
Universidade Federal de Campina Grande

Centro de Engenharia Elétrica e Informática

Coordenação de Pós-Graduação em Ciência da Computação

Danyllo Wagner Albuquerque

# Avaliação Experimental da Detecção Interativa de Anomalias de Código

Campina Grande - PB

2023

Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

# Avaliação Experimental da Detecção Interativa de Anomalias de Código

Danyllo Wagner Albuquerque

Tese submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande como parte dos requisitos necessários para obtenção do grau de Doutor em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Engenharia de Software

Hyggo Oliveira de Almeida
(Orientador)

Campina Grande, Paraíba, Brasil

A345a Albuquerque, Danyllo Wagner.
Avaliação experimental da detecção interativa de anomalias de código / Danyllo Wagner Albuquerque. - Campina Grande, 2023.
228 f. : il. color.

Tese (Doutorado em Ciência da Computação) - Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2023.
"Orientação: Prof. Dr. Hyggo Oliveira de Almeida"
Referências.

1. Engenharia de Software. 2. Anomalias de Código. 3. Detecção Interativa. 4. Avaliação Empírica. 5. Qualidade de Software. 6. Code Smells. 7. Interactive Detection. 8. Empirical Evaluation. 9. Software Quality. I. Almeida, Hyggo Oliveira de. II. Título.

CDU 004.41(043)

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECÁRIA ITAPUANA SOARES DIAS GONÇALVES CRB-15/093

MINISTÉRIO DA EDUCAÇÃO
**UNIVERSIDADE FEDERAL DE CAMPINA GRANDE**
POS-GRADUACAO EM CIENCIA DA COMPUTACAO
Rua Aprígio Veloso, 882, Edifício Telmo Silva de Araújo, Bloco CG1, - Bairro Universitário, Campina Grande/PB, CEP 58429-900
Telefone: 2101-1122 - (83) 2101-1123 - (83) 2101-1124
Site: http://computacao.ufcg.edu.br - E-mail: secretaria-copin@computacao.ufcg.edu.br / copin@copin.ufcg.edu.br

## FOLHA DE ASSINATURA PARA TESES E DISSERTAÇÕES

**DANYLLO WAGNER ALBUQUERQUE**

**AVALIAÇÃO EXPERIMENTAL DA DETECÇÃO INTERATIVA DE ANOMALIAS DE CÓDIGO**

Tese apresentada ao Programa de Pós-Graduação em Ciência da Computação como pré-requisito para obtenção do título de Doutor em Ciência da Computação.

Aprovada em: 31/07/2023

Prof. Dr. HYGGO OLIVEIRA DE ALMEIDA, UFCG, Orientador

Prof. Dr. DANILO FREIRE DE SOUZA SANTOS, UFCG, Examinador Interno

Prof. Dr. EVANDRO DE BARROS COSTA, UFAL, Examinador Interno

Prof. Dr. LENARDO CHAVES E SILVA, UFERSA, Examinador Externo

Prof. Dr. ROBERTO FELICIO DE OLIVEIRA, UEG, Examinador Externo

---

Documento assinado eletronicamente por **HYGGO OLIVEIRA DE ALMEIDA**, **PROFESSOR(A) DO MAGISTERIO SUPERIOR**, em 31/07/2023, às 11:28, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **DANILO FREIRE DE SOUZA SANTOS**, **PROFESSOR(A) DO MAGISTERIO SUPERIOR**, em 31/07/2023, às 12:17, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **LENARDO CHAVES E SILVA**, **Usuário Externo**, em 31/07/2023, às 13:07, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **Roberto Felício de Oliveira**, **Usuário Externo**, em 31/07/2023, às 19:43, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **Evandro de Barros Costa**, **Usuário Externo**, em 03/08/2023, às 08:02, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

A autenticidade deste documento pode ser conferida no site https://sei.ufcg.edu.br/autenticidade, informando o código verificador **3636116** e o código CRC **DE779B1B**.

# Resumo

Anomalias de código são estruturas que frequentemente indicam a presença de problemas no software, dificultando sua manutenção e evolução. Existem várias anomalias catalogadas na literatura e sua detecção geralmente é feita por meio de abordagens de Detecção Não-Interativa (DNI). Essas abordagens não oferecem suporte à interação progressiva dos desenvolvedores com o código afetado, revelando ocorrências de anomalias mais globais apenas sob demanda do desenvolvedor, implicando recorrentemente na identificação tardia destas anomalias. Com o surgimento da Detecção Interativa (DI), busca-se lidar com as limitações das abordagens tradicionais, permitindo a revelação de instâncias de anomalias de código sem uma solicitação explícita do desenvolvedor, incentivando a detecção precoce como uma prática recomendada. Embora os desenvolvedores considerem o uso de abordagens DI, a literatura não oferece diretrizes claras sobre quando e como essas abordagens devem ser utilizadas no contexto das atividades de desenvolvimento de software. Neste trabalho, tem-se como objetivo a avaliação experimental da abordagem de detecção interativa de anomalias de código no contexto das atividades do processo de desenvolvimento de software. Para isso, foram conduzidos estudos no intuito de identificar a necessidade de uso de tal abordagem, avaliar métodos de detecção aderentes, desenvolver suporte automatizado com características DI, apresentar evidências empíricas sobre sua eficácia na detecção de anomalias, bem como descrever um modo de integração ao processo de desenvolvimento de acordo com o arcabouço do *Scrum*. As tarefas experimentais revelaram que o uso da DI resultou em um aumento de até 40% na medida *recall* e de até 25% na medida *precision* na detecção de anomalias durante as atividades de inspeção e desenvolvimento de código. Com base nos resultados dos estudos, conclui-se que fatores associados à DI contribuíram para detecção antecipada de um maior número de ocorrências de anomalias de código se comparada com abordagens tradicionais. Consequentemente, a utilização disciplinada da abordagem DI em um processo de desenvolvimento pode promover uma avaliação contínua e melhorar a qualidade do software.

# Abstract

Code smells generally indicate the presence of deeper problems in the software, making its maintenance and evolution difficult. Several smells are cataloged in the literature, and their detection is traditionally supported by Non-Interactive Detection (NID) approaches. These approaches do not support the progressive interaction of developers with the affected code, revealing occurrences of more global smells only at the developer's request, recurrently implying a late identification of these smells. With the emergence of Interactive Detection (ID), the aim is to deal with the limitations of traditional approaches, allowing the revelation of instances of code smells without an explicit request from the developer, encouraging early detection as a recommended practice. Although developers consider using ID approaches, guidelines were not found in the literature on when and how such approaches should be used in the context of software development activities. In this work, the objective is the experimental evaluation of the interactive detection of code smells in the context of the activities of the software development process. For doing so, studies were conducted to identify the need to use such an approach, evaluate adherent detection methods, develop automated support with DI characteristics, present empirical evidence on its effectiveness in detecting anomalies, as well as describe a way of integration to the development process according to the Scrum framework. The experimental tasks revealed that using the ID technique led to an increase of up to 40% in the recall and up to 25% in the precision in the detection of code smells during inspection activities and code development. Based on the results of the studies, it is concluded that factors associated with ID contributed to the early detection of a greater number of occurrences of code smells compared to traditional approaches. Consequently, using the ID approach in a disciplined way in a development process can promote continuous evaluation and improve the quality of the software.

# Dedicatória

Dedico este trabalho às memórias da minha mãe, Josélia Albuquerque de Farias Leite (*in memoriam*), pelo seu exemplo de perseverança, resiliência e coragem; da minha avó, Antonieta Albuquerque de Farias Leite (*in memoriam*), por sempre ter acreditado e apoiado os meus sonhos; e dos meus Tios Josemar (*in memoriam*) e Josenaldo (*in memoriam*) pelas demonstrações de responsabilidade e amor à família. Dedico também ao meu grande amigo e sogro, Genival Moraes Leal (*in memoriam*), pelos exemplos de bondade e simplicidade.

# Agradecimentos

Antes de mais nada, agradeço a Deus por estar ao meu lado o tempo todo durante toda essa jornada. Minha coragem para seguir em frente veio da minha fé inabalável.

Gostaria de agradecer a minha amada mãe Josélia Albuquerque de Farias Leite (*in memoriam*) por todo apoio e por me fazer entender que a educação é a maior força de transformação social. Suas palavras, ensinamentos e exemplos estarão sempre comigo onde eu estiver. Obrigado, mãe! Sem você eu não teria chegado até aqui. De modo igualmente especial gostaria de agradecer a minha avó Antonieta Albuquerque (*in memoriam*) por todo incentivo, palavras de apoio e suporte emocional. Obrigado, vó!

Gostaria de agradecer à minha companheira Raquel Leal pelo apoio e suporte prestado ao longo da realização deste trabalho. Externo também meus agradecimentos aos meus filhos Gabriel e Lis que compreenderam muitas vezes minha ausência ao longo destes anos de estudo. Vocês são a razão do meu esforço. Essa conquista também é de vocês!

Meu profundo agradecimento a toda minha família: meu pai Dorgival Cândido de Albuquerque, meus irmãos Dorgival Cândido de Albuquerque Júnior, Roberta Albuquerque dos Santos e Maria Virgínia da Conceição Albuquerque, meus tios José Muniz e Josefa Albuquerque, meus primos Emanuelle Marie, Leandro Leite, Severino Gabriel, Gabrielle Letícia, Danielle Albuquerque e Isabelle Cristina. Obrigado pelo apoio nesta árdua missão.

Minha profunda gratidão ao meu orientador Prof. Hyggo Almeida pelo apoio contínuo em meus estudos durante o doutorado. Tenho uma dívida eterna pela sua paciência, conselhos e conhecimentos transmitidos ao longo desses anos. Eu não tenho palavras para expressar toda a minha admiração e gratidão. Obrigado pela confiança neste aluno, bem como nesta pesquisa!

Gostaria de expressar minha gratidão aos membros da banca, Prof. Danilo Santos, Prof. Evandro Costa, Prof. Lenardo Chaves e Prof. Roberto Oliveira. Vossos conselhos, sugestões e apontamentos foram fundamentais para conclusão desta pesquisa. Obrigado por todo esforço dispensado na leitura e compreensão deste trabalho.

Agradeço a todos os professores do Programa de Pós-Graduação em Ciência da Computação (PPGCC) da Universidade Federal de Campina Grande (UFCG) pela contribuição

para a minha formação e de todos os meus colegas do programa. Meus sinceros agradecimentos também à equipe administrativa do PPGCC/UFCG, em especial às servidoras Paloma Porto e Lyana Nascimento pela solicitude e presteza em todas as demandas deste aluno.

Externo minha gratidão aos meus colegas e amigos do *Intelligent Software Engineering (ISE/UFCG)*. Em particular, gostaria de agradecer a Ademar Neto, Alexandre Braga, Alexandre Costa, Dalton Cézane, Emanuel Dantas, Ednaldo Dilorenzo, Felipe Silva, Ferdinandy Chagas, Luiz Antônio Pereira, Manuel Silva e Renata Saraiva. Minha gratidão a todos pela ajuda e companheirismo. Agradecimentos especiais a Mirko Perkusich pela significativa contribuição para o meu crescimento como pesquisador. Suas palavras e conselhos contribuíram muito para os resultados dessa pesquisa.

Meus sinceros agradecimentos também à Profa. Graziela Tonin (Insper - São Paulo), Prof. Mauricio Aniche (TU Delft - Países Baixos), Prof. Tushar Sharma (*Dalhousie University* - Canadá) e Roberta Arcoverde (*Stack Overflow* - Estados Unidos) pela colaboração nos diversos estudos deste trabalho. O apoio de todos vocês enriqueceu esta pesquisa. Externo meus agradecimentos de modo especial ao Prof. Everton Guimarães (*Penn State University* - Estados Unidos) que sempre esteve ao meu lado desde a gênese desta pesquisa.

Meu reconhecimento aos colegas da PUC-RIO que também colaboraram com minha formação enquanto pesquisador. Externo minhas saudações a Diego Cedrim, Willian Oizumi, Luciano Sampaio, Leonardo Sousa, Bruno Cafeo, Eiji Hadachi, Marx Viana, Chrystinne Fernandes, Francisco Cunha e João Neves. Agradecimentos especiais a Soeli Fiorini e Rafael Nasser que confiaram no meu trabalho em diversos projetos de Pesquisa e Desenvolvimento realizados junto ao Laboratório de Engenharia de Software daquela instituição.

Gostaria de agradecer também aos meus colegas da Universidade Federal de Campina Grande pelo apoio e suporte em fazer a minha participação no programa de doutorado possível. Agradeço ainda aos meus colegas do Instituto Federal da Paraíba (IFPB), bem como a gestão da instituição pelo apoio financeiro durante as diferentes fases desta tese por meio do Programa de Incentivo a Qualificação do IFPB (PIQ/IFPB) - Edital Nr 21/2021 PRPIG.

Por fim, externo meus sinceros agradecimentos a todos que de alguma forma contribuíram para a realização deste trabalho.

# Conteúdo

# Lista de Siglas e Abreviaturas

ADI - Ambiente de Desenvolvimento Integrado

ASD - *Agile Software Development*

AST - *Abstract Syntax Tree*

ASAT - *Automated Static Analysis Tools*

API - *Application Programming Interface*

BBN - *Bayesian Belief Network*

CBSoft - Congresso Brasileiro de Software

CD - *Continuous Delivery*

CEEI - Centro de Engenharia Elétrica e Informática

CI - *Continuous Integration*

CONCAD - *CONtinuous Code Anomalies Detection*

CPU - *Central Processing Unit*

DI - Detecção Interativa

DoD - *Definition Of Done*

DNI - Detecção Não-Interativa

DT - Dívida Técnica

ES - Engenharia de Software

FN - Falso Negativo

FP - Falso Positivo

GQM - *Goal-Question-Metric*

H - Hipótese

IC - Integração Contínua

IDE - *Integrated Development Environment*

IES - Instituição de Ensino Superior

IFPB - Instituto Federal da Paraíba

ISE - *Intelligent Software Engineering*

JDBC - *Java Database Connectivity*

JSERD - *Journal of Software Engineering Research and Development*

LLM - *Large Language Models*

MSL - Mapeamento Sistemático da Literatura

MVC - *Model-View-Controller*

OO - Orientação a Objetos

ORM - *Object-Relational Mapping*

POO - Programação Orientada a Objetos

QA - *Quality Assessment*

QCC - Qualidade de Código Contínua

QP - Questão de Pesquisa

RSL - Revisão Sistemática da Literatura

SBES - Simpósio Brasileiro de Engenharia de Software

SE - *Software Engineering*

SoftCOM - *International Conference on Software, Telecommunications, and Computer Networks*

SVM - *Support Vector Machine*

TD - *Technical Debt*

UFCG - Universidade Federal de Campina Grande

UML - *Unified Modeling Language*

VEM - *Workshop on Software Visualization, Evolution, and Maintenance*

VIRTUS - Núcleo de Pesquisa, Desenvolvimento e Inovação em Tecnologia da Informação, Comunicação e Automação

VP - Verdadeiro Positivo

# Lista de Figuras

# Lista de Tabelas

# Capítulo 1

# Introdução

Ao longo do ciclo de vida de um software, diversas modificações são realizadas a fim de lidar com sua evolução. Devido às necessidades do mercado atual, tais modificações ocorrem com restrições de tempo, exigindo entregas em períodos cada vez mais curtos. Como consequência, a qualidade do software é frequentemente negligenciada, potencializando problemas associados à arquitetura bem como ao código em desenvolvimento [93]. Assim, atividades relacionadas à preservação e melhoramento da qualidade do software têm se tornado cada vez mais importantes desde as fases iniciais do desenvolvimento [97].

Um indicador relevante para a qualidade de um sistema é o acúmulo de itens de Dívida Técnica (DT) que refletem compromissos técnicos que podem gerar benefícios a curto prazo, mas podem prejudicar a qualidade a longo prazo [17]. A DT pode ser associada ao processo de tomada de decisão sobre os atalhos e soluções alternativas adotadas no desenvolvimento de software. No entanto, essas decisões podem ter uma influência positiva ou negativa na qualidade dos artefatos de desenvolvimento de software. Itens de DT podem ocorrer devido a diversos fatores (e.g., mudanças de requisitos e prazos rígidos), impactando os mais variados artefatos de software (e.g., arquitetura, código-fonte, entre outros). Como o código-fonte é um dos artefatos mais confiáveis e frequentemente atualizados ao longo do processo de desenvolvimento [46], as organizações de software e seus profissionais devem perceber e gerenciar oportunamente itens de DT em seus projetos [109], particularmente aqueles que afetam o código-fonte.

As anomalias de código são consideradas um dos principais indicadores da DT no código-fonte [38]. Um trabalho liderado por Fowler [38] apresentou um catálogo com 20

tipos diferentes de anomalias de código (do inglês, *code smells* ou *bad smells*), bem como aspectos negativos relacionados à ocorrência destas anomalias em projetos de software. As anomalias são estruturas que podem indicar problemas, tornando o software difícil de manter e evoluir [38]. Quando essas estruturas são detectadas oportunamente, os desenvolvedores podem realizar ações de refatoração em menos tempo [75] e esforço [120]. Tais ações de refatoração são mudanças estruturais no código sem alteração do seu comportamento com o objetivo de melhorar a qualidade do software [87].

As anomalias de código têm sido investigadas a partir de diversas perspectivas, incluindo sua introdução e evolução [33], bem como seu impacto em atributos de qualidade de software como manutenibilidade [51] e confiabilidade [90]. Mais recentemente, outros trabalhos apresentaram evidências de que, além de comprometer atributos de qualidade [49], o crescimento na incidência de anomalias pode também levar à degradação da arquitetura [83] e à propensão a falhas [105]. Assim, de forma generalista, os pesquisadores tendem a confirmar que a detecção e correção de tais anomalias podem ser uma abordagem promissora para manutenção e melhoria da qualidade do software [46].

Como as anomalias de código podem passar despercebidas enquanto os desenvolvedores estão trabalhando, diversas abordagens para detecção destas anomalias foram propostas. Estas abordagens visam alertar os desenvolvedores sobre a presença dos diversos tipos de anomalias de código bem como para ajudá-los na compreensão de suas causas. Nas abordagens manuais, desenvolvedores podem realizar inspeções diretas no código em desenvolvimento. Contudo, a utilização de abordagens manuais implica em atividades não-repetíveis, não-escaláveis e dispendiosas em termos de tempo e esforço [67].

Diante dessas dificuldades, pesquisadores desenvolveram abordagens automatizadas para detecção das anomalias que levam em consideração o código-fonte, características estáticas e dinâmicas do sistema e também do processo de desenvolvimento. De modo geral, essas abordagens são compostas por dois componentes: (i) *Mecanismo de detecção*, que permite que os desenvolvedores escolham ou definam algoritmos para a detecção das anomalias [105]; e (ii) *Interface do usuário*, que representa o meio de comunicação entre desenvolvedores e os resultados providos pelo mecanismo de detecção. Com base na interação do desenvolvedor com esses componentes bem como na capacidade de interação direta com trechos de código "anômalos", utilizou-se no contexto deste trabalho a seguinte taxonomia de classificação:

*Detecção Não-Interativa (DNI)* e *Detecção Interativa (DI)*.

É evidente na literatura da área que a maior parte das técnicas suportam a abordagem DNI [33] [89] [99] [102] [37] [78]. De modo geral, a abordagem DNI tem como objetivo revelar uma lista global de anomalias do projeto apenas quando o código-fonte é concluído e mediante demanda explícita do desenvolvedor. Adicionalmente, esta abordagem não oferece aos desenvolvedores os meios para interagir diretamente com os elementos de código afetados durante a produção, edição ou inspeção de fragmentos de código. Além disso, a abordagem DNI não provê informações locais e contextualizadas para auxiliar na compreensão das causas e espalhamento das anomalias [79]. Por fim, o uso de DNI encoraja a detecção tardia de anomalias, comprometendo significativamente aspectos associados à qualidade do software [46][80][7].

Na literatura encontra-se conhecimento empírico a respeito da eficácia relacionada à abordagem DNI [110] [37] [102]. Alguns estudos apontaram que o uso da abordagem DNI induz um baixo número de anomalias detectadas corretamente [79], impactando negativamente em sua eficácia [77]. Outros estudos sugeriram que esta abordagem pode influenciar desenvolvedores a realizar ações de refatoração ineficazes [108] [103]. Finalmente, outros estudos comprovaram o impacto negativo que a utilização da abordagem DNI pode acarretar na qualidade do software [111] [116].

Por outro lado, as abordagens DI foram propostas mais recentemente com o objetivo de lidar com as limitações das abordagens DNI [28]. Mediante suporte de DI, desenvolvedores realizam concomitantemente atividades de programação em conjunto com detecção das anomalias, sem uma solicitação explícita, encorajando a detecção precoce de anomalias como uma boa prática. Ainda, o uso da abordagem DI permite que os desenvolvedores interajam direta e oportunamente com os elementos afetados por anomalias à medida que criam, editam ou analisam fragmentos de código, promovendo uma manutenção contínua do código durante as atividades de programação. Por fim, as suas informações contextuais e localizadas a respeito da ocorrência de anomalias contribuem significativamente para que os desenvolvedores compreendam suas causas, origens e nível de espalhamento.

O presente trabalho está inserido no contexto de detecção de anomalias de código. Mais especificamente, o foco neste trabalho é na avaliação da utilização da abordagem de detecção interativa de anomalias de código. Em linhas gerais, busca-se responder se o uso da

abordagem DI traz benefícios em relação ao uso de abordagens tradicionais para detecção de anomalias de código e melhoria da qualidade de software.

## 1.1 Problemática

A abordagem DI pode auxiliar desenvolvedores na manutenção e melhoria da qualidade do software em desenvolvimento [90]. Porém, há uma escassez de conhecimento sobre como integrar a abordagem DI dentro de um ambiente de desenvolvimento de software [28] [120]. A concepção de uma abordagem DI vai além de simplesmente tornar o mecanismo de detecção de uma abordagem DNI tradicional em funcionamento contínuo e em segundo plano. Os custos envolvidos no processamento requerido pelos mecanismos de detecção podem degradar o funcionamento do ambiente de desenvolvimento, comprometendo a realização da atividade de programação principal e desencorajando desenvolvedores a utilizarem esta abordagem. Além disso, embora os desenvolvedores considerem utilizar DI em suas atividades no processo de desenvolvimento de software, não foram encontradas na literatura diretrizes sobre *quando* e *como* essa abordagem deve ser aplicada com vistas a melhoria da qualidade de software.

Enuncia-se, então, o problema de negócio abordado neste trabalho: Como utilizar a abordagem DI integrada ao processo de desenvolvimento de software para melhorar a qualidade do software em desenvolvimento?

A utilização das abordagens DNI e DI já tem sido discutida desde a década passada no contexto da detecção de anomalias de código. Por exemplo, Murphy-Hill e Black [80] definiram e discutiram duas práticas distintas que os desenvolvedores podem utilizar para realizar a detecção de anomalias de código: (i) de forma esporádica, e (ii) de modo frequente e disciplinado. É importante notar que a primeira prática está alinhada com a abordagem DNI enquanto que a última está alinhada com a abordagem DI. Embora os autores tenham discutido as vantagens percebidas com uso da segunda abordagem, eles concluíram que existe uma dificuldade de implantar DI por, entre outros fatores, falta de conhecimento de como adequar a abordagem ao processo de desenvolvimento, escassez de suporte automatizado para realização desta abordagem e falta de treinamento dos desenvolvedores [81] [80].

Mais recentemente, outros estudos descreveram a utilização de técnicas que suportam

características da abordagem DI. Murphy-Hill e Black [79] propuseram uma abordagem de detecção de anomalias que fornece uma visualização diferenciada dos resultados de detecção. A abordagem foi avaliada através de um experimento controlado considerando aspectos relacionados a sua usabilidade. Similarmente, Ganea *et al.* [40] propuseram um suporte automatizado integrado ao *Eclipse* IDE para detecção de anomalias e sugestão de ações de refatoração. Este trabalho limitou-se a apresentar detalhes sobre a implementação bem como a eficiência associada à ferramenta. Por fim, Quang Do *et al.* [28] apresentaram o conceito de análise estática *Just-In-Time* (JIT) que intercala o desenvolvimento de código e a correção de anomalias em um ambiente de desenvolvimento integrado. Os autores descreveram as diretrizes gerais para projetar análises JIT e para transformar análises de fluxo de dados estáticos em análises JIT por meio de um conceito de execução de análise em camadas.

De modo resumido, os trabalhos descritos até o presente momento na literatura não avaliam técnicas que suportam a abordagem DI durante a detecção de anomalias de código, com o objetivo específico de discutir seu impacto na realização desta atividade bem como na qualidade do software. Adicionalmente, esses trabalhos não demonstram evidências empíricas relacionadas à utilização de técnicas que suportam a abordagem DI no contexto das atividades do processo de desenvolvimento de software.

Diante disso, enuncia-se o problema técnico específico abordado neste trabalho: Como realizar a detecção interativa de anomalias de código de forma integrada ao processo de desenvolvimento ágil de software?

É importante mencionar que a abordagem DI está aderente às práticas ágeis de desenvolvimento de software. Por exemplo, o uso da abordagem DI poderá trazer benefícios no contexto da Integração Contínua (IC), em que os desenvolvedores integram constantemente suas mudanças no projeto por meio de um processo de construção automatizado. Outro ponto de convergência com práticas ágeis diz respeito à Qualidade de Código Contínua (QCC). Esta atividade inclui a execução contínua e constante de análises estáticas e dinâmicas do código com o objetivo de promover e garantir sua qualidade [30]. A abordagem DI tem essa capacidade de promover alertas precoces e constantes de ocorrências de anomalias, reduzindo assim os custos associados à manutenção da qualidade uma vez que estas atividades estão diluídas juntamente ao próprio processo de desenvolvimento de software.

## 1.2 Objetivos

O objetivo neste trabalho é realizar uma avaliação experimental da abordagem de detecção interativa de anomalias de código no contexto das atividades do processo de desenvolvimento de software.

Através de um método baseado em métricas e heurísticas de forma combinada, implementada em uma ferramenta integrada ao ambiente de desenvolvimento, viabilizou-se a detecção interativa. Com o suporte da ferramenta, foram realizados experimentos no contexto da análise e criação de código em projetos reais, indicando que a abordagem DI se mostra promissora para a detecção de anomalias de maneira integrada às atividades do processo de desenvolvimento de software.

Para alcançar o objetivo principal anteriormente descrito, os seguintes objetivos específicos foram definidos:

- Caracterizar a abordagem de detecção interativa de anomalias de código em contraste com abordagens DNI tradicionais;
- Validar os requisitos e o método de detecção aderentes à abordagem de detecção interativa através da implementação de um suporte automatizado;
- Avaliar a eficácia da detecção interativa de anomalias de código em contraste com técnicas tradicionais no contexto das atividades do processo de desenvolvimento de software;
- Propor a aplicação da abordagem de detecção interativa no processo ágil de desenvolvimento de software.

## 1.3 Metodologia

O presente trabalho utilizou os passos metodológicos definidos por Wieringa *et al.* [124] para definição do problema de pesquisa, questões de pesquisa em conjunto com suas respostas bem como para definição de artefatos para resolução do problema. A seguir são apresentadas as Questões de Pesquisa (QP), as Hipóteses (H) e, por fim, as etapas definidas para a execução do trabalho.

- **QP1.** Existe a necessidade de uma abordagem para dar suporte à detecção interativa de anomalias de código?
- **QP2.** Quais métodos de detecção se adéquam à abordagem de detecção interativa?
- **QP3.** Quais fatores uma abordagem de detecção interativa deve considerar em sua concepção e utilização?
- **QP4.** A aplicação da abordagem de detecção interativa contribui para melhoria da eficácia na detecção de anomalias de código durante a análise de código?
- **QP5.** A aplicação da abordagem de detecção interativa contribui para reduzir a quantidade de anomalias de código remanescentes durante o desenvolvimento do código?
- **QP6.** Como a abordagem DI pode ser integrada ao fluxo de trabalho do processo ágil de desenvolvimento?

Com base nas QPs descritas anteriormente, a hipótese (H1) deste trabalho é definida da seguinte forma:

- **H1.** A utilização da abordagem de detecção interativa eleva a eficácia na detecção de anomalias de código e contribui para elevar a qualidade do software em desenvolvimento.

Para a realização deste trabalho, bem como a validação da sua hipótese principal, foram definidas as seguintes etapas e seus relacionamentos com as QPs apresentadas anteriormente:

1. realizar uma revisão da literatura para identificar as principais características das abordagens DI e DNI, propondo um arcabouço de classificação destas abordagens (**QP1**);
2. realizar um estudo qualitativo na indústria para avaliar a necessidade de uma abordagem DI para detecção de anomalias de código (**QP1**);
3. realizar uma revisão da literatura para identificar e analisar os métodos de detecção e, com base em critérios, definir os métodos que se adequam à abordagem DI (**QP2**);
4. identificar os tipos de anomalias mais relevantes e sua adequabilidade no contexto da DI (**QP2**);

5. identificar e avaliar os fatores e características que uma abordagem DI deve considerar em sua concepção e uso nas atividades do processo de desenvolvimento de software(**QP3**);

6. avaliar a eficácia da abordagem DI por meio de experimentos controlados utilizando desenvolvedores profissionais e estudantes no contexto da análise de código(**QP4**);

7. aplicar a abordagem DI em experimentos controlados utilizando desenvolvedores profissionais e estudantes visando avaliar a redução de anomalias de código remanescentes no contexto do desenvolvimento do código (**QP5**);

8. propor e avaliar a utilização da abordagem DI de forma integrada ao processo de desenvolvimento de software no contexto de projetos ágeis (**QP6**).

Com a realização das **etapas 1 e 2**, espera-se identificar as características da abordagem DI e confirmar sua necessidade na atividade de detecção de anomalias. Com a realização das **etapas 3, 4 e 5**, tem-se a expectativa de identificar aspectos relevantes para integrar a detecção interativa ao ambiente de desenvolvimento. Com a realização da **etapa 6**, será avaliada a eficácia da abordagem DI na detecção de anomalias de código durante a análise e desenvolvimento de código. Similarmente, com a realização da **etapa 7**, será analisada se a aplicação da abordagem DI contribui para diminuição de anomalias remanescentes durante o desenvolvimento de novos módulos de software. Finalmente, com a realização da **etapa 8**, espera-se uma propositura e validação da abordagem DI de forma integrada ao processo de desenvolvimento utilizando como base o arcabouço *Scrum*.

## 1.4 Relevância e Contribuições

Manutenção é uma atividade essencial para qualquer sistema de software [56]. De acordo com alguns estudos, entre 50% e 80% dos custos de um software são associados às atividades de manutenção, entre elas a detecção e refatoração de anomalias de código [62] [113]. Contudo, a manutenção do software é uma atividade difícil de realizar em virtude da falta de documentação útil associada ao software. Desse modo, o código-fonte torna-se uma das poucas fontes confiáveis de informações, sendo a preservação de sua qualidade essencial para sua longevidade [46].

Por outro lado, é amplamente aceito que as anomalias de código são uma das principais ameaças à qualidade do software [49]. Nesse sentido, espera-se que técnicas de detecção de anomalias de código auxiliem os desenvolvedores de forma eficaz nesta identificação e refatoração destas anomalias. Conforme exposto, a abordagem DI pode contribuir na melhor identificação e, consequentemente, na remoção das anomalias de código, implicando na diminuição do tempo e esforço associado à execução das atividades de manutenção do software. Com base nos resultados obtidos, as principais contribuições desta tese são apresentadas a seguir:

- um guia de classificação das abordagens de detecção de anomalias, a partir do qual a comunidade científica e desenvolvedores profissionais podem identificar um conjunto de características que distinguem as abordagens DI e DNI;

- avaliação de mecanismos de detecção e construção de um suporte automatizado aderente as caraterísticas DI denominado *Eclipse ConCAD*. Esta ferramenta foi projetada visando otimizar a identificação e correção de anomalias de código. O *Eclipse ConCAD* não apenas proporciona uma interface intuitiva para a aplicação eficaz da DI, mas também incorpora mecanismos de detecção e funcionalidades de interação que aprimoram significativamente a experiência dos desenvolvedores;

- análise da eficácia da abordagem DI comparada com DNI. Este resultado respalda as pesquisas futuras sobre aplicação de DI nas atividades do processo de desenvolvimento de software. De fato, os desenvolvedores conseguiram identificar mais instâncias de anomalias de código a partir do uso dessa abordagem. Portanto, organizações podem considerar sua adoção para melhorar a eficácia na detecção de anomalias de código;

- fatores e características da abordagem DI que podem impactar na eficácia da detecção de anomalias. Desenvolvedores que utilizam DI estão constantemente cientes das instâncias de anomalias ao analisar diferentes fragmentos de código. Ainda, podem detectar antecipadamente um maior número de instâncias devido à disponibilidade de informações relacionadas às anomalias. No geral, os fatores observados nos estudos podem auxiliar as organizações em aplicar cuidadosamente DI para identificação e correção de anomalias de código;

- a abordagem integrada ao processo de desenvolvimento permite vislumbrar uma potencial aplicação prática em projetos da indústria, assim como viabiliza novas pesquisas alinhadas com a aplicação das abordagens DI e DNI em projetos reais.

Finalmente, espera-se que os resultados deste trabalho possibilitem insumos nas futuras pesquisas que investiguem mais profundamente a abordagem DI. Acredita-se que esta abordagem irá beneficiar equipes na indústria, especialmente em times vinculados a núcleos de inovação tecnológica. No contexto destes núcleos existe uma tendência maior de ocorrência de anomalias de código em virtude do caráter flexível dos requisitos, necessidade de entregas contínuas e frequentes bem como pelo uso de múltiplas tecnologias.

Ainda, este trabalho tem relevância para o avanço nas pesquisas do grupo ISE (*Intelligent Software Engineering*) da UFCG, que investiga a aplicação de técnicas inteligentes para a melhoria da produtividade na prática de Engenharia de Software. Finalmente, espera-se que a aplicação disciplinada da abordagem DI no processo ágil de desenvolvimento de software contribua para a avaliação e a melhoria contínua dos artefatos de software produzidos pelos diversos projetos do VIRTUS, que também é parte da UFCG.

## 1.5 Estrutura do Documento

Os capítulos restantes que compõem este documento estão organizados da seguinte forma:

- **Capítulo 2: Fundamentação Teórica**. Apresentam-se definições gerais dos temas abordados neste documento;

- **Capítulo 3: Trabalhos Relacionados**. Discutem-se os principais trabalhos relacionados à anomalias de código, estratégias de detecção, bem como análises comparativas entre tais estratégias;

- **Capítulo 4: Motivação para a Abordagem de Detecção Interativa**. Apresenta-se o estudo com utilização de desenvolvedores profissionais que motivou o estudo da abordagem DI;

- **Capítulo 5: Avaliação de Mecanismos para Detecção Interativa**. Apresenta-se uma avaliação dos métodos de detecção de anomalias de código que podem ser utilizadas

na abordagem de detecção interativa;

- **Capítulo 6: Definição dos Requisitos para uma abordagem DI**. Apresentam-se os requisitos para uma abordagem de DI a ser integrada em um ambiente de desenvolvimento;

- **Capítulo 7: Avaliação da Abordagem DI Durante a Análise de Código**. Apresenta-se uma avaliação experimental realizada com projetos reais através de uma ferramenta desenvolvida para DI, denominada *Eclipse ConCAD*, no contexto da inspeção e análise de módulos de código pertencentes a projetos de software industriais;

- **Capítulo 8: Avaliação da Abordagem DI Durante o Desenvolvimento de Código**. Apresenta-se uma avaliação experimental realizada com projetos através da ferramenta *Eclipse ConCAD* no contexto do desenvolvimento de código;

- **Capítulo 9: Aplicação da abordagem DI no Processo Ágil de Desenvolvimento de Software**. Apresenta-se uma proposta da abordagem DI integrada ao processo de desenvolvimento de software baseada no arcabouço *Scrum*;

- **Capítulo 10: Considerações Finais**. Apresentam-se as considerações finais do trabalho, incluindo uma revisão dos resultados, contribuições e sugestões para trabalhos futuros.

# Capítulo 2

# Fundamentação Teórica

Neste capítulo são apresentados conceitos necessários ao entendimento desta tese. Nas próximas seções são fundamentados os conceitos acerca de dívida técnica e anomalias de código (Seção 2.1), ações de refatoração (Seção 2.2), estratégias de detecção de anomalias de código (Seção 2.3), conceitos dos estudos empíricos (Seção 2.4) e, por fim, o arcabouço *Scrum* (Seção 2.5).

## 2.1 Dívida Técnica e Anomalias de Código

**Dívida Técnica e seus Tipos**. A dívida técnica (DT) é uma metáfora que reflete compromissos técnicos que podem gerar benefícios de curto prazo, mas podem prejudicar a qualidade de um sistema de software a longo prazo [17]. A DT pode ser associada a diferentes artefatos de software [71]. Por exemplo, DT de código acomete o código-fonte enquanto que a DT de testes é verificada nos diversos artefatos associados ao arcabouço de teste de um sistema. Uma taxonomia amplamente aceita na literatura classifica os tipos de DT em dez categorias, e cada uma delas foi classificada em vários subtipos com base nos artefatos e causas-raiz da DT [60][20].

**Gestão da DT**. Além disso, existe uma série de atividades associadas à gestão da DT que lidam com sua ocorrência e acúmulo, tornando as DTs visíveis e controláveis para manter um equilíbrio entre o custo e o valor do projeto de software [20]. Li *et al.* [60] classificaram a gestão da DT em oito atividades, sendo as atividades de identificação, pagamento e monitoramento as mais recorrentemente citadas na literatura. A primeira atividade é considerada de

extrema importância pois é o estágio inicial que habilita a gestão da DT de forma adequada [17].

Diversos fatores como prazos rígidos, más práticas de programação, mudança de requisitos e baixa cobertura de testes podem contribuir para a introdução da DT durante o desenvolvimento de software [38]. Em essência, a DT pode surgir de um conjunto de decisões abaixo do ideal que afetam negativamente a qualidade de um artefato em particular. Por exemplo, no contexto das atividades do desenvolvimento de software, tem-se o código-fonte como um dos principais e mais confiáveis artefatos [71][46]. Assim, a aplicação de más práticas de programação pode incorrer no surgimento de anomalias de código (também conhecidas por *bad smells* ou *code smells*) que são um dos principais indicadores da DT manifestada no código-fonte.

**Anomalias de Código**. As anomalias de código indicam estruturas no código que precisam ser alteradas a fim de melhorar a qualidade do projeto [38]. Diferentemente de um *bug*, uma anomalia de código não causa necessariamente uma falha, mas pode levar a outras consequências negativas, impactando na manutenção e evolução do software. É inegável que o conceito de anomalias de código foi adotado, primeiro, pela comunidade de desenvolvimento ágil de software como forma de apontar algo errado ou um ponto de melhoria [19].

Devido a sua importância para qualidade de software, diversos pesquisadores propuseram e catalogaram diferentes tipos de anomalias de código [38], [68], [105]. Contudo, o uso do termo "*code smell*" tornou-se popular principalmente devido ao trabalho original de Fowler que o utilizou para identificar padrões de código que contêm problemas estruturais e, portanto, devem ser aprimorados [38]. Fowler e colegas foram pioneiros na identificação e discussão de anomalias de código e no fornecimento de um guia prático de técnicas para resolvê-los [38].

**Tipos de Anomalias**. Exemplos típicos de anomalias de código são *God Class* e *Long Method*. A primeira é caracterizada por classes de grande tamanho, baixa coesão e diversas dependências com outras classes de dados do sistema [38]. Sua ocorrência pode prejudicar a compreensão do programa e a manutenibilidade do software [51] [90]. A última pode ser observada quando os métodos implementam mais de uma funcionalidade (ou abordam mais de uma preocupação) [38], e sua ocorrência pode diminuir o entendimento do programa e

tornar o código-fonte mais sujeito a alterações e falhas [51] [90]. Na Tabela 2.1 é fornecida uma breve descrição de cada um dos tipos de anomalias de código apresentadas no catálogo de Fowler [38].

Tabela 2.1: Anomalias de Código do Catálogo de Fowler.

| **Anomalia** | **Breve Descrição** |
|---|---|
| *Duplicated Code* | Estrutura de código que aparece em mais de um lugar. |
| *Long Method* | Um método que é muito longo e possui muitas responsabilidades. |
| *God Class* | Se refere a classes que tendem a centralizar a inteligência do sistema. |
| *Long Parameter List* | Um método que possui muitos parâmetros. |
| *Feature Envy* | Métodos que usam muito mais dados de outras classes do que da classe em que estão definidos. |
| *Data Clumps* | Diferentes partes de código apresentam grupos idênticos de variáveis. |
| *Primitive Obsession* | Tipos primitivos de dados são usados em demasia no software. |
| *Switch Statements* | Casos de um comando *switch/case* apresentam duplicação de código. |
| *Speculative Generality* | Código criado para favorecer funcionalidades que nunca serão implementadas. |
| *Temporary Field* | Verifica-se em um objeto que possui uma variável de instância que apenas é utilizada em certas circunstâncias. |
| *Message Chains* | Objeto cliente que requisita um objeto que é requisitado por outro. |
| *Middle Man* | Ocorre quando grande parte dos métodos de uma classe delega a execução para métodos de outra classe. |
| *Inappropriate Intimacy* | Verifica-se quando uma classe utiliza inadequadamente as estruturas internas de uma outra classe. |
| *Data Class* | Classes que possuem apenas atributos e métodos get/set. |
| *Refused Bequest* | Ocorre quando uma subclasse praticamente não utiliza os dados e métodos herdados da classe pai. |

| | |
|---|---|
| *Divergent Change* | Ocorre quando uma classe é comumente modificada de diferentes formas e por diferentes razões. |
| *Shotgun Surgery* | É o oposto da anomalia *Divergent Change*. Ocorre quando uma modificação em uma determinada classe requer uma série de pequenas mudanças em outras classes. |
| *Parallel Inheritance Hierarchies* | Caracteriza-se como um caso especial de *Shotgun Surgery* ligado a estrutura de herança. Sempre que uma determinada classe é modificada uma outra também precisa ser. |
| *Lazy Class* | Classes que não fazem nada no sistema e deveriam ser eliminadas. |
| *Comments* | Comentário utilizado para justificar uma implementação ruim. |
| *Alternative Classes* | Classes que fazem coisas similares mas possuem assinaturas diferentes. |
| *Incomplete Library Class* | Ocorre quando as bibliotecas de classes utilizadas não estão completas. |

**Categorização das Anomalias**. Uma forma interessante de compreender as anomalias de código é por meio da categorização baseada nas possíveis relações entre estas anomalias [65]. Por exemplo, Wake [123] propôs uma classificação das anomalias de código catalogados por Fowler com a seguinte divisão:

- *Anomalias Intraclasses*. Nesta categoria pode-se encontrar anomalias identificadas com métricas simples (e.g. *comments, long method, large class, long parameter list*), nomes que precisam de melhorias (e.g., *inconsistent names*), trechos de código que precisam ser removidos (e.g., *magic numbers, duplicated code, alternative classes with different interfaces*), e problemas na lógica condicional (e.g., *null check, complicated boolean expression, special case*).

- *Anomalias Interclasses*. Nesta categoria pode-se encontrar anomalias com comportamento inadequado (e.g., *primitive obsession, data class, data clump, temporary field*), relação entre hierarquias de classe (e.g., *refused bequest, inappropriate intimacy, lazy class, combinatorial explosion*), balanceamento de responsabilidades (e.g., *feature envy, message chains, middle man*) e alterações de código (e.g., *divergent change,*

*shotgun surgery, parallel inheritance hierarchies*).

Uma outra forma de categorizar as anomalias de código foi proposta por Mantyla [65]. Resumidamente, esta categorização buscou organizar as anomalias de código propostas por Fowler de acordo com suas características principais:

- *Bloaters*. Representa qualquer elemento no código que se tornou muito grande e não pode ser manipulado de forma eficaz. Em geral, estas anomalias são difíceis de entender e modificar. As anomalias pertencentes a esta categoria são: *Long Method, Large Class, Primitive Obsession, Long Parameter List* e *Data Clumps*;

- *Object-Orientation Abusers*. Todas essas anomalias são aplicações incompletas ou incorretas dos princípios de programação orientada a objetos (POO). Nesta categoria pode-se encontrar as seguintes anomalias: *Switch Statements, Temporary Field, Refused Bequest, Alternative Classes with Different Interfaces* e *Parallel Inheritance Hierarchies*;

- *Change Preventers*. Essas anomalias significam que, se for necessário alterar algo em um local do código, muitas alterações terão de ser realizadas em outros locais também. Como resultado, o desenvolvimento do programa se torna muito mais complicado e caro. As anomalias pertencentes a esta categoria são *Divergent Change* e *Shotgun Surgery*;

- *Dispensables*. Um dispensável é algo sem sentido e desnecessário, cuja ausência tornaria o código mais limpo, eficiente e fácil de entender. Nesta categoria pode-se encontrar as seguintes anomalias: *Duplicated Code, Lazy Class, Data Class* e *Speculative Generality*;

- *Couplers*. Todas as anomalias neste grupo contribuem para o acoplamento excessivo entre as classes ou mostram o que acontece se o acoplamento for substituído por delegação excessiva. As anomalias pertencentes a esta categoria são: *Feature Envy* e *Inappropriate Intimacy*.

## 2.2 Refatoração

Apesar de fornecer subsídios para o entendimento geral de cada tipo de anomalia, o catálogo apresentado por Fowler não fornece mecanismos para detectar precisamente as anomalias em código [38]. Em contraste, o catálogo visa apresentar exemplos de situações em que ações de refatoração podem ser aplicadas para remoção de tais anomalias e melhorar a qualidade do software [38].

**Definição**. O termo "refatoração" veio do trabalho de Opdyke [87], que o define como estratégias de reorganização que suportam uma mudança em um elemento de software. A refatoração ajuda a tornar o código mais legível e eliminar possíveis problemas, além de melhorar os atributos de qualidade interna do software [73]. Finalmente, a refatoração também pode ser usada para reengenharia, permitindo tornar mais modular e estruturado um sistema específico (e.g., código legado).

Existem diferentes níveis de abstração e tipos de artefatos de software que podem ser aplicados na refatoração. Por exemplo, é possível aplicar refatoração em modelos UML (do inglês, *Unified Modeling Language*), esquemas de banco de dados, requisitos, arquitetura de software e estruturas de uma linguagem [73]. Portanto, a refatoração se concentra não apenas no código-fonte, mas também em outros artefatos e, por esse motivo, é necessário manter todos os artefatos sincronizados.

A refatoração normalmente ocorre em dois níveis: alto nível e baixo nível. Refatoração de alto nível (ou refatoração composta) pode ser definida como aquelas que se referem a mudanças de *design* significativas (geralmente macro ou arquiteturais). Em contrapartida, as refatorações de baixo nível são aquelas para pequenas mudanças aplicadas em nível de código. O trabalho de Opdyke também introduz um elemento fundamental para a refatoração: garantir que a transformação preserve o comportamento do programa [87].

Esta condição deve ser verificada antes de aplicar a transformação para garantir que ela não irá introduzir problemas de compilação ou alterar o comportamento do programa. Por exemplo, a refatoração *Extract Method* verifica se o fragmento de código selecionado contém elementos quebrados antes de executar a refatoração. Opdyke afirma que, para realizar a refatoração de alto nível, é necessário realizar a refatoração de baixo nível. Isso denota a importância das ações de refatoração em nível de código para manutenção local e global da

qualidade do código em desenvolvimento.

**Tipos**. Fowler [38] identificou mais de 70 tipos diferentes de refatoração, que variam de mudanças locais em um elemento de código específico (por exemplo, Extrair Variável Local) a uma mudança global (por exemplo, Extrair Classe). Na Tabela 2.2 apresentam-se algumas das técnicas de refatoração dentre as mais recorrentes [56].

Tabela 2.2: Refatorações de Código do Catálogo de Fowler.

| **Refatoração** | **Breve Descrição** |
|---|---|
| *Extract Interface* | Criar uma interface comum para as classes existentes. |
| *Extract Class* | Extrair uma superclasse do código compartilhado por classes existentes. |
| *Move Class* | Mover a classe de um pacote para outro pacote. |
| *Rename Class* | Renomear uma classe. |
| *Extract Method* | Criar um método baseado em declarações de um método existente. |
| *Inline Method* | Incorporar o corpo de um método em um método existente. |
| *Move Method* | Mover o método de uma classe para outra classe. |
| *Push Down Method* | Mover o método de uma classe principal para uma ou mais classes secundárias. |
| *Pull Up Method* | Mover o método de uma classe filha para sua classe pai. |
| *Rename Method* | Renomear um método. |

**Classificação**. Para ajudar na compreensão e na melhor organização dessas ações de refatoração, Fowler [38] as organizou em seis categorias distintas de acordo com similaridades e características. Nas lista a seguir apresenta-se cada uma das categorias e alguns exemplares dos tipos de refatoração dentre os mais recorrentes [56].

- *Composing Methods*. As técnicas de refatoração neste grupo simplificam os métodos, removem a duplicação de código e abrem o caminho para melhorias futuras. *Extract Method e Extract Variable* são exemplos de técnicas de refatoração desta categoria.

- *Moving Features between Objects*. Essas técnicas de refatoração mostram como mover com segurança a funcionalidade entre classes, criar novas classes e ocultar detalhes de

implementação do acesso público. *Move Method* e *Extract Class* são exemplos de técnicas de refatoração desta categoria.

- *Organizing Data*. Essas técnicas de refatoração ajudam no tratamento de dados, substituindo os primitivos por uma rica funcionalidade de classe. Outro resultado importante é o desembaraço das associações de classe, o que torna as classes mais modulares e reutilizáveis. *Replace Data Value with Object* e *Encapsulate Field* são exemplos de técnicas de refatoração desta categoria.

- *Simplifying Conditional Expressions*. As condicionais tendem a se tornar cada vez mais complicadas em sua lógica com o tempo, e ainda existem mais técnicas para combater isso também. *Consolidate Conditional Expression* e *Decompose Conditional* são exemplos de técnicas de refatoração desta categoria.

- *Simplifying Method Calls*. Essas técnicas tornam as chamadas de método mais simples e fáceis de entender. Isso, por sua vez, simplifica as interfaces de interação entre as classes. *Rename Method* e *Add Parameter* são exemplos de técnicas de refatoração desta categoria.

- *Dealing with Generalization*. A abstração tem seu próprio grupo de técnicas de refatoração, principalmente associadas à movimentação de funcionalidade ao longo da hierarquia de herança de classes, criando novas classes e interfaces e substituindo herança por delegação e vice-versa. *Pull Up Method* e *Extract Subclass* são exemplos de técnicas de refatoração desta categoria.

**Processo e automação**. Cada refatoração pode ser composta de um conjunto de etapas básicas simples. Esse conjunto de etapas básicas é denominado processo de refatoração. Mens e Tourwé [73] identificaram um processo de refatoração comum baseado em um conjunto de atividades, conforme lista a seguir:

1. detectar trechos de código com oportunidades de refatoração;

2. determinar qual refatoração pode ser aplicada no trecho de código selecionado;

3. certificar-se de que a refatoração selecionada preserva o comportamento;

4. aplicar a refatoração escolhida nos respectivos locais;

5. avaliar o efeito da refatoração nas características de qualidade do software;

6. manter a consistência entre o artefato refatorado e outros artefatos de software.

Ao mesmo tempo, a refatoração pode se tornar uma ferramenta de melhoria contínua de software, principalmente se a equipe for formada por desenvolvedores preocupados com a qualidade do código [56]. De acordo com Parnin *et al.* [94], um problema relacionado à refatoração é que os benefícios da qualidade do software obtidos por meio dessas práticas são frequentemente diluídos pelos altos custos e baixa prioridade quando comparados à urgência de correções de *bugs* e implementação de novas funcionalidades. Esse problema ocorre porque cerca de 40% do tempo investido em manutenção de software é o custo para entender como o software (e sua arquitetura) vai evoluir [113].

Um aspecto relevante sobre a refatoração diz respeito ao momento de sua realização. Murphy-Hill e Black [80] apresentaram dois termos que podem resumir a postura em relação à refatoração: (i) *Floss Refactoring*, ou seja, adotar as técnicas de refatoração do dia-a-dia de forma saudável e disciplinada; e (ii) *Root Channel Refactoring*, quando não há o hábito de limpeza e isso pode ser muito caro ao longo do tempo, com a necessidade de planejamento.

## 2.3 Detecção de Anomalias de Código

Nesta seção discute-se sobre as formas principais de detecção (Seção 2.3.1), os conceitos envolvidos nesta atividade (Seção 2.3.2), a forma como os desenvolvedores interagem com essas estratégias (Seção 2.3.3) e os principais conceitos e modo de funcionamento das abordagens mais tradicionais de detecção de anomalias de código (Seção 2.3.4).

### 2.3.1 Formas de Detecção

Diante da relevância e da popularização das anomalias de código, diversos trabalhos propuseram técnicas focadas na detecção destas anomalias, a fim de auxiliar desenvolvedores a manter e melhorar a qualidade dos sistemas construídos por eles. A seguir são apresentadas as principais propostas para a detecção de tais anomalias, abordando como essa atividade pode ser realizada de forma manual e automática.

**Detecção Manual**. A detecção de anomalias de código através de inspeções manuais realizadas pelos desenvolvedores é discutida por Tavassos *et al.* [114]. Neste trabalho, os autores definiram uma técnica de inspeção que busca aumentar a efetividade dos revisores a partir de algumas diretrizes usadas para examinar um software Orientado a Objetos (OO) na busca por problemas no projeto. Tais diretrizes foram propostas pelos autores como um tipo de técnica de inspeção que tenta rastrear informações principais do projeto através de seus requisitos e documentos. Embora os resultados deste trabalho apontem que os desenvolvedores reconhecem avanços para a detecção de problemas em projetos OO, a técnica ainda carece de artefatos semânticos que ajudem no julgamento das anomalias encontradas.

Apesar das técnicas manuais permitirem que a detecção seja realizada de forma coerente com a percepção do desenvolvedor sobre os diversos tipos de anomalias de código, o que tornaria a detecção mais precisa, esse tipo de técnica não favorece a repetição do processo em outras situações parecidas. Assim, as abordagens manuais, de maneira geral, tornam-se tarefas não-repetíveis, não-escaláveis e que consomem muito tempo e esforço para a realização [67].

**Detecção Automática**. Diante das dificuldades relacionadas à detecção manual, pesquisadores investigaram técnicas que pudessem realizar a identificação de anomalias de código de forma automática. Marinescu [67] investigou o uso de métricas de software para a identificação de anomalias de código utilizando uma técnica que evoluiu para o conceito de estratégias de detecção. De acordo com o autor, uma estratégia de detecção consiste em uma expressão quantificável onde fragmentos de código que estão em conformidade com a regra podem ser detectados.

Em um outro trabalho [68], Marinescu detalhou o processo para a definição das estratégias de detecção de acordo com os seguintes passos: (i) seleção de métricas; (ii) definição dos limiares de aceitação; e (iii) composição da regra de detecção. O autor destacou que existe um grande desafio relacionado à definição de limiares apropriados para compor a regra de detecção. Afinal, a escolha destes limiares vai determinar a acurácia da regra na identificação de anomalias de código para os desenvolvedores. Por exemplo, em algumas ferramentas a estratégia de detecção para identificar *Long Method* tem sido definida como:

$$MLOC(m) > \alpha = LongMethod(m) \tag{2.1}$$

MLOC denota a métrica que computa o número de linhas do método *m* e *alpha* define um número inteiro com o limiar necessário para se considerar um método comum como *Long Method*, a partir do seu número de linhas.

Um estudo recente proposto por Sharma *et al.* [105] organizou os principais métodos de detecção de anomalias de código em cinco grupos: (i) baseado em métricas; (ii) baseado em aprendizado de máquina; (iii) baseado em histórico; (iv) baseado em heurísticas; e (v) baseado em otimização. É importante mencionar que a estratégia de detecção baseada em métricas ainda é uma das mais recorrentes utilizadas pelos desenvolvedores na construção das suas abordagens para detecção de anomalias de código [33] [2] [56].

### 2.3.2 Conceitos na Detecção de Anomalias de Código

Os conceitos de detecção de anomalias de código abrangem desde a definição das anomalias até os termos elementares relativos à detecção. Estes conceitos são fundamentais para a compreensão da presente pesquisa e são relatados na Figura 2.1. Na figura, modelam-se as relações entre os diferentes conceitos usando a notação UML [22][86]. Cada retângulo representa um conceito básico. Além disso, cada linha contínua representa uma relação hierárquica entre dois conceitos. Essa relação significa que o conceito representado no nível mais elevado é decomposto nos conceitos dos níveis mais baixos. Finalmente, as linhas pontilhadas representam os relacionamentos de dependência entre vários conceitos.

Um conceito básico central apresentado na Figura 2.1 é o *elemento de código* que consiste na unidade de decomposição básica de um sistema de software [39]. Esta unidade de decomposição é uma *classe*, que é composta por *métodos* e *atributos*, ou um método, que é composto por *statements* [58]. Determinados elementos de código também são suspeitos de conter instâncias de anomalias de código. Uma *suspeita de anomalia de código* consiste em um elemento de código que é possivelmente afetado por uma anomalia de código, mas ainda não confirmado ou refutado pelos desenvolvedores como realmente afetado pela anomalia de código. Por sua vez, uma *anomalia de código* é uma estrutura de código anômala que geralmente indica um ou mais problemas de manutenção em um sistema de software [38] [126].

Com relação às anomalias de código, cada instância destas anomalias são classificadas em um *tipo* específico. Além disso, cada tipo de anomalia tem uma *granularidade* distinta,

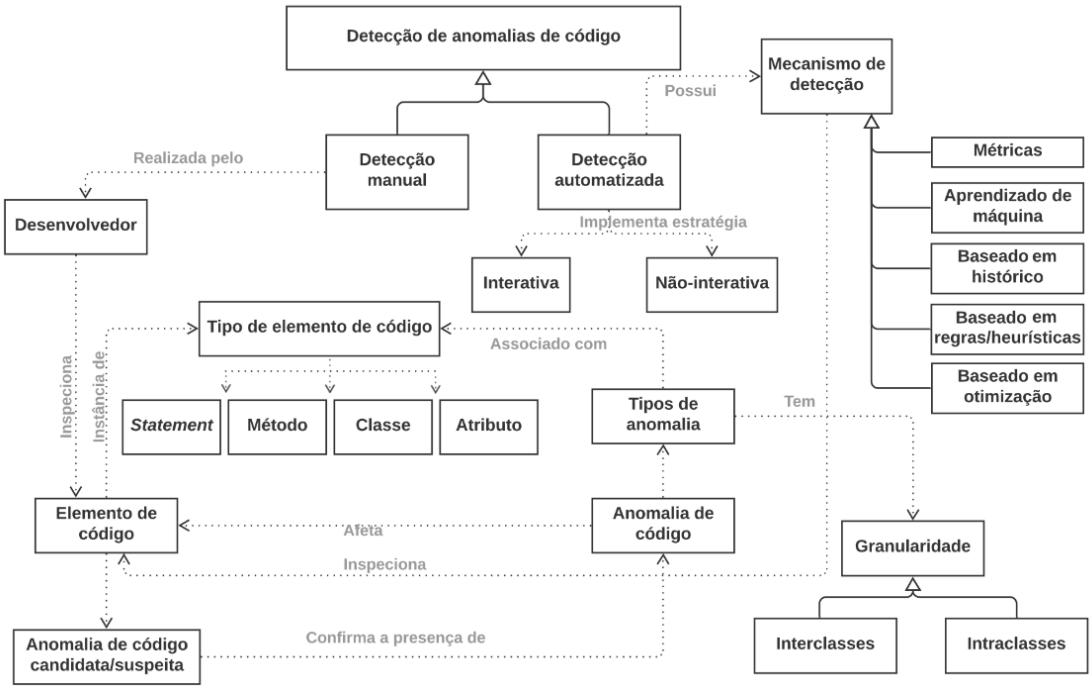


Figura 2.1: Conceitos da Detecção de Anomalias.

sendo estas classificadas em *intraclasse* ou *interclasse* [38] [70]. As anomalias intraclasse são estruturas de código anômalas que afetam uma classe particular do sistema, como *Long Method* ou *Long Parameter List*. As anomalias interclasses são aquelas que afetam várias classes juntas, como *Feature Envy* e *Message Chain*.

A *detecção das anomalias de código* pode ser realizada a partir de três etapas: (i) identificar suspeitos de anomalias no código-fonte; (ii) inspecionar cada indicação; e (iii) confirmar ou refutar cada suspeita de anomalia de código como uma ocorrência verdadeira [88]. Importante mencionar que os desenvolvedores realizam a detecção de anomalias a partir de abordagens *manuais* ou *automatizadas*. Um componente importante nesse contexto é o *mecanismo de detecção* e a forma de *interação* dos desenvolvedores com esses elementos. Tais conceitos serão definidos e discutidos detalhadamente na subseção a seguir.

### 2.3.3 Interação com as Estratégias de Detecção

Conforme apresentado anteriormente, os desenvolvedores de software normalmente usam técnicas automatizadas para guiar seus esforços na detecção de anomalias de código [67] [80] [58][96]. De modo geral, essas técnicas são compostas por dois componentes principais

[79] [7]:

- *Mecanismo de Detecção*. O mecanismo de detecção é o principal componente. Ele permite que os desenvolvedores escolham ou definam algoritmos para a detecção dos diversos tipos de anomalias de código [105]. Os desenvolvedores podem escolher algumas métricas e ajustar os limites para compor a estratégia de detecção [66]. Com base nessas regras, aprendizagem ou casamento de padrões, o mecanismo de detecção aponta as ocorrências de anomalias de código;

- *Interface do Usuário*. Após a detecção ser concluída, a interface do usuário exibe as instâncias de anomalias de código detectadas. Esse componente representa o meio de comunicação entre os desenvolvedores e os resultados providos pelo mecanismo de detecção. Existem duas formas principais de apresentar os resultados: (i) exibição das ocorrências de anomalias de código através de listas ou arquivos estruturados em XML ou JSON; e (ii) exibição das ocorrências de anomalias de código diretamente no fragmento de código afetado pelas anomalias.

Com base no "modo de interação" do desenvolvedor com os componentes mencionados acima, as técnicas podem ser classificadas de duas maneiras diferentes: Detecção Não-Interativa (DNI) e Detecção Interativa (DI). A primeira abordagem não propicia meios de interação com trechos de códigos anômalos e está propensa a exibir uma lista global de ocorrências de anomalias em todo o projeto de software. A segunda abordagem encoraja a interação direta do desenvolvedor com trechos de código anômalos e utiliza o próprio código-fonte onde o desenvolvedor realiza alguma atividade de programação para alertar ocorrências locais das anomalias. Na subseção a seguir, serão descritas as principais características e o modo de funcionamento da abordagem DNI, enquanto que as características e o modo de funcionamento da abordagem DI proposta será descrita na subseção 6.2.

### 2.3.4 Detecção Não-Interativa

De modo geral, técnicas que implementam a abordagem DNI têm como objetivo revelar uma lista global de anomalias apenas quando o código-fonte é concluído e mediante demanda explícita do desenvolvedor. Tais técnicas não oferecem aos desenvolvedores os meios para

interagir diretamente com os elementos de código afetados durante a produção, edição ou inspeção de fragmentos de código. Adicionalmente, encorajam a detecção tardia de anomalias de código além de não prover informações locais e contextualizadas para auxiliar na compreensão das causas e espalhamento das anomalias.

As técnicas de detecção de anomalias de código que implementam a abordagem DNI não oferecem suporte a interação direta do desenvolvedor com elementos de código afetados por anomalias de código devido a sua forma de funcionamento (Figura 2.2). A seguir, detalha-se cada uma das etapas do fluxo principal de funcionamento da abordagem DNI:

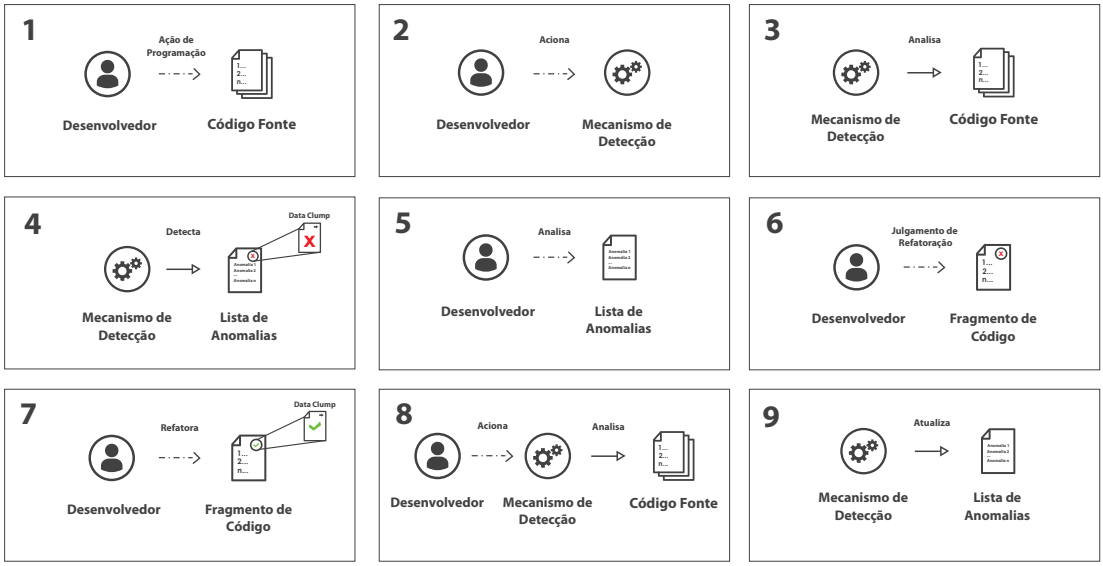


Figura 2.2: Etapas da Detecção Não-Interativa.

