UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

KIRT: UMA TÉCNICA DE LEITURA BASEADA EM INFORMAÇÕES-CHAVE E VISUALIZAÇÃO PARA INSPEÇÃO DE CÓDIGO COM SUPORTE COMPUTACIONAL

ANDERSON BELGAMO

São Carlos - SP
Dezembro/2015

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

KIRT: UMA TÉCNICA DE LEITURA BASEADA EM INFORMAÇÕES-CHAVE E VISUALIZAÇÃO PARA INSPEÇÃO DE CÓDIGO COM SUPORTE COMPUTACIONAL

ANDERSON BELGAMO

Tese apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Doutor em Ciência da Computação, área de concentração: Engenharia de Software.
Orientadora: Dra. Sandra Camargo Pinto Ferraz Fabbri.

São Carlos - SP
Dezembro/2015

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária UFSCar
Processamento Técnico
com os dados fornecidos pelo(a) autor(a)

Belgamo, Anderson
B429k KIRT : uma técnica de leitura baseada em informações-chave e visualização para inspeção de código com suporte computacional / Anderson Belgamo. -- São Carlos : UFSCar, 2016.
154 p.

Tese (Doutorado) -- Universidade Federal de São Carlos, 2015.

1. Inspeção de código. 2. Técnicas de leitura. 3. Visualização. 4. Engenharia de software experimental. I. Título.

UNIVERSIDADE FEDERAL DE SÀO CARLOS

Centro de Ciências Exatas e de Tecnologia

Programa de Pós-Graduação em Ciência da Computação

Folha de Aprovação

Assinaturas dos membros da comissão examinadora que avaliou e aprovou a defesa de Tese de Doutorado do candidato Anderson Belgamo, realizada em 10/12/2015.

Profa. Dra. Sandra Camargo Pinto Ferraz Fabbri
(UFSCar)

Prof. Dr. Fabiano Cutigi Ferrari
(UFSCar)

Prof. Dr. Manoel Gomes de Mendonça Neto
(UFBA)

Prof. Dr. Alfredo Goldman Vel Lejbman
(USP)

Profa. Dra. Silvia Regina Vergílio
(UFPR)

Certifico que a sessão de defesa foi realizada com a participação à distância dos membros Prof. Dr. Manoel Gomes de Mendonça Neto, Prof. Dr. Alfredo Goldman Vel Lejbman e Profa. Dra. Silvia Regina Vergílio e, depois das arguições e deliberações realizadas, os participantes à distância estão de acordo com o conteúdo do parecer da comissão examinadora redigido no relatório de defesa do aluno Anderson Belgamo.

Profa. Dra. Sandra Camargo Pinto Ferraz Fabbri
Presidente da Comissão Examinadora
(UFSCar)

Toda conquista é especial e única, mas sua obtenção é conseguida por caminhos não conhecidos e muitas vezes difíceis. Facilitadores desses caminhos são pessoas únicas que nos ajudam e contribuem para que a jornada se torne mais suave. Obrigado a esses facilitadores, especialmente a Dra. Sandra, grande responsável por tornar essa conquista prazerosa e mais suave.

AGRADECIMENTOS

Agradeço ao bom Deus por me abençoar e permitir que eu levante todos os dias com saúde e paz.

A Dra. Sandra Fabbri, que sempre acreditou em mim, me aceitando para ser novamente seu orientado. Todo esse tempo de convívio, desde o mestrado até esse momento, foi enriquecedor e de um aprendizado que jamais teria se não fosse sua orientação. Obrigado pelo carinho, compreensão, amizade e pelo compartilhamento de conhecimento.

Agradeço aos avaliadores da minha qualificação de doutorado, por terem incentivado a continuidade da pesquisa e mostrarem novos caminhos. Antecipo meus agradecimentos aos avaliadores desta tese.

Também agradeço o prof. Manoel Mendonça pelas ideias apresentadas durante reuniões do projeto Universal na qual este projeto estava vinculado.

Um especial agradecimento aos meus pais, Wanderley e Claudia, meu irmão Alessandro e minha vó Maria José (*in memoriam*), pelo amor, carinho, ajuda e exemplos de vida. Vocês foram, são e sempre serão referências para mim. A minha cunhada Juliana e minhas lindas sobrinhas, Brenda e Alyssa. Obrigado por fazer o tio Dinho muito feliz. Tenham certeza que vocês são parte integrante deste e outros novos momentos que virão. À minha Juliana, pela compreensão da ausência, das dificuldades encontradas e das superações obtidas.

A todos os colegas do LaPES com quem convivi nesses anos, em especial a Elis Hernandes, André Di Thommazo, Erik Antonio, Kamila Camargo, Juciara Nepomuceno, Fábio Octaviano, Rafael Gastaldi, Bento Siqueira e André Abade. Não poderia deixar de fora pessoas fantásticas como Augusto Zamboni e Cleiton Silva, os quais me auxiliaram no desenvolvimento da ferramenta CRISTA. Obviamente, um agradecimento ao pai da ferramenta, Daniel Porto, pela competência e excelente qualidade de código.

Aos colegas do IFSP Câmpus Piracicaba por me ajudar inúmeras vezes nas substituições de aulas e sempre me auxiliando em todas as decisões. Meu muito obrigado.

Aos colegas da Universidade Metodista de Piracicaba (UNIMEP), aos quais tive a honra de ser aluno e posteriormente colegas de profissão.

Aos alunos e profissionais da indústria de software que participaram dos estudos experimentais realizados.

Aos professores e funcionários do Departamento de Computação da UFSCar.

CERTEZA

De tudo ficaram 3 coisas:
A certeza de que estamos sempre começando;
A certeza de que precisamos continuar;
A certeza de que seremos interrompidos antes de terminar.
Portanto devemos:
Fazer da interrupção um novo caminho,
Da queda, um passo de dança,
Do medo, uma escada,
Do sonho, uma ponte,
Da procura, um encontro.

Fernando Pessoa

RESUMO

Contexto: A inspeção de código é uma importante atividade para garantia da qualidade de software, mostrando, às vezes, desempenho superior em relação à atividade de teste. Sua aplicação depende da utilização de técnicas de leitura que auxiliem o inspetor tanto na compreensão do código como na condução da atividade em si. Apesar de existirem algumas técnicas de leitura de código que dão suporte à inspeção, a *Stepwise Abstraction*, por exemplo, foi definida para o paradigma procedimental e exige que até as partes mais simples sejam abstraídas; outras técnicas como *Checklist*, Baseada em Casos de Usos, etc., são suscetíveis à experiência do inspetor, pois não possuem um procedimento sistemático e, além disso, não contam com recursos facilitadores para sua aplicação. Uma iniciativa de suporte computacional com o facilitador de visualização agregado à técnica *Stepwise Abstraction* está implementada na ferramenta CRISTA, desenvolvida em outro trabalho deste grupo de pesquisa, que mostrou ser uma abordagem efetiva e eficiente para inspeção de código. **Objetivo**: Apresentar a técnica de leitura KIRT (*Key-Information Reading Technique*) que foi definida por meio de cenários operacionais, com o propósito de estabelecer diretrizes sistemáticas para realizar inspeção de código. Ela foi baseada em padrões de leitura extraídos da análise do comportamento de inspetores, contemplando particularidades do paradigma orientado a objetos, e fazendo uso de metáforas visuais que guiam o inspetor nas tarefas a serem realizadas. **Metodologia**: Considerando a necessidade de compreender como a inspeção de código é realizada na prática, a pesquisa foi baseada por meio de quatro ciclos investigativos que serviram de base para a definição e para a avalição da técnica de leitura KIRT. Em cada ciclo executaram-se estudos que geraram resultados parciais que embasaram os ciclos subsequentes. A técnica KIRT foi definida com o propósito principal de identificação de defeitos e, adicionalmente, de melhorias no código-fonte. Após definida, três estudos experimentais foram realizados: (i) para avaliar a contribuição da metáfora visual implementada na CRISTA, para a identificação de defeitos; (ii) para avaliar a técnica de leitura KIRT usando duas métricas distintas em relação à efetividade e eficiência na identificação de defeitos e melhorias no código; (iii) para avaliar a técnica de leitura KIRT usando uma métrica específica em relação à técnica *Checklist*. **Resultados:** os resultados dos experimentos mostraram que (i) a metáfora visual influenciou em um maior número de melhorias de código identificadas e um menor número de falso-positivos; (ii) a técnica KIRT com as duas métricas distintas utilizadas são complementares; (iii) a efetividade e eficiência da técnica KIRT foram melhores que a técnica *Checklist* da literatura. **Conclusão**: Nos diferentes estudos em que foi avaliada, a técnica KIRT apresentou resultado satisfatório em relação à efetividade e eficiência na identificação de defeitos, gerando baixo número de falso-positivos. Além disso, ela se mostrou efetiva na identificação de melhorias no código. Com esses resultados pode-se dizer que a técnica KIRT constitui uma alternativa para a inspeção de código na prática, uma vez que contam com apoio da CRISTA, que oferece suporte para os procedimentos definidos, com o facilitador de metáforas visuais. Consequentemente, a aplicação da técnica KIRT pode promover a qualidade geral do software.

Palavras-chave: inspeção de código, técnicas de leitura, visualização, engenharia de software experimental.

ABSTRACT

Background: Code inspection is an important activity to assure the software quality that, in some cases, shows a better performance than the testing activity. Its application depends on the use of reading techniques that assist the inspector both in understanding the code and in executing the inspection. Although there are some reading techniques that support code inspection, Stepwise Abstraction, for example, was set to the procedural paradigm and requires that even the simplest code be abstracted. Other techniques such as Checklist, Use Case based, etc., are prone to the inspector's experience since they do not provide a systematic procedure and they are not supported by facilitating resources. A computer-aided code inspection providing visualization facilities for applying Stepwise Abstraction – the CRISTA tool - was developed in another work of this research group. This tool has shown to be an effective and efficient approach for code inspection. **Objective**: To present the reading technique KIRT (Key-Information Reading Techniques) that was defined through operational scenarios, in order to establish systematic guidelines for conducting the code inspection. KIRT was based on reading patterns obtained from the analysis of the inspectors' behaviors, considering features of the object-oriented paradigm, and using visual metaphors that guide the inspector during the inspection. **Method**: Considering the intention of understanding how the code inspection is carried out in practice, the research was based evolutive cycles, performed through four cycles that were the basis for defining and evaluating KIRT. In each cycle, studies were performed to generate subsidies for supporting subsequent cycles. The KIRT technique was set with the main purpose of identifying defects and, in addition, source code improvements. Once defined, three experimental studies were conducted to: (i) evaluate the contribution of the visual metaphor implemented in CRISTA tool for defects identification; (ii) evaluate KIRT technique (using two different metrics) regarding the effectiveness and efficiency in the identification of defects and source code improvements; (iii) evaluate KIRT technique in relation to the Checklist technique. **Results**: The results of the experimental studies showed that (i) the visual metaphor influenced the number of source code improvements and false positives identified; (ii) KIRT technique (instantiated by two distinct metrics) was complementary; (iii) the effectiveness and efficiency of the KIRT technique were better than Checklist. **Conclusion**: In the experimental studies the KIRT technique showed satisfactory results regarding the effectiveness and efficiency in identifying defects, and generated few false positives. Moreover, the KIRT technique identified more source code improvements than Checklist. Based on the results we can say that KIRT technique can be used for code inspection in practice, mainly because they are supported by CRISTA, which provides procedures and visual metaphors for conducting the code inspection activity. Consequently, if they reach the objective of identifying defects, we can say that they contribute to the overall quality of the software.

Keywords: code inspection, reading techniques, visualization, experimental software engineering.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

Capítulo 1

INTRODUÇÃO

Este capítulo apresenta o contexto do trabalho que está relacionado à inspeção de código e as dificuldades envolvidas com essa atividade que geraram a motivação para o desenvolvimento do trabalho. Além disso, justifica-se o uso de uma metodologia de trabalho evolutiva, uma vez que as investigações foram feitas em ciclos evolutivos baseadas em experimentação.

1.1 Contexto

O uso de software está cada vez mais presente no cotidiano de qualquer pessoa, desde uma ligação telefônica até a realização de um procedimento cirúrgico. Em decorrência disso, a exigência por qualidade tem sido cada vez maior. Assim, o desenvolvimento de qualquer software deve seguir critérios de qualidade e ser fundamentado nos princípios da Engenharia de Software.

A Qualidade de Software, segundo Pressman (2010), é a conformidade do software com: os requisitos funcionais e de desempenho; os padrões e convenções de desenvolvimento pré-estabelecidos; e os atributos implícitos que todo software desenvolvido profissionalmente deve possuir.

Assim, tendo como objetivo a qualidade, existe um conjunto de atividades, denominado Atividades de Garantia de Qualidade de Software, que deve acompanhar todo o processo de desenvolvimento. Dentre essas atividades estão as de V&V (Verificação e Validação), como por exemplo, inspeção e teste de software. Ambas têm por objetivo melhorar a qualidade do artefato em questão, identificando defeitos que possam estar presentes nesses artefatos, deixando-os com um nível de qualidade determinado pela equipe de desenvolvimento. O teste de software é considerado uma atividade dinâmica, pois implica na execução do artefato que está sendo avaliado com o objetivo de encontrar defeitos (MYERS, 2004). Já a inspeção de software é considerada uma atividade de análise estática

que tem o benefício de ser aplicável a qualquer artefato produzido ao longo do processo de desenvolvimento (AURUM; PETERSSON; WOHLIN, 2002).

O termo inspeção de software foi introduzido no trabalho de Fagan (1986) e é considerado como uma revisão formal de artefatos de software e geralmente está associada a um processo de inspeção formal, com papéis e fases de execução bem definidas. O trabalho de Hernandes *et al.* (2013) apresenta um comparativo de diversos processos formais de inspeção.

Se tomarmos como referência o código-fonte, por exemplo, apesar de ser natural avaliar a qualidade do código por meio de atividades de teste, a atividade de inspeção também pode ser aplicada e, em alguns momentos, se mostra inclusive mais efetiva do que a própria atividade de teste. O trabalho de Wilkerson *et al.* (2002) mostra uma ocorrência desse fato, pois é apresentado um experimento comparando a inspeção de código com desenvolvimento dirigido a testes (sigla TDD, em inglês) e os resultados mostram que a inspeção de código é mais efetiva do que TDD na redução de defeitos, apesar de ter um custo maior para ser implementada. Já o trabalho de Morales *et al.* (2015) mostra o impacto que as práticas de inspeção de código geram na qualidade do software, especialmente levando em consideração os aspectos de cobertura do código, participação e experiência dos inspetores.

Assim, dada a reconhecida contribuição da inspeção para a qualidade do software, em particular para o código, essa atividade é o foco deste trabalho.

Para que a atividade de inspeção de software possa ser aplicada, ela deve fazer uso de técnicas denominadas técnicas de leitura, que devem, segundo Basili *et al.* (1996), possuir as seguintes características:

- Ser associada com um documento em particular (por exemplo, documento de requisitos) e a notação na qual o documento é escrito (por exemplo, texto em Português).
- Ser adaptável, baseada nas características do ambiente e do projeto. Assim, se o domínio do problema mudar, a técnica de leitura também deve mudar.
- Ser detalhada, fornecendo ao leitor um processo bem definido.
- Ser específica no sentido de que cada leitor tenha um propósito particular ou um objetivo para ler o documento e os procedimentos apoiem esse objetivo. Isto pode variar de projeto para projeto.
- Ser estudada empiricamente para determinar se ela é efetiva e em quais condições.

Em 1996 Basili *et al.* (1996) publicaram uma árvore que representa uma taxonomia das técnicas de leitura. Essa árvore foi atualizada por Shull *et al.* (2003) e Belgamo (2004), e está apresentada na Figura 1.1.

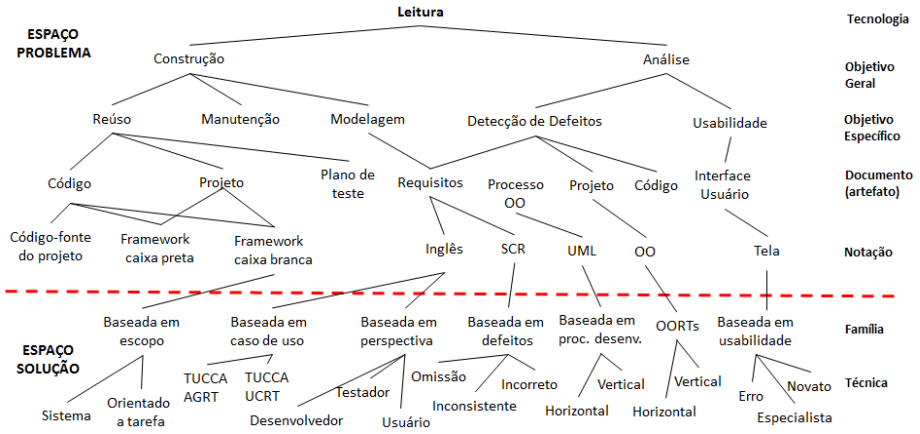


Figura 1.1. Família de Técnicas de Leitura (adaptada de Basili *et al.* (1996))

Percebe-se na Figura 1.1 que existe uma divisão (linha tracejada) entre o espaço do problema (problemas que podem ser direcionados pela leitura) e o espaço da solução (soluções específicas a serem direcionadas para problemas particulares). À direita da Figura 1.1 existe a classificação de cada nível da árvore (tecnologia, objetivo geral, etc.), sendo que estes fornecem as características para as quais a leitura pode ser aplicada.

A primeira ramificação das famílias de técnicas de leitura consiste da leitura para construção e leitura para análise.

A leitura para construção tem por objetivo responder a seguinte questão: Dado um sistema existente, como é possível entendê-lo para usá-lo como parte de um novo sistema? Observa-se, portanto, que a leitura para construção é importante para a compreensão do que o sistema faz, quais requisitos existem e não existem, ajudando abstrair a informação importante do sistema (BASILI *et al.*, 1996).

A leitura para análise tem por objetivo responder a seguinte pergunta: Dado um documento, como são avaliadas as características de qualidade desse documento? A leitura para análise auxilia no entendimento e verificação dos artefatos produzidos ao longo do processo de desenvolvimento, na melhoria das técnicas de desenvolvimento de artefatos de software e, principalmente, no auxílio para encontrar e entender os vários tipos de defeitos procurados (BASILI *et al.*, 1996).

Observa-se na Figura 1.1 que o ramo de identificação de defeitos de código não apresenta registros no "espaço solução". No entanto, em um mapeamento sistemático realizado encontraram-se nove técnicas de identificação de defeitos em código mencionadas

na literatura e que poderiam ser inseridas na família de técnicas de leitura da Figura 1.1 no ramo de *Leitura para Análise –Detecção de Defeitos – Código*.

Independentemente da utilização de uma técnica de leitura para auxiliar na identificação de defeitos do processo de inspeção, qualquer artefato inspecionado deve ser compreendido. No caso de código-fonte, Tao *et al.* (2012) mostram que desenvolvedores gastam uma quantidade significativa de tempo para compreender mudanças do software.

Assim, na Engenharia de Software existe uma área de estudo denominada Compreensão de Software que, segundo Vinz e Etzkorn (2006), se refere a qualquer atividade que usa métodos estáticos ou dinâmicos para revelar propriedades do software. Em geral, a compreensão de software auxilia no entendimento de sistemas de software existentes e, segundo (LETOVSKY, 1987), não existe um modelo robusto que possa explicar o comportamento envolvido durante a compreensão do software, pois, dependendo do problema a ser resolvido, vários modelos de compreensão podem ser utilizados.

Nesse sentido, técnicas de leitura para inspeção de software devem possuir características que também auxiliem o inspetor na compreensão do artefato a ser inspecionado. Assim, durante a compreensão do artefato a ser inspecionado surge uma oportunidade para identificação de melhorias no código.

No caso do artefato inspecionado ser o código-fonte, uma das técnicas de leitura mais usadas é a dirigida à abstração ou *Stepwise Abstraction* (LINGER; MILLS; WITT, 1979). Essa técnica aplica um procedimento específico para a compreensão do código, de forma que sua funcionalidade possa ser abstraída de maneira sistemática, começando das estruturas mais internas e caminhando para as estruturas mais externas.

Embora possa ser utilizada para a compreensão e inspeção de código, a técnica *Stepwise Abstraction* foi desenvolvida em uma época em que o paradigma de desenvolvimento era apenas o procedimental, no qual o software era decomposto funcionalmente. Atualmente, considerando a evolução dos paradigmas de programação e, mais especificamente, a utilização do paradigma orientado a objetos, surgiram novos desafios para a inspeção de código-fonte relacionados às próprias características desse paradigma. Dessa forma, a técnica *Stepwise Abstraction* deixou de atender todas as características dos novos paradigmas.

Nos trabalhos de (DUNSMORE; ROPER; WOOD, 2000a), (DUNSMORE; ROPER; WOOD, 2000b) e (DUNSMORE; ROPER; WOOD, 2003) foram identificados alguns desafios com a inspeção de código orientado a objetos, relatados a seguir:

- Complexidade: o software desenvolvido atualmente possui alto grau de complexidade.
- O problema da distribuição da funcionalidade (*delocalização*): a funcionalidade relacionada ao código-fonte não é contígua e está "espalhada" entre as diversas classes do software.
- *Chuncking*: decorrente do problema anterior, o código "espalhado" entre as classes caracteriza um alto grau de interdependência entre elas, dificultando a escolha de quais trechos de código devem ser inspecionados.
- Estratégia de leitura: ler sistematicamente todo código orientado a objeto em todo contexto de execução possível é impraticável.

No entanto, ressalta-se que, independentemente dos desafios trazidos pelo paradigma de programação, a inspeção é considerada uma atividade efetiva (DENGER; KOLB, 2006), mesmo quando aplicada de forma manual, tornando-a demorada, propensa a erros e trabalhosa. Assim, se mesmo nessas condições a inspeção ainda é vantajosa, o suporte computacional é justificável para minimizar o tempo, diminuir a possibilidade de erros e o trabalho.

Nesse contexto, uma ferramenta deve dar suporte ao inspetor durante a própria leitura do código, como também no tratamento dos defeitos identificados. Além disso, como a atividade de inspeção envolve a compreensão do código, recursos que facilitem a atividade de compreensão também são relevantes e essenciais para a conformidade a um processo de inspeção mais sistemático.

Uma ferramenta que foi desenvolvida com esse propósito, no contexto deste grupo de pesquisa é a CRISTA (*Code Reading Implemented with Stepwise Abstraction*), implementada por um trabalho de mestrado. Essa ferramenta dá apoio à inspeção de código com o uso da técnica de leitura *Stepwise Abstraction* com suporte de visualização para facilitar a atividade de compreensão (PORTO; MENDONÇA; FABBRI, 2009). Apesar da ferramenta estar instanciada para diferentes linguagens de programação (Java, C, C++ e Cobol) e de ter se mostrado bastante efetiva para a realização de um processo de inspeção, a versão original possuía algumas limitações que restringiam o seu uso em um ambiente real de trabalho. Por exemplo, a ferramenta possibilitava a leitura e a visualização de código restrita a uma classe por vez e não possuía uma visualização de todas as classes de um projeto e os relacionamentos existentes. Além disso, um dos estudos conduzidos por Porto *et al.* (2009) mostrou que a técnica *Stepwise Abstraction* pode não ser efetiva em termos práticos uma vez

que ela exige que todo o código seja abstraído, inclusive até as estruturas mais simples cuja compreensão é considerada óbvia.

Assim, para tornar a ferramenta passível de uso em um ambiente real de desenvolvimento de software, várias frentes de pesquisa precisavam ser trabalhadas, como por exemplo, a visualização do relacionamento de todas as classes de um projeto, a abertura de todas as classes de um projeto, a definição de técnicas de leitura mais eficientes que guiassem o inspetor na atividade de inspeção de código com o suporte de visualização.

Em resumo, são esses tópicos comentados anteriormente que caracterizam o contexto em que este trabalho está inserido: (i) a relevância da atividade de inspeção de código para a qualidade de software; (ii) a necessidade de técnicas de leitura para que a inspeção seja efetiva e eficiente; (iii) a utilidade da visualização para compreender o código que precisa ser inspecionado; e (iv), a existência de uma ferramenta de apoio à inspeção com essas características, mas que precisava ser melhorada para se tornar viável em termos práticos.

1.2 Motivação e objetivos

O contexto apresentado anteriormente gerou a motivação de desenvolver técnicas de leitura que pudessem contribuir com o progresso do estado da arte de inspeção de código, de modo a oferecer técnicas de leitura sistemáticas que tratassem os aspectos inerentes do paradigma orientado a objetos.

Com base na motivação apresentada estabeleceram-se os seguintes objetivos para o desenvolvimento deste trabalho:

- Identificação de informações sobre o código que possam ser utilizadas pelos inspetores durante o processo de inspeção. Neste objetivo pretendeu-se identificar quais informações podem ser úteis para direcionar a leitura do código com o propósito de identificação de defeitos;
- Exploração e uso de diferentes formas de visualização para apoiar a compreensão de código com o propósito de identificar defeitos. Neste objetivo pretendeu-se verificar se a visualização de código-fonte pode auxiliar no processo de inspeção de código;
- Definição de técnicas de leitura de inspeção de código-fonte com suporte de visualização e outros recursos de informações. Neste objetivo combinaram-se as informações obtidas nos dois objetivos anteriores;

- Evolução da ferramenta CRISTA para incorporar as técnicas definidas. Neste objetivo concretizaram-se os resultados dos objetivos anteriores, de forma a deixar disponível para a comunidade de engenharia de software uma ferramenta que possa tornar a atividade de inspeção mais eficiente e efetiva. Espera-se que havendo uma ferramenta com essas características disponível para o uso, os desenvolvedores de software possam usá-la na prática com o objetivo de melhorar a qualidade de seus produtos.

Assim, com base na motivação e nos objetivos descritos, a tese deste trabalho de doutorado pode ser redigida da seguinte forma:

"Técnica de leitura para inspeção de código baseada em visualização e informações do próprio código contribuem para a identificação de defeitos e consequentemente para a melhoria da qualidade geral do software"

1.3 Metodologia de pesquisa

Considerando que as informações de código que pudessem facilitar a atividade de inspeção deveriam ser investigadas experimentalmente para identificar como usa-las em conjunto com os recursos de visualização, optou-se por adotar uma metodologia de trabalho evolutiva, a qual é composta das seguintes etapas:

1. Motivação: consiste em identificar evidências que possibilitem a evolução da pesquisa;
2. Planejamento: consiste na definição das ações de como os elementos devem ser investigados;
3. Método: consiste na implementação das ações planejadas;
4. Análise dos Resultados: consiste na análise e discussão dos resultados e extração de evidências a serem explorados nos próximos ciclos.

Assim, foram realizados quatro ciclos compostos das etapas apresentadas anteriormente.

- **Ciclo 1 (apresentado no Capítulo 3)**
    1) Motivação: constatação (ou não) da contribuição sobre o uso de visualização e da técnica *Stepwise Abstraction* durante atividade de inspeção de código em relação à efetividade e eficiência.
    2) Planejamento: análise dos resultados de um experimento controlado (conduzido anteriormente a este trabalho) avaliando o uso da visualização e da

técnica *Stepwise Abstraction* implementada na ferramenta CRISTA em relação a uma abordagem *ad-hoc*.

3) Método: tabulação e aplicação de análises estatísticas dos dados do experimento.
4) Análise dos Resultados: constatação das seguintes evidências – (i) o uso da visualização e da técnica *Stepwise Abstraction* permite uma melhor compreensão do código por parte dos inspetores; (ii) a necessidade de abstração de todo e qualquer trecho de código torna o processo mais demorado o que, eventualmente, pode impactar a relação custo-benefício agregada à atividade de inspeção; (iii) inspetores relataram dificuldade na tomada de decisão de qual classe deveria ser a primeira a ser inspecionada, além da definição da ordem de inspeção das demais classes;

Como resultado deste ciclo, tem-se a publicação do artigo "*Code Inspection Supported by Stepwise Abstraction and Visualization: An Experimental Study*" no *International Conference on Enterprise Information System*s em 2014.

- **Ciclo 2 (apresentado no Capítulo 4)**
  1) Motivação: com base nas evidências (i) e (ii) do Ciclo 1, decidiu-se realizar uma investigação mais detalhada sobre técnicas de leitura existentes para inspeção de código.
  2) Planejamento: definição de um mapeamento sistemático com o objetivo de identificar técnicas de leitura para inspeção de código, bem como suas particularidades no uso de artefatos de software auxiliares, métricas, etc.
  3) Método: aplicação do mapeamento sistemático planejado nas bases Scopus, IEEE, ACM, Web of Science.
  4) Análise dos Resultados: constatação das seguintes evidências – (i) identificação de nove técnicas de leitura de código, com maior incidência para a técnica Checklist; (ii) dificuldade de adaptação das técnicas encontradas em relação ao paradigma orientado a objetos; (iii) baixo uso de visualização em técnicas de inspeção de código; (iv) uso de documento de requisitos e diagramas para apoiar a atividade de inspeção.

Como resultado desse ciclo, tem-se a publicação dos artigos "*Experimental studies in software inspection process: a systematic mapping*" no *International Conference on Enterprise Information Systems* em 2013 e "*An overview of experimental studies on software inspection process*" no *Lecture Notes in Business Information Processing* em 2014. Além disso, salienta-se que as técnicas obtidas foram estudadas com o intuito de servirem de base para a definição de novas técnicas e para o levantamento de quais informações-chave são utilizadas durante a aplicação de cada uma delas, o que foi investigado no próximo ciclo.

- **Ciclo 3 – (apresentado no Capítulo 5)**
  1) Motivação: investigar se o uso de informações-chave como métricas de software, diagramas UML e documento de requisitos são úteis durante a atividade de inspeção.
  2) Planejamento: definição de um estudo exploratório por meio de um experimento.
  3) Método: aplicação do experimento envolvendo alunos dos cursos de graduação em Ciência da Computação e Engenharia da Computação da Universidade Federal de São Carlos.
  4) Análise dos Resultados: constatação das seguintes evidências – (i) identificação de 15 evidências relacionadas ao comportamento e desempenho dos inspetores utilizando informações-chave; (ii) inspetores que utilizaram informações-chave, especialmente métricas de software, apresentaram um melhor desempenho na identificação de defeitos.

Como resultado deste ciclo, tem-se a submissão do artigo intitulado "*Using auxiliary artifacts during code inspection activity: findings from an exploratory study*" para um periódico especializado. Além disso, as evidências identificadas neste ciclo, juntamente com as características das técnicas de inspeção de código identificadas pelo mapeamento sistemático (ciclo 2), subsidiaram a criação de técnica de leitura com o uso de fontes de informações e visualização do código, conforme apresentado no próximo ciclo.

- **Ciclo 4 – (apresentado nos Capítulos 6 e 7)**

  As atividades desse ciclo estão divididas em dois capítulos: no Capítulo 6 relata-se a motivação com a apresentação da técnica de leitura KIRT definida neste trabalho e no Capítulo 7 relatam-se as demais atividades do ciclo, no qual são apresentados os estudos experimentais conduzidos para avaliação da técnica KIRT.

  1) Motivação: formalização da técnica KIRT com base nas evidências obtidas nos ciclos anteriores e sua avaliação.
  2) Planejamento: definição da técnica KIRT, descrição das novas funcionalidades da ferramenta CRISTA e definição de três estudos experimentais para avaliação da técnica.
  3) Método: formalização da técnica KIRT por meio de um cenário operacional, apresentação de exemplos que ilustram as funcionalidades implementadas no contexto deste trabalho e descrição detalhada de três de estudos experimentais para avaliar a técnica de leitura KIRT.
  4) Análise dos Resultados: os três estudos experimentais realizados apresentaram resultados satisfatórios e mostraram que a técnica de leitura KIRT auxilia, de fato, não apenas na identificação de defeitos, mas também, indiretamente, na identificação de melhorias no código, o que certamente pode levar à melhoria da qualidade do software.

Os resultados deste ciclo foram analisados e utilizados em um artigo submetido para um periódico especializado da área.

1.4 Organização

Esta tese está organizada em oito capítulos. Este capítulo apresentou o contexto no qual a proposta da pesquisa está inserida, bem como a motivação, os objetivos e a metodologia de pesquisa seguida.

No Capítulo 2 é apresentada a Fundamentação Teórica dos temas que estão relacionados à pesquisa, como técnicas de inspeção de código e suporte de visualização.

No Capítulo 3 é apresentada uma avaliação de um estudo experimental realizado com o objetivo de avaliação do uso da técnica de leitura *Stepwise Abstraction* e recursos de visualização presentes na ferramenta CRISTA.

No Capítulo 4 é apresentado um mapeamento sistemático para identificação de

técnicas de leituras utilizadas.

No Capítulo 5 é apresentado um estudo exploratório como objetivo de avaliar quais fontes de informação e métricas são mais comumente usadas por revisores durante a inspeção de código.

No Capítulo 6 é apresentada a técnica de leitura KIRT e a ferramenta *CRISTA*, que viabiliza sua aplicação.

No Capítulo 7 são apresentados os estudos experimentais conduzidos para avaliar a pesquisa realizada.

Finalmente, no Capítulo 8 são apresentadas as conclusões, contribuições e limitações da pesquisam, assim como as lições aprendidas e oportunidades futuras. Uma lista com as publicações obtidas durante essa pesquisa de doutorado encerra o capítulo.

No Apêndice A é apresentado o *checklist* utilizado pela técnica KIRT.

Capítulo 2

FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta as teorias que embasam este trabalho e a revisão da literatura científica referente aos principais temas relacionados a esta pesquisa. Essa revisão concentrou-se na investigação de técnicas de inspeção de código e nas suas características bem como na identificação de metáforas visuais para representação de código-fonte.

2.1 Considerações iniciais

Considerando o contexto de inspeção de código e a necessidade de técnicas de leitura que auxiliem o processo de inspeção de código, este capítulo tem como objetivo apresentar a fundamentação teórica da área de inspeção de software, especialmente a inspeção de código. Além disso, tem como objetivo apresentar os fundamentos de visualização de informações e algumas metáforas de visualização de código-fonte que podem ser utilizadas durante a inspeção de código.

Dessa forma, este capítulo está organizado da seguinte maneira: na Seção 2.2 são apresentados conceitos sobre inspeção de software e técnicas de inspeção de código; na Seção 2.3 são apresentados os conceitos sobre visualização de código; na Seção 2.4 é apresentada a ferramenta CRISTA e suas principais funcionalidades. Por fim, na Seção 2.5 são apresentadas as considerações finais deste capítulo.

2.2 Processo de Inspeção de Software

O processo de Inspeção de software foi criado em 1972 por Fagan, para a IBM, com o objetivo de melhorar a qualidade de software e aumentar a produtividade dos programadores. Trata-se de um método de análise estática para verificar se os produtos gerados pelo processo

de desenvolvimento de software satisfazem o cliente/usuário (FAGAN, 1986) (FAGAN, 1999).

O processo de Inspeção parte do princípio que a detecção de defeitos logo nas primeiras fases de desenvolvimento do software pode garantir um menor tempo gasto em retrabalho, assegurando um software que tenha uma maior probabilidade de atender aos requisitos do usuário, de ser entregue no prazo estabelecido, de cumprir a estimativa de custo estipulada, etc. A Figura 2.1 apresenta os dados de uma situação relatada em (WHEELER; BRYKCZYNSKI; MEESON JR., 1996), na qual os números entre parênteses representam a quantidade de defeitos passados para a próxima fase sem o processo de Inspeção e os números fora dos parênteses mostram a quantidade de defeitos passados para a próxima fase utilizando-se o processo. De acordo com esse exemplo, pode-se perceber que o retrabalho quando se usou o processo de inspeção deve ter sido reduzido, pois a quantidade de defeitos ao longo do processo também foi reduzida.

Embora o custo de implantação e execução do processo de inspeção seja alto, o custo de retrabalho necessário tende a diminuir, devido ao fato dos defeitos serem encontrados logo no início do processo de desenvolvimento de software. Já a não utilização do processo de inspeção resulta em mais retrabalho o que faz com que o custo de reparo do defeito detectado seja de 10 a 100 vezes mais alto e, além disso, isso acontece em fases mais adiantadas do processo de desenvolvimento (FAGAN, 1986).

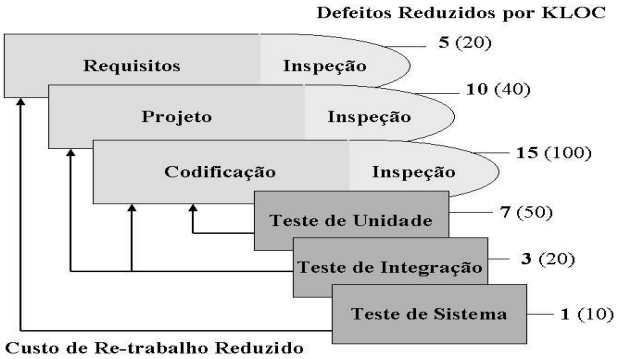


Figura 2.1. Quantidade de defeitos encontrados com e sem o processo de Inspeção (adaptado de (WHEELER; BRYKCZYNSKI; MEESON JR., 1996)**)**

Além disso, a inspeção fornece um meio de melhorar a manutenibilidade do software permitindo detectar certos tipos de defeitos que não são detectáveis usando outras técnicas (SIY; VOTTA, 2001). Exemplo disso são os defeitos de *evolvability* relacionados a defeitos funcionais os quais são identificados apenas com o processo de inspeção (MÄNTYLÄ; LASSENIUS, 2009).

O processo de inspeção pode ser aplicado nos diversos artefatos desenvolvidos, como por exemplo, especificações de requisitos, documentação, planos e casos de teste, melhorando a qualidade do produto final (software). Segundo Fagan (1999), o fato da inspeção ser aplicada logo no início do processo de desenvolvimento contribui muito para o processo, pois é nessa etapa que é injetada a maior quantidade de defeitos. Ressalta-se que o processo de inspeção não substitui os testes, mas são processos que devem ser combinados (RUSSELL, 1991).

É salientado em Fagan (1999) que o processo de inspeção não deve, de forma alguma, acarretar em um desmerecimento da pessoa que realizou o produto sendo inspecionado. Deve, sim, incentivar a pessoa a melhorar e a aprender com os erros cometidos. Alguns relatos mostram evidências de que desenvolvedores que participam de inspeções de seus próprios produtos, criam menos defeitos em trabalhos futuros.

O processo de inspeção, segundo Fagan (1986), consiste das seguintes etapas:

- Planejamento: os critérios de saída bem como os artefatos a serem inspecionados devem estar especificados. Os participantes do processo de inspeção devem ser escolhidos e o local e o horário devem ser agendados.
- Visão Geral: apresenta-se aos participantes o artefato a ser inspecionado e atribuem-se os papéis a todos os participantes.
- Preparação: os participantes estudam os artefatos e preparam-se para execução de seus papéis.
- Reunião de Inspeção: nesta operação os artefatos são examinados na tentativa de se encontrar defeitos. Utiliza-se um *checklist* ou qualquer outra técnica de leitura, e uma classificação dos tipos de defeitos procurados.
- Retrabalho: o autor do artefato inspecionado corrige todos os defeitos encontrados durante a inspeção.
- Acompanhamento: o moderador checa e verifica e cada correção.

A literatura apresenta diversos outros processos de inspeção com diferentes etapas (HERNANDES; BELGAMO; FABBRI, 2013). Um exemplo de outro processo de inspeção é o proposto por Sauer *et al.* (2000), no qual as atividades de Preparação e a Reunião de Inspeção são substituídas por três atividades sequenciais: Detecção de Defeitos, Coleção de Defeitos e Discriminação de Defeitos.

Os diversos tipos de processo de inspeção apresentados na literatura são justificados pelos autores de acordo com as suas características. Por exemplo, no processo de Sauer *et al.*

(2000), a variação no processo possibilita a realização da detecção de defeitos em paralelo por diversos inspetores o que, segundo Lanubile e Mallardo (2003), aumenta a probabilidade de encontrar defeitos difíceis de serem identificados. No entanto, Kalinowski e Travassos (2007) apresentam que o custo para a realização do processo de inspeção é maior quando o número de inspetores envolvidos é aumentado.

Todos os processos apresentados anteriormente são caracterizados como baseado em Fagan (1986) e, portanto, seguem uma abordagem formal com diversas etapas e diferentes pessoas envolvidas em encontros presenciais.

No entanto, independentemente do processo de inspeção adotado, Aurum *et al.* (2002) dividem os estudos relacionados à inspeção de software em grupos focados em:

i. Novos modelos de processo reestruturando o processo de inspeção definido por Fagan. Como exemplo, pode ser citado o processo de inspeção definido por Sauer *et al.* (2000).
ii. Métodos, ferramentas e modelos que apoiam a estrutura do processo de inspeção, como por exemplo, técnicas de leitura (apresentadas na próxima subseção do capítulo).

