode ter sua arquitetura descaracterizada por completo em longo prazo.

2.2 Degradação Arquitetural

De acordo com a especificação IEEE 1471 [Maier et al., 2004], um sistema é uma coleção de componentes organizados para realizar um conjunto de funcionalidades. O sistema possui uma organização fundamental denominada arquitetura de software que é composta por seus componentes, as relações entre eles e o ambiente e os princípios que guiam o seu *design* e evolução. O ambiente determina a configuração e as circunstâncias de desenvolvimento sobre o sistema, como por exemplo, as linguagens que são utilizadas e o seu impacto estrutural [Maier et al., 2004]. Essas linguagens exercem um impacto importante na arquitetura, em razão das suas particularidades que determinam como os componentes e suas interfaces serão estruturados.

Arquiteturas de sistemas multilinguagem possuem diversos componentes desenvolvidos por diferentes linguagens. Ou seja, a arquitetura de software emerge a partir de decisões de *design* amplamente distintas em cada um de seus componentes e, potencialmente, por diferentes desenvolvedores. Essas decisões são tomadas durante todo o ciclo de desenvolvimento do software [Booch, 2007]. Sistemas possuem áreas de interesse que influenciam de forma significativa na arquitetura [Lippert and Roock, 2006] chamados de interesses arquiteturais.

A arquitetura de software de um sistema é aprimorada com o tempo em função de novos requisitos do sistema. Esse aprimoramento depende diretamente de tarefas relacionadas à compreensão e manutenção [Hochstein and Lindvall, 2005]. Em especial, este aprimoramento depende de estratégias eficazes para detecção de sintomas de degradação arquitetural. No entanto, ela

vai sendo refinada e aprimorada ao longo do tempo. A arquitetura prescritiva representa a arquitetura pretendida ou planejada do sistema que deve ser acatada durante a implementação [Taylor et al., 2009]. Por outro lado, a arquitetura que de fato foi construída é denominada arquitetura descritiva [Taylor et al., 2009].

Mudanças no código associadas a novos requisitos devem ser acompanhadas de ações preventivas que evitam o aumento da complexidade dos componentes implementados. Se essas ações não forem tomadas, ocorre um acúmulo crescente de violações de *design*. Esse evento é denominado degradação arquitetural [Hochstein and Lindvall, 2005].

A manifestação da degradação arquitetural pode ocorrer através do desvio arquitetural (*drift*) que é a introdução de decisões na arquitetura descritiva que não estão incluídas ou não são implicações das descrições contidas na arquitetura prescritiva [Hochstein and Lindvall, 2005]. No entanto, essas decisões podem violar princípios de modularidade arquitetural como o princípio da única responsabilidade e o da separação de interesses arquiteturais nos componentes [Martin, 2003]. Cada desvio arquitetural está diretamente relacionado a um problema arquitetural [Garcia et al., 2009] e o presente trabalho tem o foco em problemas arquiteturais.

2.3 Problemas Arquiteturais

Problema arquitetural é uma decisão arquitetural que tem impacto negativo na qualidade do sistema. Garcia et al. catalogou e documentou quatro categorias de problemas arquiteturais [Garcia et al., 2009]. Várias destas podem ser candidatas a ocorrerem frequentemente em sistemas multilinguagem. Um exemplo de categoria de problema arquitetural é a *Scattered Functionality*. Alguns componentes de um sistema com sintomas de *Scattered Functionality* possuem a característica de implementar o mesmo interesse arquitetural e alguns deles serem responsáveis também por outros interesses. O impacto dessa categoria de anomalia está relacionado a aspectos de manutenção e compreensão do sistema, tendo em vista violação do princípio da separação de interesses. Visto que esta é uma anomalia que afeta vários componentes de um sistema, a detecção da mesma requer a consideração de elementos desenvolvidos em diferentes linguagens.

Outro exemplo é a *Ambiguous Interface* que é uma interface que oferece serviços através de um genérico e único ponto de entrada do componente. Esse ponto de entrada faz o tratamento de todas as requisições e redistribui internamente para outras operações. Um exemplo de *Ambiguous Interface* é

dado na Figura 2.2. Um componente **A** constituído de arquivos escritos na linguagem JavaScript (representado pela extensão js) oferece uma interface para a funcionalidade TSV. TSV - Tab Separeted Values - é uma formatação de informações gerenciais que permite a geração de diferentes tipos de relatórios. A interface possui a operação selecionaTipoTsv, que funciona como genérico e único ponto de entrada com o papel de redistribuir a requisição para um arquivo JSP adequado ao tipo recebido. É importante notar que esta é uma anomalia que pode ocorrer em sistemas multilinguagem. Por exemplo, na Figura 2.2 existem componentes escritos em duas linguagens: JavaScript, tal como o componente A e JSP, tais como os arquivos bTSV e aTSV.

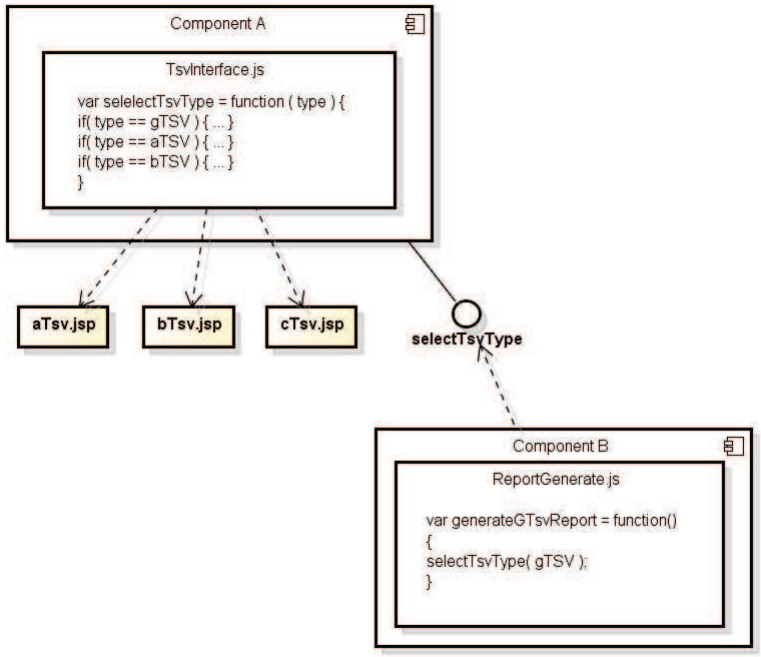

Figura 2.2: Exemplo de Ambiguous Interface em um sistema multilinguagem

As características da *Ambiguous Interface* impactam de forma negativa na qualidade do sistema, uma vez que os clientes desses serviços necessitam conhecer o funcionamento interno do componente para saber como requisitá-los. Esse problema arquitetural é a manifestação de um sintoma de degradação arquitetural ocorrida no sistema. Esse exemplo foi identificado no sistema multilinguagem avaliado no estudo empírico descrito no Capítulo 3.

Pesquisadores documentaram o efeito de anomalias de código na modularidade do sistema e a sua relação com problemas arquiteturais [Macia et al., 2012b] [Fowler, 1999]. Em especial, Macia et al. afirmou que anomalias de código de forma individual, ou em conjunto, frequentemente são indicadores chaves de degradação arquitetural no sistema [Macia et al., 2012b]. Essa relação será descrita na próxima seção.

2.4 Anomalias de Código de Relevância Arquitetural

Anomalia de código (ou *code smell*) é um problema estrutural na implementação que pode indicar problemas arquiteturais. Por exemplo, quando um método possui uma grande quantidade de linhas de código é diagnosticado como método longo. Quando a partir de uma mudança são necessárias mudanças em várias outras classes, é diagnosticado como *Shotgun Surgery* [Fowler, 1999]. Problemas arquiteturais são decisões de *design* que têm um impacto negativo em atributos de qualidade do sistema, tais como manutenibilidade e compreensibilidade [Fowler, 1999]. Quando uma anomalia de código está relacionada a um problema arquitetural, temos que essa anomalia de código é arquiteturalmente relevante.

A Listagem 2.1 representa, com maiores detalhes, o método selectTsvType, em código JavaScript, que foi referenciado na Figura 2.2 e foi identificado no estudo empírico inicial que será reportado no Capítulo 3.

```
var selectTsvType = function ( type ) {
    ...
    var param = {}
        if( type == aTSV ) {
        param["numPag"] = "1";
                $( location ).attr( 'href', "TSV/aTSV.jsp");
        } else if( type == bTSV ) {
        param["numPag"] = "2";
                $( location ).attr( 'href', "TSV/bTSV.jsp");
        } else if( type == cTSV ) {
        param["numPag"] = "1";
                $( location ).attr( 'href', "TSV/cTSV.jsp"); }
    ...
}
```

Listagem 2.1: Detalhamento do método selectTsvType em código JavaScript

O método na Listagem 2.1 acima dá exemplos de anomalias de código. O método selectTsvType possui uma grande quantidade de linhas de código e diversos interesses associados. Ela está associada a diferentes tipos de relatórios no formato TSV que estão associados a propósitos diferentes. Tendo em vista estas características, o método foi diagnosticado como um método longo [Fowler, 1999].

Além disso, o método também foi diagnosticado como *Shotgun Surgery*, pois quando, por exemplo, é necessário realizar mudanças relativas à como são definidos os valores dos parâmetros das chamadas de cada arquivo de

relatório (por exemplo, $(location).attr('href', "TSV/aTSV.jsp")) são necessárias mudanças em cada arquivo que implementa as funcionalidades de cada tipo. Por exemplo, a variável **numPag** que está sendo definida no trecho **param["numPag"] = "1"** define o número de páginas em que o relatório será gerado. Nesse caso, o relatório do tipo **gTSV** será gerado com uma página.

O método selectTsvType é escrito em JavaScript. Nessa linguagem, a definição do tipo de uma variável é dinâmica. Ou seja, a mesma variável pode ter atribuídos valores de tipos diferentes durante o seu uso. Nesse caso, a variável é definida inicialmente como String, uma vez que os valores atribuídos são do tipo String. Vamos supor que, como se trata de um número, deseja-se que o tipo da variável seja alterado para inteiro. Dessa forma, todos os arquivos que implementam a geração do relatório vão precisar processar o valor da variável de maneira diferente, uma vez que todos a reconhecem como String.

Algumas dessas anomalias de código podem afetar negativamente a arquitetura, ou seja, representar um problema arquitetural (Seção 2.3) no código fonte. Desta forma, dizemos que uma anomalia de código é arquiteturalmente relevante se representa um sintoma na implementação de um problema arquitetural [Macia et al., 2012a]. A ocorrência de cada uma das anomalias de código, método longo e *Shotgun Surgery*, representa evidências de um problema arquitetural que é a *Ambiguous Interface* (Seção 2.3). Pelo fato do método selectTsvType ser o único ponto de entrada do componente e ser responsável pela redistribuição das requisições, ele lida com diversos interesses. O sintoma método longo indica que este método lida com interesses diversos como **aTSV** e **bTSV**. Cada um dos interesses representam dois tipos diferentes de relatórios que poderiam ser implementados em diferentes funções. Um dos benefícios na implementação de cada interesse em diferentes funções seria a de coesão, uma que cada mudança seria situada em um método específico. Já o *Shotgun Surgery* captura os efeitos em cascata de possíveis modificações no ponto de entrada, pois o número de funcionalidades dependentes é grande.

No entanto, nesse exemplo, somente a existência da anomalia de código método longo poderia não ser suficiente para diagnosticar que esse é um problema arquitetural. Isso acontece, pois muitas vezes é inerente a natureza do método ser longo, como métodos que implementam um analisador léxico de um compilador. Macia et al. [Macia, 2013] observaram então que certos padrões de anomalias de código tendem a ser melhores indicadores de problemas arquiteturais. Por exemplo, a coocorrência de certas anomalias de código tende a ser forte indicador de problemas arquiteturais. Com isso, no exemplo da Figura 2.2, a coexistência das anomalias de código método longo e *Shotgun*

Surgery na interface é um melhor indicador do que a avaliação individual delas.

Macia et al. catalogaram as padrões de anomalias de código inter-relacionadas mais recorrentes e categorizou-as [Macia et al., 2012a]. Um exemplo de anomalias inter-relacionadas é o *Multiple Anomaly*. De acordo com essa anomalia inter-relacionada, certos elementos podem estar infectados por mais de uma anomalia, ou seja, ocorra uma coocorrência de anomalias de código. Essa anomalia inter-relacionada pode indicar que a distribuição de responsabilidades entre os elementos do componente pode estar sendo feita de maneira inadequada. Além disso, pode indicar um aumento na complexidade esperada para esses elementos. O exemplo da Figura 2.2 se enquadra nessa categoria. Porém, é importante mencionar que o estudo em [Macia et al., 2012b] foi conduzido apenas em sistemas monolinguagem.

2.5 Arquitetura de Software Multilinguagem: Principais Conceitos

Alguns dos principais conceitos apresentados nesse trabalho são representados na literatura através de metamodelos. Por exemplo, Macia et al. apresentaram um metamodelo para análise de anomalias de código na implementação de sistemas [Macia, 2013]. No entanto, esse metamodelo possui a limitação de não representar os conceitos de componentes que são desenvolvidos em diferentes linguagens e as anomalias inter-relacionadas. Dessa forma, foi realizada uma atualização do metamodelo permitindo a inserção desses novos conceitos. Como pode ser visto na Figura 2.3.

Através da Figura 2.3 é possível instanciar sistemas de software mono e multilinguagem. Ou seja, a figura mostra as relações entre elementos que um sistema de software pode possuir, e as relações com problema arquitetural e anomalia de código. Um sistema é composto por 1 ou mais componentes arquiteturais que pode ser desenvolvido por 1 ou mais linguagens. Além disso, ele pode ter problemas arquiteturais. Um componente arquitetural está associado a elementos de código. Elementos de código podem ter dependências entre si e podem ter anomalias de código. Uma anomalia de código pode ser arquiteturalmente relevante ou arquiteturalmente irrelevante, se estiver associada a pelo menos um problema arquitetural.

Além disso, o metamodelo define quais interesses arquiteturais podem ser mapeados em elementos de código que podem ser operações, atributos, declarações e módulos. Nesse caso, em sistemas Java, temos como exemplo, respectivamente, métodos, atributos, classes e pacotes. E, por fim, uma anomalia inter-relacionada é composta de pelo menos uma anomalia de código de relevância arquitetural.

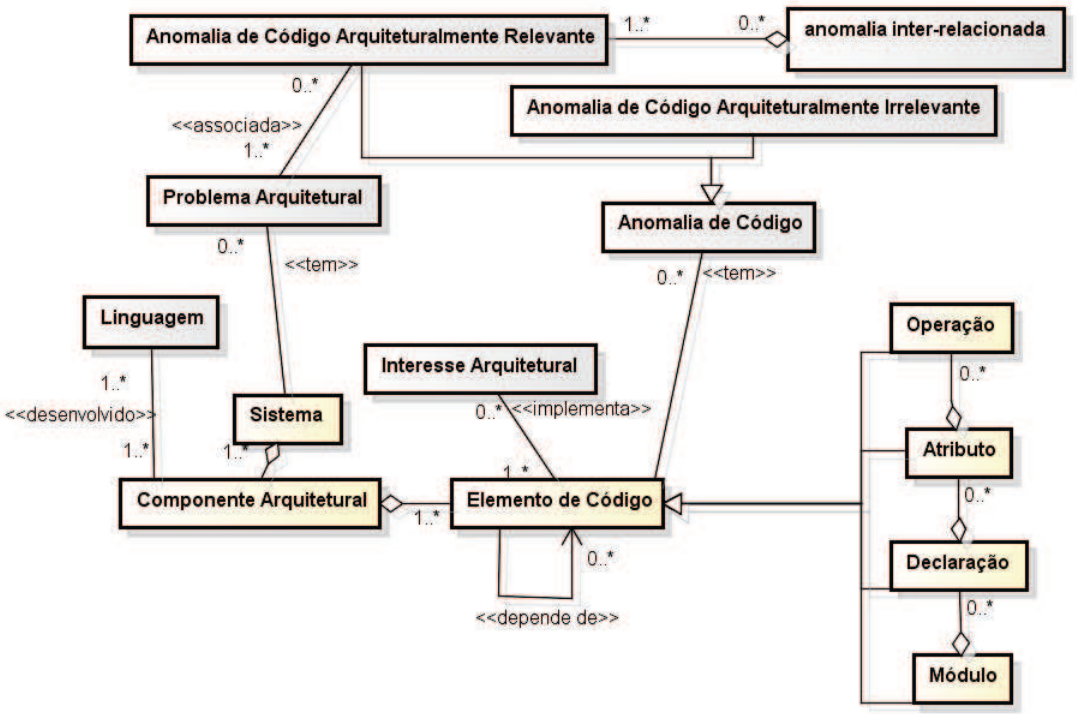

Figura 2.3: Metamodelo de sistema. Baseado na fonte: [Macia, 2013].

Quando um sistema é desenvolvido usando mais de uma linguagem, é possível que componentes arquiteturais sejam escritos em diferentes linguagens. Ou seja, elementos de código escritos em uma linguagem podem depender de elementos escritos em outra linguagem. A esses elementos de código nós chamamos de elementos híbridos e os componentes que os contém de componentes híbridos. Além disso, suponha que um elemento escrito em uma linguagem A seja afetado por uma anomalia de código inter-relacionada. Suponha também que essa anomalia esteja associada a um problema arquitetural que envolve outro elemento de código escrito em uma linguagem B. Nesse caso, chamamos essa anomalia de anomalia inter-relacionada híbrida.

Dessa forma, um sistema pode ser afetado por diversas anomalias de código de relevância arquitetural. Se ações preventivas não forem tomadas, a contínua introdução de anomalias de código de relevância arquitetural pode acarretar em aumento da sua complexidade e requerer consideráveis reestruturações. Entretanto, a identificação de anomalias de código de relevância arquitetural ainda não é efetiva, devido ao: (1) Alto número de falsos negativos gerados em razão da falta de suporte a um ambiente heterogêneo como identificado no estudo empírico, descrito no Capítulo 3 e (2) Alto esforço na identificação,

uma vez que o desenvolvedor precisa saber e avaliar diversas informações e características ao mesmo tempo como: (2.1) O que é uma anomalia de código para cada linguagem; (2.2) Qual a sua relação com violações de *design* e (2.3) Como diagnosticar problemas na comunicação entre as interdependências dos componentes.

2.6 Abordagens para Detecção de Anomalias de Relevância Arquitetural

Como as abordagens atuais de identificação de anomalias de código de relevância arquitetural são falhas, o esforço no filtro dos candidatos a problemas arquiteturais é grande [Kullbach et al., 1998] [Linos et al., 2003] [Kontogiannis et al., 2006] [Mayer and Schroeder, 2012] [Alves et al., 2011]. Isso se deve principalmente ao fato de que são abordagens que auxiliam de maneira restrita o arquiteto na identificação de problemas arquiteturais. Além disso, são limitados também os auxílios na identificação de anomalias de código em sistemas multilinguagem [Terceiro et al., 2010] [Linos et al., 2007] e, consequentemente, a lista de candidatos tende a ser extensa e incompleta. Na próxima seção serão definidas as abordagens atuais para identificação de anomalias de código de relevância arquitetural em sistemas multilinguagem.

2.6.1 Inspeção Manual e Técnicas Automatizadas

Existe uma grande quantidade de abordagens que permitem a identificação de anomalias de código de relevância arquitetural [Linos et al., 2003] [Kontogiannis et al., 2006] [Wilkerson et al., 2012]. Essas abordagens podem utilizar uma forma manual ou automatizada de detecção. Em relação a abordagens manuais, a inspeção de software é uma técnica formal leve para inspecionar artefatos de software, a fim de identificar, por exemplo, problemas arquiteturais [Wilkerson et al., 2012]. Essa abordagem é também, atualmente, uma das mais comumente usadas para detectar anomalias de código em ambientes reais de desenvolvimento [Wilkerson et al., 2012]. No entanto, como será visto no Capítulo 3, essa abordagem exige um alto esforço para detecção.

Por outro lado, em relação às abordagens automatizadas, as abordagens atuais podem ser categorizadas em dois tipos: (i) extração e análise de dependências entre elementos de um sistema multilinguagem e (ii) extração e análise de métricas de software em sistemas multilinguagem. A primeira categoria agrupa trabalhos que fazem a reengenharia de sistemas gerando estruturas como grafos, que permitem a identificação das dependências entre os elementos e a construção de consultas para avaliação dessas dependências

[Kullbach et al., 1998] [Linos et al., 2003] [Kontogiannis et al., 2006] [Mayer and Schroeder, 2012] [Alves et al., 2011]. Essa abordagem permite a avaliação das dependências, mas é ineficaz na avaliação estrutural do código-fonte, uma vez que não avalia as suas várias outras propriedades estruturais. Por exemplo, número de linhas de código, coesão e complexidade ciclomática. A não avaliação dessas propriedades pode ocultar problemas que podem só ser avaliados através do uso dessas informações. Por exemplo, um método com uma grande quantidade de dependências pode não ser um problema se ele for um ponto de entrada de uma interface. No entanto, se ele possuir uma grande quantidade de linhas de código, ele pode estar assumindo responsabilidades que não deveria.

A segunda categoria agrupa trabalhos que avaliam a estrutura do código por meio de métricas que contabilizam propriedades estruturais do código. Esses trabalhos fazem a contabilização e a avaliação de métricas para diversas linguagens de programação. Essa contabilização é feita gerando um valor da métrica para elementos do programa em cada linguagem [Terceiro et al., 2010] [Nicolay et al., 2013]. No entanto, essa abordagem não oferece um bom suporte à avaliação de dependências entre componentes desenvolvidos em linguagens distintas. Outros trabalhos geram uma contabilização única de métricas para diferentes linguagens a partir de linguagens intermediárias, o que minimiza os problemas decorrentes da complexidade de avaliar diferentes sintaxes [Linos et al., 2007] [Nguyen et al., 2012]. As linguagens intermediárias permitem representar os elementos de código das diferentes linguagens de forma única. Nesse caso, muitas anomalias de código em cada linguagem podem requerer a definição de estratégias baseadas em propriedades estruturais específicas como, por exemplo, profundidade de herança em Java. Ou seja, a unificação da avaliação pode ocultar problemas decorrentes das especificidades das linguagens.

Estudos afirmam também que o uso restrito de métricas gera um conjunto de problemas [Marinescu, 2004] [Macia et al., 2012b]. Os principais problemas são o alto esforço na compreensão de como usar as métricas e a limitação da sua análise pelo uso individual de métricas, que, em muitos casos, reduz o número de resultados relevantes [Marinescu, 2004]. Essas limitações podem diminuir a eficácia na detecção de potenciais anomalias. No entanto, a identificação de anomalias inter-relacionadas é difícil com o uso individual de métricas. Dessa forma, o uso de estratégias baseadas na combinação de métricas pode melhorar a eficácia e reduzir o esforço na identificação de anomalias relevantes.

Dessa forma, a identificação de anomalias de código de relevância arquitetural requer o suporte de abordagens que utilizem como base métricas de

software. No entanto, devem mesclar estratégias que permitam a avaliação de dependências entre elementos implementados em diferentes linguagens em sistemas multilinguagem. Em outras palavras, as características chaves das duas categorias de abordagens mencionadas acima, deveriam ser unificadas em uma abordagem única.