- *Etapa 1*. O desenvolvedor realiza alguma atividade de programação (e.g., implementação, modificação ou inspeção) com auxílio do seu Ambiente de Desenvolvimento Integrado;

- *Etapa 2*. O desenvolvedor aciona explicitamente o mecanismo de detecção de anomalias de código através de uma requisição para que o mesmo inicie seu funcionamento;

- *Etapa 3*. Este componente irá analisar o código-fonte com objetivo de realizar a identificação de anomalias (e.g., baseado em métricas, aprendizado de máquinas, baseado em heurísticas) considerando todo o projeto de software;

- *Etapa 4*. O mecanismo de detecção identifica a presença das anomalias de código, gerando uma lista global destas ocorrências ao longo de todo código-fonte do projeto considerado;

- *Etapa 5*. O desenvolvedor analisa a lista global de anomalias do código sem interagir diretamente com o código-fonte. Isso ocorre em virtude desta lista estar separada do código-fonte em que o desenvolvedor realiza alguma atividade de programação;

- *Etapa 6*. O desenvolvedor julga a necessidade de refatorar esta anomalia de código com base em critérios como criticidade e severidade da anomalia de código;

- *Etapa 7*. O desenvolvedor realiza a refatoração sem informações contextuais da anomalia (e.g., nível de espalhamento, elementos de código afetados, relacionamento com outras anomalias). Tão logo tomou sua decisão sobre a refatoração, deve realizar nova solicitação explícita ao mecanismo de detecção;

- *Etapa 8*. O desenvolvedor aciona novamente o mecanismo de detecção de anomalias de código através de uma requisição para que este componente analise o código-fonte com objetivo de realizar a identificação de anomalias;

- *Etapa 9*. O desenvolvedor analisa a lista global de anomalias de código remanescentes para concluir se a anomalia foi removida após a realização da refatoração (Etapa 7) antes de retornar para sua atividade de programação.

Conforme apresentado na Figura 2.2, os desenvolvedores que usam uma técnica que implementa a abordagem DNI devem sair do contexto da atividade de programação para identificar ocorrências de anomalias de código. Isso ocorre em virtude deles terem de mudar o contexto de um fragmento de código para analisar listas ou outras formas de visualização que apresentam resultados globais da ocorrência de anomalias de código. Adicionalmente, muitas destas técnicas que implementam a abordagem DNI não estão totalmente integradas ao ambiente de desenvolvimento. Requerendo muitas vezes o uso de suporte ferramental externo a este ambiente para apresentação dos resultados associados à detecção de anomalias de código.

## 2.4 Conceitos dos Estudos Empíricos

Os conceitos dos estudos empíricos consistem na definição de termos elementares que são fundamentais para a compreensão dos estudos quantitativos e qualitativos que serão apresentados. De modo resumido, eles se referem a conceitos avaliados durante a condução de estudos empíricos. Os principais conceitos do estudo empírico são apresentados na Figura 2.3 através do uso de uma notação inspirada na UML [22] e adaptada do trabalho de Oliveira [86]. No que segue, discute-se em detalhes cada conceito associados aos estudos empíricos.

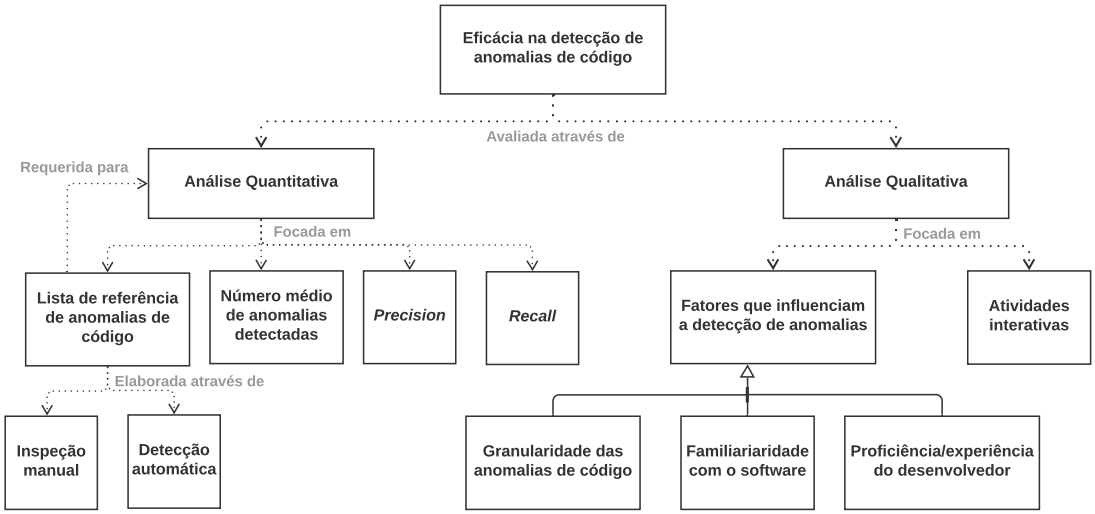


Figura 2.3: Conceitos dos Estudos Empíricos.

A eficácia na detecção das anomalias de código é um dos critérios mais importantes para a escolha de uma técnica para realizar esta atividade [80] [110]. Uma técnica de detecção de anomalias de código pode ser considerada eficaz ao detectar um grande número de instâncias de anomalias de código. Além disso, técnicas eficazes devem detectar apenas instâncias de anomalias de código associadas a problemas de manutenção [33]. Se os desenvolvedores usarem técnicas eficazes, eles podem detectar corretamente as instâncias de anomalias de código e, consequentemente, realizar ações de refatoração para melhorar a qualidade do software [110] [38].

Esta pesquisa avalia a eficácia da detecção de anomalias de código de duas maneiras, ou seja, investiga-se tal eficácia com base em análises quantitativas e qualitativas. Ambas análises são complementares, porque cada uma delas fornece diferentes pontos de vista e descobertas sobre a eficácia dos desenvolvedores na detecção de anomalias de código a partir

do uso de diferentes estratégias de detecção. Ambas análises serão descritas de acordo com o que segue.

**Análise quantitativa**. O presente estudo adota *precision* e *recall* como medidas para avaliar e comparar a eficácia na detecção de anomalias de código usando técnicas que implementam as abordagens DI e DNI. Essas medidas são definidas nas Equações 2.2 e 2.3. As equações foram adaptadas de Erich e Euzenat [31] e foram amplamente utilizadas em outros estudos recentes voltados para a comparação de técnicas de detecção de anomalias de código [89] [37] [115].

$$Precision = \frac{TP}{TP + FP} \tag{2.2}$$

$$Recall = \frac{TP}{TP + FN} \tag{2.3}$$

Todas essas medidas dependem da lista de referência de anomalias de código (também conhecida por "Oráculo"), que são construídas com base no relatório de uma ou mais estratégias de detecção de anomalias e validadas manualmente por desenvolvedores experientes em detecção e refatoração de anomalias de código. Uma lista de referência de anomalias de código relaciona as anomalias de código que realmente representam problemas em um sistema de software [34]. Para fins de esclarecimento, cada componente de medida pode ser brevemente descrito no seguinte:

- *True Positive* (TP) são instâncias de anomalias de código confirmadas por especialistas (por exemplo, desenvolvedores e engenheiros de software) que representam problemas de manutenibilidade no sistema em desenvolvimento;

- *False Positive* (FP) são candidatos a anomalias de código identificados pelos desenvolvedores usando uma técnica de detecção, mas não estão incluídos no oráculo original fornecido pelos especialistas;

- *False Negative (FN)* são instâncias de anomalias de código não identificadas pelos desenvolvedores (de forma manual ou mediante suporte automatizado) mas fazem parte do oráculo original como um indicador de problema de manutenção.

O *Precision* quantifica a taxa de TP pelo número de anomalias detectadas, enquanto *Recall* quantifica a taxa de TP pelo número de anomalias de código existentes. Na prática, taxas elevadas de *Precision* significam que uma técnica de detecção retornou resultados substancialmente mais relevantes (TPs) do que irrelevantes (FPs). Da mesma forma, altas taxas de *Recall* significam que uma técnica de detecção retornou a maioria das anomalias existentes no código.

**Análise qualitativa**. Esta medida concentra-se nos *fatores* e *atividades* da abordagem DI que contribuem para melhorar a eficácia dos desenvolvedores na atividade de detecção de anomalias. *Fatores* são características relacionadas aos sistemas de software, equipes de desenvolvimento ou organizações que contribuem para a eficácia da detecção de anomalias de código. Por sua vez, as *atividades* são ações com objetivos específicos que são realizadas pelos desenvolvedores durante a identificação de anomalias de código, tais como as ações realizadas para confirmar ou refutar uma suspeita de anomalia de código.

Com relação aos *fatores*, o foco deste trabalho concentra-se em três fatores. Primeiro, a granularidade da anomalia de código, uma vez que granularidades diferentes são inerentemente difíceis de inspecionar por desenvolvedores individuais. Com base em um trabalho anterior [112], considera-se a granularidade como o nível de abstração do código afetado por um suspeito de anomalia de código: no nível da classe (geral) ou nível de método (específico). Assim, a estratégia de detecção pode fazer com que tal inspeção seja realizada de modo facilitado. Em segundo lugar, a natureza da atividade de programação, o que significa que os desenvolvedores implementando novas funcionalidades ou avaliando trechos de códigos que não estão familiarizados, o uso da detecção interativa pode ser útil para melhorar a eficácia dos desenvolvedores neste contexto. Terceiro, o nível de proficiência dos desenvolvedores implicando que o tempo de experiência em projetos de software bem como o nível de entendimento dos conceitos associados à qualidade de software (e.g., anomalias de código e refatoração) podem impactar na eficácia da detecção de anomalias de código.

É digno afirmar que as atividades de detecção e remoção de anomalias de código são componentes de um conjunto maior de atividades disciplinadas associada ao desenvolvimento de software. Normalmente tais atividades são organizadas através de uma estrutura de gerenciamento que auxilia os diferentes papéis na definição das atividades, seu cronograma bem como seus responsáveis. No que segue, expõe-se uma estrutura de gerenciamento de

projetos atualmente relevante em projetos de software, sendo esta aderente a detecção interativa de anomalias de código.

## 2.5 *Scrum*

O *Scrum* é uma estrutura de gerenciamento de projetos que pode ser empregada para gerenciar e controlar as diversas atividades de desenvolvimento de software [104]. O termo foi introduzido por Ken Shwaber e Jeff Sutherland na década de 1990 em um artigo elaborado de modo conjunto entre estes pesquisadores. Este *framework* é usado para gerenciar projetos de software em um ambiente de frequente mudança. *Scrum* não foi concebido como um processo ou técnica empregado para construção de produtos. Em vez disso, é uma estrutura dentro da qual se pode empregar vários processos e técnicas [21]. *Scrum* tem a vantagem de ser muito visível e seu foco em software funcional mostra resultados para a gestão que eles podem ver e se entusiasmar [24].

*Scrum* avança as etapas de um projeto melhorando a comunicação e colaboração entre os membros da equipe (também conhecido por time). Em mais detalhes, a estrutura *Scrum* consiste no time e seus papéis, eventos, artefatos e regras associados. Cada componente dentro deste *framework* serve a um propósito específico e é essencial para o sucesso da utilização do *Scrum* [45].

A Tabela 2.3 apresenta os componentes essenciais do *Scrum*, juntamente com os itens correspondentes, cujos detalhes serão abordados de maneira mais aprofundada nas subseções subsequentes. Para uma representação visual mais clara, a Figura 2.4 ilustra o *Scrum* e seus componentes de forma gráfica. A análise conjunta da Tabela 2.3 e da Figura 2.4 permite identificar as atividades centrais intrínsecas ao *framework*, e correlacioná-las com seus componentes associados. As atividades fundamentais do *Scrum* incluem:

1. Preparação do *backlog* do produto;
2. Reunião de planejamento da *sprint* e preparação da *Sprint Backlog*;
3. *Sprint*;
4. Reunião diária;
5. Revisão da *Sprint* e apresentação de incremento;
6. Retrospectiva da *Sprint*.

Tabela 2.3: Componentes do *Scrum*.

| **Componente** | **Descrição dos itens** |
|---|---|
| Time | (i) Dono do produto, (ii) Time de desenvolvimento e (iii) *Scrum Master*. |
| Eventos | (i) *Sprint*, (ii) Reunião de planejamento da *Sprint*, (iii) Reunião diária, (iv) Revisão da *Sprint* e (v) Retrospectiva da *Sprint*. |
| Artefatos | (i) *Backlog* do produto, (ii) *Backlog* da *Sprint* e (iii) Incremento. |
| Regras | As regras do *Scrum* unem os papéis, eventos e artefatos, governando os relacionamentos e interação entre eles. |

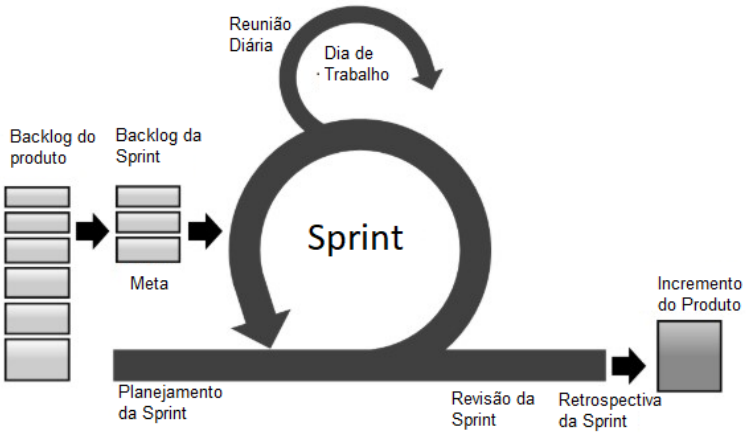


Figura 2.4: Arcabouço *Scrum*

Essas atividades representam uma estrutura simples e coesa que será utilizada no contexto deste trabalho para contextualizar os resultados descritos no Capítulo 9. Nos itens a seguir, um breve resumo de cada componente do *Scrum* é fornecido.

**Time *Scrum***. Os times *Scrum* são auto-organizados e multifuncionais [104][21] e são projetados para otimizar a flexibilidade, criatividade e produtividade. O time possui os principais papéis conforme descrito a seguir:

- *Dono do produto* é responsável por criar e priorizar o *Backlog* do produto, escolhendo o que será incluído na próxima *Sprint* e revisando o produto com outras partes interessadas sempre ao final de cada *Sprint* [104];

- *Scrum master* é responsável por garantir que os valores, práticas e regras do *Scrum* sejam entendidos e aplicados de modo correto. Este papel é responsável por facilitar os eventos *Scrum*, conduzindo diariamente as reuniões, revisando e avaliando a *sprint*.

Ele também é responsável por identificar dificuldades, treinar o time, mantendo-o motivado e focado na conclusão das tarefas e nas entregas;

- *Time de desenvolvimento* é responsável por projetar, construir e testar o produto desejado [45]. O time geralmente é composto entre cinco a nove pessoas e seus membros devem ter as habilidades necessárias para produzir software de qualidade. O time é auto-organizado visando permitir que os seus membros realizem todo o projeto, desenvolvimento e testes juntos. Os membros do time têm autoridade total para fazer o que for necessário para atingir a meta da *sprint*.

**Eventos *Scrum***. Eventos prescritos são usados para criar regularidade e minimizar a necessidade de reuniões não definidas no *Scrum* [98][104]. Todo evento possui uma janela de tempo que implica em uma duração máxima. Os principais eventos deste *framework* são apresentados a seguir:

- *Sprint* é considerada como o coração do *Scrum*. Consiste em um período de tempo, normalmente contido entre uma e quatro semanas, durante o qual é produzido um incremento do produto em desenvolvimento [98]. Durante este evento, a equipe deve ter total autoridade sobre suas tarefas e nenhuma influência externa (inclusive do dono do produto) é permitida [24];

- *Reunião de planejamento da sprint* é um evento onde o proprietário do produto e o time concordam com os objetivos definidos para a próxima iteração, definindo basicamente o que a próximo *sprint* deve alcançar [104]. Usando o objetivo da *sprint*, o time de desenvolvimento revisa o *backlog* do produto para identificar a prioridade mais alta dos itens que serão incluídos e realizados na próxima *sprint*;

- *Reunião diária*. Como o próprio nome sugere, são eventos que ocorrem diariamente para determinar o andamento das tarefas e para responder aos problemas encontrados ao longo de sua execução [104]. Trata-se de um evento de 15 minutos liderado pelo *Scrum Master*, que repete as mesmas três perguntas para cada membro do time todos os dias com intuito de avaliar o progresso do desenvolvimento: (i) O que você fez ontem?; (ii) O que você vai fazer hoje?; e (iii) Quais problemas enfrentou?;

- *Revisão da sprint* é um evento realizado sempre ao final da *sprint* para inspecionar o incremento e adaptar o *backlog* do produto, se necessário [104]. Na revisão do *sprint*, o time e as partes interessadas discutem sobre o trabalho que foi realizado na *sprint* e as tarefas que poderiam ser feitas. Neste evento, a apresentação do incremento destina-se basicamente para obter *feedback* e promover colaboração;

- *Retrospectiva da Sprint* é o evento que ocorre frequentemente após a revisão da *sprint* e antes do próximo planejamento da próxima *sprint* [104]. É útil para toda equipe *Scrum* inspecionar a si mesma e identificar as possíveis melhorias a serem consideradas durante a próxima *sprint* [98].

**Artefatos do *Scrum***. Os principais artefatos deste *framework* são: *backlog* do produto, *Sprint backlog* e incremento [104].

- *Backlog do produto*. Uma das atividades do dono do produto é gerar uma lista refinada e priorizada de tarefas que definem o produto [104]. Então, ele organiza tal lista como sendo o *backlog* do produto que evolui à medida que o produto evolui. Para o desenvolvimento contínuo do produto, o *backlog* do produto também pode conter novos recursos, alterações em recursos existentes, defeitos que precisam de reparo, melhorias técnicas e assim por diante [98];

- *Sprint Backlog*. É o conjunto de itens retirados do *backlog* do produto e que foram selecionados para a *sprint*, além de um plano para entregar o incremento do produto e realizar os objetivos da *sprint*. A *Sprint Backlog* é uma previsão da equipe de desenvolvimento sobre qual funcionalidade deverá ser entregue no próximo incremento e o trabalho necessário para entregar essa funcionalidade [104];

- *Incremento*: O incremento é o total de todos os itens do *backlog* do produto realizados durante a *sprint* atual e todas as *sprints* anteriores. Ao final de cada *sprint*, o dono do produto receberá uma entrega, e será capaz de ver o crescimento incremental do produto [104].

Um aspecto interessante relacionado ao incremento é a "definição de pronto" (do inglês *Definition of Done* [DoD]). A utilização deste conceito ajuda o time a elencar os pontos

necessários para que uma determinada tarefa seja classificada como concluída [107]. É importante mencionar que não existe uma definição de pronto que deve ser seguida por todas as organizações e equipes de desenvolvimento. Cada tarefa, item do *backlog*, *release* ou mesmo versões do software tem complexidades distintas e critérios de aceitação diferentes. Cada equipe deve desenvolver sua definição de pronto, levando sempre em consideração aspectos como desafio do desenvolvimento, qualidade do artefato, uso de ferramentas, necessidades dos usuários, dentre outros.

## 2.6 Considerações Finais do Capítulo

Este capítulo apresentou de forma detalhada os principais conceitos que servem de suporte para o presente trabalho. Definiu-se o conceito de dívida técnica, sua associação com as anomalias de código bem como as principais ações de refatoração requeridas para remoção das anomalias de código. Em seguida, as principais estratégias de detecção de anomalias, bem como as abordagens não-interativas e interativas foram descritas em detalhes. Adicionalmente, os principais conceitos associados aos estudos empíricos que serão apresentados ao longo do presente trabalho foram explicitados. Por fim, exibiu-se uma estrutura de gerenciamento de projetos ágeis comumente empregada para o controle das atividades de desenvolvimento de software e forneceu-se os primeiros detalhes da associação da detecção interativa com essa estrutura.

A análise dos trabalhos relacionados desempenha um papel crucial no avanço do conhecimento, permitindo contextualizar a pesquisa dentro do cenário existente. Essa exploração detalhada da literatura não apenas revela lacunas no estado da arte, mas também identifica oportunidades de contribuição e inovação. Ao conectar as descobertas dos trabalhos anteriores com os objetivos e resultados da pesquisa atual, é possível demonstrar como esta tese proporciona um avanço significativo, abrindo novos caminhos e aprimorando a compreensão no campo da detecção de anomalias de código e qualidade de software. Isso garante um embasamento sólido e embutido na evolução das pesquisas nessa área. Desse modo, no capítulo 3 será abordada uma análise abrangente dos estudos correlatos a esta tese, destacando como a presente pesquisa avança em relação ao estado atual do conhecimento no âmbito da detecção de anomalias de código e na melhoria da qualidade de software.

# Capítulo 3

# Trabalhos Relacionados

Neste capítulo são apresentados os trabalhos relacionados a esta tese. Na Seção 3.1 é exposta uma visão geral da pesquisa relacionada às anomalias de códigos. Na Seção 3.2 são descritos alguns trabalhos que evidenciam que a presença de anomalias impacta na qualidade do software. Na Seção 3.3 é dado um foco específico nas estratégias de detecção relevantes. Finalmente, na Seção 3.4 são discutidos alguns trabalhos com relação direta à abordagem de detecção interativa proposta neste estudo e na Seção 3.5 são apresentadas as considerações finais deste capítulo.

## 3.1 Anomalias de Código

As anomalias de código foram investigadas sob várias perspectivas [27], incluindo desde sua introdução [117], evolução [77], bem como o impacto na confiabilidade [90] [126] e facilidade de manutenção [51] [111]. Além disso, estudos relataram como os desenvolvedores percebem tais anomalias [77][91]. Vale destacar que nos últimos anos notou-se um crescente número de estudos secundários associados às anomalias de código [33], [42], [23], [27], [18], [2], e [29]. No que segue, descreve-se brevemente cada um desses estudos de acordo com sua cronologia.

Gupta *et al.* [42] realizaram uma Revisão Sistemática da Literatura (RSL) com intuito de fornecer uma visão geral das pesquisas existentes na área, identificando as técnicas de detecção e a correlação entre estas técnicas, além de destacar os tipos de anomalias que demandaram mais atenção nas abordagens de detecção. Cairo *et al.* [23] também realizaram

uma RSL visando identificar estudos que forneceram evidências da influência das anomalias de código na ocorrência de *bugs*. Este trabalho evidenciou que anomalias remanescentes em diversas versões do código apresentaram correlação direta com a presença de *bugs*.

Paulo Sobrinho *et al.* [27] conduziram uma extensiva RSL cobrindo décadas de pesquisa associada às anomalias de código. Os autores identificaram os tipos de anomalias mais pesquisadas bem como os pesquisadores mais relevantes nessa área de pesquisa. Os autores destacaram uma alta diversidade na gama de abordagens utilizadas nas configurações experimentais para lidar com anomalias. No entanto, muitos estudos não compartilharam as implementações de suas técnicas e abordagens, limitando sua utilização bem como a reprodutibilidade dos resultados. Azeem *et al.* [18] desenvolveram um estudo visando fornecer uma visão geral e discutir o uso de abordagens de aprendizado de máquina no suporte à detecção de anomalias de código. Entre as contribuições deste trabalho, destaca-se a apresentação de uma síntese abrangente dos estudos primários, incluindo quatro temas principais: (i) tipos de anomalias de código; (ii) configuração das abordagens de aprendizado de máquina; (iii) estratégias de avaliação; e (iv) análise de desempenho dos modelos propostos.

Abuhassam *et al.* [2] realizaram uma RSL para identificar estudos relacionados à detecção de anomalias de código. Com base em 145 estudos, os autores avaliaram várias questões relacionadas à análise das técnicas de detecção em termos de nível de abstração (*design* ou código), tipos de anomalias, métricas usadas, implementação e validação. Dos Reis *et al.* [29] desenvolveram um estudo visando identificar as principais técnicas e ferramentas de detecção de anomalias de código discutidas na literatura. Os autores analisaram em que medida as técnicas de visualização foram aplicadas para apoiar tais ferramentas.

Mais recentemente, alguns estudos terciários foram realizados com objetivo de mapear e organizar o conhecimento associado à área de anomalias de código [56][128]. Lacerda *et al.* [56] coletaram dados de vários estudos secundários com objetivo de identificar as principais observações e desafios ainda a serem explorados sobre anomalias de código e refatoração. Similarmente, Yaqoob *et al.* [128] cobriram e integraram informações associadas às anomalias tais como métodos e técnicas, formas de uso, desafios e avanços. Ademais, todos os estudos secundários e terciários descritos acima evidenciaram todo esforço de pesquisa realizada ao longo de quase três décadas desde que o termo “anomalias de código” foi cunhado por Fowler [38].

## 3.2 Impacto na Qualidade do Software

É unânime entre os pesquisadores da área que a presença e o acúmulo de anomalias de código tendem a degradar diversos atributos de qualidade de software. Nos trabalhos a seguir serão expostas evidências associadas a esta correlação negativa. Kaur *et al.* [49] conduziram uma RSL de estudos empíricos que investigaram o impacto das anomalias de código em atributos de qualidade de software (e.g., manutenibilidade, legibilidade e evolutividade). Os resultados deste estudo indicaram que o impacto das anomalias na qualidade do software não é uniforme, pois diferentes anomalias impactam diferentes atributos de qualidade. Semelhantes conclusões foram apresentadas no trabalho de Martins *et al.* [72]. Nesse estudo, os autores investigaram o impacto das ocorrências das anomalias em vários sistemas, sendo observado que anomalias de código tendem a ocorrer de modo conjunto e que existe maior impacto na remoção destas ocorrências conjuntas de anomalias de código nos atributos de qualidade.

Paulo Sobrinho *et al.* [27] e Lacerda *et al.* [56] realizaram estudos secundários sobre anomalias de código. Ambos estudos indicaram que a presença de anomalias de código pode levar a problemas de manutenção e evolução no software. Contudo, os autores apontaram que esta é uma área que ainda precisa de atenção e que mais estudos empíricos são requeridos para avaliar e mensurar o impacto das anomalias de código nestes atributos de qualidade. Por outro lado, Abbes *et al.* [1] estudaram as interações entre anomalias de código e seus efeitos, concluindo que anomalias quando apareciam em aglomerações tendiam a requerer mais esforço de manutenção. Fernandes *et al.* [32] realizaram um grande estudo quantitativo sobre o efeito de refatoração nos atributos de qualidade. Como resultado, os autores identificaram que a maioria das ações de refatoração tendiam a melhorar um ou mais atributos de qualidade.

Yamashita e Moonen [127] analisaram o impacto das relações entre anomalias na manutenibilidade de quatro sistemas industriais de médio porte escritos em Java. Os autores concluíram que as relações entre anomalias estão associadas a problemas durante as atividades de manutenção. Li e Shatnavi [59][106] investigaram a relação entre a probabilidade de erro em uma classe baseado na presença de anomalias de código utilizando experimentos em três versões do *Eclipse* IDE. Olbrich *et. al* [84][85] analisaram que as classes que estavam

infectadas com anomalias de código têm uma frequência maior de alterações e precisam de mais manutenção quando comparadas com as classes não infectadas. Conclusões similares foram obtidas nos estudos de Khomh *et. al* [50][51].

## 3.3 Estratégias de Detecção de Anomalias de Código

Detectar e identificar anomalias de código é uma tarefa desafiadora [33][2]. Para auxiliar os desenvolvedores nessa tarefa, diversas estratégias para detecção automática ou semiautomática foram propostas na literatura, particularmente nas últimas duas décadas [38]. A implementação dessas estratégias de detecção permite que as ferramentas destaquem as entidades que mais provavelmente apresentam fragmentos de código "anômalos". Felizmente, existem muitas ferramentas de análise de software disponíveis para detectar anomalias.

De acordo com Lacerda *et al.* [56], as ferramentas mais utilizadas para detecção de anomalias de código são: CCFinder [48], JCosmo [118], DECOR [76], inCode [69], StenchBlosson [79], IPlasma [25], JDeodorant[115] e JSpirit [122]. Além dessas ferramentas de código aberto, muitas outras ferramentas proprietárias se propõem a identificar e expor resultados associados a ocorrências de anomalias de código. PMD, Designite, InFusion, SonarQube e ReSharper são exemplos de ferramentas desse tipo [56].

Recentemente, alguns estudos secundários realizaram o mapeamento da pesquisa associada às estratégias de detecção. Fernandes *et al.* [2] identificaram 84 ferramentas distintas para detecção de anomalias de código. Tais ferramentas proviam suporte a detecção de um conjunto de 61 anomalias distintas, contando com pelo menos seis estratégias de detecção diferentes. As descobertas dos autores evidenciaram que as ferramentas forneciam resultados de detecção redundantes para o mesmo tipo de anomalia. Similarmente, Hadj-Kacem e Bouassida [43] realizaram uma avaliação das abordagens existentes em uma taxonomia representada por três dimensões diferentes: métodos utilizados, critérios de análise e avaliação. Além de seu valor de conhecimento, esta taxonomia pode sugerir oportunidades promissoras para a criação de novas abordagens ou através da combinação entre técnicas existentes.

Um aspecto relevante relacionado às estratégias de detecção tem íntima relação com a avaliação da sua eficácia. Diversos estudos avaliaram e demonstraram a capacidade de retornar problemas relevantes associados a anomalias de código. Paiva *et al.* [89] realizaram

uma avaliação empírica comparando diferentes técnicas de detecção de anomalias de código no contexto de dois sistemas de código aberto. Similarmente, Pecoreli *et al.* [95] propuseram um estudo em larga escala para comparar empiricamente o desempenho de técnicas baseadas em heurísticas em contraste com técnicas baseadas em aprendizado de máquina na atividade de detecção de anomalias de código. Ainda, Mumtaz *et al.* [78] avaliaram o impacto do uso de diferentes técnicas de visualização no suporte à detecção de anomalias de código.

É importante mencionar que a maioria dessas estratégias propostas na literatura focam na implementação da Detecção Não-Interativa (DNI) de anomalias de código [2][56][33][43]. Embora alguns estudos demonstrem a eficácia das estratégias de detecção através de medidas como *recall* e *precision* [92] [37] [115], a maior parte desses estudos não fornece uma análise mais aprofundada sobre a precisão dessas ferramentas, incluindo o número de falsos positivos observados quando diferentes técnicas de detecção são utilizadas. Além disso, nenhum desses estudos analisou se características da abordagem de Detecção Interativa (DI) têm a capacidade de impactar a detecção de anomalias de código, bem como na identificação de oportunidades de refatoração.

## 3.4 Limitado Conhecimento da Detecção Interativa

Inicialmente, serão expostas algumas técnicas que apresentam algumas das características da abordagem de detecção interativa e serviram de inspiração para o presente trabalho. Primeiramente, Murphy-Hill e Black [79] propuseram uma abordagem de detecção de anomalias que fornece uma visualização diferenciada dos resultados de detecção. Este ambiente foi projetado para fornecer aos desenvolvedores uma visão geral rápida e de alto nível das anomalias presentes no código, auxiliando na compreensão das suas origens. Os resultados do experimento controlado sugeriram que o uso da ferramenta não é intrusivo e provê meios adequados para detecção de anomalias e julgamento sobre ações de refatoração requeridas para resolução das anomalias.

Em seguida, Ganea *et al.* [40] propuseram uma ferramenta integrada ao *Eclipse* IDE para detecção de anomalias e sugestão de ações de refatoração. Esta ferramenta visa transformar a avaliação da qualidade e as inspeções de código de uma atividade autônoma através de um processo contínuo, totalmente integrado no ciclo de vida do desenvolvimento. Os autores

apresentaram os recursos da ferramenta, os objetivos do projeto bem como as decisões arquiteturais envolvidas na sua construção. Ainda, descreveram um experimento controlado projetado para validar a eficiência e usabilidade da ferramenta.

Por fim, Quang Do [28] apresentaram o conceito de análise estática *Just-In-Time* (JIT) que intercala o desenvolvimento de código e a correção de *bugs* em um ambiente de desenvolvimento integrado. Os autores relataram que ferramentas de análise JIT apresentam avisos aos desenvolvedores de código ao longo do tempo, fornecendo os resultados mais relevantes rapidamente e computando resultados menos relevantes incrementalmente mais tarde. Por fim, os autores descreveram as diretrizes gerais para projetar análises JIT e para transformar análises de fluxo de dados estáticos em análises JIT por meio de um conceito de execução de análise em camadas.

Embora algumas abordagens tenham sido propostas e avaliadas sob alguns aspectos, faz-se necessário compreender o impacto da detecção interativa e contínua na qualidade do código em desenvolvimento. Nesse sentido, Vassalo *et al.* [120] realizaram uma investigação empírica com o objetivo de compreender o quão rigorosamente os desenvolvedores seguem a Qualidade Contínua de Código (QCC). Com base no conjunto de dados sobrepondo informações históricas associadas às alterações no código provenientes do *SonarCloud* e *TravisCI*, os autores investigaram o impacto da prática da QCC na qualidade do software e as circunstâncias em que os desenvolvedores são particularmente encorajados a verificar a qualidade do código com mais frequência. O estudo revelou uma forte discrepância entre teoria e prática: os desenvolvedores não realizam inspeção contínua, mas sim o controle de qualidade apenas no final de uma *sprint* e, na maioria das vezes, apenas na *branch* de lançamento.

Similarmente, Saidani *et al.* [101] realizaram um estudo com objetivo de explorar e compreender a utilização da QCC e seu impacto nas práticas de refatoração em termos de frequência, tamanho e desenvolvedores envolvidos. Para isso, os autores coletaram quase 100 mil *commits* e cerca de 90 mil ações de refatoração extraídas a partir de 39 projetos de código aberto que adotam o *TravisCI*. Os autores apontaram que a adoção de QCC está associada a uma queda na frequência e tamanho da refatoração, indicando que é menos provável que a refatoração seja aplicada no contexto da QCC. Finalmente, os autores apontaram que desenvolvedores precisam de suporte automatizado mais personalizado no contexto da QCC para manter e desenvolver melhor seus sistemas.

Na medida do nosso conhecimento, apenas dois estudos avaliaram a utilização da abordagem de detecção interativa [79] [40]. No entanto, ambos estudos priorizaram aspectos relacionados às diretrizes de usabilidade (e.g., como disponibilidade e sensibilidade ao contexto) e aspectos associados à eficiência do uso da ferramenta (e.g., consumo de processamento computacional e memória). Consequentemente, eles não analisaram se a utilização de abordagens de detecção interativa promoveram melhor eficácia na atividade de detecção de anomalias de código e, consequentemente, na identificação de oportunidades para realizar ações de refatoração. Além disso, ambos estudos não apresentaram indicativos de como os desenvolvedores devem utilizar a abordagem de detecção interativa no contexto das atividades de um processo de desenvolvimento de software.

Nesse sentido, a presente tese se diferencia dos demais estudos relacionados em virtude das seguintes contribuições esperadas: (i) identificar e relacionar as características que diferenciam as abordagem DI e DNI; (ii) propor um suporte ferramental automatizado aderente às características da abordagem DI; (iii) avaliar a utilização desta abordagem automatizada de DI em contraste com a abordagem de DNI com intuito de avaliar seu impacto na detecção de anomalias de código; e (iv) avaliar a utilização da abordagem DI no contexto das atividades do processo ágil de desenvolvimento de software.

## 3.5 Considerações Finais do Capítulo

Conforme visto ao longo deste capítulo, as anomalias de código têm sido pesquisadas pela comunidade de pesquisa em engenharia de software nas últimas décadas. Diversos estudos foram publicados sobre anomalias, discutindo suas implicações na qualidade do software, seu impacto na manutenção e evolução, bem como na proposição de diferentes abordagens para sua correta detecção e remoção. De fato, estudos anteriores apresentaram evidências de que a presença de anomalias de código em um dado programa pode prejudicar sua compreensão [1], degradar a arquitetura [83], aumentar sua propensão a falhas [116] e impactar na manutenibilidade do software [90].

A maioria dos estudos sobre a detecção de anomalias são estritamente focados na avaliação de abordagens não-interativas [102] [110] [37]. Esses estudos apontaram que as abordagens DNI induzem a um baixo número de anomalias de código detectadas corretamente,

impactando diretamente em sua eficácia. A realização das ações de refatoração depende principalmente da eficácia das técnicas de detecção de anomalias de código. Nesse sentido, estudos preliminares [108] [103] expuseram consequências negativas na qualidade do código sempre que ações ineficazes e tardias de refatoração foram realizadas. Portanto, os desenvolvedores precisam identificar as instâncias de anomalias de código de forma mais eficaz e oportuna.

A abordagem de detecção interativa de anomalias de código tem sido apontada como uma possível forma de aumentar a longevidade de sistemas de software [79] [7]. Com a adoção da abordagem DI, os desenvolvedores conseguem identificar oportunidades de refatoração assim que as anomalias são introduzidas no código-fonte [38] [80] [77]. É importante destacar que, quanto mais tempo essas anomalias permanecem no software, mais difícil e demorada se torna sua remoção por meio das ações de refatoração.

Baseado na discussão apresentada no presente capítulo, identificou-se que existe um baixo conhecimento empírico a respeito do impacto da abordagem de detecção interativa na atividade de detecção de anomalias de código. O primeiro passo para endereçar este problema consiste na identificação da forma como os desenvolvedores profissionais realizam a identificação e gerência das anomalias de código em seus sistemas de software. Assim, o capítulo 4a presentará um *survey* realizado com profissionais para levantar a real necessidade de uso da abordagem de detecção interativa no contexto das atividades de desenvolvimento de software.

# Capítulo 4

# Motivação para a Abordagem de Detecção Interativa

Antes de entender a abordagem de Detecção Interativa (DI), é importante entender o processo que motivou a concepção de um trabalho nesta área. Neste capítulo são apresentados os resultados de um *survey* com profissionais de diversas organizações de software no intuito de verificar como ocorre a identificação e gerenciamento de itens de dívida técnica (DT), incluindo as anomalias de código. Este estudo foi realizado dentro de um contexto mais amplo de DT, mas o foco deste trabalho foi entender se realmente existe a demanda por uma abordagem para dar suporte à detecção interativa de anomalias de código. Ademais, os resultados desse estudo foram publicados no Simpósio Brasileiro de Engenharia de Software (SBES'22)[14].

Este capítulo está organizado da seguinte forma: na Seção 4.1 são descritos os detalhes metodológicos da pesquisa. Na Seção 4.2 são apontados os principais resultados associados à pesquisa. Na Seção 4.3 são expostas algumas conclusões obtidas a partir dos resultados deste estudo. Na Seção 4.4 são detalhadas as principais ameaças à validade deste estudo. Por fim, na Seção 4.5 são apresentadas as considerações finais do estudo.

## 4.1 Metodologia do Estudo

Utilizando o paradigma *Goal-Question-Metric* (GQM) [119], o objetivo deste estudo foi analisar o conceito mais amplo de Dívida Técnica (DT) e sua gestão, com o objetivo de ca-

racterizar, quanto ao nível de conhecimento e as estratégias, atividades e tecnologias, a partir do ponto de vista dos praticantes de software, no contexto das organizações de software. Embora o objetivo do estudo tenha sido uma análise mais ampla sobre a gestão de práticas associadas à DT, os dados obtidos a partir deste estudo darão suporte para responder a seguinte Questão de Pesquisa (QP):

- **QP1.** Existe a necessidade de uma abordagem para dar suporte à detecção interativa de anomalias de código?

### 4.1.1 Projeto do Questionário

Um questionário foi elaborado de acordo com as diretrizes apresentadas no trabalho de Linåker *et al.* [61]. As atividades associadas à gestão da DT foram organizadas conforme taxonomia proposta por Li *et al.* [60]. Para cada atividade, identificou-se um conjunto de estratégias específicas e tecnologias utilizadas para sua execução. A partir desta informação, um questionário foi elaborado e perguntas específicas foram incluídas para cada atividade da gestão de itens de DT.

O questionário foi dividido em 14 seções distintas e organizadas de acordo com a similaridade. As seções foram compostas principalmente por perguntas fechadas. Um pequeno número de perguntas abertas foi necessário para obter mais informações do participante. Perguntas parcialmente fechadas também foram incluídas para lidar com questões relacionadas a ferramentas, bem como as estratégias associadas a cada uma das atividades da gestão da DT.

Na Tabela 4.1, é exibido um extrato do questionário contendo um conjunto selecionado de perguntas associadas à identificação da Dívida Técnica (DT). Cada seção é introduzida com uma breve explicação sobre o conteúdo abordado, acompanhada por instruções específicas. A operacionalização desse questionário foi realizada por meio da plataforma *Google Forms*, assegurando a confidencialidade dos participantes. Estes participantes foram requisitados a responder às perguntas com base em suas experiências nos projetos de software atuais ou mais recentes na respectiva organização. O questionário completo encontra-se disponibilizado no material suplementar deste estudo [5].

Em relação a cada uma das atividades envolvidas na gestão da DT, procurou-se compre-

Tabela 4.1: Extrato da Seção do Questionário.

| **Questão** | **Opções de Resposta** |
|---|---|
| Você realiza a identificação de itens de DT? | ( ) Sim, de modo FORMAL. |
| | ( ) Sim, de modo INFORMAL. |
| | ( ) Não, a atividade não é considerada. |
| A estratégia de identificação da DT é aplicada obrigatoriamente por todos os envolvidos nesta atividade? | ( ) Sim, a aplicação da estratégia de identificação da DT é obrigatória para todos os envolvidos. |
| | ( ) Não, a aplicação da estratégia de identificação da DT é opcional para os envolvidos. |
| Marque na lista abaixo todas as soluções que são usadas para a identificação da DT | [ ] Inspeção manual do código |
| | [ ] Análise de dependências |
| | [ ] Análise estática de código (e.g., sonar) |
| | [ ] *Checklist* |
| | [ ] *Findbugs* |
| | [ ] *CodeVizard* |
| | [ ] *Clio* |
| | [ ] Nenhuma opção |
| | [ ] Outra opção |
| Em sua percepção profissional, como você acredita que deveria ser realizada a identificação da DT? | ( ) De modo CONTÍNUO, tão cedo quanto fosse adicionada a DT deveria ser identificada. |
| | ( ) NÃO-CONTÍNUO, pelo menos uma vez a cada *sprint* devemos realizar a identificação da DT. |
| | ( ) NÃO-CONTÍNUO, a identificação da DT deveria ocorrer apenas sob demanda. |

ender o método pelo qual essas atividades são executadas. Tais atividades podem acontecer de maneira formal ou informal, dependendo das particularidades do contexto e das exigências do projeto em questão. No “modo formal”, as atividades são executadas de forma estruturada, sendo empregas técnicas específicas para analisar e definir as tarefas envolvidas para execução sistemática da atividade. Em contrapartida, no “modo informal”, a execução da atividade acontece de maneira mais flexível e intuitiva, muitas vezes guiada pela intuição e pela experiência acumulada por parte dos envolvidos. Além disso, existe a opção de “não realizar a atividade”, caso sua relevância para o processo de gestão da DT seja considerada mínima.

### 4.1.2 Execução do Piloto

Um teste piloto foi conduzido usando os mesmos artefatos e procedimentos projetados para a pesquisa com um pequeno número de participantes [61]. Seis praticantes foram convidados para os testes-piloto. Todos os participantes possuíam experiência prévia com DT, principalmente na indústria. Os participantes foram convidados a responder o questionário e retornar seus comentários sobre o tempo de resposta, compreensão adequada das perguntas entre outros aspectos. Todos os participantes do teste-piloto responderam à pesquisa dentro de um período de uma semana. O tempo médio de resposta foi de 25 minutos. As críticas e sugestões dos participantes foram posteriormente discutidas, e modificações foram aplicadas ao questionário final. No geral, não observou-se comentários negativos ou dúvidas sobre as opções de respostas ou as descrições das perguntas, sugerindo que o questionário tinha qualidade suficiente para ser empregado no estudo.

### 4.1.3 Amostra e Coleta de Dados

Para atingir os objetivos da pesquisa, os praticantes de indústria de software foram selecionados como público-alvo do estudo. O desenho amostral adotado na pesquisa é acidental e não-probabilístico (i.e., não se pode observar aleatoriedade nas unidades selecionadas da população). Um convite direto para responder à pesquisa foi enviado através de listas de *e-mails* que foram encaminhadas para uma série de empresas e grupos de pesquisa na área de desenvolvimento de software. Esta pesquisa contou com apoio de outros pesquisadores que fizeram uso de sua rede primária de contatos no intuito de maximizar o número de respondentes desta pesquisa. Outros convites foram enviados por meio da rede social profissional *LinkedIn*.

## 4.2 Resultados do Questionário

A pesquisa foi realizada no período compreendido entre outubro de 2020 e setembro de 2021. No total, 138 participantes responderam à pesquisa, com 120 respostas completas. Dentre as 18 respostas que não foram incluídas na análise, houve três desistências e outras 15 respostas incompletas. A seguir, serão apresentados os resultados associados ao perfil dos

participantes, a percepção da DT, bem como o modo de gestão dos itens da DT.

### 4.2.1 Caracterização da Amostra

*Perfil dos Participantes*. Mais de 50% dos entrevistados possuíam faixa etária compreendida entre 30 e 39 anos. Apenas quatro entrevistados relataram ter graduação incompleta, enquanto os 116 respondentes restantes possuem pelo menos uma graduação completa na área de computação e afins. Cerca de 60% dos participantes têm uma experiência média de trabalho de mais de cinco anos em projetos de software. Menos de 15% dos participantes indicaram ser "novatos" ou "iniciantes" em termos de proficiência em desenvolvimento de software. Finalmente, cerca de 70% dos participantes afirmaram ter o perfil de desenvolvedor, líder técnico ou arquiteto de software.

*Perfil das Organizações*. A maioria destas organizações (cerca de 55%) eram dos setores de (i) Tecnologia da Informação; (ii) Pesquisa, Desenvolvimento e Inovação; ou (iii) Finanças. Cerca de 60% das organizações possuíam até 500 colaboradores e apenas 25% das organizações possuíam mais de 2000 colaboradores. Finalmente, a maior parte destas organizações (70%) possuíam times de até 10 membros. Uma minoria das organizações apresentava times com mais de 21 pessoas (menos de 15%).

*Perfil dos Projetos*. A maioria dos projetos eram *web* (cerca de 85%) e *mobile* (cerca de 40%). Varejo, finanças ou sistemas de gestão empresarial eram os domínios mais recorrentes (cerca de 45%). Quase 75% dos projetos possuíam mais de um ano e apenas 15% possuíam mais de 10 anos de existência. Finalmente, 90% dos projetos utilizavam abordagens ágeis ou híbridas para gestão do desenvolvimento.

Devido ao anonimato do questionário, não foi possível identificar precisamente o número de organizações representadas na pesquisa. No entanto, foi possível estimá-lo de forma aproximada, com base nas informações fornecidas pelos participantes. Portanto, estima-se que cerca de 72 organizações e 97 projetos foram incluídos na pesquisa.

### 4.2.2 Percepção da Dívida Técnica

Em relação ao conhecimento sobre DT, a partir das 120 respostas válidas, foi identificado que apenas 12 entrevistados (10%) afirmaram não estar cientes da metáfora da DT. Do uni-

verso de 108 participantes que afirmaram conhecer o conceito da DT, eles tiveram contato com o conceito da DT a partir de experiências prévias em projetos de desenvolvimento de software (65%) e através de *blogs* e fóruns de discussão relacionados a desenvolvimento (52%). Finalmente, o conceito da DT proposto por Avgeriou *et al.* [17] foi exposto aos participantes. Para quase 86% dos participantes o conceito estava "próximo" ou "muito próximo" do entendimento deles.

Em relação à percepção da DT, a maior parte dos respondentes (82%) associaram a presença das anomalias de código como a principal manifestação da DT em seus projetos de desenvolvimento de software. Finalmente, a definição imprecisa dos requisitos e a baixa cobertura de testes foram as formas mais recorrentes de manifestação da DT para 74% e 72% dos respondentes, respectivamente.

### 4.2.3 Gestão da Dívida Técnica

Na Figura 4.1 resumem-se as respostas do questionário em termos percentuais, indicando a quantidade de participantes que realizam as atividades da gestão da DT de modo *Formal* (na cor azul), *Informal* (na cor vermelho) e que deliberadamente não realizam qualquer atividade (na cor amarelo) associada à gestão da DT.

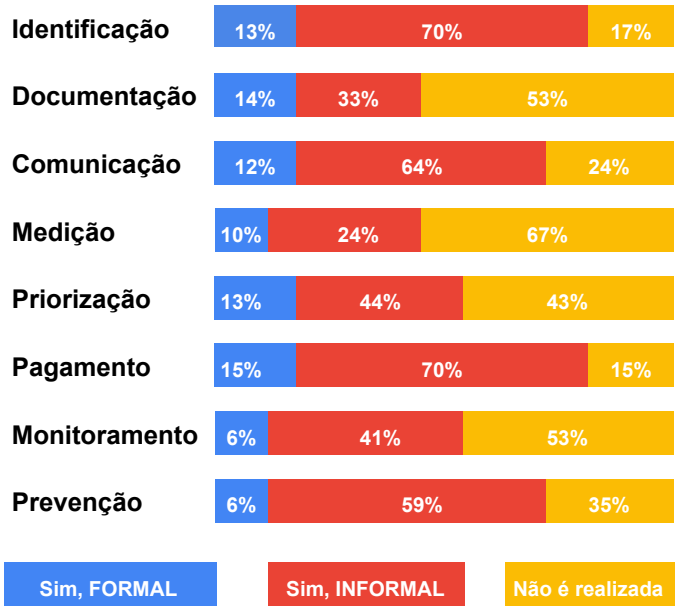


Figura 4.1: Visão Geral da Gestão da DT.

Notou-se que todas as atividades de gestão da DT eram realizadas pelos participantes, evidenciando sua importância e relevância. Identificação e Pagamento da DT foram as atividades mais recorrentemente realizadas pelos respondentes desta pesquisa, sendo a primeira atividade de interesse maior deste estudo. Notou-se que para cerca de 70% dos respondentes a identificação da DT é realizada apenas de modo *Informal* (i.e., sem um processo minimamente prescrito para realização da atividade). Outros 17% indicaram que esta atividade não era realizada. Para cerca de 73% dos respondentes a aplicação da estratégia de identificação da DT é opcional para os envolvidos na atividade. Relacionado às ferramentas e técnicas para identificação da DT, a inspeção manual de código (83%), análise estática (62%) e análise de dependências (42%) foram as mais recorrentemente citadas.

A respeito do momento em que a DT é identificada, notou-se que para 62% dos respondentes a atividade de identificação não ocorre em momentos "definidos". Essa amostra identifica itens de DT apenas esporadicamente ao longo do projeto ou sempre que eles percebem algum problema nos artefatos produzidos, sendo o código-fonte o mais citado. Adicionalmente, foi dada a oportunidade dos respondentes exporem a forma como eles acreditam que seria a "ideal" para realização da identificação da DT. Para 86% dos respondentes, a identificação da DT deveria ocorrer de modo contínuo (i.e., tão cedo quanto fosse adicionado o item de DT, o mesmo deveria ser identificado e gerenciado).

Embora os participantes acreditem que os itens de DT devam ser continuamente identificados e gerenciados, para cerca de 70% da amostra de respondentes, as ferramentas e técnicas do estado da arte não fornecem suporte adequado a essa estratégia. Entre os principais benefícios obtidos com a detecção contínua estão o não acúmulo de itens de DT (78%), maior facilidade em termos de tempo e esforço para refatoração desses itens (69%) e maior transparência sobre o nível de qualidade dos artefatos produzidos (67%).

Embora a atividade de identificação tenha recebido elevado nível de interesse por parte dos respondentes, cerca de 70% dos respondentes afirmaram não realizar qualquer classificação dos itens de DT após esta atividade. Para 88% dos respondentes, as anomalias de código são os tipos mais recorrentes de itens de DT identificados e gerenciados em seus projetos. Cerca de 65% dos respondentes identificam esse tipo de DT de modo "frequente" ou "muito frequente".

Na questão aberta associada à identificação de itens de DT, a maior parte dos responden-

tes afirmaram que realizam a identificação de itens de DT mais recorrentemente durante o desenvolvimento de novas funcionalidades. Normalmente isso é realizado ao final de ciclos de desenvolvimento (e.g., *sprints*) através de ferramentas de análise estática, inspeções manuais ou revisões de código. Os respondentes confirmaram que dão maior interesse aos itens de DT associados ao código (i.e., anomalias de código).

## 4.3 Discussão

No que segue, discutem-se aspectos relacionados a percepção e conhecimento sobre a DT (Seção 4.3.1), as atividades de gestão da DT (Seção 4.3.2), as soluções Adotadas na gestão da DT (Seção 4.3.3), bem como as causas dos itens de DT (Seção 4.3.4).

### 4.3.1 Percepção e Conhecimento Sobre a DT

Embora a maior parte dos respondentes afirmaram perceber a DT em seus projetos de software, notou-se uma falta de disciplina nas atividades de gestão. Menos de 20% dos participantes afirmaram adotar atividades formais para mais de uma atividade de gestão de DT, possivelmente indicando a existência de uma grave lacuna na perspectiva geral da qualidade do produto.

Agrupando as atividades de gestão mais importantes do ponto de vista dos respondentes, foi possível identificar os principais sintomas associados aos principais tipos de DT mencionados pelos respondentes. A análise em questão é descrita em detalhes na Tabela 4.2:

Tabela 4.2: Sintomas Associados aos Tipos de DT.

| **Tipos de DT** | **Sintomas Associados** |
|---|---|
| *Design* | Anomalias de código (79%) |
| | Classes e métodos complexos (64%) |
| | Mudanças Frequentes (45%) |
| Código | Código duplicado (67%) |
| | Código complexo (58%) |
| | Baixa qualidade do código (58%) |

Foi notado também que o modo de gestão dos itens de DT varia conforme o tamanho das organizações. Observou-se que um maior percentual de participantes de organizações

com maior número de colaboradores realiza gestão de modo mais disciplinado e formal quando comparado às organizações de menor porte. Isso pode indicar que organizações de maior porte (geralmente sendo ativas por um período mais longo e tendo processos mais sólidos para gerenciar projetos de software) poderiam ter uma perspectiva mais ampla sobre o conceito e a gestão da DT. Achados similares foram obtidos em outros estudos relacionados à presente pesquisa [129] e [26].

Finalmente, pôde-se identificar que os participantes de fato dispensam mais atenção e cuidado à manifestação de itens de DT associados ao código-fonte do sistema em desenvolvimento. As anomalias de código são um dos principais indicadores de DT manifestado no código-fonte. Desse modo, estratégias para correta identificação e gestão destes itens de DT são essenciais do ponto de vista dos desenvolvedores. Esses achados têm relação direta com o trabalho de Hozano *et al.* [46] onde seus resultados apontaram que a manutenção do software é uma atividade difícil de realizar em virtude da falta de artefatos bem documentados (e.g., documento de requisitos, descrição arquitetural, arcabouço de testes, entre outros). Assim, o código-fonte torna-se uma das poucas fontes confiáveis de informações sobre um sistema.

### 4.3.2 Atividades de Gestão da DT

Os resultados desta pesquisa mostraram que não houve consenso em relação às atividades de gestão da DT mais relevantes para os projetos de software pesquisados. A respeito dos resultados associados a estas atividades, os resultados aqui descritos estão de acordo com os trabalhos de Yli-Huumo *et al.*[129] e Da Silva *et al.* [26], que indicaram que a identificação é mais comumente adotada pelas equipes de desenvolvimento, seguida por pagamento e priorização.

Outra análise tem relação direta com o nível de importância que cada uma das atividades de gestão da DT tem do ponto de vista dos respondentes. Na Tabela 4.3, apresentam-se os resultados das seis atividades consideradas mais importantes pelos participantes desta pesquisa. Nota-se que as atividades de identificação e pagamento estão de fato alinhadas com os resultados associados às atividades de gestão da DT (subseção 4.2.3).

Tabela 4.3: Nível de Importância das Atividades de Gestão da DT.

| Atividade | Importância | % | Atividade | Importância | % |
|---|---|---|---|---|---|
| Identificação | Essencial | 56 | Pagamento | Essencial | 50 |
| | Muito Importante | 28 | | Muito Importante | 23 |
| | Importante | 12 | | Importante | 20 |
| | Pouco Importante | 2 | | Pouco Importante | 5 |
| | Dispensável | 2 | | Dispensável | 2 |

### 4.3.3 Soluções Adotadas na Gestão da DT

Na Tabela 4.4 sumarizam-se as soluções (e.g. ferramentas, técnicas ou métodos) mais comumente utilizadas pelos desenvolvedores para cada uma das atividades da gestão dos itens de DT.

Tabela 4.4: Soluções Adotadas nas Atividades de Gestão da DT.

| Atividade | Soluções Adotadas |
|---|---|
| Identificação | Inspeção manual do código (83%), Análise estática (62%), Análise de dependências (41%), *CheckList* (21%) e *CheckStyle* (19%). |
| Documentação | Lista ou *backlogs* (62%), *Issue Tracker* (62%), Ferramenta de Gestão (24%), Planilhas (16%) e Wikis (8%). |
| Comunicação | Reuniões de Projeto (46%), Reuniões específicas (37%), Listas de DT (35%), Fóruns de discussão (25%) e Ferramentas de Gestão (18%). |
| Medição | *Issue Tracker* (56%), Análise estática (48%), Manualmente (33%), Ferramenta de gestão (12%) e Wikis (7%). |
| Priorização | Análise estática (54%), *Issue Tracker* (48%), Manualmente (38%), Reuniões (34%) e Ferramentas de Gestão (14%). |
| Pagamento | Refatoração (95%), Reescrita de código (78%), *Redesign* arquitetural (53%), Reuniões (27%) e Alteração de artefatos (15%). |
| Monitoramento | Manualmente (74%), Análise estática (66%), *Issue Tracker* (47%), Ferramenta de teste (26%) e Wikis (21%). |
| Prevenção | Revisões de código (88%), Padrões de codificação (81%), Testes automatizados (67%), *Guidelines* (62%) e Reuniões (54%). |

Conforme resultados de estudos anteriores [129] [26], notou-se que existe uma predominância da utilização de ferramentas de análise estática para identificação, bem como outras

três atividades da gestão da DT. Sabe-se que essas ferramentas realizam análise das estruturas do código-fonte para identificar e alertar sobre prováveis inconsistências. Isso pode indicar que esse tipo de ferramenta pode ser importante para correta gestão de itens de DT, principalmente aqueles associados ao código-fonte, que foi o artefato mais recorrentemente citado pelos respondentes.

Outro aspecto relevante sobre o uso desse tipo de solução é que ela deve prover suporte para identificação contínua e direta dos itens de DT. Embora cerca de 85% dos respondentes apontaram que a identificação da DT deveria ocorrer de modo *contínuo* (i.e., tão cedo quanto fosse adicionado o item de DT o mesmo deveria ser identificado), o que ocorre na realidade é que para mais de 70% dos respondentes não ocorrem momentos pré-definidos ou são identificadas apenas tardiamente quando se é notado algum problema. Uma das causas para isso é que as soluções do estado da arte não são adequadas à detecção contínua de itens de DT.

### 4.3.4 Causas dos Itens de DT

Com base nas experiências prévias dos participantes, perguntou-se quais das causas mais contribuem para ocorrência de itens de DT. A *pressão do tempo* foi a causa mais apontada pelos respondentes (82%). Muitos deles citaram que existia uma necessidade de entregas em períodos de tempo cada vez mais curtos. Isso impactou significativamente na qualidade dos artefatos produzidos. A *falta de boas práticas de programação* (72%) também foi citada como causa frequente da incidência de itens de DT. Os respondentes associaram o mau uso dos princípios da orientação a objeto bem como a falta de uso de padrões de projeto como fatores que promoviam a ocorrência de anomalias de código. Adicionalmente, eles citaram que a ausência de boas práticas degradava algumas métricas de software (e.g., tamanho dos métodos, tamanho das classes) que implicavam também em anomalias de código.

Finalmente, para mais de 50% dos respondentes a *falta de experiência* e o *uso de novas tecnologias* também contribuem significativamente para existência da DT. Em ambos casos, os respondentes citaram que estas causas estavam intimamente associadas à falta de conhecimento técnico para uso de uma determinada tecnologia (e.g., linguagem de programação, bibliotecas, *plugins*). Isso poderia implicar em mau uso desta tecnologia e provocar itens de DT em decorrência desse contexto de falta de conhecimento técnico.

## 4.4 Ameaças à Validade

Esta pesquisa possui algumas ameaças à validade discutidas de acordo com o arcabouço de classificação proposto por Wohlin *et al.* [125], juntamente com algumas ações adotadas com intuito de mitigar os seus efeitos.

*Validade Interna*. Uma potencial ameaça tem relação com os participantes e o mau entendimento dos termos e conceitos do questionário. Com o objetivo de reduzir os efeitos dessa ameaça, os enunciados das questões, bem como seus termos e conceitos, foram aprimorados ao longo da realização dos testes-piloto. Cabe ressaltar que no início de cada seção eram expostos os principais termos necessários ao entendimento das questões, bem como as conceituações cabíveis.

*Validade de Construção*. Há também uma ameaça de uma pesquisa tendenciosa, da perspectiva dos pesquisadores e das informações coletadas da literatura técnica, como as atividades da gestão de DT organizadas descritas em Li *et al.* [60]. Para reduzir o impacto desta ameaça, foram conduzidos ciclos de revisão durante o desenvolvimento da pesquisa com pelo menos dois pesquisadores. Além disso, testes-piloto foram executados, seguidos de uma revisão final por todos os participantes dos testes-piloto.

*Validade Externa*. Observou-se uma ameaça relacionada com a representatividade e baixa adesão dos respondentes convidados para a pesquisa. Como parte da estratégia de divulgação envolvia apresentar o formulário a grupos de pesquisa, redes de contatos dos pesquisadores e na rede profissional *LinkedIn*, alguns de nossos resultados podem apresentar algum tipo de viés. Para minimizar os efeitos desta ameaça, buscou-se assegurar que a divulgação tivesse alcance tanto para pesquisadores quanto profissionais da indústria, evitando focar apenas em um segmento e equilibrando as oportunidades de participação. Outra ameaça pode estar associada ao tamanho do questionário. Para minimizar os efeitos dessa ameaça, buscou-se otimizar as estruturas condicionais das perguntas para minimizar a extensão do questionário, garantindo que apenas as questões pertinentes a cada respondente sejam apresentadas e respondidas.

*Validade de Conclusão*. Uma vez que a amostra alvo não é probabilística, não é possível determinar “a priori” o tamanho da população, bem como o número total esperado de participantes. Portanto, o nível de confiança dos resultados pode ser baixo, dificultando a

generalização dos resultados para toda a população associada às organizações de software. Contudo, os procedimentos metodológicos foram utilizados desde a fase de planejamento deste estudo até a sua execução, visando reduzir o nível de tal ameaça. Adicionalmente, buscou-se uma diversidade na amostra de respondentes com objetivo de fortalecer a integridade e a representatividade dos resultados da pesquisa.

## 4.5 Considerações Finais do Capítulo

Neste capítulo, foram apresentados os resultados de uma pesquisa que investigou como profissionais de diversas organizações de software percebem e gerenciam os itens de DT em seus projetos. A partir das conclusões delineadas no estudo, é possível deduzir de maneira inequívoca que emerge a necessidade por uma abordagem que ofereça assistência à detecção interativa de anomalias de código. Diversos aspectos abordados no estudo corroboram com essa demanda:

- *Identificação de Anomalias de Código como Principal Manifestação da DT*: Os resultados indicam que a maioria dos respondentes associou a presença de anomalias de código como a principal manifestação da DT em seus projetos de desenvolvimento de software. Isso ressalta a importância da detecção e gestão adequadas dessas anomalias;

- *Ferramentas de Análise Estática*: O estudo mostrou que as ferramentas de análise estática são comumente utilizadas para a identificação de anomalias de código. Isso sugere que as equipes reconhecem a importância de utilizar ferramentas automatizadas para auxiliar na detecção precoce de problemas no código;

- *Detecção Contínua*: Embora a maioria dos respondentes tenha expressado o desejo de identificar itens de DT de forma contínua, muitos afirmaram que a identificação ocorre apenas esporadicamente ou quando problemas são notados. Isso destaca uma lacuna na forma como a identificação é atualmente abordada e aponta para a necessidade de uma abordagem que permita a detecção contínua e interativa de anomalias de código à medida que elas são adicionadas ao sistema;

- *Importância da Identificação e Pagamento*: A identificação e o pagamento de itens de DT foram identificados como atividades essenciais ou muito importantes por uma

parcela significativa dos participantes. Isso reforça a necessidade de uma abordagem que forneça suporte eficaz para a detecção e ação sobre esses problemas;

- *Causas da DT*: A pressão do tempo e a falta de boas práticas de programação foram citadas como causas frequentes da DT. Uma abordagem que permita a detecção interativa de anomalias de código poderia auxiliar as equipes a lidar com essas causas, identificando problemas à medida que surgem e possibilitando ações corretivas imediatas.

A partir destes indícios, torna-se evidente que uma abordagem que agilize a detecção interativa de anomalias de código traria consideráveis vantagens às equipes de desenvolvimento de software. Os desfechos extraídos do estudo forneceram a sustentação necessária para responder à QP1, conforme ilustrado no quadro abaixo.