Em um estudo realizado por Porter *et al.* (1998) é concluído que a melhoria da efetividade da inspeção se dá por melhores técnicas de detecção de defeitos do que melhores estruturas de processo. Nesse sentido, a literatura apresenta diversas técnicas de leitura que auxiliam a inspeção de software com o objetivo de detectar os defeitos com maior facilidade. Por exemplo, no caso do documento de requisitos, exemplos de técnicas de leituras são *Checklist* e PBR *(Perspective Based Reading*) (BASILI *et al.*, 1996); para modelos de projeto descritos em UML podem-se usar as OORTs *(Object Oriented Reading Techniques*) (TRAVASSOS et al., 2002) ou as OORTs/ProDeS (MARUCCI et al., 2002). Existem também técnicas de leitura que podem ser usadas para inspeção e construção de artefatos, como é o caso da TUCCA *(Technique for Use Case Construction and Requirement Document Analysis*) (BELGAMO; FABBRI; MALDONADO, 2005) que serve para análise do documento de requisitos e construção do modelo de casos de uso.

2.2.1 Inspeção de Código

Especificamente relacionado à inspeção de código, Jones (2008) afirma que a inspeção de código pode atingir 85% de efetividade para identificação de defeitos e em média cerca de 65% de eficiência. Além disso, em um *survey* realizado por Bacchelli e Bird (2013), 337

programadores (39% da população pesquisada) relataram que uma importante motivação para inspeção de código é a "melhoria do código-fonte" obtida a partir da identificação de defeitos.

A inspeção de código pode ser apoiada por técnicas que auxiliam na identificação de defeitos. Algumas dessas técnicas são apresentadas a seguir:

- ***Ad-hoc*:** segundo Aurum *et al.* (2002), não há um procedimento a seguir pelo revisor e nenhum treinamento é necessário. Revisores utilizam os próprios conhecimentos e experiências para a identificação de defeitos.
- ***Checklist*:** a técnica de *checklist* é baseada em uma série de questões específicas com o objetivo de guiar o inspetor em busca de fontes comuns de defeitos. De acordo com Gilb e Graham (1995) e Humphrey (1995), *checklists* deveriam ser baseados em informação histórica local. Laitenberger e Debaud (1997) apresentam alguns pontos negativos em relação ao *checklist*: as questões são genéricas e não suficientemente adaptadas para um ambiente de desenvolvimento particular; instruções concretas de como usar um *checklist* geralmente não são fornecidas e; as questões do *checklist* são limitadas a identificação de defeitos que pertencem a tipos de defeitos específicos.

  Um exemplo de *checklist* que pode ser aplicado para inspeção de código segundo Dunsmore *et al.* (2003), é apresentado na Figura 2.2.

	Feature	Question
For each class:		
1	Inheritance	Is all inheritance required by the design implemented in the class?
2		Is the inheritance appropriate?
3	Class Constructor	Are all instance variables initialised with meaningful values?
4		If a call to super is required in the constructor, is it present?
For each method:		
5	Data Referencing	Are all parameters used within a method?
…		
14	Method Behavior	Are all assignments and state changes made correctly?
15		For each return statement, is the value returned and its type correct?
16		Does the method match the specification?
For each class:		
17	Method Overriding	If inherited methods need to behave differently, are they overridden?
18		Are all uses of method overriding correct?

Figura 2.2. *Checklist* exemplo extraído de Dunsmore *et al.* (2003)

Com relação à efetividade de aplicação de *checklist* na atividade de inspeção, um estudo experimental realizado por Hatton (2008) não encontrou evidência significativa se a experiência do inspetor aumenta ou não a efetividade da atividade de inspeção. Em Almeida *et al*. (2003), o sucesso do uso de *checklist* em um processo de inspeção de software crítico é apresentado.

No trabalho de Chernak (1996) é apresentado que o processo de inspeção de código se torna mais produtivo e os inspetores encontram mais defeitos quando o *checklist* é representado por dois componentes: (a) onde procurar um defeito; (b) como detectar um defeito.

- **Técnica dirigida à Abstração:** essa técnica utiliza os conceitos da *Stepwise Abstraction* (SA). A SA é uma técnica para a leitura de código, na qual a funcionalidade do programa é determinada pelas abstrações funcionais que são geradas a partir do código fonte. A SA consiste em realizar leituras do código fonte, a partir das quais os revisores escrevem sua própria especificação para o programa (LINGER; MILLS; WITT, 1979).

  Nessa técnica, o inspetor abstrai, passo a passo, cada trecho do programa, começando das partes mais internas para as mais externas determinando uma especificação para cada um deles, de forma que, combinando-se todas as especificações já abstraídas, obtém-se a especificação completa do código. Então, essa especificação é comparada com a especificação original que foi usada para construir o programa. Com base nessa comparação, são identificadas as discrepâncias entre as especificações e, a partir delas, são procurados os defeitos no código.

  Segundo Mcmeekin *et al.* (2008), uma vantagem dessa técnica é que as especificações em linguagem natural criadas podem ser usadas para inspeções futuras bem como para servir de documentação para referência futura. Uma desvantagem é o tempo de aplicação da técnica forçando ao inspetor gastar muito tempo examinando código fora do escopo da inspeção.

- **Técnica de Caso de Uso:** segundo Dunsmore *et al.* (2003), a técnica de caso de uso utiliza casos de uso da UML e tem como objetivo checar se cada objeto é capaz de responder corretamente a todos possíveis modos aos quais ele deve ser usado. Mais especificamente, a técnica verifica se:

  i. Os métodos corretos estão sendo chamados pelos usuários da classe.
  ii. As decisões e as mudanças de estados feitas em cada método são corretas e consistentes.

  A abordagem básica da técnica é criar vários cenários do caso de uso e examinar como a classe sendo inspecionada lida com os diversos cenários levantados.

- **Leitura Baseada em Perspectiva (*Perspective-Based Reading* – PBR):** foi desenvolvida por Basili *et al.* (1996) e já foi objeto de estudos experimentais (BASILI et al., 1996) (LAITENBERGER; EMAM; HARBICH, 2001), especialmente relacionados à inspeção de documento de requisitos. A técnica define cenários operacionais em que o leitor do documento a ser inspecionado deve seguir e pode ser caracterizada como uma fusão de questões e modelos (BASILI et al., 1996). Durante a pesquisa bibliográfica, apenas um estudo de aplicação da PBR em código-fonte foi encontrado em Laitenberger e Debaud (1997).
- **TDI (*Task-Directed Inspection*)**: apresentada por Kelly e Shepard (2004), é baseada em uma estratégia na qual a inspeção é realizada de acordo com alguma tarefa. Os autores apresentam um experimento comparando a técnica *checklist* com duas variações de TDI (uma voltada a escrever uma descrição em linguagem natural do trecho de código a ser inspecionado e outra variação exigindo a escrita de um esboço de plano de teste unitário para o trecho de código atribuído).
- **Abordagem Baseada em Comparação**: de acordo com Li (1995), essa abordagem envolve duas tarefas: (1) examinar e comparar várias partes relacionadas do código-fonte, levantando questões sobre possíveis defeitos; (2) verificar a especificação de requisitos para conseguir respostas satisfatórias às questões. Em Li (1995) é apresentado um exemplo de aplicação da técnica, a qual envolve um formalismo e o autor declara que a técnica deveria ser utilizada de maneira complementar em relação a outras técnicas de inspeção de código.

2.2.2 Estudos Experimentais sobre técnicas de inspeção de código

Algumas técnicas de inspeção apresentadas na subseção anterior foram avaliadas empiricamente e alguns resultados são apresentados a seguir.

Dunsmore *et al.* (2003) realizaram um experimento em um ambiente acadêmico dividindo os participantes em três grupos de 23 participantes, um grupo para aplicação de cada uma das seguintes técnicas: *Checklist*, Técnica Dirigida à Abstração e Caso de Uso. O objetivo de cada participante dos grupos era identificar defeitos injetados pelos pesquisadores utilizando a técnica atribuída a cada participante em um tempo limite máximo de 90 minutos.

Em termos do número de defeitos identificados, os participantes que utilizaram a técnica *checklist* tiveram melhor desempenho, identificando, em média, 52% dos defeitos. A aplicação da técnica dirigida à abstração resultou na identificação de 44% dos defeitos e a técnica de caso de uso em 40% dos defeitos.

Em relação ao tempo de aplicação das técnicas e o número de defeitos identificados, os autores observaram que participantes utilizando a técnica *checklist* encontraram mais defeitos e em uma porcentagem mais rápida de tempo, embora o desempenho tenha diminuído após os 60 minutos de início do experimento.

No caso das técnicas dirigidas à abstração e caso de uso, os autores relatam que a taxa de identificação de defeitos são similares, talvez pelo fato das duas técnicas possuírem uma maior sobrecarga de trabalho inicial, como por exemplo, abstrair o código a ser inspecionado (no caso da técnica dirigida à abstração) e criar os cenários dos casos de uso (no caso da técnica de caso de uso).

Ao contrário dos resultados obtidos por Dunsmore *et al.* (2003), Laitenberger e Debaud (1997) e Laitenberger *et al.* (2001) compararam a leitura baseada em perspectiva com checklist em código C e os resultados sugerem uma vantagem da primeira técnica.

2.3 Visualização de Informações

De acordo com Gershon *et al.* (1998), a visualização é um processo de transformar os dados, informações e conhecimentos em forma visual, fazendo uso da capacidade visual natural dos seres humanos, fornecendo uma interface entre dois sistemas de tratamento da informação: a mente humana e o computador.

Especificamente relacionado à visualização de informações, Tukey (1977) define que a área de visualização de informações está relacionada com a aplicação de técnicas de computação gráfica, comumente interativas, visando auxiliar o processo de análise e compreensão de um conjunto de dados por meio de representações gráficas.

Os dados visualizados por representações gráficas são considerados abstratos, obrigando que as imagens geradas por esses dados sejam baseadas nos relacionamentos ou informações que podem ser inferidas pelos mesmos.

O uso da visualização de informações permite ao leitor explorar o sentido humano (visão) que possui a maior capacidade de captação de informações por unidade de tempo (NASCIMENTO; FERREIRA, 2005) possibilitando observações de padrões ou de

características visuais que permitem uma melhor compreensão do que a simples observação dos dados em sua forma abstrata.

De acordo com Nascimento e Ferreira (2005), os dados abstratos a serem visualizados podem ser classificados em:

- Nominal: conjunto de elementos distintos, sem uma relação de ordem entre eles. Exemplo: João, Maria, Alberto;
- Ordinal: conjunto de elementos distintos, mas com uma relação de ordem entre os mesmos. Exemplo: Segunda, Terça, Quarta, ou Janeiro, Fevereiro;
- Quantitativo: faixa de valores numéricos. Essa categoria pode ser dividida em Intervalos (com valores discretos) e Razão (representando uma faixa contínua de valores).

De posse dos dados abstratos é possível realizar o processo de construir a visualização de informações. A literatura apresenta um modelo de referência clássico (vide Figura 2.3) para a visualização de informações (CARD; MACKINLAY; SHNEIDERMAN, 1999), o qual é composto em três etapas (Transformação de Dados, Mapeamento Visual e Transformações Visuais) e que utiliza como entrada os dados brutos que são transformados, após a aplicação das etapas, em visões para o usuário.

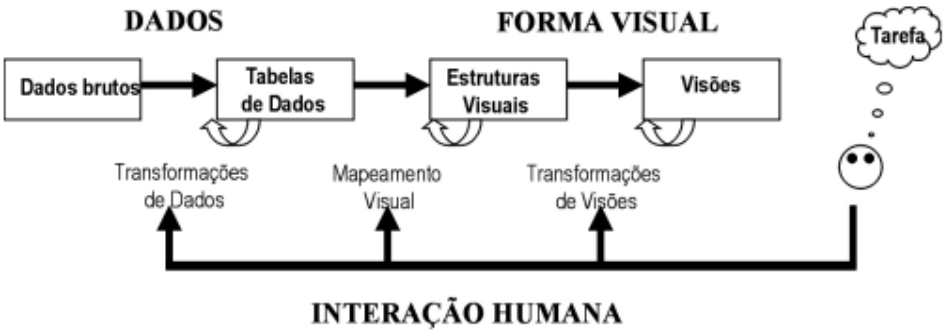


Figura 2.3. Modelo de Referência Clássico para Visualização de Informações adaptado de (CARD; MACKINLAY; SHNEIDERMAN, 1999)

A seguir são apresentadas as etapas a serem realizadas (NASCIMENTO; FERREIRA, 2005):

- **Transformações dos Dados**: os dados brutos são processados e organizados em uma representação lógica mais estruturada, geralmente na forma de uma ou mais tabelas em um banco de dados relacional. Durante o processamento algumas atividades são realizadas como: eliminação de dados redundantes, errados ou incompletos, bem como a filtragem e o agrupamento dos dados.

- **Mapeamento Visual**: os dados alimentados nas tabelas são usados para construir uma estrutura visual que os representem, podendo ser decomposta em três partes:
    1. **Substrato visual:** caracteriza o espaço para a visualização, sendo normalmente representado por eixos, tais como os eixos X e Y do plano cartesiano;
    2. **Marcas visuais**: são símbolos gráficos utilizados para representar os itens de dados, como figuras geométricas, linhas, áreas e volume;
    3. **Propriedades gráficas das marcas**: são os atributos visuais que as caracterizam, como direção, cor, forma, inclinação, textura, etc.
- **Transformações Visuais**: nessa etapa é possível interagir com as estruturas visuais através de operações básicas como teste de localização (obter informações adicionais sobre um item da tabela de dados), controles de ponto de vista (ampliação, redução e deslocamento da imagem para obter visões diferentes) e distorções da imagem (criar ampliações de uma região específica da imagem em detrimento de outra).

O estudo de Lengler e Eppler (2007) se destaca pelo fato de caracterizar seis métodos de visualização e apresentar diversos exemplos de técnicas para cada método de acordo com a analogia de uma tabela periódica, conforme pode ser consultado em http://www.visual-literacy.org/periodic_table/periodic_table.html. Os métodos de visualização caracterizados pelos autores são apresentados a seguir:

- **Visualização de dados**: são representações visuais de dados quantitativos de forma esquemática - com ou sem eixos. Exemplos dessa categoria são os gráficos de pizzas e os *boxplots* da análise estatística.
- **Visualização de informação**: é definida como a utilização de representações visuais interativas para ampliar a cognição. Isto significa que os dados são transformados em uma imagem que podem ser manipuladas pelos usuários de acordo com o trabalho a ser realizado. Exemplos dessa categoria são os Diagramas Entidade Relacionamento e o Diagrama de Venn da matemática.
- **Visualização conceitual**: são métodos para elaborar, na maioria das vezes, conceitos qualitativos, ideias, planos e análises. Exemplos dessa categoria são os Mindmap e os gráficos da técnica PERT/CPM.
- **Visualização estratégica**: é definida como o uso sistemático de representações visuais complementares para análise, desenvolvimento, formulação, comunicação

e implementação de estratégias em organizações. Esse tipo de visualização é o mais específico de todos os grupos e exemplos dessa categoria são o Diagrama de Ishikawa e o Organograma.

- **Visualização metafórica**: esse método posiciona informação graficamente para organizar e estruturar a informação. Esse método também transmite informação sobre a informação representada através de características chaves da metáfora a ser empregada. Exemplo dessa categoria são os mapas do metrô.
- **Visualização composta**: é a utilização complementar de diferentes formatos de representação visual em um único esquema ou quadro, fazendo uso de vários dos formatos acima mencionados. Exemplos são as imagens ricas e murais de informação.

Como no contexto deste trabalho, o uso de visualização de informações está associado à visualização de código-fonte, a subseção a seguir apresenta os principais tipos de visualização de código-fonte identificados na literatura.

2.3.1 Visualização de Código-Fonte

De acordo com Rilling *et al.* (2002), à medida que os programas se tornam mais complexos e maiores, o volume de informações apresentadas aos envolvidos torna-se impraticável a sua compreensão.

Para tanto, conceitos, técnicas e ferramentas associadas visualização de software podem ser utilizadas. O objetivo da visualização de software é adquirir conhecimento suficiente sobre um sistema de software compreendendo artefatos do software e entendendo seus relacionamentos (VON MAYRHAUSER; VANS, 1998) (KNIGHT; MUNRO, 1999).

A visualização de código-fonte pode ser considerada um tipo de visualização de software e na maioria das vezes está associada a uma atividade de compreensão. No caso específico deste trabalho, a visualização de código como uma atividade de compreensão deve ser usada como fonte de informações para auxiliar o inspetor na identificação de defeitos.

Para tanto, pesquisou-se na literatura formas de visualização de código-fonte que pudessem auxiliar na atividade de compreensão do código-fonte e consequentemente na identificação de defeitos durante a inspeção.

A visualização de código-fonte pode ser categorizada em visões estáticas e dinâmicas. As visões estáticas são baseadas na análise estática do código-fonte e sua informação associada e fornecem uma visão alto-nível mais genérica do sistema e seu código-fonte. Já a

visão dinâmica é baseada na informação da análise da execução monitorada ou gravada de um programa (RILLING; SEFFAH; BOUTHLIER, 2002).

Conforme mencionado por Mayrhauser e Vans (1998), os tipos de visões deveriam ser utilizadas de forma complementar e não exclusivas. A seguir, são apresentadas algumas técnicas de visualização de código-fonte:

- ***Fisheye***: nessa técnica, o código-fonte é visualizado e uma linha deve ser selecionada (chamada linha foco) enquanto nas demais linhas do código-fonte são atribuídas valores (chamado grau de interesse), calculado pela distância de cada linha em relação à linha foco. Linhas com valores abaixo de um limiar são ocultas, apresentando apenas as linhas de interesse (JAKOBSEN; HORNBAK, 2006).
- ***Treemap***: desenvolvida originalmente para identificar, através da visualização, quais programas usavam mais espaço de disco (JOHNSON; SHNEIDERMAN, 1991). A *treemap* representa uma árvore hierárquica de dados como um conjunto de retângulos aninhados, onde cada nó é representado como um retângulo. O tamanho do retângulo pode indicar propriedades do nó sendo representado. A ferramenta CRISTA, utilizada no contexto deste trabalho, utiliza a metáfora *treemap* para representação dos elementos de código de uma classe, conforme pode ser visualizado na Figura 3.4. Cada retângulo representado na Figura 2.4 representa uma instrução do código da classe visualizada e as cores indicam se o código foi abstraído ou não.

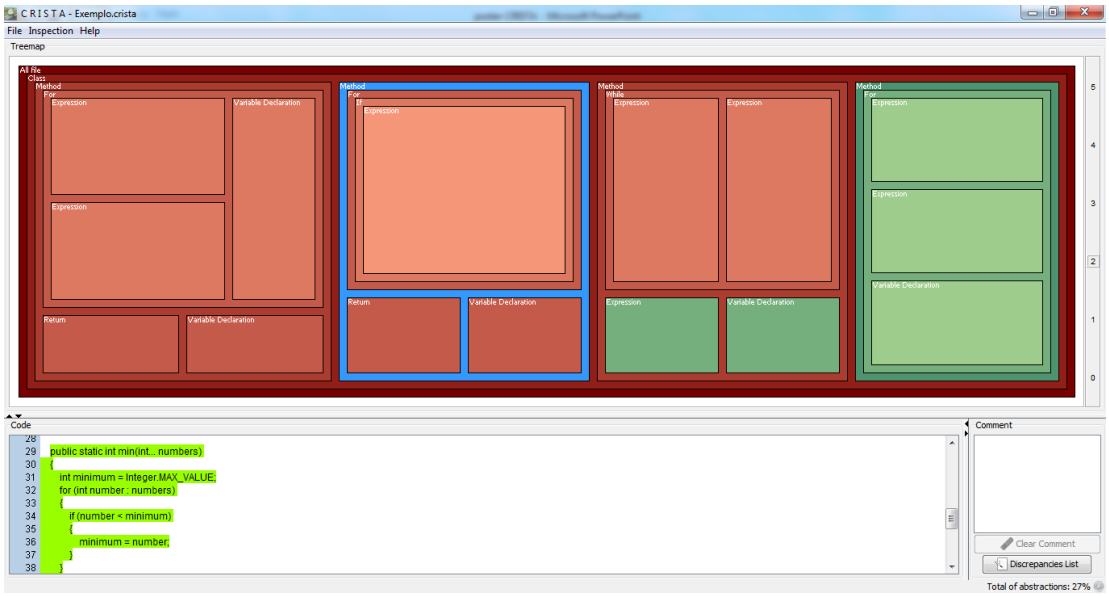


Figura 2.4. Exemplo do uso da técnica *Treemap* na ferramenta CRISTA original

- **Árvore Hiperbólica**: a técnica de visualização Árvore Hiperbólica usa geometria hiperbólica para fornecer um efeito de "*fisheye*". Essa técnica tem como objetivo mostrar e manipular uma grande estrutura hierárquica, combinando uma visão detalhada em uma pequena área (foco), enquanto fornece uma visão geral de toda estrutura (contexto) (LAMPING; RAO; PIROLLI, 1995) (LAMPING; RAO; PIROLLI, 1995).
- **Metáfora da Cidade**: essa metáfora é baseada no conceito da habitabilidade, a qual, segundo Wettel e Lanza (2007), transmite ao desenvolvedor a noção de que o sistema de software é um espaço físico com pontos de orientação. Para tanto, uma técnica de visualização 3D é proposta, possibilitando que os conceitos de habitabilidade pelo uso da metáfora da cidade sejam representados. Na metáfora da cidade aplicada ao código orientado a objetos, classes são representadas como edifícios localizados nos distritos das cidades, os quais representam os pacotes do sistema. Os tamanhos e os tipos de edifícios (arranha-céus, escritórios, apartamentos, mansões e casas) representam valores para diferentes métricas obtidas do código-fonte. Um exemplo da metáfora da cidade é representado na Figura 2.5, que foi gerada a partir da ferramenta CodeCity (WETTEL; LANZA; ROBBES, 2011).

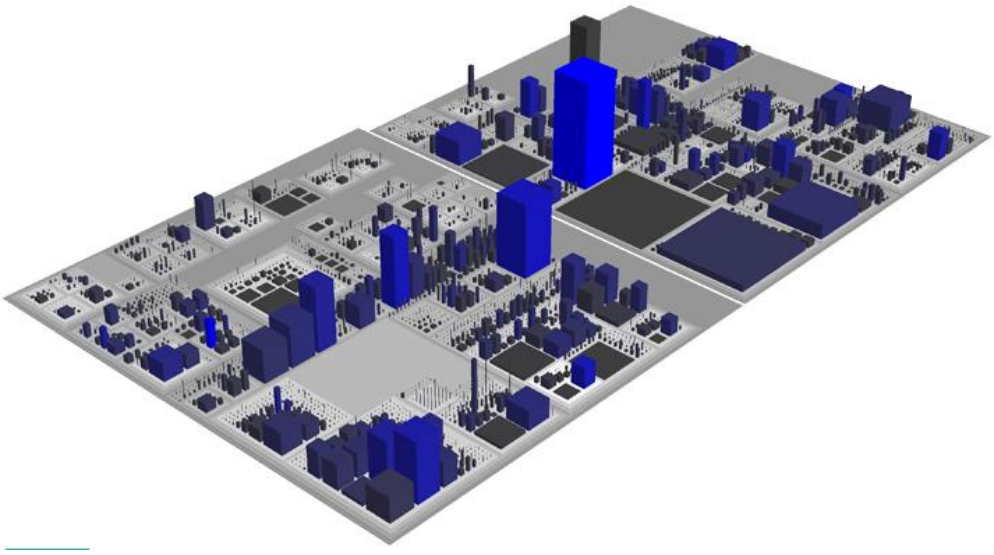


Figura 2.5. Exemplo da metáfora da cidade gerada a partir da ferramenta CodeCity (WETTEL; LANZA; ROBBES, 2011)

- **Visão Polimétrica**: a visão polimétrica utiliza um gráfico de duas dimensões para visualizar software orientado a objetos. Os nós representam as entidades de software ou abstrações dessas entidades, enquanto que as arestas

representam relacionamentos entre essas entidades. Lanza e Ducasse (2003) utilizam a visão polimétrica para apresentar métricas de software. Um exemplo dessa visualização pode ser obtida pela ferramenta SourceMiner (CARNEIRO et al., 2008), a qual é apresentada na Figura 2.6. Na representação Polimétrica da Figura 2.6 pode ser visualizada as classes usando um *layout* de árvore. A largura e a altura dos nós representam os valores das métricas de software.

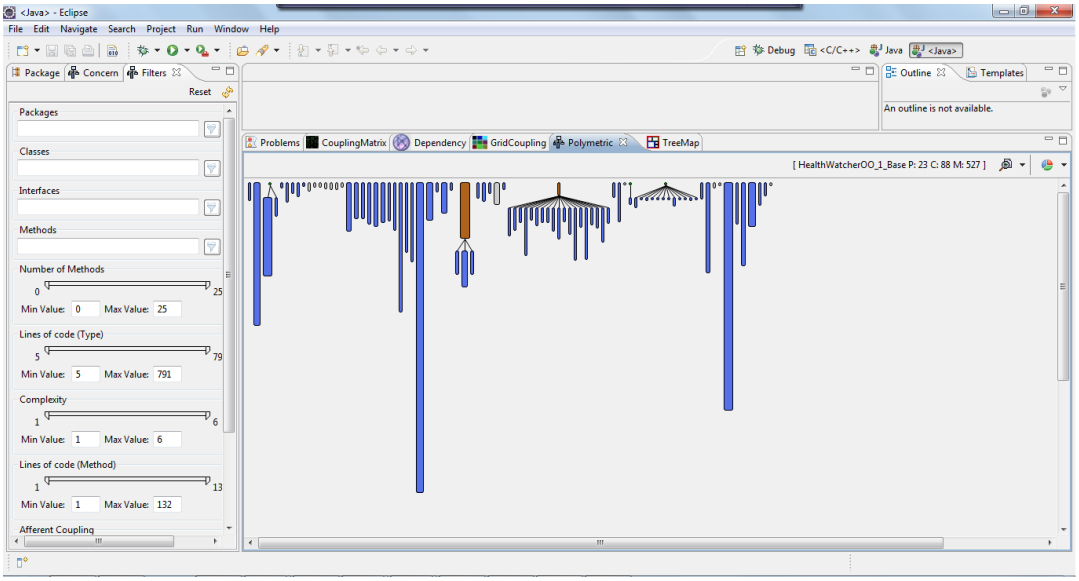


Figura 2.6. Exemplo da visão polimétrica usada na ferramenta SourceMiner (CARNEIRO et al., 2008)

Destaca-se que durante o levantamento bibliográfico referente à visualização de código-fonte, grande parte das técnicas de visualização identificadas estava relacionada à representação de métricas de software para auxiliar nas atividades de compreensão de software e nas atividades de evolução de software.

Por fim, um interessante trabalho publicado por Caserta e Zendra (2011) apresenta uma pesquisa sobre técnicas de visualização representando os aspectos estáticos do software e sua evolução, fornecendo uma categorização das técnicas de acordo com o seu foco (propriedades de linha, métricas, organização, relacionamentos, mudanças, evolução de métricas) a fim de auxiliar na comparação e na identificação das técnicas e ferramentas mais relevantes para um dado problema.

2.4 A Ferramenta CRISTA

A ferramenta CRISTA (*Code Reading Implemented with STepwise Abstraction*) foi criada com a motivação inicial de apoiar a inspeção de código com a técnica de leitura *stepwise abstraction.*

Nela é possível fazer a leitura do código, o registro das discrepâncias, e a junção das listas de discrepâncias elaboradas por cada inspetor. Além disso, uma característica peculiar da ferramenta CRISTA é o seu auxílio na abstração do código pela técnica *stepwise abstraction* e pelos recursos de visualização.

Nesse sentido, a ferramenta CRISTA adota uma metáfora visual que representa visualmente os blocos hierárquicos do código sendo analisado. A metáfora visual *treemap* oferece um simples modo de olhar a estrutura do código. Ela separa blocos de código como retângulos aninhados de acordo com sua hierarquia. Ela também trabalha com um *feedback* visual para o processo de análise do código, mudando as cores do retângulo de acordo com a abstração dos blocos sob análise.

A Figura 2.7 apresenta a tela principal da ferramenta CRISTA, na qual se observam três áreas principais: 1 – uma metáfora visual que representa visualmente os blocos hierárquicos do código sendo analisado; 2 – essa janela pode ser navegada e ela é logicamente ligada a área de visualização. Qualquer retângulo que seja clicado na metáfora visual, o corresponde código-fonte na janela de código é realçado e vice-versa; e 3 – uma vez que um bloco seja selecionado, sua abstração pode ser realizada na área de documentação. Esse comentário é fisicamente associado ao bloco selecionado, e pode ser usado posteriormente para produzir pseudo-código ou documentação do código. Quando um comentário é associado com um bloco de código, o retângulo correspondente a esse bloco é colorido em verde. Isso permite ao leitor facilmente acompanhar o progresso do processo de abstração de código.

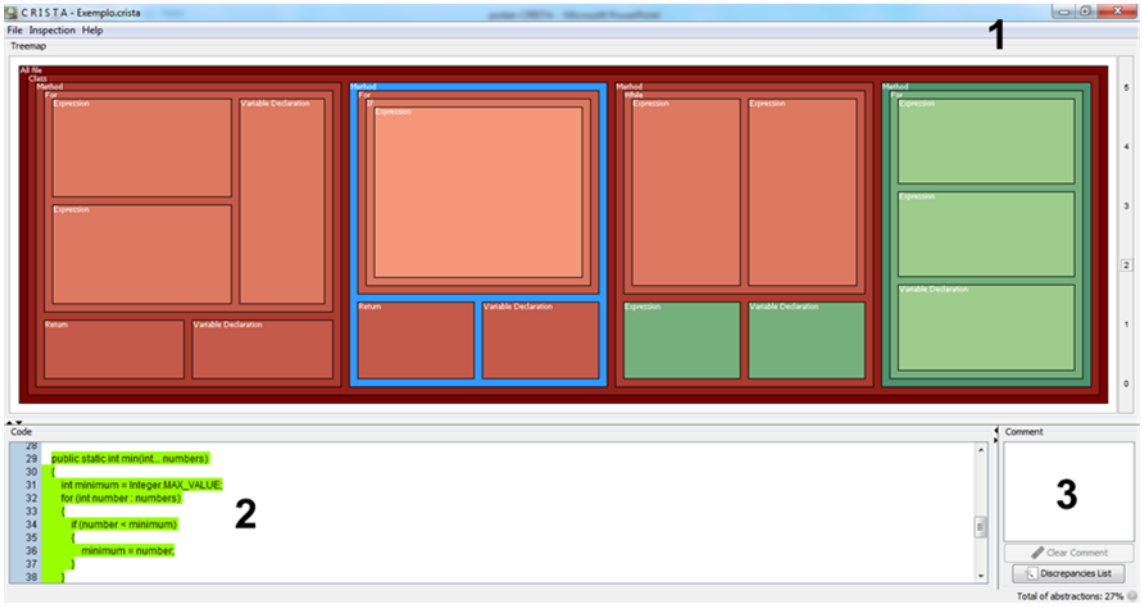


Figura 2.7. Tela Principal da Ferramenta CRISTA

Considerando a diversidade de linguagens de programação encontradas no mercado, a CRISTA foi projetada para ser instanciada para outras linguagens, como apresentado em (Porto et al., 2008). Atualmente ela aceita código em Java, C, C++ e Cobol85. No entanto, qualquer outra linguagem pode ser utilizada na ferramenta CRISTA desde que ela seja instanciada para a linguagem escolhida, conforme apresentado na Figura 2.8.

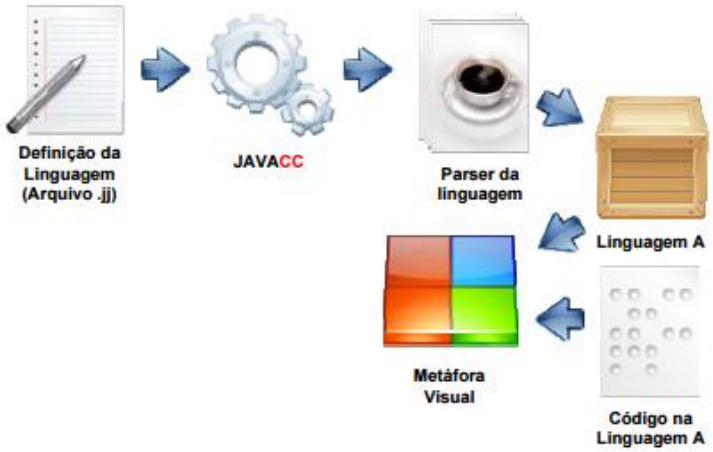


Figura 2.8. Instanciação de novas linguagens na ferramenta CRISTA

De acordo com a Figura 2.8, para a utilização de uma nova linguagem na ferramenta CRISTA é necessária a definição da gramática da linguagem (arquivo .jj) baseada no JavaCC a fim de gerar o *parser*. Com o *parser* gerado, basta a geração do arquivo .jar e a inserção no diretório da ferramenta CRISTA que já é possível a inspeção de código de novas linguagens.

Mais detalhes da ferramenta CRISTA bem com os aspectos de implementação podem ser encontrados em Porto (2009).

2.5 Considerações finais

Neste capítulo foram apresentados os temas envolvidos diretamente com a tese: inspeção de código e visualização de código-fonte. Embora esses temas sejam passíveis de discussões mais extensas e completas, apresentam-se aqui, os principais conceitos e dá-se ênfase as definições de inspeção, características das técnicas e técnicas de visualização.

Sobre inspeção de software foram apresentados os conceitos relacionados a essa atividade, os processos de inspeção existentes e as técnicas de leitura que podem ser aplicadas durante a identificação de defeitos em código-fonte.

Alguns conceitos sobre visualização de software foram apresentados e um maior foco foi dado na visualização de código-fonte por se tratar de tema relevante para esta pesquisa. Ressalta-se que a proposta deste trabalho não tem como objetivo criar novas formas de visualização de código-fonte e sim utilizar as existentes com o intuito de auxiliar na compreensão do código e na atividade de identificação de defeitos.

Além disso, foram apresentadas algumas técnicas de inspeção de código. Ressalta-se que algumas dessas técnicas não contemplam apropriadamente as características do paradigma orientado a objetos e, embora com certa sistematização, não apresentam diretrizes bem definidas que padronizem o comportamento dos inspetores durante a atividade de inspeção, a fim de minimizar a subjetividade inerente ao processo. Além disso, a falta de uma sistematização não permite a avaliação das próprias técnicas, ou seja, não é possível a verificação de quais trechos das técnicas podem ser melhorados. Portanto, após o levantamento realizado, constatou-se que este trabalho poderia contribuir com a melhoria da atividade de inspeção na definição de técnica de leitura que contemplasse aspectos ainda não tratados pelas técnicas existentes.

Capítulo 3

CICLO 1: AVALIAÇÃO DA TÉCNICA *STEPWISE ABSTRACTION* COM SUPORTE DE VISUALIZAÇÃO EM RELAÇÃO À TÉCNICA *AD-HOC*

Este capítulo apresenta um estudo experimental para avaliação dos recursos de visualização e da técnica de leitura Stepwise Abstraction durante a atividade de inspeção usando os recursos da ferramenta CRISTA original.

3.1 Considerações iniciais

Dada a contribuição da ferramenta CRISTA e a motivação em dar continuidade à evolução da ferramenta, o primeiro passo, como mencionado anteriormente, foi o entendimento aprofundado do trabalho de Porto *et al.* (2009) e da análise dos resultados de um experimento conduzido por ele e que ainda não havia sido analisado.

Assim, neste capítulo são apresentados os resultados obtidos a partir da análise do estudo experimental realizado por Porto *et al.* (2009). Esse estudo teve como objetivo investigar a efetividade e eficiência do uso da técnica de leitura *Stepwise Abstraction* com suporte de visualização implementado na ferramenta CRISTA original, em relação à técnica *ad-hoc*.

Nas próximas seções relatam-se as principais etapas do experimento, conforme o processo definido por Wohlin *et al.* (2000).

3.2 Planejamento do Experimento

O planejamento do experimento foi proposto baseado no modelo GQM (BASILI; CALDIERA; ROMBACH, 1994) e pode ser resumido da seguinte forma:

Analisar	o uso da técnica *Stepwise Abstraction* com suporte de visualização
Com o propósito de	avaliar
Em relação à	efetividade e eficiência
Do ponto de vista	de pesquisadores da área de inspeção de código
No contexto de	Estudantes do Curso de Ciência da Computação e Engenharia da Computação da Universidade de Federal de São Carlos.

3.3 Seleção de Contexto

O experimento foi considerado uma atividade da disciplina de Engenharia de Software dos cursos de Bacharelado em Ciência da Computação e Engenharia da Computação da UFSCar - Universidade Federal de São Carlos - em que os alunos não receberam uma nota pelo desempenho no resultado, mas apenas uma nota pela participação. Eles foram conscientizados de que a responsabilidade e a colaboração efetiva deles estariam ajudando e contribuindo para a pesquisa. Todos se mostraram dispostos em participar e demonstraram isso no decorrer do experimento.

De acordo com as quatro dimensões que caracterizam o contexto, segundo Wohlin *et al.* (2000), tem-se que: o experimento foi do tipo *off-line* por não ser baseado no desenvolvimento de software pela indústria; os participantes são estudantes de graduação; o problema tem o porte de um problema *toy* e corresponde também a um problema específico não podendo ser generalizado para outros contextos.

No entanto, os resultados alcançados podem auxiliar na melhoria da ferramenta e na sua contribuição para as atividades de compreensão de código e identificação de defeitos.

3.4 Seleção das Variáveis

Para esse experimento foi considerada uma variável independente, chamada "TécnicaUsada", a qual representa o uso ou o não uso da "técnica *stepwise abstraction* + visualização". A variável independente possui dois tratamentos:

- StepVis: representado pelo uso da "técnica *stepwise abstraction* + visualização" (implementada pela ferramenta CRISTA).
- *Ad-hoc*: representado pelo uso da técnica "ad-hoc".

Foram também consideradas outras variáveis independentes:

- Experiência em Java: foi investigada se a experiência na linguagem Java teria alguma influência nos resultados.
- Experiência em inspeção de código: foi investigada se experiência em inspeção de código teria alguma influência nos resultados.

Foram consideradas duas variáveis dependentes:

- Efetividade: definida como a quantidade de defeitos reais identificados pelo inspetor.
- Eficiência: definida como a razão do número de defeitos reais identificados e o tempo de inspeção.

Salienta-se que durante a inspeção, o inspetor pode relatar discrepâncias que, muitas vezes, não são consideradas como "defeitos reais". Essas discrepâncias serão denominadas de "falso-positivos" e não foram consideradas para o cômputo da efetividade e eficiência.

3.5 Seleção dos Participantes

A seleção dos participantes foi baseada na conveniência, isto é, a disciplina de Engenharia de Software estava sendo ministrada aos alunos dos cursos de Ciência da Computação e Engenharia da Computação da Universidade Federal de São Carlos. Dessa forma, o tópico inspeção de software fazia parte da ementa da disciplina. De acordo com o levantamento realizado, a maior parte dos participantes do experimento não possuía conhecimento prévio em inspeção de software.

3.6 Projeto Experimental

O projeto experimental foi realizado por meio de dois grupos (G1 - StepVis e G2 - Ad-Hoc) compostos, respectivamente, por 31 e 28 participantes. A diferença no número de

participantes deu-se pelo fato de existirem duas turmas da disciplina de Engenharia de Software com números diferentes de alunos.

Dessa forma, o Grupo 1 utilizou a "técnica *stepwise abstraction* + visualização" e o Grupo 2 utilizou a técnica "ad-hoc" para inspeção de código. Detalhes do projeto experimental são apresentados na Tabela 3.1.

Tabela 3.1. Projeto Experimental

Dia	**G1 - StepVis**	**G2 – Ad-hoc**	**Duração**
Dia 1	Caracterização dos Participantes - Questionário		10 minutos
Dia 1	Treinamento em Inspeção de Software		30 minutos
Dia 1	Treinamento em Inspeção de Código-Fonte e Realização de Exercícios		60 minutos
Dia 2	Treinamento na ferramenta CRISTA	Exercícios de inspeção *ad-hoc*	30 minutos
Dia 3	Realização do experimento com CRISTA	Realização do experimento com a técnica *ad-hoc*	90 minutos

Ressalta-se que a divisão dos participantes nos dois grupos foi realizada após o levantamento do perfil dos participantes de modo que os perfis ficaram equilibrados em cada um dos grupos.

3.7 Instrumentação

Os instrumentos do experimento foram o software *Paint*, por meio de seu código-fonte e a ferramenta CRISTA, para dar suporte à visualização de código e à técnica dirigida à abstração.

O software *Paint* é um pequeno editor de figuras escrito na linguagem Java. Suas funcionalidades permitem desenhar traços simples (modo "*free hand*"), escolher a cor do traço combinando as cores primitivas vermelho, verde e azul (sistema RGB) e por fim, apagar todo ou partes do desenho feito. É possível também desfazer cada ação do usuário (*undo*). O programa *Paint* já foi utilizado em outros estudos experimentais, conforme relatado em Ko *et al.* (2006).