Estudos afirmam que o uso de estratégias baseadas em métricas pode ser um caminho eficaz para identificá-las [Macia et al., 2012b] [Marinescu et al., 2010] [Moha et al., 2010] [Schumacher et al., 2010] [Wedyan et al., 2009]. Estratégias baseadas em métricas permitem raciocinar sobre a estrutura do código de forma mais abstrata e independente de linguagem. Além disso, elas usam heurísticas que capturam desvios quantificáveis de princípios de bom *design*. As estratégias permitem melhorar a compreensão das métricas, pois os analistas trabalham em um nível mais abstrato. Além disso, ele utiliza mecanismos de filtro e composição de métricas. O mecanismo de filtro permite limitar os resultados relacionados a propriedades específicas e a composição permite combinar diferentes métricas para produzir uma regra customizada [Marinescu, 2004]. Um exemplo de estratégia de detecção de anomalias de código é mostrado na Listagem 2.2. Essa estratégia identifica elementos de código infectados pela anomalia *God Class*. A métrica de software usada é a LOC, número de linhas de código. O limiar definido é 200 e o mecanismo de filtro indica que só serão coletados os elementos de código com LOC maior que 200.

```
God Class: (LOC > 200);
```

Listagem 2.2: Exemplo estratégias de detecção

Algumas ferramentas [Macia et al., 2012b] [Marinescu et al., 2010] se baseiam no uso de estratégias de detecção na identificação de anomalias individuais ou nas anomalias inter-relacionadas. Essas ferramentas auxiliam seus usuários, principalmente, em tarefas de manutenção, como refatoração e modificação das funcionalidades. De maneira geral, ferramentas atuais proveem suporte ao processo de identificação baseado em três etapas. A primeira etapa é a geração de arquivos auxiliares como diagramas de classes. Depois disso, as estratégias de detecção geram uma lista de entidades suspeitas. No último passo, os resultados são avaliados manualmente de forma a serem detectadas as anomalias relevantes e, se necessário, remoção de falsos positivos e identificação de falsos negativos. O primeiro e o último passo necessitam de intervenção de analistas.

2.6.2 SCOOP: Detectando Anomalias Arquiteturalmente Relevantes

Para permitir a identificação de anomalias de relevância arquitetural no presente trabalho, foi necessário selecionar uma das abordagens atuais para estratégias de detecção baseadas em métricas. Dentre as ferramentas disponíveis no estado da arte, a SCOOP possui um diferencial em relação às outras ferramentas. Além de oferecer suporte a todas as etapas citadas na Seção 2.6.1 anterior, ela permite o mapeamento de elementos de código e interesses arquiteturais.

Macia et al. [Macia et al., 2012b] propuseram novas estratégias de detecção com a exploração de dois tipos de informações adicionais. O primeiro são as relações entre elementos arquiteturais e os seus elementos de código correspondentes, aqui chamado de traços de código arquitetural. O segundo são relações entre diferentes tipos de anomalias de código. Dois tipos de traços arquiteturais são suportados: (i) mapeamento entre elementos arquiteturais, tais como componentes e conectores, e elementos de código e (ii) mapeamento entre elementos de código e interesses arquiteturais que não estão modularizados em componentes. O uso destes dois tipos de informações permite o desenvolvimento de métricas sensíveis à arquitetura. Portanto, elas proveem uma melhora em termos de eficácia no uso de estratégias baseadas em métricas. Por exemplo, métricas baseadas nos mapeamentos dos interesses são elaboradas e suportadas de forma a quantificar propriedades destes interesses arquiteturais no código fonte.

Macia et al. propuseram uma ferramenta, denominada SCOOP, que é um plugin para o Eclipse que suporta a definição de estratégias baseadas em métricas. Além disso, ele provê um ambiente de desenvolvimento que permite desenvolvedores construírem suas próprias estratégias de detecção no processo de identificação [Macia et al., 2012b]. Um exemplo de estratégia de detecção definida no SCOOP está representado na Figura 4.

```
codeanomaly<method> longmethod: (LOC > 15) or (CC > 5);

LOC: Número de Linhas de Código
CC: Complexidade Ciclomática
```

Listagem 2.3: Definindo estratégias no SCOOP usando métricas

Na Listagem 2.3 só são utilizadas métricas comuns como a número de linhas de código (LOC). No entanto, o SCOOP oferece também suporte a métricas baseadas em traços arquiteturais como o número de interesses por classe (NACC). Esse conjunto de regras baseadas em métricas define uma estratégia que identifica potenciais ocorrências de métodos longos.

Através do conjunto de estratégias baseadas em métricas, o SCOOP realiza a identificação das anomalias inter-relacionadas citadas na Seção 2.4. A utilização de referências de coocorrências de anomalias diminui o número de falsos positivos, tornando a identificação de anomalias de código de relevância arquitetural menos custosa.

No entanto, estudos recentes pouco mostram sobre a eficácia e esforço no uso das estratégias atuais, em especial, em sistemas que são implementados em diversas linguagens. Dessa forma, seria importante avaliar de forma mais ampla esses aspectos em sistemas monolinguagem e multilinguagem. Essa avaliação vai servir de insumo para a obtenção de melhores resultados em termos de eficácia e esforço na identificação de anomalias de código de relevância arquitetural como será apresentado no próximo capítulo.

3
Análise das Estratégias de Detecção de Código Convencionais

No Capítulo anterior, foram apresentados os principais conceitos relacionados ao presente trabalho e ao entendimento do presente capítulo. O foco no presente trabalho é na detecção de problemas arquiteturais através da identificação de anomalias inter-relacionadas, como visto na Seção 2.6.1. No entanto, existem poucos estudos que avaliem a eficácia dessas abordagens e o esforço associado, seja em software mono ou multilinguagem. Logo, o objetivo deste capítulo é responder a pergunta de pesquisa RQ1: *"Qual é a eficácia e o esforço associado ao aplicar estratégias na identificação de anomalias de código de relevância arquitetural?"*

Dessa forma, foi realizado um estudo de caso avaliando a eficácia e o esforço associado ao uso de estratégias convencionais na identificação de anomalias de código em um sistema multilinguagem da indústria. Os resultados desse estudo permitiram uma maior compreensão da eficácia e esforço usado nas estratégias atuais, identificando possíveis pontos de aperfeiçoamento. Muitos desses pontos compreendem tanto sistemas monolinguagem como sistemas multilinguagem. Porém, vamos tentar observar problemas que são exclusivos ou amplificados em sistemas multilinguagem. Com base nestas observações vamos identificar o(s) tal(tais) problema(s) para desenvolvimento de uma abordagem adequada para detecção de anomalias arquiteturalmente relevantes em sistemas multilinguagem. Esse estudo foi publicado no Simpósio 29th *Symposium On Applied Computing* - SAC 2014 [Ferreira et al., 2014].

Primeiramente será apresentado o desenho do estudo realizado, onde foram definidas as questões de pesquisa, os indicadores utilizados, o sistema alvo no qual foi executado o estudo de caso e, por fim, os procedimentos executados neste estudo de caso. Em seguida, são elucidados os resultados e discussões acerca da eficácia e do esforço, sem deixar de apresentar as ameaças à validade. O capítulo será finalizado com as conclusões sobre este estudo de caso.

3.1 Desenho do Estudo

O objetivo deste estudo é avaliar tanto eficácia quanto esforço associado quando aplicadas estratégias baseadas em métricas para detectar anomalias de código de relevância arquitetural. A avaliação é conduzida em um ambiente real de desenvolvimento de software multilinguagem. Para atingir esse objetivo, nós executamos um estudo de caso em uma empresa de software no qual nós comparamos uso de estratégias baseadas em métricas com uma estratégia baseada em inspeção *ad hoc*. O estudo de caso foi o método mais apropriado a adotar, dadas as dificuldades em controlar várias variáveis presente em uma configuração de ambiente real. Além disso, inspeção de software é uma técnica usada para inspecionar artefatos de software como visto na Seção 2.6.1. Ela tem sido o foco de mais de 400 estudos nos últimos 30 anos [Wilkerson et al., 2012]. Essas razões foram citadas e discutidas na Seção 2.6.1. De fato, inspeção *ad hoc* foi adotada na empresa de software onde realizamos nosso estudo de caso. Desta forma, podemos usar os resultados da inspeção *ad hoc*, já aplicada pela empresa, como base para uma avaliação comparativa.

As implicações desses resultados, no contexto de sistemas multilinguagem, são discutidas na Seção 3.2. Com base nos resultados dessa avaliação comparativa, fomos capazes de identificar possíveis melhorias para estratégias baseadas em métricas existentes para detecção de anomalias de código de relevância arquitetural (Seção 3.2.2). O restante desta Seção está estruturado da seguinte forma: Seção 3.1.1 apresenta as questões de pesquisa e define os indicadores utilizados neste estudo; Seção 3.1.2 descreve o sistema alvo do estudo. Finalmente, a Seção 3.1.3 relata os procedimentos seguidos durante a execução do estudo de caso.

3.1.1 Questões de Pesquisa e Indicadores

As seguintes sub-questões de pesquisa foram criadas para servir como norte para responder a RQ1: (RQ1.1) Qual é a diferença entre a eficácia de estratégias baseadas em métricas e inspeções *ad hoc* para detectar anomalias código de relevância arquitetural?, (RQ1.2) Qual é a diferença entre o esforço necessário para aplicar estratégias baseadas em métricas e inspeções *ad hoc* para detectar anomalias código de relevância arquitetural? Para nos ajudar a responder a estas questões, quatro indicadores foram definidos: *Pontuação de Eficácia, Precisão, Consistência e Esforço*. Os indicadores de Pontuação de Eficácia e Esforço foram usados para responder diretamente, RQ1.1 e RQ1.2, respectivamente. Os demais indicadores foram utilizados na análise de questões

de pesquisa para avaliação de propriedades complementares relacionadas com a eficácia e esforço. Eles fornecem resultados obtidos com estratégias baseadas em métricas contra inspeções *ad hoc*. Cada indicador é definido a seguir.

Pontuação de eficácia é um indicador que quantifica a qualidade dos resultados obtidos pela aplicação da estratégia avaliada. Nesse caso, qualidade significa quanto relevante são os resultados e o quão bons são os processos de detecção suportados pela estratégia. O valor desse indicador foi dado por um guia como o resultado de uma entrevista (Subseção 3.1.3). A unidade de medida é uma variação da pontuação de 0 a 10, onde 0 representa o menor nível de qualidade e 10 o maior.

Precisão mensura até que ponto a estratégia está habilitada a identificar o conjunto de anomalias de código de relevância arquitetural documentadas pelo guia. Este indicador representa a relação entre: (i) o número de anomalias detectadas pela estratégia que foi confirmado pelo guia, e (ii) o número de anomalias detectadas. Eq 3-1 mostra a fórmula usada para calcular este indicador.

$$\text{Precisão} = \frac{|guiaconfirmado \cap \text{estratégia}|}{|\text{estratégia}|} \tag{3-1}$$

Consistência representa a relação entre: (i) o número de anomalias detectadas pela inspeção *ad doc* confirmadas pelo guia e (ii) o número de anomalias de código detectadas pela estratégia baseada em métricas confirmadas como de relevância arquitetural pelo guia. Eq 3-2 mostra a fórmula usada para calcular este indicador.

$$\text{Consistência} = \frac{|guiaconfirmado \cap ad\ hocdetectado|}{|guiaconfirmado \cap \text{baseado métricas}|} \tag{3-2}$$

Esforço computa a quantidade de tempo (em horas) usada para identificar a lista de anomalias de relevância arquitetural. Esforço considera o tempo gasto com passos de configuração e detecção: (i) o passo 'geração de arquivos auxiliares', onde arquivos auxiliares são produzidos antecipadamente e são entradas para detecção de anomalia de código (ex. mapeamento de interesses arquiteturais), e (ii) o passo 'detecção', que representa as atividades atuais para detecção de anomalia.

3.1.2 Sistema Alvo

O sistema alvo deste estudo é um sistema web desenvolvido na empresa onde executamos nosso estudo de caso. O sistema V é um sistema multilinguagem, uma vez que, tem uma arquitetura em Java Enterprise Edition – JEE – e seus front end são implementado em Java Server Pages – JSP – e JavaScript. O sistema envolveu durante todo o desenvolvimento um total de 12 desenvolvedores. O sistema V foi escolhido porque cumpre alguns critérios relevantes para o objetivo principal do estudo. Primeiro, existia um grande número de versões de software disponível (210 versões), em razão do grande fluxo de mudanças originadas de refatorações e novas funcionalidades. Segundo, o sistema V é um sistema legado que entrou em uma fase crítica de degradação do projeto no qual as mudanças no sistema exigem um elevado esforço. À medida que o sistema evolui, altos investimentos em horas trabalhadas são gastos para melhorar e reestruturar a arquitetura. Assim, o sistema V promove uma diversidade de fonte de dados para nossa análise. Entre os diferentes cenários de manutenção avaliados, nos focamos nos cenários mais relevantes e complexos para o projeto de acordo com o guia (Seção 3.1.3), tal como a 'Refatoração TSV'. Cenários de manutenção são importantes em razão do seu foco em melhoria do sistema.

Refatoração TSV. A refatoração TSV é um exemplo de tarefa de manutenção importante que envolveu muitas modificações na estrutura do sistema V. Foi necessária a modificação de 23 classes Java e 7 outros tipos de arquivo, abrangendo mais de 60 horas-homem de trabalho realizadas por dois diferentes analistas. Além disso, dada a complexidade das mudanças, foi necessária cerca de 20 versões para fazer todas as modificações. O principal foco da refatoração TSV foi melhorar aspectos estruturais relacionados à geração de Valores separados por tabulação (acrônimo *Tab Separated Values* – TSV). A equipe de desenvolvimento falou sobre a dificuldade de reutilizar os módulos que cria ou como altera arquivos TSV. Desta maneira, a tarefa de refatoração TSV destinava-se a aumentar a facilidade de utilização dos módulos responsáveis pelos arquivos TSV. Refatoração TSV estava preocupada principalmente com a identificação e remoção de anomalias de relevância arquitetural.

3.1.3 Procedimentos do Estudo de Caso

Guia. O primeiro passo foi eleger desenvolvedores na companhia como guia para o estudo de caso. Ter um guia era necessário por muitas razões. Primeiro, nós precisávamos de alguém que nos ajudasse a coletar os dados

necessários para conduzir nosso estudo. Também, nós precisávamos de alguém que tivesse conhecimento sobre as decisões de projeto tomadas durante o cenário de refatoração TSV, então aqueles indicadores subjetivos deste estudo poderiam ser computados. Assim, o gerente atual do projeto do sistema V, que é também o arquiteto e o principal desenvolvedor do sistema, foi eleito como o guia para este estudo. Ele é gerente de projetos há mais de 2 anos e tem sido programador por mais de 10 anos, realizando sistematicamente inspeções de código para garantir a qualidade do código-fonte do sistema. No entanto, nós consultamos outros desenvolvedores sempre que foi necessário.

Estratégia baseada em inspeção *ad hoc*. O segundo passo era entender melhor a estratégia baseada em inspeção *ad hoc* pela equipe do sistema V. A inspeção *ad hoc* adotada pela equipe V não utiliza análise sistemática e o foco do revisor é genérico, ou seja, não é destinado a uma determinada lista de problemas arquiteturais (Seção 2.3). Revisores usam sua própria intuição e experiência. O processo de detecção de anomalias adotado pela equipe V seguiu esses passos: A equipe revisou o código procurando por anomalias de código de relevância arquitetural. Os fatores usados para identificar essas anomalias incluem: (i) informações de bugs reportados em sistemas de rastreamento de issues, (ii) número de linhas duplicadas no código (isto é, blocos de código clone), (iii) e dificuldade percebida na manutenção e extensão. A equipe também identificou principais classes que sofrem de anomalias e danos avaliados em classes vizinhas (classes dependentes ou classes do mesmo pacote).

Coleção de dados. O terceiro passo durante nosso estudo foi coletar dados primários em relação aos cenários de refatoração. As fontes de onde nós coletamos esses dados foram: documentação do sistema (manual do usuário e relatórios em sistema de gerenciamento de issues) e código fonte. A primeira parte do procedimento de coleta de dados foi extrair relatórios de cada refatoração do rastreamento de issues do sistema V. Esses relatórios serviriam para identificar quais foram as anomalias identificadas e demais informações úteis para a sa identificação, como o número de horas gastas na identificação. Desta maneira, nós consultamos no rastreamento de issues palavras chave especificas e identificamos 198 referências. O guia filtrou e aumentou essa lista com novos relatórios, resultando em um conjunto de 26 relatórios. Pela análise desses 26 relatórios, nós identificamos as versões mais relevantes em termos de atividades de refatoração. O guia foi consultado para confirmar todos esses achados e, como resultado um total de 5 versões foi escolhido. Essas versões foram selecionadas baseadas em proximidade temporal da refatoração e pequenas quantidades de modificações.

Infraestrutura criada. Com objetivo de aplicar a estratégia baseada em métricas, arquivos auxiliares foram fornecidos pelos desenvolvedores usando a ferramenta SOOP (Seção 2.6.2). O primeiro arquivo gerado foi o mapeamento de interesses, que define o mapeamento entre interesses arquiteturais e elementos de código. Interesses foram confirmados através de entrevista com o guia. O mapeamento dos 11 interesses no sistema foi baseado na nomenclatura de classes e análise de características desenvolvidas, de acordo com o conhecimento dos desenvolvedores. Desenvolvedores usaram estratégias baseadas em métricas encontradas por ser efetiva em detectar anomalias de relevância arquitetural em estudos anteriores [Macia et al., 2012a] [Macia et al., 2012b]. Estratégias utilizadas nesse estudo estão melhores descritas no Apêndice B denominado *Detection Strategies and Thresholds* do trabalho de Macia [Macia, 2013]. Além disso, os desenvolvedores especializaram as estratégias para utilizar métricas ou limiares que os desenvolvedores sentiram apropriado para o sistema V.

Computação dos valores dos indicadores. O próximo passo foi calcular os valores dos indicadores do estudo. Primeiro, nós identificamos anomalias de código baseando nos resultados da coleção de dados e categorizamo-las em tipos de Fowler et al. [Godfrey and Lee, 2000]. Depois disso, nós contamos o número de cada tipo de anomalias detectadas e o número de anomalias de código de relevância arquitetural. Nós usamos esses números na geração dos indicadores de precisão e de consistência (Seção 3.1.1). Depois disso, nós avaliamos as coocorrências de anomalias de código para identificar candidatas a anomalias de código de relevância arquitetural. Então, nós executamos o processo de entrevista com o guia para medir precisão e consistência da estratégia baseada em métricas e a Pontuação da eficácia. O processo para medir a eficácia seguiu os seguintes passos. No primeiro momento, nós apresentamos para o guia a lista de anomalias detectadas pela inspeção *ad hoc*. Então, o guia deu pontuações para a estratégia de acordo com a relevância das anomalias e o processo de detecção apoiado pela estratégia. Depois disso, a lista de anomalias detectadas pela estratégia baseada em métricas foi apresentada assim como o processo de detecção que foi explicado. Então, de novo, o guia deu uma pontuação para essa estratégia. Depois, nós fizemos questões subjetivas para melhor entender o relacionamento das pontuações do guia. Exemplos dessas questões são: *Q1 – Quais são as propriedades positivas da estratégia usada? Q2 – Quais são as dificuldades em usar cada uma das estratégias? Q3 – Quais as sugestões você dá para melhorar essa estratégia?* O indicador de esforço foi computado para a estratégia baseada em métricas no final baseado no tempo gasto para gerar os arquivos auxiliares e o tempo para configurar e executar a ferramenta SCOOP (Seção 2.6.2). O esforço para a inspeção *ad hoc* foi computado baseado

no tempo gasto registrando no JIRA.

3.2 Resultados e Discussão

Nessa Seção serão apresentados os resultados gerados a partir do estudo de caso. Além disso, é realizada uma discussão sobre os resultados e possibilidades de aperfeiçoamento das estratégias baseadas em métricas. Particularmente em sistemas multilinguagem, os resultados irão permitir avaliar se, por exemplo, as abordagens atuais permitem identificar anomalias relacionadas a componentes híbridos.

3.2.1 Eficácia e Esforço

Essa Subseção apresenta os resultados da eficácia, seguido da discussão sobre os indicadores de esforço. Os resultados são ilustrados baseados nas ocorrências individuais de anomalias de código identificadas na "Refatoração TSV". O guia reportou a lista a seguir como sendo as anomalias críticas e frequentes no sistema V: *Shotgun Surgery*, *Divergent Change*, Método Longo e *God Class*. Essas são anomalias simples que o guia considerou de alguma forma relevante, mas não necessariamente com uma grande relevância, ou seja, não necessariamente com impacto arquitetural. Ocorrências de anomalias de código *God Class*, por exemplo, são frequentemente localizadas em interfaces chave ou em classes abstratas em programas. Esse tipo de anomalias causa problemas na manutenção, pois quando era necessário fazer mudanças na interface, dependências arquiteturais eram quebradas. Outras ocorrências de anomalias seguem tendências similares. A Tabela 3.1 mostra a medição da precisão (2º coluna) para cada um dos tipos nas anomalias observadas na "Refatoração TSV". A tabela também distingue entre as quantias de anomalias simples detectadas com estratégias de detecção baseadas em métricas (3º coluna) daquelas consideradas relevantes para os desenvolvedores, isto é, o guia (4º coluna).

Precisão e consistência de estratégias baseadas em métricas. A análise da Tabela 3.1 revela que alguns falsos positivos (FP) ocorrem na detecção de anomalias de código usando estratégias baseadas em métricas. Essa descoberta pode ser observada nas linhas correspondentes aos resultados das anomalias *Divergent Change* e Método Longo. FP representa aquelas anomalias de código identificadas pelas estratégias baseadas em métricas (3º coluna) que não sejam consideradas relevantes para o guia (4º coluna). Somente uma instância de cada tipo de anomalia de código supracitada foi classificada

como falso positivo. Esse resultado é embasado pela análise de outro indicado: consistência. Nossa análise de consistência mostrou que todas as categorias tinham 1 como resultado. Baseado na análise dessas anomalias de código, nós identificamos um subconjunto de 14 anomalias múltiplas com impacto arquitetural. Esse tipo de anomalias teve resultado 1 na precisão em ambas as estratégias: baseadas em métricas e *ad hoc*. Esse é um indicador sobre a eficácia na identificação de anomalias de código com relevância arquitetural com as estratégias baseadas em métricas. O que significa que podemos identificar um número equivalente de anomalias de código relevantes se comparado com a inspeção *ad hoc*, que é estritamente baseada na experiência dos desenvolvedores atuais.