**QP1 - Existe a necessidade de uma abordagem para dar suporte à detecção interativa de anomalias de código?**
Com base nos resultados do estudo, é evidente a necessidade de uma abordagem que ofereça suporte à detecção interativa de anomalias de código. A identificação frequente de anomalias de código como principal manifestação da DT, o uso comum de ferramentas de análise estática e a importância destacada da identificação e pagamento da DT indicam que uma abordagem que permita a detecção contínua e ação imediata sobre problemas no código seria altamente benéfica.

De forma geral, a abordagem DI poderia desempenhar um papel crucial na identificação contínua de problemas no código, sendo incorporada nas atividades de desenvolvimento para fornecer um *feedback* imediato aos desenvolvedores. Essa integração facilitaria a melhoria da qualidade do código, a redução da acumulação das manifestações de DT no código-fonte e o enfrentamento de causas destacadas no estudo, como a pressão do tempo e a falta de adesão às boas práticas de programação. O passo seguinte para avaliar a abordagem DI é identificar os métodos de detecção do estado da arte que melhor se alinham a esta abordagem. Assim, o Capítulo 5 irá apresentar um mapeamento mais abrangente desses métodos de detecção e, através de uma análise criteriosa, destaca aqueles mais adequados para apoiar a implementação da abordagem DI.

# Capítulo 5

# Avaliação de Mecanismos para Abordagem de Detecção Interativa

No Capítulo 4 foi apresentada a motivação para a abordagem de Detecção Interativa (DI). Neste capítulo, apresenta-se o primeiro passo para a construção da abordagem: identificar os mecanismos de detecção de anomalias de código adequados à DI. Para isso, foi realizado um estudo com o objetivo de mapear os métodos de detecção do estado da arte com objetivo de apontar qual destes é mais adequado para prover suporte à abordagem DI.

Inicialmente são descritos os passos metodológicos do estudo (Seção 5.1). Em seguida, são demonstradas as categorias de métodos de detecção (Seção 5.2), os processos empregados para avaliação (Seção 5.3) e a escolha do método mais adequado à abordagem DI (Seção 5.4). Por fim, são apresentadas as considerações finais sobre esta avaliação (Seção 5.5).

## 5.1 Configuração do Estudo

A realização deste estudo tem por objetivo responder à seguinte Questão de Pesquisa (QP):

- **QP2 - Quais métodos de detecção se adéquam à abordagem de detecção interativa?**

Para guiar os esforços de pesquisa serão seguidas algumas etapas que encontram-se ilustradas no diagrama da Figura 5.1.

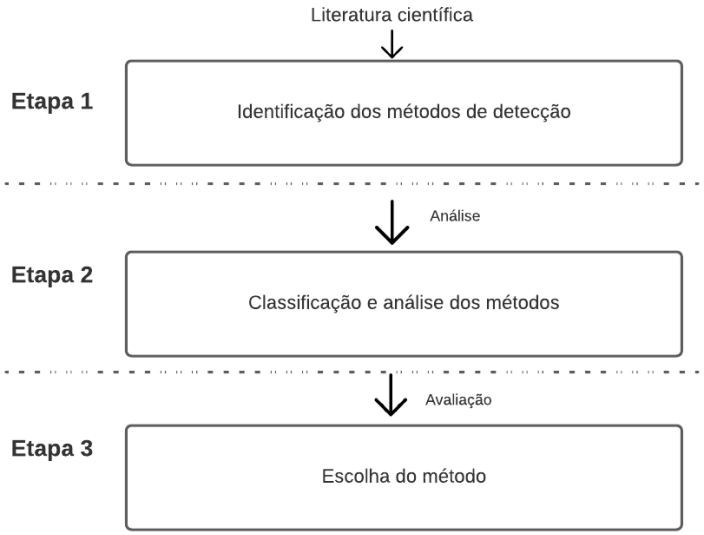


Figura 5.1: Etapas para Avaliação e Escolha do Método de Detecção.

Estas etapas foram planejadas para execução sequencial de modo que as saídas de uma etapa anterior são utilizadas como entradas da etapa subsequente. Na *Etapa 1* foi realizada uma revisão da literatura para identificação e catalogação dos métodos de detecção mais relevantes. Na *Etapa 2* foi realizada a classificação e análise dos métodos existentes com base em suas características e similaridades. Finalmente, na *Etapa 3* foi realizada a escolha do método de detecção mais aderente à abordagem DI. Para dar suporte a esse processo decisório, especialistas na área de detecção de anomalias de código e refatoração criaram uma lista de critérios e, em seguida, avaliaram os métodos de detecção com base nesses critérios. Cada uma dessas etapas é descrita em detalhes nas próximas seções.

## 5.2 Identificação dos Métodos de Detecção

Conforme já descrito no Capítulo 3, nota-se a existência de um vasto conhecimento em detecção de anomalias de código. Diversos estudos na literatura expõem várias técnicas distintas que têm sido empregadas para realizar a detecção destas anomalias. Utilizando como base as taxonomias propostas por Sharma *et al.* [105] e Haque *et al.* [44], os métodos para detecção de anomalias podem ser classificados em cinco grandes categorias, de acordo com suas características e similaridades.

Na Figura 5.2 ilustra-se um fluxo de processo abstrato em camadas onde são sintetizados os métodos existentes para detecção de anomalias utilizando como base as cinco categorias pertencentes à classificação utilizada no estudo. Esta figura demonstra em alto nível o modo de funcionamento de cada uma das categorias de métodos de detecção. Cada método realiza suas etapas para detecção a partir do código-fonte (ou artefato de origem), passando em seguida por diversas etapas subsequentes até chegar à detecção das anomalias propriamente dita. A direção das setas mostra a direção do fluxo e as anotações nas setas mostram o método de detecção (primeira parte) e o número da etapa (segunda parte). A seguir, são descritos os detalhes de cada uma dessas categorias.

É importante mencionar que não existe um consenso na literatura a respeito de uma taxonomia comum para classificação dos métodos de detecção de anomalias. Contudo, a taxonomia apresentada no presente trabalho mostrou-se efetiva em identificar e classificar estudos relevantes associados à detecção de anomalias de código. Buscou-se tornar essa classificação o mais abrangente possível com intuito de ampliar sua cobertura, minimizando desse modo a possibilidade de não coletar dados associados a métodos relevantes do estado da arte.

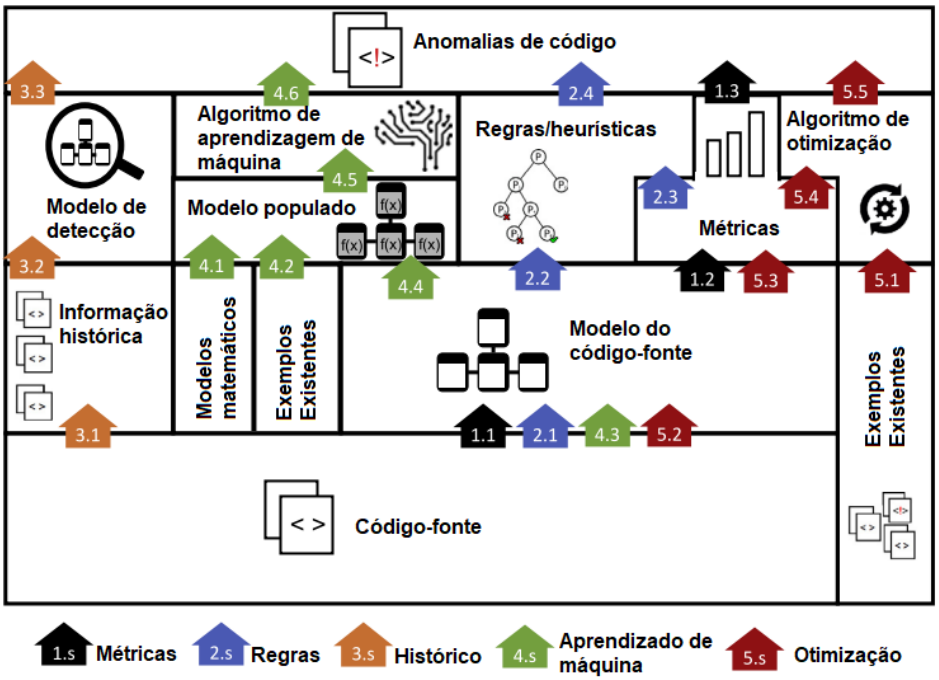


Figura 5.2: Visão em Camadas dos Métodos de Detecção.

### 5.2.1 Detecção Baseada em Métricas

Um método típico de detecção baseado em métricas utiliza o código-fonte como entrada, prepara um modelo de código-fonte (como uma *Abstract Syntax Tree* [AST]) (Etapa 1.1 na Figura 5.2) normalmente usando uma biblioteca de terceiros, detecta um conjunto de métricas de software (Etapa 1.2) que capturam as características de um conjunto de anomalias e, por fim, realiza a detecção através de um limiar adequado [44] (Etapa 1.3). Por exemplo, uma instância da anomalia *God Class* pode ser detectada usando o seguinte conjunto de métricas: WMC (*Weighted Methods per Class*), ATFD (*Access To Foreign Data*) e TCC (*Tight Class Coesion*) [68]. Essas métricas devem ser comparadas com limiares predefinidos e combinadas usando operadores lógicos. Além disso, a comunidade frequentemente utiliza diversas outras métricas tais como NOC (*Number of Children*), NOM (*Number of Methods*), CBO (*Coupling Between Objects*), RFC (*Response For Class*) e LCOM (*Lack of Cohesion of Methods*) que são utilizadas de modo conjunto para detectar outras instâncias de anomalias de código. Exemplares de métodos de detecção baseados em métricas podem ser melhor compreendidos com a análise dos trabalhos de Marinescu [68] e Lanza *et al.* [57].

### 5.2.2 Detecção Baseada em Regras/Heurísticas

Métodos de detecção de anomalias que definem regras ou heurísticas [76] (Etapa 2.2 na Figura 5.2) normalmente usam o modelo de código-fonte (Etapa 2.1) e às vezes métricas de software adicionais (Etapa 2.3) como entradas. Eles detectam um conjunto de anomalias quando as regras/heurísticas definidas previamente são satisfeitas. Existem muitas anomalias que não podem ser detectadas apenas pelo uso de métricas. Por exemplo, não se pode detectar as anomalias *Rebellious Hierarchy*, *Missing Abstraction*, *Cyclic Hierarchy* e *Empty Catch Block* através do uso de métricas de software. Em tais casos, regras ou heurísticas podem ser usadas para detectar essas anomalias.

Para fins de ilustração, a *Cyclic Hierarchy* é uma típica anomalia que ocorre quando o super-tipo tem conhecimento sobre seus subtipos. Esta anomalia pode ser facilmente detectada através da definição de uma regra que verifica se uma classe está se referindo as suas subclasses. Exemplares de métodos de detecção baseados em regras ou heurísticas podem ser melhor compreendidos com a análise dos trabalhos de Moha *et al.* [76] e Sharma *et al.*

[105].

### 5.2.3 Detecção Baseada em Informações Históricas

Alguns métodos detectam anomalias de código através do uso de informações associadas à evolução do código-fonte [92]. Tais métodos extraem informações estruturais do código e como ele foi modificado ao longo de um determinado período de tempo (Etapa 3.1 na Figura 5.2). Essas informações são usadas por um modelo de detecção (Etapa 3.2) para inferir sobre a ocorrência de anomalias de código. Por exemplo, através da aplicação de regras em um conjunto de métodos que foram alterados frequentemente no sistema de controle de versão, a anomalia *Divergent Change* pode ser detectada [92]. Exemplares de métodos de detecção baseados em informações históricas podem ser melhor compreendidos com a análise dos trabalhos propostos por Palomba *et al.* [92] e Tsantalis *et al.* [115].

### 5.2.4 Detecção Baseada em Aprendizado de Máquina

Vários métodos baseados em algoritmos de aprendizado de máquina, como *Support Vector Machines (SVM)* e *Bayesian Belief Networks (BBN)*, têm sido usados para detectar anomalias [37]. Um método típico para detecção de anomalias com uso de aprendizado de máquina começa com um modelo representando o problema de detecção de anomalias (Etapa 4.1 na Figura 5.2). Exemplos existentes (Etapa 4.2) em conjunto com o modelo do código-fonte (Etapa 4.3 e 4.4) podem ser utilizados para auxiliar a instanciação do modelo. O método resulta em um conjunto de anomalias aplicando um algoritmo de aprendizado de máquina escolhido (Etapa 4.5) no modelo preenchido. Por exemplo, um classificador SVM pode ser treinado usando atributos de métricas de software para cada classe de um dado sistema. Então o classificador pode ser usado em outros programas juntamente com dados de métricas correspondentes para detecção de anomalias [95]. Exemplares de métodos de detecção baseados em aprendizagem de máquina podem ser melhor compreendidos com a análise dos trabalhos propostos por Fontana *et al.* [37] e Pecorelli *et al.* [95].

### 5.2.5 Detecção Baseada em Otimização

Os métodos inseridos nesta categoria aplicam algoritmos de otimização (e.g., algoritmos genéticos e programação dinâmica) para detecção de anomalias de código. Esses métodos comumente aplicam um algoritmo de otimização (Etapa 5.5 na Figura 5.2) em métricas de software computadas (Etapa 5.4) a partir do código-fonte e sua representação (Etapas 5.2 e 5.3). Em alguns casos, exemplos existentes (Etapa 5.1) de anomalias também são utilizados para detectar novas instâncias. Exemplares de métodos de detecção baseados em otimização podem ser melhor compreendidos com a análise dos trabalhos de Sahin *et al.* [100] e Ghannem *et al.* [41].

## 5.3 Análise dos Métodos de Detecção

Nesta seção do trabalho, será realizada uma análise de cada método de detecção de acordo com a taxonomia de classificação apresentada. É importante destacar que, com base nos artefatos utilizados e nas etapas de realização, cada uma das categorias de métodos de detecção apresenta um conjunto de pontos fortes e possíveis fraquezas, que serão descritos a seguir.

Os métodos baseados em histórico lidam com a evolução a partir da implementação de mudanças que são parte comum no ciclo de vida do software. Desse modo, um sistema torna-se cada vez mais complexo ao longo do tempo, o que pode conduzir à introdução de anomalias de código nas versões subsequentes do sistema.

No entanto, os métodos baseados em histórico têm uma aplicabilidade limitada porque apenas algumas anomalias estão associadas a mudanças evolutivas. Um elemento de código (e.g., um método ou uma classe) que não necessariamente evoluiu de modo a ser impactado com uma ocorrência de anomalias não pode ser detectado por métodos baseados em histórico. Ainda, mesmo para anomalias que estão associadas a mudanças evolutivas, faz-se necessária a disponibilização de informações históricas. Isso pode ser um fator complicador quando o sistema encontra-se em estágios iniciais [44]. Mesmo que sejam utilizadas informações históricas de outros sistemas, tais informações podem não ser completamente aderentes ao sistema em virtude de diversos fatores como domínio, uso de tecnologias e padrões de codificação.

Os métodos baseados em aprendizado de máquina utilizam algoritmos para classificar e

detectar anomalias com base em um conjunto de exemplos fornecidos para cada anomalia. Os métodos desta categoria são benéficos em vários aspectos pois reduzem a carga cognitiva para analisar as anomalias de código, são capazes de fornecer resultado mais confiável ao estimar a conformidade entre o resultado obtido e o esperado e podem determinar o conjunto de métricas que mais influenciam na detecção de determinadas anomalias. Contudo, os métodos baseados em aprendizado de máquina dependem muito dos dados de treinamento e a falta de tais conjuntos de dados pode se tornar um fator impeditivo à sua implementação e uso [50]. Além disso, ainda não se sabe se a detecção baseada em aprendizado de máquina pode escalar para o grande número de anomalias conhecidas.

Os métodos de detecção de anomalias baseados em otimização dependem de dados de métricas de software e limiares correspondentes. Este fato os faz sofrer de limitações semelhantes aos métodos baseados em métricas. Contudo, muito esforço computacional pode ser empregado para escolha dos valores de limiares mais adequados a uma dada estratégia [105]. Isso pode se tornar um desafio se uma abordagem implementa a detecção de um conjunto elevado de anomalias de código. Ainda, este método de detecção necessita também de dados suplementares (e.g., exemplos existentes) para operar com níveis de eficácia aceitáveis.

Métodos de detecção baseados em regras/heurísticas são desenvolvidos com base nas informações estruturais do software. Embora estes métodos dependam de diferentes métricas para medir o sucesso do processo na maioria dos cenários, mesmo regras especificadas manualmente são relevantes para este método. As regras/heurísticas podem ser utilizadas para expandir o horizonte de outros métodos de detecção, fortalecendo-os com o poder de heurísticas definidas em entidades de código-fonte [43]. Portanto, métodos baseados em regras/heurísticas combinados com outros métodos de detecção oferecem mecanismos de detecção que podem revelar uma elevada proporção das anomalias de código conhecidas.

Finalmente, a detecção de anomalias baseada em métricas é conveniente e relativamente fácil de implementar. As métricas de software têm papel vital na automação (e escalabilidade) da detecção de anomalias de código [44]. A maioria das abordagens de detecção automatizadas foram desenvolvidas com base na computação de um conjunto de métricas de software. No entanto, como discutido anteriormente, não é possível detectar um conjunto mais amplo de anomalias usando apenas métricas comumente conhecidas. Outra crítica importante dos métodos baseados em métricas é sua dependência associada à escolha de um

conjunto apropriado de limites, que é uma atividade complexa.

## 5.4 Escolha do Método de Detecção

Para a escolha do método de detecção mais adequado para fornecer suporte à detecção interativa, deve-se identificar critérios relevantes para o projeto desta abordagem, bem como verificar a aderência dos métodos de detecção existentes a tais critérios. No contexto do presente estudo foram definidos os seguintes critérios:

- *Eficácia na detecção*. Este critério está intimamente relacionado à capacidade do método de detecção retornar substancialmente mais resultados relevantes do que irrelevantes sobre anomalias de código. O critério tem relevância porque abordagens com baixa eficácia podem desencorajar seu uso por parte dos desenvolvedores, além de promover um acúmulo de anomalias ao longo do ciclo de desenvolvimento de software por não apresentarem alertas adequados e oportunos da ocorrência de anomalias;

- *Capacidade de automação*. Este critério tem relação com a possibilidade de execução do processo de detecção de anomalias de forma automatizada, sem necessidade de pré-processamentos manuais ou mesmo de intervenção humana durante o processo. Este critério é relevante pois a abordagem interativa requer uma detecção contínua de anomalias de código, requerendo automatização de diversos processos para alcançar esta característica e tornar a abordagem exequível e utilizável;

- *Cobertura de anomalias*. Este critério está associado à capacidade que um dado método possui em detectar um maior número de tipos de anomalias de código dentre aquelas catalogadas. A detecção interativa tem, por sua natureza, o interesse direto em informar tão cedo quanto possível a existência de anomalias. Quanto mais tipos forem cobertos pela abordagem, maior será a capacidade de atender as necessidades dos desenvolvedores;

- *Independência de dados externos*. Este critério está associado à necessidade de dados para processamento além daqueles provenientes do próprio código-fonte. Este critério é relevante em virtude da necessidade de escalabilidade do processamento requerido

para habilitar a detecção interativa. Caso haja a necessidade de utilização de dados externos ou históricos, isso pode degradar o desempenho da abordagem, comprometendo sua funcionalidade e uso por parte dos desenvolvedores;

- *Necessidade de poucos recursos computacionais*. Este critério está associado à necessidade da abordagem em termos de processamento computacional e uso de memória. Quanto menos processamento e utilização de memória, mais viável será para habilitação da detecção interativa. Isso ocorre em virtude da necessidade do processamento em segundo plano do mecanismo de detecção ocorrer sem onerar o uso do ambiente integrado de desenvolvimento;

- *Análise do código-fonte em diferentes níveis de granularidade*. Este critério está associado à capacidade de um método em realizar análises em diferentes elementos de código (e.g., método, classe e pacote) com granularidades distintas de modo eficiente. Uma vez que a detecção interativa se preocupa com anomalias associadas ao contexto da programação atual, diferentes níveis de análises serão requeridos para prover suporte a esta característica. A abordagem interativa irá priorizar o cálculo de métricas associadas ao fragmento de código da atividade principal, em detrimento a outros elementos de código que não têm relação direta com este fragmento. Este critério possibilita um cálculo seletivo das métricas, priorizando as métricas mais locais e postergando o cálculo de métricas mais globais.

Estes critérios foram definidos através de um processo colaborativo envolvendo três especialistas na área de detecção de anomalias de código e refatoração. Com base na análise das categorias de métodos de detecção de anomalias considerados neste estudo, estes especialistas avaliaram também a satisfatibilidade de cada um dos critérios previamente definidos. Para auxiliar o processo avaliativo foi criado um formulário eletrônico com seis questões onde os especialistas utilizaram uma escala *Likert* de três fatores, sendo Total (T), Parcial (P), Não atende (N) os valores possíveis como resposta para cada questão. Cada um dos especialistas assinalou suas respostas neste formulário que auxiliou na centralização e análise dos resultados.

Das 30 possíveis respostas (i.e., a partir da combinação de cinco categorias e seis critérios) para cada especialista envolvido nesta avaliação, sete delas (cerca de 23%) apresenta-

ram algum tipo de discrepância. Isto ocorreu sempre que as respostas de pelo menos dois dos especialistas eram discrepantes para o mesmo critério. Por exemplo, um especialista assinalou o critério "eficácia" como sendo atendido de forma "total", enquanto os demais assinalaram esse critério como "parcial". Cada uma das respostas com algum tipo de discrepância foi submetida a um processo de discussão colaborativa envolvendo todos os especialistas. Após esse processo, os resultados dos apontamentos dos especialistas foram consolidados. Na Tabela 5.1 apresenta-se um resumo dos resultados da avaliação dos critérios.

Tabela 5.1: Critérios de Avaliação dos Métodos de Detecção.

| | **Métodos de detecção baseados em** | | | | |
|---|---|---|---|---|---|
| **Critério** | Métricas | Heurísticas | Aprendizado | Histórico | Otimização |
| Eficácia na detecção | P | T | T | T | T |
| Capacidade de automação | T | T | P | P | P |
| Cobertura de anomalias | T | P | P | P | P |
| Independência de dados | T | T | N | N | N |
| Poucos recursos | T | P | P | N | N |
| Granularidade da análise | T | P | P | P | P |

De acordo com os dados descritos na Tabela 5.1, a categoria baseada em métricas atende a maioria dos critérios considerados como relevantes para a abordagem de detecção interativa. De fato, alguns estudos expuseram que a detecção de anomalias baseada em métricas é conveniente e relativamente fácil de implementar [76][122]. Contudo, notou-se que a eficácia deste método de detecção apresenta resultados dependentes diretamente dos valores limiares para as métricas de software [58][69]. A escolha incorreta desses valores compromete significativamente a eficácia na detecção.

Por outro lado, a categoria de detecção baseada em regras e heurística também atendeu maior parte dos critérios, mesmo que de forma parcial. Os métodos baseados em regras/heurísticas podem "expandir o horizonte" da detecção que são baseadas em outros métodos. A partir da aplicação de heurísticas definidas em elementos do código-fonte, os métodos de detecção podem ter seu "poder de detecção" ampliado [105].

Portanto, a escolha que se mostra mais promissora é a utilização de métodos baseados em regras/heurísticas combinados com métricas. Esta combinação pode oferecer mecanismos de detecção com boa capacidade de automação e possibilitar a detecção de um grande conjunto

de anomalias conhecidas. Além disso, a combinação desses métodos tende a requerer uma baixa necessidade de dados externos além do próprio código-fonte, com processos razoáveis em termos de processamento e memória e com grande capacidade de prover análises em diferentes níveis de granularidade.

Além da escolha do método de detecção, um fator importante na construção da abordagem de detecção interativa diz respeito às anomalias suportadas. De acordo com os resultados da revisão da literatura descrito na Seção 2.1, as anomalias podem ser classificadas em diversas categorias (i.e., *Bloaters, Abusers, Change Preventers, Dispensables* e *Couplers*). Os *Bloaters* e os *Couplers* são as categorias de anomalias que devem merecer mais atenção por parte da abordagem de detecção interativa.

A primeira categoria está associada a métodos e classes que aumentaram para proporções tão gigantescas que são difíceis de manter e evoluir. Normalmente, essas anomalias não surgem imediatamente, mas se acumulam ao longo do tempo à medida que o programa evolui (e especialmente quando o desenvolvedor não faz qualquer ação para erradicá-las). Elas são introduzidas em um código principalmente quando novos recursos são adicionados a um sistema existente [44]. Portanto, tão cedo quanto possível essa categoria de anomalias for detectada, menos tempo e esforço serão empregados na sua remoção. Exemplos típicos de anomalias classificadas como *Bloaters* são o *God Class* e o *Long Method*.

Na segunda categoria, as anomalias contribuem para o acoplamento excessivo entre classes ou mostram o que acontece se o acoplamento for substituído por delegação excessiva. São conhecidos os efeitos do forte acoplamento em sistemas de software [39]: sistemas são alterados com esforço e custos adicionais; a integração com outros módulos e sistemas é dificultada; e impacto negativo na capacidade de compreensão e leitura do sistema. Desse modo, anomalias desta categoria necessitam ser detectadas antecipadamente para evitar que muitos módulos sejam afetados e promovam forte acoplamento entre eles. De modo similar, tão cedo quanto as anomalias desta categoria forem detectadas, menos tempo e esforço serão utilizados para refatorá-las. Exemplos típicos de anomalias classificadas como *Couplers* são o *Feature Envy* e o *Disperserd Coupling*.

## 5.5 Considerações Finais do Capítulo

O estudo descrito neste capítulo apresentou informações detalhadas sobre a escolha do método de detecção de anomalias mais promissor, considerando os critérios definidos para uma abordagem de detecção interativa de anomalias de código integrada a um processo de desenvolvimento.

As etapas do estudo descritas deram suporte para responder à QP2, conforme quadro abaixo:

**QP2 - Quais métodos de detecção se adéquam à abordagem de detecção interativa?**
De acordo com a análise dos métodos de detecção elencados no presente estudo, os métodos de detecção baseados em métricas em associação com regras/heurísticas se adéquam à detecção interativa e podem ser utilizados como base para a abordagem DI.

A definição dos métodos de detecção de anomalias de código é apenas um dos passos cruciais para a aplicação da abordagem DI. Para avaliar a viabilidade desses métodos na prática, é fundamental a definição dos requisitos para implementação desta abordagem. Esses requisitos devem levar em consideração aspectos técnicos, como a integração dos métodos de detecção de anomalias ao ambiente de desenvolvimento e a capacidade de monitoramento e gerenciamento contínuo dos itens de DT. Além disso, também devem abranger aspectos organizacionais, como a aceitação e engajamento dos desenvolvedores na utilização da abordagem DI para gestão das anomalias em projetos de software.

Somente a partir da definição e implementação adequada desses requisitos será possível realizar uma avaliação experimental com projetos reais e verificar se os métodos de detecção de anomalias suportam efetivamente a abordagem DI e podem melhorar a eficácia da detecção de anomalias de código. No capítulo serão apresentados e detalhados os requisitos para a construção e utilização de uma abordagem DI.

# Capítulo 6

# Definição dos Requisitos para Abordagem DI

Neste capítulo, são apresentados os requisitos para a construção e utilização de uma abordagem de Detecção Interativa (DI). É importante estabelecer essa diferenciação, uma vez que a abordagem DI possui diferentes objetivos e características em relação a abordagem de Detecção Não-Interativa (DNI). Definir requisitos pode garantir que a abordagem atenda às necessidades e expectativas dos desenvolvedores, bem como melhorar a eficácia na detecção de anomalias de código no contexto das atividades do processo de desenvolvimento de software.

Na Seção 6.1 descrevem-se os detalhes metodológicos do estudo. Nas seções 6.2 e 6.3 detalham-se o modo de funcionamento, bem como uma lista de características da abordagem DI, respectivamente. Na Seção 6.4 define-se um conjunto de diretrizes e demonstra-se um mapeamento com as características da abordagem DI. Na Seção 6.5 detalham-se os resultados associados à validação das diretrizes a partir da execução das atividades experimentais. Finalmente, são expostas as considerações finais do estudo na Seção 6.6.

## 6.1 Configuração do Estudo

O presente estudo objetivou prover suporte adequado para responder a seguinte Questão de Pesquisa (QP):

- **QP3 - Quais fatores uma abordagem de detecção interativa deve considerar em**

**sua concepção e utilização?**

Para este fim, será seguido um conjunto de etapas metodológicas conforme ilustrado na Figura 6.1.

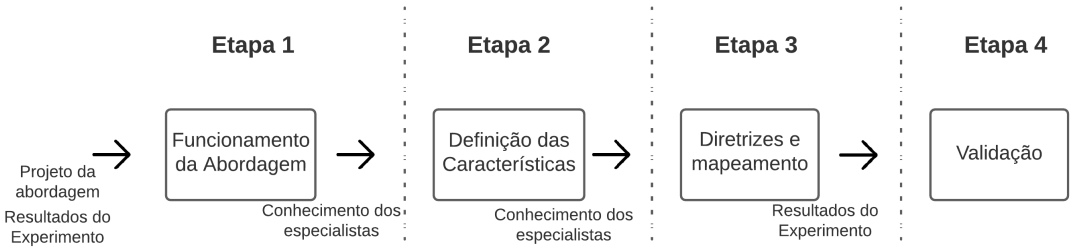


Figura 6.1: Etapas para Definição das Características da Abordagem DI.

Inicialmente, os resultados do experimento do uso de DI na prática (Capítulo 7) bem como o conhecimento adquirido no projeto e construção da ferramenta *Eclipse ConCAD* serviram de entrada para o entendimento do funcionamento de uma abordagem DI (Etapa 1). Em seguida, a análise do funcionamento em conjunto com o conhecimento de especialistas conduziu à definição das características desejáveis para uma abordagem DI (Etapa 2). Essas características foram mapeadas em diretrizes (Etapa 3) que foram em seguida validadas através de atividades experimentais (Etapa 4). Cada uma dessas etapas definidas na Figura 6.1 serão descritas em detalhes a seguir.

## 6.2 Funcionamento de uma Abordagem DI

Nesta parte do estudo detalha-se o funcionamento típico da abordagem DI de acordo com os vários passos necessários para sua realização. Na Figura 6.2 são descritos os principais passos que representam o funcionamento da abordagem:

- **Passo 1**. O desenvolvedor realiza alguma atividade de programação (e.g., implementação, modificação ou inspeção) no código-fonte com auxílio do seu Ambiente de Desenvolvimento Integrado. De modo geral, o código-fonte possui diversos arquivos de código que são organizados em pacotes de acordo com suas características e funcionalidades;

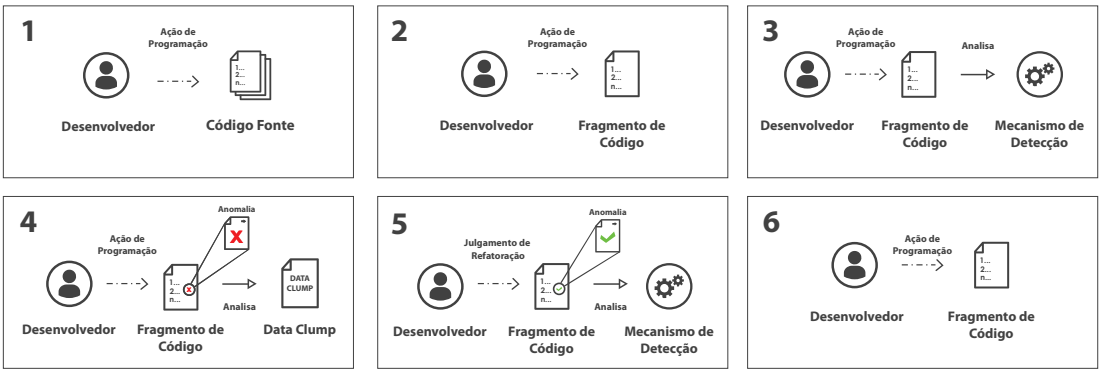


Figura 6.2: Etapas da Detecção Interativa.

- **Passo 2**. O desenvolvedor muda seu contexto para realizar alguma atividade de programação em um arquivo de código particular. Nesta ocasião, o desenvolvedor necessita focar suas atividades de programação em fragmentos específicos que são parte constituinte do arquivo de código que encontra-se ativo no Ambiente de Desenvolvimento Integrado;

- **Passo 3**. O desenvolvedor finaliza alguma atividade de programação no fragmento de código. Tão logo quanto ele conclua esta atividade de programação, faz-se necessário o salvamento deste arquivo de código particular visando a incorporação das alterações no código. Neste momento o mecanismo de detecção de anomalias de código atua - independentemente de uma requisição explícita do desenvolvedor - calculando métricas associadas a esse fragmento, bem como aos arquivos de código que possuem relação direta com o fragmento de código em questão;

- **Passo 4**. O mecanismo de detecção realiza uma análise detalhada dos valores associados às métricas e seus limiares. Esta análise tem o intuito de identificar possíveis violações das regras para detecção das anomalias. Caso ocorra uma violação nas estratégias previamente definidas para detecção, o desenvolvedor é alertado precocemente sobre a presença de anomalias no fragmento de código considerado tão cedo quanto ela foi inserida;

- **Passo 5**. O desenvolvedor recebe oportunamente a informação associada à ocorrência de uma anomalia e julga a necessidade de realizar ações de refatoração visando sua remoção. Para amparar seu processo decisório, o desenvolvedor pode fazer uso de in-

formações contextuais da anomalia (e.g., nível de espalhamento, elementos de código afetados, relacionamento com outras anomalias, dados históricos do sistema, valores de métricas entre outros) disponíveis no ambiente de desenvolvimento integrado;

- **Passo 6**. Este passo pode conduzir a dois desdobramentos principais onde o desenvolvedor: (i) conclui sobre a realização imediata da refatoração; ou (ii) conclui sobre a realização posterior da refatoração. No primeiro caso, o desenvolvedor deverá reiniciar o ciclo de passos de funcionamento da abordagem DI (Passo 1). No segundo caso, o desenvolvedor realiza alguma atividade de programação visando a correta aplicação da refatoração (Passo 2). Após a realização desta atividade, o desenvolvedor irá salvar o arquivo de código e, de forma simultânea, o mecanismo irá atuar no cálculo de métricas (Passo 3) e, imediatamente após a conclusão desta atividade, avalia se anomalia foi removida no trecho de código considerado (Passos 4 e 5).

Um fato interessante da abordagem DI é que o mecanismo de detecção analisa continuamente - em segundo plano - fragmentos de código onde o desenvolvedor realiza alguma atividade de programação (Figura 6.2). Isso propicia que instâncias de anomalias de código sejam detectadas concomitantemente às atividades de programação. Em outras palavras, os desenvolvedores que fazem uso de estratégia DI são alertados sobre a inserção ou remoção de anomalias de código imediatamente após a conclusão de alguma atividade de programação.

## 6.3 Características da Abordagem DI

Notadamente, a abordagem DI apresenta funcionamento de modo distinto às estratégias de DNI (Seção 2.3). Por conseguinte, a abordagem DI emerge no intuito de suprir as falhas existentes nas abordagens mais tradicionais.

Com base na experiência adquirida com o projeto e construção da ferramenta*Eclipse ConCAD* e analisando os principais passos alusivos ao funcionamento da abordagem DI, elenca-se um conjunto de características e fatores que devem ser considerados no contexto da abordagem DI. Essa lista preliminar foi submetida à apreciação de três especialistas na área de detecção de anomalias que analisaram e aprimoram seus componentes. No que segue, descreve-se detalhadamente cada uma das principais características da abordagem DI

considerada no presente trabalho, bem como apresenta-se um comparativo desta abordagem com as abordagens DNI tradicionais.

### 6.3.1 Mecanismo de Detecção

Na detecção interativa o mecanismo de detecção deverá funcionar em segundo plano e de forma contínua sem a necessidade de uma requisição explícita do desenvolvedor para seu funcionamento. O mecanismo de detecção deverá atuar computando métricas bem como avaliando a conformidade dessas métricas com as estratégias de detecção. Cada um dos diversos tipos de anomalias de código possui estratégias de detecção distintas uma vez que fazem uso de métricas e limiares específicos. Importante mencionar que os mecanismos de detecção das abordagens de detecção mais tradicionais, de modo geral, necessitam de uma requisição explícita do desenvolvedor para seu acionamento. Tal característica pode ser considerada um fator comprometedor para detecção de modo contínuo como boa prática.

A necessidade do mecanismo de detecção funcionar de forma contínua tem como base a promoção da qualidade de código contínua diluída no processo de desenvolvimento. Durante a realização das atividades de programação ao longo de um ciclo (e.g. *sprint*) é interessante que o desenvolvedor tenha ciência tão cedo quanto possível da ocorrência das anomalias de código. A partir disso, o desenvolvedor pode tomar a decisão de remover imediatamente uma ocorrência de anomalia ou postergar esta atividade através de ações de refatoração. Fatores como tempo demandado e esforço requerido serão levados em consideração nesse processo decisório.

### 6.3.2 Escala da Detecção

As ações de computar métricas relevantes de código e associá-las a estratégias de detecção são duas das atividades mais relevantes no contexto da detecção de anomalias de código. No caso específico da detecção interativa, faz-se necessária uma mudança de paradigma para realização destas atividades. A abordagem DI se preocupa mais com as ocorrências de anomalias associadas ao fragmento de código pertencente ao arquivo ativo no ambiente de desenvolvimento integrado. Nesse sentido, a escala de detecção deve priorizar elementos de código locais associados ao fragmento de código considerado em detrimento a elementos de

código mais globais, que não contribuem para ocorrência de uma anomalia de código em particular.

Do ponto de vista do processo de desenvolvimento, esta modalidade pode contribuir para análise de anomalias de código de modo mais detalhado durante o planejamento das atividades de desenvolvimento e manutenção (e.g. planejamento da *sprint*). Por exemplo, integrantes do time de desenvolvimento podem avaliar trechos de código específicos mediante suporte da abordagem DI para avaliar o nível de esforço e tempo que serão requeridos para remoção de uma anomalia em particular. No contexto da conclusão das atividades de programação, o uso da abordagem DI pode ser utilizado como critério de aceitação de itens de software concluídos ao final do ciclo iterativo, antes de ser integrado ao produto final de software (e.g., definição de pronto).

### 6.3.3 Integração ao Ambiente

A detecção interativa pressupõe o uso de um suporte automatizado obrigatoriamente interligado ao ambiente de desenvolvimento. Isso proporciona uma integração suficiente para habilitar as características da abordagem DI sem a necessidade da utilização de recursos extras (e.g., ferramentas *standalone*, *dashboards web* ou *plugins* extras) para atividade de detecção de anomalias de código.

Sob a ótica do processo de desenvolvimento, esta característica está intimamente ligada às práticas de integração e qualidade contínua. Tais práticas tem por objetivo encontrar e investigar os problemas associados aos artefatos o mais cedo possível, melhorar a qualidade do software e reduzir o tempo que se leva para validar e lançar novas atualizações. Mediante integração ao ambiente de desenvolvimento e, consequentemente, realizando de modo concomitante às atividades de programação e manutenção do software, tais práticas são asseguradas de modo facilitado.

### 6.3.4 Natureza da Detecção

Para proporcionar a capacidade de interação direta com trechos de código anômalos ao longo das atividades de programação, a abordagem DI deverá realizar continuamente a detecção de anomalias de código. Isso deve ocorrer para que, tão cedo quanto introduzidas, os desen-

volvedores tenham consciência da existência de tais anomalias. Cabe o comentário que essa característica tem forte associação com o funcionamento contínuo e em segundo plano do mecanismo de detecção já descrito anteriormente.

Um desafio relevante nesse contexto está associado à possibilidade de obstrução das atividades de programação. A detecção contínua deve ser bem planejada para evitar uma "inundação" de informações ao desenvolvedor que podem inibir ou comprometer suas atividades de programação principal. É conhecido na literatura que os desenvolvedores se incomodam facilmente com avisos invasivos em tempo real que venham a interferir com suas atividades de programação atuais [79][80]. No entanto, os alertas associados à presença de anomalias de código tendem a ser mais eficazes quando são fornecidos mais cedo. Assim, deve ser priorizada uma solução que forneça alertas de maneira contínua, porém de uma forma extremamente discreta. Isso tem associação direta com as características de integração com ambiente de desenvolvimento e com as formas de visualização destes alertas.

### 6.3.5 Formas de Visualização

Visando habilitar a capacidade de ter consciência da existência das anomalias tão cedo quanto forem introduzidas, são requeridas novas e diferentes formas de visualização [79]. No caso da detecção interativa, essas formas de visualização devem ser embutidas diretamente no código-fonte e também devem ser menos invasivas quanto possível.

As formas de visualização devem alertar não apenas detecções precisas de anomalias de código, bem como fornecer mecanismos para exploração de informações contextuais (e.g., causas da ocorrência, nível de espalhamento e trechos de código afetados) durante as atividades de programação. Tais informações contextuais tem objetivo de guiar os esforços do desenvolvedor e tornar eficazes suas ações de refatoração. Do ponto de vista do processo de desenvolvimento, esses meios específicos de exploração tornam a melhoria da qualidade uma atividade mais viável e eficiente em termos de tempo.

### 6.3.6 Interação Imediata com Código Anômalo

Através da interação direta com trechos de código anômalos, os desenvolvedores podem obter rapidamente um bom resumo da ocorrência de uma anomalia particular, sem perder seu

contexto da atividade de programação principal. Isso permite que eles tomem uma decisão no melhor (e mais oportuno) momento para resolver esta ocorrência através de ações de refatoração. Esta capacidade de interação direta é uma das principais características da DI e ela pode contribuir para um aumento na produtividade dos desenvolvedores e a qualidade de seus projetos de duas maneiras. Primeiramente, emitindo alertas sobre a ocorrência de anomalias assim que elas são inseridas no código. Segundo, fornecendo informações contextualizadas associadas a cada instância de anomalias com vias a facilitar sua identificação e correção.

Essa característica tem sua utilidade evidenciada ao longo do ciclo iterativo de atividades de programação e manutenção. É importante mencionar que a correção de anomalias geralmente requer: (i) a compreensão de múltiplos fragmentos de código que estão amplamente dispersos; e (ii) o emprego de várias ações de refatoração para corrigir uma determinada ocorrência de anomalia. Por consequência, a interação indireta com código pode dificultar a execução destes passos, comprometendo assim a manutenção da qualidade do código através da identificação e correção das anomalias.

### 6.3.7 Informações Contextuais

A DI deve prover informações contextualizadas (e.g., nível de espalhamento da anomalia, módulos afetados, módulos dependentes entre outros) da ocorrência de uma anomalia de código em vez de confrontar programadores com apenas números (i.e., valores de métricas anormais). Tais números são de difícil compreensão e interpretação por parte dos desenvolvedores, o que pode inibir ações efetivas que conduzam a melhoria da qualidade do código. Embora as anomalias de código sejam detectadas usando regras e heurísticas baseadas em métricas [57][58], o problema é descrito em termos de conceitos de *design*, ao invés de simplesmente utilizar números. A abordagem DI deve listar os elementos de código, bem como suas relações que contribuem - direta ou indiretamente - para a ocorrência de uma anomalia específica, visando guiar as ações que o usuário pode realizar para correta identificação e correção de tais ocorrências. Ter essas informações contextuais de fácil acesso pode ser produtivo em termos de atividades do processo de desenvolvimento. Muitas vezes se faz necessário a mudança de contexto da atividade de programação para acessar tais informações. Isso além de obstruir a atividade de programação principal tende a ser dispendiosa em

termos de tempo e esforço.

### 6.3.8 Correção das Anomalias

No contexto da abordagem DI, a correção das anomalias é realizada de modo concomitante à atividade de programação principal. Em outras palavras, o desenvolvedor não necessita mudar seu contexto de atividade de programação uma vez que as características de detecção contínua sem a requisição do desenvolvedor, a interação direta com código "anômalo", bem como a visualização embutida no código-fonte propiciam a realização das duas atividades de modo simultâneo sem qualquer prejuízo. Olhando para o processo de desenvolvimento, esta característica pode auxiliar significativamente na identificação e correção de anomalias, transformando o fluxo de atividades mais intuitivo ao desenvolvedor e, como consequência, facilitando a manutenção e melhoria do código em desenvolvimento.

### 6.3.9 Configuração dos Limiares

Os resultados da detecção de anomalias de código são fortemente afetados pela diversidade na estimativa dos valores dos limiares associados às métricas de software [74]. Devido à falta de um método padrão para medir limiares adequados, as abordagens de detecção calculam esses valores de maneiras diferentes [36]. As abordagens tradicionais utilizam prioritariamente a definição de limiares fixos com base no conhecimento especializado. É válido mencionar que a estimativa de limiares apropriados depende de muitos fatores, como o tamanho do sistema, seu domínio de aplicação, as melhores práticas da organização, a percepção do desenvolvedor quem define esses valores [35]. No entanto, um valor de limiar fixo não pode lidar com a variação das informações do sistema impostas pela sua evolução.

A abordagem DI utiliza limiares adaptativos e configuráveis. Ao contrário do uso de limiares fixos, a utilização desta abordagem tende a ser mais flexível, pois é baseada no conjunto de características do sistema sobre as quais seu valor é formulado de diferentes maneiras. De acordo com sua experiência e preferências, os desenvolvedores podem selecionar valores de limiares mais adequados às suas diversas estratégias de detecção. Para o processo de desenvolvimento, isso é particularmente útil uma vez que em diferentes estágios do desenvolvimento, diferentes critérios podem ser aplicados, demandando modificações nestes

limiares.

### 6.3.10 Comparação das Características das Abordagens DI e DNI

Nas subseções anteriores foi feita a explanação das principais características que compõem a abordagem de detecção interativa (DI). Tais características servem como base para criação de uma taxonomia de classificação das abordagens existentes de acordo com a capacidade de prover meios de "interação" do usuário com trechos de código afetados pelas anomalias. Assim, o atendimento a estas características proporcionam às abordagens existentes a capacidade de serem "interativas". Finalmente, na Tabela 6.1 expõe-se um resumo dessas características, bem como realiza-se uma análise comparativa frente às abordagens de detecção não-interativas (DNI) tradicionais.

Tabela 6.1: Comparação das Estratégias DI e DNI

| **Característica** | **DI** | **DNI** |
|---|---|---|
| Mecanismo de detecção | Funcionamento contínuo em segundo plano | Funcionamento mediante requisição do desenvolvedor |
| Integração com ambiente de desenvolvimento | Mandatória | Opcional |
| Natureza da detecção | Detecção contínua | Detecção sob demanda |
| Forma de visualização das anomalias | Embutida diretamente no código-fonte | Em listas ou arquivos estruturados |
| Escala da detecção | Detecção local no fragmento de código | Detecção global no projeto de software inteiro |
| Interação com código anômalo | Direta | Indireta |
| Informações contextuais das anomalias | Simples e de fácil acesso | Necessita de mudanças de contextos |
| Correção da anomalia | Concomitante à atividade de programação | Atividade individual e exclusiva |
| Configuração de limiares | Adaptáveis e flexíveis | Fixos |

## 6.4 Definição de Diretrizes

Com base nas características descritas na seção anterior e com auxílio de três especialistas na área de detecção de anomalias e refatoração, foi derivada uma série de fatores que acredita-se que possa ser útil em qualquer abordagem para detecção interativa. Na Tabela 6.2, capturou-se esses fatores como um conjunto de diretrizes estabelecidas de forma centrada no desenvolvedor propenso a utilizar a referida abordagem.

Tabela 6.2: Diretrizes para Abordagens DI.

| **Diretriz** | **Raciocínio** |
|---|---|
| LIMITAÇÃO. A abordagem não deve sobrecarregar o usuário com os resultados da detecção de anomalias. | Às vezes, as anomalias emergem de diversos fragmentos de código. Outras vezes, um fragmento emite muitas anomalias. Assim, a abordagem não deve exibir informações da detecção de forma que uma proliferação de anomalias sobrecarregue o desenvolvedor. |
| RELACIONAMENTO. A abordagem deve exibir as relações entre os elementos do programa afetados pelas anomalias. | Algumas anomalias não de um único ponto no código, mas da relação entre vários fragmentos não contíguos. Assim, a abordagem deve exibir informações das anomalias de forma relacional quando esta for causada pela relação entre fragmentos de código. |
| PARCIALIDADE. A abordagem deve enfatizar anomalias que são difíceis de ver a olho nú. | Os programadores podem achar que há mais valor em ter uma abordagem que os informe sobre certas anomalias e menos valor sobre outras. Assim, a abordagem deve enfatizar aquelas anomalias difíceis de reconhecer sem um suporte automatizado. |
| DESOBSTRUÇÃO. A abordagem não deve distrair o desenvolvedor em sua atividade de programação principal enquanto detecta anomalias. | As atividades de programação e detecção de anomalias são frequentemente entrelaçadas. Assim, uma abordagem não deve impedir o desenvolvedor da atividade de programação enquanto reúne, analisa e exibe informações sobre anomalias. |
| DISPONIBILIDADE. A abordagem deve disponibilizar informações sobre anomalias de forma oportuna. | A tática mais popular para refatoração ocorre quando um programador intercala refatorações com atividades de programação. Uma vez que analisar anomalias faz parte dessa intercalação, uma abordagem que suporte essa tática deve ajudar os programadores a encontrar anomalias rapidamente, sem forçá-los a passar por um longo processo para identificar alguma anomalia. |
| SENSIBILIDADE AO CONTEXTO. A abordagem deve detectar prioritariamente as anomalias relacionadas ao código da atividade de programação atual. | Remover uma anomalia que não está relacionado à tarefa de programação atual é uma distração dessa tarefa. Portanto, corrigir anomalias de maneira insensível ao contexto pode ser uma maneira ineficiente de usar recursos ou pode até ser contraproducente. Assim, uma abordagem de detecção deve indicar anomalias relevantes para o contexto de programação atual. |
| LUCIDEZ. A abordagem deve informar as razões da ocorrência das anomalias além de detectá-las. | Anomalias podem ser complexas e difíceis de entender, porque podem ser sutis ou flagrantes, difundidas ou localizadas entre outras. Uma abordagem que comunique essas propriedades pode colaborar no processo decisório do desenvolvedor em confirmar e remover as ocorrências. Assim, uma abordagem deve auxiliar o desenvolvedor na descoberta das fontes do problema explicando porque a anomalia existe. |

Estes conceitos foram utilizados na avaliação empírica descrita na próxima seção (Seção 6.5). Acredita-se que enumerar essas diretrizes é importante porque captura detalhes associados à abordagem DI de forma reutilizável. Elas podem também ser utilizadas por outros

projetistas de abordagens de detecção de anomalias, visando a escolha de quais fatores são desejáveis nas suas abordagens.

Diferentemente das características da abordagem DI que foram descritas do ponto de vista de projetistas, as diretrizes descritas na Tabela 6.2 foram criadas com objetivo de capturar a percepção dos usuários finais da abordagem de detecção. No entanto, para se ter uma visão colaborativa entre esses dois artefatos, buscou-se mapear as características da abordagem DI com as diretrizes descritas neste estudo. Esse mapeamento será importante para validação das diretrizes e, consequentemente, das características da abordagem DI do ponto de vista dos desenvolvedores usuários. Na Figura 6.3 descreve-se o mapeamento entre esses conceitos.

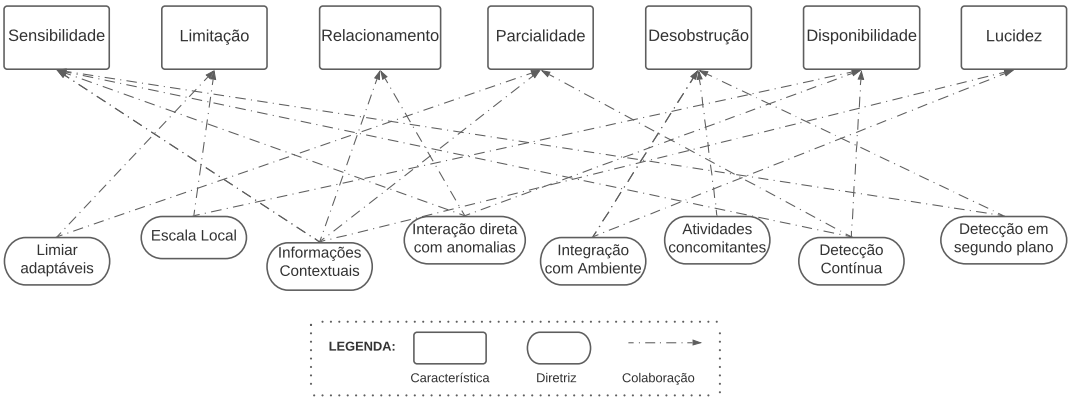


Figura 6.3: Mapeamento das Características e Diretrizes.

Cabe a ressalva de que esse mapeamento se dá através da colaboração entre esses conceitos. De acordo com a Figura 6.3 nota-se o modo como as diretrizes contribuem para que as características sejam satisfeitas. Por exemplo, a utilização de limiares adaptáveis e flexíveis colaboram com a diretriz de limitação, pois a escolha de valores incorretos de limiares pode conduzir a um número grande de falsos positivos. Isso pode sobrecarregar o desenvolvedor com resultados não relevantes de anomalias de código. Similarmente, a utilização de informações contextuais contribui para a diretriz sensibilidade. O uso dessas informações ajuda na detecção e análise prioritária de anomalias associadas ao fragmento de código associado à atividade principal.

## 6.5 Resultados da Validação

O mapeamento das características e diretrizes habilita a possibilidade de uma validação mútua desses conceitos sob o ponto de vista dos desenvolvedores usuários da abordagem de detecção. Para realização desta validação foram realizadas atividades experimentais no contexto do estudo descrito no Capítulo 7. Após a realização das atividades de detecção, os 16 participantes do experimento foram convidados a responder questões diretamente associadas às diretrizes apresentadas neste estudo.

Na Tabela 6.3, lista-se como os participantes classificaram cada diretriz previamente definida para o estudo (Tabela 6.2). Na coluna da esquerda, o nome da diretriz é listado. Importante citar que o participante leu a descrição completa da diretriz (não apenas seu nome) com intuito de minimizar qualquer possível viés associado ao entendimento da questão. As colunas da direita listam quantos participantes classificaram cada diretriz de acordo com o nível de importância: Não importante (NI), pouco importante (PI), importante (I), muito importante (MI) e Essencial (E). Por exemplo, um participante assinalou o critério Disponibilidade como sendo "Pouco Importante", enquanto nove a marcaram como "Essencial".

Tabela 6.3: Resultados Associados às Diretrizes.

| | Nível de Importância | | | | |
|---|---|---|---|---|---|
| **Critério** | NI | PI | I | MI | E |
| Desobstrução | 0 | 0 | 0 | 3 | 13 |
| Limitação | 0 | 1 | 1 | 2 | 12 |
| Parcialidade | 0 | 1 | 1 | 3 | 11 |
| Sensibilidade | 0 | 0 | 2 | 3 | 10 |
| Disponibilidade | 0 | 1 | 2 | 4 | 9 |
| Relacionamento | 0 | 2 | 3 | 3 | 8 |
| Lucidez | 0 | 3 | 2 | 3 | 8 |

Os resultados desta validação confirmaram a hipótese de que as diretrizes representam considerações de projeto desejáveis para a abordagem DI. Notou-se, em linhas gerais, que os participantes assinalaram maior parte das respostas como sendo "Importantes", sugerindo que eles também acreditam que as diretrizes são importantes para a abordagem DI. Por outro lado, uma minoria dos participantes pareceu acreditar que algumas diretrizes não são de fato importantes. Por exemplo, a diretriz postulada que foi julgada a menos importante, lucidez,

foi julgada como "Pouco Importante" por três participantes. Curiosamente, esses três participantes eram todos estudantes, e apresentaram menor nível de experiência se considerada a média da amostra. Uma possível interpretação é que, talvez, programadores menos experientes não valorizem um suporte automatizado que explique seu raciocínio porque acreditam que precisar de tal explicação é um sinal de falta de habilidade em programação.

## 6.6 Considerações Finais do Capítulo

Neste capítulo foram apresentados requisitos inerentes a uma abordagem DI. Os resultados do estudo descritos deram suporte para responder a QP3, conforme quadro abaixo:

**QP3 - Quais fatores uma abordagem de detecção interativa deve considerar em sua concepção e utilização?**
Os resultados do estudo apontaram para uma série de fatores relevantes no contexto da detecção interativa de anomalias de código:

- Interação direta com código anômalo;
- Mecanismo de detecção com funcionamento contínuo em segundo plano;
- Integração com ambiente de desenvolvimento mandatória;
- Natureza da detecção contínua;
- Forma de visualização das anomalias embutida no código-fonte;
- Escala local de detecção no fragmento de código da atividade de programação;
- Informações contextuais simples e de fácil acesso;
- Correção da anomalia concomitante à atividade de programação; e
- Configuração de limiares adaptáveis e flexíveis.

A abordagem DI se mostra promissora para auxiliar no desenvolvimento de software, permitindo a identificação e correção de anomalias em tempo real. A integração com o ambiente de desenvolvimento e a escala local de detecção tornam o processo de correção mais fácil e reduzem o tempo de desenvolvimento. A visualização das anomalias no código-fonte e informações contextuais auxiliam na compreensão do problema e na escolha da melhor solução. A correção simultânea à atividade de programação e a configuração de limiares

flexíveis tornam a detecção interativa mais eficiente. A incorporação desses fatores em uma ferramenta de detecção interativa pode trazer grandes benefícios para o processo de desenvolvimento de software, aumentando a produtividade do programador e melhorando a qualidade do código produzido.

A implementação de um suporte automatizado aderente a abordagem DI deve ser cuidadosa e estratégica para garantir a integração com o ambiente de desenvolvimento e a usabilidade da ferramenta. A ferramenta deve permitir a detecção contínua em segundo plano, com uma interface visual clara e informativa e a configuração de limiares adaptáveis. Esses fatores contribuirão de forma direta para que os desenvolvedores possam produzir código de maior qualidade.

Dessa forma, o Capítulo 6 apresentará uma proposta de abordagem DI integrada ao ambiente de desenvolvimento *Eclipse*, juntamente com a avaliação experimental da DI em projetos reais. Os resultados dessa avaliação experimental serão fundamentais para a verificação do impacto da abordagem DI na detecção de anomalias de código em projetos reais. Com base nesses resultados, poderão ser identificadas oportunidades de melhorias na abordagem DI e nos métodos de detecção de anomalias, visando à sua aplicação em cenários ainda mais complexos e desafiadores.

# Capítulo 7

# Avaliação da Abordagem DI na Análise de Código

Após identificar um conjunto de requisitos para a abordagem DI em um estudo apresentado no Capítulo 6, foi desenvolvida a ferramenta *Eclipse ConCAD* para validar a relevância desses requisitos e auxiliar na avaliação da eficácia da abordagem DI. Em seguida, foi realizada uma avaliação experimental para verificar se a abordagem DI é capaz de trazer benefícios em comparação à Detecção Não-Interativa (DNI) no contexto da detecção de anomalias durante a inspeção de código em projetos reais.

A abordagem DI na inspeção de código traz uma distinção significativa em relação as abordagens DNI tradicionais. A DI envolve a interação direta entre os desenvolvedores e a abordagem de análise, permitindo uma avaliação mais precisa e direcionada das anomalias presentes no código. Ao contrário dos métodos estáticos de DNI, nos quais as detecções são feitas automaticamente sem envolvimento humano direto, a DI capacita os desenvolvedores a fornecerem *insights* contextuais e conhecimento especializado durante o processo de análise e inspeção. Isso cria uma dinâmica mais colaborativa, onde a abordagem atua como um auxílio na identificação de problemas, ao mesmo tempo em que os desenvolvedores têm a oportunidade de interpretar e ajustar as detecções com base em sua compreensão do código e do contexto do projeto. Características como a capacidade de customização das regras de detecção, *feedback* em tempo real e foco nas particularidades do projeto podem ser vantajosas na DI, permitindo um ajuste mais preciso das detecções de anomalias e uma análise mais abrangente do código-fonte. Isso pode resultar em uma abordagem mais adaptável e eficaz

para melhorar a qualidade e a confiabilidade do software.

Neste capítulo, são apresentados os resultados da avaliação experimental da DI durante a inspeção de código em projetos reais. É importante mencionar que os resultados desse estudo foram publicados no *International Conference on Software, Telecommunications and Computer Networks* (SoftCOM'22) [8] e no *IEEE Access* [15]. Na Seção 7.1 são detalhados os procedimentos metodológicos do estudo. Na Seção 7.2 são descritos o método de seleção dos participantes. Na Seção 7.3 é apresentada a ferramenta desenvolvida especificamente para este experimento. Na Seção 7.4 são apontados aspectos relacionados à execução do experimento. Em seguida, na Seção 7.5 são expostos e discutidos os resultados do estudo enquanto que na Seção 7.6 são expostas as ameaças à validade do estudo. Finalmente, na Seção 7.7, são apresentadas as considerações finais do estudo.

## 7.1 Escopo do Experimento

A análise de código consiste na revisão e avaliação do código fonte para identificação de problemas, entre elas as anomalias de código. Esta atividade está contida no processo de desenvolvimento de software, podendo ser realizada manualmente por desenvolvedores ou mediante suporte de ferramentas automatizadas. O objetivo da análise de código é garantir que o código seja confiável e obedeça boas práticas de programação. Além disso, a análise de código também ajuda a promover a manutenibilidade e legibilidade do código a longo prazo.

A detecção e correção de anomalias de código durante a análise de código pode economizar tempo e recursos durante o ciclo de vida do projeto, além de melhorar a qualidade geral do software. Assumiu-se neste trabalho que a abordagem DI, ao permitir a interação direta entre os desenvolvedores e os resultados da análise, pode oferecer uma vantagem significativa em termos da detecção precisa e resolução eficaz das anomalias. A capacidade de combinar o conhecimento humano com a análise automatizada possibilita uma abordagem mais abrangente e adaptável para a detecção de problemas de código. Através da DI, espera-se não apenas otimizar a detecção de anomalias, mas também promover uma compreensão mais profunda do código-fonte e das complexidades do projeto, resultando em um software mais robusto e confiável. Embora o uso desta abordagem pareça promissora, há uma falta

de conhecimento empírico quanto à sua eficácia na atividade de análise de código. Esta observação levanta a seguinte Questão de Pesquisa (QP):

- **QP4 - A aplicação da abordagem de detecção interativa contribui para melhoria da eficácia na detecção de anomalias de código durante a análise de código?**

Para responder esta QP, foi avaliado se a utilização de suporte de DI proporciona aos desenvolvedores, no contexto específico da análise de código, uma capacidade maior de detectar anomalias em comparação com a mesma atividade suportada pela abordagem DNI. Essa questão é fundamental para entender se o uso da abordagem DI pode realmente melhorar a eficácia da detecção de anomalias em código-fonte. Desse modo, visando prover melhor suporte à resposta da QP4, sub-questões de pesquisa foram definidas para o estudo e encontram-se sumarizadas na Tabela 7.1.

Tabela 7.1: Subquestões de Pesquisa (QP4).

| **QP** | **Descrição** |
|---|---|
| SQP4.1 | O uso da abordagem DI promove melhoria na medida de eficácia *precision* na detecção de anomalias de código? |
| SQP4.2 | O uso da abordagem DI promove melhoria na medida de eficácia *recall* na detecção de anomalias de código? |

Para a **SQP4.1** comparam-se as abordagens DI e DNI através da medida de eficácia *precision*. Esta análise é essencial para entender como as abordagens de detecção retornaram resultados substancialmente mais relevantes (i.e., verdadeiro positivo) do que irrelevantes (i.e., falso positivo). Para a **SQP4.2**, comparam-se as abordagens DI e DNI através da medida de eficácia *recall* na atividade de detecção de anomalias de código usando DI e DNI. O *recall* também é uma medida importante porque permite observar qual abordagem retornou a maioria dos resultados relevantes.

Um bom equilíbrio entre essas medidas é importante, especialmente no contexto da detecção de anomalias de código. Elevados valores de *precision* não garantem a detecção de todas as anomalias existentes enquanto que elevados valores de *recall* pode produzir muitos falsos positivos. Idealmente, deve-se buscar um equilíbrio que maximize tanto a *precision* quanto o *recall*, garantindo que a detecção de anomalias seja precisa e abrangente o sufici-

ente para melhorar a qualidade do software. Mais detalhes sobre essas medidas de eficácia podem ser obtidos na Seção 2.4.

Para cada subquestão descrita anteriormente na Tabela 7.1, foram definidas Hipóteses (H) que estão resumidas na Tabela 7.2.

Tabela 7.2: Hipóteses (H).

| H | Descrição |
|---|---|
| H4.1 | A abordagem DI tem uma melhor medida de *precision* se comparada a abordagem DNI na atividade de detecção de anomalias. |
| H4.2 | A abordagem DI tem uma melhor medida de *recall* se comparada a abordagem DNI na atividade de detecção de anomalias. |

As hipóteses descritas na Tabela 7.2 foram definidas em virtude de evidências empíricas encontradas no estudo prévio de Murphy-Hill e Black [79], onde os autores indicaram que o uso da abordagem DI poderia aumentar o número de instâncias de anomalias de código detectadas. Portanto, espera-se que o uso da abordagem DI possa melhorar a eficácia da detecção de anomalias de código em termos das medidas de eficácia consideradas. Se os números associados a *precision* que os participantes obtiverem ao usar a abordagem DI for maior do que os números associados a *precision* quando os participantes usarem a abordagem DNI, então a **H4.1** é confirmada. Da mesma forma, se os números associados a *recall* que os participantes obtiverem ao usar a abordagem DI for maior do que os números associados a *recall* quando os participantes estiverem usando a abordagem DNI, a **H4.2** é confirmada.