O programa *Paint* não possui nenhum tipo de documentação e é composto por oito classes que possuem, em média, 73 linhas de código. O programa *Paint* possui quatro defeitos conhecidos:

- Defeito 1: Não é possível escolher a cor amarela.

- Defeito 2: A opção de desfazer a última ação (*undo*) não funciona até que o cursor seja posicionado na tela de desenho.
- Defeito 3: A opção de desfazer começa habilitada sem nenhuma ação ter sido executada.
- Defeito 4: A opção de desenhar uma linha reta não funciona.

3.8 Formulação das Hipóteses

As seguintes hipóteses foram definidas:

- Hipótese 1: Existe diferença no número de defeitos identificados na inspeção do código do *Paint* quando comparadas a técnica "*stepwise abstraction* + visualização" com a técnica "ad-hoc".
- Hipótese 2: Existe diferença no tempo gasto na inspeção do código do *Paint* quando comparadas a técnica "*stepwise abstraction* + visualização" e a técnica "ad-hoc".
- Hipótese 3: A experiência dos participantes em inspeção de código afeta o número de defeitos identificados quando utilizada a técnica "*stepwise abstraction* + visualização" e a técnica "ad-hoc".
- Hipótese: A experiência dos participantes na linguagem de programação Java afeta o número de defeitos identificados usando a técnica "*stepwise abstraction* + visualização" e a técnica "ad-hoc"

3.9 Ameaças à Validade

A seguir relacionam-se alguns itens que poderiam representar ameaças à validade do experimento. No caso das validades interna e de conclusão, algumas medidas puderam ser tomadas para reduzir tais ameaças.

Validade Interna

1. Interferência no desempenho do participante devido à atribuição de nota à atividade: como as notas foram atribuídas pela participação do estudante, considera-se que o risco associado ao desempenho tenha sido minimizado.

2. Erro no registro do tempo de aplicação: esse risco não pode ser minimizado, pois cada estudante foi responsável pela marcação do seu tempo.

Validade Externa

- Os resultados não podem ser generalizados para outros perfis de inspetores e aplicações de outro porte. Essa ameaça não pode ser minimizada, pois não houve possibilidade de aplicar o experimento com outros tipos de profissionais.

Validade de Conclusão

1. Considera-se que não houve ameaça à validade de conclusão, pois os dados foram avaliados quanto à sua distribuição normal por meio do teste de Kolmogorov-Smirnov. Somente após a confirmação da normalidade dos dados é que foram utilizados testes não-paramétricos.

3.10 Preparação e Execução do Experimento

Cada participante recebeu o material para realizar as seguintes tarefas:

- Tarefa 1: preencher um questionário de caracterização, o qual incluía informações sobre detalhes pessoais e técnicos.
- Tarefa 2: treinamentos dos grupos da seguinte maneira: Grupo 1 (inspeção de código utilizando a "técnica *stepwise abstraction* + visualização") e Grupo 2 (inspeção de código de forma *ad-hoc*). Essa tarefa foi realizada uma semana antes do experimento.
- Tarefa 3: aplicação da inspeção no software *Paint* de acordo com a técnica estipulada para cada um dos grupos. Durante o experimento, foi solicitado aos participantes que não se comunicassem.

3.11 Coleta de Dados

A coleta de dados se deu por meio de questionários que deveriam ser preenchidos por cada participante do experimento. Devido ao fato de um grupo utilizar a ferramenta, um dos questionários possuía questões relacionadas ao uso da ferramenta.

3.12 Análise e Interpretação

As análises foram executadas com o uso da ferramenta estatística MiniTab e os resultados são apresentados a seguir.

3.12.1 Análise Descritiva

A Figura 3.1 apresenta as porcentagens de participantes que encontraram os defeitos existentes no *Paint*.

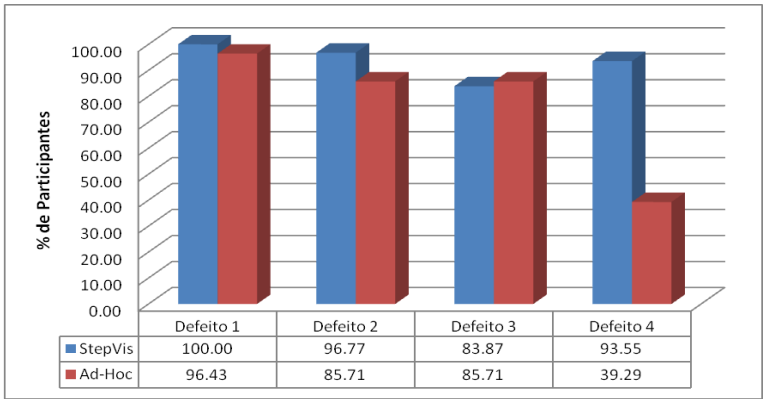


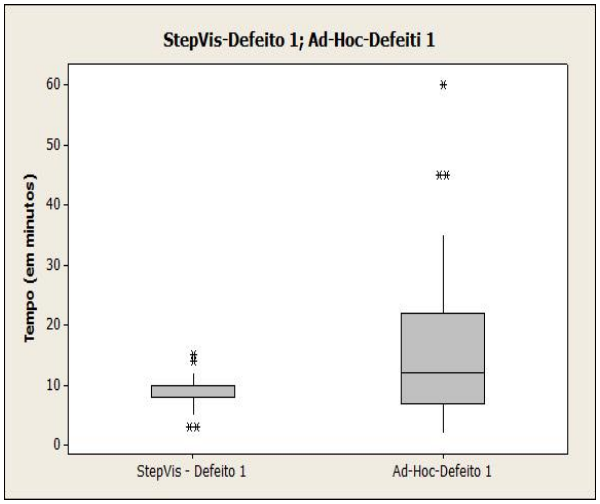
Figura 3.1. Porcentagem de participantes que encontraram os defeitos no *Paint*

Observa-se pela Figura 3.1, que todos os participantes que utilizaram a técnica "*stepwise abstraction* + visualização" encontraram o Defeito 1. No caso do defeito 2, quase 100% dos participantes que utilizaram a técnica "*stepwise abstraction* + visualização" o encontraram. A maior diferença diz respeito ao Defeito 4, no qual 93% dos participantes que utilizaram a técnica "*stepwise abstraction* + visualização" o encontraram, enquanto que para os participantes que utilizaram a técnica "ad-hoc" aproximadamente 39% dos participantes o encontraram. Observa-se também que para o Defeito 3, os participantes que aplicaram a técnica "*stepwise abstraction* + visualização" tiveram um desempenho levemente inferior em relação aos participantes que aplicaram a técnica "ad-hoc". Em decorrência desse fato, o tipo de defeito e sua localização foi motivo de investigação neste trabalho e será apresentado nos outros experimentos realizados.

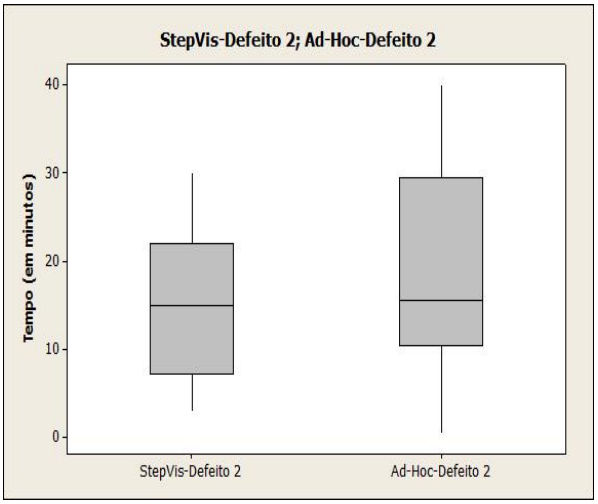
A seguir são apresentados *boxplots* para cada um dos defeitos em relação ao uso da técnica "*stepwise abstraction* + visualização" e da técnica "ad-hoc". Todo *boxplot* é representado em relação ao tempo e ao defeito identificado.

Uma breve análise dos *boxplots* revela que os participantes que utilizaram a técnica "*stepwise abstraction* + visualização" identificaram a maior parte dos defeitos quase que no mesmo intervalo de tempo, uma vez que a variabilidade existente na aplicação da técnica

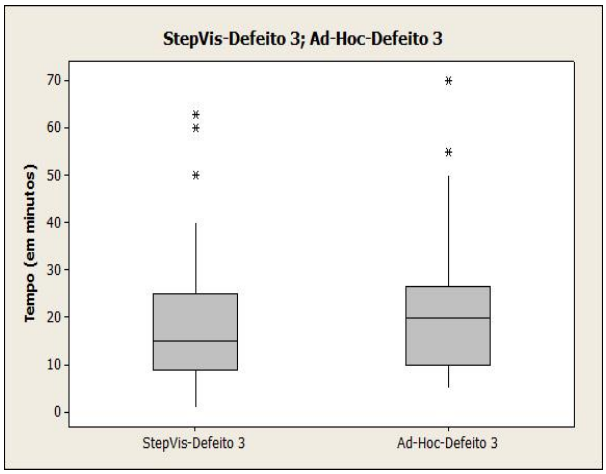
"*stepwise abstraction* + visualização" é menor do que a variabilidade na aplicação da técnica "ad-hoc" (Observe a diferença nos tamanhos das caixas do gráfico da Figura 3.2 (a)). Essa informação é interessante, pois o defeito 1 corresponde a um defeito de fácil identificação e o uso da técnica "*stepwise abstraction* + visualização" proporcionou a identificação do defeito por todos os participantes, com uma baixa variabilidade de tempo.



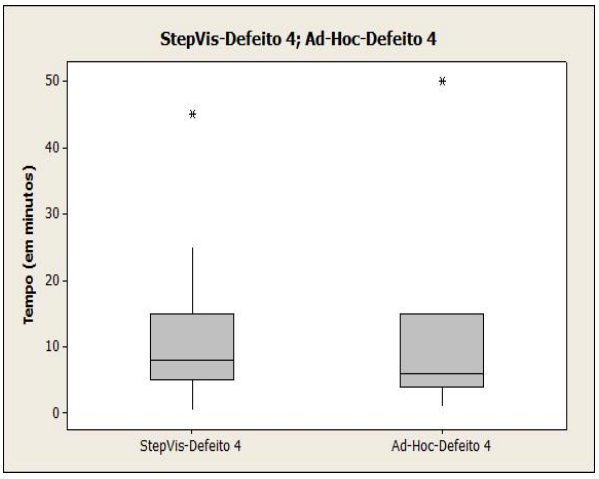
(a) Defeito 1



(b) Defeito 2



(c) Defeito 3



(d) Defeito 4

Figura 3.2. Tempo gasto inspecionando o código para encontrar os defeitos do *Paint*

3.12.2 Análise Estatística

Considerando as hipóteses formuladas anteriormente, apresenta-se, a seguir, a análise realizada para respondê-las, apresentando apenas o texto da hipótese nula.

> $H_{0,1}$: Não há diferença no número de defeitos identificados na inspeção do código do *Paint* quando comparadas a técnica "*stepwise abstraction* + visualização" com a técnica "ad-hoc".

Para essa hipótese, o número de defeitos encontrados pelos participantes que aplicaram as duas técnicas foi analisado pelo teste do *chi-quadrado*. O resultado do *p-value* obtido foi de 0,189, o que significa que a hipótese nula não pode ser rejeitada. Ou seja, nesse experimento não se tem indícios de que o número de defeitos encontrado é dependente da técnica utilizada. No entanto, se considerarmos o defeito 4 observamos que o número de participantes que o encontraram com a técnica "*stepwise abstraction* + visualização" (93,55%) foi superior ao número de participantes que encontraram o mesmo defeito com a técnica "ad-hoc" (39,29%).

> $H_{0,2}$: Não há diferença no tempo gasto na inspeção do código do *Paint* quando comparadas as técnica "*stepwise abstraction* + visualização" e a técnica "ad-hoc".

Como os dados obtidos não apresentavam uma distribuição normal, foi aplicado o teste não-paramétrico Mann-Whitney. Com um nível de significância de 95%, obteve-se como resposta um *p-value* de 0.0252, o que permite rejeitar a hipótese nula de que não há diferença de tempo na identificação de defeitos usando as técnicas "*stepwise abstraction* + visualização" e "ad-hoc". Ou seja, o uso da técnica "*stepwise abstraction* + visualização" implementada na ferramenta CRISTA teve influência no tempo de identificação de defeitos.

> $H_{0,3}$: A experiência dos participantes em inspeção de código não afeta o número de defeitos identificados quando utilizada a técnica "*stepwise abstraction* + visualização" e a técnica "ad-hoc".

Para responder essa hipótese foi utilizada a correlação de Pearson para análise estatística. Em ambos os casos não se pode rejeitar a hipótese nula, ou seja, os valores obtidos

nesse experimento indicam que a experiência em inspeção não teve influência no número de defeitos identificados, independentemente da abordagem utilizada. Em relação à experiência em inspeção, os valores do *p-value* para os grupos que aplicaram as técnicas "*stepwise abstraction* + visualização" e "ad-hoc" foram 0,073 e 0,0935, respectivamente.

A experiência em inspeção de software está apresentada na Figura 3.3 para os dois grupos de participantes definidos no experimento.

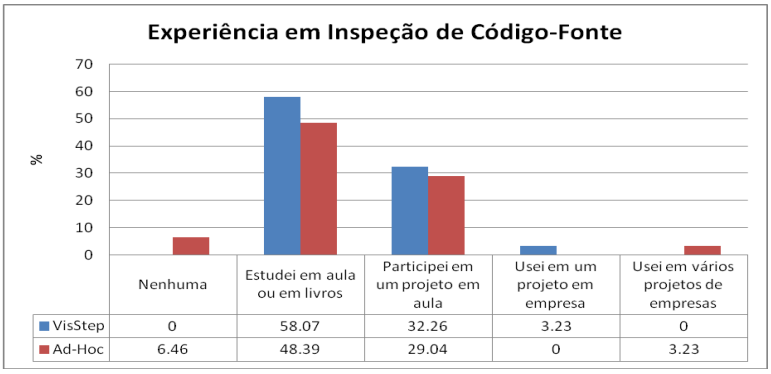
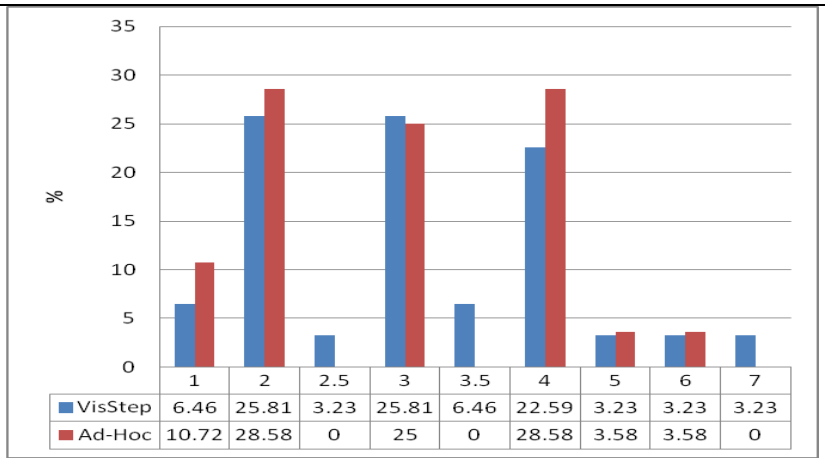


Figura 3.3. Experiência dos participantes em inspeção de código-fonte

$H_{0,4}$: A experiência dos participantes na linguagem de programação Java não afeta o número de defeitos identificados usando a técnica "*stepwise abstraction* + visualização" e a técnica "ad-hoc".

Para responder essa hipótese a análise estatística foi realizada utilizando a correlação de Pearson. Em ambos os casos não se pode rejeitar a hipótese nula, ou seja, os valores obtidos nesse experimento indicam que a experiência na linguagem de programação Java não teve influência no número de defeitos identificados, independentemente da técnica utilizada. Em relação à experiência em Java, os valores de *p-value* para os grupos que aplicaram as técnicas "*stepwise abstraction* + visualização" e "ad-hoc" foram 0,285 e 0,475, respectivamente.

A experiência em Java está apresentada na Figura 3.4, para os dois grupos de participantes definidos no experimento.



	1	2	2.5	3	3.5	4	5	6	7
VisStep	6.46	25.81	3.23	25.81	6.46	22.59	3.23	3.23	3.23
Ad-Hoc	10.72	28.58	0	25	0	28.58	3.58	3.58	0

Figura 3.4. Experiência (em anos) dos participantes do experimento na linguagem de programação Java

3.13 Considerações Finais

De acordo com o experimento e as análises realizadas observa-se que o uso da técnica "*stepwise abstraction* + visualização" auxiliou na identificação de defeitos e que para o contexto em que o experimento foi realizado, a experiência na atividade de inspeção e na linguagem de programação não teve influência sobre o resultado.

Pelos resultados obtidos e pela análise dos questionários observaram-se os seguintes pontos:

- O fato de utilizar uma técnica de leitura, como a "*stepwise abstraction* + visualização", possibilitou que os inspetores pudessem ter uma maior compreensão do código-fonte já que a técnica exige a abstração de todos os elementos do código, do mais interno ao mais externo por mais simples que eles sejam.
- Embora o fato de ter que abstrair cada elemento de código possa ser uma desvantagem em relação a uma técnica "ad-hoc", foi observado que a aplicação da técnica "*stepwise abstraction* + visualização" levou a um melhor entendimento de todo projeto, o que facilitou a identificação dos defeitos.
- O fato da aplicação da técnica "*stepwise abstraction* + visualização" com apoio computacional (ferramenta CRISTA) possibilita que os dados da própria inspeção sejam manipulados e permitam a geração de diversos relatórios que auxiliam na compreensão do código e, consequentemente, na própria atividade de identificação de defeitos.

Em geral, os resultados obtidos fornecem indícios de que a técnica "*stepwise abstraction* + visualização" auxilia o inspetor na compreensão de código e na identificação de defeitos do código-fonte.

Capítulo 4

CICLO 2: MAPEAMENTO SISTEMÁTICO

Este capítulo apresenta os resultados de um mapeamento sistemático realizado no contexto de técnicas de inspeção de código-fonte.

4.1 Considerações iniciais

O mapeamento sistemático realizado utilizou como base para sua definição o paradigma GQM, conforme ilustrado na Figura 4.1. O paradigma GQM (BASILI; CALDIERA; ROMBACH, 1994) foi adaptado para utilização no planejamento do mapeamento sistemático, onde o *Goal* corresponde ao objetivo do mapeamento sistemático, *Question* corresponde a questão primária do mapeamento sistemático e *Metric* corresponde aos atributos para extração de dados do mapeamento sistemático.

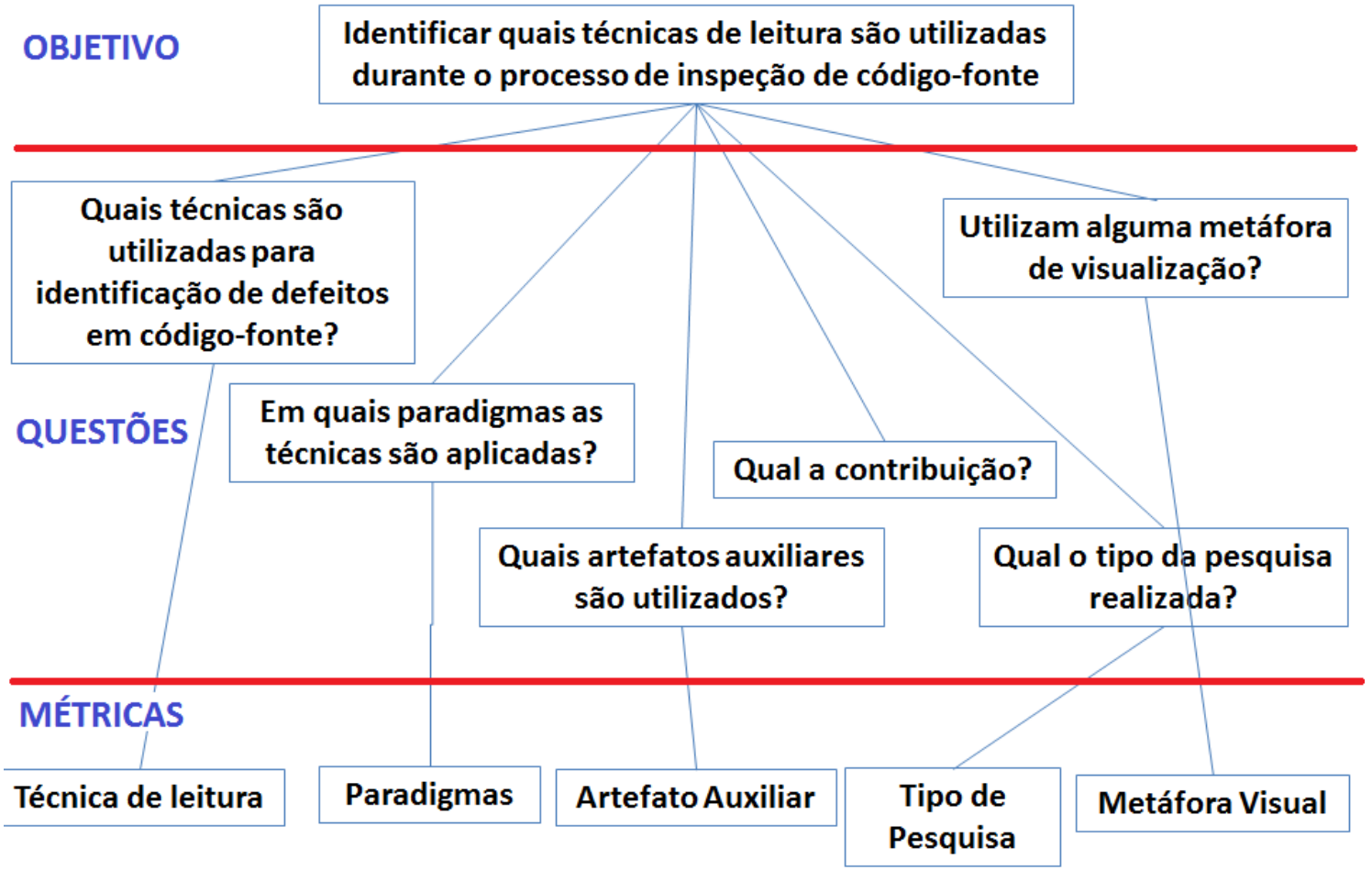


Figura 4.1. GQM de planejamento da revisão bibliográfica

O protocolo do mapeamento sistemático foi realizado de acordo com Petersen *et al*. (2008) e a ferramenta StArt (FABBRI et al., 2012) foi utilizada como apoio computacional para essa atividade. A Tabela 4.1 apresenta o protocolo desenvolvido.

Tabela 4.1. Protocolo de condução do mapeamento sistemático para a questão definida

Title:	Técnicas de Inspeção de Código
Researchers:	Anderson Belgamo; Sandra Fabbri;
Description:	Levantar o estado da arte de técnicas de leitura para inspeção de código-fonte.
Objectives:	Identificar quais técnicas de leitura para detecção de defeitos são utilizadas durante o processo de inspeção de código-fonte.
Main Question:	Quais técnicas de leitura são utilizadas para detecção de defeitos de código-fonte durante o processo de inspeção de software?
Intervention:	artigos que descrevem técnicas de leitura para inspeção de código-fonte
Control:	-
Population:	artigos que relatem o uso de técnicas de leitura para inspeção de código
Results:	técnicas de leitura para inspeção de código
Application:	inspeção de código
Keywords:	code inspection; code review
Source Selection Criteria:	exportar .bib;
Studies Languages:	português ou inglês;
Source Search Methods:	a *string* base será definida na Scopus, refinada e quando for julgado que a *string* é adequada, será traduzida para as outras máquinas de busca;
Source Engine:	ACM; IEEE; Web of Science; Scopus; Engineering Village
Studies inclusion (I) and exclusion (E) criteria:	(I) code inspection technique; (E) introdução de proceedings; (E) não apresenta técnica de inspeção de código; (E) não está em inglês ou em português; (E) o full paper não está disponível na web ou no COMUT da UFSCar; (E) introdução a pôster
Studies types definition:	todos os tipos de estudos serão considerados;
Initial studies selection:	Os critérios de inclusão e exclusão serão aplicados com base no *abstract*, *keywords* e título.
Studies quality evaluation:	não será avaliada a qualidade dos estudos;
Information Extraction Fields:	Tipo de contribuição (ferramenta, método, processo); Tipo de pesquisa (proposta de solução, validação, avaliação, proposta conceitual, relato de experiência, opinião); Nome da técnica de inspeção; Abordagem de Avaliação (Experiência, Estudo de Caso, Estudo empírico); Técnica de visualização de código (qual?); Contexto (industrial ou acadêmico); Ano de publicação; Canais de publicação; Domínio de Aplicação (Sistema de Informação, Sistema Embarcado, Não classificado); Paradigma de Linguagem de Programação (Estruturado, OO, Funcional, Lógico, Não classificado); Artefato Auxiliar (Requisitos, UML, não definido)
Results Summarization:	Os resultados serão tabulados e analisados de maneira *ad-hoc*.

Para definir as *strings* de busca do mapeamento sistemático, ensaios foram feitos na máquina de busca SciVerse Scopus[1] para encontrar a string apropriada e permitir que esta fosse aplicada em cada uma das máquinas de busca selecionadas.

[1] www.scopus.com

Com as strings definidas, foi criado um protocolo de pesquisa para cada questão. A mesma string foi aplicada nas máquinas de busca IEEE, Web of Science, ACM, Science Direct e Engineering Village.

A seguir, um resumo do mapeamento sistemático conduzido é apresentado, assim como observações acerca dos estudos lidos por completo. Ressalta-se que os gráficos, tabelas e imagens apresentadas neste capítulo e que exibem dados referentes ao mapeamento consideram apenas os estudos aceitos.

Considerando o objetivo da tese optou-se por identificar quais técnicas de leitura para inspeção de código-fonte são relatadas na literatura, bem como o levantamento de ferramentas utilizadas, ano de publicação do trabalho, local de publicação, paradigma de programação envolvido e utilização de outros artefatos que apoiem a atividade de identificação de defeitos.

Alguns atributos de extração de dados tiveram suas opções de respostas definidas previamente durante o planejamento do protocolo do mapeamento sistemático, conforme relatado a seguir:

- Tipo de contribuição: dividida nas seguintes categorias:
    1. Ferramenta: refere-se às contribuições que focam em ferramentas de apoio ao processo de inspeção, particularmente na identificação de defeitos.
    2. Técnica: refere-se às contribuições que apresentam uma técnica de inspeção de código.
    3. Processo: refere-se às contribuições que apresentam um processo de inspeção de código-fonte.
- Tipo de pesquisa: baseado no esquema proposto por Wieringa *et al*. (2006) para classificação de pesquisa conforme as seguintes categorias:
    1. Proposta de Solução: resolve um problema fornecendo uma solução inédita ou uma extensão significativa de uma técnica existente.
    2. Pesquisa de Validação: examina uma proposta de solução que ainda não tem sido aplicada na prática. Deve ser conduzida de maneira sistemática usando experimentos, protótipos, simulações, análise matemática, etc.
    3. Pesquisa de Avaliação: ao contrário da pesquisa de validação, a pesquisa de avaliação auxilia em examinar uma solução que já foi aplicada na prática. Investiga-se a implementação prática da solução e apresenta resultados usando estudos de casos, etc.

4. Proposta Conceitual: apresenta um arranjo para visualizar algo que já existe, de uma maneira inédita. Entretanto, não resolve precisamente um problema particular. Exemplos são taxonomias, frameworks teóricos, etc.
5. Relato de Experiência: relata a experiência pessoal do autor em um ou mais projetos da vida real.
6. Relato de Opinião: relata uma experiência pessoal de um autor no uso de uma técnica ou ferramenta específica

Por meio do mapeamento sistemático conduzido foi possível identificar que grande parte dos estudos divide-se em duas categorias, conforme relatado em Aurum *et al.* (2002):

- Processo de Inspeção: referente a estudos que se concentram no processo de inspeção, especificamente em termos de fases do processo, número de inspetores necessários, tempo de condução da inspeção, etc.
- Técnicas de Inspeção: referente ao uso de técnicas, métodos e ferramentas durante a fase de detecção de defeitos do processo de inspeção.

A Figura 4.2 apresenta a quantidade de artigos aceitos, duplicados e excluídos do mapeamento sistemático realizado.

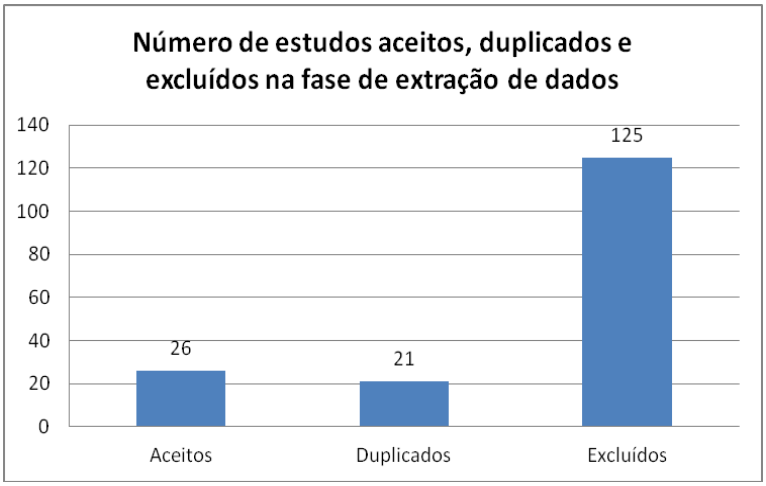
- 

Figura 4.2. Número de estudos aceitos, excluídos e duplicados na fase de extração de dados.

Ressalta-se que o grande número de artigos que foram excluídos na fase de extração se deu pelo fato de que na fase de seleção (na qual os artigos são selecionados baseados na leitura do seu *abstract*) grande parte dos artigos mencionava o uso de inspeção de código. No entanto, durante a leitura do artigo como um todo, percebeu-se que não era relatada uma técnica de inspeção de código-fonte e, portanto, o artigo foi excluído da extração de dados.

A Figura 4.3 apresenta a distribuição dos artigos em relação aos canais de publicação e a Figura 4.4 a distribuição dos artigos aceitos em relação ao ano de suas publicações.

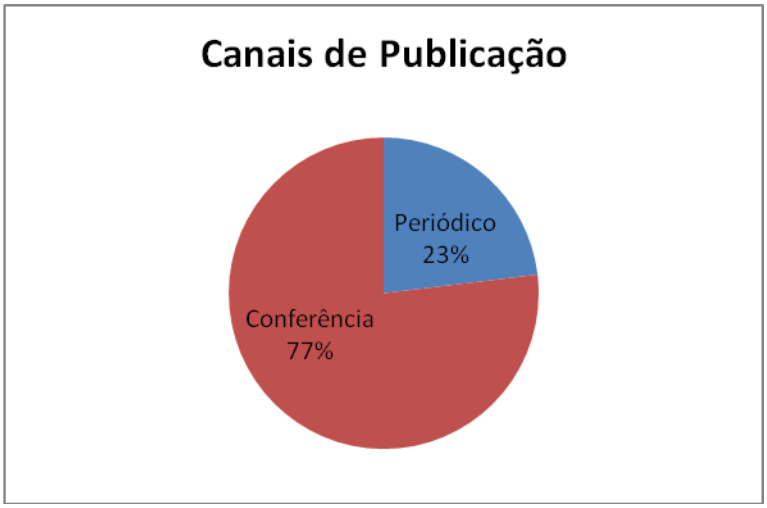


Figura 4.3. Canais de Publicação dos Artigos Selecionados.

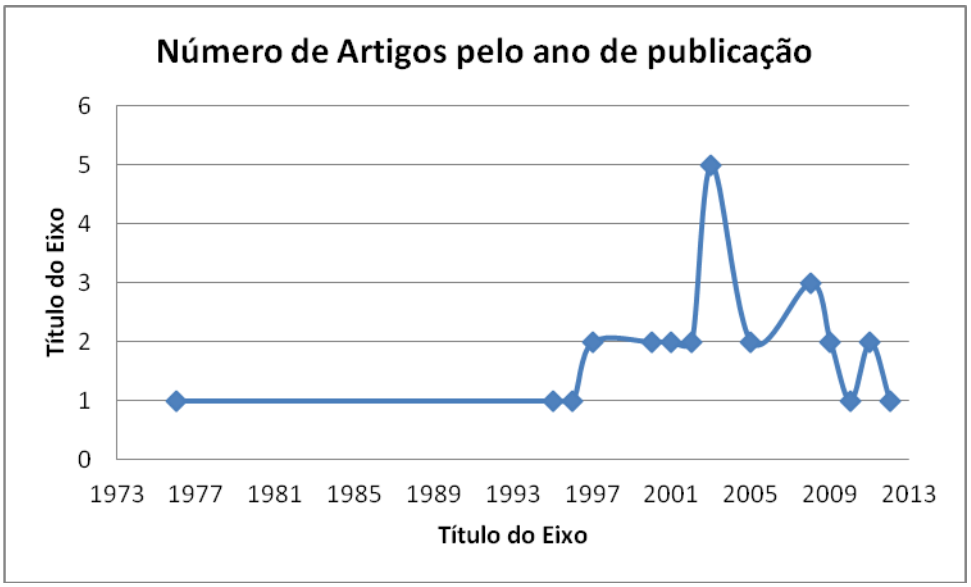


Figura 4.4. Número de artigos pelo ano de publicação

Dos 26 artigos analisados (artigos aceitos), foram identificados relatos de nove técnicas de leitura que foram definidas para a detecção de defeitos em código-fonte durante o processo de inspeção. O gráfico exibido na Figura 4.5 mostra quais são essas técnicas e o número de vezes em que elas foram utilizadas nos artigos.

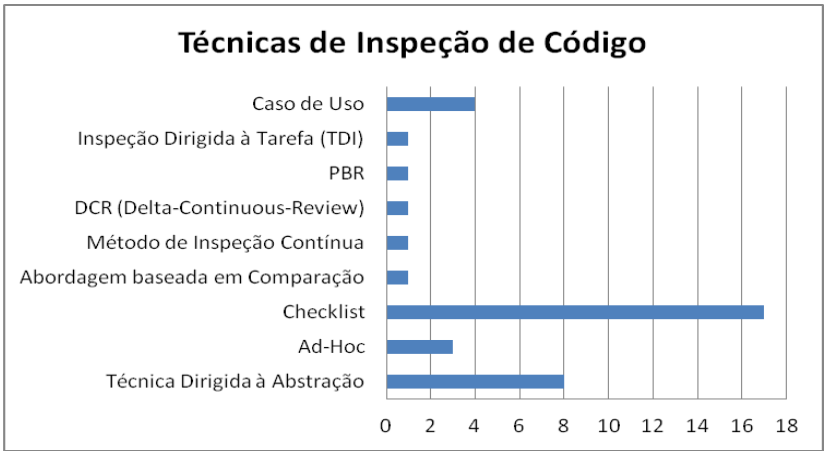


Figura 4.5. Técnicas de inspeção de código utilizadas nos artigos selecionados

No momento de classificação dos artigos observou-se que algumas técnicas eram mencionadas com termos distintos mas, ao verificar os artigos verificou-se que se tratavam da mesma técnica. Por exemplo, alguns autores relataram os termos "*Abstraction-driven Technique*" e "*Stepwise Abstraction*" se referenciado a mesma técnica. Portanto, optou-se pelo termo "Técnica Dirigida à Abstração" (em inglês, "*Abstraction-driven Technique*").

Relacionando a questão de pesquisa desse mapeamento sistemático, é possível observar que a técnica *checklist* é a mais citada nos artigos aceitos, sendo seguida pela técnica dirigida à abstração e caso de uso.

Especificamente relacionado a técnica *checklist*, os trabalhos de Dusnmore *et al.* (2003a), Dunsmore *et al.* (2003b), Kelly e Shepard (2004) e Mcmeekin *et al.* (2008) utilizam a técnica *checklist* como base para comparação com outras técnicas de leitura.

Em diversos trabalhos são apresentados diferentes tipos de *checklists*, variando na forma de utilização da técnica, nos formatos das questões elaboradas e nos propósitos de uso, como por exemplo, para sistemas críticos (BELLI; CRISAN, 1996) (DE ALMEIDA et al., 2003) (HAVELUND; HOLZMANN, 2011) e sistemas de informação (FAGAN, 1999) (Fagan, 1976) (FISHER; CUKIC, 2001) (HÖST; JOHANSSON, 2000) (HATTON, 2008) (JUN-SUK OH; HO-JIN CHOI, 2005) (MCMEEKIN et al., 2008).

Com relação ao tipo de contribuição dos artigos selecionados, o gráfico exibido na Figura 4.6 apresenta os resultados obtidos.

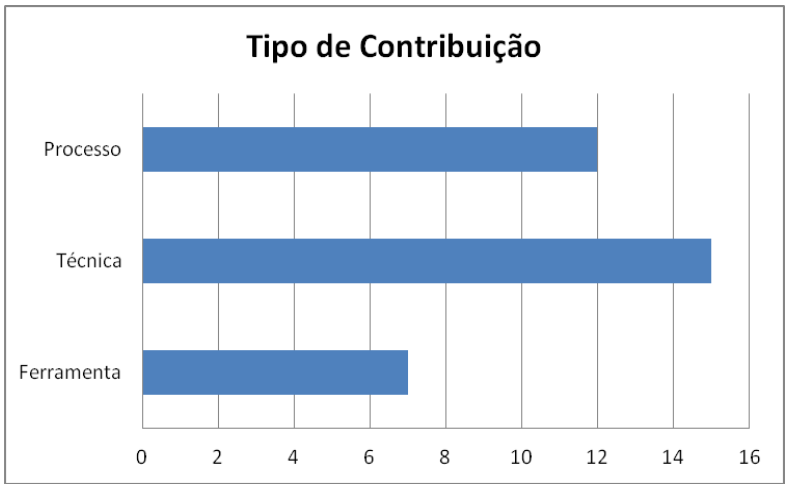


Figura 4.6. Tipo de contribuição dos artigos selecionados

Observa-se no gráfico da Figura 4.6 que a maior contribuição foi para técnica, seguida por processo e ferramenta. Essa característica deve-se ao fato de que a aceitação dos artigos para extração deveria passar pelo critério de possuir a descrição de uma técnica de inspeção de código ou ferramenta. Ressalta-se que um mesmo artigo pode apresentar mais de uma contribuição.

Com relação ao tipo de pesquisa relatada nos artigos selecionados verifica-se que grande parte apresenta uma proposta de solução com alguma avaliação ou validação. Dos 26 artigos, apenas sete artigos apresentam uma proposta de solução sem uma avaliação ou validação. O gráfico da Figura 4.7 apresenta os tipos de pesquisa identificados em cada artigo selecionado. Ressalta-se que um mesmo artigo pode apresentar mais de um tipo de pesquisa.

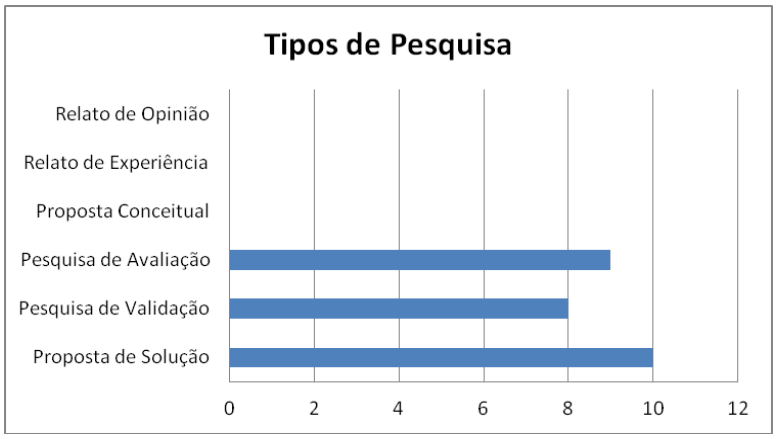


Figura 4.7.Tipo de Pesquisa descrito nos artigos

Observam-se na Figura 4.8 os artigos classificados de acordo com os tipos de pesquisa e quais os artigos são classificados em mais de um tipo de pesquisa. Os números representados nos conjuntos representam os artigos selecionados e listados na Tabela 4.2.