Tabela 3.1: Precisão de Estratégias Baseadas em Métricas na Detecção de Anomalias de Código Simples

Anomalia de Código	Precisão	Identificadas	Relevantes
Shotgun Surgery	1.00	16	16
Divergent Change	0.93	15	14
Método Longo	0.94	16	15
God Class	1.00	3	3

RQ1.1 - Eficácia: *Ad hoc* vs. Estratégias baseadas em Métricas. Nós analisamos se existia alguma diferença entre a eficácia das estratégias baseadas em métricas e a *ad hoc* para detecção de anomalias de código de relevância arquitetural. As pontuações dos desenvolvedores (Tabela 3.1) revelam que a eficácia é similar comparando as duas estratégias. Entretanto, nós olhamos de forma mais específica para os resultados da eficácia e esforço de agora se baseando em outros indicadores, assim como as respostas dos desenvolvedores de uma lista específica de perguntas. Descobertas dessa análise estão presentes abaixo:

Tabela 3.2: Resultados da Pontuação de Eficácia

Estratégia	Pontuação Eficácia
Inspeção *Ad Hoc*	8.5
Estratégias Baseadas em Métricas	8.5

Equívocos típicos: Anomalias relevantes. Nossos resultados quantitativos mostram que a mensuração da precisão é muito similar entre as duas estratégias. Em outras palavras, a inspeção *ad hoc* alcança resultados muito similares aos apresentados na Tabela 3.1. Entretanto, nas entrevistas, a percepção dos desenvolvedores indicava que a inspeção *ad hoc* era mais eficaz que as estratégias baseadas em métricas para identificar anomalias de código de re-

levância arquitetural. Em particular, observou-se que várias destas anomalias relevantes estavam associadas com módulos Java, usados por módulos JavaScript. Esta observação foi feita através do uso de inspeção *ad hoc*, já que as estratégias de detecção eram somente aplicáveis ao código Java. Como resultado, tornou-se claro que seria necessário investigar mais a existência de anomalias arquiteturalmente relevantes em componentes híbridos e como tratá-las para permitir a sua identificação. Existe um fator chave justificando essa percepção uniforme: os falsos positivos observados na "Refatoração TSV", por exemplo, representam a fragilidade consistentemente relatada pelos atuais desenvolvedores. As estratégias baseadas em métricas são menos eficazes para distinguir problemas arquiteturais reificados pelos elementos de código que realizam um interesse difuso (por exemplo, instâncias de *Divergent Change* relacionada à *Scattered Functionality*). Além disso, de acordo com os desenvolvedores, existe uma dificuldade no processo de calibração das métricas para detectar as anomalias mais relevantes nesse caso. Notamos que instâncias de *Scattered Functionality* também afetaram implementações de elementos em JavaScript. Portanto, o problema arquitetural de espalhamento de interesses somente poderia ser diagnosticado apropriadamente se houvesse suporte a análise multilinguagem.

RQ1.2 - Esforço: *Ad hoc* vs. Estratégias baseadas em métricas. Tabela 3.3 reporta os resultados relacionados ao indicador de esforço. Nós focamos primeiro na comparação e discussão do esforço requerido na etapa de detecção (3º coluna). O longo tempo requerido para identificar anomalias e código de relevância arquitetural foi uma desvantagem significativa no uso da inspeção *ad hoc*. O esforço total gasto pelos desenvolvedores aplicando a inspeção *ad hoc* foi oito vezes maior que na aplicação da estratégia baseada em métrica: 16 vs. 2 horas (3º coluna). Desenvolvedores gastam mais de oito vezes mais na identificação e análise dos elementos de código candidatos na apresentação das anomalias e segregação daquelas eram consideradas de relevância arquitetural.

De fato, a detecção com a inspeção *ad hoc* compreende diversas atividades: revisão do código fonte, filtro dos elementos de código, identificação das anomalias de código em potencial (candidatas), e seleção das candidatas. Entretanto, quando as estratégias baseadas em métricas são usadas, essa atividade basicamente compreende o filtro dos elementos de código principais e a seleção dos candidatos. Particularmente em relação a análise de um sistema multilinguagem, observou-se que parte do esforço na identificação de anomalias usando inspeção *ad hoc* foi dedicado no diagnóstico de elementos de código híbridos. Ou seja, uma análise mais ampla da identificação de anomalias em elementos híbridos permitiria oferecer um suporte mais amplo ao diagnóstico

das anomalias em sistemas multilinguagem.

Tabela 3.3: Métrica de Esforço: Etapas de Configuração e Detecção

Estratégia	Configuração (hora)	Detecção (hora)	Total (hora)
Inspeção *Ad Hoc*	0	16	16
Estratégias Baseadas em métricas	20	2	22

O esforço total com as estratégias baseadas em métricas é muito maior. A última coluna da Tabela 3.3 representa o esforço total gasto considerando ambas as etapas, configuração e detecção. Os resultados apresentados nessa tabela mostram que, em geral, e considerando as duas etapas, as estratégias baseadas em métricas requerem 37.5% mais esforço que a *ad hoc*. Essa notável diferença é principalmente devido ao alto tempo requerido no fornecimento das configurações (Seção 3.1.3). Entendemos que estas discussões de esforço de configuração seriam amplificadas se ferramentas para cada linguagem fossem utilizadas, por que permitiria identificar uma gama maior de anomalias em elementos híbridos. Além disso, o tratamento de dependências entre elementos híbridos permitiria o diagnóstico de anomalias inter-relacionadas híbridas, até então ocultadas pelas abordagens de estratégias baseadas em métricas atuais.

A Tabela 3.4 mostra o quanto de esforço foi gasto na geração de cada um dos conjuntos de arquivos auxiliares. O arquivo de representação do projeto é automaticamente gerado usando uma ferramenta de recuperação de projeto que analisa o código. O mapeamento de interesse foi fornecido pelo guia como discutido na Seção 3.1.3. A coluna "outros arquivos" envolve outros arquivos com a associação do conjunto de elementos do programa com informações arquiteturais no código. Resultados na Tabela 3.4 também mostram que 60% do esforço foi gasto na produção do mapeamento de interesses. O esforço gasto nos vestígios arquiteturais é também significante (30%) assim como esses mapeamentos não são frequentemente totalmente gerados automaticamente. Descobertas dessas análises são apresentadas na Seção 3.2.2.

Tabela 3.4: Métrica de Esforço: Etapas de Configuração e Detecção

Estratégia	RP (Hora)	MI (Hora)	OA (Hora)	Total (Hora)
Estratégias Baseadas em Métricas	2	12	6	20

RP=Representação do Projeto, MI=Mapeamento de Interesse, OA=Outros Arquivos

3.2.2 Aperfeiçoando as Estratégias Baseadas em Métricas

Nós também observamos fatores que poderiam melhorar a eficácia e reduzir o esforço da aplicação de estratégias baseadas em métricas. Resultados

na Tabela 3.2 revelam que a eficácia de ambas as estratégias não foi diferente. Entretanto, uma desvantagem no uso das baseadas em métricas é o elevado número de falsos positivos (Tabela 3.1) e negativos (Tabela 3.5). Nós identificamos que esses erros foram frequentemente relacionados a anomalias de relevância arquitetural, afetando os elementos do programa do Sistema V, implementado em múltiplas linguagens. Esse Sistema é implementado nas linguagens Java, JavaScript e JSP. Nós percebemos que as relações entre os elementos de código de integração implementados em duas ou mais linguagens podem ser muitas vezes a causa de algumas anomalias. Um exemplo de falso negativo identificado durante o estudo foi a respeito da classe TsvGenerator. Por exemplo, tsvGenerator.js é uma classe JavaScript que tem 195 LOC e tem sido caracterizada como uma God Class [Fowler, 1999]. Essa é uma classe com muitas linhas de código e contém alguma função que tem 190 LOC, caracterizando como um Método Longo. Também, 3 diferentes arquivos JSP dependem dessa classe, gerando modificações em muitos arquivos que foram caracterizados como uma *Shotgun Surgery*. Essas anomalias de relevância arquitetural representam um problema arquitetural chave. Tabela 3.5 mostra anomalias simples identificadas nesse contexto e resultados de consistência em relação a diferentes tipos de linguagens. Esta estrutura de tabela é similar à vista na Tabela 3.1.

Melhorando a eficácia da detecção. Tabela 3.5 mostra os resultados do indicador de consistência considerando. Observamos que ao considerar muitas linguagens nós identificamos anomalias que não foram inicialmente identificadas. Portanto, é necessário um conjunto de estratégias de detecção em partes do sistema implementado em diferentes linguagens. Isto é uma limitação no estado da arte porque atualmente não existem estratégias para identificar essas anomalias que combinam diretamente estruturas de código (e respectivas medidas) com base em diferentes linguagens. Esta é também uma limitação chave das ferramentas existentes de suporta às estratégias baseadas em métricas. Este problema é reforçado pelo fato que muitos dos projetos atuais são implementados com pelo menos quatro diferentes linguagens [Karus and Gall, 2011].

Tabela 3.5: Resultados da Consistência (Java, JavaScript e JSP)

Anomalia de Código	Consistência	Estratégia AH (relevante)	Estratégia BM (relevante)
Shotgun Surgery	1.13	18	16
Método Longo	1.13	17	15
God Class	1.67	5	3
Múltiplas Anomalias	1.14	16	14

AH=Estratégia Ah Doc; BM = Estratégia Baseada em Métricas

Outra questão crítica é o uso restrito de informação arquitetural nas estratégias baseadas em métricas existentes. Por exemplo, as classes AGenerator

e ATGenerator são mapeadas como parte percebida do Interesse TSVGenerator, ao passo que as classes IABean e ASBean são mapeadas como parte do Interesse Persistência. De acordo com o mapeamento de interesse fornecido pelo guia, alto acoplamento entre os interesses TSVGenerator e Persistência não devem existir. Entretanto, nós identificamos forte dependência entre as classes AGenerator e IABean, e também entre as classes ATGenerator e ASBean. AGenerator apresentou alto grau de acoplamento de classe, o que foi um sintoma de anomalia de código de relevância arquitetural [Godfrey and Lee, 2000]. Contudo, essas potenciais anomalias não foram detectadas com as estratégias baseadas em métricas não consideram acoplamentos entre múltiplos interesses arquiteturais realizados no programa. Os exemplos supracitados ilustram como estratégia baseadas em métricas poderiam explorar ainda mais informação adicional de mapeamento para melhorar a detecção de anomalias de relevância arquitetural.

Reduzindo esforço. Nossos resultados na Tabela 3.3 mostra que as estratégias baseadas em métricas requerem um esforço mais alto quando comparadas com aquelas *ad hoc*. Neste contexto, o primeiro passo poderia ser considerado com o objetivo de reduzir o esforço. Neste passo considera o esforço sobre a prestação de arquivos auxiliares nossas avaliações revelaram que, em especial, o maior esforço das estratégias baseadas em métricas resulta da geração de mapeamento de interesses como mostrado na Tabela 3.4. Uma ferramenta automatizada estável para gerar o mapeamento de interesses poderia reduzir o esforço, explorando ferramentas do estado da arte. No entanto, as ferramentas atuais de mapeamento manual e automático só permitem o mapeamento de elementos de código escritos em uma linguagem, e não são adequadas a sistemas multilinguagem [Feigenspan et al., 2010] [pure::variants, 2014]. Ou seja, o aperfeiçoamento das ferramentas atuais para o suporte a esse tipo de mapeamento é necessário.

3.2.3 Ameaças à Validade

A primeira ameaça à validade é a associação entre anomalias de código identifica pelas estratégias e aquelas consideradas como anomalias de código de relevância arquitetural. Foram confirmadas com o guia as indicações realizadas pela ferramenta e foram apresentadas as regras para o oráculo para minimizar erros na classificação de anomalias de código. Outra ameaça é a seleção das refatorações e versões analisadas. Tentamos mitigar essa ameaça por meio da identificação, com a equipe do sistema V, de quais foram as anomalias e refatorações mais relevantes que tiveram um impacto arquitetural elevado. Outra

ameaça à validade diz respeito à associação incorreta entre os interesses arquiteturais e elementos de código. No entanto, limitamos tal ameaça considerando apenas os mapeamentos produzidos por desenvolvedores e confirmados pelo gerente de projeto. Uma ameaça de validade da seleção de indicadores é a falta de um indicador recall, que é utilizado em alguns estudos para avaliar estratégias de detecção de anomalias (por exemplo, [Macia et al., 2012a] [Marinescu, 2004] [Wong et al., 2011]). No entanto, este indicador não pôde ser usado porque a equipe de desenvolvimento do sistema V não tem conhecimento explícito em relação a uma lista completa das anomalias de código dado o seu grande tamanho. Portanto, não poderíamos calcular falsos negativos.

3.3 Conclusão

Nosso estudo de caso longitudinal nos permitiu revelar que as inspeções *ad hoc* e baseada em métricas foram consideradas similares em termos de eficácia. As estratégias baseadas em métricas foram comparadas com a inspeção de código *ad hoc* executada pelos desenvolvedores. Isso indica que as estratégias baseadas em métricas podem ser aplicadas com um confiança considerável, especialmente quando os desenvolvedores mais experientes não estão disponíveis para revisar a arquitetura do código fonte.

Os resultados nos permitiram concluir também que o esforço gasto na detecção baseada em métricas de anomalias de código de relevância arquitetural pode representar um gargalo para os desenvolvedores. Uma elevada percentagem de tempo gasto foi necessária para identificar anomalias de relevância arquitetural. No entanto, nós também revelamos potenciais melhorias com o objetivo de reduzir o esforço. Em geral, a automação para as etapas de configuração também devem receber mais atenção dos pesquisadores.

Por outro lado, visando aumentar a eficácia no diagnóstico de anomalias de código de relevância arquitetural foram identificadas potenciais oportunidades. O processo de calibração das métricas para detectar anomalias mais relevantes vem sendo estudado por outros trabalhos como o de Silva [Silva, 2013]. Sobre o estudo de interesses arquiteturais, existem diversos trabalhos que vem estudando esse assunto como [Sant'Anna et al., 2007] [Boucké and Holvoet, 2006].

Foram identificadas também lacunas nas abordagens atuais de mapeamento e configuração para sistemas multilinguagem. As abordagens atuais não suportam o mapeamento automático para elementos de código escritos em diferentes linguagens. O suporte a esse tipo de mapeamento gera benefícios na redução do esforço na identificação de anomalias. No entanto, como o foco do

presente trabalho é na eficácia das estratégias baseadas em métricas, o aperfeiçoamento dessas ferramentas fica como sugestão de trabalho futuro.

Além disso, foram identificadas anomalias inter-relacionadas que requerem estratégias de detecção, considerando partes e estruturas do sistema implementadas em diferentes linguagens. De fato, esse problema deve ser ressaltado em razão do menor número de trabalhos que focam em problemas arquiteturais em sistemas multilinguagem [Linos et al., 2003] [Kontogiannis et al., 2006] [Wilkerson et al., 2012] em relação aos que avaliam sistemas monolinguagem. Diferente das outras potenciais oportunidades, esse problema envolve características inerentes a sistemas multilinguagem que são pouco explorados nos atuais trabalhos. Adicionalmente, esse cenário tende a ser pior quando a abordagem de detecção usada é baseada em métricas, pois a maioria dos trabalhos atuais se baseia no suporte a sistemas monolinguagem. Ou seja, apesar de nos últimos anos o número de sistemas multilinguagem terem aumentado significativamente [Mayer and Schroeder, 2012], o foco da maioria dos trabalhos é no desenvolvimento de suporte, ferramentas e realização de estudos em sistemas monolinguagem.

Por outro lado, os resultados mostram que é possível obter melhores resultados relacionados à eficácia com o tratamento de elementos e componentes híbridos. Além disso, o esforço associado pode ser reduzido em razão da análise de um número maior de elementos de código em um sistema multilinguagem. Logo, o desenvolvimento de um suporte a identificação de problemas arquiteturais em sistemas multilinguagem através do uso de estratégias baseadas em métricas traria grandes benefícios aos analistas e contribuiria com uma área pouco explorada no estudo da arte. Para permitir oferecer um suporte a sistemas multilinguagem é importante saber como definir estratégias para identificação de problemas arquiteturais em sistemas multilinguagem. Essas estratégias devem permitir identificar propriedades específicas da linguagem para cada elemento de código e dependências entre componentes híbridos.

Visando aperfeiçoar a identificação de anomalias de código inter-relacionadas em sistemas multilinguagem, o presente trabalho oferece um suporte a essa identificação de forma que o desenvolvedor possa detectar problemas arquiteturais de forma eficaz. O desenvolvimento do suporte, assim como a definição das estratégias para identificar problemas arquiteturais, vão ser discutidos no Capítulo 4 e a sua avaliação no Capítulo 5.

4
Estratégias de Detecção em Sistemas Multilinguagem

O Capítulo anterior mostrou um estudo sobre a utilização de estratégias baseadas em métricas na identificação de anomalias de código. O foco foi particularmente em anomalias de relevância arquitetural em um sistema multilinguagem do mercado. Essa avaliação apontou a necessidade de melhorias no suporte à identificação de anomalias de código. Em especial, o aumento da sua eficácia para identificar sintomas de degradação arquitetural. Notou-se no estudo anterior que existiam evidências iniciais de que anomalias arquiteturalmente relevantes não eram detectadas devida a falta de suporte multilinguagem. Além disso, sem este suporte não é possível investigar como anomalias arquiteturalmente relevantes se manifestam em um sistema multilinguagem. Tão menos é possível estudar se essas manifestações se diferem de anomalias da mesma natureza em sistemas multilinguagem.

Uma dessas necessidades é a melhoria da eficácia através do desenvolvimento de estratégias de detecção que consideram componentes do sistema implementadas em diferentes linguagens. Dentre os principais problemas discutidos na Seção 3.2.2, o foco desse trabalho é no tratamento de dependências em componentes e elementos de código híbridos. O objetivo deste Capítulo é responder a RQ2 do presente trabalho: *Como poderão ser definidas estratégias para identificação de problemas arquiteturais em componentes dependentes em sistemas multilinguagem?* Neste capítulo, serão apresentados os procedimentos para o desenvolvimento de uma abordagem que permite o suporte a análise de anomalias inter-relacionadas. A abordagem visa detectar sintomas de degradação arquitetural de forma eficaz em sistemas multilinguagem. A avaliação da eficácia da abordagem será discutida no Capítulo 5.

Este Capítulo apresentará as etapas sugeridas para identificar anomalias inter-relacionas em sistemas multilinguagem. Em seguida, tem-se a abordagem proposta por este trabalho. Nesta abordagem, é elucidada com mais clareza cada etapa realizada nesta construção. Primeiramente, configuração de um ambiente multilinguagem, depois, definição de métricas e estratégias, em seguida, detecção de anomalias inter-relacionadas híbridas, mais adiante, seleção dos tipos de anomalias inter-relacionadas e, por fim, modificação e configuração

de uma ferramenta existente. Além disto, foi realizada a atualização de métricas de acoplamento e a criação de novas métricas. Finaliza-se a abordagem fazendo o carregamento de estrutura do sistema multilinguagem em memória e adequação mecanismo de consulta e conclui-se o texto com os cometários finais.

4.1 Identificando Anomalias Inter-relacionadas em Sistemas Multilinguagem

Sistemas multilinguagem sofrem modificações com o tempo. Quando a manutenção desses sistemas não é feita de maneira adequada, podem surgir anomalias de código que tenham impacto arquitetural. Existem diversas abordagens [Kullbach et al., 1998] [Linos et al., 2003] [Kontogiannis et al., 2006] [Mayer and Schroeder, 2012] [Alves et al., 2011] que auxiliam o desenvolvedor na identificação dessas anomalias. Particularmente, as estratégias baseadas em métricas permitem raciocinar sobre a estrutura do código de forma mais abstrata e independente de linguagem (2.6.1). A avaliação dessa abordagem em um sistema do mercado (Capítulo 3) permitiu identificar necessidades de melhorias na sua eficácia e, consequentemente, na redução do esforço. Uma dessas necessidades é o suporte a identificação de anomalias de código que estão relacionadas a trechos do sistema desenvolvidos em diferentes linguagens (Seção 3.2.2).

A solução usada na identificação de anomalias de código em sistemas multilinguagem consiste no aperfeiçoamento das estratégias baseadas em métricas considerando características dos elementos de programa nas múltiplas linguagens. O diferencial destas estratégias é que, diferente das abordagens existentes, elas capturam anomalias nestas diferentes linguagens. Ainda mais importante é o fato que elas permitem detectar anomalias inter-relacionadas ocorrendo em componentes híbridos (Seção 2.5). Desta forma, desenvolvedores podem detectar diretamente estas anomalias inter-relacionadas. Além disso, desenvolvedores e pesquisadores podem estudar a relação delas com problemas arquiteturais. É importante ressaltar que a identificação de anomalias só foi feita nos níveis de declaração e operação (Subseção 2.5). Ou seja, não foram coletadas informações sobre módulos. Isso ocorreu em razão do fato que a vasta maioria dos tipos de anomalias documentadas nas diferentes linguagens sejam nos níveis de declarações e operações. A solução englobou os seguintes procedimentos:

– Configuração de um ambiente multilinguagem propício à identificação de anomalias inter-relacionadas;

– Definição de métricas e estratégias para diagnóstico de anomalias inter-relacionadas. Para isso foi usada a definição de anomalias em cada linguagem e o tratamento de dependências entre os componentes;

– Detecção de anomalias inter-relacionadas híbridas devido ao tratamento de dependências entre componentes híbridos, a partir de tais métricas e estratégias;

– Seleção de tipos de anomalias inter-relacionadas;

– Modificação e configuração de uma ferramenta existente para permitir o suporte a essas novas estratégias. A solução proposta será descrita em detalhes a partir da próxima Seção.

4.2 Abordagem Proposta

A natureza heterogênea de um sistema multilinguagem faz com que as abordagens atuais de estratégias baseadas em métricas necessitem de um aperfeiçoamento visando aumentar a sua eficácia (Seção 3.2.2). O aumento da sua eficácia vai permitir a identificação de um número maior de anomalias arquiteturalmente relevante no sistema e a diminuição do número de falsos negativos (Seção 3.2.2) no processo de identificação como foi diagnosticado na Seção 3.2.1. A abordagem proposta foi balizada nas 5 etapas mostradas na Figura 4.1. Cada uma das etapas será mais bem descrita nas próximas subseções.

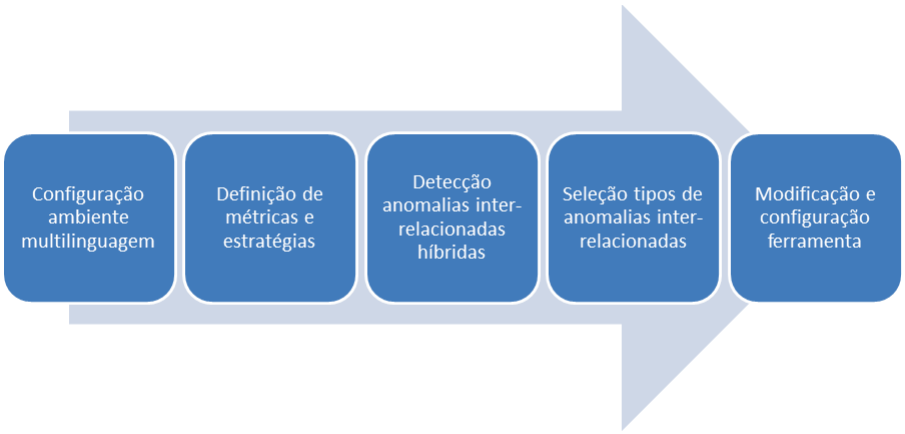

Figura 4.1: Etapas da solução proposta

4.2.1 Configuração do Ambiente Alvo

A configuração do ambiente tinha como objetivo definir um ambiente multilinguagem propício à identificação dessas anomalias. O ambiente deve oferecer condições de investigar a identificação de anomalias inter-relacionadas em sistemas compostos por duas ou mais linguagens, recorrentemente usadas em conjunção. Dessa forma, optamos por selecionar um subconjunto de linguagens frequentemente utilizadas em projetos de software. Existe uma grande variedade de linguagens que compõem sistemas multilinguagem no mercado e academia. Visando definir um escopo de avaliação adequado, selecionamos três linguagens tipicamente usadas em sistemas seguindo arquitetura cliente-servidor [Buschmann et al., 2007]. As linguagens escolhidas são as mais utilizadas para implementar estes tipos de arquiteturas. As linguagens escolhidas foram:

Servidor: Java e JSP **Cliente:** JavaScript

Java é uma linguagem de programação orientada a objetos popular entre os desenvolvedores de software. Ela foi desenvolvida com o intuito de permitir que, através do desenvolvimento de um único sistema, fosse possível rodar esse software em diversas arquiteturas de hardware. Essa interoperabilidade acontece, pois os sistemas são executados em uma máquina virtual denominada JVM [Arnold et al., 2000]. JavaServer Pages, ou JSP, é uma tecnologia que ajuda a geração dinâmica de páginas web baseada em HTML, XML e outras linguagens [Patzer et al., 2004]. Já o JavaScript é uma linguagem de programação interpretada que permite o desenvolvimento de scripts para interação com o usuário na camada cliente [Osmani, 2012].

