UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

CRISTA: Um Apoio Computacional para Atividades de Inspeção e Compreensão de Código

DANIEL DE PAULA PORTO

São Carlos - SP
Maio/2009

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária da UFSCar

P853ca

Porto, Daniel de Paula.
CRISTA : um apoio computacional para atividades de inspeção e compreensão de código / Daniel de Paula Porto. -- São Carlos : UFSCar, 2009.
246 p.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2009.

1. Engenharia de software. 2. Inspeção de software. 3. Visualização de software. 4. Compreensão de dados. 5. Manutenção de programas. 6. Engenharia reversa. I. Título.

CDD: 005.1 (20ª)

Universidade Federal de São Carlos

Centro de Ciências Exatas e de Tecnologia

Programa de Pós-Graduação em Ciência da Computação

"CRISTA: Um Apoio Computacional para Atividades de Inspeção e Compreensão de Código"

DANIEL DE PAULA PORTO

Dissertação de Mestrado 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.

Membros da Banca:

Profa. Dra. Sandra Camargo P. Ferraz Fabbri
(Orientadora – DC/UFSCar)

Prof. Dr. Manoel G. de Mendonça Neto
(Co-orientador - UFBA)

Profa. Dra. Maria Cristina Ferreira de Oliveira
(ICMC/USP)

Prof. Dr. Marcos Lordello Chaim
(EACH/USP)

São Carlos
Maio/2009

Dedico a quem sempre esteve comigo, mesmo estando tão longe de mim. A quem sempre me abençoou e nas noites mais inquietantes me deu sono tranqüilo. A quem, nos períodos de angústia e desânimo, me lembrou que a persistência é gratificante. A quem me fez sentir saudade, pois sua presença sempre foi muito especial.

Dedico este trabalho a minha mãe.

AGRADECIMENTO

Inicialmente a Deus, pelo dom da vida, pela proteção e por todos os itens listados abaixo.

A minha orientadora, Profª. Drª. Sandra C.P.F. Fabbri, pela confiança no meu trabalho, pela presença constante, além da excelente orientação, transmitindo sábios ensinamentos dos mais variados assuntos.

Aos meus professores Nicolas Anquetil, Káthia Oliveira e Rejane Figueiredo da Universidade Católica de Brasília, pelo apoio e incentivo desde o início.

À minha mãe por me fazer compreender que certos ensinamentos moldam o caráter e a honestidade das pessoas e que esses ensinamentos não são lecionados em nenhuma faculdade.

Ao meu pai, que muitas vezes não me deu o que eu quis, mas me fez perceber que o que ele me proporcionou era tudo o que eu precisava.

À minha irmã pelo carinho e torcida constante desde o início da jornada.

À todos os meus parentes pelo apoio, incentivo e torcida.

À minha namorada, Marina, pelo amor, carinho, incentivo, ânimo e presença em todos os momentos possíveis e imagináveis.

Aos amigos da turma de 2007 do PPGCC por toda cumplicidade.

Aos amigos Lapesianos Elis, Maru, Denis, Luiz, Renan, Deyse, Fábio, e Henrique pela amizade, companheirismo e momentos de descontração.

Aos alunos da graduação, da pós-graduação da UFSCar e da pós-graduação da USP pela compreensão na aplicação dos experimentos.

Ao aluno Augusto Zamboni, pela ajuda nos experimentos e pela ajuda com a ferramenta CRISTA.

A CAPES pelo apoio financeiro.

Defeitos não fazem mal, quando há vontade e poder de os corrigir.
Machado de Assis

A mente que se abre a uma nova idéia jamais voltará ao seu tamanho original.
Albert Einstein

RESUMO

Inspeção de software é uma atividade chave de garantia de qualidade de software que pode ser aplicada durante todo o processo de desenvolvimento uma vez que é uma atividade estática, baseada essencialmente em técnica de leitura. Dependendo do artefato inspecionado, é preciso aplicar a técnica apropriada. No caso de inspeção de código uma técnica comumente utilizada é a *Stepwise Abstraction* (SA). No entanto, sua aplicação é trabalhosa e consome muito tempo. Com o objetivo de auxiliar e facilitar a aplicação da SA, este trabalho apresenta a CRISTA (*Code Reading Implemented with Stepwise Abstraction*), uma ferramenta que apóia o processo de inspeção baseado em SA. Essa ferramenta usa uma metáfora visual para facilitar a navegação pelo código e possui vários recursos que ajudam na compreensão do código e em sua documentação. Devido a esses recursos, a CRISTA também auxilia nas atividades de engenharia reversa, re-engenharia e manutenção. Foram realizados três estudos experimentais com o objetivo de se obter uma realimentação sobre a usabilidade e a utilidade da ferramenta em atividades de inspeção e manutenção. Os resultados fornecem evidências de que a CRISTA é fácil de ser utilizada e apóia adequadamente o processo de inspeção, bem como a leitura de código utilizando a *Stepwise Abstraction*. Além disso, no contexto de manutenção, os recursos da ferramenta ajudam a diminuir o tempo dessa atividade.

Palavras-chave: Inspeção de Código, *Stepwise Abstraction*, Visualização de Software, Compreensão de Código, Manutenção, Engenharia Reversa e Reengenharia.

ABSTRACT

Software inspection is a key activity of software quality assurance that can be applied in the whole development process since it is a static activity essentially based on reading. Depending on the artifact that is being inspected, we need to apply the appropriated reading technique. Stepwise Abstraction (SA) is a reading technique commonly used in code inspections. However, its application is laborious and time consuming. Aiming to help and facilitate the application of SA, this work presents CRISTA (Code Reading Implemented with Stepwise Abstraction), a tool to support SA-based inspection processes. This tool uses a visual metaphor to facilitate code navigation and has several resources to help program understanding and documentation. Due to these resources, CRISTA is also helpful for reverse engineering, re-engineering and maintenance activities. Three experimental studies were carried out to get feedback on the tool usability and usefulness for inspections and maintenance activities. The results provide insights that CRISTA is easy to use and adequately supports the inspection process as well as code reading by Stepwise Abstraction. Besides, in the context of maintenance, its resources make this activity less time-consuming.

Key-words: Code Inspection, Stepwise Abstraction, Software Visualization, Code Comprehension, Maintenance, Reverse Engineering and Re-engineering.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

HTML - *HyperText Markup Language*

PNG - *Portable Network Graphics*

IDE - *Integrated Development Environment*

SUMÁRIO

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

Atualmente grande parte da população mundial depende de aplicações de software para realizar as suas atividades diárias. Caso alguns sistemas amplamente utilizados deixarem de funcionar, aproximadamente 40% da população mundial sentirão as conseqüências do problema (REED, 2000 apud ROCHA; MALDONADO; WEBER, 2001). Diante disso, a questão da qualidade de software passa a ser de vital importância para o sucesso, já que as empresas de software que forem capazes de integrar, harmonizar e acelerar seus processos de desenvolvimento e manutenção terão a primazia do mercado (CURTIS, 2000 apud ROCHA; MALDONADO; WEBER, 2001).

Para se obter um software com qualidade, é necessário gerenciar o esforço, a produtividade, o tempo e custo de seu desenvolvimento e, caso um artefato de software apresente defeitos, isso contribui negativamente para a sua qualidade (KALINOWSKI, 2004). Segundo Pressman (2006), enfatizar atividades de garantia de qualidade ao desenvolver um software, faz com que haja uma redução na quantidade de trabalho a ser refeito. Isso implica diretamente nos custos do software, além de diminuir o prazo para a colocação do software no mercado.

Boehm e Basili (2001) concordam que o custo do retrabalho para correção dos defeitos aumenta à medida que o processo de desenvolvimento progride. Segundo Presman (2006), o custo de correções em fases mais adiantadas de desenvolvimento pode chegar até 1000 vezes o valor de uma correção nas fases iniciais. Desta forma, devem ser

realizadas atividades que busquem encontrar e corrigir defeitos tão logo eles sejam introduzidos nos artefatos.

As atividades de VV&T (Validação, Verificação e Teste) são consideradas parte das atividades de garantia de qualidade de software, as quais devem ser adotadas durante todo o processo de desenvolvimento para garantir maior qualidade aos artefatos liberados em cada fase (BASILI *et al.*, 1996b). Tais atividades são explicitamente tratadas nos modelos de qualidade de processo como CMMI (SEI, 2008), ISO 12.207 (ISO/IEC12207, 1998) e MPS-BR (SOFTEX, 2006) e são essenciais para a implantação desses modelos.

Para as atividades de validação e verificação existem as inspeções de software, as quais auxiliam na detecção de defeitos (FAGAN, 1976). Por ser uma atividade estática, ela pode ser aplicada desde as primeiras fases do ciclo de vida de desenvolvimento de software, 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, as quais são apropriadas para o artefato em questão. Assim, técnicas que apóiam, por exemplo, a inspeção de requisitos são diferentes das técnicas que apóiam a inspeção de código.

Associada à atividade de inspeção de código, tem-se a atividade de compreensão que, segundo Vinz e Etzkorn (2006), refere-se a qualquer atividade que use métodos estáticos ou dinâmicos para identificar suas propriedades. Essas atividades visam entender o código, analisando suas estruturas e construindo representações de alto nível que o representem. 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.

A compreensão de código apóia várias atividades da engenharia de software, por exemplo: teste, inspeção, manutenção e re-engenharia (WALKINSHAW; ROPER; WOOD, 2005). Sendo assim, uma boa compreensão do código pode implicar no sucesso de todas essas atividades e, conseqüentemente, no sucesso do software.

Com o aumento da complexidade do software desenvolvido atualmente, têm sido investigadas algumas formas inovadoras para facilitar seu entendimento e compreensão. Exemplo disso são as técnicas de visualização de software (SENSALIRE; OGAO, 2007).

A visualização tem sido apontada com uma possível solução para dar suporte à compreensão de sistemas complexos. O processo cognitivo de seres humanos é mais

intuitivo, efetivo e eficiente quando apoiado por recursos visuais tais como imagens, gráficos e sinais (TERGAN; KELLER, 2005). Segundo Nascimento e Ferreira (2005), o processo de visualização está relacionado com a transformação de algo abstrato em imagens (mentais ou reais) que possam ser visualizadas pelos seres humanos.

O objetivo final da visualização é auxiliar no entendimento de determinado assunto, o qual, sem uma visualização, exigiria maior esforço para ser compreendido. Assim, a visualização de informações tem sido apontada como uma possível solução para apoiar a compreensão de sistemas complexos, procurando, no caso do código, maneiras de representá-lo de forma gráfica, para facilitar o seu entendimento (SENSALIRE; OGAO, 2007). Facilitando a compreensão do código, sua inspeção fica também facilitada.

Tilley e outros (1996) já afirmavam que ferramentas de software para apoio à compreensão de programas são de grande importância para ajudar a reduzir a complexidade inerente do processo sistemático de compreensão. As ferramentas auxiliam o programador na construção de um modelo mental do programa, através de mecanismos de análise, exploração e visualização da informação em diferentes níveis de abstração.

1.2 Motivação e Objetivos

Face ao exposto, e caracterizada a importância da inspeção de código para as atividades de VV&T e para a qualidade do software, o objetivo deste trabalho foi desenvolver uma ferramenta para auxiliar a inspeção de software. Como a inspeção utiliza uma técnica de leitura para ser realizada, investigaram-se alguns estudos sobre inspeção de código (BASILI; CALDIERA; SHULL, 1996), e escolheu-se a técnica de leitura *Stepwise Abstraction* para dar suporte à compreensão sistemática do código na ferramenta proposta. Além disso, considerando que recursos de visualização têm sido considerados um facilitador para a compreensão do código, decidiu-se também prover a ferramenta de uma metáfora visual para facilitar a compreensão.

Apesar de existirem na literatura algumas ferramentas (CARNEIRO; ORRICO; MENDONÇA, 2007; COOPER *et al.*, 2006; LINTERN *et al.*, 2003) que dão apoio à compreensão de software com uso de visualização, elas não atendem ao processo de inspeção, como a ferramenta aqui proposta. Além disso, considerando que a atividade de compreensão é essencial também para uma boa manutenção, para a engenharia reversa de um produto e sua reengenharia, a ferramenta aqui proposta se preocupa em registrar o entendimento de um código. Esse registro permite fazer uma re-documentação, de forma a minimizar o retrabalho sempre que novas manutenções forem necessárias.

1.3 Metodologia de Execução do Trabalho

O trabalho foi iniciado com uma revisão sistemática sobre inspeção de software, que deu subsídios para a concepção inicial da ferramenta CRISTA. Desde o início foi decidido que a compreensão do código, necessária à atividade de inspeção, deveria usar recurso de visualização e que a ferramenta deveria ser o mais genérica possível, permitindo usá-la facilmente para inspecionar e compreender código em várias linguagens de programação. Dessa forma, foi decidido também que a ferramenta seria independente, não estando acoplada a nenhuma IDE.

Tomadas essas decisões, fez-se um estudo de viabilidade sobre o JavaCC, identificando-se que ele permitiria um fácil reconhecimento de diferentes linguagens, tornando a ferramenta CRISTA genérica, como era desejado. Quanto ao recurso de visualização, optou-se pela técnica *Treemap* para representar o código e suas estruturas hierárquicas, pois essa técnica aproveita todo o espaço disponível na tela e provê uma boa informação sobre o estado atual do que já foi inspecionado e compreendido no código fonte. Assim, com base em uma versão inicial que contemplava grande parte das funcionalidades requeridas inicialmente para a ferramenta, a CRISTA foi aprimorada e complementada até atingir sua versão atual por meio de um processo evolutivo. Algumas funcionalidades foram idealizadas ao longo desse processo uma vez que a ferramenta pode apoiar várias atividades diferentes que dependam de compreensão de código, como é o caso de manutenção, engenharia reversa, re-engenharia e redocumentação de código.

Atingida uma versão muito próxima à versão atual, foram realizados três estudos experimentais para validar o que se tinha construído. No primeiro estudo foram avaliadas as características de usabilidade da ferramenta, cujos resultados foram muito satisfatórios. No segundo estudo foi avaliado o apoio dado pela ferramenta no contexto de inspeção, com um grupo de controle que realizou a inspeção de forma manual. Os resultados foram semelhantes com ou sem o uso da ferramenta, tanto no tempo gasto como na quantidade de defeitos encontrados, mostrando que a ela não interfere negativamente na atividade e provê o inspetor de várias informações que não se consegue de forma manual. O terceiro estudo avaliou a contribuição da ferramenta em um contexto de manutenção para um programa de domínio gráfico, após ele ter passado por uma inspeção. Os resultados mostraram a contribuição da ferramenta CRISTA que facilitou a localização dos defeitos e a redocumentação do código.

Os resultados obtidos com os estudos experimentais foram publicados nos seguintes eventos da área: Sessão de Ferramentas do Simpósio Brasileiro de Engenharia de Software (SBES 2008 - Menção Honrosa), *International Conference on Program Comprehension*

(ICPC 2009), *International Conference on Software Engineering and Knowledge Engineering* (SEKE 2009) e Workshop de Manutenção de Software Moderna (WMSWM 2009). Além desses, foi submetido um artigo (que ainda não se tem resultado) para o *International Symposium on Empirical Software Engineering and Measurement* (ESEM 2009).

1.4 Organização do Trabalho

Este trabalho é composto por 8 capítulos e 5 apêndices, cujos objetivos estão descritos a seguir:

Capítulo 1 - Introdução: esse capítulo caracteriza a área de pesquisa em que o trabalho foi desenvolvido, evidenciando as lacunas que levaram à motivação e objetivos. **Capítulo 2 - Inspeção de software**: Uma vez que a proposta principal deste trabalho foi a construção da ferramenta CRISTA, que apóia a atividade de inspeção, neste capítulo apresenta-se um levantamento bibliográfico a respeito do assunto, para caracterizar o processo de inspeção e as técnicas usadas nessa atividade. **Capítulo 3 - Visualização de informações**: Esse capítulo se fez necessário, pois para facilitar a aplicação da técnica de leitura *Stepwise Abstraction*, implementada na CRISTA para fazer a leitura do código, decidiu-se usar uma técnica de visualização para tornar a compreensão do código mais efetiva. **Capítulo 4 - Experimentação**: Para validação da ferramenta proposta foram realizados estudos experimentais, que são indispensáveis para a avaliação de nova tecnologia. Assim, apresentar os conceitos principais acerca de experimentação era fundamental para que o leitor pudesse acompanhar os estudos experimentais apresentados no trabalho. **Capítulo 5 - Trabalhos relacionados**: Algumas das idéias para a criação e avaliação da ferramenta CRISTA foram baseadas em alguns trabalhos já realizados. Assim, esse capítulo apresenta os principais trabalhos encontrados na literatura que tiveram uma maior contribuição para o desenvolvimento da CRISTA. **Capítulo 6 - A ferramenta CRISTA**: Esse capítulo é a parte principal deste trabalho, pois corresponde a um relato detalhado dos requisitos e funcionalidades contidos na ferramenta. **Capítulo 7 - Estudos experimentais**: Depois do Capítulo 6, esse capítulo é outro ponto importante deste trabalho, pois nele são apresentados os estudos que já foram realizados com o objetivo de se estabelecer uma primeira caracterização da ferramenta CRISTA. **Capítulo 8 - Conclusões**: Nesse capítulo apresenta-se uma caracterização bem geral da ferramenta, destacando-se as contribuições e limitações do trabalho, bem como os possíveis trabalhos que poderão ser realizados a partir deste. **Apêndice A - Guia de instanciação de novas linguagens**: Esse apêndice é muito importante para que o leitor compreenda como a ferramenta permite as instanciações

para diferentes linguagens. As decisões de projeto relacionadas com esse ponto da ferramenta atribuem a ela uma grande contribuição no contexto de compreensão e redocumentação de código, cruciais para as atividades de manutenção, que se deparam freqüentemente com código legado. **Apêndice B - Apresentações utilizadas nos estudos experimentais 1 e 2**: Considerou-se importante deixar registrado neste trabalho as apresentações utilizadas no treinamento da ferramenta durante os estudos experimentais 1 e 2. **Apêndice C - Programas utilizados nos estudos experimentais 1 e 2**: Assim como as apresentações, considerou-se igualmente importante que os códigos utilizados nos estudos experimentais ficassem registrados no trabalho. **Apêndice D - Questionários utilizados nos estudos experimentais 1 e 2**: Assim como os dois apêndices anteriores, considerou-se que os questionários utilizados nos experimentos 1 e 2 também deveriam ficar registrados neste documento. **Apêndice E - Dados obtidos dos experimentos 1 e 2**: Esse apêndice apresenta todos os gráficos elaborados com os resultados obtidos dos experimentos. Alguns deles estão comentados com maior detalhe no Capítulo 7, mas nesse apêndice todos eles estão disponíveis.

Na Figura 1.1 a organização dos capítulos está modelada de forma ilustrativa, em uma representação visual similar à técnica de visualização de informações *Treemap*, usada como recurso facilitador para a compreensão de código na ferramenta CRISTA. Espera-se que essa figura transmita já nessas primeiras páginas desta monografia a contribuição desse recurso para a atividade de compreensão em geral.

Figura 1.1 - Organização do trabalho

Capítulo 2

INSPEÇÃO DE SOFTWARE

Este capítulo apresenta uma revisão bibliográfica a respeito da inspeção de software. São apresentados os benefícios oriundos da inspeção, 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 (FAGAN, 1976) é um tipo particular de revisão que pode ser aplicada a todos os artefatos produzidos ao longo do ciclo de desenvolvimento do 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. Essas atividades, por sua vez, devem ser adotadas durante todo o processo de desenvolvimento para garantir maior qualidade aos artefatos liberados em cada fase do ciclo de desenvolvimento (BASILI *et al.*, 1996b). Tais atividades são explicitamente tratadas nos modelos de qualidade de processo como CMMI (SEI, 2008), ISO 12.207 (ISO/IEC12207, 1998) e MPS-BR (SOFTEX, 2006) e são essenciais para a implantação desses modelos.

No contexto de VV&T a terminologia usada na Engenharia de Software, de acordo com o padrão da IEEE (1990) é: (i) Erro (*error*): 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; (ii) Defeito (*fault*): é um passo, processo ou definição de dados incorreto. O defeito, que é geralmente conhecido como *bug*, é a causa hipotética da falha; (iii) Falha (*failure*): 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; e (iv) Engano (*mistake*): é uma ação humana que produz um resultado incorreto.

Dois termos bastante usados em inspeção de software são *discrepância* e *defeito*. O termo discrepância se refere a algo que diverge do que era esperado e costuma-se usá-lo quando se compara a abstração feita pelo inspetor com a especificação original do código que foi inspecionado. Nesse sentido, uma discrepância poderá ser considerada de fato um defeito, depois que for analisada. Por outro lado, se ao fazer uma inspeção de código o inspetor identificar um defeito, de acordo com a definição do padrão IEEE (1990), o defeito é referenciado como defeito e não como discrepância.

O restante do capítulo está organizado da seguinte forma: na Seção 2.2 são apresentados alguns benefícios de se realizar inspeções nos artefatos produzidos ao longo do processo de desenvolvimento de software. Na Seção 2.3, é descrito detalhadamente o processo de inspeção de software e suas características, bem como as evoluções sofridas durantes os anos. Em seguida, na Seção 2.4, são apresentados os conceitos sobre técnicas de leitura para detecção de defeitos em artefatos de software. Já as características peculiares das inspeções de código são apresentadas na Seção 2.5. A Seção 2.6 aborda as ferramentas que apóiam o processo de inspeção. Por fim, a Seção 2.7 conclui este capítulo.

2.2 A Inspeção de Software e seus Benefícios

A inspeção de software tem por objetivo detectar defeitos nos artefatos de software tão logo eles sejam inseridos. Fagan (1976) introduziu o método de inspeção na IBM em 1976 e, desde então, a inspeção de software é considerada uma das melhores práticas de detecção de defeitos da engenharia de software (ANDERSON; REPS; TEITELBAUM, 2003; ANDERSON *et al.*, 2003).

A atividade de inspeção é uma abordagem estruturada e sistemática, que possui um processo de detecção de defeitos rigoroso e bem definido. Ela visa encontrar defeitos em vários tipos de documentos como: documentos de texto, modelos, gráficos, fragmentos de código, etc. (WINKLER; THURNHER; BIFFL, 2007). A Figura 2.1 ilustra possíveis momentos para se realizar inspeções nos diferentes artefatos de software durante todo o seu ciclo de vida (KALINOWSKI, 2004).

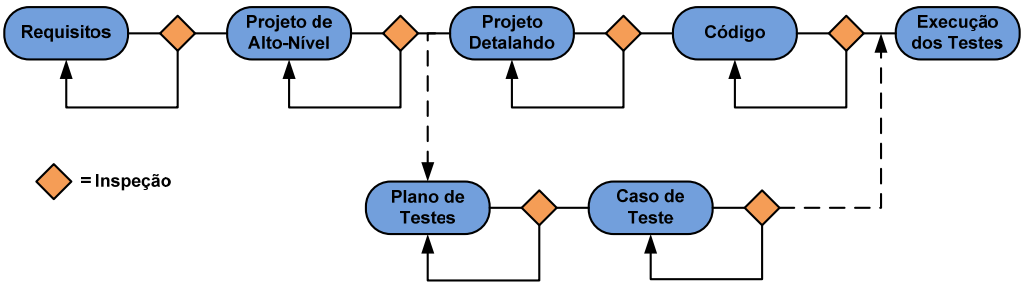


Figura 2.1 - Momentos propícios para realizar inspeções de software (Adaptado de KALINOWSKI, 2004)

Como pode ser visto na Figura 2.1, a inspeção não requer um código executável como as atividades de teste, e pode ser aplicada logo nas primeiras fases do ciclo de vida de desenvolvimento de software. Sendo assim, diferentemente da atividade de teste, a inspeção é considerada uma atividade estática, pois não implica na execução do artefato que está sendo inspecionado (MYERS *et al.*, 2004). 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).

Uma das maiores vantagens da inspeção de software é que ela pode ser realizada antes da atividade de teste (BOOGERD; MOONEN, 2006). Isso se torna interessante, uma vez que o custo da correção de defeitos cresce exponencialmente quando os defeitos são passados para as próximas fases do ciclo de vida do software, como é mostrado na Figura 2.2. Assim, um defeito inserido na fase de requisitos e identificado na fase de codificação gera um custo dez vezes maior para ser corrigido nessa fase ao invés de ser corrigido na própria fase de requisitos. Essa situação foi modelada em 1981 por Bohem e continua sendo citada em trabalhos mais recentes como o de Pressman (2006).

Pesquisas mostram que a inspeção aplicada nos primeiros estágios do ciclo de vida de desenvolvimento de software reduz significativamente o custo e o tempo de se encontrar defeitos (CHOWDHURY; LAND, 2004; THELIN *et al.*, 2004a). Os defeitos de especificação de requisitos não detectados nessa fase são considerados muito caros, uma vez que eles irão afetar todas as fases posteriores do desenvolvimento (BERLING; THELIN, 2004). Sendo assim, é altamente desejável, do ponto de vista econômico e de confiabilidade, que os defeitos sejam detectados e mitigados o quanto antes (BOOGERD; MOONEN, 2006; GUPTA; PATNAIK; GOEL, 2003; KALINOWSKI; TRAVASSOS, 2004). Por isso, de acordo com Chowdhury e Land (2004), é conveniente aumentar a eficácia da inspeção de requisitos.

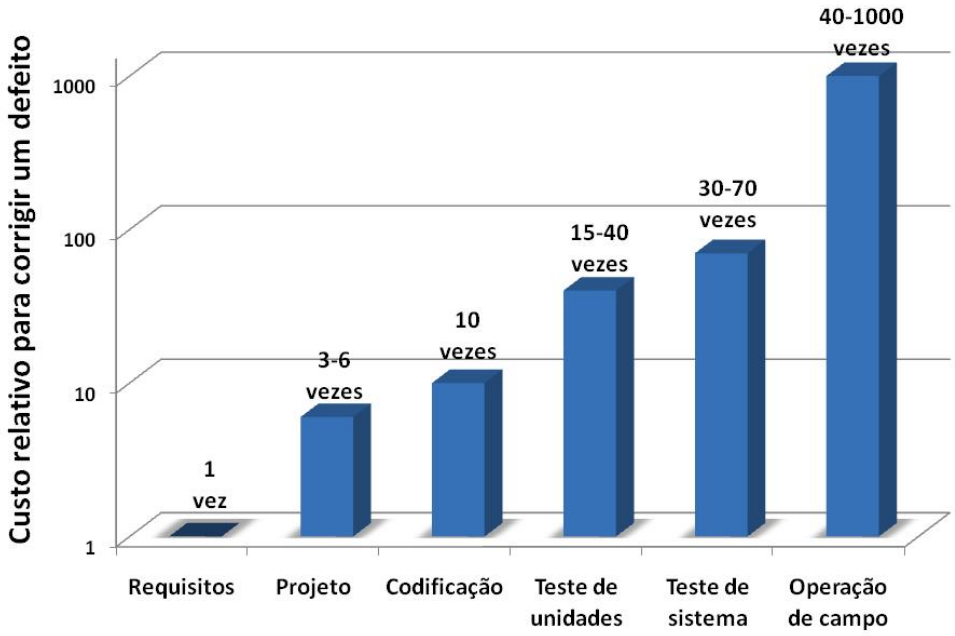


Figura 2.2 - Custo relativo para corrigir um defeito (BOEHM, 1981 apud PRESSMAN, 2006)

Segundo alguns autores, como Denger e Shull (2007), as inspeções identificam a maioria dos defeitos existentes. De acordo com a Tabela 2.1, as inspeções formais de projeto e de código se destacam pela eficácia dentre outras alternativas. Harjumaa, Tervonen e Vuorio (2004), também comentam que as inspeções de software constituem o método de garantia de qualidade que promove o maior retorno de investimento.

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 70%**
Teste de unidades	15% a 50%
Teste de integração	25% a 40%
Teste do sistema	25% a 55%
Beta Teste (< 10 clientes)	24% a 40%
Beta Teste (> 1000 clientes)	60% a 85%

Aumentar a qualidade dos artefatos, detectando os defeitos nele contidos, é um dos principais propósitos da inspeção, embora ela também seja útil para fornecer informações que podem ser utilizadas no suporte às tomadas de decisões acerca do projeto (THELIN *et al.*, 2004b). Nesse sentido, as inspeções podem contribuir auxiliando os gerentes de projeto

a gerenciar os riscos, além de auxiliar os engenheiros de software, apontando melhorias específicas no processo (BIFFL; GRÜNBACHER; HALLING, 2006).

Segundo Winkler, Thurnher e Biffl (2007), a literatura atual considera a inspeção de software como um método apropriado para a detecção de defeitos, aprendizado e introdução de novos membros na equipe de desenvolvimento.

2.3 O Processo de Inspeção de Software

O processo de inspeção de software foi proposto inicialmente por Fagan (1976) e passou por algumas modificações ao longo do tempo. Ele é formado por etapas rigorosas e bem definidas e executado por várias pessoas, com papéis bem definidos.

As evoluções do processo ocorreram principalmente nas etapas-chave do processo proposto por Fagan: “Preparação”, que corresponde a uma preparação individual do inspetor e a “Inspeção”, que corresponde a uma reunião de inspeção. Tais evoluções são descritas a seguir, a começar pelo primeiro processo de inspeção definido (FAGAN, 1976).

2.3.1 O processo tradicional de inspeção segundo Fagan (1976)

Em seu trabalho, Fagan (1976) propôs um processo de inspeção com as seguintes etapas: *Apresentação*, *Preparação*, *Inspeção*, *Retrabalho* e *Continuação*. Cada uma das etapas possui as seguintes características:

1) **Apresentação:** São repassadas aos membros da equipe todas as informações sobre o artefato que são necessárias à inspeção.

2) **Preparação**: Os participantes estudam os artefatos individualmente, tentando entender seu projeto, objetivos e lógica. Eventualmente são feitas anotações sobre os artefatos, produzindo assim uma lista preliminar de defeitos.

3) **Inspeção**: É feita uma reunião com o objetivo de procurar defeitos. O Leitor escolhido pelo Moderador (geralmente o Implementador) descreve como foi codificado o que foi projetado. À medida que os defeitos são encontrados, eles são anotados sem a preocupação de resolvê-los. Depois da reunião o Moderador produz um relatório contendo todos os defeitos identificados.

4) **Retrabalho**: Todos os defeitos encontrados são corrigidos pelo autor do artefato.

5) **Acompanhamento**: O Moderador verifica cada correção feita pelo autor e verifica a necessidade de uma nova inspeção.

Ao propor o processo de inspeção, Fagan (1976) também propôs papéis bem definidos para a equipe de inspeção. Segundo o autor, a equipe de inspeção deve ser composta por participantes de diversas áreas, pois cada um vai analisar 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, conforme proposto originalmente por Fagan (1976), é a seguinte:

a) **Moderador:** O Moderador é a pessoa chave do processo de inspeção. Além de liderar a equipe, ele também é o responsável por garantir o sucesso da inspeção. Não necessariamente deve ser um especialista no produto que está sendo analisado. Ele é responsável por escolher o local mais adequado para a reunião, além de acompanhar o que foi realizado na etapa de Retrabalho. O Moderador deve usar toda a sua sensibilidade pessoal e habilidade para obter uma sinergia entre os participantes. Para melhores resultados o Moderador deve ser treinado (um breve, mas efetivo treinamento).

b) **Projetista:** Pessoa responsável por projetar o software.

c) **Implementador:** Pessoa responsável por codificar o que foi previamente projetado.

d) **Testador:** Pessoa responsável por escrever e/ou executar casos de teste ou, de qualquer outra forma, testar o produto do projetista e do implementador.

A Figura 2.3 apresenta as etapas e papéis envolvidos no processo de inspeção.

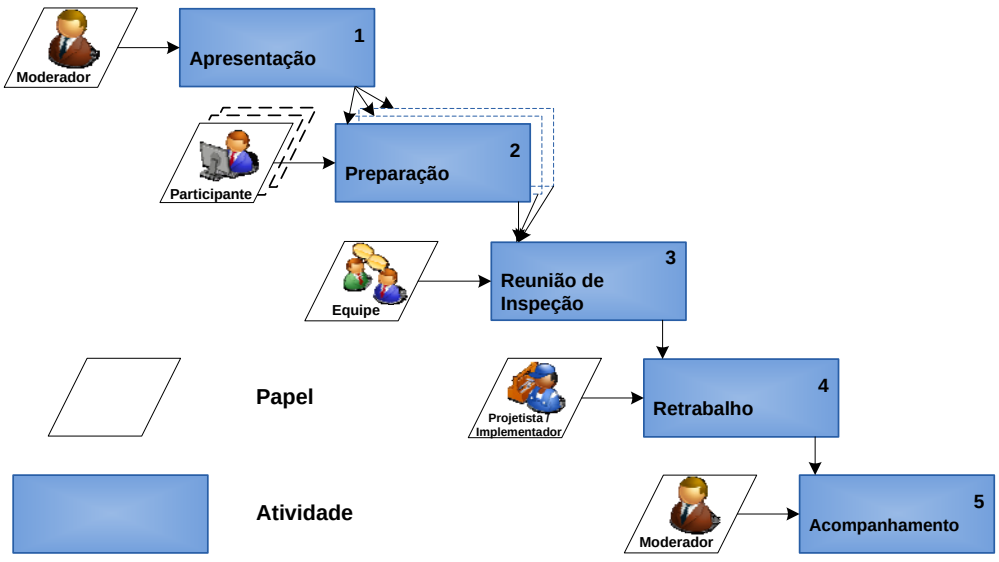


Figura 2.3 - Visão do processo de inspeção proposto por Fagan em 1976 (Adaptada de KALINOWSKI, 2004)

2.3.2 Evolução do processo tradicional 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.

2.3.3 O processo de inspeção segundo Humphrey (1989)

Em 1989, Humphrey propôs uma evolução do processo de inspeção proposto por Fagan (1986). Sua principal diferença consiste em mudar o foco da atividade de Preparação, que era de tentar entender o artefato a ser inspecionado, para efetivamente encontrar seus defeitos. Sendo assim, ao final dessa atividade, cada um dos inspetores entrega uma lista de defeitos para o autor do documento antes da reunião na etapa de Inspeção. Dessa forma, durante a reunião não se procura mais por defeitos, e passa-se a discutir os defeitos encontrados e classificá-los como defeito ou falso-positivo. As etapas definidas no processo de Humphrey são: Inicialização (Planejamento), Preparação, Inspeção e Pós-inspeção.

2.3.4 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). Esse guia é rico em detalhes e explicações acerca de cada etapa e papel envolvido em uma inspeção. Nele, o processo de inspeção é dividido nas etapas: Planejamento, Apresentação, Preparação, Reunião de Inspeção, Terceiro Momento, Retrabalho e Continuação.

A etapa Inspeção encontrada nos processos de Fagan e Humphrey, foi renomeada pela NASA para Reunião de Inspeção. Entretanto, o objetivo dessa etapa continuou a ser a discussão dos defeitos encontrados individualmente, como proposto por Humphrey. Outra diferença foi o acréscimo da etapa Terceiro Momento, a qual oferece um tempo adicional para discussões, soluções de problemas ou fechamento de questões levantadas durante a Reunião de Inspeção. Segundo a NASA, essa etapa é opcional.

Esse guia ainda especifica de maneira clara todos os papéis e suas respectivas responsabilidades em um contexto de inspeção. Dentre esses papéis, cabe ressaltar a denominação de dois novos papéis:

a) 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 os defeitos no artefato. Os principais candidatos a inspetores são as pessoas envolvidas com o produto a ser inspecionado (envolvidas nas etapas anteriores, corrente e posteriores do ciclo de vida). 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.

b) Documentador: O Documentador é a pessoa responsável por registrar na lista de defeitos cada defeito encontrado, bem como registrar as decisões e recomendações feitas durante a Reunião de Inspeção. Essas atividades eram anteriormente realizadas pelo Moderador no processo de Fagan (1976).

Como dito anteriormente, 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.

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

A partir das propostas de processo de inspeção anteriores, Johnson (1994) propôs um processo chamado FTArm (acrônimo em inglês para o termo “Método de Revisão Técnica Formal Assíncrona”). O FTArm leva em conta que reuniões de inspeção podem ser evitadas, reduzindo assim, custos e conflitos de alocação de recursos sem sacrificar a eficácia da inspeção. As etapas do processo FTArm são: Configuração (Planejamento), Orientação (Apresentação), Revisão Particular (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, 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 básico de cada uma dessas etapas é definido a seguir:

- A etapa de **Revisão Pública** tem por objetivo discutir os problemas identificados durante a etapa de Revisão Particular. A diferença dessa etapa para a etapa de Reunião de Inspeção no processo da NASA, é que essa etapa é feita de forma assíncrona e usa votações para chegar a um consenso entre os inspetores a cerca de cada ponto levantado durante a Revisão Particular. Essa etapa termina quando todos os inspetores revisaram todos os pontos e a votação tenha se estabilizado.

- A etapa de **Consolidação** é realizada pelo moderador da inspeção que agrupa 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 (FAGAN, 1976).

- A etapa de **Reunião de Revisão em Grupo** serve para se discutir os problemas não resolvidos observados pelo relatório produzido na etapa de Consolidação.

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

2.3.6 O processo de inspeção segundo Sauer e outros (2000)

Com base em diversos estudos experimentais sobre inspeções de software, Sauer e outros (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 tradicional de Fagan (1986), com as reuniões assíncronas do processo FTArm de Johnson (1994) e considerando equipes geograficamente distribuídas.

Basicamente, a variação de Sauer substitui as atividades de Preparação e de Inspeção (Reunião de Inspeção) do processo tradicional por três novas atividades seqüenciais: Detecção de Defeitos, Coleção de Defeitos e Discriminação de Defeitos. Entretanto, o processo de inspeção mantém a mesma estrutura.

Kalinowski (2004) apresenta uma representação visual para o processo sugerido por Sauer e outros (2000), a qual é apresentada na Figura 2.4.

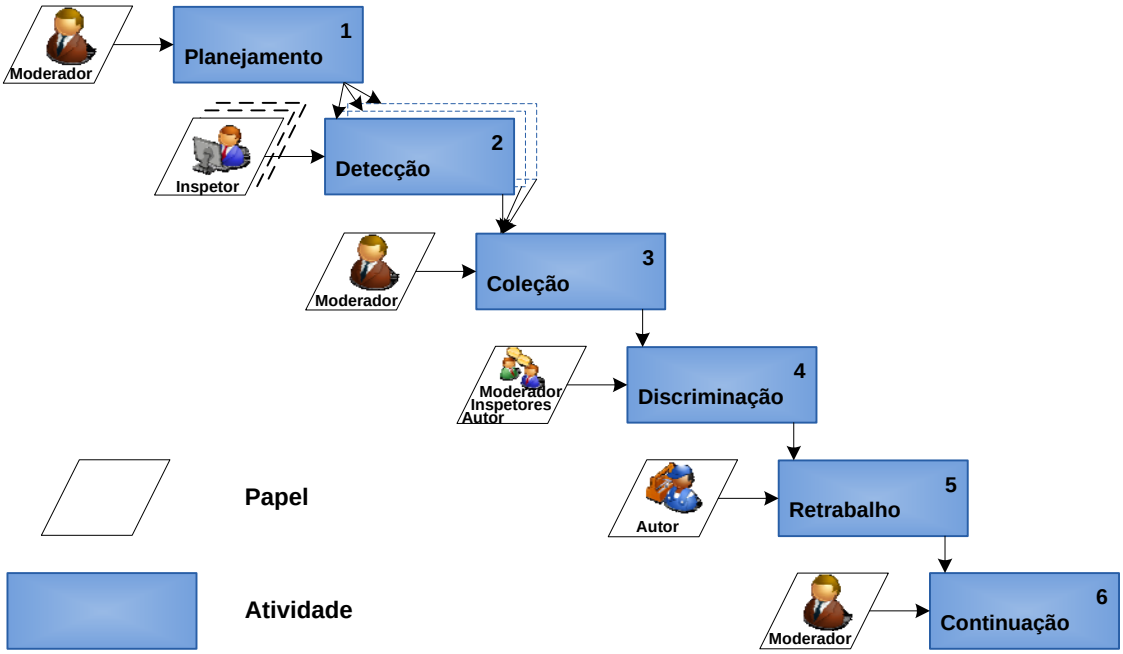


Figura 2.4 - Visão do processo de inspeção proposto por Sauer em 2000 (Adaptado de KALINOWSKI, 2004)

A etapa de Detecção corresponde à etapa de Preparação do processo da NASA (1993), ou seja, tem por objetivo encontrar os defeitos nos artefato. A Recém criada etapa de Coleção possui o mesmo papel da etapa de Consolidação de Johnson (1994), ou seja, tem pó objetivo condensar os resultados obtidos durante a etapa de Detecção. Entretanto, Sauer e seus companheiros acrescentaram que, dependendo da equipe de inspeção e da

taxa de falso-positivos[1], os defeitos encontrados pelos inspetores podem passar direto para a etapa de Retrabalho.

A etapa de Discriminação sugerida por Sauer e outros (2000) veio substituir a antiga etapa de Reunião de Inspeção. Nessa etapa, o Moderador, o Autor do artefato e os Inspetores discutem os defeitos encontrados de forma assíncrona. Durante esta discussão, alguns defeitos serão classificados como falso-positivo e outros como defeito. Os falso-positivos serão descartados e os defeitos serão registrados em uma lista de defeitos, que então será passada para o Autor para que a correção possa ocorrer (etapa de Retrabalho).

Segundo Grünbacher e outros (2003), os falso-positivos podem se tornar um problema se ocorrerem com freqüência, pois eles podem se passar por defeitos verdadeiros, gastando-se muito tempo tentando consertá-los desnecessariamente.

Essa variação do processo de inspeção proposto por Sauer e outros (2000) permite a utilização de um número grande de inspetores em paralelo para a detecção de defeitos, o que, segundo Lanubile e Mallardo (2003) aumenta a probabilidade de se encontrar defeitos difíceis de serem encontrados. Kalinowski (2004) argumenta que um grande número de inspetores na equipe tem efeito no custo. Todavia, não tem efeito no tempo de detecção de defeitos e também não implica em problemas de coordenação.

