UNIVERSIDADE FEDERAL DE SERGIPE
CAMPUS ALBERTO CARVALHO
DEPARTAMENTO DE SISTEMAS DE INFORMAÇÃO

CLEVERTON DOS SANTOS

UMA TÉCNICA PARA RECOMENDAR REFATORAÇÕES DE MÉTODOS LONGOS BASEADO NO CONTEXTO ARQUITETURAL DA CLASSE

ITABAIANA
2016

UNIVERSIDADE FEDERAL DE SERGIPE
CAMPUS ALBERTO CARVALHO
DEPARTAMENTO DE SISTEMAS DE INFORMAÇÃO

CLEVERTON DOS SANTOS

UMA TÉCNICA PARA RECOMENDAR REFATORAÇÕES DE MÉTODOS LONGOS BASEADO NO CONTEXTO ARQUITETURAL DA CLASSE

Trabalho de Conclusão de Curso submetido ao Departamento de Sistemas de Informação da Universidade Federal de Sergipe como requisito parcial para a obtenção do título de Bacharel em Sistemas de Informação.

Orientador: MARCOS BARBOSA DÓSEA

ITABAIANA
2016

Sobrenome, Nome.

Título do trabalho / Nome completo – Itabaiana: UFS, Ano.

99f. (indica o número de páginas do trabalho); 99 cm (indica o tamanho)

Trabalho de Conclusão de Curso (graduação) – Universidade Federal de Sergipe, Curso de [Nome do curso], Ano.

1. Assunto. 2. Área de Concentração - TCC. 3. Curso. I. Título.

CLEVERTON DOS SANTOS

UMA TÉCNICA PARA RECOMENDAR REFATORAÇÕES DE MÉTODOS LONGOS BASEADO NO CONTEXTO ARQUITETURAL DA CLASSE

Trabalho de Conclusão de Curso submetido ao corpo docente do Departamento de Sistemas de Informação da Universidade Federal de Sergipe (DSIITA/UFS) como parte dos requisitos para obtenção do grau de Bacharel em Sistemas de Informação.

Itabaiana, (___ , maio de 2016).

BANCA EXAMINADORA:

_____

Profº Marcos Barbosa Dósea, Mestre
Orientador
DSIITA/UFS

_____

Profº Alcides Xavier Benicasa, Doutor
DSIITA/UFS

_____

Profº Methanias Colaço Rodrigues Junior, Doutor
DSIITA/UFS

AGRADECIMENTOS

Meus agradecimentos,

A Deus, que com sua imensa grandeza me deu força e me manteve no caminho desta jornada.

Aos meus pais José Erlito e Joselita pelo amor, carinho e por sempre me apoiar em seguir nos caminhos do estudo e do conhecimento.

A meus irmãos e irmãs José Gildernir, Cleyton, Josenilda e Maria José por compartilhar meus bons e maus momentos.

Ao orientador Profº Marcos Barbosa Dósea pela excelente orientação no caminho da pesquisa, pelas críticas visando buscar o melhor empenho e pela maestria e paciência em propagar o conhecimento.

A todos meus antigos e novos amigos que conquistei durante o curso pelas palavras de incentivo, companheirismo, brincadeiras, discussões e ótimo convívio.

Epígrafe

“Quem diz que não pode ser feito nunca deve interromper aquele que está fazendo”
(Monkey D. Luffy)

DOS SANTOS, Cleverton. **Uma técnica para recomendar refatorações de métodos longos baseado no contexto arquitetural da classe**. 2016. Trabalho de Conclusão de Curso – Curso de Sistemas de Informação, Departamento de Sistemas de Informação, Universidade Federal de Sergipe, Itabaiana, 2016.

RESUMO

A atividade de implementação do software pode introduzir anomalias no código. Detectar e remover essas anomalias são tarefas importantes para manter a qualidade do código em desenvolvimento. Várias técnicas manuais e automáticas são propostas para identificação de anomalias introduzidas no código. As abordagens automáticas que ajudam os desenvolvedores na tomada de decisão ou recomendam itens de interesse funcionam como sistemas de recomendação. Uma das anomalias mais comuns e detectadas pela maioria das ferramentas de análise automática de código são os métodos longos. Entretanto, a identificação de métodos longos nessas abordagens, baseia-se na utilização de valores limiares genéricos, utilizados para todas as classes do sistema, levando ao incremento de falsos positivos e falsos negativos na lista de anomalias detectadas. Nesse contexto, este trabalho propõe uma nova técnica para identificar valores limiares para métodos longos que consideram o contexto arquitetural da classe e são extraídos de uma versão com a mesma arquitetura do software. Adicionalmente foi desenvolvida uma ferramenta para dar suporte à técnica proposta que recomenda ações ao desenvolvedor durante a implementação do código. Foi realizado um estudo exploratório para avaliar a abordagem proposta utilizando dez versões do software MobileMedia. Na avaliação realizada a técnica proposta obteve 100% de recall, significando que todos os métodos longos definidos previamente por especialistas foram identificados. A precisão do método alcançou 32%. Dessa forma, o método proposto torna-se promissor em relação às abordagens existentes por diminuir o número de falsos positivos e falsos negativos.

Palavras-chave: Interesse Arquitetural, Métodos Longos, Recomendação.

ABSTRACT

The software implementation activity can introduce code anomalies. Detect and remove these anomalies are essential tasks to keep the quality of source code. Several manual and automated are proposed to identify code anomalies. The automatic approaches help developers to make decision or recommend items like a recommendation systems. Long method is a common code anomalies detected by code analysis tools. But, the identification of long methods are based in generic thresholds, used for all system classes, leading to increase of false positives and false negatives. In this context, this work propose a new approach to identify thresholds for long methods that consider the class architectural context and are extracted from a project with the same architecture. In addition, we develop a tool to support the proposed technique that recommends actions for the developer during the source code implementation. A case study was used to evaluate the proposed approach using ten versions of MobileMedia software. In the assessment, the proposed technique had 100% recall, it means that all the long methods previously defined by experts were identified. The precision of the method reached 32%. So, the proposed method is promising considering the existing approaches to shorten the number of false positives and false negatives.

Key-words: *Architectural Interest. Long Methods. Recommendation.*

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
AST	*Abstract Syntax Tree*
CPD	*copy-paste-detector*
CVS	*Version Control System*
FE	*Feature Envy*
GC	*God Class*
GM	*God Method*
HTML	*HyperText Markup Language*
JDT	*Java Development Tools*
KLOC	*Kilo Lines of Code*
LM	*Long Method*
LOC	*Line of Code*
Md	Mediana
MNOB	*Maximum Number of Branches*
MOOD	*Object Oriented Design Metrics*
NOP	*Number of Parameters*
NOLV	*Number of Local Variables*
OSGi	*Open Services Gateway Initiative*
OO	Orientados à Objetos
PDE	*Plug-in Developer Environment*
PHP	*Hypertext Preprocessor*
Pp	Posição do Percentil
PPM	*Prediction by Partial Matching*
QMOOD	*Quality Model for Object Oriented Design*
SDK	*Software Development Kit*
SPL	Linha de Produto de Software
SRES	Sistemas de Recomendação para Engenharia de *Software*
SWT	*Standard Widget Toolkit*
UML	*Unified Modeling Language*

SUMÁRIO

1 INTRODUÇÃO

A evolução inevitável dos sistemas de *software* resulta no aumento de seu tamanho e complexidade, o que, por sua vez, pode levar a degradação da arquitetura (GUIMARAES; GARCIA; CAI, 2014). Um dos principais sintomas de degradação do sistema é a manifestação progressiva de anomalias no código. A detecção e remoção dessas anomalias são tarefas importantes para prolongar a longevidade do sistema (ARCOVERDE et al., 2012).

Fowler (2004) propõe um catálogo com anomalias, denominadas *code smells*, que são comuns em códigos de *software*. Uma das anomalias propostas e bastante discutida na literatura são os métodos longos.

Várias técnicas vêm sendo propostas para identificar anomalias introduzidas no código McCabe (1976), Munson e Elbaum (1998), Bansiya e Davis (2002), Abreu e Carapuça (1994), Chidamber e Kemerer (1994), Briand, Devanbu e Melo (1997), Gupta, Goyal e Goyal (2015). Segundo (RADJENOVIĆ et al., 2013). Algumas dessas técnicas têm sido validadas apenas em um pequeno número de estudos e outras têm sido propostas, mas nunca utilizadas (RADJENOVIĆ et al., 2013).

Ferramentas de detecção de anomalias automatizam essas técnicas para auxiliar a identificação de anomalias. FindBugs, PMD e JDeodorant são exemplos desses tipos de ferramenta.

Algumas dessas ferramentas funcionam como sistemas de recomendação para ajudar desenvolvedores na tomada de decisão. Sistemas de Recomendação para Engenharia de *Software* (SRES) estão surgindo para ajudar os desenvolvedores em várias atividades (ROBILLARD; WALKER; ZIMMERMANN, 2010), entre elas a fase de implementação do código.

1.1 Problemática

Um problema das abordagens existentes é que elas utilizam valores limiares fixos para identificação de métodos longos nas classes. Na literatura vários estudos sugerem valores limiares para métodos longos extraídos da análise de diversos software. Mesmo sendo possível calibrá-los usando a experiência da equipe de desenvolvimento, esses valores são utilizados para analisar todas as classes independente da sua função na arquitetura. Quando o

contexto arquitetural é desconsiderado um grande número de falsos positivos e falsos negativos são gerados por essas abordagens.

Adicionalmente, a maioria das abordagens de identificação de métodos longos são executadas depois que o código-fonte é desenvolvido. O desenvolvedor nem sempre tem tempo ou motivação para executar todas as correções sugeridas pelas ferramentas após a implementação do código.

1.2 Objetivo Geral

Definir e avaliar uma técnica para identificar e recomendar a realização de refatorações em métodos longos baseada no contexto arquitetural da classe.

1.3 Objetivos Específicos

- Definir uma técnica para identificação de métodos longos baseada em valores limiares obtidos de um projeto exemplo com arquitetura semelhante e no interesse arquitetural da classe avaliada;
- Implementar uma ferramenta que recomende métodos longos com base na técnica proposta;
- Avaliação da ferramenta, através da análise de diversas versões de um sistema de software.

1.4 Metodologia do Trabalho

A pesquisa utilizada no projeto será uma pesquisa-ação.

> A pesquisa-ação é um tipo de pesquisa social com base empírica que é concebida e realizada em estreita associação com uma ação ou tom a resolução de um problema coletivo e no qual os pesquisadores e os participantes representativos da situação ou do problema estão envolvidos de modo cooperativo ou participativo (THIOLLENT, 1996).

A mesma tem como objetivo resolver ou, pelo menos, esclarecer os problemas da situação que está sendo observada.

A configuração de uma pesquisa-ação depende dos seus objetivos e do contexto no qual é aplicado (THIOLLENT, 1996). Dessa forma, a metodologia visa atingir os objetivos e pode ser dividida em cinco etapas a seguir:

Etapa 1 – Fase exploratória. Consiste na realização de um estudo sobre o fenômeno que está sendo pesquisado. O mesmo corresponde a um levantamento inicial sobre (I) técnicas para identificar e evitar anomalias de código; (II) técnicas para recomendar ações para o desenvolvedor.

Etapa 2 – Colocação dos problemas. Trata-se em definir a problemática na qual o tema escolhido adquira sentido. O problema ocorre devido às abordagens existentes para identificação de métodos longos não levarem em consideração o contexto do código que está sendo analisado, gerando muitos falsos positivos e falsos negativos. Além disso, a maioria das técnicas existentes precisam ser executadas pelo desenvolvedor, deixando o processo de avaliação do código para o final.

Etapa 3 – Objetivo da pesquisa. O objetivo determina a utilização de uma abordagem para ajudar desenvolvedores a identificar oportunidades de refatorações em métodos, levando em consideração o número de linhas e o contexto arquitetural da classe.

Etapa 4 – Construção de uma ferramenta para recomendar refatorações em métodos, com base na abordagem proposta.

Etapa 5 – Avaliação. Será feita uma análise da ferramenta construída, observando versões de um sistema de software.

1.5 Trabalhos Relacionados

Vários trabalhos relacionados identificam anomalias de código e recomendam ações para os desenvolvedores. Relacionamos alguns trabalhos que realizam a análise de código fonte para investigar aspectos que influenciam na qualidade do software.

Silva, Terra e Valente (2015) propõem uma ferramenta que analisa e recomenda oportunidades de *Extract Method*. Com base em regras predefinidas, trechos de métodos que podem ser refatorados com o *Extract Method* são selecionados, filtrados e então apresentados aos desenvolvedores. A identificação de oportunidades de refatorações é feita com base na similaridade entre as séries de dependência. Na filtragem os trechos de métodos basicamente são ordenados e filtrados com base nas recomendações máximas por método (três) e pontuação por valor mínimo (valor definido pelo usuário). Para avaliar a ferramenta, foram realizados dois estudos. O primeiro estudo levou em consideração a *recall* e *precision* em relação a ferramenta JDeodorant. O JExtract foi configurado com três configurações diferentes e o JDeodorant usando a configuração padrão. No segundo estudo foi replicado o

primeiro com dez populações de sistemas abertos desenvolvidos em Java, porém, os dados não foram comparados com a ferramenta JDeodorant. Como resultados, no primeiro experimento JExtract superou JDeodorant independente da configuração utilizada, já no segundo os valores variaram tendo uma média de *recall* aumentada e diminuição da *precision*. Porém, mesmo com a diminuição da *precision* os valores foram considerados como aceitáveis. O estudo se assemelha ao nosso em alguns pontos: (i) a técnica foi automatizada como um plug-in para eclipse, (ii) utiliza regras pré-definidas para encontrar trechos de códigos que necessitam ser refatorados e (iii) para avaliar a ferramenta levou em consideração a *recall* e *precision* da ferramenta. Entretanto nossa abordagem tem como foco analisar a estrutura do código existente e o contexto arquitetural das classes para identificação dos métodos longos no código em desenvolvimento.

Terra, Valente e Bigonha (2015) apresentam um trabalho que visa a reparação de violações no código em níveis arquitetônicos. Para tal, desenvolveram um sistema de recomendação, chamado DCLfix, que sugere recomendações de refatorações para violações detectadas como resultado de um processo de arquitetura em conformidade com DLC, uma linguagem de restrição de arquitetura. DCLfix trabalha em conjunto de restrições DCL (especificadas pelo arquiteto de *software*), um conjunto de violações arquitetônicas (levantada pela própria ferramenta), e o código fonte do sistema para fornecer um conjunto de recomendações de refatoração para orientar o processo de remoção das violações detectadas. Como forma de avaliação, os autores, avaliaram a aplicação da ferramenta em dois grandes sistemas industrial. No primeiro sistema, denominado Geplanes, recomendou refatorações corretamente em 31 violações das 41 encontradas. No segundo sistema, chamado TCOM, desencadeou refatorações corretamente para 624 violações das 787 encontradas. A pesquisa também aborda a utilização da ferramenta como um *plug-in* para o eclipse que usa informações da arquitetura que precisam ser especificadas. Nosso trabalho difere porque não é necessário descrever as regras da arquitetura que serão verificadas para encontrar o método longo, essas regras são extraídas automaticamente do código fonte existente.

Garcia et al. (2011) propõem uma nova técnica que aproveita os interesses do sistema para automatizar a recuperação de componentes e conectores. O objetivo é recuperar automaticamente a arquitetura do software. Em sua abordagem além de explorar informações estruturais para descobrir componentes, primeiramente recupera interesses do sistema implementado usando uma técnica de recuperação de informação e, logo após, combina os interesses com as informações estruturais para identificar automaticamente componentes e

conectores. Para obter os interesses do sistema, utiliza um modelo estatístico de linguagem utilizada na recuperação de informação chamado *Latent Dirichlet Allocation* (LDA). Como parte da avaliação foram utilizados os *frameworks* MALLET, Weka e Soot para extrair os interesses, realizar aprendizado supervisionado e extrair informações estruturais, respectivamente. A comparação desse trabalho com o nosso estar em considerar os interesses arquiteturais para recuperar automaticamente informações dos sistemas. Porém, enquanto o trabalho tenta recuperar componentes e conectores, nós tentamos obter valores limiares a serem considerados na identificação de métodos longos.

