UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

SISTEMATIZAÇÃO DA INSPEÇÃO DE CÓDIGO SQL UTILIZANDO A TÉCNICA STEPWISE ABSTRACTION E A FERRAMENTA CRISTA

AUGUSTO BINDILATTI ZAMBONI

ORIENTADORA: PROFA. DRA. SANDRA CAMARGO PINTO FERRAZ FABBRI

São Carlos – SP

Março/2015

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

SISTEMATIZAÇÃO DA INSPEÇÃO DE CÓDIGO SQL UTILIZANDO A TÉCNICA STEPWISE ABSTRACTION E A FERRAMENTA CRISTA

AUGUSTO BINDILATTI ZAMBONI

Dissertação 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 Mestre em Ciência da Computação, área de concentração: Engenharia de Software

Orientadora: Profa. Dra. Sandra Camargo Pinto Ferraz Fabbri

São Carlos – SP

Março/2015

À minha família.

AGRADECIMENTOS

À minha orientadora, Prof[a]. Dr[a]. Sandra C. P. F. Fabbri, pelos ensinamentos compartilhados, semeando ideias com muita dedicação e sabedoria, enriquecendo não apenas a minha formação profissional mas, acima de tudo, minha formação intelectual. Minha especial admiração e gratidão.

Aos meus pais, que sempre me incentivaram e contribuíram para o meu crescimento profissional, por serem também exemplos a serem seguidos, muito obrigado pela presença e carinho incondicional de vocês.

À minha namorada, Liane, por entender meus momentos de ausência, por compartilhar minhas ideias e me incentivar sempre que necessário, dividindo comigo as expectativas, frustrações e conquistas. Obrigada pelo carinho.

À todos meus amigos Lapesianos pela cooperação, disponibilidade, críticas, sugestões e apoio que me ajudaram a transformar ideias em ações.

Aos meus amigos Anderson Belgamo, André Di Thommazo, Elis Hernandes e Fábio Octaviano por toda a ajuda e companheirismo, essenciais para o bom desenvolvimento deste trabalho.

Aos meus colegas de profissão que responderam ao *survey* e tanto contribuíram com suas experiências e sabedorias.

Aos alunos da graduação do curso de Bacharelado em Ciência da Computação da UFSCar pelo comprometimento na aplicação do experimento.

Que os nossos esforços desafiem as impossibilidades, nos lembrando de que as grandes coisas do homem foram conquistadas do que parecia impossível.

Charles Chaplin

RESUMO

Contexto: A inspeção é uma atividade de garantia de qualidade de software que pode ser aplicada durante todo o processo de desenvolvimento e em diferentes tipos de artefatos, uma vez que é uma atividade estática, baseada essencialmente em técnicas de leitura. No contexto de inspeção de códigos fonte, uma técnica de leitura comumente utilizada é a *Stepwise Abstraction* (SA). Dentre os artefatos que compõem um software, destacam-se aqueles relacionados a bancos de dados como, por exemplo, códigos fontes SQL, sendo que para estes não foram identificadas propostas de técnicas de leitura para apoiar a condução de inspeção. **Objetivo:** Nesse cenário, este trabalho apresenta uma proposta de sistematização da atividade de inspeção de códigos fonte SQL utilizando a técnica *Stepwise Abstraction*, com o apoio computacional da ferramenta CRISTA. **Metodologia:** Devido à escassez de relatos e avaliações do uso da inspeção em códigos fonte SQL na literatura, elaborou-se um *survey* para identificar como o processo de inspeção de código SQL é conduzido e quais as dificuldades encontradas por profissionais de TI durante sua execução. Os resultados do *survey* forneceram diretrizes para a elaboração da proposta de sistematização da inspeção de código fonte SQL e implementação de novas funcionalidades na ferramenta CRISTA. Um estudo experimental foi conduzido para avaliar a viabilidade de uso da proposta. **Resultados:** Os resultados obtidos por meio do estudo experimental foram satisfatórios e dão indícios positivos com relação à viabilidade de uso da proposta. Quando a atividade de inspeção foi conduzida seguindo a proposta de sistematização, foram encontrados 9,3% a mais de defeitos em relação à abordagem *adhoc*. No que diz respeito aos falso positivos, o uso da proposta de sistematização resultou em zero falso positivos, enquanto que a abordagem *adhoc* teve 2 ocorrências. **Conclusão:** Com base nesses resultados, a proposta de sistematização da atividade de inspeção de código SQL com o apoio da técnica de leitura *Stepwise Abstraction* e da ferramenta CRISTA se mostrou viável de ser utilizada. Em comparação com a inspeção *adhoc*, utilizada na prática, a efetividade na detecção de defeitos aumenta enquanto que os falso positivos diminuem. Adicionalmente, ressalta-se a relevância da pesquisa para a área de engenharia de software, haja vista a escassez de trabalhos relacionados à inspeção de código SQL na literatura.

Palavras-chave: Inspeção de código, *Stepwise Abstraction*, SQL, *Structured Query Language*.

ABSTRACT

Context: Inspections are a software quality assurance activity which can be applied throughout the development process and in different types of artifacts, since it is a statistic activity, based essentially on reading techniques. In the context of source code inspection, a commonly used reading technique is the Stepwise Abstraction (SA). Among the artifacts that comprise a software, the ones related to databases stand out, for instance, SQL source code, of which no reading techniques proposals were identified to support the inspection. **Objective:** In this setting, this work shows a proposal for systematizing the SQL source code inspection activity using the Stepwise Abstraction technique, with the computational support of the CRISTA tool. **Methodology:** Due to the shortage of reports and assessment of inspection use in SQL source code in the literature, a survey was developed to identify how the SQL code inspection process is conducted and what the difficulties encountered by IT professionals during its execution are. The survey results provided directives for the development of the systematization proposal for the SQL source code inspection and implementation of new features on the CRISTA tool. An experimental study was conducted to evaluate the feasibility of use for the proposal. **Results:** The results obtained through the experimental study were satisfactory and show positive indication relating to the feasibility of use for the proposal. When the inspection activity was conducted following the systematizing proposal, 9.3% more defects were found relating to the adhoc approach. Regarding to false positives, the systematization proposal use resulted in zero false positives, while the adhoc approach had two occurrences. **Conclusion:** Based on these results, the proposal for systematizing the SQL code inspection activity, with the support of the Stepwise Abstraction reading technique and CRISTA tool, showed itself viable for use. In comparison to the adhoc inspection, used in practice, the effectiveness in defects detection increased, while the number of false positives decreased. Additionally, it should be noted the research relevance for the software engineering area, taking into account the shortage of studies related to the SQL code inspection in the literature.

Keywords: Code inspection, Stepwise Abstraction, SQL, Structure Query Language

LISTA DE FIGURAS

2.1 Custo relativo para correção de defeitos. Adaptado de (BOEHM; BASILI, 2005). . 22

2.2 Momentos propícios para realizar inspeções de software. Adaptado de (KALINOWSKI, 2004). 24

2.3 Estimativa do esforço de desenvolvimento utilizando e não utilizando inspeções. Adaptado de (WHEELER; BRYKCZYNSKI; MEESON, 1996). 25

2.4 Visão do processo de inspeção de propostp por Fagan (1976). Adaptado de (KALINOWSKI, 2004). 27

2.5 Visão do processo de inspeção proposto por Sauer et al. (2000). Adaptado de (KALINOWSKI, 2004). 32

2.6 Exemplo de aplicação da técnica Stepwise Abstraction. Adaptado de (DÓRIA, 2001). 35

2.7 Processos de inspeção de software utilizados nos estudos experimentais. Adaptado de (HERNANDES; BELGAMO; FABBRI, 2013). 37

2.8 Artefatos inspecionados nos estudos experimentais. Adaptado de (HERNANDES; BELGAMO; FABBRI, 2013). 37

2.9 Exibição dos resultados após a análise da ferramenta FindBugs (HOVEMEYER; PUGH, 2004). 39

2.10 Exemplo de uso da ferramenta JLint (ARTHO, 2006). 40

2.11 Exemplo de uso da ferramenta PMD (COPELAND, 2005). 40

2.12 Exemplo de uso da ferramenta CheckStyle (BURN, 2007). 41

2.13 Tela da ferramenta JCSC na qual o usuário pode definir os padrões que os códigos fontes devem seguir (JOCHAM, 2005). 42

2.14 Tela da ferramenta CRISTA com a metáfora visual *Treemap* (região "A"), código fonte (região "B") e registro das abstrações (região "C"). Adaptado de (PORTO, 2010). 43

3.1 Distribuição dos participantes do *survey* por tempo de experiência em TI (A) e em desenvolvimento SQL (B). 47

3.2 Utilização dos diferentes bancos de dados relatados pelos participantes do *survey*. 48

3.3 Comparação da dificuldade em inspecionar código SQL em relação a POO (A) e documentos utilizados como apoio à inspeção (B). 49

3.4 Porcentagem de falhas observadas no software em decorrência de defeitos em objetos SQL. 49

3.5 Fatores que dificultam a compreensão do código SQL durante a atividade de inspeção. 50

3.6 Primeiros passos da etapa de *Planejamento* utilizando a ferramenta CRISTA. . 59

3.7 Definição dos níveis de severidade e prioridade e últimos passos da etapa de *Planejamento* na ferramenta CRISTA. 60

3.8 Regiões apresentadas pela ferramenta CRISTA e o sincronismo entre a metáfora visual e código fonte. 61

3.9 Passos para registrar uma discrepância do código fonte na ferramenta CRISTA. 62

3.10 Diferentes cores para representação de discrepâncias na *Treemap*, dependendo de sua severidade e prioridade. 63

3.11 Lista de discrepâncias detectadas pelo inspetor. 64

3.12 Suporte da ferramenta CRISTA à etapa de *Coleção de Defeitos* 65

3.13 Suporte da ferramenta CRISTA à etapa de *Discriminação de Defeitos* 66

4.1 Tempo de experiência com desenvolvimento de códigos SQL dos participantes do estudo experimental 73

4.2 Tipo de experiência com desenvolvimento de códigos SQL dos participantes do estudo experimental 74

4.3 Análise dos defeitos relatados com o suporte de visualização da ferramenta Insight (HERNANDES et al., 2014). 80

LISTA DE TABELAS

2.1 Eficácia das Técnicas para a Identificação e Correção de Defeitos (JONES, 1996) 24

3.1 Taxonomia de defeitos em códigos fonte. 51

4.1 Resumo do estudo experimental de acordo com os objetivos e participantes. . . 69

4.2 Resultados para as variáveis 'tempo' e 'quantidade de defeitos relatados' dos participantes do tratamento com o processo de sistematização da inspeção de código SQL. 78

4.3 Resultados para as variáveis 'tempo' e 'quantidade de defeitos relatados' dos participantes do tratamento *adhoc*. 78

SUMÁRIO

CAPÍTULO 1 – INTRODUÇÃO **13**

1.1 Contexto 13

1.2 Motivação e Objetivos 16

1.3 Metodologia de Execução do Trabalho 17

1.4 Organização do Trabalho 18

CAPÍTULO 2 – INSPEÇÃO DE SOFTWARE **19**

2.1 Considerações Iniciais 19

2.2 Custo da Qualidade de Software 20

2.3 A Inspeção de Software e seus Benefícios 23

2.4 O Processo de Inspeção de Software 25

2.4.1 O processo de inspeção segundo Fagan (1976) 25

2.4.2 Evolução do processo de inspeção também proposto por Fagan (1986) . 27

2.4.3 O processo de inspeção segundo Humphrey (1989) 28

2.4.4 O processo de inspeção segundo NASA (1993) 28

2.4.5 O processo de inspeções assíncronas: FTArm (JOHNSON, 1994) 29

2.4.6 O processo de inspeção segundo Sauer et al. (2000) 30

2.5 Inspeção de Código 32

2.5.1 Técnica de leitura Stepwise Abstraction 34

2.6 Estudos Experimentais Sobre Inspeção de Software 36

2.7 Ferramentas de Apoio ao Processo de Inspeção 38
2.8 Considerações Finais 44

CAPÍTULO 3 – SISTEMATIZAÇÃO DA INSPEÇÃO DE CÓDIGO SQL COM SUPORTE COMPUTACIONAL 45

3.1 Considerações Iniciais 45
3.2 Survey 46
3.2.1 Planejamento 46
3.2.2 Amostra Obtida 47
3.2.3 Resultados do Survey 47
3.2.4 Discussão 50
3.3 Proposta de sistematização da inspeção de código SQL 51
3.3.1 Planejamento 52
3.3.2 Detecção de Defeitos 54
3.3.3 Coleção de Defeitos 56
3.3.4 Discriminação de Defeitos 57
3.3.5 Retrabalho 57
3.3.6 Acompanhamento 58
3.4 Exemplo 58
3.5 Considerações Finais 66

CAPÍTULO 4 – ESTUDO EXPERIMENTAL 68

4.1 Considerações Inciais 68
4.2 Estudo Experimental: Viabilidade de Uso da Proposta de Sistematização da Inspeção de Códigos SQL 69
4.2.1 Definição 69
4.2.2 Planejamento 70
4.2.2.1 Seleção de Contexto 71

4.2.2.2 Formulação das Hipóteses 71
4.2.2.3 Seleção das Variáveis 72
4.2.2.4 Seleção de Indivíduos 73
4.2.2.5 Projeto do Experimento 73
4.2.2.6 Instrumentação 74
4.2.2.7 Avaliação da Validade 75
4.2.3 Operação 76
4.2.4 Análise dos Dados 77
4.2.5 Discussão 79
4.3 Considerações Finais 81

CAPÍTULO 5 – CONCLUSÕES **82**

5.1 Contribuições e Limitações 83
5.2 Trabalhos Futuros 84

REFERÊNCIAS **85**

APÊNDICE A – SURVEY PARA CARACTERIZAÇÃO DA ATIVIDADE DE INSPEÇÃO DE CÓDIGO SQL NO CONTEXTO EMPRESARIAL **91**

APÊNDICE B – INSTRUMENTOS DO EXPERIMENTO **95**

B.1 Apresentação de Treinamento dos Participantes 95
B.2 Apresentação de Contextualização dos Participantes ao Domínio do Código Utilizado no Experimento 96
B.3 Questionário de Caracterização 97
B.4 Códigos SQL Utilizados no Treinamento 98
B.5 Código SQL Utilizado no Experimento 100
B.6 Especificação do Código SQL Utilizado no Experimento 101
B.7 Lista de Erros Conhecidos do Código SQL Utilizado no Experimento 102

Capítulo 1

INTRODUÇÃO

Neste capítulo é apresentado o contexto e as questões que motivaram a realização deste trabalho. É apresentada também a organização desta dissertação.

1.1 Contexto

O crescente aumento da concorrência entre as empresas fez com que, ao longo dos anos, houvesse uma busca por produtos de melhor qualidade. O mercado consumidor aumenta cada vez mais os níveis de exigência por qualidade, fazendo com que as indústrias enfrentem árduos desafios para promover a melhoria dos produtos comercializados. As indústrias fabricantes de software não são exceções a essa tendência e também necessitam garantir produtos de qualidade para permanecer no mercado. Empresas de software que forem capazes de integrar, harmonizar e acelerar seus processos de desenvolvimento e manutenção terão a primazia do mercado (ROCHA; MALDONADO; WEBER, 2001).

Segundo Pressman (2011), o processo típico de desenvolvimento de software consiste de cinco atividades: *Comunicação*, *Planejamento*, *Modelagem*, *Construção* e *Entrega*. Porém essas atividades nem sempre são executadas da mesma maneira. Há quatro diferentes fluxos: a) processo linear, ou seja, começando com a atividade de comunicação e culminando com a atividade da entrega; b) processo iterativo, no qual se repete uma ou mais atividades antes de prosseguir para a atividade seguinte; c) processo evolucionário, no qual as atividades são executadas em um formato circular e que, a cada passagem pelas cinco atividades, uma versão mais completa do software é entregue; d) processo paralelo, no qual uma ou mais atividades são executadas ao mesmo tempo que outras atividades.

Porém, o fato de uma empresa desenvolvedora de software seguir um dos processos de

desenvolvimento de software não garante que o produto final terá a qualidade desejada. Vários autores definiram qualidade no âmbito de desenvolvimento de softwares. Garvin (1984) sugere que "qualidade é um conceito complexo e multifacetado" e que pode ser mensurado a partir de cinco visões distintas. A *visão transcendental* representa a qualidade que se é reconhecida imediatamente, mas que não se pode ser definida explicitamente. A *visão do usuário* percebe a qualidade em termos de metas específicas de um usuário final, ou seja, se o produto atinge suas metas, ele apresenta qualidade. A *visão do fabricante* está relacionada ao atendimento das especificações iniciais do produto. A *visão do produto* sugere que a qualidade está ligada a características inerentes ao próprio produto como, por exemplo, suas funções e recursos. Por último, a *visão baseada em valor* define qualidade com base no quanto um usuário estaria disposto a pagar pelo produto.

Para Glass (1998), qualidade está intimamente relacionada à satisfação do cliente, sendo que a satisfação pode ser mensurada através da somatória de três fatores: produto compatível com sua finalidade, boa qualidade e entrega dentro do orçamento e do prazo previsto. Corroborando com a definição de Glass, DeMarco e Lister (1998) afirmam que a "qualidade de um produto é função do quanto ele transforma o mundo para melhor", ou seja, se um produto de software satisfaz a seus usuários finais, é possível que eles estejam dispostos a tolerar problemas ocasionais de desempenho ou confiabilidade.

A questão de qualidade de sistemas de software cresceu à medida que os sistemas de software se tornaram parte integrante do cotidiano das pessoas. Atualmente, é difícil apontar alguma atividade humana que não tenha, em maior ou menor grau, intervenção de algum software. A preocupação com relação à qualidade dos sistemas de software é proporcional às consequências que podem ser decorridas de uma possível falha. Como exemplo, pode-se deduzir que o controle de qualidade de um software que regula o nível de oxigênio no interior de uma espaçonave é maior que o controle de qualidade de um software que regula a temperatura de um refrigerador, pois as possíveis consequências de uma falha no primeiro sistema são mais graves que no segundo.

Estima-se que entre 50% e 60% do esforço total gasto no desenvolvimento de um software está relacionado a tarefas de controle e avaliação de qualidade (OSTERWEIL, 1996). O percentual pode ser ainda maior se o sistema de software a ser desenvolvido for um sistema crítico, cujo mal funcionamento ou falha possa resultar em morte ou ferimentos graves a pessoas (KOLLANUS; KOSKINEN, 2009). Sendo assim, as atividades de garantia de qualidade de software, mais especificamente, atividades de VV&T (Validação, Verificação e Teste) têm um papel fundamental dentro do processo de desenvolvimento do software, pois asseguram que os

artefatos gerados atendam aos requisitos de qualidade necessários e estejam compatíveis com a especificação do usuário, de forma a gerar o produto correto e com qualidade (SOMMERVILLE, 2011).

Dentre as atividades de VV&T, pode-se destacar a inspeção de software, utilizada para aumentar a qualidade do software, documentos e quaisquer outros artefatos produzidos durante o processo de desenvolvimento do software (FAGAN, 1976). Por ser uma atividade estática, ela pode ser aplicada desde as primeiras atividades do processo, detectando defeitos nos artefatos tão logo eles sejam inseridos (ALMEIDA et al., 2003; ANDERSON; REPS; TEITELBAUM, 2003). Esses artefatos incluem documento de requisitos, modelos de análise e projeto, código e casos de teste. Segundo Denger e Kolb (2006), a inspeção pode levar à detecção e correção de 50% a 90% dos defeitos. Apesar de sua efetividade, a inspeção é feita, em geral, de forma manual, tornando-se uma atividade muito demorada, propensa a erros e trabalhosa. Por ser essencialmente uma atividade de leitura do artefato que está sendo avaliado, ela é apoiada por técnicas de leitura, que podem variar de acordo com o artefato a ser inspecionado. Assim, técnicas de leitura que apoiam, por exemplo, a inspeção de requisitos são diferentes das técnicas de leitura que apoiam a inspeção de código.

Inerente à atividade de inspeção de código, pode-se observar a atividade de compreensão. Essa atividade refere-se a qualquer método estático ou dinâmico utilizado para identificar propriedades funcionais ou estruturais de um sistema de software (VINZ; ETZKORN, 2006). Essas atividades visam entender os artefatos, analisando suas estruturas e construindo representações de alto nível que os representam. Dessa forma, enquanto algumas técnicas de compreensão tentam padronizar abstrações derivadas de comentários de programas e nomes de variáveis, outras tentam entender o programa repetindo regras de transformação para derivar conceitos abstratos que representem trechos de código.

Porém, assim como a inspeção, uma vez que a compreensão seja realizada de forma manual ou sem instrumentos voltados especificamente para esse fim, ela se torna bastante trabalhosa, pouco efetiva e pouco eficiente. Somando-se isso à crescente complexidade dos softwares desenvolvidos atualmente, têm sido investigadas algumas alternativas para facilitar as atividades de VV&T. Exemplos disso são as técnicas de visualização de software (SENSALIRE; OGAO, 2007).

O processo de visualização está relacionado à transformação de algo abstrato em imagens, sejam elas mentais ou reais, que possam ser visualizadas pelos seres humanos (NASCIMENTO; FERREIRA, 2005). Como o processo cognitivo de seres humanos é mais intuitivo, efetivo e eficiente quando apoiado por recursos visuais (TERGAN; KELLER, 2005), a visualização tem

sido apontada como uma possível solução à compreensão de sistemas de software complexos.

Entre os artefatos que compõem os sistemas computacionais, pode-se destacar aqueles vinculados aos bancos de dados relacionais como, por exemplo, *stored procedures*, *views*, *functions* e *triggers*. Em sistemas de software grandes e complexos, o volume de dados armazenados em bancos de dados relacionais pode ser extremamente grande e uma boa qualidade dos artefatos que os manipulam reflete significativamente na qualidade dos sistemas de software como um todo. Porém, apesar da importância do banco de dados e sua direta influência na qualidade dos sistemas de software, há uma carência na literatura por trabalhos que abordem a aplicação de atividades de V&V, mais especificamente as atividades de compreensão e inspeção, no contexto de artefatos de bancos de dados relacionais.

Nesse cenário, a elaboração e avaliação de uma abordagem sistêmica à atividade de inspeção de códigos fonte escritos na linguagem SQL, utilizando uma técnica de leitura e apoio computacional para executá-la, mostram-se válidas e de grande potencial.

1.2 Motivação e Objetivos

Face ao exposto, caracterizada a importância da inspeção dentre as atividades de VV&T para se controlar e garantir a qualidade do produto final, e a carência de abordagens sistêmicas dessa atividade no contexto de artefatos de bancos de dados relacionais, o objetivo deste trabalho foi propor e avaliar a sistematização da atividade de inspeção no contexto de código fonte SQL, utilizando a técnica de leitura *Stepwise Abstraction* e o apoio computacional da ferramenta CRISTA. A ferramenta CRISTA (PORTO; MENDONÇA; FABBRI, 2008) foi escolhida para dar o apoio computacional, uma vez que ela implementa um processo de inspeção de código com a técnica de leitura *Stepwise Abstraction*.

Ressalta-se que o autor deste trabalho atuou em uma importante instituição financeira, em que o volume de dados armazenados em bancos de dados relacionais era significativo e havia-se a necessidade de garantir que o acesso a esses dados dava-se de forma consistente e eficiente. Porém, na época, não havia uma abordagem sistemática para guiar a atividade de inspeção de códigos SQL por parte dos desenvolvedores, fazendo com que, na maioria das vezes, a atividade fosse executada de forma *adhoc*, diminuindo sua eficiência, eficácia e, na maioria das vezes, gerando retrabalho.

1.3 Metodologia de Execução do Trabalho

O trabalho foi iniciado com uma revisão sistemática sobre qualidade de software, com maior ênfase na caracterização da atividade de inspeção de código no contexto de banco de dados relacionais. Porém poucos trabalhos relacionados a esse assunto foram encontrados na literatura e, daqueles encontrados, nenhum abordava exatamente o tema pesquisado. Nesse cenário, optou-se por elaborar um *survey* com o intuito de caracterizar a atividade de inspeção de código no contexto de banco de dados relacionais, por meio de um questionário respondido por profissionais de TI com experiência em desenvolvimento de códigos SQL no ambiente empresarial.

Com os resultados obtidos do *survey* e as experiências do autor deste trabalho em executar a atividade de inspeção de códigos SQL de forma *adhoc* no mercado de trabalho, foi possível propor uma abordagem sistêmica para a realização da inspeção. Como apoio computacional à abordagem, optou-se por utilizar a ferramenta CRISTA pois ela reunia os requisitos básicos desejados como, por exemplo, suporte à técnica de leitura *Stepwise Abstraction* e criação da metáfora visual do código fonte por meio da *Treemap*. Outro fator que influenciou na escolha da ferramenta CRISTA foi o fato dela ter sido desenvolvida no trabalho de mestrado de outro aluno deste grupo de pesquisa e o autor desta dissertação ter tido a oportunidade de participar de algumas atividades durante esse desenvolvimento, parceria que inclusive promoveu uma publicação conjunta (PORTO et al., 2009).

Tomadas essas decisões, fez-se uma análise de quais funcionalidades deveriam ser adicionadas à ferramenta CRISTA para satisfazer todas as necessidades da abordagem sistêmica da atividade de inspeção de código fonte SQL. Dessa forma, a CRISTA foi aprimorada com novas funcionalidades implementadas seguindo o processo evolutivo de desenvolvimento de software.

Atingida uma versão estável da ferramenta CRISTA, após a implementação das funcionalidades necessárias, foi realizado um estudo experimental para avaliar a viabilidade de uso da abordagem sistêmica da atividade de inspeção com o apoio computacional da CRISTA no contexto de inspeções de código SQL. Os resultados obtidos nesse experimento foram satisfatórios, porém inconclusivos, sendo necessário a elaboração e condução de outros estudos experimentais a fim de generalizar quaisquer resultados.

Os resultados preliminares obtidos no estudo experimental e as descrições tanto da abordagem sistêmica para a realização da atividade de inspeção de código SQL, quanto da ferramenta CRISTA, estão sendo sumarizados para dar início à escrita de um artigo científico para submissão em eventos da área.

1.4 Organização do Trabalho

Esta dissertação é composta por 5 capítulos e 2 apêndices, cujos objetivos estão descritos a seguir:

No Capítulo 1, *Introdução*, é caracteriza a área de pesquisa em que o trabalho foi desenvolvido, evidenciando as lacunas que levaram à motivação e objetivos.

No Capítulo 2, *Inspeção de Software*, o conceito de inspeção de software e, mais especificamente, inspeção de código, são introduzidos e seus benefícios são apresentados. Nesse capítulo ainda é discutido o estado atual da atividade de inspeção na literatura e o apoio ferramental atualmente existente.

No Capítulo 3, *Proposta de Sistematização da Inspeção de Código SQL com Suporte Computacional*, é descrita a abordagem sistêmica à atividade de inspeção de código SQL, por meio da instanciação de um processo de inspeção já existente ao contexto de códigos SQL apoiado computacionalmente pela ferramenta CRISTA. Nesse capítulo é também apresentado, detalhadamente, o uso da ferramenta CRISTA no contexto de inspeção de códigos SQL.

No Capítulo 4, *Estudo Experimental*, é descrito um estudo experimental para avaliar a viabilidade de uso da abordagem sistêmica à atividade de inspeção de código SQL por meio do apoio computacional da ferramenta CRISTA.

No Capítulo 5, *Conclusões*, apresenta-se uma caracterização geral da proposta deste trabalho, destacando as suas contribuições e limitações, bem como os possíveis trabalhos que poderão ser realizados a partir deste.

O Apêndice A, *Survey para Caracterização da Atividade de Inspeção de Código SQL no Contexto Empresarial*, contém o *survey* aplicado neste trabalho na íntegra, listando todas as questões e as possíveis respostas nos casos de questões fechadas.

O Apêndice B, *Instrumentos do Experimento*, contém os seguintes instrumentos utilizados na operação do experimento: apresentação de treinamento dos participantes, apresentação de contextualização dos participantes ao domínio do código utilizado no experimento, questionário de caracterização, códigos SQL utilizados no treinamento, código SQL utilizado no experimento, especificação do código SQL utilizado no experimento e lista de erros conhecidos no código SQL utilizado no experimento.

Capítulo 2

INSPEÇÃO DE SOFTWARE

Este capítulo apresenta o papel da inspeção de software dentro das atividades de garantia de qualidade de software. São apresentados os benefícios oriundos da inspeção de software, juntamente com o seu processo, dando ênfase à inspeção de código. São exploradas também as técnicas de leitura e as ferramentas de apoio ao processo de inspeção.

2.1 Considerações Iniciais

A inspeção de software é um tipo particular de revisão que pode ser aplicada a todos os artefatos produzidos ao longo do ciclo de desenvolvimento de software. Juntamente com as atividades de teste, as inspeções compõem as atividades de VV&T (Verificação, Validação e Teste) e são consideradas parte das atividades de garantia de qualidade de software (FAGAN, 1976). Essas atividades devem estar presentes durante todo o processo de desenvolvimento de software caso deseje-se garantir maior qualidade aos artefatos liberados em cada fase do ciclo de desenvolvimento (BASILI et al., 1996). Tais atividades são explicitamente tratadas nos modelos de qualidade de processo como CMMI (SEI, 2010), ISO 12.207 (ISO/IEC12207, 1998) e MPS-BR (MPS-BR, 2012) e são essenciais para a implantação desses modelos.

No contexto de VV&T existem quatro terminologias bastante utilizadas que, se não definidas com clareza, podem-se se confundir entre elas e dificultar o entendimento das atividades de garantia de qualidade de software. São elas: *erro*, *defeito*, *falha* e *engano*. Suas definições, de acordo com o padrão da IEEE (1990), são apresentadas a seguir:

- *Erro* – um erro ocorre quando alguma parte do software assume como resposta um valor ou um estado indesejado. O erro ocorre pela ativação de um defeito ou mais defeitos;
- *Defeito* – é um passo, processo ou definição de dados incorretos. O defeito, que é geral-

mente conhecido como *bug*, é a causa hipotética da falha;

- *Falha* – é o comportamento operacional do software diferente do esperado pelo usuário. Uma falha pode ter sido causada por diversos defeitos e alguns defeitos podem nunca causar uma falha. A principal característica da falha é que ela é o resultado final percebido pelo usuário;

- *Engano* – é uma ação humana que produz um resultado incorreto.

No contexto mais específico de inspeção de software, há duas terminologias muito importantes: *discrepância* e *defeito*. A definição de *discrepância* é dada por algo que diverge do que se era esperado inicialmente e costuma-se usá-lo quando se compara a abstração feita pelo inspetor com a especificação original do artefato que foi inspecionado. O termo *defeito* mantém a mesma definição dada no contexto de VV&T. Uma discrepância pode vir a ser classificada como um defeito após sua análise e verificação de que realmente a discrepância corresponde à um passo, processo ou definição de dados incorretos.

A atividade de inspeção de software está intimamente ligada à necessidade de se garantir níveis de qualidade aos artefatos produzidos durante o processo de desenvolvimento de software. Como resultado, espera-se não apenas um produto melhor e mais confiável, mas também a redução dos custos envolvidos no processo de desenvolvimento.

O restante do capítulo está organizado da seguinte forma: na Seção 2.2 é abordada a relação entre qualidade de software e o custo de se desenvolver e manter um sistema de software. Na Seção 2.3 são apresentados alguns benefícios de se realizar inspeções nos artefatos produzidos ao longo do processo de desenvolvimento de software. Na Seção 2.4, é descrito detalhadamente o processo de inspeção de software e suas características, bem como as evoluções sofridas durantes os anos. Já as características peculiares das inspeções de código são apresentadas na Seção 2.5. Em seguida, na Seção 2.6, é apresentado o estado atual do processo de inspeção de software na condução de estudos experimentais disponíveis na literatura. A Seção 2.7 aborda algumas das ferramentas que apoiam o processo de inspeção. Por fim, a Seção 2.8 conclui este capítulo.

2.2 Custo da Qualidade de Software

Não há dúvidas de que software de qualidade é um dos principais objetivos de equipes de desenvolvimento de software. Porém, muitas vezes, qualidade de software torna-se algo

bastante subjetivo e de difícil mensuração, fazendo com que não se tenha uma linha de separação clara entre softwares com qualidade e softwares sem qualidade.

A subjetividade da qualidade de software conduz a um problema conhecido como "dilema da qualidade de software". Se um software é desenvolvido com péssima qualidade, perde-se porque ninguém terá interesse em utilizá-lo. Se, por outro lado, for empreendido um esforço extremamente grande e virtuosas somas de dinheiro para construir um software teoricamente perfeito, o custo de produção será tão alto que o produto final não terá viabilidade econômica. Dessa forma, é necessário encontrar um meio termo entre qualidade e custo para que o produto não seja rejeitado logo de cara em decorrência de sua baixa qualidade, mas também que não consuma todos os recursos na tentativa de desenvolvê-lo com um perfeccionismo exagerado (VENNER, 2003).