A seleção dessas linguagens também se baseou em requisitos relacionados à: (i) variedade de linguagens pertencentes a cada camada da arquitetura cliente-servidor; (ii) popularidade da linguagem e (iii) linguagens que são comumente encontradas sendo usadas em conjunto. O primeiro requisito é contemplado levando em consideração que foram selecionadas duas linguagens que atuam na camada servidor e uma na cliente. Para avaliar o segundo requisito foi utilizado o difundido índice da empresa de software Tiobe [Tiobe, 2013]. O Tiobe realiza uma compilação mensal de uma lista das linguagens mais populares através do cálculo da frequência de palavras chaves em mecanismos de busca como Google [Google, 2014], MSN [MSN, 2014] e Yahoo [Yahoo, 2014].

Na Figura 4.2 é possível observar o índice do Tiobe [Tiobe, 2013]

relacionado a agosto de 2013, período da configuração do ambiente alvo. Java e JavaScript estão entre as 10 primeiras linguagens (TOP 10) do índice. Java está posicionada no 1º lugar, enquanto JavaScript está na 9º lugar. É importante ressaltar que de acordo com a coluna "*Delta in Position*", todas as duas linguagens subiram de colocação em relação ao mesmo período do ano de 2012.

Para que uma linguagem seja selecionada para participar da listagem ela deve ter: (i) deve indicar claramente de que se trata de uma linguagem de programação e (ii) deve ser Turing completa. Por conta do tópico (i), a linguagem JSP não está no índice. Isso acontece, pois JSP é considerada uma linguagem de script e, além disso, deve ser usada em conjunto com Java. Essa última afirmação é o ponto de partida para justificar o terceiro requisito. JSP é uma linguagem que roda no servidor, mas tem o papel de gerar código HTML/CSS/JavaScript de forma dinâmica. Além de gerar JavaScript, é possível haver trocas de informações entre arquivos JSP e arquivos originalmente escritos em JavaScript. Essas dependências podem ser relevantes para identificação de anomalias inter-relacionadas com impacto arquitetural.

Position Aug 2013	Position Aug 2012	Delta in Position	Programming Language	Ratings Aug 2013	Delta Aug 2012	Status
1	2	↑	Java	15.978%	-0.37%	A
2	1	↓	C	15.974%	-2.96%	A
3	4	↑	C++	9.371%	+0.04%	A
4	3	↓	Objective-C	8.082%	-1.46%	A
5	6	↑	PHP	6.694%	+1.17%	A
6	5	↓	C#	6.117%	-0.47%	A
7	7	=	(Visual) Basic	3.873%	-1.46%	A
8	8	=	Python	3.603%	-0.27%	A
9	11	↑↑	JavaScript	2.093%	+0.73%	A
10	10	=	Ruby	2.067%	+0.38%	A
11	9	↓↓	Perl	2.041%	-0.23%	A
12	15	↑↑↑	Transact-SQL	1.393%	+0.54%	A
13	14	↑	Visual Basic .NET	1.320%	+0.44%	A
14	12	↓↓	Delphi/Object Pascal	0.918%	-0.09%	A--
15	20	↑↑↑↑↑	MATLAB	0.841%	+0.31%	A--
16	13	↓↓↓	Lisp	0.752%	-0.22%	A
17	19	↑↑	PL/SQL	0.751%	+0.14%	A
18	16	↓↓	Pascal	0.620%	-0.17%	A-
19	23	↑↑↑↑	Assembly	0.616%	+0.11%	B
20	22	↑↑	SAS	0.580%	+0.06%	B

Figura 4.2: Índice de Linguagem de programação do Tiobe referente ao mês de agosto de 2013 [Tiobe, 2013]

Já a Figura 4.3 mostra o número de projetos, número de contribuidores (pessoas que trabalham nos projetos) e número de contribuições (*commits*) relacionados a cada linguagem desde o período de 1996. Esses dados foram retirados do sítio eletrônico Ohloh [Ohloh, 2013]. Ohloh provê serviços *web* e uma plataforma eletrônica que tem como objetivo mapear o panorama de software de código aberto. É possível observar que essas linguagens foram utilizadas em milhares de projetos e contribuidores ao longo do tempo.

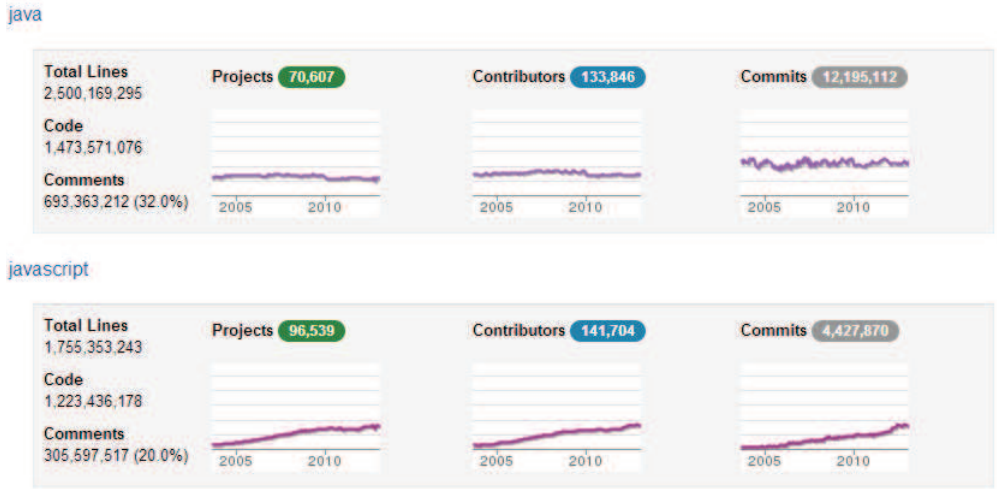

Figura 4.3: Número de projetos, contribuidores de projetos e commit [Ohloh, 2013]

É importante também ressaltar que a avaliação de elementos de código escritos na linguagem Java só compreendeu a avaliação de código exclusivamente escrito em Java. Assim como os elementos escritos em JavaScript, somente código JavaScript foi avaliado. Por outro lado, a avaliação dos elementos JSP compreenderam código de anotações JSP e código HTML.

4.2.2 Definição de Métricas e Estratégias

Conforme apresentado anteriormente, primeiramente foi definido o conjunto de linguagens que farão parte do ambiente alvo. Após a configuração do ambiente através da seleção do conjunto de linguagens, o próximo passo foi a definição das métricas e estratégias para detecção de anomalias de relevância arquitetural e anomalias inter-relacionadas em sistemas multilinguagem. Dessa forma, o primeiro passo foi a definição do conjunto de anomalias que seria identificado nos sistemas multilinguagem.

Definição do Conjunto de Anomalias

O conjunto de anomalias que foram identificadas em sistemas multilinguagem englobam as anomalias de relevância arquitetural e anomalias inter-relacionadas referentes a uma linguagem em específico. É importante ressaltar que essas anomalias já são identificadas na avaliação de sistemas monolinguagem. Porém, o conjunto engloba também anomalias inter-relacionadas híbridas (Seção 2.5) que consideram dependências em componentes híbridos. A identificação de anomalias como essas não seria possível sem o tratamento adequado a

essas características de sistemas multilinguagem. Ou seja, o uso das abordagens convencionais oculta essas anomalias em razão de desconsiderarem dependências entre componentes híbridos.

É importante notar que esse conjunto pode ser constituído de subconjuntos de anomalias de relevância arquitetural e inter-relacionadas, específicas a cada linguagem suportada no ambiente. Ou seja, se houver um componente A escrito na linguagem A e outro componente B na linguagem B, é possível que o conjunto total de anomalias de relevância arquitetural identificadas no sistema tenha anomalias de código para linguagens A e B. Consequentemente, foi necessário definir o subconjunto de anomalias que seriam detectadas para cada linguagem e para as anomalias inter-relacionadas híbridas. As métricas e estratégias selecionadas estão relacionadas ao subconjunto de linguagens que foi anteriormente definido (Seção 4.2.1).

Por exemplo, de acordo com a Figura 2.2, as declarações TsvInterface.js e ReportGeneric.js foram desenvolvidas usando a linguagem JavaScript. Por outro lado, os arquivos aTsv.jsp, bTsv.jsp e cTsv.jsp foram desenvolvidos usando Jsp. Ao identificar as anomalias de código individualmente para cada linguagem, como visto na Seção 2.4, teremos que a operação selectTsvType foi diagnosticada como um método longo. Nenhum dos outros elementos de código envolvidos no problema arquitetural teriam indicações de anomalias de código. Apesar da anomalia método longo poder indicar um problema arquitetural, a avaliação das anomalias inter-relacionadas é um forte indicador de problemas arquiteturais (Seção 2.4).

Dessa forma, somente o tratamento das dependência entre elementos híbridos permitiria identificar com rigor que se trata de um problema arquitetural. O tratamento das dependências entre elementos híbridos permitiria identificar o acoplamento entre esses elementos. Consequentemente, teríamos o diagnóstico de que a operação selectTsvType também é afetada pela anomalia *Shotgun Surgery*. Consequentemente, poderíamos ter uma indicação da anomalia inter-relacionada *Multiple-Anomaly*.

Baseado nas necessidades descritas anteriormente, foram definidas as diretrizes usadas para caracterização de anomalias de código para cada uma das linguagens selecionadas. A definição de anomalia de código de relevância arquitetural em sistemas multilinguagem vai seguir a mesma definição realizada na Seção 2.4. Além disso, foi necessário definir as diretrizes usadas para identificar dependências entre componentes escritos em diferentes linguagens. Isso vai permitir identificar anomalias inter-relacionadas híbridas.

Na definição das anomalias de código para linguagem Java foi utilizada a referência do Fowler [Fowler, 1999]. Por exemplo, *God Class*, *Shotgun Surgery*

e *Divergent Change*. Para a definição das anomalias de código para linguagem JSP não foi encontrada na literatura uma referência que reporta uma lista de anomalias de código em componentes que usam essa linguagem. Dessa forma, foram utilizadas as referências de Patzer, que define boas práticas de projeto para implementações JSP [Patzer et al., 2004], e a de Fowler [Fowler, 1999]. Ou seja, foram identificadas quais das anomalias genéricas reportadas por Fowler que se adequam as boas práticas reportadas por Patzer. Dessa forma, foram selecionadas as anomalias *God Class*, *Shotgun Surgery* e *Divergent Change* reportadas por [Fowler, 1999]. O conceito dessas anomalias para componentes JSP é análogo ao utilizado para Java.

Na definição das anomalias de código para linguagem JavaScript foram utilizadas as definições descritas por Fard [Fard and Mesbah, 2013]. Fard apresenta 13 definições de anomalias de código. As definições do Fard foram selecionadas, pois representam diferentes problemas no código JavaScript. Esses problemas podem ser desde Lista Longa de Parâmetros, que representa um grande número de parâmetros para uma função, até Encadeamento de Escopo Longo, quando funções possuem múltiplos escopos. Além disso, o trabalho do Fowler foi selecionado, pois vem sendo utilizado nos últimos anos como referência para vários trabalhos.

As 13 anomalias definidas por Fard foram baseadas na adaptação de algumas anomalias genéricas para linguagens orientadas a objetos e na definição de novas anomalias. Na adaptação das anomalias genéricas foi necessário substituir a noção de classe para objeto, pois JavaScript é uma linguagem livre do uso de classes (*class-free*).

Dessas anomalias, 7 são anomalias genéricas. A primeira foi a *Empty catch blocks* que ocorre quando o bloco *catch* está vazio. Esses blocos estão relacionados ao tratamento de exceções. O problema associado a essa anomalia é que temos um baixo entendimento do bloco *try*. *Large object* e *Long functions* são análogas aos conceitos de *God class* e método longo, respectivamente [Fowler, 1999]. *Long parameter* ocorre quando uma operação possui uma grande quantidade de parâmetros. Nesse caso, seria interessante a utilização de um objeto para representação das propriedades que são passadas como parâmetro. *Switch statements* ocorre quando um bloco *switch* possui uma grande quantidade de opções. Nesse caso, o alto número de opções de decisão torna alta a complexidade da análise do código e aumenta a possibilidade de duplicação de código. *Unused/dead code* representa trechos de código que nunca são executados. Ou seja, indica a existência de trechos de código que tornam mais complexa a análise do código, mas não contribuem como funcionalidades no sistema.

Além disso, foram definidas 6 novas anomalias de código para JavaScript. A primeira foi a *Closure Smells* que é uma anomalia que indica problemas nas definições *Closure*. O *Closure* é utilizado, pois permite emular noções de orientação a objetos. Em declarações JavaScript é possível usar código HTML e CSS, por exemplo. No entanto, esse uso torna alto o acoplamento entre essas linguagens. Logo, a anomalia *Coupling between JavaScript, HTML, and CSS* indica o uso de outras linguagens em arquivos estritamente usados para uma linguagem.

A anomalia *Excessive Global Variables* indica que existe uma grande quantidade de variáveis globais definidas. Ou seja, variáveis que podem ser usadas em qualquer local do sistema somente com o nome. O problema nesse caso é que uma variável A pode ser definida como global em um local e, consequentemente, sobrescrever uma variável A local. A anomalia *Long Message Chain* indica que existe uma cadeia de chamadas longa no sistema usando o ".". Por fim, *Nested Callback* ocorre quando existe um encadeamento de callback em uma operação do sistema.

Das anomalias JavaScript anteriormente definidas, só serão utilizadas as seguintes anomalias no presente trabalho: *Large Object*, *Lazy Object*, *Long parameter list*, *Switch statements*, *Long Message Chain*, *Nested Callback* e *Refused Bequest*.

4.2.3 Detecção de Anomalias Inter-relacionadas Híbridas

Componentes híbridos podem ter dependências. A avaliação dessas dependências compreendeu: (i) definição do modelo de dependências entre os componentes escritos em diferentes linguagens; (ii) seleção de técnicas para identificação desses modelos no código e (iii) integração da técnica à abordagem proposta. Os pontos (ii) e (iii) serão descritos na Seção 4.2.5.

Em relação ao ponto (i), a definição do modelo de dependências tem como objetivo identificar qual modelo que será utilizado para definir o que é uma dependência entre componentes híbridos. As dependências em componentes híbridos são realizadas através da utilização de *frameworks*. O uso de um *framework* permite o suporte à interação entre códigos escritos entre diferentes linguagens.

Os modelos são definidos a partir dos padrões pré-estabelecidos de dependências entre linguagens definidos pelas referências dos *frameworks* utilizados. É importante ressaltar que, nesse caso, o conjunto de modelos compreendem aqueles que suportam o subconjunto de linguagens selecionados na Seção 4.2.1. A definição do modelo teve como base os seguintes critérios: (i) popularidade

do *framework*; (ii) Baixa variabilidade nas formas de utilização do *framework*. Esse critério foi útil para facilitar o tratamento de todas as formas de comunicação entre os componentes e (iii) Disponibilidade de documentação de referência dos padrões de dependência do *frameworks*.

Os padrões selecionados para cada uma das dependências entre componentes híbridos estão listados a seguir. A descrição do processo de integração será realizada na Subseção 4.2.5.

- **Comunicação de JavaScript para Java:** O padrão utilizado foi o do *framework* Dwr [DWR, 2014]. O Dwr é um *framework* que permite fazer chamadas do JavaScript para Java, assim como o inverso. No entanto, no presente trabalho, somente o primeiro caso foi avaliado devido ao critério (i). Um exemplo de uso do Dwr é mostrado na Figura 4.4. Esse é um exemplo retirado do Tudu-Lists, sistema de gerenciamento de tarefas. O sistema será mais bem descrito na Seção 5.2. A operação JavaScript **completeTodo** realiza duas chamadas para os serviços Java disponibilizados através da interface **todos**. As chamadas são para os métodos **completeTodo** e **forceGetCurrentTodoLists**. Cada uma dessas chamadas é contabilizada como uma dependência.
- **Comunicação de JSP para Java:** Os padrões utilizados foram o do *framework* Struts [Struts, 2014] e Spring [Spring, 2014]. A Figura 4.5 mostra um exemplo de chamada usando Spring. A chamada é realizada através da definição **/tudu/rss**. A indicação **rss** indica qual a operação que será chamado através de um mapeamento prévio na declaração Java como pode ser visto na Figura 4.6.

```
function completeTodo(todoId) {
    dwr.engine.beginBatch();
    todos.completeTodo(todoId, replyRenderTable);
    todos.forceGetCurrentTodoLists(replyCurrentTodoLists);
    dwr.engine.endBatch();
    tracker('/ajax/completeTodo');
}
```

Figura 4.4: Exemplo de dependência entre elementos de código JavaScript e Java usando DWR [DWR, 2014]

```
<c:if test="${not empty todoList}">
 <c:if test="${todoList.rssAllowed eq true}">
  <a href="${ctx}/tudu/rss?listId=${todoList.listId}">
   <img width="30" height="14" alt="RSS" src="${staticConte
```

Figura 4.5: Exemplo de dependência entre elementos de código JSP e Java

```
@RequestMapping("/rss")
public ModelAndView showRss(@RequestParam String listId,
        throws Exception {

    ModelAndView mv = new ModelAndView();
    TodoList todoList = todoListsService.unsecuredFindTod

    if (todoList.isRssAllowed()) {
        mv.addObject("todoList", todoList);
        mv.addObject("link", request.getScheme() + "://"
                + request.getServerName() + ":" + request
                + request.getContextPath() + "/tudu/lists

        mv.setView(new InternalResourceView("/servlet/rss
    } else {
        if (log.isDebugEnabled()) {
            log.debug("Rendering RSS feed for Todo List I
                    + todoList.getListId() + "' is not al
        }
```

Figura 4.6: Exemplo de declaração Java mapeada para uma chamada no código JSP

4.2.4 Seleção das Categorias de Anomalias Inter-relacionadas

Macia et al mostrou evidências de que a utilização de anomalias inter-relacionadas [Macia et al., 2012b] pode ser eficaz no processo de identificação de anomalias arquiteturalmente relevantes, como afirmado na Seção 2.4. Macia et al definem categorias de anomalias inter-relacionadas com relevância arquitetural que facilitam o seu entendimento e análise. A identificação das anomalias inter-relacionadas foi realizada tanto em componentes comuns como os componentes híbridos. Consequentemente, também foram avaliados elementos de código comuns e híbridos. Ou seja, as anomalias inter-relacionadas foram utilizadas também em razão do seu caráter independente de linguagem e, portanto, torna-se possível identificar instâncias dos elementos implementados em diferentes linguagens. De acordo com os resultados reportados na Seção 3.2.2, a incorporação dos componentes e elementos de código híbridos na análise vai permitir o aumento da eficácia na identificação de anomalias inter-relacionadas.

Estão listadas a seguir as categorias mais relevantes para o presente trabalho, pois foram selecionadas as anomalias inter-relacionadas que pudessem ser englobadas pela maior parte das linguagens definidas anteriormente. Ou seja, as anomalias inter-relacionadas não utilizadas eram restritas a avaliação hierárquica, tais como *Hereditary Anomaly* e *Mutant Anomaly*. Essa restrição tornaria inviável a avaliação das linguagens JavaScript e JSP que não possuem

suporte hierárquico semelhante a linguagens orientadas a objetos. Além disso, não foram utilizadas também as anomalias inter-relacionadas que exigiam informações de interesses arquiteturais como o *Concern Overload* e *Misplaced Concern*, pois a avaliação das anomalias inter-relacionadas relacionadas a interesses arquiteturais não faz parte do escopo do trabalho.

- *Multiple-Anomaly*: São elementos de código que estão infectados por mais de uma anomalia de código. Esse padrão pode indicar que a distribuição de responsabilidades entre os elementos do componente pode estar sendo feita de maneira inadequada. Além disso, pode indicar um aumento na complexidade esperada para esses elementos. No caso de sistemas multilinguagem, esse padrão é especialmente importante na análise de componentes híbridos. Isso acontece pelo fato de que componentes híbridos podem, de forma inadequada, assumir responsabilidades de componentes que são escritos em outra linguagem. A identificação desse problema tem uma maior complexidade em componentes híbridos, pois exige do analista um conhecimento mais aprofundado em ambas as linguagens em que o componente híbrido e os componentes dependentes foram desenvolvidos.
- *Similar Anomalous Neighbors*: É a existência de sintomas similares de anomalias de código entre elementos de código pertencentes ao mesmo componente. Esse padrão pode indicar de maneira geral grande número de funcionalidades complexas relacionadas a esses elementos ou problemas em componentes externos que esses elementos são dependentes. Em sistemas multilinguagem essa anomalia inter-relacionada é particularmente interessante na avaliação de componentes híbridos que são pontos de entrada de comunicação entre diferentes camadas do sistema.
- *External Addictors*: Elementos que utilizam informações de diversos componentes. O "uso de informação" é classificado como invocação a operações, acesso a atributos ou hierarquia. Esse padrão pode indicar a existência de um grande número de responsabilidades a esses elementos e acoplamentos inesperados. Além disso, ele pode ser considerado um elemento instável uma vez que sofre impacto de mudanças de diferentes componentes. Esse padrão pode ser avaliado no contexto de uma declaração ou de uma operação. Em sistemas multilinguagem, a existência dessa anomalia inter-relacionada em um componente híbrido pode indicar um grande uso de informações de componentes escritos em outras linguagens. Isso é particularmente danoso em componentes híbridos, pois se houver dependências com um componente de tipagem dinâmica como o JavaScript, as mudanças em um componente podem ser ocultadas em outros, gerando problemas nas dependências.

– *External Attractors*: Relacionado a um elemento que é acessado por muitos elementos externos definidos em vários componentes. A identificação dessa anomalia inter-relacionada pode sugerir dentre outros problemas centralização de diferentes serviços e a existência, consequentemente, de uma interface anômala que permite o acesso a esses diferentes recursos. Esse padrão pode ser avaliado no contexto de uma declaração ou de uma operação. Em sistemas multilinguagem, a existência dessa anomalia inter-relacionada gera impactos análogos ao *External Addictors*.

– *Replicated External Network*: Elementos do mesmo componente que usa as mesmas informações de diferentes elementos externos. Esse padrão pode indicar a presença de uma interface redundante no módulo ocasionando aumento do acoplamento e dependências inesperadas entre os módulos. Em sistemas multilinguagem essa anomalia inter-relacionada pode indicar problemas análogos ao reportado na anomalia inter-relacionada *External Addictors*.

Dessa forma, baseado nas evidências nos estudos reportados por Macia et al [Macia et al., 2012b], foram utilizadas as anomalias inter-relacionadas na identificação de problemas arquiteturais visando tornar eficaz a detecção desses problemas. Além disso, na Seção 5 serão apresentados os resultados na avaliação dessa proposta de solução baseado nas categorias de anomalias inter-relacionadas.