2.4 Técnicas de Leitura de Artefatos de Software

Vários autores já relataram a importância das inspeções para aumentar a qualidade de software (BERLING; THELIN, 2004; BOOGERD; MOONEN, 2006; CHOWDHURY; LAND, 2004; KALINOWSKI, 2004; TRAVASSOS, 2007; VITHARANA; RAMAMURTHY, 2003). Entretanto, existem vários fatores que têm um impacto significativo no número de defeitos detectados em um artefato de software. Exemplo disso são a habilidade e a experiência do inspetor na realização de inspeções. Entretanto, muitos estudos experimentais têm colocado o foco em outros fatores como técnicas de leitura e reuniões de equipe (CHOWDHURY; LAND, 2004).

As reuniões de equipe são uma atividade chave no processo de inspeção e, por isso, deve-se tomar muito cuidado para conduzí-las de maneira correta, como por exemplo, direcionando as críticas ao artefato ao invés do autor como menciona Kollanus (2005). Pressman (2006) também define algumas diretrizes para uma boa reunião.

[1] Os falso-positivos são defeitos relatados que não foram considerados como defeitos após a reunião de consenso.

Outro fator que tem forte impacto no processo de inspeção são as técnicas de leitura, que descrevem como o revisor deve ler o artefato para encontrar defeitos (BERLING; THELIN, 2004). Essas técnicas ajudam os inspetores no processo de detecção de defeitos provendo uma seqüência bem definida de passos para a leitura dos artefatos (WINKLER; THURNHER; BIFFL, 2007).

Segundo Basili e outros (1996b), as técnicas de leitura são um conjunto de passos para a análise individual de um artefato que permitem alcançar a compreensão necessária para uma determinada tarefa. Essas técnicas, segundo Kalinovski (2004), visam aumentar o número de defeitos encontrados pelos inspetores, além de reduzir a influência do fator humano nos resultados de uma inspeção. Além disso, essas técnicas podem representar modelos para escrever artefatos de maior qualidade.

Entre as técnicas de leitura mais conhecidas estão a *ad-hoc* e a *checklist*. As *checklists* são utilizadas tanto para detectar defeitos quanto para avaliar características de qualidade do artefato. Existem ainda técnicas de leitura mais rígidas, com procedimentos mais explícitos e sistemáticos, como a técnica baseada em perspectiva (PBR) (SHULL; RUS; BASILI, 2000) e a técnica de projetos orientados a objeto (OORT's) (TRAVASSOS *et al.*, 1999). PBR apóia a detecção de defeitos em documentos de requisitos, enquanto OORT's contempla diagramas e documentos de projeto orientados a objeto (BASILI *et al.*, 1996b). Na Tabela 2.2, é apresentada uma comparação entre as técnicas de leitura *Ad-Hoc*, *Checklist* e técnicas mais rígidas.

Tabela 2.2 - Comparação entre as técnicas de inspeção de software (TRAVASSOS, 2007)

Técnica	Notação	Sistemática?	Focada?	Melhoria Controlada?	Adaptável?	Treinamento necessário?
Ad-Hoc	Qualquer	Não	Não	Não	Não	Não
Checklist	Qualquer	Parcialmente	Não	Parcialmente	Sim	Parcialmente
Técnicas mais rígidas	Linguagem Natural	Sim	Sim	Sim	Sim	Sim

Inspecionar o artefato com uma técnica de leitura *Ad-hoc* implica em o revisor checar o artefato usando suas habilidades de encontrar defeitos. A técnica *ad-hoc* não dá nenhum direcionamento específico para o revisor sobre como verificar o documento. Nessa abordagem simplesmente é assumido que todos participantes inspecionam o artefato efetivamente (MENDES, 2006). Sendo assim, a técnica *ad-hoc* apresenta algumas desvantagens (BERLING; THELIN, 2004):

- O número de defeitos encontrados fica dependente exclusivamente da habilidade e experiência do inspetor;
- O resultado da inspeção pode variar bastante entre vários inspetores;

- Se o inspetor achar que uma leitura rápida no artefato é suficiente, não existe nenhum guia para apoiar uma inspeção mais profunda dos artefatos;
- Os revisores não aprendem uns com os outros; e
- O procedimento não é documentado, nenhum conhecimento é transferido entre os inspetores e nem de uma inspeção para outra.

Um pouco mais controlada que a *ad-hoc*, a técnica *checklist* provê uma lista (geralmente é constituída de questões que devem ser respondidas) para os revisores usarem durante a inspeção. O propósito do *checklist* é apoiar o revisor a detectar mais defeitos à medida que se vai respondendo as questões, as quais procuram manter o foco em detalhes específicos do artefato (BERLING; THELIN, 2004). Segundo Abdelnabi e outros (2004), essas questões podem ser muito superficiais ou até mesmo sofisticadas e complexas.

A idéia por trás dos *checklists* é a captura de experiências, isto é, os conhecimentos sobre problemas e defeitos típicos são transformados em perguntas. Entretanto, os *checklists* também apresentam alguns pontos negativos (MENDES, 2006):

- Para que o *checklist* não seja muito específico, as perguntas devem ser muito genéricas o que acaba deixando o documento muito extenso;
- Os *checklists* são geralmente longos e contêm muitas perguntas. Com isso, a equipe tende a ignorar algumas perguntas, para não deixar o processo muito tedioso e demorado;
- Os *checklists* impõem à equipe que verifique todas as informações dos documentos inspecionados, o que pode fazer com que haja dedicação a detalhes desnecessários do documento; e
- Todos os integrantes da equipe usam a mesma lista de verificação. Em uma equipe de inspeção, todos verificam os mesmos aspectos do documento.

Conforme apresentado na Tabela 2.2, as técnicas *Ad-Hoc* e *checklist* possuem pouco formalismo. Com isso, muitas técnicas de leitura têm sido relatadas na literatura no intuito de deixar a leitura do código mais rígida e formal (ANDERSON; REPS; TEITELBAUM, 2003; THELIN *et al.*, 2004b).

Como exemplo de técnicas de leituras mais formais temos: *Defect-based Reading* (DBR), *Usability based Reading* (UBR), *Perspective-based Reading* (PBR), e *Object-Oriented Reading Technique* (OORT) (BIFFL; GRÜNBACHER; HALLING, 2006; THELIN, 2003; TRAVASSOS *et al.*, 1999). Essas técnicas são detalhadas a seguir:

a) DBR (*Defect-Based Reading*)

A técnica de leitura DBR representa uma família de técnicas de leitura para a detecção de defeitos em documentos de requisitos. Cada técnica DBR foi projetada para focar uma determinada classe de defeitos específicos (MAFRA; TRAVASSOS, 2005). Sua idéia principal é fazer os diferentes inspetores focarem em classes de defeitos diferentes enquanto estão inspecionando o artefato (BERLING; THELIN, 2004).

Para cada classe de defeitos, existe uma série de questões para identificá-la. Essas questões são compostas de uma série de passos, chamados de cenários, a serem seguidos enquanto a leitura está sendo realizada. Enquanto é feita a leitura do documento e são seguidos os passos, o leitor tenta responder as questões do cenário (BASILI; CALDIERA; SHULL, 1996).

Embora a técnica DBR existente seja direcionada principalmente a documentos de requisitos (BERLING; THELIN, 2004), ela também pode ser aplicada em outros documentos. Conseqüentemente, pode ser necessário criar uma lista de classes de defeitos para os outros documentos, e criar técnicas de leitura que foquem classes específicas de defeito (MENDES, 2006).

b) UBR (*Use-Based Reading*)

A técnica UBR utiliza casos de uso para guiar o revisor durante a inspeção. Essa técnica pretende inspecionar o artefato com o foco no ponto de vista do usuário para detectar os defeitos mais críticos de forma mais eficiente (BERLING; THELIN, 2004; MENDES, 2006). Uma das principais características da UBR é que ela assume que diferentes defeitos possuem importâncias diferentes para o usuário (THELIN, 2003).

Na técnica UBR os casos de uso são utilizados como guias para os artefatos inspecionados (MENDES, 2006). A idéia é melhorar a eficiência e a efetividade ao direcionar os esforços da inspeção para os casos de uso mais importantes sob o ponto de vista do usuário. Assim, a inspeção procura encontrar defeitos que têm um maior impacto negativo na qualidade, sob o ponto de vista do usuário (BERLING; THELIN, 2004; THELIN, 2003; THELIN *et al.*, 2004b).

Em termos de contribuição para o negócio, a UBR é a melhor abordagem, já que tem o seu foco em cenários e casos de uso em uma ordem pré-definida (WINKLER; BIFFL, 2006). Os cenários são específicos para cada projeto, o que significa que os casos de uso poderiam ser utilizados apenas dentro do projeto nos quais foram desenvolvidos. Todavia, os casos de uso podem ser utilizados para inspeções de requisitos, projeto, código, além de apoiar a especificação de teste (MENDES, 2006).

c) PBR (*Perspective-Based Reading*)

A técnica PBR é aplicada em inspeção de documentos de requisitos escritos em linguagem natural. Ela provê um conjunto de instruções específicas para os três papéis envolvidos diretamente com o documento de requisitos: o testador, o projetista e o usuário (SILVA; TRAVASSOS, 2004).

Figura 2.5 - Expectativas diferentes de acordo com as perspectivas em uma inspeção de requisitos com a técnica PBR (Adaptado de SILVA; TRAVASSOS, 2004)

De acordo com a Figura 2.5, cada inspetor deve receber instruções de como inspecionar o documento de requisitos de acordo a sua perspectiva (SILVA; TRAVASSOS, 2004). O objetivo é encontrar defeitos com essas diferentes perspectivas, e assim reduzir a sobreposição de inspeções (BERLING; THELIN, 2004; MENDES, 2006). Segundo Travassos (2007) coletar medidas e realizar observações baseadas no uso da técnica possibilita o seu aprimoramento contínuo. Além disso, esse acompanhamento pode:

- Possibilitar a inclusão de novas perspectivas de interesse para a organização;
- Possibilitar a troca da combinação das perspectivas. Ex: Se é esperada a predominância de certo tipo de defeito, devem ser incluídos mais revisores utilizando a perspectiva relevante;
- Possibilitar a alteração da taxonomia de defeitos; e
- Possibilitar a alteração no nível de detalhamento.

Segundo Berling e Thelin (2004), a PBR tem sido usada em diferentes tipos de artefatos, como por exemplo, documentos de requisitos, partes funcionais de código, e documentos de projeto de software orientado a objetos. A técnica consiste de três componentes principais (BERLING; THELIN, 2004):

1) **Introdução -** Descreve o artefato a ser inspecionado e como o artefato é importante para a perspectiva em foco, enfatizando as partes de qualidade mais relevantes que a perspectiva requer.

2) **Instruções -** Descreve como os documentos devem ser usados, como devem ser inspecionados e como as informações devem ser extraídas dos artefatos. É durante a extração de informações do artefato que provavelmente serão encontrados os defeitos.

3) **Questões -** As questões de controle ajudam a lembrar os aspectos importantes da perspectiva e ajudam o inspetor a inspecionar o artefato de uma maneira estruturada.

A técnica também verifica, através de cenários, a qualidade das especificações de requisitos ao solicitar que cada revisor assuma a perspectiva de um usuário específico do documento (projetistas, responsáveis por testes e usuários finais). O cenário consiste em construir um modelo do documento a ser revisado a fim de aumentar o entendimento e responder questões sobre o modelo, focando na resolução de problemas de interesse da organização (MENDES, 2006).

A técnica PBR oferece vários benefícios como: focar as responsabilidades de cada participante e diminuir a sobreposição das atividades realizadas, possibilitando que o documento seja completamente verificado, sem haver análises duplicadas de informações e/ou falta de análise, garantindo assim uma varredura total do documento (MENDES, 2006). Isso vai de encontro a um problema apontado por (MILLER; YIN, 2004), que diz que o gargalo da inspeção em grupo é a tendência dos membros do grupo encontrarem sempre os mesmos defeitos. Além disso, segundo He e Carver (2006), a PBR é a técnica existente mais efetiva, sistemática, focada, orientada a objetivo, customizável e transferível por treinamento.

d) OORT (*Object-Oriented Reading Technique*)

A técnica OORT representa uma família de técnicas de leitura que fornecem um procedimento para revisões dos diferentes diagramas e documentos de projeto orientados a objeto. O processo de leitura utilizando OORT deve ser realizado em duas dimensões (MENDES, 2006):

1) **Leitura Horizontal** - diferentes diagramas de projeto são verificados para assegurar que estejam consistentes entre si.

2) **Leitura Vertical** - é necessária a validação entre a especificação dos requisitos e os diagramas de projeto, para assegurar que o projeto esteja correto em relação aos requisitos.

Além das técnicas citadas acima, existem outras técnicas de leitura. Alguns exemplos são: SBR (*Systematic Order-based Reading*), FBR (*Functionality-based Reading*), UCDR (*Use case-driven Reading*) (ABDELNABI *et al.*, 2004) e TBR (*traceability-based reading*) (THELIN *et al.*, 2004b). Assim sendo, existem várias técnicas de leitura atualmente, contudo, a seleção e aplicação apropriada das técnicas de leitura são fundamentais para se obter os melhores resultados na detecção de defeitos (WINKLER; THURNHER; BIFFL, 2007).

Assim como as técnicas de leitura são importantes para a atividade de inspeção, existem outras estratégias que podem direcionar a inspeção de forma mais eficaz e eficiente. Por exemplo: uma vez que não seja possível inspecionar todos os documentos, em vez de selecioná-los de maneira aleatória, deve-se usar uma técnica sistemática para selecionar quais documentos serão inspecionados, priorizando os documentos em que se espera encontrar mais defeitos (RUNESON; ANDREWS, 2003).

2.5 Inspeção de código

Segundo alguns autores, o principal propósito da inspeção é aumentar a qualidade, detectando os defeitos nos artefatos, como por exemplo, o código (ALMEIDA *et al.*, 2003; LANUBILE; MALLARDO, 2007; THELIN *et al.*, 2004b). Quando se trata de garantir a qualidade do código, existem duas técnicas: os testes e as inspeções. Ao se usar o teste, é faz necessário o código executável do software (DENGER; KOLB, 2006), ao passo que, quando se usa a inspeção, não é obrigatório que todo o código esteja escrito e executando (WINKLER; THURNHER; BIFFL, 2007), podendo ser realizada até mesmo com um código incompleto.

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

De toda forma, mesmo com todas as vantagens do uso das inspeções, os testes ainda se fazem necessários para analisar certos tipos de comportamentos dinâmicos. Assim, as inspeções procuram complementar os testes, os quais, segundo Kothari e outros (2004), aumentam bastante a confiabilidade das inspeções.

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 fácil de manter, etc.

Entretanto, encontrar defeitos no código não é uma tarefa trivial. Segundo Chan e Jiang e Karunasekera (CHAN; JIANG; KARUNASEKERA, 2005), pesquisas têm mostrado que inspeções de código em projetos orientados a objetos possuem problemas semelhantes a testes orientados a objetos. Como exemplos de problemas em códigos orientados a objetos, podem ser citados (CHAN; JIANG; KARUNASEKERA, 2005):

- Entrelaçamento - torna difícil inspecionar uma classe separadamente;
- Referências - uma classe que faz muitas referências a outras classes tende a provocar distorções na própria inspeção;
- Comportamento dinâmico do código orientado a objetos - coloca certo limite na inspeção de código, pois certas informações só serão determinadas no momento de execução; e
- Polimorfismo e a Associação dinâmica - fazem com que o sistema se comporte diferentemente quando executado e quando interpretado o código fonte de maneira estática.

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 apóia várias atividades de engenharia de software como manutenção, testes e inspeções. Como 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 os engenheiros gastem, aproximadamente, 90% do tempo na atividade de compreensão.

Segundo Vinz e Etzkorn (VINZ; ETZKORN, 2006), a compreensão de código refere-se a qualquer atividade que usa métodos estáticos ou dinâmicos para revelar suas propriedades. Assim, a compreensão de código envolve o processo de extrair propriedades de um código para se ter um melhor entendimento do que ele faz.

Na tentativa de facilitar a compreensão, procura-se cada vez mais, escrever códigos enxutos e encapsulados. Essa é a principal idéia por traz da orientação a objetos, a qual, segundo Rilling e Klemola (2003), tem sido amplamente adotada, tornando-se popular com C++ e, mais recentemente, com Java.

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 pobre, com métodos de programação desestruturados, e manutenções mal gerenciadas, contribuem bastante para uma pobre qualidade do código, o que irá afetar diretamente a sua compreensão.

Da mesma forma que para inspeções de outros artefatos de software, para realizar inspeções de código são utilizadas técnicas de leitura que auxiliam na atividade de compreensão. Uma dessas técnicas é a que foi usada neste trabalho – *Stepwise Abstraction* (SA) – apresentada na subseção seguinte.

2.5.1 Stepwise Abstraction

Existem estudos que apontam para a necessidade de uma compreensão sistemática do código, caso se deseje encontrar todos os defeitos (RUNESON; ANDREWS, 2003). Uma compreensão sistemática se faz necessária principalmente para os trechos de código de alta complexidade cognitiva.

Uma abordagem sistemática comum para entender o programa é analisar a sua estrutura e construir uma representação de alto nível para o mesmo. Muitas técnicas tentam padronizar abstrações derivadas de comentários de programas e nomes de variáveis. Outros métodos tentam entender o programa 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; MILLS; WITT, 1979), a qual pode ser considerada como uma técnica de compreensão sistemática do código, 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 fonte. A SA consiste em realizar leituras do código fonte, a partir das quais os revisores escrevem sua própria especificação para o programa.

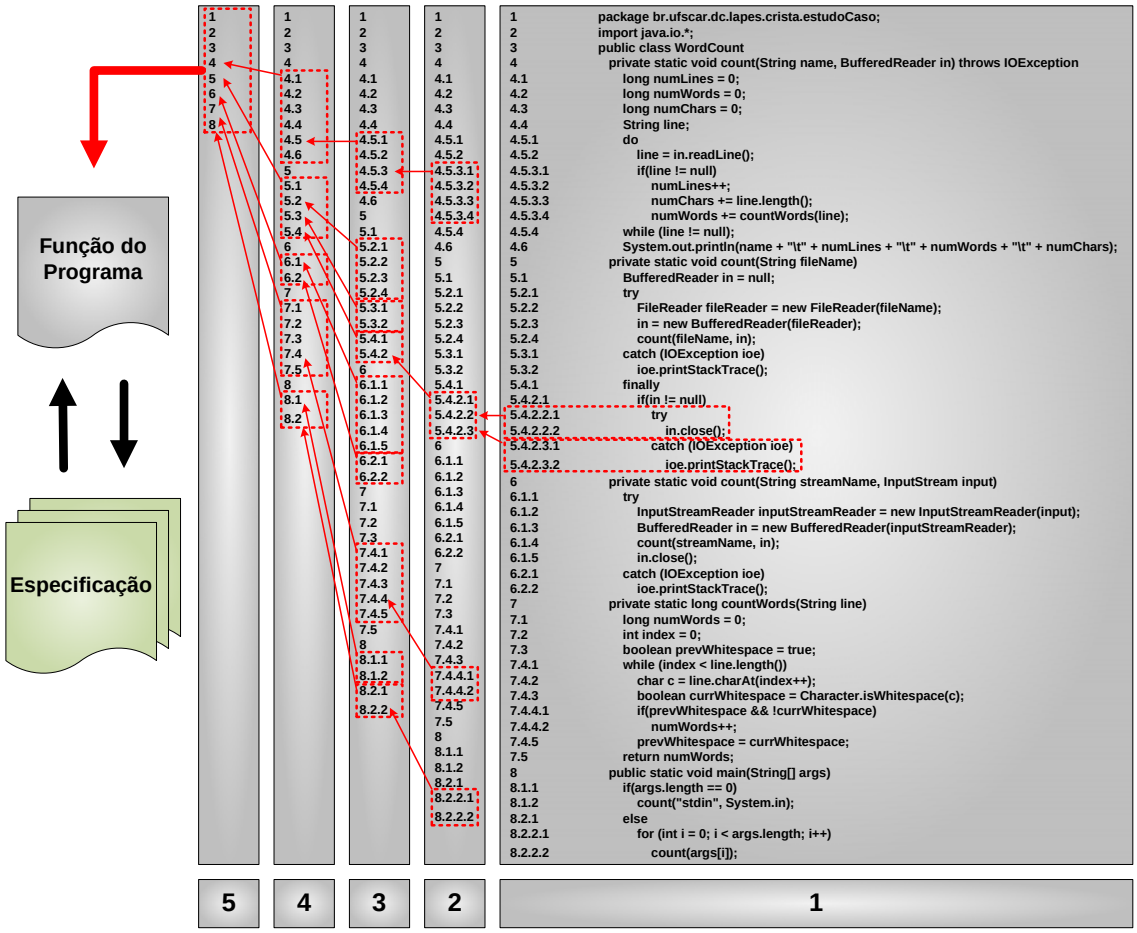


Figura 2.6 - Exemplo de aplicação da técnica *Stepwise Abstraction* (Adaptado de DÓRIA, 2001)

Como pode ser exemplificado pela Figura 2.6, o primeiro passo é identificar as instruções mais internas no código fonte (instruções 5.4.2.2.1, 5.4.2.2.2, 5.4.2.3.1, 5.4.2.3.2). Essas instruções são abstraídas pelo leitor, o qual escreve com suas palavras o que cada instrução no código representa.

No próximo passo, o leitor escolhe novamente as instruções mais internas que ainda não foram abstraídas (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, 8.2.2.2). Da mesma forma que no passo anterior, as instruções mais internas são abstraídas.

Ou seja, após serem feitas as abstrações para cada bloco, novas abstrações são feitas para blocos maiores, que englobam blocos já abstraídos, dessa forma, as abstrações são combinadas gerando uma abstração mais geral.

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

Em um processo de inspeção de código que use a técnica *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, o inspetor deve compará-la com a especificação original do mesmo, a fim de identificar se existem inconsistências entre ambas. Essas inconsistências são registradas como discrepâncias e são encaminhadas ao moderador para serem discutidas na reunião de inspeção. Esse processo pode ser observado na Figura 2.7.

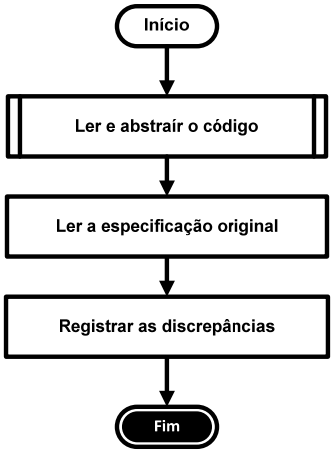


Figura 2.7 - Processo de inspeção de código com a técnica *Stepwise Abstraction*

Assim, embora a SA seja mais utilizada em atividades de inspeção elaborando uma descrição funcional do programa e comparando com o que se esperava que o programa fizesse, ela é uma forma sistemática para compreensão de código e pode ser utilizada em qualquer outra atividade que requeira o entendimento do código.

2.6 Ferramentas de Apoio à Inspeção

Segundo Lucia e outros (2007), de maneira geral, a inspeção de software é inerentemente manual, feita em papel e, além de gastar muito tempo, é uma atividade muito trabalhosa. Kothari e outros (2004) dizem que, além de consumirem tempo e serem caras, as inspeções manuais podem ser muito tediosas e tendenciosas a erros, especificamente quando a inspeção requer uma análise completa do software. Ainda assim, a inspeção formal é considerada uma importante atividade de garantia de qualidade de software, possuindo uma boa relação entre custo e eficiência para se encontrar defeitos nos artefatos de software (BREEN, 2006).

O sucesso da inspeção depende de um procedimento sistemático para sua condução (PARNAS; LAWFORD, 2003). Como conseqüência, surge a necessidade de um apoio computacional para aumentar a eficiência do processo de inspeção (LUCIA *et al.*, 2007). Segundo Halling, Biffl e Grünbacher (2003), as ferramentas que apóiam as inspeções são planejadas para acelerar as tarefas tediosas ajudando os inspetores a concentrar esforços em tarefas que agreguem um alto valor ao produto.

O uso de ferramentas no processo de inspeção pode influenciar significativamente o custo e o desempenho dessa atividade. Vários autores comentam que essas ferramentas podem apoiar o tratamento de uma grande variedade de documentos, auxiliar na coleta de dados, além de possibilitar a colaboração de diferentes envolvidos, tanto na preparação individual, quanto nas reuniões de inspeção (GRÜNBACHER; HALLING; BIFFL, 2003; LUCIA *et al.*, 2007; VITHARANA; RAMAMURTHY, 2003).

Um dos pontos mais controversos relacionados com o processo de inspeção se refere à fase de reunião e seu sincronismo (LUCIA *et al.*, 2007). O foco do desenvolvimento atual de ferramentas de inspeção está em apoiar o trabalho distribuído, mudando a natureza do processo de inspeção para outra totalmente independente de tempo e lugar, provendo meios necessários para inspecionar documentos e participar de discussões de uma maneira distribuída e assíncrona (HEDBERG; LAPPALAINEN, 2005).

Conforme pode ser visto na Figura 2.8, tendo como base os tipos de reuniões do processo de inspeção, as ferramentas evoluíram das Inspeções Síncronas para as Inspeções Distribuídas, que por sua vez evoluíram para Inspeções assíncronas.

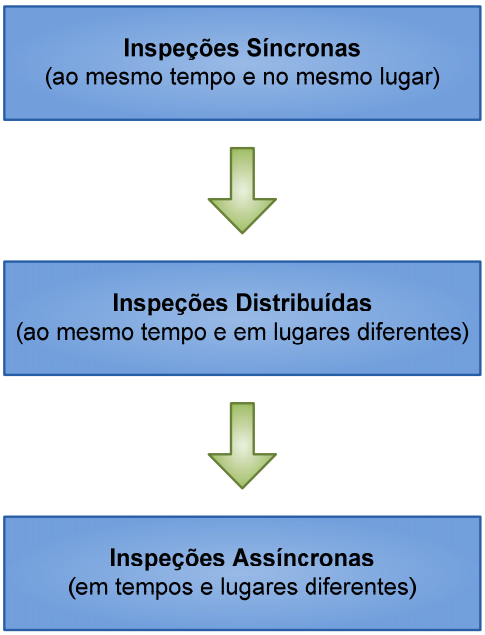


Figura 2.8 - Evolução das ferramentas para apoiar reuniões (Adaptado de HEDBERG; LAPPALAINEN, 2005)

Mendes (2006) descreve claramente os tipos de inspeções, baseadas nas reuniões de inspeção do ponto de vista da independência de tempo e lugar:

- Inspeções Síncronas são aquelas realizadas em um determinado lugar e todos os envolvidos devem estar presentes em determinada data;
- Inspeções Distribuídas são aquelas em que os participantes podem estar geograficamente distribuídos e a reunião se realiza em determinada data, por meio de uma videoconferência, por exemplo; e
- Inspeções Assíncronas são aquelas em que os envolvidos não precisam se reunir num mesmo lugar para que o processo ocorra. Isto é possível com a utilização de ferramentas que armazenam as informações e que todos os envolvidos possam atualizar as informações e consultá-las a qualquer momento.

Conforme apresentado anteriormente nesse capítulo, o processo de inspeção original incluía reuniões presenciais (FAGAN, 1976) e, por isso, as primeiras ferramentas foram desenvolvidas para dar apoio a esse tipo de reunião. Todavia, o apoio para trabalho distribuído vem sendo desenvolvido nas ferramentas de inspeção, culminando com uma total liberdade de tempo e lugar (HEDBERG, 2004).

No trabalho de Hedberg (2004), as ferramentas de inspeção de software podem ser agrupadas em quatro gerações de ferramentas, baseadas na classificação de groupware e das transições tecnológicas. Essas gerações são:

- Primeiras Ferramentas (*Early tools*);
- Ferramentas Distribuídas (*Distributed tools*);
- Ferramentas Assíncronas (*Asynchronous tools*); e
- Ferramentas baseadas em WEB (*Web-based tools*).

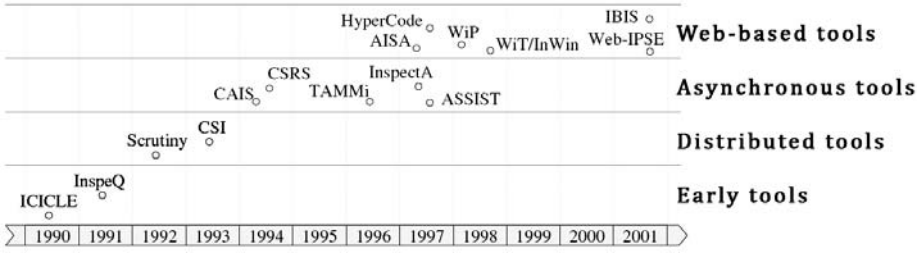


Figura 2.9 - Quatro gerações de ferramentas de apoio a inspeções (Adaptada de HEDBERG, 2004)

Ferramentas de apoio para o processo de inspeção evoluíram desde os anos 90 (HEDBERG; LAPPALAINEN, 2005). A Figura 2.9 apresenta algumas ferramentas na linha do tempo de acordo com as quatro gerações citadas acima.

Tendo em vista os benefícios agregados à utilização de apoio computacional para a área de inspeção de software, segundo Hedberg (2004), muitos esforços têm sido gastos desenvolvendo novas ferramentas para inspeção de software. Muitas dessas ferramentas apóiam particularmente inspeções de código, já existindo muitas que apóiam e até mesmo realizam análise estática do código para detectar possíveis defeitos. Cada uma dessas ferramentas procura por algum tipo de vulnerabilidade de segurança, erros de código, práticas de programação pobres e violação de estilos (SPACCO; HOVEMEYER; PUGH, 2006).

Existem ferramentas que apóiam vários tipos de artefatos, ou até mesmo o processo como um todo. Entretanto, como o foco deste trabalho encontra-se na inspeção de código, são apresentadas a seguir algumas das principais ferramentas existentes para inspeção de código.

Codestriker: É uma aplicação web de código livre que apóia revisões de código *on line*. Ela possui integração com sistemas de controle de versão como CVS e Subversion. A ferramenta permite ao inspetor ver o código e, ao identificar defeitos no software, inserir comentários a respeito dos mesmos além de acompanhar o progresso da correção dos defeitos durante o processo de inspeção. Assim que um comentário é feito, o autor do código recebe a notificação por email.

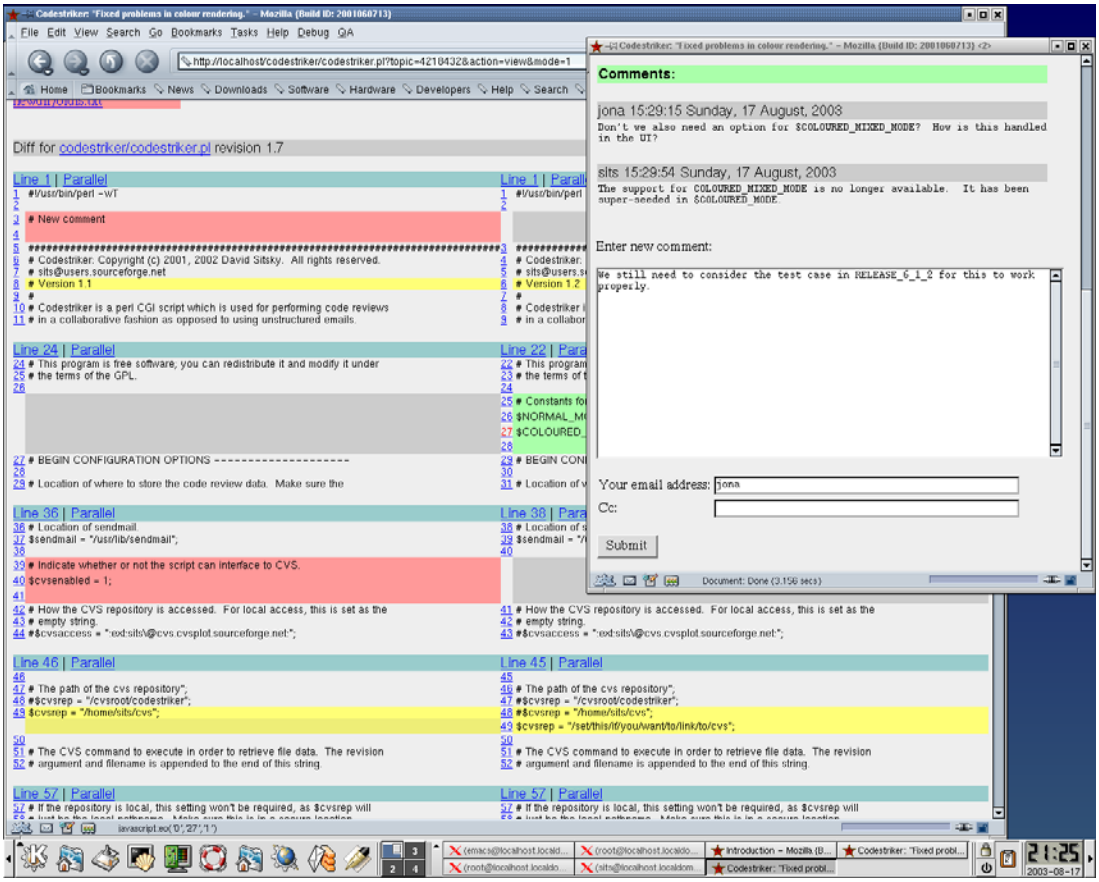


Figura 2.10 - Tela da ferramenta Codestriker (CODESTRIKER, 2008)

A Codestriker permite ao autor rejeitar ou aceitar o defeito identificado. A Figura 2.10 apresenta uma tela da ferramenta Codestriker, na qual pode ser visto o código ao fundo e a janela de inclusão de um novo comentário à direita. É possível ver também um histórico com os comentários anteriores.

A Codestriker ferramenta, além de facilitar a comunicação entre os membros da equipe por meio de emails (VITHARANA; RAMAMURTHY, 2003), possibilita que a quantidade de papel seja diminuída, uma vez que os comentários e decisões são armazenados em uma base de dados. Segundo os autores, a ferramenta oferece um excelente ambiente para realizar inspeções de código (CODESTRIKER, 2008). Codestriker foi escrito em PERL, é executado como um script CGI e usa o banco de dados MySQL.

Jlint: É uma ferramenta de inspeção automática de código Java. Apesar de não possuir interface gráfica, seu aprendizado é muito fácil. Jlint Le códigos escritos em Java e procura por defeitos comuns, como problemas de: inconsistências, fluxos de dados e sincronização. Além de sua execução ser muito rápida, Jlint gasta pouco esforço, uma vez que os defeitos são encontrados de forma automática (JLINT, 2008).

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 2.11 - Exemplo de uso da ferramenta Jlint (JLINT, 2008)

A Figura 2.11 apresenta um programa e a respectiva saída da ferramenta Jlint. É possível observar que o código apresenta problemas de sincronização, onde dois métodos tentam prender dois recursos de maneira alternada.

FindBugs: É outra ferramenta de análise estática de código que procura por defeitos em programas Java baseada em padrões de defeitos conhecidos (FINDBUGS, 2008). Alguns exemplos de padrões de defeitos detectados pelo FindBugs incluem (COLE *et al.*, 2006):

- Estruturas de repetição recursivas infinitas. Métodos que invocam a si mesmos novamente em *loops* recursivos terminando em um *Stack Overflow Error*;

- Instruções ou ramificações que, se executadas, irão provocar um *Null Pointer Exception*;
- Verificação de *casts* que lançam *Class Cast Exception*;
- Valor de retorno ignorado quando não pode ser ignorado; e
- Atributos públicos e estáticos que podem ser modificados por um código não confiável.

Tanto o FindBugs quanto o Jlint são ferramentas livres e inspecionam o bytecode Java na tentativa de encontrar padrões de defeitos. Ambas as ferramentas têm a capacidade de relatar vários tipos de padrões de defeitos; FindBugs pode detectar defeitos relacionados à interface enquanto Jlint pode detectar *deadlocks* (WOJCICKI; STROOPER, 2006). A Figura 2.12 mostra uma tela da ferramenta FindBugs. Na parte esquerda superior é mostrada a árvore de possíveis defeitos dentro de cada pacote. Quando um possível defeito é selecionado, seu código fonte é mostrado à direita, e sua descrição é mostrada na parte de baixo. A ferramenta FindBugs permite ainda ao inspetor escrever comentários a respeito do possível defeito.

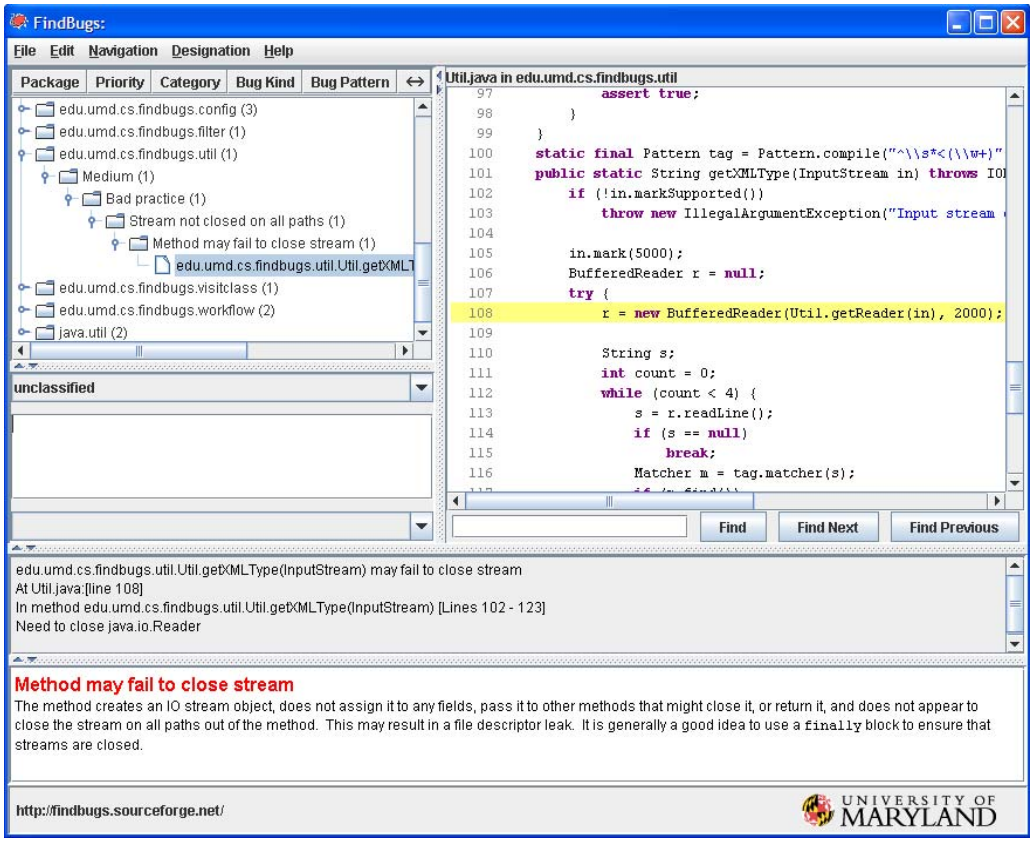


Figura 2.12 - Tela da ferramenta FindBugs (FINDBUGS, 2008)

Coffee Grinder: É uma ferramenta para apoiar inspeções de software orientado a objetos. Coffee Grinder provê uma infra-estrutura de *scripts* e uma interface de controle e

grafos de dependência. Os *scripts* são usados para encontrar defeitos ou potencias áreas para inspeção. Os grafos de dependência permitem escrever os *scripts* e, de uma forma interativa, atualizá-los para cobrir um conjunto de itens de um *checklist* (COOPER *et al.*, 2006). Um grafo de dependência da ferramenta Coffee Grinder pode ser visto na Figura 2.13.

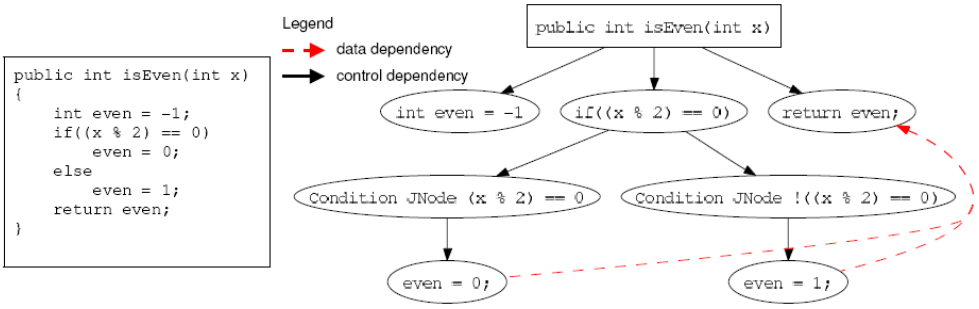


Figura 2.13 - Exemplo de grafo de dependência de uma declaração condicional (COOPER *et al.*, 2006)

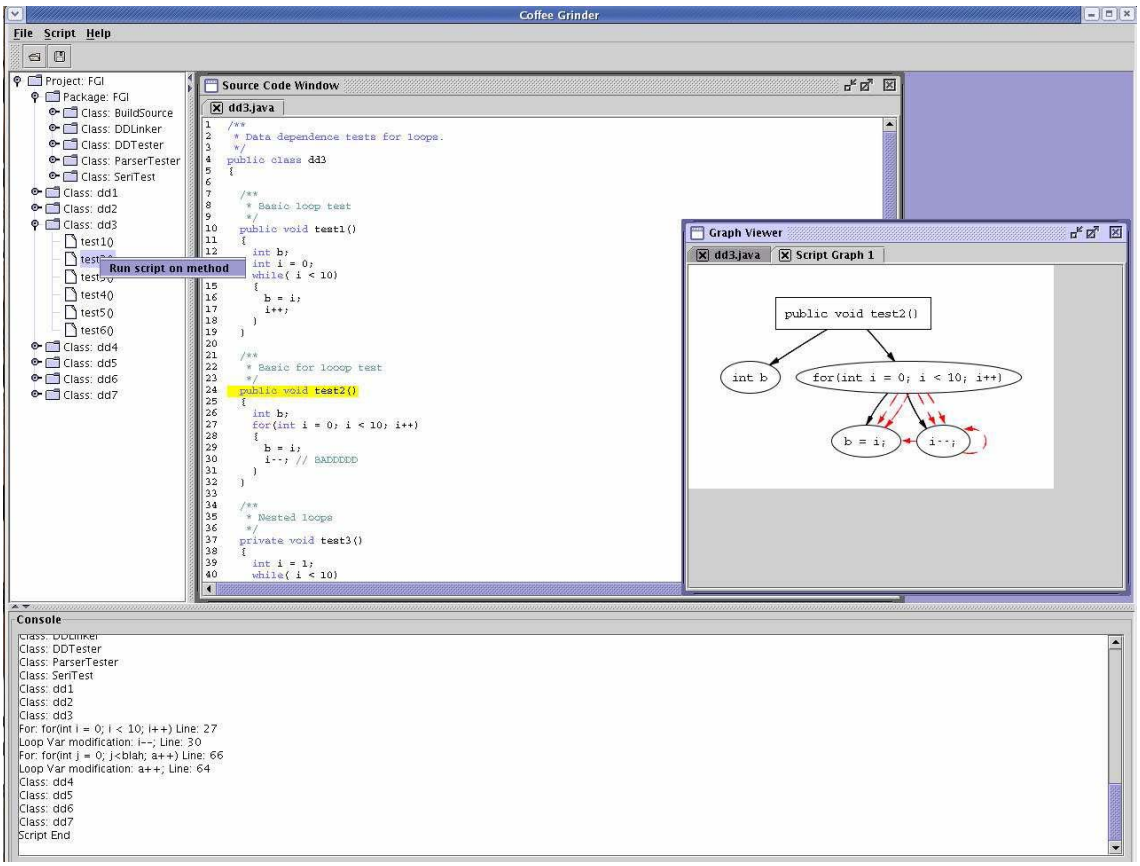


Figura 2.14 - Interface da ferramenta Coffee Grinder (COOPER *et al.*, 2006)

Na interface da ferramenta Coffee Grinder (Figura 2.14). É possível observar na figura que para a classe selecionada e para o método selecionado, é mostrado o código,

além de sua representação em grafo. Com a visualização em grafo do código, é possível identificar com mais facilidade, as dependências de cada variável dentro do método.