Nesse contexto, surge o conceito de "bom o suficiente". Software bom o suficiente fornece funcionalidades de alta qualidade, principalmente aquelas que os usuários mais anseiam e que, provavelmente, serão utilizadas com mais frequência. Porém, ao mesmo tempo fornece outras funcionalidades menos utilizadas ou com menor importância ao usuário, com defeitos conhecidos. O fornecedor espera que a grande maioria dos usuários ignore as falhas, ou pelo menos aceite conviver com elas, pois estão suficientemente satisfeitos com as funcionalidades que possuem alta qualidade. Porém essa filosofia não é aplicável em todos os domínios de software pois pode conduzir a empresa desenvolvedora de software à litígios custosos e ainda constituir crime quando um erro, conhecido ou não, tiver consequências graves à manutenção da vida de pessoas (PRESSMAN, 2011; BREEN, 2002).

O custo da qualidade de software deve ser calculado a partir da soma de todos os custos de recursos gastos na busca pela qualidade e também os custos causados pela falta dela. Pode-se dividir esses custos em três grupos: *custos de prevenção*, *custos de falhas internas* e *custos de falhas externas*.

- *Custos de prevenção* – incluem o custo de gerenciamento e execução das atividades de garantia de qualidade de software e o custo relacionado ao treinamento e posterior análise dos resultados gerados por essas atividades.
- *Custos de falhas internas* – são os custos que não existiriam caso nenhum defeito tivesse sido inserido ao produto antes da entrega ao cliente. Normalmente, esses custos são relacionados ao retrabalho para se corrigir um defeito e os possíveis efeitos colaterais resultantes dessa correção.
- *Custos de falhas externas* – são os custos relacionados aos defeitos encontrados após o

produto ter sido entregue ao cliente como, por exemplo, resolução de reclamações, devolução e substituição de produtos, suporte telefônico/e-mail e retrabalho para se corrigir o defeito.

Outra importante característica relacionada ao custo da qualidade de software é o aumento exponencial do custo ao descobrir e corrigir um defeito nos diferentes momentos do ciclo de desenvolvimento de software. Quanto mais tardio for o momento de correção, maior será o custo (BOEHM; BASILI, 2005). Ou seja, o *custo de falhas externas* é maior que o *custo de falhas internas* que, por sua vez, é maior que o *custo de prevenção*. A Figura 2.1 ilustra essa característica e deixa bastante claro a diferença dos custos em se corrigir um defeito ao longo do ciclo de desenvolvimento, sendo que pode ser 100 vezes mais caro corrigir um defeito na fase de pós-implantação se comparado com o valor da correção ainda na fase de levantamento de requisitos.

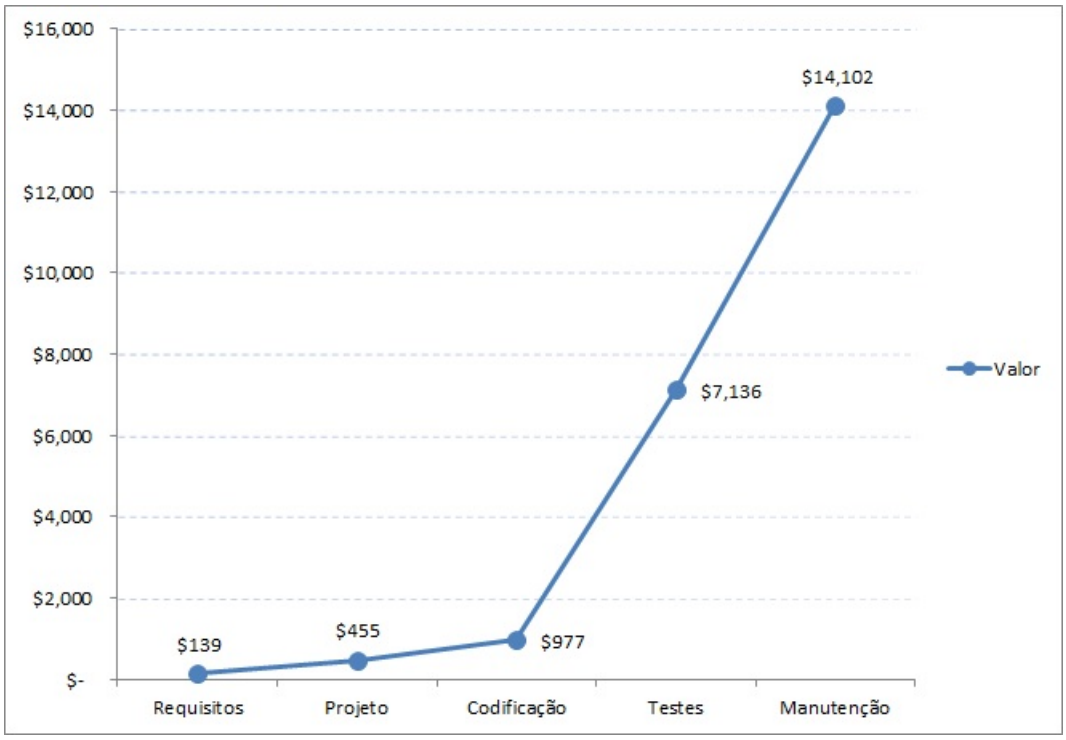


Figura 2.1: Custo relativo para correção de defeitos. Adaptado de (BOEHM; BASILI, 2005).

Nesse cenário, pode-se vislumbrar a importância de se investir e aplicar atividades que possibilitem encontrar e corrigir defeitos tão logo eles sejam inseridos nos artefatos que compõem o sistema de software. Mesmo que essa estratégia aumente os custos de prevenção, ela pode diminuir os custos relacionados às falhas internas e externas. Dessa forma, a atividade de inspeção de software desempenha um papel importante nessa estratégia, uma vez que ela tem o objetivo de encontrar e corrigir defeitos em diferentes artefatos, e pode ser aplicada desde as

primeiras fases do ciclo de desenvolvimento de software, onde o custo de se corrigir um defeito é muito menor em relação às fases seguintes.

2.3 A Inspeção de Software e seus Benefícios

A atividade de inspeção é uma abordagem estruturada e sistemática, que possui um processo de detecção de defeitos rigoroso e bem definido. Introduzida por Fagan em 1976 (FAGAN, 1976) nos laboratórios da IBM, a inspeção de software tem por objetivo detectar defeitos nos artefatos que compõem um software tão logo eles sejam inseridos e pode ser aplicada em vários tipos de documentos, tais como: documentos de texto, modelos, gráficos, fragmentos de código, etc. (WINKLER; THURNHER; BIFFL, 2007). Desde o primeiro processo proposto por Fagan, a atividade de inspeção tem sido considerada uma das melhores práticas de detecção de defeitos da engenharia de software (ANDERSON; REPS; TEITELBAUM, 2003; ANDERSON et al., 2003).

Como visto na Seção 2.2, referente ao custo de se corrigir defeitos ao longo do ciclo de desenvolvimento de software, a oportunidade oferecida pela inspeção de software de detectar defeitos e corrigi-los na mesma fase de desenvolvimento em que eles são inseridos passa a ter um peso bastante grande no momento de definição de quais atividades serão aplicadas para garantir a qualidade de um software. A importância das inspeções no contexto da qualidade do software produzido é bem documentada na literatura (ACKERMAN; BUCHWALD; LEWSKI, 1989; BOOGERD; MOONEN, 2006; DENGER; SHULL, 2007; FAGAN, 1976; GUPTA; PATNAIK; GOEL, 2003; HARJUMAA; TERVONEN; VUORIO, 2004; KALINOWSKI; TRAVASSOS, 2004; SAUER et al., 2000).

Diferentemente das necessidades das atividades de teste, a inspeção de software não requer que o artefato a ser inspecionado esteja concluído. Essa característica faz com que a atividade de inspeção seja classificada como uma atividade estática dentro das atividades de VV&T (MYERS; SANDLER, 2004). Dessa forma, ela pode ser aplicada em diferentes momentos, por diferentes pessoas e utilizando diferentes abordagens. A Figura 2.2 mostra os momentos durante o ciclo de desenvolvimento do software em que é possível a realização da atividade de inspeção de software (KALINOWSKI, 2004). Sempre que a inspeção de software é executada e defeitos são encontrados, deve-se avaliar a necessidade de voltar à fase anterior para retrabalho. Essa decisão é representada pelas flechas que saem dos quadrados amarelos (momentos de inspeção) e chegam nos retângulos verdes (fases do processo de desenvolvimento de software).

Vários estudos mostram que a atividade de inspeção de software é capaz de identificar a maioria dos tipos de defeitos existentes, fazendo com que haja uma significativa redução de tempo e esforço em atividades de VV&T posteriores como, por exemplo, testes. De acordo

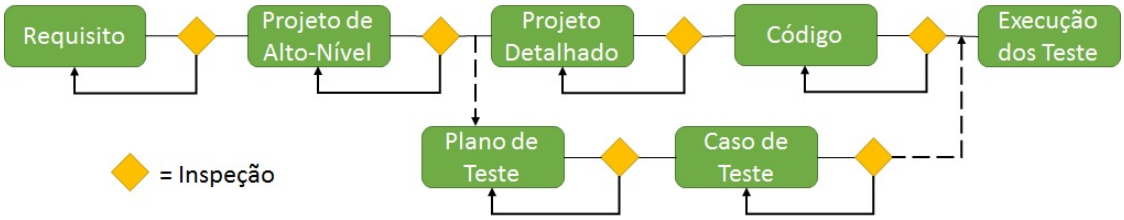


Figura 2.2: Momentos propícios para realizar inspeções de software. Adaptado de (KALINOWSKI, 2004).

com a Tabela 2.1, as inspeções formais de projeto e de código se destacam pela eficácia dentre outras alternativas.

Tabela 2.1: Eficácia das Técnicas para a Identificação e Correção de Defeitos (JONES, 1996)

ATIVIDADE	*EFICÁCIA*
Revisões informais de projeto	25% a 40%
Inspeções formais de projeto	45% a 65%
Revisões informais de código	20% a 35%
Inspeções formais de código	45% a 65%
Teste de unidade	15% a 50%
Teste de integração	25% a 40%
Teste do sistema	25% a 55%
Beta teste ($\leq$10 clientes)	24% a 40%
Beta teste ($\leq$1000 clientes)	60% a 85%

Wheeler, Brykczynski e Meeson (1996) publicaram uma comparação de esforço empreendido no desenvolvimento de software entre projetos que aplicaram inspeção e projetos que não aplicaram inspeção. A Figura 2.3 apresenta o resultado da comparação, o qual deixa bem claro a redução do esforço ao longo do tempo e a conclusão do projeto em menos tempo quando se aplica inspeção.

Outros benefícios secundários da inspeção são um software melhor documentado, aumentando a eficiência do compartilhamento de conhecimento entre membros ou a inclusão de novos membros à equipe de desenvolvimento e ainda diminuição das chances de adicionar novos defeitos em atividades posteriores no ciclo de desenvolvimento do software (DENGER; SHULL, 2007). Além disso, ela também se faz útil para fornecer informações que podem ser utilizadas no suporte às tomadas de decisões acerca do projeto (THELIN et al., 2004). Nesse sentido, as inspeções podem contribuir auxiliando os gerentes de projeto no gerenciamento de riscos, além de auxiliar os engenheiros de software, apontando melhorias específicas no processo analisado (BIFFL; GRÜNBACHER; HALLING, 2006).

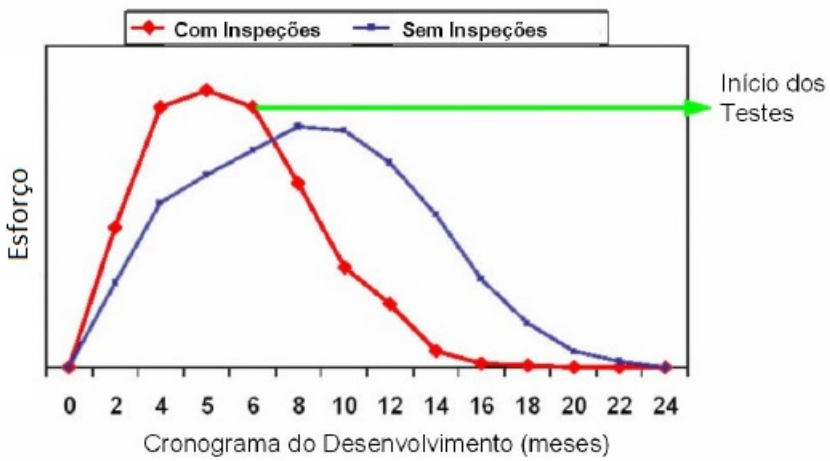


Figura 2.3: Estimativa do esforço de desenvolvimento utilizando e não utilizando inspeções. Adaptado de (WHEELER; BRYKCZYNSKI; MEESON, 1996).

2.4 O Processo de Inspeção de Software

O processo de inspeção de software foi proposto inicialmente por Fagan em 1976, dentro dos laboratórios da IBM (FAGAN, 1976). Desde então, passou por algumas modificações principalmente nas etapas-chave do processo conhecidas como *Preparação*, na qual ocorre a preparação individual do inspetor e *Reunião de inspeção*, que corresponde à atividade de se inspecionar o artefato em busca de discrepâncias.

O processo de inspeção é formado por etapas rigorosas e bem definidas e executado por várias pessoas, com papéis bem definidos. As evoluções, papéis e etapas são descritas a seguir, a começar cronologicamente pelo primeiro processo proposto para inspeção.

2.4.1 O processo de inspeção segundo Fagan (1976)

O processo tradicional de inspeção, proposto inicialmente em (FAGAN, 1976) possui cinco etapas bem definidas: *Apresentação*, *Preparação*, *Inspeção*, *Retrabalho* e *Acompanhamento*. Além disso, há quatro papéis bem definidos que são desempenhados pelos membros da equipe de inspeção: *Moderador*, *Projetista*, *Desenvolvedor* e *Testador*.

Segundo Fagan (1976), a equipe de inspeção tem sua melhor formação quando composta por pessoas de diferentes áreas, pois cada um analisará o produto com base em sua própria perspectiva e experiência, facilitando a identificação de mais defeitos. A composição da equipe de inspeção e seus respectivos papéis é a seguinte:

1. *Moderador* – a principal pessoa para uma inspeção bem sucedida. O *moderador* deve

ser um profissional competente e experiente, mas não necessariamente deve conhecer a fundo o software a ser inspecionado. O *moderador* deve guiar a equipe pelo processo de inspeção, sendo a figura representativa de um líder, procurando sempre manter a sinergia da equipe para a obtenção de um melhor resultado. Além disso, é dever do *moderador* cuidar da logística de reuniões, reportar o status da inspeção diariamente e acompanhar os resultados do retrabalho.

2. *Projetista* – o profissional responsável por projetar o software.

3. *Desenvolvedor* – o profissional responsável em codificar o que foi previamente projetado.

4. *Testador* – o profissional responsável em escrever e/ou executar os casos de teste ou, de qualquer outra forma, testar o produto desenvolvido pelo *projetista* e *desenvolvedor*.

Junto aos papéis, foram também definidas as etapas que compõem o processo de inspeção. Manter a objetividade de cada uma das etapas é uma das principais funções do *moderador* (FAGAN, 1976). As etapas são descritas a seguir:

1. *Apresentação* (participação de toda a equipe) – o *moderador* reúne toda a equipe para que seja feita uma passagem de conhecimento acerca do software a ser inspecionado. O *projetista* descreve o escopo do software e os aspectos de projeto e implementação. As documentações do software são distribuídas à todos os membros da equipe de inspeção ao final da apresentação.

2. *Preparação* (individualmente) – os participantes, em posse da documentação distribuída na etapa de *Apresentação*, estudam os artefatos, tentando entender seu projeto, objetivos e lógica. Eventualmente são feitas anotações sobre os artefatos, produzindo assim uma lista preliminar de discrepâncias entre a abstração do artefato e sua documentação.

3. *Inspeção* (participação de toda a equipe) – é feita uma reunião com o objetivo de procurar defeitos. Um "leitor" é escolhido pelo *moderador* (geralmente o *desenvolvedor*) que descreve como o software projetado foi codificado. À medida que as discrepâncias são encontradas, elas são anotadas, discutidas sem a preocupação de resolvê-las e posteriormente, cada uma das discrepâncias é classificada pelo *moderador* como defeito ou falso positivo. Depois da reunião o *moderador* produz um relatório contendo todos os defeitos identificados.

4. *Retrabalho* – todos os defeitos encontrados durante a etapa de *Inspeção* são corrigidos pelo *desenvolvedor* e/ou *projetista*.

5. *Acompanhamento* (moderador) – é responsabilidade do *moderador* verificar se todos os defeitos reportados na etapa de *Inspeção* foram devidamente corrigidos durante a etapa de *Retrabalho*. Se cinco por cento ou mais dos artefatos foram alterados, a equipe deve se reunir novamente e executar todo o processo de inspeção. Por outro lado, se menos de cinco por cento dos artefatos foram alterados, o *moderador* deve avaliar a qualidade do retrabalho e reunir a equipe para avaliar a necessidade de se realizar uma nova inspeção de todos os artefatos ou apenas daqueles alterados.

A Figura 2.4 apresenta as etapas e papéis envolvidos no processo de inspeção proposto por Fagan (1976).

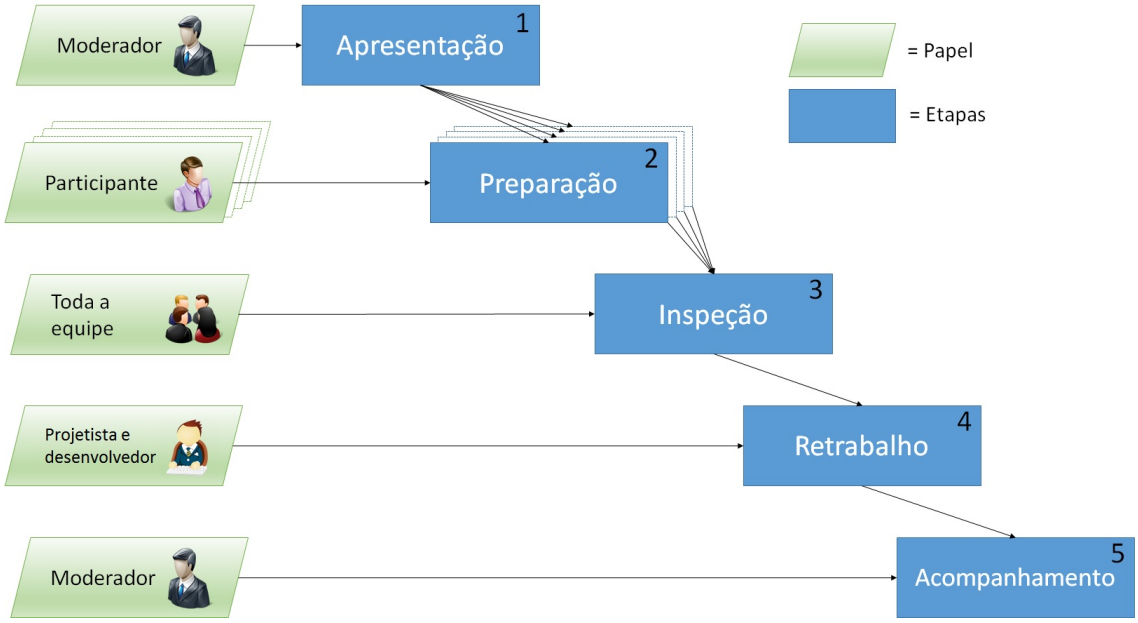


Figura 2.4: Visão do processo de inspeção de propostp por Fagan (1976). Adaptado de (KALINOWSKI, 2004).

2.4.2 Evolução do processo de inspeção também proposto por Fagan (1986)

Com base em seu próprio trabalho de 1976, Fagan apresentou, em 1986 (FAGAN, 1986), uma reorganização do processo de inspeção. Esse novo processo contava com as seguintes etapas: *Planejamento*, *Apresentação*, *Preparação*, *Inspeção*, *Retrabalho* e *Acompanhamento*. As principais mudanças em relação ao processo de 1976 foram:

- Foi criada a etapa de *Planejamento*, na qual o *moderador* define o material a ser inspecionado, seleciona as pessoas que vão inspecionar o artefato e agenda a reunião da etapa de *Inspeção*.

- A etapa de *Apresentação* sofreu uma pequena mudança em sua execução. No novo processo, nessa etapa são apresentados os artefatos a serem inspecionados, bem como a forma de se inspecioná-los. Fagan ainda diz que essa etapa pode ser omitida (com pequenos riscos) caso a equipe já possua conhecimento sobre o projeto e os artefatos que devem ser inspecionados.

No trabalho de 1986, Fagan não mudou os papéis da equipe de inspeção. No entanto ele destaca a importância do papel do *moderador*, indicando que ele deve possuir um treinamento especial para conduzir as inspeções da melhor maneira, obtendo uma sinergia entre os participantes. Fagan ressalta ainda que, para garantir a objetividade do processo, o *moderador* não deve estar envolvido no desenvolvimento do artefato que está sendo inspecionado, mas estar trabalhando em projetos similares.

2.4.3 O processo de inspeção segundo Humphrey (1989)

Em 1989, Humphrey propôs uma variação dos processos de inspeção propostos em (FAGAN, 1976, 1986). Sua principal diferença consiste em alterar o foco da etapa de *Preparação*, que era de tentar entender o artefato a ser inspecionado através de sua abstração, para efetivamente encontrar as discrepâncias. Sendo assim, ao final dessa etapa, cada um dos *participantes* entrega uma lista de discrepâncias ao *moderador* da inspeção antes da reunião na etapa de *Inspeção*. Dessa forma, durante a reunião não se procura mais por discrepâncias, mas passa-se apenas a discutir as que já foram encontradas e classificá-las como defeito ou falso-positivo. As etapas definidas no processo de Humphrey (1989) são: *Inicialização*, *Preparação*, *Inspeção* e *Pós-inspeção*.

2.4.4 O processo de inspeção segundo NASA (1993)

Em 1993, a NASA (*National Aeronautics and Space Administration*) publicou um guia para inspeções de software (NASA, 1993). Nele, o processo de inspeção é dividido nas etapas: *Planejamento*, *Apresentação*, *Preparação*, *Reunião de Inspeção*, *Terceiro Momento*, *Retrabalho* e *Acompanhamento*.

A etapa de *Inspeção* encontrada nos processos de Fagan (1976, 1986) e Humphrey (1989), foi renomeada pela NASA para *Reunião de Inspeção*, porém tanto seu objetivo quanto as pessoas envolvidas continuam os mesmos, ou seja, essa etapa continua sendo utilizada para discutir as discrepâncias encontradas individualmente, como proposto em (HUMPHREY, 1989). Diferenças mais substanciais incluem o acréscimo da etapa *Terceiro Momento*, a qual oferece um tempo

adicional para discussões, soluções de problemas ou fechamento de questões levantadas durante a etapa de *Reunião de Inspeção*. Segundo a NASA, essa etapa é opcional. Além disso, cabe ressaltar a denominação de dois novos papéis:

1. *Inspetor* – todos os membros da equipe de inspeção são considerados inspetores, independentemente do papel já assumido. Além do papel de *inspetor*, os membros podem assumir os papéis de *moderador*, *autor*, *leitor* e *documentador*, quando apropriado. Os *inspetores* são as pessoas responsáveis por identificar as discrepâncias nos artefatos. Os principais candidatos a inspetores são as pessoas envolvidas com o artefato a ser inspecionado (envolvidas nas etapas anteriores, corrente e posteriores do ciclo de desenvolvimento do software). Por exemplo, para avaliar um documento de projeto de sistema, bons inspetores estariam entre os que escreveram os requisitos e os que irão codificar o programa. Vale lembrar que essas funções sempre foram realizadas desde o processo de Fagan (1976), entretanto, esse papel não estava explicitamente mencionado. O mesmo aconteceu com o papel de *autor*, referenciado pela NASA e subentendido nos processos anteriores.

2. *Documentador* – o *documentador* é a pessoa responsável por registrar na lista de defeitos cada discrepância analisada e classificada como defeito, bem como registrar as decisões e recomendações feitas durante a etapa de *Reunião de Inspeção*. Essas atividades eram realizadas pelo *moderador* nos processos anteriores.

O guia criado pela NASA é rico em detalhes, fornecendo explicações para partes omissas e/ou subentendidas nos processos anteriores. Isso torna o guia uma ótima referência para o aprendizado do processo de inspeção.

2.4.5 O processo de inspeções assíncronas: FTArm (JOHNSON, 1994)

A partir das propostas de processo de inspeção anteriores e do trabalho de Votta (1993), no qual ele defende que reuniões de inspeção podem ser evitadas, reduzindo custos e conflitos de alocação de recursos sem sacrificar a eficiência da inspeção, Johnson (1994) propôs um processo chamado FTArm (acrônimo em inglês para o termo "Método de Revisão Técnica Formal Assíncrona"). As etapas do processo FTArm são: *Configuração* (análogo à etapa de Planejamento), *Orientação* (análogo à etapa de Apresentação), *Revisão Particular* (análogo à etapa de Preparação), *Revisão Pública*, *Consolidação* e *Reunião de Revisão em Grupo*.

Como listado acima, as etapas de *Configuração*, *Orientação* e *Revisão Particular* são as mesmas do processo da NASA (1993), com exceção dos nomes. Entretanto, as etapas *Revisão*

Pública, *Consolidação* e *Reunião de Revisão em Grupo* não podem ser comparadas diretamente com as etapas já existentes em outros processos. Assim, o objetivo de cada uma dessas etapas é definido a seguir:

- *Revisão Pública* – tem por objetivo discutir as discrepâncias identificadas durante a etapa de *Revisão Particular*. A diferença dessa etapa para a etapa de *Reunião de Inspeção* no processo da NASA (1993), é que essa etapa é feita de forma assíncrona e usa votações para chegar a um consenso entre os inspetores a cerca de cada discrepância relatada durante a etapa de *Revisão Particular*. Essa etapa termina quando todos os inspetores revisaram todas as discrepâncias e a votação tenha se estabilizado.

- *Consolidação* – é realizada pelo *moderador* da inspeção e tem como objetivo agrupar os resultados obtidos durante as etapas de *Revisão Particular* e *Revisão Pública*. Vale lembrar que essa atividade sempre esteve presente no processo de inspeção e era executada como uma das tarefas da etapa de *Inspeção* de Fagan (1976, 1986) e Humphrey (1989).

- *Reunião de Revisão em Grupo* – tem por objetivo discutir os questões gerais não resolvidas observados pelo relatório produzido na etapa de Consolidação.

Vale destacar que as etapas de *Retrabalho* e *Continuação* descritos no processo da NASA (1993) estão implícitas no processo de Johnson (1994).

Outros trabalhos (JOHNSON; TJAHJONO, 1997, 1998; PORTER; JOHNSON, 1997) relatam a execução e os resultados de um estudo experimental realizado para avaliar o argumento de Votta (1993), sobre a não necessidade de uma reunião de inspeção no processo de inspeção de software. Os resultados indicam não haver uma diferença substancial na efetividade da inspeção com ou sem reuniões de inspeção. Entretanto, observou-se que o processo com reuniões de inspeção encontrou, ainda que em pequeno número, classes de defeitos não encontradas em inspeções sem reuniões. Além disso, observou-se também que inspeções com reuniões de inspeção produziram menos falsos positivos.

2.4.6 O processo de inspeção segundo Sauer et al. (2000)

Com base em diversos estudos experimentais sobre inspeções de software, Sauer et al. (2000) propuseram uma reorganização do processo tradicional de inspeção. A reorganização de Sauer introduziu mudanças para redução do custo e do tempo total para realizar uma inspeção, adequando o processo de Fagan (1986), com as reuniões assíncronas do processo FTArm de Johnson (1994) e considerando equipes geograficamente distribuídas.

Basicamente, a variação proposta por Sauer et al. (2000) substitui as etapas de *Preparação* e de *Inspeção* (*Reunião de Inspeção*) dos processos anteriores por três novas etapas sequenciais: *Detecção de Defeitos*, *Coleção de Defeitos* e *Discriminação de Defeitos*. Essas atividades podem ser descritas da seguinte forma:

- *Detecção de Defeitos* – cada um dos *inspetores* selecionados pelo *moderador* na etapa de *Planejamento* realiza uma atividade de abstração do artefato inspecionado e busca por discrepâncias entre a sua abstração e a documentação do artefato. O principal objetivo dessa atividade consiste em produzir uma lista de discrepâncias para serem analisadas e classificadas posteriormente entre defeitos ou falso positivos.

- *Coleção dos Defeitos* – o *moderador* reúne as listas de discrepâncias produzidas pelos *inspetores* e cria uma lista única, eliminando discrepâncias duplicadas (encontradas por mais de um *inspetor*).

- *Discriminação dos Defeitos* – o *moderador*, o *autor* do artefato e os *inspetores* discutem as discrepâncias de forma assíncrona. Durante essa discussão, as discrepâncias são classificadas como defeito ou falso positivo. Os falsos positivos são descartados e os defeitos são registrados em uma lista de defeitos, que então é passada para o *autor* para que sejam feitas as devidas correções.

Kalinowski (2004) apresenta uma representação visual para o processo sugerido por Sauer et al. (2000), que é apresentada na Figura 2.5.

Esse processo, por não depender de reuniões presenciais ou sincronismo entre os inspetores, permite um número maior de membros na equipe de inspeção, aumentado a probabilidade de se encontrar defeitos que, dependendo do nível de dificuldade, não seriam encontrados por uma equipe de inspeção menor (LANUBILE; MALLARDO, 2002). Kalinowski (2004) argumenta que um grande número de inspetores na equipe tem efeito no custo, aumentando-o, uma vez que é necessário financiar a participação de mais pessoas. Porém não tem efeito no tempo de detecção de defeitos e também não implica em problemas de coordenação.

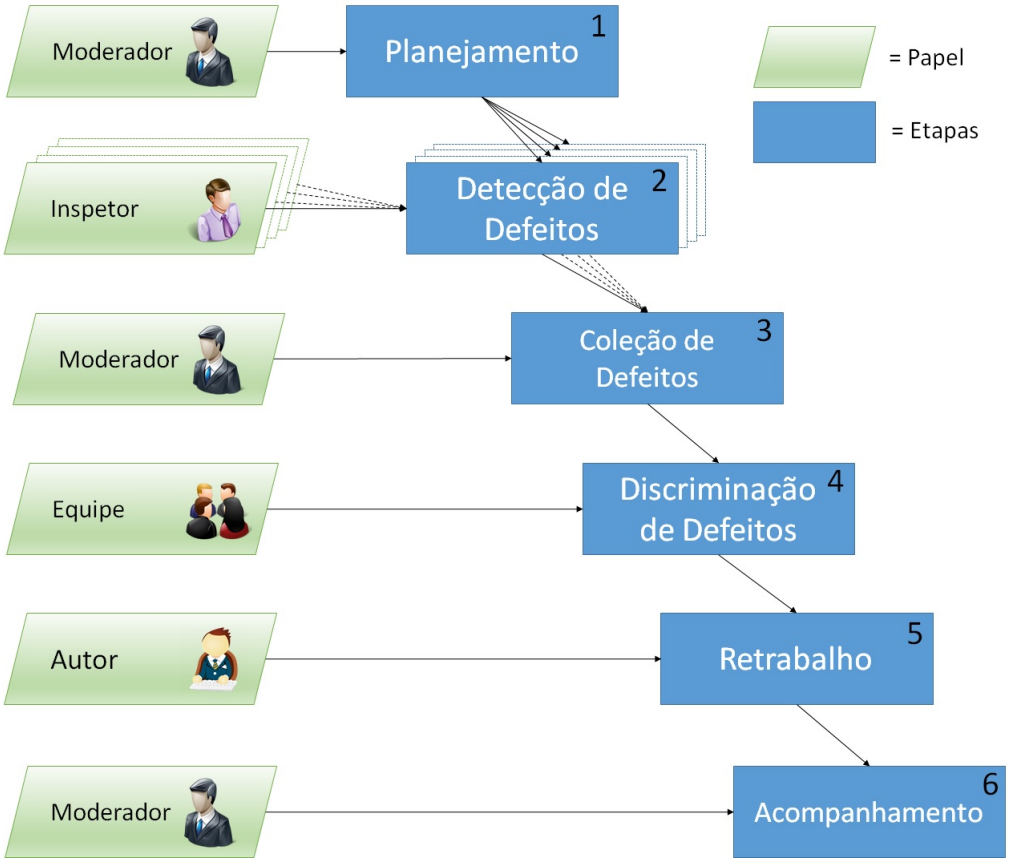


Figura 2.5: Visão do processo de inspeção proposto por Sauer et al. (2000). Adaptado de (KALINOWSKI, 2004).