4.2.5 Modificação e Configuração de uma Ferramenta Existente

Com o objetivo de identificar problemas arquiteturais através da identificação de anomalias inter-relacionadas em sistemas multilinguagem, foi selecionada uma ferramenta existente no estado-da-arte. Essa ferramenta vai permitir também automatizar a tarefa de identificação. Como dito na Seção 2.6, não existem ferramentas que ofereçam suporte a identificação de anomalias inter-relacionadas que avaliem a estrutura interna dos elementos e dependências em componentes híbridos. Ou seja, foi necessário escolher uma ferramenta que mais se assemelhe com os objetivos do trabalho e estendê-la.

O SCOOP foi a ferramenta escolhida para o desenvolvimento de uma abordagem que suporte sistemas multilinguagem. Ela foi escolhida em razão de ser uma ferramenta representativa no estado da arte, pois suporta estratégias baseadas em métricas e permite mapeamento de componentes arquiteturais. Além disso, possui o potencial de permitir fazer o mapeamento de interesses arquiteturais com elementos de código fornecidos por analistas, como dito na Seção 2.6.2.

As modificações realizadas no SCOOP foram baseadas em três pontos específicos. O primeiro ponto foi o suporte a identificação de anomalias de código para cada linguagem. O segundo ao suporte ao modelo de dependências que irá permitir identificar anomalias em componentes híbridos. O terceiro ponto foi a adequação ao suporte de anomalias inter-relacionadas em sistema multilinguagem.

Para oferecer suporte a identificação de anomalias de código de cada linguagem em específico, não foram feitas modificações na infraestrutura da ferramenta. No entanto, foi necessário atualizar a lista de métricas suportadas pela ferramenta visando incorporar as novas métricas que serão usadas para permitir a coleta de informações para a identificação de anomalias de código para as linguagens diferentes de Java. Por exemplo, número de parâmetros. Essa métrica contabiliza o número de parâmetros de uma operação. Além disso, foi necessário criar as regras para identificação das anomalias de código de acordo com as definições dos padrões que foram reportados na Seção 4.2.2.

Em relação ao segundo ponto, a identificação de dependências entre elementos de código escritos na mesma linguagem já era suportada pela ferramenta. Por outro lado, foi necessária a criação de um mecanismo que permitiu a identificação das dependências entre os elementos de código como definido pelo modelo de dependências na Seção 4.2.2. Esse mecanismo foi criado através de um conjunto de *parsers* que identificaram as dependências definidas anteriormente. Esse mecanismo permitiu a criação de novas métricas, tal como *Number Of Used External Classes Per Function From JavaScript To Java (NOEC-F)* que será descrita na Seção 4.2.5 e a atualização dos valores de métricas já existentes tal como FanIn.

A criação de novas métricas foi necessária em razão da inexistência de métricas específicas para a contabilização de dependências entre elementos híbridos relevantes para o sistema. O uso de métricas específicas permitiria criar regras com filtros e *thresholds* para estratégias de detecção adequadas à identificação de certas anomalias de código para elementos híbridos tal como *Shotgun Surgery* em uma interface com um ponto de comunicação entre camadas diferentes do sistema. Nesse caso, desejava-se diferenciar as estratégias *Shotgun Surgery* para detecção de anomalias em outros elementos do sistema e elementos híbridos entre camadas específicas. Essa abordagem permitiu obter referências relacionadas ao conjunto alvo, ou seja, elementos híbridos.

Além disso, hoje não existem ferramentas que contabilizam as dependências entre elementos híbridos. Ou seja, as métricas relacionadas a dependências entre esses elementos como a FanOut são geradas apenas para dependências entre elementos de mesma linguagem. Dessa forma, foi desenvolvido um meca-

nismo que permitiu fazer a atualização dos valores para as métricas relacionadas a dependências, ou seja, métricas de acoplamento, como FanIn e FanOut.

Em relação ao ponto três, não foram criados mais padrões de anomalias inter-relacionadas. Dessa forma, as modificações em relação a esse ponto consistiram em estender a identificação das anomalias existentes para o suporte a elementos híbridos que as novas linguagens estão envolvidas.

O SCOOP é uma ferramenta que atualmente só suporta a linguagem Java. Foram necessárias mudanças em algumas estruturas internas do SCOOP para que ele fosse capaz de suportar a identificação de anomalias inter-relacionadas em sistemas multilinguagem. As estruturas que foram modificadas são: (i) carregamento da estrutura do projeto a ser avaliado em memória. Foi realizada uma extensão da abordagem para permitir a avaliação de arquivos JavaScript e JSP; (ii) uso de uma estrutura similar à geração de estruturas Prolog para realizar consultas nas linguagens diferentes de Java. A abordagem anterior não suportava outras linguagens; (iii) Mecanismo para atualização de métricas de acoplamento para componentes escritos em diferentes linguagens e (iv) Criação de novas métricas de acoplamento para dependências entre componentes escritos em diferentes linguagens.

Na Figura 4.7 é possível observar a nova arquitetura da ferramenta. Em vermelho estão indicados os pontos de modificação no uso da ferramenta. Os arquivos *metrics.csv* e *Rule.rule* incorporaram novas métricas e novas estratégias para identificação de anomalias relacionadas às novas linguagens, respectivamente. Além disso, o componente *Logical Statements Generator* foi modificado de acordo com os quesitos (i) e (ii), gerando uma nova árvore AST (*Abstract Syntax Tree*) anotada com métricas de software. O mecanismo de Coleta de Métricas foi estendido para permitir a modificação do quesito (iii). Além disso, foram criadas duas novas métricas de acordo com o quesito (iv). O componente relacionado à detecção de anomalias inter-relacionadas também foi modificado para permitir identificar anomalias inter-relacionadas em sistemas multilinguagem.

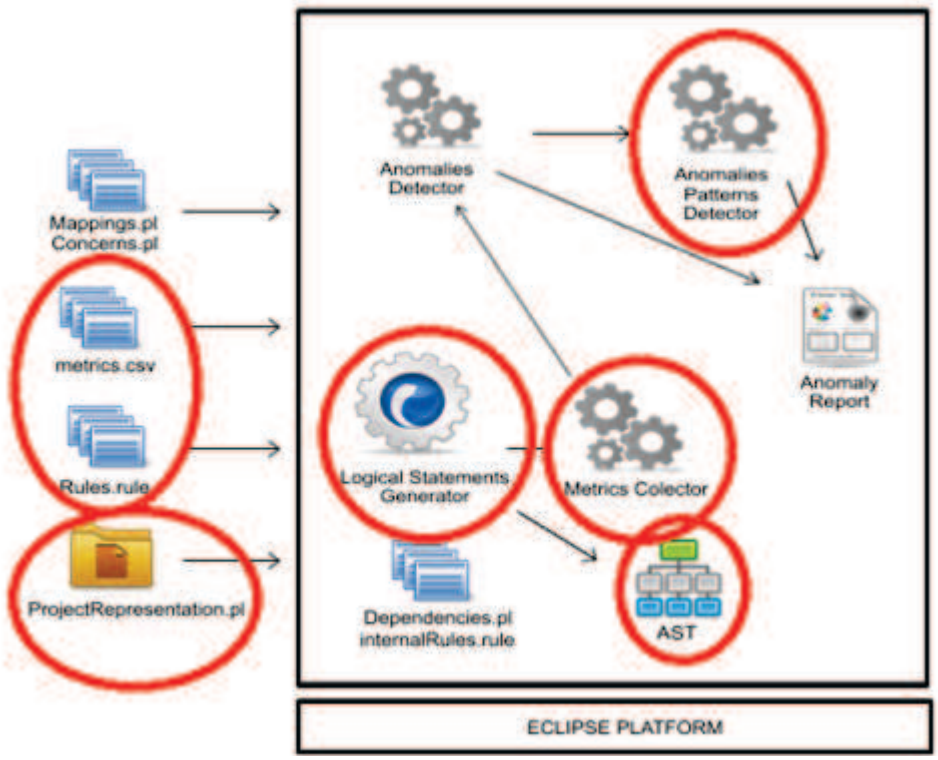

Figura 4.7: Arquitetura da solução proposta. Figura baseada na fonte: [Macia, 2013]

Os próximos tópicos irão descrever em detalhes as modificações realizadas na ferramenta para permitir oferecer suporte à nova abordagem. Essas modificações foram divididas em três categorias: (i) Atualização de métricas de acoplamento; (ii) Criação de novas métricas. (iii) Carregamento de estrutura do sistema multilinguagem em memória e adequação mecanismo de consulta.

Atualização de Métricas de Acoplamento

As principais métricas de acoplamento utilizadas nessa abordagem para componentes escritos em diferentes linguagens são a *FanIn* e *FanOut*. Essas métricas foram selecionadas, pois são as mais utilizadas por outros estudos para análise de acoplamento entre componentes [Macia et al., 2012b] [Marinescu et al., 2010] e podem ser aplicadas nas definições de diferentes anomalias tais como *Divergent Change* e *Feature Envy*. No entanto, não existem ferramentas de coleta de métricas que realizem a contabilização de FanIn e FanOut em componentes híbridos. Além disso, a utilização desses valores sem considerar esses componentes poderiam causar inconsistências na identificação das anomalias. Dessa forma, foi criado um mecanismo que realiza a atualização dos valores de FanIn e FanOut entre os arquivos. O padrão de dependência em

componente híbrido usado segue as definições ditas na Subseção 4.2.2.

O mecanismo funciona da seguinte forma. Primeiro foi realizada uma avaliação de todos os arquivos visando identificar dependências entre elementos híbridos. Por exemplo, a Figura 4.4 mostra a operação completeTodo e ela segue o padrão definido no modelo de dependência. Uma vez identificada uma referência como a completeTodo, os valores de FanIn e FanOut do elemento híbrido, a declaração que o contem, as operações dependentes e as declarações que os contem são atualizados. No exemplo da Figura 4.4, a operação completeTodo tem duas dependências, logo vai ter o seu valor de FanOut incrementado de 2. Assim como a declaração Todos e as operações completeTodo e forceGetCurrentTodoLists tiveram o valor de FanIn incrementado de 1. Essa avaliação foi feita tanto para o nível de declaração como para nível de operação.

Criação de Novas Métricas

Através dos resultados reportados no estudo anterior descrito na Seção 3.2.2 foi possível perceber a importância na análise de dependências entre componentes híbridos. Além disso, como reportado na Seção 2.1, tal análise exige conhecimentos diferentes acerca do que é uma anomalia de código para cada e quais as melhores soluções de software. Essa análise foi realizada na Seção 4.2.2. Com isso, como descrito na Seção 4.2.5, foi necessária a criação de novas métricas de contabilização de dependências de acoplamento para elementos híbridos. Foram criadas duas métricas que estão listadas abaixo que englobam as dependências definidas pelo modelo de dependência descrito na Seção 4.2.2 para novas linguagens que oferecem suporte a operação. Ou seja, o foco foi na linguagem JavaScript.

- **Number Of Used External Classes Per Function From JavaScript To Java (NOEC-F):** Essa métrica contabiliza o número de declarações Java que são usadas em uma operação JavaScript. Ou seja, o número de declarações em que operações são chamadas em uma determinada operação JavaScript. Essa métrica permite identificar se uma operação JavaScript possui um alto acoplamento em relação a elementos de código Java.
- **Number Of Used External Function Per Function From JavaScript To Java (NOEF-F):** Essa métrica contabiliza o número de operações Java que são usadas em uma operação JavaScript. Ou seja, o número de operações que são chamadas em uma determinada operação JavaScript. Essa métrica permite identificar se uma operação JavaScript possui um alto acoplamento em relação a elementos de código Java.

Carregamento de Estrutura do Sistema Multilinguagem em Memória e Adequação Mecanismo de Consulta

A estrutura genérica final para representação da árvore AST em memória pelo SCOOP não foi modificada. Isso ocorreu em razão do seu caráter genérico e transparente a qualquer linguagem. Por outro lado, para permitir construir essas árvores com nós de elementos de código relacionados a todas as linguagens suportadas, foi necessário realizar modificações na forma de estruturação em memória de árvores específicas e consultas para coleta de métricas.

Para permitir a avaliação de sistemas multilinguagem é importante que seja possível a análise de componentes híbridos. A ferramenta SCOOP só permitia o carregamento em memória de componentes escritos na linguagem Java. O SCOOP utiliza a biblioteca JDT [JDT, 2014] para a realização de consultas entre elementos escritos na linguagem Java. Dessa forma, o suporte ao carregamento de elementos de código JavaScript foi realizado através do uso da biblioteca JSDT [JSDT, 2014]. Essa biblioteca é uma versão para JavaScript da biblioteca JDT [JDT, 2014].

Não existe uma versão do JDT ou de outra biblioteca para carregamento em memória de elementos de código JSP. Consequentemente, o suporte à linguagem JSP foi realizada através do desenvolvimento de um *parser*. Esse parser se baseou na identificação de arquivos JSP e dependências definidas no modelo de dependências.

Após a realização do carregamento em memória dos componentes híbridos, foi possível realizar consultas em cada um deles. A realização de consultas utiliza como base diferentes níveis de encapsulamento em sistemas. A utilização de níveis como módulos, declarações e operações é importante na organização de declarações, operações e funcionalidades semelhantes, respectivamente. Ou seja, ao identificar uma anomalia de código é importante saber qual nível ela afeta. Esse conhecimento vai permitir saber qual o impacto da sua mudança no sistema e se existe a possibilidade de estar afetando outros elementos no mesmo nível.

São suportados diretamente pela linguagem representada pelos seguintes conceitos: módulos, declarações e operações. Cada um desses níveis já existe em Java. Portanto, podem ser identificados diretamente no código-fonte em Java. No entanto, em outras linguagens, esses níveis são suportados. Dessa forma, foi necessário fazer a equivalência de níveis para as linguagens JSP e JavaScript.

A equivalência de níveis consistiu em identificar correspondências entre conceitos em diferentes linguagens, mas que tinham o mesmo papel na estru-

tura dos sistemas. Ou seja, conceitos com nomes diferentes, mas que exerciam funções semelhantes. A linguagem Java não sofreu alterações na definição dos conceitos base, uma vez que já suporta todos os conceitos de maneira literal. Segue a seguir as equivalências feitas para as linguagens do ambiente alvo:

- **Módulo:** O conceito de módulos é comum a todas as linguagens. Também está englobado no conceito de módulos o conceito de pacotes.
- **Declaração:** Declaração representa os conceitos de arquivo para JavaScript e JSP e classe para Java.
- **Operação:**O conceito operação representa os conceitos de função para JavaScript e método para Java. JSP não possui o conceito de operação.

Existe uma exceção para a equivalência de função em código JavaScript. Quando uma função possui outras funções internamente e não está definida em uma função, ela é considerada uma declaração e não uma operação. Essa é uma definição inerente à linguagem JavaScript como definido em [Crockford, 2008].

Após a realização das equivalências, foi possível realizar consultas genéricas e demais avaliações na estrutura de módulos, declarações e operações disponíveis. Como por exemplo, ao avaliar a anomalia de código *God Class*, tanto classes Java como arquivos JSP são avaliados e reconhecidos no sistema da mesma.

Com a realização das consultas, foi possível identificar anomalias de código e suas dependências. No entanto, nos casos em que eram necessárias avaliações do código fonte e estrutura interna de declarações e operações, foram necessárias modificações. Essas modificações permitiram o suporte a consultas de todas as linguagens. Essas modificações foram feitas usando a biblioteca JSDT ou código do parser desenvolvido para componentes JSP.

4.3 Conclusão

A abordagem proposta tem como objetivo oferecer suporte a identificação de problemas arquiteturais através da detecção de anomalias de relevância arquitetural em um sistema multilinguagem. Para que isso fosse possível, alguns procedimentos foram realizados. Esses procedimentos englobaram, inicialmente, a configuração de um ambiente multilinguagem em que fosse possível identificar anomalias inter-relacionadas. Posteriormente, definição de métricas e estratégias para diagnóstico de anomalias inter-relacionadas. Essa definição permitiu detectar anomalias inter-relacionadas híbridas através do tratamento das dependências entre componentes híbridos. Por fim a seleção dos tipos de

anomalias a serem utilizados e a modificação e configuração de uma ferramenta que permitiu o suporte atual a abordagem.

O desenvolvimento da abordagem permitiu concluir que a definição das estratégias para identificação de problemas em componentes em sistemas multilinguagem deve, em especial, fazer o tratamento de componentes híbridos e suas dependências. Para que isso fosse possível foi necessário fazer a identificação de anomalias em elementos de código de acordo com o ambiente de configuração definido.

No próximo Capítulo é feita uma avaliação das características relacionadas a sistemas multilinguagem que permitem identificar problemas arquiteturais de forma eficaz. Essa avaliação foi realizada através da execução de um estudo realizado em três aplicações comerciais. O foco principal deste estudo foi avaliar a eficácia desta abordagem na identificação de anomalias de código de relevância arquitetural em sistemas multilinguagem.

5
Avaliação

O Capítulo anterior mostrou os procedimentos para o desenvolvimento de suporte a identificação de anomalias de código de relevância arquitetural em sistemas multilinguagem. Esse suporte tem como foco permitir o desenvolvedor detectar problemas arquiteturais de forma eficaz. Neste Capítulo, o objetivo é responder a RQ3 do presente trabalho: *Quais as características das anomalias de código podem indicar problemas arquiteturais em um sistema multilinguagem?*. Responder essa pergunta vai permitir avaliar "se" e "como" a análise de características relacionadas a sistemas multilinguagem permite identificar problemas arquiteturais de forma eficaz.

Este Capítulo apresentará a construção e execução de um estudo realizado em três aplicações comerciais. Nesse estudo, foram usadas as estratégias baseadas em métricas para a identificação de anomalias inter-relacionadas em sistemas multilinguagem (Seção 4.1). Os resultados permitiram identificar que, de fato, é fundamental o suporte explícito à identificação de problemas arquiteturais em software multilinguagem. Além disso, observamos quais características foram importantes para melhora da eficácia na detecção de tais problemas.

O método científico usado na execução do estudo foi o estudo de caso exploratório. O estudo foi executado nos sistemas, descritos na Subseção 5.2, em uma empresa de software de médio porte. Esse método foi escolhido como mais apropriado, pois foi executado um estudo usando estratégias baseadas em métricas em um sistema real com menor controle sobre os eventos comportamentais do que usando experimentos. Além disso, não definimos proposições ou hipóteses para ser falseada, permitindo levantar problemas, identificar variáveis relacionadas ao fenômeno e investigar possíveis causas e consequências dos eventos [Yin, 2009].

5.1
Desenho do Estudo

Alguns trabalhos fizeram análises sobre o impacto arquitetural de anomalias de código em sistemas monolinguagem (Seção 2.1) [Macia et al., 2012b]

[Macia, 2013] [Schumacher et al., 2010] [Marinescu, 2004] [Wedyan et al., 2009] [Moha et al., 2010]. No entanto, pouco se sabe sobre esse fenômeno em sistemas multilinguagem. Visando entender melhor as características das anomalias de relevância arquitetural em sistemas multilinguagem e auxiliar na resposta da RQ3, foram definidas duas perguntas de pesquisa secundárias. As perguntas de pesquisa secundárias estão listadas abaixo:

RQ3.1 Quais características das anomalias de relevância arquitetural são semelhantes ou diferentes em sistemas multilinguagem comparados a sistemas monolinguagem?

RQ3.2 O tratamento das características diferentes em sistemas multilinguagem na abordagem proposta influenciou na eficácia observada?

Essas perguntas secundárias irão permitir efetuar: (i) uma melhor análise das características das anomalias de relevância arquitetural identificadas e (ii) a relação destas características com os problemas arquiteturais. A primeira questão de pesquisa secundária (RQ3.1) tem como objetivo avaliar quais tipos de semelhanças e diferenças nas características das anomalias de relevância arquitetural foram identificados durante o estudo. As semelhanças e diferenças serão identificadas através de um comparativo entre o que foi identificado no estudo anterior (Capítulo 3) e os resultados do presente estudo em sistemas multilinguagem. Responder essa pergunta vai permitir a obtenção de oportunidades de melhorias nas abordagens de detecção e confirmar ou refutar a eficácia dos procedimentos propostos no Capítulo 4. Por exemplo, trabalhos anteriores mostram que a variedade sintática e semântica entre linguagens pode ser uma característica que influencia no diagnóstico de anomalias de relevância arquitetural em componentes híbridos [Kontogiannis et al., 2006] [Linos et al., 2003].

A outra questão de pesquisa secundária (RQ3.2) visa avaliar se as diferenças nas características apresentadas entre sistemas multilinguagem em relação a sistemas monolinguagem influenciaram nos resultados da eficácia do estudo. Por exemplo, primeiramente visa identificar se a variedade sintática e semântica é uma característica existentes nos problemas arquiteturais diagnosticados em sistemas multilinguagem. Então será feita uma avaliação da influência da existência dessa característica nos resultados de eficácia apresentados. Essa avaliação vai permitir identificar oportunidades de reflexões e aperfeiçoamento da abordagem proposta no Capítulo 4.

Baseado no *template* de definição de objetivos do Wohlin et al. [Wohlin et al., 2012], o objetivo do estudo foi:

Analisar: Estratégias baseadas em métricas com suporte a sistemas multilinguagem.

Com o propósito de: Quantificar sua eficácia

Com respeito a: Identificação de anomalias inter-relacionadas de relevância arquitetural.

Do ponto de vista do(s): arquitetos, desenvolvedores e pesquisador.

No contexto de: três (03) sistemas de software de múltiplos domínios, desenvolvidos usando o subconjunto de linguagens definido na Subseção 4.2.1.

É importante ressaltar que não faz parte do escopo do estudo as anomalias inter-relacionadas que tenham como base informações de interesses arquiteturais e hierarquia de classes. Essa decisão foi descrita e justificada na Seção 4.2.2. Além disso, arquitetos e desenvolvedores das aplicações alvo participaram da etapa de validação das indicações (Seção 5.3.3). Eles serão chamados a partir de agora de avaliadores.

Nesse trabalho foi avaliado um total de 255 anomalias inter-relacionadas. Esse conjunto de 255 anomalias representa 66% do total de indicações de anomalias inter-relacionadas realizadas pela ferramenta SCOOP (Subseção 2.6.2) para todas as três aplicações comerciais (Subseção 5.2). Este percentual de 66% das anomalias avaliadas representa o subconjunto de indicações mais relevantes arquiteturalmente de acordo com os avaliadores e pesquisador. Portanto, o restante equivalente a 34% foi descartado.

5.2 Aplicações Alvo

Após a definição do objetivo, o próximo passo foi a seleção das aplicações alvo do estudo. Um conjunto de critérios foi utilizado na seleção das aplicações alvo. Esses critérios estão listados na Tabela 5.1.

Tabela 5.1: Critérios usados na seleção das aplicações alvo

C1	A existência de documentação ou avaliadores disponíveis para avaliação dos resultados
C2	Presença de uma diversa variedade de tipos de anomalias de código
C3	Presença de uma diversa variedade de problemas arquiteturais
C4	Implementação em um subconjunto de linguagens definidas na Subseção 4.2.1
C5	Desenvolvimento feito por desenvolvedores com diferentes níveis de conhecimento

A seleção de cada critério tinha como objetivo a obtenção de resultados mais confiáveis, independentes de particularidades específicas de cada projeto. O critério C1 permitiu reduzir a geração de resultados finais com ruídos. Para isso, a avaliação foi auxiliada pelas documentações de usuário e arquitetural atualizadas, de acordo com os avaliadores, ou a participação de avaliadores

durante avaliação. A C2, C3 e C5 permitiram ter disponível uma variedade de tipos de anomalias, problemas arquiteturais e experiências de desenvolvedores, de forma a avaliar a abordagem de detecção em sistemas multilinguagem em diversas situações. A C4 permitiu a avaliação de componentes desenvolvidos usando as linguagens suportadas pela abordagem para sistema multilinguagem.