CIT: É uma ferramenta designada para apoiar inspeção de código. Ela permite que o Moderador crie um projeto, e a partir dele, cadastre os inspetores para formar a equipe de inspeção. Dentre as principais características da CIT estão (CHAN; JIANG; KARUNASEKERA, 2005):

- Apoio navegacional;
- Checagem dinâmica de código usando diagramas UML de projeto;
- Tratamento de documentos;
- Checagem estática; e
- Coleta de métricas;

Além disso, a ferramenta CIT permite ao inspetor marcar os trechos de código como possíveis defeitos. Essa marcação pode ser feita ainda com uma escala de prioridade.

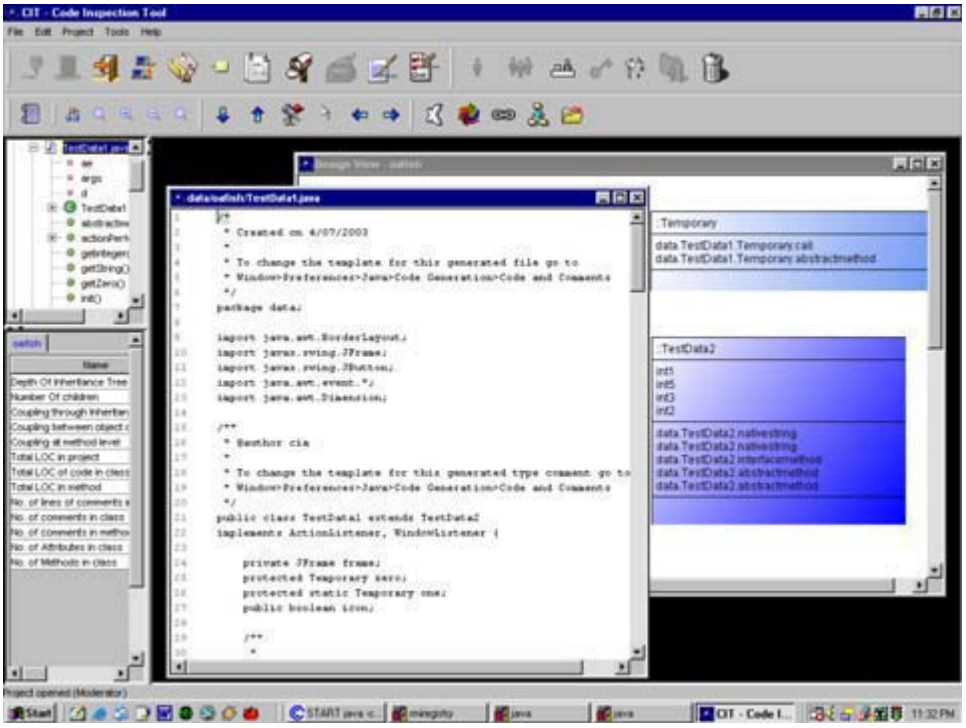


Figura 2.15 - Navegadores de código e diagrama de classes da ferramenta CIT (CHAN; JIANG; KARUNASEKERA, 2005)

A Figura 2.15 apresenta uma tela da ferramenta CIT na qual pode ser observado o navegador hierárquico do código (à esquerda), o navegador do código (ao centro) e o diagrama de classes (à direita). Esses navegadores provêem um apoio navegacional padronizado de destaque do código e referências cruzadas. Ao prover essas informações, a CIT almeja apoiar o trabalho dos inspetores, tornando a inspeção mais eficaz.

2.7 Considerações Finais

Este capítulo apresentou uma revisão da literatura sobre inspeção de software, a qual foi conduzida por meio de uma revisão sistemática (KITCHENHAM, 2004). Assim, neste capítulo foram apresentados alguns conceitos sobre essa atividade, seu processo e as técnicas de leitura que a apóiam. Segundo Kalinowski (2004), é muito importante que a inspeção retenha o conhecimento obtido durante sua aplicação para que este conhecimento seja utilizado de forma adequada. Isso só é possível aplicando o processo de inspeção de forma sistemática, da maneira como foi descrita na Seção 2.4.

No intuito de otimizar o processo de inspeção, fala-se bastante de técnicas especializadas e ferramentas, as quais buscam reduzir a sobrecarga administrativa e aumentar a eficácia e eficiência do processo de inspeção (BIFFL; GRÜNBACHER; HALLING, 2006). Segundo Kothari e outros (2004), muitas empresas ainda têm adotado as inspeções manuais pela falta de ferramentas que atendam a um domínio especifico.

Nem toda inspeção ocorre com sucesso. Segundo Kollanus e Koskinen (2006), o principal problema no processo de inspeção é a falta de treinamento adequado, o que acaba limitando a formalidade da inspeção e levando a métricas de inspeção imaturas. Além disso, muitas empresas simplesmente param de realizar as inspeções depois de certo tempo. Denger e Shull (2007) apontam algumas razões para o abandono das inspeções:

- É difícil traçar um paralelo entre o esforço da inspeção e a qualidade do produto final, como satisfação, segurança e reusabilidade;
- Inspeções não são suficientemente bem conduzidas para o contexto dado. Os projetos tentam aplicar o processo importado de outro projeto, sem considerar a capacidade da equipe, as questões específicas que são importantes para a equipe, ou o esforço disponível; e
- O real investimento parece ser muito amplo, pois não existem conexões diretas entre as inspeções e as atividades usuais dos envolvidos.

Mesmo com os problemas apresentados acima, segundo Anderson e outros (2003), as inspeções de software se destacam como uma prática muito efetiva da engenharia de software.

A inspeção de código, em particular, é muito dependente da atividade de compreensão e assim, ao se melhorar a compreensão do código consegue-se também fazer com que a inspeção seja mais efetiva. Pesquisas têm sido conduzidas nessa área e a visualização tem se mostrado como um recurso promissor para ajudar na compreensão de código. Como na ferramenta apresentada neste trabalho utiliza uma metáfora visual para

apoiar a aplicação da técnica *Stepwise Abstraction*, no próximo capítulo explora-se a visualização de informações, com uma breve revisão sobre o assunto.

Capítulo 3

VISUALIZAÇÃO DE INFORMAÇÕES

Este capítulo descreve como o uso de visualizações pode auxiliar no processo de compreensão. É apresentado também o processo de visualização, bem como as técnicas de visualização mais relevantes para este trabalho e seus respectivos exemplos. Por fim, é descrito o uso da visualização no contexto de software, seguido por exemplos de ferramentas de visualizações de software.

3.1 Considerações Iniciais

Conforme dito anteriormente, a compreensão de software tem papel estratégico na engenharia de software. Ela consome tempo, esforço e representa um custo considerável tanto nas atividades de desenvolvimento como nas de manutenção. Existem vários modelos propostos na literatura para compreensão (MARQUES; MENDONÇA; SANTOS, 2004), mas, independentemente do modelo considerado, a compreensão está sempre relacionada à transferência e construção de conhecimento a respeito do software, a partir dos artefatos disponíveis para análise.

Segundo Tergan e Keller (2005), o processo cognitivo de seres humanos é mais intuitivo, efetivo e eficiente quando apoiado por recursos visuais tais como imagens, gráficos e sinais. Isso porque os gráficos comunicam o conhecimento visualmente ao invés de verbalmente e, quando bem projetados, podem transmitir uma grande quantidade de informações mais complexas. Como diz o ditado “uma imagem vale por mil palavras”, um gráfico mostra ao invés de dizer (TIDWELL, 2005).

Almejando a construção de representações visuais de dados, a área emergente de Visualização de Informações se preocupa em facilitar o entendimento e/ou ajudar na descoberta de novas informações (NASCIMENTO; FERREIRA, 2005). De acordo com Keim

(KEIM, 2002) e Rhyne (RHYNE, 2000), a visualização pode ser usada para apresentação de informações e análise exploratória, além de apoio à decisão.

É sabido que a capacidade humana de lidar com informações de forma visual é muito maior do que com dados textuais. Isso se torna muito mais evidente em uma situação em que se tenha uma grande quantidade de informação (IEPSEN; LUZZARDI; LOH, 2007). Nesses casos, podem-se utilizar técnicas de visualização de informações para contornar as dificuldades de selecionar as informações relevantes de um grande conjunto de dados. Por meio dessas técnicas pode-se obter uma representação visual que, por um lado abstrai detalhes do conjunto de informações e, por outro, propicia uma organização desse conjunto segundo algum critério (FREITAS *et al.*, 2001).

O objetivo deste capítulo é apresentar, ainda que resumidamente, um pouco do tema Visualização, uma vez que a técnica de visualização *Treemap* foi usada na ferramenta CRISTA para mostrar a estrutura do código fonte a ser inspecionado. Este capítulo foi escrito tendo como base o trabalho de Nascimento e Ferreira (2005).

O restante deste capítulo está organizado da seguinte forma: na Seção 3.2 é apresentado o processo de visualização de informações. Na Seção 3.3, apresentam-se algumas técnicas de visualização. Em seguida, na Seção 3.4 são traçadas considerações sobre o uso de visualização no contexto de engenharia de software bem como ferramentas de apoio, e por fim, a Seção 3.5 apresenta as considerações finais deste capítulo.

3.2 Processo de Visualização

Segundo vários autores, (IEPSEN; LUZZARDI; LOH, 2007; NASCIMENTO; FERREIRA, 2005), o processo de visualização está relacionado com a transformação de algo abstrato em imagens (mentais ou reais) que possam ser visualizadas pelos seres humanos. O objetivo final é auxiliar no entendimento de determinado assunto, o qual, sem uma visualização, exigiria maior esforço para ser compreendido.

De modo simplificado, o processo de obtenção de novas informações por meio da visualização começa pelos dados abstratos (dados brutos) que, a partir de uma ferramenta computacional, são processados e organizados em uma representação lógica mais estruturada, para ser posteriormente criada uma representação visual, a qual será apresentada ao usuário na forma de figura. Ao analisar essa figura que representa os dados brutos, o usuário pode então tirar as informações desejadas para executar alguma tarefa. A Figura 3.1 mostra esse processo mais detalhado.

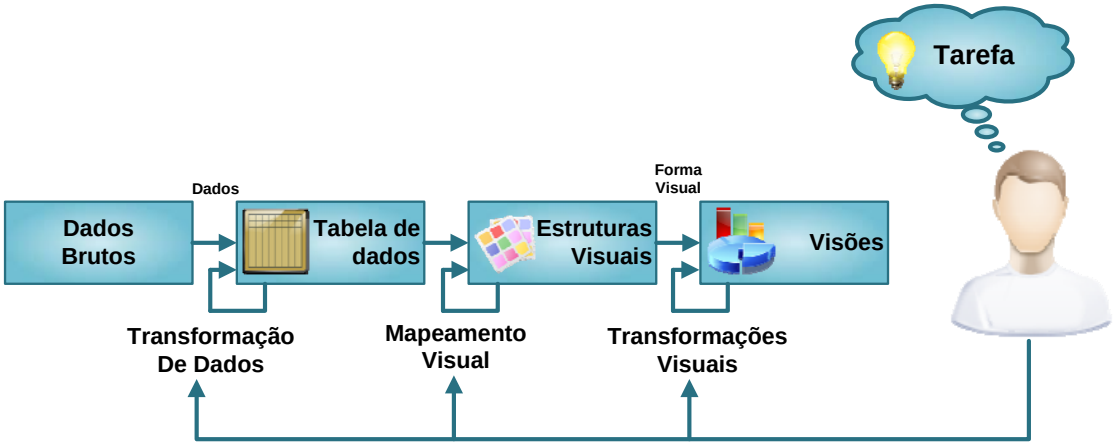


Figura 3.1 - Modelo de Referência para Visualização (Adaptada de CARD et al, 1999 apud NASCIMENTO; FERREIRA, 2005)

Na etapa de Transformações dos Dados, um conjunto de dados brutos é processado e organizado na forma de uma ou mais tabelas. Nesse processamento pode haver a eliminação de dados redundantes, errados ou incompletos, bem como a filtragem e o agrupamento de dados relevantes. Além disso, resultados de análises estatísticas (média, soma total, desvio padrão, etc.) podem ser incluídos.

Na etapa seguinte, que é o Mapeamento Visual, é definida uma estrutura visual que represente os dados da tabela. Dessa forma, esse mapeamento consiste em associar os atributos dos dados a marcas visuais (símbolos gráficos) para serem apresentados na tela.

A última etapa é a de Transformações Visuais, em que é permitido modificar e estender as estruturas visuais interativamente através de operações básicas como:

- Testes de localização, que possibilitam obter informações adicionais sobre um item da tabela de dados;
- Controles de ponto de vista, os quais permitem ampliar, reduzir e deslocar a imagem com o objetivo de oferecer visões diferentes; e
- Distorções da imagem, visando criar ampliações de uma região específica em detrimento de outra.

Segundo Iepsen, Luzzardi e Loh (2007), vale destacar que uma das principais características das ferramentas desenvolvidas para a visualização de informações, é que elas devem permitir a interação do usuário sobre as representações geradas. Assim, a ferramenta exibe apenas uma representação visual, a partir da qual o usuário deve ser capaz de gerar outras representações de forma a explorar a imensa capacidade de percepção visual humana.

Existem várias técnicas de visualização para as etapas de Mapeamento Visual e Transformações Visuais. Cada técnica procura representar e manipular os dados brutos de uma maneira graficamente diferente. Na seção seguinte são apresentadas algumas das principais técnicas de visualização existentes.

3.3 Técnicas de Visualização

Segundo Freitas e outros (FREITAS *et al.*, 2001), as técnicas de visualização de informações procuram representar os dados graficamente de modo que essa representação visual explore a capacidade de percepção humana. Assim, a partir das relações espaciais exibidas pela representação visual, o ser humano interpreta e compreende as informações apresentadas, possibilitando a produção de novos conhecimentos.

Existem várias técnicas de visualização disponíveis atualmente, cada qual mais apropriada a determinados tipos de dados e/ou tipos de explorações necessárias. A escolha da técnica de visualização mais apropriada para um determinado conjunto de dados depende de diversos fatores (IEPSEN; LUZZARDI; LOH, 2007), como por exemplo: o tamanho do conjunto de dados e a quantidade de informações que se deseja observar ao mesmo tempo.

Considerando que as técnicas de visualização visam, sobretudo, apresentar visualmente uma dada informação e permitir interação, elas podem ser consideradas essencialmente um processo de mapeamento. Esse mapeamento se dá entre as representações computacionais (tabela de dados) e as representações perceptíveis (estruturas visuais). Assim, esse mapeamento deve ser feito de forma a maximizar a comunicação e o entendimento humano (CEMIN, 2001). A seguir são apresentadas algumas das principais técnicas de visualização, bem como exemplos de suas aplicações.

3.3.1 Grafos

Uma das técnicas de visualização mais conhecidas para mostrar relações entre objetos, pessoas e estruturas hierárquicas são os grafos. Eles são modelos matemáticos formados por estruturas simples, que consistem de um conjunto de vértices (muitas vezes chamados de nós) e um conjunto de arestas. Nos grafos, os vértices geralmente representam objetos concretos ou abstratos, e as arestas indicam as relações entre esses objetos.

Figura 3.2 - Redes sociais desenhada por grafos (PREFUSE, 2008)

A técnica de visualização por grafos é uma técnica muito flexível, permitindo a sua utilização em um conjunto diverso de situações como: processos, redes sociais, mapas, etc. A Figura 3.2 apresenta um exemplo de utilização de grafos para representar uma rede social.

Outra área em que os grafos podem ser usados com freqüência é a computação, sendo um exemplo clássico, a representação de máquinas de estados finitos. Conforme pode ser observado na Figura 3.3, cada estado da máquina é representado por um vértice, e as relações que fazem a máquina passar de um estado para outro são representadas pelas arestas.

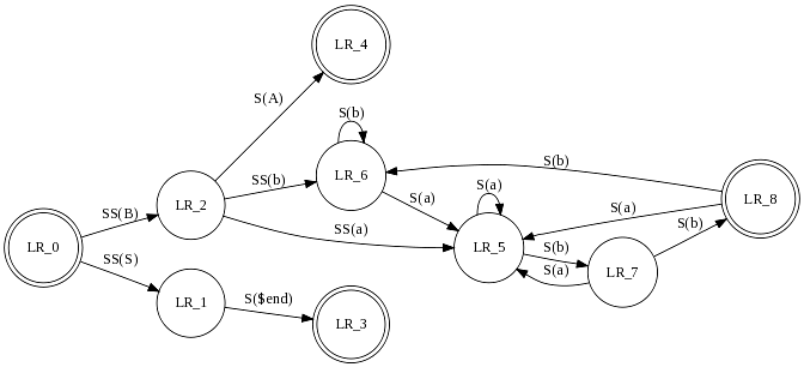


Figura 3.3 - Representação de máquina de estados finitos por grafos (GRAPHVIZ, 2008)

Na Engenharia de Software os grafos são amplamente utilizados para representar estruturas modulares de programas e a hierarquia de classes e de objetos, principalmente por meio de linguagens visuais como a *Unified Modeling Language* (UML) (NASCIMENTO; FERREIRA, 2005).

3.3.2 Fish-eye

A técnica de visualização *Fish-eye* é uma técnica muito interessante, pois permite uma visão mais detalhada de uma região de interesse sem haver perda de seus arredores. Isso se dá por meio de uma taxa maior de ampliação no centro da região de interesse e decrescente no sentido da periferia da imagem. Um exemplo dessa técnica de visualização é mostrado na Figura 3.4, em que o menu (a) é um menu normal que não apresenta nenhuma diferença entre o item selecionado e os demais. Já o menu (b), apresenta um efeito que aumenta o tamanho do item selecionado e de seus vizinhos com o passar do mouse, de forma que o item selecionado fique em destaque.

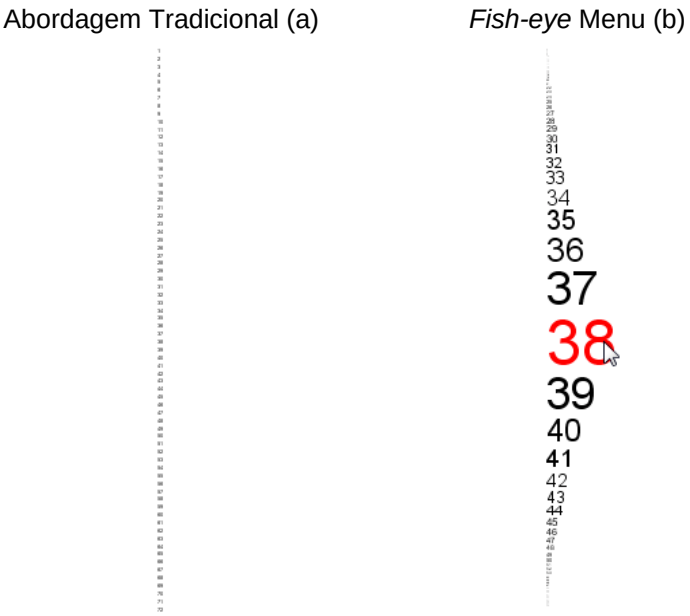


Figura 3.4 – Exemplo de menu tradicional e com a técnica *Fish-eye* (PREFUSE, 2008)

De maneira semelhante à figura acima, a *Dock bar* (ORDING; JOBS; LINDSAY, 2008) do sistema operacional Mac OS apresenta um menu conforme pode ser visto na Figura 3.5 (a). Ao passar o mouse sobre algum item do menu, ele fica em destaque, juntamente com os arredores, conforme pode ser visto na Figura 3.5 (b).

(a)

(b)

Figura 3.5 - *Dock bar* do sistema operacional Mac OS

3.3.3 Browser Hiperbólico

A técnica de *Browser* Hiperbólico é utilizada para a exploração de grandes árvores hierárquicas. Essas árvores são mapeadas em um plano hiperbólico, de forma que o centro da figura fica em destaque e na região periférica as informações são compactadas. Com essa representação, o *Browser* Hiperbólico consegue disponibilizar cerca de dez vezes mais vértices de uma árvore do que utilizando uma visualização no plano cartesiano.

A navegação em um *Browser* Hiperbólico é também mais efetiva que a do plano cartesiano, pois ao se mover o vértice selecionado para o centro da tela, ocorre uma compactação das informações que estão distantes do mesmo. Além disso, o sistema realiza transições gradativas e suaves da mudança do ponto de foco, de maneira a preservar o mapa mental construído para aquela estrutura.

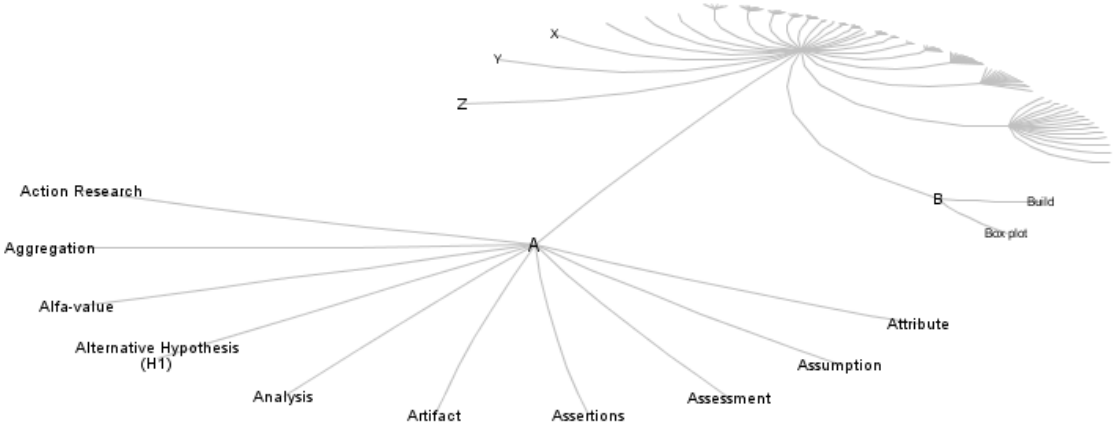


Figura 3.6 – Exemplo de menu Hiperbólico (ESE-COPPE, 2008)

A Figura 3.6 mostra o menu de um glossário de termos de engenharia de software experimental que usa a técnica (ESE-COPPE, 2008). Nesse glossário, é possível ir

caminhando pelas letras do alfabeto, a fim de se encontrar o termo procurado. Vale destacar a semelhança entre a técnicas *Browser* Hiperbólico e a *Fish-eye*, uma vez que nas duas,quando é dado um destaque maior para o que está selecionado, é tirada a atenção para o que não está selecionado.

3.3.4 Coordenadas Paralelas

A técnica de Coordenadas Paralelas mapeia um espaço n-dimensional em uma estrutura bidimensional que usa n eixos paralelos verticais eqüidistantes, denominados coordenadas. Conforme pode ser visto na Figura 3.7, essa técnica permite o mapeamento de uma grande quantidade de variáveis. Os eixos verticais (coordenadas) representam as dimensões ou atributos dos dados, são do mesmo tamanho, e os valores dos dados são dispostos no eixo de forma proporcional. Uma linha representativa de cada item de dado conecta os eixos nos seus respectivos valores, o que permite a observação de padrões.

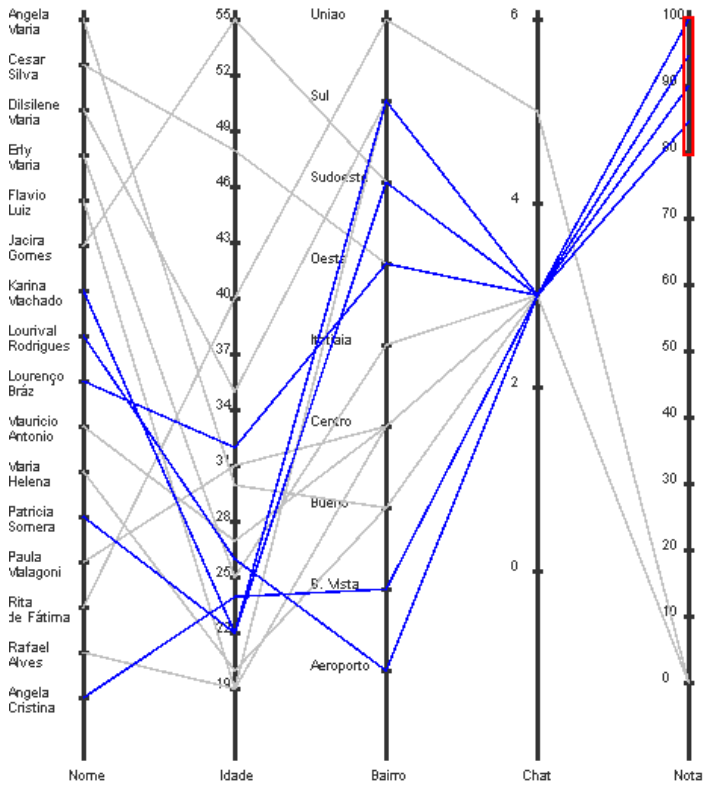


Figura 3.7 - Exemplo da técnica Coordenadas Paralelas (NASCIMENTO; FERREIRA, 2005)

Além de trabalhar com uma grande quantidade de dados, a técnica de coordenadas paralelas permite relacionar as informações entre si. A técnica enfatiza principalmente, as

relações entre eixos adjacentes e conjuntos de dados que possuem padrões similares (IEPSEN; LUZZARDI; LOH, 2007).

3.3.5 Glyphs

Assim como a técnica de Coordenadas Paralelas, a técnica *Glyphs* (também chamada de ícones) é utilizada para a visualização de dados multidimensionais. Os ícones são compostos por atributos geométricos, tais como forma, tamanho, orientação, posição ou direção, e atributos de aparência, como cor, textura e transparência.

Um exemplo da técnica de *Glyphs* pode ser visto na Figura 3.8. Nesse exemplo, vários dados dos alunos, como sexo, quantidade de acesso a um curso e nota foram mapeados em atributos visuais de um rosto. Nesse exemplo, o número de acessos ao site do curso foi mapeado pela quantidade de cabelo; a nota do aluno foi associada com a boca; e o sexo do aluno como a cor da face.

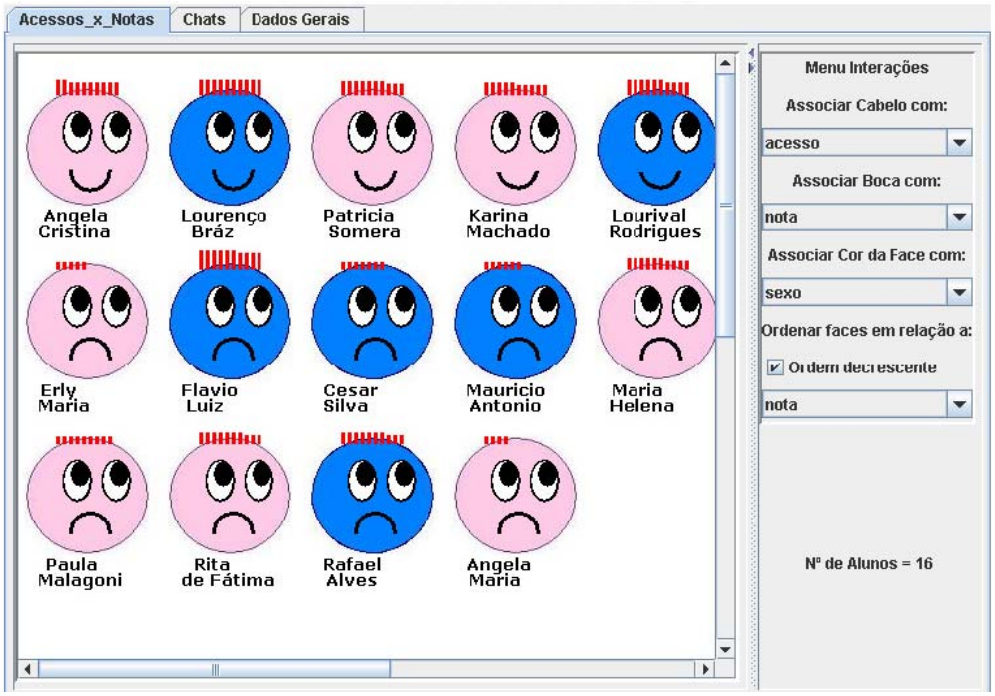


Figura 3.8 - Exemplo da técnica *Glyphs* (NASCIMENTO; FERREIRA, 2005)

Na Figura 3.8, os alunos foram ordenados pelas respectivas notas. Assim, através de uma simples análise da figura, é fácil perceber, por exemplo, que os alunos que obtiveram notas mais baixas (última linha) também acessaram pouco o site do curso (poucos cabelos).

3.3.6 Treemap

A *Treemap*, ou mapa em árvore, consiste em representar o nível mais alto da hierarquia como uma região retangular que preenche 100% do espaço disponível na tela. Os níveis mais baixos são desenhados recursivamente como retângulos dentro da região maior, sendo que o tamanho de cada retângulo é proporcional ao número de itens nos níveis imediatamente abaixo na hierarquia.

A Figura 3.9 mostra uma árvore e *Treemaps* associados a ela. O número de retângulos corresponde à numeração dos respectivos nós da árvore. O *Treemap* mostrado na Figura 3.9 (b) apresenta somente os nós folhas da árvore, enquanto na Figura 3.9 (c), todos os nós são aninhados, provendo mais informações acerca da hierarquia, contudo gastando um pouco mais de espaço visual para isso (PFEIFFER; GURD, 2006).

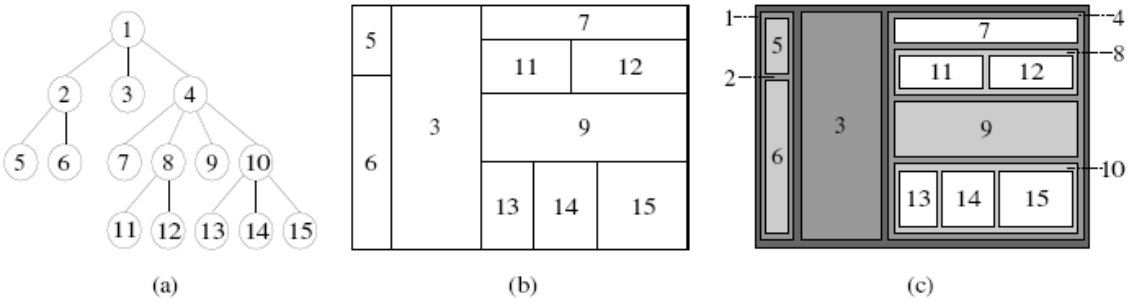


Figura 3.9 - Exemplo de uma estrutura em árvore (a), e possíveis visualizações com a técnica *Treemap* (b) e (c) (PFEIFFER; GURD, 2006)

Um exemplo interessante do uso de visualização *Treemap* é a visualização de notícias apresentadas no site Newsmap (MARUMUSHI, 2008). O Newsmap é uma aplicação que usa a *Treemap* para mostrar, de forma efetiva, uma enorme quantidade de notícias na tela. O Newsmap permite ao usuário filtrar as notícias apresentadas de acordo com o tema, data e país, conforme é apresentado na Figura 3.10.

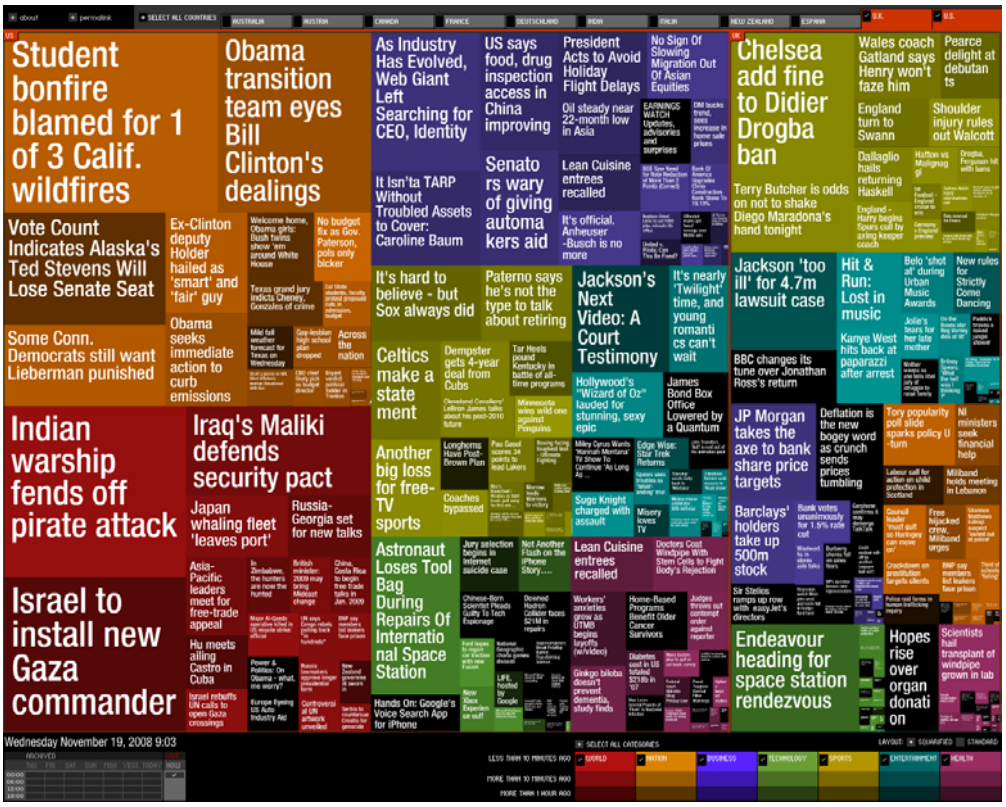


Figura 3.10 – Newsmap: exemplo de apresentação de notícias com a técnica *Treemap* (MARUMUSHI, 2008)

Assim como os grafos, a visualização *Treemap* tem muitos usos na Engenharia de Software através da compreensão de programas. Um exemplo prático dessa aplicação é através do programa SourceMiner (CARNEIRO; ORRICO; MENDONÇA, 2007), apresentado na Figura 3.11. No SourceMiner, um mapa em árvore apresenta como as classes de um software estão aninhadas em pacotes, assim como os métodos e variáveis estão aninhados nas classes. Além das informações acerca da estrutura hierárquica, no SourceMiner podem ser vistas também informações como:

- O tamanho de pacotes, classes e métodos, feitos pela associação entre o número de linhas de código e o tamanho dos retângulos; e
- A complexidade ciclomática por meio da associação de cores.

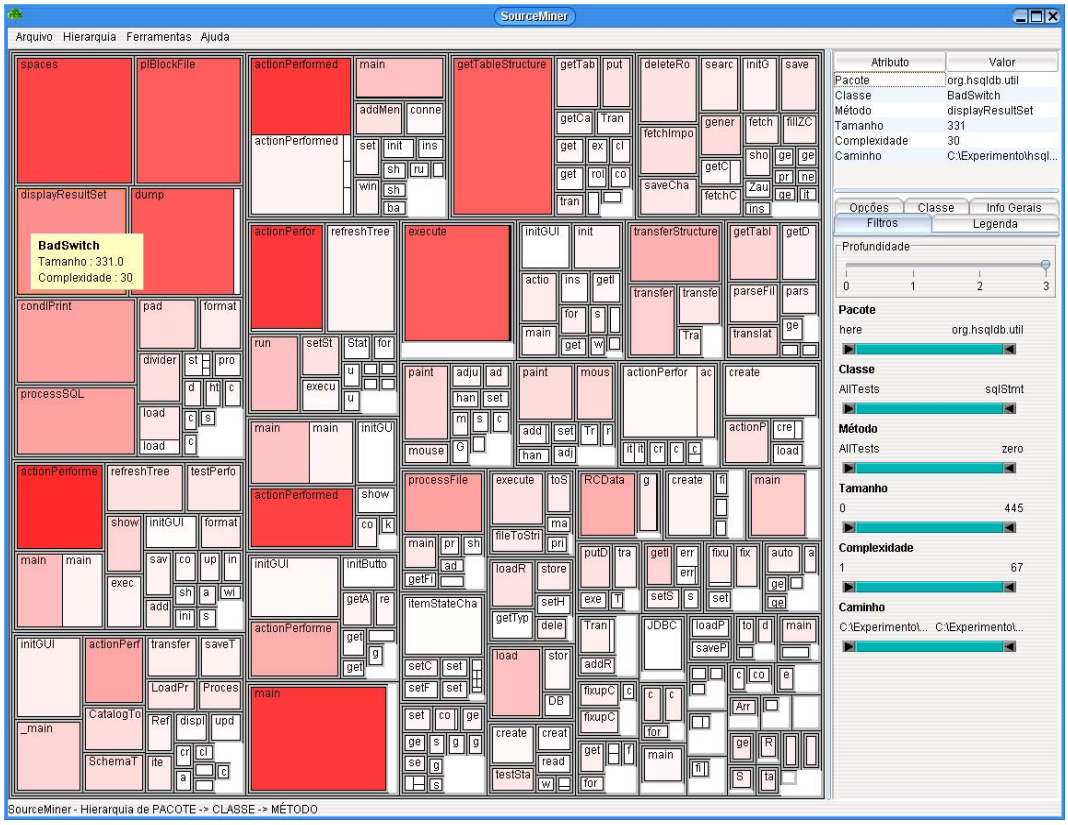


Figura 3.11 - Programa SourceMiner. Usado para atividades de compreensão de software (CARNEIRO; ORRICO; MENDONÇA, 2007)

Assim, conforme apresentado nesta seção, técnicas de visualização têm diversas aplicações em deferentes áreas da computação. Uma das principais é a visualização de software a qual é apresentada na seção seguinte.

3.4 Visualização de Software e ferramentas de apoio

Atualmente, os computadores se tornaram uma importante ferramenta para criar visualizações e auxiliar o usuário a compreender melhor os fenômenos complexos. Como conseqüência, a visualização tornou-se uma disciplina da computação (DIEHL, 2007).

Gershon (1994 *apud* DIEHL, 2007) define a visualização como sendo o processo de transformar a informação em um formato visual, permitindo que os usuários possam observar as informações. Assim, o resultado visual mostrado permite ao usuário perceber visualmente características que estão escondidas nos dados, mas que, são necessárias para a sua exploração e análise.

De acordo com Diehl (2007), a visualização é muito utilizada na indústria mecânica, química, física, e médica. Os cientistas da computação desenvolveram sistemas sofisticados para produzir visualizações para estas disciplinas. No entanto, pouco tem sido feito para o uso de visualização como ferramenta de apoio para a concepção, execução e manutenção de software. Conforme é ilustrado na charge da Figura 3.12, muitas vezes os programadores se adaptam às representações fornecida pelo computador, em vez de adaptar o computador para representar as mesmas informações de maneira mais perceptiva ao programador.

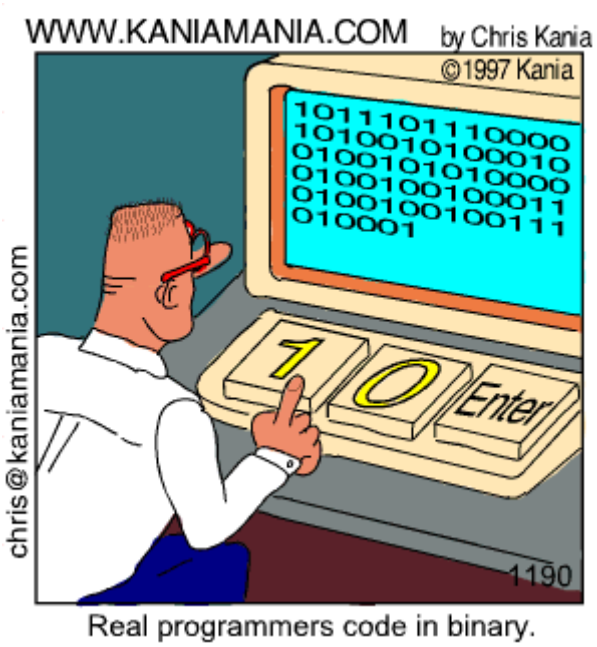


Figura 3.12 - Programadores sem uso de visualizações (DIEHL, 2007)

Mesmo com toda sua notação formal e obscura, a terminologia da ciência da computação é rica em metáforas, como por exemplo: fitas, árvores, folhas, filas, arquivos, pastas e documentos. O objetivo dessas metáforas é evocar imagens mentais para memorizar melhor os conceitos e explorar analogias para entender melhor as estruturas ou funções utilizadas na área (DIEHL, 2007).