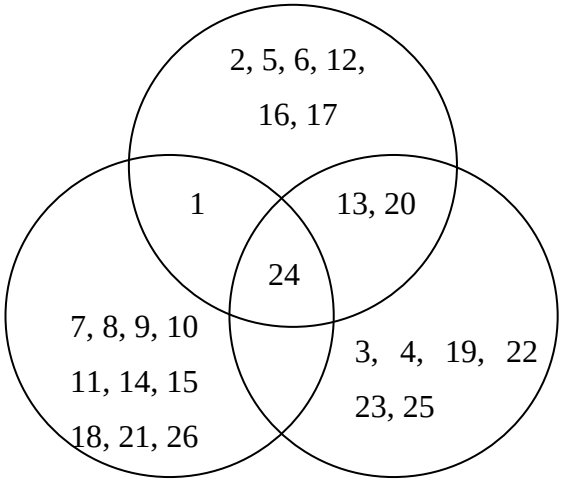


Figura 4.8. Artigos apresentados de acordo com sua classificação

Tabela 4.2. Nome e Ano de Publicação dos Artigos Selecionados

#	Título	Ano de Publicação
1	Empirical performance analysis of computer-supported code-reviews	1997
2	Towards automation of checklist-based code-reviews	1996
3	A Task-Based Code Review Process and Tool to Comply with the DO-278/ED-109 Standard for Air Traffic Management Software Development: An Industrial Case Study	2011
4	Applying continuous code reviews in airport operations software	2012
5	A Tool to Support Perspective Based Approach to Software Code Inspection	2005
6	Best practices in code inspection for safety-critical software	2003
7	The development and evaluation of three diverse techniques for object-oriented code inspection	2003
8	Further investigations into the development and evaluation of reading techniques for object-oriented code inspection	2002
9	Practical code inspection techniques for object-oriented systems: an experimental comparison	2003
10	Systematic object-oriented inspection - An empirical study	2001
11	Object-oriented inspection in the face of delocalisation	2000
12	Design and code inspections to reduce errors in program development	1976
13	Automating techniques for inspecting high assurance systems	2001
14	Evaluation of code review methods through interviews and experimentation	2000
15	Testing the value of checklists in code inspections	2008
16	Software certification - Coding, code, and coders	2011
17	SCRUB: A tool for code reviews	2010
18	A reflective practice of automated and manual code reviews for a studio project	2005
19	Qualitative observations from software code inspection experiments	2002
20	A comparison-based approach for software inspection	1995
21	Checklist Based Reading's Influence on a Developer's Understanding	2008

22	Checklist Inspections and Modifications: Applying Bloom's Taxonomy to Categorise Developer Comprehension	2008
23	Measuring cognition levels in collaborative processes for software engineering code inspections	2009
24	CRISTA: A tool to support code comprehension based on visualization and reading technique	2009
25	An empirical evaluation of defect detection techniques	1997
26	Detection or isolation of defects? An experimental comparison of unit testing and code inspection	2003

Observa-se que de todos os artigos selecionados, apenas o trabalho de Porto *et al.* (2009) apresentou o uso de visualização de código-fonte para a inspeção de código-fonte. Esse fato pode ser um indício de que o uso da visualização de código com o objetivo de identificar defeitos é ainda pouco explorado.

Com relação às técnicas apresentadas nos artigos, observa-se que a técnica *Checklist* é a que ocorre com maior frequência (conforme Figura 4.5) e nos trabalhos 8, 9, 10, 20 e 24 da Tabela 4.2, a técnica de *checklist* é utilizada como critério de comparação com outras técnicas.

Nos trabalhos de Dunsmore *et al.* (2000a), Dunsmore *et al.* (2003a) e Dunsmore *et al.* (2003b) são apresentados a técnica de leitura baseada em caso de uso e os resultados empíricos de estudos realizados comparando as técnicas *checklist*, dirigida à abstração e caso de uso. Em Dunsmore *et al.* (2000a) são apresentadas evidências que o fato do código orientado a objetos ser "espalhado" é um problema real para a inspeção de software. Em Dunsmore *et al.* (2003a) e Dunsmore *et al.* (2003b) observaram-se uma diferença significativa no número de defeitos descobertos por participantes utilizando as técnicas *checklist*, baseada em abstração e baseada em caso de uso.

Em relação ao paradigma de programação na qual a técnica de inspeção pode ser aplicada, os estudos selecionados apresentam que 13 artigos relatam aplicação das técnicas em código orientado a objetos, 10 artigos relatam aplicação das técnicas em código estruturado e em 4 artigos não foi possível realizar a classificação pois não havia informações suficientes. Ressalta-se que no artigo de Havelund e Holzmann (2011) são apresentadas técnicas que podem ser aplicadas tanto no paradigma orientado a objetos quanto no estruturado.

Por fim, também foi extraído se as técnicas descritas nos artigos selecionados utilizavam algum tipo de artefato auxiliar (por exemplo, o Documento de Requisitos,

Diagrama da UML, etc.) como apoio para a inspeção de código. Os resultados são apresentados no gráfico da Figura 4.9.

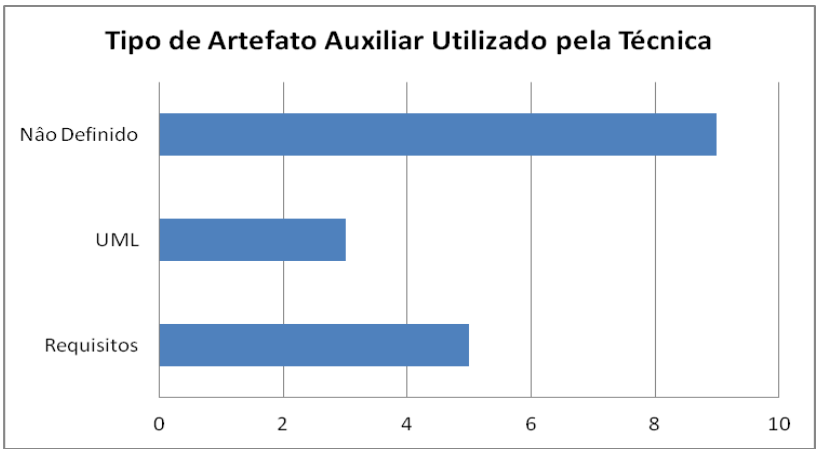


Figura 4.9. Tipo de Artefato Auxiliar utilizado pela técnica de leitura durante o processo de inspeção

Capítulo 5

CICLO 3: UM ESTUDO EXPLORATÓRIO PARA INVESTIGAÇÃO DO USO DE ARTEFATOS AUXILIARES DURANTE A ATIVIDADE DE INSPEÇÃO DE CÓDIGO

Este capítulo apresenta os resultados de um estudo exploratório realizado para investigar como artefatos auxiliares (documentos de requisitos, diagramas UML e métricas de software) são utilizados por inspetores durante a atividade de inspeção de código.

5.1 Considerações iniciais

Conforme observado na literatura, existem diversas técnicas de leitura que utilizam artefatos auxiliares durante a inspeção de software como, por exemplo, PBR (documentos de requisitos de software, diagrama de análise, código-fonte), UBR e Use Case (casos de uso e cenários de usos), OORTs (documento de requisitos de software e diagramas UML). Além disso, métricas de software têm sido utilizadas durante atividades de manutenção e tarefas de evolução de software, como apresentado nos trabalhos de Nishizono *et al.* (2011) e Counsell *et al.* (2005).

Com base no uso de informações auxiliares, este capítulo tem como objetivo apresentar evidências obtidas do estudo exploratório que avaliou como os inspetores usaram informações-chave (documento de requisitos, diagramas UML e métricas de software) durante a atividade de inspeção de código-fonte.

Nas próximas seções relatam-se as principais etapas do estudo, conforme o processo definido por Wohlin *et al.* (2000).

5.2 Escolha das Informações-Chave

Para auxiliar o inspetor de código na compreensão e na identificação de defeitos, estabeleceram-se como informações-chave para uso dos inspetores as seguintes informações:

- Documento de Requisitos: base de referência para o desenvolvimento de qualquer software.
- Diagrama de Classes: representa a arquitetura da aplicação, mostrando as principais classes do negócio e seus relacionamentos.
- Métricas: representam medidas quantitativas do código-fonte que podem auxiliar o inspetor durante a definição da ordem de inspeção das classes de um sistema orientado a objetos.

As métricas de software foram os artefatos utilizados com mais frequência durante a atividade de inspeção, que pode estar associado ao fato de que uma das principais dificuldades relatadas na literatura diz respeito à ordem na qual as classes de um sistema orientado a objeto deve ser inspecionada. Tal dificuldade é inerente do próprio processo de inspeção de código orientado a objetos e já foi relatado nos trabalhos de Dunsmore *et al.* (2000a) e Dunsmore *et al.* (2003).

Nesse sentido partiu-se do princípio do uso de métricas de software para a geração inicial da ordem de classes a serem inspecionadas. Para tanto, as métricas utilizadas foram as métricas definidas no trabalho de Chidamber e Kemerer (1994). A escolha das métricas de Chidamber e Kemerer (1994) deve-se ao fato das técnicas de leitura ser focada em software orientado a objetos e devido ao fato de ser um conjunto de métricas bem estabelecido e utilizado com frequência em diversos trabalhos, como em Subramanyam e Krishnan (2003), Olague *et al.* (2007) e Srivastava e Kumar (2013).

As métricas utilizadas, bem como as definições, são sucintamente apresentadas:

- Acoplamento entre Objetos (CBO - *Coupling Between Objects*): classes são acopladas quando os métodos em uma classe usam métodos ou variáveis de instância definidas em uma classe diferente. CBO é a medida de quanto acoplamento existe. Um valor alto para o CBO de uma classe significa que a classe

é altamente dependente e, portanto, maior a probabilidade de uma mudança em uma classe afetar outras classes do sistema.

- Falta de coesão em métodos (LCOM - *Lack Of Cohesion in Methods*): é calculada considerando os pares de métodos em uma classe. LCOM é a diferença entre o número de pares de métodos sem atributos compartilhados. O valor dessa métrica tem sido amplamente discutido e possui variações de como calculá-la.
- Métodos ponderados por classe (WMC - *Weighted Methods per Class*): o número de métodos em cada classe ponderados pela complexidade de cada método. Pode ser dita simplesmente como a soma da complexidade de cada método. Pode-se usar a complexidade ciclomática ou simplesmente atribuir o valor 1 como complexidade de cada método.
- Árvore de Profundidade de Herança (DIT - *Depth of Inheritance Tree*): representa o número de níveis na árvore de herança em que as subclasses herdam atributos e operações (métodos) de superclasses. Quanto mais profunda a árvore de herança, mais complexo o projeto. Em casos em que envolvem múltiplas heranças, o DIT seria o tamanho máximo determinado entre a subclasse e a classe principal.
- Número de filhos (NOC - *Number Of Children*): é uma medida do número de subclasses imediatas em uma classe. Ele mede a largura de uma hierarquia de classe, considerando que o DIT mede sua profundidade. Um valor alto para NOC pode indicar um maior reúso. Isso pode significar que mais esforço deve ser realizado na validação de classes de base por causa do número de subclasses que dependem delas.
- Resposta para uma classe (RFC - *Response For a Class*): é a medida do número de métodos que poderiam ser executados em resposta a uma mensagem recebida por um objeto dessa classe. RFC está relacionada com a complexidade e é obtida pela soma dos métodos chamados para realizar uma determinada funcionalidade. Quanto maior o valor de RFC, mais complexa é a classe e, portanto, mais provável que possua erros.

5.3 Definição do Experimento

O planejamento do experimento foi elaborado de acordo com a estrutura a seguir:

Analisar	a atividade de inspeção de código-fonte
Com o propósito de	caracterizar
Em relação à	padrões comportamentais relacionados a decisões tomadas por inspetores quando informações-chave são ou não usadas.
Do ponto de vista	de pesquisadores de inspeção de código
No contexto de	estudantes do Curso de Ciência da Computação e Engenharia da Computação da Universidade de Federal de São Carlos.

5.4 Seleção do Contexto

O experimento foi realizado durante a disciplina de Engenharia de Software dos cursos de Ciência da Computação e Engenharia da Computação da Universidade Federal de São Carlos. Os participantes receberem uma nota apenas pela participação na atividade e não pelo desempenho. Ressalta-se que qualidade do relato do processo utilizado de cada participante durante a atividade de inspeção foi levado em consideração para a atribuição de notas, de modo que os participantes pudessem registrar o maior nível de detalhes possível do processo utilizado.

De acordo com Wohlin *et al.* (2000), o contexto do experimento pode ser caracterizado de acordo com aos quatro dimensões a seguir:

- *Off-line*: o software inspecionado não foi desenvolvido pela indústria e o estudo foi executado como parte de uma disciplina de Engenharia de Software.
- Estudante: os participantes eram estudantes da graduação.
- *Toy*: embora a aplicação selecionada tivesse todas as características do paradigma orientado a objetos, a ferramenta *Paint* não era uma aplicação real desenvolvida pela indústria.
- Específico: os resultados do experimento não podem ser generalizados para outros contextos.
-

5.5 Seleção dos Participantes

A seleção dos participantes foi baseada na conveniência, isto é, a disciplina de Engenharia de Software II estava sendo ministrada aos alunos dos cursos de Engenharia da Computação e Ciência da Computação da Universidade Federal de São Carlos. Nesse sentido, os alunos já tinham conhecimento prévio de modelagem de sistemas, documento de requisitos e métricas de software e estavam sendo apresentados aos conceitos de verificação, validação e testes.

5.6 Projeto Experimental

Os participantes foram divididos em dois grupos (G1 e G2) que tinham como objetivo descrever os passos realizados durante a inspeção e identificar defeitos no software *Paint* usando a ferramenta Crista. Os dois grupos receberam treinamento no uso da ferramenta Crista e então o G1 recebeu a documentação do *Paint*, bem como valores de métricas obtidas a partir do código-fonte. Detalhes do projeto experimental são apresentados na Tabela 5.1.

Tabela 5.1. Projeto Experimental

Dia	G1 (16 participantes)	G2 (17 participantes)	Duração
Dia 1	Caracterização dos Participantes - Questionário		10 minutos
	Treinamento em Inspeção de Software		30 minutos
	Treinamento em Inspeção de Código-Fonte usando a ferramenta CRISTA e realização de exercícios		60 minutos
	Treinamento em interpretação de valores de métricas, documento de requisitos e diagramas UML.	Não houve nenhuma atividade	40 minutos
Dia 2	Realização da inspeção do Paint com a CRISTA e uso de métricas, documento de requisitos e diagramas UML	Realização da inspeção do Paint com a CRISTA	90 minutos
Dia 3	Discussão dos resultados		90 minutos

Ressalta-se que o grupo G2 não recebeu nenhum artefato auxiliar, apenas o código-fonte para inspeção. Desse modo tinha-se o objetivo de verificar se o uso de artefatos auxiliares influenciava na identificação de defeitos usando a ferramenta CRISTA. Além disso, nenhum dos .grupos tinha ciência da quantidade de defeitos existentes no software Paint

5.7 Instrumentação

O sistema alvo utilizado para o estudo foi o *Paint*. O software *Paint* é um simples editor de figura escrito na linguagem Java. Ele permite ao usuário desenhar, apagar, limpar e desfazer elementos coloridos (usando o padrão RGB). Esse software também foi utilizado nos estudos de Ko *et al.* (2006) e Robillard *et al.*, (2004).

Não havia nenhuma documentação oficial do software *Paint* e o código não possui comentários. O *Paint* foi implementado com nove classes Java com uma média de 73 linhas de código. O software *Paint* não é um sistema complexo, no entanto, o objetivo de estudar um sistema de menor porte permitiu-nos investigar as diferentes fontes de informações utilizadas pelos participantes para a identificação de defeitos em código-fonte.

Como o *Paint* não possuía documentação oficial, toda a documentação e valores das métricas foram desenvolvidos e obtidos pelos autores do experimento. A Tabela 5.2 apresenta um breve resumo dos artefatos recebidos por cada grupo:

Tabela 5.2. Artefatos utilizados pelos participantes dos grupos

Artefato	Grupo 1	Grupo 2
Código-fonte do *Paint*	SIM	SIM
Documento de Requisitos do *Paint*	SIM	NÃO
Diagrama de Classes do *Paint*	SIM	NÃO
Métrica de Software (acoplamento, coesão, complexidade ciclomática, LOC)	SIM	NÃO

Além disso, ambos os grupos receberam uma planilha para marcação dos passos usados durante o processo de inspeção. Os campos dessa planilha são apresentados na Tabela 4.3, com um exemplo do seu preenchimento.

Pela Tabela 5.3, observa-se que o inspetor partiu do documento de requisitos (Requisito Funcional RF1) no instante de tempo 11h15min e resolveu ir para a classe *Pencil* de acordo com as justificativas apresentadas na primeira linha da última coluna. No instante de tempo 11h20min ele acessou a classe *Pencil* e registrou um defeito de número 1 na ferramenta CRISTA e resolveu ir para a classe *Actions* de acordo com o motivo relatado na

segunda linha da última coluna. É importante ressaltar que o inspetor preenche a coluna **#Defeito do Código** com o número de defeito gerado pela ferramenta CRISTA. O tempo exato do registro do defeito fica automaticamente armazenado na ferramenta CRISTA.

Tabela 5.3. Planilha de Preenchimentos dos Participantes

Tempo	Informação Utilizada	# Defeito do Código	Motivo que levou na tomada de decisão de ir da classe atual para a próxima classe
11h15min	Requisito Funcional RF1		O requisito RF1 mencionava dados sobre a funcionalidade de desenho, então resolvi ir para a classe Pencil pois aparentemente deveria tratar da funcionalidade descrita no requisito.
11h20min	Classe Pencil	1	A Classe Pencil referencia uma chamada ao método da classe Actions.
11h27min	Classe Actions	.....	.....

5.8 Preparação e Execução

O estudo foi dividido em três fases: fase de treinamento em inspeção de software, fase de treinamento na ferramenta Crista e a fase de inspeção de código efetivamente. Nas fases de treinamentos em inspeção de software e na ferramenta Crista não houve divisão de grupos visto que o objetivo era apresentar os objetivos, características e nomenclaturas utilizadas na inspeção de código-fonte.

Na fase de treinamento em inspeção de software, os participantes foram apresentados à teoria de inspeção de software (processo de inspeção, participantes, taxonomia de defeitos, relatório de discrepâncias), especificamente voltado à inspeção de código. Para fixação dos conceitos, uma aplicação de inspeção de código foi realizada em um estudo de caso utilizando um sistema orientado a objetos escrito em Java.

Na fase de treinamento na ferramenta Crista, os participantes foram treinados no uso da ferramenta Crista para inspeção de código, sendo apresentada a técnica *Stepwise Abstraction* e o uso de visualização presentes na ferramenta. Após o treinamento, os participantes também fizeram um estudo de caso de inspeção de código com o uso da ferramenta Crista. Durante o estudo de caso realizado nesse treinamento, os participantes foram divididos em dois grupos: grupo G1 recebeu os artefatos auxiliares da aplicação exemplo e o grupo G2 recebeu apenas o código-fonte da aplicação exemplo.

Na fase de inspeção de código os participantes dos dois grupos foram mantidos: Grupo G1 (uso da ferramenta Crista com apoio de artefatos auxiliares - documento de requisitos,

diagrama de classes da UML e uso de métricas de software como LOC, Complexidade Ciclomática, Acoplamento, Coesão) e Grupo G2 (uso da ferramenta Crista apenas). O objetivo dessa divisão foi identificar estratégias utilizadas pelos participantes durante a inspeção de código e a sua consequente efetividade e eficiência na identificação de defeitos.

Os participantes trabalharam de forma individual usando um microcomputador com acesso a ferramenta Crista. Os participantes do Grupo G1 receberam a documentação do software impressa antes do início do experimento. Os participantes tinham como tarefa identificar defeitos no código e anotar todos os passos realizados durante a inspeção do código, inclusive anotando o tempo em que cada atividade realizada foi executada, conforme apresentado na Tabela 5.3.

5.9 Instrumentação

Os materiais usados durante a execução do experimento foram o formulário de consentimento do participante, o formulário de passos executados, o formulário de *feedback*, o material de treinamento, código-fonte e documento do *Paint* e a ferramenta CRISTA. Conforme apresentado no Capítulo 3, havia quatro defeitos originalmente conhecidos e relatados durante o desenvolvimento do software *Paint*, a saber:

- Defeito 1: Usuário não pode selecionar a cor amarela.
- Defeito 2: O botão "undo my last stroke" não funciona apropriadamente.
- Defeito 3: O botão "undo my last stroke" é habilitado mesmo quando nenhuma ação tem sido realizada.
- Defeito 4: Há uma opção para desenho de Linha mas ela não funciona.

Outros defeitos foram injetados no software *Paint* a fim de que uma maior variedade de tipos de defeitos pudesse ser contemplada. Os defeitos injetados foram os seguintes, apresentados em relação ao código original:

- Defeito 5: Considerado um defeito cosmético de mudança do valor de um rótulo de um botão de *Pencil* para *Penci*. Exemplo:

```
DE: pencilAction.putValue(Action.NAME, "Pencil");
PARA: pencilAction.putValue(Action.NAME, "Penci");
```

- Defeito 6: Atribuição de uma ação incorreta para o objeto pencilAction. Exemplo:

```
DE:eraserlAction.putValue(Action.NAME, "Eraser");
```

```
PARA:pencilAction.putValue(Action.NAME, "Eraser");
```

- Defeito 7: a cor “preta” foi atribuída ao objeto Eraser ao invés da cor “branca”. Exemplo:

```
DE: this.color = Color.white;
PARA: this.color = Color.black;
```

- Defeito 8: uma atribuição incorreta de tamanho de vetor. Exemplo:

```
DE: Point[] points = new Point[2];
PARA: Point[] points = new Point[3];
```

- Defeito 9: objeto incorreto inserido em um painel. Exemplo:

```
DE: colorPanel.add(bPanel);
PARA: colorPanel.add(rPanel);
```

- Defeito 10: uso incorreto do operador relacional. Exemplo:

```
DE: for(int pointIndex = points.length - 1; pointIndex >=
0; pointIndex--)
PARA: for(int pointIndex = points.length - 1; pointIndex >
0; pointIndex--)
```

- Defeito 11: uso incorreto do incremento na variável pointIndex. Exemplo:

```
DE: for(int pointIndex = points.length - 1; pointIndex >=
1; pointIndex--)
PARA: for(int pointIndex = points.length - 1; pointIndex
>= 1; pointIndex++)
```

5.10 Coleta de Dados

Os seguintes tipos de dados foram coletados: discrepâncias identificadas e relatadas na ferramenta CRISTA, a porcentagem de abstração de código coletada pela ferramenta CRISTA e os passos relatados pelos inspetores durante a atividade de inspeção de código.

5.11 Análise de Dados

A primeira atividade para a realização da análise de dados consistiu da compilação e formatação de todos os passos realizados pelos participantes para a inspeção de código bem

como as discrepâncias identificadas em cada passo, a classe que continha a discrepância relatada e o instante de tempo em que ela foi registrada. A partir dessa compilação e formatação inicial dos dados de todos os participantes deu-se início a análise dos dados obtidos.

A seguir, a análise dos dados é apresentada de acordo com os objetivos pretendidos. Durante a análise algumas evidências foram identificadas e apresentadas como resultados do estudo exploratório. As evidências identificadas podem ser divididas em Padrões Comportamentais e Análise dos Defeitos, Falso-Positivos e Melhorias identificados.

5.11.1 Frequência que uma determinada classe foi utilizada como ponto de partida para a inspeção de código

As Tabelas 5.4 e 5.5 apresentam os resultados da frequência que uma classe específica foi utilizada como ponto de partida para a inspeção de código.

Tabela 5.4. Classe Inicial para o Grupo G1

Classe usada como Ponto de Partida para Inspeção do Grupo G1	Frequência
Actions	1
PaintWindow	1
EraserPaint	1
PaintCanvas	1
PaintObject	12

Tabela 5.5- Classe Inicial para o Grupo G2

Classe usada como Ponto de Partida para Inspeção do Grupo G2	Frequência
PaintObjectConstructorListener	1
PaintWindow	2
Actions	3
PaintObject	5
EraserPaint	6

Observando os números apresentados nas Tabelas 5.5 e 5.5, 75% dos participantes do grupo que utilizou a documentação (G1) iniciaram a inspeção com a classe PaintObject, enquanto que os participantes do grupo que não utilizou a documentação tiveram uma maior heterogeneidade na escolha da primeira classe a ser inspecionada, com a classe EraserPaint possuindo a maior frequência (35%).

Evidência 1: os inspetores que utilizaram artefatos auxiliares mostraram um comportamento mais homogêneo na escolha da classe para iniciar a inspeção do software.

Ressalta-se que o fato da escolha de uma determinada classe pela maior parte dos integrantes do Grupo 1 não significa necessariamente que a efetividade do mesmo foi maior que a do Grupo 2. Essa análise será realizada posteriormente.

Outra análise que pode ser efetuada é a partir da classe que possui o menor acoplamento entre as classes do *Paint*. Nesse caso, a classe PaintObject possuía essa característica.

Evidência 2: classes com menor acoplamento são fortes candidatas a serem escolhidas como ponto inicial de uma inspeção de código.

5.11.2 Frequência que uma informação específica ou artefato de software foi utilizado como ponto de partida para a inspeção de código

As Tabelas 5.6 e 5.7 apresentam os resultados da frequência que uma determinada informação ou artefato de software foi utilizado como ponto de partida para a inspeção de código.

Tabela 5.6. Frequência do uso de informação/artefato como ponto de partida para inspetores do Grupo G1

Informação/Artefato	Frequência
Aleatório	1
Diagrama de Classes – Identificação da Classe Núcleo	1
Diagrama de Classes – Herança	1
Métrica – Menor Complexidade Ciclomática	1
Documento de Requisitos – Requisito Funcional RF1	2
Métrica – Menor Acoplamento	10

Tabela 5.7. Frequência do uso de informação/artefato como ponto de partida para inspetores do Grupo G2

Informação/Artefato	Frequência
Método main	1
Interface	1
Interface Gráfica (GUI)	1
Herança – SuperClasse	2
Classe com complexidade menor	3
Aleatório	9

É interessante observar que quando do uso de artefatos auxiliares, os inspetores diminuem a possibilidade de escolha de um motivo aleatório como ponto de partida para inspeção. Ao contrário, em grupos sem documentação, nove inspetores utilizaram a aleatoriedade para escolher o ponto inicial para inspeção de código.

Além disso, grande parte dos inspetores do Grupo 1 (62,5%) escolheu o uso da métrica "menor acoplamento" como decisão para a escolha da classe a ser inspecionada. Esse número corrobora com a *Evidência 2* apresentada anteriormente.

Outra característica observada diz respeito a maior heterogeneidade em relação a decisão da escolha dos inspetores do Grupo 2, sendo que a decisão "aleatório" representou aproximadamente 53% das escolhas dos inspetores, a decisão "Classe mais simples" representou aproximadamente 18% e "Herança – SuperClasse" representou aproximadamente 12%.

Evidência 3: mesmo possuindo um maior número de fontes de informações e artefatos auxiliares, inspetores do grupo G1 tiveram maior homogeneidade na escolha da fonte de informação/artefato para iniciar a inspeção do software.

5.11.3 Artefatos auxiliares e Informações usadas para a Tomada de Decisão pelos Inspetores Durante a Inspeção

Após analisar quais artefatos foram usados pelos inspetores, um total de 23 pares de decisão (artefatos + informação) distintos foram identificados. Por exemplo, alguns inspetores usaram o par de decisão Diagrama de Classe (artefato) mais herança (informação). Os pares de decisão são apresentados na Tabela 5.8, bem como a frequência e as porcentagens que determinada decisão foi utilizada pelos grupos, independentemente do momento e do contexto em que tal decisão tenha sido utilizada. A porcentagem foi obtida levando em consideração a somatória de todas as decisões usadas por todos os participantes dos grupos.

O número total de pares de decisões utilizadas pelo grupo G1 foi de 266 pares de decisões e pelo grupo G2 foi de 207 pares de decisões.

Os pares de decisões mais frequentemente escolhidos pelos inspetores do grupo G1 foram "Métricas de Software + Complexidade Ciclomática", "Código-fonte + Referência a Classe", e "Código-fonte + Última classe a ser inspecionada". Essas escolhas representam mais do que 48% de todos os pares de decisão usados pelo grupo G1. Em relação ao grupo G2, os mais frequentes pares de decisões foram "Código-fonte + Referência a Classe" e "Código-fonte + Aleatoriedade", os quais representam mais de 50% de todos os pares de decisões usados pelos inspetores do grupo G2.

Evidência 4: mesmo usando vários pares de decisões para tomada de decisão, alguns deles são mais utilizados do que outros. Por exemplo, "Métricas de Software +

Acoplamento". Esses pares de decisões são utilizados para guiar os inspetores na escolha das classes a serem inspecionadas.

Tabela 5.8. Pares de decisões tomadas pelos inspetores durante a atividade de inspeção de código

Pares de Decisões	Grupo 1	Grupo 2
Código-Fonte (Referência a Classe)	42 (15.7%)	57 (27.5%)
Código-Fonte (herança)	9 (3.3%)	27 (13.0%)
CRISTA (Ordem de Apresentação das Classes na Crista)	14 (5.2%)	4 (1.9%)
Código-Fonte (Iniciar pelo método main)	0 (0%)	2 (0.9%)
Aleatório	22 (8.2%)	49 (23.6%)
Código-Fonte (Última classe a ser analisada)	35 (13.1%)	7 (3.3%)
Código-Fonte (Classe Pequena)	0 (0%)	30 (14.5%)
Código-Fonte (Classe Semelhante)	0 (0%)	3 (1.4%)
Código-Fonte (Retorno a classe anteriormente inspecionada)	16 (6%)	20 (9.7%)
Código-Fonte (Nome semelhante das classes)	2 (0.7%)	2 (0.9%)
Código-Fonte (Mudança devido a interface)	0 (0%)	3 (1.3%)
Código-Fonte (Classe implementa a Interface)	0 (0%)	1 (0.4%)
Código-Fonte (Classe robusta)	0 (0%)	1 (0.4%)
Código-Fonte (Forte Acoplamento)	0 (0%)	1 (0.4%)
Código-Fonte (Interface Gráfica)	22 (8.2%)	0 (0%)
Métricas de Software (Menor Acoplamento)	4 (1.5%)	0 (0%)
Métricas de Software (Menor Acoplamento e Menor Complexidade Ciclomática)	3 (1.1%)	0 (0%)
Diagrama de Classes (Mudança devido a herança)	15 (5.6%)	0 (0%)
Métrica de Software (Menor Complexidade Ciclomática)	53 (19.9%)	0 (0%)
Diagrama de Classe (Facilidade de Entendimento da Classe)	7 (2.6%)	0 (0%)
Documento de Requisitos (Requisitos Funcionais)	14 (5.2%)	0 (0%)
Diagrama de Classes (Relacionamento entre as Classes)	7 (2.6%)	0 (0%)
Diagrama de Classes (Classe representa o núcleo do software)	1 (0.3%)	0 (0%)

5.11.4 Frequência de defeitos, falso-positivos e melhorias identificadas durante a inspeção de código-fonte

A frequência de defeitos, falso-positivos e melhorias no código identificadas por cada participante em cada um dos grupos durante o processo de inspeção e apresentada nas Tabelas 5.9 e 5.10.

Uma breve análise quantitativa revela um melhor desempenho na frequência de identificação de defeitos e melhorias pelos participantes do grupo G1 (uso de documentação e métricas).

Evidência 5: o uso da documentação e métricas do software a ser inspecionado pode auxiliar o inspetor no desempenho da atividade de inspeção.

Tabela 5.9. Frequência de defeitos, falso-positivo e melhorias para participantes do Grupo 1

Inspetores do Grupo 1	# Defeitos	# Falso-Positivo	# Melhorias
Insp. 1	0	1	0
Insp. 2	3	2	0
Insp. 3	5	3	2
Insp. 4	2	0	0
Insp. 5	5	6	0
Insp. 6	6	0	0
Insp. 7	6	1	0
Insp. 8	1	0	0
Insp. 9	2	0	0
Insp. 10	4	0	0
Insp. 11	4	2	0
Insp. 12	8	2	1
Insp. 13	8	0	3
Insp. 14	2	4	0
Insp. 15	0	0	0
Insp. 16	3	0	0

Tabela 5.10. Frequência de defeitos, falso-positivo e melhorias para participantes do Grupo 2

Inspetores do Grupo 2	# Defeitos	# Falso-Positivo	# Melhorias
Insp. 1	2	0	0
Insp. 2	2	3	0
Insp. 3	3	0	0
Insp. 4	4	1	0
Insp. 5	3	0	0
Insp. 6	4	1	0
Insp. 7	1	3	0
Insp. 8	1	17	0
Insp. 9	5	4	0
Insp. 10	0	0	1
Insp. 11	3	2	0
Insp. 12	3	0	0
Insp. 13	3	0	1
Insp. 14	2	8	0
Insp. 15	3	2	0
Insp. 16	1	7	0
Insp. 17	3	1	0

Questões de melhorias de código também podem ser identificadas durante um processo de inspeção de código. Do experimento realizado, identificou-se que diversas questões de melhorias do código e de possíveis refatorações foram identificadas por inspetores do grupo G1. Tal resultado deve-se, possivelmente, a um melhor entendimento do sistema devido ao fato dos inspetores possuírem artefatos auxiliares.

Evidência 6: o uso da documentação e métricas do sistema auxilia o inspetor na identificação de melhorias no código.

Outro resultado que merece destaque é em relação ao número de falso-positivos identificados pelos inspetores de ambos os grupos. Quando o grupo G1 realizou a inspeção do

código-fonte, a soma do número de falso-positivos encontrado (21) é substancialmente menor do que a soma do número de falso-positivo identificado pelo grupo G2 (49).

Evidência 7: o uso de artefatos auxiliares auxilia o inspetor na diminuição do número de falso-positivos identificados durante o processo de inspeção.

5.11.5 *Precision* e *Recall*

Uma análise geral dos resultados dos dois grupos envolvidos no experimento pode ser obtida a partir das métricas *precision* e *recall*. Se o valor de *precision* for 100%, então significa que todos os defeitos identificados são realmente defeitos, embora ainda possam existir defeitos que não foram identificados. Se o valor de *recall* for de 100%, então todos os defeitos conhecidos foram identificados embora os inspetores possam ter identificado falso-positivos.

Para computar as métricas *precision* e *recall*, dois pesquisadores analisaram os relatórios de discrepâncias gerados pela ferramenta Crista e classificaram os defeitos como defeitos verdadeiros, falso-positivos ou melhorias do código.

O uso das métricas *precision* e *recall* está relacionado ao número de defeitos identificado pelos inspetores nos dois grupos. Inicialmente, foi realizada uma análise das métricas *precision* e *recall* para cada inspetor e posteriormente para todo o grupo.

No caso da avaliação individual as métricas *precision* e *recall* foram definidas como segue:

$$precision = \frac{|D \cap T|}{|T|}$$

$$recall = \frac{|D \cap T|}{|D|}$$

onde:

D = {*d* | *d* Є *defeitos conhecidos*}
T = {*t* | *t* Є *defeitos identificados*}

As Tabelas 5.11 e 5.12 apresentam os valores das métricas *precision* e *recall* para os inspetores dos grupos G1 e G2, respectivamente.

Tabela 5.11. *Precision* e *Recall* para Inspetores do Grupo 1

Inspetores do Grupo 1	*Precision*	*Recall*
Insp. 1	0,00 %	0,00 %
Insp. 2	60,00 %	27,27 %
Insp. 3	62,50 %	45,45 %
Insp. 4	100,00 %	18,18 %
Insp. 5	45,45 %	45,45 %
Insp. 6	100,00 %	54,55 %
Insp. 7	85,71 %	54,55 %
Insp. 8	100,00 %	9,09 %
Insp. 9	100,00 %	18,18 %
Insp. 10	100,00 %	36,36 %
Insp. 11	66,67 %	36,36 %
Insp. 12	80,00 %	72,73 %
Insp. 13	100,00 %	72,73 %
Insp. 14	33,33 %	18,18 %
Insp. 15	0,00 %	0,00 %
Insp. 16	100,00 %	27,27 %

Tabela 5.12. *Precision* e *Recall* para Inspetores do Grupo 2

Inspetores do Grupo 2	*Precision*	*Recall*
Insp. 1	100,00 %	18,18 %
Insp. 2	40,00 %	18,18 %
Insp. 3	100,00 %	27,27 %
Insp. 4	80,00 %	36,36 %
Insp. 5	100,00 %	27,27 %
Insp. 6	80,00 %	36,36 %
Insp. 7	25,00 %	9,09 %
Insp. 8	5,56 %	9,09 %
Insp. 9	55,56 %	45,45 %
Insp. 10	0,00 %	0,00 %
Insp. 11	60,00 %	27,27 %
Insp. 12	100,00 %	27,27 %
Insp. 13	100,00 %	27,27 %
Insp. 14	20,00 %	18,18 %
Insp. 15	60,00 %	27,27 %
Insp. 16	12,50 %	9,09 %
Insp. 17	75,00 %	27,27 %

Para avaliação dos grupos, as métricas *precision* e *recall* foram calculadas a partir da média aritmética das métricas individuais dos inspetores, conforme apresentado na Tabela 5.13.

Tabela 5.13. Média das métricas *Precision* e *Recall* para os grupos G1 e G2

Grupo	*PRECISION*	*RECALL*
G1	70,85 %	33,52 %
G2	59,62 %	22,99 %

Evidência 8: os valores de precision e recall foram consideravelmente melhores quando do uso de artefatos auxiliares durante a identificação de defeitos do processo de inspeção.

5.11.6 Tempo de Inspeção

As Tabelas 5.14 e 5.15 apresentam o tempo total que cada inspetor levou para encerrar a inspeção do Paint. Além disso, é apresentada a medida de eficiência para cada inspetor dos grupos G1 e G2. Já a Tabela 5.16 apresenta os valores médios de tempo para cada um dos grupos.

Tabela 5.14. Eficiência dos inspetores do grupo 1

Inspetores do Grupo 1	Tempo (em minutos)	Total de defeitos identificados	Eficiência (defeitos/minuto)
Insp. 1	107	0	0.0
Insp. 2	63	3	4.8
Insp. 3	144	5	3.5
Insp. 4	117	2	1.7
Insp. 5	160	5	3.1
Insp. 6	68	6	8.8
Insp. 7	100	6	6.0
Insp. 8	94	1	1.1
Insp. 9	89	2	2.2
Insp. 10	82	4	4.9
Insp. 11	149	4	2.7
Insp. 12	79	8	10.1
Insp. 13	72	8	11.1
Insp. 14	93	2	2.2
Insp. 15	92	0	0.0
Insp. 16	70	3	4.3

Tabela 5.15. Eficiência dos inspetores do grupo 2

Inspetores do Grupo 2	Tempo (em minutos)	Total de defeitos identificados	Eficiência (defeitos/minuto)
Insp. 1	99	2	2.0
Insp. 2	47	2	4.3
Insp. 3	107	3	2.8
Insp. 4	103	4	3.9
Insp. 5	59	3	5.1
Insp. 6	85	4	4.7
Insp. 7	94	1	1.1
Insp. 8	178	1	0.6
Insp. 9	100	5	5.0
Insp. 10	87	0	0.0
Insp. 11	89	3	3.4
Insp. 12	96	3	3.1
Insp. 13	103	3	2.9
Insp. 14	63	2	3.2
Insp. 15	101	3	3.0
Insp. 16	106	1	0.9
Insp. 17	129	3	2.3

Tabela 5.16. Medidas de tempo dos grupos participantes

MEDIDAS	GRUPO 1	GRUPO 2
MÉDIA (tempo em minutos)	98.68	96.82
MEDIANA (tempo em minutos)	92	99
MÍNIMO (tempo em minutos)	63	47
MÁXIMO (tempo em minutos)	160	178
MÉDIA (eficiência)	4.2	2.8

Quantitativamente não existe uma grande diferença na média dos tempos de cada grupo. No entanto, se verificarmos a eficiência de cada inspetor e do grupo como um todo, observa-se que o grupo G1 (uso de artefatos auxiliares) foi quase 50% mais eficiente do que o grupo G2.

Evidência 9: inspetores com um maior número de artefatos de software (documentação e métricas) para compreensão do sistema a ser inspecionado foram mais eficientes do que inspetores sem o uso desses recursos.

5.11.7 Análise das decisões tomadas pelos inspetores durante a atividade de inspeção e o impacto nos defeitos identificados

As decisões mais comuns realizadas pelos inspetores nos dois grupos do experimento são apresentadas a seguir:

- Decisões mais realizadas do grupo G1 (em ordem crescente de uso): Classe que representa Interface Gráfica, Mudança devido a herança – Verificando Diagrama de Classes, Retorno a classe anterior, Aleatório, Última Classe a ser analisada, Referência a Classe, Menor Complexidade Ciclomática.
- Decisões mais realizadas do grupo G2 (em ordem crescente de uso): Retorno a classe anterior, Mudança devido a herança – Verificando código-fonte, Classe Pequena, Aleatório, Referência a Classe.

Do conjunto de todas as decisões tomadas pelos inspetores, as decisões mais utilizadas pelos inspetores do grupo G1 (representando mais de 48% das decisões) são: Menor Complexidade Ciclomática, Referência a Classe, Última classe a ser analisada. Em relação aos inspetores do grupo G2, as decisões mais utilizadas (representando mais de 50% das decisões) são: Referência a Classe e Aleatório.

As Tabelas 5.17 (Grupo 1) e 5.18 (Grupo 2) ilustram o total de decisões tomadas por cada inspetor. Ressalta-se que uma mesma decisão pode ser tomada várias vezes durante a inspeção do código-fonte.