2.5 Inspeção de Código

O principal objetivo da inspeção de software é aumentar a qualidade do software, detectando defeitos em artefatos tão logo eles tenham sido inseridos. Um dos principais artefatos investigados através de inspeção é o código fonte (HERNANDES; BELGAMO; FABBRI, 2013; LANUBILE; MALLARDO, 2003). Quando se trata de garantir a qualidade do código fonte, existem duas técnicas: o teste e a inspeção. O teste, por ser uma atividade dinâmica, é realizado através da execução do código, geralmente guiado por casos de teste. Assim, como é necessário executar o código, ele já foi totalmente implementado. Por outro lado, quando se usa a inspeção, não é necessário que todo o código já tenha sido implementado e esteja pronto para execução (WINKLER; THURNHER; BIFFL, 2007), podendo ser realizada mesmo em um código fonte incompleto.

As inspeções são baseadas na análise estática e não na execução do código. Com isso, segundo Kothari et al. (2004), pode-se economizar uma quantia significativa de recursos, uma vez que as inspeções podem identificar defeitos no código antes que os defeitos causem falhas no software.

Além de encontrar defeitos no código e nos documentos relacionados, inspeções podem ajudar a verificar outros fatores, como por exemplo (PARNAS; LAWFORD, 2003):

- Determinar se as diretrizes de estilo de codificação estão sendo seguidas pelos programadores;
- Os comentários no código são relevantes e de tamanho apropriado;
- A convenção de nomes é clara e consistente;
- O código é legível, de fácil compreensão e de fácil manutenção.

Um ponto muito importante para uma boa inspeção de código é conseguir compreendê-lo muito bem. De acordo com Runeson e Andrews (2003), para a detecção e o isolamento de defeitos no nível do código é necessário compreender a intenção e o real comportamento do programa.

De acordo com Walkinshaw, Roper e Wood (2005), a compreensão de código é uma atividade chave que apoia várias atividades de engenharia de software como, por exemplo, manutenção, testes e inspeções. Segundo Vinz e Etzkorn (2006), a compreensão de código envolve o processo de extrair propriedades de um código para se ter um melhor entendimento do que ele faz. Por exemplo, se um programador tem que realizar uma tarefa de manutenção em um software (que provavelmente não foi escrito por ele), é necessário entender quais partes do código serão inclusas na manutenção. Se a tarefa resultar em mudanças, é preciso compreender muito bem o código para saber como a mudança irá afetar o resto do sistema.

Softwares grandes, que evoluem durante décadas, fazem da manutenção a atividade mais cara e demorada do ciclo de desenvolvimento de software por causa da atividade de compreensão. Neginhal e Kothari (2006) estimam que o esforço para compreender o código fonte e suas relações consome aproximadamente 90% do tempo quando se faz uma atividade de manutenção.

À medida que o tempo passa, as tarefas de manutenção e compreensão, que estão intrinsecamente ligadas à qualidade do código, se tornam mais complexas e caras (RILLING; KLEMOLA, 2003). Ou seja, um projeto com métodos de programação desestruturados e manutenções mal gerenciadas, contribuem bastante para uma qualidade do código ruim, o que irá afetar diretamente a sua compreensão.

Para realizar inspeções de códigos fonte são utilizadas técnicas de leitura que auxiliam na atividade de compreensão. Uma dessas técnicas é a *Stepwise Abstraction* (SA). Além disso,

a utilização de técnicas de visualização também interfere positivamente na atividade de compreensão de códigos fonte, tendo a técnica *Treemap* como exemplo. Ambas as técnicas são apresentadas nas subseções seguintes.

2.5.1 Técnica de leitura Stepwise Abstraction

Existem estudos que apontam para a necessidade de uma abordagem sistemática para a atividade de compreensão de código fonte caso se deseje encontrar o maior número de defeitos possível, principalmente para os trechos de código de alta complexidade cognitiva (RUNESON; ANDREWS, 2003).

Uma abordagem sistemática para entender o código fonte tem como base analisar a sua estrutura e construir uma representação de alto nível para o mesmo. Muitas técnicas tentam padronizar abstrações derivadas de comentários inseridos no código fonte e/ou nomes de variáveis. Outros métodos tentam entender o código fonte repetindo regras de transformações para derivar conceitos abstratos que representem trechos de código (VINZ; ETZKORN, 2006).

Uma das técnicas de leitura existentes é a Stepwise Abstraction (LINGER; WITT; MILLS, 1979), a qual pode ser considerada como uma técnica de compreensão sistemática, uma vez que existe um processo bem formalizado para a criação de abstrações dos trechos do código.

A Stepwise Abstraction (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. A SA consiste em realizar leituras do código fonte, a partir das quais os revisores escrevem sua própria especificação para o programa. As abstrações são obtidas parcialmente, sendo que, inicialmente, os trechos de código mais internos são abstraídos, passando para as estruturas mais externas. A Figura 2.6 mostra um exemplo da técnica sendo aplicada na compreensão de um código fonte.

O primeiro passo é identificar as instruções mais internas no código fonte, ou seja, instruções 5.4.2.2.1, 5.4.2.2.2, 5.4.2.3.1, 5.4.2.3.2, representadas na coluna 1 da Figura 2.6. Essas instruções são abstraídas pelo leitor, o qual escreve com suas palavras o que cada instrução no código representa. No próximo passo, representado pela coluna 2, o leitor escolhe novamente as instruções mais internas que ainda não foram abstraídas, nesse caso, instruções 4.5.3.1, 4.5.3.2, 4.5.3.3, 4.5.3.4, 5.4.2.1, 5.4.2.2, 5.4.2.3, 7.4.4.1, 7.4.4.2, 8.2.2.1 e 8.2.2.2. De forma recursiva, na coluna 3 são escolhidas novamente as instruções mais internas que ainda não tiverem sido abstraídas. O processo de abstração é repetido sempre das instruções mais internas para as mais externas, até que se tenha todas as instruções e abstrações no mesmo nível, como mostra a co-

luna 5 da Figura 2.6. Nesse momento, as instruções e abstrações são reunidas em uma única abstração a qual representa a função do programa.

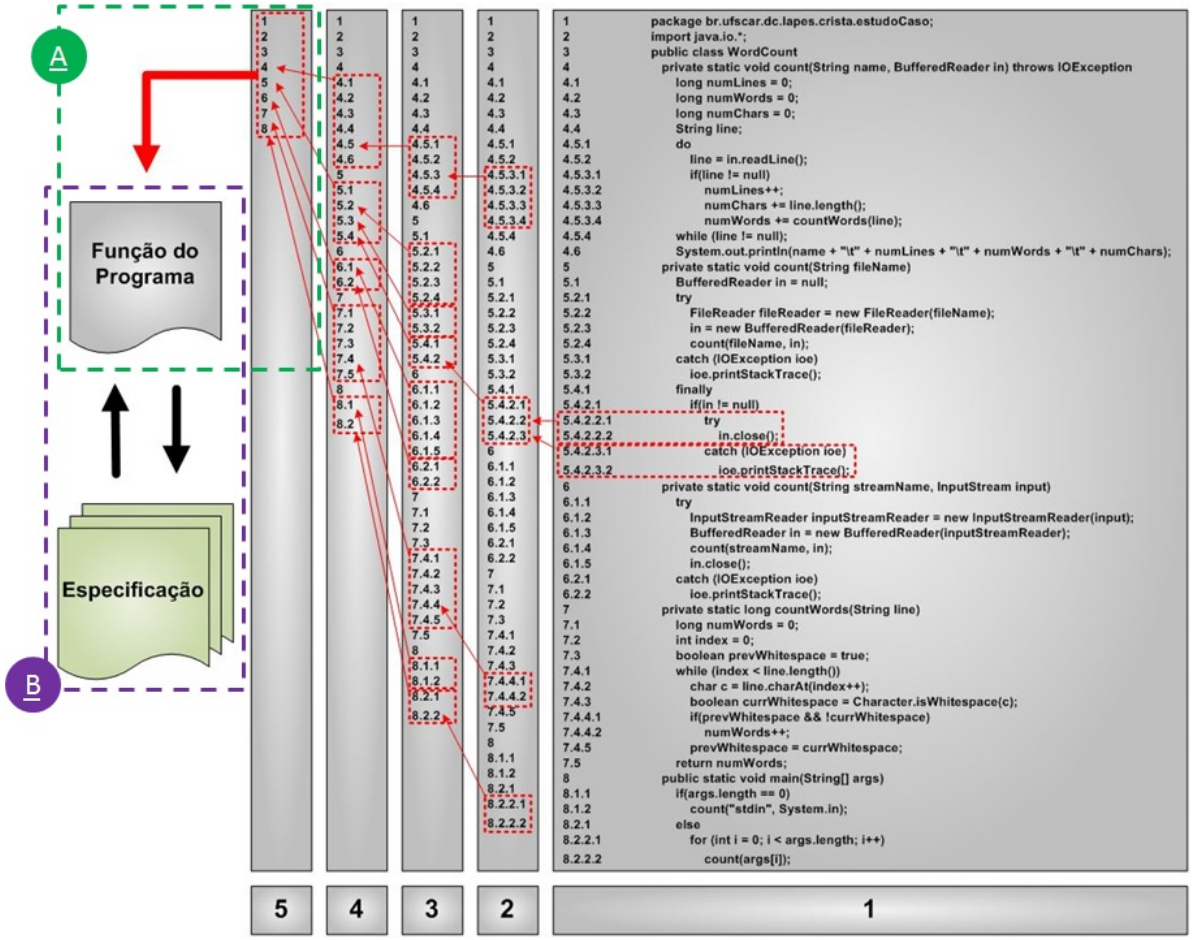


Figura 2.6: Exemplo de aplicação da técnica Stepwise Abstraction. Adaptado de (DÓRIA, 2001).

Em um processo de inspeção de código que usa a técnica de leitura Stepwise Abstraction, primeiramente o inspetor realiza a leitura do código de acordo com a técnica de leitura, e uma vez de posse da função do programa (região "A", destacada em verde na Figura 2.6), o inspetor deve compará-la com a especificação original do mesmo, a fim de identificar se existem inconsistências entre ambas (região "B", destacada em roxo na Figura 2.6). Essas inconsistências são registradas como discrepâncias e são encaminhadas ao moderador para serem discutidas na reunião de inspeção.

Embora a SA seja mais utilizada em atividades de inspeção, com a finalidade de apoiar a elaboração de uma descrição funcional do código fonte, ela é uma abordagem sistemática genérica para compreensão de código e pode ser utilizada em qualquer outra atividade que requeira o entendimento do código.

2.6 Estudos Experimentais Sobre Inspeção de Software

Ao longo das últimas quatro décadas, muito conhecimento tem sido produzido na área de inspeção de software, incluindo variantes do processo tradicional de inspeção, técnicas de estimativa da quantidade de defeitos em diferentes tipos de artefatos, técnicas de leitura de documentos visando aumentar o número de defeitos encontrados por inspetores, diretrizes para pontos de tomada de decisão do processo inspeção, dentre outros assuntos. Parte desse conhecimento tem sido avaliado em estudos experimentais (KALINOWSKI, 2004).

Hernandes, Belgamo e Fabbri (2013) realizaram um mapeamento sistemático para caracterizar o uso dos processos de inspeção de software em estudos experimentais publicados na literatura. Entre os objetivos do mapeamento sistemático destacam-se dois: identificar quais processos de inspeção de software têm sido investigados e em que tipos de artefatos eles vêm sendo aplicados. Os autores identificaram 249 estudos primários, dos quais 116 eram duplicados e dos 133 restantes, 79 foram aceitos pelos critérios de inclusão.

Como resultado da análise dos 79 artigos aceitos, visando responder à questão "*Qual processo de inspeção de software foi investigado no estudo experimental?*", tem-se que os processos de Fagan (FAGAN, 1976, 1986) foram os mais citados, somando um total de 22 ocorrências. Sua relevância fica ainda mais evidente se os processos adaptados (aqueles em que uma ou mais etapas do processo original não foram executadas) forem contabilizados (7 ocorrências). O processo proposto por Sauer et al. (2000) também merece destaque, com 10 ocorrências levando-se também em consideração suas adaptações. Os processos propostos por Graham e Gilb (1993) (3 ocorrências), HyperCode (PERPICH et al., 1997) (2 ocorrências) e N-fold (MARTIN; TSAI, 1990) (2 ocorrências) foram outros processos explicitamente citados nos artigos analisados. A Figura 2.7 mostra graficamente a distribuição de citações dos processos dentro dos 79 trabalhos analisados, lembrando que um mesmo trabalho pode citar 1 ou mais processos.

Com relação ao objetivo de se identificar os tipos de artefatos que vêm sendo investigados no contexto de inspeção de software, destacaram-se dois: documento de requisitos, com 35 ocorrências, e código fonte, com 30 ocorrências. A Figura 2.8 mostra os tipos de artefatos investigados nos estudos experimentais.

O fato de os artefatos relacionados à código fonte ocuparem um lugar de destaque entre os tipos de artefatos inspecionados (responsável por 29% do total) é muito importante para este trabalho devido ao contexto no qual ele está inserido. Porém, outro fato que merece atenção é em qual linguagem de programação esses artefatos foram desenvolvidos. Hernandes, Belgamo e Fabbri (2013) não chegaram a esse nível de detalhamento nas análises decorrentes

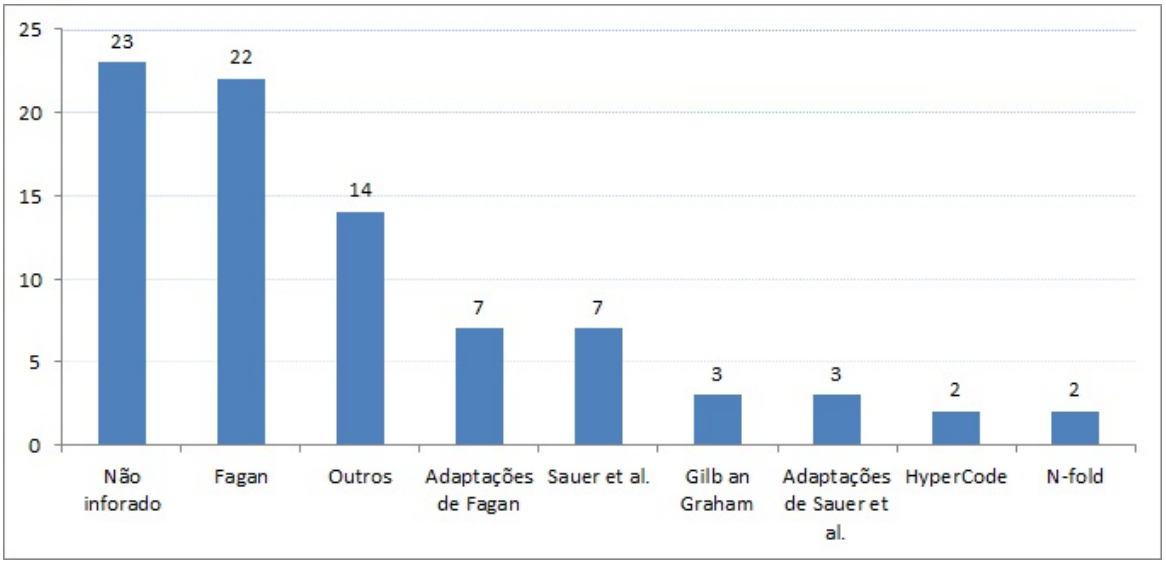


Figura 2.7: Processos de inspeção de software utilizados nos estudos experimentais. Adaptado de (HERNANDES; BELGAMO; FABBRI, 2013).

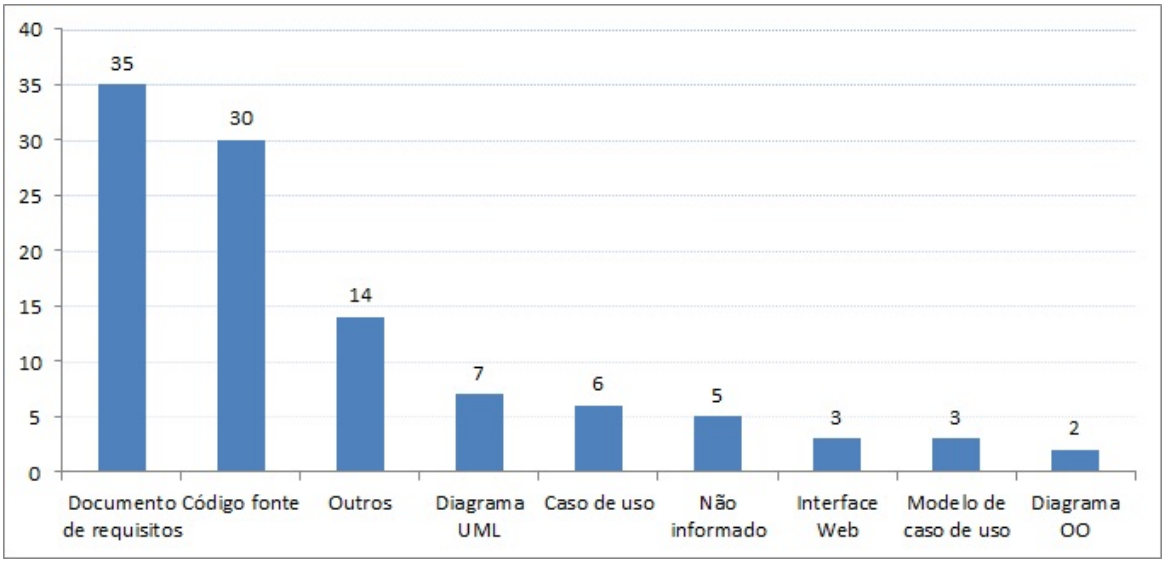


Figura 2.8: Artefatos inspecionados nos estudos experimentais. Adaptado de (HERNANDES; BELGAMO; FABBRI, 2013).

do mapeamento sistemático, porém, durante o levantamento bibliográfico realizado pelo autor deste trabalho no contexto de inspeção de software para caracterização da área, ficou bastante evidente que a grande maioria dos códigos fontes utilizados em experimentos envolvendo a atividade de inspeção foram desenvolvidos nos paradigmas de programação procedural, que tem como representantes as linguagens C e Pascal, entre muitas outras, ou programação orientada a objetos, representado principalmente pelas linguagens Java e Visual Basic.NET.

Várias pesquisas foram realizadas em máquinas de busca *online*, principalmente na Sco-

pus[1], conhecida por indexar um grande volume de trabalhos científicos. Todas as buscas tentavam relacionar dois ou mais dos seguintes termos: "*software inspection*", "*code inspection*", "*inspection*", "SQL", "*Structured Query Language*", "*database*" e "*tool*". Porém, nenhum dos trabalhos científicos retornados pelas *strings* de busca descreviam a aplicação da atividade de inspeção em artefatos relacionados a banco de dados. Isso mostra que a aplicação do processo de inspeção, no contexto de código fonte SQL, tem sido pouco explorada no meio acadêmico.

2.7 Ferramentas de Apoio ao Processo de Inspeção

Entre as ferramentas disponíveis para apoio ao processo de inspeção estão FindBugs (HOVEMEYER; PUGH, 2004), JLint (ARTHO, 2006), PMD (COPELAND, 2005), CheckStyle (BURN, 2007), JCSC (JOCHAM, 2005) e CRISTA (PORTO, 2010). Uma descrição dessas ferramentas encontra-se a seguir:

- *FindBugs* – A ferramenta FindBugs foi inicialmente desenvolvida por Hovemeyer e Pugh (2004). Ela analisa estaticamente arquivos Java compilados - bytecode - a procura de trechos de códigos fonte que possuem padrões de desenvolvimento já conhecidos que podem levar a um erro. Quando esses padrões são encontrados no código analisado, a ferramenta destaca o trecho de código em questão e apresenta um aviso ao usuário, indicando qual defeito pode ter sido inserido àquele trecho de código. FindBugs contempla uma série de padrões de códigos defeituosos incluindo, por exemplo, referência a objetos não inicializados, não verificação de argumentos nulos por parte dos métodos e conversões implícitas entre diferentes tipos de objetos. Esses padrões podem ser classificados em diferentes gêneros, tais como: não utilização de boas práticas de programação, vulnerabilidade do código, falta de padronização do código, performance, etc. A Figura 2.9 mostra o resultado de uma análise realizada pela ferramenta.

- *JLint* – É uma ferramenta de inspeção automática de código Java. Apesar de não possuir interface gráfica, seu aprendizado é muito fácil. Jlint, assim como Findbugs, lê os arquivos Java compilados e procura de padrões de código que induzam a error já conhecidos. A ferramenta JLint constrói, com base no código fonte, o fluxo do programa analisado e verifica trechos de códigos inacessíveis ou que podem entrar em repetições infinitas. Os padrões verificados pela JLint podem ser classificados nos seguintes gêneros: falhas de sincronização e no fluxo do programa. Além de sua execução ser muito rápida, JLint requer pouco esforço por parte do desenvolvedor, uma vez que os defeitos são encontrados

[1] http://www.scopus.com/

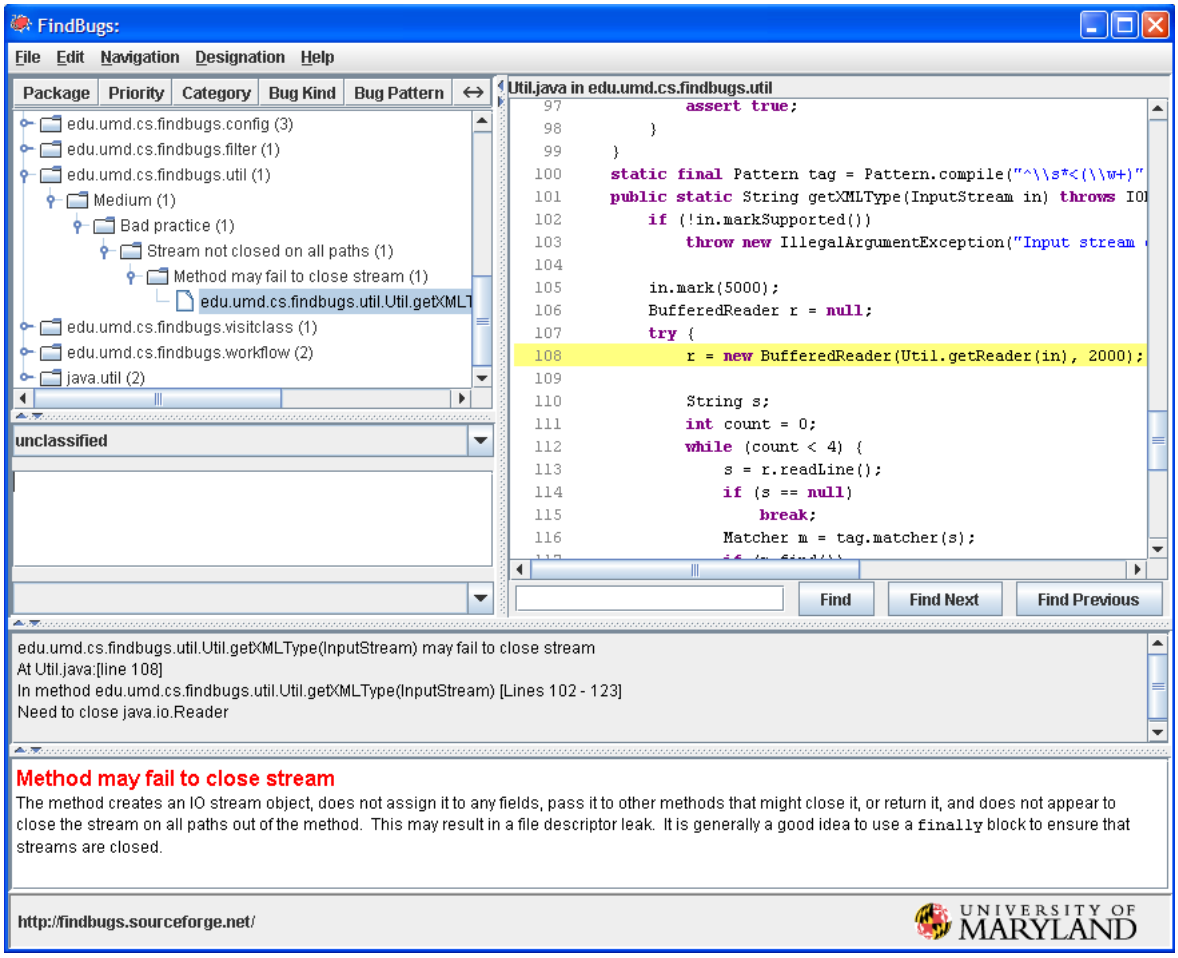


Figura 2.9: Exibição dos resultados após a análise da ferramenta FindBugs (HOVEMEYER; PUGH, 2004).

de forma automática (ARTHO, 2006). A Figura 2.10 mostra o resultado de uma análise feita utilizando-se a JLint.

- *PMD* – A PMD, desenvolvida por Copeland (2005), é uma ferramenta de inspeção de código que tem como objetivo encontrar trechos de códigos defeituosos, através da comparação com uma base de dados de defeitos já conhecidos. Os principais padrões de defeitos encontrados pela PMD são: trechos de código inacessíveis, lógicas de baixa performance e expressões demasiadamente complexas, o que pode esconder diversos defeitos, *deadlock*, conversões inadequadas entre tipos de objetos, etc. Diferentemente das ferramentas Findbugs e JList, PMD analisa arquivos de código Java não compilados, diretamente na IDE (*Integrated Development Environment*), sendo instalada como um *plugin*. A Figura 2.11 mostra a utilização da ferramenta PMD instalada como um *plugin* na IDE Eclipse.

- *CheckStyle* – A ferramenta CheckStyle, desenvolvida por Burn (2007), impõe ao desenvolvedor um determinado padrão de codificação. O padrão pode ser customizado de

Programa:

```
public class Deadlock {
   Object a = new Object();
   Object b = new Object();
   public void foo() {
      synchronized (a) {
         synchronized (b) { }
      }
   }
   public void bar() {
      synchronized (b) {
         synchronized (a) { }
      }
   }
}
```

Saída:

```
Deadlock.java:13: Lock Deadlock.a is requested while holding lock Deadlock.b,
                  with other thread holding
Deadlock.java:7: Lock Deadlock.b is requested while holding lock Deadlock.a,
                  with other thread holding Deadlock.
```

Figura 2.10: Exemplo de uso da ferramenta JLint (ARTHO, 2006).

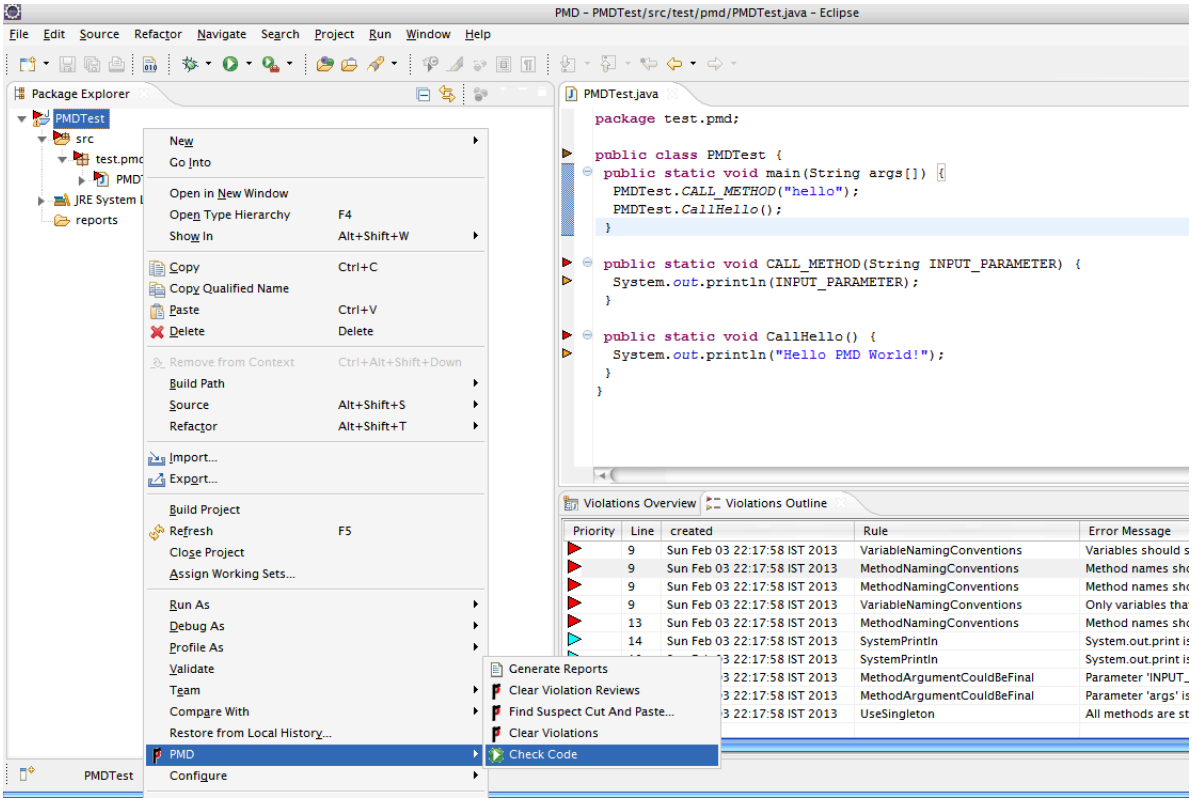


Figura 2.11: Exemplo de uso da ferramenta PMD (COPELAND, 2005).

acordo com as necessidades do projeto mas, por *defaut*, a ferramenta impõe o padrão estabelecido em *Sun Code Conventions*. Similar à ferramenta PMD, CheckStyle também analisa códigos Java não compilados na tentativa de identificar os seguintes tipos de defeitos: inconsistências na relação entre os objetos que compõem o sistema, códigos

repetidos, falta de comentários, falta de padrão na identificação de variáveis, métodos, campos e classes, etc. Assim como a PMD, a ferramenta CheckStyle também é um *plugin* que pode ser utilizado junto às IDEs, como por ser observado na Figura 2.12.

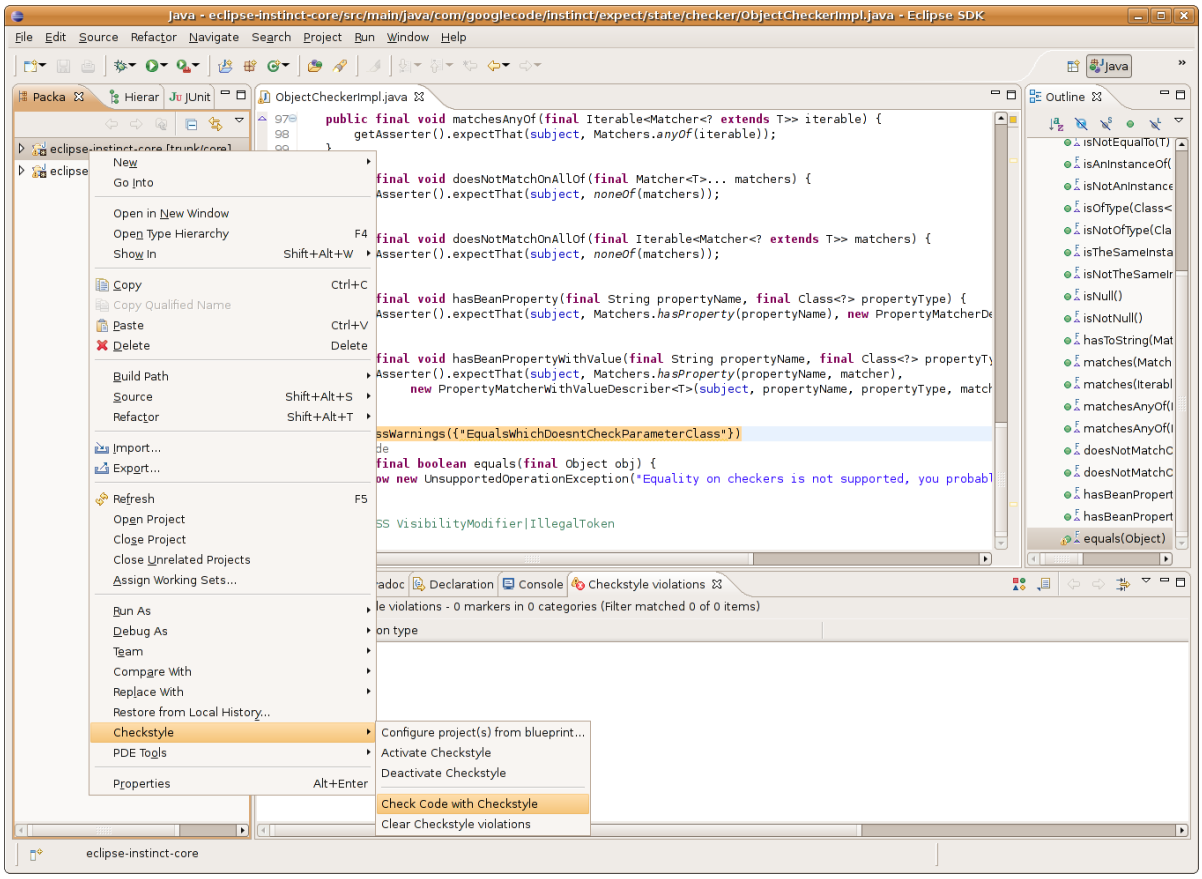


Figura 2.12: Exemplo de uso da ferramenta CheckStyle (BURN, 2007).