No trabalho proposto por Holmes e Murphy (2005), é descrita uma abordagem que usa a estrutura do código em desenvolvimento para encontrar exemplos relevantes em um repositório. O contexto estrutural que é utilizado para formar uma consulta é extraído automaticamente a partir do código escrito pelo desenvolvedor. Para procurar exemplos relacionados em um repositório o desenvolvedor precisa emitir um pedido de pesquisa, através de uma combinação de teclas. O repositório de exemplos é extraído automaticamente a partir de aplicativos existentes que usam a estrutura da consulta. Para investigar a abordagem foi desenvolvida a ferramenta cliente-servidor Strathcona. A parte cliente é um plug-in para o Eclipse, já a parte servidor da ferramenta abriga o repositório de exemplos e seleciona exemplos utilizando um conjunto de heurísticas estruturais correspondentes. Para avaliar a abordagem foi realizada uma avaliação quantitativa, onde dois sujeitos replicaram quatro casos que consistiam em tarefas de programação relacionadas a escrever *plug-ins* para o Eclipse. Na avaliação o repositório foi preenchido com o código fonte do sistema Eclipse, por retratarem um bom exemplo do uso de *frameworks*. Este trabalho tenta encontrar e recomendar exemplos de código relacionados ao que estar sendo desenvolvido, nós também utilizamos informações do *design* do código existente para encontrar valores limiares usado na identificação de métodos longos e realizar recomendações relevantes enquanto o software é desenvolvido.

Takuya e Masuhara (2011) desenvolvem um sistema de recomendação de código chamada Selene, uma ferramenta de recomendação de código-fonte com base em um motor de busca baseada em texto ao longo de um repositório de código fonte. Selene (i) utiliza toda a edição do código como fonte de pesquisa e (ii) busca e exibe fragmentos de código semelhantes de um repositório de programas de exemplo. O objetivo de Selene é recomendar expressões idiomáticas de bibliotecas e *frameworks* adequados ao contexto. Os principais componentes de Selene são um *plug-in* Eclipse com parte cliente da ferramenta e um motor de

busca associativa no lado do servidor. O motor de busca associativa usado é o GETA, cuja pesquisa recupera a partir de um conjunto de palavras-chave documentos que contém conjuntos de palavras-chave semelhantes. Com base nos experimentos realizados os autores mostraram que o tempo médio para a pesquisa é de 2,7 segundos, com um desvio padrão de 1,25 para um ensaio de 50 vezes. Este trabalho também utiliza o código fonte existe para recomendação, mas o nosso difere por tentar identificar falhas no *design* ao invés de exemplos de código.

Por fim, Arcoverde et al. (2012) apresenta uma abordagem para recomendação e classificação (*rank*) de anomalias de código arquitetonicamente relevantes. O sistema é enraizado sobre a ideia de detecção de padrões de anomalias de código. Seu objetivo é ajudar os desenvolvedores a encontrar estruturas de código que causam problemas de arquitetura com mais precisão, e, então, ajudar a priorizar essas anomalias. A fim de identificar correta e automaticamente anomalias relevantes, teve-se que decidir o tipo de informação que seria levada em consideração para a classificação de anomalias como relevantes ou não. Esse tipo de informação pode variar desde classes que implementam o que diz respeito a estrutura estática do código, como dependências entre as classes. Do ponto de vista da arquitetura, há duas heurísticas que consideram como relevantes: o número de anomalias encontradas por módulo, e o papel da arquitetura do componente. Em relação ao primeiro, classes com mais anomalias são consideradas alvos de alta prioridade para refatoração. Quanto ao segundo, quando a arquitetura de informação está disponível, o papel de um componente desempenha no modelo global da arquitetura influência sobre seu nível de prioridade. Nosso trabalho também analisa o papel do componente arquitetural, mas não tem como objetivo priorizar as anomalias encontradas. Nosso objetivo é melhorar o processo de identificação de anomalias relevantes para diminuir o número de falsos positivos e falsos negativos.

1.6 Organização do Trabalho

O restante do trabalho está organizado da seguinte forma:

O Capítulo 2 apresenta a Fundamentação Teórica sobre os principais temas relacionados à pesquisa, a saber: Arquitetura de Software, Qualidade de Código, Anomalias de Código, Ferramentas para Detecção de Anomalias de Código, Sistemas de Recomendação para Evitar Anomalias de Código e *Plug-ins* do Eclipse como Ferramenta de Recomendação.

O Capítulo 3 explica o Método Sensível a Arquitetura para Buscar Métodos Longos, contendo uma descrição detalhada sobre o método proposto e a ferramenta desenvolvida para apoiar o uso do método.

No Capítulo 4, é apresentada a realização da Avaliação realizada sobre o método proposto.

Por fim, no Capítulo 5 o trabalho é encerrado com a Conclusão e propostas de possíveis trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo aborda os conceitos utilizados para entender os principais temas relacionados ao tema proposto. A Seção 2.1 explica a importância da definição adequada da arquitetura para evolução do software. A Seção 2.2 discute conceitos relativos à qualidade de código e como avaliar, por meio de métricas, se o resultado de uma medida é feito da maneira correta. Na Seção 2.3 abordamos anomalias de código, com destaque para os métodos longos. A Seção 2.4 aborda ferramentas para detecção de anomalias de código apresentando algumas ferramentas e como elas são classificadas. A Seção 2.5 conceitua sistemas de recomendação para engenharia de software, além de destacar alguns tipos de sistemas de recomendação e apresentar algumas ferramentas. Por fim, na Seção 2.6 são explicados alguns conceitos sobre a construção de *plug-ins para o Eclipse*.

2.1 Arquitetura de Software

A arquitetura de *software* é uma solução estrutural que desempenha papel fundamental no gerenciamento da complexidade do software. Segundo Bass, Clements e Kazman (2003) a estrutura ou estruturas do sistema, que abrange os componentes de *software*, as propriedades externamente visíveis desses componentes e as relações entre eles, é denominado arquitetura de *software*.

Sistemas grandes são sempre decompostos em subsistemas que fornecem um conjunto de serviços que estejam relacionados (SOMMERVILLE, 2011). Para enfatizar o papel dos componentes de *software* de qualquer arquitetura Pressman (2011) define que a arquitetura não é o *software* operacional, mas sim, uma representação que permite (i) analisar a efetividade do projeto no atendimento dos requisitos declarados, (ii) considerar alternativas de arquitetura em um momento em que realizar mudanças tenham um menor custo e (iii) reduzir os riscos associados à construção do *software*.

Levando essa definição para o contexto de projeto de arquitetura, Pressman (2011) relata que um componente de *software* pode ser algo tão simples quanto um módulo de um programa ou uma classe orientada a objetos, porém pode ser estendido para abranger banco de dados que possibilitam a criação de redes clientes e servidores. Sommerville (2011) classifica esses dois níveis de abstração como arquiteturas em pequena e larga escala, respectivamente.

Para Bosch (2000) a arquitetura de *software* é importante por que afeta fatores como desempenho, robustez, capacidade de distribuição e manutenibilidade de um sistema. Para o mesmo, requisitos funcionais do sistema são implementados por componentes individuais. Já os requisitos não funcionais dependem da arquitetura do sistema, podendo ser influenciados por componentes individuais. Devido à estreita relação entre os requisitos não funcionais e a arquitetura do *software*, o estilo e a estrutura da arquitetura que é escolhida para um sistema devem depender de requisitos não funcionais do sistema como desempenho, proteção, segurança, disponibilidade e manutenção (PRESSMAN, 2011).

Bass, Clements e Kazman (2003) explicam três razões-chave para projetar e documentar explicitamente uma arquitetura de *software*:

- Comunicação de *stakeholders*. As representações da arquitetura são um facilitador para estabelecer uma melhor comunicação entre todos os envolvidos no desenvolvimento de um projeto de um sistema *software*.
- Análise do sistema. Tornar a arquitetura do sistema explícita evidencia decisões de projetos iniciais que terão impacto em todo o trabalho de engenharia posterior.
- Reuso em larga escala – A arquitetura é um modelo compacto e administrável de como um sistema está estruturado e de como os componentes operam entre si. Para sistemas com requisitos similares a arquitetura do *software* é muitas vezes a mesma e, portanto, pode apoiar o reuso em larga escala.

Segundo Sommerville (2011), uma arquitetura quando adequadamente documentada, dentre os diversos fatores, facilita a compressão da estrutura de um sistema e evita a deterioração do código fonte ou anomalias arquiteturais. Um sistema apropriado possui uma arquitetura que implementa todos os requisitos e atenda o objetivo do sistema. As modificações realizadas nos sistemas, muitas vezes, são feitas ao longo de um período de tempo, ocasionando danos à estrutura do sistema. Dessa forma, o mesmo pode se tornar inadequado devido às anomalias de código que possam vir a surgir.

Essa importância da arquitetura do software foi um dos motivadores para usarmos o código existente como base para definição de valores limiares para identificação de métodos longos. Para aumentar a precisão da recomendação consideramos a estrutura arquitetural do código existente. Desconsiderar essas especificidades acaba aumentando o número de falsos positivos e falsos negativos gerados pelas abordagens automáticas existentes que detectam anomalias de código.

2.2 Qualidade de Código

Koscianski (2007) mostra como o conceito de qualidade é flexível e pode receber diferentes significados. Sob a perspectiva de *software*, o assunto qualidade é bastante extenso. Segundo Pressman (2011) a qualidade de software é a conformidade com requisitos funcionais e de desempenho explicitamente declarados, padrões de desenvolvimento explicitamente documentados e características implícitas, que são esperadas em todo o software desenvolvido profissionalmente. A qualidade de software ainda depende principalmente do correto emprego de boas metodologias pelos desenvolvedores.

Qualidade de código aborda problemas relativos à comunicação entre os envolvidos em um projeto de *software*, aspectos de desenvolvimento e soluções de implementação ligadas à leitura, melhoria da escrita, documentação e reaproveitamento de código. Assim, escrever o código corretamente auxilia na busca de objetivos como redução de defeitos e validação de requisitos.

Para identificar se um *software* possui defeitos e está de acordo com o especificado as técnicas de verificação e validação (SOMMERVILLE, 2011) são fundamentais. Essas atividades servem para assegurar que o *software* funcione de acordo com os requisitos estabelecidos pelos *stakeholders*. A verificação é o processo de determinar se o *software* está sendo construído da maneira correta, de acordo com os requisitos especificados. A validação é o processo de confirmação de que o *software* é aquilo que os *stakeholders* querem.

Mesmo com todo trabalho de avaliação, se não for possível obter dados confiáveis, a avaliação é posta em risco. Assim, é preciso verificar se o resultado de uma medida usada para julgar um *software* é feito da maneira correta. Para avaliar a qualidade de produtos de software existem métricas mencionadas a seguir:

Métricas de Funcionalidade – Verificadas através de revisões e testes. Uma parte da avaliação pode ser feita no início do desenvolvimento, ainda na fase de projeto de arquitetura. Ainda no início do desenvolvimento, com a subcaracterística de interoperabilidade, pode-se verificar se a arquitetura contempla a implementação das funções necessárias. Algumas métricas de funcionalidade são tarefas de interoperabilidade confirmadas em testes, número de testes de interoperabilidade feitos, número de funções projetadas e implementadas, funções cuja acurácia foi validada ao serem executadas.

Métricas de Manutenibilidade – São importantes para o gerenciamento da atividade de manutenção de software. Podem ser aplicadas para prever o esforço necessário para manter

o software ou criar uma base de dados histórica para acompanhamento do desenvolvimento. Algumas métricas de manutenibilidade bastante conhecidas são Medidas de tamanho como medidas de linha de código e pontos de função; Complexidade estrutural que engloba complexidade ciclomática e métrica de *Halstead*; Medidas baseadas no fluxo de dados; Acoplamento e coesão; e *Unified Modeling Language* (UML) e orientação a objetos.

Métricas de Usabilidade – Possui uma dependência forte de aspectos subjetivos, que são fatores não quantificáveis pelo avaliador em função de características próprias ao software. Utiliza lista de verificação (*checklists*) e contagem de itens para verificar problemas de interfaces e ergonomia. Embora esteja longe de um estudo de laboratório, essa técnica é simples de usar, possui baixo custo e pode ser muito efetiva.

Métricas de confiabilidade – Significa, de maneira geral estimar a probabilidade de ocorrência de falhas. Como falha é a manifestação de um defeito, confiabilidade também pode estar associada a estimativa da densidade de defeitos presente no *software*. Mesmo sendo difícil de quantificar com precisão, a confiabilidade pode ser medida por medidas de disponibilidade e classificação de falhas.

Métricas de Eficiência – Medem o comportamento temporal e a utilização de recursos. Medidas de comportamento temporal são fáceis de definir e dependem do contexto da aplicação, já as medidas de utilização de recursos se relacionam com os elementos físicos que o software utiliza durante sua execução. Alguns exemplos de métricas de eficiência são número de transações, tempo total para execução de uma tarefa, tempo de reposta de uma tarefa e tempo total para execução de testes.

Métricas de Portabilidade – Indica que um produto será adaptado para uso em outra plataforma diferente. A adaptabilidade reflete o esforço necessário para que um produto seja adaptado para ser executado em outro ambiente. Desse modo, as métricas irão se preocupar com manutenção de código em diferentes plataformas. Alguns exemplos são número de instalações efetuadas com sucesso e número de problemas enfrentados ao instalar o produto.

Essas métricas servem como forma de avaliação quantitativa na administração da qualidade, servindo como base de requisitos e definição de objetivos de qualidade de um produto (KOSCIANSKI; DOS SANTOS SOARES, 2007). Nosso trabalho foca na métrica de linhas de código por método, uma métrica de manutenibilidade que avalia o tamanho.

2.3 Anomalias de Código

Anomalias de código[1] são problemas no código que podem dificultar a manutenibilidade do software. Fowler (2004) conceitua anomalias de código como problemas relacionados ao uso de más práticas que afetam a evolução do código. Essas anomalias podem influenciar na inserção de erros responsáveis por futuras falhas, ou seja, são responsáveis pelas dificuldades encontrados na manutenção do sistema. Elas descrevem uma situação que pode indicar possíveis falhas de projeto do software.

O tratamento de anomalias de código pode ser realizado de forma preventiva enquanto o projeto é desenvolvido. Pata tal, é necessário identificar as partes do código que violam a estrutura e as propriedades desejadas do projeto.

Às vezes, o *software* modificado perde boa parte de sua estrutura, ocasionando diminuição da capacidade de manutenção. Umas das técnicas que visam melhorar a capacidade de manutenção é a refatoração (MEANANEATRA; RONGVIRIYAPANISH; APIWATTANAPONG, 2011). Segundo Li e Shatnawi (2007) refatoração é o processo de alteração de um sistema de *software* de modo que o comportamento externo do código não mude, mas sua estrutura interna seja melhorada.

Refatoração não é a salvação para todos os problemas, mas sim uma ferramenta que serve de suporte para ajudar a manter o código livre de problemas. Dessa forma, a refatoração pode e deve ser usada para diversos propósitos: (i) melhorar o projeto do *software*, (ii) tornar o *software* mais fácil de entender, (iii) ajudar a encontrar falhas e (iv) ajudar a programar mais rapidamente.

As refatorações podem ser executadas a qualquer momento durante a construção e/ou manutenção do software. Para Fowler, (2004) essa é uma atividade que deve ser feita o tempo todo em pequenas rajadas, ou seja, não é uma atividade que separa um tempo para ser fazer, a refatoração é feita por que alguma coisa deve ser feita e aplica-la ajuda a fazer. O mesmo, ainda apresenta três ocasiões de quando se deve refatorar: (i) quando acrescentar funções, (ii) quando precisar acrescentar uma falha e (iii) enquanto revisa o código.

Fowler (2004) e Monteiro e Fernandes (2006) apresentam diversas anomalias de código. Algumas delas são:

Duplicated Code, o número um no *ranking* de anomalias, ocorre quando existe o mesmo código em mais de um lugar. Alguns exemplos comuns de código duplicado é quando tem a mesma expressão em dois métodos da mesma classe, em duas subclasses irmãs ou até em duas ou mais classes não relacionadas.

[1] Nesse documento anomalias de código é utilizado como tradução para *code smells*.

Uma classe candidata à anomalia *God Class* é identificada quando reúne muito conhecimento sobre o sistema. Dessa forma, quando uma classe tenta fazer muita coisa, pode representar um impacto negativo na evolução do sistema como um todo.

A *Shotgun Surgery* é uma anomalia que ocorre quando uma mudança é executada e por causa dessa mudança é preciso fazer muitas alterações em muitas classes diferentes. Quando as alterações estão em muitos lugares são difíceis de encontrar, o que ocasionar no risco de deixar de fazer alguma alteração importante.

A investigação rápida das anomalias de código em um sistema é crucial para que sua remoção seja feita o mais cedo possível. Não remover as anomalias de código pode gerar um grande impacto na arquitetura. Pois, um software difícil de manter ou que apresenta envelhecimento, implica em perda de qualidade ou até descontinuidade do mesmo. A subseção seguinte apresenta em detalhes, a anomalia de código focada nesse estudo com o intuito de encontrar falhas e evitar degradação da arquitetura.