Além do total de decisões tomadas por cada inspetor, as Tabelas 5.17 e 5.18 apresentam o número de decisões tomadas que correspondem às decisões mais utilizadas por cada grupo. Ressalta-se que essas decisões correspondem a aproximadamente 50% das decisões tomadas em cada grupo do experimento.

Tabela 5.17. Número de decisões utilizadas, porcentagem média de abstração de código realizada e eficiência dos participantes do grupo 1

Inspetores do Grupo 1	Total de decisões utilizadas	Total de decisões dentro das 3 mais utilizadas	% Decisões	% Defeitos encontrados	% Média da abstração de código realizada	Eficiência
Insp. 1	10	0	0.00 %	0.00 %	83.5 %	0.0
Insp. 2	36	4	11.11 %	27.27 %	42.6 %	4.8
Insp. 3	18	11	61.11 %	45.45 %	0.0 %	3.5
Insp. 4	13	6	46.15 %	18.18 %	0.0 %	1.7
Insp. 5	18	17	94.44 %	45.45 %	81.9 %	3.1
Insp. 6	23	6	26.09 %	54.55 %	0.3 %	8.8
Insp. 7	10	6	60.00 %	54.55 %	6.8 %	6.0
Insp. 8	23	6	26.09 %	9.09 %	0.0 %	1.1
Insp. 9	17	9	52.94 %	18.18 %	1.4 %	2.2
Insp. 10	16	9	56.25 %	36.36 %	5.6 %	4.9
Insp. 11	16	11	68.75 %	36.36 %	0.0 %	2.7
Insp. 12	15	11	73.33 %	72.73 %	76.3 %	10.1
Insp. 13	16	9	56.25 %	72.73 %	15.1 %	11.1
Insp. 14	15	13	86.67 %	18.18 %	96.1 %	2.2
Insp. 15	13	10	76.92 %	0.00 %	38.0 %	0.0
Insp. 16	7	2	28.57 %	27.27 %	7.5 %	4.3

Tabela 5.18. Número de decisões utilizadas, porcentagem média de abstração de código realizada e eficiência dos participantes do grupo 2

Inspetores do Grupo 2	Total de decisões utilizadas	Total de decisões dentro das 3 mais utilizadas	% Decisões	% Defeitos encontrados	% Média da abstração de código realizada	Eficiência
Insp. 1	12	4	33.33 %	18.18 %	4.3 %	2.0
Insp. 2	9	6	66.67 %	18.18 %	0.4 %	4.3
Insp. 3	8	4	50.00 %	27.27 %	27.3 %	2.8
Insp. 4	10	4	40.00 %	36.36 %	1.6 %	3.9
Insp. 5	6	4	66.67 %	27.27 %	33.4 %	5.1
Insp. 6	10	2	20.00 %	36.36 %	56.1 %	4.7
Insp. 7	14	4	28.57 %	9.09 %	44.4 %	1.1
Insp. 8	21	1	4.76 %	9.09 %	8.8 %	0.6
Insp. 9	15	9	60.00 %	45.45 %	0.8 %	5.0
Insp. 10	8	4	50.00 %	0.00 %	28.1 %	0.0
Insp. 11	11	3	27.27 %	27.27 %	16.3 %	3.4
Insp. 12	16	8	50.00 %	27.27 %	56.4 %	3.1
Insp. 13	15	9	60.00 %	27.27 %	4.0 %	2.9
Insp. 14	13	13	100.00 %	18.18 %	4.3 %	3.2
Insp. 15	10	6	60.00 %	27.27 %	0.0 %	3.0
Insp. 16	18	15	83.33 %	9.09 %	0.0 %	0.9
Insp. 17	10	8	80.00 %	27.27 %	21.9 %	2.3

De acordo com a Tabela 5.17, por exemplo, das 18 decisões utilizadas pelo inspetor 3 do grupo G1 durante a inspeção do Paint, 11 decisões estão compreendidas entre as 3 mais utilizadas do grupo G1.

A coluna "% de Decisões" das tabelas 5.17 e 5.18 apresenta a porcentagem do número de decisões utilizadas pelos inspetores que estão contempladas nas três decisões mais utilizadas por cada inspetor de cada grupo. Observa-se na Tabela 5.17, por exemplo, que o inspetor 3 do grupo G1 utilizou em suas decisões aproximadamente 61% das decisões mais utilizadas pelo grupo G1.

Observando com mais detalhes a coluna "% de Decisões" da Tabela 5.17, verifica-se que quase todos os inspetores utilizaram decisões compreendidas entre as 50% mais usadas decisões de todo o grupo. Essa característica fornece indícios de que as decisões mais comuns podem ser parte de uma estratégia efetiva e eficiente de técnica de inspeção de código-fonte.

Evidência 10: o uso de informações de outros artefatos do software e de métricas de código-fonte pode ser usada como fonte de informação para guiar o inspetor durante um processo de inspeção de código-fonte efetivo e eficiente.

As Tabelas 5.17 e 5.18 também possuem duas colunas interessantes para análise, a saber: "% de defeitos encontrados" e "% de abstração realizada". A coluna "% de defeitos encontrados" apresenta a porcentagem de defeitos encontrados por cada inspetor em relação ao total de defeitos conhecidos. A coluna "% de abstração realizada" apresenta a porcentagem de código que foi abstraído quando da realização de inspeção do *Paint* usando a ferramenta Crista. Por exemplo, o inspetor 2 do grupo G1 (Tabela 4.16) teve 27%, aproximadamente, de defeitos identificados realizando uma abstração de 42%, aproximadamente, do código-fonte do *Paint*.

A análise em conjunta dessas duas últimas colunas das Tabelas 5.17 e 5.18 pode ser utilizada para verificar se, entre outras coisas, a abstração do código tem influência sobre o número de defeitos identificados. Por exemplo, a média de abstração do grupo G1 foi de 28% enquanto que no grupo G2 foi de 18%. No entanto, a média de defeitos encontrados para o grupo G1 foi de 33,5% enquanto que para o grupo G2 foi de 22,9%. De forma simples, esses resultados podem indicar a influência da abstração na identificação de defeitos. Além disso, acreditamos que a abstração tenha sido maior para os participantes do grupo G1 devido a existência de documentação para o entendimento dos requisitos e projeto do Paint.

Evidência 11: a cobertura de abstração de código é mais bem alcançada quando há existência de documentação para auxiliar o inspetor na compreensão do código.

Evidência 12: a realização de abstração de código pode auxiliar o inspetor no desempenho da identificação de defeitos em código-fonte.

Para verificar se o uso da decisão "Aleatória" teria algum padrão entre os inspetores que a utilizaram foi feita uma análise para associar o uso da decisão aleatória e da classe a ser inspecionada baseada nessa decisão. A Tabela 5.19 apresenta os resultados dessa análise. Observa-se que grande parte dos inspetores do grupo G2 quando usaram como decisão "Aleatória" optou por inspecionar a classe Actions. Essa decisão pode ter sido influenciada pelo fato da classe Actions ser a primeira classe apresentada na lista de classes da ferramenta Crista.

Tabela 5.19. Frequência de uso de classes quando a decisão "Aleatória" é utilizada

Classe	Grupo 1	Grupo 2
Actions	2	15
PaintWindow	5	6
PencilPaint	3	5
PaintObject	3	3
PaintObjectConstructor	3	6
PaintObjectConstructorListener	3	4
EraserPaint	3	5
PaintCanvas	0	2

Evidência 13: a ordem de apresentação das classes pode influenciar na escolha da classe a ser inspecionada.

Por outro lado, quando analisado os participantes do grupo G1 uma interessante observação está relacionada com a decisão "Aleatória". Em três inspetores distintos do grupo G1, sempre que uma decisão "Aleatória" foi tomada, três classes específicas são sempre analisadas em conjunto, a saber: `PaintObject`, `PaintObjectConstructor` e `PaintObjectConstructorListener`. Esse resultado pode indicar, por exemplo, que o inspetor utilizou a proximidade do nome da classe para tomar a decisão de qual classe ser inspecionada.

Evidência 14: a similaridade dos nomes das classes pode auxiliar o inspetor de código-fonte na tomada de decisão de qual classe inspecionar.

5.11.8 Análise dos defeitos identificados por cada inspetor

As tabelas 5.20 e 5.21 apresentam quais defeitos foram identificados pelos participantes dos dois grupos.

Tabela 5.20. Defeitos identificados por cada participante do grupo 1

INSPETORES	D1	D2	D3	D4	D5	D6	D7	D8	D9	D10	D11
Insp. 1											
Insp. 2					X		X		X		
Insp. 3	X		X	X			X				X
Insp. 4					X		X				
Insp. 5			X		X	X	X				
Insp. 6	X				X	X	X		X		
Insp. 7			X	X	X	X	X				X
Insp. 8									X		
Insp. 9					X		X				
Insp. 10	X	X	X				X				
Insp. 11			X		X	X	X				
Insp. 12	X	X	X		X	X	X		X		X
Insp. 13	X		X	X	X	X	X		X	X	
Insp. 14							X			X	
Insp. 15											
Insp. 16	X	X					X				
TOTAL	**6**	**3**	**7**	**3**	**9**	**6**	**13**	**0**	**5**	**2**	**3**

Tabela 5.21. Defeitos identificados por cada participante do grupo 2

INSPETORES	D1	D2	D3	D4	D5	D6	D7	D8	D9	D10	D11
Insp. 1					X	X					
Insp. 2					X		X				
Insp. 3					X		X				X
Insp. 4	X				X	X					X
Insp. 5	X	X			X						
Insp. 6	X		X		X						X
Insp. 7					X						
Insp. 8					X						
Insp. 9	X	X	X		X						X
Insp. 10											
Insp. 11			X				X				X
Insp. 12	X			X							X
Insp. 13		X	X				X				
Insp. 14					X	X					
Insp. 15	X	X					X				
Insp. 16					X						
Insp. 17					X		X				X
TOTAL	**6**	**4**	**4**	**1**	**12**	**3**	**6**	**0**	**0**	**0**	**7**

Algumas observações são possíveis de serem realizadas a partir das Tabelas 5.20 e 5.21:

- O único defeito não identificado pelos inspetores dos dois grupos foi o defeito 8. Os defeitos 7, 10 e 11 possuem as mesmas características do defeito 8 e foram identificados pelos inspetores.
- Inspetores do grupo 2 não identificaram dois defeitos além do defeito 8: 9 e 10. Isso significa que do total de 11 defeitos, aproximadamente 27% dos defeitos não foram identificados por nenhum dos 17 participantes do grupo 2. Já os

participantes do grupo 1 tiveram, em média, um desempenho de aproximadamente 21% na identificação dos defeitos 9 e 10.

5.11.9 Análise dos defeitos de acordo com seus descritores

A seguir são apresentados descritores dos defeitos de acordo com o trabalho de Dunsmore *et al.* (2003):

- **Objeto incorreto utilizado**: enviar mensagem a um objeto incorreto
- **Método incorreto chamado**: enviar mensagem incorreta
- **Parâmetro incorreto em uma chamada de método**: parâmetros incorretos na chamada do método
- **Algoritmo/computação**: erro no algoritmo (exemplo: sequência de passos faltando ou ordem incorreta)
- **Erro de fluxo de dados**: variável incorreta/faltando ou valor incorreto
- **Erro de especificação**: inconsistência com a especificação
- **Omissão**: código faltando
- **Comissão**: código incorreto ou supérfluo
- **Localidade**: área do código a ser verificada para identificar o defeito. Valores possíveis:

  (M)étodo: informação exigida para identificar defeito está presente em nível de método.

  (C)lasse: informação exigida para identificar defeito está presente em nível de classe.

  (S)istema: informação exigida para identificar defeito está presente em nível de sistema.
- **Tamanho do método**: tamanho do método onde está presente o defeito

  S = 0-4 linhas de código

  M = 5-10 linhas de código

  L = 11 ou mais linhas de código

Na Tabela 5.22 os 11 defeitos presentes no Paint são classificados de acordo com os descritores de Dunsmore *et al.* (2003).

Tabela 5.22-Classificação dos Defeitos do Paint de acordo com os descritores de Dunsmore *et al.* (2003)

INSPETORES	D1	D2	D3	D4	D5	D6	D7	D8	D9	D10	D11
Objeto incorreto						X					
Método incorreto									X		
Parâmetro incorreto		X									
Computação							X	X		X	X
Fluxo de dados			X								
Erro de Especificação				X							
Omissão	X			X							
Comissão					X						
Localidade	C	S	S	S	C	C	M	M	M	M	M
Tamanho	L	L	S	---	L	L	S	L	L	M	M

Algumas análises individualizadas dos defeitos são realizadas bem como os resultados obtidos pelos dois grupos de participantes.

Em relação ao tipo de defeito "Objeto incorreto usado" (Defeito 6), os dois grupos de participantes tiveram o mesmo número de defeitos encontrados. Já com relação ao tipo de defeito "Método incorreto chamado" (Defeito 9), o grupo G1 teve uma porcentagem de 31,25% de participantes que encontraram o defeito 9 enquanto que o grupo G2 não teve nenhum participante que o encontrou.

O tipo de defeito "Parâmetro incorreto" (Defeito 2) foi mais encontrado pelos participantes do grupo G2 (23,52% contra 18,75% do grupo G1).

Como o software Paint possuía quatro defeitos do tipo "Algoritmo/Computação", a análise será realizada individualmente para cada defeito, conforme segue:

- O defeito 7 teve uma maior identificação pelos participantes do grupo G1 (81,25%) contra 35,29% do grupo G2. Tal diferença pode ser considerada uma vantagem pelo uso dos recursos do grupo G1, principalmente pelo fato de que o defeito encontra-se em um método considerado de tamanho pequeno para análise. Tal facilidade de identificação do grupo G1 poderia ser atribuída ao fato da abstração dos trechos de código pela aplicação da técnica Stepwise Abstraction.
- O defeito 8 não foi identificado por nenhum participante dos dois grupos. Ele possui a localidade restrita ao método e é considerado de tamanho grande em termos de linhas de código.
- O defeito 10 não foi encontrado com muita frequência pelos participantes do grupo G1 (cerca de 10% dos participantes o encontraram). No entanto, nenhum participante do grupo G2 identificou tal defeito.
- O defeito 11 foi mais identificado pelos participantes do grupo G2 (41,17%) contra 18,75% da frequência identificada pelos participantes do grupo G1. É interessante observar que os defeitos 10 e 11 encontram-se no mesmo método e que o número de linhas do método é de 13 linhas.

O tipo de defeito "Fluxo de dados" (Defeito 3) foi encontrado por quase metade dos participantes do grupo G1 (43,75%) enquanto que no grupo 2 apenas 23,25% dos participantes encontraram esse defeito. É interessante observar que o defeito 3 trata-se de um defeito associado ao sistema, na qual a sua identificação envolve múltiplas classes do software e portanto está associada a uma das principais dificuldades apontadas por Dunsmore (2003) na inspeção de código orientado a objetos: *delocalization* do código.

O tipo de defeito "Erro de especificação" (Defeito 4) foi encontrado por 18,75% dos participantes do grupo G1 enquanto que os participantes do grupo G2 tiveram apenas 5,88% de participantes o identificando. O defeito 4 é considerado um defeito do sistema e a diferença no percentual encontrado por cada inspetor pode estar associado ao fato de que apenas o grupo G1 recebeu a especificação de requisitos do software Paint e portanto estava menos complexa a atividade de identificar a falta de uma classe para essa funcionalidade. Dessa forma, o defeito 4 também está associado ao tipo de defeito "Omissão".

Além do defeito 4, o tipo de defeito "Omissão" também está associado ao defeito 1, o qual possui, em números relativos, praticamente o mesmo número de participantes que o identificaram (37,5% para o Grupo G1 e 35,29% para o Grupo G2).

Para o tipo "Comissão" (Defeito 5), o número de participantes do grupo 2 que encontraram o defeito foi maior (70,58% para o grupo G2 e 56,25% para o grupo G1). Vale destacar que o Defeito 5 é um defeito cosmético inserido em um método com 32 linhas.

A Figura 5.1 apresenta o percentual médio de participantes que identificaram os defeitos de acordo com as localidades do defeito (Método, Classe, Sistema).

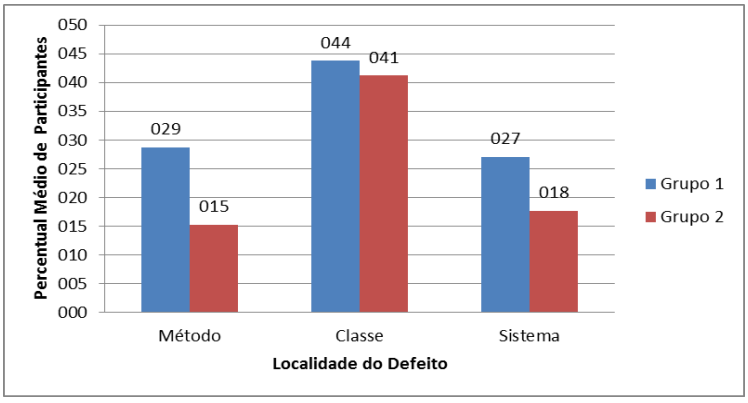


Figura 5.1-Percentual médio de participantes que identificaram defeitos de acordo com suas localidades

Observa-se pela Figura 5.1 que o grupo G1, em média, teve um melhor desempenho na identificação dos defeitos de acordo com a localidade. A maior diferença está nos defeitos residentes no próprio método, seguido pelo sistema e pela classe. Ressalta-se que quando o

defeito está classificado como Sistema, a característica de delocalização ressaltada por Dunsmore (2003) está presente e os resultados obtidos apresentam um melhor desempenho ao grupo que utilizou fontes de informações extras durante a inspeção de código.

Evidência 15: o uso de artefatos auxiliares durante a atividade de inspeção pode auxiliar na identificação de defeitos com característica de delocalização.

5.12 Ameaças a Validade

As ameaças à validade podem ser classificadas em interna, externa, de construção e de conclusão (WOHLIN et al., 2000).

A validade interna refere-se a fatores não controlados que podem influenciar nas variáveis dependentes, destacando:

- Participantes: o uso de estudantes como participantes poderia influenciar nas variáveis dependentes. No entanto, os participantes do experimento foram estudantes de pós-graduação com uma média de experiência profissional de 8 anos e meio, caracterizando certa maturidade.
- Comunicação: a comunicação entre participantes do próprio grupo e entre grupos não era permitida.
- Fadiga: a fadiga por parte dos inspetores foi minimizada, pois a atividade de inspeção foi planejada para consumir, no máximo, 1 hora e meia de aplicação.
- Treinamento: uma possível ameaça estava relacionada à falta de familiaridade com o uso das técnicas. Essa ameaça foi minimizada com os treinamentos e os feedbacks fornecidos antes da execução do experimento.
- Métricas de Software: os participantes que utilizaram as métricas de software tinham prévio conhecimento apenas das métricas acoplamento, coesão e complexidade ciclomática. No entanto, eles receberam treinamento para todas as métricas disponíveis na técnica KIRT.

A validade externa refere-se à capacidade de generalizar os resultados do experimento, destacando:

- Participantes: o uso de estudantes como participantes não permite a generalização dos resultados obtidos no experimento para outros tipos de perfis de inspetores. No

entanto, essa ameaça foi minimizada, como já explicado anteriormente, com a participação de estudantes de pós-graduação com experiência profissional.

- Sistema: o sistema *Paint* utilizado no estudo experimental, embora não desenvolvido profissionalmente, possui todas as características de um sistema orientado a objetos e já foi utilizado em outros estudos experimentais relatados na literatura.

A validade de conclusão está relacionada com as questões que afetam a interpretação de uma conclusão correta, destacando:

- Teste estatístico: mesmo com um número reduzido de participantes em cada grupo, foi possível aplicar o teste não-paramétrico (Mann-Whitnney) para avaliar a efetividade e eficiência na identificação de defeitos e os números de melhorias identificadas.

A validade de construção está relacionada ao grau com que as métricas utilizadas para avaliar os resultados do experimento (variáveis dependentes) de fato representam as medidas necessárias para caracterizar o objeto de estudo (as técnicas de leitura definidas), destacando:

- Métricas (efetividade, eficiência e número de melhorias): essas métricas têm sido utilizadas por vários pesquisadores em outros estudos experimentais com objetivos similares a este.

5.13 Considerações finais

Este capítulo apresentou um estudo exploratório realizado para identificar quais informações são utilizadas por inspetores durante um processo de inspeção de código-fonte. Os resultados obtidos por esse estudo exploratório foram resumidos em 15 evidências. As principais evidências estão relacionadas ao uso de artefatos auxiliares (documento de requisitos, diagrama UML e métricas de código-fonte) para guiar inspetores na tomada de decisão. Padrões comportamentais foram observados para inspetores do grupo G1. Além disso, inspetores do grupo G1 também tiveram um melhor desempenho em relação aos inspetores do grupo G2 em termos de efetividade e eficiência na identificação de defeitos.

Outro resultado interessante é o fato que o grupo G1 identificou um número menor de falso positivos comparado ao grupo G2. Essa evidência foi baseada nas métricas de *precision* e *recall*.

As evidências relatadas serviram de base para a definição da técnica de leitura KIRT implementadas na ferramenta CRISTA e que são apresentadas no Capítulo 6 deste trabalho.

Capítulo 6

CICLO 4 - TÉCNICA DE LEITURA KIRT: DEFINIÇÃO E APOIO COMPUTACIONAL

Este capítulo apresenta a técnica de leitura proposta neste trabalho e implementada na ferramenta CRISTA. Salienta-se que o Ciclo 4 é dividido em dois capítulos de modo a registrar a definição da técnica de leitura e o apoio computacional (capítulo em questão) e os estudos experimentais realizados (Capítulo 7).

6.1 Considerações iniciais

Durante a revisão da literatura sobre técnicas de inspeção de código-fonte foi identificado que as técnicas de leitura *Ad-Hoc* e *Checklist* são as mais populares técnicas de leitura usadas para a identificação de defeitos. No entanto, também é encontrado na literatura que essas técnicas de leitura possuem algumas deficiências. Por exemplo, no caso da técnica *Ad-Hoc*, ela não oferece nenhum guia ao inspetor durante a atividade de identificação de defeitos e é altamente dependente do conhecimento e experiência do inspetor. Já a técnica *Checklist* oferece um simples guia de inspeção, por meio de questões que o inspetor precisa responder durante a atividade de identificação de defeitos. O *Checklist* também possui forte dependência da experiência do inspetor e da qualidade das questões utilizadas.

Ressalta-se que em ambas as técnicas (*Ad-Hoc* e *Checklist*), não existem instruções de leitura concretas e essa característica pode explicar porque fatores como experiência tem uma influência significativa no número de defeitos identificados (Laitenberger e DeBaud, 1997).

Variações na construção dos *Checklists* podem minimizar os problemas dessa técnica, como por exemplo, especificando-se onde procurar um defeito e como detecta-lo, conforme proposto por (CHERNAK, 1996).

Uma das técnicas que tenta minimizar os problemas identificados pelas técnicas *Ad-Hoc* e *Checklist* é a técnica *Perspective-Based Reading* (PBR). A base dessa técnica está no oferecimento de instruções para um inspetor na forma de cenários operacionais. Um cenário operacional é definido como uma diretriz algorítmica de como inspetores deve proceder enquanto lêem um documento de software.

Dado o contexto apresentado anteriormente e as observações descritas no Capítulo 5, decorrentes da execução do Ciclo 3, novas técnicas de leitura de código-fonte foram definidas com base no uso de informações auxiliares, como métricas de software, e visualização.

Nesse sentido, este capítulo tem como objetivo apresentar as técnicas definidas neste trabalho bem como os recursos computacionais implementados na ferramenta CRISTA para automatização dessas técnicas.

Este capítulo está organizado da seguinte forma: na seção 6.2 são mostradas as técnicas de leitura definidas neste trabalho, na seção 6.3 é mostrada a evolução da ferramenta com as novas funcionalidades implementadas para apoiar a aplicação das técnicas de leitura, na seção 6.4 é mostrado um exemplo de aplicação de uma técnica de leitura e na seção 5.5 são apresentadas as considerações finais do capítulo.

6.2 Técnica de Leitura para Inspeção de Código-Fonte baseadas em Informações-Chave do Software e Visualização

Dada a efetividade dos cenários operacionais da técnica de leitura PBR e o amplo uso da técnica *Checklist*, as técnicas de leitura definidas neste trabalho foram inspiradas nas características dessas duas técnicas. Além disso, os indícios identificados no Ciclo 3 serviram de base para a definição de como informações-chave deveriam ser utilizadas nas técnicas de leitura. Somam-se a isso, o uso de visualização já presente na ferramenta CRISTA e as novas funcionalidades adicionadas à ferramenta para dar suporte às técnicas definidas.

O conceito de cenário operacional está baseado em três elementos: uma taxonomia de defeitos, um modelo subjacente e um conjunto de procedimentos que orienta o inspetor na leitura do artefato sendo avaliado. A taxonomia de defeitos deve retratar os defeitos característicos daquele tipo de artefato e que devem ser procurados pelo inspetor durante a inspeção. O modelo subjacente corresponde a um modelo auxiliar que deve ser construído pelo inspetor como base para verificar se o artefato sob inspeção possui os requisitos de

qualidade necessários para construí-lo. O conjunto de procedimentos corresponde a um conjunto de passos e/ou atividades que o inspetor deve realizar para que, ao construir o modelo subjacente, os defeitos pertencentes à taxonomia de defeitos possam ser investigados no artefato sob inspeção.

Com base nessa definição de cenário operacional, utilizado na técnica de leitura PBR (LAITENBERGER; DEBAUD, 1997), a Figura 6.1 representa o cenário operacional para geração das técnicas de leitura baseadas em informações-chave definidas neste trabalho.

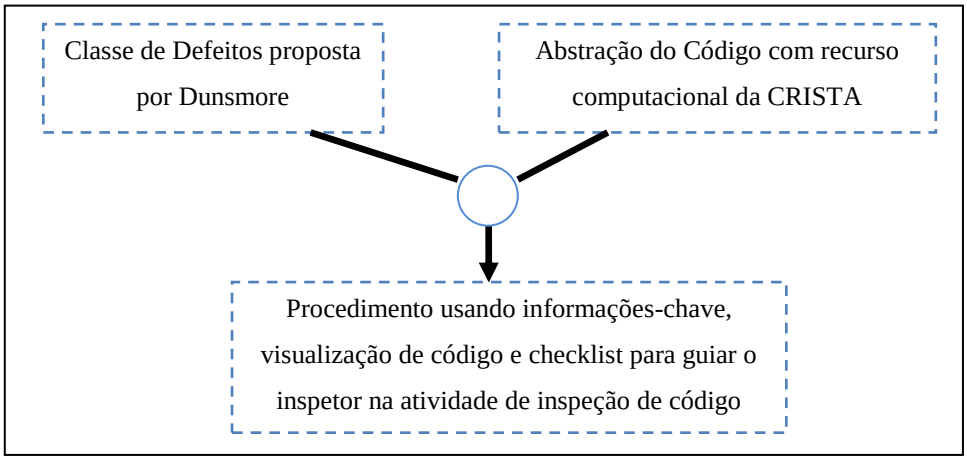


Figura 6.1. Modelo para Geração das Técnicas de Leitura

De acordo com a Figura 6.1, as técnicas de leitura definidas neste trabalho devem ser instanciadas com o uso das seguintes informações do cenário operacional:

- Classe de Defeitos: contempla a taxonomia utilizada para a classificação dos defeitos identificados durante a atividade de inspeção de código. Por se tratar de código orientado a objetos, optou-se por utilizar a classe de defeitos baseada em Dunsmore *et al.* (2003b), que é composta dos seguintes tipos:
  - **Objeto incorreto utilizado**: enviar mensagem a um objeto incorreto
  - **Método incorreto chamado**: enviar mensagem incorreta
  - **Parâmetro incorreto em uma chamada de método**: parâmetros incorretos na chamada do método
  - **Algoritmo/computação**: erro no algoritmo (exemplo: sequência de passos faltando ou ordem incorreta)
  - **Defeito de fluxo de dados**: variável incorreta/faltando ou valor incorreto
  - **Defeito de especificação**: inconsistência com a especificação
  - **Omissão**: código faltando
  - **Comissão**: código incorreto ou supérfluo

- **Localidade**: área do código a ser verificada para identificar o defeito. Valores possíveis:
  - (M)étodo: informação exigida para identificar defeito está presente em nível de método.
  - (C)lasse: informação exigida para identificar defeito está presente em nível de classe.
  - (S)istema: informação exigida para identificar defeito está presente em nível de sistema.

- Modelo subjacente: o modelo subjacente deve servir como base para auxiliar o inspetor durante a atividade de inspeção. Para a aplicação das técnicas de leitura aqui definidas utiliza-se como construção do modelo subjacente, a construção da abstração do código-fonte, utilizando a ferramenta CRISTA por meio da aplicação da *Stepwise Abstraction* (opcionalmente) e da metáfora visual. Ressalta-se que pelo fato da técnica KIRT utilizar metáforas visuais implementadas na ferramenta CRISTA, é necessária sua utilização para aplicação da técnica.
- Procedimento: a partir da definição da taxonomia de defeitos e do modelo subjacente, as técnicas de leitura aqui definidas são constituídas de um procedimento algorítmico levando em consideração as informações-chave (documento de requisitos, diagrama de classes e métricas de software), a visualização e um *checklist* desenvolvido com base em outros propostos na literatura. Tal procedimento deve guiar o inspetor durante a compreensão do código e identificação dos defeitos. Assim, com base nas métricas selecionadas e apresentadas no Capítulo 5, estabeleceu-se um procedimento único que faz uso do documento de requisitos, do diagrama de classe e da métrica selecionada. Dessa forma, como foram selecionadas sete métricas, a técnica KIRT pode ser instanciada a partir das sete métricas utilizadas.

Um exemplo de técnica de leitura KIRT baseada na métrica Acoplamento (CBO) é apresentado a seguir:

1. Use a ferramenta CRISTA para gerar a ordem de inspeção das classes usando a métrica *Acoplamento (CBO)*.
2. Para cada classe estabelecida na ordem gerada pela métrica *Acoplamento (CBO)* faça:
   a. Para cada método destacado na metáfora visual faça:

i. Abstraia a funcionalidade do método usando as informações-chave documento de requisitos e diagrama de classes. Durante a abstração registre, usando a ferramenta CRISTA, quaisquer possíveis discrepâncias ou melhorias que possam ser realizadas no código inspecionado.

b. Para cada seção do *checklist*, use a metáfora visual para destacar os blocos de código que devem ser verificados por meio de suas questões.

i. Para cada bloco de código destacado, registre na ferramenta CRISTA, possíveis discrepâncias ou melhorias que possam ser realizadas no código inspecionado.

Ressalta-se que os procedimentos realizados são os mesmos variando apenas a métrica utilizada. Ou seja, apenas a ordem de inspeção das classes pode sofrer alteração. Além disso, em qualquer momento o inspetor tem a liberdade de consultar outras classes e fazer uso de outras informações-chave (documento de requisitos e diagrama de classe).

Um trecho do Checklist utilizado no procedimento das técnicas de leitura é apresentado na Figura 6.2. O *checklist* integral está apresentado no Apêndice A.

LaPES Checklist

Checklist de Inspeção de Código

Projeto:			
Inspetor:		Data :	

DEFEITO				
#	I - Variável	Sim	Não	N/A
1.	Os nomes de variáveis e de constantes são usados de acordo com as convenções de nome?	☐	☐	☐
2.	Toda variável está corretamente tipada?	☐	☐	☐
3.	Toda variável está apropriadamente inicializada?	☐	☐	☐
4.	Existem variáveis que deveriam ser constantes?	☐	☐	☐
5.	Existem atributos que deveriam ser variáveis locais?	☐	☐	☐
6.	Todos atributos possuem modificadores de acesso apropriados (private, protected, public)?	☐	☐	☐
7.	Existem atributos estáticos que deveriam ser não-estáticos ou vice-versa?	☐	☐	☐
8.	Há variáveis que possuem nomes similares que se confundem?	☐	☐	☐
9.	Todas variáveis estão apropriadamente definidas com nomes significantes, consistentes e claros?	☐	☐	☐
10.	Existem variáveis ou atributos redundantes ou não utilizados?	☐	☐	☐

Figura 6.2. Trecho do *Checklist* utilizado pelas técnicas de leitura

Dada a árvore de família de técnicas de leitura apresentada no Capítulo 1, as técnicas de leitura aqui propostas se inserem naquela árvore conforme a Figura 6.3.

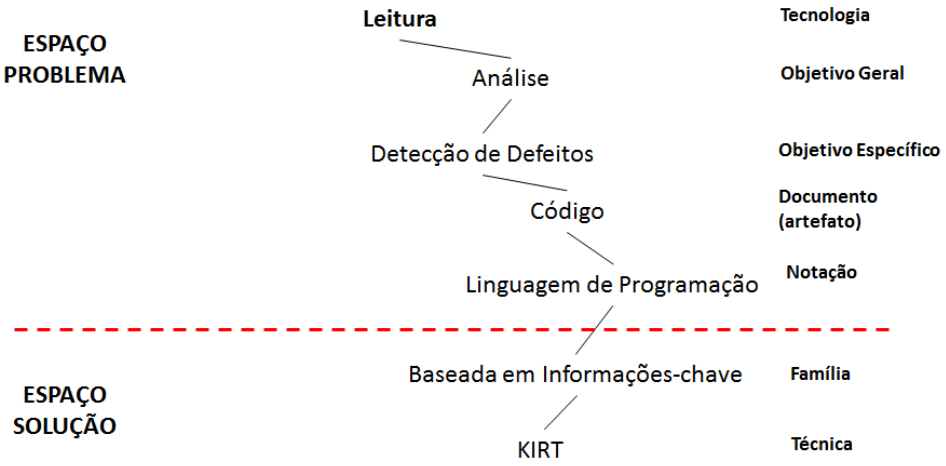


Figura 6.3. Classificação da Técnica de Leitura baseada em Informações-chave de acordo com árvore de Família de Técnicas de Leitura

No espaço de solução, a técnica de leitura aqui definidas é classificada como uma família *baseada em informações-chave*, sendo estas o documento de requisitos, o diagrama de classe e métricas do código. Dessa forma, a técnica é denominada de KIRT e possui sete instanciações distintas, uma para cada métrica considerada.

Para aplicação da técnica KIRT foi necessário o desenvolvimento de novas funcionalidades na ferramenta CRISTA, as quais estão relatadas na subseção seguinte.

6.3 Evolução da ferramenta CRISTA para suporte à técnica KIRT

Com o objetivo de apoiar a aplicação da técnica KIRT, novas funcionalidades foram implementadas na ferramenta CRISTA e são brevemente descritas a seguir:

- Suporte à visualização de um projeto completo em Java.

Como mencionado anteriormente, a CRISTA não possuía recursos para abertura de todas as classes de um projeto escrito na linguagem Java. Dessa forma, a inspeção de todo o sistema era dificultada, pois implicaria na criação de um projeto de inspeção para cada classe existente. Isso exigia do inspetor a abertura de várias instâncias da CRISTA (uma para cada classe) ou, alternativamente, que o inspetor fechasse o projeto da classe atual e abrisse um projeto para outra classe.

Para solucionar esse problema foi implementada a funcionalidade de criação de inspeção de um projeto para todo o sistema. Assim, cada classe do sistema passa a

corresponder a uma aba que apresenta a metáfora visual *treemap* equivalente ao seu código. A Figura 6.4 ilustra essa funcionalidade.

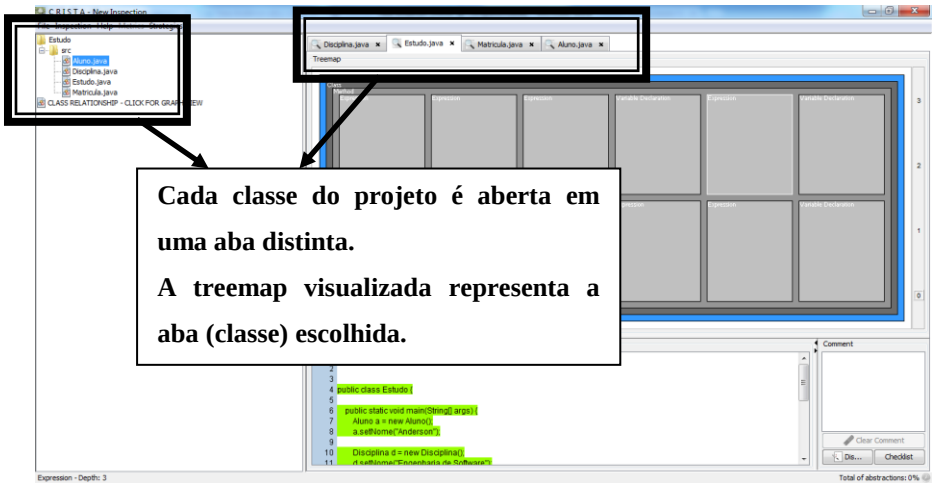


Figura 6.4. Suporte à visualização de todo projeto

- Visualização dos relacionamentos existentes entre as classes por meio de um grafo.

Durante a inspeção de código é importante conhecer os relacionamentos existentes entre as classes do sistema o que não era possível na versão original da CRISTA. Para solucionar essa restrição foi implementada uma funcionalidade que permite gerar uma metáfora visual de relacionamento entre as classes do sistema por meio de um grafo. A tela para visualização do grafo de relacionamento entre as classes é apresentada na Figura 6.5.

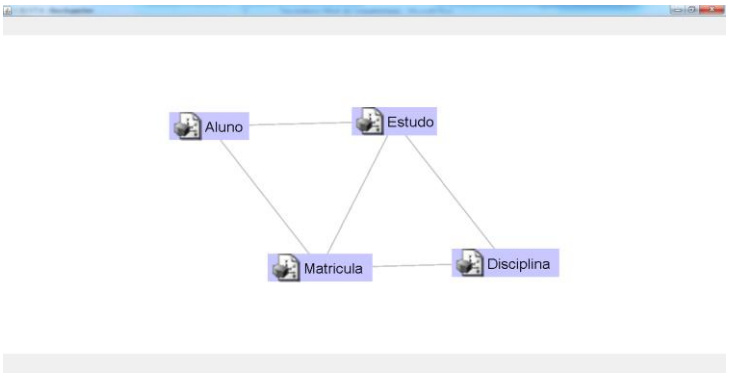


Figura 6.5. Exemplo da metáfora visual de relacionamento entre classes por meio de um grafo

No momento da inspeção, o ideal seria que o inspetor utilizasse dois monitores de modo que o grafo fosse visualizado em um monitor e a *treemap* da classe selecionada em outro. Dessa forma, o inspetor teria uma visão geral de todo o sistema por meio do grafo ao mesmo tempo em que está inspecionando uma classe específica.

- Cálculo e uso das métricas de software orientado a objetos.

Eventualmente, para o inspetor realizar a inspeção de código ele pode ter interesse em verificar os valores das métricas de cada classe existente no sistema, além do relacionamento entre elas. Para visualizar o relacionamento entre as classes, foi implementada a funcionalidade apresentada anteriormente. Para fornecer o cálculo das métricas de software orientado a objeto foi integrada à CRISTA a ferramenta externa chamada *ckjm* (SPINELLIS, 2005).

Para que o inspetor tivesse acesso às métricas calculadas para cada classe, optou-se por integrar os resultados obtidos com a ferramenta *ckjm* ao grafo de relacionamento entre as classes. Dessa forma, o inspetor tem a possibilidade de clicar no nó representativo da classe desejada e ver as métricas computadas para ela bem como seu relacionamento com as outras classes. A Figura 6.6 apresenta as métricas obtidas quando o inspetor clica no nó da classe denominado *Matrícula*. Observe que a janela aberta apresenta as associações da classe *Matrícula* e os valores das sete métricas calculadas.

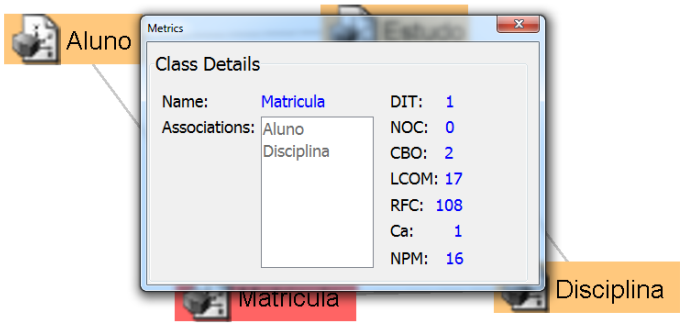
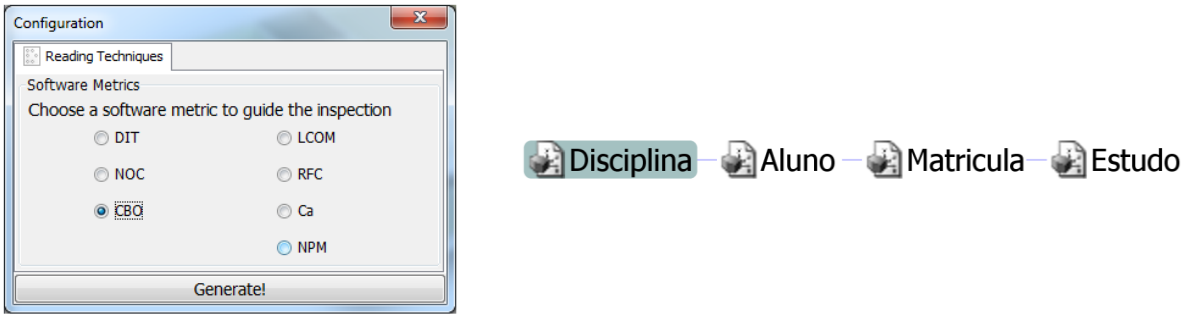


Figura 6.6. Métricas computadas para as classes

- Escolha da técnica de leitura da família KIRT a ser aplicada.