## 7.2 Planejamento, Seleção de Participantes e Avaliação dos Resultados

Para projetar o experimento controlado, utilizou-se um conjunto de recomendações descritas nos trabalhos de Wohlin *et. al* [125] e Jedlitschka *et. al* [47]. Os participantes executaram tarefas relacionadas à detecção de anomalias de código no contexto da análise e inspeção de código-fonte. Todas as tarefas foram realizadas mediante suporte das abordagens DI e DNI. Para a abordagem DI, foi desenvolvida uma ferramenta, denominada *Eclipse ConCAD* (Apêndice D). A inspeção manual foi usada como abordagem DNI. Esta abordagem DNI foi

amplamente empregada para comparar técnicas automatizadas de detecção de anomalias de código [89] [78] [82] [110].

A comparação entre as abordagens DI e DNI permite concluir se características particulares de DI (e.g., detecção precoce e interação direta com trechos de código "anômalos") trazem (des)vantagens aparentes. Esta decisão foi também baseada no fato de que não há outra solução automatizada robusta que forneça suporte às características de DI para os mesmos tipos de anomalias. Além disso, muitos considerariam DI e DNI complementares em vez de abordagens competitivas, visto que podem ser direcionadas a diferentes estágios de desenvolvimento. Todavia, embora abordagens DI e DNI possam ser usadas de forma complementar, elas também podem ser usadas com o mesmo propósito durante uma atividade de programação (e.g., análise de fragmentos de código).

Em relação aos participantes deste estudo, foram recrutados estudantes de graduação e desenvolvedores profissionais, totalizando 16 participantes nas atividades experimentais. Todos os participantes foram recrutados por meio de uma mensagem de *e-mail* com base no interesse explícito e voluntário em participar do experimento. Era esperado que os participantes estivessem pelo menos moderadamente familiarizados com a linguagem Java. No entanto, não esperava-se o amplo conhecimento dos participantes sobre anomalias de código ou técnicas de detecção usadas no experimento.

Todos os participantes estudavam e/ou trabalhavam em Campina Grande-PB. Os estudantes eram do curso de Engenharia de Computação e cursavam uma disciplina de Programação para *Web*. Os desenvolvedores foram oriundos de uma mesma organização de software e desempenhavam atividades profissionais associadas a programação. Todos os participantes deste estudo participaram de uma sessão de treinamento para nivelar o conhecimento sobre anomalias de código, abordagens de detecção, refatoração e utilização do ambiente de desenvolvimento integrado no contexto das atividades experimentais.

Aplicou-se análise estatística, utilizando a ferramenta R [53], sobre os dados obtidos nas atividades. Esta ferramenta fornece meios adequados para o cálculo dos testes estatísticos considerados neste estudo: (i) *Wilcoxon signed-rank test* [55] - aplicado aos valores associados às instâncias de anomalias detectadas. Este teste foi selecionado uma vez que estes dados não seguem uma distribuição normalizada; e (ii) *Paired T-Test* [55] - aplicado aos valores das medidas *recall* e *precision*. O teste foi selecionado uma vez que as medidas obtidas

seguem uma distribuição normalizada. A execução das tarefas experimentais derivou dados para duas amostras: (i) a amostra com o auxílio da abordagem DI e (ii) a amostra com o auxílio da abordagem DNI. Os testes estatísticos mencionados acima puderam ser aplicados, uma vez que cada observação da primeira amostra podia ser pareada com uma observação da segunda.

## 7.3 Ferramenta *Eclipse ConCAD*

Para viabilizar o experimento do uso da DI em projetos reais, foi necessário o desenvolvimento de uma ferramenta integrada ao ambiente de desenvolvimento de software. No que segue, serão exibidas as principais funcionalidades (Seção 7.3.1), seus detalhes arquiteturais (Seção 7.3.2), bem como o modo de funcionamento da ferramenta (Seção 7.3.3). Ademais, os resultados associados a construção e validação desta ferramenta foram publicados nos anais do X *Workshop* de Visualização, Evolução e Manutenção de Software (VEM'22) [9] e no *Journal of Software Engineering Research and Development* (JSERD) [11].

### 7.3.1 Funcionalidades

Visando um mínimo impacto no atual processo de codificação de software, decidiu-se criar um *plugin* para uma das ferramentas mais utilizadas para desenvolvimento de software no mundo - a plataforma Eclipse[1]. Essencialmente, a ferramenta *Eclipse ConCAD* utiliza como entrada o código-fonte Java de um projeto de software e produz como saída uma exibição dos resultados associados à detecção de anomalias de código. Na Figura 7.1 encontra-se descrito o fluxo de trabalho da ferramenta em termos de atividades.

Basicamente, existem dois perfis principais de utilização da ferramenta: desenvolvedor e usuário. O primeiro tem relação direta com a construção (ou expansão) da ferramenta, enquanto que o segundo tem relação direta com a utilização de suas funcionalidades. As seguintes funcionalidades estão disponíveis na ferramenta:

- **Mapeamento de Informações**. O usuário pode incluir informações externas através de mapeamentos para o código-fonte. Por exemplo, o projeto arquitetural do sistema

[1] https://www.eclipse.org/

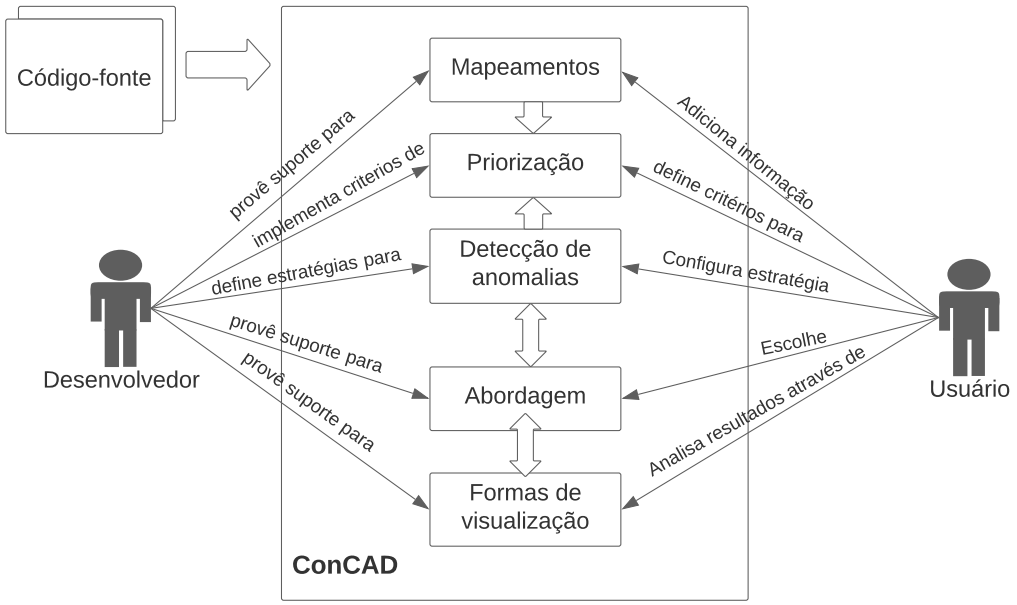


Figura 7.1: Fluxo de Trabalho da Ferramenta *Eclipse ConCAD*.

pode ser mapeado para o código com objetivo de indicar quais classes têm relação direta com cada componente da arquitetura;

- **Detecção de Anomalias**. O código-fonte do projeto atual do desenvolvedor é analisado através de diversas métricas relevantes no contexto da Programação Orientação a Objetos (POO) e as anomalias de código são detectadas automaticamente por meio de estratégias de detecção;

- **Priorização de Anomalias**. As anomalias de código são avaliadas de acordo com sua importância com base em diferentes critérios de priorização, que podem ser facilmente customizados pelos usuários de acordo com suas preferências ou mesmo de acordo com características do projeto;

- **Escolha da Abordagem de Detecção**. A ferramenta foi concebida para implementar as abordagens interativa e não-interativa de detecção de anomalias de código. O desenvolvedor pode optar pela escolha de alguma das estratégias de acordo com a adequabilidade delas à sua atividade de programação;

- **Formas de Visualização**. Uma vez que a ferramenta foi projetada para dar suporte a abordagens distintas, formas de visualização dos resultados devem ser aderentes a tais

abordagens. Enquanto na abordagem DI os resultados mais localizados (i.e., associados ao arquivo de código-fonte que o desenvolvedor está trabalhando atualmente) são requeridos, no caso da abordagem DNI existe uma necessidade de exibir resultados mais globais (i.e., associados ao projeto de software como um todo).

O modo de detecção contínua de anomalias e a capacidade de interação direta com fragmentos de código anômalos são as principais características embutidas na ferramenta. Tais características indicam a necessidade de expor resultados de detecção mais localizados aos fragmentos de código em uso por parte do desenvolvedor. No entanto, uma imagem global dos resultados da detecção de anomalias ainda é útil especialmente para desenvolvedores que não detêm conhecimento do sistema e podem precisar entender suas características gerais, bem como os problemas legados para os fragmentos de código nos quais eles trabalharão.

A ferramenta é baseada na análise da qualidade do código-fonte através de métricas de software. Uma vez que a ferramenta foi concebida para análise de projetos na linguagem Java, a ferramenta *Eclipse ConCAD* utiliza como entrada os diversos arquivos com extensão ".java", componentes do projeto. Estes arquivos são analisados por meio da sua representação através de uma Árvore de Sintaxe Abstrata (do inglês, *Abstract Syntax Tree* [AST]). A representação do código em AST é utilizada como entrada para o cálculo das métricas. A *Eclipse ConCAD* provê suporte ao cálculo de 30 tipos distintos de métricas de qualidade de software. Tais métricas associadas ao Paradigma Orientado a Objetos (POO) foram escolhidas em virtude de serem componentes da estratégia de detecção de estudos relevantes da área [57][58][69][122]. A lista completa dessas métricas é descrita no Apêndice A.

Estas métricas são utilizadas para compor as estratégias de detecção para os diversos tipos de anomalias. Conforme discutido na Seção 2.3, uma estratégia de detecção consiste em uma expressão quantificável e lógica onde fragmentos de código que não estão em conformidade com a regra são detectados. O processo para a definição das estratégias de detecção contempla os seguintes passos: (i) seleção de métricas; (ii) definição dos limiares de aceitação; e (iii) composição da regra de detecção. É importante mencionar que a ferramenta fornece suporte semi-automatizado para os passos acima descritos. Isso visa prover suporte à configuração da estratégia de detecção de anomalias a partir de preferências do usuário que podem estar previamente armazenadas em banco de dados. A lista completa das estratégias de detecção pré-configuradas na ferramenta é descrita no Apêndice B.

Baseado nas métricas de software e nas estratégias de detecção pré-configuradas, a ferramenta provê suporte inicial à detecção de 10 tipos distintos de anomalias de código. É importante destacar que com base nas diversas métricas já implementadas na ferramenta, novas e diferentes estratégias de detecção podem ser facilmente configuráveis visando expandir as anomalias suportadas pela ferramenta. A lista de anomalias suportadas pela ferramenta é a seguinte: *Brain Class, Brain Method, Feature Envy, Data Class, Dispersed Coupling, God Class, Intensive Coupling, Refused Bequest, Tradition Breaker* e *Shotgun Surgery*. Uma descrição mais detalhada a respeito de cada uma dessas anomalias é apresentada no Apêndice C.

A quantidade de anomalias de código pode representar uma grande dificuldade ao analisar sistemas de grande porte. Portanto, a ferramenta provê meios de priorizar cada instância dos 10 tipos distintos de anomalias. Desde a sua concepção, esta ferramenta foi planejada para ser flexível o suficiente para encontrar um equilíbrio entre (i) critérios de priorização que requerem pouca intervenção dos usuários, mas funcionam melhor depois de várias versões do sistema, e (ii) critérios que contam com (uma quantidade razoável de) informações externas fornecidas com antecedência pelo usuário, mas produzem bons resultados para as versões recentes do sistema. Isso permite que os usuários priorizem a inspeção e a correção dos fragmentos mais críticos de acordo com suas preferências.

### 7.3.2 Decisões Arquiteturais

Um dos maiores desafios de implementação foi garantir que a ferramenta respondesse a cada mudança de código sem afetar o desempenho geral do ambiente de desenvolvimento integrado Eclipse. Embora este ambiente forneça mecanismos para rastrear mudanças, a análise de código-fonte deve ser executada em segundo plano (e.g., computação de métricas ou detecção de anomalias), conduzindo a um uso intensivo de recursos, tanto em termos de processamento quanto de uso de memória. Isso conduziu às seguintes decisões arquiteturais para tornar viável o uso e o experimento com a ferramenta:

- **Execução *multithread***. Para manter os resultados atualizados, foram definidos *listeners* em dois tipos de eventos: (i) alterações de código salvas no arquivo atual; e (ii) mudar o foco do editor para outro arquivo. Quando esses eventos ocorrem, o *Java*

*Builder* é lançado e a ferramenta inicia um trabalho de marcador pelo qual o arquivo atual é analisado. Este trabalho sempre é executado em uma *thread* separada, garantindo que a interface do usuário permaneça responsiva. Além disso, a fim de evitar o perigo de vários trabalhos de marcador estando simultaneamente ativos, foram utilizadas estruturas de dados simultâneas a fim de garantir que todos os trabalhos de marcador desatualizados sejam cancelados (e.g., quando o foco do editor é alternado para um arquivo de origem diferente).

- **Redução de uso de ASTs durante a construção do modelo**. No Eclipse JDT (ambiente de desenvolvimento Java), construir um AST é custoso, tanto em termos de processamento quanto de consumo de memória. Assim, em vez de envolver grandes objetos baseados em AST, a ferramenta utiliza de modo prioritário objetos baseados no *Java Model*. No entanto, evitar objetos AST completamente não é possível porque algumas das métricas envolvem informações de referência cruzada (e.g., acessos variáveis e complexidade ciclomática). Desse modo, o desempenho da ferramenta ainda pode ser otimizado com base na seguinte observação: quando um AST é construído para um método, o AST contém de fato as informações para todos os métodos da mesma unidade de compilação (i.e., arquivo). Assim, no ConCAD, os ASTs são construídos apenas uma vez por arquivo de origem; e quando o AST é criado, a *Eclipse ConCAD* busca as informações necessárias para todos os métodos definidos naquele arquivo. Eventualmente, depois que todas as informações são extraídas, os objetos baseados em AST são descartados da memória.

- **Uso de *cache***. As métricas envolvidas na detecção de anomalias, bem como as informações adicionais fornecidas pelos usuários, envolvem cálculos demorados. A ferramenta é projetada para evitar quaisquer recálculos desnecessários, por meio de *cache* e atualização seletiva. Portanto, para cada elemento de código, as relações centrais são armazenadas em *cache*. Isso permite que métricas e outras propriedades possam ser recalculadas muito rapidamente. Quando ocorre uma mudança, as entidades do programa afetadas pela respectiva mudança são atualizadas, garantindo que nenhum cálculo adicional desnecessário seja executado. No entanto, o armazenamento pode implicar em elevado consumo de memória, e isso pode se tornar um problema para

projetos grandes. Por isso, conforme descrito antes, a ferramenta armazena principalmente referências a objetos *Java Model* e mantém o uso e armazenamento de objetos baseados em AST em mínimo possível. Como resultado, a ferramenta tende a analisar grandes projetos Java com um mínimo de sobrecarga em seu desempenho.

### 7.3.3 Funcionamento

A interface de usuário da ferramenta é completamente integrada ao Eclipse. Na Figura 7.2, ilustra-se a interface da ferramenta com rótulos de letras que são detalhadas a seguir.

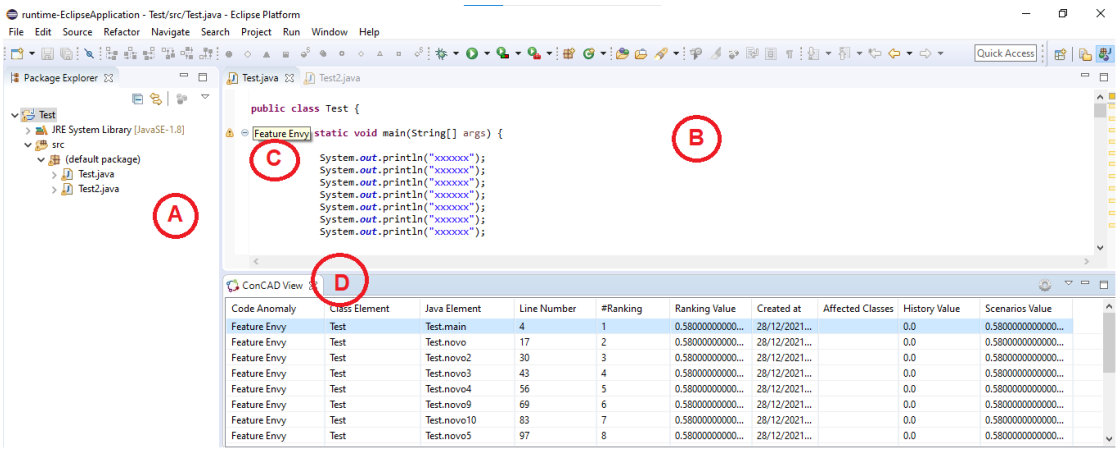


Figura 7.2: Visão Geral da Ferramenta *Eclipse ConCAD*.

Depois que um projeto é carregado no ambiente *Eclipse*, o usuário pode instruir a ferramenta para identificar e analisar as possíveis anomalias de código de um dado fragmento. Isso é feito clicando no botão "*ConCAD Smell Detector -> Enable ConCAD*" em um menu contextual no projeto (Figura 7.2.A). Depois de clicar nesta opção, o usuário muda seu contexto para realizar alguma atividade de programação em um arquivo de código particular. Nesta ocasião o usuário necessita focar suas atividades de programação em fragmentos de código que são parte constituinte do arquivo que encontra-se ativo no Ambiente de Desenvolvimento Integrado (Figura 7.2.B).

O usuário realiza alguma atividade de programação no fragmento de código. Neste momento o mecanismo de detecção de anomalias atua - independentemente de uma requisição explícita deste usuário - calculando métricas associadas ao fragmento, bem como aos arquivos de código-fonte relacionados. Este mecanismo realiza uma análise detalhada dos

valores das métricas e, caso ocorra uma violação nas estratégias previamente definidas para detecção, o desenvolvedor é alertado sobre a presença de anomalias através de marcadores no fragmento de código considerado (Figura 7.2.C). Finalmente, o desenvolvedor também pode ter uma visão mais global dos resultados associados às anomalias de código presentes no projeto. Todas essas ocorrências estão listadas e classificadas na *ConCAD View* (Figura 7.2.D).

É importante destacar que os marcadores da ferramenta são dinâmicos: conforme um novo código é escrito ou modificado, novos marcadores podem aparecer ou os existentes desaparecerem. Embora as anomalias sejam detectadas usando regras baseadas em métricas, o problema é descrito em termos de conceitos de projeto, ao invés de números. Nesse sentido, a ferramenta lista as entidades e relações reais que contribuem para uma anomalia de código específica, ajudando o usuário da ferramenta na correta identificação e remoção das anomalias.

Por exemplo, no caso de ocorrer *Feature Envy*, a regra de detecção afirma que: (i) o método usa muitos atributos externos, de poucas classes; e (ii) usa nenhum ou quase nenhum atributo de sua classe. Assim, a ferramenta mostrará os atributos externos reais que são usados, bem como os atributos de sua própria classe que o método está usando (se houver). Além disso, a descrição contém um conjunto de *hiperlinks* que permitem ao desenvolvedor se concentrar em explorar em detalhes o contexto relevante de uma determinada anomalia.

As demais funcionalidades da ferramenta, incluindo descritivo das telas da interface com usuário, são descritas no Apêndice D.

## 7.4 Execução do Experimento

Os participantes realizaram as tarefas experimentais associadas à análise e inspeção de código manipulando arquivos da linguagem Java extraídos do projeto "*Java Core Library*" [4]. Este projeto foi selecionado por se tratar de um sistema de código aberto, facilitando a replicação deste estudo por pesquisadores independentes. Para detecção de anomalias, os participantes manipularam dois arquivos de código com tamanho (i.e., em termos de módulos e linhas de código) e número semelhantes de anomalias de código (i.e., cerca de 25 ocorrências de oito tipos de anomalias diferentes).

Para a realização das atividades experimentais, os participantes foram divididos em dois grupos: controle e experimental. Os integrantes do primeiro grupo irão realizar as atividades mediante suporte da abordagem DNI. Os integrantes do segundo grupo irão realizar as atividades mediante suporte da abordagem DI. Os grupos foram balanceados em termos de tempo de experiência e formação para evitar qualquer tipo de viés nos resultados obtidos a partir das atividades experimentais.

Os participantes foram submetidos ao "cruzamento fatorial de dois tratamentos" [121] para as tarefas de detecção de anomalias de código. O cruzamento estabelece diferentes sequências de aplicação de tratamento (isto é, grupos experimentais), e os participantes são atribuídos a essas sequências. Na Tabela 7.3 ilustram-se quatro sequências possíveis através do cruzamento de tratamentos (i.e., abordagens DI e DNI) e objetos (i.e., arquivos de código A e B). Além disso, para contrabalancear alguns dos efeitos da aplicação dos tratamentos ou objetos em uma ordem particular, cada sequência foi realizada por quatro participantes (i.e., dois desenvolvedores e dois estudantes).

Tabela 7.3: Cruzamento Fatorial de Dois Tratamentos.

| | **Período 1** | **Período 2** |
|---|---|---|
| **Sequência I** | DI, Arquivo de código A | DNI, Arquivo de código B |
| **Sequência II** | DNI, Arquivo de código A | DI, Arquivo de código B |
| **Sequência III** | DI, Arquivo de código B | DNI, Arquivo de código A |
| **Sequência IV** | DNI, Arquivo de código B | DI,Arquivo de código A |

Para ter confiança nos resultados associados às atividades experimentais, era necessário ter um "oráculo" representando a lista de anomalias de código que realmente representam problemas de manutenibilidade no sistema. Para a geração do oráculo foram realizadas três atividades: (i) os códigos foram submetidos à ferramenta JSpirit [122] para obter métricas de qualidade de software; (ii) essas métricas tinham limites visando a identificação "automática" da instância de anomalias de código; e (iii) essas instâncias foram analisadas e validadas por dois pesquisadores especialistas na detecção de anomalias de código com uso da inspeção manual. Após a realização dessas atividades, gerou-se um oráculo contendo as instâncias das anomalias de código.

O ambiente de execução das tarefas experimentais, incluindo os arquivos e suporte ferramental, foi disponibilizado aos participantes. Além disso, cada tarefa experimental foi

conduzida individualmente com um pesquisador como um "observador" e outro como um "auxiliar". O tempo máximo que cada participante tinha disponível para detectar anomalias de código e identificar oportunidades de refatoração era de 60 minutos. Uma descrição detalhada das tarefas experimentais pode ser encontrada no material suplementar [4]. Cada uma das fases do experimento será detalhada nos itens que seguem, conforme ilustrado na Figura 7.3.

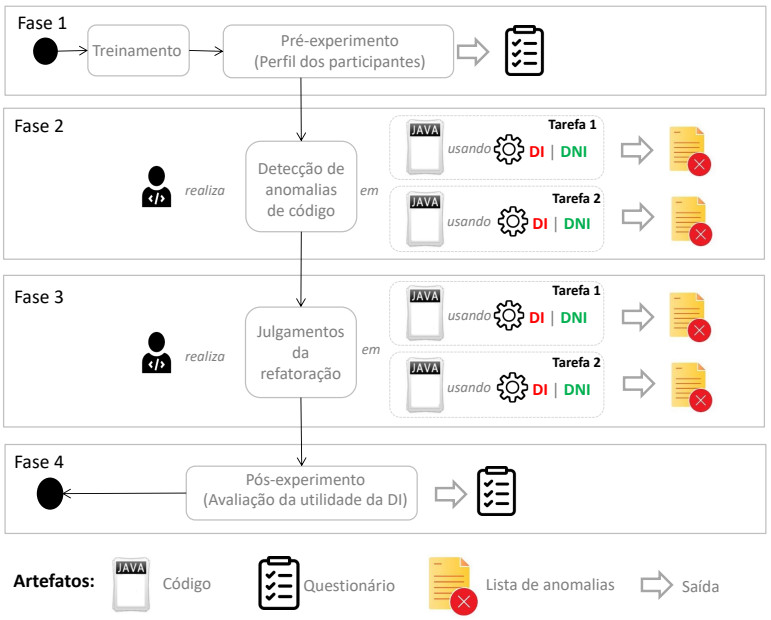


Figura 7.3: Etapas de Execução do Experimento.

**Fase 1: Pré-experimento**. Inicialmente, todos os participantes receberam material com a definição de oito anomalias e exemplos da ocorrência de cada uma delas. Um prazo de 15 minutos (máximo) foi concedido para que os participantes entendessem essas definições. Em seguida, os participantes passaram por um treinamento sobre as abordagens de DI e DNI utilizadas no experimento e sobre o ambiente de desenvolvimento integrado. Todas essas ações objetivaram o nivelamento do conhecimento dos participantes. Finalmente, todos os participantes foram convidados a responder um questionário contendo informações demográficas (e.g., formação, tempo de experiência, conhecimento de técnicas, entre outros). Essa ação tinha o objetivo principal de identificar o perfil dos participantes.

**Fase 2: Detecção de anomalias de código**. Os participantes realizaram análise de código a fim de detectar oito instâncias diferentes de anomalias de código (i.e., *God Class, Long Method, Feature Envy, Dispersed Coupling, Intensive Coupling, Tradition Breaker, Shotgun*

*Surgery* e *Refused Bequest*). Para tanto, os participantes realizaram duas tarefas associadas à detecção de anomalias durante a análise de código: (i) uma com suporte da abordagem DI; e (ii) outra com suporte da abordagem DNI. Como mencionado anteriormente, os indivíduos foram submetidos ao "tratamento cruzado" (Tabela 7.3) para realização dessas tarefas. Durante o experimento, os participantes poderiam concordar ou discordar da detecção proposta pelas abordagens de detecção. Assim, "falsos positivos" decorrentes das abordagens poderiam ser omitidos quando os participantes utilizassem seu conhecimento para tomar decisões sobre as instâncias de anomalias de código existentes. Os dados relacionados à detecção de anomalias de cada tarefa foram transcritos para uma lista contendo dados relacionados ao número total (T) de anomalias de código detectado (AD), verdadeiros positivos (VP), falsos positivos (FP) e falsos negativos (FN). Os resultados das tarefas transcritos nestas listas foram usados para calcular as medidas de *recall* e *precision* e, consequentemente, avaliar as hipóteses (H4.1 e H4.2) do presente estudo.

**Fase 3: Julgamento da refatoração**. Nesta fase os participantes realizaram julgamentos de refatoração a respeito da ocorrência de uma anomalia de código particular (i.e., *Feature Envy*). Cada participante deveria inferir a respeito da aplicabilidade de uma ação de refatoração. Em caso positivo, o participante deveria responder adequadamente às seguintes questões: (i) quão espalhada estava a instância de anomalia de código; (ii) qual a probabilidade de remover a instância de anomalia; e (iii) quais ações de refatoração seriam requeridas para remoção dessa instância de anomalia. As questões acima estão diretamente relacionadas aos julgamentos de refatoração [38][3]. É importante mencionar que nessa fase os participantes realizaram duas tarefas associadas aos julgamentos de refatoração através do uso das abordagens DI e DNI.

**Fase 4: Pós-experimento**. Nesta etapa, os participantes foram questionados sobre alguns aspectos do uso da abordagem DI utilizada. Para tanto, os participantes foram submetidos a um questionário contendo diversas questões abertas e fechadas. Na análise dos dados, considerou-se a concordância (C) ou Discordância (D) - e suas diferentes formas de classificação, como Totalmente (T) e Parcialmente (P) - para algumas afirmações relacionadas à utilidade da Detecção Interativa.

## 7.5 Resultados

Nesta seção são apresentados os dados relacionados ao perfil dos participantes e descritos os resultados relacionados às tarefas experimentais. A seguir, são discutidos alguns aspectos relacionados ao uso da Detecção Interativa (DI) no contexto da análise de código e algumas implicações para pesquisadores e profissionais. Informações detalhadas sobre cada uma das atividades experimentais podem ser encontradas no material suplementar [4] deste estudo.

### 7.5.1 Atividades Pré-Experimento (Fase 1)

A primeira fase do experimento controlado envolveu a realização de um treinamento para garantir que os participantes adquirissem as habilidades e informações necessárias para realizar o experimento com sucesso e produzir resultados precisos e confiáveis. Essa atividade foi realizada ao longo de dois encontros de cerca de 90 minutos. O primeiro encontro objetivou nivelar o conhecimento sobre anomalias de código e refatoração. O segundo encontro objetivou o nivelamento em termos do uso do ambiente de desenvolvimento integrado bem como as abordagens de detecção. Cada encontro foi composto por uma fase expositiva e uma fase prática. Ao final dos encontros todos os participantes foram submetidos a uma avaliação para assegurar a compreensão dos conceitos e técnicas necessários à realização das atividades experimentais.

Em seguida, um questionário foi submetido com vistas a determinar o perfil dos participantes. No total, 16 participantes realizaram o experimento controlado, sendo oito estudantes de graduação e oito desenvolvedores profissionais. No questionário adotou-se uma escala de 0 a 4, onde 0 significa “não proficiente” e 4 “muito proficiente”. Todos os participantes possuíam experiência prévia na utilização da IDE Eclipse, e 50% tinham entre cinco e oito anos de experiência em desenvolvimento de software. Em relação à linguagem Java, 32% dos participantes responderam 2 e 68% dos participantes responderam 3. Por sua vez, aproximadamente 80% dos participantes responderam 1 ou 2 sobre a proficiência na detecção de anomalias. Em contraste, cerca de 60% dos participantes responderam 3 ou 4 em relação à experiência de refatoração. No geral, o perfil dos participantes atendeu às expectativas de nosso estudo, pois eles apresentaram conhecimento intermediário em Java, detecção de anomalias de código e refatoração.

### 7.5.2 Detecção de Anomalias (Fase 2)

A segunda fase do experimento controlado envolveu as tarefas relacionadas à detecção de anomalias de código usando abordagens DI e DNI no contexto da análise de código. É importante mencionar que o tempo limite de 30 minutos foi estabelecido para realização de cada tarefa de detecção de anomalias. Na Tabela 7.4 descrevem-se os resultados para todos os participantes, considerando os seguintes componentes de medição de eficácia: Anomalias Detectadas (AD); Verdadeiros Positivos (VP); Falsos Positivos (FP); e Falsos Negativos (FN).

Tabela 7.4: Resultados da Detecção de Anomalias.

| | **DNI (Grupo Controle)** | | | | **DI (Grupo Experimental)** | | |
|---|---|---|---|---|---|---|---|
| | **VP** | **FP** | **FN** | | **VP** | **FP** | **FN** |
| Desenvolvedor 1 | 5 | 1 | 17 | Desenvolvedor 1 | 8 | 1 | 14 |
| Desenvolvedor 2 | 7 | 2 | 15 | Desenvolvedor 2 | 16 | 2 | 6 |
| Desenvolvedor 3 | 10 | 1 | 12 | Desenvolvedor 3 | 16 | 3 | 6 |
| Desenvolvedor 4 | 6 | 2 | 16 | Desenvolvedor 4 | 10 | 2 | 12 |
| Desenvolvedor 5 | 10 | 2 | 12 | Desenvolvedor 5 | 13 | 1 | 9 |
| Desenvolvedor 6 | 11 | 3 | 11 | Desenvolvedor 6 | 14 | 3 | 8 |
| Desenvolvedor 7 | 7 | 1 | 15 | Desenvolvedor 7 | 9 | 2 | 13 |
| Desenvolvedor 8 | 8 | 2 | 14 | Desenvolvedor 8 | 13 | 1 | 9 |
| **Soma desenvolvedores** | **64** | **14** | **112** | **Soma desenvolvedores** | **99** | **15** | **77** |
| **Média desenvolvedores** | **8** | **1.75** | **14** | **Média desenvolvedores** | **12.37** | **1.87** | **9.62** |
| Estudante 1 | 5 | 1 | 17 | Estudante 1 | 11 | 2 | 11 |
| Estudante 2 | 5 | 1 | 17 | Estudante 2 | 9 | 2 | 13 |
| Estudante 3 | 8 | 2 | 14 | Estudante 3 | 12 | 3 | 10 |
| Estudante 4 | 6 | 1 | 16 | Estudante 4 | 10 | 1 | 12 |
| Estudante 5 | 6 | 3 | 16 | Estudante 5 | 9 | 2 | 13 |
| Estudante 6 | 5 | 2 | 17 | Estudante 6 | 9 | 3 | 13 |
| Estudante 7 | 6 | 2 | 16 | Estudante 7 | 9 | 1 | 13 |
| Estudante 8 | 5 | 3 | 17 | Estudante 8 | 10 | 1 | 12 |
| **Soma estudantes** | **46** | **13** | **130** | **Soma estudantes** | **79** | **15** | **97** |
| **Média estudantes** | **5.75** | **1.87** | **16.25** | **Média estudantes** | **9.87** | **1.87** | **12.13** |
| **Soma Total** | **110** | **27** | **242** | **Soma Total** | **178** | **30** | **174** |
| **Média Total** | **6.87** | **1.81** | **15.13** | **Média Total** | **11.12** | **1.87** | **10.87** |

### Abordagem DI aumenta AD e VP

Em relação aos resultados de AD, o interesse do experimento era mensurar o número total de instâncias de anomalias que o desenvolvedor poderia detectar, independentemente se a instância de anomalias de código representava um problema de manutenção (ou seja, VP) ou não (ou seja, FP). Em outras palavras, o AD pode ser obtido pela soma do VP e FP. Os resultados experimentais mostraram que os participantes que utilizaram DNI obtiveram 137 AD, enquanto que com a abordagem DI obtiveram 208 AD. Conclui-se que o uso da abordagem DI leva os participantes a aumentar o número de AD em cerca de 50% em comparação com o utilização da DNI. Em relação aos resultados associados ao VP, notou-se incremento semelhante. Participantes usando a abordagem DNI detectaram 110 VP, enquanto que usando DI detectaram 178 VP. Portanto, o uso de DI aumentou cerca de 60% do total de VP na detecção de anomalias de código. Além disso, os resultados estatísticos relacionados a VP (*alfa* = 0,05, p = 0,00022, tamanho do efeito = 0,88, SD = 1,6931, valor Z = -3,515) e AD (*alfa* = 0,05, p = 0,00044, tamanho do efeito = 0,43, SD = 19,34, valor Z = -3,5162) obtidos a partir do uso das abordagens DI e DNI foram significantes através do uso do teste de Wilcoxon [55].

### Abordagem DI diminui FN

Em relação aos resultados de FN, o interesse do estudo era conhecer quantas instâncias de anomalias de código que representavam problemas de manutenção (ou seja, VP) os participantes não conseguiram detectar. Os resultados experimentais revelaram que os participantes identificaram 242 FN com a abordagem DNI, enquanto os participantes com DI conseguiram reduzir o número de FN em 28%, totalizando 174 FN. O uso da abordagem DI levou os participantes a identificarem a maioria das instâncias de anomalias de código que representavam problemas de manutenção. Embora se tenha notado um aumento no FP, ainda existem oportunidades para melhorias no mecanismo de detecção. Além disso, os dados relacionados a FN gerado com o auxílio das abordagens DI e DNI foram estatisticamente significantes usando o teste Wilcoxon (*alfa* = 0,05, p = 0,000219, tamanho do efeito = 0,88, SD = 1,6931, valor Z = 3,515) [55].

**Abordagem DI aumenta FP**

Com relação aos resultados de FP, o interesse do estudo era identificar quantas instâncias de anomalias o participante detectaria e se essas instâncias realmente representavam problemas de manutenção. Observou-se que os participantes identificaram 27 FP ao utilizar DNI. O número de FP é aproximadamente 10% maior em comparação com o uso da abordagem DI (i.e., 30 FP). Além disso, os resultados estatísticos relacionados a VP (p = 0,00022, tamanho do efeito = 0,88, SD = 1,6931, valor Z = -3,515) e FP (p = 0,0218, tamanho do efeito = 0,25, SD = 0,9105, valor Z = -0,7765) obtidos a partir do uso das abordagens DI e DNI foram significantes usando o teste de Wilcoxon [55].

Um achado importante deste estudo é o efeito do nível de experiência dos participantes nos resultados do estudo. Em relação aos resultados associados a AD, o uso da abordagem DI conduziu os estudantes a aumentar AD em cerca de 65% em comparação com o uso da abordagem DNI, enquanto considerando os desenvolvedores esse aumento foi de cerca de 35%. Da mesma forma, nos resultados associados a TP, a abordagem DI levou os estudantes a aumentar o TP em cerca de 70%, enquanto considerando os desenvolvedores, esse aumento foi de cerca de aproximadamente 50%. Por fim, considerando os resultados associados a FP, a abordagem de DI leva os estudantes a aumentarem cerca de 15%, enquanto considerando os desenvolvedores, esse aumento foi de cerca de 7%.

Uma possível razão para esses resultados é que os desenvolvedores (com mais experiência) ao longo das suas atividades profissionais foram expostos a uma ampla variedade de bases de código. Eles tendem a desenvolver um senso do que constitui uma boa qualidade de código e podem estar mais bem preparados para identificar anomalias de código. Este perfil pode estar mais propenso a detectar problemas e padrões no código que podem indicar a presença de uma anomalia. Além disso, eles também têm uma compreensão mais profunda das consequências dessas anomalias na manutenção, escalabilidade e desempenho do código. No entanto, é importante observar que mesmo desenvolvedores experientes podem ter opiniões divergentes sobre o que constitui uma anomalia, uma vez que sua estratégia de detecção pode ser subjetiva.

### Avaliação das medidas de eficácia

Para fornecer uma perspectiva adicional sobre a eficácia das abordagens DI e DNI, foram calculadas as medidas de eficácia *Precision* (P) e *Recall* (R) a partir do uso dos componentes descritos na Tabela 7.4. Os resultados associados a essas medidas podem ser vistos na Figura 7.4.

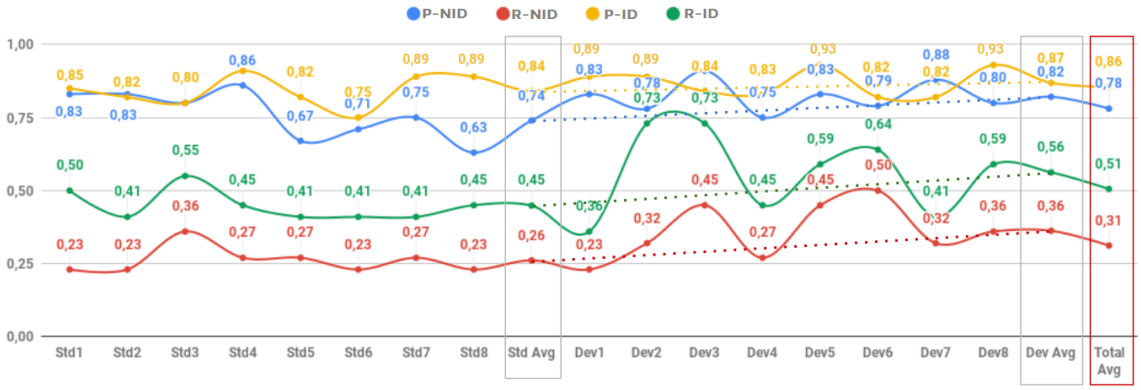


Figura 7.4: Resultados em Termos de *Precision* e *Recall*.

**Abordagem DI aumenta a *precision***. O valor médio desta medida com uso da abordagem DI foi de 0,86, enquanto com uso da DNI atingiu 0,78. A diferença nos resultados não foi significativa considerando uma variação de cerca de 11%. Também observou-se que a experiência de trabalho dos participantes afetou diretamente os resultados. Embora o uso da abordagem DI aumentou o número de VP, ela também contribuiu para elevação do FP, impactando diretamente os valores da medida *precision*, uma vez que esses componentes de medição de eficácia são usados para calcular esta medida. Na Figura 7.4 indica-se que os participantes que utilizaram DI obtiveram melhor *precision* do que os participantes que utilizaram a abordagem DNI. Finalmente, uma análise de comparação considerando os resultados da medida *precision* alcançada com ambas as técnicas foi estatisticamente significante mediante uso do *Paired T-Test* [55] (*alfa* = 0,05, p = 0,01161, SD = 6,9858, t = 2,6823, tamanho do efeito = 0,47). Portanto, *é possível confirmar a primeira hipótese (H4.1)*.

É importante notar que a abordagem DI promoveu uma elevação significativa na medida *precision* considerando apenas a amostra dos estudantes, visto que eles alcançaram 0,74 com o uso da abordagem DNI. Ao usar DI, obteve-se um aumento de mais de 13% (i.e, 0,84). Contudo, resultados semelhantes não podem ser vistos na amostra dos desenvolvedores, na

qual observou-se um aumento de apenas 6% na mesma comparação. Esses resultados podem indicar que o nível de conhecimento dos participantes no experimento pode ter influenciado os resultados. Quanto mais baixo for o nível de conhecimento dos assuntos (i.e., amostra de estudantes), maior será o benefício que essa amostra pode ter com o uso de uma abordagem DI em termos da medida *precision*.

**Abordagem DI aumenta o *recall*.** Com base nos resultados associados a FN e VP, espera-se que o uso da abordagem DI contribua significativamente para melhorar o *recall* na detecção de anomalias. Em média, observou-se que os participantes que utilizaram a DI obtiveram 0,51, enquanto que os participantes que utilizaram a DNI atingiram 0,31, representando uma diferença de cerca de 60% a favor da abordagem DI. Ao analisar diferentes amostras, os desenvolvedores melhoram os valores da medida *recall* em 50%, enquanto os estudantes melhoram em cerca de 70%. Finalmente, os resultados relacionados à *recall* nesta tarefa por meio das abordagens DI e DNI foram estatisticamente significantes usando *Paired T-Test* [55] (*alfa* = 0,05, p = 0,0002951, SD = 11,2089, t = 4,8890, tamanho do efeito = 0,86).

Também observou-se que a medida *recall* sofre influência direta em relação à experiência de trabalho dos participantes, o que implica que quanto menor o nível de conhecimento dos participantes, maior o benefício que esta amostra pode ter com o uso da abordagem DI em termos de medida de *recall*. Além disso, o uso de DI pode afetar diretamente os valores desta medida, uma vez que sua utilização implica que os participantes interajam com os elementos de código afetados durante a análise de fragmentos de código. Portanto, os desenvolvedores podem observar uma cobertura melhor com a abordagem DI ao detectar corretamente instâncias de anomalias de código. Finalmente, *é possível confirmar a segunda hipótese (H4.2)*, uma vez que o uso do DI levou a melhores resultados de *recall* em comparação com a abordagem DNI.

Vale ressaltar que os resultados não indicaram impacto negativo no uso da abordagem DI para detecção de anomalias. É provável que os participantes se beneficiem da detecção de anomalias antecipadamente, quando recebem *feedback* de modo regular e contínuo. Além disso, a disponibilidade constante e a maior quantidade de informações por meio da DI levaram os participantes a aceitarem um maior número de anomalias de código sugeridos pela abordagem. Os dados descritos na Tabela 7.4 permitem confirmar esta suposição. Desenvolvedores mais experientes usando DI obtiveram menos FPs do que os estudantes (com menos

experiência de trabalho) usando a mesma abordagem. Da mesma forma, os desenvolvedores identificaram um número maior de VP em comparação com os estudantes.

### 7.5.3 Julgamento da Refatoração (Fase 3)

Na terceira fase, os participantes realizaram julgamentos de refatoração usando ambas as abordagens de detecção. No contexto desta atividade, foi analisado se os participantes realizaram refatoração ineficaz (RI) ou refatoração eficaz (RE) relacionada à ocorrência da anomalia "*Feature Envy*'. Esta fase também foi composta com duas tarefas onde todos os participantes foram submetidos ao cruzamento fatorial. Para cada tarefa de julgamento de refatoração foi concedido o tempo máximo de 15 minutos. Os resultados gerais sobre a (in)eficácia da proposição das ações de refatoração, com base na detecção de anomalias no contexto da análise de código, são ilustrados na Tabela 7.5.

Tabela 7.5: Resultados dos Julgamentos da Refatoração.

| **Participante** | **DNI (Grupo Controle)** | | **Participante** | **DI (Grupo Experimental)** | |
|---|---|---|---|---|---|
| | **RI** | **RE** | | **RI** | **RE** |
| Desenvolvedor 1 | - | x | Desenvolvedor 1 | - | x |
| Desenvolvedor 2 | - | x | Desenvolvedor 2 | - | x |
| Desenvolvedor 3 | - | x | Desenvolvedor 3 | - | x |
| Desenvolvedor 4 | - | x | Desenvolvedor 4 | - | x |
| Desenvolvedor 5 | - | x | Desenvolvedor 5 | - | x |
| Desenvolvedor 6 | x | - | Desenvolvedor 6 | - | x |
| Desenvolvedor 7 | - | x | Desenvolvedor 7 | - | x |
| Desenvolvedor 8 | x | - | Desenvolvedor 8 | x | - |
| **Total Des.** | **2** | **6** | **Total Des.** | **1** | **7** |
| Estudante 1 | x | - | Estudante 1 | - | x |
| Estudante 2 | - | x | Estudante 2 | - | x |
| Estudante 3 | x | - | Estudante 3 | x | - |
| Estudante 4 | - | x | Estudante 4 | - | x |
| Estudante 5 | - | x | Estudante 5 | - | x |
| Estudante 6 | x | - | Estudante 6 | x | - |
| Estudante 7 | - | x | Estudante 7 | - | x |
| Estudante 8 | x | - | Estudante 8 | - | x |
| **Total Est.** | **4** | **4** | **Total Est.** | **2** | **6** |
| **Amostra Total** | **6** | **10** | **Amostra Total** | **3** | **13** |

**Abordagem DI eleva ações de RE**. Observou-se que os participantes realizaram 10 ações de RE ao usar a abordagem DNI, enquanto os participantes realizaram 13 ações de RE ao usar a abordagem DI. O uso da abordagem DI ocasionou um aumento de 30% nas ações de RE realizadas pelos participantes. Similarmente, os participantes realizaram 6 ações de RI ao usar a abordagem DNI, enquanto ao usar DI realizaram apenas 3 ações de RI, representando uma diminuição de 50%. Ao analisar os resultados alcançados pela amostra dos desenvolvedores, os participantes realizaram apenas 1 ação de RI ao usar o DI, enquanto 2 ações de RI foram realizadas quando o DNI foi empregado. A mesma proporção de crescimento (ou seja, 50%) ocorre nos resultados obtidos na amostra de estudantes, pois observamos que 4 ações de RI foram realizadas quando o DNI foi usado, enquanto os participantes que usaram o DI realizaram apenas 2.

Em resumo, observou-se que os participantes que usam DI são propensos a realizar mais ações de RE do que os participantes que usam DNI. Além disso, observou-se que a experiência de trabalho também influencia os resultados dessa tarefa, já que os desenvolvedores realizaram 50% menos ações de RI do que os estudantes. Durante a segunda fase experimental, observou-se que a maioria das ocorrências de FP estava relacionada à anomalia de código *Feature Envy* - onde os estudantes identificaram a maioria dos FP. Pode-se concluir que as ocorrências de FP podem estar diretamente associadas à experiência de trabalho dos desenvolvedores, e o uso da abordagem DI afeta diretamente as ações de refatoração.

Ao analisar os dados coletados associados aos julgamentos de refatoração (Tabela 7.5), conclui-se que o uso da abordagem DI pode induzir os desenvolvedores a realizar mais ações de RE, embora o uso da técnica de DI possa, até certo ponto, também aumentar o número de falsos positivos. Cabe o comentário de que o esforço para realizar esta tarefa pode não contribuir para melhorar a manutenibilidade do sistema se o desenvolvedor realizar ações de refatoração em uma ocorrência de falso positivo de anomalia de código. Características da abordagem DI contribuíram positivamente para que os participantes do experimento inferissem com mais confiança a respeito da dispersão, probabilidade de remoção e ações requeridas para remoção das anomalias.

### 7.5.4 Atividades Pós-Experimento (Fase 4)

A quarta fase do experimento controlado envolveu a coleta de dados através de um questionário em relação a alguns aspectos do uso da abordagem DI. Resumidamente, essa análise compreendeu um conjunto de cinco questões (Q) de múltipla escolha para verificar a concordância (C) ou discordância (D) - e suas diferentes formas de classificação, como Totalmente (T) e Parcialmente (P) - para algumas afirmações relacionadas à utilidade da abordagem DI. No geral, obteve-se *feedback* positivo quanto ao uso da abordagem DI, considerando que cerca de 90% dos participantes concordaram (total ou parcialmente) com as afirmações relacionadas à sua utilidade. Os resultados da utilidade da abordagem DI para detecção de anomalias de código são descritos na Tabela 7.6.

Tabela 7.6: Utilidade da Detecção Interativa.

| **Nr** | **Questão** | **DT** | **DP** | **C** | **CP** | **CT** |
|---|---|---|---|---|---|---|
| Q1 | A técnica DI usada no experimento foi útil na realização das tarefas atribuídas? | 0 | 0 | 3 | **8** | 5 |
| Q2 | A técnica DI encontrou informações que eu não teria encontrado tão rapidamente sem ela? | 0 | 1 | 1 | 4 | **10** |
| Q3 | A técnica DI encontrou informações que eu não teria encontrado sem ela? | 0 | 1 | 3 | **6** | **6** |
| Q4 | Sem a técnica DI, era difícil identificar a ocorrência de todas as dez anomalias de modo concomitante? | 0 | 0 | 0 | 5 | **11** |
| Q5 | Se uma técnica DI estiver disponível, eu a usaria ao codificar? | 0 | 0 | 0 | 7 | **9** |

No geral, os participantes consideraram útil a abordagem DI (ou seja, Q1), pois a maioria das respostas se concentrou nas colunas PC e TC - cerca de 90% concordaram (total ou parcialmente) com as afirmações relacionadas à utilidade da técnica DI. Em apenas duas questões (ou seja, Q2 e Q3) notou-se algum grau de discordância. É importante mencionar que os dois participantes que discordaram das afirmações eram desenvolvedores. Essas questões dizem respeito à incapacidade de detectar anomalias sem o auxílio da abordagem DI. É provável que o nível de experiência desses desenvolvedores tenha contribuído para a escolha das respostas.

Em seguida, realizou-se um conjunto de cinco questões abertas, e as respostas foram rotuladas como: (i) Concordância (C), que representa a concordância do participante (total ou parcialmente) com o questionamento; (ii) Discordância (D), que representa que o participante discorda (total ou parcialmente). Os resultados da pseudo-entrevista foram resumidos na Tabela 7.7.

Tabela 7.7: Resultados da Pseudo-Entrevista.

| Nr | Questão | C | D |
|---|---|---|---|
| Q6 | Usando essa técnica DI (durante a codificação), você acredita que ela chamaria sua atenção nos momentos certos? | **11** | 5 |
| Q7 | Você achou que a quantidade e disponibilidade de informações fornecidas por essa técnica impactou negativamente sua atividade de programação? | 6 | **10** |
| Q8 | A ferramenta pode deixá-lo mais confiante sobre seus julgamentos sobre ações de refatoração? | **14** | 2 |
| Q9 | Você acha que ajudou você (por exemplo, com mais e melhores informações) a realizar tais julgamentos? | **14** | 2 |
| Q10 | Você gostaria de mudar algo neste (ou em qualquer outro) detector de anomalias? | 6 | **10** |

Em relação a **Q6 e Q7**, Tabela 7.7, verificou-se que a maioria dos participantes acredita que o uso da técnica DI chama a atenção nos momentos certos, e as características da DI não são "perturbadoras" porque a técnica pode ser desativada a qualquer momento. Por sua vez, alguns participantes mencionaram que usar a técnica DI pode ser perturbador se os desenvolvedores estiverem interessados em identificar apenas algumas anomalias de código específicas. Isso ocorre pois a abordagem DI fornece os resultados associados a 10 tipos distintos de anomalias sem a possibilidade de desabilitar a detecção de alguns desses tipos. Além disso, os resultados referentes a **Q8 e Q9** revelaram que a maioria dos participantes considerou que o uso da técnica DI permite realizar as ações de refatoração por fornecer mais informações. Os participantes mencionaram que a representação dos resultados da detecção (por meio de gradação de cores), auxiliava na criação de um "mapa mental" que ajuda a verificar a real propagação das anomalias de código no projeto analisado.

Finalmente, na questão **Q10**, foram coletadas as seguintes sugestões para melhorar a técnica DI: (i) fazer os ajustes necessários para uso em versões mais novas do IDE Eclipse;

(ii) expandir o suporte a outros tipos de anomalias; (iii) permitir a análise de anomalias de código previamente escolhidas (nem sempre mostram todas as anomalias ao mesmo tempo); (iv) utilizar gradação de cores para mostrar a "força" associada à ocorrência das anomalias do código e; (v) os "limiares" do mecanismo de detecção de anomalias de código devem ser parametrizados para que o desenvolvedor possa ajustar a estratégia de detecção.

## 7.6 Ameaças à Validade

Nesta seção são discutidas as ameaças à validade do experimento controlado e as ações realizadas com objetivo de mitigá-las. Para tanto, foi seguido o esquema de classificação proposto por Wohlin *et al.* [125].

*Validade de Construção*. Em primeiro lugar, foram consideradas as ameaças à validade associadas: (i) à dificuldade em compreender os arquivos de código escolhidos para o experimento; e (ii) à natureza das tarefas experimentais. Para minimizar a primeira ameaça, os arquivos de código foram selecionados de acordo com seu tamanho, quantidade de anomalias de código e tempo empregado para cada tarefa. Além disso, foi realizado um experimento piloto para ajustar o tempo necessário para conclusão das tarefas experimentais. Com o objetivo de mitigar a segunda ameaça, todas as tarefas experimentais foram monitoradas por um pesquisador especialista em detecção de anomalias de código. Além disso, todos os participantes receberam (i) instruções para o preenchimento dos questionários; e (ii) demonstrações antes da realização das tarefas experimentais.

*Validade Externa*. No contexto das atividades experimentais foi utilizado o IDE Eclipse devido à compatibilidade da abordagem DI. Assim, a experiência dos participantes no uso desta versão IDE pode ter sido prejudicada. No entanto, as observações dos participantes não apresentaram qualquer problema ou influência no uso das funcionalidades desta IDE. Ademais, com o objetivo de minimizar essa ameaça, foi fornecido treinamento específico sobre o uso da versão específica do Eclipse.

*Validade Interna*. A seleção das anomalias de código usadas no experimento controlado pode ser considerada uma ameaça. Restringiu-se à discussão de apenas oito tipos de anomalias de código. Em contraste, Fowler catalogou uma lista com mais de 20 anomalias de código [38]. Portanto, as oito anomalias de código usadas nas tarefas experimentais podem

não ser necessariamente uma amostra representativa das anomalias encontradas nos sistemas de modo geral.

*Validade da Conclusão*. A primeira ameaça está relacionada ao cálculo de medidas de eficácia (e.g., *recall* e *precision*). Os cálculos foram realizados com suporte automatizado da ferramenta R e revisados por dois pesquisadores. Outra ameaça está relacionada ao tamanho da amostra uma vez que apenas 16 participantes entre estudantes e profissionais realizaram o experimento controlado. Os resultados podem ter influência direta do tamanho da amostra e a experiência de trabalho dos participantes nas atividades de detecção e refatoração de anomalias. Para mitigar essa ameaça, foi escolhida uma amostra equilibrada composta por estudantes e desenvolvedores, e foram conduzidas sessões de treinamento para nivelamento de conhecimento dos assuntos relativos a esses tópicos, bem como a adequada execução das atividades experimentais.

*Generalização de resultados*. Apenas um único experimento foi realizado envolvendo um número limitado de participantes e um único sistema alvo. Certamente, mais estudos são necessários antes que os resultados possam ser generalizados. No entanto, foi definida uma configuração experimental bastante comum em empresas de software: desenvolvedores com um nível médio de experiência lidando com uma base de código desconhecida - uma situação que os desenvolvedores frequentemente encontram ao mudar de projeto - e sendo confrontados com tipos de problemas de código (correlacionados) que a abordagem DI é capaz de detectar. No entanto, vários aspectos podem influenciar a generalidade dos resultados: (i) é possível que trabalhar em uma base de código familiar tenha reduzido a lacuna entre as realizações dos dois grupos; (ii) as especificidades do sistema (por exemplo, área de negócio, domínio, tamanho do projeto, linguagem, *framework*, entre outros) podem levar a variações nos resultados; e (iii) usar desenvolvedores com diferentes níveis de experiência pode levar a resultados diferentes.

## 7.7 Considerações Finais do Capítulo

Neste capítulo foi apresentada a primeira avaliação experimental da eficácia da detecção de anomalias por meio das abordagens DI e DNI durante a análise de código. Com base em experimentos envolvendo 16 participantes, foi constatado que a utilização da abordagem DI

pode permitir que os desenvolvedores realizem outras atividades de programação enquanto detectam anomalias, estejam cientes constantemente das anomalias ao analisar diferentes fragmentos de código e detectem antecipadamente um maior número de instâncias graças às informações disponíveis sobre as anomalias.

A avaliação entre DI e DNI na atividade de inspeção de código foi de grande importância por várias razões. Primeiramente, essa avaliação proporcionou *insights* fundamentais sobre a eficácia e os benefícios relativos de cada abordagem. Compreender as diferenças de desempenho e precisão entre DI e DNI é essencial para orientar a seleção da abordagem mais adequada para cenários específicos. Além disso, a avaliação permitiu examinar como a interação humana influenciou o processo de detecção de anomalias. Uma vez que DI introduz um nível de conhecimento humano e *expertise* contextual, isso tende a impactar positivamente a qualidade da análise. Através da comparação direta com a DNI, foi possível determinar que essa interação contribuiu significativamente para a detecção de anomalias de código.

A partir da análise dos resultados, conclui-se que, usando DI, os desenvolvedores puderam identificar até 50% mais instâncias de anomalias de código se comparado à mesma atividade mediante suporte de DNI. Portanto, a abordagem DI permite que os desenvolvedores identifiquem esses sintomas precocemente, o que pode beneficiar a manutenção do sistema. Além disso, esta técnica reduziu em até 10% o número de falsos positivos. Diminuir esse indicador pode economizar tempo e esforço dos desenvolvedores, reduzindo os alarmes falsos que eles precisam investigar, permitindo que eles se concentrem em instâncias reais das anomalias. Finalmente, a utilização da abordagem DI pode reduzir o número de falso negativo em mais de 35%. Diminuir esse indicador ajuda a garantir que quaisquer anomalias sejam identificadas e resolvidas prontamente, contribuindo para melhoria da qualidade geral do código.

Outro aspecto da análise diz respeito às medidas de eficácia obtidas a partir dos indicadores descritos acima. Os resultados revelaram que a abordagem DI melhora cerca de 60% dos valores de *recall* na detecção de anomalias. Similarmente, o uso da DI também elevou os valores de *precision* em cerca de 15%. Também observou-se que a experiência de trabalho dos participantes pode afetar diretamente os resultados associados a essas medidas. Quanto maior a experiência de trabalho dos participantes, maiores os valores observados para as

medidas de eficácia consideradas neste estudo.

Os resultados do experimento controlado descritos no presente trabalho deram suporte para responder a QP4, conforme quadro abaixo:

**QP4 - A aplicação da abordagem de detecção interativa contribui para melhoria da eficácia na detecção de anomalias de código durante a análise de código?**
As medidas de eficácia utilizadas no estudo demonstraram que a abordagem DI foi capaz de detectar mais anomalias de código que realmente representam problemas de manutenibilidade no contexto da análise de código. Por consequência, a abordagem DI se mostrou mais eficaz na realização da detecção de anomalias em comparação com a mesma atividade suportada pela DNI.

Os resultados da avaliação apresentada neste capítulo indicaram que a DI é adequada no contexto da análise de código. Isso significa que esta abordagem pode ser utilizada para verificar o código em busca de instâncias de anomalias de código. No entanto, a abordagem DI pode também ser usada durante o desenvolvimento de código para fornecer *feedback* ao desenvolvedor e ajudar a garantir a qualidade do código durante sua produção, além de contribuir para melhorar suas habilidades na detecção de anomalias.

No Capítulo 8 serão descritos os resultados da segunda avaliação experimental com suporte da abordagem DI quando a atividade de programação é realizada progressivamente (i.e., quando um desenvolvedor está navegando ou editando o código). Assim, será possível explorar mais a fundo as características da abordagem DI visando reunir suas (des)vantagens em relação às abordagens DNI tradicionais durante a detecção de anomalias de código no desenvolvimento de novos módulos de software.

# Capítulo 8

# Avaliação da Abordagem DI no Desenvolvimento de Código

O estudo realizado no Capítulo 7 evidenciou que a utilização da abordagem DI melhora a eficácia na detecção de anomalias durante a análise e inspeção de código. No entanto, é necessário investigar se essa abordagem também é efetiva na detecção de anomalias durante a atividade progressiva de desenvolvimento de software, a fim de avaliar sua promissora aplicação nesse contexto.

Neste capítulo, são apresentados os resultados de uma avaliação experimental da DI durante o desenvolvimento progressivo de software. O uso da abordagem DI no desenvolvimento de software é promissor por diversas razões. A DI permite *feedback* imediato aos desenvolvedores enquanto escrevem ou modificam o código, prevenindo erros futuros. Além disso, é flexível e pode ser personalizada para atender às necessidades específicas do projeto. A DI contribui para o aprendizado constante e compreensão mais profunda do código, enquanto a detecção precoce de anomalias impulsiona a eficiência do ciclo de desenvolvimento, economizando tempo e recursos. Em resumo, a DI pode potencializar o desenvolvimento de software ao oferecer *feedback* imediato, personalização, compreensão profunda do código e melhoria contínua, resultando em um processo de desenvolvimento mais eficaz e de alta qualidade.

É importante mencionar que os resultados desse estudo foram publicados no *International Conference on Software, Telecommunications and Computer Networks* (SoftCOM'23) [12]. Na Seção 8.1 são detalhados os procedimentos metodológicos do estudo. Na Seção

8.2 descreve-se o modo de seleção dos participantes enquanto que na Seção 8.3 detalha-se a execução do experimento. Em seguida, na Seção 8.4 são expostos e discutidos os resultados desta avaliação experimental. Por fim, na Seção 8.5 são elencadas as ameaças à validade do estudo, enquanto que na Seção 8.6 são apresentadas as considerações finais.

## 8.1 Escopo do Experimento

A atividade de codificação é a parte do desenvolvimento de software que envolve a escrita de código para implementação de funcionalidades desejadas em um sistema. É considerada uma atividade crítica pois é onde a lógica e as especificações do software são transformadas em um conjunto de instruções compreendidas pelo computador. A atividade de codificação é feita prioritariamente por desenvolvedores usando linguagens de programação (e.g., Java, Python e C++). Nesta atividade é importante que o código escrito seja claro e compreensível, de modo a garantir que o software funcione conforme previsto pelas suas especificações e restrições.

O estudo descrito no Capítulo 7 apontou que a abordagem de Detecção Interativa (DI) promove uma identificação precoce de anomalias de código, permitindo que os desenvolvedores possam realizar ações de refatoração eficazes no contexto da análise e inspeção de código. De modo análogo, a interação direta com trechos anômalos de código mediante uso da abordagem DI também pode ser benéfica durante a execução de atividades de desenvolvimento. Isso pode evitar que as anomalias de código se acumulem ao longo do tempo, diminuindo potencialmente a quantidade de anomalias remanescentes e, consequentemente, reduzindo o tempo e o esforço necessários para corrigi-las. Embora a utilização da abordagem DI pareça promissora, há uma falta de conhecimento quanto ao impacto desta abordagem na detecção de anomalias de código durante o desenvolvimento de novos módulos de software. Esta observação levanta a seguinte Questão de Pesquisa (QP):

- **QP5 - A aplicação da abordagem de detecção interativa contribui para reduzir a quantidade de anomalias de código remanescentes durante o desenvolvimento do código?**

Para responder esta QP, é importante avaliar se a abordagem DI pode contribuir para a redução da quantidade de anomalias de código remanescentes durante o desenvolvimento do

código. Uma vez que a abordagem DI permite que os desenvolvedores interajam diretamente com trechos anômalos, isso pode possibilitar que os problemas sejam detectados e corrigidos mais rapidamente, antes que possam se acumular e se tornar mais complexos de resolver.

No entanto, é importante notar que o sucesso da aplicação da abordagem DI na redução da quantidade de anomalias de código remanescentes pode depender da capacidade dos desenvolvedores de identificar e corrigir essas anomalias de forma eficaz. Para prover melhor suporte à resposta da QP5, sub-questões de pesquisa foram definidas para o estudo e estão sumarizadas na Tabela 8.1.

Tabela 8.1: Subquestões de Pesquisa (QP5).

| **QP** | **Descrição** |
|---|---|
| SQP5.1 | O uso da abordagem DI promove melhoria na medida de eficácia *precision* na detecção de anomalias de código? |
| SQP5.2 | O uso da abordagem DI promove melhoria na medida de eficácia *recall* na detecção de anomalias de código? |

Do ponto de vista de anomalias remanescentes no código, é importante avaliar o número de anomalias detectadas mediante suporte das abordagens, compreendendo quais representam de fato problemas de manutenibilidade (i.e., verdadeiros positivos), bem como as que foram apontadas como "anomalias" e, contudo, não representam problemas de manutenibilidade (i.e., falsos positivos). Essa avaliação é importante pois permitirá que sejam priorizadas as correções das anomalias que realmente impactam na qualidade e facilidade de manutenção do código, evitando o desperdício de tempo e recursos com a correção de falsos positivos.