Uma das áreas em que a compreensão de informações é muito importante é na compreensão de software. Entender como o software está escrito é uma atividade chave que apóia várias atividades de engenharia de software como manutenção, teste e inspeção (WALKINSHAW; ROPER; WOOD, 2005). A atividade de compreensão pode, muitas vezes, determinar o sucesso de qualquer uma dessas atividades.

Segundo Freitas e outros (2001), ao se combinar os aspectos de computação gráfica, interface homem-computador e mineração de dados, a visualização permite

apresentar os dados de forma gráfica, de modo que o usuário utilize sua percepção visual para melhor analisar e compreender as informações.

Essa necessidade por ferramentas e técnicas para tratamento de dados se torna cada vez mais urgente na medida em que se aumenta o volume de informações a serem compreendidas. Isso é o que ocorre com a maioria dos programas, que estão cada dia maiores e mais complexos. Assim, a visualização de software pode ser útil, por exemplo, auxiliando o programador a analisar e a entender a estrutura e o funcionamento de um programa em um nível maior de abstração do que quando comparada a uma simples leitura do código fonte (NASCIMENTO; FERREIRA, 2005).

Muitas ferramentas de visualização de software já foram desenvolvidas para ajudar os programadores a entenderem e desenvolvem programas (LINTERN *et al.*, 2003). Um exemplo dessas ferramentas é a Coffee Grinder, que apóia inspeções por meio de grafos de dependência, como já apresentado na Seção 2.6.1. A seguir são descritas algumas das principais ferramentas de existentes atualmente.

Team Tracks: Team Tracks é outra ferramenta de visualização que auxilia na compreensão de código. Uma vez que para grandes projetos de software, geralmente, o programador deve realizar mudanças em códigos desconhecidos, a ferramenta Team Tracks facilita a compreensão do sistema, mostrando os padrões navegacionais do código fonte para toda a equipe de desenvolvimento (DELINE; CZERWINSKI; ROBERTSON, 2005).

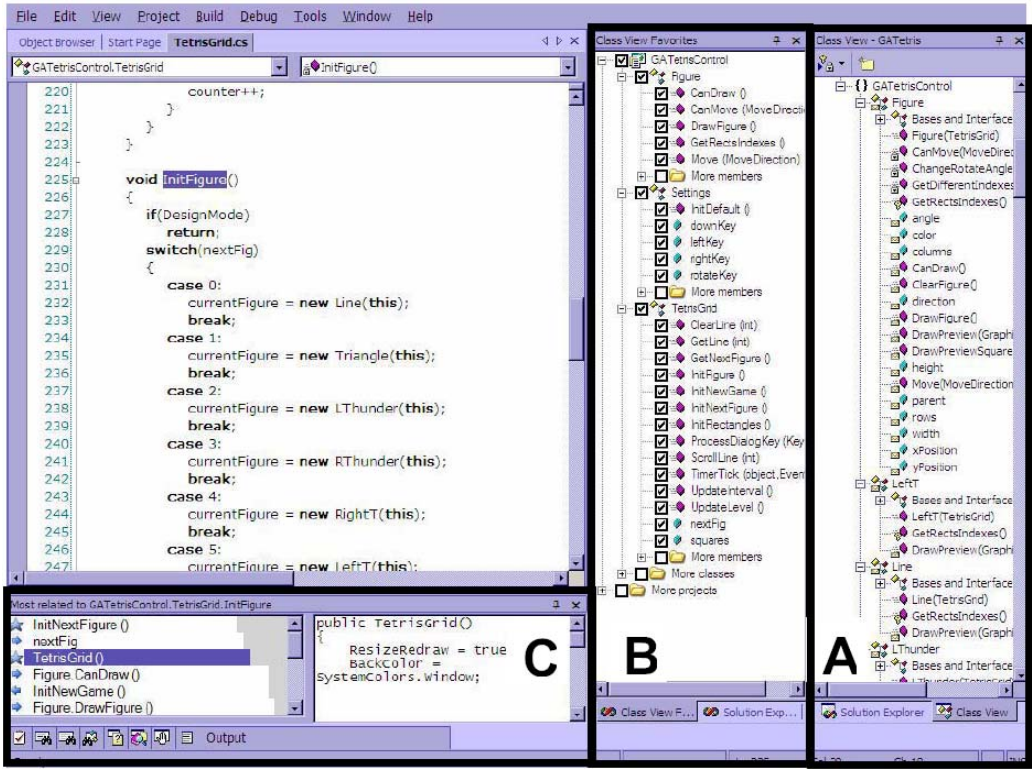


Figura 3.13 - Tela da ferramenta Team Tracks (DELINE; CZERWINSKI; ROBERTSON, 2005)

A Figura 3.13 mostra a tela da ferramenta Team Tracks, sendo que em (A) é apresentada a estrutura hierárquica das classes (muito comum em IDEs), em (B) a mesma visão hierárquica somente com as classes favoritas do usuário e em (C) a visão com os métodos e classes relacionadas com a classe selecionada.

SHriMP: De maneira geral, as ferramentas de visualização são aplicações separadas e dificilmente integradas com as ferramentas que os desenvolvedores usam. Dessa forma, é difícil avaliar a sua utilidade em um contexto do mundo real (LINTERN *et al.*, 2003). Com essa motivação, foi desenvolvida a ferramenta SHriMP.

SHriMP é uma ferramenta para apoiar a compreensão e entendimento de programas por meio de grafos que mostram a estrutura hierárquica de programas Java (Figura 3.14). Os nós folhas do grafo correspondem a entidades no software, como métodos e tipos de dados; já os arcos representam as dependências entre essas entidades, podendo mostrar também relacionamentos de herança, composição e associação. Além disso, o programador pode navegar no código fonte ou na documentação seguindo os links disponíveis na ferramenta (LINTERN *et al.*, 2003).

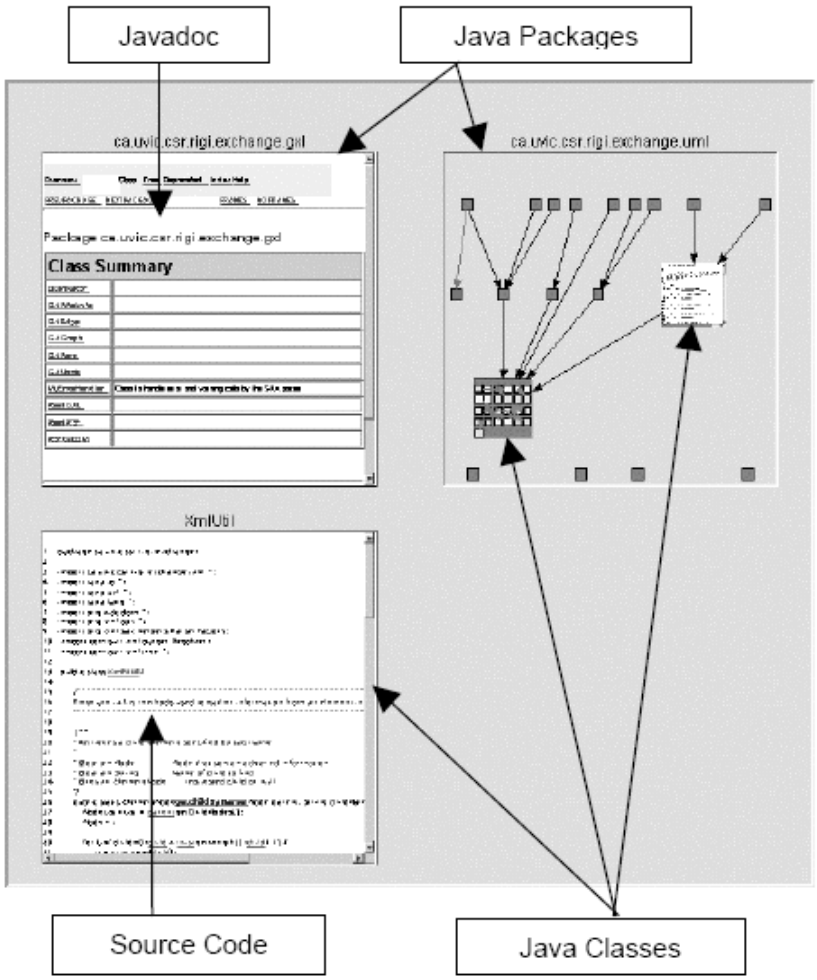


Figura 3.14 - Visão de um programa Java na ferramenta SHriMP (LINTERN *et al.*, 2003)

SHriMP foi originalmente construída como uma alternativa para a ferramenta de engenharia reversa Rigi, que é uma das primeiras ferramentas a apresentar os artefatos de software e seus relacionamentos usando grafos (LINTERN *et al.*, 2003). Com a integração da ferramenta SHriMP com a IDE Eclipse, foi também adicionado acesso a novas fontes de informação através dos *plugins* existentes na IDE. Na Figura 3.15 é apresentada a ferramenta SHriMP integrada ao Eclipse. Como pode ser observado, ela apresenta todas as informações sincronizadas: a visão hierárquica no painel superior esquerdo, a visão principal no painel inferior e o código fonte no painel superior direito (LINTERN *et al.*, 2003).

O objetivo de integrar a ferramenta de visualização SHriMP com a IDE Eclipse foi melhorar a navegação dos programadores pelos programas, apoiar o entendimento do programa e prover um apoio para a colaboração da equipe e o gerenciamento do projeto. Com a integração da ferramenta com a IDE, foi adicionada a técnica de visualização *Treemap* (que pode ser observada no centro na região inferior da Figura 3.15), para fazer um melhor uso do espaço da tela (LINTERN *et al.*, 2003).

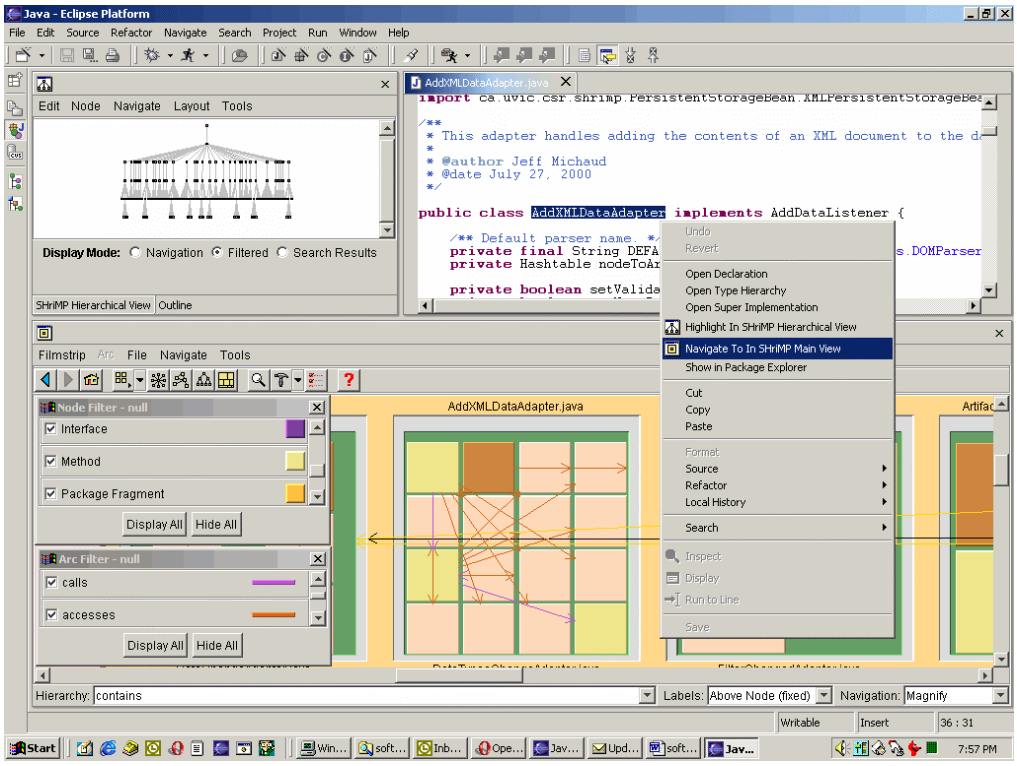


Figura 3.15 - Ferramenta SHriMP integrada com a IDE Eclipse (LINTERN *et al.*, 2003)

3.5 Considerações Finais

Como foi visto neste capítulo, pelo fato da visualização apresentar as informações graficamente, isso faz com que elas possam ser entendidas com maior facilidade, inclusive no contexto de compreensão de código, que é uma atividade que se faz presente e que é fundamental em vários momentos do desenvolvimento de software (WALKINSHAW; ROPER; WOOD, 2005). Exemplo disso são as atividades de inspeção e manutenção que necessitam compreender várias informações que estão espalhadas pelo código. Dessa forma, uma vez que a compreensão de código possui um papel tão importante nas atividades de engenharia de software, a visualização pode facilitar diversas tarefas que dependam da compreensão.

A maneira como essas informações são apresentadas é um ponto muito importante durante o processo de visualização. Por isso, conforme apresentado neste capítulo, existem várias técnicas de visualização. Cada técnica é mais adequada a certos tipos de informação. Por isso, a escolha correta da técnica de visualização é fundamental para que se possa obter uma contribuição efetiva da técnica. Também foram apresentadas neste capítulo várias ferramentas que auxiliam na compreensão de software, destacando-se a importante influência da visualização adotada nessas ferramentas.

Como dito anteriormente, para apoiar a técnica de leitura *Stepwise Abstraction*, que trabalha hierarquicamente com as estruturas que compõem o código fonte, foi escolhida a técnica de visualização *Treemap*. Essa escolha baseou-se nos fatos dessa técnica representar estruturas hierárquicas e de não desperdiçar o espaço disponível na tela. Além disso, como os retângulos representados na *Treemap* não possuem uma ordem seqüencial, isso contribui para o processo da técnica de leitura *Stepwise Abstraction*, uma vez que nessa técnica o código não deve ser lido seqüencialmente, mas sim das estruturas mais internas para as mais externas. Espera-se que essa técnica proporcione uma forma mais simples e intuitiva de entender melhor e mais rapidamente o significado das instruções presentes no código. Isso poderá facilitar a própria aplicação da técnica de leitura, além de tornar a atividade de inspeção ainda mais efetiva na detecção de defeitos. No próximo capítulo serão abordados os conceitos sobre Experimentação. Esse assunto é importante para este trabalho, pois toda proposta nova deve ser avaliada por meio de estudos experimentais que possam caracterizá-la. No caso da ferramenta CRISTA, foram conduzidos alguns estudos que serão apresentados no Capítulo 7.

Capítulo 4

EXPERIMENTAÇÃO

Este capítulo descreve como a experimentação pode ser útil no meio científico. São apresentados os tipos de experimentos, bem como o processo detalhado de experimentação. Por fim, é discutido o uso da experimentação no contexto da engenharia de software.

4.1 Considerações Iniciais

Existe um interesse crescente em estudos experimentais, que avaliem a atividade humana de modo sistemático, disciplinado e controlado. Os estudos experimentais servem tanto para validar tecnologias maduras, quanto para guiar melhorias em tecnologias mais recentes. Corandi e outros (2001) afirmam que a experimentação pode proporcionar uma base de conhecimento para reduzir incertezas sobre quais teorias, ferramentas e metodologias são adequadas, como também descobrir novas áreas de pesquisa ou conduzir as teorias para direções promissoras.

Segundo Basili e outros (1996a), novos métodos, técnicas, linguagens e ferramentas não deveriam ser apresentados sem passar antes por um processo de experimentação, a fim de ser validado. Ou seja, as novas invenções e sugestões devem ser comparadas com as já existentes de modo a comprovar suas reais contribuições.

Dentre as várias áreas existentes, uma em que os estudos experimentais tem se destacado é a engenharia de software, mostrando-se um meio efetivo para avaliar técnicas, bem como adquirir conhecimento (LOTT; ROMBACH, 1996). Segundo Travassos, Gurov e Amaral (2002), a experimentação no contexto específico da engenharia de software, tem como objetivo caracterizar, avaliar, prever, controlar ou melhorar tanto os produtos, como também os processos, recursos, modelos ou teorias.

Como este trabalho aborda a construção de um apoio computacional para o auxílio às inspeções de software com a aplicação da técnica de leitura *Stepwise Abstraction* e com o uso de técnica de visualização, a experimentação nos deu suporte para avaliar o uso da ferramenta posposta, bem como adquirir conhecimento a respeito da mesma.

O restante deste capítulo está organizado da seguinte forma: na Seção 4.2 são caracterizados os tipos de experimentos. Na Seção 4.3, é explorado o processo de experimentação, detalhando cada uma de suas etapas. São feitas algumas considerações relevantes sobre o empacotamento na Seção 4.4. Em seguida, a Seção 4.5, aborda a experimentação no contexto específico da Engenharia de Software. Por fim, a Seção 4.6 apresenta as considerações finais.

4.2 Tipos de Experimentos

Segundo Wohlin e outros (2000), existem diferentes estudos experimentais. A escolha do tipo mais apropriado depende, por exemplo, das técnicas, dos métodos, das ferramentas e das propriedades do processo de software usado durante a experimentação. Os três principais estudos experimentais são (WOHLIN *et al.*, 2000):

- *Survey*;
- Estudo de caso;
- Experimento.

O levantamento de dados ou ***survey*** é uma investigação executada em retrospectiva, por exemplo, quando uma ferramenta ou técnica já foi usada por um determinado período (WOHLIN *et al.*, 2000). No *survey* o pesquisador procura por relações entre as variáveis gerando os resultados que, são generalizados para a população da qual a amostra foi selecionada. Os objetivos gerais do *survey* são (WOHLIN *et al.*, 2000):

- Descritivo – determinar a distribuição de atributos ou características;
- Explanatório – explicar porque os desenvolvedores escolheram uma das técnicas; e
- Exploratório – um estudo preliminar para uma investigação mais profunda.

Segundo Wohlin e outros (2000), os meios principais para coletar a informação quantitativa e qualitativa preliminar são os questionários. Dessa forma, o *survey* possui capacidade de levantar um grande número de variáveis a serem avaliadas. Entretanto, ele

não oferece nenhum controle sobre a execução ou medição, além de ser sempre impossível manipular as variáveis.

O **estudo de caso** é utilizado para monitorar os projetos, atividades e atribuições (WOHLIN *et al.*, 2000). Segundo Travassos, Gurov e Amaral (2002), os estudos de caso objetivam observar um atributo específico (um caso particular) e, posteriormente, estabelecer relacionamentos entre atributos diferentes (princípios gerais).

O estudo de caso é caracterizado pelo baixo controle por parte do pesquisador, mas ao contrário de *survey*, o estudo de caso possui o controle sobre a medição das variáveis (TRAVASSOS; GUROV; AMARAL, 2002).

O **experimento** geralmente é realizado em laboratório e oferece o maior nível de controle (WOHLIN *et al.*, 2000). Um experimento é um evento planejado e fortemente controlado pelo pesquisador, além de existir um total controle sobre o processo e as variáveis envolvidas. Assim, uma ou algumas variáveis são manipuladas enquanto as outras são mantidas fixas e, ao final, é medido o resultado.

Segundo Travassos, Gurov e Amaral (2002), os experimentos são apropriados para confirmar as teorias, confirmar o conhecimento convencional, explorar os relacionamentos, avaliar a predição dos modelos, ou validar as medidas. O que o experimento apresenta de melhor é o controle total sobre o processo e as variáveis, além da possibilidade de ser repetido. Os experimentos podem ser feitos de dois tipos:

a) in-vitro - sob condições de laboratório; ou

b) in-vivo - sob condições normais

As principais características para diferenciar os tipos de estudos experimentais são o controle de execução, controle de medição, controle de investigação, facilidade de replicação e custo, conforme apresentado pela Tabela 4.1.

Tabela 4.1 - Características dos estudos experimentais (TRAVASSOS; GUROV; AMARAL, 2002)

Fator	*Survey*	Estudo de caso	Experimento
Controle de execução	Nenhum	Nenhum	Tem
Controle de medição	Nenhum	Tem	Tem
Controle de investigação	Baixo	Médio	Alto
Facilidade de replicação	Alta	Baixa	Alta
Custo	Baixo	Médio	Alto

4.3 Processo de Experimentação

A execução de um experimento presume a realização de um processo bem definido com diferentes atividades. Segundo Wohlin e outros (2000) a experimentação é caracterizada por um processo que tem um início a partir da etapa de definição do experimento, seguido pelas etapas de planejamento, operação, análise e interpretação, até a apresentação e empacotamento. A Figura 4.1 representa as fases do processo experimental que será detalhado a seguir.

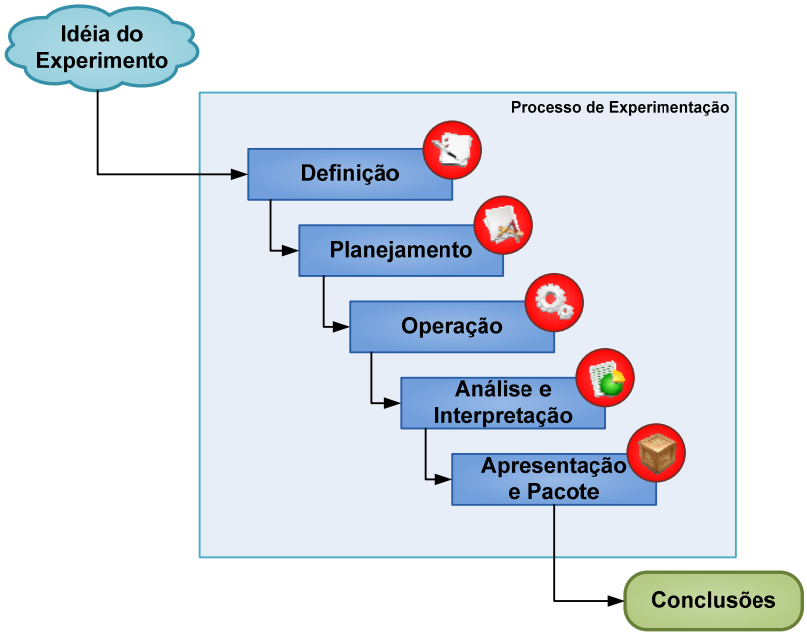


Figura 4.1 -O processo de Experimentação e suas fases (Adaptado de WOHLIN *et al.*, 2000)

4.3.1 Definição

A primeira etapa é a definição do experimento, e é nela que o experimento é definido em termos de problema, objetivo e idéias. Ou seja, nessa etapa é determinada a base do experimento. Se essa etapa não for realizada de maneira adequada, pode implicar em retrabalho ou o experimento pode até não ser usado para o estudo desejado (WOHLIN *et al.*, 2000).

Nessa fase é elaborado um esqueleto de definição de idéias com o propósito de assegurar que todos os aspectos importantes do experimento sejam definidos antes que o planejamento e a execução aconteçam. A seguir, são apresentadas a estrutura do esqueleto de idéias e a definição dos termos usados segundo Wohlin e outros (2000):

Análisar	**<objeto de estudo>** É a entidade que está sendo estudada no experimento. Por exemplo: produtos, processos, recursos, modelos, métricas ou teorias.
Com o propósito de	**<propósito>** Qual é a intenção do experimento. Por exemplo: avaliar duas técnicas ou caracterizar a curva de aprendizado de uma organização.
Com respeito a	**<foco de qualidade>** Indica o principal aspecto que está sendo estudado. Por exemplo: eficiência, custo, confiabilidade, etc.
Do ponto de vista da	**<perspectiva>** Exibe o ponto de vista no qual os resultados do experimento são interpretados. Por exemplo: perspectiva do desenvolvedor, cliente, gerente de projeto e pesquisador.
No contexto de	**<contexto>** É o ambiente no qual o experimento será realizado. Pode ser caracterizado como números de pessoas e artefatos envolvidos no estudo.

4.3.2 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):

- O ambiente no qual o experimento será executado;
- A formulação de hipóteses;
- A escolha das variáveis;
- O projeto experimental;
- A implementação do experimento; e
- A avaliação da validade do experimento.

4.3.3 Operação

Depois de o experimento ter sido projetado e esboçado ele deve ser executado para se coletarem os dados a serem analisados. Isso é feito na etapa de operação, a qual é dividida em três fases (WOHLIN *et al.*, 2000):

1) **Preparação**: os participantes e os materiais são escolhidos, organizados e preparados para a execução do experimento.

2) **Execução**: os participantes desempenham suas tarefas de acordo com o que foi planejado e os dados são coletados.

3) **Validação de Dados**: os dados que foram coletados na etapa anterior são validados, a fim de conferir se tudo foi coletado corretamente.

Segundo Travassos, Gurov e Amaral (2002), o aspecto mais importante da fase de operação é que a parte humana entra em cena. É importante que os participantes sejam preparados para a tarefa, pois resultados errôneos podem ocorrer devido ao mal-entendido ou falta de interesse dos participantes.

4.3.4 Análise e Interpretação

Nessa etapa, são feitas análises e interpretações a partir dos dados coletados na fase de operação. A primeira fase é a estatística descritiva, a qual explora a tendência central, dispersão, etc. Depois disso, os dados com pontos anormais e falsos são excluídos. Assim, o conjunto de dados é reduzido, facilitando a aplicação de outros métodos estatísticos para se testarem as hipóteses (WOHLIN *et al.*, 2000).

4.3.5 Apresentação e Pacote

A última fase do processo de experimentação é a fase da apresentação e empacotamento. Ao fim da realização do experimento, normalmente deseja-se apresentar os resultados obtidos e documentá-los de forma que o experimento possa ser repetido (WOHLIN *et al.*, 2000). Isso pode ser feito com um artigo em uma conferência, um relatório para tomada de decisões, um pacote para replicação de experimento ou na forma de um material educacional (WOHLIN *et al.*, 2000).

Além de permitir que outros pesquisadores possam reutilizar o estudo e repetir o experimento, o empacotamento também pode ser útil em empresas, melhorando e ajudando a entender diferentes processos existentes (WOHLIN *et al.*, 2000).

4.4 Empacotamento

Amaral e Travassos (2002) relataram que uma parte considerável das publicações na área de Ciência da Computação não apresenta um experimento para comprovar o que

foi escrito. Quando comparada com outras ciências, a Ciência da Computação possui um baixo percentual de pesquisadores que validam o seu trabalho.

Um agravante para esse problema é a falta de visibilidade dos experimentos já realizados, uma vez que somente quem os elaborou conhece os seus detalhes. Para outra pessoa trabalhar com o mesmo assunto, ou mesmo repetir o experimento já realizado, ela vai ter que começar o estudo praticamente do zero. Para que esse problema na replicação de experimentos seja minimizado, é necessário o empacotamento do experimento (AMARAL; TRAVASSOS, 2002).

Segundo Amaral e Travassos (2002), o empacotamento do experimento permite armazenar todos os artefatos que foram usados durante o processo de experimentação, impedindo assim, a perda de informações importantes durante a execução do mesmo. Com o empacotamento, torna-se muito mais fácil a sua replicação, que é uma das características mais importantes de um experimento (TRAVASSOS; GUROV; AMARAL, 2002).

Com a replicação, tem-se um aumento no aprendizado dos conceitos investigados e também a calibração das características do experimento. Dessa forma, o empacotamento padronizado dos dados experimentais pode servir como base para a criação de bibliotecas de experimentação (TRAVASSOS; GUROV; AMARAL, 2002).

Em seu trabalho, Amaral e Travassos (2002) propuseram uma evolução no processo de experimentação de Wohlin e outros (2000), no qual a etapa de Apresentação e Empacotamento é executada em paralelo com as outras fases, de forma a evitar perda de informações ao longo do processo. A Figura 4.2 apresenta o processo de experimentação proposto por Amaral e Travassos (2002).

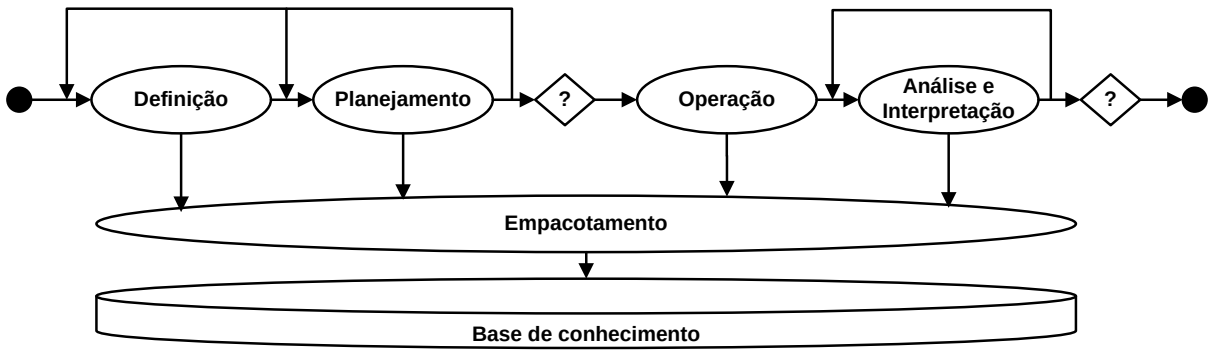


Figura 4.2 - Processo de experimentação proposto por Amaral e Travassos (2002)

Para que um pacote possa ser útil, fácil de ser encontrado e reutilizado, ele deve estar bem rotulado. Esse rótulo, no caso dos estudos experimentais, pode se dar por meio de seu resumo, o qual, de acordo com Kitchenham (2004) deve ser apresentado de forma mais estruturada, contendo as informações de cenário, objetivo, método, resultados e

conclusões. Assim, o resumo provê uma maneira fácil e rápida de descobrir a relevância, qualidade e generalidade do artefato (ou pacote) a quem estiver procurando.

Vale lembrar que a replicação é o atributo mais importante associado ao processo de experimentação, pois permite que outros pesquisadores reproduzam o experimento, possibilitando que as hipóteses sejam validadas. Assim, a facilidade de replicação do experimento é dependente da qualidade do seu empacotamento.

Por se tratar de uma área nova, o empacotamento de experimentos não possui normas internacionais aprovadas (TRAVASSOS; GUROV; AMARAL, 2002). No livro "*Guide to Advanced Empirical Software Engineering*" Shull e outros (2007) apresentam vários tópicos a respeito dos experimentos na engenharia de software. Um dos capítulos de destaque nesse livro é o capítulo 8, "*Guideline for Reporting Controlled Experiments*" (JEDLITSCHKA; CIOLKOWSKI; PFAHL, 2008), o qual propõe um guia para relatar experimentos em engenharia de software, baseado em uma revisão de literatura sobre os diversos guias já propostos, entre eles o trabalho de Kitchenham e outros (2008). O capítulo apresenta detalhadamente o conteúdo esperado nas seções e subseções de um relatório de experimento. Os experimentos contidos no capítulo 7 desta dissertação foram escritos baseando-se no modelo proposto por Jedlitschka e outros (2008).

4.5 Experimentação em Engenharia de Software

Experimentos controlados têm se mostrado um meio efetivo para avaliar as técnicas de Engenharia de Software, bem como adquirir o entendimento necessário sobre sua utilidade (LOTT; ROMBACH, 1996). Segundo Travassos, Gurov e Amaral (2002), é necessário estabelecer metodologias específicas para ajudar a criar uma base de engenharia e de ciência para a Engenharia de Software, devido ao duplo caráter dessa área: parte ciência e parte engenharia.

Basili e outros (1996a) consideram que novos métodos, técnicas, linguagens e ferramentas não deveriam apenas ser sugeridos, publicados ou apresentados para venda sem passarem pela experimentação e validação. No caso específico da Engenharia de Software, a experimentação tem como objetivo caracterizar, avaliar, prever, controlar ou melhorar tanto os produtos, como também os processos, recursos, modelos ou teorias (TRAVASSOS; GUROV; AMARAL, 2002).

Wohlin e outros (2000) listam quatro métodos para se conduzir um experimento: o científico; o de engenharia; o empírico e o analítico.

O **método científico** permite construir um modelo baseado nas observações do mundo real (WOHLIN *et al.*, 2000). Por exemplo: quando se tenta entender o processo, produto ou ambiente de software, o método tenta extrair do mundo real algum modelo que possa explicar um fenômeno e avaliar se o modelo é realmente representativo para o fenômeno sob observação.

Diferentemente do método científico, o **método de engenharia** não sugere um modelo novo, mas sugere soluções mais adequadas para as soluções já existentes (WOHLIN *et al.*, 2000). Isto é uma abordagem orientada à melhoria evolutiva, desenvolvendo, medindo e analisando até que nenhuma melhoria seja possível.

O **método experimental**[2] avalia uma teoria ou um modelo, utilizando a experimentação para obter os dados qualitativos e/ou quantitativos, que serão medidos e analisados para que o modelo ou a teoria possam ser avaliados. O método experimental tem como premissa a repetição do processo, isto é, a replicação do experimento, a fim de aumentar a credibilidade dos resultados obtidos (WOHLIN *et al.*, 2000).

O **método analítico** (ou matemático) define teorias formais e deriva os resultados a partir das mesmas. Ou seja, esse é um método dedutivo que não precisa de um projeto experimental (no seu sentido estatístico), mas oferece uma base analítica para o desenvolvimento de novos modelos (WOHLIN *et al.*, 2000).

Travassos, Gurov e Amaral (2002) afirmam que a abordagem mais apropriada para a experimentação na área de Engenharia de Software é o método experimental, pois este considera a proposição e avaliação do modelo por meio de estudos experimentais. Entretanto, podem-se utilizar outros métodos de acordo com o propósito. Por exemplo, de acordo com Travassos, Gurov e Amaral (2002) o método científico pode ser usado para compreender a construção de um software, a fim de verificar se o mesmo pode ser utilizado para automatizar o processo da organização; o método de engenharia pode ser útil para mostrar que uma ferramenta possui um desempenho melhor do que outra; já o método analítico pode provar modelos matemáticos para conceitos, como o crescimento da confiabilidade, a complexidade do software, etc.

Vários outros estudos já foram desenvolvidos a respeito da aplicação da experimentação no contexto da engenharia de software (BASILI; SELBY, 1987; CONRADI *et al.*, 2001; DÓRIA, 2001; HETZEL, 1976; KAMSTIES; LOTT, 1995; LOTT; ROMBACH, 1996; MYERS *et al.*, 2004; WOOD *et al.*, 1997).

Outro trabalho que merece ser destacado é o de Shull, Carver e Travassos (2001) que, a partir do crescente interesse em estudo empíricos na Engenharia de Software, definiram um processo sistemático e incremental para validação de tecnologias maduras, e

[2] Também referenciado na literatura como método empírico.

um guia para a melhoria de novas tecnologias. Esse processo é composto pelas etapas: Estudo de viabilidade, Estudo observacional, Estudo de caso em um ciclo de vida real, e Estudo de caso na indústria, conforme mostra a Figura 4.3.

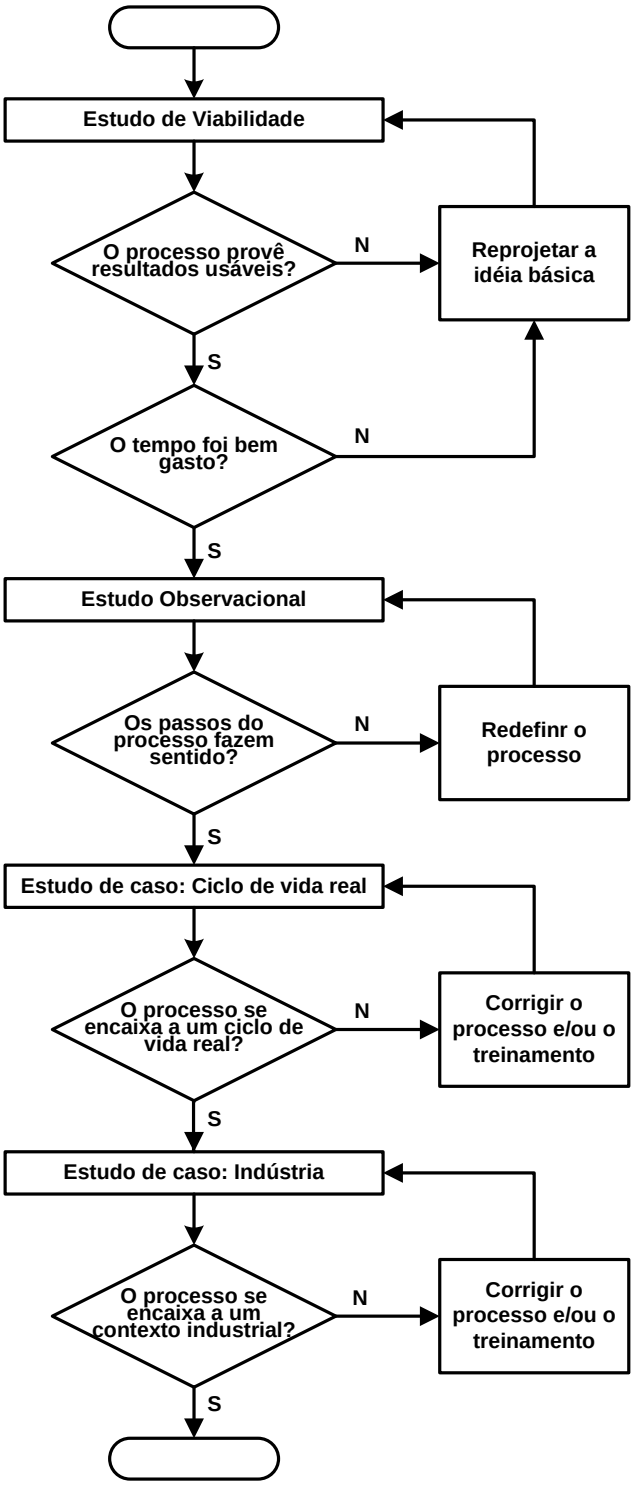


Figura 4.3 - Processo para validação de tecnologias proposto por Shull, Carver, Travassos (2001)

A proposta de uma nova metodologia deve ser avaliada iniciando-se com um estudo de viabilidade, o qual vai explorar se o novo processo atende ao objetivo geral para o qual foi criado. Depois disso, no estudo observacional, a proposta será avaliada para verificar se cada etapa do processo é efetiva e se a ordem em que são executadas faz sentido. O próximo passo é garantir que o novo processo é adequando a um ciclo de desenvolvimento real. Essa etapa é realizada por meio de um estudo de caso. O ultimo passo é verificar se o novo processo é aplicável em um contexto industrial sem nenhum resultado negativo inesperado. Esse passo também é realizado por meio de um estudo de caso.

A avaliação da ferramenta CRISTA, descrita no Capítulo 6, está seguindo esse processo e as fases de Estudo de viabilidade e Estudo observacional estão relatadas no Capítulo 7, apresentado os dados coletados até o momento.

4.6 Considerações Finais

Este capítulo apresentou uma revisão da literatura a respeito dos estudos experimentais, que estão sendo utilizados cada vez mais na computação, principalmente na área de Engenharia de Software. Foram apresentados os tipos de estudos experimentais, o processo de experimentação e a etapa de empacotamento, que é fundamental tanto para a transferência de conhecimento como também para permitir uma replicação do experimento realizado.

Também se deu ênfase à importância da experimentação no contexto da Engenharia de Software, mostrando os avanços e necessidades nesse contexto específico. Com essa revisão bibliográfica, foi possível perceber a importância dos estudos experimentais para a introdução de novas tecnologias no mercado. A partir disso, foi possível planejar as avaliações da ferramenta CRISTA, a qual é apresentada no Capítulo 6, de acordo com o processo definido por Shull, Carver, Travassos (2001).

Todos os conceitos apresentados neste capítulo dão suporte ao planejamento e entendimento dos estudos realizados neste trabalho, descritos no Capítulo 7.

Concluindo o conteúdo de revisão bibliográfica contida neste trabalho, o capítulo seguinte apresenta alguns dos principais artigos encontrados, explorando suas idéias centrais e o seu relacionando com o presente trabalho.

Capítulo 5

TRABALHOS RELACIONADOS

Este capítulo apresenta um resumo dos artigos que se destacaram no levantamento bibliográfico. Eles são apresentados individualmente, explorando e analisando suas características peculiares, no contexto deste trabalho.

5.1 Considerações Iniciais

Como encerramento da revisão bibliográfica referente a este trabalho, este capítulo apresenta com maiores detalhes alguns trabalhos relacionados a inspeção, visualização e experimentação de software, assuntos que já foram discutidos nos capítulo anteriores. Vale lembrar que os artigos citados nessa revisão bibliográfica foram provenientes de um levantamento bibliográfico *ad-hoc* e também da aplicação de uma Revisão Sistemática sobre inspeção de software baseada nos trabalhos de Kitchenham (2004) e Biolchini e outros (2005). Do conjunto de artigos encontrados, são apresentados, sucintamente, apenas alguns deles que mais se relacionam com este trabalho.

Assim, na Seção 5.2 são apresentados e discutidos os artigos individualmente e, na Seção 5.3, são apresentadas as considerações finais, destacando-se os pontos mais importantes para o contexto deste trabalho.

5.2 Literatura relevante

Os artigos aqui apresentados são somente alguns dos que subsidiaram a criação da ferramenta aqui proposta. Para cada artigo, é apresentado o título, os autores, o ano de publicação, e uma discussão acerca dos principais pontos associados ao trabalho.

- **1 *Introducing the Next Generation of Software Inspection Tools* (HEDBERG, 2004)**

Esse artigo, já mencionado no Capítulo 2, apresenta uma revisão da literatura a respeito das ferramentas de apoio à inspeção de software na década de 1990. Um aspecto interessante do artigo é que o autor categoriza 16 ferramentas existentes em quatro gerações, de acordo com os tipos de reuniões apoiadas por elas (Primeiras ferramentas, Ferramentas distribuídas, Ferramentas assíncronas e Ferramentas baseadas na web).

Ao final do artigo, o autor estabelece alguns critérios que deverão ser seguidos pelas próximas ferramentas de inspeção, chamando a atenção para quatro pontos:

- Gerenciamento dos artefatos;
- Flexibilidade quanto ao processo de inspeção;
- Geração de resultados; e
- Resumo de informações.