Considerando que a técnica KIRT pode ser instanciada pelas métricas de software, foi implementada uma funcionalidade que permite ao inspetor a escolha da métrica a ser utilizada para instanciar a técnica KIRT, como mostrado na Figura 6.7(a). Uma vez escolhida a métrica desejada, a CRISTA gera a visualização da ordem das classes a serem inspecionadas, conforme apresentado na Figura 6.7(b).



(a) (b)

Figura 6.7. (a) Tela para escolha da métrica que estabelece a técnica de leitura. (b) Ordem de inspeção das classes segundo a métrica escolhida

Observe na Figura 6.7(a), que o inspetor escolheu a métrica CBO para o sistema a ser inspecionado. Uma vez feito a escolha, imediatamente a CRISTA gera a ordem em que as classes devem ser inspecionadas, conforme Figura 6.7(b). Ressalta-se que na Figura 6.7(b) a funcionalidade apresentada anteriormente também está habilitada, ou seja, ao clicar em uma classe as informações mencionadas na Figura 6.6 também são apresentadas.

- Uso da metáfora visual para identificar trechos de código que devem ser abstraídos e/ou inspecionados a partir do *checklist*.

Para a aplicação dos passos 2a e 2b da KIRT é necessário o uso de destaques na metáfora visual utilizada na CRISTA. Ao selecionar uma classe no passo 2, o passo 2a requer que o inspetor abstraia a funcionalidade de cada método. Assim, os métodos ficam destacados na metáfora visual *treemap*, conforme apresentado na Figura 6.8.

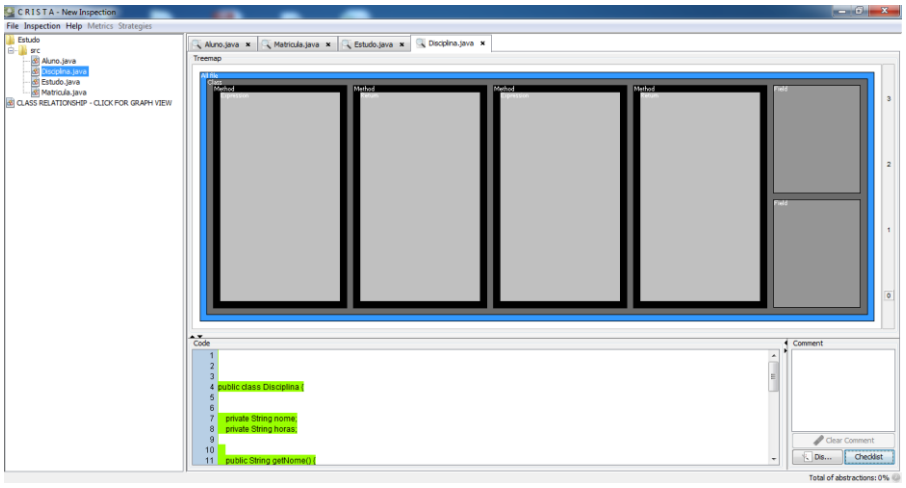


Figura 6.8. Destaque (cor preta) dos blocos de código referentes aos métodos

Considerando que no passo 2b o inspetor deve aplicar o *checklist* e ele está dividido em seções específicas (*Variáveis, Métodos, Construtores, Computação, Condição, Repetição*), uma vez escolhida qual seção a ser inspecionada, a CRISTA destaca o bloco correspondente. Observe que no exemplo da Figura 6.9, o inspetor escolheu inspecionar blocos "*Variáveis*", uma vez que são esses os blocos destacados na metáfora visual.

A implementação dessas funcionalidades tem como objetivo auxiliar o inspetor na aplicação da técnica KIRT e, possivelmente contribuir para a melhoria da qualidade do software.

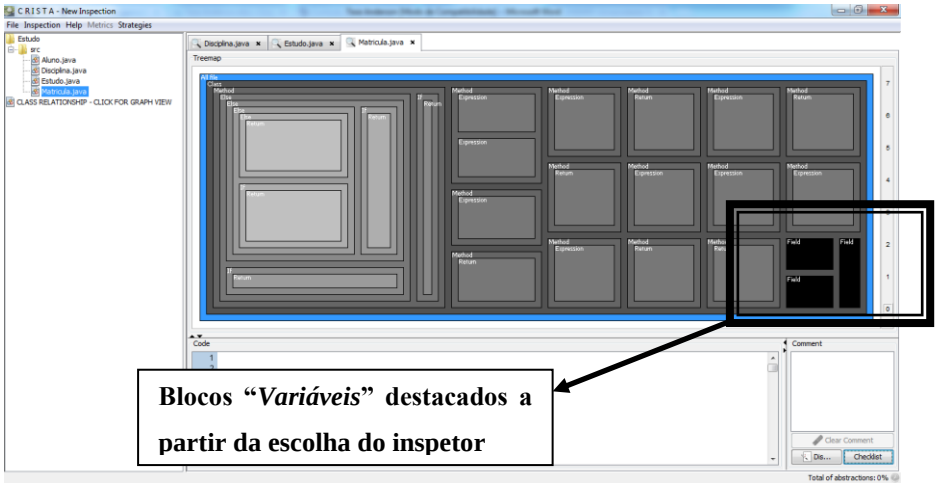


Figura 6.9. Destaque dos blocos de código de acordo com a seleção do bloco "*Variáveis*"

- Registro de Melhorias no Código.

Ao aplicar a KIRT observou-se que melhorias no código também eram identificadas durante a inspeção do código. A versão original da CRISTA só permitia o registro de discrepâncias, não diferenciando entre possíveis defeitos e melhorias. Dessa forma, programou-se uma funcionalidade na qual o inspetor pode escolher, no momento do registro da discrepância, se ele considera essa discrepância como uma melhoria no código. A Figura 6.10 apresenta a tela de registro de discrepância com a opção de classificá-la como melhoria.

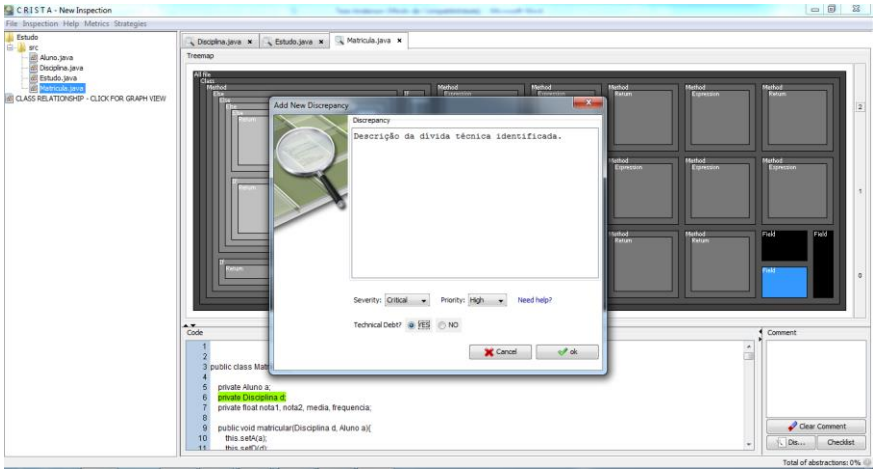


Figura 6.10. Registro de uma melhoria na CRISTA

6.4 Exemplo de aplicação da KIRT

Para efeito de exemplificação da KIRT será utilizado um sistema de matrículas de estudantes em disciplinas e acompanhamento do desempenho nas avaliações. Esse sistema é composto de quatro classes representadas no diagrama de classes da Figura 6.11.

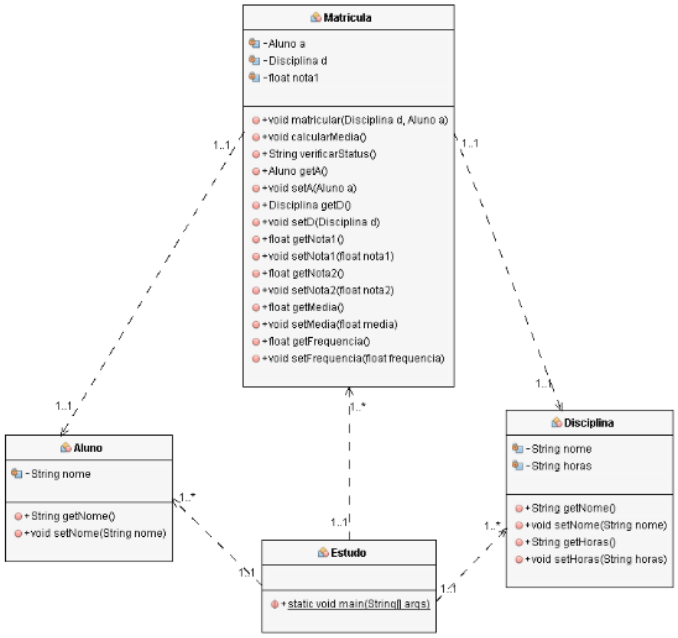


Figura 6.11. Diagrama de Classes do Sistema de Matrículas

A seguir, cada um dos passos da técnica será aplicado nesse exemplo.

Passo 1. Use a ferramenta CRISTA para gerar a ordem de inspeção das classes usando a métrica *Acoplamento (CBO)*.

Considerando que o inspetor tenha selecionado o uso da KIRT-Acoplamento (CBO) (vide Figura 6.7a), a ordem gerada pela CRISTA para inspeção das classes é Disciplina, Aluno, Matricula e Estudo (vide Figura 6.7b). Como mencionado anteriormente, ao clicar no nó Disciplina, a CRISTA apresenta as métricas calculadas, conforme Figura 6.12.

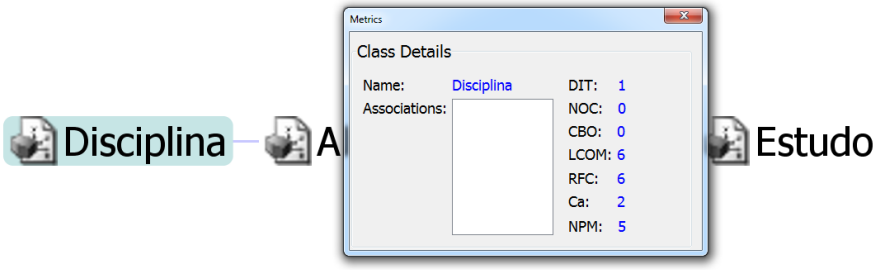


Figura 6.12. Métricas calculadas da classe Disciplina

Passo 2. Para cada classe estabelecida na ordem gerada pela métrica *Acoplamento (CBO)* faça:

Dada a ordem gerada pela CRISTA, o inspetor deve iniciar a inspeção pela classe Disciplina. Uma vez escolhida essa classe, a CRISTA apresenta a tela da Figura 6.13.

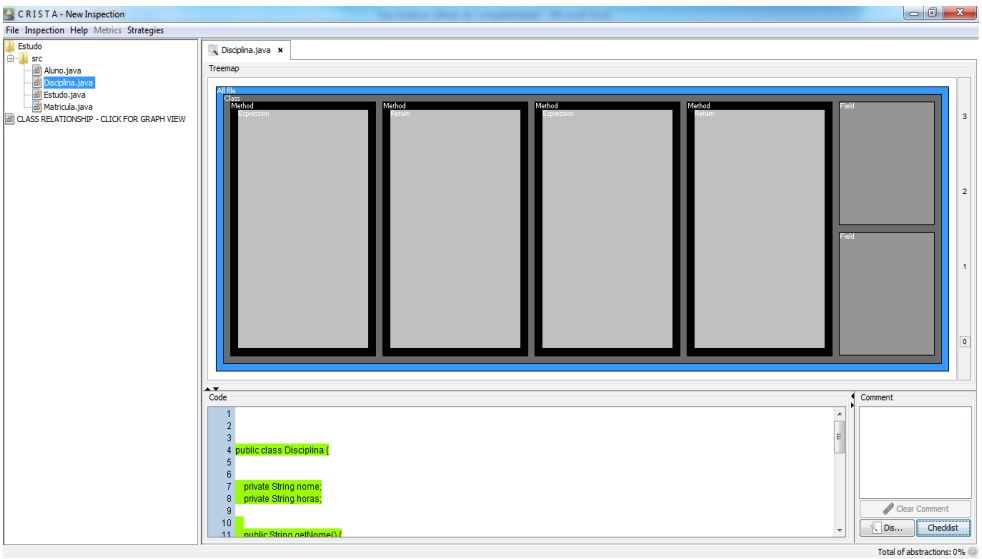


Figura 6.13. Métodos destacados na *treemap* da classe Disciplina

Observe na Figura 6.13 que todos os blocos destacados em preto correspondem a métodos da classe Disciplina.

Passo 2a. Para cada método destacado na metáfora visual faça:

i. Abstraia a funcionalidade do método usando as informações-chave documento de requisitos e diagrama de classes. Durante a abstração registre, usando a ferramenta CRISTA, quaisquer possíveis discrepâncias ou melhorias que possam ser realizadas no código inspecionado.

Considerando a Figura 6.14, o inspetor selecionou o método *setHoras* uma vez que o bloco da *treemap* está destacado em azul. Nesse momento ele pode abstrair a funcionalidade desse método por meio da área de texto *Comment* no canto inferior direito da tela.

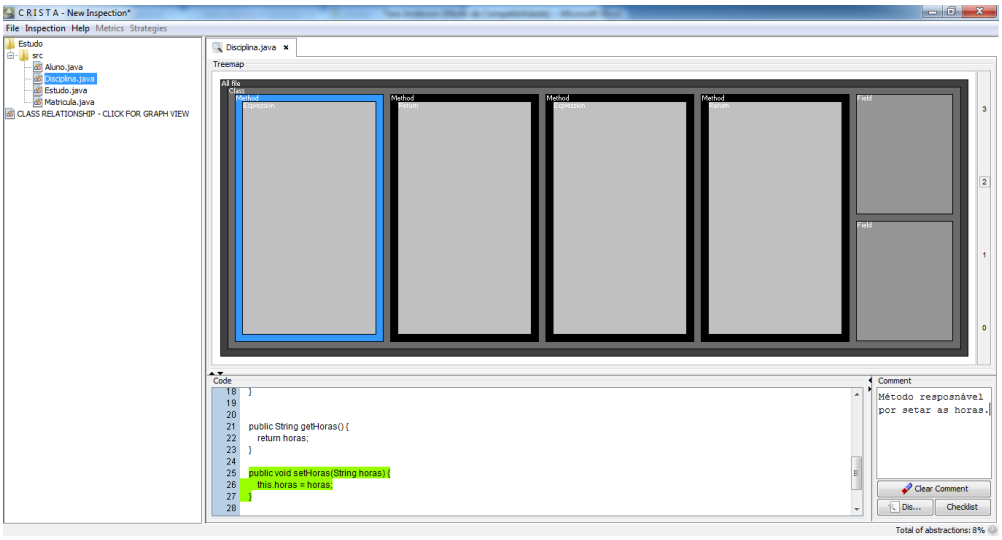


Figura 6.14. Abstração do método *setHoras* da classe Disciplina

Durante a abstração de cada método, o inspetor pode identificar discrepâncias no código-fonte. Para registrar uma discrepância o inspetor deve clicar com o botão direito do mouse no trecho de código que possui a discrepância. Uma nova janela é aberta (Figura 6.15) para que o inspetor possa relatar a discrepância.

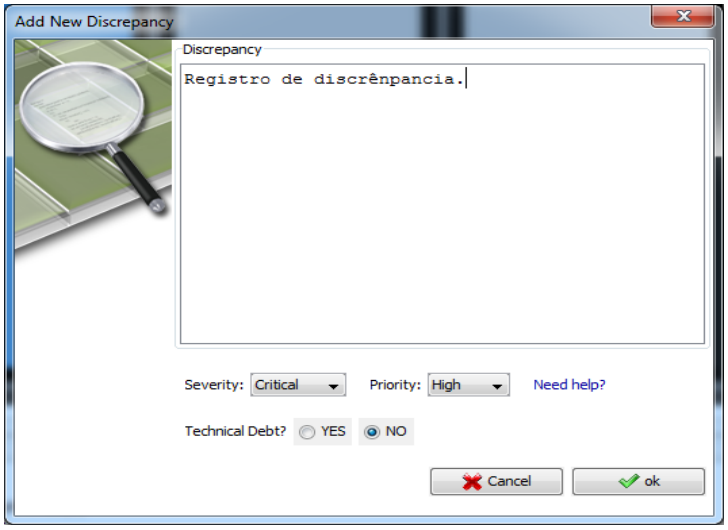


Figura 6.15. Tela de registro de discrepâncias na CRISTA

Observe na tela de registro de discrepâncias (Figura 6.15) que o inspetor classifica a discrepância em níveis de severidade e prioridade. Os níveis de severidade e prioridade escolhidos são utilizados para geração da cor dos blocos que possuem discrepâncias. As cores dos blocos variam de tons que vão do vermelho escuro (mais críticos e mais prioritários) até o amarelo claro (menos críticos e menos prioritários). A Figura 6.16 apresenta um bloco com uma discrepância registrada com severidade crítica e prioridade máxima (cor vermelha) e outra discrepância registrada com severidade trivial e prioridade média (cor amarela).

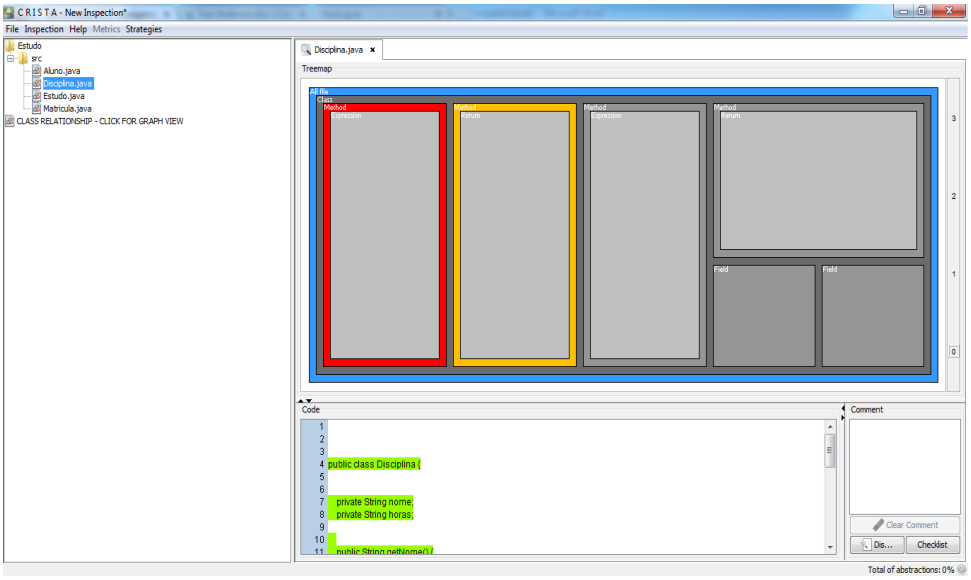


Figura 6.16. Blocos de código com discrepâncias registradas

Passo 2b. Para cada seção do *checklist*, use a metáfora visual para destacar os blocos de código que devem ser verificados por meio de suas questões.

i. Para cada bloco de código destacado, registre na ferramenta CRISTA, possíveis discrepâncias ou melhorias que possam ser realizadas no código inspecionado.

Considerando que o *checklist* está dividido nas seções *Variáveis*, *Métodos*, *Construtores*, *Computação*, *Condição*, *Repetição* e o inspetor está inspecionando a seção *Computação*, a CRISTA destaca blocos de código relacionados à seção escolhida, conforme apresentado na Figura 6.17.

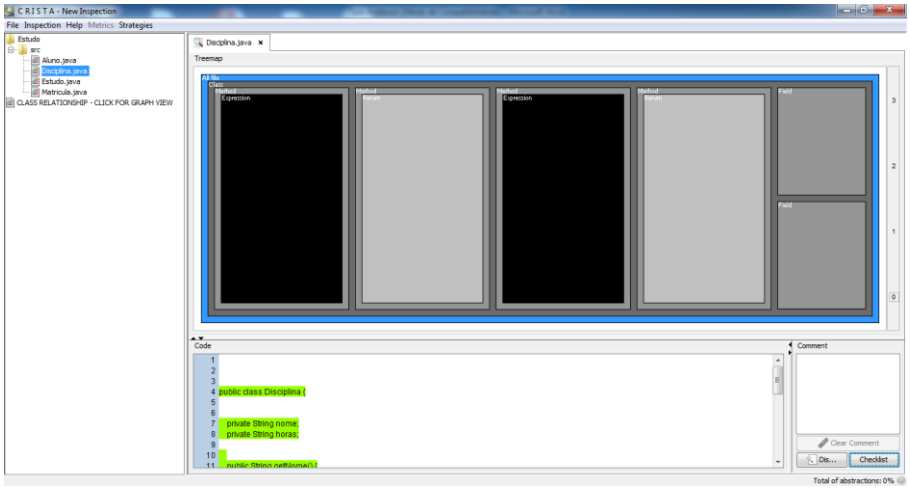


Figura 6.17. Blocos destacados (em cor preta) relacionados à seção *Computação* do *checklist*

Ressalta-se que os passos ilustrados anteriormente devem ser repetidos até que todas as classes do sistema tenham sido inspecionadas. Ressalta-se também que caso o inspetor selecione uma métrica diferente de acoplamento, o procedimento a ser realizado é o mesmo, exceto pela ordem de inspeção das classes, que será dependente da métrica escolhida.

6.5 Considerações Finais

Esse capítulo apresentou a técnica de leitura KIRT que foi definida com base em cenários operacionais (LAITENBERGER; DEBAUD, 1997) e constitui uma técnica sistemática e algorítmica. A definição da técnica foi fundamentada a partir das evidências obtidas no estudo exploratório relatado no Capítulo 5 e no desempenho do uso de visualização

relatado no Capítulo 3. Tais evidências podem ser resumidas no seguinte ponto: o uso do documento de requisitos, diagrama de classes e métricas de software influenciam nos resultados da atividade de inspeção. Essa influência provoca homogeneidade no comportamento dos inspetores, na efetividade da identificação de defeitos e de melhorias e diminuição no número de falso-positivos identificados.

Dessa forma, a técnica KIRT baseia-se no uso das informações-chave utilizadas no estudo do Capítulo 5: documento de requisitos, diagrama de classes e métricas de software. Além disso, por ser uma técnica de leitura o inspetor deve ser guiado a identificar defeitos pertinentes ao código. Portanto, uma taxonomia de defeitos e um modelo subjacente também eram necessários para compor a técnica de leitura baseada em um cenário operacional. Como taxonomia de defeitos foi utilizada a classificação definida por Dunsmore *et al.* (2003b) e como modelo subjacente foi utilizada a abstração do código-fonte guiada pela metáfora visual. Salienta-se que a metáfora visual apresenta dois níveis de abstração do código – nível de unidade (*treemap*) e o nível de integração de classes (grafo de relacionamento), o que permite estabelecer um paralelo com a atividade de teste de unidade e de integração. Utilizando-se a taxonomia de defeitos e o modelo subjacente estabeleceu-se um algoritmo para sistematizar a atividade de inspeção de código. Esse algoritmo é instanciado pela métrica de software utilizada, que altera ordem de leitura do código de acordo com a métrica selecionada.

Nesse sentido, tem-se um procedimento baseado em cenários operacionais para instanciação de outras técnicas de leitura. Por exemplo, um inspetor pode utilizar o cenário operacional e definir uma nova instanciação da técnica KIRT por meio de outras métricas (por exemplo, classes mais alteradas em um sistema de controle de versões) e/ou outro checklist que represente a realidade da empresa na qual a técnica está sendo aplicada.

Para ilustrar a aplicação da técnica KIRT foi utilizada a técnica KIRT instanciada com a métrica Acoplamento a fim de apresentar o algoritmo a ser aplicado para leitura do código-fonte bem como os recursos computacionais presentes na ferramenta CRISTA que auxiliam na aplicação da técnica.

Portanto, a técnica KIRT foi extraída da observação dos comportamentos de inspetores durante a atividade de inspeção e dos resultados evidenciados. Os principais diferenciais da técnica KIRT estão no uso de metáforas visuais que guiam o inspetor na navegação pelo código, e no estabelecimento da sistematização da leitura propriamente. Salienta-se que a abordagem mais sistemática encontrada na literatura é a de Dunsmore *et al.*

(2003b). No entanto, esse autor não propõe o uso de visualização e de um guia passo-a-passo para leitura do código.

Embora sistemáticas, a técnica KIRT não exige uma abstração de todo o código como a *Stepwise Abstraction* fazia no paradigma procedimental. Dessa forma, o inspetor fica livre para não abstrair trechos de código com funcionalidades evidentes. Assim, a técnica KIRT é uma alternativa de uso de outras técnicas identificadas na literatura possibilitando melhoria na qualidade do software produzido, contribuindo na identificação de defeitos e melhorias.

A avaliação da técnica KIRT foi realizada por meio de estudos experimentais que estão descritos no próximo capítulo.

Capítulo 7

CICLO 4 - TÉCNICA DE LEITURA KIRT: ESTUDOS EXPERIMENTAIS

Este capítulo apresenta três estudos experimentais realizados para avaliar a técnica de leitura KIRT: o primeiro explora a visualização; o segundo faz um comparativo entre KIRT-Acoplamento e KIRT-Coesão e o terceiro compara a técnica KIRT-Acoplamento com uma técnica Checklist.

7.1 Considerações iniciais

Considerando os diferenciais da técnica de leitura KIRT, foram planejados três estudos experimentais para avaliação da técnica.

O primeiro experimento teve como objetivo avaliar especificamente a contribuição da visualização para apoiar o *checklist* desenvolvido para a técnica KIRT. Dessa forma, o direcionamento da leitura estabelecido pela métrica não foi utilizado pois a principal análise estava relacionada a facilidade de aplicação do *checklist* em conjunto com a visualização, contribuindo com a identificação dos defeitos e melhorando o desempenho da atividade de inspeção.

O segundo experimento teve como objetivo comparar a técnica de leitura KIRT-Acoplamento e KIRT-Coesão em relação ao desempenho associado com a identificação de defeitos e melhorias pois se considera que possam existir contribuições diferenciadas dependendo da técnica aplicada.

O terceiro experimento teve como objetivo comparar a técnica KIRT-Acoplamento com a técnica Checklist proposta por Dunsmore *et al.* (2003b), que foi identificada como a abordagem que apresentava maior similaridade a este trabalho e que foi considerada pelo autor como a técnica que obteve melhor desempenho nos estudos por ele realizado.

O capítulo está organizado da seguinte maneira: na Seção 7.2 é apresentado o estudo experimental I; o estudo experimental II é abordado na Seção 7.3 e o estudo experimental III na Seção 7.4. A Seção 7.5 apresenta as considerações finais do capítulo.

7.2 Estudo Experimental I

Esta seção apresenta os detalhes do estudo experimental realizado para avaliar a contribuição da visualização para apoiar o *checklist* desenvolvido para a técnica KIRT. Para tanto, foram utilizadas duas versões da ferramenta CRISTA: uma com todas as metáforas visuais disponíveis, incluindo a específica para apoio ao *checklist* e outra versão em que a metáfora visual associada ao *checklist* não estava disponível.

Dessa forma, utilizando o modelo proposto na técnica GQM (BASILI; CALDIERA; ROMBACH, 1994) o planejamento do experimento pode ser resumido da seguinte forma:

Analisar	o uso de visualização para apoio ao *checklist*
Com o propósito de	caracterizar
Em relação	aos defeitos identificados no código
Do ponto de vista	de pesquisadores da área de inspeção de código
No contexto de	estudantes do Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos

7.2.1 Seleção do Contexto e dos Participantes

O experimento foi considerado uma atividade da disciplina Verificação e Validação de Software do Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos. Os participantes seriam contemplados com uma nota para participação no experimento, sendo que esta não estaria associada ao desempenho na identificação de defeitos. Ressalta-se que, de acordo com o formulário de caracterização dos participantes, 85% já havia realizado alguma atividade de inspeção profissionalmente. Além disso, 83% dos participantes possuíam experiência em desenvolvimento de sistemas com linguagens orientadas a objetos, especialmente a linguagem Java.

7.2.2 Seleção das Variáveis

Para esse experimento foi considerada uma variável independente denominada "VisualizaçãoChecklist", a qual representa o uso ou não de visualização associada ao *checklist*. A variável independente possui dois tratamentos:

- CRISTA-Checklist: representada pelo uso da ferramenta CRISTA com a visualização para o suporte do *checklist* habilitada.
- CRISTA: representada pelo uso da ferramenta CRISTA com a visualização para o suporte do *checklist* não habilitada.

Foram consideradas três variáveis dependentes:

- Efetividade: definida como a quantidade de defeitos reais identificados pelo inspetor.
- Eficiência: definida como a taxa entre defeitos reais identificados e o tempo de inspeção.
- Melhorias no código: quantidade de melhorias que podem ser realizadas no código e que foram identificadas pelos inspetores.

7.2.3 Projeto Experimental

O projeto experimental foi realizado com dois grupos de quatro participantes. Um grupo utilizou o tratamento CRISTA-Checklist e outro grupo o tratamento CRISTA. Cada participante realizou o experimento individualmente. As atividades realizadas por cada grupo estão descritas na Tabela 7.1.

7.2.4 Instrumentação

Os instrumentos do experimento foram o software de Sistema de Matrículas e Acompanhamento de Alunos, documento de requisitos, diagrama de classes, *checklist* da técnica KIRT e as duas versões da ferramenta (CRISTA-Checklist e CRISTA).

O Sistema de Matrículas e Acompanhamento de Alunos é implementado em Java, possui quatro classes e três defeitos de código conhecidos.

7.2.5 Preparação e Execução do Experimento

O experimento foi realizado em três dias conforme apresentado na Tabela 7.1. Ressalta-se que os participantes foram divididos em dois grupos após o preenchimento do questionário de caracterização, permitindo a criação de grupos homogêneos.

Tabela 7.1. Atividades realizadas no Estudo Experimental I

	GRUPO 1	**GRUPO 2**
Dia 1	1. Treinamento em Inspeção de Código (30 minutos) 2. Treinamento na ferramenta CRISTA sem suporte à visualização do *checklist* (30 minutos) 3. Aplicação em um sistema exemplo (30 minutos) 4. Discussão dos resultados obtidos e das dúvidas encontradas pelos participantes (50 minutos)	
Dia 2	1. Inspeção no sistema de matrícula usando CRISTA (90 minutos)	1. Treinamento na ferramenta CRISTA-Checklist (20 minutos) 2. Inspeção no sistema de matrícula usando CRISTA-Checklist (90 minutos)
Dia 3	1. Discussão dos resultados (60 minutos)	

7.2.6 Análise e Discussão dos Resultados

Todas as discrepâncias registradas nas duas versões da ferramenta foram analisadas a fim de classificá-las em defeitos, falso-positivos ou melhorias. A Tabela 7.2 apresenta um resumo da frequência de discrepâncias, melhorias, falso-positivos e defeitos identificados por cada participante em relação a cada classe do Sistema de Matrícula e Acompanhamento de Alunos.

É interessante observar que os três defeitos conhecidos foram identificados por todos os participantes independentemente de qual versão da ferramenta CRISTA foi utilizada. No entanto, os participantes que utilizaram a versão CRISTA-Checklist tiveram um maior número de melhorias identificadas e um menor número de falso-positivos, conforme observado na Figura 7.1.

Tabela 7.2. Resumo dos registros obtidos pelos participantes

		Classes	Discrepâncias Registradas	Melhorias	Falso-Positivo	Defeitos
Grupo 1 (CRISTA)	Inspetor 1	Aluno	0	0	0	0
		Disciplina	0	0	0	0
		Estudo	0	0	0	0
		Matrícula	4	0	1	3
	Inspetor 2	Aluno	0	0	0	0
		Disciplina	0	0	0	0
		Estudo	0	0	0	0
		Matrícula	11	6	2	3
	Inspetor 3	Aluno	0	0	0	0
		Disciplina	0	0	0	0
		Estudo	0	0	0	0
		Matrícula	10	5	2	3
	Inspetor 4	Aluno	0	0	0	0
		Disciplina	0	0	0	0
		Estudo	0	0	0	0
		Matrícula	5	2	0	3
Grupo 2 (CRISTA-Checklist)	Inspetor 5	Aluno	0	0	0	0
		Disciplina	0	0	0	0
		Estudo	0	0	0	0
		Matrícula	5	0	2	3
	Inspetor 6	Aluno	0	0	0	0
		Disciplina	1	0	1	0
		Estudo	0	0	0	0
		Matrícula	6	3	0	3
	Inspetor 7	Aluno	1	1	0	0
		Disciplina	1	1	0	0
		Estudo	3	2	1	0
		Matrícula	10	7	0	3
	Inspetor 8	Aluno	0	1	0	0
		Disciplina	0	1	0	0
		Estudo	0	2	1	0
		Matrícula	4	1	0	3

De acordo com a Tabela 7.3, ambos os grupos foram igualmente efetivos na identificação de defeitos. No entanto, ao analisar os valores médios de cada grupo verifica-se que o grupo que utilizou a CRISTA-Checklist apresentou um menor número de falso-positivos e mais melhorias dentre as discrepâncias registradas. Outra informação importante diz respeito ao tempo médio de aplicação dos dois grupos. Observe que mesmo havendo mais efetividade nas discrepâncias registradas (mesmo número de discrepâncias e menor número de falso-positivos), o tempo médio do grupo CRISTA-Checklist foi bem próximo do grupo CRISTA.

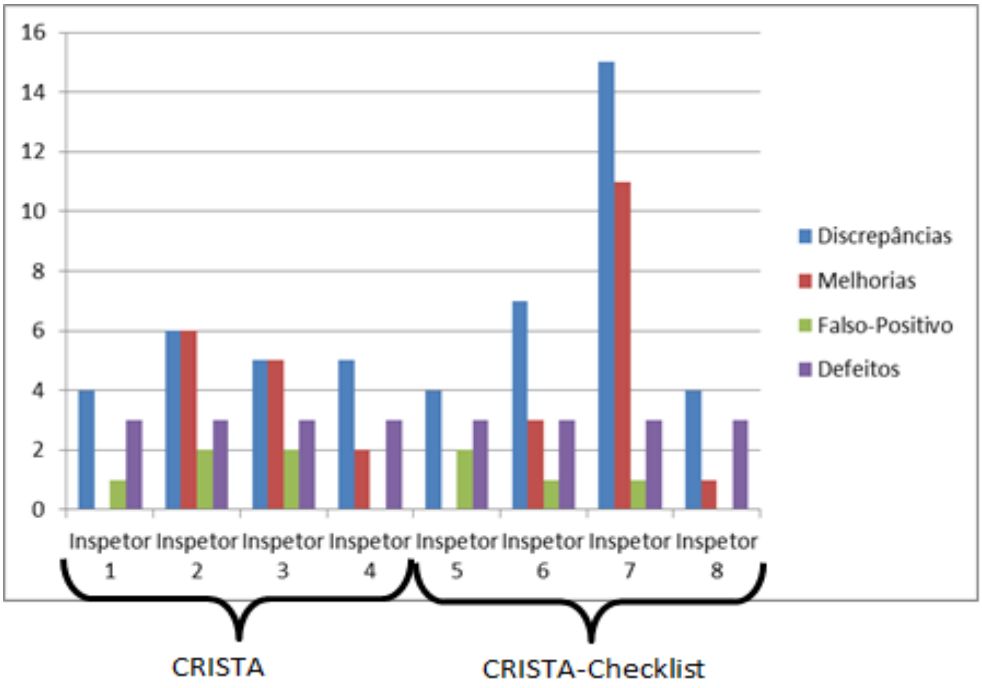


Figura 7.1. Resultados obtidos por cada participante

Tabela 7.3. Valores médios dos participantes dos dois grupos

	Discrepâncias	**Melhorias**	**Falso-Positivos**	**Defeitos**	**Tempo**
CRISTA	7,5	3,25	1,25	3	34 min.
CRISTA-Checklist	7,5	3,75	1,00	3	39 min.

Algumas observações interessantes a partir dos resultados dos experimentos dizem respeito ao número considerável de melhorias que foram identificadas pelos participantes do grupo CRISTA-Checklist, especialmente do inspetor 7. Ressalta-se que os inspetores do grupo CRISTA-Checklist conseguiram aplicar o *checklist* de maneira completa e em todos os trechos de código-fonte. Tal facilidade foi conseguida pelo uso da visualização de acordo com entrevistas realizadas com os participantes após a discussão dos resultados do experimento. Um dos inspetores relatou que "o uso da ferramenta possibilitou concentrar em trechos específicos do código quando era usado o *checklist*". Em relação aos participantes do grupo CRISTA algumas questões do *checklist* não foram utilizadas devido à dificuldade de gerenciar sua aplicação em todas as classes. Daí, talvez, explica-se o fato de que o tempo médio do grupo CRISTA tenha sido próximo do tempo médio do grupo CRISTA-Checklist que comentou ter aplicado todo o *checklist* em decorrência do suporte da metáfora visual.

Observa-se que nenhum dos grupos desse estudo utilizou qualquer outro guia para orientá-los durante a inspeção, como por exemplo, a ordem das classes a serem inspecionadas. Nesse sentido, a intenção era apenas verificar a influência do uso da metáfora visual para facilitar a aplicação do *checklist*.

7.2.7 Ameaças à Validade

As ameaças à validade podem ser classificadas em interna, externa, de construção e de conclusão (WOHLIN et al., 2000).

A validade interna refere-se a fatores não controlados que podem influenciar nas variáveis dependentes, destacando:

- Participantes: os participantes eram estudantes. No entanto, como mencionado por Carver *et al.* (2003), há vantagens no uso de estudantes no contexto de experimentos na área de engenharia de software, como por exemplo, a obtenção de evidências preliminares para confirmar ou refutar uma hipótese, controlar fatores que podem afetar o estudo, apresentar às empresas de software a relevância da pesquisa, produzir um kit experimental, entre outras. Em particular, no caso deste estudo, os participantes eram todos estudantes de pós-graduação e tinham alguma experiência (em média 8 anos e meio) com inspeção e, principalmente, com desenvolvimento de software.
- Comunicação: a comunicação entre participantes do próprio grupo e entre grupos não era permitida.
- Fadiga: a fadiga por parte dos inspetores foi minimizada, pois a atividade de inspeção foi planejada para consumir, no máximo, 1 hora e meia de aplicação.
- Treinamento: uma possível ameaça estava relacionada à falta de familiaridade com o uso das técnicas. Essa ameaça foi minimizada com os treinamentos e os *feedbacks* fornecidos antes da execução do experimento.

A validade externa refere-se à capacidade de generalizar os resultados do experimento, destacando:

- Participantes: os participantes desse experimento eram estudantes, o que não permite a generalização dos resultados.
- Sistema: o sistema de Matrículas utilizado no estudo experimental era muito simples.

A validade de construção está relacionada ao grau com que as métricas utilizadas para avaliar os resultados do experimento (variáveis dependentes) de fato representam as medidas necessárias para caracterizar o objeto de estudo (as técnicas de leitura definidas), destacando:

- Métricas (efetividade, eficiência e número de melhorias): essas métricas têm sido utilizadas por vários pesquisadores em outros estudos experimentais com objetivos similares a este.

Salienta-se que, mesmo sendo possível aplicar um teste estatístico quando a quantidade de participantes é pequena, os resultados não justificariam uma análise estatística baseada no número de defeitos uma vez que a quantidade de defeitos identificados por cada técnica foi a mesma.

7.2.8 Conclusões do Estudo Experimental I

O estudo experimental relatado nesta seção avaliou o uso do recurso de visualização aplicado ao *checklist*. Os resultados apresentados fornecem indícios que o suporte de visualização aplicado ao *checklist* auxilia o inspetor na atividade de compreensão e inspeção do código uma vez que ele se concentra em trechos específicos de código relacionados a seção do *checklist* sendo utilizada.

Alguns participantes relataram que o destaque de trechos de código a serem inspecionados minimizou o tempo necessário para aplicação do checklist além de focar em trechos de código condizentes com cada seção do *checklist*.

Salienta-se também que o número de falso-positivos identificados pelos inspetores que utilizam a metáfora visual habilitada para o *checklist* foi menor que os inspetores que utilizaram o *checklist* sem a metáfora visual habilitada. Tal resultado também pode ser um indício que a metáfora visual para aplicação do *checklist* melhora o desempenho da atividade de inspeção de código.