- *JCSC* – Desenvolvida por Jocham (2005), a ferramenta JCSC, assim como a CheckStyle, força o desenvolvedor a seguir um determinado padrão de codificação e, ao mesmo tempo, verifica a existência de códigos potencialmente defeituosos. A ferramenta analisa arquivos Java não compilados. Entre os padrões verificados pela JCSC estão a padronização da nomenclatura de classes, interfaces, campos, parâmetros, métodos e também a estrutura dos arquivos como, por exemplo, o lugar em que cada uma dessas informações deve ser inserida dentro de uma classe. A padronização dos comentários também pode ser definida dentro da ferramenta. JCSC também inspeciona o código fonte a fim de identificar os seguintes defeitos: blocos *catch/finally* vazios, instrução *switch* sem o comando *default*, códigos com baixa performance, etc. A Figura 2.13 mostra a funcionalidade que a ferramenta FCSC oferece de customizar as regras que irão compor o padrão de desenvolvimento do código fonte.

- *CRISTA* – A ferramenta CRISTA, desenvolvida no trabalho de mestrado de outro aluno

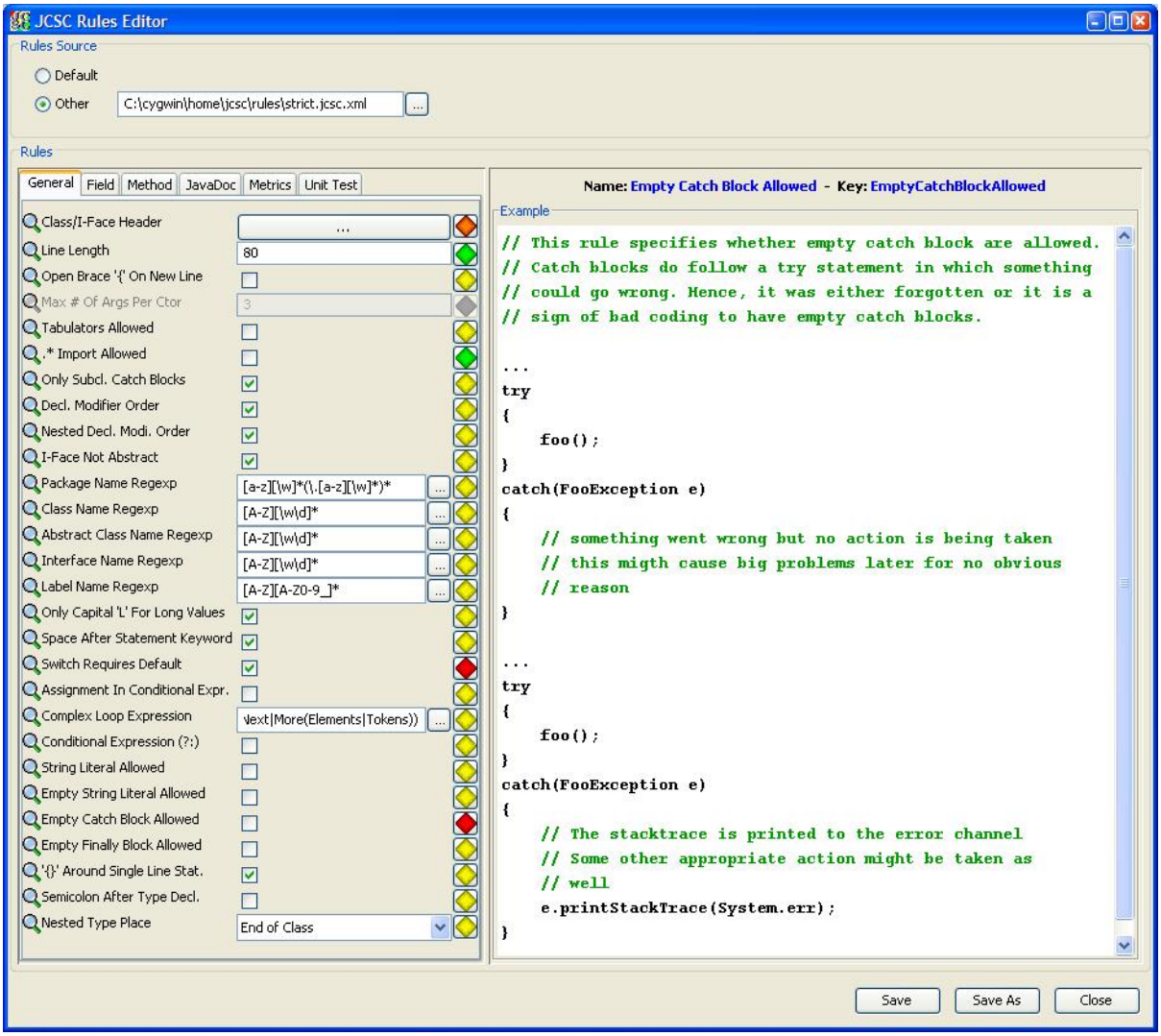


Figura 2.13: Tela da ferramenta JCSC na qual o usuário pode definir os padrões que os códigos fontes devem seguir (JOCHAM, 2005).

deste grupo de pesquisa (PORTO, 2010), tem como finalidade apoiar as etapas do processo de inspeção proposto por Sauer et al. (2000), no contexto de códigos fonte. A CRISTA implementa a técnica de leitura *Stepwise Abstraction* para auxiliar na compreensão e abstração do código e ainda cria a metáfora visual do código por meio da técnica de visualização *Treemap*. Além disso, possui a funcionalidade de registrar e manipular discrepâncias que eventualmente tenham sido detectadas pelo inspetor e que serão analisadas e classificadas pelo moderador e outros membros da equipe de inspeção. Em sua primeira versão, a ferramenta CRISTA suportava inspeções de códigos desenvolvidos na linguagem JAVA, C, C++ e COBOL. A Figura 2.14 mostra a principal tela da ferramenta CRISTA, dividida em três regiões: região "A", que contém a metáfora visual do código fonte criada através do algoritmo *Treemap*; a região "B" contém o código fonte; e a região "C" contém o espaço reservado para descrever as abstrações feitas pelo inspetor em cada

instrução do código fonte.

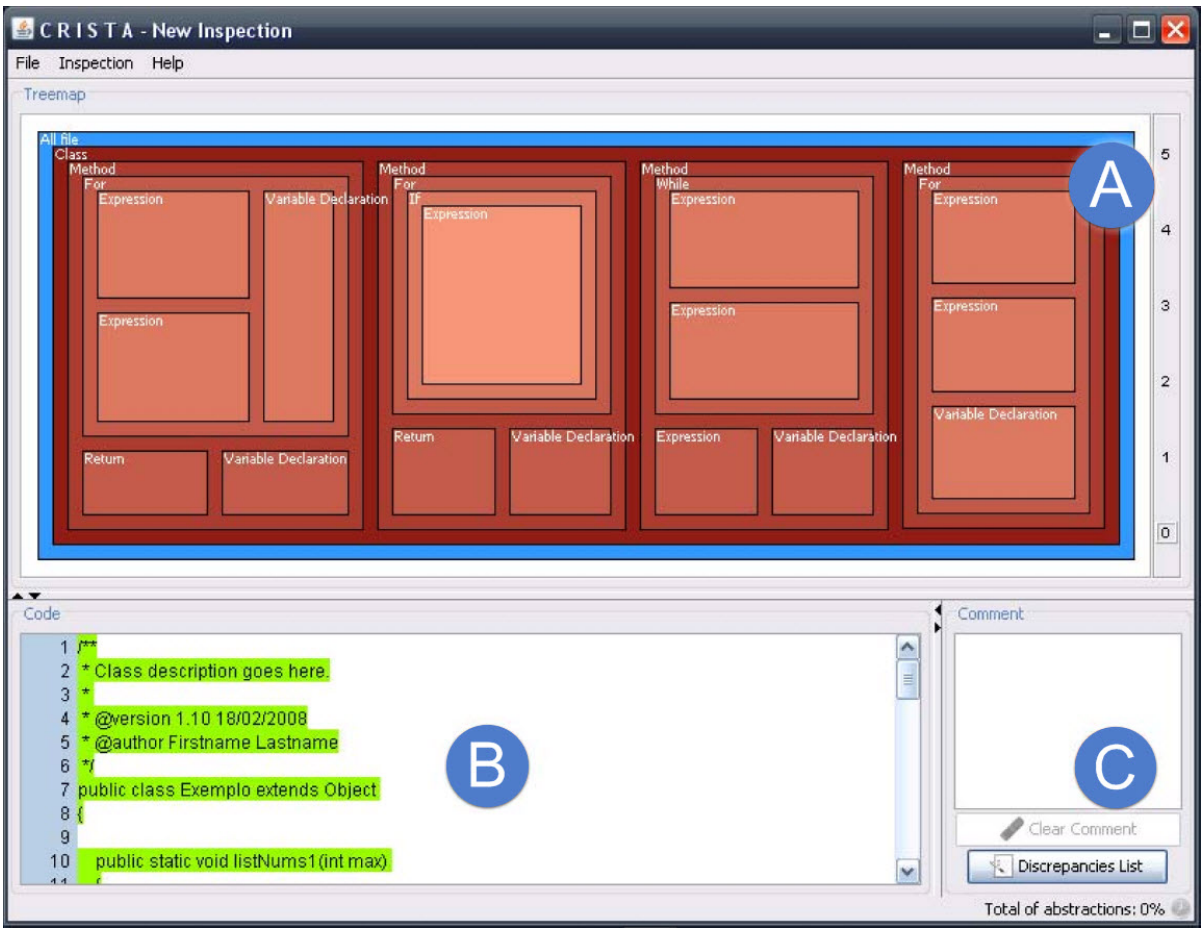


Figura 2.14: Tela da ferramenta CRISTA com a metáfora visual *Treemap* (região "A"), código fonte (região "B") e registro das abstrações (região "C"). Adaptado de (PORTO, 2010).

Thung et al. (2015) realizaram um estudo experimental com o objetivo de comparar a eficácia entre as cinco primeiras ferramentas citadas nesta seção (FindBugs, JLint, PMD, CheckStyle e JCSC), utilizando-as para inspecionar o código fonte de três programas de código aberto (Lucene, Rhino e AspectJ). Como resultado, Thung et al. (2015) identificaram que a maioria dos defeitos poderiam ser parcialmente ou totalmente identificados através da combinação das análises das cinco ferramentas de inspeção analisadas. Além disso, verificou-se que as ferramentas FindBugs e PMD tiveram os menores índices de falso negativos. Outro resultado importante foi que, na maioria das vezes, os relatórios disponibilizados pelas ferramentas apresentavam informações genéricas, fazendo com que fosse necessário uma inspeção manual por parte do desenvolvedor.

Com relação à ferramenta CRISTA, Belgamo et al. (2014) realizaram um estudo experimental para avaliar a eficiência e eficácia de seu uso na detecção de defeitos em códigos Java, quando comparado com a abordagem *adhoc*. Os resultados revelaram que o uso da ferramenta,

que implementa a técnica de leitura *Stepwise Abstraction* e a técnica de visualização *Treemap*, trouxe ganhos à performance da atividade de detectar defeitos.

2.8 Considerações Finais

O objetivo de inspeções de software é melhorar a qualidade de artefatos de software através de sua análise, detectando e corrigindo defeitos antes que os artefatos sejam passados para as próximas fases do processo de desenvolvimento de software. Como visto na seção 2.3, a aplicação de inspeções pode trazer diversos benefícios ao projeto de desenvolvimento de software.

O processo de inspeção foi inicialmente proposto por Fagan (1976). Trabalhos científicos e estudos experimentais publicados desde então, visando avaliar esse processo, indicaram reuniões assíncronas como uma evolução do processo. Tendo em vista as inspeções com reuniões assíncronas, Sauer et al. (2000) apresentaram uma reorganização do processo de inspeção com mudanças que visam reduzir o custo e o tempo total da realização da inspeção de software.

Há uma grande quantidade de trabalhos científicos descrevendo e avaliando a atividade de inspeção de software na literatura. Muitos deles relatam a atividade no contexto de inspeção de código fonte, porém a grande maioria limita-se a códigos escritos em linguagens de programação procedurais ou orientadas a objetos. As buscas realizadas durante o levantamento bibliográfico para a escrita desse capítulo não trouxeram trabalhos que descrevessem ou avaliassem a atividade de inspeção no contexto de código SQL. De forma semelhante, as ferramentas computacionais de apoio ao processo de inspeção existentes atualmente limitam-se a linguagens de programação procedurais ou orientadas a objetos.

Nesse cenário, identificou-se ser possível elaborar uma abordagem sistemática, com apoio computacional ao processo de inspeção de software proposto por Sauer et al. (2000), no contexto de código SQL, contando com o apoio da técnica de leitura *Stepwise Abstraction* na compreensão do código e apoio ferramental da CRISTA. A proposta em questão está descrita no Capítulo 3.

Capítulo 3

SISTEMATIZAÇÃO DA INSPEÇÃO DE CÓDIGO SQL COM SUPORTE COMPUTACIONAL

Este capítulo apresenta um survey realizado para identificar como a inspeção de código SQL se dá no âmbito empresarial e quais são as maiores dificuldades enfrentadas. Com base nos resultados e na experiência do autor, é apresentada uma proposta de sistematização da atividade de inspeção de código SQL, utilizando a técnica de leitura Stepwise Abstraction e o apoio computacional da ferramenta CRISTA.

3.1 Considerações Iniciais

O processo de inspeção é uma abordagem bem estruturada e sua aplicação segue passos bem definidos (FAGAN, 1976). Segundo Jones (1996), a inspeção é a atividade de VV&T de maior eficácia para a detecção de remoção de defeitos. Entre os benefícios em se utilizar inspeção, que são vários, destacam-se : i) possibilidade de detectar e remover defeitos em diferentes tipos de artefatos na mesma fase de desenvolvimento que são inseridos (WINKLER; THURNHER; BIFFL, 2007); e ii) construir e manter a documentação do sistema atualizada (DENGER; SHULL, 2007). Porém, apesar da eficácia da inspeção, se ela for aplicada de forma manual, sem o apoio de técnicas de leitura ou apoio computacional, ela se torna uma atividade demorada, propensa a erros e muito trabalhosa.

Como relatos e avaliações da aplicação do processo de inspeção em artefatos relacionados a bancos de dados relacionais, mais especificamente códigos fontes desenvolvidos na linguagem SQL, são escassos na literatura, optou-se em realizar uma investigação do tipo *survey* para identificar como o processo de inspeção de código SQL é conduzido atualmente no ambiente empresarial, mapeando as principais dificuldades em se realizar tal atividade e qual o impacto

que defeitos em códigos SQL podem trazer ao sistema de software como um todo.

Os resultados obtidos por meio do *survey* foram devidamente analisados e forneceram apoio para tomadas de decisões em relação à construção da proposta de sistematização da inspeção de código SQL apresentada neste capítulo.

O restante deste capítulo está organizado da seguinte forma: Na Seção 3.2 é apresentado o *survey* realizado para caracterizar a atividade de inspeção de código SQL no ambiente empresarial. A Seção 3.3 apresenta a proposta de sistematização da inspeção de código SQL com apoio computacional da ferramenta CRISTA. Na Seção 3.4 é apresentado um exemplo detalhado de uma inspeção de código SQL usando a ferramenta CRISTA. Já na Seção 3.5 são apresentadas as considerações finais deste capítulo.

3.2 Survey

Nesta seção serão apresentados a forma como o *survey* foi planejado, uma breve descrição da amostra obtida e, por fim, o resultado das análises.

3.2.1 Planejamento

O questionário foi projetado com o apoio da ferramenta *Lime Survey* (TEAM, 2015), a qual permite organizar as questões em grupos e visualizá-las em páginas *web* distintas. O questionário foi dividido em dois grupos de questões. O primeiro grupo, relacionado ao perfil dos participantes, teve como objetivo identificar a experiência e o conhecimento dos participantes no desenvolvimento e manutenção de códigos SQL. Já o segundo grupo de questões teve como objetivo extrair como a atividade de inspeção é conduzida pelos participantes e quais as principais dificuldades encontradas para sua realização. O questionário completo pode ser visto no Apêndice A.

Como o objetivo do *survey* foi de caracterizar a atividade de inspeção no contexto empresarial, fez parte do planejamento limitar os participantes àqueles que estivessem, de alguma maneira, envolvidos com projetos dessa natureza. Para isso, o autor deste trabalho solicitou a colegas de profissão que respondessem e, quando possível, repassassem o pedido de participação a outros profissionais da área de TI.

3.2.2 Amostra Obtida

O *survey* ficou disponível por 60 dias, período no qual houve 32 acessos e desses 22 participantes responderam todas as perguntas do questionário. Após esse período o questionário foi desativado para que não houvesse novas participações e as análises pudessem ser iniciadas. Foram consideradas para análise somente as respostas dos participantes que responderam todas as questões obrigatórias do questionário.

3.2.3 Resultados do Survey

Com relação ao primeiro grupo de questões, responsável pela caracterização dos participantes, observou-se que a idade média dos participantes foi de 31 anos. Além disso, todos os participantes concluíram o ensino superior, sendo que 55% concluíram também cursos de especialização *lato sensu* ou *stricto sensu*. Sobre tempo de experiência profissional na área de TI, observou-se que 50% dos participantes possuem 10 anos ou mais de experiência (Figura 3.1 (A)). Com relação ao tempo de experiência em desenvolvimento SQL, observou-se que a maioria (45%) possui entre 3 e 6 anos de experiência. Além disso, uma porcentagem considerável dos participantes (27%) declaram possuir mais de 10 anos de experiência em desenvolvimento SQL (Figura 3.1 (B)).

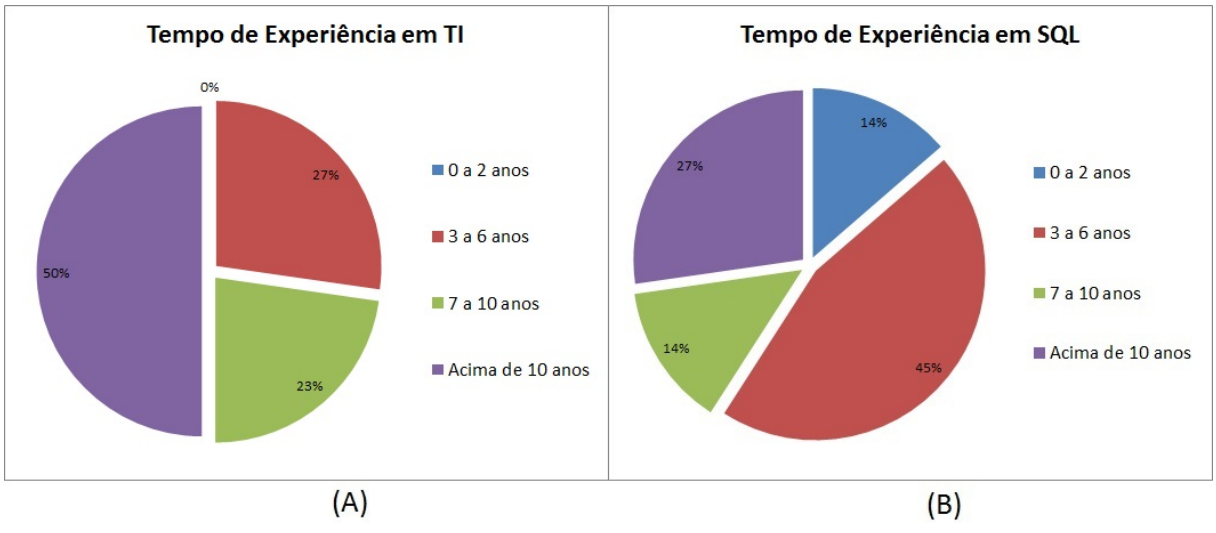


Figura 3.1: Distribuição dos participantes do *survey* por tempo de experiência em TI (A) e em desenvolvimento SQL (B).

No que diz respeito a utilização dos diferentes bancos de dados, os participantes puderam escolher 1 ou mais bancos de dados que já haviam tido algum contato, dentro de uma lista pré-definida. Além disso, havia-se a possibilidade de cadastrar outro, caso não estivesse na lista. A análise dos dados, consolidada na Figura 3.2, mostra que o banco de dados “Microsoft

SQL Server" (utilizado por 77% dos participantes) é o banco de dados mais utilizado, seguido de perto pelos bancos de dados "MySQL" (utilizado por 72% dos participantes) e "Oracle" (utilizado por 64% dos participantes).

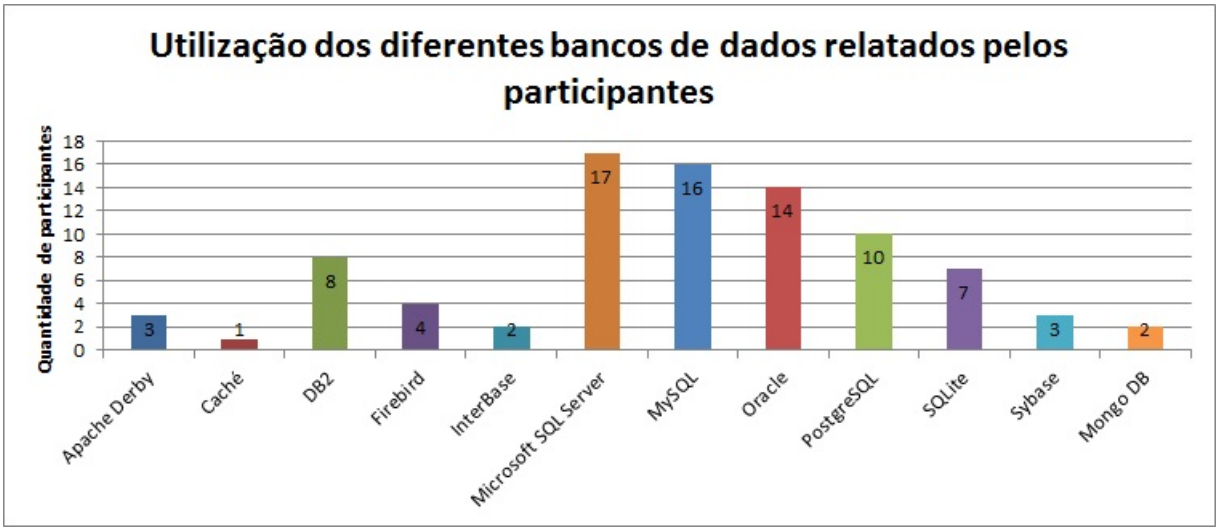


Figura 3.2: Utilização dos diferentes bancos de dados relatados pelos participantes do *survey*.

Com relação ao segundo grupo de questões, responsáveis por identificar como a atividade de inspeção é conduzida pelos participantes e quais as principais dificuldades encontradas para sua realização, observou-se que para 36% dos participantes, a dificuldade de inspecionar códigos SQL é maior em relação a códigos de linguagens de programação orientada a objetos (POO). Outros 32% declaram que possuem o mesmo nível de dificuldade e 32% acham que a inspeção em código SQL é mais fácil (Figura 3.3 (A)). Sobre os documentos auxiliares utilizados durante a atividade, 41% dos participantes declaram usar o MER (Modelo Entidade Relacionamento), 36% não utilizam documentos auxiliares, 9% utilizam os comentários inseridos no código, outros 9% voltam ao documentos de requisitos e 5% utilizam o plano de execução da *query* como apoio (Figura 3.3 (B)).

Quando questionados sobre quais defeitos são mais comumente encontrados em códigos fonte SQL, houve uma variedade bastante grande nos relatos, mas alguns defeitos se destacaram pela quantidade de vezes citados. São eles: i) *queries* mal construídas e com baixa performance, ocasionando *timeouts* (10 ocorrências); ii) defeitos na lógica de implementação (8 ocorrências); iii) defeitos nos relacionamentos entre objetos SQL por meio de *joins* ou *unions* (5 ocorrências); e iv) criação de tabelas com tipos de dados incorretos (4 ocorrências).

Uma vez questionado quais defeitos são encontrados com maior frequência em códigos SQL, queria-se mensurar qual o impacto desses defeitos no sistema computacional como um todo. Para isso, os participantes foram questionados, em termos de porcentagem, quantas falhas observadas no software têm sua origem em defeitos no banco de dados. A Figura 3.4 mostra a

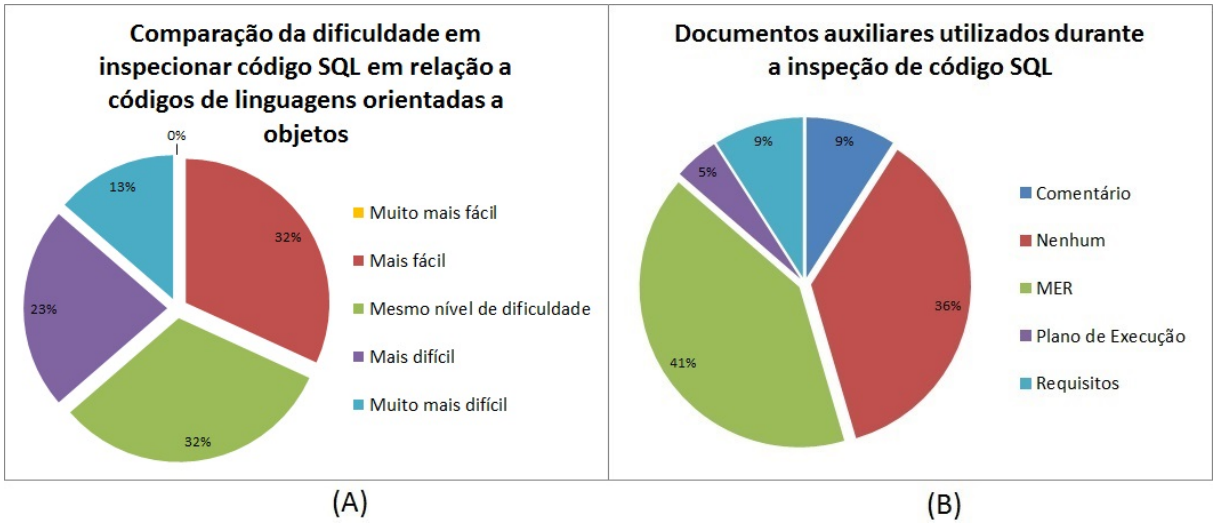


Figura 3.3: Comparação da dificuldade em inspecionar código SQL em relação a POO (A) e documentos utilizados como apoio à inspeção (B).

distribuição das respostas. A maioria dos participantes responderam que entre 11% e 30% das falhas observadas no software estão relacionadas a defeitos inseridos em objetos SQL.

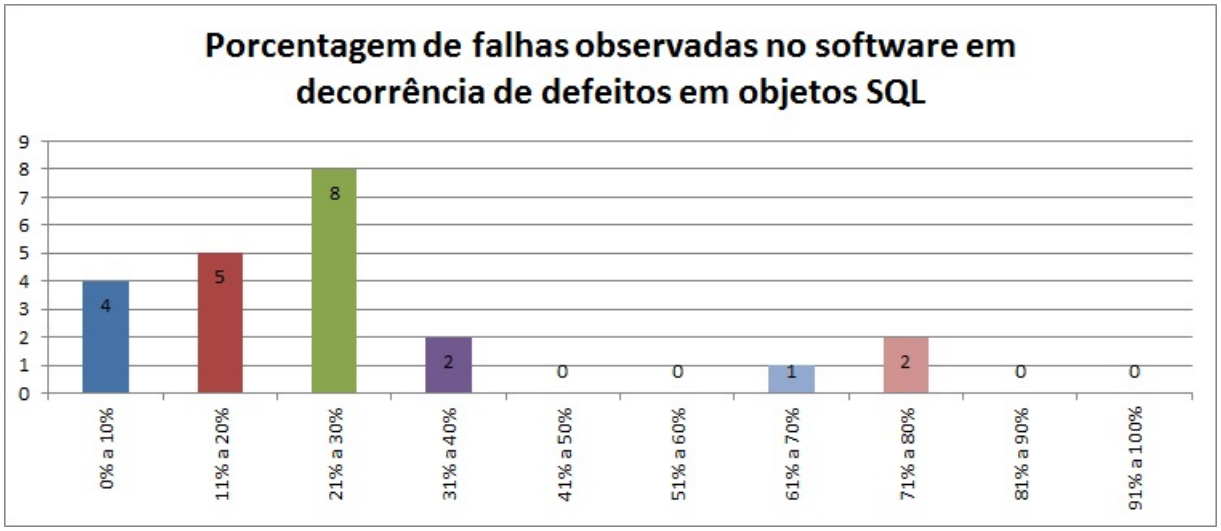


Figura 3.4: Porcentagem de falhas observadas no software em decorrência de defeitos em objetos SQL.

No que diz respeito aos passos executados para encontrar o código defeituoso, uma vez que se descobriu uma falha no software relacionado ao banco de dados, não foi possível estabelecer um padrão nas respostas dos participantes. Mesmo assim, observou-se que a depuração e a análise dos *logs* de execução do banco de dados foram as atividades mais relatadas pelos participantes.

Foi perguntado também quais são os principais fatores que dificultam a compreensão de código SQL, uma vez que se detecte que aquele código é defeituoso. Havia 6 opções já lista-

das aos participantes e ainda a possibilidade de inserir algum outro fator que não estivesse na lista, sendo que mais de uma opção poderia ser selecionada. De acordo com as análises, os 3 principais fatores são: i) falta de comentários no código (19 ocorrências); ii) falta de documentação externa como, por exemplo, documento de requisitos, MER, etc. (18 ocorrências); e iii) quantidade de objetos SQL envolvidos na consulta (16 ocorrências). A Figura 3.5 mostra a lista completa de ofensores, bem como a quantidade de ocorrências de cada um deles.

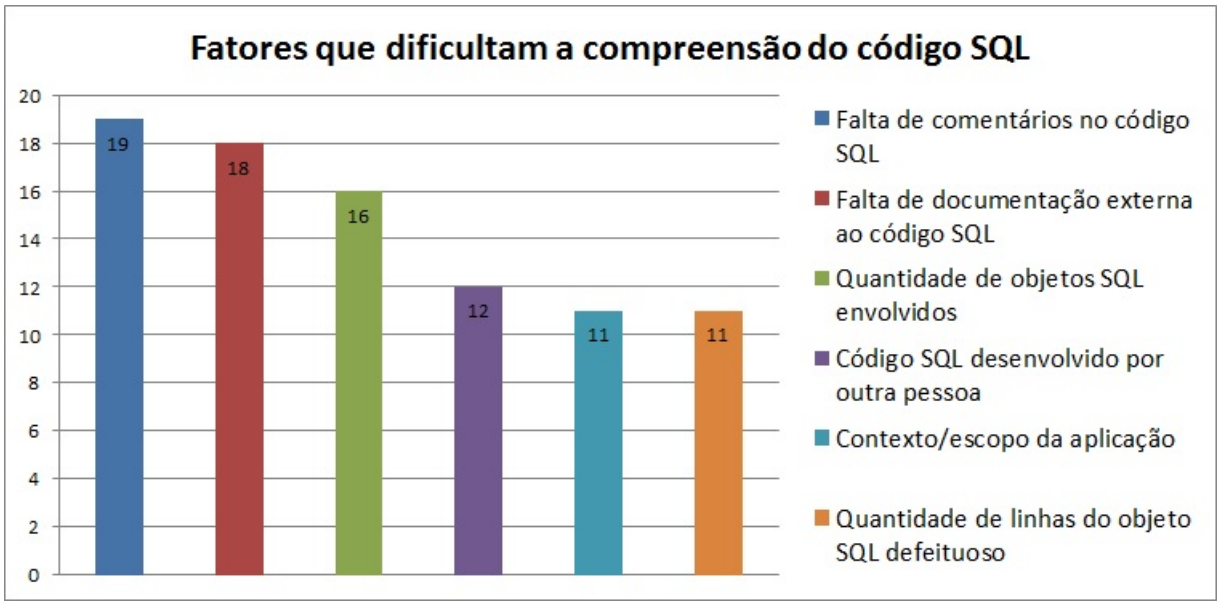


Figura 3.5: Fatores que dificultam a compreensão do código SQL durante a atividade de inspeção.