É importante mencionar que a**QP5.1** e **QP5.2** foram definidas seguindo a mesma motivação do estudo descrito Capítulo 7. Similarmente, para cada subquestão descrita anteriormente na Tabela 8.1, foram definidas Hipóteses (H) que estão resumidas na Tabela 8.2.

Tabela 8.2: Hipóteses (H).

| **H** | **Descrição** |
|---|---|
| H5.1 | A abordagem DI tem uma melhor medida de *precision* se comparada à abordagem DNI na atividade de detecção de anomalias. |
| H5.2 | A abordagem DI tem uma melhor medida de *recall* se comparada à abordagem DNI na atividade de detecção de anomalias. |

As Hipóteses **H5.1** e **H5.2** foram definidas em virtude de evidências empíricas encontradas no estudo descrito no Capítulo 7. Notou-se que a abordagem DI atingiu melhores resultados na detecção de anomalias em termos dessas medidas de eficácia no contexto da análise e inspeção de código. Deduz-se assim que o uso da abordagem DI também possa melhorar tais medidas de eficácia na detecção de anomalias no contexto do desenvolvimento de código. Desse modo, a Hipótese **H5.1** é confirmada se os valores associados à medida *precision* mediante utilização da abordagem DI forem superiores aos valores da medida *precision* mediante utilização da abordagem DNI. Por analogia, a Hipótese **H5.2** é confirmada se os resultados associados à medida *recall* através do uso da abordagem DI forem superiores aos resultados obtidos através do uso da abordagem DNI.

Essas medidas são relevantes para determinar a quantidade de anomalias remanescentes no código e, portanto, para avaliar a eficácia geral da abordagem de detecção de anomalias. Quando as medidas de *precision* e *recall* são elevadas, significa que a abordagem utilizada é eficaz em detectar a maioria das anomalias e que, consequentemente, haverá menos anomalias remanescentes no código. Por outro lado, quando as medidas são baixas, pode haver um grande número de anomalias remanescentes que precisam ser corrigidas para garantir a manutenibilidade do código.

## 8.2 Planejamento, Seleção de Participantes e Avaliação dos Resultados

Este experimento controlado foi planejado seguindo as orientações fornecidas por Wohlin *et al.*[125] e Ko *et al.*[52]. Os participantes foram conduzidos por tarefas relacionadas à detecção de anomalias de código no contexto do desenvolvimento de software. Todas as tarefas foram executadas usando as abordagens DI e DNI, incorporadas na ferramenta *Eclipse ConCAD* (Apêndice D). A comparação entre estas abordagens possibilita a conclusão de que as características distintivas da DI, como detecção antecipada e interação direta com trechos de código“anômalos”, apresentam (des)vantagens evidentes na atividade de detecção de anomalias de código.

No que diz respeito aos participantes deste estudo, uma amostra de 16 indivíduos foi recrutada, abrangendo estudantes de graduação e desenvolvedores profissionais. É relevante

observar que esses participantes diferiam daqueles envolvidos no estudo delineado no Capítulo 7. Todos os participantes foram contatados por meio de convites de *e-mail*, demonstrando interesse voluntário em participar do experimento. Embora fosse necessário que os participantes possuíssem ao menos um conhecimento moderado da linguagem Java e *Spring Framework*, não era uma expectativa que tivessem um domínio abrangente sobre anomalias de código, refatoração ou técnicas de detecção utilizadas no experimento. Entre os participantes estavam estudantes do curso de Engenharia de Computação, matriculados em uma disciplina de programação para a *Web* no Instituto Federal da Paraíba (IFPB). Além disso, também contamos com desenvolvedores profissionais do Núcleo de Pesquisa, Desenvolvimento e Inovação da Universidade Federal de Campina Grande (VIRTUS/UFCG), atuando como desenvolvedores em suas respectivas funções.

Foi empregada uma análise estatística nos resultados das medidas de eficácia obtidas nas atividades experimentais mediante suporte da ferramenta R [53]. O teste *Wilcoxon signed-rank test* [55] foi empregado nos valores associados aos componentes utilizados para calcular as medidas de eficácia (i.e., FN, VP e FP). Posteriormente, o teste *Paired T-Test* [55] foi aplicado às métricas de *recall* e *precision*. Mais detalhes sobre a motivação do uso desses testes podem ser obtidos no estudo descrito Capítulo 7.

## 8.3 Execução do Experimento

Os participantes realizaram as tarefas experimentais associadas ao desenvolvimento de um projeto de software utilizando a linguagem Java. No contexto desse estudo, os participantes deveriam realizar o desenvolvimento de um sistema *web* para simular um domínio de livraria. Este sistema foi escolhido devido a sua simplicidade bem como a capacidade de ser desenvolvido em um período de tempo adequado à realização do experimento. No que segue, são descritas as principais funcionalidades do sistema:

1. permitir cadastro de livros;
2. permitir alteração de livros cadastrados;
3. permitir exclusão de livros cadastrados;
4. permitir a filtragem dos livros cadastrados de acordo com critérios;
5. ter integração com banco de dados;

6. permitir cadastro de usuários;
7. permitir alteração de dados dos usuários;
8. possuir sistema de autenticação;
9. permitir que usuários cadastrados possam editar seus dados de cadastro.

Visando minimizar qualquer viés, este sistema foi desenvolvido com uso da linguagem Java mediante suporte do *Spring framework* para guiar as atividades de desenvolvimento. Este *framework* é bastante popular na comunidade de desenvolvedores Java, existindo diversas razões para sua escolha: possuir injeção de dependência; prover abstração para várias tecnologias (e.g., ORM e JDBC); ser aderente ao padrão arquitetural MVC (do inglês, *Model-View-Controller*); possuir um vasto ecossistema de bibliotecas e módulos que podem ser facilmente integrados; e fornecer mecanismos para construção de sistemas seguros.

Para a realização das atividades experimentais, os participantes foram divididos em dois grupos: controle e experimental. Os integrantes do primeiro grupo irão realizar as atividades mediante suporte da abordagem DNI. Os integrantes do segundo grupo irão realizar as atividades mediante suporte da abordagem DI. Os grupos foram balanceados em termos de tempo de experiência e formação para evitar qualquer tipo de viés nos resultados obtidos a partir das atividades experimentais. Assim, cada grupo deverá ter quatro desenvolvedores e quatro estudantes que foram escolhidos aleatoriamente para realização das atividades.

Para avaliação dos resultados associados às atividades experimentais foi necessário obter um "oráculo" representando a lista de anomalias de código que realmente representam problemas de manutenibilidade no sistema. Para a geração do oráculo foram realizadas os mesmos procedimentos descritos no Capítulo 7. O ambiente de execução das tarefas experimentais, incluindo os arquivos e suporte ferramental, foi disponibilizado aos participantes. Além disso, as tarefas experimentais foram supervisionadas por dois pesquisadores. O tempo total de realização do experimento foi de 30 dias, sendo realizadas duas entregas durante esse período (i.e., uma entrega ao final do 15º dia e outra entrega ao final do 30º dia). Para cada entrega foi elencado um número de requisitos e funcionalidades previstas. Em linhas gerais, o experimento foi organizado em três fases da seguinte forma, conforme descrito na Figura 8.1.

**Fase 1: Pré-experimento**. Inicialmente, todos os participantes receberam material com a definição de 10 tipos de anomalias suportadas pela ferramenta *Eclipse ConCAD* (i.e., *Brain*

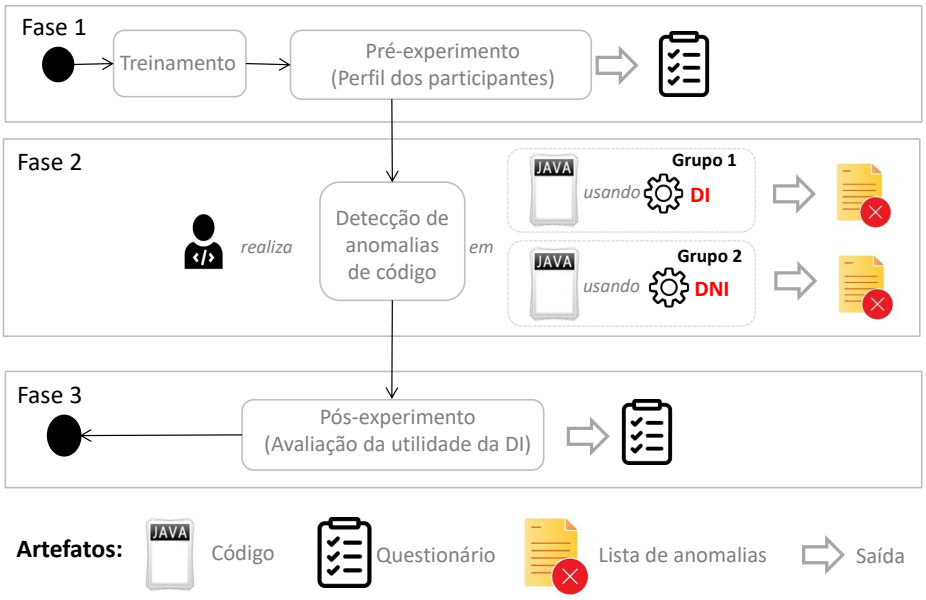


Figura 8.1: Etapas de Execução do Experimento.

*Class, Brain Method, Feature Envy, Data Class, Dispersed Coupling, God Class, Intensive Coupling, Refused Bequest, Tradition Breaker* e *Shotgun Surgery*) e exemplos da ocorrência de cada uma delas. Um prazo de 20 minutos (máximo) foi concedido para que os participantes entendessem essas definições. Em seguida, os participantes passaram por um treinamento sobre as abordagens de DI e DNI contidas na ferramenta *Eclipse ConCAD*. Finalmente, todos os participantes foram convidados a responder um questionário com objetivo principal de identificar o perfil dos participantes;

**Fase 2: Desenvolvimento de Software e Detecção de Anomalias**. Os participantes realizaram o desenvolvimento de um sistema *web* com uso da linguagem Java. Os participantes tinham o objetivo de identificar e catalogar a presença de 10 tipos distintos de anomalias de código ao longo da realização do experimento. Para tanto, os participantes utilizaram a ferramenta *Eclipse ConCAD* para auxiliar na detecção das anomalias de código durante as atividades de desenvolvimento do sistema. Cabe comentário de que durante o experimento, os participantes poderiam concordar ou discordar da detecção proposta por cada abordagem. Assim, "Falsos Positivos" decorrentes das abordagens poderiam ser omitidos quando os participantes utilizassem seu conhecimento para tomar decisões sobre as instâncias de anomalias de código existentes. As estratégias de detecção, bem como os valores limiares das métricas utilizadas, foram configurados da mesma forma para todos os participantes. Tais valores

foram baseados em estratégias de detecção relevantes descritas em estudos prévios [69][79].

Os dados relacionados à detecção de anomalias de cada tarefa foram transcritos para uma lista contendo dados relacionados ao número total (T) de anomalias de código detectado (AD), verdadeiros positivos (VP), falsos positivos (FP) e falsos negativos (FN). Os resultados das tarefas transcritos nestas listas foram usados para calcular as medidas de *recall* e *precision* e, consequentemente, avaliar as hipóteses (H5.1 e H5.2) do presente estudo. Finalmente, para evitar que o efeito de aprendizagem promovesse qualquer viés, as análises das listas com os resultados da detecção foram divulgadas apenas após a realização do experimento.

**Fase 3: Pós-experimento**. Nesta etapa, os participantes foram questionados sobre alguns aspectos do uso da abordagem DI utilizada nas atividades experimentais. Para tanto, os participantes foram submetidos a um questionário contendo diversas questões abertas e fechadas. Na análise dos dados, considerou-se a concordância (C) ou Discordância (D) - e suas diferentes formas de classificação, como Totalmente (T) e Parcialmente (P) - para algumas afirmações relacionadas à utilidade da Detecção Interativa.

## 8.4 Resultados

Nesta seção são apresentados os dados relacionados ao perfil dos participantes, bem como os resultados relacionados às tarefas experimentais. Em seguida, são discutidos alguns aspectos relacionados ao uso das abordagens DI e DNI para detecção de anomalias de código no contexto das atividades de desenvolvimento previstas no experimento.

### 8.4.1 Atividades Pré-Experimento (Fase 1)

A primeira fase do experimento envolveu a realização de um treinamento para garantir que os participantes obtivessem as habilidades e informações necessárias para realizar o experimento com sucesso e produzir resultados precisos e confiáveis. Essa atividade foi realizada ao longo de dois encontros com duração aproximada de 90 minutos cada. O primeiro encontro objetivou nivelar o conhecimento sobre anomalias de código e refatoração enquanto que o segundo objetivou o nivelamento em termos do uso do ambiente de desenvolvimento integrado bem como as abordagens de detecção. Cada encontro foi composto por uma fase

expositiva e uma fase prática. Ao final de cada encontro, os participantes foram submetidos a uma avaliação para assegurar a compreensão dos conceitos e técnicas necessárias à realização das atividades experimentais. Todos os participantes concluíram de modo satisfatório as atividades previstas para o treinamento.

Em seguida, todos os participantes foram convidados a responder um questionário para determinar o perfil. Em média, cada participante demorou 10 minutos para proceder com as respostas do questionário. Maior parte dos respondentes (cerca de 55%) possuía entre cinco e oito anos de experiência em atividades de desenvolvimento de software. O questionário utilizou uma escala de 0 a 4, onde 0 significa "não proficiente" e 4 "muito proficiente". Assim, relacionado à proficiência em linguagem Java, 50% dos participantes responderam que tinham proficiência em níveis 3 ou 4. Similarmente, em relação ao uso do ambiente de desenvolvimento integrado Eclipse, cerca de 75% dos participantes responderam que tinham proficiência em níveis 3 ou 4. Com relação à detecção de anomalias de código, cerca de 50% dos respondentes citaram que conheciam/realizavam essa atividade. No geral, o perfil dos participantes atende às expectativas do estudo uma vez que eles possuem conhecimento intermediário em Java, conhecem o ambiente de desenvolvimento integrado utilizado no experimento além das atividades de detecção de anomalias de código e refatoração fazerem parte do seu vocabulário ou rotina de trabalho.

### 8.4.2 Desenvolvimento de Software e Detecção de Anomalias (Fase 2)

A segunda fase do experimento controlado envolveu as tarefas relacionadas ao desenvolvimento de software usando abordagens DI e DNI mediante suporte da ferramenta *Eclipse ConCAD* com vistas a analisar a quantidade de anomalias de código remanescentes nas entregas previstas pelas atividades experimentais. Os participantes deveriam realizar duas entregas (i.e., uma entrega ao final do 15º dia e outra entrega ao final do 30º dia do experimento). Em cada uma dessas entregas, o participante deveria proceder a entrega de dois artefatos: (i) o projeto na linguagem Java com a implementação dos requisitos e funcionalidades previstos; e (ii) uma lista contendo 10 tipos distintos de anomalias de código remanescentes no código consideradas pelos participantes como problemas de manutenibilidade. Adicionalmente, foi gerado para cada projeto entregue um oráculo contendo uma lista de anomalias de código que realmente representam problemas de manutenibilidade. Mediante comparação

entre esses artefatos (i.e., oráculo *vs*. lista de anomalias) foi possível obter números associados a Verdadeiros Positivos (VP); Falsos Positivos (FP); e Falsos Negativos (FN) para cada uma das análises. Os resultados associados a cada um desses componentes encontram-se descritos na Tabela 8.3:

Tabela 8.3: Resultados da Detecção de Anomalias.

| | **1a análise** | | | **2a análise** | | |
|---|---|---|---|---|---|---|
| | **DI (Grupo Experimental)** | | | | | |
| | **VP** | **FP** | **FN** | **VP** | **FP** | **FN** |
| Des 1 | 17 | 5 | 7 | 15 | 4 | 6 |
| Des 2 | 15 | 8 | 8 | 14 | 6 | 8 |
| Des 3 | 20 | 7 | 9 | 19 | 5 | 6 |
| Des 4 | 13 | 5 | 6 | 13 | 4 | 5 |
| **soma des.** | **65** | **25** | **30** | **61** | **19** | **25** |
| **média des.** | **16,2** | **6,2** | **7,5** | **15,2** | **4,7** | **6,2** |
| Est 1 | 13 | 6 | 9 | 12 | 5 | 9 |
| Est 2 | 12 | 10 | 12 | 10 | 7 | 8 |
| Est 3 | 14 | 9 | 11 | 14 | 7 | 8 |
| Est 4 | 12 | 7 | 8 | 12 | 5 | 9 |
| **soma est.** | **51** | **32** | **40** | **48** | **24** | **34** |
| **média est.** | **12,7** | **8** | **10** | **12** | **6** | **8,5** |
| **soma total** | **116** | **57** | **70** | **109** | **43** | **59** |
| **média total** | **14,5** | **7,1** | **8,7** | **13,6** | **5,4** | **7,4** |
| | **DNI (Grupo Controle)** | | | | | |
| Des 5 | 23 | 7 | 9 | 18 | 6 | 8 |
| Des 6 | 27 | 8 | 11 | 21 | 8 | 10 |
| Des 7 | 22 | 8 | 13 | 17 | 7 | 12 |
| Des 8 | 19 | 9 | 8 | 16 | 8 | 7 |
| **soma des.** | **91** | **32** | **41** | **72** | **29** | **37** |
| **média des.** | **22,7** | **8** | **10,2** | **18** | **7,2** | **9,2** |
| Est 5 | 19 | 12 | 11 | 17 | 12 | 9 |
| Est 6 | 23 | 11 | 13 | 21 | 9 | 11 |
| Est 7 | 18 | 11 | 9 | 16 | 8 | 8 |
| Est 8 | 16 | 13 | 15 | 13 | 12 | 12 |
| **soma est.** | **76** | **47** | **48** | **67** | **41** | **40** |
| **média est.** | **19** | **11,7** | **12** | **16,7** | **10,2** | **10** |
| **soma** | **167** | **79** | **89** | **139** | **70** | **77** |
| **média** | **20,9** | **9,9** | **11,1** | **17,4** | **8,7** | **9,6** |

**Abordagem DI diminui AD e VP**

Os resultados experimentais mostraram que os participantes que utilizaram DNI obtiveram um total de 246 (1ª análise) e 209 (2ª análise), enquanto que com a abordagem DI obtiveram um total de 173 (1ª análise) e 142 (2ª análise). Conclui-se que o uso da abordagem DI leva os participantes a diminuir o número de AD em mais de 30% em comparação com a utilização da DNI. Uma explicação para isso é que o *feedback* constante, bem como as informações contextuais provenientes da abordagem DI, podem ter contribuído diretamente na produção de um código com menos anomalias remanescentes em cada entrega.

Outra observação interessante é que o número de AD na segunda entrega diminuiu consideravelmente em relação à primeira entrega. Isso pode ter ocorrido em virtude de que os desenvolvedores podem ter adquirido mais experiência e conhecimento sobre as melhores práticas de desenvolvimento de software, bem como sobre as restrições e especificações do sistema em desenvolvimento. Assim, os desenvolvedores podem ter adquirido uma postura mais consciente e cuidadosa na criação do código, com maior atenção à qualidade e manutenibilidade do software. Isso foi notado principalmente nos desenvolvedores que utilizaram a abordagem DI que conseguiram reduzir em cerca de 35% o número de anomalias remanescentes na 2ª entrega em comparação com o número de anomalias remanescentes na 1ª entrega.

Em relação aos resultados associados ao número de VPs, notou-se decremento similar. Participantes usando a abordagem DI detectaram 116 (1ª análise) e 109 (2ª análise), enquanto que usando DNI detectaram 167 (1ª análise) e 139 (2ª análise). Portanto, o uso de DI contribui para redução de mais 30% no número total de VP (i.e., anomalias de código remanescentes) na atividade de detecção de anomalias de código. Ao diminuir o número de VP, os desenvolvedores podem melhorar a qualidade do código-fonte e torná-lo mais fácil de manter e evoluir ao longo do tempo. Esse resultado se deve em virtude de que o uso de DI pode levar a uma abordagem mais proativa na correção das anomalias de código. Uma vez que os desenvolvedores recebem alertas constantes e contínuos sobre a inserção das anomalias tão cedo quanto possível, a quantidade de tempo e esforço requerida para sua remoção tende a ser menor. Finalmente, os resultados estatísticos relacionados a AD (*alfa* = 0,05, p = 0,00044, W = 3, MD = 5,71, Z = -1.859) e VP (*alfa* = 0,05, p = 0,0053, W = 0, MD = 5,88, Z = -2,520) obtidos pelos participantes mediante uso das abordagens DI e DNI foram

significantes através do uso do teste de Wilcoxon [55].

**Abordagem DI diminui FP e FN**

Com relação aos resultados de FP, observou-se que os participantes identificaram 79 (1ª análise) e 70 (2ª análise) ao utilizar a abordagem DNI. Por outro lado, observou-se que os participantes identificaram 57 (1ª análise) e 43 (2ª análise) mediante uso da abordagem DI. Conclui-se, portanto, que a utilização da abordagem DI pode reduzir em até 40% o número de FP. Essa diminuição foi mais acentuada durante a utilização da abordagem DI onde os desenvolvedores conseguiram reduzir em mais de 25% o número de FP entre a 1ª e 2ª entregas. Realizando a mesma análise de desenvolvedores com uso da abordagem DNI, notou-se uma redução de menos de 12% entre a 1ª e 2ª entregas. Ainda, a diminuição de FPs pode economizar tempo e esforço dos desenvolvedores, reduzindo o número de falsos alarmes que eles precisam investigar, permitindo que eles se concentrem nas verdadeiras instâncias de anomalias de código. Esse resultado pode ter ocorrido em virtude das informações contextuais providas pela abordagem DI que podem ter tornado os participantes do experimento mais confiantes na conclusão a respeito da existência (ou não) de uma anomalia de código no sistema em desenvolvimento.

Em relação aos resultados associados aos FNs, observou-se que os participantes identificaram 89 (1ª análise) e 77 (2ª análise) com suporte da abordagem DNI, enquanto os participantes com DI conseguiram 72 (1ª análise) e 60 (2ª análise). Isso denota que o uso da abordagem DI pode reduzir o número de FN em mais de 20%. É importante diminuir os FNs na detecção porque um FN significa que uma anomalia está presente no código, mas a abordagem não a identificou. Isso pode fazer com que essa instância de anomalia permaneça no código por um longo período de tempo e potencialmente causando problemas como diminuição da capacidade de manutenção do código ou aumento da probabilidade de *bugs*. Em outras palavras, a diminuição de FNs ajuda a garantir que todas as anomalias sejam identificadas e resolvidas em tempo hábil, melhorando a qualidade geral e a capacidade de manutenção do código. É importante mencionar que os resultados estatísticos relacionados a FP (*alfa* = 0,05, p = 0,00032, W = 0, MD = -1,87, Z = -3,407) e FN (*alfa* = 0,05, p = 0,00175, W = 6, MD = -3,14, Z = -2,919) obtidos a partir do uso das abordagens DI e DNI foram significantes mediante uso do teste de Wilcoxon [55].

Similarmente aos resultados obtidos no experimento do Capítulo 7, notou-se um efeito do nível de experiência dos participantes nos resultados do estudo. Em relação aos resultados associados a AD, o uso da abordagem DI conduziu os estudantes a diminuir AD em cerca de 15% em comparação com o uso da abordagem DNI, enquanto considerando os desenvolvedores esse decremento foi de cerca de 8%. Da mesma forma, nos resultados associados a TP, a abordagem DI levou os estudantes a diminuir o TP em cerca de 20%, enquanto considerando os desenvolvedores, esse aumento foi de cerca de aproximadamente 10%. Com relação a FP, o uso da abordagem DI contribuiu para diminuição desse indicador em 10% para os desenvolvedores e de cerca de 15% considerando a amostra de estudantes. Por fim, nos resultados associados a FN, a abordagem DI levou os estudantes a diminuir este indicador em cerca de 25% e desenvolvedores a diminuir em cerca de 10%. As análises acima foram restritas apenas aos resultados da segunda análise.

### Avaliação das medidas de eficácia

Para fornecer uma perspectiva adicional sobre a eficácia das abordagens DI e DNI na atividade de detecção de anomalias de código, foram calculadas as medidas de eficácia *Precision* (P) e *Recall* (R) a partir do uso dos componentes descritos na Tabela 8.3. Os resultados associados a essas medidas podem ser vistos na Tabela 8.4.

Tabela 8.4: Resultados das Medidas de Eficácia na Detecção de Anomalias.

| | **DI (Grupo Experimental)** | | | | | **DNI (Grupo Controle)** | | | |
|---|---|---|---|---|---|---|---|---|---|
| | **1a análise** | | **2a análise** | | | **1a análise** | | **2a análise** | |
| | **P** | **R** | **P** | **R** | | **P** | **R** | **P** | **R** |
| Des 1 | 0,82 | 0,77 | 0,82 | 0,75 | Des 5 | 0,71 | 0,65 | 0,71 | 0,65 |
| Des 2 | 0,77 | 0,73 | 0,78 | 0,72 | Des 6 | 0,65 | 0,58 | 0,64 | 0,58 |
| Des 3 | 0,76 | 0,71 | 0,77 | 0,71 | Des 7 | 0,71 | 0,61 | 0,73 | 0,61 |
| Des 4 | 0,79 | 0,76 | 0,80 | 0,76 | Des 8 | 0,59 | 0,62 | 0,62 | 0,65 |
| **média Des** | **0,78** | **0,74** | **0,79** | **0,73** | **média Des** | **0,67** | **0,61** | **0,68** | **0,62** |
| Est 1 | 0,76 | 0,68 | 0,77 | 0,65 | Est 5 | 0,52 | 0,54 | 0,50 | 0,57 |
| Est 2 | 0,70 | 0,66 | 0,75 | 0,72 | Est 6 | 0,52 | 0,48 | 0,53 | 0,48 |
| Est 3 | 0,67 | 0,62 | 0,70 | 0,67 | Est 7 | 0,56 | 0,61 | 0,64 | 0,64 |
| Est 4 | 0,70 | 0,67 | 0,72 | 0,59 | Est 8 | 0,48 | 0,44 | 0,50 | 0,50 |
| **média Est** | **0,70** | **0,66** | **0,74** | **0,66** | **média Est** | **0,52** | **0,52** | **0,54** | **0,55** |
| **média Total** | **0,75** | **0,70** | **0,76** | **0,70** | **média Total** | **0,59** | **0,57** | **0,61** | **0,59** |

**Abordagem DI aumenta a *precision***. O valor médio desta medida com uso da abordagem DI foi de 0,75 (1ª análise) e 0,76 (2ª análise), enquanto com uso da DNI atingiu 0,59 (1ª análise) e 0,61 (2ª análise). A diferença nos resultados da medida *precision* foi em média de até 25% em favor do uso da abordagem DI. Os resultados da medida *precision* alcançada com ambas as técnicas foram estatisticamente significantes (*alfa* = 0,05, p = 0,005103, SD = 9,2953, t = 4,6974, tamanho do efeito = 0,83, mediante uso do *Paired T-Test* [55]). Portanto, *é possível confirmar a primeira hipótese (H5.1).* É importante notar que a abordagem DI promoveu uma elevação significativa na medida *precision* considerando apenas a amostra dos estudantes, visto que eles alcançaram em média 0,53 com o uso da abordagem DNI. Ao usar DI, obteve-se um aumento de cerca de 35% (i.e, em média 0,72). Curiosamente, resultados semelhantes não podem ser vistos na amostra dos desenvolvedores, na qual observou-se um aumento de menos de 15% na mesma comparação. De modo geral, notou-se que quanto mais baixo o nível de conhecimento dos participantes (i.e., amostra de estudantes), maior o benefício que essa amostra pode ter com o uso de uma abordagem DI em termos da medida *precision*.

**Abordagem DI aumenta o *recall***. Com base nos resultados associados a FN e VP, espera-se que o uso da abordagem DI contribua significativamente para melhorar o *recall* na detecção de anomalias. Em média, observou-se que os participantes que utilizaram a DI obtiveram 0,70 (1ª análise) e 0,70 (2ª análise), enquanto que os participantes que utilizaram a DNI atingiram 0,57 (1ª análise) e 0,59 (2ª análise), representando uma diferença de até aproximadamente 25% a favor da abordagem DI. Ao analisar diferentes amostras, os desenvolvedores melhoram os valores da medida *recall* em menos de 15%, enquanto os estudantes melhoram em cerca de 40%. Finalmente, os resultados relacionados à *recall* nesta tarefa por meio das abordagens DI e DNI foram estatisticamente significantes (*alfa* = 0,05, p = 0,0002951, SD = 11,2089, t = 4,8890, tamanho do efeito = 0,86, usando *Paired T-Test* [55]).

Também observou-se que a medida *recall* sofre influência direta em relação à experiência de trabalho dos participantes. Em linhas gerais, quanto menor o nível de conhecimento dos participantes, maior o benefício que esta amostra pode ter com o uso da abordagem DI em termos de medida de *recall*. Além disso, o uso de DI pode afetar diretamente os valores desta medida, uma vez que sua utilização implica que os participantes interajam com os elementos

de código afetados durante a análise de fragmentos de código. Portanto, os desenvolvedores podem observar uma cobertura melhor com a abordagem DI ao detectar corretamente instâncias de anomalias de código. Finalmente, *é possível confirmar a segunda hipótese (H5.2)*, uma vez que o uso do DI levou a melhores resultados de *recall* em comparação com a abordagem DNI.

### 8.4.3 Atividades Pós-Experimento (Fase 3)

A terceira fase do experimento controlado envolveu responder um questionário sobre alguns aspectos do uso da abordagem DI. Resumidamente, essa análise compreendeu um conjunto de cinco questões (Q) de múltipla escolha para verificar a concordância (C) ou discordância (D) - e suas diferentes formas de classificação, como Totalmente (T) e Parcialmente (P) - para algumas afirmações relacionadas à utilidade da abordagem DI no contexto do desenvolvimento de software. Os resultados da utilidade da abordagem DI para detecção de anomalias de código são descritos na Tabela 8.5.

Tabela 8.5: Utilidade da Detecção Interativa.

| Nr | Questão | DT | DP | C | CP | CT |
|---|---|---|---|---|---|---|
| Q1 | A técnica DI usada no experimento foi útil na realização das tarefas atribuídas? | 0 | 0 | 1 | **9** | 6 |
| Q2 | A técnica DI encontrou informações que eu não teria encontrado tão rapidamente sem ela? | 0 | 0 | 1 | 3 | **12** |
| Q3 | A técnica DI encontrou informações que eu não teria encontrado sem ela? | 0 | 1 | 2 | 6 | **7** |
| Q4 | Sem a técnica DI, era difícil identificar a ocorrência de todas as dez anomalias de modo concomitante? | 0 | 0 | 0 | 3 | **13** |
| Q5 | Se uma técnica DI estiver disponível, eu a usaria ao codificar? | 0 | 0 | 0 | 6 | **10** |

No geral, obteve-se *feedback* positivo quanto ao uso da abordagem DI, considerando que cerca de 90% dos participantes concordaram (total ou parcialmente) com as afirmações relacionadas à sua utilidade para as questões Q1-Q5. Em relação a **Q1** e **Q2**, notou-se que os respondentes consideram útil a utilização da abordagem DI na detecção de anomalias.

Em relação a **Q3**, maior parte dos respondentes (85%) consideram que a abordagem DI contribuiu com afirmações relevantes e contextuais para detecção de anomalias. Em relação a **Q4**, notou-se que todos os respondentes concordam que a abordagem DI é relevante para detecção de múltiplos tipos de anomalias de forma concomitante. Por fim, na **Q5** todos os respondentes se mostraram dispostos a utilizar a abordagem DI em suas atividades de desenvolvimento, confirmando sua relevância e utilidade no contexto desta tarefa.

Em seguida, realizou-se um conjunto de cinco questões abertas, e as respostas foram rotuladas como: (i) Concordância (C), que representa a concordância do participante (total ou parcialmente) com o questionamento; (ii) Discordância (D), que representa que o participante discorda (total ou parcialmente). Os resultados da pseudo-entrevista foram resumidos na Tabela 8.6.

Tabela 8.6: Resultados da Pseudo-Entrevista.

| **Nr** | **Questão** | **C** | **D** |
|---|---|---|---|
| Q6 | Usando essa técnica DI (durante a codificação), você acredita que ela chamaria sua atenção nos momentos certos? | **12** | 4 |
| Q7 | Você achou que a quantidade e disponibilidade de informações fornecidas por essa técnica impactou negativamente sua atividade de programação? | 5 | **11** |
| Q8 | A ferramenta pode deixá-lo mais confiante sobre seus julgamentos sobre ações de refatoração? | **14** | 2 |
| Q9 | Você acha que ajudou você (por exemplo, com mais e melhores informações) a realizar tais julgamentos? | **15** | 1 |
| Q10 | Você gostaria de mudar algo neste (ou em qualquer outro) detector de anomalias? | 7 | **9** |

Em relação a **Q6** (Tabela 8.6) verificou-se que a maioria dos participantes (75%) acredita que o uso da abordagem DI chamaria a atenção nos momentos certos. Em relação a **Q7**, discutiu-se que uma vez que a abordagem DI pode ser desativada a qualquer momento, isso pode evitar que os resultados da detecção possam comprometer a atividade principal (e.g., análise de código ou codificação). Mais de 70% dos respondentes discordaram que o uso dessa abordagem poderia comprometer a atividade principal. Além disso, os resultados referentes a **Q8** e **Q9** revelaram que a maioria dos participantes considerou que o uso da técnica DI permite realizar as ações de refatoração por fornecer mais informações. Os participantes

mencionaram que a representação dos resultados da detecção (por meio de gradação de cores), auxiliava na criação de um "mapa mental" que ajuda a verificar a real propagação das anomalias de código no projeto analisado.

Finalmente, na questão **Q10**, diversas sugestões foram coletadas visando aprimorar a abordagem DI. Essas sugestões incluem algumas já obtidas no estudo descrito no Capítulo 7, tais como (i) adaptar a técnica para ser compatível com versões mais recentes do IDE *Eclipse*; (ii) ampliar o suporte para outros tipos de anomalias de código; e (iii) permitir a análise seletiva de anomalias de código escolhidas previamente, evitando a exibição simultânea de todas as anomalias. Além destas, outras sugestões interessantes foram coletadas tais como (iv) implementar funcionalidades de visualização de histórico de mudanças relacionadas às anomalias de código, permitindo aos desenvolvedores acompanhar a evolução ao longo do tempo; (v) incorporar um mecanismo de sugestões de refatoração específicas para cada tipo de anomalia detectada, auxiliando os desenvolvedores na correção dos problemas identificados; (vi) Fornecer suporte para análises em equipe, permitindo que os membros compartilhem e discutam as anomalias de código encontradas; e (vii) Oferecer uma interface personalizável que permita aos desenvolvedores configurar quais tipos de anomalias desejam destacar e receber *feedback*.

## 8.5 Ameaças à Validade

Em qualquer estudo experimental existem fatores que podem ser vistos como possíveis influências e ameaças à validade. Nesta seção discutem-se tais ameaças associadas ao estudo descrito neste capítulo, bem como as ações realizadas para mitigá-las. Para tanto, seguiu-se o esquema de classificação proposto por Wohlin *et al.* [125] conforme a seguir.

*Validade de Construção*. A primeira ameaça diz respeito à definição do que está sendo estudado e como ele será operacionalizado (medido) no estudo. Para minimizar essa ameaça, seguiu-se a definição de um protocolo que, de modo iterativo e colaborativo, foi construído com vistas a suportar as atividades experimentais. Outra ameaça diz respeito ao uso de métricas para mensurar as atividades. Isso ajuda a garantir que os resultados do estudo não sejam em decorrência de erros de medição ou mesmo algum tipo de viés. No contexto desse estudo foram utilizadas medidas relevantes e amplamente utilizadas em estudos relacionados. Fi-

nalmente, a ameaça do uso de grupos de controle e experimental de forma apropriada. Para garantir a inexistência de viés, optou-se por uma divisão balanceada dos grupos em termos de números, bem como no nível de experiência e formação destes participantes.

*Validade Externa*. Para minimizar essa ameaça buscou-se usar configurações realistas para o estudo. Isso ajuda a garantir que os resultados possam ser generalizados para outras configurações onde a variável independente e a variável dependente podem ser encontradas. Outra ameaça diz respeito ao uso apropriado de análises estatísticas. Buscou-se utilizar testes estatísticos aos dados obtidos nas tarefas experimentais. Esses testes foram escolhidos com base nos critérios da natureza dos dados, número de grupos e por serem amplamente utilizados em experimentos relacionados a esta pesquisa.

*Validade Interna*. A seleção de anomalias de código usadas no experimento controlado pode ser considerada uma ameaça. Restringiu-se à discussão de apenas dez tipos de anomalias de código. Em contraste, Fowler catalogou uma lista com mais de 20 anomalias de código [38]. Portanto, as anomalias de código usadas nas tarefas experimentais podem não ser necessariamente uma amostra representativa das anomalias encontradas em sistemas de modo geral. Outra ameaça diz respeito ao uso da atribuição aleatória. Buscou-se utilizar a atribuição aleatória de participantes para os grupos experimental e de controle. Todavia, participantes com o mesmo nível de experiência aparente podem ter habilidades distintas. Considerando também o pequeno número de participantes, pessoas com habilidades excepcionais em detecção de anomalias de código podem influenciar os resultados.

*Validade da Conclusão*. A primeira ameaça está relacionada ao cálculo de medidas de eficácia (por exemplo, *recall* e *precision*). Os cálculos foram realizados com suporte automatizado da ferramenta R e revisados por dois pesquisadores. Outra ameaça está relacionada ao tamanho da amostra. 16 participantes entre estudantes e profissionais realizaram o experimento controlado. Os resultados podem sofrer influência direta do tamanho da amostra e do nível de experiência de trabalho dos participantes na detecção e refatoração de anomalias. Para mitigar essa ameaça, escolheu-se uma amostra equilibrada composta por estudantes e desenvolvedores, e foram conduzidas sessões de treinamento para nivelamento de conhecimento dos assuntos relativos a esses tópicos. Por fim, em relação às tarefas, foi evitado qualquer viés em benefício da abordagem DI, garantindo que todos os participantes tenham recebido o mesmo conjunto de informações necessárias para a resolução das tarefas, as quais

foram disponibilizadas somente durante o uso da IDE Eclipse. No entanto, uma possível ameaça à validade interna é o fato de que o grupo experimental é capaz de ver constantemente os alertas associados à detecção de anomalias, o que pode permitir que eles removam tais problemas com muito mais facilidade, pois podem supor quais melhorias são requeridas na remoção.

*Generalização dos resultados*. Realizou-se apenas um único experimento que envolveu um número limitado de participantes e um único sistema-alvo. Certamente se faz necessária a realização de mais estudos antes que os resultados possam ser generalizados. No entanto, definiu-se uma configuração experimental que é bastante comum em empresas de software: desenvolvedores com um nível médio de experiência, lidando com uma base de código desconhecida - uma situação que os desenvolvedores frequentemente encontram quando mudam de projeto - e sendo confrontados com tipos de problemas de código (correlacionados) que a ferramenta *Eclipse ConCAD* está detectando atualmente. No entanto, existem vários aspectos que podem influenciar a generalidade dos resultados: (i) é possível que trabalhar em uma base de código familiar reduzisse a lacuna entre as conquistas dos dois grupos; (ii) as especificidades do sistema (e.g. área de negócio, domínio, tamanho do projeto entre outros) podem levar a uma variação dos resultados; e (iii) a utilização de desenvolvedores com vários níveis de experiência pode levar a resultados diferentes. Por exemplo, pessoas que são muito habilidosas no uso da IDE *Eclipse* podem ter desempenho melhor do que o grupo de controle, enquanto no caso dos desenvolvedores menos experientes a diferença pode ser ainda maior (a favor do grupo experimental).

## 8.6 Considerações Finais do Capítulo

Este estudo apresentou uma avaliação empírica quanto à eficácia da detecção de anomalias de código com o uso das abordagens DI e DNI no contexto do desenvolvimento de software. Avaliou-se empiricamente a detecção de anomalias de código com 16 participantes (incluindo desenvolvedores profissionais e estudantes de graduação) usando as abordagens DI e DNI incorporadas à ferramenta *Eclipse ConCAD*. Os resultados mostraram que, usando DI, os desenvolvedores podem diminuir até 30% das ocorrências de anomalias remanescentes em comparação com o uso da técnica DNI durante as atividades de desenvolvimento de

software. Portanto, a técnica de DI permite que os desenvolvedores identifiquem esses problemas de modo precoce, o que pode beneficiar a manutenção do sistema. Além disso, a abordagem DI reduziu em até 40% no FP. Diminuir esse indicador pode economizar tempo e esforço dos desenvolvedores, reduzindo os alarmes falsos que eles precisam investigar, permitindo que eles se concentrem em instâncias reais de anomalias de código. Finalmente, usar DI pode reduzir o número de FN em até 20%. A diminuição deste indicador ajuda a garantir que todas as anomalias sejam identificadas e resolvidas em tempo hábil, melhorando a qualidade geral e a capacidade de manutenção do código.

Outra vertente de análise diz respeito às medidas de eficácia obtidas a partir dos indicadores descritos anteriormente. A diferença nos resultados da medida *precision* foi em média de até 25% em favor do uso da abordagem DI. Essa diferença pode ser ainda maior se considerada apenas a amostra de estudantes (cerca de 35%). Com relação à medida *recall* obteve-se uma diferença global de aproximadamente 25% a favor da abordagem DI. Utilizando apenas a amostra de estudantes esse valor é incrementado para 40%. Esses resultados podem indicar que o nível de conhecimento dos participantes no experimento pode ter influenciado os resultados. Quanto mais baixo for o nível de conhecimento dos participantes (i.e., amostra de estudantes), maior será o benefício que essa amostra pode ter com o uso de uma abordagem DI em termos da medida.

Ademais, os resultados do experimento controlado descritos no presente trabalho deram suporte para responder a QP5, conforme quadro abaixo:

**QP5 - A aplicação da abordagem de detecção interativa contribui para reduzir a quantidade de anomalias de código remanescentes durante o desenvolvimento do código?**
A utilização da abordagem DI resulta em uma redução de mais de 30% na quantidade de anomalias remanescentes em trechos de código recém desenvolvidos. Ao diminuir o número de anomalias remanescentes, os desenvolvedores podem melhorar a qualidade do código-fonte e torná-lo mais fácil de manter e evoluir ao longo do tempo.

A abordagem DI mostrou-se adequada ao desenvolvimento de código porque permite identificar e corrigir problemas rapidamente. Ao integrar ferramentas de detecção de anomalias no fluxo de trabalho de desenvolvimento de software, tais problemas podem ser identificados e corrigidos no início do processo, o que pode ajudar a garantir que o código atenda

aos padrões e boas práticas de programação, resultando em código mais confiável e fácil de manter ao longo do tempo. Além disso, o uso da abordagem DI também pode fornecer *feedback* aos desenvolvedores, ajudando-os a melhorar suas habilidades e a desenvolver código de alta qualidade.

É importante mencionar que não existe na literatura a descrição de um conjunto específico de diretrizes para realizar a detecção de anomalias de código em um processo de desenvolvimento ágil (e.g., no *Scrum*). No entanto, em geral, é importante ter um processo para identificar e lidar com anomalias e incorporar a sua detecção em um processo adequado e exequível. Essas diretrizes podem garantir que anomalias sejam detectadas e removidas desde o início, antes que tenham a chance de causar problemas mais significativos no futuro, além de melhorar a comunicação e a colaboração entre os membros da equipe e garantir que a base de código seja sustentável e escalável.

No Capítulo 9 será apresentada uma proposta de integração da abordagem DI ao processo ágil de desenvolvimento de software. Esta integração pode trazer vários benefícios, incluindo a detecção antecipada de anomalias de código, redução do retrabalho, melhoria da qualidade do código e maior eficiência do processo de desenvolvimento. Além disso, a proposta de integração da abordagem DI ao processo ágil de desenvolvimento de software pode ser adaptada a diferentes metodologias e processos de desenvolvimento de software, permitindo a sua aplicação em uma ampla variedade de projetos.

# Capítulo 9

# Aplicação da Abordagem DI no Processo Ágil de Desenvolvimento de Software

Considerando os resultados experimentais da aplicação da abordagem DI no contexto da análise de código (Capítulo 7) e do desenvolvimento de software (Capítulo 8), concluiu-se que DI pode ser promissora no contexto ágil de desenvolvimento. A análise comparativa entre DI e DNI demonstrou melhorias na eficácia da detecção de anomalias e na qualidade do código. Além disso, a capacidade de interação humana da DI se alinha bem com a natureza iterativa e adaptativa do desenvolvimento ágil. Esses resultados sugerem que a DI não só pode identificar problemas de forma mais precisa e oportuna, mas também se integra de maneira harmoniosa com as práticas ágeis, contribuindo para a melhoria contínua do software ao longo do ciclo de desenvolvimento. Portanto, a adoção da abordagem DI pode ser considerada uma estratégia promissora para otimizar o processo de desenvolvimento em um ambiente ágil.

Ao integrar essas abordagens de maneira complementar, a equipe obtém uma série de benefícios. Primeiramente, garante-se uma detecção abrangente de anomalias. Enquanto a DI lida com questões complexas e específicas, a DNI identifica problemas mais comuns, criando uma visão completa das questões presentes no código. Em segundo lugar, a complementaridade aumenta a eficiência e a agilidade do desenvolvimento. A DNI acelera varreduras automáticas, permitindo que a equipe foque nas detecções interativas mais intricadas, mantendo o processo de desenvolvimento ágil e eficaz. Além disso, a combinação das abordagens resulta em uma maior qualidade e precisão nas detecções. A expertise humana e a

compreensão contextual da DI evitam falsos positivos e enriquecem a detecção de anomalias. Por fim, a estratégia de uso complementar também fomenta o aprendizado contínuo. As correções realizadas em ambos os contextos alimentam o desenvolvimento das habilidades da equipe e aprimoram o padrão de qualidade do código.

O presente capítulo apresenta uma proposta de abordagem para desenvolvimento de software que integra a abordagem DI numa instanciação do arcabouço *Scrum*. É importante mencionar que os resultados desse estudo foram publicados no *International Conference on Software, Telecommunications and Computer Networks* (SoftCOM'23) [10] e no *Applied Sciences* [13]. Inicialmente são descritos os passos metodológicos deste estudo (Seção 9.1). Em seguida é apresentada uma visão geral da abordagem (Seção 9.2) e a adaptação dos componentes do *Scrum* (Seção 9.3). Finalmente, são descritos os resultados de duas validações (Seções 9.4 e 9.5), bem como as considerações finais do estudo deste capítulo (Seção 9.6).

## 9.1 Configuração do Estudo

A realização deste estudo visou fornecer suporte à resposta da seguinte Questão de Pesquisa (QP):

- **QP6 - Como a abordagem DI pode ser integrada ao fluxo de trabalho do processo ágil de desenvolvimento?**

Decidiu-se adotar o arcabouço *Scrum* para integrar a abordagem de Detecção Interativa (DI) ao processo de desenvolvimento de software. Essa escolha foi motivada pela capacidade do *Scrum* de lidar com mudanças frequentes nos requisitos do projeto, o que pode resultar na introdução recorrente de anomalias no código. A aplicação da abordagem DI nesse contexto pode contribuir para a detecção e remoção precoce dessas anomalias, o que, por sua vez, pode levar à melhoria dos atributos de qualidade do software em desenvolvimento.

As etapas associadas à construção da abordagem de DI integrada ao processo de desenvolvimento baseado no *Scrum* são ilustradas na Figura 9.1.

As cinco etapas foram realizadas de maneira sequencial de modo que as saídas de uma etapa são utilizadas como entradas para a etapa subsequente. A Etapa 1 consistiu na análise da literatura com intuito de elaborar uma proposta inicial do *Scrum* com atividades associadas à detecção de anomalias. Para conclusão desta etapa, diversos estudos na literatura

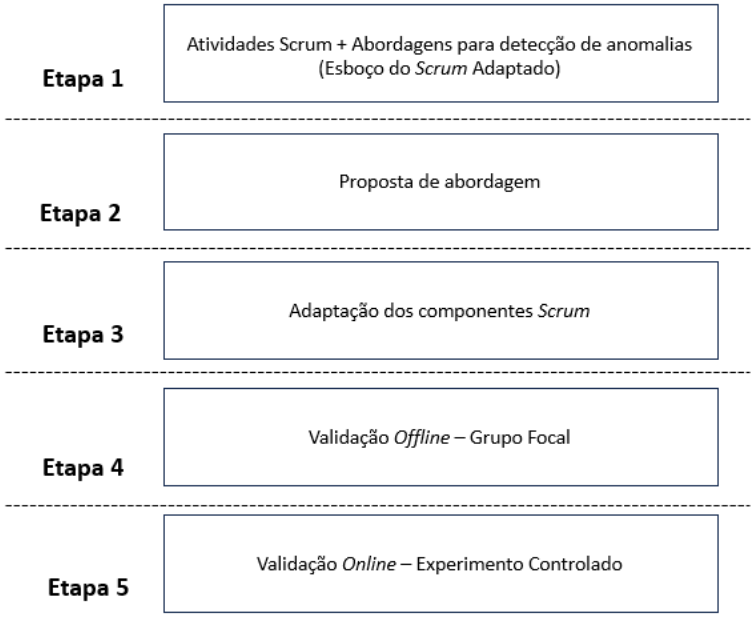


Figura 9.1: Etapas de Preparação da Abordagem Baseada no Arcabouço do *Scrum*.

científica (e.g. artigos e livros) bem como na literatura cinza (e.g., relatórios técnicos e opiniões de especialistas em *websites*) foram identificados. Estes estudos foram posteriormente analisados objetivando compreender as características do *Scrum*, bem como das atividades das abordagens de detecção interativa e não interativa (ver Seções 2.3 e 2.5). Ao final dessa etapa, com base no conhecimento obtido a respeito do *Scrum* e das abordagens para detecção, conclui-se que as atividades do *Scrum* podem ser adaptadas a partir da inclusão de um conjunto de atividades disciplinadas associadas às abordagens de detecção de anomalias discutidas no presente estudo. As demais etapas deste estudo (Etapas 2, 3, 4 e 5 da Figura 9.2) associadas à proposta da abordagem, modificação dos componentes *Scrum* e validação da proposta são discutidas nas seções a seguir.

## 9.2 Proposta da Abordagem

Nesta seção, apresenta-se uma visão geral da abordagem proposta baseada no arcabouço *Scrum* que é uma estrutura empregada para gerenciar e controlar as diversas atividades de desenvolvimento de software (Seção 2.5). Sabe-se que a existência e o acúmulo de anomalias de código em projetos de software podem conduzir à degradação de diversos atributos de

qualidade, impactando significativamente na capacidade do time de desenvolvimento manter e evoluir o sistema de software. Embora existam abordagens distintas para realizar a detecção e gestão destas anomalias, faz-se necessário o entendimento de "como" e "quando" empregar tais abordagens em um processo de desenvolvimento.

Sistematizar o uso das abordagens DI e DNI em um ambiente ágil é de suma importância por diversos motivos. Em um contexto ágil, caracterizado pela flexibilidade e constante adaptação, a sistematização proporciona uma base sólida para a detecção de anomalias. Isso garante que a equipe mantenha um padrão de qualidade consistente e evita a interrupção causada por abordagens desordenadas e inconsistentes. Ademais, é crucial preservar o uso complementar de DI e DNI devido às suas vantagens individuais. A utilização de DI pode ajudar a identificar e corrigir problemas específicos em trechos de código locais, o que pode melhorar a qualidade do código e reduzir o tempo necessário para corrigir esses problemas. Essa abordagem também pode ser mais rápida e menos dispendiosa do que a detecção globalizada, pois se concentra em áreas específicas do código que precisam de atenção. Por outro lado, o uso da abordagem DNI pode ajudar a identificar problemas mais amplos em todo o sistema de software, o que pode ajudar a melhorar a manutenibilidade, escalabilidade e desempenho do código. Essa abordagem pode ser mais útil em projetos maiores e mais complexos, onde existem muitos trechos de código interdependentes e problemas podem surgir em várias áreas do sistema. No entanto, a detecção DNI pode ser mais demorada e dispendiosa, uma vez que requer uma análise global do código-fonte.

Na Figura 9.2 ilustra-se uma visão geral da proposta de abordagem para desenvolvimento de software que integra a DI numa instanciação do arcabouço *Scrum*. A numeração existente na figura serve como base para descrição do seu funcionamento, sendo cada um desses itens descritos em detalhes a seguir.

1. Além do *backlog* do produto, existe também um artefato para gestão das ocorrências de anomalias de código. Este artefato deve ser compartilhado e mantido com auxílio de toda a equipe. As anomalias de código contidas nesta lista global do projeto serão identificadas através do uso de abordagens de Detecção Não-Interativa (DNI) no intuito de obter uma visão mais ampla da incidência desses problemas ao longo de todo projeto.

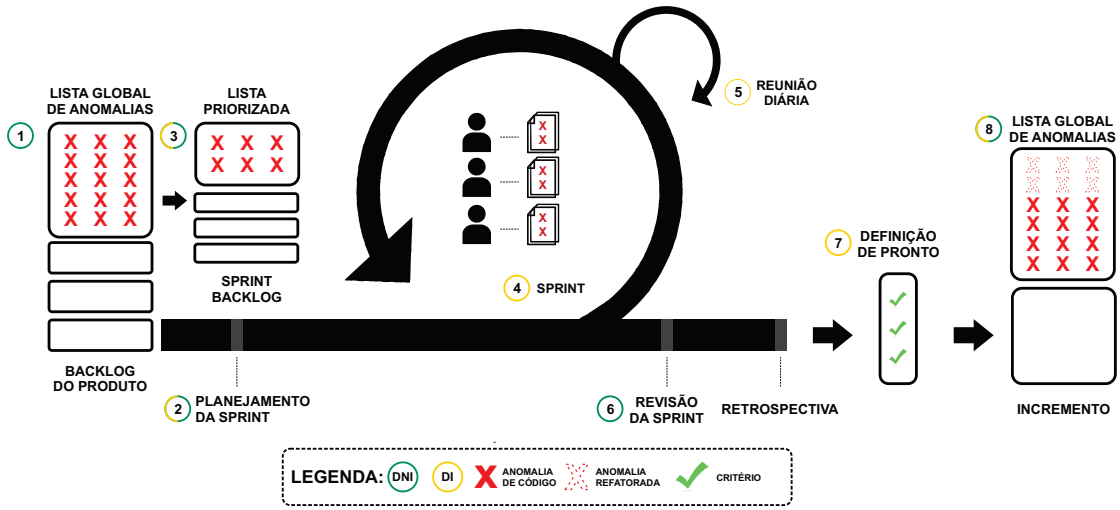


Figura 9.2: Etapas de Integração da Abordagem DI no Arcabouço do *Scrum*.

2. Ao longo da realização do planejamento da *sprint*, os itens da lista global de anomalias de código serão analisados e priorizados de acordo com critérios pré-definidos e mediante concordância do dono do produto. Para essas atividades, a equipe poderá alternar o uso de DNI e DI no intuito de retificar ou ratificar o planejamento de itens de anomalias de código. Essa ação é necessária para confirmar quais desses itens serão transformados em tarefas ao longo da próxima *sprint*.

3. Os itens da lista global de maior severidade/criticidade são priorizados e ordenados para execução na *sprint*. Cada ocorrência de anomalia (ou conjunto destas) desta lista priorizada de anomalias será transformada em tarefas específicas de acordo com a capacidade da equipe. Fatores como granularidade, nível de espalhamento, elementos de código afetados serão utilizados no processo decisório em relação ao tamanho e duração destas tarefas. Estas tarefas serão atribuídas ao time de desenvolvimento que deverá executá-las ao longo da *sprint*.

4. Os desenvolvedores deverão avaliar continuamente os artefatos de software gerados na *sprint* através das suas atividades de programação mediante uso da abordagem DI. Desse modo, sempre que uma nova instância de anomalia for inserida no código, o desenvolvedor deverá fazer uma avaliação primária baseada em diversos fatores (e.g., nível de espalhamento da anomalia, tempo investido na execução da refatoração e esforço demandado) para decidir a respeito da remoção imediata ou posterior desta

anomalia. No segundo caso, esta anomalia será incluída na lista global de anomalias e sua resolução será deliberada nas *sprints* futuras.

5. Durante a reunião diária, todos os desenvolvedores deverão informar a respeito da ocorrência de anomalias e explicitar os motivos que levaram a postergar sua remoção através das ações de refatoração. Tal decisão poderá ser discutida entre os membros da equipe que poderá ser modificada ou mantida através de um processo colaborativo entre os membros do time.

6. Na revisão da *sprint*, a equipe deverá realizar uma análise global da qualidade do código em termos de anomalias remanescentes. Essa atividade poderá ser realizada com suporte da abordagem DNI. Os resultados desta análise deverão ser identificados e catalogados para subsidiar ações futuras.

7. Antes dos itens realizados ao longo da *sprint* serem considerados "prontos", é importante que todos estes itens sejam checados de acordo com o conceito de "definição de pronto". Nesse contexto, a abordagem DI pode subsidiar a equipe *Scrum* a tomar decisões se determinadas tarefas foram de fato concluídas. A abordagem DI poderá fornecer dados e indicadores (e.g., métricas de qualidade de software, acúmulo de ocorrências de anomalias de código) que podem servir de critérios de aceitação para conclusão das tarefas da *sprint*. Caso ao final da *sprint* seja feita a entrega de um incremento em granularidade de uma *release* ou versão do software em desenvolvimento, faz-se necessária uma análise mais ampla de todo o projeto. Nesse caso, o uso da abordagem DNI poderá fornecer dados e indicadores que podem servir de critérios de aceitação de uma *release* ou versão do software.

8. Ao término do ciclo iterativo da *sprint*, o item correspondente na lista de *backlog* global de anomalias é atualizado novamente com as anomalias que foram corrigidas durante a *sprint*, bem como com as novas anomalias identificadas durante a iteração. Essa tarefa pode ser realizada utilizando ambas as abordagens. Por exemplo, a abordagem interativa de detecção pode ser utilizada para confirmar a remoção das anomalias mais locais que foram priorizadas na *sprint*. Por outro lado, a abordagem não interativa pode ser empregada para validar a detecção de novas instâncias mais globais do

projeto.

## 9.3 Adaptação dos Componentes do *Scrum*

O funcionamento da proposta de abordagem para desenvolvimento de software que integra a DI ao *Scrum* requer alteração em seus papéis, artefatos e eventos. Nos itens a seguir são expostas as principais mudanças em cada um desses componentes com o intuito de torná-los adaptados à proposta desta parte do estudo.

### 9.3.1 Papéis

Os papéis originais do *Scrum* são o dono do produto, o *Scrum master* e o time de desenvolvimento. No contexto da proposta de abordagem que integra a DI faz-se necessária a criação de um papel responsável por gerir ações associadas à manutenção e melhoria da qualidade de código em desenvolvimento pelo time. Esse papel será denominado de "Avaliador de qualidade", exercido prioritariamente por um profissional com conhecimento na área de detecção e refatoração de anomalias de código. O avaliador de qualidade deverá orientar os esforços dos integrantes do time que realizam atividades de programação.

### 9.3.2 Artefatos

Os artefatos originais do *Scrum* são *Backlog* do produto, *Sprint Backlog* e Incremento. Para operacionalização da proposta de abordagem que integra a DI no *Scrum* deve-se incrementar o *Backlog* do produto - e consequentemente o *Backlog* da *sprint* - com uma lista de ocorrências de anomalias de código. No caso do incremento do produto, os dados e indicadores provenientes das abordagens DI e DNI servirão como critérios de aceitação da "definição de pronto" dos diversos artefatos gerados pelo time.

### 9.3.3 Eventos

No contexto da proposta de abordagem que integra a DI, o foco estará restrito aos eventos diretamente associados ao desenvolvimento de software: (i) reunião de planejamento da

*sprint*; (ii) *sprint*; (iii) reunião diária; e (iv) revisão da *sprint*. No texto que segue são descritas as mudanças propostas em cada um desses eventos do *Scrum*.

1. A reunião de planejamento da *sprint* será modificada para disponibilizar um período de tempo voltado para análise do artefato global de anomalias de código do projeto. Nesse período deverá ser realizada uma análise com intuito de priorizar e ordenar as ocorrências de anomalias de acordo com seu nível de severidade e criticidade. O avaliador da qualidade deverá deliberar junto ao dono do produto e todo time de desenvolvimento para, de modo colaborativo, priorizar as ocorrências de anomalias que deverão ser removidas através de tarefas pertencentes a *sprint* atual. Esta ação está intimamente ligada ao pilar "adaptação" do *Scrum* que significa a capacidade de adaptar o projeto à necessidade de negócio.

2. A *sprint* deverá ser modificada para encorajar todos os integrantes do time de desenvolvimento a realizarem inspeção contínua dos artefatos de código. Essa atividade de inspeção deverá ser sistematizada ao longo do dia de trabalho mediante suporte da abordagem DI. Esta ação está intimamente ligada ao pilar "inspeção" do *Scrum* que recomenda a inspeção constante de tudo que está sendo gerado. Isso deve ser encarado como boa prática, uma vez que oportunidades de refatoração mais simples podem emergir com a identificação precoce das anomalias de código.

3. Na reunião diária não há alterações de processo. Apenas, deve-se garantir que no momento em que os componentes do time de desenvolvimento explicitam "Quais problemas enfrentou", a ocorrência de anomalias de código de maior criticidade e severidade sejam expostas. Estas anomalias surgiram ao longo das atividades de programação da *sprint* atual e tiveram sua remoção postergada. A decisão de postergar a ação de refatoração deverá ser baseada na dimensão da tarefa necessária para realizar a ação de refatoração. Caso essa tarefa extrapole o previsto para o *sprint backlog*, ela deverá ser postergada para *sprints* futuras.

4. Na revisão da *sprint* deve haver um momento para análise da lista global de anomalias de código. Nesta ocasião é feita uma análise da versão anterior e atual da lista com intuito de ter uma visão global do aumento (ou diminuição) da ocorrência da anomalia.

Neste momento também o time expõe, em maiores detalhes, as anomalias de código de maior criticidade e severidade. Sobretudo aquelas que foram mencionadas brevemente nas reuniões diárias e que tiveram sua resolução postergada em virtude da dimensão da tarefa de refatoração extrapolar as atividades da *sprint*. Esta ação está intimamente ligada ao pilar "transparência" do *Scrum*, que ocorre através da comunicação (verbal ou escrita) sobre os diversos artefatos gerados nos diversos eventos do arcabouço.

## 9.4 Validação *Offline* da Abordagem DI Integrada ao Processo Ágil

Após a apresentação da proposta de abordagem para desenvolvimento de software que integra a abordagem DI numa instanciação do arcabouço *Scrum*, a presente seção tem como objetivo realizar a validação desta proposta mediante aplicação de um grupo focal. A aplicação dessa técnica de pesquisa permite coletar dados qualitativos sobre as opiniões, percepções e experiências dos participantes em relação a um determinado tema [54]. Ao escolher um grupo focal, é possível obter *insights* valiosos a partir da interação entre os participantes, que podem ter diferentes perspectivas e experiências com as abordagens de detecção de anomalias, bem como o arcabouço *Scrum*.

O objetivo da abordagem DI é identificar e corrigir problemas de código de forma precoce, sendo uma estratégia promissora para melhorar o desenvolvimento de software em ambientes ágeis, como o *Scrum*. A realização de um grupo focal pode ser uma forma efetiva de avaliar a adequação da abordagem DI no contexto do processo ágil de desenvolvimento, identificando possíveis pontos fortes e fracos da abordagem em relação aos objetivos e valores do *Scrum*. Em suma, a abordagem DI pode trazer benefícios significativos para o desenvolvimento de software ágil e a realização de um grupo focal pode ajudar a avaliar sua adequação em ambientes específicos, como o *Scrum*.

No que segue, serão exibidos detalhes sobre o planejamento (Seção 9.4.1) e execução (Seção 9.4.2) do grupo focal. Em seguida, serão apresentados os dados obtidos a partir da aplicação do grupo focal (Seção 9.4.3), bem como as ameaças à validade desta técnica no contexto da validação da abordagem DI no contexto do *Scrum* (Seção 9.4.4).

### 9.4.1 Planejamento do Grupo Focal

A realização de um grupo focal envolve algumas etapas importantes que precisam ser seguidas para garantir a eficácia do método. Para assegurar a efetividade do método, seguiu-se as recomendações descritas no trabalho de Kontio et al. [54]. O *primeiro passo* envolve a definição do objetivo do grupo focal e o que se espera obter com sua realização. No contexto de um grupo focal, é crucial manter em perspectiva os objetivos que se deseja alcançar e identificar as questões que precisam ser abordadas durante as discussões em grupo. Desse modo, o objetivo principal da realização deste grupo focal consistiu na validação de uma proposta de abordagem para o desenvolvimento de software que integra DI em uma instância do arcabouço *Scrum*. A validação desta proposta envolveu a obtenção da percepção de potenciais usuários, como desenvolvedores de software e gerentes, a respeito da adoção prática e utilidade da abordagem proposta.