2.3.1 Métodos Longos

Segundo LIU et al. (2009), método longo é um método com muitas declarações, muitas variáveis ou uma lista longa de parâmetros. O aparecimento dessa anomalia no código, muitas vezes, faz com se tenha um impacto significativo na capacidade de manter o *software*.

Bryton, Brito E Abreu e Monteiro (2010) demonstram que a detecção de métodos longos pode ser uma tarefa objetiva, determinista e automática. Para esse fim, é proposto um modelo matemático para detectar métodos longos. Em outro estudo Meananeatra, Rongviriyapanish e Apiwattanapong (2011) tiveram como objetivo selecionar refatorações de métodos longos para melhorar a manutenção do software. Para os mesmos, a dificuldade de leitura e reutilização do código estão entre os principais problemas com métodos longos. Desse modo, definem as condições de refatorações em termos de métricas de *software* com base em gráficos de fluxo de controle e fluxo de dados. Por último, Rongviriyapanish, Karunlanchakorn e Meananeatra (2015) tornam a abordagem para remover métodos longos mais avançada ao propor um algoritmo para substituir automaticamente variáveis temporárias com método de consulta.

Segundo Fowler (2004), nos métodos longos: (i) quanto maior for o procedimento, mais difícil é entende-lo; (ii) toda vez que sentir a necessidade de comentar algo, deve-se escrever um método em vez disso; (iii) a chave não é o tamanho do método, mas a distância

semântica entre o que o método faz e como ele o faz; (iv) olhar os comentários é uma boa técnica para a identificar blocos que necessitam de refatoração; (v) condicionais e *loops* também dão sinais de extrações.

Métodos longos, assim como todas as anomalias apresentadas são estruturas comuns de problemas no código fonte, que indicam a necessidade de se aplicar refatorações em pequenos passos. A próxima seção apresenta algumas técnicas apresentadas na literatura para identificação dessas anomalias.

2.4 Técnicas para Detecção de Anomalias de Código

Diversas técnicas vêm sendo propostas para identificar anomalias introduzidas no código. McCabe (1976) propõe uma técnica de modularização que identifique módulos de *software* que possam se tornar difíceis de testar ou manter, tendo como base o fluxo de controle do programa.

Outros estudos coletam métricas do código desenvolvido e propõem métodos para utilizar os valores obtidos na identificação de possíveis desvios. Munson e Elbaum (1998) apresentam uma técnica que utiliza métricas de complexidade para disponibilizar informações sobre diferentes módulos de um sistema de *software* e assim avaliar a qualidade do *software* no processo de teste.

Alguns autores propõem novas métricas para aumentar a efetividade da detecção de anomalias. Chidamber e Kemerer (1994) desenvolvem e implementam um conjunto de seis novas métricas para projetos OO. Estas métricas são baseadas na teoria de medição e também refletem os pontos de vista de experientes desenvolvedores de *software* OO. Buscando avaliar quantitativamente a qualidade e a produtividade em sistemas OO Abreu e Carapuça (1994) propõem um conjunto de oito métricas conhecidas como *Object Oriented Design Metrics* (MOOD). A proposta feita por Briand, Devanbu e Melo (1997) tenta avaliar e prever a qualidade de produtos de *software* utilizando um conjunto de métricas que investigam o nível de acoplamento entre classes de sistemas OO.

Bansiya e Davis (2002) sugere o uso de onze métricas, definidas como *Quality Model for Object Oriented Design* (QMOOD) para avaliar a qualidade de projetos orientados a objetos (OO). Com ajuda dessas métricas, atributos de qualidade de *software* como reusabilidade, flexibilidade, compreensibilidade, funcionalidade, extensibilidade e efetividade podem ser medidos. Já Gupta, Goyal e Goyal (2015) propõem um modelo para calcular a

qualidade de produtos de *software* orientados a objetos, com base nos fatores densidade de defeitos, complexidade do *software* e esforço da mudança.

Desenvolvedores podem contar com o apoio de ferramentas para auxiliar no processo de detecção de anomalias de código. Segundo (MURPHY-HILL; BLACK, 2008, 2010) as ferramentas de detecção de anomalias são compostas por partes fundamentais: (i) um mecanismo de detecção de anomalias de código que permite aplicar, bem como escolher ou definir, algoritmos de detecção; e (ii) uma interface com o usuário para apresentar as anomalias detectadas.

Várias ferramentas, detalhadas na próxima seção, estão disponíveis para detecção automática de anomalias no código, entre elas os métodos longos.

2.5 Sistemas de Recomendação para Evitar Anomalias de Código

Segundo Robillard, Walker e Zimmermann (2010) Sistemas de Recomendação para Engenharia de Software (SRES) são aplicativos de *software* que apoiam os usuários em sua tomada de decisão ao interagir com grandes espaços de informação. Recomendam itens de interesse com base nas preferências do usuário e apoiam o desenvolvimento do software.

Uma recomendação é valiosa por que o desenvolvedor não estava ciente de sua necessidade ou desconhece os riscos que a recomendação provoca. Dessa forma, SRES abrangem uma alta gama de aspectos de engenharia de software e quantidades exorbitantes de dados de desenvolvimento ajudando desenvolvedores a encontrar informações que eles devem conhecer e avaliar (ROBILLARD; WALKER; ZIMMERMANN, 2010).

Ricci et al. (2011) apresenta vários tipos de sistemas de recomendação que variam na forma como a previsão de uma recomendação é feita, no tipo de recomendação utilizada e como é reunida e apresentada em resposta das solicitações do usuário. O mesmo ainda distingue seis classes diferentes de abordagens de recomendação:

Baseada em conteúdo: O sistema aprende a recomendar itens que são semelhantes aos que o utilizador gostou em um momento anterior. A similaridade dos dados é calculada de com base nas características associadas com os itens comparados. Por exemplo, se um usuário avaliou positivamente um livro que pertence ao gênero aventura, então os sistemas podem aprender a recomendar outros livros do gênero.

Filtragem colaborativa: Considerada a técnica mais popular e amplamente implementada em sistemas de recomendação é muitas vezes e referenciada como “correlação

pessoa a pessoa". A implementação mais simples e original desta abordagem é recomendar itens ao usuário que outros usuários com gostos semelhantes tenham gostado no passado. A semelhança de gostos é calculada com base na similaridade da história de classificação dos utilizadores.

Demográfica: Esse tipo de sistema recomenda itens com base no perfil demográfico do usuário. O pressuposto é que diferentes recomendações devem ser recomendadas para diferentes grupos. Por exemplo, as recomendações podem ser personalizadas com base na idade ou sexo do utilizador. Embora esse tipo de abordagem ser bem bastante popular na literatura, tem havido, relativamente, poucas pesquisas nos sistemas.

Baseada no aprendizado: Sistemas baseados em conhecimento recomendam itens com base no conhecimento de domínio especifico sobre como certas características satisfazem as preferências e necessidades do usuário. Nesses sistemas a função de similaridade estima quanto as necessidades do usuário (descrição do problema) coincidem com as recomendações (solução do problema). A pontuação de semelhança pode ser diretamente interpretada como a utilidade da recomendação pelo usuário. Este tipo de modelo de sistema de recomendação recebe informações sobre as relações sociais dos usuários e as preferências dos amigos do usuário.

Baseado em comunidade: Este tipo de sistema recomenda itens com base nas preferências do usuário. Tem como regra a epigrama "Diga-me quem são seus amigos, que eu lhe direi quem você é.". Essa evidência sugere que as pessoas tendem a confiar mais em recomendações de seus amigos do que em recomendações de indivíduos semelhantes, mas desconhecidos.

Sistemas de recomendação híbridos: Estes sistemas de recomendação baseiam-se na combinação de duas ou mais técnicas mencionadas acima. Um sistema híbrido que combina técnicas A e B tenta usar as vantagens de um para corrigir as desvantagens de outros. Por exemplo, métodos de filtragem colaborativa sofrem de problemas de novos itens, ou seja, eles não podem recomendar itens que não tenham *ranking*. Isso não limita abordagens baseadas em conteúdo uma vez que a previsão para novos itens é baseada em sua descrição e são tipicamente disponíveis. Dessa forma, a abordagem baseada em conteúdo é usada para remover o problema da filtragem colaborativa. Dada duas, ou mais, técnicas básicas de sistemas de recomendações, várias propostas podem ser feitas para combina-las e criar um novo sistema híbrido.

Nosso trabalho é um sistema de recomendação híbrido que utiliza as abordagens baseada no aprendizado e baseada em conteúdo para, respectivamente, aprender sobre os interesses arquiteturais do sistema a partir de um projeto exemplo e verificar quais métodos do projeto em desenvolvimento não estão de acordo com o conteúdo aprendido.

Há diversas generalizações sobre a arquitetura de SRES, mas a grande maioria envolve três funcionalidades principais, como apresentado na Figura 1: (i) um mecanismo de coleta de dados para coletar dados do processo de desenvolvimento e artefatos em um modelo de dados, (ii) um mecanismo de recomendação para analisar os dados e gerar recomendações e (iii) uma interface de usuário para acionar o ciclo de recomendação e apresentar seus resultados (ROBILLARD; WALKER; ZIMMERMANN, 2010).

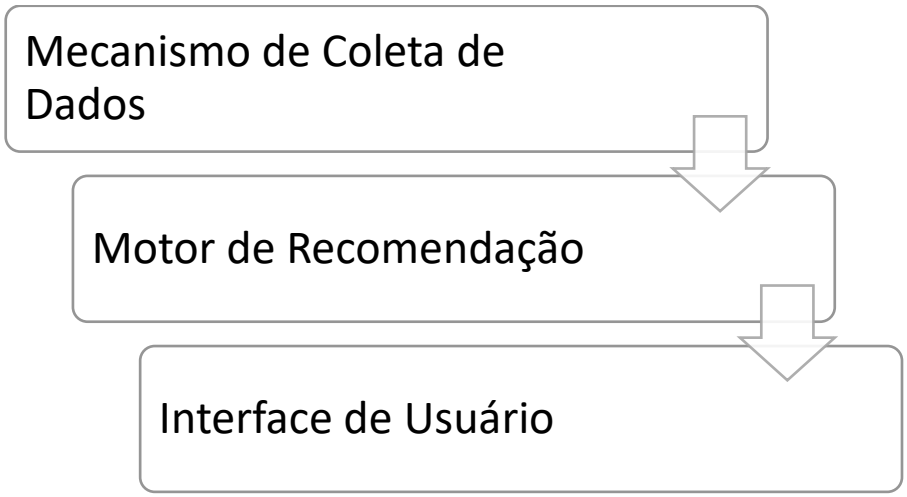

Figura 1: Principais funcionalidades de um Sistema de Recomendação

Em nossa ferramenta de suporte o mecanismo de coleta de dados recupera informações dos interesses arquiteturais de um projeto de exemplo. O motor de recomendação trabalha com os dados do projeto de exemplo e o projeto em desenvolvimento para selecionar métodos longos com base em valores limiares. Por fim, a interface do usuário apresenta através de uma *view* e do editor de texto do ambiente de desenvolvimento Eclipse os métodos longos selecionados pelo motor de recomendação.

Um exemplo de SRES é Seahawk que utiliza o Stack Overflow (2015) como fonte de dados para realizar recomendações. Fornece aos desenvolvedores a capacidade de geração de consultas a partir do código, recuperando discussões, ligando-as ao código, e importando amostras de código. Assim, através de um *Plug-in* para o Eclipse, recupera automaticamente, avalia e sugere discussões do Stack Overflow para o desenvolvedor, se um limite de confiança é ultrapassado (PONZANELLI; BACCHELLI; LANZA, 2013; PONZANELLI, 2014).

Outro exemplo de SRES é o Mentor, uma ferramenta que gerencia solicitações de mudança e faz recomendações de código fonte relacionadas a uma solicitação de alteração. Usa o algoritmo de *Prediction by Partial Matching* (PPM) e algumas heurísticas para analisar um pedido de alteração e os dados dos sistemas de controle de versão, para então recomendar código fonte potencialmente relevante (MALHEIROS et al., 2012).

Outro SRES é o Selene, um sistema de recomendação de código baseado em pesquisa. O processo de recomendação no Selene consiste basicamente em pesquisa associativa e cálculo das pontuações de similaridade local. Primeiro ele envia todo o texto na janela de edição para um servidor de pesquisa. O servidor, em seguida realiza pesquisa associativa sobre o repositório de código e retorna os arquivos mais semelhantes ao texto dado. Logo após, calcula para cada linha de cada arquivo uma pontuação de similaridade local que denota similaridade entre o texto em torno do cursor e o texto em torno da linha do arquivo retornado. Por fim, mostra textos em torno das linhas que tem maiores pontuações de similaridade local (MURAKAMI; MASUHARA, 2012).

Li e Zhou (2005) apresentam um método chamado PR-Miner que utiliza uma técnica de mineração de dados para extrair automaticamente regras gerais de programação com poucos esforços dos programadores. Para extrair regras automaticamente, o PR-Miner tenta encontrar associações entre elementos (Funções, variáveis, tipos de dados) procurando elementos que são frequentemente utilizados em conjunto no código fonte. Suas principais contribuições são a extração de regras de programação e a detecção de violações das regras de programação extraídas.

Finalmente, Holmes e Murphy (2005) constroem a ferramenta Strathcona, um sistema de recomendação que utiliza a estrutura do código em desenvolvimento para encontrar exemplos relevantes em um repositório e então recomendar aos desenvolvedores. A ferramenta é um *plug-in* para o Eclipse. Ao receber pedidos de exemplos extrai o contexto estrutural do código em desenvolvimento e envia para um servidor que abriga o exemplo e então seleciona e devolve para o desenvolvedor exemplos de código com base em heurísticas estruturais.

Algumas dessas ferramentas são *Desktop*, e outras são disponíveis como *plug-ins* do ambiente de desenvolvimento eclipse. Segundo Albuquerque et al. (2014) as ferramentas podem ser classificadas em interativas ou não interativas. As ferramentas interativas (MURPHY-HILL; BLACK, 2008, 2010) permitem ao desenvolvedor manipular o código e, ao mesmo tempo, obter constantemente informações sobre as anomalias de código

relacionadas com a instrução e o método corrente, ou seja, a medida que um método ou instrução é percorrido ou alterado, informações sobre as anomalias referentes ao método ou instrução corrente, são apresentadas ao desenvolvedor. Em contrapartida, as ferramentas não interativas (SIMON; STEINBRUCKNER; LEWERENTZ, 2001) não permitem ao desenvolvedor essa possibilidade de interação. Essas ferramentas têm como prioridade a exibição de uma lista com todas as anomalias encontradas no programa. Ou seja, independentemente de onde o desenvolvedor esteja percorrendo ou alterando métodos e instruções as anomalias contidas na lista são exibidas.

Ao comparar ferramentas para detectar e evitar anomalias de código Fontana, Braione e Zanoni (2012) afirmam que as ferramentas são úteis para avaliar partes do código que necessitam de melhoria, mas não determinam qual delas é a melhor. Entretanto, um problema que existe na utilização das ferramentas é que as análises são realizadas através de valores pré-definidos usados para analisar todas as classes da arquitetura, independente da sua função, gerando um grande número de falsos positivos e falsos negativos. Além disso, boa parte das ferramentas analisa o código-fonte a partir da execução manual do programador. Quando o programador esquece de executar essa análise, os problemas podem acumular e acabar desmotivando sua posterior resolução. Uma possível solução para antecipar decisões dos programadores é o uso de sistemas de recomendação.

SRES avançam no estado da arte em ferramentas de desenvolvimento de software, porém eles têm suas limitações. SRES normalmente recomendam código para olhar, alterar ou reutilizar. No entanto, as recomendações poderiam abordar outros aspectos de desenvolvimento de software como por exemplo recomendações para medidas de qualidade, ferramentas, gerenciamento de projetos e pessoas podiam suportar cada vez mais tarefas de engenharia de software (ROBILLARD; WALKER; ZIMMERMANN, 2010). Nosso método aborda recomendações para medidas de qualidade. Utilizamos a métrica de linhas de código, uma métrica de manutenibilidade que avalia o tamanho.

As abordagens que recomendam ações no código também não consideram as regras de *design* do sistema existente e o contexto arquitetural da classe que está sendo implementada. Vários trabalhos recentes (GUIMARAES; GARCIA; CAI, 2014; MACIA et al., 2013) vêm discutindo a importância de considerar a arquitetura em projetos de *software*. Além disso, nenhum dos SRES estudados preocupa-se em identificar e recomendar refatorações em métodos longos no momento da implementação do código. Deixar a identificação para o final da fase de implementação significa que esses métodos podem nunca

ser refatorados. A seção a seguir explica o processo de construção de um plug-in para o Eclipse. Muitas ferramentas que utilizam técnicas de recomendação para desenvolvedores são criadas como plug-ins do Eclipse.