3.2.4 Discussão

A principal observação obtida a partir das análises das respostas dos participantes é que, mesmo entre profissionais experientes, não há uma padronização na maneira de se localizar defeitos em códigos SQL nem como o código defeituoso, uma vez localizado, é compreendido. Ou seja, a atividade de inspeção em códigos fonte SQL são realizadas de forma *adhoc*, o que pode diminuir consideravelmente sua eficiência e eficácia. Essa observação pode estar relacionada a dificuldade em se realizar a atividade de inspeção de código SQL, uma vez que mais da metade dos participantes declararam que inspecionar código SQL é mais ou muito mais difícil do que inspecionar códigos de programação orientados a objetos.

Outro resultado que chamou a atenção foi a lista de defeitos mais frequentes em códigos SQL. Segundo os participantes, o desenvolvimento mal feito de códigos SQL, desrespeitando boas práticas de programação e não considerando a escalabilidade do sistema, configura a principal fonte de defeitos. Vale notar que nesses casos, o resultado final do processamento pode estar correto, porém ele não é calculado em um intervalo de tempo satisfatório, gerando lentidão

no software ou, em casos mais graves, falhas por *timeout*. Esse tipo de defeito muitas vezes é tido como um *technical debt* (LI; AVGERIOU; LIANG, 2015), ou seja, algo que não inviabiliza o uso do software, mas que eventualmente precisará ser corrigido.

Com relação aos outros defeitos mencionados, foi possível fazer uma relação entre os defeitos em códigos SQL e a taxinomia de defeitos de código já conhecida para outras linguagens de programação. A Tabela 3.1 mostra uma definição resumida de cada tipo de defeito.

Tabela 3.1: Taxonomia de defeitos em códigos fonte.

TAXONOMIA	*DESCRIÇÃO*
Defeito de Inicialização	Inicialização incorreta de uma estrutura de dados. Por exemplo, atribuir um valor errado a uma variável.
Defeito de Computação	Qualquer computação incorreta para geração do valor de uma variável. Por exemplo, um operador aritmético errado é usado em uma expressão de atribuição de valor de variável.
Defeito de Controle	Causa a execução de um caminho de controle errado para um valor de entrada. Por exemplo, valores errados no controle do comando IF-THEN-ELSE.
Defeito de Interface	Quando um módulo usa ou faz suposições sobre dados que não fazem parte do seu escopo. Por exemplo, passagem de um argumento incorreto para um procedimento.
Defeito de Dados	Uso incorreto de uma estrutura de dados. Por exemplo, determinar incorretamente o último índice de um *array*.
Defeitos de Cosméticos	Erro de escrita no programa. Por exemplo, uma mensagem de informação ao usuário com erro de ortografia.

3.3 Proposta de sistematização da inspeção de código SQL

Esta seção apresenta a proposta de sistematização da inspeção de código SQL. A proposta é baseada na utilização da técnica de leitura *Stepwise Abstraction* para ler e compreender o código fonte e o apoio computacional da ferramenta CRISTA, que implementa o processo de inspeção proposto por Sauer et al. (2000).

Essa proposta foi planejada de acordo com a experiência do autor deste trabalho em realizar inspeções em código SQL de forma *adhoc* no ambiente empresarial e pelos resultados obtidos pelo *survey* apresentado na Seção 3.2. Não por coincidência, tanto a experiência do autor deste trabalho quanto os resultados da pesquisa têm vários pontos em comum, principalmente com relação à dificuldade em se realizar a atividade e quais os fatores que contribuem para isso.

As subseções a seguir correspondem às etapas do processo proposto por Sauer et al. (2000), com a descrição das atividades que devem ser executadas, por quem, e qual a importância dentro

da proposta como um todo.

3.3.1 Planejamento

Na etapa de *Planejamento*, o moderador deve definir todos os aspectos envolvidos na inspeção como, por exemplo, os artefatos que serão inspecionados e quem serão os inspetores. Uma vez definidos, os materiais são distribuídos para os inspetores e inicia-se a etapa de *Detecção de defeitos*.

Para apoio a essa etapa, a ferramenta CRISTA já fornecia funcionalidades básicas como, por exemplo, criar um novo projeto de inspeção, definir o nome do inspetor e qual artefato seria inspecionado.

Porém identificou-se que seria importante também definir uma forma de classificação das discrepâncias encontradas pelos inspetores. Nesse sentido, foram definidos dois critérios de classificação: prioridade (define a ordem de avaliação da discrepância e correção do defeito, se for o caso) e severidade (define o impacto que o defeito, caso a discrepância seja avaliada como tal, pode causar no software).

Para cada um dos critérios, foram definidos diferentes níveis. No caso da prioridade, pode-se classificá-la como *alto*, *médio* ou *baixo*. Já para a severidade, pode-se classificá-la como *crítico*, *alto*, *moderado*, *baixo* ou *insignificante*.

Por padrão, os critérios e seus níveis podem ser descritos da seguinte forma:

- *Severidade* – o impacto que um determinado defeito tem no software como um todo, caso ele seja ativado:
  - *Crítico* – o defeito causa parada total ou parcial do sistema, causando extenso corrompimento de dados e não há alternativas para se chegar ao resultado desejado;
  - *Alto* – o defeito causa parada total ou parcial do sistema, causando extenso corrompimento de dados, porém há alternativas para se chegar ao resultado desejado;
  - *Moderado* – o defeito não resulta em parada total ou parcial do sistema, mas faz com que o resultado final do processamento seja incorreto, incompleto ou inconsistente, sem alternativas para se obter o resultado desejado;
  - *Baixo* – o defeito não resulta em parada total ou parcial do sistema, mas faz com que o resultado final do processamento seja incorreto, incompleto ou inconsistente, porém existem alternativas para se obter o resultado desejado;

  - *Insignificante* – o defeito é relacionado a questões de erros ortográficos em interfaces gráficas, mensagens de informação ou em relatórios.

- *Prioridade* – a ordem que o defeito deve ser avaliado e corrigido:
  - *Alto* – o defeito deve ser corrigido imediatamente e o software não pode ser liberado sem antes corrigi-lo;
  - *Médio* – o defeito deve ser corrigido tão logo seja possível e o software não pode ser liberado sem antes corrigi-lo;
  - *Baixo* – o defeito pode ser corrigido ao longo do ciclo de desenvolvimento do software e o software pode ser liberado mesmo sem corrigi-lo.

Todas essas descrições podem ser alteradas pelo moderador de acordo com as necessidades do projeto. Além disso, podem-se definir valores numéricos para cada um dos níveis dos dois critérios, dando pesos diferentes para cada um deles. Esses pesos podem variar de acordo com o contexto da inspeção e do planejamento do moderador e serão utilizados para ordenar as discrepâncias na etapa de *Discriminação de defeitos* e ordenar os defeitos na etapa de *Retrabalho*.

Ou seja, para a sistematização da etapa de *Planejamento* do processo de inspeção, a ferramenta CRISTA passou a prover as seguintes funcionalidades, para dar suporte à proposta deste trabalho:

1. Criação de um novo projeto de inspeção;
2. Definição de um nome representativo para o projeto de inspeção;
3. Definição do inspetor que irá realizar a atividade;
4. Definição do artefato que será inspecionado;
5. Definição de como os critérios de severidade e prioridade devem ser interpretados pelos inspetores e qual o peso que cada nível dos critérios terá no ranqueamento das discrepâncias/defeitos.

Vale ressaltar que, se necessário, o moderador deve fazer reuniões de alinhamento e passagem de conhecimento entre os inspetores nessa etapa. No entanto, essas atividades não foram mapeadas na ferramenta CRISTA.

3.3.2 Detecção de Defeitos

Na etapa de *Detecção de Defeitos*, cada um dos inspetores selecionados pelo moderador inspeciona o artefato, com o objetivo de encontrar discrepâncias entre a abstração do artefato e a sua especificação original.

Para apoio a essa etapa, a ferramenta CRISTA já fornecia a implementação da técnica de leitura *Stepwise Abstraction* e a criação da metáfora visual do código fonte no formato de *Treemap*. Além disso, fornecia a funcionalidade de cadastramento de discrepâncias encontradas no código. Porém, essas funcionalidades estavam disponíveis apenas para algumas linguagens procedurais ou orientadas a objetos como, por exemplo, Java, C++ e C.

Nesse cenário, foi necessário instanciar a linguagem SQL dentro da ferramenta CRISTA. Um guia bastante detalhado de como realizar a instanciação de novas linguagens na CRISTA pode ser encontrada em (PORTO, 2010). Em resumo, a instanciação de uma nova linguagem segue o seguinte processo:

1. Construir e modificar o arquivo de definição da linguagem desejada. Esse arquivo é comumente conhecido como "JJ" devido a sua extensão e ele possui o conjunto de regras que definem como elementos básicos de uma linguagem (símbolos) podem ser combinados para formar sentenças válidas. Em outras palavras, o arquivo de definição da linguagem possui todas as regras lexicais de uma determinada linguagem.

2. Submeter o arquivo de definição da linguagem (ou arquivo "JJ") ao JavaCC (JAVACC, 2015). O JavaCC é um compilador que lê arquivos de definição de linguagem e os converte em programas Java capazes de, por exemplo, verificar se um determinado código fonte satisfaz as regras sintáticas de uma determinada linguagem de programação.

3. Empacotar as classes geradas pelo JavaCC e adicioná-las ao projeto da ferramenta CRISTA. Feito isso, a CRISTA passa a reconhecer a sintaxe dessa linguagem e cria a metáfora visual do código fonte.

Apesar de o processo parecer simples, a primeira etapa tende a ser muito trabalhosa, pois nem sempre é possível encontrar arquivos de definição de linguagens prontos. No caso da linguagem SQL não foi possível localizar tal arquivo, sendo necessário desenvolvê-lo por completo usando como base um arquivo JJ bastante simples de uma extensão da linguagem SQL conhecida como PL/SQL. Para tanto, foi necessário um estudo profundo sobre a estrutura básica da linguagem SQL e de suas regras sintáticas.

Nesse ponto foi tomada uma decisão com base nos resultados observados do *survey* apresentado na Seção 3.2: optou-se por construir um arquivo de definição da linguagem SQL para atender códigos fonte escritos para bancos de dados MS SQL Server, uma vez que ele foi o mais citado entre os profissionais participantes da pesquisa.

Uma vez que o processo de instanciação da linguagem SQL na ferramenta CRISTA havia sido concluído, foi possível criar a metáfora visual do código SQL no formato de *Treemap* e aplicar a técnica de leitura *Stepwise Abstraction* para abstração do código fonte.

No caso de uma discrepância ser detectada pelo inspetor, a ferramenta CRISTA fornecia uma funcionalidade para registrá-la, armazenando uma descrição e o trecho do código relacionado à discrepância. Para a sistematização da inspeção de código SQL, essa funcionalidade foi aprimorada, dando ao inspetor a possibilidade de classificar a discrepância em relação a dois critérios: severidade e prioridade. Ambos os critérios são definidos pelo moderador na etapa de *Planejamento*. A combinação dos dois critérios resulta em um *score* numérico, obtido a partir da soma dos pesos cadastrados pelo moderador na etapa de *Planejamento* para cada um dos níveis dos dois critérios.

O *score* é utilizado nessa etapa para destacar na metáfora visual *Treemap* os trechos de código que possuem discrepâncias. Quanto maior o *score*, ou seja, quanto mais severa e prioritária for a discrepância, mais avermelhado será o componente gráfico relacionado ao trecho do código que possui a discrepância. No sentido oposto, quanto menor o *score*, ou seja, quanto menos severa e menos prioritária for a discrepância, mais amarelado será o componente gráfico. Dessa forma, o inspetor consegue ter uma visão clara da situação do código fonte, no que diz respeito a sua qualidade, apenas observando a metáfora visual.

Todas as discrepâncias detectadas pelo inspetor são registradas e passam a compor uma lista de discrepâncias que, ao final da etapa detecção de defeitos, deverá ser enviada ao moderador. Essa lista contém todas as informações pertinentes de cada discrepância, ou seja, sua descrição, trecho de código discrepante, severidade, prioridade e *score*.

Em resumo, para a sistematização da etapa de *Detecção de Defeitos* do processo de inspeção, a ferramenta CRISTA provê as seguintes funcionalidades:

1. Implementação da técnica de leitura *Stepwise Abstraction* e da técnica de visualização *Treemap* no apoio à compreensão do código fonte;

2. Cadastramento de discrepância, registrando sua descrição, trecho do código discrepante, severidade, prioridade e um *score*, calculado a partir da combinação da severidade e da prioridade;

3. Elaboração e exportação de uma lista contendo todas as discrepâncias detectadas pelo inspetor, que deverá ser enviada ao moderador.

3.3.3 Coleção de Defeitos

De acordo com o processo de inspeção, a etapa de *Coleção de Defeitos* é caracterizada pela união de todas as listas de discrepâncias enviadas pelos inspetores ao moderador. O moderador por sua vez, deve eliminar possíveis discrepâncias duplicadas, ou seja, aquelas detectadas por mais de um inspetor.

Para apoiar essa etapa, a ferramenta CRISTA já provia a funcionalidade de importar as listas de discrepâncias e, se necessário, classificá-las como duplicadas. Porém, como não existiam os conceitos de severidade e prioridade das discrepâncias, o moderador não poderia utilizar essas informações como apoio à tomada de decisões.

Nesse cenário, foi necessário aperfeiçoar a funcionalidade de importação e união das listas para que as novas informações pudessem ser visualizadas e avaliadas como informações adicionais no momento de decisão entre adicionar a discrepância à lista final de discrepâncias ou classificá-la como duplicada.

Uma das possíveis utilizações dessas informações adicionais seria a ordenação das discrepâncias por *score*, uma vez que, provavelmente, discrepâncias iguais encontradas por mais de um inspetor teriam classificações parecidas, já que os mesmos critérios de classificação foram utilizados por todos os inspetores.

Ao final da análise de duplicidade entre as listas de discrepâncias, o moderador deve exportar uma lista final, contendo apenas discrepâncias únicas.

Em resumo, para a sistematização da etapa de *Coleção de Defeitos* do processo de inspeção, a ferramenta CRISTA provê as seguintes funcionalidades:

1. Importação das listas de discrepâncias enviadas pelos inspetores na etapa anterior;
2. Classificação das discrepâncias como válidas ou duplicadas;
3. Exportação de uma lista final de discrepâncias, sem discrepâncias duplicadas, que deve ser objeto de discussão na etapa seguinte do processo.

3.3.4 Discriminação de Defeitos

A etapa de *Discriminação de Defeitos* é composta por uma discussão entre o moderador, autor e inspetores com o objetivo de classificar cada uma das discrepâncias contidas na lista elaborada pelo moderador na etapa anterior como defeito ou falso-positivo.

Uma vez que cada discrepância possui um *score* que representa a combinação de sua severidade e prioridade, uma boa estratégia seria iniciar a discussão pelas discrepâncias de maior *score*, pois, provavelmente, essas são mais críticas dentro do processo de inspeção.

Assim como na etapa anterior, a ferramenta CRISTA possuía uma funcionalidade para importar a lista final de discrepâncias, porém não levava em conta a severidade, prioridade e o *score* de cada uma delas. Nesse cenário, aperfeiçoou-se a funcionalidade de importação da lista final de discrepâncias, a fim de que as informações relacionadas à severidade, prioridade e *score* de cada discrepância também fossem importadas e ficassem disponíveis aos membros da inspeção.

Uma vez que a lista tenha sido importada à ferramenta, pode-se ordenar as discrepâncias de acordo com sua severidade, prioridade ou *score*. Após a análise de cada uma delas, é possível manter a discrepância na lista (equivalente a classificar a discrepância como defeito) ou removê-la da lista (equivalente a classificá-la como falso positivo).

Em resumo, para a sistematização da etapa de *Discriminação de Defeitos* do processo de inspeção, a ferramenta CRISTA provê as seguintes funcionalidades:

1. Importação da lista final de discrepâncias, resultado da etapa anterior;
2. Ordenação das discrepâncias de acordo com sua severidade, prioridade ou *score*;
3. Remoção das discrepâncias que forem classificadas como falso-positivos;
4. Exportação de uma nova lista, contendo apenas as discrepâncias que realmente caracterizam defeitos e que precisam ser corrigidos na etapa seguinte do processo.

3.3.5 Retrabalho

De acordo com o processo de inspeção, na etapa de *Retrabalho* devem-se corrigir os defeitos detectados nas etapas anteriores.

Dentro da proposta de sistematização da inspeção de código SQL, uma boa estratégia é utilizar as informações de severidade, prioridade ou *score* dos defeitos para organizar as atividades

dessa etapa. Por exemplo:

- Ordenar os defeitos pelo *score*, de forma decrescente, e iniciar a correção pelos defeitos que foram classificados como mais severos e/ou prioritários;
- Ordenar os defeitos por severidade e distribuí-los aos desenvolvedores de acordo com suas experiências, ou seja, defeitos mais severos são corrigidos por desenvolvedores mais experientes, enquanto que defeitos menos severos são corrigidos por desenvolvedores menos experientes;
- Ordenar os defeitos por prioridade e distribuí-los aos desenvolvedores de acordo com a disponibilidade de cada um, ou seja, defeitos mais prioritários são corrigidos pelos desenvolvedores com maior disponibilidade, enquanto que defeitos menos prioritários são corrigidos por desenvolvedores com menor disponibilidade;

Em resumo, para a sistematização da etapa de *Retrabalho* do processo de inspeção de código fonte SQL, a ferramenta CRISTA provê a lista final de defeitos, que pode ser ordenada de acordo com as necessidades do projeto.

3.3.6 Acompanhamento

Na etapa de *Acompanhamento*, o moderador verifica se todos os defeitos detectados durante a inspeção foram devidamente corrigidos durante a etapa de *Retrabalho*.

Essa atividade de verificação também é apoiada pela lista final de defeitos, exportada pela ferramenta CRISTA ao final da etapa de *Discriminação de Defeitos*. Como a lista contém todos os defeitos detectados, pode-se usá-la como controle de quais defeitos já foram corrigidos, quais estão em andamento e quais ainda não foram distribuídos entre os desenvolvedores.

3.4 Exemplo

Esta seção apresenta um exemplo da execução da atividade de inspeção de código SQL com o apoio computacional da ferramenta CRISTA, seguindo a proposta de sistematização. Para cada etapa do processo de inspeção, serão apresentadas as funcionalidades que as apoiam.

Na etapa de *Planejamento*, o moderador cria um novo projeto de inspeção, clicando em "File / New project inspection..." (Figura 3.6, passo "1"). No passo "2" são definidos um nome para o projeto e o nome do inspetor que irá realizar a atividade. Já no passo "3" é escolhida a

linguagem de programação que os códigos fonte a serem inspecionados foram desenvolvidos e também o diretório no qual eles estão salvos.

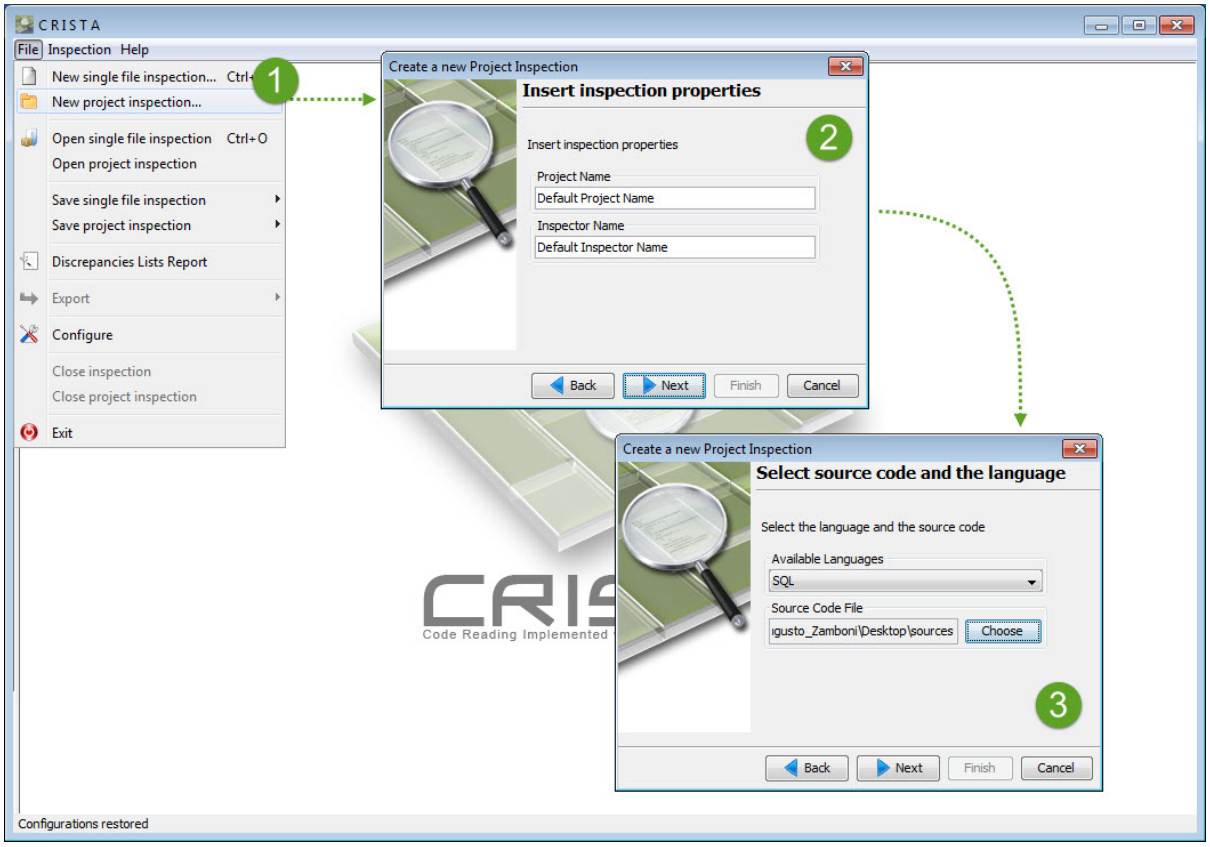


Figura 3.6: Primeiros passos da etapa de *Planejamento* utilizando a ferramenta CRISTA.

Uma vez que o projeto de inspeção tenha sido criado, o moderador define como os diferentes níveis de severidade e prioridade devem ser interpretados pelos inspetores, uma vez que é essencial que todos os inspetores utilizem os mesmos critérios no momento de classificação das discrepâncias. Além disso, os pesos de cada nível podem ser configurados de acordo com as necessidades e desejo do moderador. Essas funcionalidades são acessadas a partir do menu "File / Configure" (Figura 3.7, passo "4"). Ambas as ações são mostradas, respectivamente, nas regiões "A" e "B" da Figura 3.7.

Para encerrar a etapa de *Planejamento*, o moderador deve salvar o projeto (Figura 3.7, passo "5") e fechar a ferramenta CRISTA, como mostrado no passo "6". Nesse momento, o moderador possui um arquivo com extensão *.crista* que contém todas as definições registradas pelo moderador. Esse arquivo *.crista* deve ser entregue ao inspetor.

A partir do momento em que o inspetor recebe o arquivo *.crista*, inicia-se a etapa de *Detecção de defeitos*. Ao abrir o arquivo disponibilizado pelo moderador, a ferramenta apresenta a tela da Figura 3.8, composta das seguintes visões:

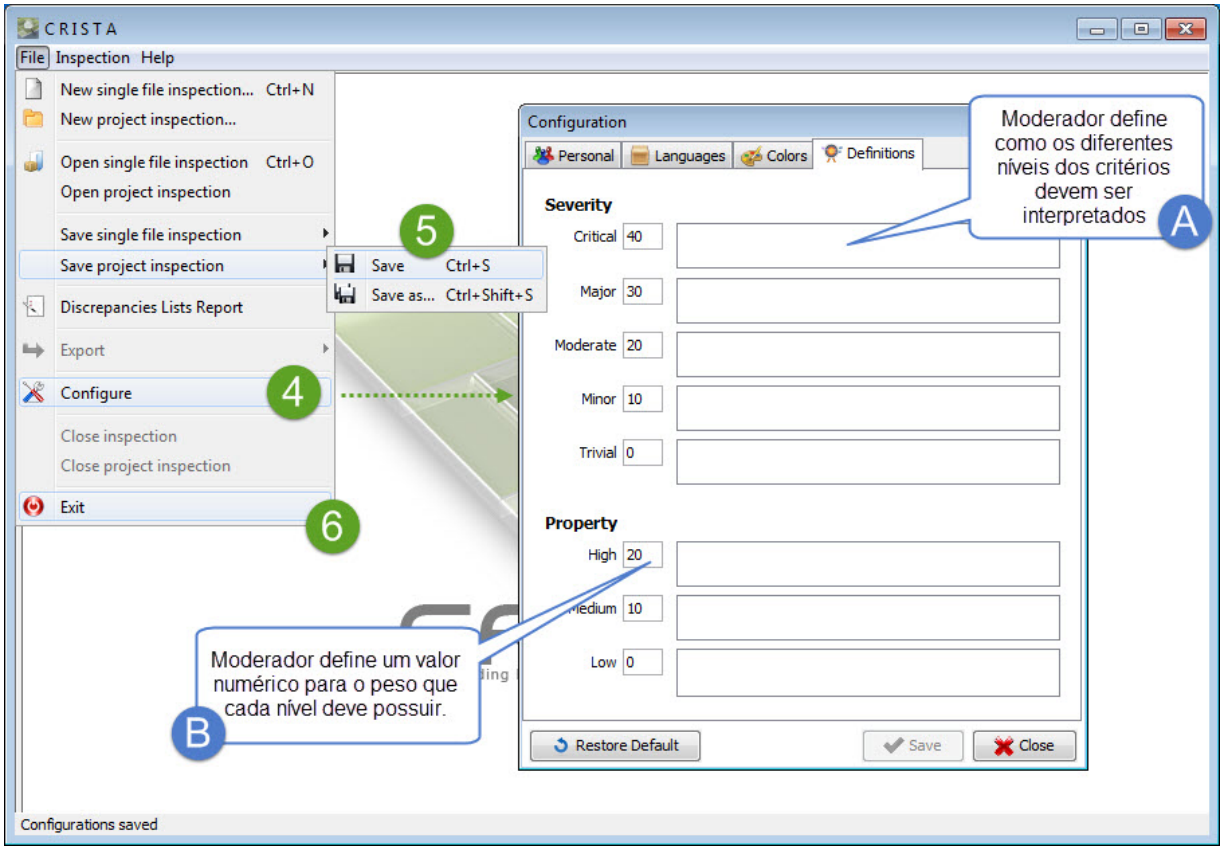


Figura 3.7: Definição dos níveis de severidade e prioridade e últimos passos da etapa de *Planejamento* na ferramenta CRISTA.

- *A* – corresponde à metáfora visual do código fonte a ser inspecionado, criada de acordo com a técnica de visualização *Treemap*;
- *B* – corresponde à caixa de texto que contém o código fonte a ser inspecionado;
- *C* – corresponde a uma caixa de texto para o inspetor inserir a abstração de cada instrução do código fonte.

As visões "A" e "B" apresentam perspectivas diferentes de um mesmo artefato. Cada instrução do código fonte da visão "B" é representado por um retângulo na visão "A". Quando o inspetor interage com a ferramenta pela visão "A", selecionando um determinado retângulo, esse retângulo é destacado com a cor azul e o trecho de código a que ele se refere é destacado na cor verde. Essa ação está representada pelo passo "1" da Figura 3.8. Nesse caso, foi selecionado o retângulo correspondente à instrução *Where* do código fonte na visão "A", que ficou destacado na visão "B". Existe também a sincronização inversa, ou seja, quando um trecho de código é selecionado na região "B", o retângulo correspondente ao trecho do código é destacado na região "A".

Inicialmente, todos os retângulos da região “A” estão pintados com cores da escala do cinza. Isso quer dizer que eles ainda não foram abstraídos. Quando uma determinada instrução do código fonte é abstraída, ou seja, é escrita sua abstração na região “C”, o retângulo correspondente à instrução é pintado com cores na escala do verde, como pode ser visto em três retângulos na região “A” da Figura 3.8.

Outra informação importante é que como a ferramenta CRISTA implementa a técnica de leitura *Stepwise Abstraction*, a abstração do código precisa ser iniciada pelas instruções mais internas para, gradativamente, abstrair as instruções mais externas.

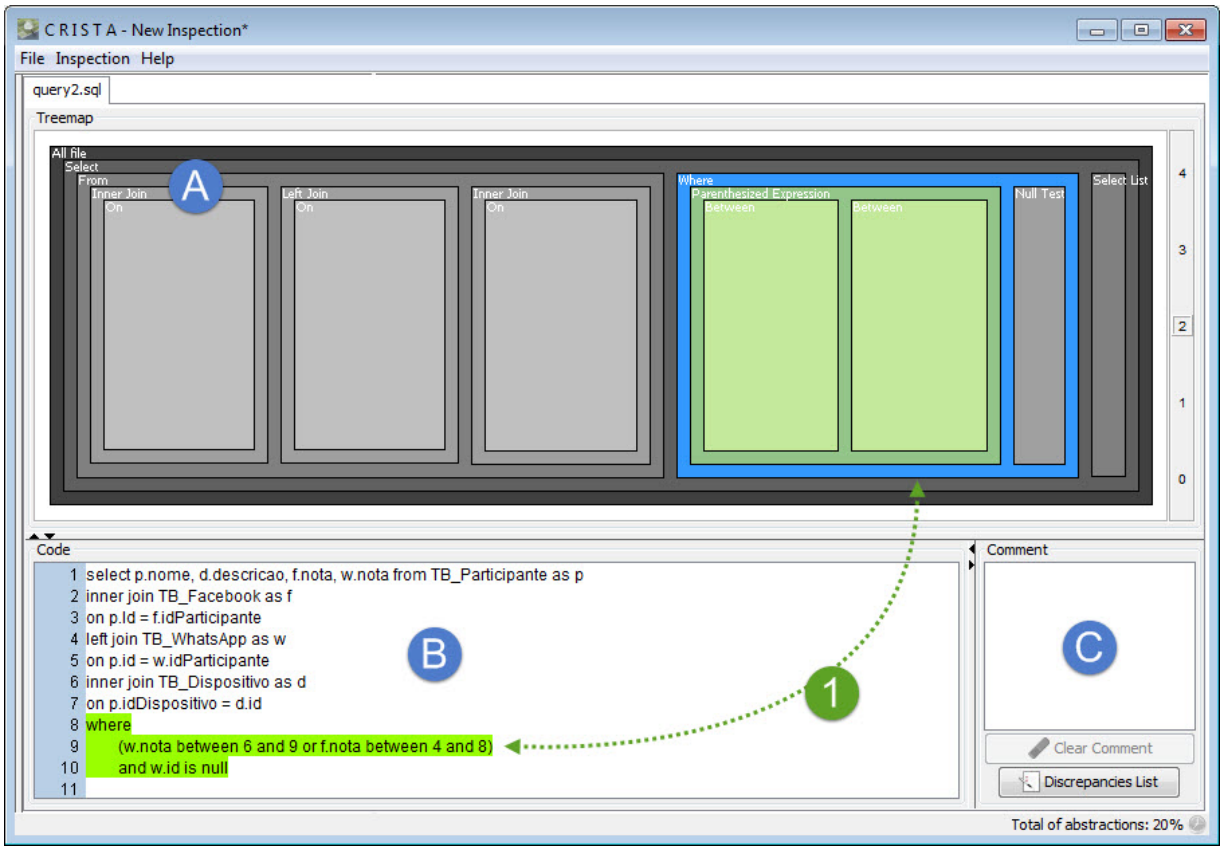


Figura 3.8: Regiões apresentadas pela ferramenta CRISTA e o sincronismo entre a metáfora visual e código fonte.

Como o próprio nome da etapa já deixa claro, o principal objetivo dessa etapa é detectar defeitos no código fonte. Quando isso ocorre, a ferramenta CRISTA fornece uma interface para que a discrepância seja registrada e passe a constar na lista de discrepâncias do inspetor. A Figura 3.9 mostra exatamente essa funcionalidade. Para registrar uma discrepância, basta clicar com o botão direito sobre o código fonte vinculado à discrepância e clicar na opção “Add Discrepancy”. Uma janela auxiliar sera exibida ao inspetor (passo “1”) que deverá preencher as informações acerca da discrepância. No passo “2”, o inspetor descreve a discrepância, no passo “3” classifica a discrepância de acordo com sua severidade e no passo “4” classifica de

acordo com sua prioridade. O quinto e último passo é clicar em “Ok” para confirmar o registro da discrepância.