O *segundo passo* no processo de realização de um grupo focal envolve a seleção cuidadosa dos seus participantes. É essencial escolher pessoas que possuam características relevantes para a pesquisa, garantindo a diversidade e representatividade do grupo. Dado que a pesquisa tem como objetivo compreender as percepções dos usuários em relação à aplicação de uma abordagem para detecção de anomalias de código no contexto do processo ágil de desenvolvimento, é importante selecionar participantes com diferentes níveis de experiência, a fim de obter perspectivas diversas e enriquecedoras para a discussão.

Após a seleção dos participantes, o *terceiro passo* para a realização de um grupo focal envolve a definição de um roteiro de perguntas e tópicos a serem abordados durante a discussão em grupo. Esse roteiro deve ser cuidadosamente elaborado para orientar o debate entre os participantes, garantindo que os objetivos da pesquisa sejam atendidos. É importante que o roteiro seja flexível o suficiente para permitir que a discussão ocorra de forma natural e para que novos tópicos relevantes possam surgir e serem abordados.

Por último, o *quarto passo* no processo de realização de um grupo focal refere-se à escolha de um local apropriado para a sua realização. É fundamental que o ambiente seja confortável e tranquilo, a fim de favorecer a discussão e a participação dos membros do grupo. Além disso, o local escolhido deve ser facilmente acessível e conveniente para os participantes, de modo a minimizar possíveis inconvenientes e garantir a participação de todos. Com essas etapas bem definidas e executadas, é possível obter informações valiosas

por meio da realização de um grupo focal.

### 9.4.2 Execução do Grupo Focal

A realização da sessão do grupo focal é considerada uma das etapas mais cruciais, visto que é neste momento que a discussão efetivamente se desenvolve. É primordial que o moderador conduza a sessão de forma a permitir que todos os participantes possam expressar suas opiniões e pontos de vista, ao mesmo tempo que promove um ambiente respeitoso e produtivo. O moderador tem como responsabilidade garantir que todas as questões do roteiro sejam abordadas durante a discussão.

O grupo focal ocorreu em maio de 2023 e teve uma duração aproximada de 110 minutos. A atividade foi conduzida utilizando a plataforma *Google Meet*, permitindo a participação remota dos participantes, o que proporcionou uma solução conveniente e econômica em termos de tempo e recursos, uma vez que os participantes estavam localizados em diferentes regiões geográficas. Foram recrutados 8 (oito) participantes por meio de uma mensagem de *e-mail* e com base no interesse explícito e voluntário em participar do grupo focal. Todos os participantes eram do Núcleo de Pesquisa, Desenvolvimento e Inovação da Universidade Federal de Campina Grande (VIRTUS/UFCG) e desempenham os papéis de desenvolvedores e gerentes de projeto. Na Tabela 9.1 encontra-se descrito o perfil dos participantes, sendo seus nomes preservados por questões de confidencialidade.

Com relação ao nível de conhecimento, era esperado que os participantes estivessem pelo menos moderadamente familiarizados com a linguagem Java e o processo ágil de desenvolvimento de software. No entanto, não esperava-se o amplo conhecimento dos participantes sobre conceitos como anomalias de código, refatoração, bem como as abordagens de detecção de anomalias consideradas nesse estudo. De modo geral, os participantes atendem às premissas de participação no grupo focal tendo em vista que estes alegaram ser “muito proficientes” ou “proficientes” em todos conceitos associados ao grupo focal (i.e., Programação OO, Linguagem Java, Detecção de anomalias, refatoração e *Scrum*). Na Figura 9.3 são exibidos em detalhes os resultados da consulta aos participantes a respeito do nível de proficiência em conceitos associados ao grupo focal.

Para guiar as atividades do grupo focal, estabeleceu-se um roteiro a ser seguido. A *primeira parte* incluiu uma apresentação sobre o método do grupo focal, o objetivo desta técnica

Tabela 9.1: Especialistas Selecionados para o Grupo Focal.

| **Especialista 1 (E1)** | | **Especialista 2 (E2)** | |
|---|---|---|---|
| Nível de Experiência | *12 anos* | Nível de Experiência | *11 anos* |
| Dados Demográficos | *Idade: 36* | Dados Demográficos | *Idade: 37 anos* |
| | *Gênero: feminino* | | *Gênero: masculino* |
| | *Formação: mestrado* | | *Formação: especialização* |
| | *Função: gerente* | | *Função: gerente* |
| Habilidades Técnicas | *DevOps, Redes de computadores, Segurança, Python, C#, scrum e testes.* | Habilidades Técnicas | *Java, Ruby, Javascript, C#, Git, IntelliJ, Desenvolvimento Web* |
| **Especialista 3 (E3)** | | **Especialista 4 (E4)** | |
| Nível de Experiência | *5 anos* | Nível de Experiência | *5 anos* |
| Dados Demográficos | *Idade: 26 anos* | Dados Demográficos | *Idade: 25 anos* |
| | *Gênero: masculino* | | *Gênero: masculino* |
| | *Formação: graduação* | | *Formação: especialização* |
| | *Função: líder técnico* | | *Função: desenvolvedor* |
| Habilidades Técnicas | *Desenvolvimento, Angular, React, Docker, Java, Scrum, Web, Linux, Scrum Master* | Habilidades Técnicas | *Typescript, Java, IntelliJ, WebStorm, Agile Development e Scrum* |
| **Especialista 5 (E5)** | | **Especialista 6 (E6)** | |
| Nível de Experiência | *9 anos* | Nível de Experiência | *3 anos* |
| Dados Demográficos | *Idade: 26 anos* | Dados Demográficos | *Idade: 31 anos* |
| | *Gênero: masculino* | | *Gênero: masculino* |
| | *Formação: especialização* | | *Formação: mestrado* |
| | *Função: desenvolvedor* | | *Função:desenvolvedor* |
| Habilidades Técnicas | *.Net, C#, C++, GIT, WEB/-Mobile, Desktop/Embarcado, Scrum, Azure, SQL* | Habilidades Técnicas | *Desenvolvimento de Software, Frontend, Backend, Node js Reactjs* |
| **Especialista 7 (E7)** | | **Especialista 8 (E8)** | |
| Nível de Experiência | *6 anos* | Nível de Experiência | *5 anos* |
| Dados Demográficos | *Idade: 26 anos* | Dados Demográficos | *Idade: 27 anos* |
| | *Gênero: masculino* | | *Gênero: masculino* |
| | *Formação: graduação* | | *Formação: mestrado* |
| | *Função: desenvolvedor* | | *Função: desenvolvedor* |
| Habilidades Técnicas | *Nodejs, Java, Kotlin, Frontend, Scrum, Testes, IntelliJ, GIT, CI/CD, Sonarqube.* | Habilidades Técnicas | *Python, GIT, R, Tensorflow, Scrum, Pytorch, Machine and Deep Learning.* |

no contexto da pesquisa, bem como uma apresentação para fins de nivelamento do conhecimentos requeridos para realização do grupo focal (i.e., atividades 1, 2 e 3). Em seguida, a

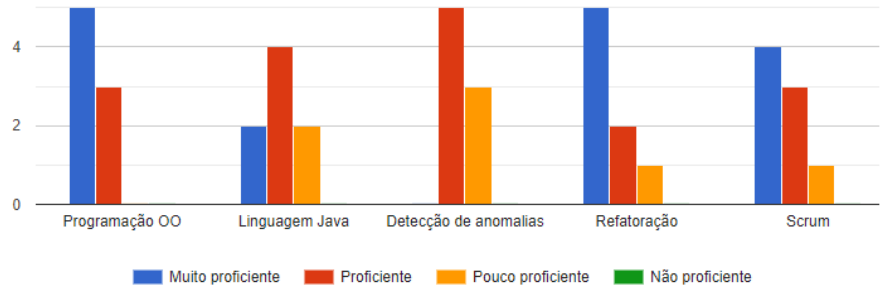


Figura 9.3: Nível de Proficiência em Conceitos Associados ao Grupo Focal.

*segunda parte* do grupo focal envolveu a apresentação da proposta de adaptação dos papéis, artefatos e eventos, e uma discussão entre os participantes sobre a pertinência das proposituras (i.e., atividades 4, 5 e 6). A *terceira parte* do grupo focal envolveu a apresentação da abordagem para desenvolvimento de software que integra a abordagem DI numa instanciação do arcabouço *Scrum*, seguida por uma discussão guiada a respeito da concordância dos participantes sobre a pertinência das atividades propostas (i.e., atividade 7). A *quarta parte* envolveu a demonstração prática do uso da abordagem DI integrada ao processo ágil em conjunto com uma discussão para identificar possíveis benefícios, desvantagens, bem como dificuldades de uso da abordagem (i.e., atividade 8). Finalmente, a *quinta parte* envolveu perguntas objetivas sobre a escalabilidade da abordagem, recomendação de uso, bem como foi dada liberdade para os participantes discutirem questões não cobertas ao longo do grupo focal (i.e., atividade 9). Mais detalhes sobre o roteiro e as atividades do grupo focal encontram-se descritas na Tabela 9.2, bem como no material suplementar do estudo [6].

Antes do início da sessão do grupo focal, obteve-se a permissão explícita dos participantes para registro de todas as atividades mediante "Termo de Consentimento Livre e Esclarecido" e apresentação das "Condições Gerais de participação". Durante a sessão do grupo focal, os participantes foram encorajados a ouvir atentamente, expressar suas opiniões, realizar perguntas e contextualizar a discussão quando necessário. Além disso, foram incentivados a compartilhar abertamente suas experiências e perspectivas. Para evitar que a discussão fosse monopolizada por um ou mais participantes, foram tomadas medidas para garantir que todos os participantes tivessem a oportunidade de falar e contribuir.

Tabela 9.2: Roteiro para Realização do Grupo Focal.

| Parte | Atividade | Duração | Descrição |
|---|---|---|---|
| I | 1 | 3 min | Apresentação dos objetivos do grupo focal |
| | 2 | 10 min | Apresentação das abordagens de detecção de anomalias |
| | 3 | 5 min | Apresentação do arcabouço *Scrum* |
| II | 4 | 5 min | Apresentação da adaptação dos papéis |
| | 4 | 5 min | Questionamentos<br>- Está clara a adaptação dos papéis?<br>- É necessária a realização de adaptações adicionais? |
| | 5 | 5 min | Apresentação da adaptação dos artefatos |
| | 5 | 5 min | Questionamentos:<br>- Está clara a adaptação dos artefatos?<br>- É necessária a realização de adaptações adicionais? |
| | 6 | 5 min | Apresentação da adaptação dos eventos? |
| | 6 | 5 min | Questionamentos:<br>- Está clara a adaptação dos eventos?<br>- É necessária a realização de adaptações adicionais? |
| III | 7 | 5 min | Apresentação da abordagem que integra a abordagem DI numa instanciação do arcabouço *Scrum* |
| | 7 | 10 min | Questionamentos:<br>- Concordância sobre a pertinência das atividades propostas |
| IV | 8 | 5 min | Apresentação de exemplo prático de uso da abordagem |
| | 8 | 5 min | Questionamentos:<br>- Benefícios, desvantagens e dificuldades de uso a abordagem |
| V | 9 | 5 min | Questionamentos Finais |
| | 9 | 2 min | Agradecimentos e conclusão |

### 9.4.3 Análise dos Dados do Grupo Focal

A análise dos dados obtidos durante a sessão do grupo focal foi conduzida através de uma abordagem qualitativa, cujo objetivo é consolidar, reduzir e interpretar os dados coletados a fim de obter uma compreensão mais aprofundada dos resultados. Na análise qualitativa dos dados, os áudios foram transcritos e organizados em uma planilha da ferramenta *Microsoft Excel*. No que segue, serão expostos os principais resultados obtidos de acordo com as

atividades do roteiro, conforme demonstrado na Tabela 9.2. É importante mencionar que maiores detalhes a respeito do planejamento e execução do grupo focal podem ser obtidos mediante acesso ao material suplementar do estudo [6].

### Adaptação dos Componentes do *Scrum*

A partir da análise dos dados da segunda parte do grupo focal, buscou-se avaliar a concordância dos participantes em relação à adaptação dos componentes do arcabouço *Scrum*. No que segue, apresentam-se os resultados associados à adaptação dos papéis, artefatos e eventos visando suportar a abordagem DI no processo ágil de desenvolvimento.

**Adaptação dos papéis**. Inicialmente, apresentam-se os resultados associados à proposta de criação de um novo papel (i.e., avaliador de qualidade) no arcabouço do *Scrum*. Na Figura 9.4 encontram-se descritos os resultados sobre a concordância dos participantes em relação a criação deste papel.

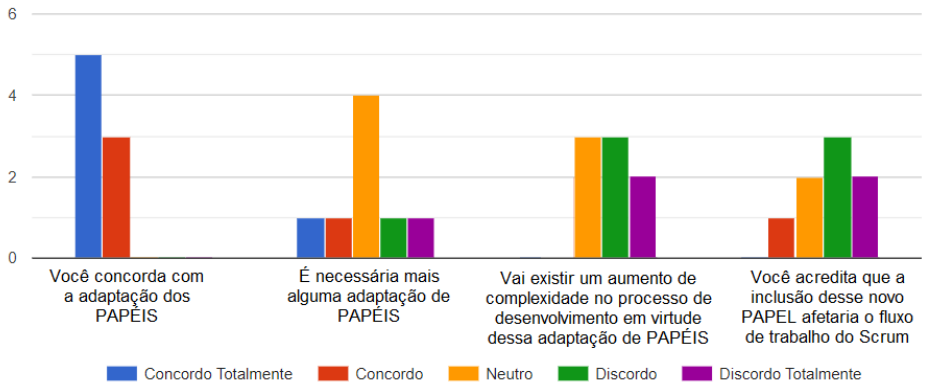


Figura 9.4: Questões Associadas à Adaptação dos Papéis.

Quando questionados sobre sua concordância com a criação deste papel, a maioria dos participantes concordou totalmente (5), enquanto apenas três concordaram parcialmente. Isso sugere que a proposta foi bem recebida pelos participantes do grupo focal e que eles veem valor na inclusão desse novo papel para melhorar a qualidade do processo de desenvolvimento e, consequentemente, da qualidade do código. No entanto, quando perguntados se outras adaptações de papéis eram necessárias, a maioria dos participantes discordou (6) e apenas dois concordaram. Isso pode indicar que os participantes veem a proposta de criação do "avaliador de qualidade" como suficiente para melhorar o processo de desenvolvimento e

que outras mudanças podem ser desnecessárias ou até prejudiciais.

Com relação ao aumento da complexidade no processo de desenvolvimento decorrente da criação deste novo papel, a maioria dos participantes discordou (5) e três foram neutros. Isso sugere que a proposta de criação do "avaliador de qualidade" não é vista como uma mudança significativa que possa complicar o processo de desenvolvimento, o que é positivo para a aceitação da proposta. Ao serem questionados se a inclusão desse novo papel afetaria o fluxo de trabalho do *Scrum*, a maioria dos participantes discordou (5), dois foram neutros e apenas um concordou. Essa constatação sugere que a adição do papel de "avaliador de qualidade" pode ser facilmente incorporada ao fluxo de trabalho do *Scrum*, contribuindo positivamente para elevar a aceitação da proposta.

**Adaptação dos artefatos**. Na presente atividade, apresentam-se os resultados referentes à proposta de adição de um novo artefato ao arcabouço do *Scrum* (i.e., Lista Global de Anomalias no *Backlog* do Produto). Na Figura 9.5 encontram-se descritos os resultados sobre a concordância dos participantes em relação a criação deste artefato.

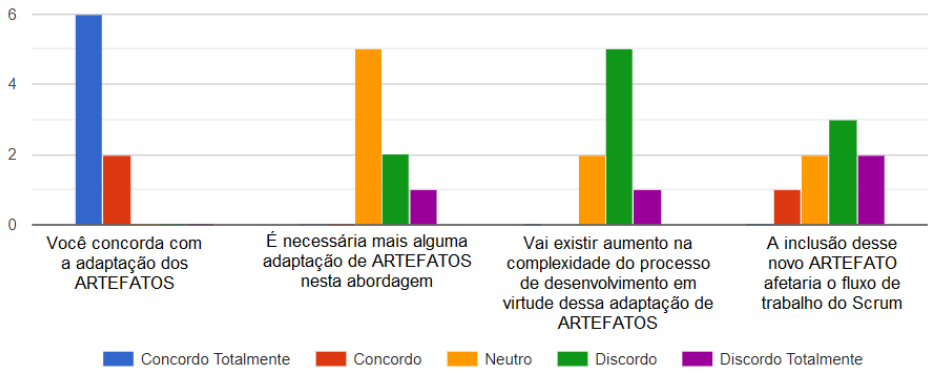


Figura 9.5: Questões Associadas à Adaptação dos Artefatos.

Os resultados obtidos durante a sessão do grupo focal indicam que a criação da "Lista Global de anomalias no *Backlog* do Produto" foi bem recebida pelos participantes. A maioria dos participantes (6) concordou totalmente com a inclusão desse artefato, o que sugere que eles reconhecem o valor que este artefato pode agregar ao processo de desenvolvimento. Isso pode ser explicado pelo fato de que a "Lista Global" pode auxiliar na gestão das anomalias de código. Além disso, esse novo artefato pode proporcionar uma visão mais clara e detalhada das atividades de detecção e refatoração das anomalias, possibilitando uma melhor comunicação e colaboração entre os envolvidos nessas atividades.

Por outro lado, quando perguntados se havia necessidade de outras adaptações de artefatos, a maioria dos participantes se mostrou neutra (5) ou discordou (3). Esses resultados podem ser explicados pelo fato de que a inclusão da "Lista Global" já é vista como uma mudança significativa o suficiente para melhorar o processo de desenvolvimento. Além disso, outras mudanças poderiam ser desnecessárias ou mesmo prejudiciais, uma vez que podem adicionar mais complexidade ao processo sem trazer benefícios aparentes.

Com relação ao aumento de complexidade no processo de desenvolvimento decorrente da adaptação de artefatos, a maioria dos participantes expressou discordância (6). Esses resultados sugerem que a introdução da "Lista Global" não é vista como uma mudança que possa complicar significativamente o processo de desenvolvimento, o que pode ser considerado um indicador positivo para a aceitação da proposta em questão. Uma possível explicação para essa percepção é que o novo artefato pode ser facilmente incorporado às atividades já existentes no *Scrum*, sem causar grandes impactos no fluxo de trabalho.

Por fim, ao serem questionados se a inclusão desse novo artefato afetaria o fluxo de trabalho do *Scrum*, a maioria dos participantes discordou (6). Essa resposta sugere que a inclusão da "Lista Global" não deve impactar significativamente o fluxo de trabalho do *Scrum*, o que é positivo para a aceitação da proposta. Isso pode ser explicado pelo fato de que este artefato pode ser integrado de forma transparente ao processo de desenvolvimento, sem interromper as atividades existentes no *Scrum*.

**Adaptação dos eventos**. Nesta atividade demonstra-se os resultados associados à proposta de adaptação dos eventos do *Scrum* (i.e., Planejamento da *Sprint*, *Sprint*, Reunião diária e Revisão da *Sprint*). Na Figura 9.6 encontram-se descritos os resultados sobre a concordância dos participantes em relação a adaptação destes eventos.

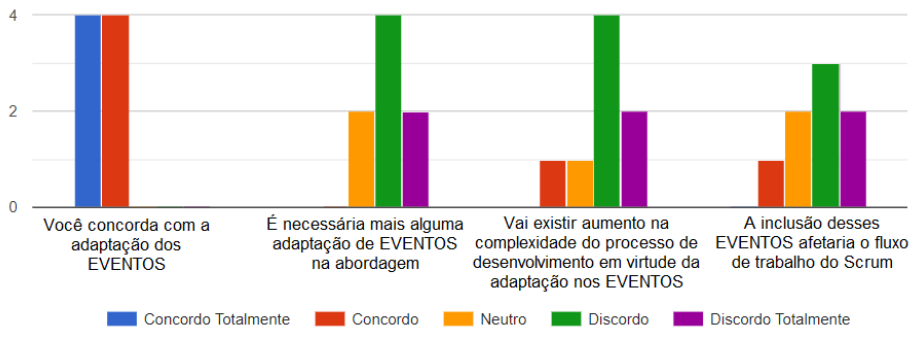


Figura 9.6: Questões Associadas à Adaptação dos Eventos.

Quando perguntados sobre a concordância em relação a essa adaptação, metade dos participantes concordaram totalmente (4) e a outra metade concordou parcialmente (4). Esses resultados podem indicar que a adaptação dos eventos foi vista como positiva pelos participantes, embora nem todos concordassem completamente. No entanto, quando perguntados se havia necessidade de mais adaptações nos eventos, a maioria dos participantes discordou (6) ou se mostrou neutra (2). Isso pode indicar que as adaptações propostas são suficientes para melhorar o processo de desenvolvimento e que outras mudanças poderiam ser desnecessárias ou mesmo prejudiciais.

Em relação ao aumento de complexidade no processo de desenvolvimento devido às adaptações nos eventos, a maioria dos participantes discordou (6). Isso sugere que as adaptações propostas não são vistas como uma mudança significativa que possa complicar o processo de desenvolvimento. No entanto, é importante lembrar que um participante concordou, indicando que a adaptação pode ter um impacto na complexidade. Ao serem questionados se a inclusão dessas adaptações afetaria o fluxo de trabalho do *Scrum*, a maioria dos participantes discordou (6), embora dois participantes concordaram com a afirmação. Isso pode indicar que a inclusão das adaptações pode afetar o fluxo de trabalho de algumas equipes, mas que em geral, elas não devem ter um impacto significativo.

#### Apresentação da abordagem DI numa instanciação do *Scrum*

A partir da análise dos dados da terceira parte do grupo focal, buscou-se colher dados associados à integração da abordagem DI ao processo ágil de desenvolvimento. Na Figura 9.7 encontram-se descritos os resultados que refletem o nível de concordância dos participantes do grupo focal em relação à questões sobre a integração da abordagem DI no arcabouço do *Scrum*.

Os resultados da pesquisa indicam que a integração da abordagem DI no arcabouço do *Scrum* pode ajudar a tornar o processo de detecção de anomalias de código mais eficaz. A maioria dos participantes (7 em 8) concordou totalmente com essa afirmação. Esse resultado pode ser justificado pela adoção da abordagem DI, a qual possibilita a detecção de anomalias de código em tempo real durante o processo de desenvolvimento. Tal prática permite que as equipes de desenvolvimento identifiquem e corrijam imediatamente os problemas encontrados, contribuindo para a redução dos custos e do tempo de retrabalho associados a anomalias

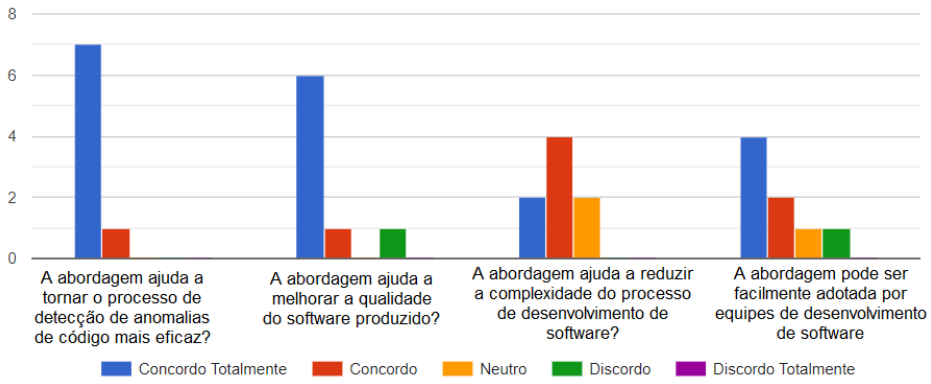


Figura 9.7: Questões Associadas à Integração da DI no Processo Ágil.

não detectadas em fases posteriores do processo de desenvolvimento de software.

Por outro lado, em relação à questão de que a abordagem DI integrada ao *Scrum* ajudar a melhorar a qualidade do software produzido, a maioria dos participantes (6 em 8) concordou totalmente com a afirmação, enquanto um participante discordou. É possível que a discordância possa ser explicada por diferenças de opinião entre os participantes sobre o que constitui uma "melhoria" na qualidade do software. No entanto, a maioria dos participantes que concordaram com a afirmação podem ter considerado que a abordagem DI pode ajudar a melhorar a qualidade do software, permitindo que os problemas de código sejam detectados e corrigidos mais rapidamente, antes que eles possam afetar a funcionalidade geral do software. Além disso, a abordagem DI pode incentivar as equipes de desenvolvimento a adotarem práticas de desenvolvimento mais colaborativas e a se comunicarem com mais eficiência, o que pode levar a um software de maior qualidade.

Um dos objetivos da integração da abordagem DI ao *Scrum* é tornar o processo de desenvolvimento de software menos complexo. Na pesquisa realizada, metade dos participantes (4) concordaram que a DI integrada ao *Scrum* ajuda a reduzir a complexidade do processo de desenvolvimento de software, enquanto outra metade foram neutros (4) em relação a essa afirmação. Estes resultados indicam que a adoção da abordagem DI pode ajudar a tornar o processo de desenvolvimento de software menos complexo e mais eficiente. Essa abordagem permite que os desenvolvedores detectem e corrijam problemas de código à medida que desenvolvem, em vez de deixar essas correções para fases posteriores de desenvolvimento. Isso pode ajudar a reduzir a complexidade do processo, permitindo que as equipes resolvam problemas de forma mais rápida e eficaz, o que pode levar a uma redução nos custos e no

tempo necessários para o desenvolvimento do software.

Outro aspecto importante da integração da abordagem DI ao *Scrum* é sua adoção por equipes de desenvolvimento de software. Na pesquisa, 6 participantes concordaram que a DI integrada ao *Scrum* pode ser facilmente adotada por equipes de desenvolvimento de software, enquanto 1 foi neutro e 1 discordou dessa afirmação. Isso sugere que a abordagem DI pode ser implementada com sucesso em equipes de desenvolvimento de software sem grandes desafios. Uma possível explicação para isso é que o *Scrum* já é amplamente adotado em muitas organizações de desenvolvimento de software, e a abordagem DI pode ser facilmente integrada aos processos existentes. Além disso, a abordagem DI é baseada em práticas ágeis e enfatiza a colaboração e a comunicação contínua entre os membros da equipe, o que pode ajudar na melhoria do processo de desenvolvimento de software.

#### Exemplo prático de uso da abordagem DI numa instanciação do *Scrum*

A partir da análise dos dados da quarta parte do grupo focal, buscou-se colher dados a respeito das percepções dos participantes a respeito da demonstração de um exemplo de uso da abordagem DI integrada ao arcabouço do *Scrum*. Nesse sentido, analisou-se os principais benefícios, dificuldades, bem como desvantagens associadas à utilização da abordagem. Tais aspectos serão apresentados e discutidos no que segue.

**Benefícios**. Ao analisar as respostas dos participantes do grupo focal, foram identificadas categorias de benefícios provenientes da utilização da abordagem DI integrada ao arcabouço do *Scrum*. A seguir, destacamos as categorias mais relevantes e transcrevemos algumas falas dos participantes associadas a essas categorias de benefícios.

1. Melhoria na qualidade de código (8 respostas). As respostas nessa categoria indicam que a abordagem DI integrada ao *Scrum* contribui diretamente para melhoraria da qualidade do código.

   Transcrição da fala de especialistas (E): E1 "*...ao usar a abordagem DI integrada ao processo ágil, iremos realizar com maior facilidade a manutenção do código, pois estaremos usando uma ferramenta e não realizando de forma manual, com isso teremos uma melhoria na qualidade do software, pois estaremos detectando e poderemos realizar a correção da anomalia, tendo assim uma maior produtividade ao realizar as*

*atividades*"; E2 "*...a adoção da DI, proporcionará uma melhoria na qualidade do código devido ao auxílio automático do recurso ao ponto de que proporcionará melhor manutenção do código visto que previamente já foi implementado usado o DI para detecção de anomalias na primeira implementação*"; E3 "*...é perceptível a melhoria na qualidade geral do código gerado que, por sua vez, gera uma melhor manutenibilidade"*; E4 "*...os benefícios na qualidade são claros* "; E5 "*Maior qualidade no código entregue, que por sua vez é sujeito a menos bugs implementados no sistema*"; E6 "*...com a abordagem DI integrada ao Scrum, conseguimos detectar problemas no código mais cedo, o que significa que podemos corrigi-los antes que se tornem um problema maior.*"; E7 "*o usar a DI, podemos desenvolver código mais limpo e organizado*"; e E8 "*...estamos sempre fazendo melhorias incrementais no código e corrigindo problemas à medida que eles surgem*".

2. Melhoria na entrega do produto (6 respostas). As respostas nessa categoria apontam que a abordagem DI integrada ao *Scrum* ajuda a melhorar a entrega do produto, permitindo que o software seja entregue com mais rapidez e qualidade.

   Transcrição da fala de especialistas (E): E2 "*...a entrega do produto será afetada positivamente, visto que dará mais segurança e velocidade de novas implementações*"; E3 "*...é perceptível a melhoria na entrega do produto ao final da Sprint"*; E6 "*...a aplicação da abordagem DI ao scrum pode promover melhora na detecção de anomalias no código, consequentemente aprimorando a qualidade final do produto*"; E8 "*...acredito que os benefícios estão diretamente associados à qualidade do produto*"; E4"*...com a DI, pode-se fazer uma integração contínua e entregas mais frequentes, resultando em um produto final mais estável e com menos erros*"; e E5 "*...identificar os problemas do software mais cedo permite corrigi-los antes da entrega, resultando em um produto final de melhor qualidade*".

3. Maior facilidade para manutenção (4 respostas). As respostas nessa categoria destacam que a abordagem DI integrada ao *Scrum* torna o código mais fácil de ser mantido, permitindo mudanças mais rápidas e eficientes.

   Transcrição da fala de especialistas (E): E4 "*...também garante uma maior facilidade de manutenção naqueles códigos que não contém anomalias*"; E7 "*...pelo fato do re-*

*torno pro desenvolvedor ser em real-time agiliza ainda mais o processo de manutenção*; E2"*...com a abordagem DI integrada ao Scrum, as mudanças no código são mais rápidas e seguras*"; e E1 "*..as características da DI podem tornar os desenvolvedores mais habilidosos na detecção de anomalias - facilitando a manutenção - em virtude das informações contextualizadas e locais*".

De modo resumido, as falas dos especialistas destacaram que a utilização da DI contribuiu diretamente para melhorar a qualidade do software, detectar problemas no código mais cedo, corrigi-los antes que se tornem um problema maior, desenvolver código mais limpo e organizado, permitir entregas mais frequentes e com menos erros, tornar o código mais fácil de ser mantido e permitir mudanças mais rápidas e eficientes. Esses benefícios foram observados em oito respostas para melhoria na qualidade de código, seis para melhoria na entrega do produto e quatro para maior facilidade para manutenção.

**Dificuldades**. Após análise das respostas dos participantes, identificaram-se algumas categorias de dificuldades relacionados à implementação da abordagem DI integrada ao *Scrum*. A seguir, apresentam-se as categorias de dificuldades mais relevantes e algumas citações dos especialistas participantes do grupo focal associadas a tais categorias.

1. Resistência à mudança (7 respostas). As respostas nesta categoria indicam resistência à mudança em relação à adoção da abordagem DI por falta de apoio da liderança ou mesmo de membros do time de desenvolvimento.

   Transcrição da fala de especialistas (E): E8 "*...a única dificuldade que observo a princípio seria adaptação das equipes a abordagem DI*"; E5 "*...como o uso desse tipo de integração não é muito utilizado atualmente, teria ali uma barreira inicial de implantação, que seria a falta de conhecimento/habilidades para implementar a abordagem. E muito provavelmente alguma resistência por parte da gestão ou time*"; E3 "*...as pessoas tendem a manter o status quo*"; E7 "*...times e organizações podem impor resistência na adição de mais um processo que poderá ser visto como uma etapa burocrática que impactará na produtividade de tempo para entrega das atividades, percepção errônea se comparada com os benefícios obtidos ao longo do tempo e maturidade e evolução do software*"; E1 "*...pessoas na liderança podem não entender completamente o valor da abordagem DI, então podem ser hesitantes em apoiar a sua*

*adoção*"; E2 "*...membros da equipe podem preferir trabalhar da maneira que estão acostumados e não querem mudar a maneira como fazem as coisas*"; E4 "*...alguns membros da equipe estão acostumados a trabalhar de uma certa maneira e não querem mudar sua rotina*" ; e E6 "*...existe um senso de conforto em fazer as coisas do jeito que sempre fizemos*".

2. Falta de conhecimento e habilidades (4 respostas). As respostas nesta categoria indicam falta de conhecimento e habilidade na aplicação da abordagem DI em virtude de treinamento e capacitação deficitários por parte da liderança ou mesmo de membros do time de desenvolvimento.

   Transcrição da fala de especialistas (E): E1 "*...as anomalias podem ser detectadas mas o desenvolvedor pode não ter conhecimento técnico suficiente para corrigi-la*"; E4 "*...acredito que a maior dificuldade, que vai ser encontrada vai ser a resistência das pessoas que iram utilizar dessa ferramenta, em aceitar que o que ele está fazendo é errado...*"; E5 "*...não temos membros da equipe que sejam especialistas em DI, então temos que aprender tudo do zero*"; e E7 "*...acredito que a abordagem DI seria boa para nosso projeto, mas não sei ainda como aplicá-la na prática*".

3. Dificuldades organizacionais (4 respostas). As respostas nesta categoria indicam dificuldades em implementar a abordagem DI devido a limitações organizacionais, tais como orçamento, recursos e cultura da empresa.

   Transcrição da fala de especialistas (E): E6 "*...creio que por tratar-se de uma pesquisa em andamento, pode haver uma resistência à mudança por parte de algumas organizações*"; E7 "*...talvez o PO possa priorizar demandas acima da resolução proposta pela DI*"; E3 "*...o ideal é implementar o processo desde o início do desenvolvimento*"; e E5 "*...a equipe de desenvolvimento pode estar sobrecarregada com muitas tarefas e projetos, então é difícil encontrar tempo e recursos para implementar a DI de forma efetiva*".

Resumidamente, os desafios do uso da detecção interativa (DI) no *Scrum* incluem resistência à mudança, falta de conhecimento e habilidades e dificuldades organizacionais. Os participantes do grupo focal apontam que a resistência pode ocorrer devido à falta de apoio

da liderança e de membros da equipe de desenvolvimento, bem como à preferência pela rotina e ao "conforto" em realizar as mesmas atividades no processo de desenvolvimento. A falta de conhecimento e habilidades pode ser uma barreira inicial para a implantação da DI, e a falta de treinamento e capacitação é um problema comum. Além disso, as dificuldades organizacionais podem incluir limitações de orçamento, recursos e cultura da empresa, bem como a sobrecarga de tarefas e projetos da equipe de desenvolvimento.

**Desvantagens**. Analisando as respostas dos participantes, foram definidas algumas categorias de desvantagens advindas da implementação da abordagem DI integrada ao processo ágil de desenvolvimento de software. No que segue apontamos as categorias mais relevantes bem como a transcrição de algumas falas dos especialistas participantes do grupo focal associadas às tais categorias de desvantagens.

1. Nenhuma desvantagem (5 respostas).

   Transcrição da fala de especialistas (E): E5 "*...nenhuma*"; E8 "*...não consegui observar desvantagens*" E7 "*...não vejo grandes desvantagens em usar o DI*"; E3 "*...não me é perceptível nenhuma desvantagem*"; e E6 "*...não consegui identificar desvantagens nesse processo*".

2. Dificuldade na adoção (3 respostas). As respostas nesta categoria indicam dificuldades em adotar a abordagem DI, especialmente se a equipe já estiver acostumada com outra abordagem ou método de desenvolvimento.

   Transcrição da fala de especialistas (E): E1 "*...acredito que por questões de resistência a mudanças do time, podemos ter uma dificuldade de adoção*"; E2 "*...devido a questões culturais de organizações, a dificuldade de adoção pode ser enfrentada*"; E5 "*...tudo que é novo causa dificuldade no primeiro momento*".

3. Dependência de Ferramentas (2 respostas). As respostas nesta categoria indicam que a abordagem DI integrada ao *Scrum* pode exigir o uso de ferramentas e tecnologias específicas, o que pode limitar as opções da equipe de desenvolvimento.

   Transcrição da fala de especialistas (E): E4 "*...por que utilizar essa abordagem e não uma ferramenta de qualidade de código já consagrada?*"; e E1 "*...podemos ter uma dificuldade de adoção da ferramenta*".

Em linhas gerais, os resultados não apresentaram desvantagens percebidas pela maioria dos participantes. É importante mencionar que alguns especialistas apontaram possíveis dificuldades na adoção da abordagem, principalmente em equipes que já estão acostumadas com outras abordagens ou métodos de desenvolvimento. Além disso, dois especialistas destacaram que a abordagem DI integrada ao *Scrum* pode exigir o uso de ferramentas e tecnologias específicas, o que pode limitar as opções da equipe de desenvolvimento.

Finalmente, a partir da análise dos dados da quinta parte do grupo focal, buscou-se avaliar a adequação da abordagem DI integrada ao *Scrum* para projetos de diferentes tamanhos, a maioria dos participantes (6) afirmou que a abordagem é adequada para projetos de qualquer tamanho. No entanto, dois participantes afirmaram que a abordagem é mais adequada para projetos de grande porte. Esses resultados podem indicar que a abordagem DI integrada ao *Scrum* pode ser aplicável em diferentes contextos de desenvolvimento de software, mas que em projetos de grande porte, a abordagem pode ter um impacto ainda maior na qualidade do software e na produtividade da equipe. Adicionalmente, perguntou-se aos participantes se eles recomendariam a adoção da abordagem DI integrada ao *Scrum* em projetos de desenvolvimento de software para outras equipes ou empresas. A grande maioria (7) respondeu afirmativamente que recomendariam a adoção dessa abordagem, enquanto apenas um respondeu com um talvez. Possíveis justificativas para essa resposta positiva podem ser o fato de que a abordagem DI pode trazer benefícios para o desenvolvimento de software, como a melhoria na qualidade do código e a diminuição do tempo de correção de erros, além de sua integração com o *Scrum*, que é uma metodologia ágil amplamente utilizada na indústria de software.

### 9.4.4 Ameaças à Validade - Grupo Focal

Em qualquer estudo experimental existem fatores que podem ser vistos como possíveis influências e ameaças à validade. Nesta seção discutem-se tais ameaças associadas à validação descrita neste capítulo, bem como as ações realizadas para mitigá-las. Para tanto, seguiu-se o esquema de classificação proposto por Wohlin et al. [125] conforme itens a seguir.

*Validade de construção*: Algumas ameaças podem ocorrer em um grupo focal como a falta de clareza nas instruções do roteiro, uso de perguntas tendenciosas ou confusas, ou a presença de viés por parte do moderador que possa influenciar a interpretação dos dados.

Para minimizar os efeitos dessas ameaças, foram realizadas revisões e refinamentos contínuos do protocolo de pesquisa, incluindo a descrição detalhada das etapas e procedimentos da abordagem. Adicionalmente, obteve-se *feedback* dos participantes do grupo focal sobre a clareza das informações apresentadas, não sendo constatado qualquer problema desta natureza.

*Validade interna*: Uma ameaça está associada ao feito de seleção, onde os participantes podem não ser representativos da população-alvo. Para mitigar os efeitos desta ameaça, buscou-se selecionar diferentes perfis de participantes que sejam representativos em relação à sua experiência em desenvolvimento de software e ao uso de abordagens ágeis. Outra ameaça tem associação ao efeito de interação, onde as discussões entre os participantes podem influenciar suas respostas, gerando um efeito de contágio. Para minimizar o efeito desta ameaça, realizou-se o registro das respostas dos participantes de forma individual, sem que os outros participantes tenham acesso, para minimizar a influência de respostas anteriores na resposta dos participantes subsequentes.

*Validade externa*: refere-se à possibilidade de generalizar os resultados para outras populações e contextos. Algumas ameaças à validade externa que podem ocorrer em um grupo focal incluem: a amostra não ser representativa da população-alvo, ou o contexto em que o grupo focal foi realizado não ser generalizável para outras situações. Em relação à amostra, buscou-se recrutar profissionais técnicos com diferentes perfis (e.g., desenvolvedores e gerentes) para assegurar uma variedade de experiências, conhecimentos e perspectivas. Isso contribuiu para enriquecer a discussão, capturando insights mais abrangentes e minimizando o risco de generalizações inadequadas. Em relação ao contexto do grupo focal, buscou-se selecionar um ambiente de discussão que fosse reflexo das dinâmicas de desenvolvimento de software comuns na indústria. Além disso, ao documentar detalhadamente as características do contexto e os procedimentos do grupo focal, pretendeu-se fornecer informações suficientes para que pesquisadores em outros contextos possam avaliar a relevância e aplicabilidade dos resultados obtidos.

*Validade de conclusão*: Uma possível ameaça está associada à interpretação subjetiva dos dados. Para minimizar o efeito desta ameaça, foram utilizadas diferentes fontes de dados (como áudios e anotações) para garantir a integridade dos dados coletados e realizar análises triangulares para confirmar os resultados obtidos. Além disso, houve a participação de mais

de um pesquisador na análise dos dados para minimizar a influência de vieses individuais.

## 9.5 Validação *Online* da Abordagem DI Integrada ao Processo Ágil

Após a apresentação da validação *offline* da proposta de abordagem para desenvolvimento de software que integra a abordagem DI numa instanciação do arcabouço *Scrum*, a presente seção tem como objetivo realizar uma validação *online* desta abordagem por meio de um experimento controlado. No que segue, serão exibidos detalhes sobre o planejamento (Seção 9.5.1) e execução (Seção 9.5.2) deste experimento. Em seguida, serão apresentados os resultados obtidos a partir da aplicação do experimento (Seção 9.5.3), bem como as ameaças à validade inerentes a realização deste estudo (Seção 9.5.4).

### 9.5.1 Planejamento do Experimento

Para projetar o experimento controlado, utilizou-se um conjunto de recomendações descritas nos trabalhos de Wohlin *et. al* [125] e Jedlitschka *et. al* [47]. Os participantes executaram tarefas relacionadas à detecção de anomalias de código com suporte da abordagem para desenvolvimento de software que integra DI numa instanciação do arcabouço *Scrum*. Todas as tarefas foram realizadas mediante suporte das abordagens DI e DNI embutidas na ferramenta *Eclipse ConCAD* (Apêndice D).

Em relação aos participantes deste estudo, foram recrutados 24 estudantes de graduação para realização das atividades experimentais. É relevante observar que esses participantes diferiam daqueles envolvidos nos estudos delineados nos Capítulos 7 e 8. Todos os participantes eram do curso de Engenharia de Computação e cursavam uma disciplina de Programação para *Web*. Estes foram recrutados por meio de uma mensagem de *e-mail* com base no interesse explícito e voluntário em participar do experimento. Era esperado que os participantes estivessem pelo menos moderadamente familiarizados com a linguagem Java e *Spring Framework*. No entanto, não esperava-se o amplo conhecimento dos participantes sobre anomalias de código ou técnicas de detecção usadas no experimento. Todos os participantes deste estudo participaram de uma sessão de treinamento para nivelar o conhecimento

sobre anomalias de código, abordagens de detecção, refatoração e utilização do ambiente de desenvolvimento integrado no contexto das atividades experimentais.

Os dados coletados durante o experimento, tanto os aspectos censitários dos participantes quanto às avaliações do código e os *feedbacks* obtidos, foram analisados de forma estatística e qualitativa para verificar a eficácia da abordagem proposta. Para realizar a análise, foram utilizados recursos da ferramenta R [53]. Os testes utilizados foram o *Wilcoxon signed-rank test* [55], aplicado aos valores das instâncias de anomalias detectadas. Esse teste foi escolhido devido aos dados não seguirem uma distribuição normalizada. Além disso, foi utilizado o *Paired T-Test* [55], aplicado aos valores das medidas de *recall* e *precision*, que seguem uma distribuição normalizada.

### 9.5.2 Execução do Experimento

As equipes de participantes realizaram as tarefas experimentais associadas ao desenvolvimento de um projeto de software utilizando a linguagem Java. No contexto desse estudo, cada equipe de participantes deveria realizar o desenvolvimento de um sistema *web* para simular um domínio de livraria. Este sistema foi escolhido devido a sua simplicidade bem como a capacidade de ser desenvolvido em um período de tempo adequado à realização do experimento. O protocolo de execução das tarefas experimentais são similares aqueles descritos no estudo do Capítulo 8.

Para a realização das atividades experimentais, as equipes de participantes foram divididas em dois grupos: controle e experimental. As equipes do primeiro grupo realizaram as atividades mediante suporte da proposta de abordagem para desenvolvimento de software que integra DI numa instanciação do arcabouço *Scrum*. As equipes do segundo grupo realizaram as atividades de desenvolvimento apenas com suporte do arcabouço *Scrum*. As equipes foram balanceadas em termos de tempo de experiência e formação para evitar qualquer tipo de viés nos resultados obtidos a partir das atividades experimentais. Assim, cada uma das 8 equipes deverá ter 3 estudantes que foram escolhidos aleatoriamente para realização das atividades.

Para avaliação dos resultados associados às atividades experimentais foi necessário obter um "oráculo" representando a lista de anomalias de código que realmente representam problemas de manutenibilidade no sistema. Para a geração do oráculo foram realizadas os

mesmos procedimentos descritos no Capítulo 8. É digno afirmar que foi necessária a geração de 8 oráculos distintos ao final de cada entrega do experimento.

O ambiente de execução das tarefas experimentais, incluindo os arquivos e suporte ferramental, foi disponibilizado às equipes participantes. Além disso, as tarefas experimentais foram supervisionadas por dois pesquisadores. O tempo total de realização do experimento foi de 45 dias, sendo realizadas duas entregas durante esse período (i.e., uma entrega ao final da segunda e da terceira *Sprint*). Para cada entrega foi elencado um número de requisitos e funcionalidades previstas. Em linhas gerais, o experimento foi organizado em três fases, conforme descrição a seguir.

**Fase 1: Pré-experimento**. Essa fase teve como propósito preparar os participantes para as atividades subsequentes do experimento, garantindo que todos tivessem o conhecimento necessário sobre os tipos de anomalias a serem identificadas e familiaridade com as abordagens de DI e DNI fornecidas pela ferramenta *Eclipse ConCAD*. Inicialmente, todas as equipes participantes receberam um material contendo a definição de 10 tipos de anomalias suportadas pela ferramenta *Eclipse ConCAD*. Junto com as definições, foram fornecidos exemplos ilustrativos de cada tipo de anomalia. Um prazo máximo de 20 minutos foi estabelecido para que os participantes pudessem compreender essas definições. Em seguida, os participantes passaram por um treinamento detalhado sobre as abordagens de Detecção de Anomalias de Código presentes na ferramenta *Eclipse ConCAD* bem como na proposta de abordagem para desenvolvimento de software que integra DI numa instanciação do arcabouço *Scrum*. Por fim, todos os participantes foram convidados a preencher um questionário com o objetivo principal de identificação do perfil. O questionário visava coletar informações relevantes, como experiência em desenvolvimento de software, conhecimento prévio sobre detecção de anomalias de código e familiaridade com a IDE Eclipse.

**Fase 2: Desenvolvimento de Software e Detecção de Anomalias**. Nesta fase, os 24 participantes foram distribuídos em 8 equipes com intuito de realizar o desenvolvimento de um sistema web utilizando a linguagem Java. Além da implementação das funcionalidades, os participantes também deveriam identificar e catalogar a presença de 10 tipos distintos de anomalias de código ao longo do experimento. Para isso, metade das equipes participantes fez uso da proposta de abordagem para desenvolvimento de software que integra DI numa instanciação do arcabouço *Scrum* durante as atividades experimentais, enquanto que a outra

metade fez uso apenas do arcabouço *Scrum*, sem no entanto utilizar abordagem DI de forma disciplinada. As equipes deveriam realizar o desenvolvimento do sistema web ao longo de 3 *Sprints* de 15 dias cada. Ao final da segunda e terceira *Sprint* cada equipe participante deveria prover uma lista contendo informações sobre o número total (T) de anomalias de código detectadas (AD), verdadeiros positivos (VP), falsos positivos (FP) e falsos negativos (FN). Os dados contidos nestas listas foram utilizados posteriormente para calcular as medidas de *recall* e *precision*. Por fim, para evitar que o efeito de aprendizagem introduzisse qualquer tipo de viés, a análise dos resultados das detecções foi realizada somente após a conclusão do experimento. Isso garantiu que as equipes participantes não fossem influenciadas pelos resultados durante o processo de detecção, mantendo a integridade e a imparcialidade dos dados coletados.

**Fase 3: Pós-experimento**. Nesta fase, os participantes foram convidados a preencher um questionário de *feedback*, com o intuito de compartilhar suas percepções sobre o uso da adaptação do *Scrum* com a abordagem DI. Especificamente, os participantes foram solicitados a destacar os benefícios e desafios encontrados no contexto das atividades experimentais. Eles foram encorajados a relatar os aspectos positivos observados, como ganhos de produtividade, melhoria na qualidade do código, facilitação da colaboração entre as equipes e aperfeiçoamento das práticas ágeis. Além disso, foram convidados a mencionar quaisquer desafios enfrentados durante a aplicação da abordagem, como dificuldades na detecção de anomalias, resistência à mudança ou problemas de integração entre as atividades do *Scrum* e DI.

### 9.5.3 Análise dos Dados do Experimento

Nesta seção são apresentados os principais resultados associados a validação *online* da proposta de abordagem para desenvolvimento de software que integra DI numa instanciação do arcabouço *Scrum*. No que segue, apresenta-se os resultados associados aos participantes do experimento. Em seguida, descrevem-se os principais resultados associados às atividades experimentais. Por fim, discute-se as percepções dos participantes sobre o uso da adaptação do *Scrum* com a abordagem DI.

### Atividades Pré-Experimento (Fase 1)

Mediante realização da Fase 1 do experimento, obteve-se dados para definição do perfil dos participantes do experimento. Os participantes eram compostos por pessoas do sexo masculino (80%) e feminino (20%). Em relação à faixa etária, 60% dos participantes tinham entre 19 e 21 anos. Em relação à experiência prévia em desenvolvimento de software, observou-se que 25% dos participantes não possuíam conhecimento prévio. 33% dos participantes tinham pouca experiência (1-2 semestres) e 29% dos participantes possuíam alguma experiência (3-4 semestres). Importante mencionar que apenas 13% dos participantes possuíam experiência significativa (5 ou mais semestres).

Em relação à experiência prévia em detecção de anomalias, 42% dos participantes não possuíam conhecimento prévio, enquanto que 29% dos participantes tinham pouca experiência (1-2 semestres). Apenas 29% dos participantes possuíam alguma experiência (3-4 semestres). Relacionado a utilização da IDE Eclipse, a maioria dos participantes (75%) estava familiarizada com o uso desta IDE, enquanto 25% dos participantes não tinham experiência com essa ferramenta específica. Em relação ao conhecimento em Java, 8% dos participantes não possuíam conhecimento prévio e 42% tinham conhecimento básico. Cerca de 50% dos participantes possuíam conhecimento intermediário ou avançado na linguagem.

Essas informações sobre o perfil dos participantes são essenciais para compreender a diversidade e experiência dos indivíduos envolvidos no experimento. Esses dados fornecem uma visão abrangente, possibilitando análises mais aprofundadas sobre como esses aspectos podem influenciar os resultados e as percepções dos participantes durante o estudo. No geral, o perfil dos participantes atende às expectativas do estudo uma vez que eles possuem conhecimento intermediário em Java, conhecem o ambiente de desenvolvimento integrado utilizado no experimento além das atividades de detecção de anomalias de código e refatoração fazerem parte do seu vocabulário ou rotina de trabalho.

### Detecção de Anomalias de Código (Fase 2)

A Fase 2 do experimento consistiu em realizar tarefas relacionadas ao desenvolvimento de software mediante suporte da proposta de abordagem que integra DI numa instanciação do arcabouço *Scrum*. O objetivo desta atividade era avaliar a eficácia e a viabilidade prática da

abordagem em termos de produtividade, qualidade do código, colaboração entre as equipes e adequação às práticas ágeis. Além disso, buscava-se verificar se a utilização da abordagem resultaria em uma detecção mais precisa e abrangente de anomalias de código em comparação com o grupo de controle que utilizou apenas o arcabouço *Scrum*.

Na realização das atividades experimentais, os participantes foram instruídos a fazer duas entregas ao final da segunda e terceira *Sprint*. Cada entrega envolvia dois artefatos: (i) o projeto em linguagem Java com a implementação dos requisitos e funcionalidades especificados, e (ii) uma lista contendo 10 tipos diferentes de anomalias de código que as equipes participantes identificaram como problemas de manutenibilidade. Para cada projeto entregue, foi gerado um oráculo contendo uma lista de anomalias de código que verdadeiramente representavam problemas de manutenibilidade. Ao comparar esses artefatos (i.e., o oráculo *versus* a lista de anomalias), foi possível obter os resultados relacionados a Verdadeiros Positivos (VP), Falsos Positivos (FP) e Falsos Negativos (FN). Os resultados correspondentes a cada um desses componentes estão descritos na Tabela 9.3.

Tabela 9.3: Resultados da Detecção de Anomalias.

| | **1a Entrega** | | | **2a Entrega** | | |
|---|---|---|---|---|---|---|
| | **Scrum + DI (Grupo Experimental)** | | | | | |
| | **VP** | **FP** | **FN** | **VP** | **FP** | **FN** |
| Equipe 1 | 11 | 3 | 6 | 9 | 3 | 4 |
| Equipe 2 | 12 | 2 | 7 | 8 | 2 | 4 |
| Equipe 3 | 13 | 3 | 6 | 9 | 2 | 5 |
| Equipe 4 | 15 | 4 | 7 | 8 | 2 | 4 |
| **Total** | **51** | **12** | **26** | **34** | **9** | **17** |
| **Média** | **12,7** | **3** | **6,5** | **8,5** | **2,3** | **4,25** |
| | **Scrum (Grupo Controle)** | | | | | |
| Equipe 5 | 14 | 5 | 9 | 11 | 5 | 7 |
| Equipe 6 | 15 | 6 | 9 | 12 | 5 | 8 |
| Equipe 7 | 16 | 6 | 9 | 14 | 5 | 9 |
| Equipe 8 | 15 | 7 | 11 | 14 | 6 | 9 |
| **Total** | **60** | **24** | **38** | **51** | **21** | **33** |
| **Média** | **15** | **6** | **9,5** | **12,5** | **5,3** | **8,3** |

**Abordagem Scrum + DI diminui AD e VP**

Em relação aos *resultados de Anomalias Detectadas (AD)*, as equipes participantes que utilizaram a abordagem identificaram um total de 63 (1ª entrega) e 43 (2ª entrega) anomalias, enquanto aqueles que não utilizaram a abordagem encontraram um total de 84 (1ª entrega) e 72 (2ª entrega) anomalias. Os resultados indicam que equipes com suporte da abordagem conseguiram diminuir em cerca de 35% do número AD.

Em relação aos *resultados de Verdadeiros Positivos (VP)*, notou-se decremento similar. Equipes participantes usando a abordagem detectaram 51 (1ª entrega) e 34 (2ª entrega), enquanto que sem o uso da abordagem detectaram 60 (1ª entrega) e 51 (2ª entrega). Portanto, o uso da abordagem contribui para redução de cerca de 25% no número total de VP (i.e., anomalias de código remanescentes) na atividade de detecção de anomalias de código. Finalmente, os resultados estatísticos relacionados a AD (*alfa* = 0,05, p = 0,00027, W = 3, MD = 5,71, Z = -2.359) e VP (*alfa* = 0,05, p = 0,0061, W = 0, MD = 4,77, Z = -2,3664) foram estatisticamente significantes através do uso do teste de Wilcoxon [55].

A abordagem *Scrum* com DI contribui para a redução de anomalias detectadas (AD) e verdadeiros positivos (VP) devido a diversos fatores. Um possível motivo é que a abordagem *Scrum* com DI incentiva uma abordagem proativa para lidar com anomalias, colocando ênfase na prevenção e detecção precoce. Isso significa que a equipe está mais preparada para evitar a introdução de anomalias e, caso elas ocorram, corrigi-las prontamente. Essa mentalidade de prevenção contribui para a diminuição do número de anomalias detectadas e dos verdadeiros positivos. Outra possível razão para o decremento de AD e VP é que o *Scrum* promove um processo de desenvolvimento iterativo e incremental, com foco na entrega contínua de software funcional. Isso permite que as equipes identifiquem e corrijam anomalias de forma mais ágil, à medida que avançam no desenvolvimento. A integração da abordagem DI no *Scrum* reforça a atenção à qualidade do código e o uso de práticas que ajudam a prevenir a introdução de anomalias.

**Abordagem Scrum + DI diminui FP e FN**

Em relação aos *resultados de Falsos Positivos (FP)*, verificou-se que os participantes identificaram 12 (na primeira entrega) e 9 (na segunda entrega) utilizando a abordagem. Por outro

lado, ao utilizar a abordagem DI, os participantes identificaram 24 (na primeira entrega) e 21 (na segunda entrega). Conclui-se, portanto, que a utilização da abordagem pode reduzir em até 50% o número de FP.

Existem algumas razões pelas quais a abordagem DI no *Scrum* pode contribuir para a redução de FP. Primeiramente, a abordagem DI enfatiza a inspeção do código desde as fases iniciais do desenvolvimento. Isso significa que os desenvolvedores são incentivados a analisar e corrigir as anomalias de código assim que são identificadas, antes mesmo de elas se propagarem e se tornarem mais complexas. Essa prática resulta em uma detecção precoce de problemas e evita que falsos alarmes se acumulem. Além disso, a abordagem DI no *Scrum* envolve uma colaboração estreita entre os membros da equipe, incluindo desenvolvedores, testadores e gerentes de projeto. Durante as inspeções de código, todos os membros têm a oportunidade de revisar e discutir as anomalias encontradas. Esse processo de revisão conjunta permite uma visão mais abrangente e uma melhor compreensão do contexto do código, reduzindo a probabilidade de identificar falsos problemas. Por fim, a abordagem DI no *Scrum* promove uma mentalidade de melhoria contínua e aprendizado. À medida que os desenvolvedores se familiarizam com as técnicas de inspeção de código e aprimoram suas habilidades, eles se tornam mais proficientes em distinguir entre anomalias legítimas e situações que não exigem intervenção imediata. Esse conhecimento acumulado e o aprimoramento constante contribuem para uma redução gradual dos FP ao longo do tempo.

Em relação aos *resultados de Falsos Negativos (FN)*, as equipes participantes identificaram 26 (1ª entrega) e 17 (1ª entrega) com suporte da abordagem, enquanto as equipes participantes sem suporte da abordagem conseguiram 38 (1ª entrega) e 33 (2ª entrega). Isso denota que o uso da abordagem DI pode reduzir o número de FN em mais de 30%. É importante mencionar que os resultados estatísticos relacionados a FP (*alfa* = 0,05, p = 0,00032, W = 0, MD = -3,38, Z = -3,407) e FN (*alfa* = 0,05, p = 0,00389, W = 0, MD = -3,62, Z = -2,520) obtidos pelos grupos do experimento foram significantes mediante uso do teste de Wilcoxon [55].

Existem algumas razões que explicam essa redução de FN com o uso da abordagem DI no contexto do *Scrum*. Em primeiro lugar, a abordagem DI enfatiza a inspeção de código e a detecção precoce de anomalias. Ao adotar práticas de inspeção contínua e revisões colaborativas, as equipes conseguem identificar um maior número de anomalias que repre-

sentam problemas de manutenção. Isso permite que essas anomalias sejam corrigidas antes que se tornem FN, ou seja, problemas que passariam despercebidos sem a intervenção da abordagem DI. Além disso, a abordagem DI promove uma mentalidade de responsabilidade compartilhada pela qualidade do código. No *Scrum*, as equipes são encorajadas a se apropriarem da qualidade do software e a buscar a excelência no desenvolvimento. Ao utilizar a abordagem DI, os desenvolvedores são mais conscientes da importância de identificar e resolver as anomalias de código que representam problemas de manutenção. Isso contribui para a redução dos FN, pois os desenvolvedores estão mais atentos e comprometidos em garantir a qualidade do código.

### Avaliação das medidas de eficácia

Para fornecer uma perspectiva adicional sobre a utilização da proposta de abordagem para desenvolvimento de software que integra DI numa instanciação do arcabouço *Scrum*, foram calculadas as medidas de eficácia *Precision* (P) e *Recall* (R) a partir do uso dos componentes descritos na Tabela 9.3. Os resultados associados a essas medidas podem ser vistos na Tabela 9.4.

Tabela 9.4: Resultados das Medidas *Precision* e *Recall*.

| | **1a Entrega** | | **2a Entrega** | |
|---|---|---|---|---|
| | **Scrum + DI (Gp. Exp.)** | | | |
| | **P** | **R** | **P** | **R** |
| Equipe 1 | 0,79 | 0,65 | 0,75 | 0,69 |
| Equipe 2 | 0,86 | 0,64 | 0,80 | 0,67 |
| Equipe 3 | 0,81 | 0,68 | 0,82 | 0,64 |
| Equipe 4 | 0,79 | 0,68 | 0,80 | 0,67 |
| **Média** | **0,81** | **0,67** | **0,80** | **0,67** |
| | **Scrum (Gp. Cont.)** | | | |
| Equipe 5 | 0,74 | 0,61 | 0,69 | 0,61 |
| Equipe 6 | 0,71 | 0,63 | 0,71 | 0,60 |
| Equipe 7 | 0,73 | 0,64 | 0,74 | 0,61 |
| Equipe 8 | 0,68 | 0,58 | 0,70 | 0,60 |
| **Média** | **0,72** | **0,60** | **0,70** | **0,60** |

**Abordagem DI aumenta a *precision*.** Com base nos resultados associados a TP e VP,

esperava-se que o uso da abordagem contribuísse para um incremento da medida *precision* na detecção de anomalias. O valor médio desta medida com uso da abordagem foi de 0,81 (1ª entrega) e 0,80 (2ª entrega), enquanto as equipes participantes sem uso da abordagem atingiram 0,72 (1ª entrega) e 0,70 (2ª entrega). A diferença nos resultados da medida *precision* foi em média de até 15% em favor do uso da abordagem. Os resultados da medida *precision* obtidos pelos grupos do experimento foram estatisticamente significantes (*alfa* = 0,05, p = 0,003087, SD = 5,1121, t = -3.5211, tamanho do efeito = 0,88) - usando o *Paired T-Test* [55]).

**Abordagem DI aumenta o *recall***. Com base nos resultados associados a FN e VP, esperava-se que o uso da abordagem DI contribuísse para melhorar o *recall* na detecção de anomalias. Em média, observou-se que as equipes participantes que utilizaram a abordagem obtiveram 0,67 (1ª entrega) e 0,67 (2ª entrega), enquanto que as equipes participantes sem utilização da abordagem atingiram 0,60 (1ª entrega) e 0,60 (2ª análise), representando uma diferença de até aproximadamente 15% a favor da abordagem. Finalmente, os resultados relacionados à *recall* obtidos pelos grupos do experimento foram estatisticamente significantes (*alfa* = 0,05, p = 0,0008386, SD = 3,4617, t = -3,0332, tamanho do efeito = 0,76), usando *Paired T-Test* [55]).

Existem alguns fatores que podem explicar esse aumento na medida de *precision* e *recall* com o uso da abordagem DI no contexto do *Scrum*. Primeiramente, a abordagem DI enfatiza a detecção e correção precoce de anomalias de código. Ao adotar práticas como inspeções contínuas e revisões colaborativas, as equipes têm a oportunidade de identificar e corrigir erros e problemas de manutenção antes que eles sejam propagados para o produto final. Isso contribui para a redução de anomalias e melhora a precisão do código entregue. Além disso, a abordagem DI promove uma cultura de qualidade e excelência no desenvolvimento de software. No *Scrum*, as equipes são incentivadas a assumir a responsabilidade pela qualidade do código e a buscar constantemente melhorias. Ao utilizar a abordagem DI, os desenvolvedores adotam uma postura mais proativa na identificação e correção de anomalias, resultando em um código de maior qualidade e, consequentemente, em uma medidas de eficácia mais elevadas.

**Atividades Pós-Experimento (Fase 3)**

Mediante realização da Fase 3 do experimento, os participantes foram convidados a preencher um questionário para compartilhar suas percepções sobre a proposta de abordagem para desenvolvimento de software que integra DI numa instanciação do arcabouço *Scrum*. A seguir, são apresentados os resultados quantitativos e trechos qualitativos das respostas dos 24 participantes do experimento controlado.

**Percepção Geral**. A primeira parte do questionário buscou captar a percepção geral dos participantes em relação ao uso da adaptação do *Scrum* com a abordagem DI. A referida abordagem foi avaliada positivamente pela maioria dos participantes, com 83% dos participantes relatando benefícios significativos em relação à qualidade do código e agilidade no desenvolvimento do sistema. No que segue, transcreve-se trechos do questionário de alguns Participantes (P):

1. P1 - *"...A adaptação do Scrum com a DI trouxe uma mudança positiva para o processo de desenvolvimento. Conseguimos identificar e corrigir anomalias de código de forma mais rápida, o que resultou em um código mais limpo e de melhor qualidade. Além disso, a abordagem ágil do Scrum nos ajudou a manter o ritmo de trabalho e entregar as funcionalidades dentro dos prazos estabelecidos. Foi uma experiência muito produtiva..."*.