2.6 Plug-ins do Eclipse como Ferramenta de Recomendação

Ferramentas desenvolvidas para estender as funcionalidades do ambiente integrado de desenvolvimento Eclipse são denominadas *plug-ins*. São grupos estruturados de código e/ou dados que contribuem adicionando funcionalidade ao sistema. A funcionalidade pode ser bibliotecas de código (classes Java com *Application Programming Interface* (API) pública), extensões da plataforma, ou até mesmo a documentação do sistema (ECLIPSE, 2015).

A própria plataforma Eclipse (*Eclipse Platform*) é estruturada como subsistemas que são implementados em um ou mais *plug-ins*. Os subsistemas são construídos em cima de um pequeno motor de tempo de execução. A Figura 2 mostra uma visão da arquitetura do Eclipse com o motor de tempo de execução denominado *Platform Runtime*. Os subsistemas apresentados na Figura 2 são:

Plataform Runtime: Implementado utilizando o modelo de serviços *Open Services Gateway Initiative* (OSGi) implementa o mecanismo de tempo de execução que inicia a base da plataforma e dinamicamente descobre e executa *plug-ins*. Seu objetivo é fazer com que o usuário final não sofra penalidades de desempenho para *plug-ins* que estão instalados, mas não são utilizados. A Figura 2 exibe os principais componentes disponibilizados pela plataforma Eclipse para construção de *plug-ins* e detalhados a seguir:

- *Workspace*: Define um modelo de recurso comum para gerir artefatos de *plug-ins*. Os plug-ins podem criar e modificar projetos, pastas e arquivos para organizar e armazenar artefatos de desenvolvimento em disco.
- *Workbench*: Implementa a interface do usuário e define uma série de pontos de extensão que permitem que outros plug-ins contribuam com ações de menu e barra de ferramentas, operações de arrastar e soltar, diálogos, assistentes e visões e editores personalizados.
- *Standard Widget Toolkit* (SWT): é um kit de ferramentas um baixo nível independente do sistema operacional que suporta a integração de plataformas e API portátil

- *JFace*: É um *framework* de interface do usuário que fornece construções de aplicação em nível superior para diálogos, assistentes, ações, e preferências de usuário.
- *Team*: Permite que outros *plug-ins* possam definir e registrar implementações para a programação da equipe, acesso a repositório, e controle de versão.
- *Help*: Implementa um mecanismo de ajuda integrado a plataforma e um servidor web. Além disso, define pontos de extensão que os plug-ins podem utilizar para ajudar outro plug-in. Outros recursos adicionais são fornecidos para integrar tópicos de ajuda em configurações de documentação nível de produto.
- *Java Development Tools* (JDT): É um *plug-in* que estende o *Workbench* fornecendo recursos especializados para edição, visualização, compilação, depuração e execução de código Java.
- *Plug-in Development Environment* (PDE): este *plug-in* fornece ferramentas que automatizam a criação, manipulação, depuração e implementação de outros *plug-ins*.

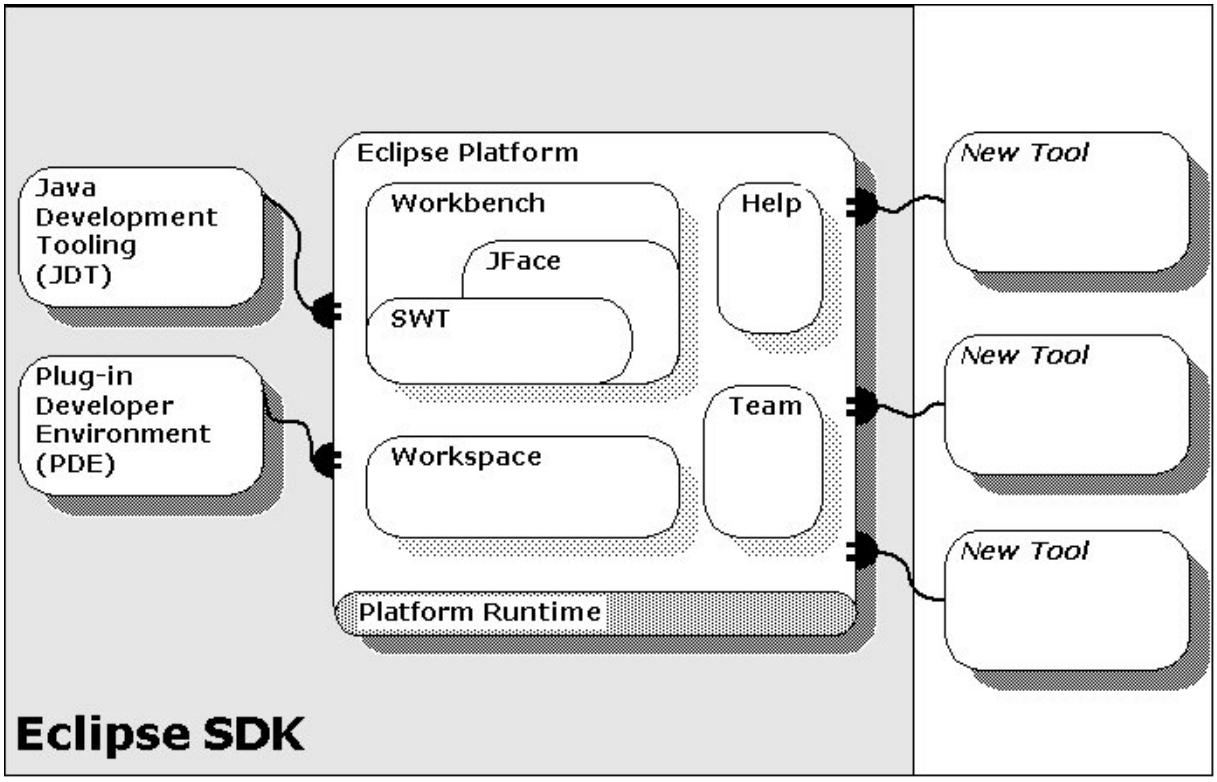

Figura 2: Arquitetura da plataforma de desenvolvimento Eclipse. Fonte: (ECLIPSE, 2015)**.**

Como pode ser observado na Figura 2 o Eclipse SDK inclui a plataforma básica mais dois *plug-ins* importantes que são úteis para o desenvolvimento de plug-in. As ferramentas JDT implementam um ambiente de desenvolvimento Java cheio de recursos. O PDE adiciona ferramentas especializadas que agilizam o desenvolvimento de plug-ins e extensões. Além disso, os três *plug-ins New Tool* ilustram a adição de novas características ao ambiente de desenvolvimento Eclipse.

A seguir são apresentados alguns *plug-ins* explicitando como eles analisam, se comportam e apresentam as anomalias de código. Esses *plug-ins* ilustram a adição de funcionalidades ao eclipse e as ferramentas (*New Tool*) da Figura 2.

SonarLint é um *plug-in* para o eclipse que fornece para os desenvolvedores *feedback* em tempo real sobre novos bugs e problemas de qualidade que são inseridos em projetos Java, JavaScript e PHP. Para utilizar o SonarLint é necessário instalar e habilitar o *plug-in* em seus projetos. A análise do SonarLint é automaticamente acionada quando o arquivo é aberto ou salvo, podendo também ser iniciada manualmente para que todas as classes do projeto sejam analisadas. SonarLint pode avaliar 234 regras em Java, 78 regras em JavaScript e 63 regras em PHP. Após realizar a análise das regras no código são adicionados marcadores nos locais das classes onde existem problemas. Os bugs e problemas de qualidade encontrados no código podem também ser observados em forma de tabela habilitando uma visão (*View*) do *plug-in* no Eclipse.

FindBugs é um programa que utiliza a análise estática para procurar erros em código Java. A ferramenta funciona analisando arquivos de classe compilados, assim não é preciso de código-fonte do programa para usá-lo. Existem diversos *Front-End* para FindBugs. Um desses *Front-End* é um *plug-in* que integra o programa ao Eclipse. A análise do projeto é iniciada de forma manual pelo desenvolvedor e realizada por um verificador de erros que checa a existência de regras no código. Dentre as várias funcionalidades FindBugs permite que programadores escrevam seus próprios padrões de erros. Ao analisar o projeto FindBugs insere marcadores nos locais das classes ondem existem problemas. Duas visões do *plug-in* podem ser destacadas. A primeira visão *Bug Explorer* apresenta uma lista com todos os problemas encontrados na análise feita. A segunda *Bug Info* apresenta uma descrição detalhada de um problema quando é selecionado.

PMD é um analisador de código fonte. É uma ferramenta voltada tanto à verificação de regras de estilo quanto à verificação de erros. Ela encontra falhas comuns de programação como variáveis não utilizadas, trechos de tratamento de exceções vazios, criação de objetos

desnecessários. Além disso, duas análises são destacadas: análise de complexidade ciclomática e a análise feita pelo *copy-paste-detector* (CPD), que detecta código duplicado, mostrando um potencial candidato a refatoração. Do mesmo modo como ocorre no FindBugs, permite que programadores escrevam seus próprios padrões de erros. A análise do código em PMD é iniciada de forma manual e após finalizar apresenta os problemas inserindo marcadores no projeto, pacote e classe e linhas de código da classe onde existem erros, podendo também ser observados através de tabelas das visões *Violations Overview* e *Violations Outiline*.

Checkstyle e uma ferramenta voltada à verificação de regras de estilo. É uma ferramenta de análise com foco no estilo de codificação e não na lógica ou integridade do código. É altamente configurável permitindo customizar as regras de estilo por projeto. Seu conjunto de regras padrão é extenso e seu uso requer uma boa configuração, visto que não é necessário ou desejável seguir todas as regras. CheckStyle pode verificar muitos aspectos de seu código-fonte. Pode-se encontrar problemas de design de classe, problemas de método de projeto. Além disso, também tem a capacidade de verificar problemas de *layout* e formatação do código. No *plug-in* para o eclipse após realizar a análise das regras CheckStyle adiciona marcadores no código para informar quais linhas não estão de acordo com as regras. Por meio de visões os problemas encontrados são apresentados em forma de gráfico e tabela.

UCDetector é um *plug-in* do eclipse para encontrar código desnecessário (Morto) em código Java. Ele também tenta tornar o código *final*, *protected* ou *private*. Além disso, encontra dependências cíclicas entre as classes. É possível executar UCDetector sem interface de usuário chamando Apache Ant. Dessa forma, relatórios HTML podem ser construídos. UCDetector cria marcadores no código para (i) códigos desnecessários, (ii) código onde a visibilidade pode ser alterada para padrão, protegido ou privado e (iii) métodos de campos que podem sem definitivos (*final*). Esses problemas, quando encontrados, aparecem na visão problemas do próprio Eclipse.

JDeodorant é um Eclipse *plug-in* de código aberto para Java, que detecta anomalias de código e resolve os problemas através da aplicação de refatorações. Essa ferramenta identifica cinco anomalias: *Feature Envy* (FE), *Long Method* (LM), *Type Checking*, *God Class* (GC) e *Duplicated Code* e suas técnicas de detecção baseiam-se em oportunidades de refatoração. JDeodorant adiciona uma nova opção na barra de menu com o nome de *Bad Smells*, onde contém cinco opções, uma para cada anomalia citada anteriormente. Ao clicar em determinada opção, ou seja, selecionar qual *bad smell* será analisado, o *plug-in* adiciona

uma nova visão na ferramenta. A análise realizada de forma manual ainda deve ser inicializada. Para tal, é necessário selecionar o projeto que será analisado e clicar no ícone da visão "*Identify Bad Smells*". Após a análise ser realizada as anomalias encontradas serão adicionadas na tabela da visão e marcadores serão adicionados no código. Esses marcadores representam as partes do código que devem, segundo a ferramenta, ser refatorados.

Neste trabalho também foi desenvolvido um *plug-in* do Eclipse com características semelhantes aos citados. Entretanto o *plug-in* construído implementa uma nova proposta para detecção de métodos longos, baseada em informações recuperadas do código fonte existente e da arquitetura do sistema. O Capítulo 3 detalha o funcionamento do método proposto.

3 IDENTIFICAÇÃO SENSÍVEL A ARQUITETURA DE MÉTODOS LONGOS

Esse capítulo tem como objetivo descrever o método sensível à arquitetura que utiliza informações do código fonte existente para identificação de valores limiares utilizados na busca por métodos longos. São avaliadas duas abordagens:

a) Uso de valores limiares calculados a partir de um projeto exemplo com a mesma arquitetura da aplicação que será avaliada.
b) Uso de valores limiares calculados a partir do interesse arquitetural da classe e de um projeto exemplo com a mesma arquitetura da aplicação que será avaliada.

As abordagens disponíveis na literatura sugerem valores limiares a partir de uma média obtida a partir da análise de vários sistemas. Nossa proposta considera apenas um único projeto que possua uma arquitetura semelhante ao avaliado, podendo ser inclusive uma versão anterior do mesmo sistema. Essa proposta tem como objetivo extrair regras de design mais próximas ao sistema que será avaliado. Nesse capítulo detalhamos apenas a segunda abordagem que considera o interesse arquitetural da classe. Essa abordagem é uma evolução da primeira e a única diferença no processo é a consideração do interesse arquitetural da classe para identificação dos valores limiares.

Para dar suporte ao método foi desenvolvido uma ferramenta interativa, pois detecta a anomalia de código enquanto o código é manipulado pelo desenvolvedor e, ao mesmo tempo, obtém constantemente informações sobre as anomalias de código relacionadas com o método corrente, ou seja, a medida que um método é percorrido ou alterado, informações sobre a anomalia do método, são apresentadas ao desenvolvedor.

A ferramenta foi desenvolvida como um *plug-in* para o ambiente de desenvolvimento Eclipse, usando a arquitetura detalhada na Seção 2.5. Para criação do *plug-in* foram usados o PDE e o JDT. O *plug-in* desenvolvido integra-se ao Eclipse exatamente da mesma maneira que outros *plug-ins* desenvolvidos para plataforma.

Com base no conceito de SRES, apresentado na Seção 2.4, podemos classificar ainda a ferramenta desenvolvida como um sistema de recomendação, pois ela aponta aos desenvolvedores métodos longos que necessitam ser refatorados para manter a qualidade do código. Observando os diferentes tipos de sistemas de recomendação apresentados, podemos classificar a nossa ferramenta como um sistema de recomendação híbrido, pois para

recomendar itens de interesse para o usuário, verifica a similaridade dos métodos analisando o seu tamanho e recomenda itens com base no conhecimento criado a partir da análise de um projeto de exemplo.

Na Seção 3.1 é detalhado o método proposto para identificação dos métodos longos e na Seção 3.2 é apresentada a ferramenta desenvolvida para dar suporte à aplicação do método proposto.

3.1 O Método Proposto

O método proposto é dividido em duas etapas. Na primeira etapa o objetivo é definir valores limiares a partir do código-fonte existente para detecção de métodos longos. Para atingir esse objetivo, inicialmente definimos o principal interesse arquitetural de cada classe em um projeto exemplo. Recomenda-se utilizar como projeto exemplo versões do software que possuam *tags* marcadas no sistema de controle de versões. Uma *tag* normalmente é criada para indicar que aquela versão possui certa estabilidade e passou por avaliações de qualidade. Esse projeto exemplo é utilizado para calcular valores limiares para o número de linhas de código em cada interesse arquitetural. Esses valores serão utilizados para análise de classes que possuam o mesmo interesse arquitetural.

Na segunda etapa é realizada a análise das classes em desenvolvimento. Para identificar se a classe possui métodos longos, inicialmente define-se qual o interesse arquitetural da classe em análise. Em seguida, utilizando-se dos valores limiares identificados na primeira etapa, verifica-se se alguns dos métodos implementados ultrapassa o valor estipulado. As subseções seguintes detalham essas duas etapas.

3.1.1 Definição dos Valores Limiares para Interesses Arquiteturais

Esta subseção aborda a primeira etapa do método proposto. A Figura 3 ilustra o processo referente à definição dos interesses arquiteturais do projeto de exemplo e ao cálculo dos valores limiares para cada interesse. A abordagem é dividida em três tarefas (i) identificar principal interesse arquitetural de cada classe do projeto exemplo (ii) agrupar classes de acordo com o interesse arquitetural e (iii) calcular valor limiar de cada interesse arquitetural. A saída desse processo é uma tabela contendo todas os interesses arquiteturais e seus respectivos valores limiares.