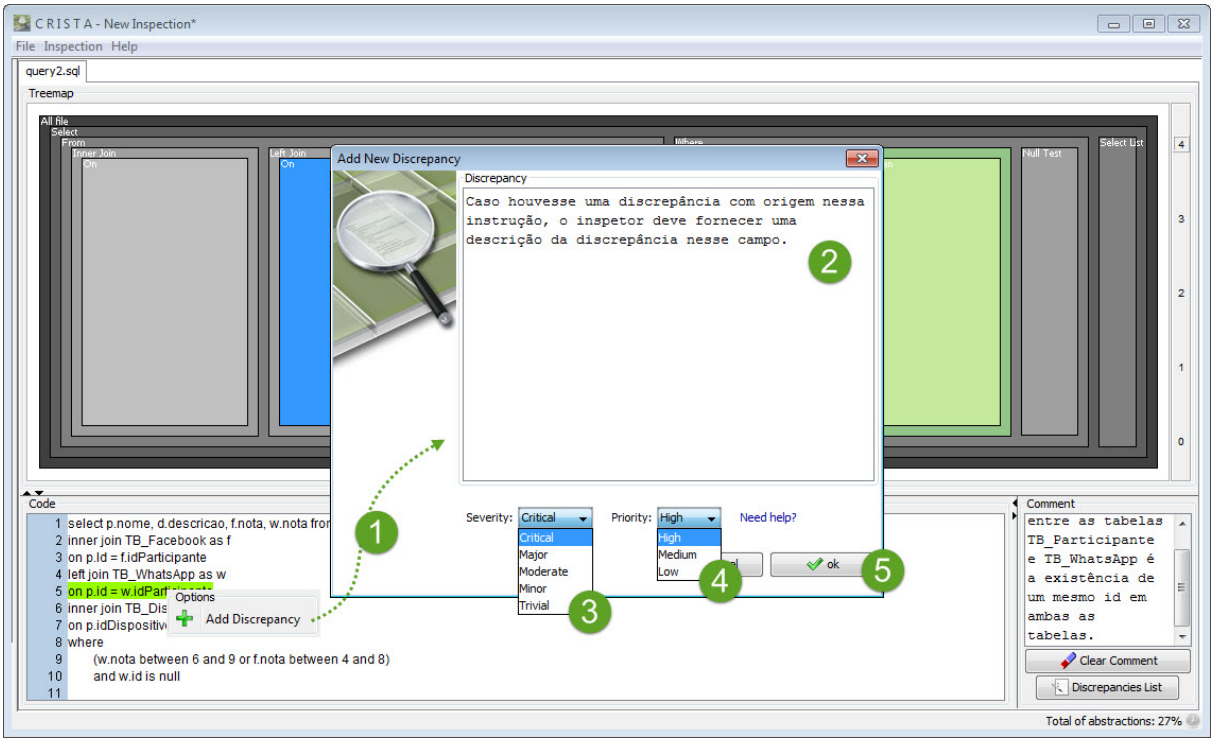


Figura 3.9: Passos para registrar uma discrepância do código fonte na ferramenta CRISTA.

Ao registrar uma discrepância do código fonte na ferramenta CRISTA, a metáfora visual do código é atualizada. O retângulo referente ao trecho de código fonte vinculado à discrepância passa a ser colorido em tons que variam do amarelo ao vermelho, dependendo da classificação dada à discrepância. O quanto menor for a severidade e prioridade da discrepância, mais amarelada será sua representação gráfica. No sentido oposto, quanto maior a severidade e prioridade, mais avermelhada será sua representação gráfica na metáfora visual *Treemap*. A Figura 3.10 mostra um exemplo em que foram registradas três discrepâncias, com diferentes níveis de severidade e prioridade.

O código fonte precisa ser abstraído até que se atinja 100% de abstração. Isso pode ser verificado através da metáfora visual *Treemap*, quando não houver mais retângulos cinzas ou pela informação disponibilizada no canto inferior direito da janela da CRISTA, circulada em roxo na Figura 3.10. Ao fim da atividade de abstração do código, todas as discrepâncias encontradas e registradas pelo inspetor estão salvas e podem ser acessadas clicando no botão “Discrepancies List”, circulado em azul na Figura 3.10.

A Figura 3.11 mostra a interface disponibilizada pela ferramenta CRISTA para que o inspetor tenha acesso às discrepâncias detectadas por ele. Na região “A”, pode-se ver a lista de

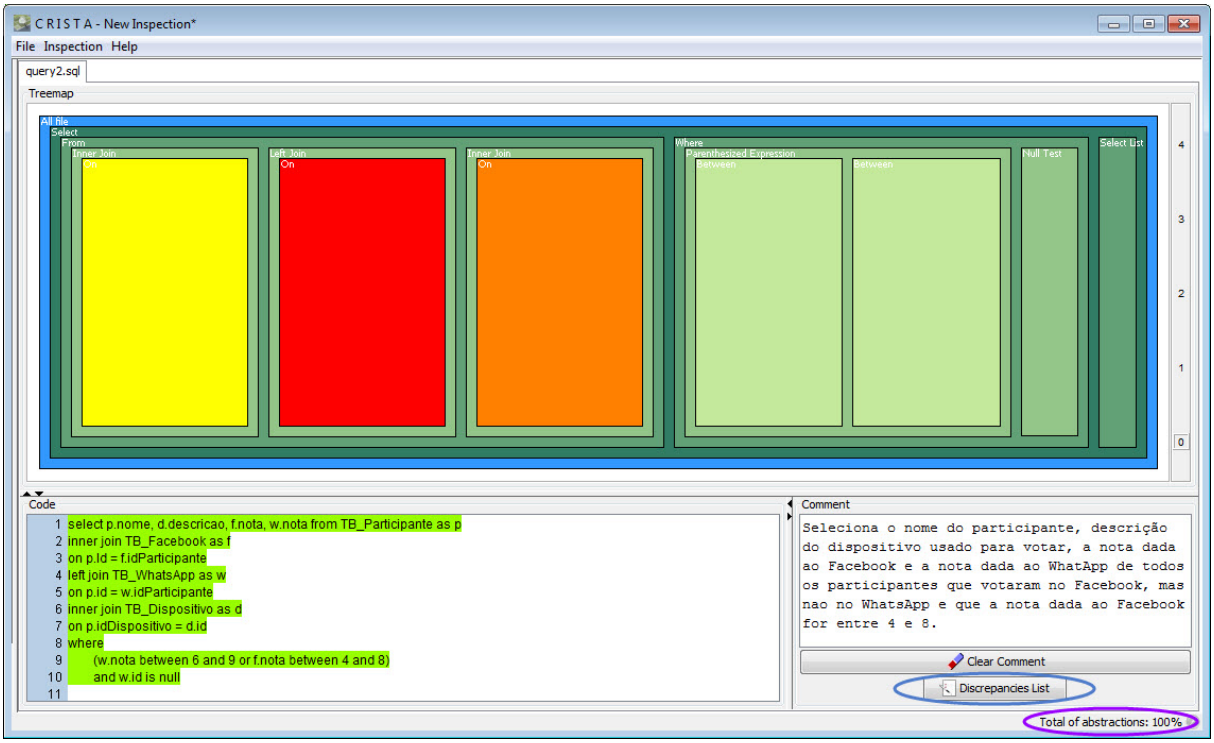


Figura 3.10: Diferentes cores para representação de discrepâncias na *Treemap*, dependendo de sua severidade e prioridade.

discrepâncias, bem como a descrição, severidade, prioridade e o *score* de cada uma delas. Na região "B" é exibido o código fonte, destacando o trecho de código vinculado a cada uma das discrepâncias. Por fim, na região "C" encontram-se os botões "Add discrepancy" e "Remove discrepancy" responsáveis por adicionar novas discrepâncias ou remover alguma discrepância previamente cadastrada. Além disso, há o botão "Show Treemap" que fecha essa interface e volta para a tela com a metáfora visual e o código fonte.

Para concluir a etapa de *Detecção de defeitos*, o inspetor deve salvar o projeto de inspeção e enviar o arquivo *.crista* para o moderador.

A etapa de *Coleção de Defeitos* inicia-se quando o moderador recebe os arquivos *.crista* contendo a lista de discrepâncias de cada um dos inspetores. Após isso, já na ferramenta CRISTA, o moderador deve acessar o menu "File / Discrepancies List Report", de acordo com o passo "1" da Figura 3.12. Uma nova interface será exibida ao moderador, na qual ele poderá abrir as listas de discrepâncias dos inspetores (passo "2"). Uma vez que as listas sejam abertas, as discrepâncias são exibidas no formato de lista, contendo as descrições dadas pelos inspetores às discrepâncias, bem como as classificações de severidade, prioridade e *score* de cada discrepância (passo "3"). A próxima ação que deve ser feita também pelo moderador é, para cada discrepância, adicioná-la à lista final de discrepâncias ou classificá-la como duplicada, clicando com o botão direito sobre a discrepância. Ao se escolher a opção de adicionar a discrepância,

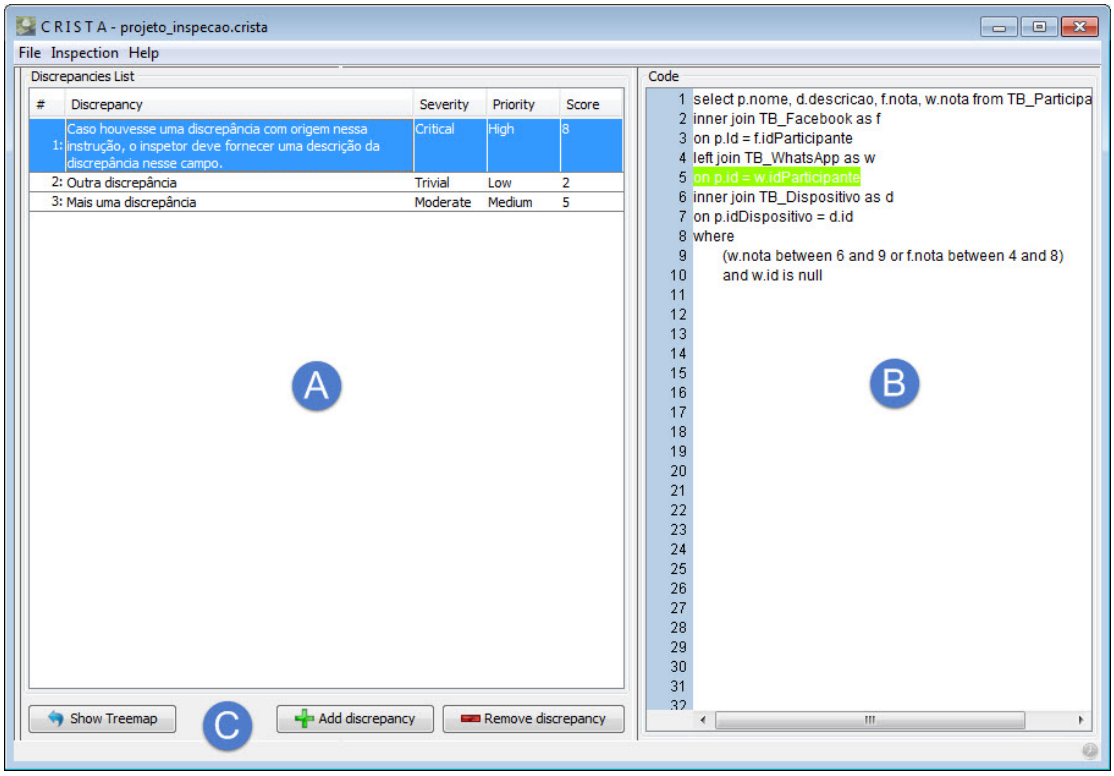


Figura 3.11: Lista de discrepâncias detectadas pelo inspetor.

ela é adiciona à lista final de discrepâncias (passo "4"). Caso seja verificado que a discrepância é duplicada, ela é removida da lista. Quando todas as discrepâncias de todos os inspetores tiverem sido analisadas, basta o moderador exportar a lista final de discrepância clicando no botão "Save Report" (passo "5").

O fim da etapa *Coleção de Defeitos* é marcada pela exportação da lista final de discrepâncias, caracterizada por um arquivo com extensão *.discrepanciesReport*.

Para realizar a etapa de *Discriminação de Defeitos*, o moderador faz a importação da lista final de discrepâncias obtida na etapa anterior clicando em "Open Report", como mostrado pelo passo "1" na Figura 3.13. Feito isso, a lista de discrepâncias é carregada e exibida na tela, conforme passo "2". As informações das discrepâncias estão à disposição da equipe de inspeção para que sejam tomadas as decisões para classificar ou não a discrepância como defeito ou rejeitá-la. Todas as colunas da lista podem ser ordenadas, como visto no passo "3" da figura 3.13, em que as discrepâncias foram ordenadas pelo *score*. Essa é uma boa estratégia, já que provavelmente as discrepâncias com maior *score* são as mais críticas ao sistema computacional inspecionado e, por isso, precisam ser analisadas primeiramente.

Uma vez estabelecida a ordem desejada das discrepâncias, inicia-se a atividade de análise de cada uma das discrepâncias. Caso a discrepância seja confirmada como defeito, ela permanece

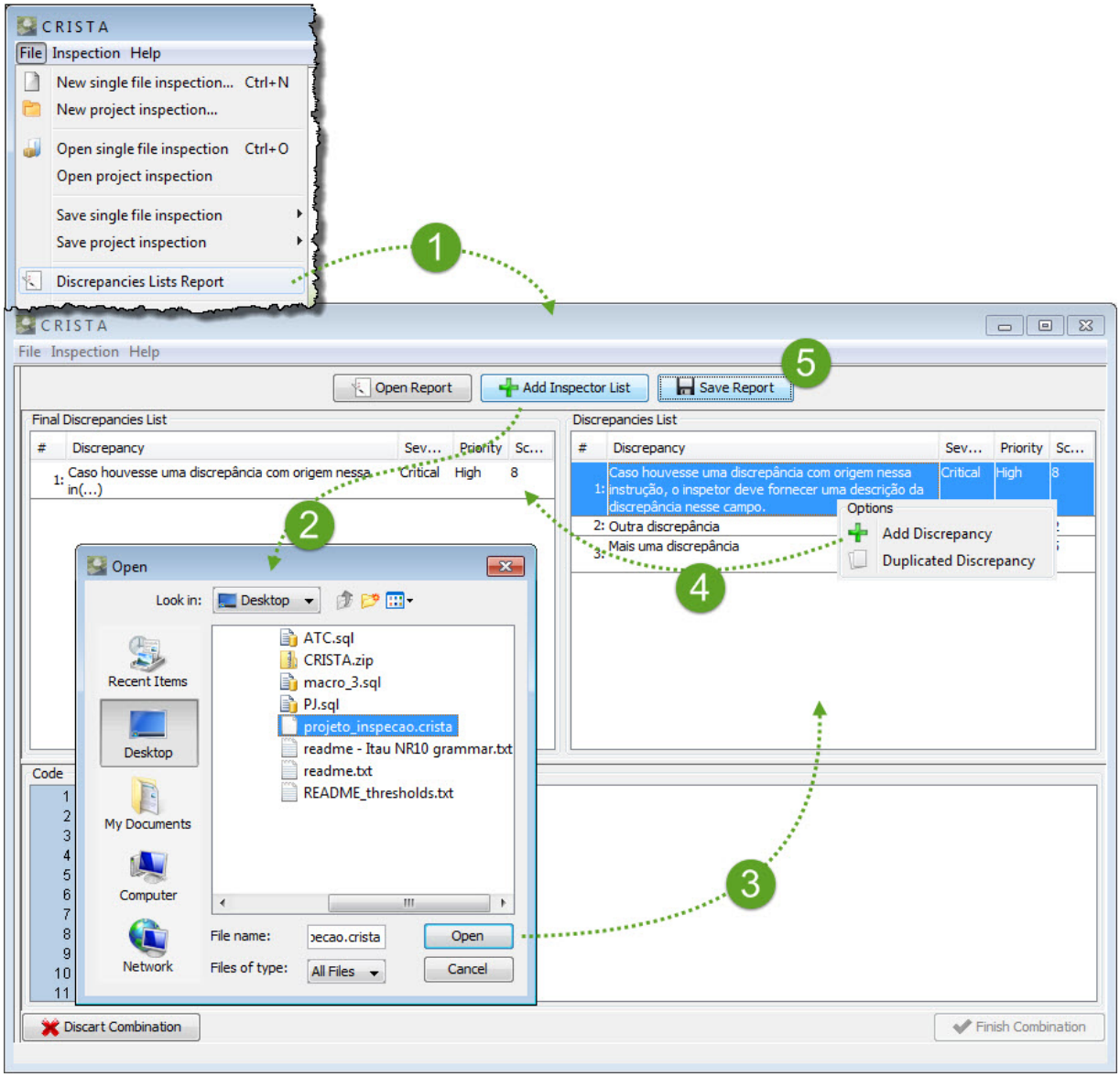


Figura 3.12: Suporte da ferramenta CRISTA à etapa de *Coleção de Defeitos*

na lista. Por outro lado, se for entendido que a discrepância não caracteriza um defeito no sistema, ela pode ser removida da lista, rejeitando-a. Essa ação é realizada clicando com o botão direito do *mouse* sobre a discrepância e escolhendo a opção "Reject Discrepancy", como mostrado pelo passo "4" da Figura 3.13. Quando todas as discrepâncias tiverem sido analisadas, a lista final de defeitos pode ser exportada clicando no botão "Finish Combination", mostrado no passo "5".

Para as etapas de *Retrabalho* e *Acompanhamento* não há funcionalidades específicas na ferramenta CRISTA. Porém, ambas as etapas podem ser conduzidas com base na lista de defeitos final exportada pela ferramenta. Como todos defeitos possuem as informações de severidade, prioridade e *score*, o retrabalho de correção pode ser realizado obedecendo uma ordem de-

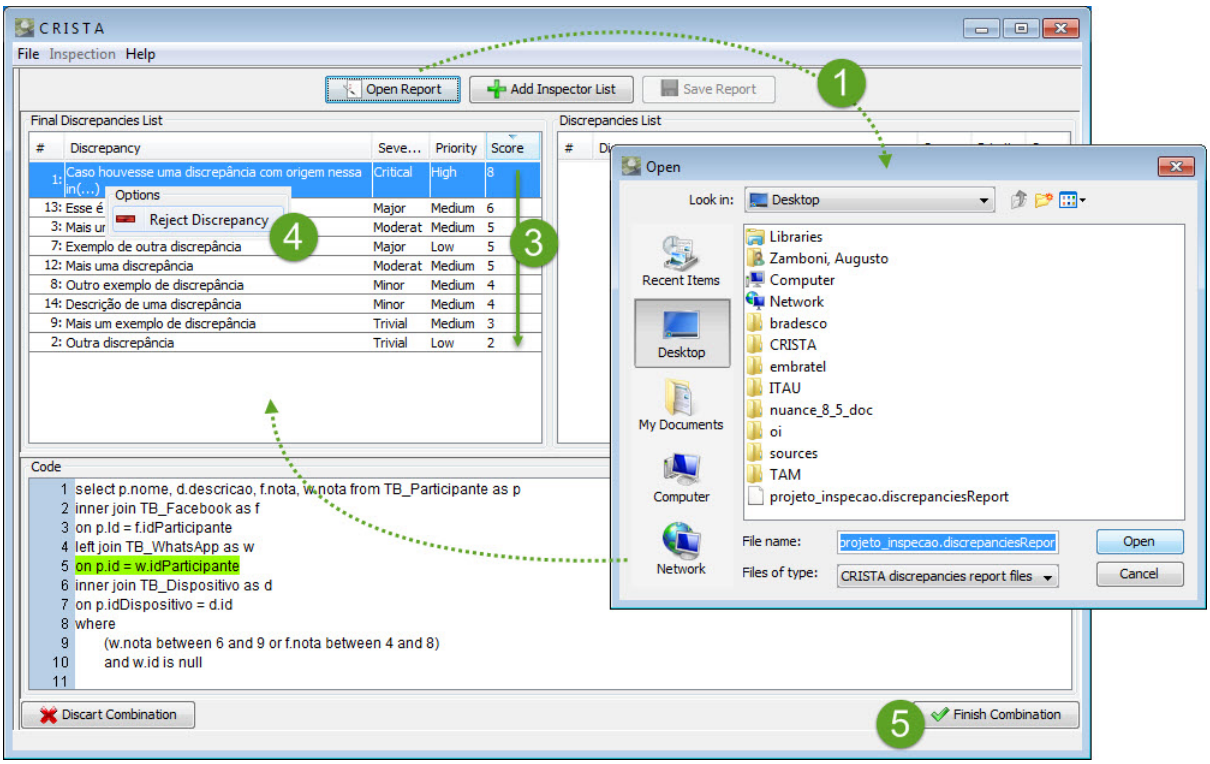


Figura 3.13: Suporte da ferramenta CRISTA à etapa de *Discriminação de Defeitos*

crescente de *score* ou então corrigindo primeiro os defeitos que tenham prioridade mais alta. Com relação ao acompanhamento, pode-se utilizar a mesma lista final de defeitos para manter o controle de quais defeitos já foram corrigidos, em qual data, por quem, etc.

3.5 Considerações Finais

Este capítulo apresentou a proposta de sistematização da atividade de inspeção de código fonte SQL com o apoio computacional da ferramenta CRISTA. Apesar da grande quantidade de trabalhos acadêmicos disponíveis na literatura sobre inspeção, não foram encontradas evidências de investigação relacionadas ao processo de inspeção em códigos fontes SQL. Dessa forma, este capítulo também apresentou um *survey*, respondido por profissionais de TI, com a finalidade de identificar como a atividade de inspeção de código SQL é conduzida no ambiente empresarial e quais as dificuldades enfrentadas por eles durante a execução da atividade.

Os resultados obtidos a partir da análise das respostas dos participantes do *survey*, somados à experiência prática do autor deste trabalho com inspeção de código SQL na indústria, embasaram a elaboração da proposta de sistematização da inspeção de código SQL composta por atividades sequenciais e bem definidas que seguem o processo de inspeção proposto por Sauer et al. (2000), com o apoio computacional da ferramenta CRISTA.

Para que a ferramenta CRISTA suportasse todas as atividades propostas, foi necessário desenvolver novas funcionalidades ou ajustar funcionalidades já existentes na versão original da ferramenta, com destaque para a instanciação do *parser* da linguagem SQL para que fosse possível carregar códigos escritos nessa linguagem na CRISTA, introdução dos conceitos de prioridade e severidade de defeitos com o objetivo de classificar e auxiliar a análise das discrepâncias e posterior correção dos defeitos e, por fim, o cálculo de um *score* em função da prioridade e severidade de cada defeito que caracteriza sua criticidade dentro das atividades de análise e correção dos defeitos encontrados durante a inspeção.

No próximo capítulo é apresentado um estudo experimental, realizado para avaliar a viabilidade de uso da proposta de sistematização da atividade de inspeção de código SQL com o apoio computacional da ferramenta CRISTA.

Capítulo 4

ESTUDO EXPERIMENTAL

Este capítulo descreve um estudo experimental conduzido para avaliar a viabilidade de uso da proposta de sistematização da atividade de inspeção de código SQL, utilizando a técnica de leitura Stepwise Abstraction e o apoio computacional fornecido pela ferramenta CRISTA.

4.1 Considerações Inciais

Segundo Travassos, Gurov e Amaral (2002), a experimentação é o centro do processo científico, sendo que, no campo da Engenharia de Software, ela oferece um modo sistemático, disciplinado, computável e controlado para avaliação de métodos, técnicas, linguagens e ferramentas. A utilização de métodos experimentais traz alguns benefícios, dentre os quais: acelera o progresso através da rápida eliminação de abordagens não fundamentadas, suposições errôneas e modismos, ajudando a orientar a engenharia e a teoria para direções promissoras.

Uma vez que a proposta de sistematização da atividade de inspeção de código fonte SQL havia sido definida e as alterações necessárias à ferramenta CRISTA para suportar a atividade encontravam-se em um nível suficientemente concluídas, fazia-se necessário avaliá-la por meio de estudos experimentais.

O restante do capítulo está organizado da seguinte forma: Na Seção 4.2 é apresentado um estudo experimental para avaliar a viabilidade de uso da proposta de sistematização da inspeção de código SQL, descrevendo seu planejamento, execução e discussão dos resultados. Por fim, a Seção 4.3 apresenta as considerações finais deste capítulo.

4.2 Estudo Experimental: Viabilidade de Uso da Proposta de Sistematização da Inspeção de Códigos SQL

Nessa seção é relatado um estudo experimental conduzido para avaliar a viabilidade de uso da proposta de sistematização da inspeção de código SQL. O estudo experimental foi realizado no ambiente acadêmico e contou com a participação de 12 pessoas, divididos em 2 grupos, referenciados como G1 e G2. A Tabela 4.1 mostra um resumo do experimento no que diz respeito ao seu objetivo e participantes.

Tabela 4.1: Resumo do estudo experimental de acordo com os objetivos e participantes.

	ESTUDO EXPERIMENTAL
Objetivo	Avaliação de viabilidade de uso da proposta de sistematização da atividade de **inspeção** de código SQL
Grupos	G1 e G2

4.2.1 Definição

A fase de definição do estudo experimental descreve os objetivos de sua realização (WOHLIN et al., 2000). No contexto deste estudo experimental, o objetivo é avaliar a viabilidade de uso da proposta de sistematização da atividade de inspeção de códigos fonte SQL, utilizando a técnica de leitura *Stepwise Abstraction* e o apoio computacional da ferramenta CRISTA. Por viabilidade entende-se que a condução da atividade de inspeção utilizando a proposta de sistematização não trará impactos negativos significantes em relação ao não uso da proposta, ou seja, de forma *adhoc*.

A estrutura a seguir, baseada no método GQM (BASILI; CALDERA; ROMBACH, 1994), define o estudo experimental.

Analisar a proposta de sistematização da atividade de inspeção de códigos fonte SQL por meio da técnica de leitura *Stepwise Abstraction* e o apoio computacional da ferramenta CRISTA

com o propósito de avaliar a viabilidade de uso da proposta

com respeito ao tempo gasto para executar a atividade (eficiência) e à quantidade de defeitos relatados ao fim da atividade de inspeção do código SQL (eficácia)

do ponto de vista do pesquisador

no contexto de alunos de graduação do curso de Bacharelado em Ciência da Computação da

UFSCar utilizando a proposta de sistematização da inspeção de código SQL, quando comparada à mesma atividade realizada de forma *adhoc*.

Assim, pretende-se avaliar as seguintes questões gerais:

- Q1: A utilização da proposta de sistematização da inspeção de código SQL implica em um tempo de conclusão da atividade de inspeção menor, quando comparada a execução da mesma atividade de inspeção de forma *adhoc*?
- Q2: A utilização da proposta de sistematização da inspeção de código SQL implica em uma quantidade de defeitos relatados maior, quando comparada a execução da mesma atividade de inspeção realizada de forma *adhoc*?

Conforme será esclarecido em maiores detalhes na próxima seção, a avaliação de viabilidade do processo proposto será feita a partir da coleta e análise de dados quantitativos durante a operação do experimento.

4.2.2 Planejamento

A fase de planejamento define e prepara como o experimento vai ser conduzido. São determinados os detalhes e os riscos do experimento. Ou seja, essa etapa implementa a fundação do experimento e o deixa totalmente elaborado e pronto para execução. Assim, nessa etapa são definidos (WOHLIN et al., 2000):

- Seleção de contexto;
- Formulação das hipóteses;
- Seleção das variáveis;
- Seleção de indivíduos;
- Projeto do experimento;
- Instrumentação;
- Avaliação da validade.

A descrição de cada uma das fases do planejamento do estudo experimental encontra-se nas subseções seguintes.

4.2.2.1 Seleção de Contexto

De acordo com Wohlin et al. (2000), o contexto pode ser caracterizado de acordo com quatro dimensões:

- Processo: *Off-line* vs. *on-line*;
- Participantes: alunos vs. profissionais;
- Realidade: problema modelado vs. problema real;
- Generalidade: específico vs. geral.

Este estudo pode ser caracterizado da seguinte forma: *off-line*, uma vez que não houve um monitoramento contínuo sobre as atividades dos participantes; os participantes foram alunos de graduação do curso de Bacharelado em Ciência da Computação do DC/UFSCar; o problema foi modelado; e a generalidade do estudo é específica para este contexto.

4.2.2.2 Formulação das Hipóteses

Para cada questão definida na seção de definição do experimento (4.2.1), foram elaboradas uma hipótese nula e uma hipótese alternativa.

1. Com relação ao tempo:
   - Hipótese nula (H_0): A utilização da proposta de sistematização da inspeção de código SQL não traz alteração à atividade quando executada de forma *adhoc* no que diz respeito ao tempo gasto em sua execução.

     (H_0): ($\mu_1 = \mu_2$), em que:
     - μ_1 é o tempo gasto para realizar a atividade de inspeção do código SQL quando utilizada a proposta de sistematização da inspeção.
     - μ_2 é o tempo gasto para realizar a atividade de inspeção do código SQL sem utilizar a proposta de sistematização da inspeção.
   - Hipótese alternativa (H_1): O tempo gasto para realizar a atividade de inspeção é menor quando a proposta de sistematização da inspeção de código SQL é utilizada.

     (H_1): ($\mu_1 < \mu_2$), em que:

  - μ_1 é o tempo gasto para realizar a atividade de inspeção do código SQL quando utilizada a proposta de sistematização da inspeção.
  - μ_2 é o tempo gasto para realizar a atividade de inspeção do código SQL sem utilizar a proposta de sistematização da inspeção.

2. Com relação à quantidade de defeitos relatados:

- Hipótese nula (H_0): A utilização da proposta de sistematização da inspeção de código SQL não traz alteração à atividade quando executada de forma *adhoc* no que diz respeito à quantidade de defeitos relatados.

  (H_0): ($\eta_1 = \eta_2$), em que:

  - η_1 é a quantidade de defeitos relatados ao fim da atividade de inspeção do código SQL quando utilizada a proposta de sistematização da inspeção.
  - η_2 é a quantidade de defeitos relatados ao fim da atividade de inspeção do código SQL sem utilizar a proposta de sistematização da inspeção.

- Hipótese alternativa (H_1): A quantidade de defeitos é maior quando a proposta de sistematização da inspeção de código SQL é utilizada.

  (H_2): ($\eta_1 > \eta_2$), em que:

  - η_1 é a quantidade de defeitos relatados ao fim da atividade de inspeção do código SQL quando utilizada a proposta de sistematização da inspeção.
  - η_2 é a quantidade de defeitos relatados ao fim da atividade de inspeção do código SQL sem utilizar a proposta de sistematização da inspeção.

4.2.2.3 Seleção das Variáveis

O experimento utiliza as seguintes variáveis:

- Variáveis independentes:
  - Proposta de sistematização da inspeção de código SQL.
- Variáveis dependentes:
  - Tempo gasto para realizar a atividade de inspeção;
  - Quantidade de defeitos relatados ao fim da atividade de inspeção. Esta variável foi calculada na forma de percentual, de acordo com a fórmula abaixo:

    Quantidade = (quantidade de defeitos relatados) / (quantidade de defeitos conhecidos do código) * 100%.

4.2.2.4 Seleção de Indivíduos

O experimento foi realizado com a participação de alunos da disciplina Engenharia de Software II, oferecida no quinto semestre do curso de graduação em Bacharelado em Ciência da Computação desta Universidade. No total, 12 alunos participaram do experimento, divididos em dois grupos (G1 e G2). A divisão foi realizada de forma aleatória, uma vez que o grupo de alunos era bastante homogêneo no que se referia ao tipo e tempo de experiência em desenvolvimento de códigos SQL. Todos os participantes preencheram um questionário de caracterização em termos de sua experiência prática com desenvolvimento de código SQL e sua formação acadêmica.

As Figuras 4.1 e 4.2 mostram, respectivamente, a distribuição dos participantes por tempo e tipo de experiência com desenvolvimento de código SQL.

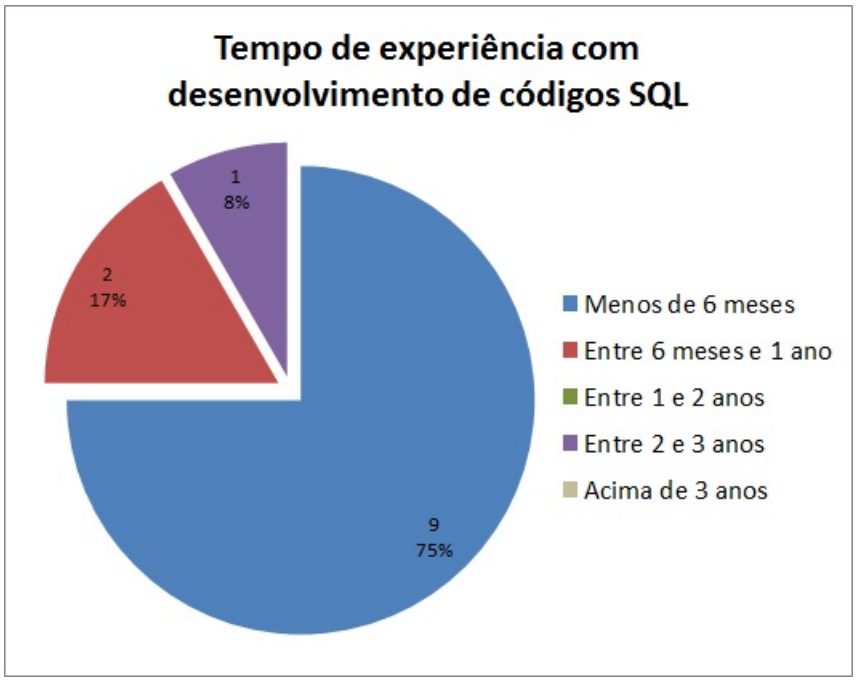


Figura 4.1: Tempo de experiência com desenvolvimento de códigos SQL dos participantes do estudo experimental

A participação no experimento substituiu uma das avaliações teóricas da disciplina, de tal maneira que a nota corresponderia não à qualidade das atividades desempenhadas pelo aluno, mas exclusivamente pela sua presença na atividade.