Quanto a geração de resultados, segundo o autor, uma ferramenta de inspeção deve necessariamente apoiar a coleta de métricas de forma automática. Com base nesse requisito, procurou-se inserir na ferramenta CRISTA a coleta automática do tempo, de forma a apoiar não só o processo de inspeção, mas também dar suporte a decisões.

O artigo apresentou um bom embasamento teórico a respeito das ferramentas existentes ao longo do tempo. Ele forneceu subsídios para direcionar a criação da ferramenta CRISTA, que seguiu a idéia básica de uma ferramenta de inspeção que, segundo o autor, é ajudar os inspetores a lidar com os artefatos baseando-se em um processo predefinido, de forma que seja permitido o armazenamento de anotações, métricas e resultados.

- **2 *A Tool to Support Perspective Based Approach to Software Code Inspection* (CHAN; JIANG; KARUNASEKERA, 2005)**

Esse artigo discute os benefícios de se utilizar a técnica de leitura PBR no contexto de inspeções de softwares orientados a objetos. Explorando esse contexto, os autores propõem a criação de uma ferramenta chamada CIT (também já mencionada no Capítulo 2), a qual apóia a aplicação da técnica de leitura PBR. Além de discutir as vantagens e desvantagens do uso de técnicas de leitura, principalmente da PBR, o artigo apresenta uma pequena lista comparando algumas ferramentas de apoio ao processo de inspeção.

Os autores do artigo defendem que as ferramentas podem automatizar certas partes do processo de inspeção, o que pode ter um grande impacto no processo como um todo. Assim, a intenção dos autores é que a ferramenta CIT possa ser mais efetiva e mais eficaz, quando comparada com inspeções manuais.

No artigo, além de serem apresentados detalhes a respeito da arquitetura da ferramenta CIT, são detalhados os aspectos funcionais da mesma. Ao fim do artigo, os autores relatam sobre o planejamento de se realizar estudos empíricos de forma a verificar a real utilidade da ferramenta.

De maneira semelhante ao artigo, o presente trabalho busca apoiar a atividade de inspeção, dando suporte à técnica de leitura *Stepwise Abstraction*. O trabalho de Chan, Jiang e Karunasekera (CHAN; JIANG; KARUNASEKERA, 2005) também se destacou ao ressaltar a importância de estudos experimentais para comprovar e/ou ao dar indícios da utilidade e eficácia da ferramenta criada.

- **3 *An Empirical Study on Groupware Support for Software Inspection Meetings* (GRÜNBACHER; HALLING; BIFFL, 2003)**

Esse artigo discute a importância das reuniões em um processo de inspeção. Segundo os autores, a reunião de inspeção é uma atividade chave para concordância dos defeitos coletados, eliminação de falso-positivos, além de disseminação do conhecimento pelos membros da equipe. Nesse sentido, os autores criaram a ferramenta GrIP (*GRoupware Supported Inspection Process*), a qual dá suporte às reuniões de inspeção, bem como ao processo de inspeção como um todo.

A ferramenta GrIP provê um framework e ferramentas colaborativas para uma equipe de inspetores e apóia detecções individuais de diferentes artefatos, reuniões de equipe, assim como o gerenciamento da inspeção. Conforme apresentado no artigo, o apoio computacional oferecido pela ferramenta GrIP possui as seguintes vantagens:

- Aumento da efetividade: Equipes usando GrIP encontram mais defeitos que equipes que realizam inspeções manuais;
- Esforço reduzido: Inspetores e equipe de inspeção apoiados pela ferramenta gastam menos tempo na detecção de defeitos que inspetores que realizam a inspeção usando papéis;
- Diminuição de defeitos sobrepostos: A sobreposição de defeitos usando o GrIP é menor que inspeções que usam papeis; e
- Aumento da eficiência: Equipes usando GrIP são mais eficientes que equipes que aplicam inspeções manuais.

Durante a reunião, a ferramenta GrIP permite um consenso da severidade dos defeitos entre os inspetores através de um sistema de votação, como pode ser observado no gráfico da Figura 5.1. Segundo os autores, a ferramenta dá um bom apoio ao moderador da inspeção, eliminando automaticamente falso-positivos (com base na votação de todos os participantes), além de dar indícios de por onde começar a reunião de inspeção.

Um ponto a ser destacado no artigo corresponde ao relato do experimento realizado com 37, alunos que investigou os efeitos da ferramenta em um ambiente de reuniões de inspeção. Nesse experimento, os autores procuraram analisar o apoio da ferramenta GrIP para a discriminação de defeitos avaliando a diferença entre a inspeção feita com e sem a ferramenta. Os resultados mostraram que o apoio computacional reduziu o número de falso-positivos, além de melhorar a discriminação de defeitos.

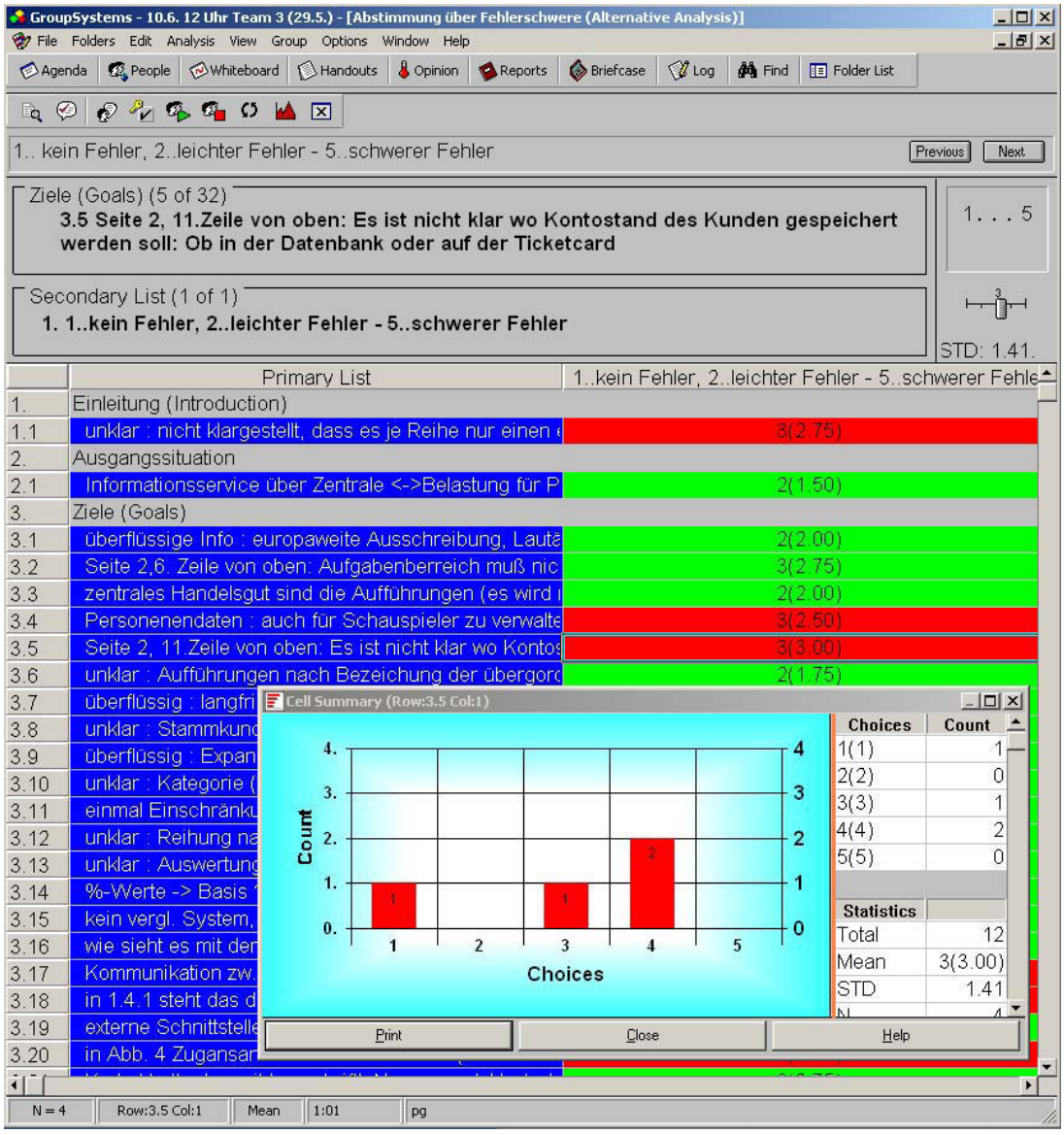


Figura 5.1 - Tela da ferramenta GrIP (GRÜNBACHER; HALLING; BIFFL, 2003)

Análogo ao trabalho de Grünbacher e outros, o presente trabalho também propôs um apoio computacional para o processo de inspeção, no qual são necessários estudos experimentais para comprovar a real eficácia e eficiência do apoio computacional da ferramenta. Em outro artigo dos mesmos autores (HALLING; BIFFL; GRÜNBACHER, 2003), o foco são os experimentos realizados com a ferramenta. Assim, novamente foi salientada a necessidade de realização de estudos experimentais para avaliação da ferramenta CRISTA, os quais são descritos a seguir, no Capítulo 7.

- **4 *An Empirical Study of Web-Based Inspection Meetings* (LANUBILE; MALLARDO, 2003)**

Esse artigo apresenta um estudo experimental para avaliar uma ferramenta para apoio de inspeções distribuídas. A ferramenta, IBIS (*Internet-Based Inspection System*), foi construída baseada na reestruturação do processo de inspeção sugerido por Sauer e outros (2000). Assim, a ferramenta contempla reuniões distribuídas por meio de fóruns de discussões além de dar suporte a votações.

Usando a ferramenta IBIS, os defeitos detectados por cada inspetor são reunidos pela ferramenta em uma única lista de defeitos. Nessa lista, o moderador da inspeção pode marcar defeitos idênticos indicados por inspetores diferentes, como duplicados, além de poder marcar um defeito como sendo verdadeiro para ser futuramente encaminhado ao autor na fase de retrabalho.

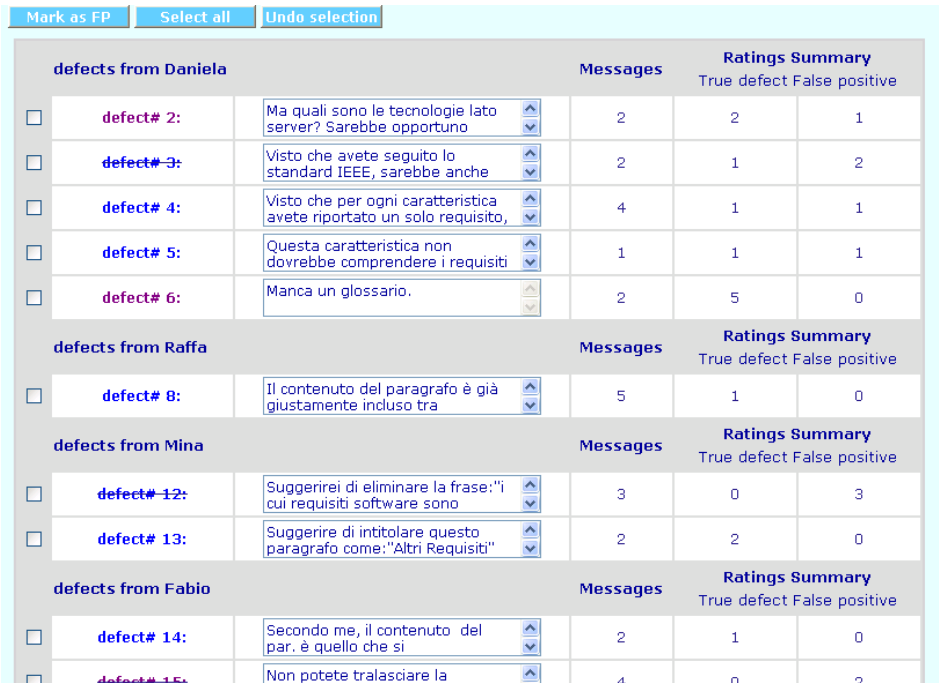


Figura 5.2 - Lista de defeitos na ferramenta IBIS (LANUBILE; MALLARDO, 2003)

Pela lista de defeitos, que pode ser vista na Figura 5.2, o moderador pode selecionar quais defeitos serão discutidos na etapa de discriminação, além de poder escolher quais inspetores participarão da reunião de consenso. Para auxiliar na reunião de consenso, a ferramenta IBIS oferece um sistema de votação em que os inspetores avaliam o grau de potencial defeito como defeito verdadeiro ou como falso-positivo.

Nesse artigo, são relatados ainda os resultados de nove inspeções geograficamente distribuídas feitas com estudantes de graduação. O foco do experimento era identificar a discriminação de defeitos verdadeiros e falso-positivos em um contexto de reuniões assíncronas.

A relação do artigo com o trabalho proposto é que a ferramenta CRISTA também apóia inspeções assíncronas. Entretanto, diferentemente do trabalho de Lanubile e Mallardo, a ferramenta CRISTA não é uma ferramenta web, mas sim uma ferramenta desktop, dotada de recursos de visualização de forma a facilitar a compreensão do artefato a ser inspecionado (código). A avaliação da ferramenta se deu por meio de experimentos, os quais são detalhados no Capítulo 7.

- **5 *Detection or Isolation of Defects? An Experimental Comparison of Unit Testing and Code Inspection* (RUNESON; ANDREWS, 2003)**

Esse artigo apresenta um estudo comparando a eficácia de testes caixa branca e inspeções de código. A motivação da pesquisa foi averiguar qual técnica é superior à outra, ou qual técnica é melhor para detectar certos tipos de defeitos. Para essa investigação, foi conduzido um experimento para avaliar as falhas descobertas com os testes e as faltas descobertas com a inspeção.

O experimento contou com a participação de 30 estudantes de graduação, e foram usadas as técnicas: inspeção baseada em uso e teste de cobertura. A experiência dos participantes foi estimada por meio de um questionário contendo 10 questões. As habilidades dos participantes foi medida a partir de dois exercícios, um para teste e outro para inspeção. O experimento foi executado separando-se os alunos em dois grupos, de acordo com a Tabela 5.1. Após a execução do experimento, os participantes responderam a um questionário.

Tabela 5.1 - Desenho experimental (RUNESON; ANDREWS, 2003)

		Programa usado	
		Counter	Correlation
Técnica usada	Inspeção	1-15	16-30
	Teste	16-30	1-15

Os resultados mostraram que, dependendo do contexto, existe uma diferença significativa entre as duas abordagens. No estudo, os testes detectaram significativamente mais defeitos (falhas), enquanto a inspeção isola melhor os defeitos detectados, ou seja, eles detectam o defeito que ocasionou a falha. Os testes possuem pouca eficiência, pois consomem significativamente mais tempo.

Os autores concluem que pode não ser muito útil comparar as duas técnicas, pois elas trabalham de forma diferente, detectando tipos diferentes de defeitos. Dessa forma, as técnicas devem ser usadas em conjunto.

Esse artigo é significante no contexto desse trabalho, pois novamente é enfatizada a necessidade de se conduzir estudos experimentais para se avaliar a atividade de inspeção. No caso desse trabalho, os estudos experimentais buscaram avaliar o comportamento da ferramenta criada nesse contexto e para o contexto de manutenção também. Para o presente trabalho, o perfil dos participantes e outros resultados também foram levantados por meio de questionários, da mesma forma que no experimento de Runeson e Andrews.

- **6 *Visualisation-Based Tool Support for the Development of Aspect-Oriented Programs* (PFEIFFER; GURD, 2006)**

Esse artigo apresenta uma ferramenta para apoiar a programação orientada a aspectos (POA). Os autores discutem a necessidade de uma ferramenta que apóie o entendimento de grandes programas orientados a aspectos. Esse entendimento é necessário, sobretudo para a identificação da intersecção dos aspectos no código. No artigo, esse entendimento foi buscado por meio da técnica de visualização *Treemap*, a qual apresenta graficamente essa intersecção.

Assim, foi criada a ferramenta Asbro, que é um *plugin* de visualização de programas AspectJ para a IDE eclipse. A proposta da ferramenta Asbro é melhorar a visualização e, por conseqüência, a compreensão dos aspectos presentes em grandes softwares.

O artigo comenta sobre os problemas em visualizar grandes quantidades de informação por meio do visualizador que já existe no AspectJ. Por causa desses problemas, os autores usaram o *Treemap* como forma alternativa de visualização. Segundo os autores, o *Treemap*, se tornou uma alternativa ao visualizador do AJDT, melhorando a visualização das informações e permitindo uma melhor navegação.

Ao apresentar a ferramenta Asbro, a qual pode ser vista na Figura 5.3, os autores defendem o uso da técnica, indicando que ela permite visualizar uma grande quantidade de informações de maneira organizada. Além disso, a técnica *Treemap* deixa as estruturas orientadas a aspectos mais explícitas, o que facilita o entendimento do código, indicando quais classes são interceptadas por um aspectos.

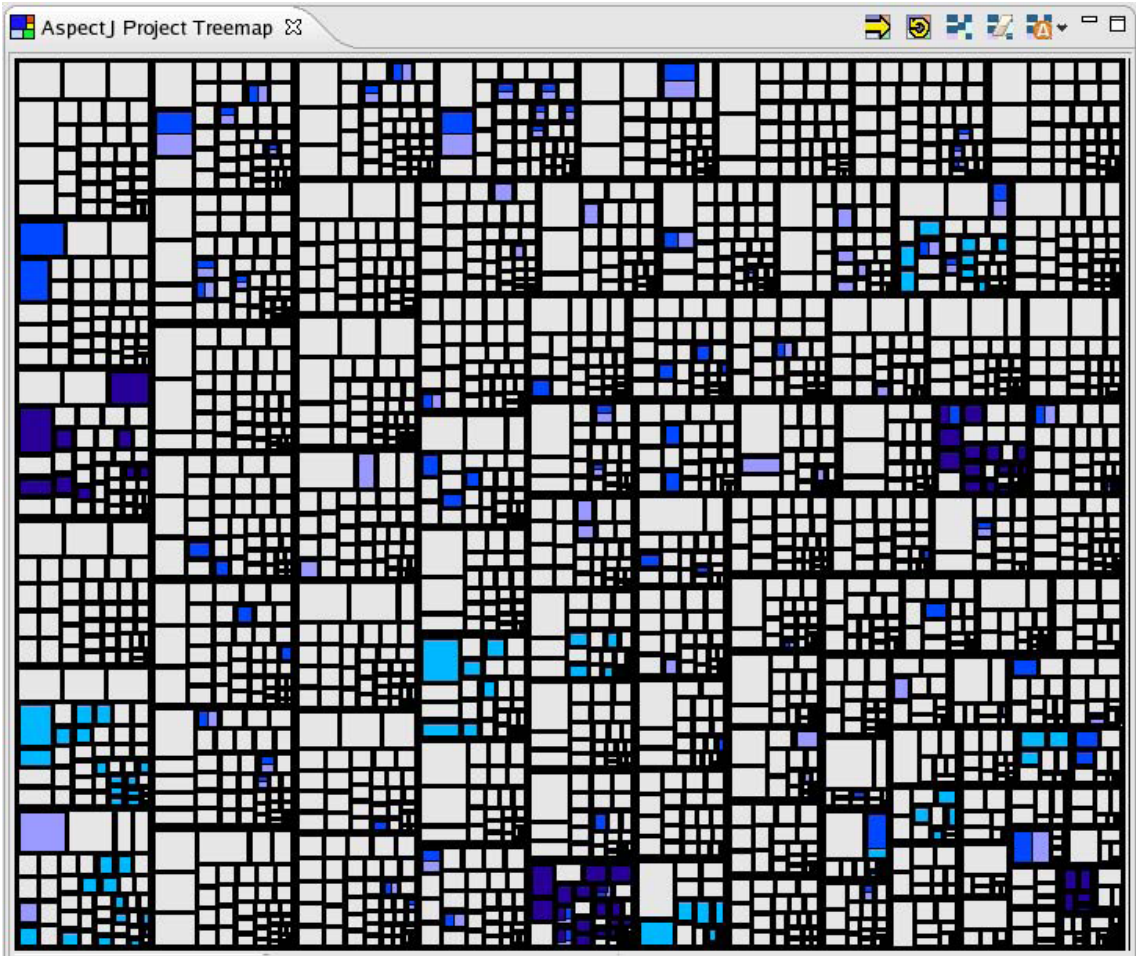


Figura 5.3 – Exemplo de visualização provida pela ferramenta Asbro (PFEIFFER; GURD, 2006)

Na Figura 5.3, cada retângulo corresponde a uma classe de um projeto de 2491 classes. Cada cor na figura corresponde a um aspecto diferente, e a visualização provida pela ferramenta Asbro indica visualmente, por meio das cores, quando um aspecto intercepta uma classe.

O artigo contempla ainda um estudo experimental para avaliação quantitativa da ferramenta. Nesse estudo, cinco pessoas tiveram algumas atividades que foram resolvidas de quatro formas diferentes: AJDT sem a visualização, AJDT com visualização, ferramenta de pesquisa Lost, e a ferramenta Asbro.

No estudo, os autores observaram que o uso da visualização com *Treemap* reduziu significativamente o tempo para entender o programa. Para algumas tarefas a redução foi ao menos 50% do tempo devido ao uso da visualização. Por fim, os autores enfatizam o fato de que a visualização com *Treemap* permite sobretudo uma redução no tempo de exploração do código fonte.

Da mesma forma que o trabalho de Pfeiffer e Gurd, o presente trabalho adota a técnica de visualização *Treemap*. Essa técnica foi adotada por o código ser uma estrutura hierárquica e é esperado obter facilidade na navegação e compreensão do código.

- **7 *Understanding Object-Oriented Source Code from the Behavioural Perspective* (WALKINSHAW; ROPER; WOOD, 2005)**

Esse artigo discute a compreensão de códigos orientados a objetos, em que, na maioria das vezes, possui a implementação de interesses espalhados por todo o sistema. Para resolver esse problema, o artigo apresenta uma técnica que, a partir de uma pequena quantidade de informação, consegue rastrear os métodos do sistema responsáveis pela implementação de um interesse de alto nível.

A técnica apresentada no artigo consiste em dividir o programa em pedaços de forma a determinar as chamadas de métodos relevantes com um requisito de alto nível específico. Para a utilização dessa técnica foi criada uma ferramenta, que, a partir de um caso de uso dado, retorna uma quantidade gerenciável de código para ser lido e entendido.

Realizar esse tipo de tarefa de forma manual é inviável devido à grande quantidade de código relacionado a um único requisito. Nesse caso, o apoio computacional restringe o grafo de chamada dos métodos, de forma que é retornada somente uma porção gerenciável de código para ser entendido. Para realizar esse corte, foram seguidas as seguintes etapas:

1- Marcar os métodos de referência de uma especificação para o grafo de chamada.

2- Identificar caminhos diretos entre os métodos traçados

3- Identificar os caminhos que podem influenciar e serem influenciados por caminhos na rede.

Para avaliar a viabilidade da proposta acima, os autores implementaram uma ferramenta para extração de código, na qual o código fonte era carregado e, a partir de então, era gerado um grafo de chamadas para o mesmo. O artigo descreve um estudo de caso em que foi possível simplificar o grafo da Figura 5.4 para o grafo da Figura 5.5.

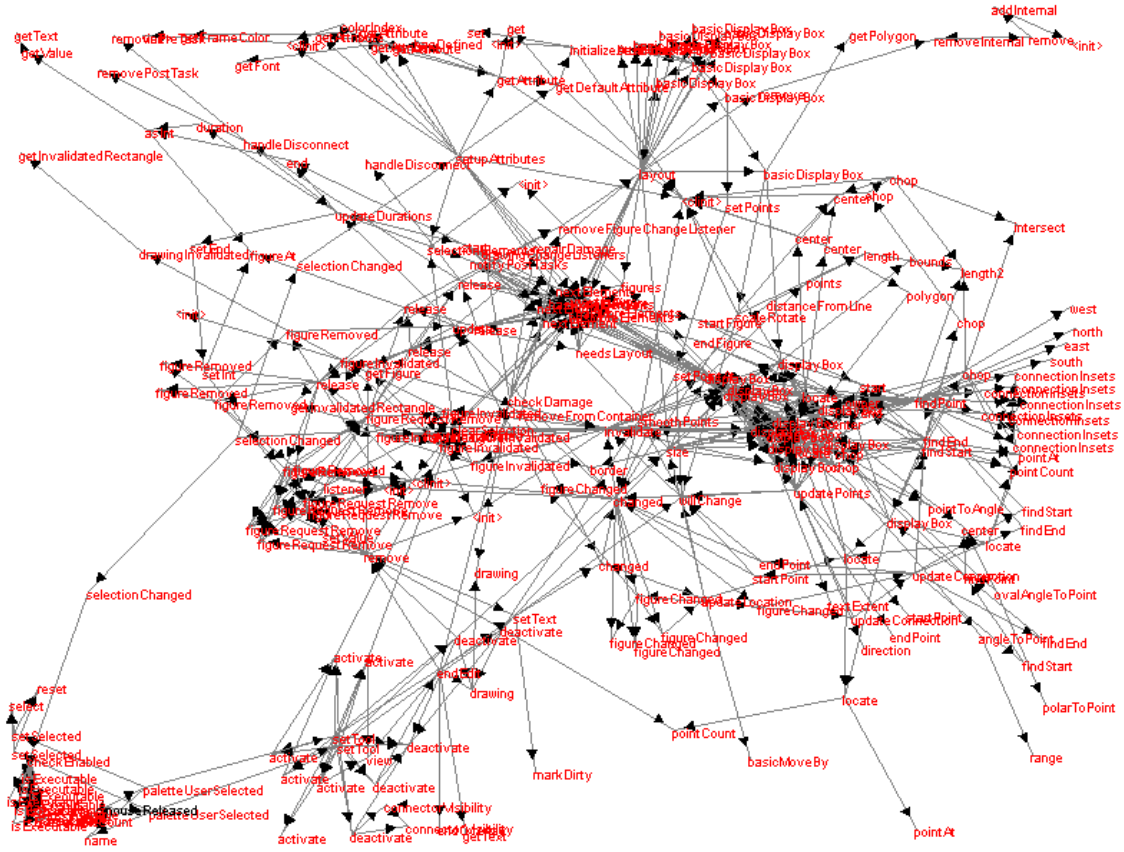


Figura 5.4 - Grafo de chamadas original (WALKINSHAW; ROPER; WOOD, 2005)

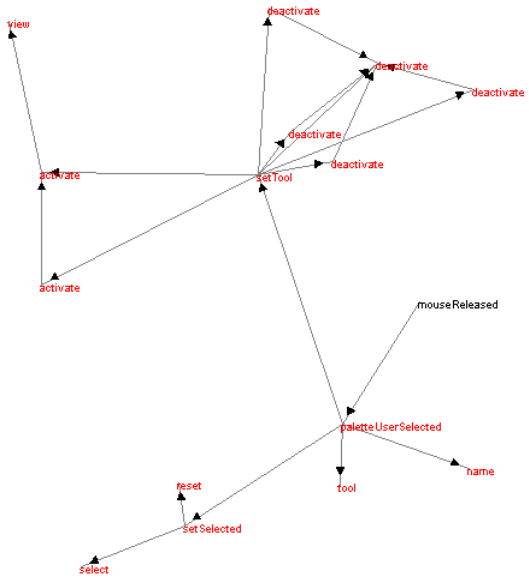


Figura 5.5 - Grafo de chamadas reduzido (WALKINSHAW; ROPER; WOOD, 2005)

Por fim, os autores afirmam que a ferramenta não apenas auxilia em atividades de compreensão de código, como também pode ajudar em atividades de análise estática de código, ou seja, inspeções de código.

A relação desse artigo com o presente trabalho é o uso de metáforas visuais como auxílio nas atividades de compreensão. Como o próprio autor deixa claro, muitas atividades como manutenção, teste e inspeções precisam fortemente do uso de técnicas de compreensão efetivas. Por isso, para este trabalho, usou-se uma técnica de visualização para apoiar a atividade inspeção de código.

- **8 *Design and Implementation of a Fine-Grained Software Inspection Tool* (ANDERSON; REPS; TEITELBAUM, 2003)**

Esse artigo descreve a experiência de Anderson e outros no projeto e implementação da ferramenta CodeSurfer (já mencionada no Capítulo 2), a qual usa recursos de visualização (grafos de dependência) para apoiar atividades de inspeção de código. Cabe ressaltar que a parte teórica de grafos de dependência está muito bem documentada no artigo.

A ferramenta CodeSurfer pode revelar pequenos detalhes da semântica dos programas por meio de pesquisas no grafo de dependência. Como observado na Figura 5.6, cada função do programa no CodeSurfer é representada como uma coleção de pontos do programa (representado pelas elipses) conectados por arestas (mostradas como setas). Uma vez montada essa visualização de grafo, a ferramenta ajuda a visualizar, por exemplo, onde estão as dependências de uma variável específica.

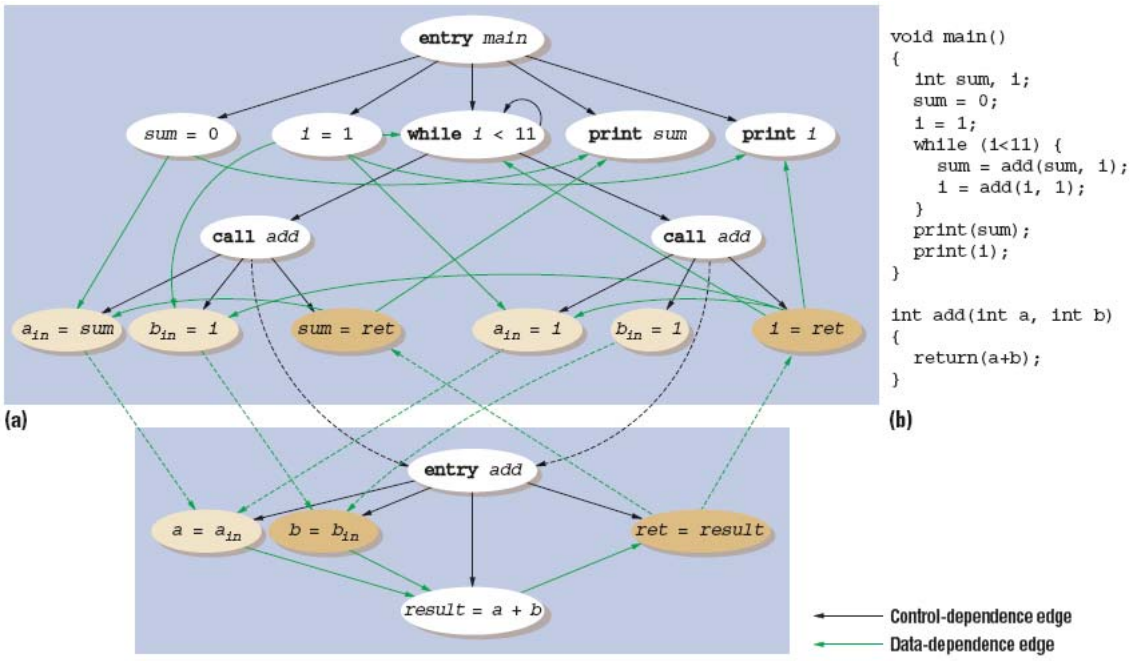


Figura 5.6 - Grafo de dependência (a) para o programa mostrado em (b) (ANDERSON; REPS; TEITELBAUM, 2003)

Uma das principais características da ferramenta CodeSurfer, é que, a partir dos grafos de dependência, ela ajuda a gerenciar grandes quantidades de informação. Além disso, ela também apresenta a funcionalidade de poder navegar (surfar) pelo código. Essa característica está presente na maioria dos compiladores e IDEs atuais. Ao final do artigo, o autor apresenta algumas ferramentas como trabalhos relacionados, e termina o artigo falando da pretensão de aumentar a escalabilidade da ferramenta de forma a usá-la em outros contextos.

A relação desse artigo com o trabalho proposto nesta dissertação está no mapeamento do código em uma metáfora visual, de forma a facilitar a compreensão do mesmo. A diferença dos dois trabalhos, além do tipo de metáfora visual utilizada, está no propósito dos mesmos. Enquanto Anderson e outros propõem utilizar o software para obter detalhes semânticos do código em uma inspeção, o presente trabalho almeja apoiar a técnica de leitura *Stepwise Abstraction*, de forma a facilitar o processo de inspeção, além de contribuir em outras atividades que requerem compreensão de código como, por exemplo, a manutenção.

- **9 *Using Dependence Graphs to Assist Manual and Automated Object Oriented Software Inspections* (COOPER *et al.*, 2006)**

Semelhantemente ao trabalho de Anderson, Reps e Teitelbaum (2003), que apresentou a ferramenta CodeSurfer, o trabalho de Cooper e outros também apresentam uma ferramenta (Coffee Grinder) para apoiar a inspeção de código. Em ambos os trabalhos, as ferramentas de apoio à inspeção utilizaram uma visualização do código na forma de grafos de dependência. Um grafo de dependência disponibilizado pela ferramenta Coffee Grinder pode ser observado na Figura 5.7, na qual é apresentado um método e os seus métodos e atributos dependentes.

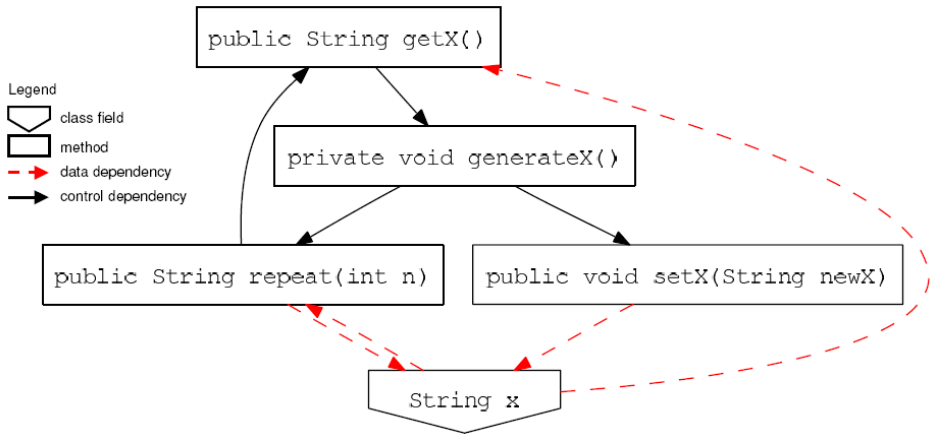


Figura 5.7 - Exemplo de grafo de dependência disponibilizado pela ferramenta Coffee Grinder (COOPER *et al.*, 2006)

O artigo de Cooper e outros (já mencionada no Capítulo 2) destaca as funcionalidades da ferramenta Coffee Grinder, bem como os detalhes de seu projeto. Diferentemente do trabalho de Anderson, Reps e Teitelbaum (2003), a ferramenta Coffee Grinder permite escrever scripts para encontrar defeitos ou localizar potenciais áreas para serem inspecionadas.

Além de o artigo oferecer bons argumentos de como a ferramenta pode ser útil no contexto de inspeção, ao final do artigo, os autores enfatizam que o uso de técnicas de visualização como grafos de dependência pode contribuir muito na compreensão do código. Esses comentários estão de acordo com o que é esperado no presente trabalho, em que se pretende apoiar a compreensão, e por conseqüência a inspeção de código, por meio de uma técnica de visualização (*Treemap*).

Uma parte interessante observada na ferramenta Coffee Grinder é que ela precisou quebrar o código fonte (Java) em *tokens* de forma que fosse possível construir a metáfora visual de grafos. Para esse processo, os autores de Coffee Grinder realizaram um *parser* do código Java usando o gerador de compiladores JavaCC (JAVACC, 2008). Como no contexto do trabalho de Cooper e outros, o JavaCC (JAVACC, 2008) serviu bem ao que se propôs, o mesmo foi escolhido como ponto de partida para a criação da metáfora visual da ferramenta CRISTA, discutida em maiores detalhes no capítulo seguinte.

5.3 Considerações Finais

Neste capítulo, de forma a completar a revisão bibliográfica, foram apresentados alguns artigos relacionados ao contexto deste trabalho. Tais artigos abordaram os temas inspeção, visualização e experimentação que influenciaram, de alguma forma, as decisões deste trabalho, tanto no que diz respeito à implementação da ferramenta CRISTA, como à condução dos estudos experimentais.

Várias características e funcionalidades presentes na ferramenta CRISTA foram inspiradas na leitura dos diversos artigos discutidos (ANDERSON; REPS; TEITELBAUM, 2003; CHAN; JIANG; KARUNASEKERA, 2005; COOPER *et al.*, 2006; GRÜNBACHER; HALLING; BIFFL, 2003; HEDBERG, 2004; LANUBILE; MALLARDO, 2003). Algumas características, como o uso do JavaCC (JAVACC, 2008) no trabalho de Cooper e outros (2006), foram igualmente adotadas neste trabalho, tornando viável a construção da CRISTA. Além do trabalho de Cooper e outros (2006), vale destacar os requisitos de gerenciamento dos artefatos, flexibilidade do processo, geração de resultados e resumo das informações presentes no trabalho de Hedberg (2004), que também foram contemplados neste trabalho.

Além do auxílio às funcionalidades da CRISTA, os trabalhos de Pfeiffer e Gurd (2006), Walkinshaw, Roper e Wood (2005), Anderson, Reps e Teitelbaum (2003) e Cooper e outros (2006) destacaram a importância do uso de visualização no processo de compreensão. Dentre esses trabalhos cabe mencionar mais uma vez o trabalho de Pfeiffer e Gurd (2006), no qual foi usada a técnica de visualização *Treemap* para a exploração de grandes quantidades de informação. Essa técnica foi adotada na ferramenta CRISTA por vários motivos citados no trabalho de Pfeiffer e Gurd (2006), dentre eles por permitir uma redução no tempo de exploração do código fonte.

Por fim, mas não menos importante, alguns trabalhos como (CHAN; JIANG; KARUNASEKERA, 2005; GRÜNBACHER; HALLING; BIFFL, 2003; LANUBILE; MALLARDO, 2003; PFEIFFER; GURD, 2006; RUNESON; ANDREWS, 2003) destacaram a necessidade e importância dos estudos experimentais para avaliação e melhoria de suas ferramentas. No caso da CRISTA, aqui proposta, também já foram realizados alguns estudos experimentais com o intuito de avaliá-la.

Como pode ser observado, os trabalhos apresentados neste capítulo deram subsídios para a fundamentação teórica da construção da ferramenta CRISTA. Além disso, eles orientaram o desenvolvimento de diversas funcionalidades e estudos acerca da mesma. O capítulo seguinte discute os aspectos de implementação da ferramenta CRISTA e suas funcionalidades, enquanto que no Capítulo 7 são relatados os estudos experimentais já realizados.

Capítulo 6

A FERRAMENTA CRISTA

Este capítulo descreve a ferramenta CRISTA, apresentando os requisitos que nortearam a criação da ferramenta, as decisões de projeto, sua arquitetura, e um exemplo detalhado de seu funcionamento.

6.1 Considerações Iniciais

A inspeção de software, segundo Lucia e outros (2007), é inerentemente manual, feita em papel, demorada e trabalhosa. O processo associado à inspeção também pode ser muito tedioso e propenso a erros, principalmente quando se fala em inspeção de código, na qual é requerida uma análise completa do software (KOTHARI *et al.*, 2004). Em vista disso, a importância de apoio computacional para aumentar a eficiência do processo de inspeção é evidente (LUCIA *et al.*, 2007).

Assim, essa carência motivou a criação da ferramenta CRISTA (*Code Reading Implemented with Stepwise Abstraction*), uma solução computacional para auxiliar a inspeção de código usando a técnica de leitura *Stepwise Abstraction*. Partindo do princípio que a inspeção de código está intrinsecamente ligada às atividades de compreensão, a CRISTA também fornece subsídios à compreensão de código, facilitando essa tarefa com os recursos disponíveis na ferramenta e com a opção de se fazer essa compreensão de forma sistemática, por meio da *Stepwise Abstraction*.

Como será visto neste capítulo, a ferramenta CRISTA adota uma metáfora visual, a qual facilita a navegação e exploração das informações contidas no código fonte. Conforme exposto anteriormente no Capítulo 3, as técnicas de visualização podem, muitas vezes, dar suporte à compreensão de código, deixando a atividade de inspeção ainda mais efetiva na detecção de defeitos, além de facilitar a própria aplicação da técnica *Stepwise Abstraction*.

A Figura 6.1 representa uma instanciação do modelo de referência de Card e outros (1999 apud NASCIMENTO; FERREIRA, 2005) para o cenário de atuação da ferramenta CRISTA. Pode-se perceber pela figura que os dados brutos são agora o código fonte compilável do software. Esse código é interpretado pela ferramenta CRISTA, que o transforma e mapeia suas estruturas em uma metáfora visual, mostrada pela ferramenta por meio da técnica de visualização *Treemap*. Essa visualização do código fonte ajuda o inspetor na tarefa de compreender o código e, conseqüentemente, em tarefas dependentes da compreensão.