7.3 Estudo Experimental II

Esta seção apresenta os detalhes do estudo experimental realizado para avaliar a efetividade e eficiência da técnica KIRT, levando em consideração as duas métricas mais utilizadas no estudo exploratório relatado no Capítulo 5 - Acoplamento e Coesão. Além disso, tinha-se o interesse de avaliar o aspecto complementar das técnicas definidas, as quais podem compor uma estratégia para inspeção em nível de unidade e de integração.

Dessa forma, utilizando o modelo proposto na técnica GQM (Basili *et al.*, 1994) o planejamento do experimento pode ser resumido da seguinte forma:

Analisar	as técnicas de leitura KIRT-Acoplamento e KIRT-Coesão
Com o propósito de	avaliar
Em relação	à eficiência e efetividade na identificação de defeitos e melhorias
Do ponto de vista	de pesquisadores da área de inspeção de código
No contexto de	estudantes do Programa de Pós-Graduação em Ciência da Computação da Universidade de Federal de São Carlos

7.3.1 Seleção do Contexto e dos Participantes

Os participantes desse experimento são os mesmos participantes do experimento anterior e o conhecimento adquirido no Estudo Experimental I teve, supostamente, pouca influência sobre este, visto que um novo treinamento foi realizado e os artefatos utilizados foram outros.

7.3.2 Seleção das Variáveis

Para esse experimento foi considerada uma variável independente, chamada "TécnicaLeitura", a qual representa a técnica de leitura utilizada pelo inspetor. Salienta-se que nesse experimento todos os participantes usaram a mesma versão da ferramenta CRISTA com todas as funcionalidades de visualização habilitadas. A variável independente possui dois tratamentos:

- KIRT-Acoplamento: representada pela aplicação da técnica KIRT-Acoplamento com o suporte computacional da ferramenta CRISTA.
- KIRT-Coesão: representada pela aplicação da técnica KIRT-Coesão com o suporte computacional da ferramenta CRISTA.

Foram consideradas três variáveis dependentes:

- Efetividade: definida como a quantidade de defeitos reais identificados pelo inspetor.
- Eficiência: definida como a taxa entre defeitos reais identificados e o tempo de inspeção.

- Melhorias de Código: quantidade de melhorias que podem ser realizadas no código e que foram identificadas pelos inspetores.

7.3.3 Projeto Experimental

O projeto experimental foi realizado com dois grupos de quatro participantes. Um grupo utilizou o tratamento KIRT-Acoplamento e outro grupo o tratamento KIRT-Coesão. Cada participante realizou o experimento individualmente.

7.3.4 Instrumentação

Os instrumentos do experimento foram o software *Paint*, documento de requisitos, diagrama de classes, *checklist* da técnica KIRT e a ferramenta CRISTA.

O software *Paint* é o mesmo utilizado em outros estudos experimentais (ROBILLARD; COELHO; MURPHY, 2004) (KO et al., 2006) (NISHIZONO et al., 2011) e também usado no estudo exploratório relatado no Capítulo 3. O *Paint* é composto por nove classes e possui um total de 11 defeitos conhecidos.

7.3.5 Preparação e Execução do Experimento

O experimento foi realizado em três dias conforme apresentado na Tabela 7.4. Ressalta-se que os participantes foram divididos em dois grupos após o preenchimento do questionário de caracterização, permitindo a criação de grupos homogêneos.

Tabela 7.4. Atividades realizadas no Estudo Experimental II

	GRUPO 1	**GRUPO 2**
Dia 1	1. Treinamento na KIRT-Acoplamento em um sistema exemplo (30 minutos) 2. Discussão dos resultados obtidos e das dúvidas encontradas pelos participantes (50 minutos)	1. Treinamento na KIRT-Coesão em um sistema exemplo (30 minutos) 2. Discussão dos resultados obtidos e das dúvidas encontradas pelos participantes (50 minutos)
Dia 2	1. Inspeção do *Paint* usando a KIRT-Acoplamento (90 minutos)	1. Inspeção do *Paint* usando a KIRT-Coesão (90 minutos)
Dia 3	1. Discussão dos resultados (60 minutos)	

7.3.6 Análise e Discussão dos Resultados

Todas as discrepâncias registradas com a aplicação das duas técnicas foram analisadas a fim de classificá-las em defeitos, falso-positivos ou melhorias. A Figura 7.2 apresenta um resumo do número de discrepâncias, melhorias, falso-positivos e defeitos encontrados por cada participante.

Observa-se na Figura 7.2 que os inspetores identificaram quase todos os 11 defeitos conhecidos do *Paint*. O participante que encontrou a menor ocorrência de defeitos foi o inspetor 4, com 6 defeitos identificados.

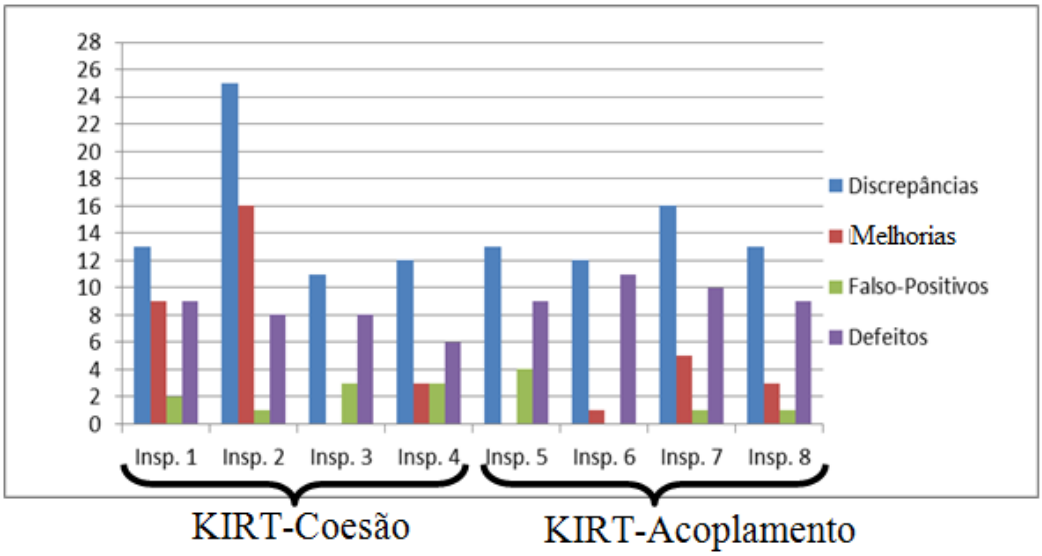


Figura 7.2. Resultados obtidos por cada participante na inspeção do sistema *Paint*

É importante salientar que o número de falso-positivos foi relativamente baixo para ambas as técnicas utilizadas. Especificamente para o Inspetor 6, nenhum falso positivo foi identificado e para os inspetores 7 e 8 apenas um falso positivo.

De acordo com a Tabela 7.5, ao analisar os valores médios obtidos por cada grupo verifica-se que o grupo KIRT-Acoplamento teve um melhor desempenho em relação ao número de defeitos identificados pelo grupo KIRT-Coesão. No entanto, o número de melhorias identificadas pelo grupo KIRT-Coesão foi, em média, três vezes maior que o grupo KIRT-Acoplamento. Tal resultado, embora não comprovado estatisticamente, pode ser um indício de que o uso da técnica de leitura baseada na métrica coesão tem um melhor desempenho para a identificação de melhorias.

Tabela 7.5. Valores médios dos participantes dos dois grupos

	Discrepâncias	**Melhorias**	**Falso-Positivos**	**Defeitos**	**Tempo**
KIRT-Coesão	15,25	5,25	2,25	7,75	66,2 min.
KIRT-Acoplamento	13,50	2,25	1,50	9,75	37,7 min.

Em uma análise individualizada das melhorias identificadas pelos inspetores do grupo KIRT-Coesão verificou-se que os primeiros métodos compreendidos e inspecionados possuíam baixa dependência de outras classes. Esse fato pode ter levado a identificação de mais melhorias visto que iniciando a compreensão do código pelas partes mais coesas, acaba-se tendo uma melhor compreensão do sistema como um todo e, portanto, uma maior facilidade de identificação de melhorias.

A Tabela 7.6 apresenta os valores das métricas *precision* e *recall* para os defeitos identificados pelos inspetores dos grupos KIRT-Coesão e KIRT-Acoplamento.

Tabela 7.6. *Precision* e *Recall* para os inspetores

Grupo	**Inspetores**	**Precision**	**Recall**
KIRT-Coesão	Insp. 1	69,2 %	81,8 %
	Insp. 2	32,0 %	72,7 %
	Insp. 3	72,7 %	72,7 %
	Insp. 4	50,0 %	54,5 %
KIRT-Acoplamento	Insp. 5	69,2 %	81,8 %
	Insp. 6	91,7 %	100,0 %
	Insp. 7	62,5 %	90,9 %
	Insp. 8	69,2 %	81,8 %

Observe que os valores de *precision* e *recall* dos inspetores do grupo KIRT-Acoplamento são melhores do que os valores do grupo KIRT-Coesão. A Tabela 7.7 apresenta a média das métricas *precision* e *recall* dos inspetores de cada grupo.

Tabela 7.7. Média das métricas *precision* e *recall* de cada grupo

Grupo	**PRECISION**	**RECALL**
KIRT-Coesão	56,6	70,5
KIRT-Acoplamento	73,2	88,6

Com base nos valores obtidos pelas médias das métricas *precision* e *recall*, a métrica *F-measure* foi calculada (BAEZA-YATES; RIBEIRO-NETO, 1999) para avaliar a efetividade de cada uma das técnicas (DELFIM et al., 2015). A fórmula de cálculo de *F-measure* é dada por:

*F-measure = 2 * (precision * recall) / (precision + recall)*

Os valores obtidos foram 48,7% e 80,1% para as técnicas de leitura KIRT-Coesão e KIRT-Acoplamento, respectivamente, mostrando, portanto, uma maior efetividade da KIRT-Acoplamento.

No entanto, é interessante analisar quais foram os defeitos identificados e quais suas características, a fim de verificar a influência da técnica na identificação de cada defeito.

A Tabela 7.8 apresenta a classificação dos defeitos do *Paint* de acordo com a taxonomia de Dunsmore *et al.* (2003b) e a frequência que cada defeito foi identificado por cada técnica.

Tabela 7.8. Classificação dos defeitos e frequência que cada defeito foi encontrado no sistema *Paint*

INSPETORES	**D1**	**D2**	**D3**	**D4**	**D5**	**D6**	**D7**	**D8**	**D9**	**D10**	**D11**
Objeto incorreto						X					
Método incorreto									X		
Parâmetro incorreto		X									
Computação							X	X		X	X
Fluxo de dados			X								
Erro de Especificação				X							
Omissão	X			X							
Comissão					X						
Localidade	C	S	S	S	C	C	M	M	M	M	M
Tamanho	L	L	S	---	L	L	S	L	L	M	M
KIRT-Coesão (%)	**75**	**100**	**50**	**25**	**100**	**50**	**100**	**50**	**25**	**100**	**100**
KIRT-Acoplamento (%)	**100**	**100**	**100**	**50**	**100**	**100**	**100**	**100**	**25**	**100**	**100**

De acordo com a Tabela 7.8, todos os inspetores identificaram praticamente a mesma quantidade de defeitos, independentemente da técnica utilizada. Para os defeitos D1, D3, D4, D6 e D8, o uso da técnica KIRT-Acoplamento teve um melhor desempenho.

Considerando a questão da localidade do defeito (Método, Classe e Sistema) as técnicas tiveram o comportamento apresentado na Figura 7.3. Observe que a técnica KIRT-Acoplamento teve um melhor desempenho em relação a todas as localizações. Ressalta-se que a maior diferença ocorreu na localidade Sistema, fornecendo indícios de que o uso da técnica KIRT-Acoplamento permite uma melhor cobertura de defeitos relacionados à delocalização.

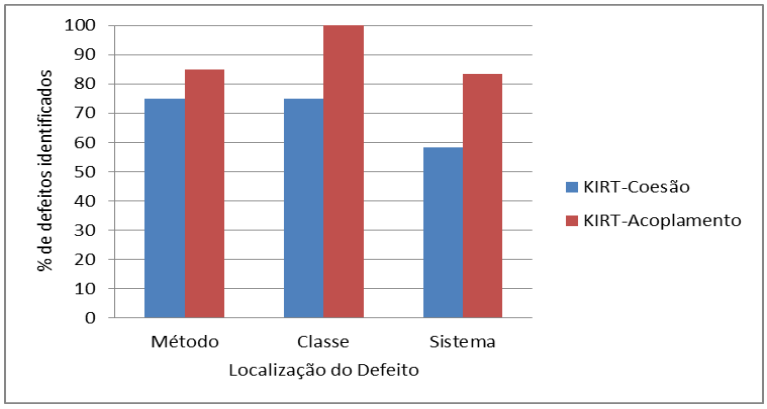


Figura 7.3. Porcentagem de defeitos identificados de acordo com a localização e técnica de leitura

A Figura 7.4 apresenta a porcentagem de identificação dos defeitos do *Paint* encontrados por cada uma das técnicas. Sabendo-se que defeitos delocalizados são mais difíceis de serem encontrados, pois implicam em uma compreensão de trechos de códigos espalhados por diversas classes, e sabendo também que os defeitos D2, D3 e D4 são defeitos delocalizados, observe que todos os participantes que utilizaram a técnica KIRT-Acoplamento identificaram os defeitos D2 e D3.

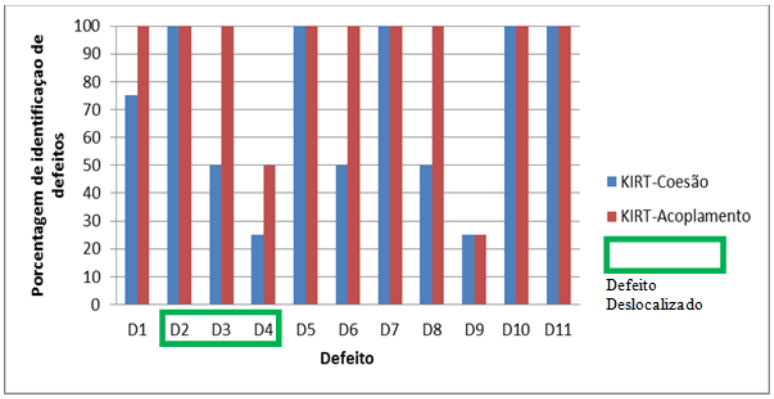


Figura 7.4. Porcentagem de identificação para os defeitos do *Paint*

Outra análise que merece atenção diz respeito ao tempo médio de aplicação das técnicas, mostrado na Tabela 7.5. Observe que o tempo médio associado à técnica KIRT-Coesão foi quase duas vezes maior que o tempo associado à técnica KIRT-Acoplamento. A Tabela 7.9 apresenta a eficiência, experiência e efetividade de cada inspetor. Em média, a eficiência dos inspetores dos grupos KIRT-Coesão e KIRT-Acoplamento foram, respectivamente, 11,9% e 26,14%.

Tabela 7.9. Eficiência, experiência e efetividade dos inspetores na inspeção do *Paint*

Grupo	Inspetores	Defeitos	Tempo (minutos)	Eficiência (defeitos/tempo)	Experiência Profissional (em meses)	Efetividade
KIRT-Coesão	Insp. 1	9	82	10,9 %	6	81,8 %
	Insp. 2	8	50	16 %	88	72,7 %
	Insp. 3	8	71	11,2 %	240	72,7 %
	Insp. 4	6	62	9,6 %	30	54,5 %
KIRT-Acoplamento	Insp. 5	9	40	22,5 %	210	81,8 %
	Insp. 6	11	34	32,3 %	60	100 %
	Insp. 7	10	36	27,7 %	8	90,9 %
	Insp. 8	9	41	21,9 %	48	81,8 %

Observe na Tabela 7.9 que, aparentemente, a experiência profissional não teve influência na eficiência dos inspetores. Por exemplo, inspetor (Insp. 7), com menor experiência profissional, teve uma melhor eficiência que inspetores mais experientes do grupo KIRT-Acoplamento. O mesmo se observa com o inspetor (Insp. 1) do grupo KIRT-Coesão.

Além das análises apresentadas anteriormente foram realizados três testes de hipóteses relacionados às três variáveis dependentes – efetividade, eficiência e melhorias no código. Devido ao projeto experimental (dois grupos com 4 participantes em cada grupo) foi utilizado o teste estatístico não-paramétrico Mann-Whitney para duas amostras independentes.

A primeira hipótese diz respeito à efetividade de cada uma das técnicas: *Há diferença significativa entre a efetividade das técnicas de leitura KIRT-Acoplamento e KIRT-Coesão.*

O p-valor resultante da aplicação do teste estatístico Mann-Whitney é de 0,0217 que, sendo menor que 0,05, representa que a hipótese nula pode ser rejeitada. Dessa forma, existe diferença significativa entre a efetividade de cada técnica.

A segunda hipótese diz respeito à eficiência de cada uma das técnicas: *Há diferença significativa entre a eficiência das técnicas de leitura KIRT-Acoplamento e KIRT- Coesão.*

O p-valor resultante da aplicação do teste estatístico Mann-Whitney é de 0, 0105 que, sendo menor que 0,05, representa que a hipótese nula pode ser rejeitada. Dessa forma, existe diferença significativa entre a eficiência de cada técnica.

A terceira hipótese diz respeito às melhorias identificadas por cada uma das técnicas: *Há diferença significativa entre o número de melhorias identificadas pelas técnicas de leitura KIRT-Acoplamento e KIRT-Coesão.*

O p-valor resultante da aplicação do teste estatístico Mann-Whitney é de 0,3864 que, não sendo menor que 0,05, representa que a hipótese nula não pode ser rejeitada. Dessa

forma, não existe diferença significativa entre o número de melhorias identificadas por cada técnica.

7.3.7 Ameaças à Validade

As ameaças à validade podem ser classificadas em interna, externa, de construção e de conclusão (WOHLIN et al., 2000).

A validade interna refere-se a fatores não controlados que podem influenciar nas variáveis dependentes, destacando:

- Participantes: o uso de estudantes como participantes poderia influenciar nas variáveis dependentes. No entanto, os participantes do experimento foram estudantes de pós-graduação com uma média de experiência profissional de 8 anos e meio, caracterizando certa maturidade.
- Comunicação: a comunicação entre participantes do próprio grupo e entre grupos não era permitida.
- Fadiga: a fadiga por parte dos inspetores foi minimizada, pois a atividade de inspeção foi planejada para consumir, no máximo, 1 hora e meia de aplicação.
- Treinamento: uma possível ameaça estava relacionada à falta de familiaridade com o uso das técnicas. Essa ameaça foi minimizada com os treinamentos e os feedbacks fornecidos antes da execução do experimento.

A validade externa refere-se à capacidade de generalizar os resultados do experimento, destacando:

- Participantes: o uso de estudantes como participantes não permite a generalização dos resultados obtidos no experimento para outros tipos de perfis de inspetores. No entanto, essa ameaça foi minimizada, como já explicado anteriormente, com a participação de estudantes de pós-graduação com experiência profissional.
- Sistema: o sistema *Paint* utilizado no estudo experimental, embora não desenvolvido profissionalmente, possui todas as características de um sistema orientado a objetos e já foi utilizado em outros estudos experimentais relatados na literatura.

A validade de conclusão está relacionada com as questões que afetam a interpretação de uma conclusão correta, destacando:

- Teste estatístico: mesmo com um número reduzido de participantes em cada grupo, foi possível aplicar o teste não-paramétrico (Mann-Whitnney) para avaliar a

efetividade e eficiência na identificação de defeitos e os números de melhorias identificadas.

A validade de construção está relacionada ao grau com que as métricas utilizadas para avaliar os resultados do experimento (variáveis dependentes) de fato representam as medidas necessárias para caracterizar o objeto de estudo (as técnicas de leitura definidas), destacando:

- Métricas (efetividade, eficiência e número de melhorias): essas métricas têm sido utilizadas por vários pesquisadores em outros estudos experimentais com objetivos similares a este.

7.3.8 Conclusões do Estudo Experimental II

O Estudo Experimental II teve como objetivo avaliar o desempenho relacionado a efetividade, eficiência e número de melhorias identificadas por inspetores que utilizaram a técnica KIRT instanciadas a partir das métricas Acoplamento e Coesão.

Os resultados e relatos obtidos apresentam indícios de que o uso da técnica KIRT-Acoplamento teve um melhor desempenho em relação a efetividade e eficiência na identificação de defeitos. Especificamente relacionado a identificação de melhorias, a técnica KIRT-Coesão teve um melhor desempenho. Esses resultados dão indícios de um aspectos de complementaridade com a técnica KIRT quando do uso da métricas Acoplamento e Coesão.

7.4 Estudo Experimental III

Esta seção apresenta os detalhes do estudo experimental realizado para avaliar a efetividade e eficiência da técnica de leitura KIRT-Acoplamento em relação à técnica *Checklist* desenvolvida por Dunsmore *et al.* (2003b), que obteve a melhor efetividade na identificação de defeitos nos experimentos por ele conduzidos.

Dessa forma, utilizando o modelo proposto na técnica GQM (BASILI; CALDIERA; ROMBACH, 1994) o planejamento do experimento pode ser resumido da seguinte forma:

Analisar	o uso das técnicas KIRT-Acoplamento e *Checklist*
Com o propósito de	avaliar
Em relação	à eficiência e efetividade na identificação de defeitos e melhorias

Do ponto de vista	de pesquisadores da área de inspeção de código
No contexto de	profissionais da área de desenvolvimento de software

7.4.1 Seleção do Contexto e dos Participantes

O experimento foi realizado com 8 profissionais da indústria de desenvolvimento de software com uma média 3,5 anos de experiência em desenvolvimento na linguagem JAVA. Todo o treinamento e a aplicação do experimento foram realizados no próprio ambiente de trabalho dos participantes. Nenhum dos participantes tinha conhecimento de técnicas de inspeção de código e a empresa em que trabalhavam possuía intenção em implantar uma área de qualidade. Nesse sentido, os participantes do experimento estavam motivados para a realização desse experimento pois havia uma oportunidade de entrar em uma nova área da empresa.

7.4.2 Seleção das Variáveis

Para esse experimento foi considerada uma variável independente, chamada "TécnicaLeitura", que representa a técnica de leitura utilizada por cada inspetor. A variável independente possui dois tratamentos:

- KIRT-Acoplamento: representada pela aplicação da técnica KIRT-Acoplamento com o suporte computacional da ferramenta CRISTA.
- *Checklist*: representada pelo uso da técnica *Checklist* proposta por Dunsmore (2003).

Foram consideradas três variáveis dependentes:

- Efetividade: definida como a quantidade de defeitos reais identificados pelo inspetor.
- Eficiência: definida como a taxa entre defeitos reais identificados e o tempo de inspeção.
- Melhorias: quantidade de melhorias que podem ser realizadas no código e que foram identificadas pelos inspetores.

7.4.3 Projeto Experimental

O projeto experimental foi realizado com dois grupos de quatro participantes. Um grupo utilizou a técnica KIRT-Acoplamento com suporte da ferramenta CRISTA e outro grupo utilizou a técnica *Checklist* desenvolvida por Dunsmore *et al.* (2003b) sem suporte computacional. Cada participante realizou o experimento individualmente.

7.4.4 Instrumentação

Os instrumentos do experimento para o grupo KIRT-Acoplamento foram o software *Flight*, documento de requisitos, diagrama de classes, *checklist* das técnicas KIRT e a ferramenta CRISTA.

Para o grupo *Checklist* os instrumentos do experimento foram o software *Flight*, documento de requisitos, diagrama de classes, o *checklist* elaborado por Dunsmore *et al.* (2003b) e um formulário para registro de discrepâncias.

O sistema *Flight* é o mesmo utilizado em outros estudos experimentais (DUNSMORE; ROPER; WOOD, 2003b), corresponde ao agendamento de passagens áreas e foi desenvolvido na linguagem Java. O sistema é composto de sete classes Java e uma média de 93 linhas de código por classe. Existem 14 defeitos conhecidos no código do software *Flight* os quais estão relatados em Dunsmore *et al.* (2003b) e são apresentados a seguir:

- Classe Reservation.java
    - Defeito 1: método isID está utilizando uma comparação incorreta entre *strings*.
    - Defeito 2: método isID possui um condição incorreta na condição if.
    - Defeito 3: método isOutOfDate está utilizando um objeto incorreto.
    - Defeito 4: método getCost está com o cálculo do custo total incorreto.
    - Defeito 5: método cancel está passando um parâmetro incorreto.
    - Defeito 6: método cancel está faltando uma linha de código que permite a remoção de passageiros devido a um cancelamento o vôo.
- Classe reservationCollection.java
    - Defeito 7: está faltando a leitura do primeiro elemento no código reservations.listIterator().
    - Defeito 8: método removeConfirmedReservation nunca utiliza o parâmetro ID recebido.

- Defeito 9: método removeConfirmedReservation possui uma chamada de método incorreta.
- Defeito 10: método removeOldReservations não funciona corretamente quando elementos estão faltando.
- Defeito 11: método removeOldReservations possui uma chama de método incorreta.
- Defeito 12: método addReservation está com a ordem dos parâmetros incorreta.
- Defeito 13: método addReservation está realizando um incremento da variável dayIdentifier de forma incorreta.
- Defeito 14: método addReservation nunca adiciona reservas na lista de reservas.

7.4.5 Preparação e Execução do Experimento

O experimento foi realizado em três dias conforme apresentado na Tabela 7.10. Ressalta-se que os participantes foram divididos em dois grupos somente após o preenchimento do questionário de caracterização, permitindo a criação de grupos homogêneos.

Tabela 7.10. Atividades realizadas no Estudo Experimental III

	GRUPO 1	**GRUPO 2**
Dia 1	1. Treinamento na KIRT-Acoplamento em um sistema exemplo (40 minutos) 2. Discussão dos resultados obtidos e das dúvidas encontradas pelos participantes (30 minutos)	1. Treinamento no *Checklist* em um sistema exemplo (40 minutos) 2. Discussão dos resultados obtidos e das dúvidas encontradas pelos participantes (30 minutos)
Dia 2	1. Inspeção do *Flight* usando a KIRT-Acoplamento (90 minutos)	1. Inspeção do *Flight* usando o *Checklist* (90 minutos)
Dia 3	1. Discussão dos resultados (60 minutos)	

7.4.6 Análise e Discussão dos Resultados

Todas as discrepâncias registradas com a aplicação das duas técnicas foram analisadas a fim de classificá-las em defeitos, falso-positivos ou melhorias. A Figura 7.5 apresenta um resumo do número de discrepâncias, melhorias, falso-positivos e defeitos encontrados por cada participante em relação ao sistema *Flight*.

Observa-se na Figura 7.5 que nenhum inspetor identificou todos os 14 defeitos do *Flight*. Os inspetores que mais identificaram defeitos foram os inspetores 2 e 3, ambos utilizando a técnica KIRT-Acoplamento. Outro dado interessante diz respeito ao Inspetor 5, o qual teve um alto índice de identificação de falso-positivos (10 falso-positivos) e um baixo índice de identificação de defeitos (2 defeitos).

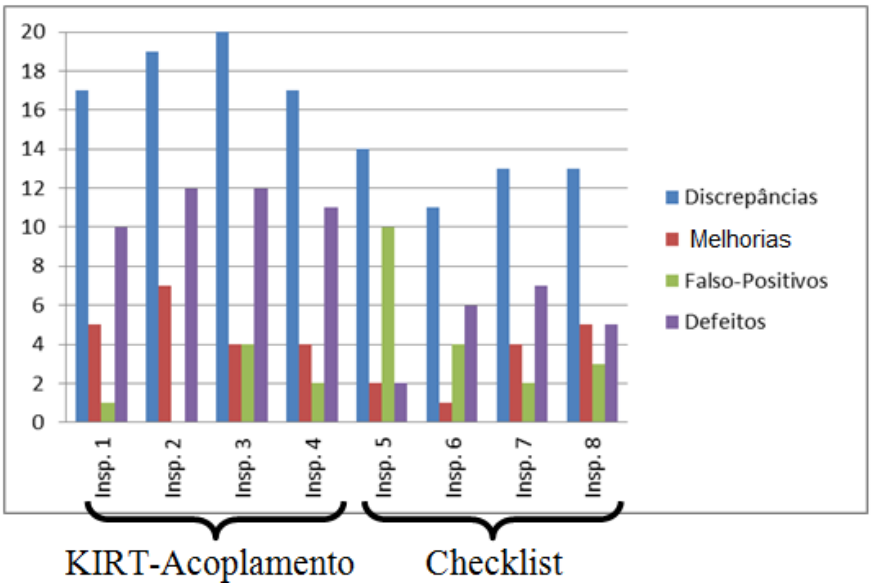


Figura 7.5. Resultados obtidos por cada participante na inspeção do sistema *Flight*

De acordo com a Tabela 7.11, ao analisar os valores médios obtidos por cada grupo verifica-se que o grupo KIRT-Acoplamento teve um melhor desempenho em relação ao número de defeitos identificados. A mesma análise é válida para os valores de discrepâncias e melhorias. Em relação ao número médio de falso-positivos identificados, a KIRT-Acoplamento também teve um melhor desempenho. Tais resultados, embora não comprovados estatisticamente, podem ser um indício de que o uso da KIRT-Acoplamento promove melhor desempenho do que a técnica Checklist.

Tabela 7.11. Valores médios dos participantes dos dois grupos

	Discrepâncias	**Melhorias**	**Falso-Positivos**	**Defeitos**	**Tempo**
KIRT-Acoplamento	18,25	5,00	1,75	11,25	68,75 min.
Checklist	12,75	3,00	4,00	5,00	57,25 min.

A Tabela 7.12 apresenta os valores das métricas *precision* e *recall* para os defeitos identificados pelos inspetores dos grupos KIRT-Acoplamento e Checklist.

Tabela 7.12. *Precision* e *Recall* para os inspetores

Grupo	Inspetores	*Precision*	*Recall*
KIRT-Acoplamento	Insp. 1	58,8 %	71,4 %
	Insp. 2	63,1 %	85,7 %
	Insp. 3	60,0 %	85,7 %
	Insp. 4	64,7 %	78,5 %
Checklist	Insp. 5	14,2 %	14,2 %
	Insp. 6	54,5 %	42,8 %
	Insp. 7	53,8 %	50,0 %
	Insp. 8	38,4 %	35,7 %

Observe que os valores de *precision* e *recall* dos inspetores do grupo KIRT-Acoplamento são melhores do que os valores do grupo Checklist. A Tabela 7.13 apresenta a média das métricas *precision* e *recall* dos inspetores de cada grupo.

Tabela 7.13. Média das métricas *precision* e *recall* para os dois grupos

Grupo	PRECISION	RECALL
KIRT-Acoplamento	61,6	80,3
Checklist	40,2	35,7

Com base nos valores obtidos pelas médias das métricas *precision* e *recall*, a métrica *F-measure* foi calculada (Baeza-Yates, 1999) para avaliar a efetividade de cada uma das técnicas (Delfim *et al.*, 2015). A fórmula de cálculo de *F-measure* é dada por:

$$F\text{-}measure = 2 * (precision * recall) / (precision + recall)$$

Os valores obtidos foram 69,7% e 37,8% para as técnicas de leitura KIRT-Acoplamento e Checklist, respectivamente, mostrando, portanto, uma maior efetividade da KIRT-Acoplamento.

No entanto, é interessante analisar quais foram os defeitos identificados e quais suas características, a fim de verificar a influência da técnica na identificação de cada defeito.

A Tabela 7.14 apresenta a classificação dos defeitos do *Flight* de acordo com a taxonomia de Dunsmore *et al.* (2003b) e a frequência que cada defeito foi identificado com o uso de cada técnica.

Tabela 7.14. Classificação dos defeitos e a frequência que cada defeito foi identificado no sistema *Flight*

INSPETORES	**D1**	**D2**	**D3**	**D4**	**D5**	**D6**	**D7**	**D8**	**D9**	**D10**	**D11**	**D12**	**D13**	**D14**
Objeto incorreto			X											
Método incorreto	X								X		X			
Parâmetro incorreto					X		X					X		
Computação		X		X						X			X	X
Fluxo de dados		X						X						
Erro de Especificação														
Omissão						X		X		X			X	X
Comissão	X	X	X	X	X		X		X		X	X		
Localidade	C	C	S	M	S	S	S	M	S	S	S	S	C	C
Tamanho	M	M	M	S	S	S	L	L	L	L	L	M	M	M
KIRT-Acoplamento (%)	**100**	**100**	**75**	**100**	**75**	**100**	**100**	**50**	**100**	**25**	**100**	**75**	**75**	**100**
Checklist (%)	**50**	**50**	**0**	**75**	**50**	**0**	**25**	**50**	**50**	**0**	**25**	**100**	**25**	**0**

Em termos numéricos, os participantes que usaram a KIRT-Acoplamento identificaram mais defeitos do que aqueles que usaram a técnica Checklist, com exceção do defeito D12. Para os defeitos D1, D2, D4, D7, D9 e D11 o uso da KIRT-Acoplamento mostrou um desempenho de 100%, ou seja, todos os participantes identificaram esses defeitos. Interessante observar que os defeitos D3, D6, D10 e D14 não foram identificados por nenhum participante que utilizou a técnica Checklist. Ressalta-se que os defeitos D3, D6 e D10 são defeitos de localidade Sistema e, portanto, associados à característica de delocalização.

Considerando a questão da localidade do defeito (Método, Classe e Sistema) as técnicas tiveram o comportamento apresentado na Figura 7.6. Observe que a KIRT-Acoplamento teve um melhor desempenho em relação a todas as localizações (Método, Classe e Sistema). Ressalta-se que a maior diferença ocorreu na localidade Classe. Além disso, os defeitos de Sistema também foram identificados com maior frequência com o uso da KIRT-Acoplamento. Tal resultado reforça, embora não comprovado estatisticamente, um melhor desempenho da KIRT-Acoplamento na identificação de defeitos com característica de delocalização.

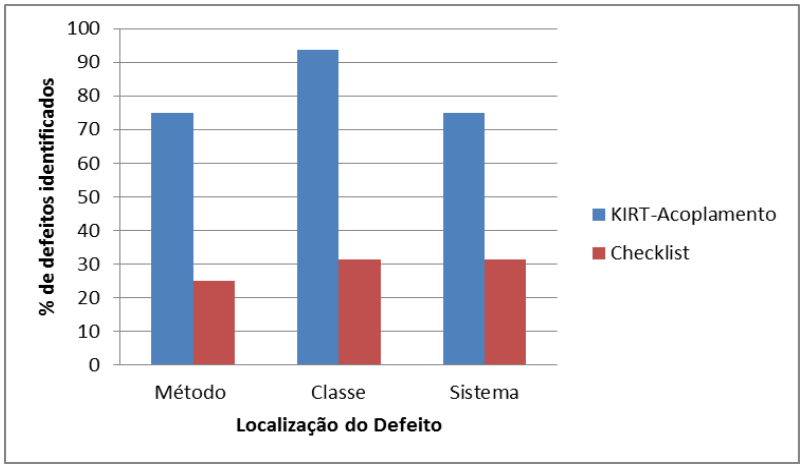


Figura 7.6. Porcentagem de defeitos identificados de acordo com a localização e técnica de leitura utilizada

A Figura 7.7 apresenta uma importante informação sobre o uso da KIRT-Acoplamento, principalmente em relação aos defeitos D3, D5, D6, D7, D9, D10, D11 e D12, que por serem defeitos associados à delocalização, são identificados mediante o entendimento de trechos de código espalhados por diversas classes. Para todos esses defeitos, a KIRT-Acoplamento promoveu melhor desempenho do que a técnica Checklist, com exceção do defeito D12, encontrado por 75% dos inspetores da KIRT-Acoplamento e 100% dos inspetores do Checklist.

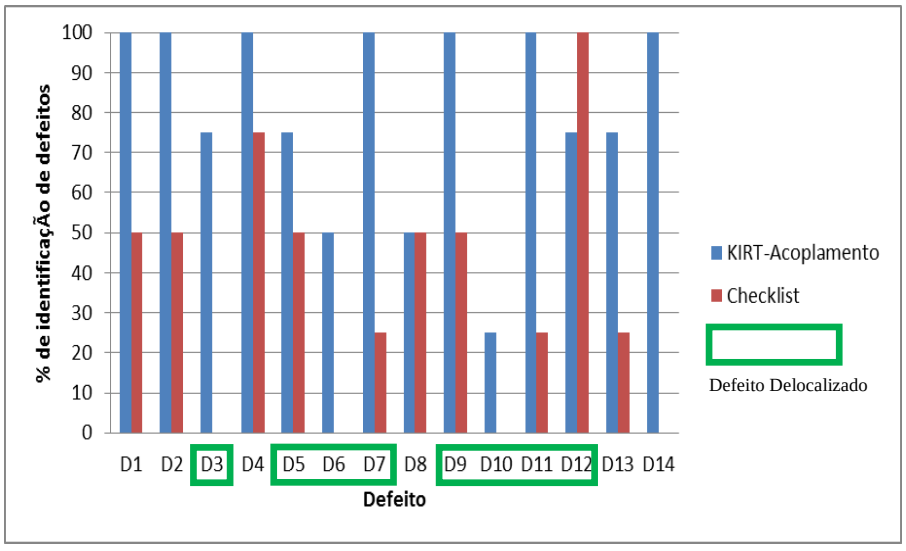


Figura 7.7. Porcentagem de identificação dos defeitos do *Flight*

Outra análise que merece atenção diz respeito ao tempo médio de aplicação das técnicas, conforme apresentado na Tabela 7.11. Para a KIRT-Acoplamento o tempo médio foi de 68 minutos e para o Checklist foi de 57 minutos. No entanto, realizando-se uma análise da eficiência de aplicação das técnicas (Tabela 7.15), isto é, a taxa entre o número de defeitos identificados e o tempo de aplicação da técnica, a eficiência média dos inspetores do grupo

KIRT-Acoplamento foi melhor que do grupo Checklist, apresentando os valores 16,4% e 8,6%, respectivamente.

Tabela 7.15. Eficiência dos inspetores em relação ao número de defeitos identificados

Grupo	Inspetores	Defeitos	Tempo (minutos)	Experiência Profissional (em meses)	Efetividade	Eficiência (defeitos/tempo)
KIRT-Acoplamento	Insp. 1	10	68	30	71,4 %	14,7 %
	Insp. 2	12	74	156	85,7 %	16,2 %
	Insp. 3	11	63	26	85,7 %	19,04 %
	Insp. 4	11	70	160	78,5 %	15,71 %
Checklist	Insp. 5	2	53	84	14,2 %	3,73 %
	Insp. 6	6	63	204	54,5 %	9,52 %
	Insp. 7	7	54	168	53,8 %	12,96 %
	Insp. 8	5	59	108	38,4 %	8,47 %

Os resultados apresentados na Tabela 7.15 referentes à experiência profissional fornecem indícios de que mesmo com a média de experiência profissional menor, a efetividade e eficiência dos participantes do grupo KIRT-Acoplamento tiveram melhores resultados. Em média, a experiência profissional dos inspetores dos grupos KIRT-Acoplamento e *Checklist* é de aproximadamente 93 meses e 141 meses, respectivamente. Inclusive, o inspetor 3 com menor experiência profissional teve uma melhor eficiência que todos os outros inspetores.

Todas as melhorias no código identificadas pela técnica Checklist também foram identificadas pela técnica KIRT-Acoplamento. Além disso, quatro melhorias foram identificadas apenas pela técnica KIRT-Acoplamento. Ressalta-se que não havia uma lista pré-conhecida de melhorias do código.

Além das análises apresentadas anteriormente foram realizados três testes de hipóteses relacionados às três variáveis dependentes – efetividade, eficiência e melhorias. Devido ao projeto experimental (dois grupos com 4 participantes em cada grupo) foi utilizado o teste estatístico não-paramétrico Mann-Whitney para duas amostras independentes.

A primeira hipótese diz respeito à efetividade de cada uma das técnicas: *Há diferença significativa entre a efetividade das técnicas de leitura KIRT-Acoplamento e Checklist.*

O p-valor resultante da aplicação do teste estatístico Mann-Whitney é 0,0105 que, sendo menor que 0,05, representa que a hipótese nula pode ser rejeitada. Dessa forma, existe diferença significativa entre a efetividade de cada técnica.

A segunda hipótese diz respeito à eficiência de cada uma das técnicas: *Há diferença significativa entre a eficiência das técnicas de leitura KIRT-Acoplamento e Checklist.*

O p-valor resultante da aplicação do teste estatístico Mann-Whitney também é 0,0105 que, sendo menor que 0,05, representa que a hipótese nula pode ser rejeitada. Dessa forma, existe diferença significativa entre a eficiência de cada técnica.

A terceira hipótese diz respeito às melhorias identificadas por cada uma das técnicas: *Há diferença significativa entre o número de melhorias identificadas pelas técnicas de leitura KIRT-Acoplamento e Checklist.*

O p-valor resultante da aplicação do teste estatístico Mann-Whitney é 0, 0970 que, não sendo menor que 0,05, representa que a hipótese nula não pode ser rejeitada. Dessa forma, não existe diferença significativa entre o número de melhorias identificadas por cada técnica.