As aplicações alvo, listadas abaixo, foram selecionadas uma vez que elas satisfaziam todos os critérios. Para cada uma delas foi selecionada uma versão disponível no sistema de controle de versão da aplicação. A seleção dessa versão foi baseada na necessidade de mudanças arquiteturais de acordo com as documentações e avaliadores envolvidos no estudo. Ou seja, nessas versões possivelmente seriam encontrados os problemas arquiteturais mais relevantes ao sistema. Segundo seus arquitetos, estes problemas arquiteturais deveriam ser detectados no código fonte de forma que o mesmo sofresse as refatorações apropriadas. Com exceção do Sistema "Tudu-Lists", os sistemas foram avaliados na própria empresa que os desenvolveu.

Sistema Tudu-Lists: A aplicação alvo "Tudu-Lists" é um sistema de código aberto web para gerenciamento de listas de tarefas. O sistema foi implementado usando a arquitetura *Java Enterprise Edition* JEE. A camada servidor utilizou código Java e *Java Server Pages* - JSP e a camada cliente foi implementado em JavaScript (Js). O sistema envolveu durante todo o desenvolvimento um total de 4 desenvolvedores. Neste sistema, foram avaliadas 100% do total de anomalias inter-relacionadas diagnosticadas pela abordagem de identificação e relacionadas ao sistema "Tudu-Lists".

Sistema V: A aplicação alvo "V" é a mesma aplicação web em que foi executado o primeiro estudo de caso reportado na Subseção 3.1.2. Neste sistema, foram avaliadas 100% do total de anomalias inter-relacionadas diagnosticadas pela abordagem de identificação e relacionadas ao sistema V foram avaliadas.

Sistema R: A aplicação alvo "R" é uma aplicação web que permite a visualização de informações complexas através de abstrações visuais tais como gráficos em diversos níveis. O sistema foi implementado usando a arquitetura *Java Enterprise Edition* JEE. A camada servidor utilizou código Java e *Java Server Pages* - JSP e a camada cliente foi implementado em JavaScript (Js). O sistema envolveu durante todo o desenvolvimento um total de 6 desenvolvedores. Neste sistema, foram avaliadas 54% do total de anomalias inter-relacionadas diagnosticadas pela abordagem de identificação e relacionadas ao sistema R foram avaliadas.

5.3 Coleta de Dados

Após a seleção das aplicações alvo, foi executada uma sequência de etapas. Essas etapas estão listadas a seguir e podem ser observadas na Figura 5.1. Cada uma das etapas permitiu coletar informações que auxiliaram na análise das características das anomalias inter-relacionadas diagnosticadas. É importante ressaltar que essa sequência de etapas foi realizada para cada aplicação alvo:

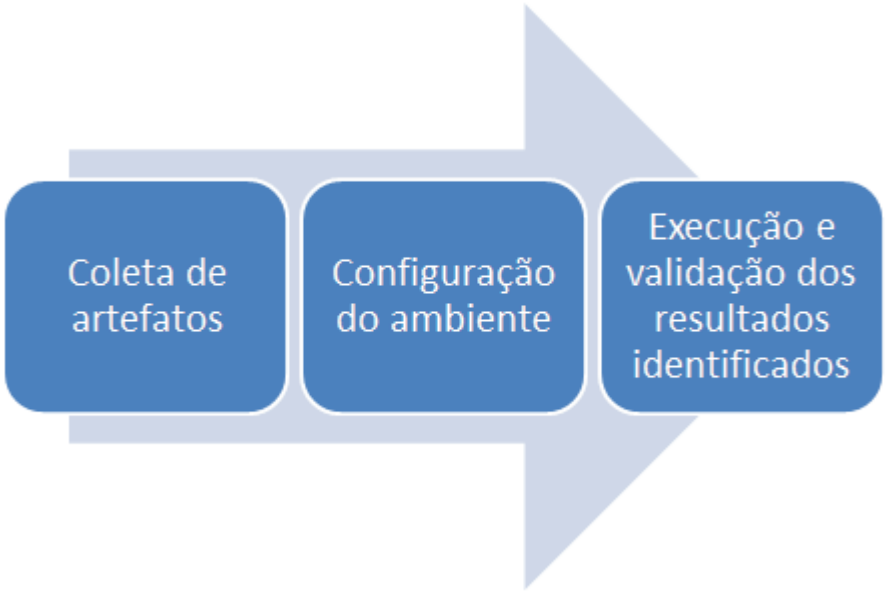

Figura 5.1: Modelo do processo de avaliação

5.3.1 Coleta de Artefatos

Foram coletados durante essa etapa: documentações de sistema da aplicação alvo como, por exemplo, representação arquitetural, manual de usuário, relatórios disponíveis em sistemas de rastreamento de erros[1], código-fonte e outras informações juntamente com os avaliadores. Essas informações serão utilizadas para identificar os problemas arquiteturais relacionados às anomalias de código diagnosticadas e suas características. Além disso, detalhes técnicos para avaliação das dependências em componentes híbridos, como, por exemplo, quais *frameworks* são usados. Alguns dos documentos utilizados foram reusados do trabalho de Silva [Silva, 2013] que também tinha foco em detecção de anomalias de código de relevância arquitetural. Consequentemente, muitas

[1]Do inglês: "bug tracking".

dessas informações relevantes para o presente trabalho já foram validadas com avaliadores no trabalho realizado anteriormente.

5.3.2 Configuração do Ambiente

Arquivos Auxiliares. Foram gerados arquivos auxiliares requeridos pela ferramenta usada (Subseção 2.6.2), visando aplicar as estratégias baseadas em métricas. O formato destes arquivos auxiliares é essencialmente o mesmo que seria gerado caso o sistema fosse monolinguagem. No entanto, esses arquivos foram incrementados com novas informações que permitiram a análise de características específicas de sistemas multilinguagem e, consequentemente, de anomalias inter-relacionadas nesses sistemas. Por exemplo, o arquivo de métricas teve a inserção de novas métricas que contabilizam informações estruturais das novas linguagens, como por exemplo, número de chamadas em cadeia de um método JavaScript.

Para a geração dos arquivos auxiliares, inicialmente, foi gerado um arquivo de mapeamento entre elementos de código (classes) e elementos arquiteturais para a aplicação alvo sendo avaliada. Elementos arquiteturais são componentes arquiteturais definidos na representação arquitetural do sistema ou definidos pelos avaliadores. Nesse mapeamento, existiam elementos de código desenvolvidos em diferentes linguagens: Java, JavaScript e JSP.

Geração de Métricas. Posteriormente, foi realizada a geração dos arquivos que continham as métricas dos elementos de código da aplicação alvo sendo avaliada. A seleção de métricas se baseou nas particularidades de cada linguagem. Portanto, um subconjunto de métricas só foi coletado em elementos de código desenvolvidos utilizando uma linguagem em específico. Essa estratégia permitiu capturar determinadas características inerentes a determinado grupo de anomalias inter-relacionadas em sistemas multilinguagem como pode ser visto na Subseção 5.4.1. No entanto, existem métricas genéricas que puderam ser coletadas em qualquer linguagem, tais como LOC e FanIn, definidas mais a frente. Além disso, dois tipos de métricas foram consideradas importantes durante a seleção: acoplamento e tamanho. Métricas de acoplamento são importantes em razão da necessidade de avaliação das dependências em componentes híbridos. Já as métricas de tamanho capturam a estrutura interna dos elementos de código, permitindo analisar propriedades importantes de vários tipos de anomalias de código.

Durante a geração dos arquivos foi necessária a seleção de ferramentas que permitiriam a coleta automática de métricas. A ferramenta Understand [Understand, 2013] foi selecionada para fazer coleta de métricas da linguagem

Java. Ela foi selecionada dentre as disponíveis em razão da variedade de tipos de métricas que são coletadas, tais como métricas de acoplamento e tamanho. Além disso, essa ferramenta permite a coleta de métricas em diversos elementos de código, como por exemplo, classes e operações. Permite também a coleta de métricas em outras linguagens, em especial, a JavaScript. Dessa forma, essa ferramenta também foi utilizada para coletar métricas na linguagem JavaScript.

Por outro lado, Understand só faz a geração de métricas estáticas e, devido ao aspecto dinâmico da linguagem JavaScript (JS), a utilização de análises dinâmicas se torna necessária. As análises dinâmicas são usadas, por exemplo, no monitoramento da criação e atualização de funções, objetos e propriedades em tempo de execução. Esta abordagem permitiu a coleta de métricas que permitiram capturar características intrínsecas a linguagens interpretadas, em especial, a JavaScript. Essas métricas foram reportadas na Tabela 5.4. Dessa forma, foi utilizada também a ferramenta JSNose [Fard and Mesbah, 2013] que permite fazer a coleta de métricas dinâmicas. A JSNose também gera métricas estáticas, mas o Understand possui uma diversidade maior de opções, possibilitando a análise de propriedades tal como de tamanho.

A coleta das métricas para código JSP foi realizada de maneira manual, uma vez que não foram encontradas ferramentas que ofereçam suporte a essa linguagem. Além disso, como reportado na Subseção 4.2.2, foram criadas novas métricas para cálculo de acoplamento entre elementos de código de diferentes linguagens.

Métricas utilizadas. As tabelas 5.2, 5.3 e 5.4 apresentam a descrição do conjunto de métricas utilizadas nesse estudo para cada uma das linguagens. O símbolo "-" indica que a métrica correspondente não foi coletada para a linguagem na coluna em questão. A Tabela 5.2 mostra a lista de métricas genéricas coletadas nesse estudo. Ou seja, as métricas que foram coletadas para todas as linguagens que fazem parte do ambiente configurado. A métrica Nº Linhas de código (LOC) [Understand, 2013] contabiliza o número de linhas de código sem linhas em branco e comentários. Ela foi coletada para os níveis de classe e operação. Já a métrica Nº chamadas realizadas por método (FanOut) [Understand, 2013] e Nº chamadas realizadas a método (FanIn) [Understand, 2013] foram coletadas para o elemento de código método. A eficácia de todas essas métricas já foi avaliada em estudos anteriores [Macia, 2013] [Silva, 2013] e se mostraram eficazes para auxiliar identificação de anomalias de código.

Tabela 5.2: Métricas Genéricas por Linguagem

Métrica	**Java**	**JavaScript**	**JSP**
Nº Linhas de código (LOC)	x	x	x
Nº chamadas realizadas por método (FanOut)	x	x	x
Nº chamadas realizadas a método (FanIn)	x	x	x

Por outro lado, algumas das métricas foram contabilizadas para somente duas linguagens: Java e JavaScript. A ferramenta de coleta de métricas Understand oferece suporte à coleta dessas métricas para ambas as linguagens. No entanto, essas métricas não foram coletadas para a linguagem Jsp em razão da falta de um suporte a sua coleta ou de estudos de como deve ser o processo de coleta dessas métricas na linguagem Jsp. A não contabilização dessas métricas para a linguagem JSP não interferiu nos resultados em razão da captura indireta e alternativa delas através das métricas dinâmicas do JavaScript e métricas Java. A Tabela 5.3 mostra o conjunto de métricas que foram coletadas para as linguagens Java e JavaScript.

A métrica Métodos ponderados por classe (WMC) [Understand, 2013] é a soma da complexidade ciclomática de todos os métodos. A avaliação do WMC pode indicar a restrição na sua reutilização uma vez que tendem a ser específicas, ou seja, foram desenvolvidas para casos específicos. Essa métrica é coletada para classes. A métrica Complexidade Ciclomática (CC) [Understand, 2013] é a contabilização de pontos de decisão tais como palavras-chave *for* e *while*. Essa métrica é coletada para métodos. Já a métrica Nível máximo de alinhamento de estruturas de controle (MaxNesting) [Understand, 2013] é o máximo nível de alinhamento de construções de controle tais como *for* e *switch* em um método. Essa métrica é coletada para classes e métodos. A métrica Número de métodos (NOM) [Understand, 2013] contabiliza o número de métodos de uma classe e é coletada para classes. A métrica Número de parâmetros (PAR) [Understand, 2013] contabiliza o número de parâmetros em um método e é coletada para métodos. Todas essas métricas já foram avaliadas em estudos anteriores [Macia, 2013] [Silva, 2013] e também se mostraram eficazes para auxiliar identificação de anomalias de código.

Tabela 5.3: Métricas específicas Java e JavaScript

Métrica	Java	JavaScript
Métodos ponderados por classe (WMC)	x	x
Complexidade Ciclomática (CC)	x	x
Nível máximo de alinhamento de estruturas de controle (MaxNesting)	x	x
Número de métodos (NOM)	x	x
Número de parâmetros (PAR)	x	x

Como dito anteriormente, algumas métricas foram coletadas somente para uma linguagem em específico. Na Tabela 5.4 são mostradas as métricas coletadas somente para a linguagem Java ou para a linguagem JavaScript. A métrica gerada só para a linguagem Java é a Acoplamento entre objetos (CBO) [Understand, 2013] que contabiliza o número de outras classes que estão acopladas a uma determinada classe. Uma classe A é acoplada a uma classe B se a classe A usa algum membro de uma classe B, como por exemplo, um método.

As outras métricas foram geradas para elementos de código JavaScript. A métrica Número de propriedades (NOP) [Fard and Mesbah, 2013] contabiliza o número de propriedades de um elemento de código que podem ser atributos de uma classe, por exemplo. A métrica Tamanho das chamadas em cadeia (LMC) [Fard and Mesbah, 2013] é o número de itens encadeados por pontos visando chamar um método específico. Essas longas chamadas podem resultar em um controle de fluxo complexo que é difícil de entender. Um método pode ter alinhamento de escopos, pois métodos podem acessar o escopo de métodos contendo-os. A métrica Número da cadeia do Escopo (LSC) [Fard and Mesbah, 2013] contabiliza o número de encadeamentos de escopos no método sendo avaliado. A métrica Número de casos (NOC) [Fard and Mesbah, 2013] contabiliza o número de casos do *switch*. Todas essas métricas já foram avaliadas em estudos anteriores [Macia, 2013] [Silva, 2013].

Já a métrica Número de Funções Externas Usadas Função de Js para Java (NOEF-F) contabiliza o número de métodos Java usados em um arquivo JavaScript sendo avaliado. A métrica Número de Classes Externas Usadas de Função de Js para Java (NOEC-F) contabiliza o número de classes Java diferentes que são usadas em um arquivo Js sendo avaliado. Nesse caso, as métricas NOEF-F e NOEC-F não foram avaliadas nos estudos anteriores e foram criadas para o presente trabalho como descrito na Subseção 4.2.5.

Tabela 5.4: Métricas específicas por Linguagem

Métrica	Java	JavaScript	JSP
Acoplamento entre objetos (CBO)	x	-	-
Número de propriedades (NOP)	-	x	-
Tamanho das chamadas em cadeia (LMC)	-	x	-
Número da cadeia do Escopo (LSC)	-	x	-
Número de casos (NOC)	-	x	-
Número de Funções Externas Usadas de Função para Função de Js para Java (NOEF-F)	-	x	-
Número de Classes Externas Usadas de Função de Js para Java (NOEC-F)	-	x	-

5.3.3 Execução e Avaliação dos Resultados

Após o ambiente estar preparado, foi realizada a execução da ferramenta SCOOP (Subseção 2.6.2) e geração de indicações suspeitas de anomalias inter-relacionadas. Posteriormente, essas indicações foram validadas visando identificar quais desses resultados representam problemas arquiteturais. A validação foi realizada através da confirmação junto com os desenvolvedores e arquitetos avaliadores. A confirmação ocorria quando as anomalias inter-relacionadas eram anomalias de código e estavam relacionadas a problemas arquiteturais. O processo de validação se baseou na análise da lista das indicações por avaliadores e pesquisador. Nas aplicações alvo R e V, os avaliadores estavam envolvidos no desenvolvimento e manutenção das aplicações e não houve a participação do pesquisador.

No caso da aplicação "Tudu-List", devido à impossibilidade de análise pelos atuais avaliadores, foi realizada uma análise por diferentes desenvolvedores. Esses desenvolvedores correspondem ao pesquisador do presente trabalho e um desenvolvedor externo. Ambos possuem pelo menos 5 anos de experiência no desenvolvimento de aplicações para a internet, em especial, utilizando o subconjunto de linguagens avaliadas no presente trabalho. Além disso, possuem experiência na modelagem arquitetural de sistemas. Ambos os desenvolvedores também utilizaram como base das suas avaliações, consultas a documentações eletrônicas disponíveis da aplicação, tais como lista de relatórios de *bugs* e refatorações.

Confirmação das anomalias. A confirmação das anomalias inter-relacionadas pelos avaliadores e pesquisador foi efetuada através de respostas sim e não. Em caso afirmativo, o avaliador ou pesquisador realizava anotações

indicando qual o problema arquitetural associado a aquela anomalia e outros comentários que desejasse sobre a anomalia. A resposta "sim" indicava que era uma anomalia inter-relacionada para aquela aplicação. No caso da reposta "não", não era considerada uma anomalia inter-relacionada. Como as avaliações levaram em consideração a opinião de diferentes avaliadores ou pesquisador, em alguns casos ocorriam discordâncias nas respostas. Quando ocorriam essas discordâncias e os avaliadores e o pesquisador estavam disponíveis, ocorria uma discussão entre os avaliadores e pesquisador até um consenso. No caso de não haver a disponibilidade de todos os avaliadores para discussão em relação aos resultados, o avaliador disponível avaliava as anotações e indicações de problemas arquiteturais realizadas por outros desenvolvedores e definia qual o resultado final.

É importante ressaltar que, uma vez confirmado que um elemento de código estava afetado por uma anomalia inter-relacionada, qualquer outra indicação realizada a esse elemento de código para outras anomalias inter-relacionadas, a indicação era confirmada. Ou seja, suponha que o elemento de código A foi indicado como afetado pela anomalia inter-relacionada I. Se o avaliador confirmar essa indicação, temos a certeza que A foi afetada por um problema arquitetural. Consequentemente, qualquer outra indicação de anomalia inter-relacionada a esse elemento de código estará associada a um problema arquitetural.

Geração de listas com referências. Após os resultados serem validados de acordo com os procedimentos acima, foram geradas listas com referências a indicações Falso Positivas (FP) e Verdadeiro Positivas (TP). Para as referências a TP, foram indicados quais os problemas arquiteturais correspondentes e possíveis regras arquiteturais quebradas. O indicador utilizado para avaliação da eficácia foi a precisão. Essa métrica foi utilizada no estudo anterior e definida na Subseção 3.1.1.

Indicações relacionadas a Falsos Negativos e Verdadeiros Negativos não foram avaliados, pois nenhum dos sistemas tinha disponível uma listagem referência de todos os problemas arquiteturais relacionados ao sistema. Essa lista de problemas arquiteturais serviria como universo total de avaliação e as indicações da abordagem utilizada poderia ser comparada com essa listagem.

Categorização dos resultados. É importante ressaltar que os resultados foram categorizados de acordo com as características das linguagens em que os elementos associados tinham dependências. Se um elemento de código tinha uma dependência para outro elemento de código que foi escrito com a mesma linguagem, ele era denominado elemento monolinguagem. Por outro lado, se fosse com uma linguagem diferente, era denominado elemento híbrido

(Seção 2.5). Na Figura 5.2 é mostrado um exemplo. Suponha que um elemento de código A e o B são escritos na linguagem Java e o elemento de código C na linguagem JavaScript. O elemento A têm dependências com os elementos B e C. Se o elemento A ou C forem afetados por anomalias inter-relacionadas (Seção 2.4), dizemos que as anomalias estão relacionadas a elementos híbridos. Se o elemento B for indicado, dizemos que as anomalias estão relacionadas a um elemento monolinguagem.

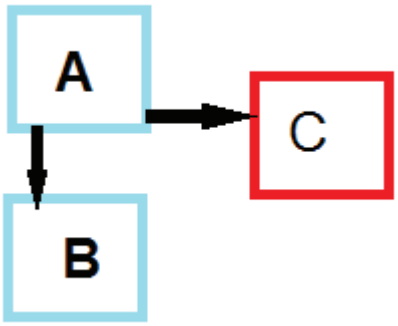

Figura 5.2: Definição de elemento monolinguagem e híbrido

Após a categorização, foram identificadas qual a frequência de elementos de código que tinham mais de uma indicações de anomalias inter-relacionadas relacionados a problemas arquiteturais. Ou seja, para cada elemento de código afetado por um problema arquitetural, foi verificado se o elemento era reincidente, sendo indicado por mais de uma anomalia inter-relacionada. A avaliação da frequência de elementos parte do pressuposto de que elementos de código que são afetados por mais de uma anomalia possuem maiores indícios de serem problemas críticos ao sistema e, consequentemente, relevantes de serem avaliados.

Após a identificação da frequência de elementos, houve uma discussão acerca dos resultados gerados pelos avaliadores e pesquisador. Essa discussão teve um caráter reflexivo e importante na indicação de melhorias na abordagem sugerida no Capítulo 4. É importante ressaltar que a discussão dos resultados não tinha como objetivo modificar os resultados já reportados anteriormente. O objetivo era de compreender de que forma os resultados poderiam ser melhorados. Além disso, essa discussão foi usada para um melhor entendimento das decisões tomadas no processo de validação dos elementos de código e no seu impacto arquitetural.

5.4 Resultados e Discussão

Essa Seção mostra os resultados do estudo de caso exploratório executado em três diferentes aplicações. Além disso, é realizada uma discussão acerca dos resultados apresentados.

5.4.1 Resultados

Visando auxiliar na avaliação da eficácia da identificação das anomalias inter-relacionadas, a Tabela 5.5 mostra os resultados relacionados à precisão da avaliação das anomalias inter-relacionadas nas aplicações alvo. As anomalias inter-relacionadas foram definidos na Seção 4.2.2. A precisão das "estratégias baseadas em métricas" é mostrada na última coluna da tabela. Além disso, são mostrados os números de Verdadeiros Positivos (TP) e Falsos Positivos (FP) para cada uma das anomalias inter-relacionadas em relação às aplicações alvo avaliadas (Subseção 5.2). Por exemplo, a precisão para a anomalia inter-relacionada *Multiple Anomaly* para a aplicação alvo R é de 0.52.

Tabela 5.5: Precisão das indicações de anomalias inter-relacionadas

Anomalia de Código	**TP**			**FP**			**Precisão**		
Inter-relacionada	Tudu	R	V	Tudu	R	V	Tudu	R	V
Multiple Anomaly	16	12	41	12	11	38	0.57	0.52	0.52
Similar Anomalous Neighbors	3	6	6	3	3	0	0.50	0.67	1
External Attractor per Class	3	4	2	1	5	4	0.75	0.44	0.33
External Addictor per Method	3	0	12	3	5	8	0.50	0	0.60
External Addictor Per Class	2	2	8	5	3	10	0.29	0.40	0.44
Replicated External Network	0	1	1	2	0	0	0	1	1

As anomalias inter-relacionadas que obtiveram para todas as aplicações alvo resultados maior ou igual a 0.50 foram a *Multiple Anomaly* e *Similar Anomalous Neighbors*. Nesse caso, a anomalia inter-relacionada *Multiple Anomaly* continuou obtendo resultados satisfatórios em relação ao estudo anterior como pode ser visto na Seção 3.2.1. Além disso, existe uma indicação de que a *Simi-*

lar Anomalous Neighbors é um forte indicador de problema arquitetural, em especial, em sistemas multilinguagem.