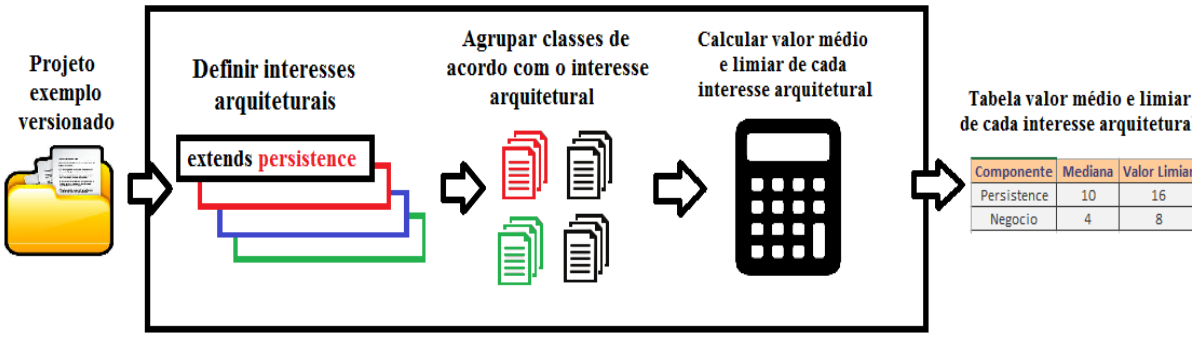

Figura 3: Definição de Valores Limiares a partir de um Projeto Exemplo.

O projeto de exemplo versionado é um projeto que deve representar os interesses arquiteturais do projeto em desenvolvimento. O ideal é que o projeto selecionado seja uma das versões anteriores do projeto que será avaliado. Quanto mais próxima são as versões analisadas em relação ao projeto exemplo usado para extrair os valores limiares, mais precisas são as análises por conta da similaridade do código.

A primeira tarefa do processo consiste em definir o principal interesse arquitetural de cada classe do projeto exemplo. Os interesses arquiteturais são definidos com base na sua hierarquia de implementação obtida através da *Abstract Syntax Tree* (AST). A partir dessa estrutura foi definida algumas regras para agrupar as classes como sendo do mesmo interesse arquitetural. Pertencem ao mesmo interesse arquitetural as classes que atendem aos critérios de acordo com a ordem:

1. Estendem uma mesma classe;
2. Implementam uma mesma interface interna ao projeto;
3. Implementam uma mesma interface externa ao projeto;
4. Não se encaixam em nenhuma das três regras anteriores.

O Algoritmo 1 descreve como as classes são agrupadas de acordo com o principal interesse arquitetural. Ele verifica se a classe analisada pode ser agrupada em alguns dos grupos existentes (invocando o Algoritmo 2). Caso não seja possível um novo grupo de interesses é criado.

Algotimo 1: Agrupamento de Classes por Interesse Arquitetural

---

Entrada: Lista de classes do projeto de exemplo
Saída: Lista de Classes agrupadas por Interesse
1 : Lista_Grupos ← vazio
2 : **para todas** classes **faça**
3 : **para todos** grupos **faça**
4 : adicionou ← adicionaClasseGrupoInteresse(classe, grupo)

```
5 :     fim para
6 :     se não adicionou então
7 :             Novo_Interesse ← classe
8 :             Lista_Grupos.add(Novo_Interesse)
9 :     fim se
10: fim para
```

O Algoritmo 2 é responsável por agrupar as classes com base nas três primeiras regras de agrupamento. Na linha 2 é verificado se a classe se encaixa na primeira regra, ou seja, é verificado se a classe estende uma classe que também é estendida pelo grupo. Na linha 4 é verificado se a classe implementa uma interface interna ao sistema que também é implementada pelo grupo. Na linha 6 é verificada a última das três regras, que consiste em verificar se a classe implementa uma interface externa ao sistema que também é implementada pelo grupo. Caso a classe não se encaixe em nenhuma dessas três regras o Algoritmo 2 retorna à informação que não conseguiu alocar a classe em um dos grupos de interesse existente. A ordem para o agrupamento considera o nível de acoplamento da classe analisada com a arquitetura. A herança é o acoplamento mais forte, seguido pela implementação de uma interface da arquitetura e por fim uma classe externa a arquitetura.

Algoritmo 2: adicionaClasseGrupoInteresse

```
Entrada: classe analisada e grupo
Saída: confirmação se a classe foi adicionada a um dos grupos existentes
1  : adicionou← true
2  : se classe extends a mesma classe das classes do grupo então
3  :     grupo.add(classe)
4  : senão se classe implements mesma interface interna das classes de grupo então
5  :     grupo.add(classe)
6  : senão se classe implements mesma interface externa das classes de grupo então
7  :     grupo.add(classe)
8  : senão
9  :     adicionou← false
10: fim se
```

Classes que não se encaixam em nenhuma das três primeiras regras irão formar grupos com apenas uma classe. Após o término da execução do Algoritmo 1 esses grupos são unidos para formar classes do interesse arquitetural da regra 4, ou seja, o interesse arquitetural de classes que não se encaixam em nenhuma das três primeiras regras.

A quantidade de grupos que cada regra pode gerar vai variar de acordo com a arquitetura analisada. As três primeiras regras podem formar nenhum ou vários grupos de interesses arquiteturais, já a regra 4 pode formar nenhum ou um único grupo.

A terceira tarefa no processo ilustrado na Figura 3, consiste em calcular os valores médios e limiares de cada interesse arquitetural. O Algoritmo 3 descreve como esse processo é realizado.

Algoritmo 3: Calcular Mediana e limiar de cada interesse arquitetural

---

Entrada: Lista de Classes agrupadas por Interesse
Saída: Valores da mediana e valor limiar de cada Grupo de Interesse
1 : **para todos** interesses arquiteturais **faça**
2 : calcularMediana(interesse arquitetural)
3 : calcularValorLimiar(interesse arquitetural)
4 : **fim para**

No Algoritmo 3 as linhas 2 e 3 calculam respectivamente, a mediana e o valor limiar de linhas de código em cada interesse arquitetural identificado. Para calcular esses valores as quantidades de linhas de código de todos os métodos das classes do interesse arquitetural são obtidas através da AST e logo após ordenados de forma crescente.

O valor médio do interesse arquitetural é definido pelo cálculo da Mediana (Md). A mediana é uma quantidade que procura caracterizar o centro da distribuição de frequências. Ela é calculada com base na ordem dos valores que formam o conjunto de dados. Dada uma distribuição de frequências, e supondo-se os valores da variável dispostos em ordem crescente ou decrescente de magnitude, há dois casos a considerar:

- 1º. A variável em estudo tem n (tamanho amostral) ímpar. Neste caso a mediana será o valor da variável que ocupa a posição de ordem $\frac{n+1}{2}$.
- 2º. A variável tem n (tamanho amostral) par. Neste caso, a mediana é o valor somado que ocupam as posições $\frac{n}{2}$ e $\frac{n+2}{2}$ dividido por 2.

O valor limiar é definido pelo cálculo do percentil. Um percentil indica que há x% de dados inferiores, ou seja, os percentis dividem o conjunto de dados em cem partes iguais. Há, portanto, noventa e nove percentis. Dada uma distribuição de frequências, e supondo-se os valores da variável estão dispostos em ordem crescente de magnitude, é calculado o valor da posição do percentil com a equação $Pp = \frac{k}{100} * n$, onde Pp é a posição do percentil na distribuição de frequências, k é o número que representa o percentil (por exemplo se quero

encontrar o percentil setenta e cinco k é igual a 75) e n é a quantidade de dados na distribuição. Com o valor da posição encontrada, dois casos devem ser considerados:

- 1º. Pp não é um número inteiro. A convenção usada é arredondar para a posição do número inteiro acima do posto e tomar o valor correspondente.
- 2º. Pp é um número inteiro. O valor a ser usado é o da média aritmética entre os dados que ocupam as posições Pp e $Pp + 1$.

O cálculo dos valores médios e limiares dos interesses arquiteturais finaliza as tarefas da primeira parte do método proposto. A saída da execução dessas tarefas é uma tabela com o valor médio e limiar para os métodos de cada interesse arquitetural como apresenta a Figura 3. A tabela na Figura 3 é uma simplificação da tabela de saída. Para melhor ilustrar a Tabela 1 exemplifica um resultado real da execução dos algoritmos descritos.

Tabela 1: Tabela detalhada com valor médio e limiar dos métodos de cada interesse arquitetural

extendsClass	implementsInternal	implementsExternal	Mediana	Valor Limiar
Persistence	Null	Null	4	8
Null	INegocio	Null	12	17
Null	Null	Serializable	2	3
Null	IController	Null	7	10
Null	Null	Null	16	22

A Tabela 1 apresenta as colunas que identificam um interesse arquitetural (colunas extendsClass, implementsInternal e implementsExternal) e as colunas que compõem a mediana (coluna Mediana) e valor limiar (coluna Valor Limiar) da quantidade de linhas dos métodos desses interesses. Nela são exibidos cinco interesses arquiteturais identificados e explicados a seguir:

- A linha 1 da tabela representa um interesse arquitetural que estende a classe Persistence, tem valor limiar 8 (o número máximo de linhas para os métodos das classes desse interesse arquitetural é de no máximo 8 linhas), valor médio de 4 linhas (o número médio de linhas do interesse é de 4 linhas) e foi gerado a partir da regra 1.

- As linhas 2 e 4 representam os interesses arquiteturais que implementam, respectivamente, as classes da arquitetura INegocio e IController, possuem como valores limiares 17 e 10 linhas de código, o número médio de linhas desses componentes são de 12 e 7 linhas e foram geradas a partir da regra 2.
- A linha 3 representa um interesse arquitetural que implementa a classe Serializable da API Java, tem valor limiar de 3 linhas de código, valor médio de 2 linhas e foi gerado a partir da regra 3.
- Por fim, a linha 5 representa um interesse arquitetural da regra 4, ou seja, o interesse arquitetural que não se encaixa em nenhuma das três primeiras regras, esse interesse pode ou não estender ou implementar outras classes, porém sempre será representada por valores *Null* nas colunas extendsClass, implementsInternal e implementsExternal, possui valor limiar de 22 linhas de código e o valor médio de linhas é de 16.

A construção dessa tabela serve para auxiliar o processo de análise dos projetos em desenvolvimento, que corresponde a segunda parte do método proposto explicado na próxima subseção.

3.1.2 Identificação de Métodos Longos com base no Interesse Arquitetural da Classe

Esta subseção aborda a segunda etapa do método proposto que visa identificar os métodos longos baseando-se em informações dos interesses arquiteturais do código existente. A Figura 4 apresenta todo o processo referente à definição do interesse arquitetural das classes do projeto em desenvolvimento e identificação dos métodos longos. Para entrada do processo é utilizada a tabela que contém o valor médio e limiar de cada interesse arquitetural. Essa tabela é gerada a partir do processo apresentado na Seção 3.1.1. Essa etapa é dividida em três tarefas (i) identificar interesse arquitetural da classe, (ii) buscar na tabela valor médio e limiar do interesse arquitetural e (iii) identificar métodos longos de acordo com os valores encontrados na tarefa anterior. A saída desse processo é uma lista contendo todos os métodos do projeto em desenvolvimento.

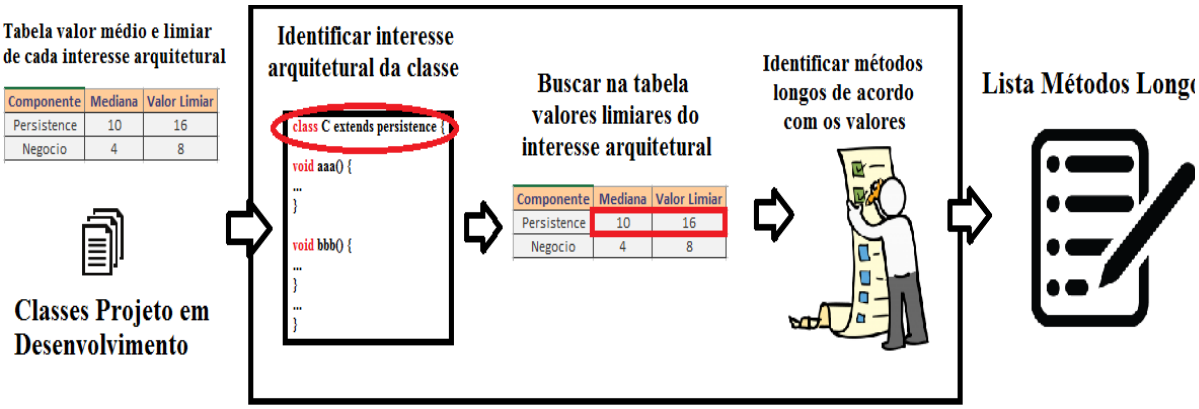

Figura 4: Tarefas referentes a definição dos interesses arquiteturais do projeto em desenvolvimento e identificação de métodos longos.

A primeira tarefa do processo consiste em identificar o interesse arquitetural da classe. O Algoritmo 4 descreve como ocorre essa identificação.

Algoritmo 4: Identificar Valor Médio e Limiar para Análise da Classe

---

Entrada: lista de classes e tabela de interesses arquiteturais
Saída: interesse arquitetural das classes
1 : **para todas** classes **faça**
2 : **se** classe *extends* mesma classe do interesse arquitetural **então**
3 : classe ← interesse arquitetural
4 : **senão se** classe *implements* mesma interface interna do interesse arquitetural **então**
5 : classe ← interesse arquitetural
6 : **senão se** classe *implements* mesma interface externa do interesse arquitetural
7 : **então**
8 : classe ← interesse arquitetural
9 : **senão**
10: classe ← interesse arquitetural formada pelas classes que não se encaixam nas três primeiras regras
11: **fim se**
12: **fim para**

Como pode ser observado no Algoritmo3 identificar o interesse arquitetural da classe consiste em definir a regra em que as classes do projeto em desenvolvimento se encaixam. No algoritmo as linhas 2, 4 e 6 verificam se o interesse da classe pertence respectivamente a regra 1, regra 2 e regra 3. Se nenhuma dessas condições forem atendidas a classe pertence ao interesse arquitetural da regra 4.

A segunda tarefa do processo utiliza as informações de identificação para buscar na tabela o valor limiar correspondente do interesse arquitetural da classe. Por exemplo, se uma "classe A" pertence a regra 1 são obtidos os interesses arquiteturais da tabela que pertencem a regra 1. Dessa forma, é verificado qual dos interesses arquiteturais selecionados estende a

mesma classe da “classe A”. O interesse que estende a mesma classe estendida pela “Classe A” tem seus valores limiares obtidos.

Com os valores limiares do interesse da classe obtidos a terceira tarefa é iniciada. Essa tarefa consiste em identificar os métodos longos da classe de acordo com os valores limiares do interesse arquitetural da classe.

Algoritmo 5 Identificação de métodos longos

```
Entrada: Lista de classes com valores limiares
Saída: Lista de métodos longos
1  : para todas classes faça
2  :     ObterValoresLimiaresPorInteresse(classe)
3  :     para todos métodos faça
4  :         se quantidadeLinhasMetodo > valorLimiarDoInteresse então
5  :             metodosLongos.add(método)
6  :         fim se
7  :     fim para
8  : fim para
```

Como pode ser observado o Algoritmo 5 executa a tarefa de identificação de métodos longos. Para tal é verificado se o método analisado possui um número de linhas de código maior que o valor limiar do interesse arquitetural da classe. Os métodos com uma quantidade de linha maiores que o valor limiar do interesse arquitetural da classe são adicionados a uma lista de métodos longos.

A lista de métodos corresponde a saída do processo da segunda parte do método proposto. Essa lista é usada para apresentar aos desenvolvedores quais métodos necessitam sofrer alterações para melhorar a qualidade do código.

3.2 Ferramenta de Suporte

Nesta seção é apresentada a ferramenta, denominada de Architecture-Sensitive Smells Detector, desenvolvida para dar suporte ao método proposto. A ferramenta é composta por quatro telas principais.