2. P2 - *"...Eu gostei da combinação do Scrum com a DI. A detecção precoce de anomalias nos permitiu corrigi-las rapidamente, evitando que se tornassem problemas maiores no futuro. Além disso, revisar e refatorar constantemente o código nos ajudou a manter um código mais limpo e organizado. Isso resultou em um desenvolvimento mais eficiente e em um sistema com menos bugs..."*.

3. P3 - *"...A adaptação do Scrum com a DI trouxe uma mudança significativa na qualidade do código que produzimos. Conseguimos identificar e resolver problemas mais cedo, o que nos permitiu entregar um sistema mais estável e confiável. Além disso, a abordagem ágil do Scrum nos ajudou a priorizar as tarefas mais importantes. Estou satisfeito com os benefícios que essa combinação trouxe ao trabalho..."*.

No entanto, 17% dos participantes mencionaram que enfrentaram desafios relacionados

à necessidade de aprendizado e adaptação às novas práticas. Dentre as respostas qualitativas, alguns participantes destacaram:

- P1 - *"...No início, tive dificuldade em me adaptar às práticas do Scrum e da DI. Foi necessário um período de aprendizado e familiarização com as novas abordagens. Porém, conforme o tempo passou, pude perceber os benefícios dessas práticas e os resultados positivos que elas trouxeram ao nosso trabalho..."*.

- P2 - *"...A introdução do Scrum e da DI trouxe um novo conjunto de práticas que precisávamos aprender e aplicar. No começo, senti-me sobrecarregado com tantas informações novas, e demorou algum tempo para me sentir confortável com essas abordagens. No entanto, com o apoio da equipe e o esforço contínuo, conseguimos aproveitar os benefícios dessas práticas..."*.

- P3 - *"...Como alguém com menos experiência em desenvolvimento ágil, enfrentei alguns desafios ao adotar o Scrum e a DI. A necessidade de aprender novas práticas e adaptar minha forma de trabalhar demandou tempo e esforço. Houve momentos em que me senti um pouco perdido, mas com o suporte da equipe consegui superar esses desafios e me tornar mais proficientes nessas abordagens..."*

**Scrum**. Na segunda parte do questionário, os participantes foram convidados a avaliar as adaptações realizadas no *framework Scrum* durante o experimento.

Em relação a adaptação de papéis, questionou-se se os participantes concordavam com a criação do novo papel de "avaliador de qualidade" no arcabouço do *Scrum*. Cerca de 60% dos participantes concordaram totalmente enquanto que cerca de 30% concordaram parcialmente com a propositura. Em linhas gerais, a proposta foi bem recebida e os participantes veem valor na inclusão desse novo papel para melhorar a qualidade do processo de desenvolvimento e do código.

Questionou-se ainda se os participantes acreditavam que outras adaptações de papéis são necessárias além do novo papel de "avaliador de qualidade". Mais de 50% dos participantes discordaram totalmente enquanto que outros 30% discordaram parcialmente desta afirmação. Os resultados revelam que a maioria dos participantes discordou da necessidade de outras adaptações de papéis além do novo papel de "avaliador de qualidade". Isso pode indicar que

os participantes consideram a proposta do novo papel suficiente para melhorar o processo de desenvolvimento, e outras mudanças podem não ser necessárias ou até mesmo prejudiciais.

Em relação a adaptação dos artefatos, questionou-se aos participantes sobre a inclusão da "Lista Global de anomalias no *Backlog* do Produto" como um novo artefato. Mais de 80% dos respondentes concordaram totalmente ou parcialmente com esta propositura. Isso sugere que os participantes reconhecem o valor que esse artefato pode agregar ao processo de desenvolvimento, especialmente em termos de gestão das anomalias de código. A inclusão desse novo artefato pode fornecer uma visão mais clara e detalhada das atividades de detecção e refatoração de anomalias, facilitando a comunicação e colaboração entre os envolvidos nessas atividades.

Questionou-se ainda se os participantes acreditavam que outras adaptações de artefatos são necessárias além da inclusão da "Lista Global de anomalias no *Backlog* do Produto". Os resultados revelam que a maioria dos participantes discordou (62%) ou ficou neutra (29%) quanto à necessidade de outras adaptações de artefatos além da inclusão da "Lista Global". Esses resultados sugerem que os participantes veem a inclusão deste artefato como uma mudança significativa o suficiente para melhorar o processo de desenvolvimento. Além disso, eles acreditam que outras mudanças podem ser desnecessárias ou até mesmo prejudiciais, pois podem introduzir mais complexidade ao processo sem trazer benefícios aparentes.

Em relação a adaptação dos eventos, questionou-se aos participantes sobre a concordância sobre as adaptações realizadas nos eventos do *Scrum* (i.e., Planejamento da *Sprint*, *Sprint*, Reunião diária e Revisão da *Sprint*). Os resultados indicam que 45% dos participantes concordaram totalmente com a adaptação dos eventos, enquanto 32% concordaram parcialmente. Isso sugere que a maioria dos participantes vê a adaptação dos eventos como positiva, embora nem todos concordem completamente.

Questionou-se ainda se os participantes acreditavam que outras adaptações nos eventos seriam necessárias além das propostas na abordagem. Os resultados revelam que a maioria dos participantes discordou (42%) ou ficou neutra (29%) quanto à necessidade de mais adaptações nos eventos além das propostas. Isso sugere que os participantes consideram as adaptações propostas suficientes para melhorar o processo de desenvolvimento e acredita que outras mudanças podem ser desnecessárias ou mesmo prejudiciais.

**Abordagem DI**. A terceira parte do questionário teve como foco a avaliação da aborda-

gem de Detecção Interativa (DI) utilizada durante as atividades experimentais. Para avaliar se a abordagem DI facilitou a identificação e compreensão das instâncias de anomalias de código, os participantes foram convidados a expressar sua concordância. As respostas foram variadas e podem ser resumidas da seguinte forma:

- 55% concordaram totalmente que a DI facilitou a identificação e compreensão das instâncias de anomalias de código. Eles relataram que a técnica forneceu uma abordagem mais eficaz para detectar e compreender as anomalias, permitindo uma análise mais aprofundada e uma visão mais clara do código problemático;
- 33% concordaram parcialmente, indicando que a DI trouxe algum benefício na identificação e compreensão de anomalias, mas também enfrentaram algumas dificuldades específicas durante o processo. Esses participantes destacaram que, apesar de alguns desafios, a técnica ainda se mostrou útil em melhorar a detecção de anomalias no código;
- 12% discordaram, afirmando que a DI não facilitou a identificação e compreensão das instâncias de anomalias de código. Esses participantes mencionaram que a abordagem não foi adequada para suas necessidades específicas ou que não notaram uma melhoria significativa em relação a métodos tradicionais de detecção de anomalias.

Em relação à influência da aplicação da abordagem DI na qualidade do código produzido, os participantes expressaram suas opiniões. Os resultados foram os seguintes:

- 67% concordaram totalmente que a aplicação da DI influenciou positivamente a qualidade do código produzido. Eles mencionaram que a técnica ajudou a identificar e corrigir problemas de código de forma mais eficiente, resultando em um código de melhor qualidade no final do processo de desenvolvimento;
- 25% concordaram parcialmente, reconhecendo que a DI teve algum impacto positivo na qualidade do código produzido, mas também observaram que outros fatores, como a experiência da equipe e o contexto do projeto, desempenharam um papel importante na qualidade geral do código;

- 8% discordaram, afirmando que a aplicação da DI não influenciou positivamente a qualidade do código produzido. Eles apontaram que outros métodos ou práticas existentes já estavam em vigor para garantir a qualidade do código, e a DI não trouxe melhorias adicionais nesse aspecto.

Em seguida, os participantes foram convidados a compartilhar os principais benefícios observados com o uso da proposta de abordagem para desenvolvimento de software que integra DI numa instanciação do arcabouço *Scrum*. As respostas dos participantes podem ser agrupadas da seguinte maneira:

- *Melhor identificação de anomalias*: 62% dos participantes relataram que a DI proporcionou uma melhor identificação das instâncias de anomalias de código. Eles mencionaram que a abordagem interativa permitiu uma detecção mais precisa e eficiente de problemas no código;

- *Compreensão aprofundada do código*: 54% destacaram que a DI auxiliou na compreensão mais aprofundada do código. Eles perceberam que a análise interativa permitiu uma visão mais detalhada das estruturas e relacionamentos do código, o que facilitou a detecção de problemas e a tomada de decisões de refatoração;

- *Detecção precoce de anomalias*: 45% dos participantes destacaram que a DI permitiu a detecção precoce de anomalias no código. Eles observaram que a abordagem interativa possibilitou identificar problemas em estágios iniciais do desenvolvimento, o que contribuiu para evitar que esses problemas se propagassem e se tornassem mais complexos;

- *Melhor colaboração e comunicação*: 38% mencionaram que a DI melhorou a colaboração e a comunicação entre os membros da equipe. Eles notaram que a capacidade de visualizar e discutir as instâncias de anomalias durante o processo de desenvolvimento ajudou a alinhar a compreensão dos problemas e facilitou a colaboração na resolução dos mesmos.

Finalmente, os participantes também compartilharam os desafios e dificuldades encontrados ao aplicar a abordagem DI durante o processo de desenvolvimento. Estas respostas podem ser resumidas da seguinte maneira:

- *Curva de aprendizado*: 58% dos participantes mencionaram que enfrentaram uma curva de aprendizado inicial ao adotar a DI. Eles relataram que leva tempo para se familiarizar com a abordagem e suas ferramentas, o que pode dificultar a aplicação efetiva da técnica;

- *Mudança cultural e resistência*: 50% dos participantes mencionaram que enfrentaram resistência à adoção da DI devido a uma cultura organizacional enraizada em métodos tradicionais de desenvolvimento. Eles apontaram que a mudança cultural e a necessidade de conscientização foram desafios a serem superados;

- *Sobrecarga cognitiva*: 38% dos participantes destacaram que a aplicação da DI pode levar a uma sobrecarga cognitiva. Eles mencionaram que a necessidade de acompanhar as instâncias de anomalias em tempo real durante a codificação pode ser exigente mentalmente, especialmente em projetos complexos;

- *Integração com ferramentas existentes*: 30% dos participantes relataram desafios ao integrar a DI com as ferramentas e processos de desenvolvimento já em uso. Eles observaram que a adoção da técnica exigiu ajustes e integração com as ferramentas existentes, o que pode demandar esforço adicional e enfrentar limitações técnicas.

**Considerações Finais**. Os participantes foram questionados se a aplicação da adaptação do *Scrum* para abrigar DI poderia contribuir para a comunicação e colaboração entre os membros da equipe. No que segue, transcreve-se trechos do questionário de alguns Participantes (P):

- P1 - *"...No Scrum com a detecção interativa, a comunicação entre as equipes é priorizada e valorizada, o que facilita a troca de informações e o trabalho colaborativo..."*.

- P2 - *"...A abordagem Scrum com a detecção interativa promove reuniões frequentes e interações constantes entre os membros da equipe, permitindo que todos estejam alinhados e engajados nos objetivos do projeto..."*.

- P3 - *"...Com o uso da detecção interativa no Scrum, notamos uma melhoria significativa na comunicação entre os desenvolvedores, pois somos encorajados a compartilhar conhecimento e experiências para resolver problemas em conjunto..."*.

- P4 - *"...Através da abordagem Scrum com a detecção interativa, temos um ambiente propício para a colaboração, onde podemos trabalhar em equipe, debater ideias e encontrar soluções de forma colaborativa e eficiente...".*

- P5 - *"...No Scrum com a detecção interativa, as equipes são incentivadas a se comunicarem de maneira clara e aberta, o que contribui para uma melhor compreensão das tarefas e minimiza possíveis ruídos na comunicação...".*

Por fim, os participantes foram convidados a compartilhar sua recomendação sobre a utilização da abordagem DI na proposta para outras equipes de desenvolvimento. Suas respostas podem ser resumidas da seguinte maneira:

- Recomendação positiva: 79% dos participantes recomendaram positivamente a utilização da abordagem para outras equipes de desenvolvimento. Eles destacaram os benefícios observados, como a facilidade de identificação e compreensão das instâncias de anomalias de código, a influência positiva na qualidade do código produzido e a melhoria na colaboração e comunicação entre os membros da equipe. Esses participantes acreditam que a aplicação disciplinada da abordagem pode trazer valor e melhorias significativas para o desenvolvimento;

- Recomendação neutra: 13% dos participantes expressaram uma recomendação neutra em relação à utilização da abordagem. Eles observaram que a eficácia da técnica pode depender do contexto específico de cada equipe e projeto. Esses participantes sugerem que equipes interessadas em adotar a abordagem devem avaliar cuidadosamente sua adequação às suas necessidades e considerar fatores como a familiaridade da equipe com a DI e a disponibilidade de recursos necessários;

- Recomendação negativa: menos de 8% dos participantes expressaram uma recomendação negativa em relação à utilização da abordagem. Eles citaram desafios e dificuldades enfrentados durante a aplicação da DI, como curva de aprendizado e sobrecarga cognitiva. Esses participantes acreditam que a DI pode não ser adequada para todas as equipes de desenvolvimento, especialmente aquelas com restrições de recursos ou com uma cultura organizacional menos propensa à adoção de abordagens interativas.

### 9.5.4 Ameaças à Validade - Experimento

Nesta seção apresenta-se as principais ameaças à validade, bem como as ações realizadas no sentido de mitigar seus efeitos. Para tanto, seguiu-se o esquema de classificação proposto por Wohlin et al. [125] conforme itens a seguir.

*Validade de construção*: A experiência prévia e o conhecimento técnico dos participantes podem variar, o que pode influenciar os resultados e a interpretação dos dados. Para mitigar esse efeito, as equipes foram balanceadas em termos de tempo de experiência e formação, a fim de evitar viés nos resultados. Adicionalmente, as equipes participantes podem ter adotado práticas de desenvolvimento diferentes, mesmo dentro do arcabouço *Scrum*. Essas diferenças podem afetar a detecção de anomalias de código e a aplicação da técnica de Detecção Interativa. Para minimizar esse efeito, as equipes receberam orientações claras sobre as práticas de desenvolvimento a serem seguidas durante o experimento, buscando uma uniformidade na abordagem.

*Validade Interna*: Os participantes podem ter sido influenciados pelo seu conhecimento prévio sobre a detecção de anomalias de código, o que pode afetar suas decisões durante o experimento. Para minimizar esse efeito, foi fornecido um treinamento detalhado sobre as abordagens de detecção de anomalias presentes na ferramenta e a proposta de abordagem para desenvolvimento de software que integra DI no *Scrum*, buscando nivelar o conhecimento dos participantes. Outra possível ameaça tem relação com o viés dos pesquisadores na validação das instâncias de anomalias. A análise e validação das instâncias de anomalias de código foram realizadas por pesquisadores especialistas na detecção de anomalias de código com uso da inspeção manual. No entanto, pode haver variações nas interpretações individuais. Para minimizar esse viés, foram adotados critérios claros e objetivos para a validação das instâncias de anomalias, e as análises foram realizadas por dois pesquisadores independentes.

*Validade externa*: O sistema de simulação do domínio da livraria pode não representar completamente um ambiente de desenvolvimento de software real, o que pode afetar a generalização dos resultados para outros contextos. No entanto, foi escolhido devido à sua simplicidade e capacidade de ser desenvolvido no período de tempo adequado ao experimento. Outra ameaça tem relação com a influência do ambiente de execução e supervisão. O ambiente de execução das tarefas experimentais foi disponibilizado às equipes participantes,

e as tarefas foram supervisionadas por pesquisadores. A presença dos pesquisadores pode ter influenciado o comportamento dos participantes. Para mitigar esse efeito, foi enfatizado que as decisões e ações deveriam refletir as práticas reais de desenvolvimento de software.

*Validade de Conclusão*: O estudo foi realizado com 24 participantes, o que pode limitar a generalização dos resultados para uma população maior. No entanto, foram adotados procedimentos metodológicos adequados para garantir a validade interna dos resultados dentro do contexto do experimento. Outra possível limitação tem relação com o nível de experiência dos participantes. É importante mencionar que os estudantes são uma fonte acessível de participantes para experimentos em ambientes educacionais (e.g., universidades e instituições de ensino), facilitando a seleção e recrutamento para o estudo. Os estudantes geralmente estão dispostos a fornecer *feedback* construtivo sobre a experiência e as melhorias percebidas, engajando-se ativamente nas atividades propostas e fornecendo *insights* valiosos para a avaliação da adaptação do *Scrum* com as técnicas DI. Outra limitação tem relação com a linguagem e tecnologia utilizadas. O uso da linguagem Java e do *Spring framework* pode limitar a generalização dos resultados para outros contextos de desenvolvimento de software com diferentes tecnologias. No entanto, a escolha dessas tecnologias foi baseada em sua popularidade e capacidade de fornecer suporte adequado para o desenvolvimento do sistema no contexto do experimento. Por fim, o sistema de simulação do domínio da livraria pode apresentar características específicas que não são encontradas em sistemas de desenvolvimento de software reais. Essas características podem limitar a generalização dos resultados para outros domínios ou contextos de desenvolvimento. No entanto, foi escolhido um sistema de simulação simples e representativo, de modo a minimizar esse efeito.

## 9.6 Considerações Finais do Capítulo

Neste capítulo apresentou-se uma proposta que integra a abordagem DI numa instanciação do arcabouço *Scrum* que foi construída com auxílio de quatro especialistas, sendo dois deles especialistas em *Scrum* e os outros dois especialistas em detecção de anomalias de código. Estes especialistas atuaram de modo colaborativo e iterativo na propositura desta abordagem e na modificação dos artefatos do *Scrum* para torná-lo aderente à presente proposta. Sempre que algum tipo de discordância neste processo ocorresse, os demais especialistas atuavam

no sentido de buscar um consenso nas atividades de construção da presente proposta.

Mediante realização de um grupo focal com 8 profissionais com perfil de desenvolvedores e gerentes de projeto, concluiu-se que a implementação da abordagem DI integrada ao *Scrum* pode trazer benefícios significativos para o desenvolvimento de software, como a melhoria na qualidade de código, a maior facilidade para manutenção, bem como a melhoria na entrega do produto. No entanto, os participantes também apontaram dificuldades em implementar a abordagem, incluindo a falta de conhecimento e habilidades, dificuldades organizacionais e resistência à mudança. Tais dificuldades podem ser superadas com a capacitação e treinamento adequado da equipe, bem como com o apoio da liderança para a adoção da abordagem DI integrada ao *Scrum*. Em geral, os resultados da avaliação indicam que a abordagem DI integrada ao processo ágil pode ser uma estratégia promissora para melhorar o desenvolvimento de software em organizações que adotam o modelo *Scrum*.

Em seguida, mediante realização de um experimento controlado com 24 participantes, concluiu-se que a utilização da proposta de abordagem de desenvolvimento que integra a DI numa instanciação do *Scrum* promoveu uma maior atenção à qualidade do código, detecção precoce de anomalias, e colaboração entre as equipes. Um dos principais achados foi uma elevação de mais de 15% nas medidas de eficácia, especificamente nas métricas de "*recall*"e "*precision*". Isso significa que a abordagem de integração da DI no *Scrum* possibilitou uma detecção mais precisa e abrangente de anomalias de código, garantindo que um número maior de anomalias fosse identificado e corrigido durante o processo de desenvolvimento. Essa detecção precoce contribuiu para evitar a propagação de erros e a ocorrência de problemas mais graves no software. Além disso, a abordagem também facilitou a colaboração entre as equipes envolvidas, promovendo uma melhor comunicação e compartilhamento de conhecimento. A aplicação disciplinada da abordagem DI incentivou uma análise mais criteriosa do código e uma revisão mais detalhada das anomalias detectadas. Isso resultou em um aumento da conscientização sobre a importância da qualidade do código em todo o processo de desenvolvimento.

As etapas do estudo descritas no presente trabalho deram suporte para responder a QP6, conforme quadro abaixo:

**QP6 - Como a abordagem DI pode ser integrada ao fluxo de trabalho do processo ágil de desenvolvimento?**
A abordagem DI pode ser integrada ao fluxo de trabalho do processo ágil de desenvolvimento por meio de uma instanciação adequada em uma metodologia ágil existente, como o Scrum. Os estudos revelaram que essa integração é viável e traz benefícios significativos para o desenvolvimento de software. Essa integração traz benefícios como a atenção à qualidade do código, detecção precoce de anomalias e colaboração entre as equipes, resultando em um software mais confiável e de melhor qualidade.

Como oportunidade de desdobramentos de pesquisa futuros, sugere-se a investigação dos desafios e estratégias para a adoção da abordagem DI em organizações de diferentes tamanhos e setores, bem como a análise dos benefícios de longo prazo dessa integração. Além disso, seria interessante explorar como a abordagem DI pode ser adaptada e estendida para lidar com aspectos específicos de desenvolvimento de software, como segurança, escalabilidade e integração contínua. Outra área de pesquisa promissora seria a avaliação comparativa da abordagem DI em relação a outras abordagens de desenvolvimento ágil, a fim de determinar suas vantagens e limitações em diferentes contextos. Esses estudos adicionais contribuiriam para a evolução e aprimoramento contínuos da abordagem DI, proporcionando *insights* valiosos para a comunidade de pesquisa e profissionais da área.

Além disso, a fim de garantir a correta utilização da proposta, é necessário estabelecer um conjunto de "diretrizes práticas"para implementar as mudanças nos papéis, eventos e artefatos do *Scrum*. Essas diretrizes podem ser desenvolvidas a partir de avaliações experimentais envolvendo profissionais da indústria de software. Para validar e aprimorar essas diretrizes, é recomendado realizar um levantamento (*survey*) com desenvolvedores de diversas organizações de software que adotam o arcabouço *Scrum* para gerir e controlar seus projetos. A análise e validação dessas diretrizes por meio do *survey* contribuirão para uma melhor compreensão das práticas eficazes de integração da abordagem DI ao fluxo de trabalho ágil, beneficiando a comunidade de profissionais e pesquisadores no desenvolvimento de software.

# Capítulo 10

# Considerações Finais

Neste capítulo são apresentadas as contribuições mais relevantes desta tese de acordo com cada Questão de Pesquisa (QP) abordada (Seção 10.1), bem como os possíveis desdobramentos futuros da pesquisa a partir dos resultados obtidos (Seção 10.2).

## 10.1 Contribuições

De acordo com os resultados dos estudos descritos na presente tese, conseguiu-se responder adequadamente as QPs definidas na presente tese (Seção 1.3). No que segue, são discutidos os principais resultados e contribuições de acordo com cada QP.

Para responder a **QP1** - *Existe a necessidade de uma abordagem para dar suporte à detecção interativa de anomalias de código?*, realizou-se inicialmente uma revisão da literatura com intuito de catalogar estudos primários e secundários que forneceram subsídios necessários para identificação e análise das principais características que distinguem a abordagem DI em contraste às abordagens DNI tradicionais. Em seguida, conseguiu-se obter evidências a respeito da necessidade da abordagem DI, mediante realização de um *survey* com desenvolvedores profissionais, provenientes de mais de 70 organizações de desenvolvimento de software.

Para responder a **QP2** - *Quais métodos de detecção se adéquam à abordagem de detecção interativa?* , utilizou-se dos resultados de estudos primários e secundários para identificar, classificar e analisar os principais métodos para detecção de anomalias do estado da arte. Baseado nos resultados desta revisão da literatura e, complementado com a opinião

de especialistas que definiram sete critérios de seleção específicos, a combinação dos métodos baseados em métricas e heurísticas mostrou-se adequado ao suporte da abordagem DI. Ainda, identificaram-se e analisaram-se as principais categorias de anomalias (i.e., *Bloaters* e *Couplers*) mais propensas a serem identificadas e analisadas utilizando a abordagem DI. Finalmente, desenvolveu-se um suporte automatizado aderente às características da abordagem de DI provendo suporte a 10 tipos distintos de anomalias de código (Apêndice C).

Para responder a **QP3** - *Quais fatores uma abordagem de detecção interativa deve considerar em sua concepção e utilização?* , realizou-se uma análise dos dados provenientes dos estudos anteriores e, com auxílio de especialistas na área de detecção de anomalias e refatoração, elencaram-se os principais fatores que uma abordagem DI deve considerar em sua concepção e utilização. Do ponto de vista da concepção, uma lista de características desejáveis foi identificada e contrastada com abordagens DNI tradicionais. Relacionado à utilização da abordagem DI, um conjunto de diretrizes centradas no usuário foram catalogadas e analisadas. Finalmente, mostrou-se a existência de uma relação direta entre esses conceitos (i.e., características e diretrizes) e validou-se sua importância a partir de um *survey* com 16 desenvolvedores de diferentes níveis de conhecimento. Os resultados da validação apontaram que os desenvolvedores, de forma geral, consideraram importantes as diretrizes descritas.

Para responder a **QP4** - *A aplicação da abordagem de detecção interativa contribui para melhoria da eficácia na detecção de anomalias de código durante a análise de código?* , realizou-se um experimento controlado envolvendo 16 desenvolvedores (entre profissionais e estudantes) para viabilizar uma análise comparativa da eficácia na detecção de anomalias de código com suporte das abordagens DI e DNI. Este experimento foi realizado no contexto da atividade de inspeção e análise de código. De modo geral, os resultados das atividades experimentais apontaram que a abordagem DI favorece a detecção de um número maior de instâncias de anomalias consideradas relevantes (Verdadeiros positivos).

Notou-se que características da abordagem DI (e.g., detecção contínua, informações contextuais e limitadas ao contexto de trabalho) contribuíram para a melhoria das medidas de eficácia consideradas no estudo. A diferença nos resultados da medida *precision* foi em média de até 25% em favor do uso da abordagem DI. Essa diferença pode ser ainda maior se considerada apenas a amostra de estudantes (cerca de 35%). Com relação à medida *recall*

obteve-se uma diferença global de aproximadamente 25% a favor da abordagem DI. Utilizando apenas a amostra de estudantes esse valor é incrementado para mais de 40%. Esses resultados podem indicar que o nível de conhecimento dos participantes do experimento pode ter influenciado os resultados. Quanto mais baixo o nível de conhecimento dos participantes (i.e., amostra de estudantes), maior tende a ser o benefício que essa amostra pode ter com o uso de uma abordagem DI em termos das medidas de eficácia consideradas no estudo.

Para responder a **QP5** - *A aplicação da abordagem de detecção interativa contribui para reduzir a quantidade de anomalias de código remanescentes durante o desenvolvimento do código?* , foi realizado um experimento controlado envolvendo 16 desenvolvedores (entre profissionais e estudantes) para avaliar se o uso da abordagem DI contribui para reduzir a quantidade de anomalias de código remanescentes durante as atividades de desenvolvimento de software. Os resultados mostraram que, usando DI, os desenvolvedores podem diminuir até 30% das ocorrências de anomalias remanescentes restantes em comparação com o uso da técnica DNI durante as atividades de desenvolvimento de software. Portanto, a técnica de DI permite que os desenvolvedores identifiquem esses problemas de modo precoce, o que pode beneficiar a manutenção do sistema. Além disso, a abordagem DI reduziu em até 40% no FP. Diminuir esse indicador pode economizar tempo e esforço dos desenvolvedores, reduzindo os alarmes falsos que eles precisam investigar, permitindo que eles se concentrem em instâncias reais de anomalias de código. Finalmente, usar DI pode reduzir o número de FN em até 20%. A diminuição deste indicador ajuda a garantir que todas as anomalias sejam identificadas e resolvidas em tempo hábil, melhorando a qualidade geral e a capacidade de manutenção do código

Durante os experimentos associados às **QP4 e QP5**, notou-se certa influência no nível de conhecimento dos participantes nos resultados experimentais. Uma suposição é que os desenvolvedores profissionais podem detectar mais anomalias em comparação com os estudantes devido à sua experiência e especialidades. Os desenvolvedores - devido às suas atividades profissionais - provavelmente encontraram uma variedade maior de anomalias e têm uma melhor compreensão das características e consequências destas. Eles também podem ter uma melhor compreensão da base de código, o que pode ajudá-los na atividade de detecção. Adicionalmente, é possível que os desenvolvedores tenham desenvolvido seu próprio conjunto de heurísticas e práticas recomendadas para a detecção de anomalias no

código. Nesse sentido, as abordagens de detecção podem complementar as decisões dos desenvolvedores nessa atividade, trazendo novas perspectivas ao processo decisório sobre a ocorrência de anomalias.

Os estudantes, por outro lado, podem não ter o mesmo nível de experiência e podem estar menos familiarizados com as características e consequências das anomalias de código, bem como com a base de código, o que pode tornar mais difícil para eles identificar anomalias. Além disso, os estudantes podem não ter desenvolvido o mesmo conjunto de heurísticas, práticas recomendadas e ferramentas que os desenvolvedores profissionais, fato que pode dificultar a detecção de anomalias. Vale ressaltar que a habilidade de detecção de anomalias pode ser aprendida e desenvolvida com o tempo, mediante prática e experiência. Assim, os estudantes podem aprimorar esta habilidade estudando e aprendendo sobre diferentes tipos de anomalias e praticando a detecção em diferentes bases de código. Além disso, eles podem se beneficiar trabalhando com desenvolvedores mais experientes e aprendendo com seus conhecimentos e práticas recomendadas.

Finalmente, para responder a **QP6** - *Como a abordagem DI pode ser integrada ao fluxo de trabalho do processo ágil de desenvolvimento?*, apresentou-se uma proposta de processo para desenvolvimento de software que integra a abordagem DI numa instanciação do *Scrum*. Quatro especialistas - sendo dois em métodos ágeis e os outros dois em detecção de anomalias e refatoração - atuaram de modo colaborativo e iterativo na propositura desta abordagem bem como na modificação dos artefatos, papéis e eventos do *Scrum* para torná-lo aderente à abordagem DI. A implementação da abordagem DI integrada ao *Scrum* foi avaliada por meio de um grupo focal com 8 profissionais, revelando benefícios como melhoria na qualidade de código, facilidade de manutenção e entrega aprimorada do produto. No entanto, foram identificadas dificuldades, como falta de conhecimento, resistência à mudança e obstáculos organizacionais. Para superar essas dificuldades, é recomendado capacitar a equipe, fornecer treinamento adequado e contar com o apoio da liderança. Além disso, um experimento controlado com 24 participantes mostrou que a abordagem de integração da DI no Scrum resultou em maior atenção à qualidade do código, detecção precoce de anomalias e colaboração entre as equipes. A utilização da abordagem proporcionou um aumento significativo nas medidas de eficácia, como *recall* e *precision*, garantindo uma detecção mais precisa e abrangente de anomalias de código. A colaboração melhorada e a análise criteriosa

do código contribuíram para um software mais confiável e de melhor qualidade. Em geral, os resultados da avaliação indicam que a abordagem DI integrada ao *Scrum* pode ser uma estratégia promissora em virtude dos pontos de convergência existentes entre ambos.

## 10.2 Trabalhos Futuros

Além das contribuições mencionadas na seção anterior, este trabalho também abre possibilidades de extensão, além de alguns pontos de melhoria. Mesmo diante da realização dos estudos experimentais descritos ao longo desse estudo, conclui-se que mais estudos podem ser requeridos com intuito de confirmar a eficácia da abordagem DI sobre abordagens tradicionais na atividade de detecção de anomalias de código. Assim, a intenção é aumentar a realização de estudos empíricos com o objetivo de avaliar os benefícios do uso da abordagem DI, bem como sua aplicabilidade na promoção contínua da qualidade. Para isso, planeja-se conduzir uma série de estudos em ambientes reais, com maior duração e um número mais diversificado de participantes. As próximas pesquisas podem explorar as seguintes questões:

1. *qual é o impacto da abordagem DI na manutenção do código e na facilidade de compreensão do mesmo?* A manutenção do código é uma atividade crucial no ciclo de vida do software, e a abordagem DI tem como objetivo melhorar a detecção e correção de anomalias de código, o que pode contribuir para a redução do tempo e custo de manutenção. Além disso, a facilidade de compreensão do código também é importante, uma vez que o código fonte deve ser legível e compreensível pelos desenvolvedores para que possa ser mantido e evoluído de forma eficaz. Desse modo, deve-se avaliar se a abordagem DI pode contribuir para a melhoria da qualidade do código e, consequentemente, para a facilidade de manutenção e compreensão do mesmo;

2. *o uso sistemático da abordagem DI leva a menos problemas associados a anomalias de código em tarefas de desenvolvimento de software?* Nesta questão de pesquisa tem-se o objetivo de avaliar se as características da abordagem DI (e.g., informações contextuais, interação direta com trechos de código anômalos, detecção antecipada e contínua, entre outras) promovem a melhoria das habilidades de programação dos desenvolvedores. Assim, mediante uso da abordagem DI, os desenvolvedores iriam

incorrer em menos instâncias de anomalias de código, bem como adquirir mais confiança em aplicar ações de refatoração para remoção;

3. *como o uso da abordagem DI afeta a experiência de produtividade dos desenvolvedores?* A literatura relata que muito do tempo e esforço é empregado em atividades de manutenção do código. Grande parte destas atividades tem por objetivo a melhoria de um ou mais atributos de qualidade. Desse modo, essa questão visa avaliar como o uso da abordagem DI afeta a experiência de produtividade dos desenvolvedores no contexto do desenvolvimento de software. Isso implica em avaliar se a adoção da abordagem DI interfere de forma positiva ou negativa na eficiência, na qualidade do trabalho realizado e na satisfação dos desenvolvedores. A resposta a essa pergunta pode ajudar a determinar se a abordagem DI é uma escolha viável para ser integrada em um processo de desenvolvimento de software em grande escala;

4. *qual é a eficácia da abordagem DI em lidar com anomalias de código em projetos de software de grande escala?* Essa questão visa avaliar a eficácia da abordagem DI na detecção e tratamento de anomalias em projetos de software de grande escala. Sabe-se que as anomalias são indícios de problemas de qualidade no código fonte, que podem afetar negativamente a manutenibilidade, a escalabilidade, a segurança e outros atributos do software. A abordagem DI, por sua vez, busca identificar esses indícios de forma interativa e com a participação ativa dos desenvolvedores, a fim de corrigi-los de forma mais eficiente e eficaz. Portanto, a questão busca avaliar se a abordagem DI é capaz de detectar e corrigir com sucesso as anomalias em projetos de grande escala, o que pode ter um impacto significativo na qualidade do software produzido.

Com relação à ferramenta *Eclipse ConCAD*, o primeiro desdobramento de trabalho futuro visando sua melhoria seria o incremento do número de anomalias de código detectadas. Atualmente a ferramenta provê suporte a detecção de 10 tipos distintos de anomalias. Cabe ressaltar que, devido a seu projeto e definição de arquitetura, tal atividade pode ser realizada com o mínimo de esforço. Outra possibilidade de trabalho futuro diz respeito à capacidade da ferramenta compreender o código em desenvolvimento em termos de diversas métricas OO e propor mudanças nas estratégias de detecção. Por exemplo, a ferramenta poderia avaliar a média de LOC (do inglês, *Lines Of Code*) de todos os módulos do projeto e, mediante

esse resultado, aplicar técnicas inteligentes (e.g., aprendizado de máquina ou algoritmos genéticos) para recomendar ao desenvolvedor o ajuste da estratégia de detecção das anomalias *God Class* e *Long Method* que utilizam prioritariamente essa métrica OO.

Outra evolução relevante para a ferramenta seria a execução automática das ações de refatoração mediante suporte do próprio ambiente de desenvolvimento integrado, o que permitiria aos desenvolvedores a possibilidade de corrigir automaticamente as anomalias de código detectadas sem a necessidade de intervenção manual. Além disso, essa abordagem de refatoração automática pode ajudar a melhorar a produtividade e a qualidade do código, reduzir os erros de programação e acelerar o processo de desenvolvimento de software. É importante mencionar que a maior parte das ferramentas do estado da arte não provêm suporte completo ao processo de gestão das anomalias de código (i.e., detecção, análise, priorização e remoção).

Adicionalmente, em versões futuras da ferramenta pode-se prover uma distinção visual entre os diferentes níveis de granularidade. Alguns participantes na avaliação sugeriram que havia diferenças fundamentais na granularidade das anomalias de código e exibir os resultados de detecção de maneira uniforme poderia ser confuso. Por exemplo, *God Class* é uma anomalia em nível de classe enquanto *Long Method* é uma anomalia em nível de método. Tal distinção visual pode ajudar os desenvolvedores a compreender a visualização mais rapidamente. Por fim, uma possível modificação na ferramenta consistiria na exibição de informações sobre as anomalias que estão "aumentando" ou "diminuindo" enquanto o programador está desenvolvendo, em vez de exibir informações sobre o código no momento da programação. Com base nessas informações históricas e nas estratégias de detecção e seus limiares, os desenvolvedores poderiam entender o impacto que suas alterações nos diversos módulos do projeto estão tendo na ocorrência ou desaparecimento de instâncias de anomalias de código. Essa abordagem permitiria aos desenvolvedores avaliar o impacto de suas ações de desenvolvimento em tempo real e ajustar suas estratégias de refatoração de forma mais eficaz.

Ademais, a integração da DI com modelos amplos de linguagem (do inglês, Large Language Models [LLM]), tais como o *ChatGPT* e *Google Bard*, pode ser perfeitamente viável e traria benefícios substanciais. Essa união poderia ser alcançada através da criação de interfaces que permitissem a colaboração entre desenvolvedores e essas ferramentas, visando

aprimorar a qualidade do código. A colaboração entre a DI e as ferramentas LLM poderia ocorrer de várias maneiras estratégicas. Primeiramente, as ferramentas LLM poderiam gerar sugestões de refatoração e alterações de código com base em padrões e boas práticas. A DI entraria nesse processo ao permitir que os desenvolvedores personalizassem e adaptassem essas sugestões de acordo com o contexto e as necessidades do projeto. Além disso, as ferramentas LLM poderiam gerar explicações e documentação para o código. A DI poderia ser empregada para refinar e aprimorar essas explicações, garantindo que fossem alinhadas às particularidades do projeto e compreensíveis para toda a equipe.

A integração entre DI e ferramentas LLM tende a ser promissora pois a colaboração entre humanos e modelos de linguagem poderia impulsionar a eficiência na detecção e correção de problemas, tornando o processo mais ágil e produtivo. Além disso, essa colaboração poderia possibilitar a personalização das sugestões automáticas das ferramentas LLM, garantindo que elas se adéquem ao código específico do projeto e às preferências da equipe. A aprendizagem contínua seria outro benefício, pois as interações humanas refinariam e melhorariam as sugestões das ferramentas LLM ao longo do tempo. Essa integração também resultaria em um código de melhor qualidade, mais legível e aderente às melhores práticas de desenvolvimento. Adicionalmente, as explicações geradas pelas ferramentas LLM auxiliariam os desenvolvedores na compreensão do código e nas mudanças propostas.

# Bibliografia


[1] Marwen Abbes, Foutse Khomh, Yann-Gael Gueheneuc, and Giuliano Antoniol. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *2011 15Th european conference on software maintenance and reengineering*, pages 181–190. IEEE, 2011.

[2] Amjad AbuHassan, Mohammad Alshayeb, and Lahouari Ghouti. Software smell detection techniques: A systematic literature review. *Journal of Software: Evolution and Process*, 33(3):1–48, 2021.

[3] Jehad Al Dallal. Identifying refactoring opportunities in object-oriented code: A systematic literature review. *Information and software Technology*, 58:231–249, 2015.

[4] Danyllo Albuquerque. Supplementary Material - On the Assessment of Continuous Detection of Code Smells: An Empirical Study (SoftCOM 2022). Figshare. https://doi.org/10.6084/m9.figshare.14210576.v2 . 3 2022.

[5] Danyllo Albuquerque. Supplementary Material - Perceptions of Technical Debt and Management Activities - A Survey of Software Practitioners (SBES 2022). Figshare. https://doi.org/10.6084/m9.figshare.20457396.v2. 2 2022.

[6] Danyllo Albuquerque. Material Suplementar - Validação da Proposta de Abordagem DI integrada ao Scrum. Figshare. https://doi.org/10.6084/m9.figshare.22777067.v1. 4 2023.

[7] Danyllo Albuquerque, Alessandro Garcia, Roberto Oliveira, and Willian Oizumi. Deteccao interativa de anomalias de codigo: Um estudo experimental. In *Proceedings of Workshop on Software Modularity, WMOD2014*. sn, 2014.

[8] Danyllo Albuquerque, Everton Guimarães, Alexandre Braga, Mirko Perkusich, Hyggo Almeida, and Angelo Perkusich. Empirical assessment on interactive detection of code smells. In *2022 International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, pages 1–6. IEEE, 2022.

[9] Danyllo Albuquerque, Everton Guimaraes, Mirko Perkusich, Hyggo Almeida, and Angelo Perkusich. Concad: A tool for interactive detection of code anomalies. In *Anais do X Workshop de Visualização, Evolução e Manutenção de Software*, pages 31–35. SBC, 2022.

[10] Danyllo Albuquerque, Everton Guimarães, Mirko Perkusich, Hyggo Almeida, and Angelo Perkusich. An approach for integrating interactive detection of code smells on agile software development. In *2023 International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, pages 1–6. IEEE, 2023.

[11] Danyllo Albuquerque, Everton Guimaraes, Mirko Perkusich, Hyggo Almeida, and Angelo Perkusich. Automated support for interactive detection of code anomalies: Proposition and evaluation. *Journal of Software Engineering Research and Development*, pages 1–15, 2023.

[12] Danyllo Albuquerque, Everton Guimarães, Mirko Perkusich, Hyggo Almeida, and Angelo Perkusich. Evaluating interactive detection of code smells on software development activities. In *2023 International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, pages 1–6. IEEE, 2023.

[13] Danyllo Albuquerque, Everton Guimarães, Mirko Perkusich, Hyggo Almeida, and Angelo Perkusich. Integrating interactive detection of code smells into scrum: Feasibility, benefits, and challenges. *Applied Sciences*, 13(15):8770, 2023.

[14] Danyllo Albuquerque, Everton Tavares Guimaraes, Graziela Simone Tonin, Mirko Barbosa Perkusich, Hyggo Almeida, and Angelo Perkusich. Perceptions of technical debt and its management activities-a survey of software practitioners. In *Proceedings of the XXXVI Brazilian Symposium on Software Engineering*, pages 220–229, 2022.

[15] Danyllo Albuquerque, Everton Guimarães, Mirko Perkusich, Thiago Rique, Felipe Cunha, Hyggo Almeida, and Angelo Perkusich. On the assessment of interactive detection of code smells in practice: A controlled experiment. *IEEE Access*, 11:84589–84606, 2023.

[16] Roberta Arcoverde, Everton Guimarães, Isela Macía, Alessandro Garcia, and Yuanfang Cai. Prioritization of code anomalies based on architecture sensitiveness. In *2013 27th Brazilian Symposium on Software Engineering*, pages 69–78. IEEE, 2013.

[17] Paris Avgeriou, Philippe Kruchten, Ipek Ozkaya, and Carolyn Seaman. Managing technical debt in software engineering (dagstuhl seminar 16162). In *Dagstuhl Reports*, volume 6. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[18] Muhammad Ilyas Azeem, Fabio Palomba, Lin Shi, and Qing Wang. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information and Software Technology*, 108:115–138, 2019.

[19] Kent Beck and Cynthia Andres. Extreme programming explained: Embrace change. 2-nd edition, 2004.

[20] Woubshet Nema Behutiye, Pilar Rodríguez, Markku Oivo, and Ayşe Tosun. Analyzing the concept of technical debt in the context of agile software development: A systematic literature review. *Information and Software Technology*, 82:139–158, 2017.

[21] Krunal Bhavsar, Vrutik Shah, and Samir Gopalan. Scrum: an agile process reengineering in software engineering. *International Journal of Innovative Technology and Exploring Engineering (IJITEE)*, 9(3):840–848, 2020.

[22] Grady Booch. *UML: guia do usuário*. Elsevier Brasil, 2006.

[23] Aloisio S Cairo, Glauco de F Carneiro, and Miguel P Monteiro. The impact of code smells on software bugs: A systematic literature review. *Information*, 9(11):273, 2018.

[24] Juyun Cho. Issues and challenges of agile software development with scrum. *Issues in Information Systems*, 9(2):188–195, 2008.

[25] M Cristina, M Radu, F Mihancea, et al. iplasma: an integrated platform for quality assessment of object-oriented design. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 77–80, 2005.

[26] Victor Machado da Silva, Helvio Jeronimo Junior, and Guilherme Horta Travassos. A taste of the software industry perception of the technical debt and its management in brazil. *Journal of Software Engineering*, 7:1, 2019.

[27] Elder Vicente de Paulo Sobrinho, Andrea De Lucia, and Marcelo de Almeida Maia. A systematic literature review on bad smells—5 w's: which, when, what, who, where. *IEEE Transactions on Software Engineering*, 2018.

[28] Lisa Nguyen Quang Do, Karim Ali, Benjamin Livshits, Eric Bodden, Justin Smith, and Emerson Murphy-Hill. Just-in-time static analysis. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 307–317. ACM, 2017.

[29] José Pereira dos Reis, Fernando Brito e Abreu, Glauco de Figueiredo Carneiro, and Craig Anslow. Code smells detection and visualization: A systematic literature review. *Archives of Computational Methods in Engineering*, pages 1–48, 2021.

[30] Paul M Duvall, Steve Matyas, and Andrew Glover. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[31] Marc Ehrig and Jérôme Euzenat. Relaxed precision and recall for ontology matching. In *Proc. K-Cap 2005 workshop on Integrating ontology*, pages 25–32. No commercial editor., 2005.

[32] Eduardo Fernandes, Alexander Chávez, Alessandro Garcia, Isabella Ferreira, Diego Cedrim, Leonardo Sousa, and Willian Oizumi. Refactoring effect on internal quality attributes: What haven't they told you yet? *Information and Software Technology*, 126:106347, 2020.

[33] Eduardo Fernandes, Johnatan Oliveira, Gustavo Vale, Thanis Paiva, and Eduardo Figueiredo. A review-based comparative study of bad smell detection tools. In *Procee-*

*dings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, pages 1–12, 2016.

[34] Eduardo Fernandes, Priscila Souza, Kecia Ferreira, Mariza Bigonha, and Eduardo Figueiredo. Detection strategies for modularity anomalies: An evaluation with software product lines. In *Information Technology-New Generations*, pages 565–570. Springer, 2018.

[35] Francesca Arcelli Fontana, Pietro Braione, and Marco Zanoni. Automatic detection of bad smells in code: An experimental assessment. *J. Object Technol.*, 11(2):5–1, 2012.

[36] Francesca Arcelli Fontana, Jens Dietrich, Bartosz Walter, Aiko Yamashita, and Marco Zanoni. Antipattern and code smell false positives: Preliminary conceptualization and classification. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, volume 1, pages 609–613. IEEE, 2016.

[37] Francesca Arcelli Fontana, Mika V Mantyla, Marco Zanoni, and Alessandro Marino. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, 21(3):1143–1191, 2016.

[38] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[39] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, and Design Patterns. *Elements of reusable object-oriented software*, volume 99. Addison-Wesley Reading, Massachusetts, 1995.

[40] George Ganea, Ioana Verebi, and Radu Marinescu. Continuous quality assessment with incode. *Science of Computer Programming*, 134:19–36, 2017.

[41] Adnane Ghannem, Ghizlane El Boussaidi, and Marouane Kessentini. On the use of design defect examples to detect model refactoring opportunities. *Software Quality Journal*, 24(4):947–965, 2016.

[42] Aakanshi Gupta, Bharti Suri, and Sanjay Misra. A systematic literature review: Code bad smells in java source code. In *International Conference on Computational Science and Its Applications*, pages 665–682. Springer, 2017.

[43] Mouna Hadj-Kacem and Nadia Bouassida. Towards a taxonomy of bad smells detection approaches. In *ICSOFT*, pages 198–209, 2018.

[44] Md Shariful Haque, Jeff Carver, and Travis Atkison. Causes, impacts, and detection approaches of code smell: a survey. In *Proceedings of the ACMSE 2018 Conference*, pages 1–8, 2018.

[45] Emam Hossain, Paul L Bannerman, and Ross Jeffery. Towards an understanding of tailoring scrum in global software development: a multi-case study. In *Proceedings of the 2011 International Conference on Software and Systems Process*, pages 110–119, 2011.

[46] Mário Hozano, Alessandro Garcia, Baldoino Fonseca, and Evandro Costa. Are you smelling it? investigating how similar developers detect code smells. *Information and Software Technology*, 93:130–146, 2018.

[47] Andreas Jedlitschka, Marcus Ciolkowski, and Dietmar Pfahl. Reporting experiments in software engineering. In *Guide to advanced empirical software engineering*, pages 201–228. Springer, 2008.

[48] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[49] Amandeep Kaur. A systematic literature review on empirical analysis of the relationship between code smells and software quality attributes. *Archives of Computational Methods in Engineering*, 27(4):1267–1296, 2020.

[50] Foutse Khomh, Massimiliano Di Penta, and Yann-Gael Gueheneuc. An exploratory study of the impact of code smells on software change-proneness. In *2009 16th Working Conference on Reverse Engineering*, pages 75–84. IEEE, 2009.

[51] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guëhëneuc, and Giuliano Antoniol. An exploratory study of the impact of antipatterns on class change-and fault-proneness. *Empirical Software Engineering*, 17(3):243–275, 2012.

[52] Amy J Ko, Thomas D LaToza, and Margaret M Burnett. A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Software Engineering*, 20(1):110–141, 2015.

[53] Eric D Kolaczyk and Gábor Csárdi. *Statistical analysis of network data with R*, volume 65. Springer, 2014.

[54] Jyrki Kontio, Johanna Bragge, and Laura Lehtola. The focus group method as an empirical tool in software engineering. *Guide to advanced empirical software engineering*, pages 93–116, 2008.

[55] Marie Kraska-Miller. *Nonparametric statistics for social and behavioral sciences*. CRC Press, 2013.

[56] Guilherme Lacerda, Fabio Petrillo, Marcelo Pimenta, and Yann Gaël Guéhéneuc. Code smells and refactoring: A tertiary systematic review of challenges and observations. *Journal of Systems and Software*, 167:110610, 2020.

[57] Michele Lanza and Radu Marinescu. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media, 2007.

[58] Michele Lanza and Radu Marinescu. Object-oriented metrics in practice: Using software metrics to characterize, evaluate, and improve the design of object-oriented systems, 2010.

[59] Wei Li and Raed Shatnawi. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *Journal of systems and software*, 80(7):1120–1128, 2007.

[60] Zengyang Li, Paris Avgeriou, and Peng Liang. A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, 101:193–220, 2015.

[61] Johan Linåker, Sardar Muhammad Sulaman, Rafael Maiani de Mello, and Martin Höst. Guidelines for conducting surveys in software engineering. 2015.

[62] Welf Löwe and Thomas Panas. Rapid construction of software comprehension tools. *International Journal of Software Engineering and Knowledge Engineering*, 15(06):995–1025, 2005.

[63] Isela Macia, Roberta Arcoverde, Elder Cirilo, Alessandro Garcia, and Arndt von Staa. Supporting the identification of architecturally-relevant code anomalies. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 662–665. IEEE, 2012.

[64] Isela Macia, Roberta Arcoverde, Alessandro Garcia, Christina Chavez, and Arndt Von Staa. On the relevance of code anomalies for identifying architecture degradation symptoms. In *2012 16Th european conference on software maintenance and reengineering*, pages 277–286. IEEE, 2012.

[65] Mika Mantyla. *Bad smells in software-a taxonomy and an empirical study*. PhD thesis, PhD thesis, Helsinki University of Technology, 2003.

[66] Mika V Mantyla. An experiment on subjective evolvability evaluation of object-oriented software: explaining factors and interrater agreement. In *2005 International Symposium on Empirical Software Engineering, 2005.*, pages 10–pp. IEEE, 2005.

[67] Radu Marinescu. Detecting design flaws via metrics in object-oriented systems. In *Proceedings 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems. TOOLS 39*, pages 173–182. IEEE, 2001.

[68] Radu Marinescu. Measurement and quality in object-oriented design. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 701–704. IEEE, 2005.

[69] Radu Marinescu, George Ganea, and Ioana Verebi. Incode: Continuous quality assessment and improvement. In *2010 14th European Conference on Software Maintenance and Reengineering*, pages 274–275. IEEE, 2010.

[70] Raúl Marticorena, Carlos López, and Yania Crespo. Extending a taxonomy of bad code smells with metrics. In *Proceedings of 7th International Workshop on Object-Oriented Reengineering (WOOR)*, page 6. Citeseer, 2006.

[71] Antonio Martini, Terese Besker, and Jan Bosch. Technical debt tracking: Current state of practice: A survey and multiple case study in 15 large organizations. *Science of Computer Programming*, 163:42–61, 2018.

[72] Júlio Martins, Carla Bezerra, Anderson Uchôa, and Alessandro Garcia. Are code smell co-occurrences harmful to internal quality attributes? a mixed-method study. In *Proceedings of the 34th Brazilian Symposium on Software Engineering*, pages 52–61, 2020.

[73] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on software engineering*, 30(2):126–139, 2004.

[74] Petru Florin Mihancea and Radu Marinescu. Towards the optimization of automatic detection of design flaws in object-oriented software systems. In *Ninth European conference on software maintenance and reengineering*, pages 92–101. IEEE, 2005.

[75] Mohamed Wiem Mkaouer, Marouane Kessentini, Mel Ó Cinnéide, Shinpei Hayashi, and Kalyanmoy Deb. A robust multi-objective approach to balance severity and importance of refactoring opportunities. *Empirical Software Engineering*, 22(2):894–927, 2017.

[76] Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and Anne-Francoise Le Meur. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, 2009.

[77] Rodrigo Morales, Zéphyrin Soh, Foutse Khomh, Giuliano Antoniol, and Francisco Chicano. On the use of developers' context for automatic refactoring of software anti-patterns. *Journal of systems and software*, 128:236–251, 2017.

[78] Haris Mumtaz, Fabian Beck, and Daniel Weiskopf. Detecting bad smells in software systems with linked multivariate visualizations. In *2018 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 12–20. IEEE, 2018.

[79] Emerson Murphy-Hill, Titus Barik, and Andrew P Black. Interactive ambient visualizations for soft advice. *Information Visualization*, 12(2):107–132, 2013.

[80] Emerson Murphy-Hill and Andrew P Black. Refactoring tools: Fitness for purpose. *IEEE software*, 25(5):38–44, 2008.

[81] Emerson Murphy-Hill and Andrew P Black. Seven habits of a highly effective smell detector. In *Proceedings of the 2008 international workshop on Recommendation systems for software engineering*, pages 36–40, 2008.

[82] Emerson Murphy-Hill and Andrew P Black. An interactive ambient visualization for code smells. In *Proceedings of the 5th international symposium on Software visualization*, pages 5–14, 2010.

[83] Willian Oizumi, Leonardo Sousa, Alessandro Garcia, Roberto Oliveira, Anderson Oliveira, OI Anne Benedicte Agbachi, and Carlos Lucena. Revealing design problems in stinky code: a mixed-method study. In *Proceedings of the 11th Brazilian Symposium on Software Components, Architectures, and Reuse*, pages 1–10, 2017.

[84] Steffen Olbrich, Daniela S Cruzes, Victor Basili, and Nico Zazworka. The evolution and impact of code smells: A case study of two open source systems. In *2009 3rd international symposium on empirical software engineering and measurement*, pages 390–400. IEEE, 2009.

[85] Steffen M Olbrich, Daniela S Cruzes, and Dag IK Sjøberg. Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10. IEEE, 2010.

[86] Roberto Felício de Oliveira. *To collaborate or not to collaborate? Improving the identification of code smells*. PhD thesis, PUC-Rio, 2017.

[87] William F Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign Champaign, IL, USA, 1992.

[88] Juliana Padilha, Juliana Pereira, Eduardo Figueiredo, Jussara Almeida, Alessandro Garcia, and Cláudio Sant'Anna. On the effectiveness of concern metrics to detect code smells: An empirical study. In *International Conference on Advanced Information Systems Engineering*, pages 656–671. Springer, 2014.

[89] Thanis Paiva, Amanda Damasceno, Eduardo Figueiredo, and Cláudio Sant'Anna. On the evaluation of code smells and detection tools. *Journal of Software Engineering Research and Development*, 5(1):7, 2017.

[90] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering*, 23(3):1188–1221, 2018.

[91] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrea De Lucia. Do they really smell bad? a study on developers' perception of bad code smells. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 101–110. IEEE, 2014.

[92] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. Detecting bad smells in source code using change history information. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 268–278. IEEE, 2013.

[93] Fabio Palomba, Andrea De Lucia, Gabriele Bavota, and Rocco Oliveto. Anti-pattern detection: Methods, challenges, and open issues. In *Advances in Computers*, volume 95, pages 201–238. Elsevier, 2014.

[94] Chris Parnin, Carsten Görg, and Ogechi Nnadi. A catalogue of lightweight visualizations to support code smell inspection. In *Proceedings of the 4th ACM symposium on Software visualization*, pages 77–86, 2008.

[95] Fabiano Pecorelli, Fabio Palomba, Dario Di Nucci, and Andrea De Lucia. Comparing heuristic and machine learning approaches for metric-based code smell detection. In

*2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 93–104. IEEE, 2019.

[96] Paula Rachow. Refactoring decision support for developers and architects based on architectural impact. In *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 262–266. IEEE, 2019.

[97] Václav Rajlich. Software evolution and maintenance. In *Future of Software Engineering Proceedings*, FOSE 2014, page 133–144. Association for Computing Machinery, New York, NY, USA, 2014.

[98] Kenneth S Rubin. *Essential Scrum: A practical guide to the most popular Agile process*. Addison-Wesley, 2012.

[99] Natthawute Sae-Lim, Shinpei Hayashi, and Motoshi Saeki. Context-based approach to prioritize code smells for prefactoring. *Journal of Software: Evolution and Process*, 30(6):e1886, 2018.

[100] Dilan Sahin, Marouane Kessentini, Slim Bechikh, and Kalyanmoy Deb. Code-smell detection as a bilevel problem. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(1):1–44, 2014.

[101] Islem Saidani, Ali Ouni, Mohamed Wiem Mkaouer, and Fabio Palomba. On the impact of continuous integration on refactoring practice: An exploratory study on travistorrent. *Information and Software Technology*, 138:106618, 2021.

[102] Luciano Sampaio and Alessandro Garcia. Exploring context-sensitive data flow analysis for early vulnerability detection. *Journal of Systems and Software*, 113:337–361, 2016.

[103] Markus Schnappinger, Mohd Hafeez Osman, Alexander Pretschner, Markus Pizka, and Arnaud Fietzke. Software quality assessment in practice: a hypothesis-driven framework. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–6, 2018.

[104] Ken Schwaber and Jeff Sutherland. The scrum guide. *URL: https://gomyskills.com/wp-content/uploads/2021/01/2020-Scrum-Guide-US.pdf*, 2020.

[105] Tushar Sharma and Diomidis Spinellis. A survey on software smells. *Journal of Systems and Software*, 138:158–173, 2018.

[106] Raed Shatnawi and Wei Li. An investigation of bad smells in object-oriented design. In *Third International Conference on Information Technology: New Generations (ITNG'06)*, pages 161–165. IEEE, 2006.

[107] Ana Silva, Thalles Araújo, João Nunes, Mirko Perkusich, Ednaldo Dilorenzo, Hyggo Almeida, and Angelo Perkusich. A systematic review on the use of definition of done on agile software development projects. In *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, pages 364–373, 2017.

[108] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. Why we refactor? confessions of github contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 858–870, 2016.

[109] Victor Machado Silva, Helvio Jeronimo Junior, and Guilherme Horta Travassos. A taste of the software industry perception of technical debt and its management in brazil. *Journal of Software Engineering Research and Development*, 7:1–1, 2019.

[110] Chris Simons, Jeremy Singer, and David R White. Search-based refactoring: Metrics are not enough. In *International Symposium on Search Based Software Engineering*, pages 47–61. Springer, 2015.

[111] Dag IK Sjøberg, Aiko Yamashita, Bente CD Anda, Audris Mockus, and Tore Dybå. Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering*, 39(8):1144–1156, 2012.

[112] Ioannis G Stamelos and Panagiotis Sfetsos. *Agile software development quality assurance*. Igi Global, 2007.

[113] Alexandru Telea and Lucian Voinea. Visual software analytics for the build optimization of large-scale software systems. *Computational Statistics*, 26(4):635–654, 2011.

[114] Guilherme Travassos, Forrest Shull, Michael Fredericks, and Victor R Basili. Detecting defects in object-oriented designs: using reading techniques to increase software quality. *ACM sigplan notices*, 34(10):47–56, 1999.

[115] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. Ten years of jdeodorant: Lessons learned from the hunt for smells. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 4–14. IEEE, 2018.

[116] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. When and why your code starts to smell bad. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 403–414. IEEE, 2015.

[117] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. When and why your code starts to smell bad (and whether the smells go away). *IEEE Transactions on Software Engineering*, 43(11):1063–1088, 2017.

[118] Eva Van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.*, pages 97–106. IEEE, 2002.

[119] Rini Van Solingen, Vic Basili, Gianluigi Caldiera, and H Dieter Rombach. Goal question metric (gqm) approach. *Encyclopedia of software engineering*, 2002.

[120] Carmine Vassallo, Fabio Palomba, Alberto Bacchelli, and Harald C Gall. Continuous code quality: are we (really) doing that? In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 790–795, 2018.

[121] Sira Vegas, Cecilia Apa, and Natalia Juristo. Crossover designs in software engineering experiments: Benefits and perils. *IEEE Transactions on Software Engineering*, 42(2):120–135, 2015.

[122] Santiago Vidal, Hernan Vazquez, J Andres Diaz-Pace, Claudia Marcos, Alessandro Garcia, and Willian Oizumi. Jspirit: a flexible tool for the analysis of code smells. In *2015 34th International Conference of the Chilean Computer Science Society (SCCC)*, pages 1–6. IEEE, 2015.

[123] William C Wake. *Refactoring workbook*. Addison-Wesley Professional, 2004.

[124] Roel J Wieringa. *Design science methodology for information systems and software engineering*. Springer, 2014.

[125] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[126] Aiko Yamashita and Leon Moonen. Do developers care about code smells? an exploratory survey. In *2013 20th working conference on reverse engineering (WCRE)*, pages 242–251. IEEE, 2013.

[127] Aiko Yamashita and Leon Moonen. Exploring the impact of inter-smell relations on software maintainability: An empirical study. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 682–691. IEEE, 2013.

[128] R Yaqoob, SUR Khan, MA Shah, et al. Tertiary study on landscaping the review in code smells. In *Competitive Advantage in the Digital Economy (CADE 2021)*, volume 2021, pages 131–136. IET, 2021.

[129] Jesse Yli-Huumo, Andrey Maglyas, and Kari Smolander. How do software development teams manage technical debt?–an empirical study. *Journal of Systems and Software*, 120:195–218, 2016.

# Apêndice A

# Métricas de Software

Tabela A.1: Métricas de Software Suportadas pela Ferramenta *Eclipse ConCAD*.

| # | Sigla | Nome da métrica |
|---|---|---|
| 1 | WMC | *Weigth Method Cyclomatic* |
| 2 | NOF | *Number of Fields* |
| 3 | NOM | *Number of Methods* |
| 4 | LOC | *Lines of Codes* |
| 5 | ATFD | *Access to Foreign Data* |
| 6 | ATLD | *Access to Local Data* |
| 7 | TCC | *Tight Class Cohesion* |
| 8 | MNL | *Maximum Nesting Level* |
| 9 | FON | *Fields of Node* |
| 10 | MFA | *Method Field Access* |
| 11 | MCFA | *Method Class Fields Access* |
| 12 | NOAV | *Number of Accessed Variables* |
| 13 | NOPA | *Number of Public Attributes* |
| 14 | NOPM | *Number of Public Methods* |
| 15 | NOAM | *Number of Accessed Methods* |
| 16 | CINT | *Coupling Intensity* |

| | | |
|---|---|---|
| 17 | CDISP | *Coupling Dispersion* |
| 18 | NOvM | *Number of Overrides Methods* |
| 19 | NPOvM | *Number of Public Overrides Methods* |
| 20 | NPOvMAns | *Number of Public Overrides Methods in Ancester* |
| 21 | AMW | *Average Method Weight* |
| 22 | CC | *Changing Classes* |
| 23 | CM | *Changing Methods* |
| 24 | NProtM | *Number of Protected Members* |
| 25 | BUR | *Base Class Usage Ratio* |
| 26 | BOvR | *Base Class Overriding Ratio* |
| 27 | MNL | *Maximum Nesting Level* |
| 28 | FDP | *Foreign Data Providers* |
| 29 | LAA | *Locality of Attribute Accesses* |
| 30 | CYCLO | *McCabe's Cyclomatic Number* |

# Apêndice B

# Estratégias de Detecção

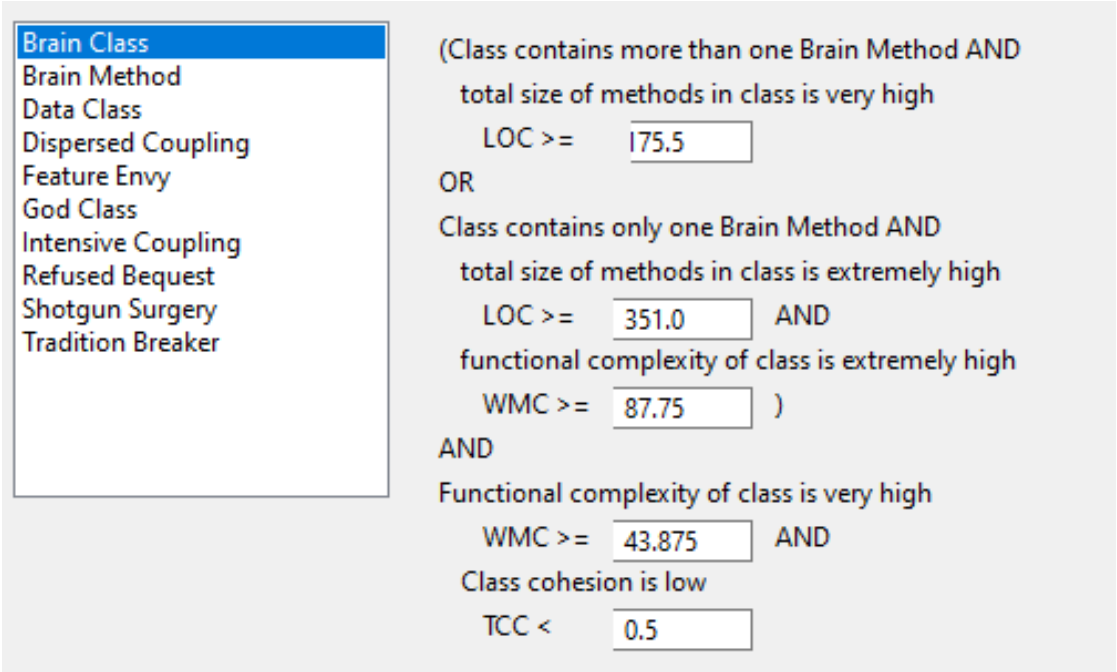


Figura B.1: Estratégia de Detecção - *Brain Class*.