As anomalias *External Addictor Per Method* e *Replicated External Network* foram bons indicadores de problemas arquiteturais na maioria dos sistemas. Onde elas não ocorreram, parece que os problemas arquiteturais correspondentes a essas anomalias não se manifestaram nesse sistema. Especificamente em relação à anomalia *Replicated External Network*, os resultados mostraram que, de maneira geral, ela não tem muitas indicações. No entanto, esses resultados foram muito satisfatórios nos sistemas R e V, através de indicações que eram muito críticas, de acordo com os analistas e arquitetos avaliadores. Por outro lado, algumas anomalias tiveram, de maneira geral, apenas uma porcentagem moderada de acertos. São os casos das anomalias *External Attractor per Class* e *External Addictor Per Class*.

Visando auxiliar em uma análise mais profunda dos dados mostrados da Tabela 5.5, foi realizada uma categorização das anomalias inter-relacionadas identificadas, conforme descrito nos procedimentos do estudo na Seção 5.3. Essa categorização foi realizada baseada nos critérios de elementos monolinguagem e elementos híbridos apresentados anteriormente na Seção 5.3. Isto permite constatar a proporção de cada critério no número de TP.

A Tabela 5.6 apresenta os resultados dessa categorização. A primeira coluna apresenta o número total de anomalias inter-relacionadas que foram confirmadas durante a avaliação (Nº Total TP - NTP). A segunda coluna apresenta o número dessas referências que estão relacionadas a elementos híbridos (Nº Elementos Híbridos). A terceira coluna mostra a porcentagem desse número em relação ao total de NTP (%).

Tabela 5.6: Proporção de Anomalias Inter-Relacionadas Híbridas

Aplicação Alvo	Nº Total TP - NTP	Nº Elementos Híbrido	%
Tudu	27	15	55.5
R	25	15	60.0
V	70	36	51.43

Na Tabela 5.6 temos que a aplicação alvo Tudu teve um número total de 27 anomalias inter-relacionadas. Desse total, 15 estão relacionadas a elementos híbridos. Dessa forma, temos que 55.5% das anomalias inter-relacionadas necessitam da avaliação de elementos de código escritos em diferentes linguagens. A porcentagem de anomalias inter-relacionadas que estão associadas a elementos híbridos apresentam resultados uniformes para as aplicações alvo

avaliadas. Esses resultados variam entre 50% e 60%, o que é uma média relevante em termos percentuais. Ou seja, existem indícios de que o diagnóstico de problemas arquiteturais em sistemas multilinguagem se beneficiou do uso de estratégias baseadas em métricas na detecção de anomalias inter-relacionadas. A não avaliação de forma adequada das dependências dos elementos escritos em diferentes linguagens inibe a análise desses elementos. Tal análise permite obter melhores resultados em pelo menos mais da metade das identificações relevantes realizadas.

Após a categorização, foram identificadas a frequências de elementos de código em relação a anomalias inter-relacionadas. A coleta da frequência foi realizada de acordo com os procedimentos definidos na Seção 5.3. A Tabela 5.7 mostra os resultados da avaliação da frequência de reincidência de anomalias. Na primeira coluna são mostradas as aplicações alvo. Na segunda coluna é indicado a frequência para elementos monolinguagem. A terceira coluna é a porcentagem dessa frequência em relação ao total de elementos reincidentes. Na quarta coluna é indicado o número da frequência para elementos híbridos.

Tabela 5.7: Relação entre Nº de Reincidências de Elementos Híbridos e Monolinguagem

Aplicação Alvo	Nº Reincidentes Monolinguagem	%	Nº Reincidentes Híbrido	%
Tudu	2	40	3	60
R	5	56	4	44
V	9	36	16	64

Os resultados da Tabela 5.7 mostram que existe uma alta porcentagem de elementos híbridos que são reincidentes. Por exemplo, nesta tabela é mostrado que o número de elementos híbridos reincidentes na aplicação V foi de 64%. É possível notar que, exceto no sistema R, a porcentagem para híbridos é bem mais alta que as demais, chegando a ser pelo menos 50% maior que a monolinguagem. Dessa forma, existem indícios de que a utilização de estratégias inerentes a linguagens específicas permitiu um melhor diagnóstico de anomalias inter-relacionadas entre componentes escritos em diferentes linguagens .

5.4.2 Discussão dos Resultados

Essa Subseção traz a discussão em relação aos resultados apresentados na Subseção anterior. A discussão dos resultados vai ser balizada nas perguntas de pesquisa secundárias definidas na Seção 5.1.

Quais características das anomalias de relevância arquitetural são semelhantes ou diferentes em sistemas multilinguagem comparados a sistemas monolinguagem?

Uma análise da Tabela 5.5 revela um alto número de indicações em relação à *Multiple Anomaly*. Nesse caso, 56.56% do total de TP para todas as aplicações são de *Multiple Anomaly*. Essa é uma semelhança encontrada em relação à análise de sistemas monolinguagem, como pôde ser visto na Seção 3.2.2. Ou seja, o número de acertos se manteve alto em relação a sistemas monolinguagem, mesmo considerando agora componentes escritos em diferentes linguagens. Pode-se concluir que o fato que várias anomalias ocorrerem em um elemento de código, escrito em qualquer linguagem, é um bom indicador de que o elemento é afetado por um problema arquitetural. Além disso, a precisão para essa anomalia é uniforme entre as diferentes aplicações avaliadas. Ou seja, para diferentes domínios de software, o valor da precisão se comporta da mesma forma.

Como visto na Seção 5.4.1, as anomalias que tiveram os melhores resultados relacionados à precisão foram a *Multiple Anomaly* e *Similar Anomalous Neighbors*, obtendo números iguais ou maiores que 0.50. A *Similar Anomalous Neighbors* obteve a precisão de 0.50 para o sistema Tudu de acordo com a Tabela 5.5. A avaliação dessa anomalia parece ser ainda mais interessante no contexto de sistemas multilinguagem, pois elementos híbridos que possuem dependências entre elementos escritos em diferentes linguagens de programação tendem a demonstrar problemas semelhantes. Como os sistemas multilinguagem normalmente seguem um padrão para esse tipo de dependência, quando o padrão não é adequado, anomalias semelhantes tendem a se espalhar pelo sistema.

Por exemplo, a Figura 5.3 mostra dependências entre componentes escritos na linguagem JavaScript (Js) e Java. Cada um deles possui um subconjunto de elementos de código que estão afetados por uma anomalia inter-relacionada *Similar Anomalous Neighbors*. O elemento de código **common.js** é responsável por executar funções comuns a diversos arquivos e pertence ao componente A. O **reportLoader.js** é responsável pelo carregamento de um relatório e pertence ao componente B. O elemento common.js pertence ao subconjunto anômalo do componente A e o reportLoader.js é um elemento do subconjunto do componente B. Cada um dos elementos possui uma dependência com o componente C através de uma chamada para um arquivo do tipo *Action*. O common.js tem uma dependência para o arquivo AAction.java e o reportLoader.js com o BAction.java.

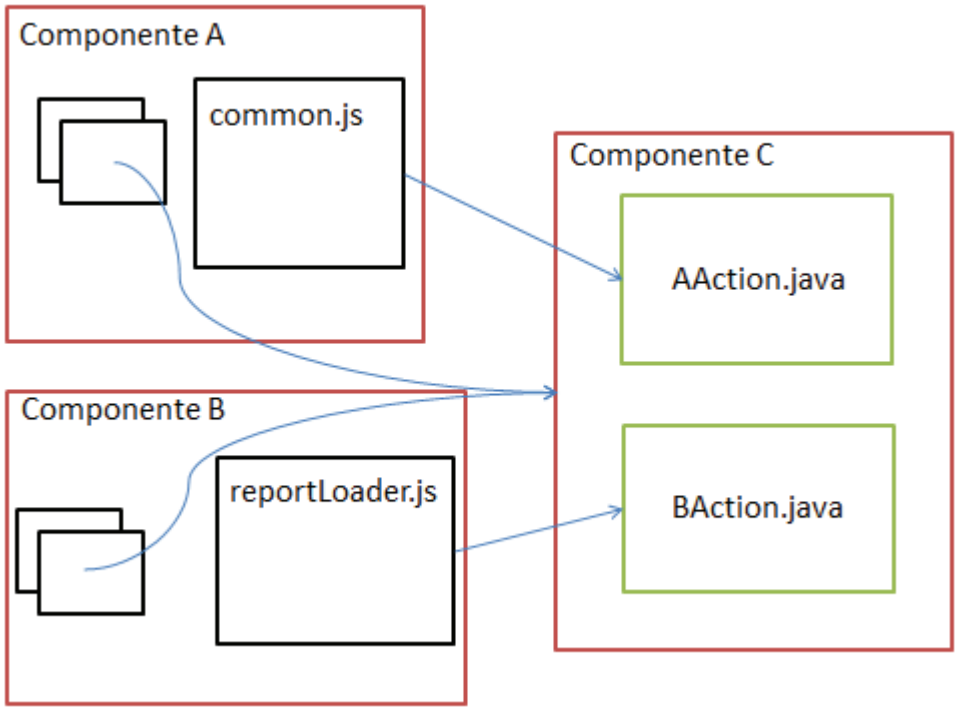

Figura 5.3: Exemplo de anomalia inter-relacionada *Similar Anomalous Neighbors* em arquivos JavaScript (Js)

O padrão de dependência entre arquivos JavaScript e arquivos Java do tipo Action é o mesmo. Portanto, foi realizada uma avaliação em relação aos problemas arquiteturais reportados para cada anomalia e foi percebido que o problema arquitetural era o mesmo. Em ambos os casos os elementos além de fazerem o tratamento das requisições para comunicação entre os componentes, eles eram responsáveis pelas funcionalidades de regra de negócio. Ou seja, o problema associado era o *Connector Envy*. Esse problema supõe que atividades de interação devem ser delegadas a um conector, visando diminuir o acoplamento entre os serviços de interação e funcionalidades específicas de negócio [Garcia et al., 2009]. Consequentemente, temos que a estrutura utilizada para dependência entre os arquivos tende a gerar problemas arquiteturais.

Como essas dependências ocorrem em várias áreas do sistema, os problemas foram espalhados por vários componentes. Além disso, é possível verificar que as características ligadas à variedade sintática e semântica entre linguagens pode ser um entrave na utilização adequada de soluções de software. Isso acontece, pois em casos como o citado, é complexa a avaliação das dependências entre os componentes e identificação de problemas arquiteturais. Dessa forma, é possível que a complexidade associada ao fato deles serem escritos em diferentes linguagens diminua com um suporte adequado ao sistema multilinguagem.

Esse fato é ainda mais evidente quando se analisa a Tabela 5.6. Nessa tabela é possível identificar que existe uma porcentagem expressiva de anomalias de código inter-relacionadas (60% para a Aplicação Alvo R) que envolvem elementos de código escritos em diferentes linguagens. Somado a isso, a Tabela 5.7 apresenta resultados relacionados à reincidência de anomalias de código inter-relacionadas e a porcentagem de reincidência em diferentes linguagens é ainda mais significativa. Ou seja, ao analisar essas anomalias de código é possível que sejam identificados problemas arquiteturais crônicos no sistema.

RQ3.2 O tratamento das características diferentes em sistemas multilinguagem na abordagem proposta influenciou na eficácia do estudo?

Apesar do bom desempenho da abordagem proposta, existe ainda uma grande porcentagem de FP para as anomalias inter-relacionadas identificadas. Ou seja, a abordagem detecta uma grande quantidade de anomalias de código que não correspondem a problemas arquiteturais. No entanto, é possível que esta correlação precise ser mais bem avaliada em sistemas multilinguagem. Apesar de existirem muitos trabalhos que reportam bons resultados de precisão na identificação de problemas arquiteturais em sistemas monolinguagem [Macia, 2013] [Garcia et al., 2009], existem poucos trabalhos que avaliam esses problemas em sistemas multilinguagem. Ou seja, devido à falta de informações, a correlação com problemas arquiteturais e anomalias de código pode não ser um procedimento simples para os analistas e arquitetos em sistemas multilinguagem.

Um exemplo dessa lacuna de informação pode ser observado através da Figura 5.4. Durante o momento de discussão dos resultados juntamente com os desenvolvedores e arquitetos avaliadores foi possível identificar alguns outros fatores que influenciaram no alto número de FP. Um dos fatores foi o desconhecimento sobre boas práticas arquiteturais para elementos de código não escritos em Java. Muitos dos desenvolvedores e arquitetos avaliadores responderam como negativo para algumas anomalias de código inter-relacionadas nesses elementos. No entanto, após a discussão a respeito do impacto arquitetural na aplicação, os avaliadores mudaram de ideia acerca da decisão anterior. Na prática, isso demonstra o potencial da abordagem proposta em revelar problemas arquiteturais não-triviais em software multilinguagem, contribuindo para o aprendizado dos desenvolvedores.

A Figura 5.4 mostra um conjunto de arquivos Jsp (**e_L2.jsp**, **e_L3.jsp** e **e_L4.jsp**). UtilitiesWeb foi indicada com um alto FanIn, ou seja, possui um alto acoplamento em relação as declarações JSP. Cada um arquivos JSP permite a geração de um tipo diferente de visualização de dados, de um nível

específico, indicado por um número. Por exemplo, o arquivo **e_L4.jsp** gera a visualização para o nível 4. No entanto, a única diferença entre elas é o número que é passado como parâmetro para a função que irá gerar a imagem na tela. Para cada novo nível criado, é necessário criar um arquivo novo. Ou seja, o crescimento do número de arquivos tende a ser exponencial. Nesse caso, uma boa solução arquitetural seria a criação de uma interface de fachada.

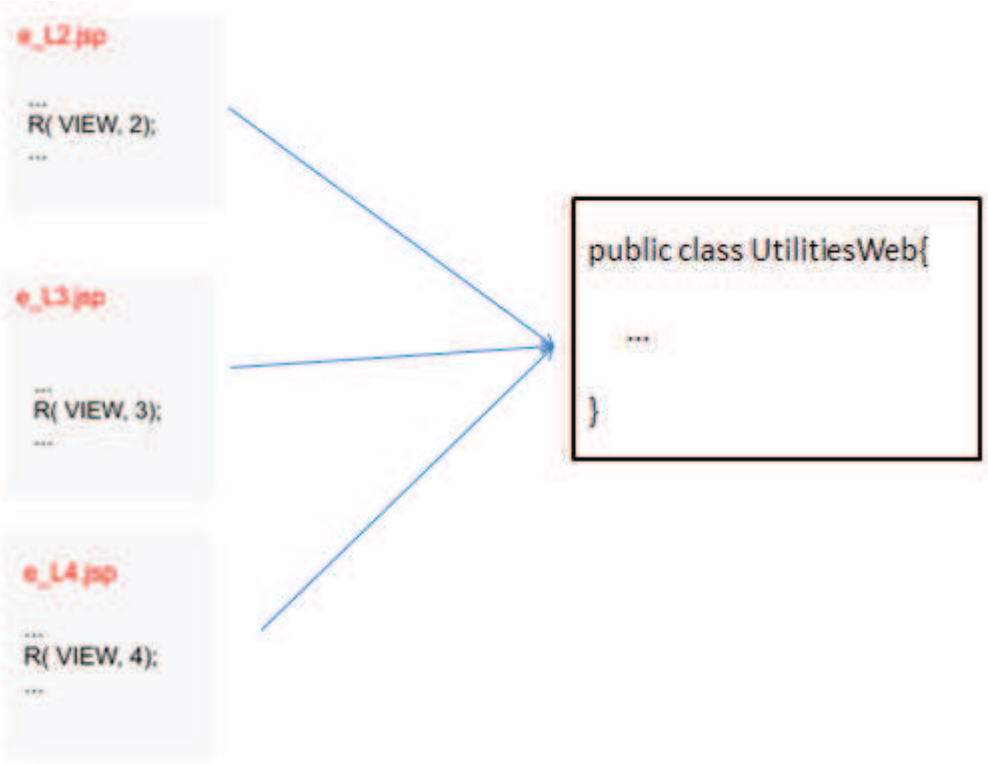

Figura 5.4: Exemplo de anomalia inter-relacionada em arquivos Jsp e Java

Outro ponto importante na avaliação das anomalias inter-relacionadas, foi o baixo índice de anomalias reportadas em arquivos escritos em linguagens diferentes de Java (Js ou Jsp). Através de uma avaliação superficial é possível afirmar que as estratégias para detecção de anomalias de código para arquivos diferentes de Java não estão maduras. Ou seja, são necessários estudos mais aprofundados em relação à frequência das anomalias de código e quais são mais adequadas para a identificação de anomalias de código de impacto arquitetural nos arquivos escritos com linguagens diferentes do Java.

É possível também que aspectos relacionados à natureza das linguagens tenham influenciado no processo de detecção. Ou seja, um estudo visando avaliar quais são as melhores estratégias de detecção para linguagens que não são orientadas a objetos ou não são definidas através dos seus conceitos de pacotes, classes e operações se faz necessário.

Portanto, através dos resultados mostrados, temos que existem características semelhantes na indicação de anomalias de código de relevância arquitetural, em especial, usando anomalias inter-relacionadas. Características essas, por exemplo, que permitem afirmar que determinadas anomalias, como a *Multiple Anomaly*, possuem uma boa média de acertos tanto para sistemas monolinguagem como para multilinguagem. Além disso, diversos resultados mostraram o impacto positivo na utilização de abordagens que permitem a identificação de anomalias inter-relacionadas em sistemas multilinguagem, permitindo identificar novos comportamentos como o apresentado no exemplo de *Similar Anomalous Neighbors*. Dessa forma, temos que tanto características presentes em sistemas monolinguagem, como novas características indicadas nesse estudo, podem indicar problemas arquiteturais em sistemas multilinguagem.

5.4.3 Ameaças à Validade

Essa Seção discute as ameaças à validade de acordo com a classificação proposta por Wohlin et al [Wohlin et al., 2012].

Validade na Construção. A primeira ameaça à validade está ligada as definições do que é uma anomalia de código para cada linguagem. A linguagem Java já possui definições amplamente reconhecidas como as reportadas por [Fowler, 1999]. No entanto, as linguagens JavaScript e Jsp não possuem definições que são de conhecimento geral. Visando minimizar isso, selecionamos um trabalho sobre anomalias de código para JavaScript que, embora recente, é muito promissor e baseado em definições de referências importantes da área como [Crockford, 2008] e [Richards et al., 2010]. Além disso, com relação ao JSP, foi realizada uma comparação com as definições indicadas por livros referência na área, que permitiram indicar um caminho para a classificação de anomalias de código.

Outra ameaça a validade é o escopo de avaliação de dependência entre linguagens diferentes. Foi definido um conjunto de linguagens a ser avaliada e um conjunto de padrões de dependência. Ou seja, linguagens que não pertençam a esse conjunto de linguagens ou não estejam adequadas a esse padrão de dependência entre elementos de código escritos em linguagens diferentes, não foram avaliadas. No entanto, visando maximizar o alcance de análise do ambiente de avaliação utilizado, fizemos uma seleção baseada em critérios como popularidade como visto na Seção 4.2.1.

Validade na conclusão. Nesse caso, a ameaça à validade é falta de um indicador revocação (*recall*), que é utilizado em alguns estudos para avaliar

estratégias de detecção de anomalias (por exemplo, [Wong et al., 2011] e [Macia et al., 2012b]). No entanto, esse indicador não pôde ser usado em razão da ausência de uma lista padrão de anomalias de código de relevância arquitetural para cada aplicação alvo. Portanto, não poderíamos calcular falsos negativos. Porém, esta é uma limitação típica quando se usa projetos de software grande e da indústria, como por exemplo nos trabalhos de Marinesco et al [Marinescu, 2004] e Olbrich et al [Olbrich et al., 2009].

Validade interna e externa. A principal limitação nesse ponto foi o nível de conhecimento e experiência dos desenvolvedores dos sistemas. Foram utilizados três sistemas que tiveram a participação de mais de 20 pessoas diferentes. Além disso, foram selecionadas aplicações de diferentes tamanhos e metodologia de desenvolvimento, visando tornar o ambiente de avaliação o mais diversificado possível.

5.5 Conclusão

O estudo de caso apresentado permitiu revelar que o uso de estratégias baseadas em métricas para identificação de anomalias inter-relacionadas é adequado ao ambiente multilinguagem. Portanto, este estudo forneceu indicadores de que a abordagem proposta é promissora. Foram realizadas as avaliações de três (3) aplicações alvo, com diferentes domínios e com um total de identificações de 255 anomalias de código inter-relacionadas.

A avaliação da precisão dessas anomalias mostrou que as anomalias *Multiple Anomaly* e *Similar Anomalous Neighbors* obtiveram os resultados com maior qualidade. Sendo a primeira delas, semelhante ao que foi identificado no trabalho anterior descrito na Seção 3.2. Por outro lado, a *Similar Anomalous Neighbors* mostrou-se adequada à identificação de anomalias em sistemas multilinguagem. Consequentemente, a avaliação do número de indicações inter-relacionadas de elementos de código desenvolvidos em diferentes linguagens e o número de anomalias reincidentes, permitiu perceber o quão é relevante à avaliação de anomalias inter-relacionadas em sistemas multilinguagem.

Por outro lado, foram identificados muitos FP nos resultados das anomalias inter-relacionadas. A análise dessas anomalias pertencentes ao conjunto de FP permitiu identificar lacunas no conhecimento dos analistas e arquitetos avaliadores. Além disso, até então no estado da arte, pouco se sabia a respeito das características dos problemas arquiteturais em sistemas multilinguagens. Essas lacunas podem ser supridas com o desenvolvimento de novos estudos em sistemas multilinguagem com o uso de abordagens que suportem a identificação de anomalias inter-relacionadas em diversas linguagens e na dependência

entre componentes escritos em diferentes linguagens.

Foi verificado na Seção 5.4.2 que a variedade sintática e semântica é uma característica existente nos problemas arquiteturais diagnosticados em sistemas multilinguagem. Com isso, foi possível identificar semelhanças e diferenças nas características das anomalias de código de relevância arquitetural em sistemas multilinguagem em relação a sistemas monolinguagem. Por exemplo, a frequência de algumas anomalias inter-relacionadas citadas neste trabalho são diferentes da frequência de algumas anomalias inter-relacionadas em sistemas monolinguagem citados no estudo do Capítulo 3. Isto permitiu verificar que a existência dessa variedade sintática e semântica influência nos resultados de eficácia apresentados na Seção 5.4.1. Além disso, identificamos que mais de 50% das anomalias diagnosticadas estavam relacionadas a componentes híbridos, tornando ainda mais consistente a observação de que essas características diferentes influenciaram nos resultados de eficácia. Essas diferenças se tornaram ainda mais evidentes quando identificou-se a falta de informação a respeito da relação entre problemas arquiteturais e anomalias de código em sistemas multilinguagem. Observou-se que não é uma tarefa simples para desenvolvedores e arquitetos avaliadores identificarem problemas arquiteturais em componentes híbridos. Ou seja, o estudo das diferentes características se torna um ponto de aprendizado para desenvolvedores e arquitetos e, inclusive, possibilita novos trabalhos no estudo da arte. Em especial, outros estudos sobre estratégias de detecção adequadas para sistemas multilinguagem se fazem necessários em razão da sua grande heterogeneidade.