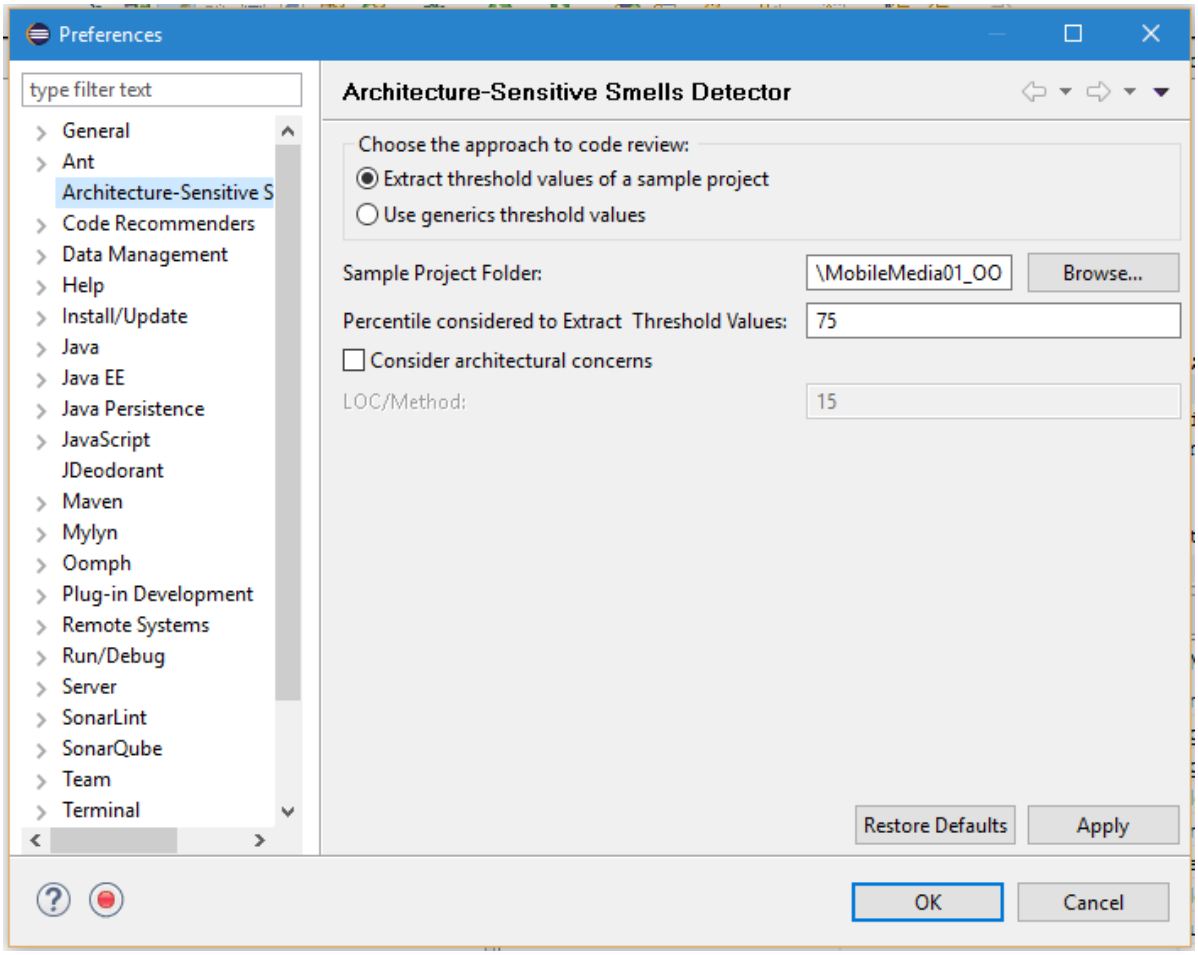

Figura 5: Tela de preferências do *plug-in*

Na Figura 5 é apresentada a tela de preferências do *plug-in* desenvolvido. Nessa tela é definido o tipo de análise realizada. A ferramenta executa três tipos de análises diferentes: (i) extrair valores limiares de um projeto exemplo considerando os interesses arquiteturais, (ii) extrair valores limiares de um projeto de exemplo sem considerar os interesses arquiteturais, e (iii) usar valor limiar genérico. Essas variações foram criadas para facilitar a validação dos resultados obtidos com a ferramenta.

A opção *Extract threshold values of a sample project* engloba duas das três análises descritas. Quando essa opção é selecionada os campos *Sample Project Folder*, *Percentile considered to Extract Threshold Values* e *Consider architectural concerns* são habilitados. Esses campos definem, respectivamente, o projeto usado como exemplo, o valor percentil da distribuição de valores que deve ser considerado para extração do valor limiar e se na análise do projeto de exemplo os interesses arquiteturais devem ser considerados. O projeto de exemplo é um projeto que deve representar os interesses arquiteturais do projeto que será avaliado (uma versão anterior do projeto pode ser utilizada).

Utilizar a opção *Extract threshold values of a sample project* e marcar o campo *Consider architectural concerns* corresponde a utilização da técnica proposta na Seção 3.1. Desse modo, os valores utilizados por essa opção definem os interesses arquiteturais do projeto, os valores limiares e mediana do número de linhas de código em cada interesse arquitetural.

Quando a opção *Use generics threshold values* é selecionada, todos os campos com exceção do LOC/Method são desabilitados. No campo LOC/Method é necessário definir um valor inteiro em número de linhas. Esse valor denota a quantidade máxima de linhas de código que um método deve possuir para não ser considerado método longo.

Por padrão é utilizado na ferramenta a opção Valor limiar, com seu valor definido como dez linhas de código. Selecionando a opção de Projeto de exemplo é utilizado por padrão um valor de percentil de setenta e cinco, porém para executar esse tipo de análise é necessário definir o projeto de exemplo.

Problems | Javadoc | Declaration | Code smells

Directory	Class	Method	Initial Line	Nº of Lines	Type
C:\Users\Kekeu\Downloads\mobilemedia\MobileMedia01_...	ImageUtil	readImageAsByteArray	39	33	Long Method
C:\Users\Kekeu\Downloads\mobilemedia\MobileMedia01_...	ImageUtil	getImageInfoFromBytes	101	32	Long Method
C:\Users\Kekeu\Downloads\mobilemedia\MobileMedia01_...	ImageUtil	getBytesFromImageInfo	161	20	Long Method
C:\Users\Kekeu\Downloads\mobilemedia\MobileMedia01_...	ImageAccessor	loadAlbums	74	30	Long Method
C:\Users\Kekeu\Downloads\mobilemedia\MobileMedia01_...	ImageAccessor	resetImageRecordStore	131	30	Long Method
C:\Users\Kekeu\Downloads\mobilemedia\MobileMedia01_...	ImageAccessor	addImageData	192	21	Long Method
C:\Users\Kekeu\Downloads\mobilemedia\MobileMedia01_...	ImageAccessor	loadImageDataFromRMS	236	24	Long Method
C:\Users\Kekeu\Downloads\mobilemedia\MobileMedia01_...	ImageAccessor	updateImageInfo	283	25	Long Method
C:\Users\Kekeu\Downloads\mobilemedia\MobileMedia01_...	ImageAccessor	deleteSingleImageFromRMS	396	17	Long Method
C:\Users\Kekeu\Downloads\mobilemedia\MobileMedia01_...	ImageAccessor	createNewPhotoAlbum	435	21	Long Method
C:\Users\Kekeu\Downloads\mobilemedia\MobileMedia01_...	AlbumData	getImageNames	70	13	Long Method
C:\Users\Kekeu\Downloads\mobilemedia\MobileMedia01_...	AlbumData	getImageFromRecordStore	106	13	Long Method
C:\Users\Kekeu\Downloads\mobilemedia\MobileMedia01_...	BaseController	init	84	14	Long Method

Figura 6: Visão para apresentação dos métodos longos encontrados

A Figura 6 exibe a visão *Code smells* disponibilizada pela ferramenta desenvolvida. Essa visão exibe em uma tabela dados detalhados sobre os métodos longos que foram encontrados na análise do projeto em desenvolvimento. A primeira coluna da tabela é a *Directory*, que exibe o diretório onde a classe está localizada, a segunda coluna denominada *Class*, mostra o nome da classe onde existem métodos longos, a terceira coluna é chamada de *Method*, e retrata o nome do método que foi considerado como longo, a quarta coluna *Initial Line* denota o número da linha onde o método longo é iniciado, a quinta coluna *Nº of Lines* apresenta a quantidade de linhas de código do método longo, por fim a última coluna *Type* apresenta o tipo de anomalia encontrada.

Além de apresentar os métodos longos e alguns detalhes sobre eles, essa tabela é usada para direcionar o desenvolvedor para o local do código onde o método longo é encontrado. Para isso, basta clicar duas vezes sobre a linha da tabela do método que deseja abrir. Após o duplo clique, a classe é aberta no editor de texto do Eclipse e o cursor é direcionado para a linha inicial do método. A visão não é aberta diretamente e pode ser exibida através do caminho *Window -> Show View -> "Other..."* do ambiente eclipse.

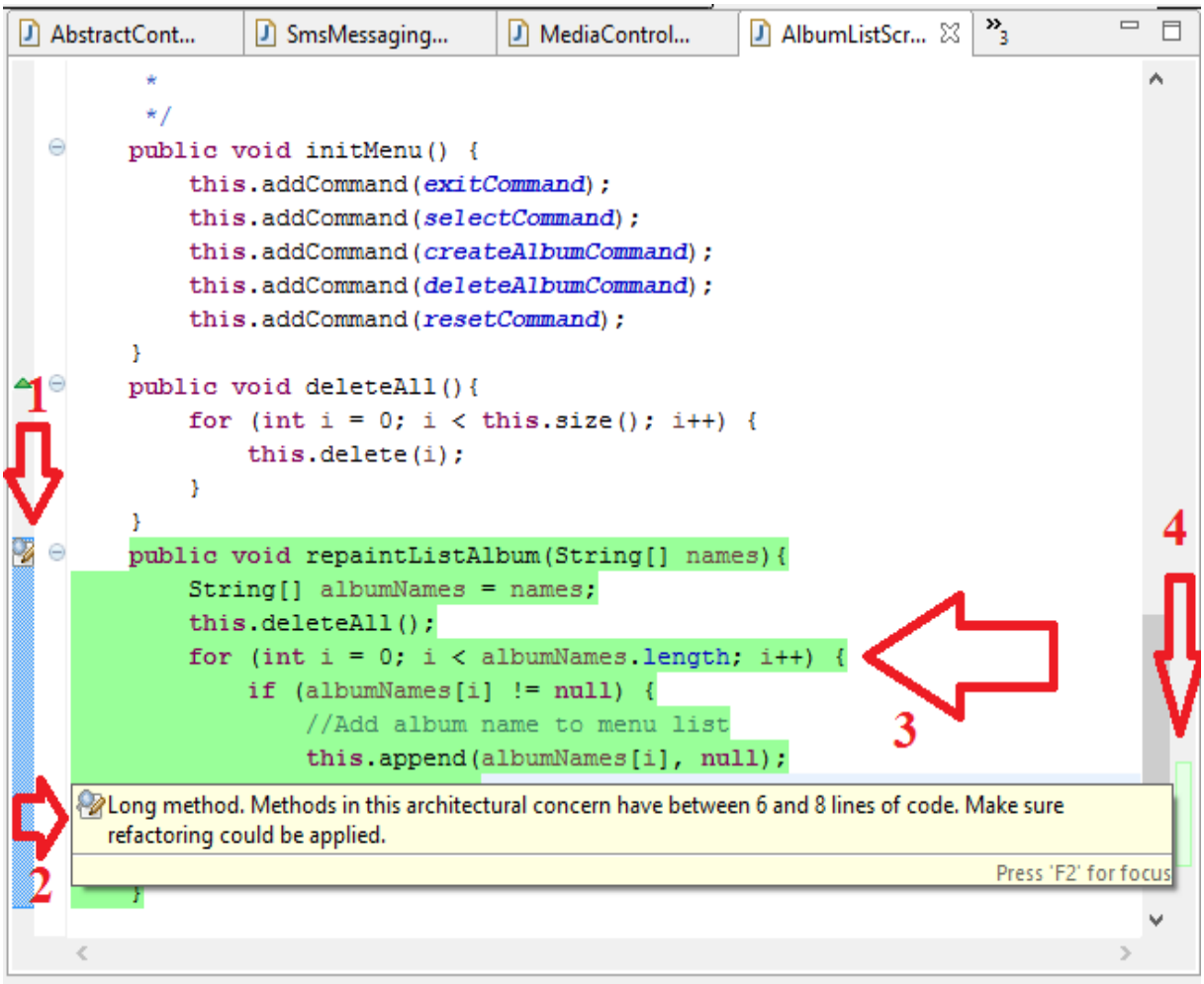

Figura 7: Editor de texto com marcadores e decoradores destacando um método longo.

Na Figura 7 temos o editor de texto do Eclipse com a classe AlbumListScreen.java. O método repaintListAlbum(String[] names) representa um método longo. Para destacar esses métodos dos demais, quatro componentes, indicados pelas setas 1, 2, 3 e 4, são adicionados nos locais dos métodos da classe.

A seta 1 destaca os marcadores (*Markers*) que são adicionados nas classes que contém métodos longos. Para cada método longo encontrado é adicionado um marcador ao lado da linha inicial do método. Esse marcador contém informações sobre onde o método é

iniciado e finalizado e a mensagem que é dada ao desenvolvedor. A mensagem é a mesma indicada pela seta 2 e pode ser visualizada colocando o cursor do *mouse* sobre o marcador.

Destacada pela seta 2, temos a mensagem dada ao desenvolvedor com detalhes sobre o método longo. Como pode ser observado, a mensagem informa a quantidade de linhas que os métodos do componente têm. Além disso, complementa informando que deve ser realizado refatoração.

Utilizado para destacar ainda mais o método longo, a nota (*Annotation*) indicada pela seta 3 altera a aparência do método, pintando o fundo com uma cor diferente. Além disso, a mensagem destacada pela seta 2 é apresentada ao colocar o cursor do *mouse* sobre alguma parte da nota.

Por fim, na Figura 7 temos a indicação feita pela seta 4. Essa é uma nota complemento da indicada pela seta 3. Utilizada ao lado da barra de rolagem do editor de texto, tem como funcionalidade direcionar o *scroll* para exibir a sua nota corresponde. Ademais, exibe a mensagem indicada pela seta 2, visualizada ao adicionar o cursor do *mouse* sobre ela.

A Figura 8 apresenta o *MenuBarPath*, um menu que é exibido ao clicar com o botão direito do *mouse* sobre projetos do *Package Explorer*. Nesse menu foram adicionadas duas novas opções. A primeira opção *Remove project of the analysis - ASSD*, tem como funcionalidade remover o projeto selecionado da análise feita pela ferramenta. Já a segunda opção *Analyze project - ASSD*, tem como funcionalidade adicionar o projeto selecionado para ser analisado pela ferramenta. Como pode ser observado na Figura 8, a primeira opção está inativa, isso quer dizer que o projeto selecionado ainda não está em análise, portanto não pode ser removido. O mesmo pode ocorrer com a segunda opção, retratando que o projeto já está sendo analisado pela ferramenta e pode apenas ser removido.

A ferramenta foi desenvolvida com o intuito de mostrar de forma clara e direta as anomalias encontradas pela análise enquanto o desenvolvedor trabalha no desenvolvimento do software. Para iniciar a recomendação é necessário adicionar o projeto para ser analisado através da opção *Analyze project – ASSD*. Após adicionar o projeto para ser analisado a ferramenta recomenda que um método pode está se tornando longo no momento que o desenvolvedor salva as alterações realizadas no código.

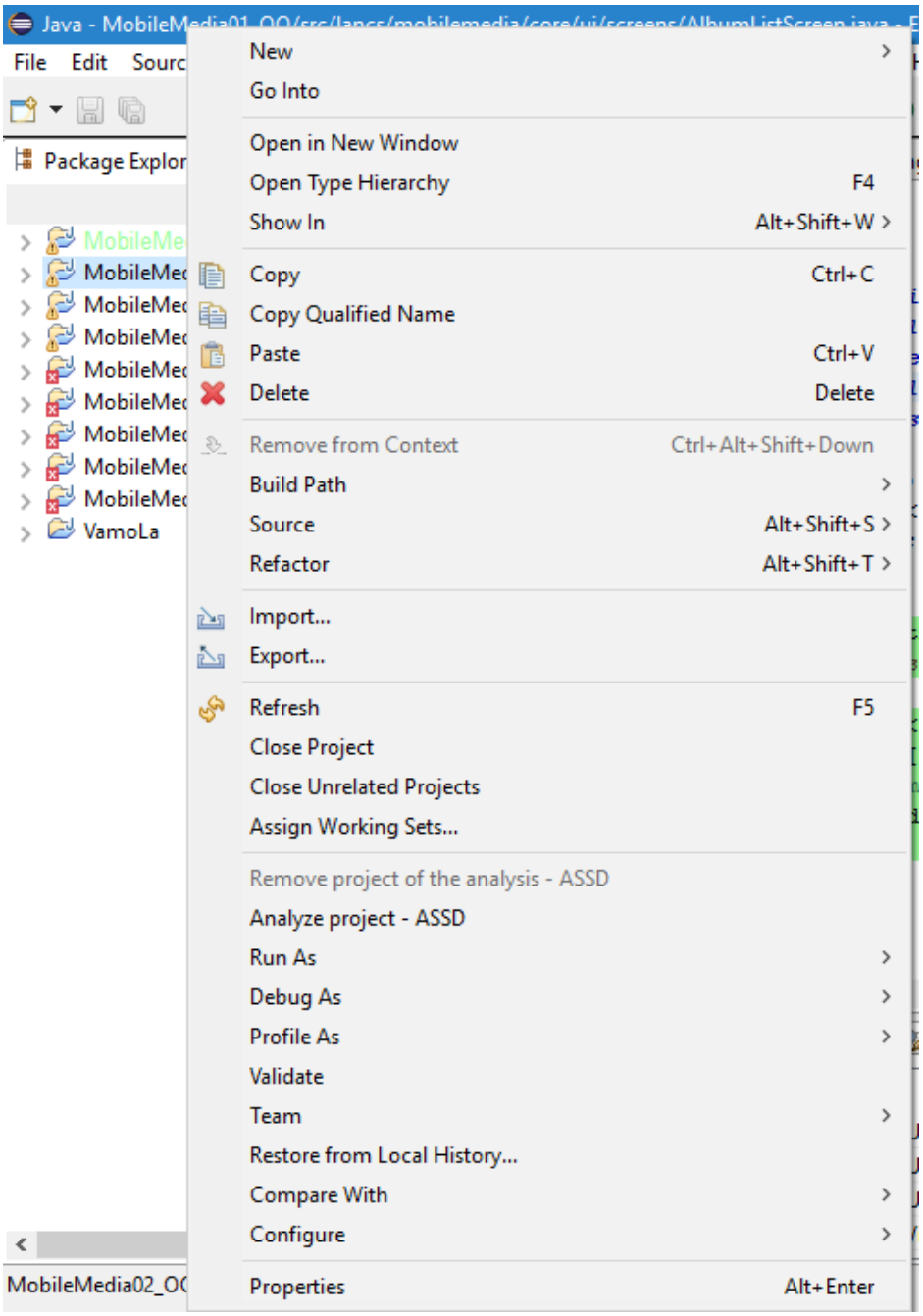

Figura 8: Adição de opções remover e adicionar projetos para análise no *MenuBarPath*.