4.2.2.5 Projeto do Experimento

O projeto do estudo experimental envolve um fator: a abordagem seguida para realização da atividade de inspeção de código SQL. Dois tratamentos foram dados a este fator: utilizar a

Figura 4.2: Tipo de experiência com desenvolvimento de códigos SQL dos participantes do estudo experimental

proposta de sistematização da inspeção de código SQL proposto neste trabalho e não utilizá-la. O experimento faz uso de um projeto balanceado, com um grupo para cada tratamento, dividindo os participantes conforme descrito na Seção 4.2.2.4. Cada tratamento teve o mesmo número de participantes e cada um dos participantes utilizou um único tratamento.

Como preparação ao experimento, sentiu-se a necessidade de realizar um treinamento preparatório para os participantes sobre a atividade de inspeção, a técnica de leitura *Stepwise Abstraction* e de utilização da ferramenta CRISTA. Para consolidar os conceitos do treinamento, foram realizadas duas atividades práticas antes da execução do experimento.

No experimento, os participantes de ambos os grupos receberam um mesmo código SQL no formato de arquivo eletrônico, contendo a implementação de 5 *Views*, bem como um arquivo contendo as especificações de cada uma delas. Além desse material, os participantes possuíam livre acesso a materiais da Internet, que foram utilizados para sanar possíveis dúvidas relacionadas à linguagem de programação SQL.

4.2.2.6 Instrumentação

Na realização dos experimentos, foram usados os seguintes materiais:

1. Apresentação de treinamento sobre inspeção de código, técnica de leitura *Stepwise Abstraction* e a ferramenta CRISTA. Apresentação disponível na Seção B.1.

2. Apresentação de contextualização ao programa SQL que foi utilizado no experimento. Apresentação disponível na Seção B.2.

3. Questionário de caracterização dos participantes do experimento. Questionário disponível na Seção B.3.

4. Exercício para aplicação da técnica de leitura *Stepwise Abstraction* em um código SQL utilizando a ferramenta CRISTA. Foram entregues aos participantes três códigos SQL, totalizando 67 linhas. Códigos disponíveis na Seção B.4.

5. Código fonte SQL utilizado no experimento. Foi entregue um código SQL responsável por criar 5 *Views*, contendo 9 defeitos inseridos propositalmente. Código fonte disponível na Seção B.5.

6. Especificação do código fonte SQL utilizado no experimento. Como o experimento estava inserido no contexto de inspeção de código, foi disponibilizado um documento com a especificação referente ao código que seria inspecionado. Documento disponível na Seção B.6.

7. Lista de defeitos. Ao final do experimento, foi entregue aos participantes uma lista contendo todos o defeitos conhecidos do código inspecionado. Documento disponível na Seção B.7.

4.2.2.7 Avaliação da Validade

As validades interna, externa, de construção e de conclusão do experimento encontram-se discutidas a seguir:

- Validade interna – Como se trata de um experimento envolvendo mais de um grupo, em que cada grupo é submetido a um tratamento distinto, a maior ameaça à validade interna é a relação dos resultados com a seleção dos participantes do estudo. Porém, como observado por meio do questionário de caracterização, os participantes possuíam tempo e tipo de experiência com desenvolvimento SQL semelhantes, fato esse que diminui a ameaça à validade interna.

- Validação externa – O estudo envolve alunos de graduação e utiliza um problema modelado. Ele se propõe a ser válido dentro desse contexto, não sendo possível generalizar os resultados para um contexto de mercado de trabalho envolvendo um problema real. Para tanto, seria necessário elaborar e executar novos estudos para ampliar a validade externa.

- Validade de construção – Para obter validade de construção, foi necessário verificar dois aspectos: que os tratamentos representam bem a construção da causa e que a saída do experimento representa bem a construção do efeito. Como o experimento possui apenas um fator e dois tratamentos, a causa pode ser facilmente mapeada no uso ou não da proposta de sistematização da inspeção de código SQL. A saída por sua vez está ligada diretamente ao efeito, uma vez que sua análise pode ser feita por meio das variáveis dependentes.

- Validade de conclusão – As ameaças à validade de conclusão deste experimentos estão relacionadas a 4 questões: i) a quantidade de participantes foi pequena, somando-se apenas 12 pessoas; ii) o tempo para realizar a atividade de inspeção no código fonte SQL foi limitado pelo horário de término da aula, totalizando um período de 60 minutos; iii) em decorrência da limitação de tempo para a execução do experimento, optou-se por utilizar um código fonte pequeno, de 49 linhas; e iv) o teste estatístico aplicado para comparar os valores obtidos em cada uma das variáveis dependentes foi o *t-test*, com intervalo de confiança de 95

4.2.3 Operação

O experimento foi conduzido com a participação de 12 alunos inscritos na disciplina Engenharia de Software II do curso de graduação em Bacharelado em Ciência da Computação, oferecido pelo Departamento de Computação da UFSCar.

O experimento foi executado em dois dias. No primeiro foi realizado um treinamento com todos os participantes, com o objetivo de nivelar o conhecimento em relação à atividade de inspeção de código e seus benefícios, ao uso da técnica de leitura *Stepwise Abstraction* e como se dá a sistematização dessa atividade para códigos SQL, utilizando a ferramenta CRISTA como apoio computacional. Esse treinamento foi composto por uma parte teórica (com duração de aproximadamente 30 minutos) e uma parte prática (com duração de aproximadamente 60 minutos). Uma vez que todos os participantes estavam satisfatoriamente treinados, foram solicitados a preencher um questionário de caracterização. Esse questionário foi utilizado para analisar a homogeneidade do grupo. Como discutido na seção 4.2.2.4, o tempo e tipo de experiência com desenvolvimento de código SQL eram semelhantes entre os participantes, viabilizando portanto, a divisão aleatória dos participantes em dois grupos.

No segundo dia, os participantes foram divididos em dois grupos, sendo que o grupo G1 foi designado para realizar a atividade de inspeção de acordo com a proposta de sistematização da

atividade, enquanto que o G2 foi utilizado como grupo de controle, para o qual não foi imposto nenhum processo.

Após a divisão dos grupos e esclarecido o papel de cada grupo dentro do experimento, foram disponibilizados a todos os participantes dois arquivos no formato digital: i) um arquivo contendo o código SQL que deveria ser inspecionado e ii) um arquivo contendo a especificação do código SQL. Além disso, os participantes do grupo G1 receberam um *link* para fazer o *download* da ferramenta CRISTA.

Após o experimento, foi solicitado aos participantes que empacotassem todo o material utilizado durante a inspeção, principalmente a lista de defeitos encontrados, e os submetessem em um diretório virtual disponibilizado pelo aplicador do experimento. No caso do grupo G1, apenas o arquivo gerado pela ferramenta CRISTA era suficiente. Já para o grupo G2, todo e qualquer arquivo utilizado deveria ser submetido.

4.2.4 Análise dos Dados

A Tabela 4.2 apresenta os valores obtidos para as variáveis 'tempo' e 'quantidade de defeitos relatados' de cada um dos participantes do grupo G1, ou seja, aqueles que realizaram a atividade de inspeção de acordo com a proposta de sistematização. Os resultados dos participantes do grupo G2, que realizaram a atividade de forma *adhoc*, encontram-se na Tabela 4.3. Os resultados marcados como *outlier* são dados fora da distribuição normal. Nessa análise foram-se considerados como *outliers* todos os resultados mais afastados da média do que o desvio padrão. As tabelas apresentam ainda a média, o desvio padrão e a média desconsiderando os *outliers* para cada uma das variáveis.

Os resultados preliminares, apresentados nas Tabelas 4.2 e 4.3 foram analisados por meio do teste estatístico *t-test* implementado pela ferramenta Action[1]. Com base nas análises, as seguintes observações podem ser feitas a respeito das hipóteses apresentadas no planejamento do experimento:

1. Com relação ao tempo:

   - **Hipótese alternativa (H_1): O tempo gasto para realizar a atividade de inspeção é menor quando a proposta de sistematização da inspeção de código SQL é utilizada.**

[1] www.portalaction.com.br

		Defeito 01	Defeito 02	Defeito 03	Defeito 04	Defeito 05	Defeito 06	Defeito 07	Defeito 08	Defeito 09	Porcentagem de defeitos relatados	Tempo (minutos)
Participantes	Participante 01	✓		✓		✓	✓		✓	✓	66.7	46 (*outlier*)
	Participante 02	✓	✓		✓		✓	✓	✓		66.7	51
	Participante 03			✓			✓				22.2 (*outlier*)	49
	Participante 04			✓		✓	✓		✓	✓	55.6	51
	Participante 05			✓		✓	✓		✓		44.4	56 (*outlier*)
	Participante 06	✓		✓		✓	✓		✓	✓	66.7	51
	Média										53.7	50.7
	Desvio Padrão										16.3	2.9
	Média sem *outliers*										60.0	50.5

Tabela 4.2: Resultados para as variáveis 'tempo' e 'quantidade de defeitos relatados' dos participantes do tratamento com o processo de sistematização da inspeção de código SQL.

		Defeito 01	Defeito 02	Defeito 03	Defeito 04	Defeito 05	Defeito 06	Defeito 07	Defeito 08	Defeito 09	Porcentagem de defeitos relatados	Tempo (minutos)
Participantes	Participante 07	✓		✓		✓			✓	✓	55.6	43
	Participante 08	✓		✓			✓			✓	44.4	58 (*outlier*)
	Participante 09	✓		✓			✓		✓	✓	55.6	42 (*outlier*)
	Participante 10			✓		✓	✓		✓		44.4	50
	Participante 11	✓		✓		✓			✓		44.4	46
	Participante 12			✓			✓				22.2 (*outlier*)	47
	Média										44.4	47.7
	Desvio Padrão										11.2	5.3
	Média sem *outliers*										48.9	46.5

Tabela 4.3: Resultados para as variáveis 'tempo' e 'quantidade de defeitos relatados' dos participantes do tratamento *adhoc*.

Aplicando *t-test* desconsiderando os *outliers*, com grau de liberdade igual a 6 e nível de confiança igual a 0.95, obtém-se P-valor = 0.98. Como P-valor não é menor que 0.05, a hipótese nula não pôde ser refutada em relação ao tempo gasto para realizar a atividade de inspeção por meio da aplicação do *t-test*.

2. Com relação à quantidade de defeitos relatados:

- **Hipótese alternativa (H_1): A quantidade de defeitos é maior quando a proposta de sistematização da inspeção de código SQL é utilizada.**

Aplicando *t-test* desconsiderando os *outliers*, com grau de liberdade igual a 8 e nível de confiança igual a 0.95, obtém-se P-valor = 0.03. Como P-valor é menor que 0.05, a hipótese nula pôde ser refutada em relação à cobertura de defeitos por meio da aplicação do *t-test*.

4.2.5 Discussão

Como visto na seção anterior, a aplicação do teste *t-test* nos dados relacionados ao tempo gasto para realizar a atividade de inspeção em códigos SQL, na tentativa de avaliar a diminuição do tempo ao se utilizar a proposta de sistematização, teve um resultado inconclusivo. Realmente, se os tempos de conclusão dos participantes do grupo G1 forem comparados com os tempos do grupo G2, observa-se que praticamente todos os participantes do grupo G1 terminaram em um tempo maior. Se a comparação for feita apenas através da média, desconsiderando os *outliers*, chega-se a conclusão que, em média, os alunos do grupo G2 realizaram a atividade 4 minutos mais rapidamente que os alunos do grupo G1. Porém, quando analisada a porcentagem de defeitos encontrados pelos dois grupos, observa-se que, apesar dos participantes grupo G1 demorarem 4 minutos a mais para concluir a atividade, eles encontraram 11% a mais de defeitos.

Outro resultado interessante pode ser visualizado na Figura 4.3. Essa figura foi criada por meio da ferramenta Insight (HERNANDES et al., 2014) que tem a proposta de utilizar visualização como apoio à análise qualitativa de dados. Ao submeter os resultados tabelados do experimento descrito neste capítulo na ferramenta Insight, foi possível observar dois fatos que corroboram o resultado positivo do experimento:

1. *Completude de defeitos* – Pode-se observar na Figura 4.3, destacados pelos retângulos de cor rosa, que os defeitos “Defeito02”, “Defeito04” e “Defeito07” foram relatados apenas por participantes do grupo G1, ou seja, aqueles que realizaram a inspeção utilizando a abordagem sistêmica e com apoio computacional da ferramenta CRISTA.

2. *Falso positivos* – Pode-se observar também que houve a ocorrência de dois falso positivos dentre os defeitos relatados pelo grupo G2. Fato esse que não foi observado no grupo G1.

Apesar dos bons resultados preliminares, novos estudos experimentais precisam ser realizados contando com um grupo maior de participantes, com um limite de tempo maior e utilizando códigos fontes SQL maiores e mais complexos. Dessa forma, as ameaças à validade do experimento, principalmente aquelas relacionadas à validade de conclusão (discutidas na seção 4.2.2.7 deste capítulo), poderão ser mitigadas.

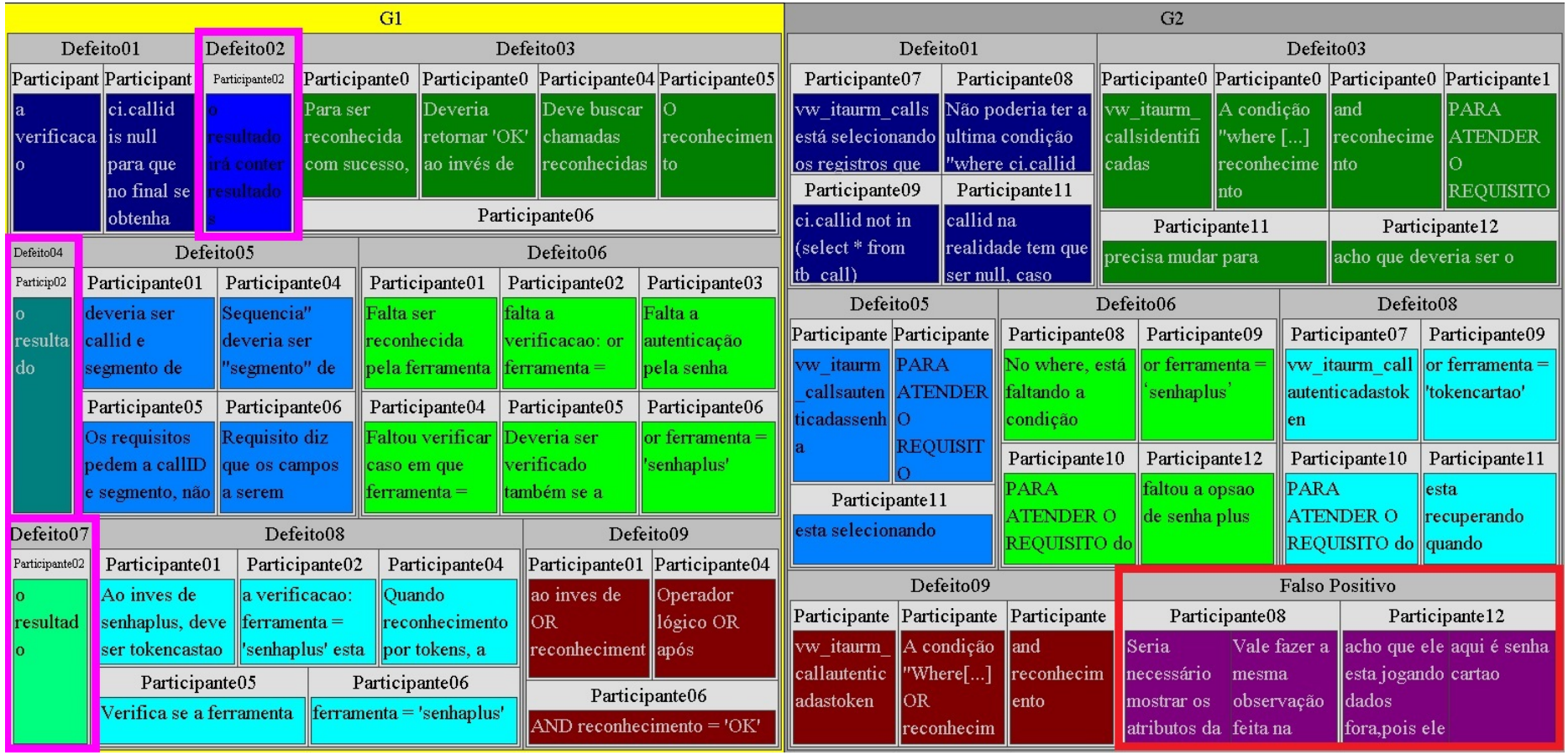


Figura 4.3: Análise dos defeitos relatados com o suporte de visualização da ferramenta Insight (HERNANDES et al., 2014).

4.3 Considerações Finais

Este capítulo apresentou um estudo experimental realizado com 12 estudantes de graduação em Bacharelado em Ciência da Computação, com o objetivo de avaliar a viabilidade de uso da proposta de sistematização da atividade de inspeção de código SQL, utilizando a técnica de leitura *Stepwise Abstraction* para abstrair o código e apoio computacional da ferramenta CRISTA. Os 12 participantes foram divididos em dois grupos, os quais receberam tratamentos distintos. O primeiro grupo executou a atividade utilizando a proposta de sistematização da atividade, enquanto que o segundo grupo executou a mesma atividade, porém de forma *adhoc*.

Os resultados preliminares obtidos a partir desse experimento são satisfatórios e dão indícios de que a execução da atividade de inspeção de código SQL realizada de maneira sistêmica e com o apoio computacional da ferramenta CRISTA, quando comparada a execução *adhoc*, pode fazer com que uma quantidade maior e mais abrangente de defeitos sejam encontrados. Além disso, observou-se que a execução da atividade de forma *adhoc* gerou a ocorrência de falso positivos, o que não ocorreu quando a abordagem sistêmica foi utilizada. Por outro lado, observou-se que, mesmo que pequeno, houve um aumento do tempo necessário para se concluir a atividade de inspeção quando utilizada a abordagem sistêmica.

Outros estudos experimentais precisam ser realizados para avaliar melhor as questões de tempo e quantidade de defeitos relatados para ambos os tratamentos. Um maior número de participantes, com uma variação maior de perfis e a utilização de um código fonte SQL maior e mais complexo são imprescindíveis para tornar os resultados aqui obtidos mais gerais e consistentes.

O estudo experimental descrito neste capítulo encerra os relatos teóricos e práticos deste trabalho. As conclusões, contribuições, limitações e trabalhos futuros são descritos em detalhes no capítulo seguinte.

Capítulo 5

CONCLUSÕES

A inspeção de software é um tipo particular de revisão que pode ser aplicada em todos os artefatos ao longo do ciclo de desenvolvimento de software e, juntamente com as atividades de teste, as inspeções compõem as atividades de VV&T (Verificação, Validação e Teste). Dentre os possíveis artefatos, destacam-se os arquivos de código fonte. O principal objetivo das inspeções, incluindo inspeções de código fonte, é trazer mais qualidade ao artefato inspecionado por meio da detecção e correção de possíveis defeitos inseridos ao longo do processo de desenvolvimento.

Este trabalho apresentou uma proposta de sistematização da atividade de inspeção de código SQL apoiada computacionalmente pela ferramenta CRISTA. A proposta foi elaborada tendo como base a experiência do autor deste trabalho no contexto de inspeção de código SQL e também os resultados de um *survey* aplicado no ambiente empresarial com o intuito de identificar como a inspeção é feita no dia a dia e quais as dificuldades enfrentadas. A proposta é composta por uma série de atividades sequencias, bem definidas e baseadas no processo de inspeção proposto por Sauer et al. (2000).

O apoio computacional da ferramenta CRISTA como parte da proposta é decorrente do conhecimento documentado na literatura de que, uma vez que o processo de inspeção seja realizado de forma manual ou *adhoc*, ela se torna muito tediosa, demorada e propensa a erros (KOTHARI et al., 2004; LUCIA et al., 2007). Para que a CRISTA suportasse todas as atividades que compõem a proposta, foi necessário desenvolver o módulo de reconhecimento da linguagem SQL, mais especificamente MS SQL Server, e também outras funcionalidades como, por exemplo, atribuição de diferentes níveis de severidade, prioridade e valores de *score* para cada discrepância detectada durante a inspeção. Além disso, é importante ressaltar a funcionalidade de ordenação das discrepâncias de acordo com sua classificação de severidade, prioridade e *score*.

Além da apresentação da proposta e sua instanciação na ferramenta CRISTA, foi também apresentado um estudo experimental conduzido para avaliar sua viabilidade de uso. Apesar das ameças à validade como, por exemplo, aplicação do experimento apenas no ambiente acadêmico e uma quantidade pequena de participantes, os resultados obtidos foram satisfatórios. Após análise dos dados coletados durante o estudo experimental, observou-se que a utilização na proposta de sistematização da inspeção de código SQL resultou em uma quantidade maior de defeitos detectados, além de um número menor de falsos positivos, quando comparada a realização da atividade de forma *adhoc*. No entanto, não foi possível tirar conclusões a respeito da interferência do uso ou não da proposta no tempo gasto para realizar a atividade.

5.1 Contribuições e Limitações

As principais contribuições deste trabalho são:

- Revisão da literatura sobre inspeção de software;
- Elaboração de uma proposta de sistematização da atividade de inspeção de códigos SQL;
- Desenvolvimento do módulo de reconhecimento da linguagem SQL para que fosse possível carregar código SQL na ferramenta CRISTA e, portanto, utiliza-la como suporte computacional à abordagem sistêmica de inspeção proposta;
- Classificação de discrepâncias com relação a sua severidade e prioridade;
- Cálculo de um *score*, em função da severidade e prioridade de cada discrepância, para guiar a análise das discrepâncias e correção dos defeitos;
- Estudo experimental para avaliar a viabilidade de uso da proposta de sistematização da inspeção de código SQL.

As principais limitações deste trabalho são:

- Apoio computacional para códigos fontes SQL desenvolvidos apenas para bancos de dados MS SQL Server;
- Implementação da metáfora visual somente na técnica *Treemap*;
- Implementação apenas da técnica de leitura *Stepwise Abstraction*;
- O estudo experimental foi realizado apenas em ambiente acadêmico.

5.2 Trabalhos Futuros

Dentre as atividades que podem ser realizadas como continuidade deste trabalho, citam-se:

- Dar manutenção à ferramenta CRISTA, de forma a evoluí-la, principalmente no que diz respeito as limitações apresentadas acima;
- Elaborar e conduzir novos estudos experimentais em contextos reais, com códigos maiores, mais complexos e em ambientes industriais;
- Investigar a relação entre a utilização da proposta de sistematização da atividade de inspeção de código SQL e o tempo gasto para concluir a atividade, quando comparado com o tempo gasto na realização da mesma atividade de forma *adhoc*;
- Adicionar novos módulos para reconhecimento de outras versões de código SQL, não apenas MS SQL Server;
- Investigar técnicas de visualização e de leitura alternativas que possam tornar a atividade de compreensão e inspeção mais eficiente.
- Implementar novas técnicas de visualização e de leitura, bem como avaliar esses contextos por meio de estudos experimentais.

REFERÊNCIAS

ACKERMAN, A. F.; BUCHWALD, L. S.; LEWSKI, F. H. Software inspections: An effective verification process. *IEEE software*, IEEE Computer Society Press, v. 6, n. 3, p. 31–36, 1989.

ALMEIDA, J. R. et al. Best practices in code inspection for safety-critical software. *Software, IEEE*, IEEE, v. 20, n. 3, p. 56–63, 2003.

ANDERSON, P.; REPS, T.; TEITELBAUM, T. Design and implementation of a fine-grained software inspection tool. *Software Engineering, IEEE Transactions on*, IEEE, v. 29, n. 8, p. 721–733, 2003.

ANDERSON, P. et al. Tool support for fine-grained software inspection. *Software, IEEE*, IEEE, v. 20, n. 4, p. 42–50, 2003.

ARTHO, C. *Jlint—find bugs in java programs*. 2006. Disponível em: <http://jlint.sourceforge.net/>. Acesso em: 20 de janeiro de 2015.

BASILI, V.; CALDERA, C.; ROMBACH, H. Goal question metric paradigm. *Encyclopedia of Software Engineering*, v. 1, p. 528–532, 1994.

BASILI, V. R. et al. The empirical investigation of perspective-based reading. *Empirical Software Engineering*, Springer, v. 1, n. 2, p. 133–164, 1996.

BELGAMO, A. et al. Code inspection supported by stepwise abstraction and visualization - an experimental study. *Proceedings of the 16th International Conference on Enterprise Information Systems*, p. 39–48, 2014.

BIFFL, S.; GRÜNBACHER, P.; HALLING, M. A family of experiments to investigate the effects of groupware for software inspection. *Automated Software Engineering*, Springer, v. 13, n. 3, p. 373–394, 2006.

BOEHM, B.; BASILI, V. R. Software defect reduction top 10 list. *Foundations of empirical software engineering: the legacy of Victor R. Basili*, v. 426, 2005.

BOOGERD, C.; MOONEN, L. Prioritizing software inspection results using static profiling. In: IEEE. *Source Code Analysis and Manipulation, 2006. SCAM'06. Sixth IEEE International Workshop*. [S.l.], 2006. p. 149–160.

BREEN, P. *Exposing the Fallacy of "Good Enough" Software*. 2002. Disponível em: <www.informit.com/articles/article.aspx?p=25141>. Acesso em: 28 de dezembro de 2014.

BURN, O. *Checkstyle*. 2007. Disponível em: <http://checkstyle.sourceforge.net/>. Acesso em: 20 de janeiro de 2015.

COPELAND, T. *PMD applied*. [S.l.]: Centennial Books San Francisco, 2005.

DEMARCO, T.; LISTER, T. *Peopleware*. [S.l.]: Dorset House, 1998.

DENGER, C.; KOLB, R. Testing and inspecting reusable product line components: first empirical results. In: ACM. *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*. [S.l.], 2006. p. 184–193.

DENGER, C.; SHULL, F. A practical approach for quality-driven inspections. *Software, IEEE*, IEEE, v. 24, n. 2, p. 79–86, 2007.

DÓRIA, E. S. *Replicação de estudos empíricos em engenharia de software.* Tese (Doutorado) — Universidade de São Paulo, 2001.

FAGAN, M. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, v. 15, n. 3, p. 182–211, 1976. ISSN 0018-8670.

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

GARVIN, D. A. What does "product quality" really mean. *Sloan management review*, v. 1, 1984.

GLASS, R. L. *Software runaways: monumental software disasters*. [S.l.]: Prentice-Hall, Inc., 1998.

GRAHAM, D.; GILB, T. *Software Inspection*. [S.l.]: Addison-Wesley Wokingham, 1993.

GUPTA, V.; PATNAIK, A. R.; GOEL, N. A system for controlling software inspections. In: IEEE. *Electrical and Computer Engineering, 2003. IEEE CCECE 2003. Canadian Conference on*. [S.l.], 2003. v. 2, p. 1343–1346.

HARJUMAA, L.; TERVONEN, I.; VUORIO, P. Improving software inspection process with patterns. In: IEEE. *Quality Software, 2004. QSIC 2004. Proceedings. Fourth International Conference on*. [S.l.], 2004. p. 118–125.

HERNANDES, E. et al. Using visualization and text mining to improve qualitative analysis. *ICEIS 2014 - Proceedings of the 16th International Conference on Enterprise Information Systems*, v. 2, p. 201–208, 2014.

HERNANDES, E. M.; BELGAMO, A.; FABBRI, S. Experimental studies in software inspection process-a systematic mapping. In: *15th International Conference on Enterprise Information Systems*. [S.l.: s.n.], 2013. p. 66–76.

HOVEMEYER, D.; PUGH, W. Finding bugs is easy. *ACM Sigplan Notices*, ACM, v. 39, n. 12, p. 92–106, 2004.

HUMPHREY, W. S. *Managing the Software Process (Hardcover)*. [S.l.]: Addison-Wesley Professional, 1989.

IEEE. Ieee standard glossary of software engineering terminology. *IEEE Std*, v. 610.12-1990, 1990.

ISO/IEC12207. Tecnologia de informação–processos de ciclo de vida de software. *Rio de Janeiro: ABNT*, 1998.

JAVACC. *Java Compiler Compiler*. [S.l.], 2015. Disponível em: <https://javacc.java.net/>. Acesso em: 06 de fevereiro de 2015.

JOCHAM, R. *JCSC - Java Coding Standard Checker*. 2005. Disponível em: <http://jcsc.sourceforge.net/>. Acesso em: 20 de janeiro de 2015.

JOHNSON, P. M. An instrumented approach to improving software quality through formal technical review. In: IEEE COMPUTER SOCIETY PRESS. *Proceedings of the 16th international conference on Software engineering*. [S.l.], 1994. p. 113–122.

JOHNSON, P. M.; TJAHJONO, D. Assessing software review meetings: A controlled experimental study using csrs. In: ACM. *Proceedings of the 19th international conference on Software engineering*. [S.l.], 1997. p. 118–127.

JOHNSON, P. M.; TJAHJONO, D. Does every inspection really need a meeting? *Empirical Software Engineering*, Springer, v. 3, n. 1, p. 9–35, 1998.

JONES, C. Software defect-removal efficiency. *Computer*, IEEE, v. 29, n. 4, p. 94–95, 1996.

KALINOWSKI, M. Infra-estrutura computacional de apoio ao processo de inspeção de software. 2004.

KALINOWSKI, M.; TRAVASSOS, G. H. A computational framework for supporting software inspections. In: IEEE. *Automated Software Engineering, 2004. Proceedings. 19th International Conference on*. [S.l.], 2004. p. 46–55.

KOLLANUS, S.; KOSKINEN, J. Survey of software inspection research. *The Open Software Engineering Journal*, v. 3, n. 1, p. 15–34, 2009.

KOTHARI, S. C. et al. A pattern-based framework for software anomaly detection. *Software Quality Journal*, Springer, v. 12, n. 2, p. 99–120, 2004.

LANUBILE, F.; MALLARDO, T. Tool support for distributed inspection. In: IEEE. *Computer Software and Applications Conference, 2002. COMPSAC 2002. Proceedings. 26th Annual International*. [S.l.], 2002. p. 1071–1076.

LANUBILE, F.; MALLARDO, T. An empirical study of web-based inspection meetings. In: IEEE. *Empirical Software Engineering, 2003. ISESE 2003. Proceedings. 2003 International Symposium on*. [S.l.], 2003. p. 244–251.

LI, Z.; AVGERIOU, P.; LIANG, P. A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, v. 101, p. 193–220, 2015.

LINGER, R. C.; WITT, B. I.; MILLS, H. D. Structured programming. Addison-Wesley, 1979.

LUCIA, A. D. et al. Assessing the effectiveness of a distributed method for code inspection: A controlled experiment. In: IEEE. *Global Software Engineering, 2007. ICGSE 2007. Second IEEE International Conference on*. [S.l.], 2007. p. 252–261.

MARTIN, J.; TSAI, W. T. N-fold inspection: A requirements analysis technique. *Communications of the ACM*, ACM, v. 33, n. 2, p. 225–232, 1990.

MPS-BR. Melhoria de processo do software brasileiro. 2012.

MYERS, G. J.; SANDLER, C. The art of software testing. *Wiley*, v. 3, p. 2, 2004.

NASA. *Software formal inspections guidebook*. 1993. Disponível em: <http://www.cs.nott.ac.uk/ cah/G53QAT/fi.pdf>. Acesso em: 02 de janeiro de 2015.

NASCIMENTO, H. A. D.; FERREIRA, C. B. Visualização de informações–uma abordagem prática. In: *XXV Congresso da Sociedade Brasileira de Computação, XXIV JAI. UNISINOS, S. Leopoldo–RS*. [S.l.: s.n.], 2005.

NEGINHAL, S.; KOTHARI, S. Event views and graph reductions for understanding system level c code. In: IEEE. *Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on*. [S.l.], 2006. p. 279–288.

OSTERWEIL, L. Strategic directions in software quality. *ACM Computing Surveys (CSUR)*, ACM, v. 28, n. 4, p. 738–750, 1996.

PARNAS, D. L.; LAWFORD, M. The role of inspection in software quality assurance. *Software Engineering, IEEE Transactions on*, IEEE, v. 29, n. 8, p. 674–676, 2003.