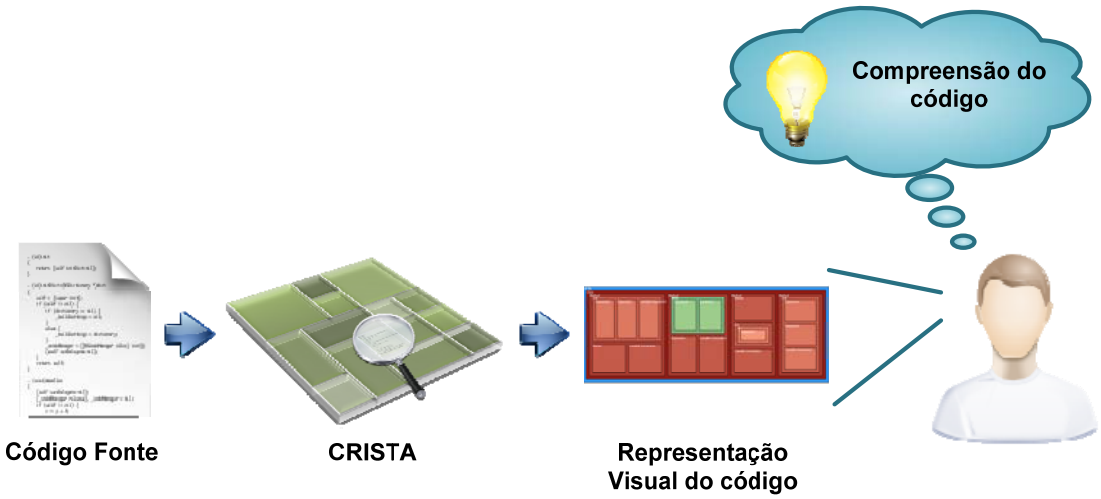


Figura 6.1 - Modelo de referência de Card e outros instanciado para a ferramenta CRISTA

O restante do capítulo está organizado da seguinte forma: na Seção 6.2 são comentados os requisitos que foram estabelecidos na concepção da ferramenta CRISTA; na seção 6.3 são apresentados todos os detalhes técnicos da implementação; na seção 6.4 são descritas detalhadamente as funcionalidades presentes na ferramenta; e na Seção 6.5 apresentam-se as Considerações Finais.

6.2 Definição dos requisitos da ferramenta CRISTA

A identificação dos requisitos da ferramenta CRISTA foi feita a partir da análise do processo de inspeção de software (SAUER *et al.*, 2000) e da análise do processo de aplicação da técnica *Stepwise Abstraction* (LINGER; MILLS; WITT, 1979). Com base nessa análise, foram identificadas as necessidades para que a ferramenta contemplasse esses processos, trazendo benefícios aos usuários.

Foi percebida com essa análise que uma atividade chave que precisaria ser apoiada pela CRISTA era a atividade de compreensão. Isso motivou a procura por outras atividades dependentes da compreensão que poderiam ser apoiadas com a ferramenta. Com isso, além do processo de inspeção, foram levantados requisitos que dessem apoio às atividades de Manutenção de Software, bem como as de Engenharia Reversa e as de Reengenharia.

Uma vez que a maioria dos softwares em manutenção são softwares legados, e que a maioria dos softwares legados são escritos em diferentes linguagens. Os requisitos e a arquitetura da ferramenta CRISTA foram calcados na necessidade básica de dar suporte à qualquer linguagem de programação.

Dessa forma, os requisitos estabelecidos para a ferramenta foram os seguintes:

R1) Dar suporte ao processo de inspeção de software com o uso da técnica *Stepwise Abstraction*:

Como objetivo principal, a CRISTA deveria dar suporte ao processo de inspeção de software e à aplicação da técnica *Stepwise Abstraction*. Isso implica que a CRISTA deveria:

- Separar as instruções do código fonte, respeitando suas hierarquias;
- Possibilitar a realização das abstrações;
- Permitir uma fácil navegação pelo código fonte;
- Auxiliar no registro das discrepâncias, por parte dos inspetores;
- Auxiliar na junção das listas de discrepâncias, por parte do moderador; e
- Auxiliar na geração de relatórios que dessem informações parciais, durante o próprio processo de inspeção, e também informações finais, depois do processo terminado.

R2) Auxiliar na compreensão de código:

Como a inspeção de código é fortemente dependente da sua compreensão, a CRISTA deveria possuir meios de dar suporte a essa compreensão, além de possibilitar usar as abstrações feitas pelos inspetores como subsídio à atividade de compreensão. Dessa forma, a CRISTA deveria:

- Permitir ler o código fonte de tal forma que os trechos já abstraídos estivessem substituídos pelas respectivas abstrações; e
- Permitir colocar as abstrações em um formato estruturado, de maneira que fosse possível transformar o código em um algoritmo, a partir das abstrações realizadas pelos inspetores.

R3) Facilitar a visualização das informações:

Conforme exposto anteriormente, as técnicas de visualização podem dar suporte à compreensão de código, deixando a atividade de inspeção ainda mais efetiva na detecção de defeitos. Dessa forma, a CRISTA deveria ser construída com base em uma metáfora visual, que apoiasse a navegação e exploração das informações contidas no código fonte. Os recursos visuais presentes na CRISTA deveriam auxiliar a:

- Visualizar as instruções contidas no código;
- Visualizar a profundidade de cada instrução, apresentando sua hierarquia;
- Facilitar na identificação de quais instruções já foram abstraídas;
- Possibilitar uma fácil associação entre as discrepâncias e os trechos de código relacionados a elas; e
- Auxiliar o inspetor, provendo um melhor entendimento sobre códigos grandes e complexos.

R4) Auxiliar a atividades de Manutenção de Software:

As atividades de Manutenção de Software, bem como as de Engenharia Reversa e as de Reengenharia, estão calcadas em uma boa avaliação do estado atual do software, isto é, na compreensão do mesmo. Dessa forma, deseja-se aproveitar a infra-estrutura disponível na CRISTA para auxiliar também as atividades de Manutenção de software. Neste caso, a CRISTA deve possibilitar:

- Armazenar o tempo gasto para a abstração do código; e
- Inserir no código as abstrações realizadas na forma de comentário;

R5) Ser independente de linguagem:

Como dito anteriormente, a CRISTA deve fornecer uma estrutura básica que permita uma fácil instanciação para o reconhecimento de outras linguagens. Com isso, a ferramenta poderá apoiar melhor softwares legados, além de possibilitar o seu uso em linguagens que ainda não foram criadas.

6.3 Aspectos de implementação

A ferramenta CRISTA foi desenvolvida utilizando-se o ciclo de vida de prototipação evolutiva. Partindo-se dos requisitos listados anteriormente, foram feitos protótipos para

validar, principalmente, a interface e o leiaute mais adequado à aplicação da técnica de leitura *Stepwise Abstraction*.

As seguintes decisões de projeto foram tomadas:

- O uso da linguagem Java, devido a sua alta popularidade e portabilidade, seguindo os padrões de codificação sugeridos pela Sun Microsystems Inc. (MICROSYSTEMS, 2008). Além disso, a existência de bibliotecas prontas para a criação de metáforas visuais foi um fator decisivo na escolha dessa linguagem.

- Uso da técnica de visualização *Treemap* devido ao caráter hierárquico do código fonte. Essa técnica contribui diretamente na identificação dos elementos a serem abstraídos, além de utilizar de forma eficiente o espaço disponível na tela. Nesse caso, para a criação dessa metáfora visual, foi utilizada a biblioteca Java chamada Prefuse (PREFUSE, 2008). Poderia ter sido utilizada qualquer outra biblioteca java para a criação da metáfora visual, como por exemplo (CHRISTOPHE, 2009) e (JTREEMAP, 2009), ou até mesmo poderia ter sido construída a metáfora visual sem o uso de uma biblioteca. O uso da Prefuse (PREFUSE, 2008) deu-se pela sua maturidade e pelas várias funcionalidades já existentes nessa biblioteca, como por exemplo, o controle de eventos externos tais como zoom, seleção das estruturas, etc..

- Opção pelo ambiente *desktop* visando, principalmente, à inspeção individual. A persistência de dados é feita em arquivos e não em bases de dados. Essa característica provê à CRISTA um ambiente portável, pois as inspeções realizadas por um inspetor podem ser enviadas e abertas por outros inspetores ou pelo moderador da inspeção. Essa propriedade foi incorporada à CRISTA pensando justamente nas inspeções assíncronas, nas quais não é necessário estabelecer nenhuma conexão com qualquer outro computador, o que deixa o inspetor livre para trabalhar onde e quando for desejado.

- Uso do padrão de projeto MVC (*Model-View-Controller*), de maneira a desacoplar as camadas de regras de negócio, de apresentação e de persistência, deixando o código mais legível e aumentando a sua manutenibilidade. A camada VIEW foi implementada usando a biblioteca Swing do Java, além das bibliotecas Jwizard (JWIZARD, 2008) e JavaHelp (JAVAHELP, 2008) para a construção das janelas de *wizard* e do *help* respectivamente.

- Uso do JavaCC (JAVACC, 2008) para a criação de módulos responsáveis por realizar o *parser* das novas linguagens, de forma que a ferramenta pudesse carregar códigos em diferentes linguagens de programação.

A Figura 6.2 representa o processo de instanciação da ferramenta para que ela aceite programas escritos em uma nova linguagem de programação.

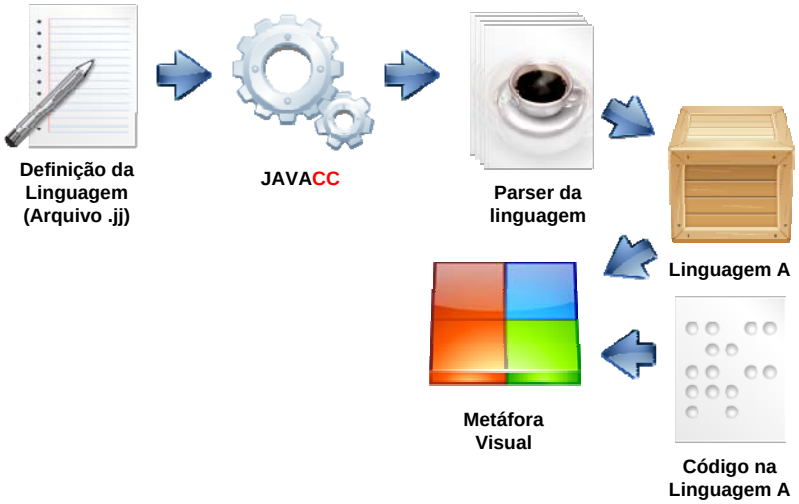


Figura 6.2 - Utilização do JavaCC na arquitetura da ferramenta CRISTA

Assim, para se disponibilizar uma nova linguagem na ferramenta CRISTA, conforme pode ser visto na Figura 6.2, é necessário realizar os seguintes passos:

1) Criar um arquivo de definição da linguagem (.jj) com algumas instruções Java específicas que serão usadas para montar a metáfora visual do código.

2) Executar o JavaCC com esse arquivo, para serem geradas as classes necessárias para realizar o *parser*.

3) Empacotar e armazenar essas classes, dentro de um arquivo jar.

4) Colocar o arquivo jar dentro de um subdiretório da ferramenta CRISTA, que lê esse arquivo dinamicamente e assim, realiza o *parser* da linguagem.

O Apêndice A contém o manual completo de instanciação da ferramenta CRISTA para novas linguagens.

A implementação da CRISTA foi realizada pelo autor deste trabalho usando aproximadamente 254 horas de implementação, considerando-se a construção de protótipos, testes funcionais e documentação. A carga horária está representada na Figura 6.3.

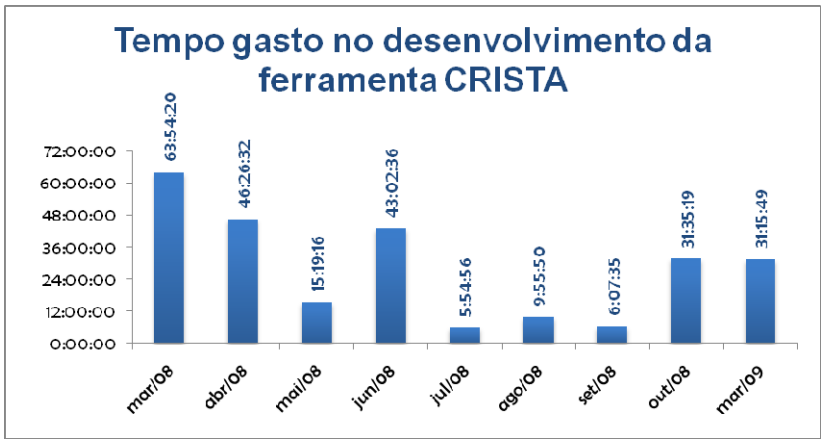


Figura 6.3 - Tempo gasto no desenvolvimento da ferramenta CRISTA mês a mês

Uma pequena parte da solução arquitetural da ferramenta CRISTA pode ser vista na Figura 6.4, em que são representadas as principais classes utilizadas no *parser* do código fonte. Cada arquivo de definição da linguagem (Arquivo .jj) correspondente a uma classe *LanguageParser*. A classe *Inspection* representa o objeto de negócio inspeção e contém, além de outras coisas, todas as instruções contidas no código fonte. Essas instruções são obtidas por um objeto *LanguageParser*, o qual realiza o *parser* do código fonte e devolve todas as instruções contidas no mesmo. Cada instrução é representada pela classe *Statement* e possui, entre outros atributos, uma lista de instruções internas (*InsideStatements*), e um delimitador de bloco (*BlockDelimiter*). A classe *BlockDelimiter* representa um delimitador de bloco e contém as linhas e colunas de início e fim de uma instrução.

Cada linguagem disponível na ferramenta CRISTA (associada a um objeto *LanguageParser*) pode ser associada a um objeto *Documenter*, o qual será usado para gerar a documentação interna do código. A classe *Documenter* é uma classe genérica e, por meio de herança, deve ser implementada para cada linguagem, uma vez que cada linguagem possui uma sintaxe específica para comentários, por meio dos quais será feita a redocumentação do código.

Como pode ser observado na Figura 6.4, a classe *LanguageParser* possui os atributos *actualStatement* e *bDelimiterTemp*, os quais são usados no processo de *parser* do código fonte, não sendo usados em nenhum outro contexto.

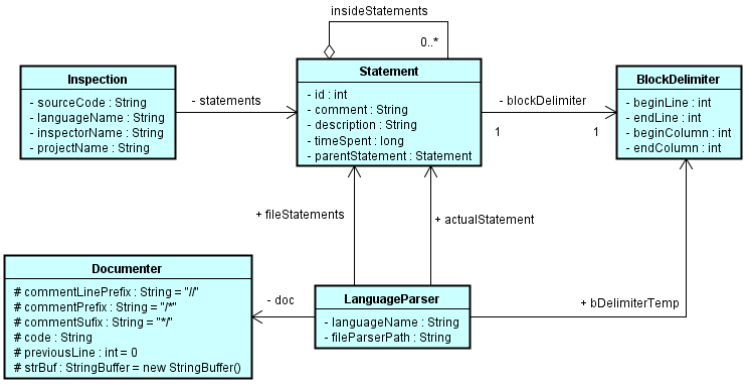


Figura 6.4 - Classes responsáveis por realizar o *parser* do código fonte

Conforme dito anteriormente, a ferramenta CRISTA utiliza a biblioteca de visualização Prefuse (PREFUSE, 2008), a qual fornece um apoio para a construção de metáforas visuais. Conforme pode se observado pela Figura 6.5, foram necessárias as classes *TreeMapDisplay*, *CRISTASquarifiedTreeMapLayout* e *TreemapWheelListener* para a construção da metáfora visual *Treemap*, bem como para o tratamento de eventos da mesma. Para tanto, bastou herdar algumas classes da biblioteca Prefuse, bem como implementar os métodos de forma que a representação visual e seus eventos correspondessem ao desejado.

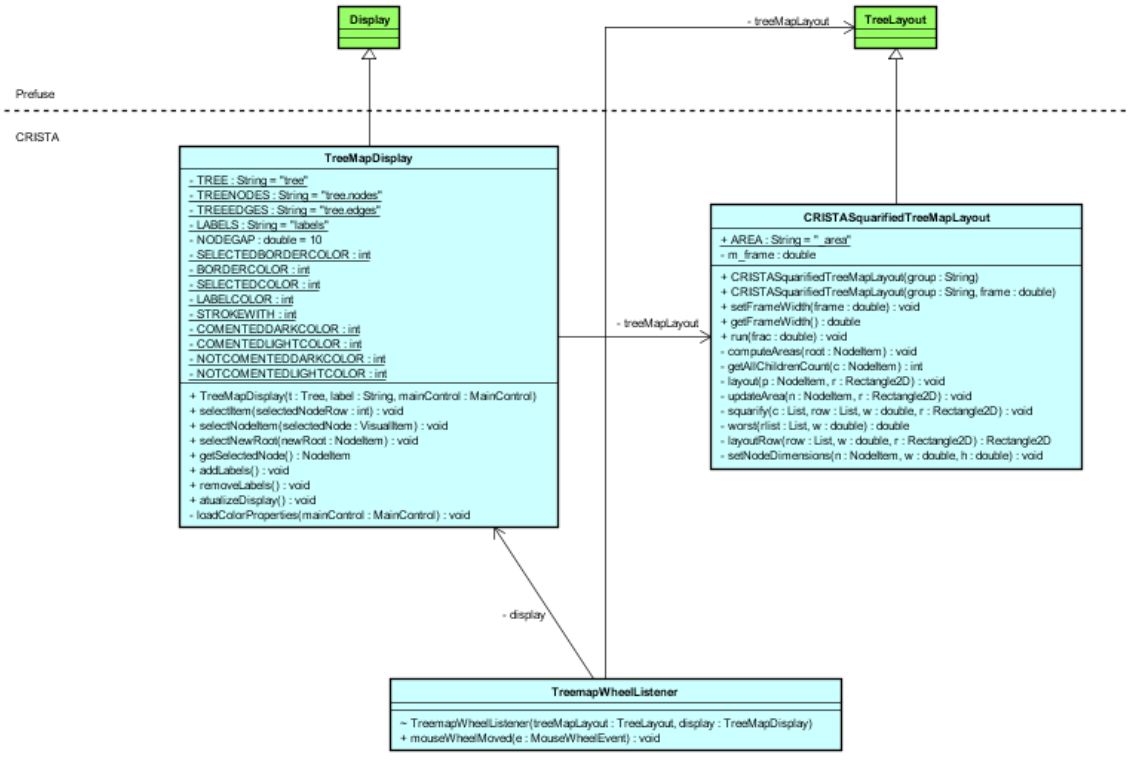


Figura 6.5 - Uso da biblioteca Prefuse para a construção da metáfora visual *Treemap*

A classe *TreeMapDisplay* , que foi herdada da classe *Display*, representa um componente Swing onde será desenha a metáfora visual desejada. Essa classe deve conter todas as regras de negócio e eventos da metáfora visual. Assim, é nessa classe que são definidos como se dá a seleção de um elemento no *Treemap*, quais elementos são visíveis a partir de determinado ponto, bem como as características visuais de rotulação dos elementos e suas cores. Já a classe *CRISTASquarifiedTreeMapLayout*, que possui uma relação de herança com a classe *TreeLayout*, contem as diretrizes para o desenho dos retângulos aninhados, de acordo com a técnica de visualização *Treemap*. Por fim, a classe *TreemapWheelListener* é uma classe separada que foi criada para ser responsável por gerenciar os eventos de zoom manual no *Treemap*.

6.4 Funcionalidades da ferramenta CRISTA

Nesta seção são ilustradas as funcionalidades da ferramenta CRISTA, as quais satisfazem os requisitos estabelecidos inicialmente para ela, apresentadas na Seção 6.2.

6.4.1 Criar uma inspeção

Para iniciar a inspeção de um novo código fonte, o usuário deve criar uma nova inspeção selecionando-se a opção *File/New Inspection*, ou utilizando-se o atalho (Ctrl+N) (Figura 6.6 (A)). Em seguida, a ferramenta apresentará um *wizard* (Figura 6.6 (B)) que guiará a criação da nova inspeção. Nesse *wizard* o usuário deve:

- Informar os dados da inspeção, isto é, o nome do projeto e o nome do inspetor (Figura 6.6 (C)), sendo que essas informações poderão ser alteradas posteriormente;
- Escolher a linguagem usada na inspeção e o arquivo que contém o código fonte (Figura 6.6 (D)); e
- Finalizar o *wizard* (Figura 6.6 (E)).

Ao finalizar o *wizard*, aparecerá a tela principal de inspeção, a qual pode ser vista na Figura 6.8, apresentada mais adiante.

Figura 6.6 - Criando uma nova inspeção

6.4.2 Salvar uma inspeção

Após ter sido aberta uma inspeção, a ferramenta permite que o usuário salve a inspeção a qualquer momento. Para tanto, basta selecionar a opção *File/Save Inspection*, ou então utilizar o atalho (Ctrl + S), conforme pode ser visto na a Figura 6.7.

Caso a inspeção não tenha sido salva ainda, ou caso se queira salvá-la em outro arquivo, deve-se selecionar a opção *File/Save Inspection As* (Ctrl + Shift + S) (Figura 6.7). A partir do momento que qualquer alteração tenha sido feita na inspeção, a ferramenta CRISTA indica essa alteração com um * (asterisco) no título da janela, ao lado no nome da inspeção.

6.4.3 Abrir uma inspeção

A CRISTA foi projetada de forma a permitir que o inspetor realize uma inspeção por etapas, conforme sua conveniência. Dessa forma, o usuário pode salvar e abrir uma inspeção para dar continuidade ao seu trabalho.

Para abrir uma sessão de inspeção já iniciada, deve-se selecionar a opção *File/Open Inspection*, ou o atalho (Ctrl + O). Essas opções podem ser vista na Figura 6.7.

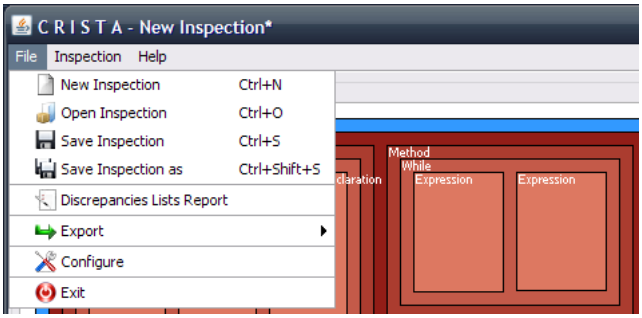


Figura 6.7 – Menu principal da ferramenta CRISTA

6.4.4 Realizar o processo de inspeção com a técnica Stepwise Abstraction

Para realizar uma inspeção é preciso que ela tenha acabado de ser criada ou que uma sessão criada anteriormente seja aberta. Em qualquer um dos casos, a ferramenta apresenta a tela da Figura 6.8, que é composta das seguintes visões:

A: que corresponde à metáfora visual do código fonte a ser inspecionado de acordo com a técnica de visualização *Treemap*.

B: que corresponde à caixa de texto que contém o código fonte a ser inspecionado.

C: que corresponde à uma caixa de texto para o inspetor inserir a abstração de cada instrução do código fonte.

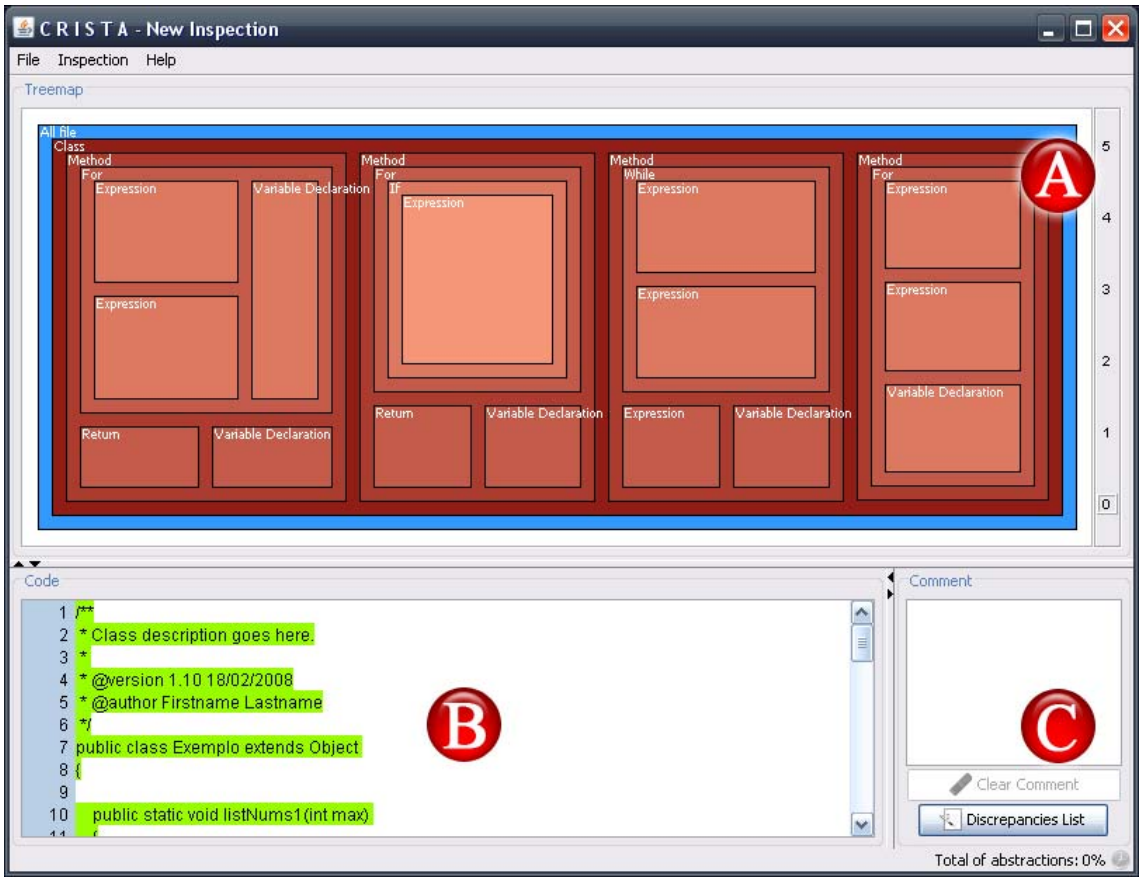
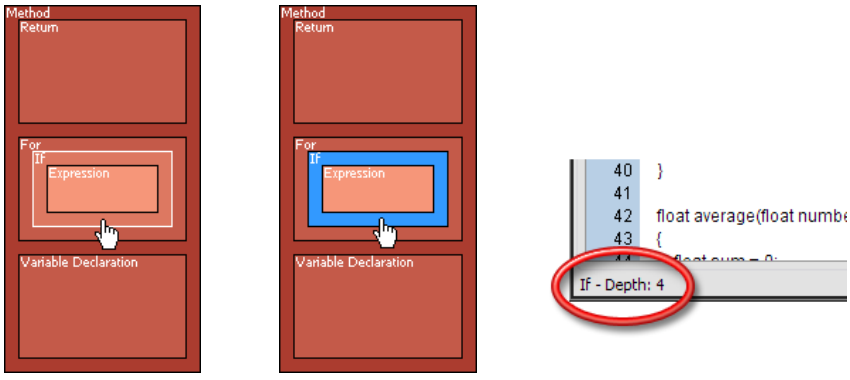


Figura 6.8 - Tela principal da ferramenta CRISTA

O projeto da interface da ferramenta deixa o usuário informado sobre sua localização no código e sobre a inspeção já realizada. Assim, ao passar com o mouse sobre uma caixa do *Treemap* (Figura 6.9 (A)), ela fica destacada, como pode ser visto Figura 6.9 (B). Aparecerão também na barra de status, o tipo da estrutura e sua distância até a raiz (Figura 6.9 (C)).



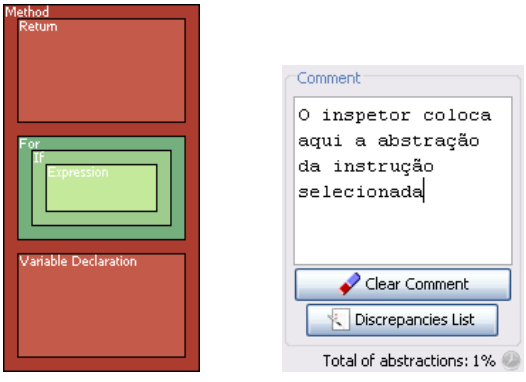
(A) (B) (C)

Figura 6.9 - Selecionando uma instrução

Para selecionar uma instrução, basta clicar com botão esquerdo do mouse em cima da instrução desejada, e a mesma ficará em destaque, como pode ser visto na Figura 6.9 (B). A visualização do código oferecida pelo *Treemap* auxilia o inspetor a dividir o código nas estruturas hierárquicas, de forma que quando selecionada uma instrução no *Treemap*, o trecho de código correspondente é destacado. Todas as cores de destaque usadas pela ferramenta são personalizáveis, e como descrito na Seção 6.4.9, poderão ser alteradas facilmente.

Outra informação que pode ser obtida pela ferramenta CRISTA é o nível de cada instrução. Como pode ser observado na Figura 6.9 (C), ao passar o mouse sobre uma caixa no Treemap, além de ficar destacada (Figura 6.9 (A)), no rodapé aparece o tipo de instrução e o nível no qual a mesma se encontra. Como pode ser observado também na Figura 6.8 (A), a ferramenta apresenta uma barra lateral, à direita do *Treemap*, o nível que se encontra a instrução selecionada. Nessa barra, é possível ver também o nível mais interno que se pode chegar a partir da instrução selecionada.

A CRISTA foi projetada para auxiliar ao máximo a realização das abstrações requeridas pela técnica *Stepwise Abstraction*. Para tanto, o usuário deve selecionar a instrução desejada e escrever sua abstração na caixa de comentário. Essa caixa fica à direita do código e pode ser vista na Figura 6.8 (C) e em destaque na Figura 6.10 (B). À medida que o código é abstraído, conforme pode ser visto na Figura 6.10 (B), a porcentagem no canto inferior direito da tela é atualizada, de forma a informar o usuário sobre o progresso da inspeção. Além disso, como podem ser vistas na Figura 6.10 (A), as estruturas no *Treemap* são coloridas à medida que o código é abstraído. Dessa forma, o usuário é informado visualmente sobre o andamento da inspeção. Nas Figuras 6.8 e 6.11 pode ser visto um código ainda não abstraído e um código todo abstraído, respectivamente.



(A) (B)

Figura 6.10 - Abstraindo o código

6.4.5 Opções da inspeção

De forma a facilitar o trabalho do usuário, a ferramenta CRISTA permite que ele possa voltar ou refazer a edição de um comentário. Para tanto, basta acessar as opções *Inspection/Undo Comment* (Crtl+Z) e *Inspection/Redo Comment* (Crtl+Y), respectivamente (Figura 6.11 (A)). Pelo botão *Clear Comment* (Figura 6.10 (B)) o usuário pode excluir de uma única vez todo o comentário referente a uma abstração.

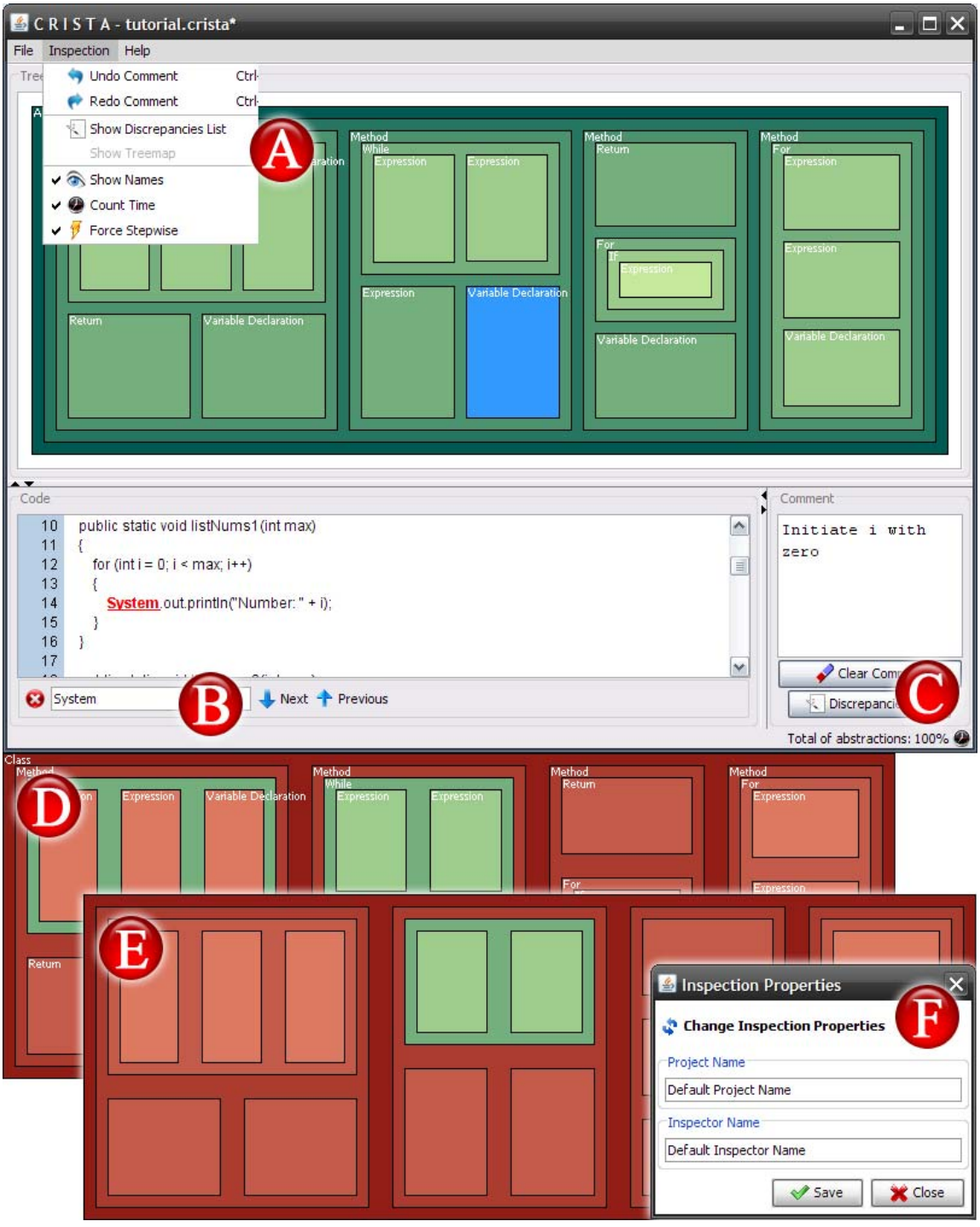


Figura 6.11 - Opções disponibilizadas ao inspetor

Como pode ser visto na Figura 6.8, a CRISTA apresenta um rótulo para cada instrução na metáfora visual, de acordo com o seu tipo. Entretanto, esses rótulos podem ser ocultados por meio da opção *Inspection/Show Names* (Figura 6.11 (A)).

Uma vez que a opção de mostrar nomes estiver habilitada, cada caixa no *Treemap* terá um rótulo indicando o tipo da instrução ou do bloco de instruções, como na Figura 6.11 (D). Entretanto, caso a opção esteja desabilitada, o *Treemap* ficará sem a indicação do que representa cada caixa, como pode ser visto na Figura 6.11 (E).

A CRISTA possibilita ao usuário pesquisar textos contidos no código fonte. Para tanto, o usuário deve acessar a opção *Inspection/Find* ou usar o atalho (Ctrl + F). Assim, é disponibilizado um campo abaixo do código, no qual o usuário insere o texto a ser procurado (Figura 6.11 (B)). Outro benefício oferecido pela ferramenta CRISTA durante a realização de inspeções de código, é que ela possibilita ao usuário armazenar quanto tempo foi gasto para abstrair cada instrução do código. Essa funcionalidade é acionada pela opção *Inspection/Count Time* (Figura 6.11 (A)).

Ao se habilitar o relógio para contar o tempo, aparecerá um ícone de um relógio no canto inferior direito da tela, conforme a Figura 6.11 (C). Isso indicará que o relógio está habilitado, e então, para cada instrução, será armazenado o tempo que ela ficar selecionada. Dessa forma, ao final da inspeção tem-se o tempo de abstração de cada instrução, inclusive de suas instruções internas.

Essa opção permite ter um parâmetro para avaliar se a inspeção foi realmente feita, ou não. Essa informação poderá ser extraída no relatório de tempo gasto, que será descrito mais adiante neste capítulo.

Por padrão, a ferramenta CRISTA força com que as abstrações sejam feitas seguindo a técnica *Stepwise Abstraction*, ou seja, as abstrações de estruturas externas só são habilitadas depois que as internas já foram abstraídas (Figura 6.11 (E)). Entretanto, a CRISTA permite desabilitar a técnica *Stepwise Abstraction*, possibilitando ao usuário abstrair as instruções em qualquer ordem. Para tanto, deve ser desmarcada a opção *Inspection/Force Stepwise* (Figura 11 (A)), e, conforme pode ser visto na Figura 6.11 (D), a ferramenta CRISTA permite ao usuário abstrair os blocos externos (verde) antes dos blocos internos (vermelhos).

A CRISTA possibilita ainda que o usuário mude as propriedades da inspeção definidas inicialmente. Para tanto, basta escolher a opção *Inspection/Change Properties* (Figura 6.11 (A)), que abre a janela mostrada em Figura 6.11 (F). Essas propriedades são usadas unicamente para fins de documentação nos relatórios gerados pela ferramenta.

6.4.6 Registrar discrepâncias

Para apoiar uma etapa muito importante de inspeção, que é o registro de discrepâncias, a ferramenta CRISTA oferece ao usuário uma maneira rápida de relatá-las. Ao clicar no botão *Discrepancies List* (Figura 6.10 (B)), será apresentada uma tela listando as discrepâncias assinaladas (Figura 6.12 (A)), juntamente com o código fonte (Figura 6.12 (B)).

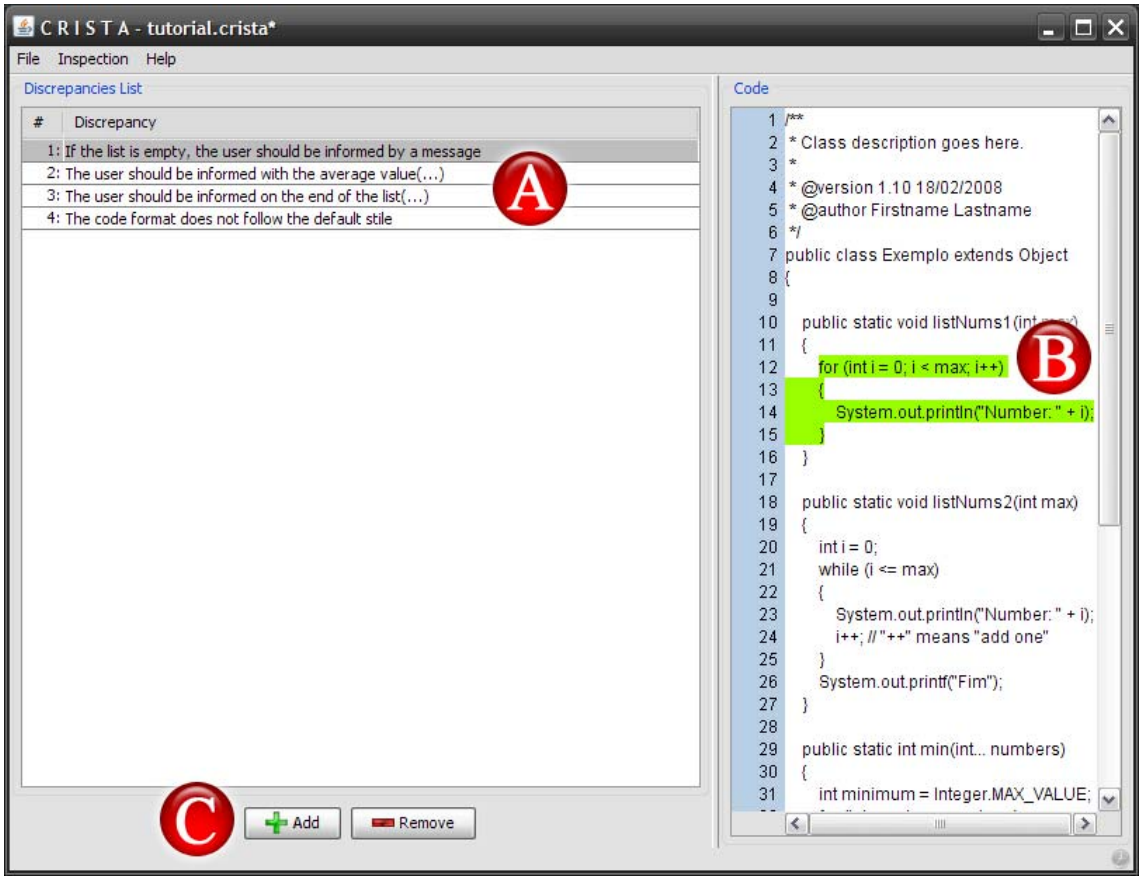


Figura 6.12 - Tela de discrepâncias

Para adicionar uma nova discrepância, basta clicar no botão *Add* (Figura 6.12 (C)), que será apresentada uma janela conforme a Figura 6.10, na qual o usuário poderá descrever a discrepância.

Figura 6.13 – Adicionando uma nova discrepância

Ao clicar no botão OK, a discrepância é adicionada à lista de discrepâncias (Figura 6.12 (A)). Para alterar o texto da discrepância, o usuário pode dar um duplo clique sobre a mesma. Assim, como mostra a Figura 6.14, o campo com o relato da discrepância se torna editável.

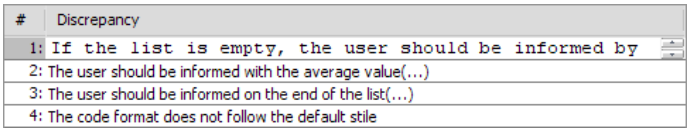


Figura 6.14 – Editando uma discrepância

A CRISTA possibilita ao usuário vincular uma discrepância a um trecho de código no qual ele considera que esteja a fonte do problema, facilitando a identificação dessa discrepância por parte dos outros usuários. Para tanto, deve-se selecionar uma discrepância e marcar no código o trecho correspondente (Figura 6.12 (B)).

6.4.7 Reunir listas de discrepâncias

No processo de inspeção, uma vez que cada inspetor já relatou as suas discrepâncias, cabe ao moderador da inspeção reuni-las. A CRISTA apóia essa atividade do moderador por meio da opção *File/Discrepancies Lists Report* (Figura 6.7).

Um exemplo da tela de junção das listas de discrepâncias pode ser visto na Figura 6.15. Essa tela contém:

- Duas barras de botões (partes A e E);
- Uma lista final de todas as discrepâncias já aceitas (parte B);
- Uma lista que exibe as discrepâncias encontradas por cada inspetor (parte C); e
- Um campo em que é exibido o código fonte (parte D).