4 AVALIAÇÃO

Há um número crescente de ferramentas de análise de software disponíveis para a detecção de anomalias de código e, em geral, uma crescente consciência de engenheiros de software sobre a qualidade estrutural de recursos em desenvolvimento (PAIVA et al., 2006). A questão é como avaliar ferramentas para definir o quão eficiente ela pode ser em uma determinada situação.

Fontana, Braione e Zanoni (2012) apresentam alguns desafios para avaliar ferramentas para detecção de anomalias. Em primeiro lugar, devido à ambiguidade e, às vezes, imprecisão de suas definições, há diferentes interpretações para cada anomalia de código. Em segundo lugar, as diferentes técnicas de detecção utilizadas pelas ferramentas, geralmente são baseadas no cálculo de um determinado conjunto de métricas, variando entre métricas orientadas a objetos padrão e métricas definidas de forma ad hoc (LANZA; MARINESCU, 2006). Por fim, mesmo utilizando as mesmas métricas, diferentes valores limiares podem ser utilizados, isso ocorre devido a utilização de diferentes fatores como o domínio e tamanho do sistema, práticas organizacionais e a compreensão de programadores e engenheiros de software responsáveis pela definição desses valores limiares.

Esses fatores dificultam a comparação de resultados gerados por diferentes técnicas. A dificuldade reside não só nas diferentes interpretações das anomalias de código, mas também em sua identificação manual. Portanto, é difícil encontrar sistemas de código aberto com listas validadas de anomalias de código para gerar uma análise mais aprofundada (PAIVA et al., 2006).

É necessário ter resultados de referência em Engenharia de Software e foi nesse contexto que foi utilizado o resultado de (PAIVA et al., 2006) para avaliação do método proposto. Foi calculada a p*recision* e r*ecall* na identificação de métodos longos relevantes em diferentes versões do sistema de software MobileMedia (FIGUEIREDO; CACHO, 2008), uma linha de produtos de software (SPL) para aplicações que manipulam foto, música e vídeo em dispositivos móveis.

Apesar dos autores disponibilizarem resultados referência para *God Methods* (GM), esses resultados também podem ser usados para *Long Methods* (LM), já que todo *God Method* também é um *Long Method*. A diferença é que além de avaliar a métrica de *Line of Code* (LOC) algumas abordagens também utilizam informações das métricas *Number of*

Parameters (NOP), *Number of Local Variables* (NOLV) e o *Maximum Number of Branches* (MNOB).

Tabela 2: Informações do sistema MobileMedia.

Dados de MobileMedia				***Code Smell* na Lista de Referência**
Versão	**Nº de Classes**	**Nº de Métodos**	**LOC**	**GM**
V1	24	124	1159	9
V2	25	143	1316	7
V3	25	143	1364	6
V4	30	161	1559	8
V5	37	202	2056	8
V6	46	238	2511	9
V7	50	269	3015	7
V8	50	273	3167	7
V9	55	290	3216	6

Nosso estudo envolveu nove versões orientadas a objetos (V1 a V9) do MobileMedia. A quantidade de linhas de código desse sistema varia de 1 KLOC na primeira versão (V1) para pouco mais de 3 KLOC na última versão (V9). A Tabela 2 mostra para cada versão o número de classes, métodos e linhas de código de MobileMedia. Além disso, exibe o número total de *God Method* de cada versão do sistema segundo avaliação de especialistas.

A Tabela 3 mostra o número de anomalias de código identificadas pelas análises realizadas, em cada uma das nove versões do MobileMedia. Foram realizadas cinco análises distintas. As duas primeiras colunas apresentam os resultados das duas análises que consideraram os interesses arquiteturais da classe analisada. A primeira utiliza o percentil 75 e a segunda o percentil 90 para definir o valor limiar a ser considerado na identificação dos métodos longos. As outras três colunas apresentam resultados das análises que não consideram os interesses arquiteturais. A terceira e quarta coluna apresentam os resultados com percentil 75 e 90 respectivamente. A última coluna apresenta o número de métodos longos identificados quando utilizado o valor limiar de 45 linhas de código. Na análise dos resultados da Tabela 3 verificamos que:

Tabela 3: Número de *Long Methods* detectadas nas análises realizadas.

	Considerando Interesse Arquitetural		Desconsiderando Interesses Arquiteturais		
Versão	**Percentil 75**	**Percentil 90**	**Percentil 75**	**Percentil 90**	**45 LOC**
V1	16	5	19	7	1
V2	16	5	20	8	1
V3	17	7	20	7	1
V4	20	9	22	11	3
V5	26	13	30	16	4
V6	32	19	36	19	5
V7	36	21	42	23	6
V8	37	21	43	23	6
V9	36	19	44	20	6

A cada versão do *software*, com exceção da nona, a quantidade de métodos longos encontrados foi aumentando. Percebe-se que na tabela referência criada por especialistas isso não aconteceu. A evolução das regras de *design* do código certamente influenciou na decisão dos especialistas. Percebe-se que alguns métodos classificados como problemáticos nas primeiras versões, mesmo sem sofrer alterações, acabaram não sendo considerados problemáticos nas versões seguintes.

- Aumentar o valor do percentil de 75 para 90 levou as duas primeiras versões (V1 e V2) a encontrar uma quantidade de anomalias menor que a quantidade que consta na lista de referência. Esse ponto indica que aumentar o valor do percentil pode fazer com que anomalias relevantes sejam desconsideradas.
- A análise realizada com um valor fixo de 45 linhas de códigos retornou uma baixa quantidade de métodos longos. Comparando com a quantidade de métodos da lista de referência, apenas na versão V9 chegou a encontrar a mesma quantidade de métodos. Isso implica na perda de anomalias relevantes recomendadas ao desenvolvedor.

Para avaliar a eficácia da ferramenta em detectar métodos longos relevantes calculou-se a *precision* e *recall* do método proposto. O cálculo dos valores de *Recall* e

Precision tem como base a Tabela 4. Uma anomalia de código relevante é uma anomalia que também está presente na lista referência de anomalias de código.

Tabela 4: *Recall* e *Precision* como modelo de avaliação. Fonte: (ZIMMERMANN; PREMRAJ; ZELLER, 2007)**.**

		Defeitos Observados	
		True	*False*
Defeitos Previstos	*Positive*	*True Positive* (TP)	*False Positive* (FP)
	Negative	*False Negative* (TN)	*True Negative* (TN)

➡ ***Precision***

⬇ ***Recall***

Recall é a razão entre o número de registos relevantes recuperados e o número total de registos relevantes analisados pela técnica. A fórmula desse cálculo é denotada por $Recall = \frac{TP}{TP+FP}$.

Precision é a razão entre o número de registos relevantes recuperados e o número total de registos relevantes e irrelevantes recuperados. A fórmula desse cálculo é denotada por $Precision = \frac{TP}{TP+FN}$.

A Tabela 5 exibe os valores de *recall* e *precision* considerando todas as versões do sistema MobileMedia. Cinco análises foram realizadas em cada uma das nove versões do sistema. As quatro primeiras extraem o valor limiar da primeira versão (V1) do Mobile Media. As duas primeiras análises consideram os interesses arquiteturais da classe do sistema. As duas seguintes não consideraram o interesse arquitetural na definição de valores limiares. Duas análises usaram como valor limiar o percentil 75 da lista ordenada de LOC/Método, outras duas usaram como valor limiar o percentil 90. A quinta e última análise considerou um valor limiar fixo de 45 linhas de código. Esse valor limiar foi obtido por Fontana et al. (2015) através da análise de 74 sistemas de *software*.

As duas análises realizadas com valor limiar no percentil 75 obtiveram taxa de *recall* de 100%, indicando que todas as anomalias identificadas pelos especialistas foram encontradas pela técnica proposta. Entretanto, em todas as análises realizadas com esse percentil a abordagem que considera os interesses arquiteturais da classe sempre obteve maior *precision.* Considerando todas as versões a precisão quando considerada o interesse

arquitetural foi de 32% enquanto a que não considerava foi de 27%. Essas taxas de *precision* indicam que alguns falsos positivos ainda foram gerados. Consequentemente, as recomendações geradas pela técnica proposta ainda exigiriam um esforço de validação dos desenvolvedores. Entretanto, esse esforço é recompensado por saber que todos os métodos apontados são possíveis candidatos.

Ainda considerando o percentil 75 para identificação do valor limiar, as versões que obtiveram maior taxa de *precision* foram as primeiras versões. Por exemplo, na análise da primeira versão (V1) considerando e desconsiderando os interesses arquiteturais foram alcançadas taxas de *precision* de 56% e 47% respectivamente. Nas versões seguintes essas taxas foram sempre reduzindo. Nossa hipótese é que por termos utilizado sempre a primeira versão (V1) do Mobile Media para analisar as demais versões, as regras de *design* do código dessa versão acabaram se distanciando das versões seguintes. Percebe-se que a última versão analisada (V9) foi a que obteve a menor taxa de *precision*. Entretanto outros estudos seriam necessários para confirmar essa hipótese.

Na análise das nove versões considerando o percentil 90 da lista de valores de LOC/método do projeto exemplo, podemos observar o incremento da *precision* em todas as versões. As versões três, quatro e cinco da análise que considera os interesses arquiteturais, por exemplo, dobrou o valor da *precision* em relação as análises realizadas com o percentil 75. Entretanto ao observar a taxa de *recall* das duas primeiras versões, ela sofreu redução obtendo 44% na primeira versão (V1) e 57% na segunda (V2) quando os interesses arquiteturais são considerados. Quando desconsideramos os interesses foram obtidos 67% na primeira versão (V1) e 86% na segunda (V2). A diminuição do *recall* indica que anomalias de código identificadas por especialistas foram desconsideradas, ou seja, cresceu o número de falsos positivos que são mais difíceis de serem identificados por uma equipe de desenvolvimento.

Tabela 5: Análise da recomendação da técnica proposta e considerando e não considerando os interesses arquiteturais nas versões do sistema MobileMedia.

	Considerando Interesse Arquitetural				Desconsiderando Interesses Arquiteturais					
	Percentil 75		Percentil 90		Percentil 75		Percentil 90		45 LOC	
Versão	**Recall**	**Precision**	**Recall**	**Precisão**	**Recall**	**Precision**	**Recall**	**Precision**	**Recall**	**Precision**
V1	100%	56%	44%	80%	100%	47%	67%	86%	11%	100%
V2	100%	44%	57%	80%	100%	35%	86%	75%	14%	100%
V3	100%	35%	100%	86%	100%	30%	100%	86%	17%	100%
V4	100%	40%	100%	89%	100%	36%	100%	73%	38%	100%
V5	100%	31%	100%	62%	100%	27%	100%	50%	50%	100%
V6	100%	28%	100%	47%	100%	25%	100%	47%	56%	100%
V7	100%	19%	100%	33%	100%	17%	100%	30%	71%	83%
V8	100%	19%	100%	33%	100%	16%	100%	30%	86%	100%
V9	100%	17%	100%	32%	100%	14%	100%	30%	83%	83%
Total	**100%**	**32%**	**89%**	**60%**	**100%**	**27%**	**95%**	**56%**	**47%**	**96%**

Observando o sistema como um todo, considerando o percentil 75 foi obtida uma taxa média de 100% de *recall* e 32% de *precision* quando os interesses arquiteturais são considerados. Desconsiderando os interesses foi obtido 100% de *recall* e 27% de *precision*. Utilizando o percentil 90 e os interesses arquiteturais a taxa de *precision* aumentou de 32% para 60%, porém a taxa de *recall* diminuiu de 100% para 89%. Desconsiderando os interesses arquiteturais a taxa de *precision* aumentou de 27% para 56% e a taxa de recall diminuiu de 100% para 95%. O aumento da *precision* reduz consideravelmente o esforço de validação do programador, mas a redução do recall acarreta na perda de anomalias de código relevantes. Percebe-se que o valor do percentil selecionado tem grande influência nos resultados de *recall* e *precision*. Pretende-se fazer novos estudos para escolha do percentil mais adequado a ser utilizado. Considerando apenas os estudos realizados, recomendamos utilizar um valor próximo a 75 do percentil para aumentar o *recalll* e consequentemente diminuir o risco de gerar falsos positivos.

Finalmente na análise realizada com o valor limiar fixado em 45 linhas de código retornou, de modo geral, uma taxa de *recall* de 47% e uma taxa de *precision* de 96%. Comparando essa análise com a técnica proposta utilizando o percentil 75 observa-se que a taxa de *precision* aumentou de 32% para 96%, mas a taxa de recall diminuiu de 100% para 47%. O aumento da taxa de *precision* pode ser explicado pela baixa quantidade de métodos retornados, sendo que esses métodos são relevantes. Porém, a baixa quantidade de métodos retornados acarretou a diminuição do *recall*. Diversos anomalias de código identificadas pelos especialistas foram perdidas nessa análise. Por exemplo, as versões V1, V2 e V3 retornaram apenas um único método longo, para as mesmas versões a lista de referência contém, respectivamente, nove, seis e sete *God Methods*.

Nosso método apesar de recomendar métodos longos com uma taxa de *precision* inferior ao tipo de análise que considera valores fixos, possuiu uma taxa de *recall* de 100% em todos os estudos realizados quando é utilizada o percentil 75, retornando dessa forma, todas as anomalias da lista de referência.

Paiva et al. (2006), utiliza em seu trabalho esses mesmos resultados referência para medir a taxa de *recall* e *precision* de três ferramentas inFusion, JDeodorant e PMD na identificação de *God Methods* (GM). A Tabela 6 exibe os resultados médios obtidos na avaliação das mesmas nove versões do sistema MobileMedia.

Tabela 6: Resultados das análises nas ferramentas inFusion, JDeodorant e PMD. Fonte: (PAIVA et al., 2006)**.**

	Recall			Precision		
Code Smell	inFusion	JDeodorant	PMD	inFusion	JDeodorant	PMD
God Class	9%	58%	17%	33%	28%	78%
God Method	26%	50%	26%	100%	35%	100%
Feature Envy	0%	48%	-	0%	13%	-

Comparando a média de *recall* e *precision*, da anomalia GM, das três ferramentas com as médias encontradas pela análise da técnica proposta neste trabalho observa-se que todas as três ferramentas obtiveram uma média de *recall* inferior à nossa proposta. A maior média de *recall* foi encontrada no JDeodorant com um valor de 50%. O menor valor encontrado em nossa análise foi de 89% usando um valor percentil 90. Para a média de *precision* foi obtido o menor valor encontrado de 33% ao usar o percentil 75. Esse valor é inferior à média encontrada pelas ferramentas inFusion, JDeodorant e PMD que tem, respectivamente, média de *precision* igual a 100%, 35% e 100%. Entretanto as ferramentas também identificaram alguns poucos métodos que existiam na lista referência (PAIVA et al., 2006). Dessa forma, apesar de menor *precision* nossa abordagem retorna todos os métodos da lista de referência.