7.4.7 Ameaças à Validade

As ameaças à validade do Estudo Experimental III são similares às do Estudo Experimental II.

A validade interna refere-se a fatores não controlados que podem influenciar nas variáveis dependentes, destacando:

- Participantes: nesse experimento foi utilizado profissionais da área de desenvolvimento de software, o que minimiza a influência do perfil associado com os participantes.
- Comunicação: a comunicação entre participantes do próprio grupo e entre grupos não era permitida.
- Fadiga: a fadiga por parte dos inspetores foi minimizada, pois a atividade de inspeção foi planejada para consumir, no máximo, 1 hora e meia de aplicação.
- Treinamento: uma possível ameaça estava relacionada à falta de familiaridade com o uso das técnicas. Essa ameaça foi minimizada com os treinamentos e os *feedbacks* fornecidos antes da execução do experimento.
- Conhecimento do domínio: apesar de nenhum dos participantes terem familiaridade com software desse domínio, essa ameaça foi minimizada, pois foi realizada uma apresentação das funcionalidades e todos os participantes já haviam interagido com um software dessa natureza em outras oportunidades.

A validade externa refere-se à capacidade de generalizar os resultados do experimento, destacando:

- Participantes: os participantes desse experimento eram profissionais da área de desenvolvimento de software, o que minimiza essa característica da validade externa.
- Sistema: o sistema *Flight* utilizado no estudo experimental, embora não desenvolvido profissionalmente, possui todas as características de um sistema orientado a objetos e já foi utilizado em outros estudos experimentais relatados na literatura.

A validade de conclusão está relacionada com as questões que afetam a interpretação de uma conclusão correta, destacando:

- Teste estatístico: mesmo com um número reduzido de participantes em cada grupo, foi possível aplicar o teste não-paramétrico (Mann-Whitnney) para avaliar a efetividade e eficiência na identificação de defeitos e os números de melhorias identificadas.

A validade de construção está relacionada ao grau com que as métricas utilizadas para avaliar os resultados do experimento (variáveis dependentes) de fato representam as medidas necessárias para caracterizar o objeto de estudo (as técnicas de leitura definidas), destacando:

- Métricas (efetividade, eficiência e número de melhorias): essas métricas têm sido utilizadas por vários pesquisadores em outros estudos experimentais com objetivos similares a este.

7.4.8 Conclusões do Estudo Experimental III

O Estudo Experimental III teve como objetivo avaliar o desempenho das técnicas KIRT-Acoplamento e Checklist proposta por Dunsmore *et al.* (2003b).

Os resultados obtidos apresentam indícios de um melhor desempenho na identificação de defeitos e melhorias com o uso da técnica KIRT-Acoplamento. Um importante resultado obtido desse experimento diz respeito a defeitos delocalizados. O desempenho da técnica KIRT-Acoplamento na identificação de defeitos delocalizados foi superior em relação a técnica Checklist proposto por Dunsmore *et al.* (2003b).

7.5 Considerações Finais

Com o intuito de avaliar o desempenho da técnicas de leitura KIRT e analisar como o uso das técnicas pode contribuir para a melhoria da qualidade de software, foram realizados três experimentos. Esses experimentos fizeram parte do Ciclo 4 da Pesquisa-Ação.

Os resultados e relatos obtidos após a execução do Estudo Experimental I apresentam indícios de que o uso da visualização dando suporte ao *checklist* das técnicas de leitura pode auxiliar na identificação de melhorias e de um menor número de falso-positivos. De acordo com entrevistas realizadas com os participantes, a metáfora visual auxilia na aplicação do *checklist* visto que o inspetor pode se concentrar em trechos de códigos específicos.

Em relação ao Estudo Experimental II, os resultados obtidos fornecem indícios de que a técnica de leitura KIRT-Acoplamento promove melhor efetividade e eficiência na identificação de defeitos do que a técnica KIRT-Coesão. No entanto, em relação às melhorias, a técnica KIRT-Coesão mostrou melhor desempenho do que a KIRT-Acoplamento. Assim, em princípio, pode-se dizer que essas técnicas são complementares.

Observa-se também que dependendo da métrica utilizada pela técnica, a inspeção fica mais concentrada nas unidades (KIRT-Coesão) ou na integração das classes (KIRT-Acoplamento). Dessa forma, com o estudo das demais técnicas aqui propostas, pretende-se estabelecer uma estratégia de inspeção em nível de unidade e integração, pois se entende que a partir desses resultados, a inspeção de código pode ser tratada em níveis diferentes, assim como a atividade de teste.

Uma das limitações do Estudo Experimental II estava relacionada ao uso de estudantes como participantes. Dessa forma, o Estudo Experimental III foi realizado com profissionais da área de desenvolvimento de software e tinha como objetivo avaliar os resultados obtidos com duas técnicas de leitura: KIRT-Acoplamento e Checklist.

Os resultados obtidos no Estudo Experimental III mostraram um melhor desempenho da técnica de leitura KIRT-Acoplamento em relação ao *Checklist* proposto por Dunsmore *et al.* (2003b), especialmente em relação à efetividade na identificação de defeitos delocalizados. Salienta-se que, no estudo de Dunsmore *et al.* (2003b), o *Checklist* teve melhor desempenho em relação a outras duas técnicas de inspeção.

Em suma, os três estudos experimentais apresentaram resultados satisfatórios e mostraram que a técnica de leitura KIRT auxilia, de fato, não apenas na identificação de defeitos, mas também, indiretamente, na identificação de melhorias, o que certamente pode levar na melhoria da qualidade do software.

Salienta-se que novos estudos experimentais precisam ser realizados, principalmente para explorar as demais instanciações da técnica KIRT de forma que resultados mais conclusivos possam ser traçados.

Capítulo 8

CONCLUSÃO

Este capítulo apresenta as conclusões, contribuições e limitações desta pesquisa, lições aprendidas, oportunidades identificadas e as publicações obtidas até o momento.

Neste trabalho definiu-se a técnica de leitura denominada KIRT para inspeção de código-fonte utilizando informações-chave, metáforas visuais e *checklist*. Os estudos experimentais realizados até o momento corroboram, preliminarmente, a tese de que técnica de leitura para inspeção de código baseada em visualização e informações-chave contribuem para a identificação de defeitos e consequentemente para a melhoria da qualidade geral do software.

O contexto real que motivou o desenvolvimento deste trabalho foi a evidência de que a atividade de inspeção de código de sistemas orientado a objetos tem o desafio de superar as questões específicas do paradigma como, por exemplo, a delocalização das funcionalidades de um sistema. Para auxiliar a atividade de inspeção, técnicas de leitura podem ser aplicadas. No entanto, poucas técnicas de leitura sistematizadas foram identificadas na literatura e sabe-se que quanto menos formal é a técnica, mais dependente da experiência do inspetor fica a atividade de inspeção. Outro ponto de destaque é que uma das técnicas de leitura mais sistemáticas é a *Stepwise Abstraction*, a qual não foi desenvolvida considerando as características do paradigma orientado a objetos, especialmente questões relacionadas à delocalização das funcionalidades de um sistema. Além disso, a *Stepwise Abstraction* apesar de ser sistemática e efetiva, muitas vezes promove certa improdutividade, pois exige que até partes mais simples do código sejam abstraídas e documentadas.

Assim, considerando o objetivo da pesquisa e que a solução dependia de um processo investigativo incremental, optou-se pela adoção de uma estratégia evolutiva para o desenvolvimento do trabalho. Dessa forma, utilizou-se como metodologia de trabalho uma abordagem composta de quatro ciclos, apresentados nos capítulos desta tese.

No Ciclo 1 foi apresentado um estudo experimental que havia sido realizado por Porto (2009), mas que os resultados ainda não haviam sido analisados. O principal objetivo do estudo experimental foi avaliar o uso da técnica *Stepwise Abstraction* com o suporte de visualização implementado na ferramenta CRISTA, em relação à técnica *ad-hoc* para inspeção de código-fonte. Os resultados obtidos mostraram que a visualização mais as abstrações de código por meio da técnica *Stepwise Abstraction* são efetivas para a identificação de defeitos, uma vez que a *Stepwise Abstraction* exige que todo trecho de código seja abstraído e documentado. Isso levou os inspetores a relatarem discrepâncias que não se tratavam apenas de defeitos propriamente, mas também de melhorias que poderiam ser realizadas no código-fonte. Dessa forma, os resultados do experimento forneceram evidências de que uma técnica de leitura com o suporte de visualização pode auxiliar não só na identificação de defeitos, mas também de melhorias no código.

No Ciclo 2 foi realizado um mapeamento sistemático da literatura com o objetivo de identificar técnicas de leitura existentes para a inspeção de código. Dessa revisão foram identificadas nove técnicas de leitura diferentes, as quais foram experimentalmente avaliadas por seus autores, sendo que na maioria das vezes a técnica *checklist* foi utilizada como técnica de controle. Essas técnicas foram avaliadas quanto às suas vantagens e desvantagens e quanto à maneira com que elas eram utilizadas. Observou-se que algumas técnicas utilizavam informações que foram consideradas relevantes neste trabalho e denominadas informações-chave, a saber, documento de requisitos, diagrama de classes e métricas de código. Assim, em relação à atividade de inspeção, a revisão da literatura evidenciou que informações-chave, juntamente com o cenário operacional da técnica PBR e o *Checklist* dividido em seções, poderiam ser uma boa alternativa para a definição das técnicas de leitura pretendidas. Nessa mesma revisão da literatura, extraíram-se informações sobre o uso de visualização para código-fonte identificando metáforas visuais e ferramentas computacionais para o seu suporte. Assim, em relação às metáforas visuais, a revisão da literatura mostrou que a metáfora utilizada na ferramenta CRISTA (*treemap*) estava apropriada para a representação do código-fonte e que o grafo seria uma alternativa para representar os relacionamentos entre as classes.

No Ciclo 3 foi realizado um estudo exploratório, por meio de um experimento, com o objetivo de identificar padrões comportamentais dos inspetores durante uma atividade de inspeção de código. Nesse estudo, utilizaram-se as informações-chave que foram identificadas durante a revisão da literatura conduzida no Ciclo 2. A suposição era de que os padrões comportamentais dos inspetores poderiam ser influenciados pelo uso dessas informações-chave durante a inspeção de código, promovendo ou não melhor desempenho. Os resultados

obtidos desse estudo exploratório foram evidências relacionadas à homogeneidade do comportamento dos inspetores que utilizaram as informações-chave. Além disso, um maior número de defeitos e de melhorias e um menor número de falso-positivos foram identificados por inspetores que utilizaram as informações-chave.

O Ciclo 4 foi subdividido em dois capítulos: no Capítulo 6 foi apresentada a técnica de leitura KIRT e um exemplo de aplicação usando a ferramenta CRISTA; no Capítulo 7 foram apresentados três estudos experimentais que tiveram como objetivo avaliar o desempenho da técnica KIRT sob diferentes pontos de vista.

Em relação à técnica de leitura KIRT, ela foi inspirada no cenário operacional da PBR e pode ser instanciada mediante o uso de uma métrica. Para a definição do cenário operacional considerou-se como modelo subjacente a abstração de código realizada por meio das metáforas visuais presentes na CRISTA, a taxonomia de defeitos definida por Dunsmore *et al.* (2003b) e os procedimentos utilizando o *checklist* e as informações-chave.

Em relação aos estudos experimentais realizados foram observados indícios de que o uso da técnica de leitura KIRT promove a efetividade e eficiência na identificação de defeitos. Além disso, verificou-se um aspecto de complementariedade entre duas instanciações da técnica de leitura KIRT (KIRT-Acoplamento e KIRT-Coesão) o que pode sugerir a criação de uma estratégia de inspeção utilizando mais de uma instância diferente de técnica de leitura. Observou-se também que a sistematização da técnica KIRT proporcionou melhor desempenho na identificação de melhorias em relação à técnica *checklist* proposta por Dunsmore *et al.* (2003b).

Sendo assim, com a condução desses estudos, considera-se ter obtido evidências que a utilização de uma técnica de leitura de inspeção de código auxiliada por informações-chave e visualização contribui para a melhoria da qualidade geral do software desenvolvido, especialmente na identificação de defeitos.

8.1 Contribuições e limitações da pesquisa

Embora existam outras técnicas de leitura para inspeção de código, conforme apresentado na revisão da literatura desta tese, as contribuições desta pesquisa podem ser caracterizadas para os seguintes pontos:

1) Resultados de um mapeamento sistemático sobre técnicas de inspeção de código que identificou a existência de nove técnicas distintas.

2) Levantamento de evidências sobre como inspetores utilizam documento de requisitos, diagrama de classes e métricas de código durante o processo de inspeção de software. Essas evidências foram caracterizadas como informações-chave para a definição da técnica KIRT.
3) Definição da técnica KIRT (*Key-Information Reading Technique*), a qual utiliza as informações-chave (documento de requisitos, diagrama de classes e métricas de software) além de metáforas visuais associadas ao *checklist* para facilitar sua aplicação, melhorando o desempenho e, consequentemente, a qualidade do software desenvolvido.
4) Utilização de metáforas visuais *treemap* e grafo para inspeção do código-fonte de modo a promover maior facilidade na aplicação da técnica KIRT.
5) Evolução da ferramenta CRISTA com o desenvolvimento de novas funcionalidades que permitem a aplicação da técnica KIRT, melhorando a eficiência e efetividade na atividade de inspeção.
6) A possibilidade de instanciação de novas técnicas de leitura a partir do uso de outras métricas (exemplo: classes mais modificadas de um sistema de versões, classes mais utilizadas, etc), outros checklist e até mesmo utilizando outras metáforas visuais. Essa característica é possível por meio da instanciação de técnicas pelos cenários operacionais da técnica KIRT.

Apesar das contribuições destacadas, limitações são inerentes. Assim, as seguintes limitações são elencadas:

1) Apenas um estudo experimental foi realizado com profissionais da área de desenvolvimento de software. Além disso, os softwares utilizados não são softwares construídos pela indústria. Essas características implicam em ameaças à validade dos resultados conforme relatado nos estudos experimentais realizados;
2) O *Paint* usado para as evidências não possui um porte de um sistema profissional, embora contemplasse todas as características de OO. Mesmo assim, algumas características podem não ter sido contempladas, de modo que novos estudos são necessários com outros portes de sistemas.
3) Atualmente, a técnica KIRT é composta de sete possíveis instanciações com o uso de sete diferentes métricas de software, mas apenas duas instanciações da técnica foram avaliadas experimentalmente neste trabalho. Portanto, novos

estudos experimentais devem ser planejados para a avaliação das demais instanciações da técnica.

4) Outras metáforas visuais poderiam ser exploradas com o objetivo de facilitar ainda mais o processo de inspeção. Tais metáforas podem ser exploradas tanto no contexto da representação do código (como a *treemap* da CRISTA) quanto no contexto da representação das métricas associadas ao código, como CodeCity (WETTEL; LANZA; ROBBES, 2011) e Visão Polimétrica (LANZA e DUCASSE, 2003).
5) A ferramenta CRISTA, que tornou viável a validação da tese, ainda é um protótipo. Suas funcionalidades não foram testadas com muitos usuários e em outros contextos diferentes dos apresentados nos estudos experimentais;
6) Não foram utilizadas técnicas apropriadas para identificação de melhorias. As melhorias relatadas com o suporte da técnica KIRT são consequência da aplicação de seus passos com o objetivo de detecção de defeitos.
7) A técnica KIRT implementada e instanciada na ferramenta CRISTA está associada com a inspeção de código orientado a objetos. No entanto, a partir do cenário operacional é possível instanciar a técnica KIRT para outros contextos, como sistemas concorrentes, orientado a aspectos, etc. Para tanto, é necessário definir as métricas específicas de cada contexto, o checklist utilizado e o modelo subjacente utilizado.

8.2 Lições aprendidas

Considerando que o desenvolvimento da técnica de leitura KIRT necessitava da exploração de como inspetores de código utilizavam informações-chave para guia-los, o uso de uma metodologia de ciclos evolutivos foi uma decisão acertada visto que se tratava de uma pesquisa exploratória e evolutiva. A cada execução de um ciclo, informações importantes eram evidenciadas e podiam ser processadas gradativamente, contribuindo com uma evolução natural da pesquisa, permitindo o planejamento das etapas futuras.

Outra lição aprendida relaciona-se com a condução de estudos experimentais no contexto de pesquisas acadêmicas. Em primeiro lugar, como já era sabido, existe uma grande dificuldade em conseguir participantes para os estudos experimentais. Assim, como mencionado por Carver *et al.* (2003), o uso de estudantes é uma alternativa para a realização

de estudos experimentais envolvendo novas tecnologias. Isso, de fato, não desmerece os resultados do estudo, uma vez que, no contexto deste trabalho, quando se conduziu um estudo com profissionais da área, observou-se que o comprometimento dos participantes acadêmicos foi parecido com o desses profissionais.

8.3 Oportunidades futuras

A seguir, apresentam-se algumas das ideias que podem dar continuidade a este trabalho:

- Condução de outros estudos experimentais com as técnicas KIRT, especialmente com profissionais da área de desenvolvimento de software e utilizando sistemas reais;
- Condução de estudos experimentais para avaliar outras instanciações da técnica de leitura KIRT que não foram avaliadas no contexto deste trabalho;
- Análise das melhorias identificadas nos experimentos como indicativo de associação a possíveis dívidas técnicas, que são definidas como uma metáfora para artefatos de software imaturos, incompletos ou inadequados que causam altos custos de manutenção e que diminuem a qualidade do software durante sua evolução (SEAMAN; GUO, 2011).
- Avaliação da usabilidade da ferramenta CRISTA, incorporando sugestões de melhorias relatadas pelos participantes dos experimentos;
- Definição de estratégias de inspeção considerando atividades de inspeção em níveis de unidade e integração;
- Implementação de funcionalidade na ferramenta CRISTA que permita o cadastro de *checklists*;
- Exploração de outras metáforas visuais que possam contribuir para a aplicação da técnica de leitura KIRT;
- Integração da ferramenta CRISTA com a ferramenta COCAR, desenvolvida no grupo de pesquisa, de forma a permitir a rastreabilidade entre defeitos no código e os respectivos requisitos associados ao código;
- Integração da ferramenta CRISTA com a ferramenta Insight, desenvolvida no grupo de pesquisa, de forma a permitir a análise dos defeitos relatados por meio de técnicas de análise qualitativa.

8.4 Publicações

Nesta seção são apresentadas as publicações do autor desta pesquisa durante o período de doutorado, explicitando sua contribuição nas publicações. As publicações estão organizadas em duas subseções: capítulos de livros e artigos completos publicados em anais de congressos (*full papers*).

8.4.1 Capítulos de livros publicados

1) *Hernandes, E. C. M.; Belgamo, A.; Fabbri, S. C. P. F. An overview of experimental studies on software inspection process. In: José Cordeiro, Leszek A. Maciaszek, Joaquim Filipe. (Org.). Lecture Notes in Business Information Processing. 1ed.Berlin: Springer, 2014*

   Essa publicação, que é uma extensão de uma publicação premiada como melhor artigo em um congresso, foi fruto de parte da revisão bibliográfica desta pesquisa.

2) *Fabbri, S. C. P. F.; Hernandes, E. C. M.; Di Thommazo, A.; Belgamo, A.; Zamboni, A.; Silva, C. Using information visualization and text mining to facilitate the conduction of systematic literature reviews. In: José Cordeiro, Leszek A. Maciaszek, Joaquim Filipe. (Org.). Lecture Notes in Business Information Processing. 1ed. Berlin: Springer, 2013, v. 141, p. 1-12.*

   Essa publicação, que é uma extensão de uma publicação premiada como melhor artigo em um congresso, foi fruto do envolvimento do autor desta pesquisa com o projeto StArt.

8.4.2 Trabalhos completos publicados em anais de congressos

1) *Di Thommazo, A.; Camargo, K.; Hernandes, E.C.M.; Belgamo, A.; Gonçalves, G.; Pedro, J.; Fabbri, S. . Using the Dependence Level Among Requirements to Prioritize the Regression Testing set and Characterize the Complexity of Requirements Change. In: 17th International Conference on Enterprise Information System, 2015, Barcelona.*

   Essa publicação foi fruto do envolvimento do autor com a ferramenta COCAR.

2) *Belgamo, A.; Hernandes, E. C. M.; Zamboni, A., Fabbri, S. C. P. F.. Code Inspection Supported by Stepwise Abstraction and Visualization: An Experimental Study. In: 16th International Conference on Enterprise Information Systems, 2014, Lisboa. Proceedings of the 16th International Conference on Enterprise Information Systems. v. 1. p. 39-48.*

   Essa publicação resultou do projeto desta pesquisa.

3) *Hernandes, E. C. M.; Belgamo, A.; Fabbri, S. C. P. F.. Experimental Studies in Software Inspection Process - A Systematic Mapping. In: 15th International Conference on Enterprise Information Systems, 2013, Angers. Proceedings of the 15th International Conference on Enterprise Information Systems. v. 1. p. 66-79.*

   Essa publicação foi fruto de parte da revisão bibliográfica realizada pelo autor desta tese. Esse trabalho recebeu o prêmio de Melhor Artigo de Estudante (*Best Student Paper*).

4) *Fabbri, S. C. P. F. ; Hernandes, E. C. M. ; Thommazo, A. ; Belgamo, Anderson ; Zamboni, A. ; Silva, Cleiton . Managing literature reviews information through visualization. In: International Conference on Enterprise Information Systems, 2012, Wroclaw. International Conference on Enterprise Information Systems, 2012. v. 1. p. 1-10.*

   Essa publicação foi fruto do envolvimento do autor desta tese com o projeto StArt. Esse trabalho recebeu o prêmio de Melhor Artigo (*Best Paper*).

REFERÊNCIAS

AURUM, A.; PETERSSON, H.; WOHLIN, C. State-of-the-art: Software inspections after 25 years. **Software Testing Verification and Reliability**, v. 12, n. February, p. 133–154, 2002.

BACCHELLI, A.; BIRD, C. **Expectations, Outcomes, and Challenges of Modern Code Review**International Conference on Software Engineering. **Anais**...Piscataway, NJ, USA: IEEE Press, 2013

BAEZA-YATES, R.; RIBEIRO-NETO, B. Modern information retrieval. **New York**, v. 9, p. 513, 1999.

BASILI, V. et al. The Empirical Investigation of Perspective-Based Reading. **Empirical Software Engineering**, v. 1, p. 133–164, 1996.

BASILI, V. R.; CALDIERA, G.; ROMBACH, H. D. The goal question metric approach. **Encyclopedia of Software Engineering**, v. 2, p. 528–532, 1994.

BELGAMO, A. **GUCCRA: Técnicas de Leitura para Construção de Modelos de Casos de Uso e Análise do Documento de Requisitos**. Universidade Federal de São Carlos, 2004.

BELGAMO, A.; FABBRI, S.; MALDONADO, J. C. **TUCCA: improving the effectiveness of use case construction and requirement analysis** International Symposium on Empirical Software Engineering. **Anais**...2005

BELLI, F.; CRISAN, R. Towards automation of checklist-based code-reviews. **Proceedings of ISSRE '96: 7th International Symposium on Software Reliability Engineering**, p. 24–33, 1996.

CARD, S. K.; MACKINLAY, J. D.; SHNEIDERMAN, B. Readings in Information Visualization: Using Vision to Think. In: **Information Display**. 1998.

CARNEIRO, G. F. et al. **An Eclipse-Based Visualization Tool for Software Comprehension**XVII Brazilian Symposium on Software Engineering. **Anais**...2008

CARVER, J. et al. **Issues in using students in empirical studies in software engineering education**Software Metrics Symposium, 2003. Proceedings. Ninth International. **Anais**...2003

CASERTA, P.; ZENDRA, O. Visualization of the Static Aspects of Software: A Survey. **Computer**, v. 17, n. 7, p. 913–933, 2011.

CHERNAK, Y. A statistical approach to the inspection checklist formal synthesis and improvement. **IEEE Transactions on Software Engineering**, v. 22, p. 866–874, 1996.

CHIDAMBER, S. R.; KEMERER, C. F. A metrics suite for object oriented design. **IEEE Transactions on Software Engineering**, v. 20, n. 6, p. 476–493, 1994.

COUNSELL, S.; SWIFT, S.; TUCKER, A. **Object-oriented cohesion as a surrogate of software comprehension: an empirical study**Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'05). **Anais**...2005

DAVISON, R. M.; MARTINSONS, M. G.; KOCK, N. Principles of canonical action

research. **Information Systems Journal**, v. 14, n. 1, p. 65–86, 2004.

DE ALMEIDA JR., J. R. et al. **Best practices in code inspection for safety-critical softwareSoftware, IEEE**, 2003.

DE ALMEIDA, J. R. et al. Best practices in code inspection for safety-critical software. **IEEE Software**, v. 20, p. 56–63+4, 2003.

DELFIM, F. M. et al. **Visual Approach for Change Impact Analysis: A Controlled ExperimentInformation Technology - New Generations (ITNG), 2015 12th International Conference on**, 2015.

DENGER, C.; KOLB, R. Testing and inspecting reusable product line components. **Proceedings of the 2006 ACM/IEEE International symposium on empirical software engineering - ISESE '06**, v. 2006, p. 184, 2006.

DUNSMORE, A.; ROPER, M.; WOOD, M. The role of comprehension in software inspection. **Journal of Systems and Software**, v. 52, n. 2–3, p. 121–129, 2000a.

DUNSMORE, A.; ROPER, M.; WOOD, M. **Object-oriented inspection in the face of delocalisation**Proceedings of the 2000 International Conference on Software Engineering. ICSE 2000 the New Millennium. **Anais**...2000b

DUNSMORE, A.; ROPER, M.; WOOD, M. The development and evaluation of three diverse techniques for object-oriented code inspection. **IEEE Transactions on Software Engineering**, v. 29, n. 8, p. 677–686, 2003a.

DUNSMORE, A.; ROPER, M.; WOOD, M. Practical code inspection techniques for object-oriented systems: an experimental comparison. **Software, IEEE**, v. 20, n. 4, p. 21–29, 2003b.

FABBRI, S. et al. **Managing Literature reviews information through visualization**ICEIS 2012 - Proceedings of the 14th International Conference on Enterprise Information Systems. **Anais**...2012

FAGAN, M. E. Advances in software inspections. **IEEE Transactions on Software Engineering**, v. SE-12, n. 7, p. 744–751, 1986.

FAGAN, M. E. Design and code inspections to reduce errors in program development. **IBM Systems Journal**, v. 38, n. 2.3, p. 258–287, 1999.

FISHER, M. S.; CUKIC, B. Automating techniques for inspecting high assurance systems. **Proceedings Sixth IEEE International Symposium on High Assurance Systems Engineering. Special Topic: Impact of Networking**, p. 117 – 26, 2001.

GERSHON, N.; EICK, S. G.; CARD, S. Information Visualization. **interactions**, v. 5, n. 2, p. 9–15, 1998.

GILB, T.; GRAHAM, D. Software inspection. **Journal of Software Maintenance: Research and Practice**, v. 7, n. 3, p. 221–222, 1995.

GUO, Y.; SPÍNOLA, R.; SEAMAN, C. Exploring the costs of technical debt management – a case study. **Empirical Software Engineering**, p. 1–24, 2014.

HATTON, L. Testing the value of checklists in code inspections. **IEEE Software**, v. 25, n. 4,

p. 82–88, 2008.

HAVELUND, K.; HOLZMANN, G. J. Software certification - coding, code, and coders. **2011 Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT)**, p. 205–210, 2011.

HERNANDES, E. M.; BELGAMO, A.; FABBRI, S. **Experimental Studies in Software Inspection Process - A Systematic Mapping**Proceedings of the 15th International Conference on Enterprise Information Systems. **Anais**...SciTePress - Science and and Technology Publications, 2013

HÖST, M.; JOHANSSON, C. Evaluation of code review methods through interviews and experimentation. **Journal of Systems and Software**, v. 52, n. 2-3, p. 113–120, 2000.

HUMPHREY, W. S. **A Discipline for Software Engineering**. 1995.

JAKOBSEN, M. R.; HORNBAK, K. Evaluating a fisheye view of source code. **Proceedings of the SIGCHI conference on**, p. 377–386, 2006.

JOHNSON, B.; SHNEIDERMAN, B. Tree-maps: a space-filling approach to the visualization of hierarchical information structures. **Proceeding Visualization '91**, p. 284–291, 1991.

JONES, C. Measuring defect potentials and defect removal efficiency. **CrossTalk**, v. 21, n. 6, p. 11–13, 2008.

JUN-SUK OH; HO-JIN CHOI. **A reflective practice of automated and manual code reviews for a studio project**Fourth Annual ACIS International Conference on Computer and Information Science (ICIS'05). **Anais**...2005

KALINOWSKI, M.; TRAVASSOS, G. H. **ISPIS: From Conception towards Industry Readiness**Proc. Int. Conf. of the Chilean Society of Computer Science (SCCC). **Anais**...2007

KELLY, D.; SHEPARD, T. Task-directed Software Inspection. **Journal of Systems and Software**, v. 73, n. 2, p. 361–368, 2004.

KNIGHT, C.; MUNRO, M. Visualising software - a key research area. **IEEE International Conference on Software Visualization**, p. 1 – 6, 1999.

KO, A. J. et al. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. **IEEE Transactions on Software Engineering**, v. 32, n. 12, p. 971–987, 2006.

KRUCHTEN, P.; NORD, R. L.; OZKAYA, I. Technical Debt: From Metaphor to Theory and Practice. **IEEE Software**, v. 29, n. 6, p. 18–21, 2012.

LAITENBERGER, O.; DEBAUD, J.-M. Perspective-based reading of code documents at Robert Bosch GmbH. **Information and Software Technology**, v. 39, n. 11, p. 781–791, 1997.

LAITENBERGER, O.; EMAM, K. EL; HARBICH, T. G. An Internally Replicated Quasi-Experimental Comparison of Checklist and Perspective-Based Reading of Code Documents. **IEEE Transactions on Software Engineering**, v. 27, n. 5, p. 387, 2001.

LAMPING, J.; RAO, R.; PIROLLI, P. A focus+context technique based on hyperbolic geometry for visualizing large hierarchies. **Proceedings of the SIGCHI conference on Human factors in computing systems - CHI '95**, p. 401–408, 1995.

LANUBILE, F.; MALLARDO, T. **An empirical study of Web-based inspection meetings**2003 International Symposium on Empirical Software Engineering, 2003. ISESE 2003. Proceedings. **Anais**...2003

LANZA, M.; DUCASSE, S. Polymetric views - A lightweight visual approach to reverse engineering. **IEEE Transactions on Software Engineering**, v. 29, n. 9, p. 782–795, 2003.

LENGLER, R.; EPPLER, M. J. Towards a Periodic Table of Visualization Methods for Management. **2007 IASTED International Conference on Graphics and Visualization in Engineering, GVE 2007**, p. 83–88, 2007.

LETOVSKY, S. Cognitive processes in program comprehension. **Journal of Systems and Software**, v. 7, p. 325–339, 1987.

LI, X. **A Comparison-based Approach for Software Inspection**Proceedings of the 1995 Conference of the Centre for Advanced Studies on Collaborative Research. **Anais**...: CASCON '95.IBM Press, 1995

LINGER, R. C.; MILLS, H. D.; WITT, B. I. **Structured Programming: Theory and Practice**. Addison-Wesley, 1979.

MÄNTYLÄ, M. V.; LASSENIUS, C. What types of defects are really discovered in code reviews? **IEEE Transactions on Software Engineering**, v. 35, n. 3, p. 430–448, 2009.

MARUCCI, R. et al. **OORTs/ProDeS: Definição de Técnicas de Leitura para um Processo de Software Orientado a Objetos**I Simpósio Brasileiro de Qualidade de Software. **Anais**...2002

MCMEEKIN, D. A. et al. **Checklist inspections and modifications: Applying Bloom's taxonomy to categorise developer comprehension**The 16th IEEE International Conference on Program Comprehension. **Anais**...2008

MORALES, R.; MCINTOSH, S.; KHOMH, F. **Do code review practices impact design quality? A case study of the Qt, VTK, and ITK projects**2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER). **Anais**...2015

MYERS, G. **The Art of Software Testing, Second edition**. 2004.

NASCIMENTO, H. DO; FERREIRA, C. Visualização de Informações–uma abordagem prática. **XXV Congresso da Sociedade Brasileira de Computação**, n. 62, p. 1262–1312, 2005.

NISHIZONO, K. et al. **Source code comprehension strategies and metrics to predict comprehension effort in software maintenance and evolution tasks - an empirical study with industry practitioners**IEEE International Conference on Software Maintenance (ICSM). **Anais**...2011

OLAGUE, H. M. et al. Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly Iterative or agile software development processes. **IEEE Transactions on Software Engineering**, v. 33, n. 6, p. 402–419, 2007.

OLIVEIRA, F.; GOLDMAN, A.; SANTOS, V. **Managing Technical Debt in Software Projects Using Scrum: An Action ResearchAgile Conference (AGILE), 2015**, 2015.

PETERSEN, K. et al. Systematic mapping studies in software engineering. **EASE'08 Proceedings of the 12th international conference on Evaluation and Assessment in Software**

Engineering, p. 68–77, 2008.

PORTER, A. et al. **Understanding the sources of variation in software inspectionsACM Transactions on Software Engineering and Methodology**, 1998.

PORTO, D. **CRISTA: Um Apoio Computacional para Atividades de Inspeção e Compreensão de Código**. Universidade Federal de São Carlos, 2009.

PORTO, D.; MENDONÇA, M.; FABBRI, S. **CRISTA: A tool to support code comprehension based on visualization and reading technique**IEEE 17th International Conference on Program Comprehension. **Anais**...2009

PRESSMAN, R. **Engenharia de Software**. Addison-Wesley, 2010.

RILLING, J.; SEFFAH, A.; BOUTHLIER, C. **The CONCEPT project - applying source code analysis to reduce information complexity of static and dynamic visualization techniquesVisualizing Software for Understanding and Analysis, 2002. Proceedings. First International Workshop on**, 2002.

ROBILLARD, M. P.; COELHO, W.; MURPHY, G. C. **How effective developers investigate source code: an exploratory studySoftware Engineering, IEEE Transactions on**, 2004.

RUSSELL, G. W. **Experience with inspection in ultralarge-scale developmentSoftware, IEEE**, 1991.

SANTOS, P. S. M. DOS; TRAVASSOS, G. H. **Action Research Use in Software Engineering: An Initial Survey**Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement. **Anais**...2009

SAUER, C. et al. The effectiveness of software development technical reviews: a\nbehaviorally motivated program of research. **IEEE Transactions on Software Engineering**, v. 26, n. 1, p. 1–14, 2000.

SEAMAN, C.; GUO, Y. Measuring and Monitoring Technical Debt. In: **Advances in Computers**. v. 82p. 25–46.

SHULL, F. et al. Replicated studies: building a body of knowledge about software reading techniques. **Software Engineering And Knowledge Engineering**, p. 39, 2003.

SIY, H.; VOTTA, L. Does the modern code inspection have value? **IEEE International Conference on Software Maintenance, ICSM**, p. 281–289, 2001.

SJØBERG, D. I. K.; DYBÅ, T.; JØRGENSEN, M. The Future of Empirical Methods in Software Engineering Research. **Future of Software Engineering**, v. SE-13, n. 1325, p. 358–378, 2007.

SPINELLIS, D. **Tool writing: a forgotten art? (software tools)Software, IEEE**, 2005.

SRIVASTAVA, S.; KUMAR, R. Indirect method to measure software quality using CK-OO suite. **2013 International Conference on Intelligent Systems and Signal Processing (ISSP)**, p. 47–51, 2013.

SUBRAMANYAM, R.; KRISHNAN, M. S. Empirical analysis of CK metrics for object-oriented design complexity: Implications for software defects. **IEEE Transactions on Software**

Engineering, v. 29, n. 4, p. 297–310, 2003.

TAO, Y. et al. How Do Software Engineers Understand Code Changes?: An Exploratory Study in Industry. **Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering**, p. 51:1–51:11, 2012.

TRAVASSOS, G. H. et al. **Reading Techniques for OO Design Inspections**. [s.l: s.n.].

TUKEY, J. W. Exploratory Data Analysis. **Analysis**, v. 2, n. 1999, p. 688, 1977.

VINZ, B. L.; ETZKORN, L. H. **A Synergistic Approach to Program Comprehension**14th IEEE International Conference on Program Comprehension (ICPC'06). **Anais**...2006

VON MAYRHAUSER, A.; VANS, A. M. Program understanding behavior during adaptation of large scale software. In: **6th International Workshop on Program Comprehension**. [s.l: s.n.]. p. 164–172.

WETTEL, R.; LANZA, M. **Program Comprehension through Software Habitability**15th IEEE International Conference on Program Comprehension (ICPC'07). **Anais**...2007

WETTEL, R.; LANZA, M.; ROBBES, R. Software systems as cities: a controlled experiment. **2011 33rd International Conference on Software Engineering (ICSE)**, p. 551–560, 2011.

WHEELER, D. A.; BRYKCZYNSKI, B.; MEESON JR., R. N. (EDS.). **Software Inspection: An Industry Best Practice for Defect Detection and Removal**. 1st. ed. Los Alamitos, CA, USA: IEEE Computer Society Press, 1996.

WIERINGA, R. et al. Requirements engineering paper classification and evaluation criteria: A proposal and a discussion. **Requirements Engineering**, v. 11, n. 1, p. 102–107, 2006.

WILKERSON, J. W.; NUNAMAKER, J. F.; MERCER, R. Comparing the defect reduction benefits of code inspection and test-driven development. **IEEE Transactions on Software Engineering**, v. 38, n. 3, p. 547–560, 2012.

WOHLIN, C. et al. **Experimentation in software engineering: an introduction**. 2000.

APÊNDICE A

CHECKLIST DA TÉCNICA KIRT

Checklist de Inspeção de Código

Projeto:			
Inspetor:		Data :	

DEFEITO				
	I - Variável	Sim	Não	N/A
1.	Os nomes de variáveis e de constantes são usados de acordo com as convenções de nome?			
2.	Toda variável está corretamente tipada?			
3.	Toda variável está apropriadamente inicializada?			
4.	Existem variáveis que deveriam ser constantes?			
5.	Existem atributos que deveriam ser variáveis locais?			
6.	Todos os atributos possuem modificadores de acesso apropriados (private, protected, public)?			
7.	Existem atributos estáticos que deveriam ser não estáticos ou vice-versa?			
8.	Há variáveis que possuem nomes similares que se confundem?			
9.	Todas variáveis estão apropriadamente definidas com nomes significantes, consistentes e claros?			
10.	Existem variáveis ou atributos redundantes ou não utilizados?			
	II – Definição de Método	Sim	Não	N/A
11.	Os nomes dos métodos são usados de acordo com as convenções de nome?			
12.	Todos os métodos possuem modificadores de acesso apropriados (private, protected, public)?			
13.	Todo valor de parâmetro de método é verificado antes de ser utilizado?			
14.	Existem métodos estáticos que deveriam ser não estáticos ou vice-versa?			
15.	Os parâmetros estão apresentados na ordem correta?			
16.	Há métodos excessivamente complexos e que deveriam ser reestruturados ou divididos em outros métodos?			
17.	Há métodos não chamados ou não necessários?			
	III – Construtor	Sim	Não	N/A

18.	Toda classe tem um construtor apropriado?			
19.	As subclasses possuem membros comuns que deveriam estar nas classes pai?			
20.	A hierarquia de herança de classe pode ser simplificada?			
	IV – Computação	Sim	Não	N/A
21.	Há computações com valores de tipos de dados distintos?			
22.	Há possíveis overflow ou underflow durante a computação?			
23.	Para cada expressão com mais de um operador: Há restrições sobre a ordem correta de avaliação e precedência?			
24.	Há parênteses para evitar ambiguidade?			
25.	O código sistematicamente previne erros de arredondamento?			
26.	O código evita adições e subtrações com grandes diferenças de magnitudes?			
27.	Os divisores são testados para divisão por zero?			
28.	Todas as exceções relevantes são capturadas?			
29.	Há uma ação apropriada para cada bloco catch?			
	V – Condição	Sim	Não	N/A
30.	Para cada teste booleano: A condição correta é verificada?			
31.	Há comparações entre variáveis e tipos inconsistentes?			
32.	Os operadores de comparação estão corretos?			
33.	Cada expressão booleana está correta?			
34.	If aninhados podem ser convertidos em switch?			
	VI – Repetição	Sim	Não	N/A
35.	Para cada loop: é a melhor escolha de estrutura de repetição?			
36.	Os loops finalizam?			
37.	Quando há múltiplas saídas de um loop, toda saída é necessária e apropriadamente manipulada?			
38.	Cada declaração switch tem um case padrão?			
39.	Se há um aninhamento profundo de loops e condições, ele está correto?			
40.	Todo método é finalizado?			
41.	Todas as exceções são manipuladas apropriadamente?			