Para reunir as listas de discrepâncias dos inspetores, o moderador deve abrir uma das listas de cada vez e indicar quais discrepâncias permanecem na lista final. Após avaliar as listas de todos os inspetores, o moderador deve salvar a lista final de discrepâncias, e gerar um relatório contendo todas as discrepâncias analisadas.

Quando é carregada a inspeção de um inspetor, todas as discrepâncias relatadas por ele aparecerão na lista da direita (Figura 6.15 (C)). Ao se escolher uma discrepância, se ela estiver vinculada a um trecho de código, esse é destacado na região D que apresenta o código fonte (Figura 6.15 (D)). Clicando-se com o botão direito do mouse sobre cada discrepância, aparecerá uma caixa com as opções de: adicionar, rejeitar e marcar a discrepância como duplicada (Figura 6.15 (G)).

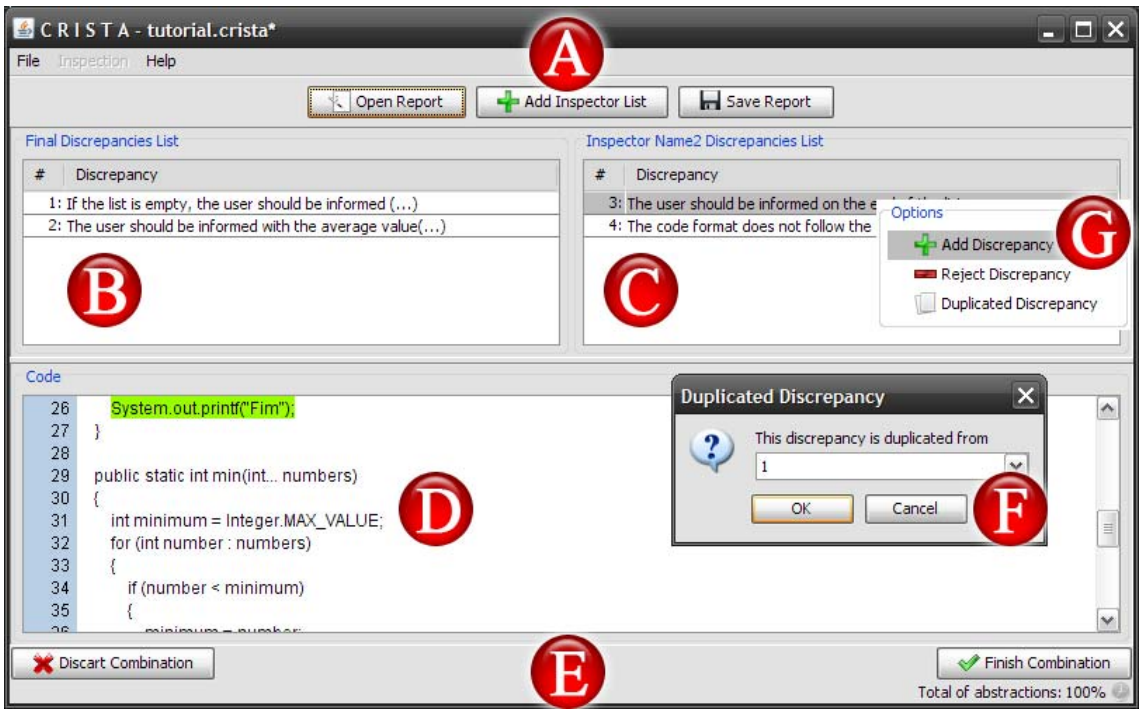


Figura 6.15 - Tela de junção das discrepâncias

No caso de o moderador concordar com a discrepância, ele deve clicar no botão *Add Discrepancy* (Figura 6.15 (G)), o que faz passar automaticamente para a lista final de discrepâncias (Figura 6.15 (B)). Caso o moderador não concorde com a discrepância, basta selecionar a opção *Reject Discrepancy* (Figura 6.15 (G)) e a discrepância é removida da lista final de discrepâncias e é armazenada em uma lista a parte.

Ao abrir a lista de discrepância de outro inspetor e esta possuir uma discrepância que já tenha sido aceite e transferida para a lista final, o moderador deve selecionar a opção *Duplicated Discrepancy* (Figura 6.15 (G)). Escolhendo essa opção, aparecerá uma janela (Figura 6.15 (F)) na qual o moderador informa com qual discrepância da lista final essa

coincide. Após ser escolhida, a discrepância é removida da lista de discrepâncias do inspetor.

Após serem tratadas todas as discrepâncias do inspetor, o moderador deve repetir o mesmo processo para as outras listas dos outros inspetores. No caso de não haver mais listas a serem tratadas, o moderador deve clicar no botão *Finish Combination* (Figura 6.15 (E)), o qual dará início à exportação do relatório final de discrepâncias. Para essa exportação, o usuário contará com a ajuda de um *wizard* (Figura 6.16 (A)) no qual o inspetor deve:

- Selecionar as informações opcionais que vão aparecer no relatório (Figura 6.16 (B)). As informações opcionais são: informar o nome do autor de cada discrepância, relatar também as discrepâncias que foram rejeitadas, e por fim, relatar também as que foram marcadas como duplicadas;
- Escolher onde o arquivo será salvo (Figura 6.16 (C)); e
- Finalizar o wizard (Figura 6.16 (D)).

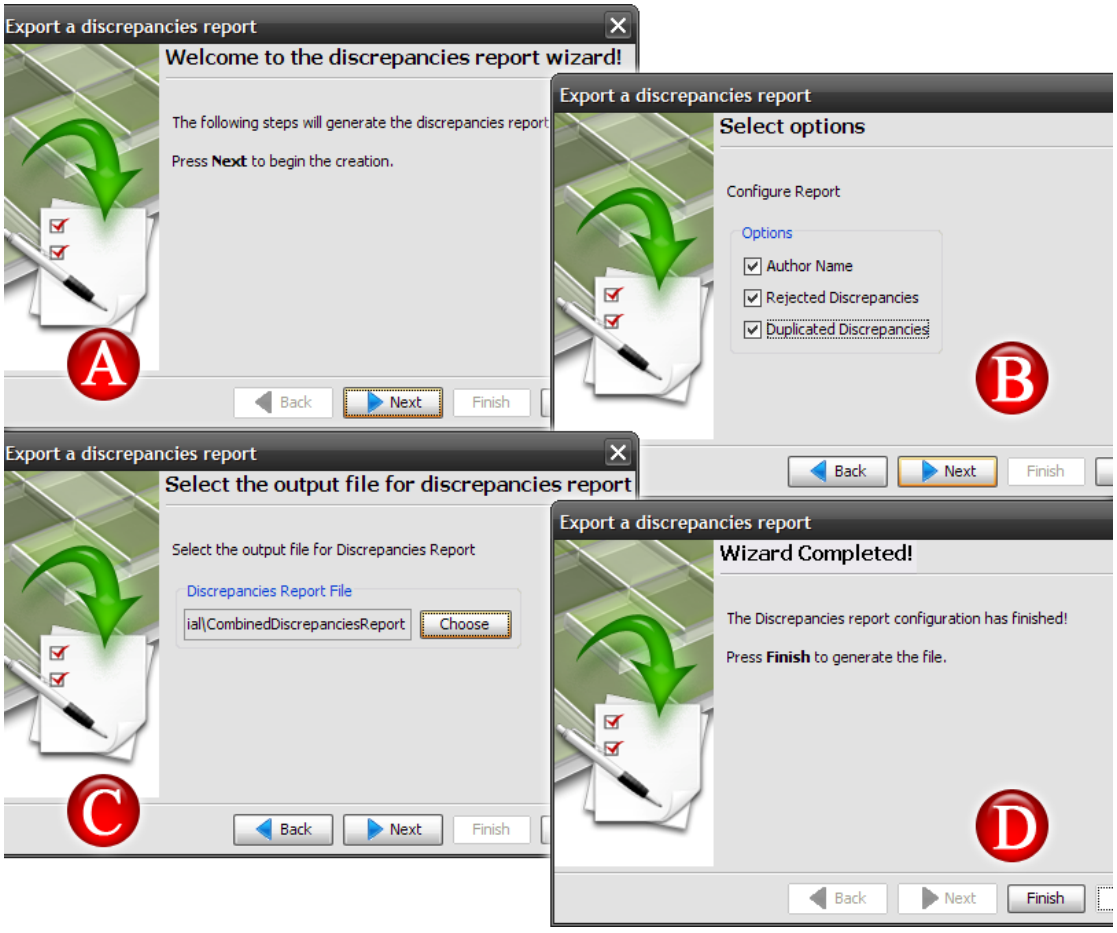


Figura 6.16 - Passos para junção das listas de discrepâncias

O relatório final de discrepâncias, que pode ser visto na Figura 6.17, é um arquivo HTML e contem a lista de todas as discrepâncias. Vinculada a cada discrepância, está o trecho de código que o inspetor associou no momento de criação da discrepância.

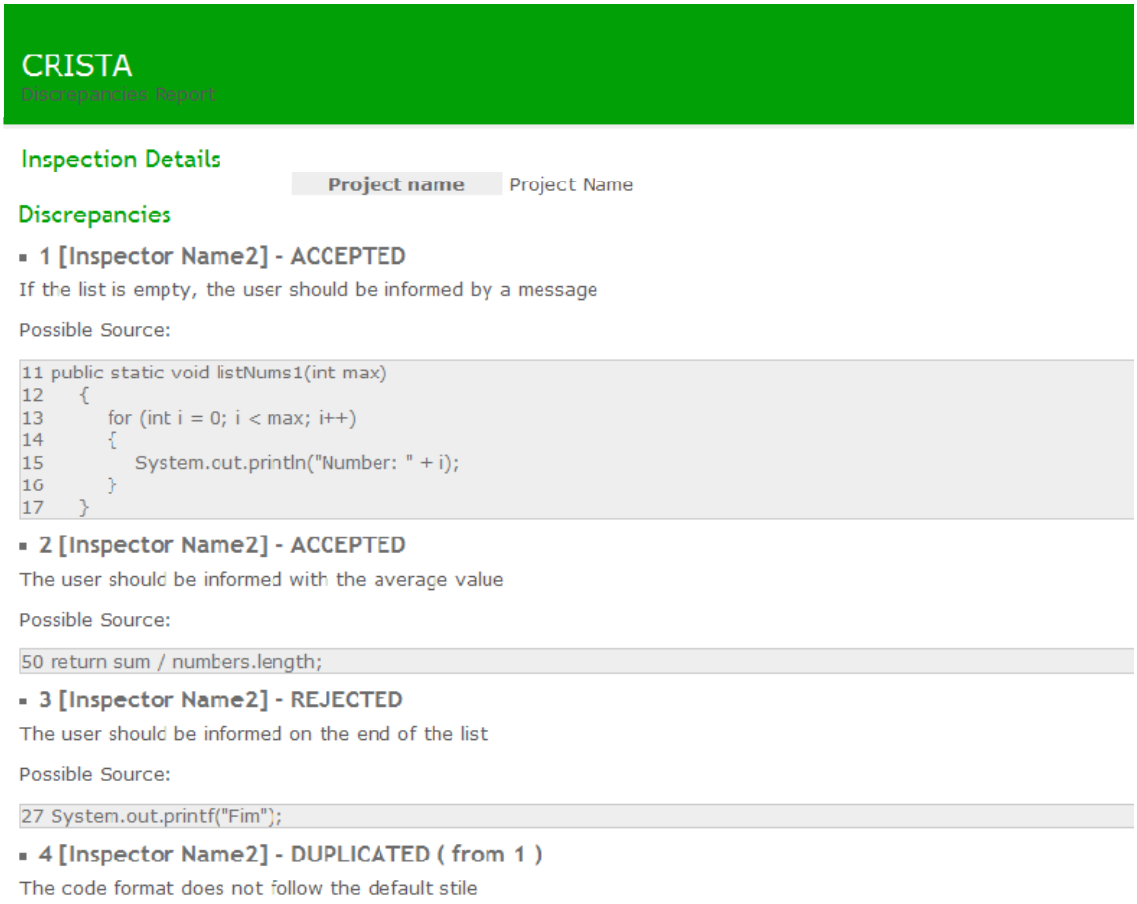


Figura 6.17 - Relatório de discrepâncias encontradas

6.4.8 Gerar relatórios

Além do relatório contendo a lista final de discrepâncias (Figura 6.17), a ferramenta CRISTA possibilita gerar vários outros tipos de relatórios selecionando-se *File/Export* (Figura 6.18). Esses relatórios são:

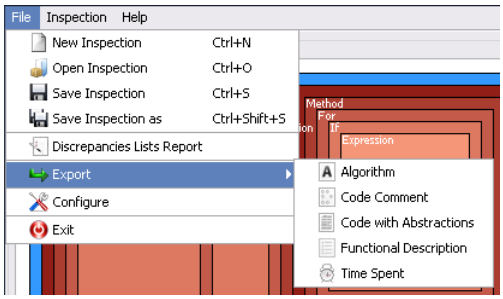


Figura 6.18 - Exportando relatórios

- **Código com abstrações**

No intuito de auxiliar o usuário a realizar as abstrações, a ferramenta CRISTA possibilita gerar um relatório com o código, substituindo os trechos de código já abstraídos pelas suas respectivas abstrações. Dessa forma, o usuário pode ler o código mais facilmente, usando as próprias abstrações para facilitar o entendimento daquilo que ainda não está abstraído. Para exportar o código com abstrações, basta selecionar a opção *File/Export/Code with Abstractions* (Figura 6.18), que aparecerá um wizard (Figura 6.19 (B), (C) e (D)) para guiar a exportação. Pelo wizard é possível escolher o local em que será salvo esse relatório. Um exemplo desse código com pode ser visto na Figura 6.19 (E).

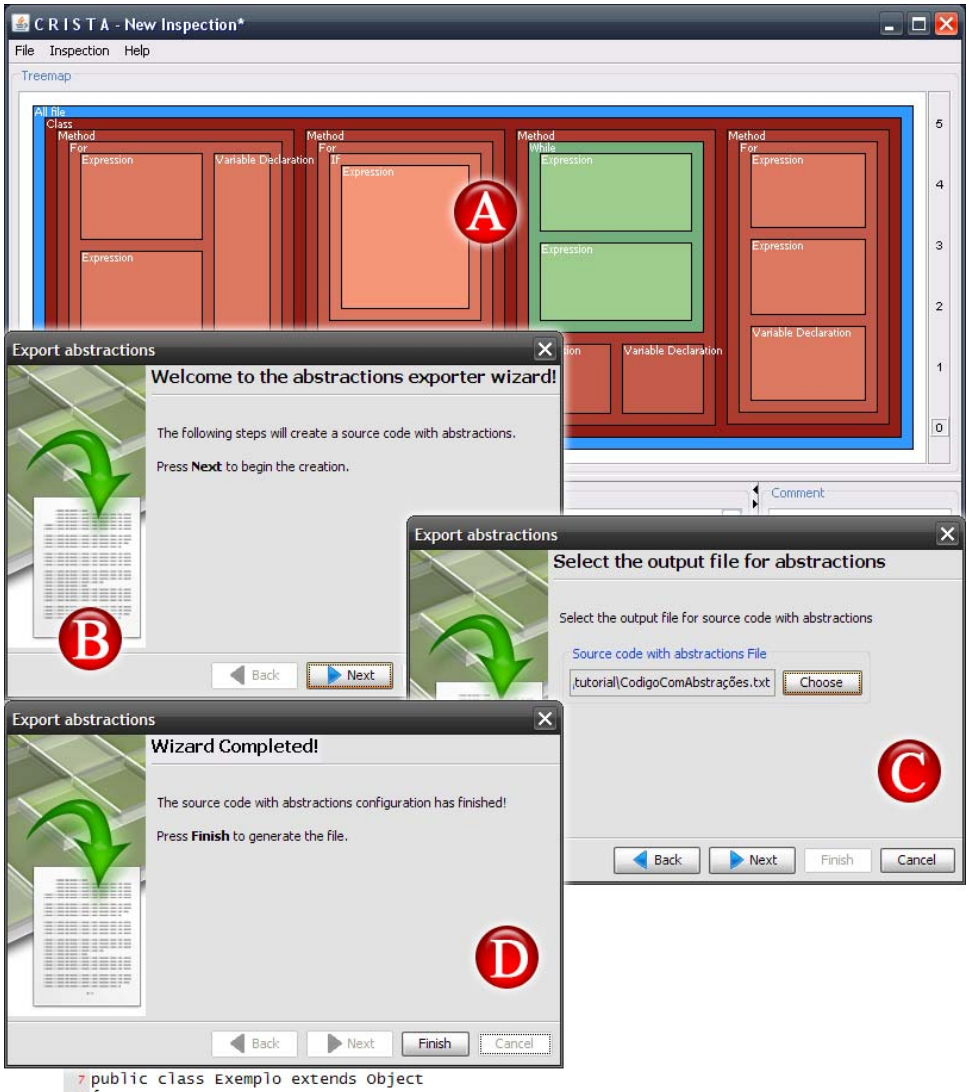
- **Abstrações em formato de algoritmo**

Uma vez que o código foi todo abstraído, com as abstrações escritas na forma de "português estruturado", a CRISTA possibilita exportar o código substituído por essas abstrações, de forma que o mesmo se converta em um algoritmo, facilitando a sua compreensão. Nesse caso, deve ser selecionada a opção *File/Export/Algorithm* (Figura 6.18).

Por meio de um *wizard* (Figura 6.20 (A), (B), (C) e (D)) a ferramenta CRISTA conduz o usuário, para a geração desse relatório:

- Selecionar quais tipos de instruções serão substituídos pelas abstrações (Figura 6.20 (B));
- Escolher onde o arquivo vai ser salvo (Figura 6.20 (C)); e
- Finalizar o wizard (Figura 6.20 (D)).

Na Figura 6.20 (E), pode ser visto um exemplo de algoritmo exportado, onde foram selecionadas todas as instruções, excetuando-se as Classes e Métodos.




```
7  public class Exemplo extends Object
8  {
9
10     public static void listNums1(int max)
11     {
12         for (int i = 0; i < max; i++)
13         {
14             System.out.println("Number: " + i);
15         }
16     }
17
18     public static void listNums2(int max)
19     {
20         int i = 0;
21         Enquanto i é menor que max, imprime o valor de i e incrementa o seu valor.
22         System.out.printf("Fim");
23     }
24
25     public static int min(int... numbers)
26     {
27         int minimum = Integer.MAX_VALUE;
28         for (int number : numbers)
29         {
30             if (number < minimum)
31             {
32                 minimum = number;
33             }
34         }
35         return (minimum);
36     }
37
38     float average(float numbers[])
39     {
40         float sum = 0;
41         for (int i = 0; i < numbers.length; i++)
42         {
43             sum += numbers[i];
44         }
45         return sum / numbers.length;
46     }
```

E

Figura 6.19 - Gerando código com abstrações

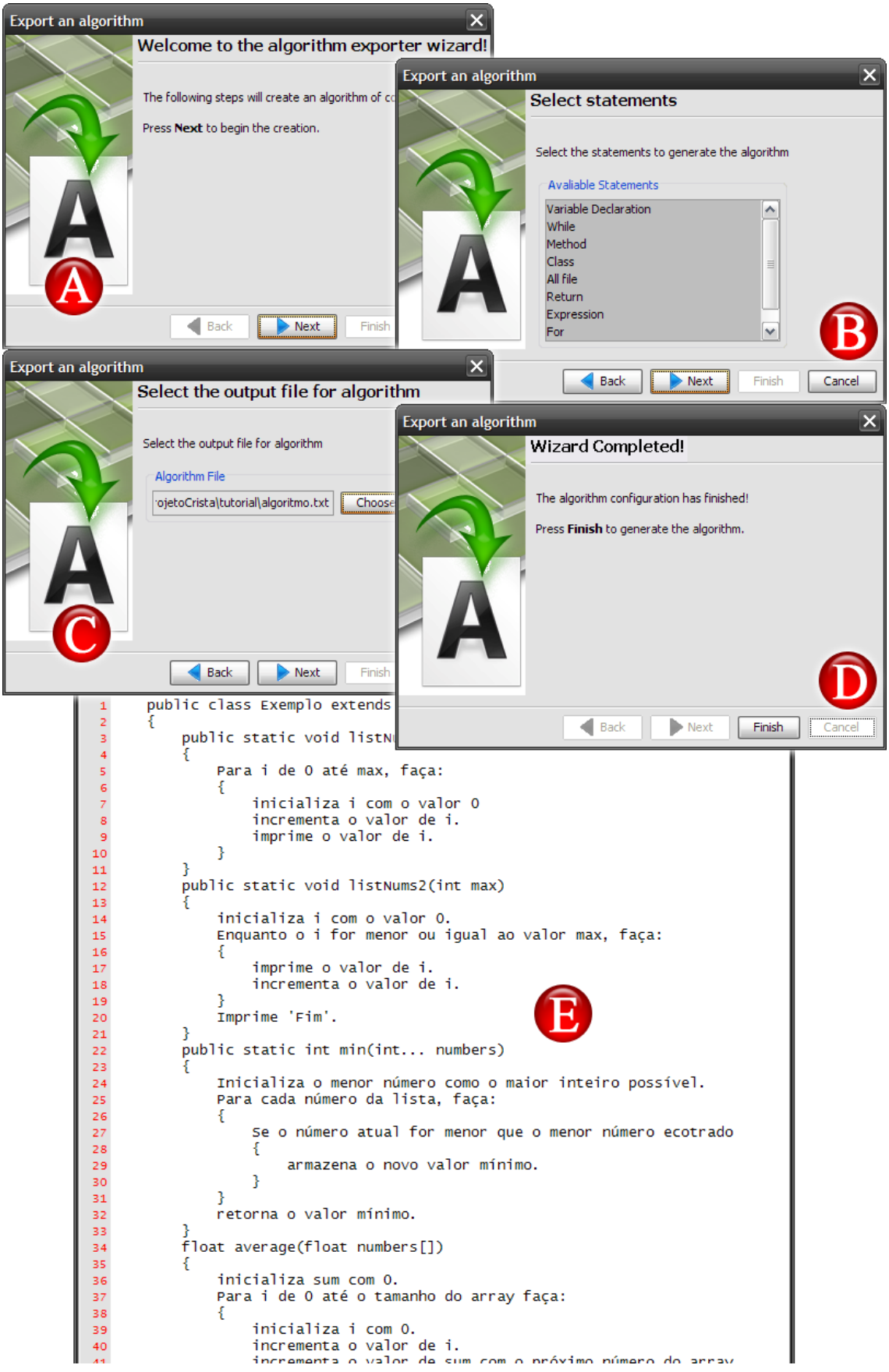


Figura 6.20 - Gerando abstrações em formato de algoritmo

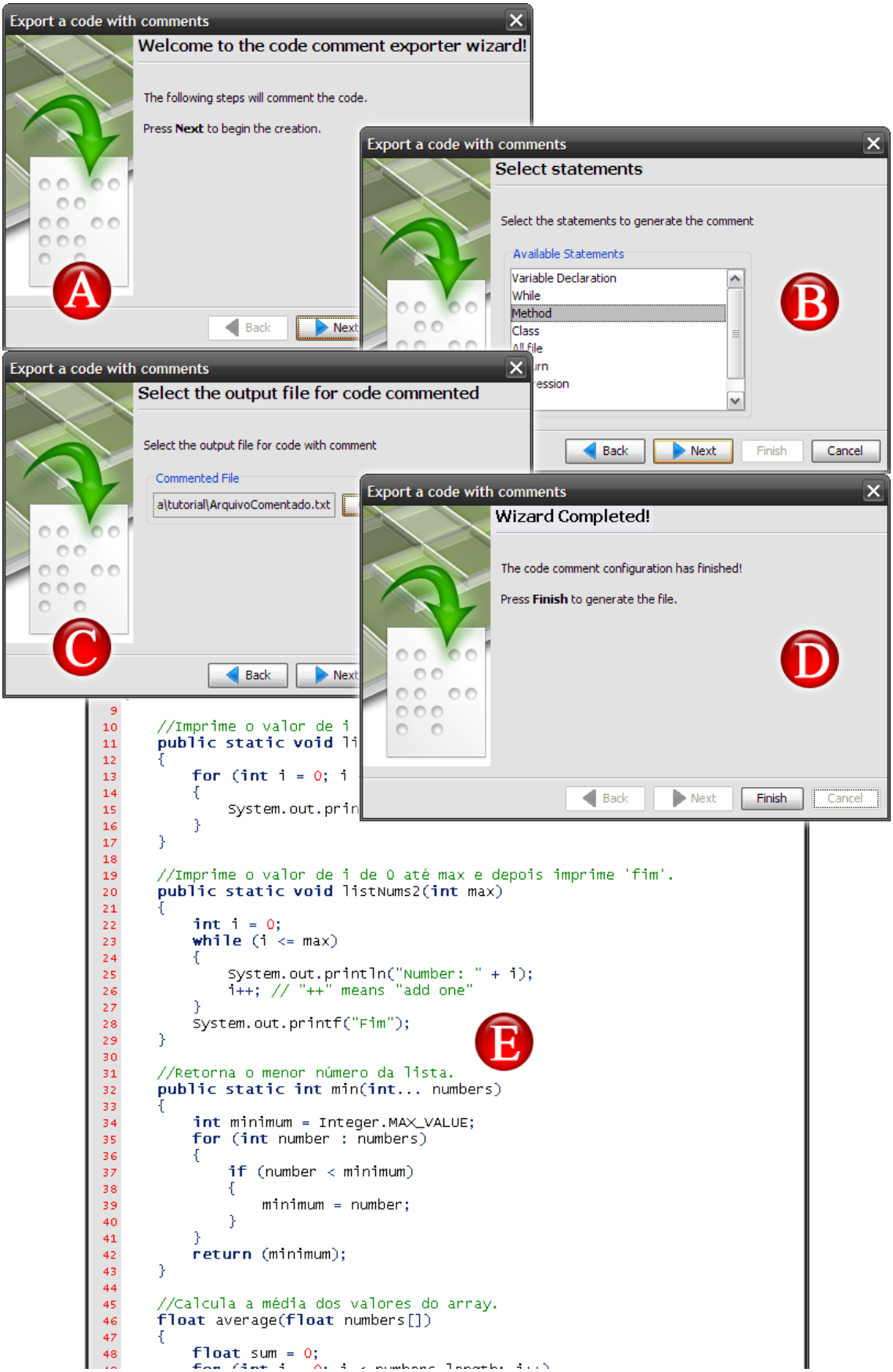
- **Código comentado**

Outro relatório fornecido pela CRISTA é composto pelo código do programa no qual se colocam as abstrações feitas pelo usuário no formato de comentário. Assim, ao se fazer uma inspeção tem-se o código comentado. Para exportar o código comentado, basta selecionar a opção *File/Export/Code Comment* (Figura 6.18 (A)).

Por meio de um *wizard* (Figura 6.21 (A), (B), (C) e (D)), a ferramenta CRISTA conduz o usuário para a exportação dos comentários no código. Nesse *wizard* o inspetor deve:

- Selecionar todos os tipos de instruções para os quais se deseja colocar um comentário na linha anterior (Figura 6.21 (B));
- Escolher onde o arquivo vai ser salvo (Figura 6.21 (C)); e
- Finalizar o wizard (Figura 6.21 (D)).

Na Figura 6.21 (E) tem-se um exemplo de um código em que foram inseridos os comentários somente para os métodos. Essa opção de inserir as abstrações novamente no código na forma de comentários, ajuda na compreensão e deixa-o mais legível. Dessa forma, a ferramenta CRISTA possibilita uma maneira sistemática e formal de redocumentação de código.




```
9
10  //Imprime o valor de i
11  public static void li
12  {
13      for (int i = 0; i
14      {
15          System.out.prin
16      }
17  }
18
19  //Imprime o valor de i de 0 até max e depois imprime 'fim'.
20  public static void listNums2(int max)
21  {
22      int i = 0;
23      while (i <= max)
24      {
25          System.out.println("Number: " + i);
26          i++; // "++" means "add one"
27      }
28      System.out.printf("Fim");
29  }
30
31  //Retorna o menor número da lista.
32  public static int min(int... numbers)
33  {
34      int minimum = Integer.MAX_VALUE;
35      for (int number : numbers)
36      {
37          if (number < minimum)
38          {
39              minimum = number;
40          }
41      }
42      return (minimum);
43  }
44
45  //Calcula a média dos valores do array.
46  float average(float numbers[])
47  {
48      float sum = 0;
```

Figura 6.21 - Comentando o código

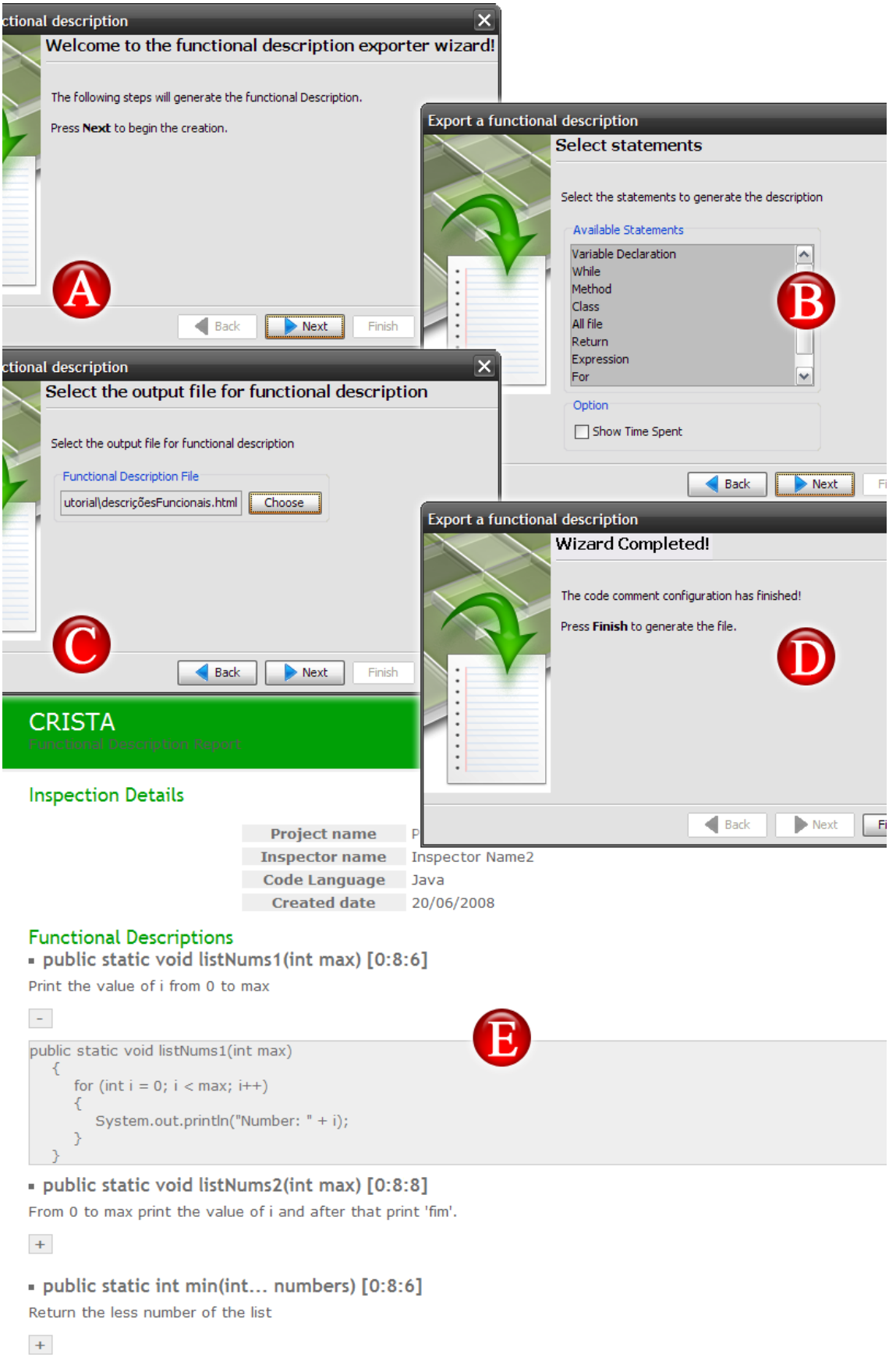
- **Descrições funcionais**

Além de inserir os comentários de volta no código, a ferramenta CRISTA oferece outro meio de apoiar a redocumentação: exportar as descrições funcionais do código, gerando um documento HTML que contém as descrições de cada parte do sistema.

Uma vez realizadas todas as abstrações, o usuário pode exportar as descrições funcionais do código, acessando a opção *File/Export/Fucntional Descriptions* (Figura 6.18). Para a exportação do relatório de descrições funcionais, o usuário contará com a ajuda de um *wizard* (Figura 6.22 (A)) no qual ele deve:

- Selecionar os tipos de instruções para os quais serão exportadas as respectivas descrições (Figura 6.22 (B));
- Escolher se será apresentado junto com as instruções o tempo que foi gasto para abstrair a mesma (Figura 6.22 (B));
- Escolher onde o arquivo vai ser salvo (Figura 6.22 (C)); e
- Finalizar o wizard (Figura 6.22 (D)).

O relatório com as descrições funcionais (Figura 6.22 (E)) é um arquivo HTML, no qual pode ser visto o enunciado de cada instrução, seguido de suas descrições funcionais. Ao se clicar no botão “+” de cada instrução é apresentado o trecho de código correspondente e, caso tenha sido marcada a opção *Show Time Spent* (Figura 6.22 (B)), o tempo para se abstrair a instrução aparecerá no final de seu enunciado.




```
public static void listNums1(int max)
   {
      for (int i = 0; i < max; i++)
      {
         System.out.println("Number: " + i);
      }
   }
```

Figura 6.22 - Gerando o relatório de descrições funcionais

- **Tempo gasto nas abstrações**

Um relatório muito importante para efeito de manutenção fornecido pela CRISTA é o relatório de tempo gasto nas abstrações. Esse relatório é gerado escolhendo a opção *File/Export/Time Spent* (Figura 6.18). Ao ser selecionada essa opção, é mostrada a tela de tempo gasto (Figura 6.23). Nessa tela é possível ver um gráfico com os tempos que foram gastos para abstrair cada instrução (Figura 6.23 (A)), bem como o trecho de código correspondente a essas instruções (Figura 6.23 (D)).

O gráfico visto na Figura 6.23 (A) apresenta, para cada instrução, dois tempos: o tempo absoluto (tempo gasto para a abstração da instrução), e o tempo acumulado (tempo gasto para a abstração de suas instruções internas). A caixa de seleção com a escala do gráfico (Figura 6.23 (B)) permite apresentar o tempo gasto em segundos ou minutos.

O gráfico foi projetado para possuir fácil navegação. Assim, ao clicar com o botão esquerdo do mouse em uma barra no gráfico, é apresentado um gráfico para as instruções internas do bloco associado à barra selecionada. Além disso, para voltar e ver os tempos das instruções um nível acima, basta clicar no botão de voltar (Figura 6.23(C)), ou clicar com o botão direito do mouse em qualquer ponto do gráfico. Também na Figura 6.23 (C), é possível aumentar ou diminuir a altura do gráfico através dos botões “+” e “–“. A Crista ainda permite salvar o gráfico como imagem no formato PNG. Para tanto, basta clicar no botão de salvar (Figura 6.23 (C)).

Observando-se esse relatório é possível identificar os trechos de código que consomem mais tempo de abstração. De maneira geral, esses trechos mais demorados podem ser encarados como os de maior complexidade e/ou menor legibilidade. Dessa forma, a CRISTA ajuda a identificar esses trechos, os quais podem receber um tratamento especial para o aumento da sua legibilidade, facilitando o processo de manutenção.

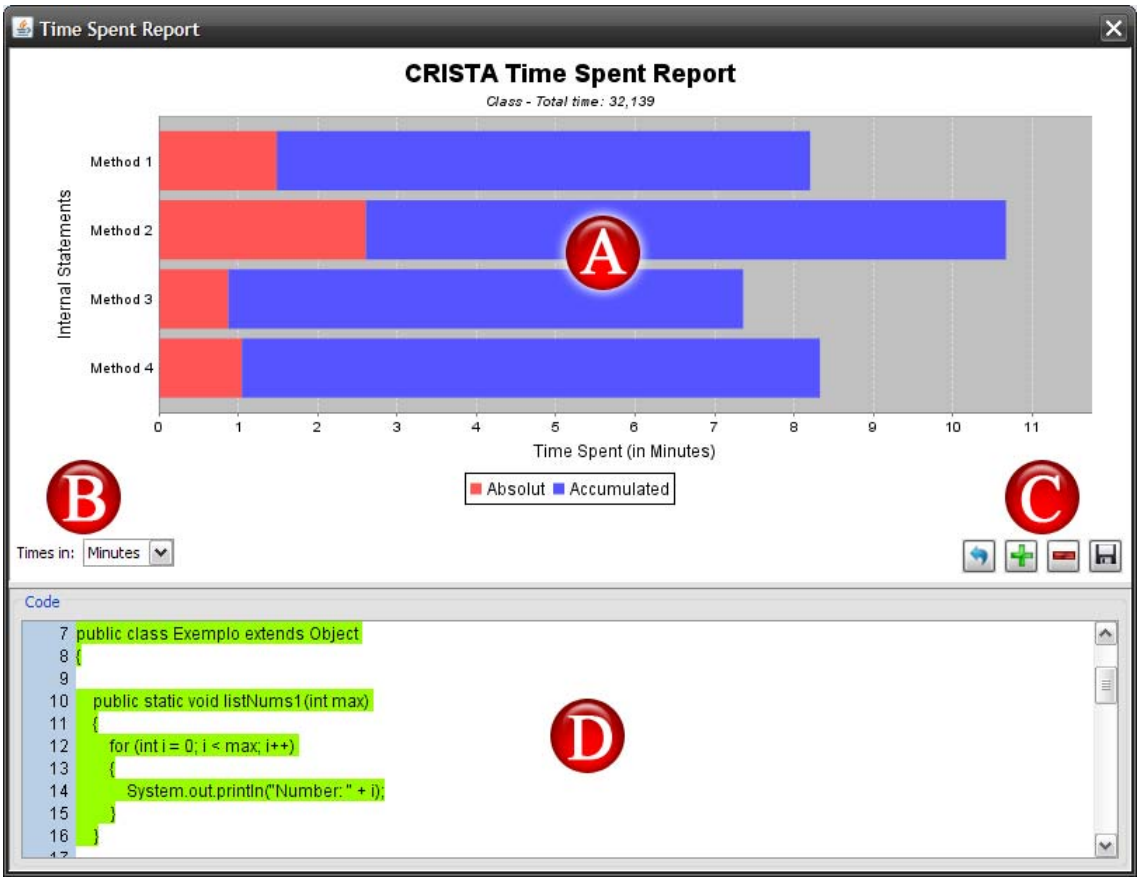


Figura 6.23 - Relatório de tempo gasto

6.4.9 Personalizar a ferramenta

A ferramenta CRISTA foi desenvolvida para ser facilmente personalizada. O usuário tem a opção de, além de mostrar ou esconder os rótulos de cada estrutura no *Treemap* (Figura 6.11 (A)), selecionar as cores que serão utilizadas na metáfora visual e na seleção do trecho de código. Esses atributos são alterados na janela de configuração da ferramenta, a qual é mostrada ao acessar a opção *File/Configure* (Figura 6.24 (A)).

Na guia *Personal* da janela de configuração (Figura 6.24 (B)), é possível definir o nome padrão para o usuário e para os projetos criados pela CRISTA. Já na guia *Languages* (Figura 6.24 (C)), é possível ver e alterar o diretório padrão no qual estão os módulos de reconhecimento das linguagens. É possível ver também todas as linguagens reconhecidas pela CRISTA.

Como mostra a Figura 6.24 (D), na guia *Colors* é possível alterar as cores que são exibidas na visualização do *Treemap* e na visualização do código, o que deixa o usuário livre para escolher todas as cores utilizadas na ferramenta.

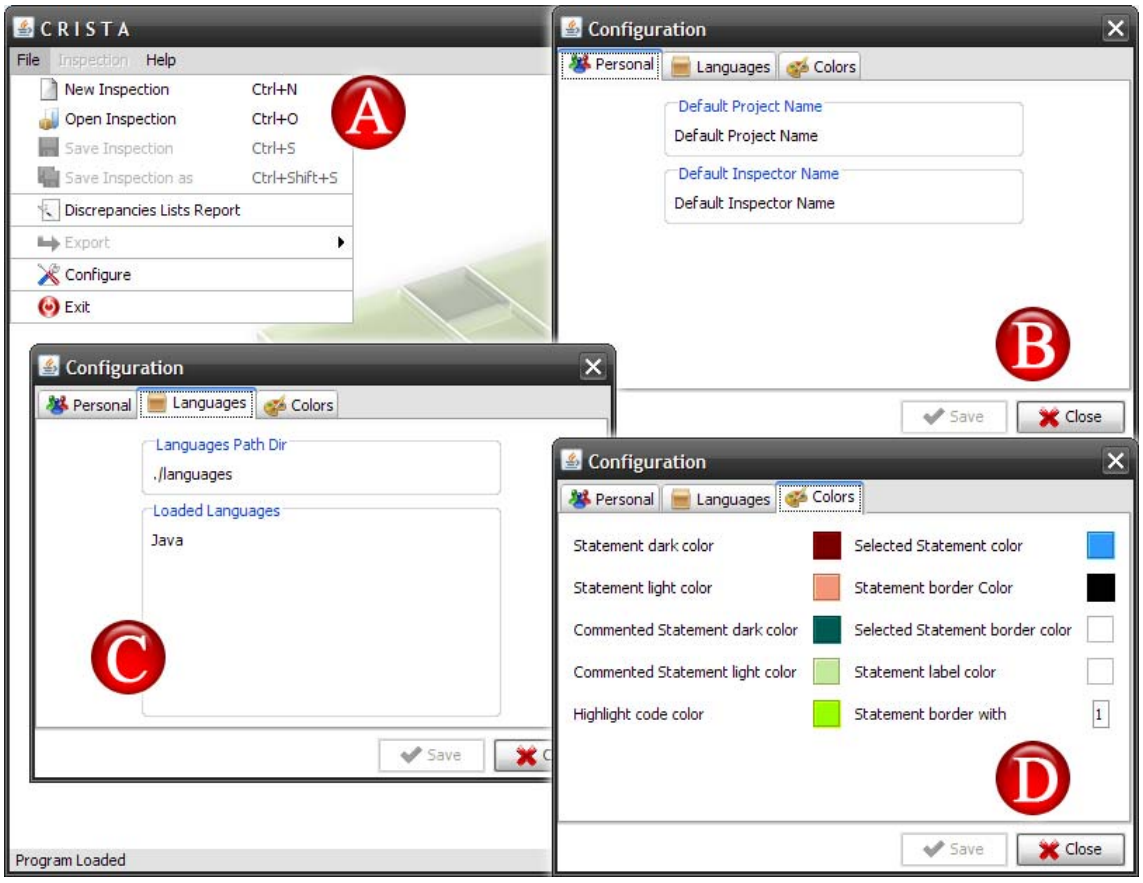


Figura 6.24 - Opções de configuração da ferramenta CRISTA

6.4.10 Ajudar o usuário no processo de inspeção

A ferramenta CRISTA disponibiliza ainda para o usuário um "*help online*" (Figura 6.25 (B)), no qual existem informações a respeito do processo de inspeção com a técnica *Stepwise Abstraction*, além de auxílio de como usar a própria ferramenta. Para acessar o *help* basta acessar a opção *Help/Help*, ou apertar a tecla F1 (Figura 6.25 (A)).