Algumas ferramentas utilizam também informações de outras métricas de código. Nossa proposta utiliza apenas a métrica LOC com valores limiares extraídos de um projeto com arquitetura semelhante para auxiliar a descoberta de métodos longos. Considerando que os especialistas definiram os GM manualmente e com base na sua experiência isso pode levar a erros e influenciar nos resultados da *precision* do método proposto. Por exemplo, na análise da primeira versão do sistema MobileMedia (V1) dois métodos com vinte e uma linhas, um deles exibido na Figuras 9, foram marcados corretamente como longos por nossa abordagem quando consideramos a lista referência dos especialistas. Entretanto, um método com vinte e duas linhas na mesma classe, exibido na Figura 10, não foi considerado *God Method*. Consideramos que esse método também deveria ser marcado como *God Method* pelos especialistas.

```
    /**
     * Show the current image that is selected
     */
    public void showImage() {
        List selected = (List) display.getCurrent();
        String name = selected.getString(selected.getSelectedIndex());
        Image storedImage = null;
        try {
            storedImage = model.getImageFromRecordStore(currentStoreName, name);
        } catch (ImageNotFoundException e) {
            Alert alert = new Alert( "Error", "The selected photo was not found in the mobile device",
                    null, AlertType.ERROR);
            Display.getDisplay(midlet).setCurrent(alert, Display.getDisplay(midlet).getCurrent());
            return;
        } catch (PersistenceMechanismException e) {
            Alert alert = new Alert( "Error", "The mobile database can open this photo", null,
                    AlertType.ERROR);
            Display.getDisplay(midlet).setCurrent(alert, Display.getDisplay(midlet).getCurrent());
            return;
        }

        //We can pass in the image directly here, or just the name/model pair and have it loaded
        PhotoViewScreen canv = new PhotoViewScreen(storedImage);
        canv.setCommandListener(this);
        setCurrentScreen(canv);
    }
```

Figura 9: Exemplo de método longo com 21 linhas identificado como *God Method*.

```
    /**
     * Show the list of images in the record store
     * TODO: Refactor - Move this to ImageList class
     */
    public void showImageList(String recordName) {
        if (recordName == null)
            recordName = currentStoreName;

        PhotoListScreen imageList = new PhotoListScreen();
        //Command selectCommand = new Command("Open", Command.ITEM, 1);
        imageList.initMenu();
        imageList.setCommandListener(this);

        String[] labels = null;
        try {
            labels = model.getImageNames(recordName);
        } catch (UnavailablePhotoAlbumException e) {
            Alert alert = new Alert( "Error", "The list of photos can not be recovered",
                    null, AlertType.ERROR);
            Display.getDisplay(midlet).setCurrent(alert, Display.getDisplay(midlet).getCurrent());
            return;
        }
        //loop through array and add labels to list
        for (int i = 0; i < labels.length; i++) {
            if (labels[i] != null) {
                //Add album name to menu list
                imageList.append(labels[i], null);

            }
        }
        setCurrentScreen(imageList);
        //currentMenu = "list";
    }
```

Figura 10: Exemplo de método longo com 22 linhas não identificado como *God Method*.

Vale ressaltar que a lista referência foi construída manualmente, utilizando o conhecimento de dois especialistas. Isso aumenta a possibilidade de falhas na definição da lista referência. Uma sugestão seria incrementar o número de especialistas que definem a essa listagem. Entretanto, como discutido no início desse capitulo, a construção de listas de referências para avaliar sistemas de recomendação ainda é pouco difundida na área de engenharia de software. Todos os resultados das avaliações realizadas estão disponíveis na Web para acesso público.[2]

[2] Toda a analise realizada pode ser encontrada em http://tinyurl.com/h4n8vsk.

5 CONCLUSÃO

Neste trabalho foi apresentada uma nova abordagem para identificar e recomendar refatorações em métodos longos considerando o contexto arquitetural. Muitos trabalhos abordam questões relacionadas a identificação, recomendação e utilização do contexto para apresentar anomalias para o desenvolvedor, porém não foram encontrados trabalhos que tenham o mesmo foco proposto neste trabalho.

Para atingir o objetivo o método proposto extrai valores limiares utilizando informações de um projeto exemplo que possua uma arquitetura semelhante àquela em avaliação. Adicionalmente o método permite identificar valores limiares considerando também o interesse arquitetural de cada classe. A abordagem tentar identificar valores limiares mais próximos da arquitetura em avaliação, diferente das abordagens propostas na literatura que usam valores limiares extraídos da análise de sistemas com arquiteturas distintas.

Implementamos uma ferramenta para recomendar refatorações. Para o desenvolvimento da ferramenta foi observado a arquitetura do ambiente de desenvolvimento Eclipse. Dessa forma, o *plug-in* desenvolvido permite utilizar a técnica proposta para apresentar e recomendar aos desenvolvedores métodos longos enquanto trabalham no desenvolvimento do software.

A avaliação da ferramenta foi realizada através da análise de nove versões do sistema de software MobileMedia. Calculamos para cada versão do software a média de *recall* e *precision* e comparamos os resultados com uma lista referência criada por especialistas. Apesar da menor precisão em relação as abordagens existentes, nossa proposta obteve alta taxa de recall, significando que apesar dos falsos positivos, também geramos poucos falsos negativos que são mais difíceis de identificar pelos desenvolvedores.

Como trabalhos futuros sugerimos: (i) utilizar outras métricas, além da LOC, na identificação de métodos longos; (ii) avaliar a técnica proposta em outros sistemas de *software*; (iii) realizar estudos para verificar se remover anomalias enquanto o sistema é desenvolvido diminui a quantidade de anomalias detectadas em novas versões do software; e (iv) automatizar o processo de identificação do percentil a ser utilizado nas análises.

REFERÊNCIAS

ABREU, F. B.; CARAPUÇA, R. **Object-Oriented Software Engineering : Measuring and Controlling the Development Process**. Disponível em: <http://ctp.di.fct.unl.pt/QUASAR/Resources/Papers/Metrics/4ICSQ.pdf>. Acesso em: 20 out. 2015.

ALBUQUERQUE, D. et al. **Detecção Interativa de Anomalias de Código: Um Estudo Experimental**. Wmod2014. **Anais**...2014Disponível em: <http://www.les.inf.puc-rio.br/opus/docs/pubs/2014/WMOD2014-Albuquerque.pdf>

ARCOVERDE, R. et al. **Automatically detecting architecturally-relevant code anomalies**. 2012 Third International Workshop on Recommendation Systems for Software Engineering (RSSE). **Anais**...IEEE, jun. 2012Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6233419>. Acesso em: 28 mar. 2015

BANSIYA, J.; DAVIS, C. G. A hierarchical model for object-oriented design quality assessment. **IEEE Transactions on Software Engineering**, v. 28, n. 1, p. 4–17, 2002.

BASS, L.; CLEMENTS, P.; KAZMAN, R. **Software Architecture in Practice**. 2ª Edition ed. Boston: Addison-Wesley, 2003.

BOSCH, J. Design and use of software architectures: adopting and evolving a product-line approach. 1 jun. 2000.

BRIAND, L. C.; DEVANBU, P.; MELO, W. **An investigation into coupling measures for C++Proceedings of the 19th international conference on Software engineering - ICSE '97**, 1997.

BRYTON, S.; BRITO E ABREU, F.; MONTEIRO, M. **Reducing Subjectivity in Code Smells Detection: Experimenting with the Long Method**. 2010 Seventh International Conference on the Quality of Information and Communications Technology. **Anais**...IEEE, set. 2010Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5655669>. Acesso em: 20 out. 2015

checkstyle – Checkstyle 6.11.2. Disponível em: <http://checkstyle.sourceforge.net/>. Acesso em: 18 nov. 2015.

CHIDAMBER, S. R.; KEMERER, C. F. Metrics suite for object oriented design. **IEEE Transactions on Software Engineering**, v. 20, n. 6, p. 476–493, 1994.
ECLIPSE. **Mars Eclipse**. Disponível em: <https://www.eclipse.org/>. Acesso em: 21 nov. 2015.

FIGUEIREDO, E.; CACHO, N. **Evolving software product lines with aspects**. Proceedings of the 13th international conference on Software engineering - ICSE '08. **Anais**...New York,

New York, USA: ACM Press, 2008Disponível em: <http://portal.acm.org/citation.cfm?doid=1368088.1368124>. Acesso em: 24 jul. 2015

FindBugs™ - Find Bugs in Java Programs. Disponível em: <http://findbugs.sourceforge.net/>. Acesso em: 18 nov. 2015.

FONTANA, F. A. et al. Automatic metric thresholds derivation for code smell detection. **International Workshop on Emerging Trends in Software Metrics, WETSoM**, v. 2015-Augus, p. 44–53, 2015.

FONTANA, F. A.; BRAIONE, P.; ZANONI, M. Automatic detection of bad smells in code: An experimental assessment. **Journal of Object Technology**, v. 11, n. 2, p. 1–38, 2012.

FOWLER, M. **Refatoração: Aperfeiçoando o Projeto de Código Existente**. Porto Alegre: Bookman, 2004.

GARCIA, J. et al. Enhancing architectural recovery using concerns. **2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE 2011, Proceedings**, p. 552–555, 2011.

GUIMARAES, E.; GARCIA, A.; CAI, Y. Exploring Blueprints on the Prioritization of Architecturally Relevant Code Anomalies -- A Controlled Experiment. **2014 IEEE 38th Annual Computer Software and Applications Conference**, p. 344–353, 2014.

GUPTA, N.; GOYAL, D.; GOYAL, M. **A hierarchical model for object-oriented design quality assessmentInternational Journal of Advances in Engineering Sciences**, 28 jul. 2015. Disponível em: <http://journals.rgsociety.org/index.php/ijse/article/view/723>. Acesso em: 20 out. 2015

HOLMES, R.; MURPHY, G. C. **Using structural context to recommend source code examples**. Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005. **Anais**...IEEe, 2005Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1553554>. Acesso em: 23 out. 2015

JDeodorant | Eclipse Plugins, Bundles and Products - Eclipse Marketplace. Disponível em: <https://marketplace.eclipse.org/content/jdeodorant>. Acesso em: 18 nov. 2015.

KOSCIANSKI, A.; DOS SANTOS SOARES, M. **Qualidade de Sotware**. 2ª Edição ed. [s.l.] Editora Novatec, 2007.

LANZA, M.; MARINESCU, R. **Object-oriented metrics in practice: Using software metrics to characterize, evaluate, and improve the design of object-oriented systems**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006.

LI, W.; SHATNAWI, R. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. **Journal of Systems and Software**, v. 80, n. 7, p. 1120–1128, jul. 2007.

LI, Z.; ZHOU, Y. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. **ACM SIGSOFT Software Engineering Notes**, v. 1, n. 5, p. 306–315, 2005.

LIU, H. et al. **Facilitating software refactoring with appropriate resolution order of bad smells**. Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering on European software engineering conference and foundations of software engineering symposium - E. **Anais**...New York, New York, USA: ACM Press, 24 ago. 2009Disponível em: <http://dl.acm.org/citation.cfm?id=1595696.1595738>. Acesso em: 5 nov. 2015

MACIA, I. et al. Enhancing the Detection of Code Anomalies with Architecture-Sensitive Strategies. **2013 17th European Conference on Software Maintenance and Reengineering**, p. 177–186, 2013.

MALHEIROS, Y. et al. **A Source Code Recommender System to Support Newcomers**. 2012 IEEE 36th Annual Computer Software and Applications Conference. **Anais**...IEEE, jul. 2012Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6340250>. Acesso em: 15 abr. 2015

MCCABE, T. J. A Complexity Measure. **IEEE Transactions on Software Engineering**, v. SE-2, n. 4, p. 308–320, dez. 1976.

MEANANEATRA, P.; RONGVIRIYAPANISH, S.; APIWATTANAPONG, T. **Using software metrics to select refactoring for long method bad smell**. The 8th Electrical Engineering/ Electronics, Computer, Telecommunications and Information Technology (ECTI) Association of Thailand - Conference 2011. **Anais**...IEEE, maio 2011Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5947882>. Acesso em: 8 out. 2015

MUNSON, J. C.; ELBAUM, S. G. **Code churn: a measure for estimating the impact of code change**. Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272). **Anais**...IEEE Comput. Soc, 1998Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=738486>. Acesso em: 20 out. 2015

MURAKAMI, N.; MASUHARA, H. **Optimizing a search-based code recommendation system**. 2012 Third International Workshop on Recommendation Systems for Software Engineering (RSSE). **Anais**...IEEE, jun. 2012Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6233414>. Acesso em: 17 abr. 2015

MURPHY-HILL, E.; BLACK, A. P. **Seven habits of a highly effective smell detector**. Proceedings of the 2008 international workshop on Recommendation systems for software engineering - RSSE '08. **Anais**...New York, New York, USA: ACM Press, 10 nov. 2008Disponível em: <http://dl.acm.org/citation.cfm?id=1454247.1454261>. Acesso em: 30 abr. 2016

MURPHY-HILL, E.; BLACK, A. P. **An interactive ambient visualization for code smells**. Proceedings of the 5th international symposium on Software visualization - SOFTVIS '10. **Anais**...New York, New York, USA: ACM Press, 25 out. 2010Disponível em: <http://dl.acm.org/citation.cfm?id=1879211.1879216>. Acesso em: 30 abr. 2016

P. MONTEIRO, M.; M. FERNANDES, J. Towards a catalogue of refactorings and code smells for AspectJ. **Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)**, v. 3880 LNCS, p. 214–258, 2006.

PAIVA, T. et al. **Experimental Evaluation of Code Smell Detection Tools**, 2006.

PMD. Disponível em: <http://pmd.github.io/>. Acesso em: 18 nov. 2015.

PONZANELLI, L. **Holistic recommender systems for software engineering**. Companion Proceedings of the 36th International Conference on Software Engineering - ICSE Companion 2014. **Anais**...New York, New York, USA: ACM Press, 2014Disponível em: <http://www.scopus.com/inward/record.url?eid=2-s2.0-84903554827&partnerID=tZOtx3y1>. Acesso em: 17 abr. 2015

PONZANELLI, L.; BACCHELLI, A.; LANZA, M. **Seahawk: Stack overflow in the IDE**. Disponível em: <http://2013.icse-conferences.org/content/seahawk-stack-overflow-ide.html>. Acesso em: 28 maio. 2015.

RADJENOVIĆ, D. et al. Software fault prediction metrics: A systematic literature review. **Information and Software Technology**, v. 55, n. 8, p. 1397–1418, 2013.

RICCI, F. et al. (EDS.). **Recommender Systems Handbook**. Boston, MA: Springer US, 2011.

ROBILLARD, M.; WALKER, R.; ZIMMERMANN, T. Recommendation Systems for Software Engineering. **IEEE Software**, v. 27, n. 4, p. 80–86, jul. 2010.

RONGVIRIYAPANISH, S.; KARUNLANCHAKORN, N.; MEANANEATRA, P. **Automatic code locations identification for replacing temporary variable with query method**. 2015 12th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON). **Anais**...IEEE, jun. 2015Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7207086>. Acesso em: 20 out. 2015

S. PRESSMAN, R. **Engenharia de Software Uma Abordagem Profissional**. 7ª Edição ed. Porto Alegre: Bookman, 2011.

SECURITY, S. **Stack Overflow**. Disponível em: <http://stackoverflow.com/>. Acesso em: 5 nov. 2015.

SILVA, D.; TERRA, R.; VALENTE, M. T. JExtract: An Eclipse Plug-in for Recommending Automated Extract Method Refactorings. 19 jun. 2015.

SIMON, F.; STEINBRUCKNER, F.; LEWERENTZ, C. **Metrics based refactoring**. Proceedings Fifth European Conference on Software Maintenance and Reengineering. **Anais**...IEEE Comput. Soc, 2001Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=914965>. Acesso em: 30 abr. 2016

SOMMERVILLE, I. **Engenharia de Software**. 9ª edição ed. [s.l: s.n.].

SonarLint for Eclipse. Disponível em: <http://www.sonarlint.org/eclipse/index.html>. Acesso em: 14 abr. 2016.

TAKUYA, W.; MASUHARA, H. A spontaneous code recommendation tool based on associative search. **Proceedings of the 3rd international workshop on Search-driven development: users, infrastructure, tools, and evaluation - SUITE '11**, n. Suite, p. 17–20, 2011.

TERRA, R.; VALENTE, M.; BIGONHA, R. DCLfix: A recommendation system for repairing architectural violations. **arXiv preprint arXiv:**, 2015.

THIOLLENT, M. **Metodologia da Pesuisa-ação**. 7º edição ed. [s.l.] Editora São Paulo: Cortez, 1996.

UCDetector. Disponível em: <http://www.ucdetector.org/>. Acesso em: 14 abr. 2016.
Welcome to JDeodorant. Disponível em: <http://users.encs.concordia.ca/~nikolaos/jdeodorant/>. Acesso em: 14 abr. 2016.

ZIMMERMANN, T.; PREMRAJ, R.; ZELLER, A. Predicting defects for eclipse. **Proceedings - ICSE 2007 Workshops: Third International Workshop on Predictor Models in Software Engineering, PROMISE'07**, 2007.