PERPICH, J. M. et al. Anywhere, anytime code inspections: using the web to remove inspection bottlenecks in large-scale software development. In: ACM. *Proceedings of the 19th international conference on Software engineering*. [S.l.], 1997. p. 14–21.

PORTER, A. A.; JOHNSON, P. M. Assessing software review meetings: Results of a comparative analysis of two experimental studies. *Software Engineering, IEEE Transactions on*, IEEE, v. 23, n. 3, p. 129–145, 1997.

PORTO, D.; MENDONÇA, M.; FABBRI, S. Crista-code reading implemented with stepwise abstraction. *proceedings of Simpósio Brasileiro de Engenharia de Software - Sessão de Ferramentas*, 2008.

PORTO, D. et al. Manutenção de código apoiada pela ferramenta crista. *VI Workshop de Manutenção de Software Moderna*, Ouro Preto-MG, 2009.

PORTO, D. de P. Crista: um apoio computacional para atividades de inspeção e compreensão de código. Biblioteca Digital de Teses e Dissertações da Universidade Federal de São Carlos, 2010.

PRESSMAN, R. S. *Engenharia de software*. [S.l.]: McGraw Hill Brasil, 2011.

RILLING, J.; KLEMOLA, T. Identifying comprehension bottlenecks using program slicing and cognitive complexity metrics. In: IEEE. *Program Comprehension, 2003. 11th IEEE International Workshop on*. [S.l.], 2003. p. 115–124.

ROCHA, A. R. C. da; MALDONADO, J. C.; WEBER, K. C. *Qualidade de software: teoria e prática*. [S.l.]: Prentice Hall, 2001.

RUNESON, P.; ANDREWS, A. Detection or isolation of defects? an experimental comparison of unit testing and code inspection. In: IEEE. *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on*. [S.l.], 2003. p. 3–13.

SAUER, C. et al. The effectiveness of software development technical reviews: A behaviorally motivated program of research. *Software Engineering, IEEE Transactions on*, IEEE, v. 26, n. 1, p. 1–14, 2000.

SEI. *CMMI® for Development, Version 1.3*. nov. 2010. Disponível em: <www.sei.cmu.edu/library/abstracts/reports/10tr033.cfm>. Acesso em: 28 de dezembro de 2014.

SENSALIRE, M.; OGAO, P. Tool users requirements classification: how software visualization tools measure up. In: ACM. *Proceedings of the 5th international conference on Computer graphics, virtual reality, visualisation and interaction in Africa*. [S.l.], 2007. p. 119–124.

SOMMERVILLE, I. *Software Engineering, 9th edition*. [S.l.]: Addison–Wesley, Ed, 2011.

TEAM, T. L. *Software Lime Survey*. 2015. Disponível em: <www.limesurvey.org>. Acesso em: 02 de fevereiro de 2015.

TERGAN, S. O.; KELLER, T. *Knowledge and information visualization: Searching for synergies*. [S.l.]: Springer, 2005.

THELIN, T. et al. A replicated experiment of usage-based and checklist-based reading. In: IEEE. *Software Metrics, 2004. Proceedings. 10th International Symposium on*. [S.l.], 2004. p. 246–256.

THUNG, F. et al. To what extent could we detect field defects? an extended empirical study of false negatives in static bug-finding tools. *Automated Software Engineering*, Springer, p. 1–42, 2015.

TRAVASSOS, G. H.; GUROV, D.; AMARAL, E. A. G. do. Introdução à engenharia de software experimental. 2002.

VENNER, B. *Design by Contract: A Conversation with Bertrand Mayer*. 2003. Disponível em: <www.artima.com/intv/contracts.html>. Acesso em: 28 de dezembro de 2014.

VINZ, B. L.; ETZKORN, L. H. A synergistic approach to program comprehension. In: IEEE. *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*. [S.l.], 2006. p. 69–73.

VOTTA, L. G. Does every inspection need a meeting? In: ACM. *ACM SIGSOFT Software Engineering Notes*. [S.l.], 1993. v. 18, n. 5, p. 107–114.

WALKINSHAW, N.; ROPER, M.; WOOD, M. Understanding object-oriented source code from the behavioural perspective. In: IEEE. *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on*. [S.l.], 2005. p. 215–224.

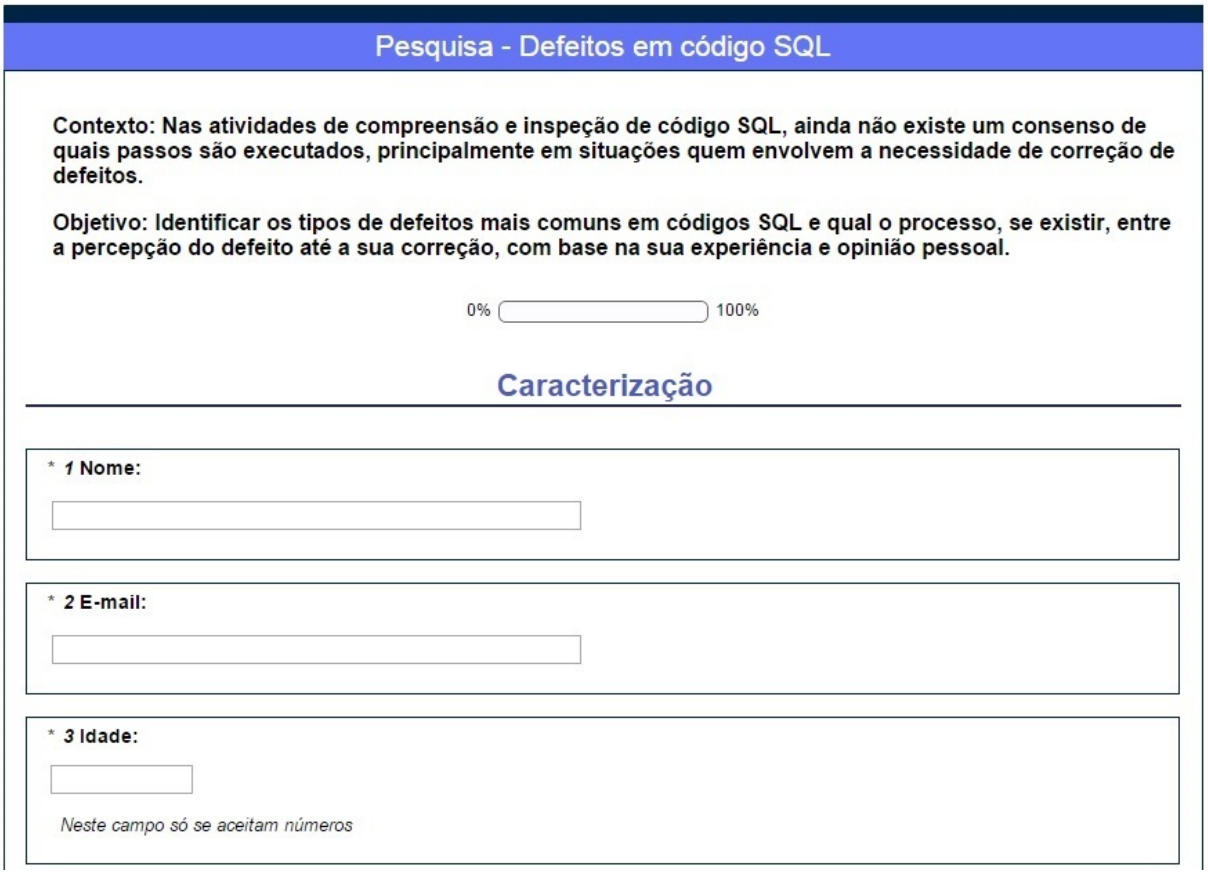
WHEELER, D. A.; BRYKCZYNSKI, B.; MEESON, R. N. *Software Inspection: An Industry Best Practice for Defect Detection and Removal*. [S.l.]: IEEE Computer Society Press, 1996.

WINKLER, D.; THURNHER, B.; BIFFL, S. Early software product improvement with sequential inspection sessions: an empirical investigation of inspector capability and learning effects. In: IEEE. *Software Engineering and Advanced Applications, 2007. 33rd EUROMICRO Conference on*. [S.l.], 2007. p. 245–254.

WOHLIN, C. et al. Experimentation in software engineering: an introduction. Kluver Academic Publishers, 2000.

Apendice A

SURVEY PARA CARACTERIZAÇÃO DA ATIVIDADE DE INSPEÇÃO DE CÓDIGO SQL NO CONTEXTO EMPRESARIAL



Pesquisa - Defeitos em código SQL

Contexto: Nas atividades de compreensão e inspeção de código SQL, ainda não existe um consenso de quais passos são executados, principalmente em situações quem envolvem a necessidade de correção de defeitos.

Objetivo: Identificar os tipos de defeitos mais comuns em códigos SQL e qual o processo, se existir, entre a percepção do defeito até a sua correção, com base na sua experiência e opinião pessoal.

0% 100%

Caracterização

* *1* **Nome:**

* *2* **E-mail:**

* *3* **Idade:**

Neste campo só se aceitam números

* ***4*** **Cargo atual:**
Escolha uma das seguintes respostas

- ○ Gerente
- ○ Coordenador
- ○ DBA
- ○ Analista de negócio
- ○ Analista de sistemas
- ○ Analista de suporte
- ○ Desenvolvedor
- ○ Testador
- ○ Outro:

* ***5*** **Nível de escolaridade:**
Escolha uma das seguintes respostas

- ○ Pós-doutorado
- ○ Doutorado
- ○ Mestrado
- ○ Especialização / MBA
- ○ Superior completo
- ○ Superior incompleto
- ○ Ensino médio

* ***6*** **Tempo de experiência na área de TI:**
Escolha uma das seguintes respostas

- ○ 0 a 2 anos
- ○ 3 a 6 anos
- ○ 7 a 10 anos
- ○ Acima de 10 anos

* ***7*** **Tempo de experiência em desenvolvimento SQL**
Escolha uma das seguintes respostas

- ○ 0 a 2 anos
- ○ 3 a 6 anos
- ○ 7 a 10 anos
- ○ Acima de 10 anos

* ***8*** **Quais dos Bancos de Dados abaixo você já utilizou?**
Seleccione todas as que se apliquem

- ☐ Apache Derby
- ☐ Caché
- ☐ DB2
- ☐ Firebird
- ☐ InterBase
- ☐ Microsoft SQL Server
- ☐ MySQL
- ☐ Oracle
- ☐ PostgreSQL
- ☐ SQLite
- ☐ Sybase
- ☐ Outro:

Processo

* ***9***

Considerando os atores abaixo, indique, em ordem de prioridade, os responsáveis pela identificação de falhas nos sistemas computacionais desenvolvidos.

1 - O que mais identifica falhas

5 - O que menos identifica falhas

Clique num item na lista à esquerda, começando com o item de maior classificação, e percorrendo os itens até ao menor.

As suas escolhas:	A sua classificação:
Analista de Negócio Analista de Sistemas Desenvolvedor Testador Cliente / Usuário	1: 2: 3: 4: 5:

Carregue na tesoura próxima de cada item à direita para remover o último registo da sua lista de classificação

* ***10*** **Considerando os intervalos abaixo, qual é a porcentagem de falhas relacionadas a defeitos em código SQL? (Por exemplo, se de cada 10 falhas, 3 são relacionadas a defeitos em código SQL, então a porcentagem é de 30%).**
Escolha uma das seguintes respostas

- ○ 0% a 10%
- ○ 11% a 20%
- ○ 21% a 30%
- ○ 31% a 40%
- ○ 41% a 50%
- ○ 51% a 60%
- ○ 61% a 70%
- ○ 71% a 80%
- ○ 81% a 90%
- ○ 91% a 100%

* ***11*** **É utilizada alguma ferramenta de bug tracking para defeitos associados a código SQL?**

○ Sim ○ Não

* ***12*** **Se sim, qual(is)?**

* ***13*** **Quais são os defeitos mais comuns em código SQL?**

* ***14*** **Considerando que uma falha é proveniente de defeitos em código SQL, quais são os passos realizados para localizar o objeto (Stored Procedure, View, Trigger, Function etc) no qual o defeito se encontra?**

* *15* **Uma vez que o objeto tenha sido localizado, algum documento adicional é utilizado para a identificação do trecho de código SQL defeituoso? (Por exemplo, documento de requisitos, UML, fluxogramas, modelo entidade-relacionamento etc).**

* *16* **É adotada alguma estratégia para auxiliar a compreensão do código SQL defeituoso?**

- ○ Sim
- ○ Não

* *17* **Se sim, descreva a estratégia.**

* *18* **É utilizada alguma IDE para o o desenvolvimento de código SQL?**

- ○ Sim
- ○ Não

* *19* **Se sim, qual(is)?**

* *20* **Quais recursos da IDE são utilizados para facilitar a compreensão de código SQL?**

* *21* **Quais são as dificuldades encontradas durante a compreensão do código SQL defeituoso?**
Seleccione todas as que se apliquem

- ☐ Quantidade de objetos SQL envolvidos
- ☐ Quantidade de linhas do objeto SQL defeituoso
- ☐ Contexto/escopo da aplicação
- ☐ Código SQL desenvolvido por outra pessoa
- ☐ Falta de documentação externa ao código SQL (doc. de requisitos, UML, modelo entidade-relacionamento, etc)
- ☐ Falta de comentários no código SQL
- ☐ Outro:

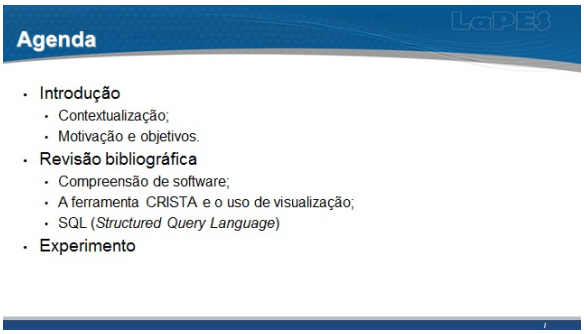
* *22* **Na sua opinião, qual é o nível de dificuldade em compreender e encontrar defeitos em códigos SQL, comparado com a identificação de defeitos em códigos de linguagens de programação orientadas a objeto.**
Escolha uma das seguintes respostas

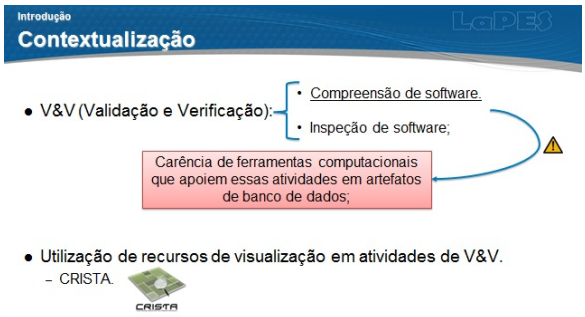
- ○ Muito mais difícil
- ○ Mais difícil
- ○ Mesmo nível de dificuldade
- ○ Mais fácil
- ○ Muito mais fácil

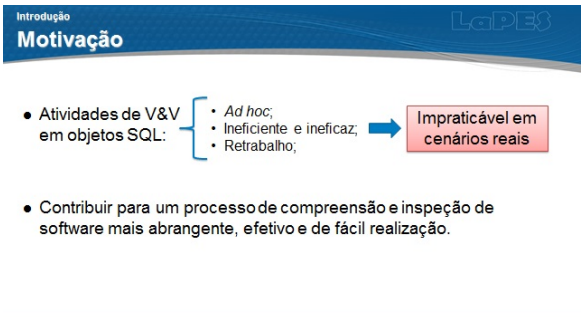
Apendice B

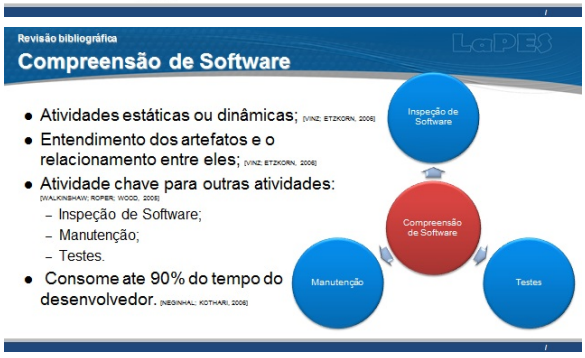
INSTRUMENTOS DO EXPERIMENTO

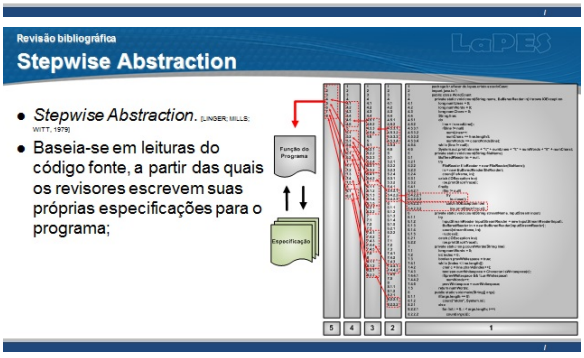
B.1 Apresentação de Treinamento dos Participantes











Revisão bibliográfica

Visualização

- O processo de visualização está relacionado à transformação de algo abstrato em imagens, sejam elas reais ou mentais, que possam ser visualizadas pelos seres humanos. O objetivo principal é auxiliar no entendimento de determinado assunto, o qual, sem uma visualização, exigiria maior esforço para ser compreendido;

[NASCIMENTO; FERREIRA, 2005] , [KEPSEN; LUZZARDI; LOH, 2007].

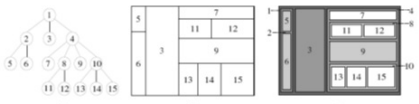
Revisão bibliográfica

CRISTA e a técnica *Treemap*

- CRISTA (*Code Reading Implemented with Stepwise Abstraction*);

[PORTO; MENDONÇA; FABBRI, 2008]

- Metáfora visual *Treemap*;



Uma árvore (a), e possíveis visualizações com *treemaps* (b) e (c) [PFEIFFER; GURD, 2006]

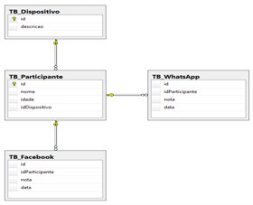
Revisão bibliográfica

SQL (*Structured Query Language*)

- Linguagem de consulta a banco de dados relacional, criada em 1976, nos laboratórios da IBM no contexto de um projeto denominado *System R*; [ASTRAHAN et al., 1976]
- Ao longo do tempo, a linguagem sofreu alterações para se adequar aos novos ambientes computacionais e, hoje, encontra-se na versão SQL:2011; [ISO/IEC9075]
- Principais comandos:
  - DML: Linguagem de Manipulação de Dados (ex: *INSERT*);
  - DDL: Linguagem de Definição de Dados (ex: *CREATE*);
  - DCL: Linguagem de Controle de Dados (ex: *GRANT*).

Treinamento – Exemplo prático

- Sistema fictício;



- Cada participante é cadastrado e vota nas ferramentas através de um dispositivo;
- Um mesmo participante pode votar em 1 ou nas duas aplicações;
- Nota entre 0 (péssimo) e 9 (excelente);

B.2 Apresentação de Contextualização dos Participantes ao Domínio do Código Utilizado no Experimento

SISTEMATIZAÇÃO DA INSPEÇÃO DE CÓDIGO SQL UTILIZANDO A TÉCNICA *STEPWISE ABSTRACTION* E A FERRAMENTA CRISTA

Introdução aos experimentos 1 e 2

Aluno: Augusto Bindilatti Zamboni
Orientadora: Profa. Dra. Sandra Camargo P. F. Fabbri

Sumário

- Introdução
  - Objetivo
  - Familiarização com os relatórios
- Estrutura do banco de dados
  - Estrutura do banco de dados
  - Views, Stored Procedures, TablesProcesso de geração dos relatórios

Introdução

Objetivo

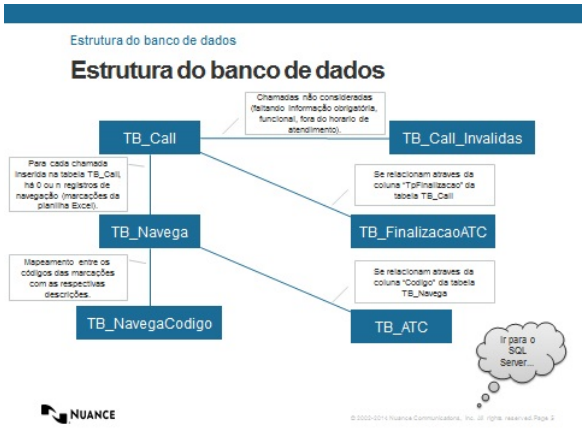
- Comportamento da URA
  - Volume de chamadas
  - Chamadas repetidas
  - Retenção
  - Derivação
  - Hang-up
  - Auto-atendimento
  - Levantamento dos assuntos mais requisitados
- Encontrar pontos de melhoria
  - Servico X com baixa retenção...
  - Hang-up muito alto no estado Y...
  - Alto indice de No-Match/No-Input no menu Z...

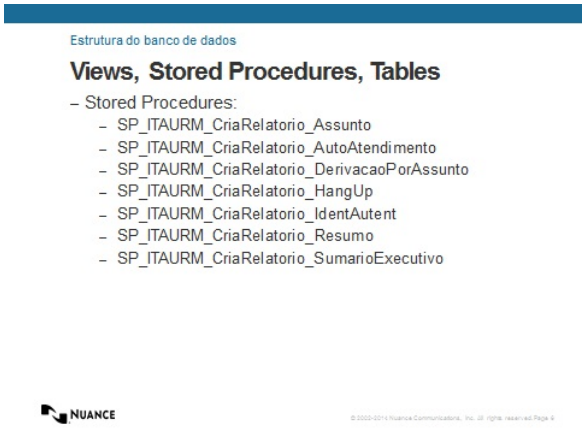
Introdução

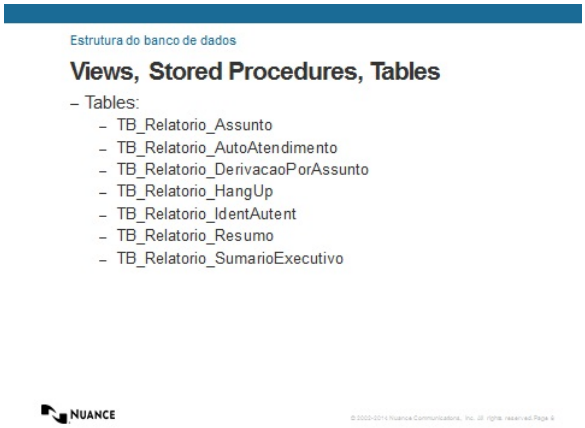
Familiarização com os relatórios

- Link: Relatórios

Termo	Significado
Chamadas repetidas	Qtd de chamadas recebidas de um mesmo número de telefone (DDI, DDD, telefone, agencia e conta), no mesmo dia para o mesmo segmento.
Retenção	O cliente realizou a tarefa toda dentro da URA
Derivação	Chamada derivou para o atendimento humano
Hang-up	O cliente desligou a ligação em algum ponto da URA







B.3 Questionário de Caracterização

CARACTERIZAÇÃO DOS PARTICIPANTES

A) DADOS PESSOAIS

1. Idade: _____
2. Sexo
   ( )Masculino ( )Feminino
3. Semestre:_____

B) CARACTERÍSTICAS TÉCNICAS

4. Leitura em Inglês
   ( ) Bem ( )Médio ( )Ruim
5. Qual sua experiência anterior com o desenvolvimento de scripts SQL?
   ( ) Nunca desenvolvi scripts SQL
   ( ) Já desenvolvi scripts SQL sozinho
   ( ) Já desenvolvi scripts SQL em grupo na aula
   ( ) Já desenvolvi scripts SQL em empresas
6. Quanto tempo você tem de experiência em programação SQL?
   ( ) Menos de 6 meses
   ( ) Entre 6 meses e 1 ano
   ( ) Entre 1 ano e 2 anos
   ( ) Entre 2 anos e 3 anos
   ( ) Mais de 3 anos

7. Quais linguagens de programação você conhece?

   _____

8. Qual a sua experiência em inspeção de software?
   ( ) Nenhuma
   ( ) Estudei em aula ou em livros
   ( ) Usei em um projeto em aula
   ( ) Usei em um projeto de empresas
9. Quanto tempo você tem de experiência em inspeção de software?
   ( ) Menos de 6 meses
   ( ) Entre 6 meses e 1 ano
   ( ) Entre 1 ano e 2 anos
   ( ) Entre 2 anos e 3 anos
   ( ) Mais de 3 anos
10. Você já usou algum software de visualização de informações? Qual?

    _____

B.4 Códigos SQL Utilizados no Treinamento

```
select p.nome, d.descricao, f.nota, w.nota from TB_Participante as p
inner join TB_Facebook as f
on p.Id = f.idParticipante
inner join TB_WhatsApp as w
on p.id = w.idParticipante
inner join TB_Dispositivo as d
on p.idDispositivo = d.id
where
    (w.nota between 6 and 9
    and f.nota between 4 and 8)
```

```
select p.nome, d.descricao, f.nota, w.nota from TB_Participante as p
inner join TB_Facebook as f
on p.Id = f.idParticipante
left join TB_WhatsApp as w
on p.id = w.idParticipante
inner join TB_Dispositivo as d
on p.idDispositivo = d.id
where
    (w.nota between 6 and 9 or f.nota between 4 and 8)
    and w.id is null
```

```
create view vw_itaurm_verificaretencao
as

    select c.callid, c.derivou, c.segmento, c.tpfinalizacao, atc.
    finalizacao from vw_itaurm_calls as c
    inner join tb_finalizacaoatc as atc
    on (c.tpfinalizacao = atc.desligamentoss or c.tpfinalizacao = atc.
    desligamentopf or c.tpfinalizacao = atc.desligamentopj or c.
```

```
    tpfinalizacao = atc.desligamentoplus
7       or c.tpfinalizacao = atc.tranferenciass or c.tpfinalizacao = atc.
    tranferenciapf or c.tpfinalizacao = atc.tranferenciapj or c.
    tpfinalizacao = atc.tranferenciaplus)
8    left join vw_itaurm_callsderivadas as d
9    on c.callid = d.callid
10   where
11      (atc.finalizacao like '%desligouapos%'
12          or atc.finalizacao = 'atendimentoforahorarioapp30h')
13      and d.callid is null
14 go
15
16 create view vw_itaurm_verificahangup
17 as
18    select c.callid, c.segmento, fatc.finalizacao, c.derivou from
    tb_finalizacaoatc as fatc
19    inner join vw_itaurm_calls as c
20    on (c.tpfinalizacao = fatc.desligamentoss
21       or c.tpfinalizacao = fatc.desligamentopf
22       or c.tpfinalizacao = fatc.desligamentopj
23       or c.tpfinalizacao = fatc.desligamentoplus)
24    left join vw_itaurm_callsderivadas as d
25    on c.callid = d.callid
26    left join vw_itaurm_verificaretencao as r
27    on c.callid = r.callid
28
29    where
30       d.callid is null
31       and r.callid is null
32 go
33
34 create view VW_ITAURM_CallsDerivadas
35 as
36    select * from VW_ITAURM_Calls
37    where Derivou = 'S'
38 go
39
40 create view vw_itaurm_calls
41 as
42    select c.* from tb_call as c
43    left join tb_call_invalidas as ci
44    on c.callid = ci.callid
45    where
```

```
46          ci.callid is null
47  go
```

B.5 Código SQL Utilizado no Experimento

```
1   create view vw_itaurm_calls
2   as
3       select c.* from tb_call as c
4       left join tb_call_invalidas as ci
5       on c.callid = ci.callid
6       where
7           ci.callid is not null
8   go
9
10  create view vw_itaurm_callsidentautent
11  as
12      select c.callid, c.segmento, c.derivou, n.codigo, a.ferramenta, a.
        reconhecimento, n.sequencia from vw_itaurm_calls as c
13      inner join tb_navega as n
14      on c.callid = n.callid
15      inner join tb_atc as a
16      on (n.codigo = a.codigoss or n.codigo = a.codigopf or n.codigo = a.
        codigopj or n.codigo = a.codigoplus)
17      where a.form = 'TRANS' and reconhecimento in ('OK', 'NOK')
18  go
19
20  create view vw_itaurm_callsidentificadas
21  as
22      select callid, segmento from vw_itaurm_callsidentautent
23      where
24          ferramenta = 'agenciaconta'
25          and reconhecimento = 'NOK'
26  go
27
28  create view vw_itaurm_callsautenticadassenha
29  as
30      select callid, sequencia from vw_itaurm_callsidentautent
31      where
32          (
33          ferramenta = 'senhacartao'
34          or ferramenta = 'senhaeletronica'
```

```
35          )
36          and reconhecimento = 'OK'
37 go
38
39 create view vw_itaurm_callsautenticadastoken
40 as
41      select callid, segmento from vw_itaurm_callsidentautent
42      where
43          (
44          ferramenta = 'tokenapp'
45          or ferramenta = 'tokensms'
46          or ferramenta = 'senhaplus'
47          )
48          or reconhecimento = 'OK'
49 go
```

B.6 Especificação do Código SQL Utilizado no Experimento

Introdução:

- Chamadas identificadas: Chamadas que forneceram com sucesso o número da agencia e o número da conta;
- Chamadas autenticadas: Chamadas que forneceram com sucesso a senha e/ou token;
    - Se autenticada por senha, pode ser por três meios diferentes:
        - Senha Cartão;
        - Senha Eletrônica;
        - Senha PLUS.
    - Se autenticadas por token, pode ser por três meios diferentes:
        - Token Aplicativo;
        - Token SMS;
        - Token cartão.
- OK: Sucesso;
- NOK: Fracasso.

Requisitos:

1. Deve ser possível extrair o id e segmento de cada chamada que foi identificada com sucesso.
2. Deve ser possível extrair o id e segmento de cada chamada que foi autenticada através de senha com sucesso.
3. Deve ser possível extrair o id e segmento de cada chamada que foi autenticada através de token com sucesso.

Para satisfazer os requisitos, foram criadas 3 VIEWs (uma para cada requisito) e mais 2 VIEWs auxiliares.

Para o requisito 1, foi criada a VIEW `vw_itaurm_callsidentificadas`

- A busca deve ser feita na VIEW auxiliar `vw_itaurm_callsidentautent`
- Os campos que devem ser retornados correspondem ao id e ao segmento de cada chamada;
- A pesquisa deve retornar apenas ids e segmentos de chamadas que foram reconhecidas com sucesso no estado 'agenciaconta';
- O resultado não pode conter ids duplicados (deve haver apenas 1 registro para cada chamada).

Para o requisito 2, foi criada a VIEW `vw_itaurm_callsautenticadassenha`

- A busca deve ser feita na VIEW auxiliar `vw_itaurm_callsidentautent`
- Os campos que devem ser retornados correspondem ao id e ao segmento de cada chamada;
- A pesquisa deve retornar apenas ids e segmentos de chamadas que foram reconhecidas com sucesso nos seguintes estados: 'senhacartao', 'senhaeletronica' ou 'senhaplus';
- O resultado não pode conter ids duplicados (deve haver apenas 1 registro para cada chamada).

Para o requisito 3, foi criada a VIEW `vw_itaurm_callsautenticadastoken`

- A busca deve ser feita na VIEW auxiliar `vw_itaurm_callsidentautent`
- Os campos que devem ser retornados correspondem ao id e ao segmento de cada chamada;
- A pesquisa deve retornar apenas ids e segmentos de chamadas que foram reconhecidas com sucesso nos seguintes estados: 'tokenapp, 'tokensms' ou 'tokencartao';
- O resultado não pode conter ids duplicados (deve haver apenas 1 registro para cada chamada).

Há outras duas VIEWs auxiliares, que são utilizadas de alguma maneira pelas três VIEWs principais:

`vw_itaurm_calls`
Deve trazer todas as colunas da tabela `tb_call`, eliminando as chamadas inválidas, contidas na tabela `tb_call_invalidas`.

`vw_itaurm_callsidentautent`
Deve buscar as informações necessárias para que seja possível selecionar as chamadas identificadas e autenticadas. (essa VIEW não precisa ser inspecionada)

B.7 Lista de Erros Conhecidos do Código SQL Utilizado no Experimento

Defeitos:

`vw_itaurm_calls`

- Verificação de null errada. Não pode haver o ‘not’.

`vw_itaurm_callsidentificadas`

- Não possui ‘distinct’ na coluna ‘callId’
- Verificacao do reconhecimento errado. Deveria ser ‘OK’

`vw_itaurm_callsautenticadassenha`

- Não possui ‘distinct’ na coluna ‘callId’
- Esta trazendo ‘Sequencia’ ao invés de ‘Segmento’
- Esta faltando uma verificação ‘or ferramenta = 'senhaplus'’

`vw_itaurm_callsautenticadastoken`

- Não possui ‘distinct’ na coluna ‘callId’
- 'senhaplus' deveria ser ‘tokencartao’
- ‘or reconhecimento = 'OK'’ deveria ser ‘and’