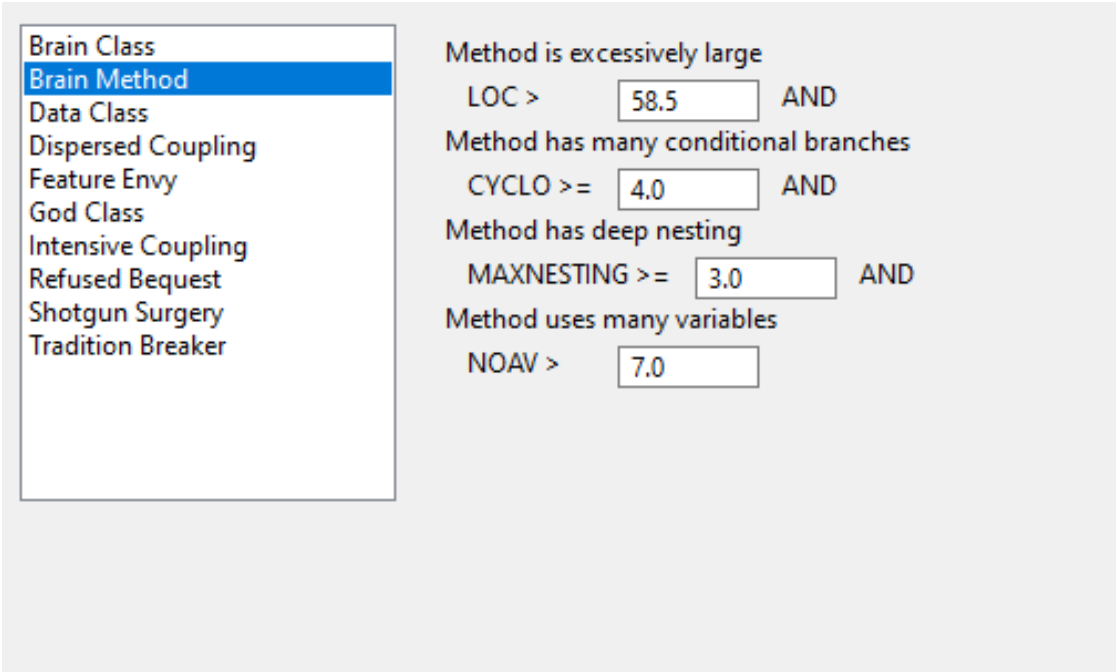


Figura B.2: Estratégia de Detecção - *Brain Method*.

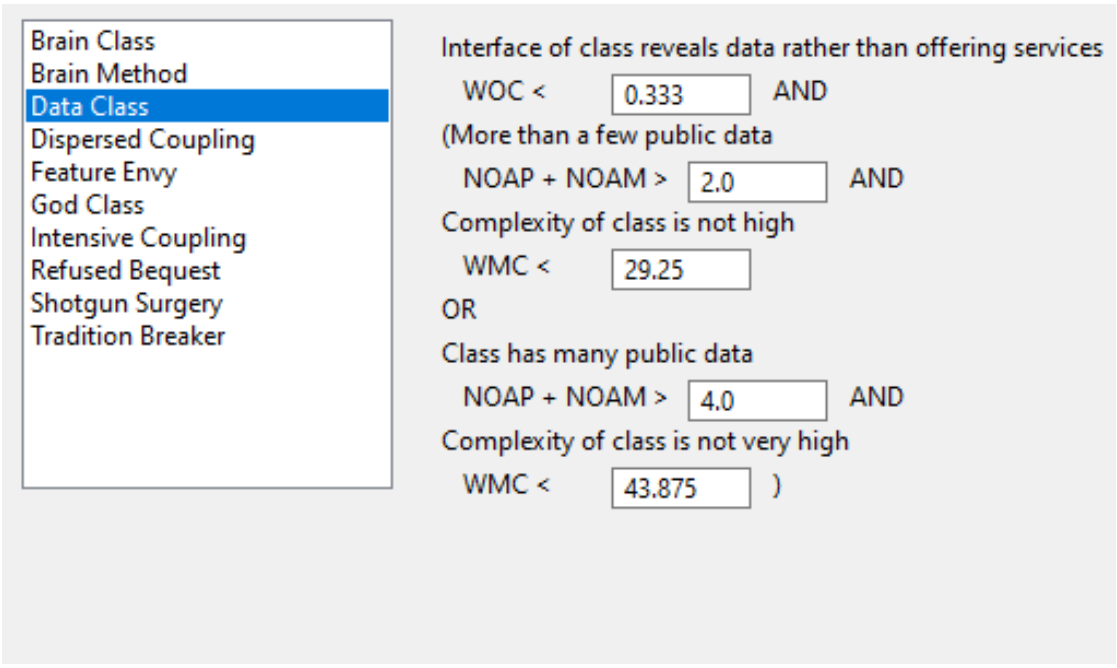


Figura B.3: Estratégia de Detecção - *Data Class*.

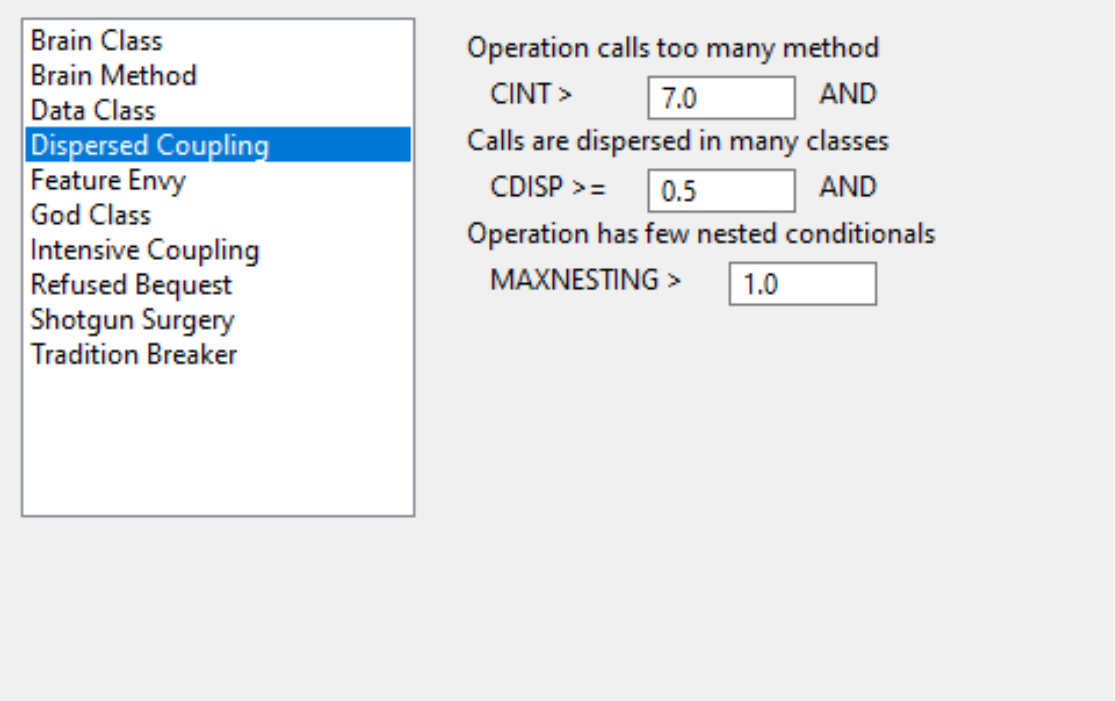


Figura B.4: Estratégia de Detecção - *Dispersed Coupling*.

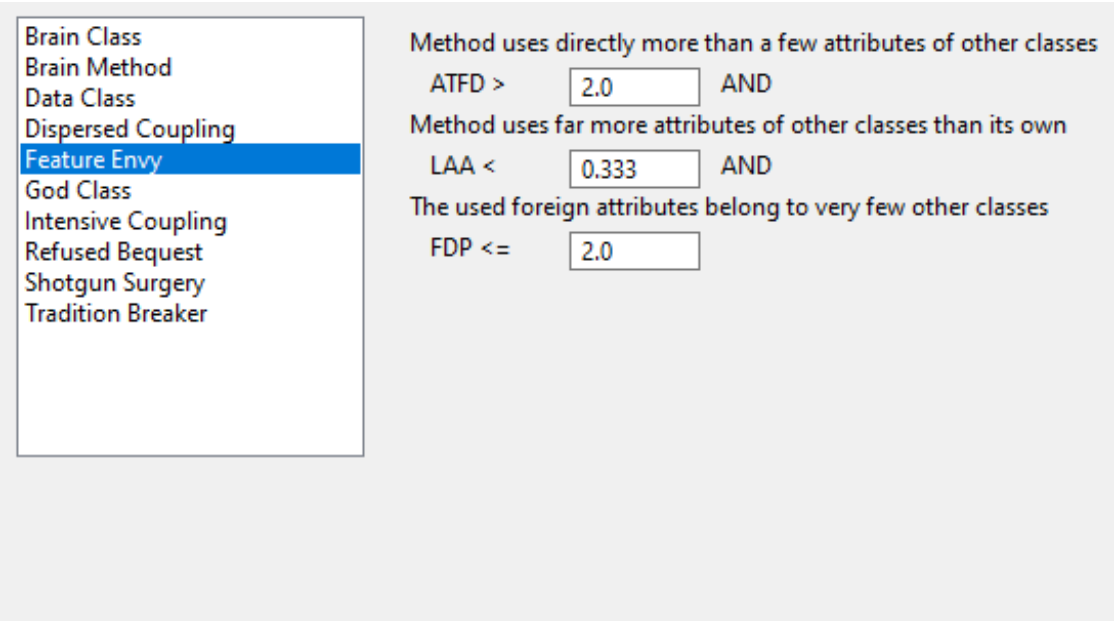


Figura B.5: Estratégia de Detecção - *Feature Envy*.

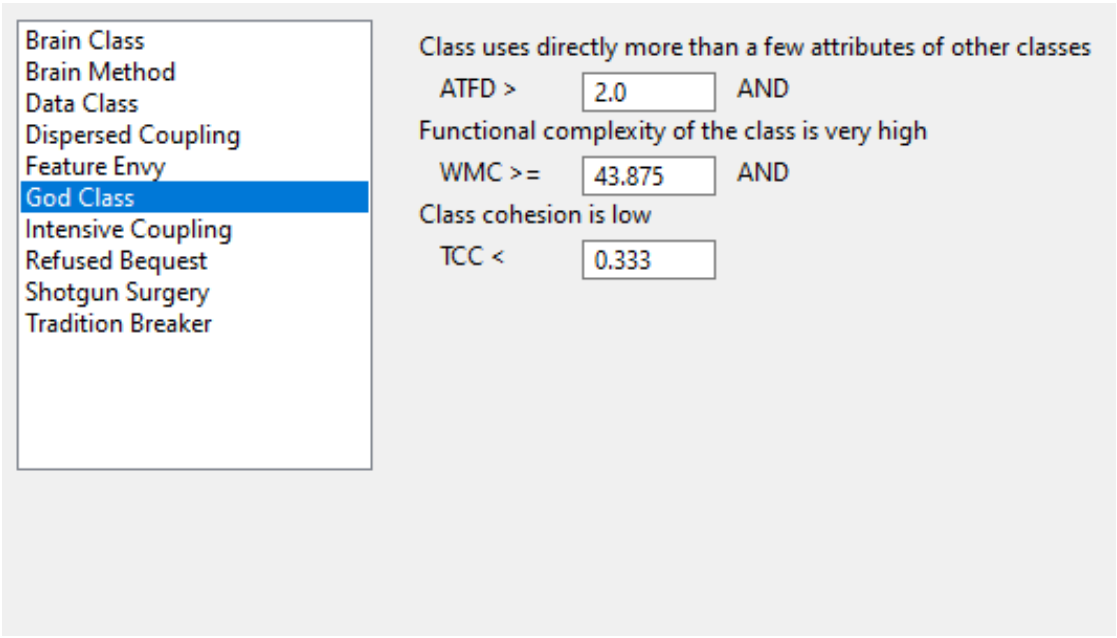


Figura B.6: Estratégia de Detecção - *God Class*.

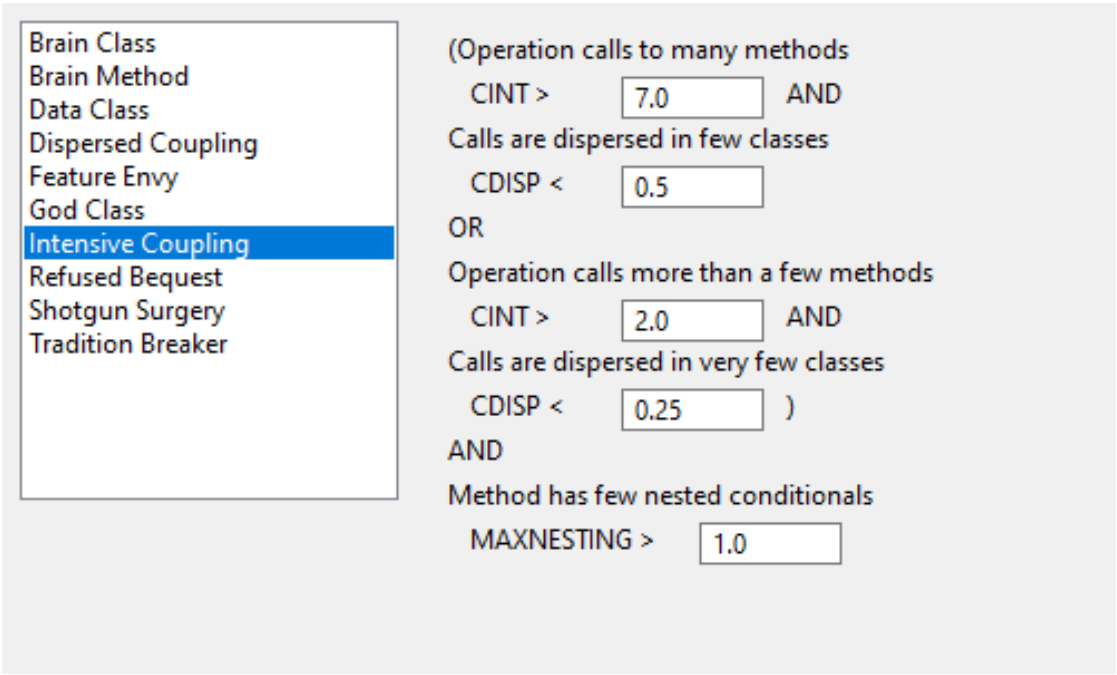


Figura B.7: Estratégia de Detecção - *Intensive Coupling*.

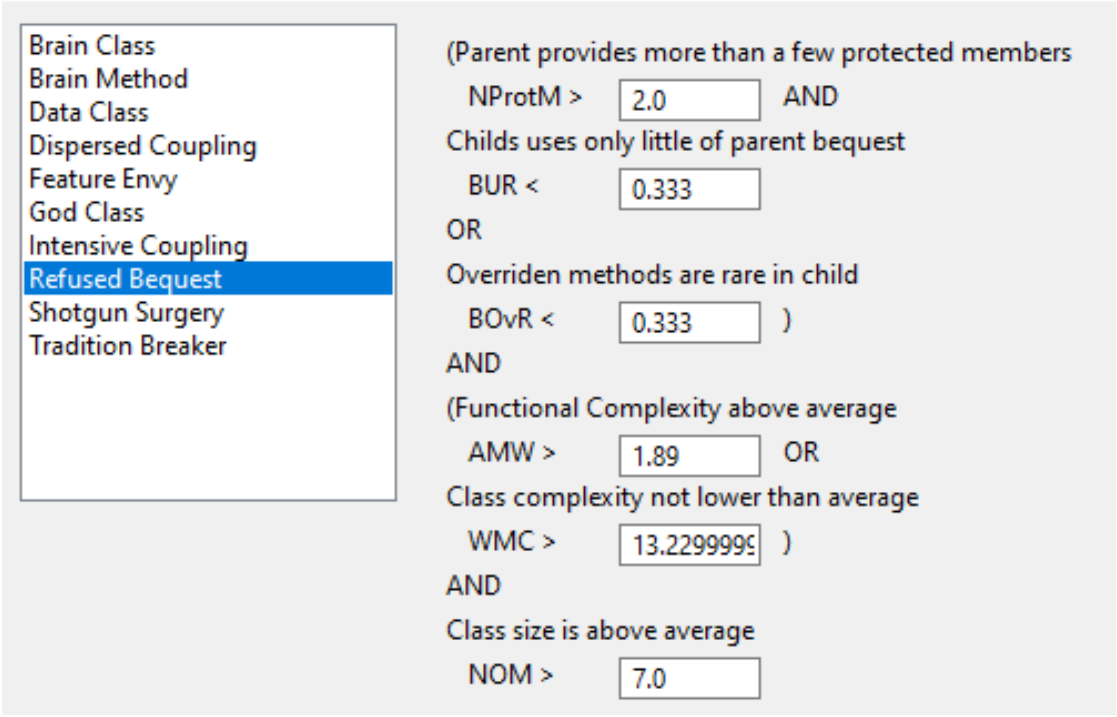


Figura B.8: Estratégia de Detecção - *Refused Bequest*.

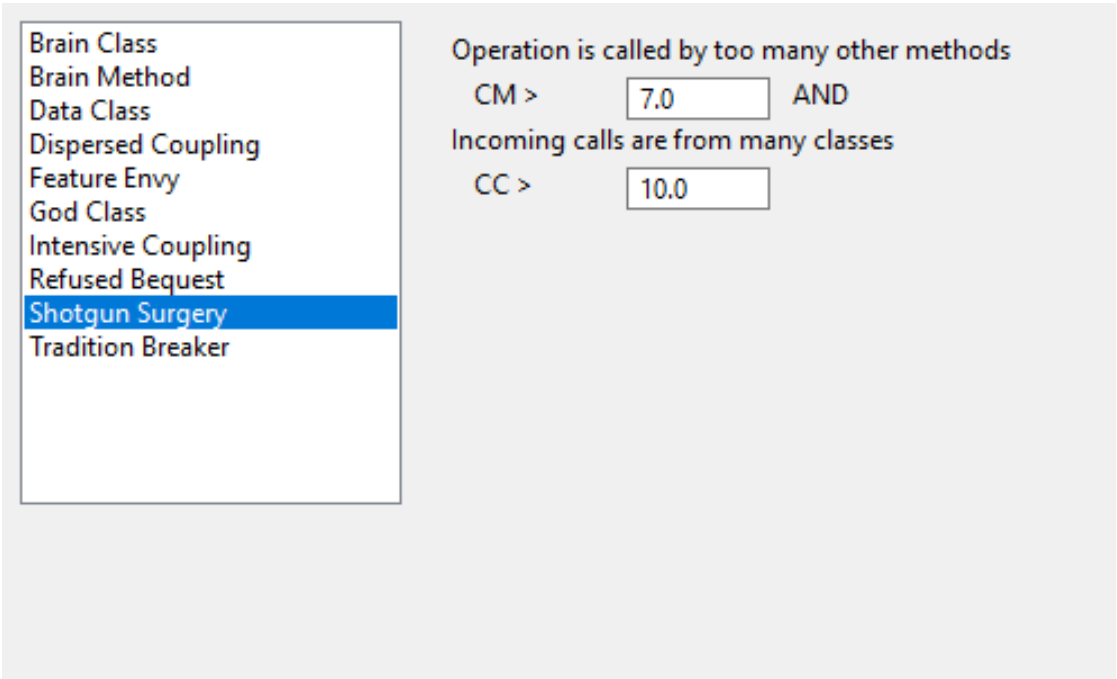


Figura B.9: Estratégia de Detecção - *Shotgun Surgery*.

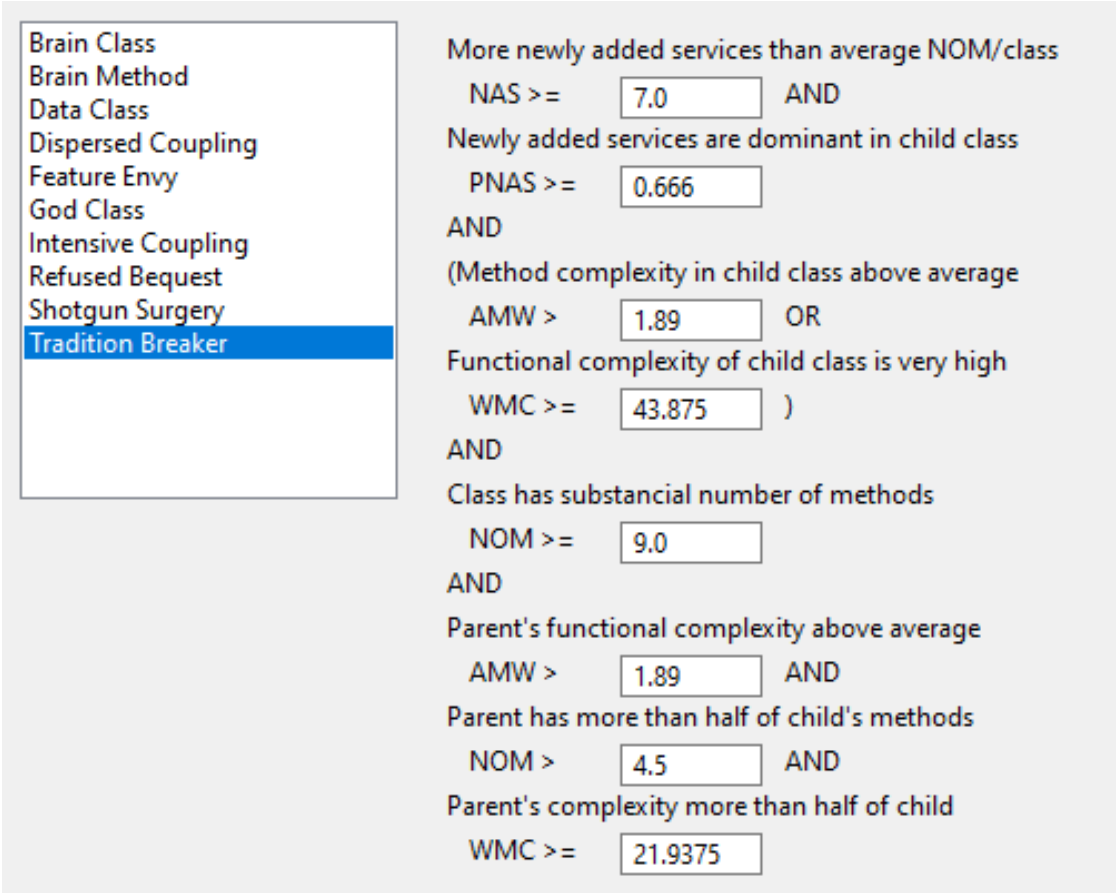


Figura B.10: Estratégia de Detecção - *Tradition Breaker*.

# Apêndice C

# Anomalias de Código

Tabela C.1: Anomalias de Código Suportadas pela Ferramenta *Eclipse ConCAD*.

| **Nome** | **Descrição** |
|---|---|
| *Brain Class* | Classes complexas que acumulam muitas responsabilidades e muitos Brain Methods. |
| *Brain Method* | Um método que é muito longo e possui muitas responsabilidades. |
| *Feature Envy* | Se refere a métodos que usam muito mais dados de outras classes do que da classe em que estão definidos. |
| *Data Class* | Classes que contém dados sem nenhum tipo comportamento associado a esses dados. |
| *Dispersed Coupling* | Método que chama mais métodos de uma classe externa do que da própria classe. |
| *God Class* | Classes longas e complexas que centralizam a inteligência do sistema. |
| *Intensive Coupling* | Método que chama vários métodos que são implementado em uma ou poucas classes. |
| *Refused Bequest* | Ocorre quando uma subclasse praticamente não utiliza os dados e métodos herdados da classe pai. |
| *Tradition Breaker* | Subclasse que não especializa a superclasse. |
| *Shotgun Surgery* | Método chamado por muitos métodos que são implementados em classes diferentes. |

# Apêndice D

# Detalhamento da Ferramenta *Eclipse ConCAD*

O presente Apêndice irá apresentar mais detalhes sobre a ferramenta *Eclipse ConCAD*. Na Seção D.1 são apresentados detalhes referentes à arquitetura bem como sobre a implementação da ferramenta. Em seguida, na Seção D.2 são descritas as principais funcionalidades desta ferramenta. Ademais, os resultados associados a construção e validação desta ferramenta foram publicados nos anais do X *Workshop* de Visualização, Evolução e Manutenção de Software (VEM'22) [9] e no *Journal of Software Engineering Research and Development* (JSERD) [11].

## D.1 Detalhes Arquiteturais

A ferramenta *ConCAD* foi implementada como um *plugin* do ambiente de desenvolvimento integrado *Eclipse*. A sua arquitetura foi projetada de acordo com seis componentes principais conforme descrito na Figura D.1.

**Plataforma Eclipse**. Sendo um *plugin* do Eclipse, o ConCAD possui dependência direta do núcleo da Plataforma Eclipse, mais precisamente dos três seguintes componentes: *workspace*, *workbench* e o *incremental project build*. O *workspace* é utilizado para obter acesso aos recursos dos projetos Eclipse e suas interdependências. O *workbench* é usado para definir os marcadores que a ferramenta irá utilizar para anotar os arquivos de código-fonte com as anomalias detectadas no projeto e para definir as formas de visualização que o ConCAD

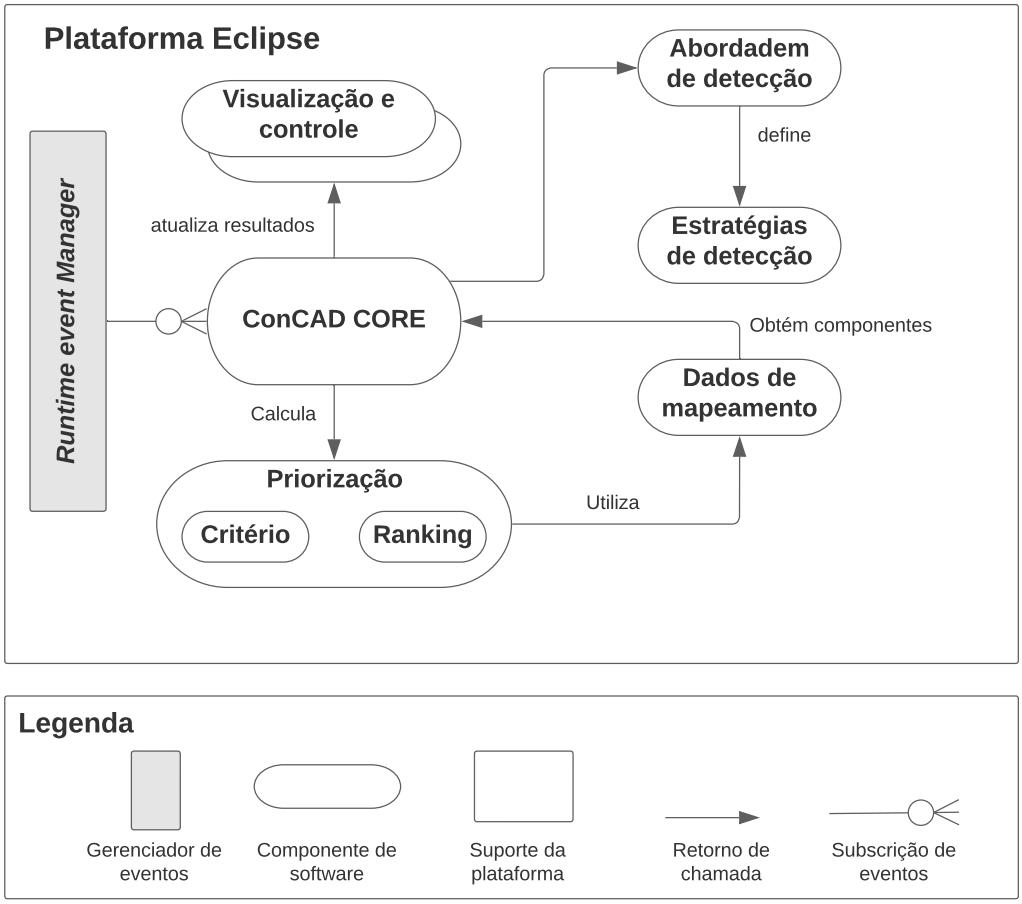


Figura D.1: Arquitetura da Ferramenta *Eclipse ConCAD*.

precisa para apresentar seus resultados de análise mais localizados. Finalmente, mecanismo de *incremental project build* é usado obter os elementos de código (e.g., *statements*, métodos, classes e pacotes) que mudaram desde a última compilação e também o fator de mudança que descreve as mudanças reais em um determinado elemento.

A dependência mais importante da ferramenta ConCAD está associada ao projeto Eclipse JDT [1] (do inglês, *Java Development Tools*). Este projeto provê os meios necessários para desenvolvimento de *plugins* para o ampliar as funcionalidades do ambiente de desenvolvimento integrado Eclipse. Assim, o ConCAD precisa de informações sobre os elementos de código que estão envolvidos em suas várias métricas, regras de detecção e meios de visualização e exploração. A maioria dessas informações é fornecida pelo *Java Model* que é a representação de um projeto Java, fornecendo um modelo mais leve associado aos elementos de código. No

[1] https://www.eclipse.org/jdt/overview.php

entanto, informações mais detalhadas como referências cruzadas (e.g., chamadas de método, acessos variáveis) e elementos de programa de baixo nível (e.g., parâmetros, variáveis locais, tipos de retorno) estão disponíveis apenas por meio dos objetos baseados na AST (do inglês, *Abstract Syntax Tree*). É importante mencionar que a maior parte das análises do ConCAD são dependentes dessas informações baseadas nos modelos *Java Model* e AST.

**ConCAD Core**. É o componente responsável por gerenciar o carregamento e análise dos arquivos de código-fonte na linguagem java através de sua interação direta com a plataforma Eclipse. Desse modo, esse componente se preocupa em obter e tratar as informações baseadas em *Java Model* e AST para prover as funcionalidades da ferramenta. Em linhas gerais, ele gerencia o fluxo de trabalho da ferramenta através da detecção de anomalias de código bem como da interação com o componente de priorização para gerar a classificação das anomalias detectadas.

**Estratégias de Detecção**. Este componente implementa uma ampla variedade de estratégias para detecção de cada um dos tipos de anomalias de código (vide estratégias de detecção da ferramenta no Apêndice B). Através deste componente, valores de limiares das métricas previamente configurados ou obtidos do usuários através de mecanismos de visualização e controle são utilizados nas estratégias de detecção. Além disso, novas e diferentes regras de detecção para anomalias de código podem ser facilmente adicionadas por desenvolvedores através deste componente.

**Dados de Mapeamentos**. Este componente suporta a estrutura de mapeamento de informações externas para o código-fonte. O usuário também pode configurar novos tipos de mapeamentos a serem usados pelas estratégias de priorização. Uma forma de aproveitar as informações externas quando as anomalias são analisadas e priorizadas é através do mapeamento dos elementos do modelo de *design* (e.g., módulos, responsabilidades e cenários) aos elementos de código implementados (e.g., pacotes, classes, métodos). Desta forma, o ConCAD pode ser instanciado por um desenvolvedor para levar em conta diferentes tipos de requisitos e documentação de arquitetura, como visualizações de componentes e conectores, cenários baseados em atributos de qualidade entre outros. Em todos os casos, um elemento do modelo de *design* será mapeado, pelo usuário da ferramenta, para uma ou mais classes (ou pacotes) para indicar sua rastreabilidade.

Por exemplo, o ConCAD pode ser instanciado para permitir ao usuário vincular um cená-

rio particular de modificação de elementos de código. Um cenário de modificação descreve uma propriedade relacionada à mudança que é desejável em um sistema. A definição de cenários de modificação pode ajudar a avaliar o quão fácil (ou difícil) tende a ser a realização de modificações em um determinado grupo de elementos de código.

**Critérios de Priorização**. Este componente calcula uma classificação para um conjunto de anomalias detectadas pela ferramenta. As formas em que o cálculo são realizados bem como da aplicação dos critérios são extensíveis. Ou seja, novos critérios de priorização podem ser adicionados por desenvolvedores sem alteração a arquitetura, graças a um conjunto de interfaces.

Uma vez que as anomalias de código são descobertas, elas devem ser classificadas de acordo com sua importância. Um usuário pode instanciar o ConCAD para priorizar anomalias mediante diferentes critérios. Por exemplo, a relevância do tipo de anomalia, o histórico de versões do sistema, as diferentes métricas de software, entre outros critérios. Além disso, o usuário pode usar informações externas para melhorar a priorização. Neste caso, o usuário pode criar um critério para classificar primeiro aquelas anomalias que comprometem a arquitetura do sistema em um dado cenário de modificação. Essa análise pode ser realizada, por exemplo, definindo um critério com base na relação entre as anomalias e os cenários de modificação do sistema. Então, as informações externas incorporadas ao ConCAD tornam-se importantes para determinar a criticidade de cada instância de anomalia.

**Visualizações e Controles**. Este componente representa um conjunto de interfaces com o usuário (e.g., visualizações, caixas de diálogo e manipuladores de ação) que visam fornecer as funcionalidades da ferramenta para seus usuários. Desse modo, o usuário pode interagir com a interface do usuário para mapear as informações externas para o código-fonte, definir suas estratégias de detecção, definir seus critérios de priorização bem como analisar os resultados associados a detecção de anomalias de código.

### D.1.1 Atributos de Qualidade

A construção da ferramenta ConCAD foi conduzida por um conjunto de objetivos de *design* que tornam o ConCAD uma ferramenta mais complexa e versátil do que os usuários podem perceber ao interagir com seus mecanismos de visualização e controle. No que se segue, declara-se os três principais objetivos de *design* que impulsionaram a criação do ConCAD e

discutem-se as decisões de implementação adotadas como resultado desses objetivos.

1. **Integração**. ConCAD é um *plugin* do Eclipse, portanto, é importante fornecer uma integração perfeita com o ecossistema deste ambiente de desenvolvimento integrado, reutilizando extensivamente a infraestrutura de análise e os componentes de interface com usuário do Eclipse. A primeira versão do ConCAD foi entregue para o Eclipse Oxygen. No entanto, como o ConCAD depende exclusivamente da API pública do Eclipse, tanto para extração dos modelos quanto para integração com os componentes de interface com usuário, manteve sua compatibilidade mesmo com a versão mais recente do Eclipse Photon. Na verdade, enquanto a API pública não mudar, o ConCAD tende a continuar com seu funcionamento regular para as versões subsequentes do Eclipse.

2. **Reuso**. A ferramenta ConCAD é mais do que um *plugin* do Eclipse que calcula métricas e detecta anomalias de código. É também uma estrutura de análise, que permite facilmente conectar várias análises, desde métricas associadas as regras de detecção para problemas de design e análises complexas baseadas em estruturas estáticas e dinâmicas dos projetos de software. Na literatura foram desenvolvidos uma grande variedade de métricas e técnicas de avaliação de qualidade usando estratégias de detecção. Portanto, o ConCAD é projetado para (i) reutilizar esse vasto número de análises definidos e implementados; (ii) facilitar a criação de novas análises sobre as existentes; e (iii) manter mínima a dependência de análises da plataforma Eclipse.

3. **Extensibilidade**. Decidiu-se por uma arquitetura que permitisse o ConCAD evoluir naturalmente de acordo com as preferências do usuário e as intenções de um desenvolvedor. Uma vez que o cálculo de métricas e as estratégias de detecção são, em sua maior parte, independentes do Eclipse JDT, o ConCAD pode ser facilmente estendido para outras linguagens (e.g., C++ baseado no Eclipse CDT), ou mesmo para outros IDEs. Adicionalmente, pode-se evoluir o ConCAD em outras dimensões através da adição de novas métricas, diferentes estratégias de detecção, granularidade distintas dos problemas detectados entre outros incrementos.

4. **Desempenho** - O ConCAD deve ser executado continuamente no ambiente de desenvolvimento integrado Eclipse. Isso implica dizer que neste ambiente, o desenvolvedor

irá realizar suas atividades de programação e de detecção de anomalias de modo simultâneo e contínuo. A execução destas atividades não deve afetar negativamente a experiência de uso do Eclipse nem onerar os recursos computacionais existentes (i.e., processamento e memória). Esta é uma tarefa extremamente desafiadora, especialmente quando da utilização do ConCAD em projetos de grande escala que apresentem diversos arquivos de códigos para serem analisados e múltiplas instâncias de anomalias de código detectadas.

## D.2 Funcionamento da Ferramenta

Nesta seção demonstra-se de forma prática o funcionamento do ConCAD Eclipse *Plugin*. No que se segue, serão expostos como as anomalias são detectadas e expostas nesta ferramenta (Seção D.2.1), como são configuradas as estratégias de detecção (Seção D.2.2) e as formas de priorização das anomalias detectadas (Seção D.2.3).

### D.2.1 Detecção de Anomalias

Conforme citado anteriormente, esta ferramenta também implementa características de abordagens mais tradicionais de detecção de anomalias de código com intuito de prover uma visão mais global dos resultados associados a detecção das anomalias de código no projeto. No que se segue são descritas a detecção de anomalias de código de acordo com as abordagens de detecção interativa e não-interativa.

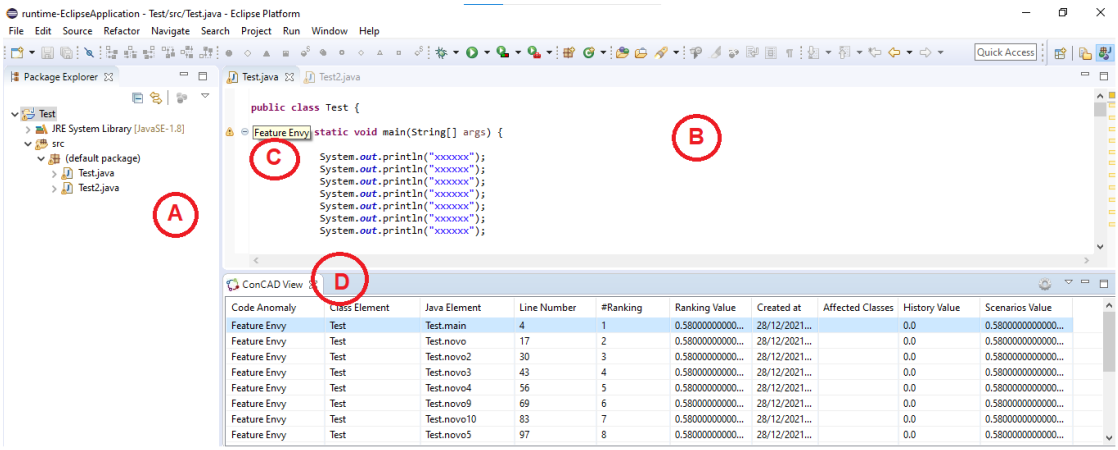


Figura D.2: Visão Geral da Ferramenta *Eclipse ConCAD*.

**Funcionamento Interativo**. Depois que um projeto é carregado no ambiente Eclipse, o usuário pode instruir o ConCAD para identificar e analisar as possíveis anomalias de código de um dado fragmento. Isso é feito clicando no botão "*ConCAD Smell Detector -> Enable ConCAD*" em um menu contextual no projeto (Figura D.2.A). Depois de clicar nesta opção, o usuário muda seu contexto para realizar alguma atividade de programação em um arquivo de código particular. Nesta ocasião o usuário necessita focar suas atividades de programação em fragmentos de código que são parte constituinte do arquivo que encontra-se ativo no Ambiente de Desenvolvimento Integrado (Figura D.2.B).

O usuário realiza alguma atividade de programação no fragmento de código. Neste momento o mecanismo de detecção de anomalias atua - independentemente de uma requisição explícita deste usuário - calculando métricas associadas ao fragmento bem como aos arquivos de código-fonte relacionados. Este mecanismo realiza uma análise detalhada dos valores das métricas e, caso ocorra uma violação nas estratégias previamente definidas para detecção, o desenvolvedor é alertado sobre a presença de anomalias através de marcadores no fragmento de código considerado (Figura D.2.C). Finalmente, o desenvolvedor também pode ter uma visão mais global dos resultados associados às anomalias de código presentes no projeto. Todas essas ocorrências estão listadas e classificadas na *ConCAD View* (Figura D.2.D).

É importante destacar que os marcadores da ferramenta ConCAD são dinâmicos: conforme um novo código é escrito ou modificado, novos marcadores podem aparecer ou os existentes desaparecer. Embora as anomalias sejam detectadas usando regras baseadas em métricas, o problema é descrito em termos de conceitos de projeto, ao invés de números. Nesse sentido o ConCAD lista as entidades e relações reais que contribuem para uma anomalia de código específica, ajudando o usuário da ferramenta na correta identificação e remoção das anomalias.

Por exemplo, no caso de um *Feature Envy*, a regra de detecção afirma que: (i) o método usa muitos atributos externos, de poucas classes e (ii) usa nenhum ou quase nenhum atributo de sua classe. Assim, o ConCAD mostrará os atributos externos reais que são usados, bem como os atributos de sua própria classe que o método está usando (se houver). Além disso, a descrição contém um conjunto de *hiperlinks* que permitem ao desenvolvedor se concentrar em explorar em detalhes o contexto relevante de uma determinada anomalia.

**Funcionamento não-interativo**. Depois que um projeto é carregado no ambiente

Eclipse, o usuário pode instruir o ConCAD para identificar e analisar as possíveis anomalias de código presentes no projeto de modo global. Isso é feito clicando no botão "*ConCAD Smell Detector -> Find Code Smells*" em um menu contextual no projeto (Figura D.2.A). Depois de clicar nesta opção, a ferramenta irá analisar as métricas associadas a todos os elementos de código do projeto em questão. Tão logo o mecanismo de detecção identifique as anomalias presentes no projeto, essas ocorrências serão listadas e classificadas na *ConCAD View* (Figura D.2.D).

O usuário muda seu contexto para realizar alguma atividade de programação em um arquivo de código particular. Nesta ocasião o usuário necessita focar suas atividades de programação em fragmentos de código que são parte constituinte do arquivo de código que encontra-se ativo no Ambiente de Desenvolvimento Integrado (Figura D.2.B). Ao concluir a atividade de programação, o usuário necessita requisitar explicitamente a ferramenta para que o mecanismo de detecção realize suas análises em busca de anomalias de código. Mais importante, nesse modo de funcionamento o usuário é privado de resultados localizados da detecção de anomalias (e.g., marcadores no código). Como consequência, novas ocorrências de anomalias podem ser inseridas e serem detectadas apenas tardiamente quanto muitos módulos dependem ou são afetados por esses trechos anômalos (vide detalhes do cenário de uso da abordagem DNI na Seção 2.3.4). Conforme já discutido, isso dificulta sobremaneira a realização das ações de refatoração, tornando-as mais caras em termos de tempo e esforço.

### D.2.2 Configuração das Estratégias de Detecção

De acordo com o previamente definido, a ferramenta ConCAD provê suporte inicial à detecção de 10 tipos distintos de anomalias de código. Uma descrição mais detalhada a respeito de cada uma dessas anomalias pode ser obtida através do Apêndice C. A ferramenta ConCAD permite que seus usuários selecionem diferentes limiares para as métricas usadas para detecção de anomalias.

Na Figura D.3 pode-se identificar a estratégia de detecção para a anomalia classificada como *God Class*. Esta anomalia possui uma estratégia baseada em três métricas distintas (i.e., ATFD, WMC e TCC) em conjunto com seus valores limiares. Embora o desenvolvedor que instancia a ferramenta possa pré-configurar os limites por padrão, o usuário da ferramenta pode alterar os limites para adaptar as estratégias de detecção para as caracterís-

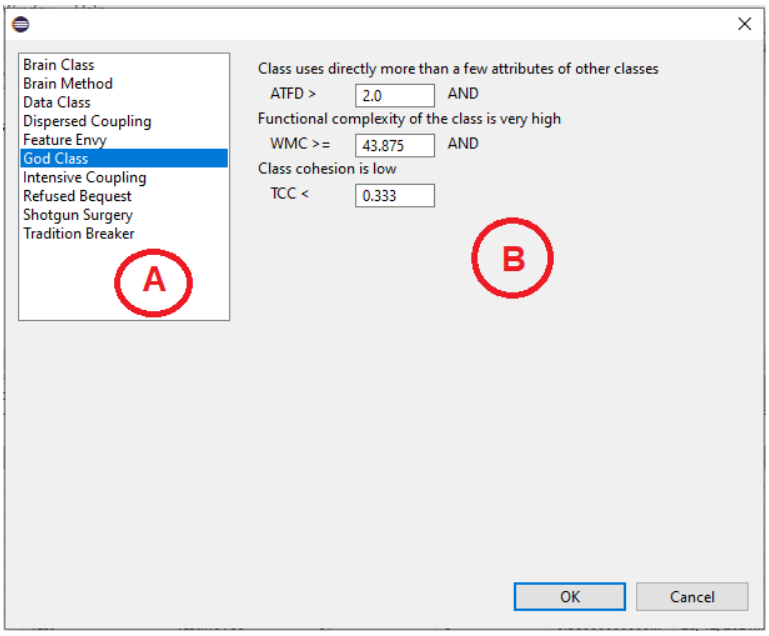


Figura D.3: Definição da Estratégia de Detecção.

ticas do aplicativo que está sendo analisado. Vale destacar que cada uma das 10 anomalias distintas apresentam estratégias de detecção similarmente adaptáveis e configuráveis. Uma descrição mais detalhada a respeito de cada uma dessas estratégias pode ser obtida através do Apêndice B.

### D.2.3 Critérios para Priorização da Detecção

Uma vez que as anomalias de código são descobertas, elas devem ser classificadas de acordo com sua importância. Um desenvolvedor pode instanciar a ConCAD para priorizar anomalias de código por diferentes critérios. A primeira estratégia diz respeito à relevância do tipo de anomalia de código (Figura D.4.A).

O usuário pode assinalar valores (entre 1 e 5) para estabelecer um nível de importância para cada um dos tipos de anomalias suportadas pela ConCAD. Na Figura D.4.B, pode-se notar que o usuário pode facilmente atribuir valores de importância a cada um dos tipos. Esses valores são ordenados posteriormente, indicando a preferência deste usuário.

Outro critério de priorização está associado ao cenário de modificação. Por exemplo, alguns trabalhos mostraram que ao refatorar anomalias de código relacionados a problemas arquiteturais, a degradação da arquitetura poderia ser interrompida [64], [63], [16]. Neste

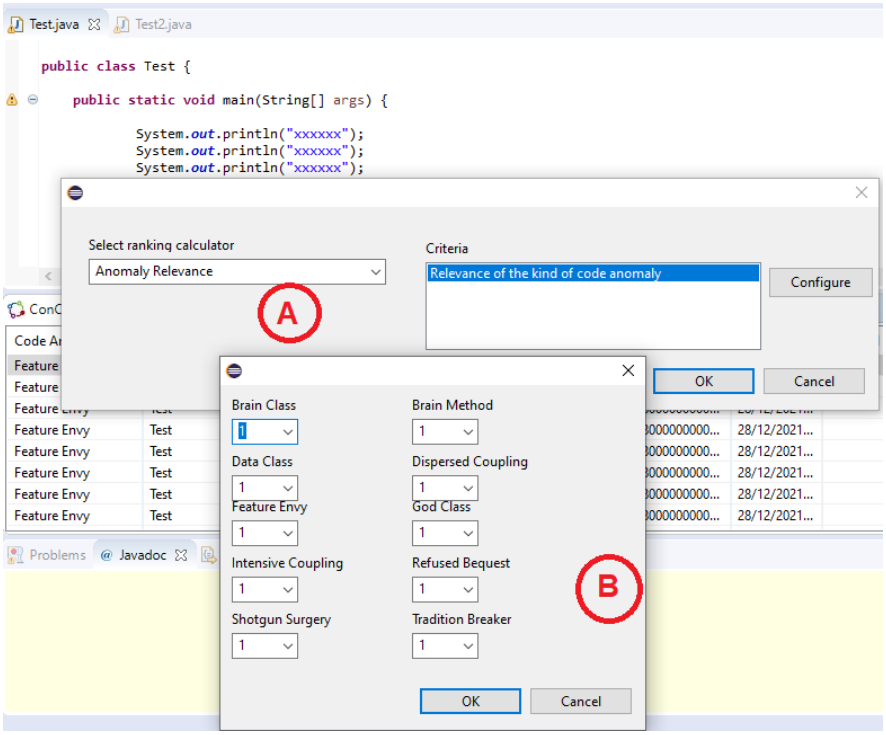


Figura D.4: Priorização por Tipo de Anomalia.

caso, o desenvolvedor pode criar um critério para classificar primeiro aquelas anomalias que comprometem a arquitetura do sistema em um dado cenário de modificação. Essa análise pode ser realizada, por exemplo, definindo um critério com base na relação entre as anomalias de código e os cenários de modificação do sistema. Então, as informações externas incorporadas ao ConCAD tornam-se importante determinar a criticidade de cada instância de anomalias de código.

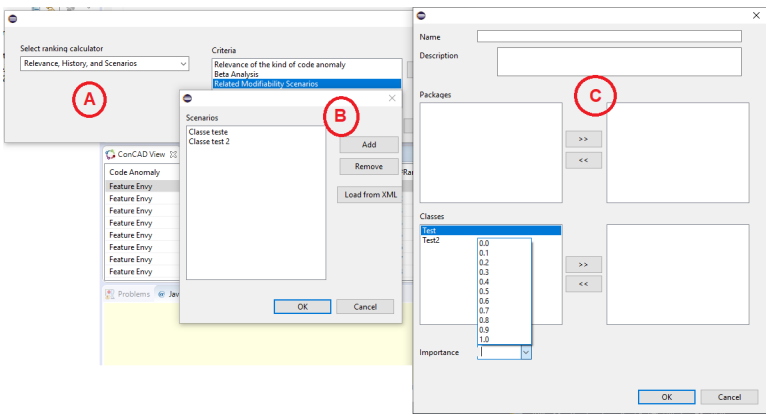


Figura D.5: Priorização por Cenário de Modificação.

Conforme Figura D.5, para esse tipo de priorização, o usuário deve selecionar as classes e pacotes do sistema que compõem o cenário, escolhendo-os de diferentes listas fornecidas pelo ConCAD. Além disso, ele pode selecionar um valor de importância entre [0..1] para indicar quão crítico é a satisfação do cenário.

Finalmente, o usuário poderá priorizar as anomalias de código de acordo com informações históricas do projeto (Figura D.6). Nesse caso o usuário deverá indicar a localização do projeto no computador em uso e carregar a versão para armazenamento dos resultados associados a anomalia de código. As anomalias recebem valores normalizados entre [0..1]. Valores próximos de 1 indicam que a anomalia é remanescente em várias versões do sistema. Valores próximos de 0 indicam que a anomalia foi inserida em versões mais recentes do sistema.

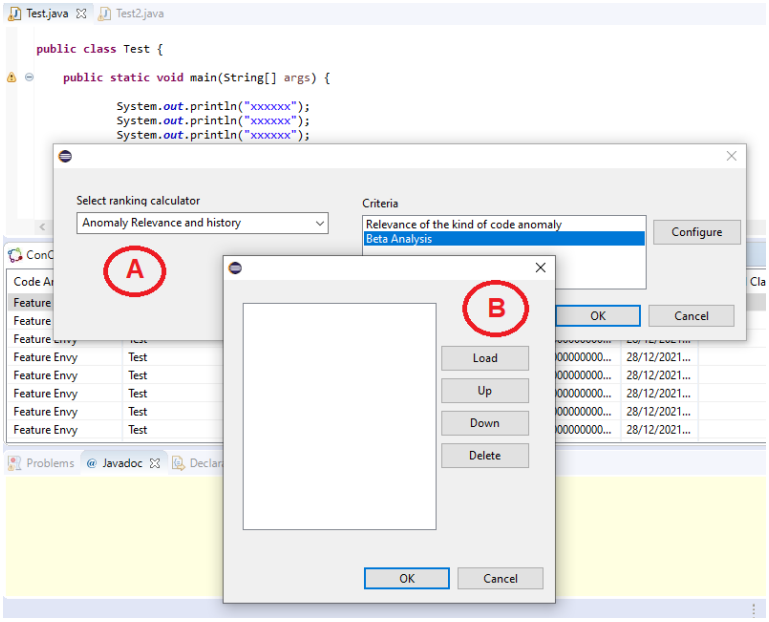


Figura D.6: Priorização Baseada no Histórico.

Como é mostrado, a ferramenta ConCAD é flexível para implementar critérios baseados em diferentes tipos de informações: histórico do código-fonte, *feedback* explícito de uma escala ordinal e um mapeamento de cenários de modificação. É importante salientar que novos e diferentes critérios de priorização podem ser implementados pelo desenvolvedor devido às características de implementação desta ferramenta. Uma discussão detalhada sobre aspectos de priorização de anomalias de código está além do escopo deste trabalho. Contudo, o assunto pode ser analisado com maior nível de detalhes nos Apêndices deste trabalho.

## D.3 Avaliação de Desempenho da Ferramenta

Nesta parte do trabalho são descritos os resultados de um estudo para avaliação do desempenho da ferramenta *Eclipse ConCAD*. Inicialmente, são exibidos os detalhes metodológicos da avaliação (Seção D.4). Em seguida, os resultados dessa avaliação para a ferramenta operando em modo DI (Seção D.5) e DNI (Seção D.6) são apontados e discutidos.

## D.4 Método de Avaliação

Quando se trata de desempenho da utilização da ferramenta no contexto de uma IDE, uma questão é relevante nesse cenário: considerando a complexidade das análises de qualidade, é possível usar a ferramenta ConCAD enquanto ainda executa em paralelo as tarefas usuais de desenvolvimento, sem enfrentar nenhuma desvantagem de desempenho? Essa questão tem duas facetas: (i) quais recursos a ferramenta ConCAD requer em modo interativo? e (ii) quais recursos a ferramenta ConCAD requer ao ser acionado manualmente para analisar de modo não-interativo um projeto inteiro? No que segue serão expostos detalhes a cada um desses dois aspectos.

Nesse contexto avaliou-se o tempo de execução da abertura de um arquivo no editor. Ao abrir um arquivo no Eclipse, um gatilho é acionado com vistas a analisar todos os elementos do programa definidos naquele arquivo, a fim de detectar as diversas anomalias de código. Observe que, embora a ferramenta ConCAD esteja focada apenas na avaliação dos elementos do programa no arquivo atual, o os cálculos podem ter implicações em vários outros arquivos do sistema. Por exemplo, a verificação de "duplicação de código" requer cada método no arquivo para ser comparado com todos os outros métodos em todo o sistema.

Assim, realizou-se essas medições utilizando 05 (cinco) sistemas de código aberto: JHotDraw 9.1[2], Maven 3.8[3], ArgoUML 0.34[4], Vuze 5.7[5] e Eclipse JDT 3.32[6]. Esses projetos foram escolhidos por terem sido utilizados em trabalhos relacionados e por serem de código aberto, facilitando a reprodução das atividades por pesquisadores independentes. É

[2]https://github.com/wrandelshofer/jhotdraw

[3]https://github.com/apache/maven-compiler-plugin/

[4]https://github.com/argouml-tigris-org/argouml/

[5]https://github.com/svn2github/vuze

[6]https://github.com/eclipse-jdt

importante mencionar que os projetos possuem tamanhos distintos, variando de projetos de tamanho médio, de mais de 100KLOC a grandes projetos, que tenham mais de 1,4 MLOC. Ademais, todas as medições de desempenho foram realizadas em um notebook Lenovo IdeaPad, Intel Core i7-6500U de 2,5 GHz, 16 GB de RAM, sistema operacional Windows 10.

## D.5 Desempenho usando ConCAD em DI

A primeira avaliação que foi realizada refere-se à atividade de exibir os resultados de detecção de anomalias de código para o arquivo de código fonte ativo no ambiente de desenvolvimento. Para cada um dos cinco projetos, foram selecionados dois arquivos de origem com base em seu tamanho para realização das atividades experimentais. Mais exatamente para cada projeto, utilizou-se o maior arquivo (Amostra 1) e um arquivo aleatório de tamanho médio (Amostra 2). Usando esses arquivos mediu-se o tempo de execução da ferramenta ConCAD em três cenários para cada um dos projetos:

1. abrir a Amostra 1 imediatamente após iniciar o Eclipse;
2. abrir a amostra 2 imediatamente após iniciar o Eclipse;
3. abrir a Amostra 2 após a Amostra 1, ou seja, abrir um arquivo após a ferramenta já ter analisado outro arquivo.

A razão pela qual exercitamos o terceiro cenário, ou seja, medindo o tempo de execução para o segundo arquivo aberto, é que esperou-se que os tempos de execução da da ferramenta ConCAD nos arquivos subsequentes sejam menores do que os necessários para a análise do primeiro arquivo.

Ao analisar os números resumidos na Tabela D.1, observa-se que o tempo de execução diminui drasticamente para um valor marginal (ou seja, menos de um segundo por arquivo) quando um arquivo é aberto após outro arquivo ter sido analisado. Relacionado ao caso em que o arquivo de tamanho médio é aberto após o arquivo maior, valores semelhantes são obtidos mesmo quando invertendo a ordem de abertura dos arquivos. Também notou-se que quando um arquivo é analisado pela primeira vez, uma grande fração da análise tempo é usado para analisar a duplicação de código. Para projetos de médio porte, o tempo de

execução para abrir o primeiro arquivo no Eclipse juntamente com a ferramenta ConCAD demanda menos de 15 segundos (no máximo), mesmo para o arquivo maior (cenário 1). Há duas observações importantes sobre este atraso inicial:

1. O atraso é esperado considerando as observações anteriores e o fato de que a análise de duplicação de código compara o arquivo contra todos os outros arquivos de origem no projeto. Além disso, observe que a ferramenta ConCAD será capaz de fornecer informações quase instantâneas dos resultados em qualquer outro arquivo aberto. Isso ocorre em virtude dos resultados anteriores serem armazenados em *cache* e, cada vez que uma alteração é feita, apenas as entidades afetadas são removidas do *cache*;

2. A ferramenta ConCAD é executada em um *thread* separada, o que significa que o desenvolvedor nunca é impedido de trabalhar em virtude do atraso inicial no carregamento da ferramenta ConCAD. Assim, os desenvolvedores podem continuar realizando seu trabalho em paralelo com a ConCAD. Atualmente, esta ferramenta usa apenas um *thread* para tarefas de marcação, mas no futuro planejamos usar os recursos multinúcleo do CPUs modernas e ter um número configurável de *threads* executando trabalhos de marcador em paralelo.

Resumidamente, simular uma edição no maior arquivo de cada projeto aciona um trabalho de análise que é executado em cerca de metade do tempo nos três primeiros projetos, enquanto para Vuze e Eclipse JDT, a análise é cerca de 35% mais rápida.

Tabela D.1: Desempenho Utilizando a Ferramenta *Eclipse ConCAD* em DI.

| | **Cenário 1** | **Tempo de execução** | | **Cenário 2** | **Tempo de execução** | | **Cenário 3** |
|---|---|---|---|---|---|---|---|
| | **Maior arquivo** | **sem verif. de duplicação** | **verif. de duplicação** | **Arquivo aleatório** | **sem verif. de duplicação** | **verif. de duplicação** | **verif. de duplicação** |
| JHotDraw 9.1 | 143 KB | 3s | 5s | 50KB | 1s | 5s | 0,1s |
| Maven 3.8 | 188 KB | 3s | 4s | 109KB | 1s | 4s | 0,5s |
| ArgoUML 0.34 | 160 KB | 3s | 7s | 16KB | 2s | 5s | 0,2s |
| Vuze 5.7 | 147 KB | 7s | 14s | 11KB | 1s | 9s | 0,6s |
| Eclipse JDT 3.32 | 439 KB | 8s | 15s | 34KB | 2s | 11s | 0,8s |

## D.6 Desempenho usando ConCAD em DNI

A segunda avaliação que foi realizada refere-se à atividade computacionalmente mais intensiva da ferramenta ConCAD, que é calcular a visão geral dos resultados para um projeto inteiro. Isso significa que as regras de detecção para todas as anomalias de código são executado para todas as classes e métodos no sistema. Identificamos dois fatores principais que têm um impacto potencial no desempenho:

1. a análise de duplicação de código para grandes bases de código é um grande consumidor de tempo e memória; este é um conhecido fato que geralmente é verdadeiro para técnicas de duplicação de código;

2. a memória *heap* alocada para o ambiente Eclipse pode ter uma influência importante, especialmente para sistemas grandes, pois define o limite geral de uso de memória para todos os *plug-ins* , que são executados em um ambiente de trabalho do Eclipse, incluindo a ferramenta ConCAD.

Conscientes destes dois fatores, mediu-se então os tempos de execução em quatro configurações diferentes: com três *heap* diferentes tamanhos de memória (i.e., 1 GB, 2 GB e 4 GB) e para cada tamanho de *heap*, com e sem análise de duplicações de código. Os resultados descritos na Tabela D.2 revela que o tempo de execução cresce quase linearmente com o tamanho dos projetos, a uma taxa de aproximadamente 40 segundos por 100 KLOC. Assim, para projetos de médio porte, a duração é pequena, e mesmo analisando projetos de grande porte, como Eclipse JDT, o tempo de análise ainda permanece razoável.

Tabela D.2: Desempenho Utilizando a Ferramenta *Eclipse ConCAD* em DNI.

| | | **Tempo de execução para visão geral ConCAD** | | | | | | **Mémoria Extra** |
|---|---|---|---|---|---|---|---|---|
| | | **1 GB (heap)** | | **2 GB (heap)** | | **4 GB (heap)** | | **4 GB (heap)** |
| | **Tamanho do Sistema** | **sem verif. de duplicação** | **verif. de duplicação** | **sem verif. de duplicação** | **verif. de duplicação** | **sem verif. de duplicação** | **verif. de duplicação** | |
| JHotDraw 9.1 | 137KLOC | 48s | 57s | 44s | 53s | 42s | 50s | + 240MB |
| Maven 3.8 | 158KLOC | 49s | 62s | 46s | 58s | 43s | 54s | +350 MB |
| ArgoUML 0.34 | 348KLOC | 96s | 113s | 90s | 109s | 85s | 103s | +370MB |
| Vuze 5.7 | 908KLOC | 189s | 225s | 180s | 212s | 168s | 201s | +1.120MB |
| Eclipse JDT 3.32 | 1.441KLOC | - | - | 302s | 357s | 289s | 332s | +1.760MB |

Em relação ao consumo de memória, a última coluna da Tabela D.2 mostra como muita memória adicional (não apenas heap) é usada pelo processo do Eclipse após a execução da

visão geral da ferramenta ConCAD. A memória usada pelo Eclipse antes de executar as análises necessárias para preenchimento da visão geral da ConCAD varia entre 200 MB para o menor projeto e 400 MB para Eclipse JDT. Embora projetos maiores exijam mais memória, o crescimento é quase linear. Além disso, em um ambiente em que grandes projetos de software estão sendo desenvolvidos, as estações de trabalho precisam ser equipadas com mais RAM. Assim, acreditamos que este é um problema menor, já que os requisitos da ferramenta ConCAD estão em sincronia com aqueles que grandes projetos definiriam de qualquer maneira com no que diz respeito à configuração dos computadores.

**Considerações sobre a avaliação da ferramenta**

Em relação a avaliação do desempenho, foi realizado experimentos com a ferramenta *Eclipse ConCAD* para detecção de anomalias de código em projetos de diferentes tamanhos. Foram identificados dois fatores que afetam o desempenho da ferramenta: a análise de duplicação de código e a memória alocada para o ambiente Eclipse. Os resultados mostraram que o tempo de execução aumenta quase linearmente com o tamanho dos projetos, a uma taxa de aproximadamente 40 segundos por 100 KLOC. Em relação ao consumo de memória, a quantidade de memória adicional usada pelo processo do Eclipse após a execução da ferramenta *Eclipse ConCAD* é relativamente pequena, e os requisitos da ferramenta estão em sincronia com aqueles que grandes projetos definiriam de qualquer maneira em termos de configuração de computadores. Em resumo, o estudo mostrou que a ferramenta *Eclipse ConCAD* é eficiente em detectar anomalias de código em projetos de diferentes tamanhos e pode ser usada com sucesso em ambientes de desenvolvimento de software.