No próximo Capítulo são apresentadas as conclusões finais, reunindo todas as perguntas respondidas por este trabalho. São explicitadas também as principais contribuições desta dissertação. Por fim, são apresentadas as limitações do trabalho e as sugestões de trabalhos futuros.

6
Conclusão

Sistemas multilinguagem são sistemas desenvolvidos usando pelo menos duas linguagens. Os aspectos de heterogeneidade relacionados ao uso de diferentes linguagens dificulta a concepção de soluções que apoiem os desenvolvedores na identificação de anomalias de código de relevância arquitetural. A identificação dessas anomalias vai permitir o desenvolvimento de sistemas com maior qualidade.

Diversas abordagens têm surgido com o objetivo de auxiliar os desenvolvedores na tarefa de detecção de problemas arquiteturais a partir do código fonte [Kontogiannis et al., 2006] [Mayer and Schroeder, 2012] [Alves et al., 2011]. Em especial, as estratégias baseadas em métricas têm sido exploradas nos últimos anos neste contexto. No entanto, poucos esforços existem na realização de estudos e na concepção de técnicas e ferramentas que apoiem a detecção de anomalias arquiteturalmente relevantes em sistemas multilinguagem.

Essa dissertação teve como objetivo responder 3 questões de pesquisa. A RQ1 é "Qual é a eficácia e o esforço associado ao aplicar estratégias atuais na identificação de anomalias de código de relevância arquitetural?". Através de um estudo de caso realizado em um sistema do mercado foi possível identificar quais eram os benefícios e dificuldades no uso de estratégias baseadas em métricas na identificação de anomalias de código. Além disso, foram identificadas oportunidades de melhorias no uso dessas estratégias. Em especial, algumas das oportunidades foi a identificação de anomalias de código de relevância arquitetural que requeriam estratégias de detecção considerando partes e estruturas do sistema implementadas em diferentes linguagens. Esse foi o ponto de partida para a avaliação da RQ2.

A RQ2 é "Como poderão ser definidas estratégias para identificação de problemas arquiteturais em componentes dependentes em sistemas multilinguagem?". Visando responder essa questão foi elaborada uma abordagem que tinha 5 fases: (i) Configuração do ambiente através da definição de um subconjunto de linguagens (Java, JavaScript e JSP); (ii) definição de métricas e estratégias que foram definidas através de estudos sobre anomalias de código

em cada uma das linguagens. Foram definidos também que tipos de métricas são mais adequados a cada contexto e quais as estratégias de detecção para anomalias de código em componentes escritos em cada linguagem; (iii) detecção de anomalias inter-relacionadas híbridas devido ao tratamento de dependências entre componentes híbridos, onde foi definido modelo de dependências entre componentes escritos em diferentes linguagens; (iv) seleção de tipos de anomalias inter-relacionadas; e (v) modificação e configuração de uma ferramenta. Nessa etapa o SCOOP foi adequado à nova abordagem sendo que as maiores modificações foram divididas em quatro categorias: (a) carregamento da estrutura de projetos a ser avaliado em memória; (b) modificação da estrutura de consultas; (c) desenvolvimento de Mecanismo de atualização de métricas de acoplamento para componentes escritos em diferentes linguagens e (d) criação de novas métricas de acoplamento para dependências entre componentes escritos em diferentes linguagens.

Essa abordagem foi avaliada através de um estudo de caso que permitiu responder a RQ3. A RQ3 é "Quais características das anomalias de relevância arquitetural são semelhantes ou diferentes em sistemas multilinguagem comparados a sistemas monolinguagem?". Para auxiliar a responder a RQ3 foram desenvolvidas perguntas de pesquisa auxiliares: A RQ3.1 era "Quais tipos de características das anomalias de código de relevância arquitetural são semelhantes e quais são diferentes num sistema multilinguagem em relação a um sistema com a avaliação de uma linguagem?". Resultados mostraram que houve semelhanças quando aos resultados de algumas anomalias inter-relacionadas. No entanto, mostrou uma porcentagem expressiva de anomalias que estavam relacionadas a elementos que implementavam comunicação entre componentes escritos em diferentes linguagens. Além disso, essas anomalias possuíam um alto número de reincidência. A RQ3.2 era "O tratamento das características diferentes em sistemas multilinguagem na abordagem proposta influenciou na eficácia do estudo?". A avaliação dos resultados mostrou uma alta porcentagem de reincidência de anomalias relacionadas a elementos que fazem comunicação entre componentes híbridos. Foi verificado também na Seção 5.4.2 que a variedade sintática e semântica é uma característica existentes nos problemas arquiteturais diagnosticados em sistemas multilinguagem. Com isso, foi possível identificar semelhanças e diferenças nas características das anomalias de código de relevância arquitetural através, por exemplo, da frequência de algumas anomalias inter-relacionadas. Isto permitiu verificar que a existência dessa variedade sintática e semântica influência nos resultados de eficácia (Seção 5.4.1).

6.1 Contribuições do Trabalho

A primeira contribuição foi um estudo de caso longitudinal que permitiu avaliar as estratégias baseadas em métricas em comparação com a estratégia *ad hoc*. Essa avaliação mostrou o passo-a-passo na utilização das estratégias baseadas em métricas e conclusões acerca da eficácia e esforço no uso dessa técnica. Se por um lado os resultados de eficácia foram similares, o que indica que estratégias podem substituir especialistas, por outro os resultados do esforço indicaram que é necessário o aperfeiçoamento das técnicas para geração de arquivos auxiliares. Em especial se tratando de sistemas multilinguagem, onde deve ser possível o mapeamento de elementos de código escritos em diferentes linguagens. Adicionalmente, um dos principais resultados do estudo foi a identificação de anomalias de código de relevância arquitetural em partes e estruturas do sistema implementadas em diferentes linguagens.

Outra contribuição foi uma abordagem que permite a identificação de anomalias de código inter-relacionada em sistemas multilinguagem. Ou seja, essa abordagem permite a identificação de anomalias de código inter-relacionadas arquiteturalmente relevantes em trechos de um sistema em que são utilizadas diferentes linguagens. A abordagem foi aplicada em sistemas envolvendo as 3 linguagens: Java, JavaScript e JSP. Porém, os procedimentos da abordagem foram definidos de forma a serem aplicados ou estendidos para outras linguagens.

Além disso, a avaliação da abordagem proposta proporcionou a identificação de diversas conclusões acerca das características das anomalias de código de relevância arquitetural em sistemas multilinguagem. Os resultados apontaram indícios de que o diagnóstico de problemas arquiteturais em sistemas multilinguagem se beneficiou do uso de estratégias baseadas em métricas na detecção de anomalias inter-relacionadas. A avaliação de forma adequada das dependências dos elementos escritos em diferentes linguagens permitiu obter melhores resultados em pelo menos mais da metade das identificações relevantes realizadas. A alta reincidência de anomalias de código inter-relacionadas híbridas mostra que é possível que sejam identificados problemas arquiteturais no sistema. A abordagem proposta se demonstrou capaz de revelar problemas arquiteturais não triviais em sistema multilinguagem, contribuindo para o aprendizado dos desenvolvedores. Essas conclusões apontaram diversas questões para prosseguimento da pesquisa.

6.2 Limitações e Trabalhos Futuros

Esse trabalho possui alguns pontos específicos que restringem a sua aplicação. As limitações relacionadas ao estudo de caso usando as estratégias baseadas em métricas já foi detalhado na Seção 3.2.3.

A abordagem proposta possui um ambiente de avaliação previamente definido e configurado. Essa limitação ocorreu em decorrência da grande quantidade de linguagens disponíveis e o fato da abordagem se basear na avaliação das particularidades relacionadas a cada uma das linguagens. Essas particularidades estão relacionadas a que tipos de anomalias de código e métricas são mais adequadas para cada linguagem. As limitações relacionadas ao estudo de caso exploratório para a abordagem proposta foi detalhado na Seção 5.4.3.

Trabalhos Futuros

Durante o decorrer da pesquisa para este trabalho de dissertação foram identificados trabalhos futuros:

- Como visto no presente trabalho, o número de sistemas multilinguagem têm crescido e com ele a necessidade de sua análise e melhora na qualidade do código [Kontogiannis et al., 2006] [Mayer and Schroeder, 2012] [Alves et al., 2011]. No entanto, o número de trabalhos relacionados a sistema monolinguagem é maior que para sistemas multilinguagem, tornando a área de estudos em sistemas multilinguagem carentes de trabalhos [Kontogiannis et al., 2006] [Mayer and Schroeder, 2012] [Alves et al., 2011]. Em especial, estudos relacionados a outras atividades relacionadas a manutenção de sistemas multilinguagem tais como refatoração, documentação e testes em sistemas multilinguagem.
- Através dos resultados reportados na Seção 5.4.2, observou-se a necessidade de uma melhor análise de problemas arquiteturais em sistemas multilinguagem. Um vez que em sistemas multilinguagem são utilizadas linguagens com diferentes paradigmas e estruturas,investigar se a combinação de certos paradigmas de programação em um único sistema tende a induzir uma taxa maior ou menor de problemas arquiteturais poderá trazer benefícios no estudo da eficácia das abordagens de identificação desses problemas.
- Como visto na Subseção 4.2.5, foi necessário realizar uma equivalência de níveis para permitir a utilização de estratégias de detecção para todas as linguagens. No entanto, existem outros paradigmas e conceitos que não foram utilizados na atual abordagem. Logo, é importante estender essa

abordagem para outras linguagens que possuam paradigmas e conceitos de estruturas internas (classe, método) diferentes.

– Foi descrito na Subseção 4.2.4, para esse trabalho foram utilizadas as anomalias inter-relacionadas mais relevantes ao presente trabalho. Dessa forma, seria importante estender essa abordagem para a avaliação das anomalias inter-relacionadas relacionadas a informações hierárquicas [Macia et al., 2012a].

– Existe uma grande diversidade de domínios em sistemas multilinguagem como afirmado em Kullbach et al. [Kullbach et al., 1998] e Jerraya et al. [Jerraya and Ernst, 1999]. Dessa forma, seria importante aplicar e avaliar as características das anomalias em sistemas multilinguagem. Essa avaliação permitiria a identificação das anomalias propostas em sistemas de outros domínios.

– Estender o catálogo de referência para anomalias inter-relacionadas, visando diminuir as lacunas de conhecimento a respeito das anomalias inter-relacionadas em sistemas multilinguagem, como visto na Subseção 5.4.2. Essa extensão permitiria dar dicas de que anomalias são mais comuns nesses sistemas e de que forma elas podem aparecer.

– Na Subseção 3.2.2 foi visto que as ferramentas atuais de mapeamento manual e automático só permitem o mapeamento de elementos de código escritos em uma linguagem, e não são adequadas a sistemas multilinguagem [Feigenspan et al., 2010] [pure::variants, 2014]. Ou seja, o aperfeiçoamento das ferramentas atuais para o suporte a configuração e mapeamento automático de elementos de código escritos em diferentes linguagens é necessário.

– Como visto na Subseção 3.2.1, existe uma grande diferença de esforço total gasto pelos desenvolvedores ao aplicar diferentes estratégias para detecção de anomalias. Entretanto, esse calculo de esforço foi avaliado, neste trabalho, apenas na comparação entre *Ad hoc* vs. Estratégias baseadas em métricas. Desta forma, é sugerida uma avaliação do esforço no uso da abordagem proposta para identificação de problemas arquiteturais em sistema multilinguagem.

– Identificar qual a melhor forma de tratar diferentes limiares relacionados a diferentes linguagens para o uso em estratégias de detecção genéricas. Ou seja, estratégias de detecção para anomalias que são comuns a elementos de códigos de qualquer linguagem.

Referências Bibliográficas

[Alves et al., 2011] Alves, T. L., Hage, J., and Rademaker, P. (2011). A comparative study of code query technologies. In *: Source Code Analysis and Manipulation (SCAM), 2011 11th IEEE International Working Conference on*, pages 145–154. IEEE.

[Arnold et al., 2000] Arnold, K., Gosling, J., and Holmes, D. (2000). *The Java programming language*, volume 2. Addison-wesley Reading.

[Binkley, 2007] Binkley, D. (2007). Source code analysis: A road map. In *2007 Future of Software Engineering*, pages 104–119. IEEE Computer Society.

[Booch, 2007] Booch, G. (2007). The economics of architecture-first. *Software, IEEE*, 24(5):18–20.

[Boucké and Holvoet, 2006] Boucké, N. and Holvoet, T. (2006). Relating architectural views with architectural concerns. In *: Proceedings of the 2006 international workshop on Early aspects at ICSE*, pages 11–18. ACM.

[Buschmann et al., 2007] Buschmann, F., Henney, K., and Schimdt, D. (2007). *Pattern-oriented Software Architecture: On Patterns and Pattern Language*, volume 5. John Wiley & Sons.

[Crockford, 2008] Crockford, D. (2008). *JavaScript: the good parts*. O'Reilly Media, Inc.

[de F Carneiro et al., 2009] de F Carneiro, G., Mendonça, M., and Magnavita, R. (2009). An experimental platform to characterize software comprehension activities supported by visualization. In *Software Engineering-Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*, pages 441–442. IEEE.

[DWR, 2014] DWR (2014). Dwr. `http://directwebremoting.org/dwr/index.html`. Accessed: 2014.

[Fard and Mesbah, 2013] Fard, A. M. and Mesbah, A. (2013). Jsnose: Detecting javascript code smells. In *: Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*, pages 116–125. IEEE.

[Feigenspan et al., 2010] Feigenspan, J., Kastner, C., Frisch, M., Dachselt, R., and Apel, S. (2010). Visual support for understanding product lines. In *The 18th IEEE International Conference on Program Comprehension, ICPC 2010, Braga, Minho, Portugal, June 30-July 2, 2010*, pages 34–35. IEEE Computer Society.

[Ferreira et al., 2014] Ferreira, M., Barbosa, E., Macia, I., Arcoverde, R., and Garcia, A. (2014). Detecting architecturally-relevant code anomalies: A case study of effectiveness and effort. In *: Proceedings of the 29th Symposium on Applied Computing (SAC)*. ACM.

[Fowler, 1999] Fowler, M. (1999). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

[Garcia et al., 2009] Garcia, J., Popescu, D., Edwards, G., and Medvidovic, N. (2009). Identifying architectural bad smells. In *: Software Maintenance and Reengineering, 2009. CSMR09. 13th European Conference on*, pages 255–258. IEEE.

[Ghanam and Carpendale, 2008] Ghanam, Y. and Carpendale, S. (2008). A survey paper on software architecture visualization. *University of Calgary, Tech. Rep*.

[Godfrey and Lee, 2000] Godfrey, M. W. and Lee, E. H. (2000). Secrets from the monster Extracting mozillas software architecture. In *: Proceedings of Second Symposium on Constructing Software Engineering Tools (CoSET00)*.

[Google, 2014] Google, I. (2014). Google inc. `http://www.google.com`. Accessed: 2014.

[Harris et al., 1996] Harris, D. R., Yeh, A. S., and Reubenstein, H. B. (1996). Extracting architectural features from source code. In *Reverse engineering*, pages 109–138. Springer.

[Hochstein and Lindvall, 2005] Hochstein, L. and Lindvall, M. (2005). Combating architectural degeneration: a survey. *Information and Software Technology*, 47(10):643–656.

[JDT, 2014] JDT (2014). Jdt. `http://www.eclipse.org/jdt/`. Accessed: 2014.

[Jerraya and Ernst, 1999] Jerraya, A. and Ernst, R. (1999). Multi-language system design. In *: Proceedings of the conference on Design, automation and test in Europe*, page 134. ACM.

[Jones, 1998] Jones, T. C. (1998). *Estimating software costs*. McGraw-Hill, Inc.

[JSDT, 2014] JSDT (2014). Jsdt. `http://www.eclipse.org/webtools/jsdt/`. Accessed: 2014.

[Karus and Gall, 2011] Karus, S. and Gall, H. (2011). A study of language usage evolution in open source software. In *: Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 13–22. ACM.

[Kontogiannis et al., 2006] Kontogiannis, K., Linos, P., and Wong, K. (2006). Comprehension and maintenance of large-scale multi-language software applications. In *: Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on*, pages 497–500. IEEE.

[Kullbach et al., 1998] Kullbach, B., Winter, A., Dahm, P., and Ebert, J. (1998). Program comprehension in multi-language systems. In *: Reverse Engineering, 1998. Proceedings. Fifth Working Conference on*, pages 135–143. IEEE.

[Linos et al., 2007] Linos, P., Lucas, W., Myers, S., and Maier, E. (2007). A metrics tool for multi-language software. In *: Proceedings of the 11th IASTED International Conference on Software Engineering and Applications*, pages 324–329. ACTA Press.

[Linos et al., 2003] Linos, P. K., Chen, Z.-h., Berrier, S., and O'Rourke, B. (2003). A tool for understanding multi-language program dependencies. In *: Program Comprehension, 2003. 11th IEEE International Workshop on*, pages 64–72. IEEE.

[Lippert and Roock, 2006] Lippert, M. and Roock, S. (2006). *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*. John Wiley & Sons.

[Macia, 2013] Macia, I. (2013). On the detection of architecturally-relevant code anomalies in software systems. Phd, DI, PUC-Rio.

[Macia et al., 2012a] Macia, I., Arcoverde, R., Cirilo, E., Garcia, A., and von Staa, A. (2012a). Supporting the identification of architecturally-relevant code anomalies. In *: Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 662–665. IEEE.

[Macia et al., 2012b] Macia, I., Garcia, J., Popescu, D., Garcia, A., Medvidovic, N., and von Staa, A. (2012b). Are automatically-detected code anomalies relevant to architectural modularity? an exploratory analysis of evolving systems. In *: Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, pages 167–178. ACM.

[Maier et al., 2004] Maier, M. W., Emery, D., and Hilliard, R. (2004). Ansi/ieee 1471 and systems engineering. *Systems Engineering*, 7(3):257–270.

[Marinescu, 2004] Marinescu, R. (2004). Detection strategies: Metrics-based rules for detecting design flaws. In *: Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 350–359. IEEE.

[Marinescu et al., 2010] Marinescu, R., Ganea, G., and Verebi, I. (2010). incode: Continuous quality assessment and improvement. In *: Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*, pages 274–275. IEEE.

[Martin, 2003] Martin, R. C. (2003). *Agile software development: principles, patterns, and practices*. Prentice Hall PTR.

[Mayer and Schroeder, 2012] Mayer, P. and Schroeder, A. (2012). Cross-language code analysis and refactoring. In *: Source Code Analysis and Manipulation (SCAM), 2012 IEEE 12th International Working Conference on*, pages 94–103. IEEE.

[Moha et al., 2010] Moha, N., Gueheneuc, Y.-G., Duchien, L., and Le Meur, A. (2010). Decor: A method for the specification and detection of code and design smells. *Software Engineering, IEEE Transactions on*, 36(1):20–36.

[MSN, 2014] MSN (2014). Msn. `http://www.msn.com`. Accessed: 2014.

[Nguyen et al., 2012] Nguyen, H. V., Nguyen, H. A., Nguyen, T. T., Nguyen, A. T., and Nguyen, T. N. (2012). Detection of embedded code smells in dynamic web applications. In *: Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*, pages 282–285. IEEE.

[Nicolay et al., 2013] Nicolay, J., Noguera, C., De Roover, C., and De Meuter, W. (2013). Determining dynamic coupling in javascript using object type inference. In *: Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*, pages 126–135. IEEE.

[Ohloh, 2013] Ohloh (2013). Ohloh, languages statistics. `http://www.ohloh.net/languages?query=java&sort=name2013`. Accessed: 2013-09-10.

[Olbrich et al., 2009] Olbrich, S., Cruzes, D. S., Basili, V., and Zazworka, N. (2009). The evolution and impact of code smells: A case study of two open source systems. In *: Proceedings of the 2009 3rd international symposium on empirical software engineering and measurement*, pages 390–400. IEEE Computer Society.

[Osmani, 2012] Osmani, A. (2012). *Learning JavaScript Design Patterns*. "OReilly Media, Inc.".

[Patzer et al., 2004] Patzer, A., Moodie, M., and Weaver, J. L. (2004). *Foundations of JSP design patterns*, volume 42. Springer.

[pure::variants, 2014] pure::variants (2014). pure::variants. `http://www.pure-systems.com/pure_variants.49.0.html`. Accessed: 2014.

[Richards et al., 2010] Richards, G., Lebresne, S., Burg, B., and Vitek, J. (2010). An analysis of the dynamic behavior of javascript programs. In *: ACM Sigplan Notices*, volume 45, pages 1–12. ACM.

[Schumacher et al., 2010] Schumacher, J., Zazworka, N., Shull, F., Seaman, C., and Shaw, M. (2010). Building empirical support for automated code smell detection. In *: Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, page 8. ACM.

[Shaw and Garlan, 1996] Shaw, M. and Garlan, D. (1996). *Software architecture: perspectives on an emerging discipline*, volume 1. Prentice Hall Englewood Cliffs.

[Silva, 2013] Silva, A. (2013). Reuso de estratégias sensíveis a dominio para deteccao de anomalias de codigo Um estudo de multiplos casos. Master, DI, PUC-Rio.

[Spring, 2014] Spring (2014). Spring. `http://projects.spring.io/spring-framework/`. Accessed: 2014.

[Struts, 2014] Struts (2014). Struts. `http://struts.apache.org/`. Accessed: 2014.

[Taylor et al., 2009] Taylor, R. N., Medvidovic, N., and Dashofy, E. M. (2009). *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing.

[Terceiro et al., 2010] Terceiro, A., Costa, J., Miranda, J., Meirelles, P., Rios, L. R., Almeida, L., Chavez, C., and Kon, F. (2010). Analizo: an extensible multi-language source code analysis and visualization toolkit. In *: Brazilian Conference on Software: Theory and Practice (CBSoft)–Tools, Salvador-Brazil*, volume 29.

[Sant'Anna et al., 2007] Sant'Anna, C., Figueiredo, E., Garcia, A., and Lucena, C. (2007). On the modularity assessment of software architectures do my architectural concerns count. In *: Proc. International Workshop on Aspects in Architecture Descriptions - AARCH. 07, AOSD*, volume 7.

[Tiobe, 2013] Tiobe (2013). Tiobe programming community index. `http://www.tiobe.com/tpci.htm`. Accessed: 2013.

[Understand, 2013] Understand (2013). Understand. `http://www.scitools.com/`. Accessed: 2013.

[Wedyan et al., 2009] Wedyan, F., Alrmuny, D., and Bieman, J. M. (2009). The effectiveness of automated static analysis tools for fault detection and refactoring prediction. In *: Software Testing Verification and Validation, 2009. ICST'09. International Conference on*, pages 141–150. IEEE.

[Wilkerson et al., 2012] Wilkerson, J. W., Nunamaker Jr, J. F., and Mercer, R. (2012). Comparing the defect reduction benefits of code inspection and test-driven development. *Software Engineering, IEEE Transactions on*, 38(3):547–560.

[Wohlin et al., 2012] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2012). *Experimentation in software engineering*. Springer.

[Wong et al., 2011] Wong, S., Cai, Y., Kim, M., and Dalton, M. (2011). Detecting software modularity violations. In *: Proceedings of the 33rd International Conference on Software Engineering*, pages 411–420. ACM.

[Yahoo, 2014] Yahoo (2014). Yahoo. `http://www.yahoo.com`. Accessed: 2014.

[Yin, 2009] Yin, R. K. (2009). *Case study research Design and methods*, volume 5. sage.