Figura 6.25 - Help da ferramenta CRISTA

6.5 Considerações Finais

Este capítulo apresentou a ferramenta CRISTA, a qual apóia o processo de inspeção de código através da técnica *Stepwise Abstraction*. Além de enumerar os requisitos da ferramenta CRISTA, foi descrita a solução arquitetural adotada, a qual viabilizou o seu uso em outros contextos além do de inspeções de código. Foi feita uma descrição minuciosa de todas as funcionalidades oferecidas pela ferramenta, dentre as quais cabe ressaltar:

- O auxílio à inspeção de código com a técnica *Stepwise Abstraction*: o uso dessa técnica impõe uma compreensão de forma sistemática, a qual é apoiada na ferramenta pela metáfora visual *Treemap*. Além disso, os passos do processo de inspeção também são contemplados na CRISTA.
- A facilidade oferecida à compreensão do código: a ferramenta provê a geração de diferentes relatórios que auxiliam o próprio processo de inspeção, por meio de uma interação constante do usuário com a CRISTA, solicitando essa realimentação por meio

dos relatórios e possibilidade de transferir as abstrações realizadas para o código, em forma de comentário, gerando uma redocumentação do mesmo.

- A identificação dos trechos de código de baixa legibilidade: a ferramenta permite marcar o tempo gasto para abstrair cada trecho do código, gerando gráficos de barra que mostram claramente os trechos que consumiram maior tempo, os quais têm grande chance de corresponder a trechos de baixa legibilidade ou grande complexidade.
- A flexibilidade e customização da ferramenta: a arquitetura idealizada para a ferramenta dotou-a de uma facilidade grande para que ela seja instanciada para diferentes linguagens de programação. Isso é essencial para que ela possa ser usada no contexto de manutenção, quando se pensa na diversidade de software legado no mercado.
- A ajuda ao usuário: a ferramenta oferece um sistema de *wizards*, que guia o usuário na realização de várias tarefas. Além disso, a CRISTA disponibiliza para o inspetor uma seção de ajuda, na qual existem informações a respeito do processo de inspeção com a *Stepwise Abstraction*, além de auxílio de como usar a própria ferramenta.

As idéias propostas para a ferramenta, que foram descritas nesse capítulo, fazem da CRISTA uma ferramenta muito útil no contexto de inspeções de código. Cabe ressaltar que a ferramenta foi aceita e premiada com menção honrosa no salão de ferramentas do SBES 2008 – 22º Simpósio Brasileiro de Engenharia de Software (PORTO; MENDONÇA; FABBRI, 2008), principal evento nacional sobre ferramentas de engenharia de software.

No próximo capítulo são apresentados os estudos experimentais realizados para avaliar a ferramenta. Realizaram-se estudos que exploraram o uso da CRISTA tanto no contexto de inspeção como de manutenção.

Capítulo 7

ESTUDOS EXPERIMENTAIS PARA AVALIAÇÃO DA CRISTA

Este capítulo descreve dois estudos experimentais realizados para avaliar o uso da ferramenta CRISTA. O primeiro avalia a usabilidade da ferramenta e seu uso em um contexto de inspeção. O segundo avalia o uso da ferramenta CRISTA em um contexto de manutenção. Ao final de cada estudo são apresentados e discutidos os resultados obtidos.

7.1 Considerações Iniciais

Uma vez implementada a ferramenta CRISTA, como foi apresentado no capítulo anterior, fazia-se necessário avaliá-la por meio de estudos experimentais, pois, como afirmam Basili e outros (1996a), os novos métodos, técnicas, linguagens e ferramentas não deveriam ser apresentados sem passar primeiro por um processo de experimentação a fim de serem validados.

Principalmente na área de engenharia de software, grande ênfase tem sido dada aos estudos experimentais, que se mostram um meio efetivo para avaliar técnicas, bem como adquirir conhecimento (LOTT; ROMBACH, 1996). Como mencionado no Capítulo 4, Shull, Carver e Travassos (2001), definiram um processo sistemático e incremental para validação de tecnologias maduras, e um guia para a melhoria de novas tecnologias. Esse processo é composto pelas etapas: Estudo de viabilidade, Estudo observacional, Estudo de caso em um ciclo de vida real, e Estudo de caso na indústria. Seguindo esse processo, tendo em vista a necessidade de avaliar a ferramenta CRISTA aqui apresentada, nossos primeiros estudos contemplam as etapas de Estudo de viabilidade e Estudo observacional.

Com o objetivo de padronizar os relatos de estudos experimentais na área de engenharia de software Jedlitschka (2008) sugeriu um guia compatível com o proposto por Wohlin e outros (2000). Alguns itens propostos nesse guia foram contemplados no relato dos experimentos apresentados neste capítulo.

Assim, este capítulo relata três estudos experimentais. Os dois primeiros avaliaram a usabilidade e utilidade da ferramenta CRISTA em um contexto de inspeção de código. Os resultados desses dois estudos foram submetidos aos eventos: ESEM 2009 - *3º International Symposium on Empirical Software Engineering and Measurement* (Em avaliação) e SEKE 2009 - *21º International Conference on Software Engineering and Knowledge Engineering*. Já o terceiro estudo avaliou o comportamento da ferramenta CRISTA em um contexto de manutenção. Esse estudo foi relatado em um artigo do WMSWM 2009 - 6º Workshop de Manutenção de Software Moderna. A Tabela X apresenta um resumo dos estudos experimentais realizados e dos respectivos grupos participantes, sendo G alunos de graduação, M1 e M2 alunos de pós-graduação e A um único aluno de graduação.

Tabela 7.1 - Caracterização dos participantes

	Estudo Experimental 1	Estudo Experimental 2	Estudo Experimental 3
Objetivo	Usabilidade (Inspeção)	Apoio ao processo (Inspeção)	Inspeção + Manutenção
Grupo	G	G M1 M2	A

O restante deste capítulo está organizado da seguinte forma: na Seção 7.2 são apresentados os estudos experimentais 1 e 2, para avaliação da ferramenta CRISTA em um contexto de inspeção. Na Seção 7.3, é relatado o estudo experimental 3, para avaliação da ferramenta no contexto de manutenção. Por fim, a Seção 7.4 apresenta as considerações finais deste capítulo.

7.2 Estudos experimentais 1 e 2 - Avaliação da ferramenta CRISTA no contexto de Inspeção

Nessa seção são relatados os estudos experimentais 1 e 2, os quais buscaram avaliar o uso da ferramenta CRISTA em um contexto de inspeção de código. Cabe lembrar

nesse momento que foi usado o modelo proposto por Jedlitschka (2008). Entretanto, alguns itens desse modelo foram omitidos, por não serem necessários nos estudos.

7.2.1 Planejamento

O planejamento desses estudos foi feito com base na técnica *Goal-Question-Metric* (GQM) (BASILI; CALDIERA; ROMBACH, 1994). Seguem-se abaixo a descrição dos objetivos, participantes, materiais, hipóteses, desenho experimental, procedimentos de execução e de análise dos estudos experimentais 1 e 2.

7.2.1.1 Objetivos

Esses estudos tiveram o objetivo de avaliar o uso da ferramenta CRISTA no contexto de inspeção de código. Assim, no estudo experimental 1 buscou-se avaliar a usabilidade da ferramenta no contexto de inspeção, enquanto que o estudo experimental 2 buscou avaliar a utilidade da ferramenta no mesmo contexto.

7.2.1.2 Participantes

Para realizar os experimentos, foram selecionados três grupos de alunos:

- Grupo G: Alunos de graduação em Ciência da Computação da Universidade Federal de São Carlos (UFSCar)
- Grupo M1: Alunos de mestrado em Ciência da Computação da Universidade Federal de São Carlos (UFSCar)
- Grupo M2: Alunos de mestrado em Ciência da Computação e Matemática Computacional da Universidade de São Paulo (ICMC-USP).

Os artefatos a serem inspecionados nesses estudos experimentais estavam escrito em Java e, com base na caracterização dos participantes observada na Tabela 7.2, o grupo G possuía um conhecimento limitado na linguagem. Toda via, foi verificado que os participantes do grupo G foram capazes de ler e compreender a lógica do programa.

Tabela 7.2 - Caracterização dos participantes

	Número de estudantes	Linguagens conhecidas	% de estudantes com conhecimento em Java	Segunda metade do curso	% de estudantes com experiência em desenvolvimento em empresas
G	20	C, C++, Pascal	30%	Sim	6%
M1	7	C, C++ e Java	100%	Sim	43%
M2	14	C, C++, Pascal, PHP, Delphi, Java	85,7%	Sim	43%

Como mostrado na Tabela 7.2, o grupo G conhecia principalmente as linguagens de programação C, C++ e Pascal, sendo poucos alunos os que tiveram contato com a linguagem Java. A Tabela 7.1 mostra que poucos estudantes do grupo G já desenvolveram software em empresas. A Figura 7.1 apresenta um gráfico mais detalhado a respeito da experiência dos participantes do grupo G com relação ao desenvolvimento de software.

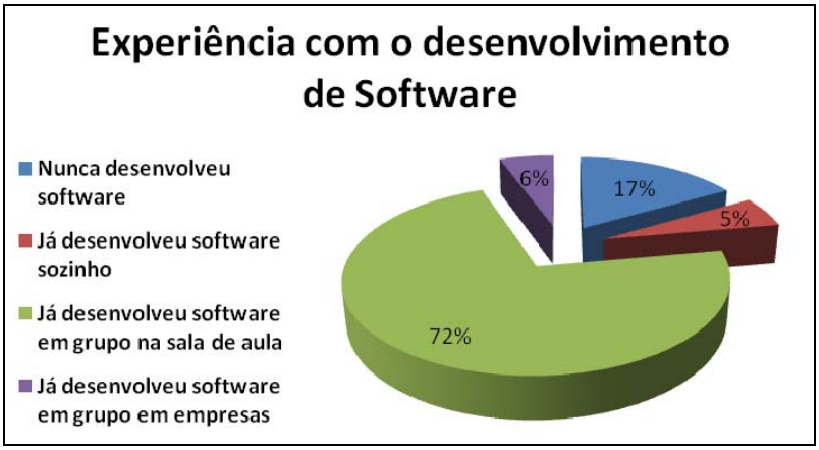


Figura 7.1 - Experiência do grupo G com o desenvolvimento de Software

A partir da figura acima, é possível perceber que somente uma pequena parcela dos participantes já desenvolveu software em empresas, restringindo o seu conhecimento em programação apenas às aulas de graduação.

Os grupos M1 e M2 são muito parecidos entre si e, ao mesmo tempo, muito diferentes do grupo G, em relação às linguagens conhecidas e a experiência em desenvolvimento de software. Ambos os grupos M1 e M2 possuíam conhecimento na linguagem Java e, conforme é mostrado nas Figuras 7.2 e 7.3, possuíam uma boa experiência em desenvolvimento de software em empresas.

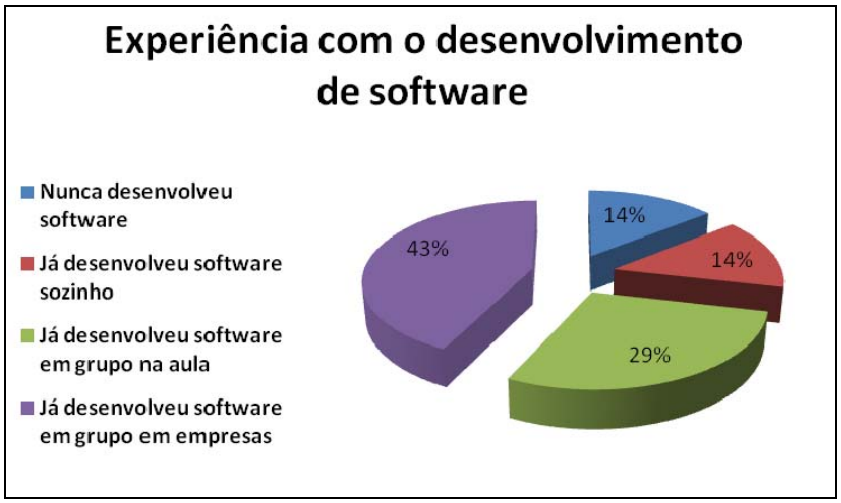


Figura 7.2 - Experiência do grupo M1 com o desenvolvimento de Software

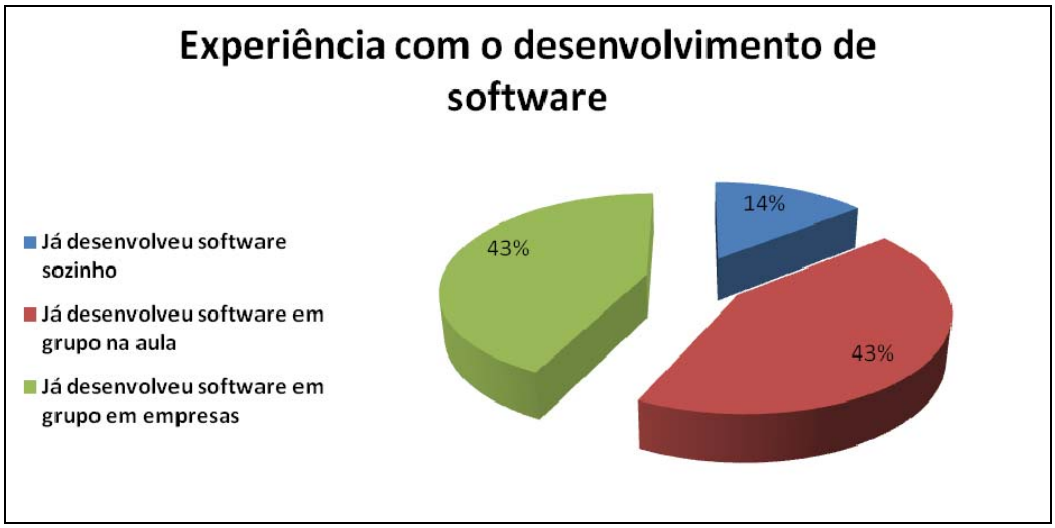


Figura 7.3 - Experiência do grupo M2 com o desenvolvimento de Software

Além das informações contidas na Tabela 7.2, todos os três grupos possuíam um bom conhecimento de inglês, além de possuíram o conhecimento relativo à inspeções restritos à sala de aula e livros. Além disso, somente o grupo M2 possuía conhecimento a cerca de ferramentas de visualização, sendo que cerca de 21,5% dos participantes já haviam trabalhado com algum tipo de ferramenta de visualização de informações.

Para os alunos do grupo G, as atividades dos experimentos vieram a substituir uma das avaliações teóricas da disciplina, de tal sorte que a nota corresponderia não à qualidade das atividades desempenhadas pelo aluno, mas exclusivamente pela sua presença nas atividades. Já os alunos de mestrado (grupos M1 e M2) foram convidados a participar dos experimentos, não tendo nenhum tipo de benefício agregado à participação.

7.2.1.3 Materiais do experimento

Na realização dos experimentos, foram usados os seguintes materiais:

1) Treinamento sobre inspeção e a técnica *Stepwise Abstraction*;

   Foi montada uma apresentação para explicar os passos e regras da inspeção, bem como explicar e mostrar um exemplo de aplicação da técnica *Stepwise Abstraction* em um código simples de 22 linhas (Disponível no Apêndice B).

2) Exercício para aplicação da técnica *Stepwise Abstraction*;

   A fim de consolidar os conhecimentos adquiridos no treinamento sobre inspeção e na *Stepwise Abstraction*, foi montado um exercício para os estudantes abstraírem um programa. O código possuía 49 linhas e calculava o maior valor primo menor que o valor informado pelo usuário (Disponível no Apêndice C).

3) Apresentação sobre heurísticas de usabilidade de software;

Uma vez que os alunos do grupo G não possuíam experiência em usabilidade de software, foi disponibilizada aos participantes uma apresentação a qual abordava as 10 heurísticas de usabilidades propostas por Jacob Nielsen (1993). Essa apresentação serviu de referência na avaliação da usabilidade da ferramenta CRISTA (Disponível no Apêndice B).

4) Questionários;

Como forma de obter o *feedback* dos participantes, foram aplicados três questionários (Q1, Q2 e Q3) (Disponíveis no Apêndice D).

Q1: para caracterização dos participantes, dividido em duas partes: (i) dados pessoais e (ii) características técnicas do participante.

Q2: para avaliação do treinamento oferecido

Q3: para avaliação da ferramenta, dividido em 4 partes: (i) usabilidade, (ii) funcionalidades, (iii) avaliação dos relatórios, e (iv) comentário geral.

5) Programas para execução do experimento

Dois programas Java foram criados para o estudo experimental 1: WordCount (com 90 linhas e contava o número de linhas, palavras e caracteres de um arquivo texto) e MergeSort (com 134 linhas, fazia o ordenamento recursivo de uma lista de inteiros) (Disponíveis no Apêndice C).

Para o estudo experimental 2 foram selecionados mais dois programas Java: Ntree (com 280 linhas e 10 defeitos conhecidos) e Nametbl (com 318 linhas e 11 defeitos conhecidos) (Disponíveis no Apêndice C). Ambos os programas já tinham sido usados nos experimentos de Basili, Caldiera e Shull (BASILI; CALDIERA; SHULL, 1996), entretanto nesses experimentos, os programas estavam originalmente escritos na linguagem C.

7.2.1.4 Hipóteses, parâmetros e variáveis

Considerando que, de acordo com a classificação apresentada por Wohlin e outros (2000), classificamos esses estudos experimentais como sendo exploratórios e que, de acordo com a metodologia de validação de novas tecnologias apresentada por Shull, Carver e Travassos (SHULL; CARVER; TRAVASSOS, 2001), esses estudos encontram-se nas categorias de estudo de viabilidade e observacional, nenhuma hipótese foi definida formalmente para ser analisada durante a realização dos mesmos. A intenção predominante nesses estudos foi de avaliar os aspectos de usabilidade da CRISTA, no sentido de verificar se ela era fácil de usar, fácil de aprender, e se apoiava de maneira satisfatória a atividade de inspeção com o uso da técnica *Stepwise Abstraction*.

7.2.1.5 Desenho experimental

O desenho experimental dos estudos é descrito a seguir.

Estudo experimental 1: o objetivo foi avaliar a ferramenta CRISTA do ponto de vista de sua usabilidade básica. Buscando avaliar quão fácil era entender como a CRISTA ajuda no processo de inspeção, não foi dado nenhum treinamento na ferramenta. Os participantes foram treinados somente no processo de inspeção e na técnica *Stepwise Abstraction*. Além disso, foram dados *guidelines* para se avaliar a usabilidade da ferramenta. Somente o grupo G realizou o estudo 1, o qual é representado na Tabela 7.3.

Tabela 7.3 - Desenho experimental do estudo 1

Treinamento no processo de inspeção e na técnica *Stepwise Abstraction*		
Inspeção manual para reforçar os conhecimentos na técnica SA (exemplo)		
Seção	Inspeção com a CRISTA	Programa
1	G	P1 (WordCount)
2	G	P2 (MergeSort)

Estudo experimental 2: o objetivo foi avaliar a ferramenta CRISTA do ponto de vista de seu suporte ao processo de inspeção com a técnica SA. De forma a obter uma resposta sobre esse suporte, diferentemente do estudo experimental 1, os participantes foram treinados na ferramenta da melhor forma possível, explorando todas as suas funcionalidades e recursos. Esse estudo foi replicado três vezes, uma para cada grupo de estudantes.

Para a execução desse estudo, cada grupo de estudante foi dividido em dois subgrupos, de forma que a inspeção manual e assistida pela ferramenta fosse alternada. A Tabela 7.4 mostra o desenho experimental do estudo 2.

Tabela 7.4 - Desenho experimental do estudo 2

Treinamento no processo de inspeção e na técnica *Stepwise Abstraction*			
Treinamento na ferramenta CRISTA			
Seção	Inspeção Manual	Inspeção com a CRISTA	Programa
1	Subgrupo1	Subgrupo2	P1 (Ntree)
2	Subgrupo2	Subgrupo1	P2 (Nametbl)

7.2.1.6 Procedimento

Os estudos experimentais foram divididos em tarefas e, de forma a resumir seus procedimentos, será usado o seguinte modelo:

Número da Tarefa	*Executor*	*Duração (minutos)*
	Descrição da Tarefa	*Material*

Usando esse modelo, as tarefas dos estudos experimentais foram as seguintes:

Estudo experimental 1

Dia 1

1	Pesquisador	30
	Treinamento no processo de inspeção e na técnica *Stepwise Abstraction*	Slides
2	Pesquisador	30
	Aplicação manual da técnica *Stepwise Abstraction* em um pequeno programa	22 linhas de código
3	Participantes	90
	Aplicação manual da técnica *Stepwise Abstraction* em um pequeno programa	49 linhas de código
4	Participantes	30
	Preenchimento dos questionários	Q1 e Q2

Dia 2

1	Participantes	Até 210
	Inspeção com a ferramenta CRISTA	Programa WordCount
2 (opcional)	Participantes	Dentro do limite de 210 definidos para a tarefa 1
	Inspeção com a ferramenta CRISTA	Programa MergeSort
3	Participantes	30
	Preenchimento do questionário	Q3

Observações: (i) observar que não foi oferecido nenhum treinamento na ferramenta CRISTA; (ii) a tarefa 2 do dia 2 foi opcional. Somente os dois estudantes mais rápidos, iniciaram a tarefa 2, enquanto os outros estavam finalizando a tarefa 1.

Estudo experimental 2

Dia 1

1	Pesquisador	30
	Treinamento no processo de inspeção e na técnica *Stepwise Abstraction*	Slides
2	Pesquisador	30
	Treinamento na ferramenta CRISTA	Slides
3	Participantes – subgrupo 1	Até 210
	Inspeção manual	Programa NTree
	Participantes – subgrupo 2	Até 210
	Inspeção com a ferramenta CRISTA	Programa NTree
4	Participantes	5
	Enviar ao pesquisador a lista individual de discrepâncias	Arquivo texto
5	Pesquisador	5
	Enviar a cada participante todas as listas de discrepâncias	Arquivo texto

6	Participantes	5
	Enviar ao pesquisador a lista final de discrepâncias e o tempo gasto na atividade	Arquivo texto

Observação: Para o grupo G, a tarefa 1 não foi realizada, uma vez que já tinha sido realizada no dia 1 do estudo experimental 1.

Dia 2

1	Participantes – subgrupo 2	Até 210
	Inspeção manual	Programa Nametbl
	Participantes – subgrupo 1	Até 210
	Inspeção com a ferramenta CRISTA	Programa Nametbl
2	Participantes	5
	Enviar ao pesquisador a lista individual de discrepâncias	Arquivo texto
3	Pesquisador	5
	Enviar a cada participante todas as listas de discrepâncias	Arquivo texto
4	Participantes	5
	Enviar ao pesquisador a lista final de discrepâncias e o tempo gasto na atividade	Arquivo texto
5	Participantes	30
	Preenchimento dos questionários	Q1, Q2 e Q3

Observação: O grupo G respondeu somente o questionário Q3 novamente.

7.2.1.7 Procedimento de Análise

A partir dos objetivos definidos, os dados dos questionários, os tempos e discrepâncias encontradas nas atividades do estudo experimental 2 foram organizados em uma planilha eletrônica, de forma a facilitar o controle estatístico. Os dados foram analisados de forma a identificar padrões e situações que deixem clara a interferência da ferramenta CRISTA no processo de inspeção.

7.2.2 Execução

A seguir, são descritos os procedimentos necessários para a preparação e desvios na execução das tarefas descritas a cima.

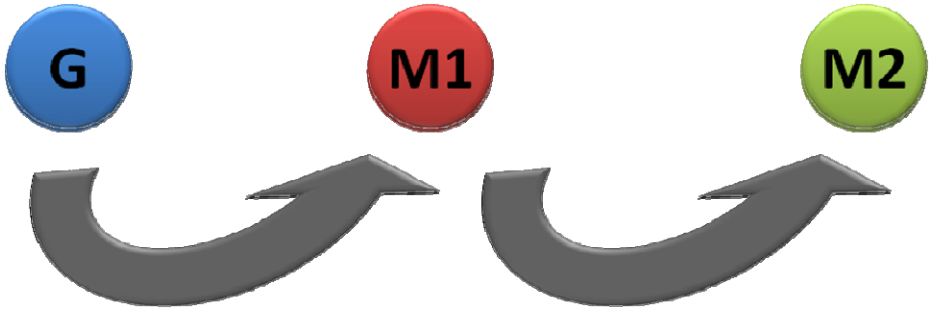
7.2.2.1 Preparação

Como preparação para a execução do experimento, foi necessário preparar todo o material teórico contido no treinamento dos participantes, bem como os programas e questionários.

Foi necessário também preparar o ambiente onde o estudo experimental foi realizado. Para tanto foi reservada uma sala composta por vinte computadores e um projetor. Foi instalada a ferramenta CRISTA em todos os computadores, e o projetor foi usado para as apresentações dos treinamentos.

7.2.2.2 Desvios

Ocorreram três desvios durante a execução do estudo experimental 2. A Figura 7.4 ilustra os desvios ocorridos com relação aos três grupos de participantes.



- Análise da lista de discrepâncias
- O momento em que as discrepâncias foram registradas
- Versão da ferramenta

- Versão da ferramenta

Figura 7.4 - Desvios para o estudo experimental 2 entre os grupos G, M1 e M2

1) Análise da lista de discrepâncias:

Ocorreu uma diferença na forma como foram analisadas as listas de discrepâncias do grupo G e dos grupos M1 e M2.

Grupo G:

A reunião da inspeção manual foi feita coletivamente, ou seja, todos os participantes que realizaram a inspeção manual se reuniram em roda e discutiram cada discrepância. Por outro lado, os participantes que fizeram a inspeção com a ferramenta CRISTA reuniram as listas individualmente na ferramenta, ou seja, sem uma discussão com todos da equipe.

Grupos M1 e M2:

A execução diferenciada entre os subgrupos que realizaram inspeção manual e com a ferramenta tornou difícil a comparação entre eles. Assim, achou-se melhor mudar a reunião das discrepâncias para os grupos M1 e M2. Para esses grupos, a reunião de discrepâncias para a equipe que fez a inspeção manual ocorreu também individualmente,

assim como na equipe que usou a ferramenta. O objetivo dessa mudança era poder comparar melhor os tempos de junção da lista final de discrepâncias.

2) O momento em que as discrepâncias foram registradas:

Grupo G:

O grupo foi instruído a registrar as discrepâncias somente ao final da inspeção, como originalmente definido pela técnica *Stepwise Abstraction*.

Grupos M1 e M2

Como alguns participantes do grupo G relataram que encontraram alguns possíveis defeitos durante a leitura do código, os grupos M1 e M2 foram instruídos a registrar os possíveis defeitos a qualquer momento. Essa mudança na aplicação da *Stepwise Abstraction* realmente melhorou a efetividade dos participantes, os quais registraram um maior número de discrepâncias no total, conforme é apresentado na análise dos resultados.

3) Melhorias na ferramenta CRISTA

Foi conscientemente decidido realizar melhorias na ferramenta CRISTA assim que possível. Dessa forma, com a indicação de melhorias sugeridas pelos grupos G e M1, foi decidido implementá-las. Com isso, a versão da ferramenta CRISTA usada pelo grupo M2 estava ligeiramente diferente da versão utilizada pelos grupos G e M1. Apesar de serem feitas melhorias na versão utilizada pelo grupo M2, nessa versão foi inserido um defeito que impossibilitava selecionar a instrução por meio do código fonte. Fato esse que teve certo impacto na execução do estudo, como será apresentado mais a frente.

7.2.3 Análise

7.2.3.1 Descrição estatística

Todos os dados estatísticos dos estudos experimentais 1 e 2 encontram-se no Apêndice E.

7.2.3.2 Preparação do conjunto de dados

Os dados coletados com os questionários foram tabulados em uma planilha eletrônica. Para normalizar as questões em que nem todos os participantes responderam, ou as situações em que algum dos participantes faltou à atividade, foi feita uma média de todos os dados sempre levando em conta o número de pessoas que responderam a cada questão, ou realizaram a atividade.

7.2.4 Discussão

Nessa seção serão discutidos alguns dos resultados obtidos nos estudos experimentais 1 e 2, descritos anteriormente, sendo que a apresentação completa de todos os dados obtidos encontra-se no Apêndice E.

7.2.4.1 Avaliando os resultados e implicações

a) **Estudo experimental 1 + Início do estudo experimental 2**: Avaliação da usabilidade da ferramenta CRISTA contrastando a aplicação do treinamento (realizada apenas para o grupo G)

❖ Mensagens de erro / Relatórios

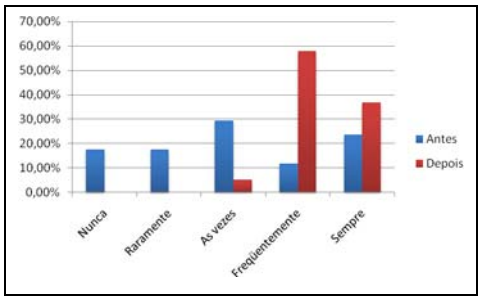


Figura 7.5 - A ferramenta provê boas mensagens de erro, mostrando claramente como resolver o problema? Correspondente à questão 12 do questionário Q3

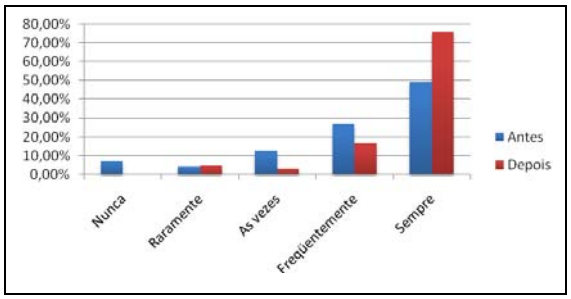


Figura 7.6 - Os relatórios estavam corretos? Correspondente à média das questões 28 a 32 do questionário Q3

Analisando os dados das Figuras 7.5 e 7.6, nota-se que depois do treinamento a avaliação da ferramenta foi mais positiva em ambos os casos. Considerando como exemplo os relatórios, o que aconteceu foi que antes do treinamento alguns dos participantes não entenderam a informação trazida pelo relatório e, portanto, consideravam a sua informação como sendo errada. Veja por exemplo que ocorreram marcações na alternativa “Nunca” antes do treinamento que não persistiram após o mesmo. Veja também que na coluna “Sempre”, a quantidade de marcações depois do treinamento aumentou, mostrando que agora os participantes entenderam melhor o que o relatório deveria apresentar. Analogamente os dados da Figura 7.5 devem ser analisados.

❖ Tempo de recuperação quando um engano é cometido

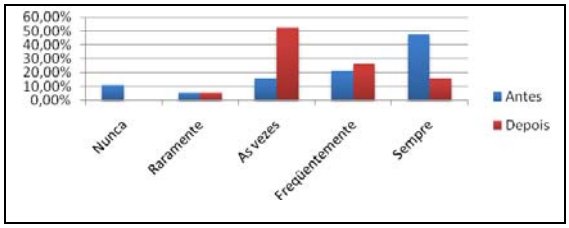


Figura 7.7 - Quando um engano é cometido usando a ferramenta, a recuperação é fácil e rápida? Correspondente à questão 11 do questionário Q3

A Figura 7.7 representa o seguinte fato: antes do treinamento, por não ter conhecimento algum sobre a ferramenta, os participantes aceitaram o tempo de resposta produzido por ela, pois a estavam usando de maneira exploratória. Já depois do treinamento, quando eles adquiriram maior conhecimento, tanto da ferramenta quanto da aplicação da SA via ferramenta, eles consideraram que em alguns momentos o tempo de resposta poderia ser melhor. Por isso, que se observa, por exemplo, na alternativa "Sempre" houve um decréscimo da satisfação dos participantes.

❖ A ferramenta deixa o participante à vontade para realizar o seu papel de inspetor?

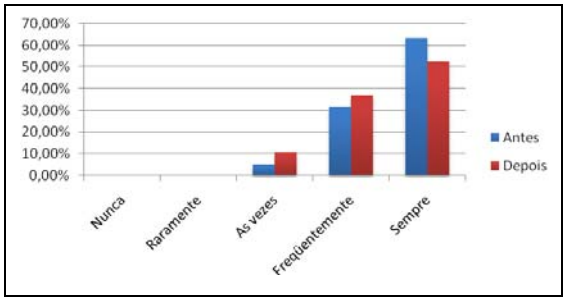


Figura 7.8 - Em algum momento você pensou em desistir de usar a ferramenta por ela não ser apropriada para o propósito que você esperava? Correspondente à questão 15 do questionário Q3

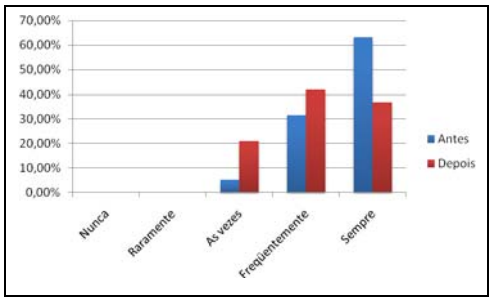


Figura 7.9 - O trabalho fica mais rápido usando a ferramenta ao invés de fazer manualmente? Correspondente à questão 21 do questionário Q3

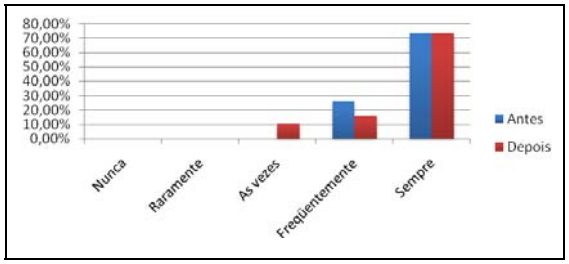


Figura 7.10 - A visualização das instruções do código influenciou na inspeção? Correspondente à questão 33 do questionário Q3

As Figuras 7.8, 7.9 e 7.10, serão comentadas em conjunto, pois, apesar de abordarem pontos específicos, em essência elas tinham por objetivos verificar se a ferramenta ajudava o inspetor em suas tarefas. Como pode ser observado, as três figuras apresentam um pequeno decréscimo da satisfação dos participantes depois que eles receberam o treinamento na ferramenta. Após analisar todo o conjunto de dados, percebemos que o que ocorreu foi que, depois de um maior domínio da ferramenta, outros pontos representaram dificuldades maiores para os participantes, a saber, o conhecimento em Java e a complexidade dos programas como será apresentado mais adiante na Figura 7.20, em que os dados referentes exclusivamente ao estudo 2 serão comentados.

Resumindo, o que pode ser concluído do estudo experimental 1, relativa ao grupo G, é que a usabilidade da ferramenta foi bastante satisfatória, uma vez que raramente algum item de usabilidade apresentou os alternativas “Nunca”, “Raramente” ou “As vezes”, tanto antes como depois do treinamento. Assim, 80,3% dos atributos de usabilidade antes de receber o treinamento eram “Freqüentemente” ou “Sempre”. Após receber o treinamento esse valor passou a ser 87,5%.

Como pode ser observado pelas Figuras 19 a 23 do Apêndice E, podemos concluir também que com o treinamento os participantes puderam perceber melhor a contribuição da ferramenta como um apoio à técnica *Stepwise Abstraction*.

Salienta-se que o *help* da ferramenta não provocou nenhuma distorção nos resultados, uma vez que exatamente metade dos participantes iniciou a exploração da ferramenta após ler o *help*, enquanto que a outra metade não o leu. Outro dado interessante, é que, mesmo para o grupo G, formado por participantes menos experientes, foi notório o fato de que a ferramenta pode contribuir também para as atividades de redocumentação, engenharia reversa, reengenharia, manutenção e análise de código.

b) Estudo experimental 2: Avaliação da usabilidade da ferramenta CRISTA considerando a aplicação do treinamento (realizada pelos três grupos G, M1 e M2)

Ao analisar os dados dos grupos G, M1 e M2, pode-se observar que, de maneira geral, eles foram bastante similares, no sentido de estarem concentrados principalmente nas alternativas “Freqüentemente” e “Sempre”. Isso foi mais notório com relação à avaliação dos relatórios e das métricas de usabilidade (Figuras 33, 34, 36, 38, 40, 42, 49 do Apêndice E). No entanto, com relação a outros pontos, os resultados não foram tão concentrados nas mesmas alternativas:

- A ferramenta facilita o trabalho do inspetor?

Como pode ser observado pelas Figuras 7.11, 7.12 e 7.13, as respostas foram bastante distribuídas entre as diversas alternativas do questionário. Considerando as

questões abordadas nessas figuras, um único uso da ferramenta apoiando a atividade de inspeção não permite uma avaliação precisa de sua influência quando comparada a uma inspeção manual. Nesses gráficos é possível observar que as alternativas “Nunca”, “Raramente” e “As vezes” foram mais assinaladas do que em outras questões, principalmente no que se refere ao grupo M2. Essa tendência do grupo M2 foi também observada em outros momentos, como comentado a seguir.

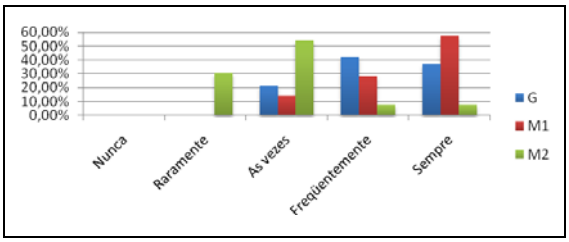


Figura 7.11 - O trabalho fica mais rápido usando a ferramenta ao invés de fazer manualmente? Correspondente à questão 21 do questionário Q3

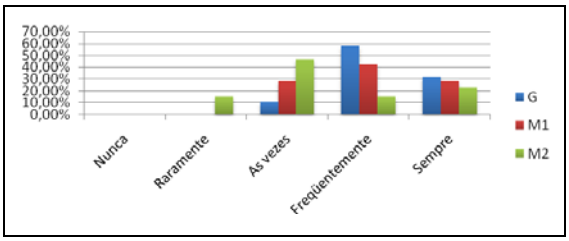


Figura 7.12 - A inspeção usando a ferramenta é tão ou mais eficiente que a inspeção manual? Correspondente à questão 22 do questionário Q3

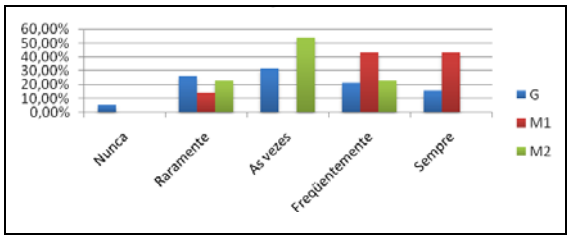


Figura 7.13 - Você acha que o uso da ferramenta ajudou a encontrar mais discrepâncias? Correspondente à questão 34 do questionário Q3

❖ <u>Dificuldades sentidas somente pelo grupo M2</u>

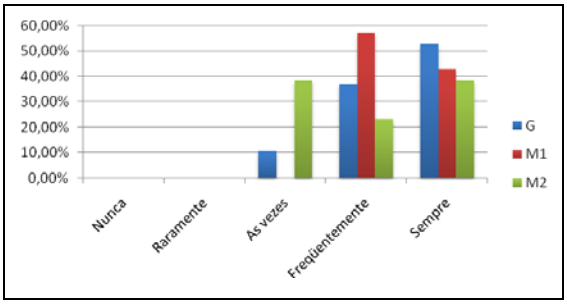


Figura 7.14 - Em algum momento você pensou em desistir de usar a ferramenta por ela não ser apropriada para o propósito que você esperava? Correspondente à questão 15 do questionário Q3

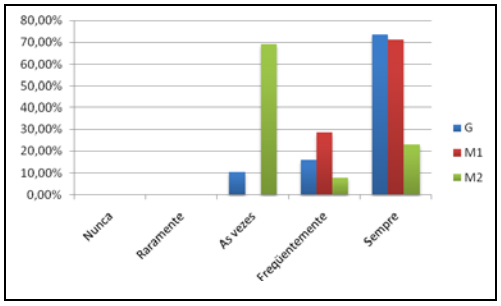


Figura 7.15 - A ferramenta tem todas as funcionalidades que se espera possuir? Correspondente à questão 19 do questionário Q3

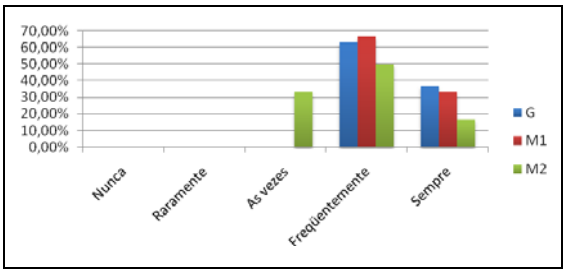


Figura 7.16 - De maneira geral você ficou satisfeito com a ferramenta? Correspondente à questão 37 do questionário Q3

As questões exploradas nas Figuras 7.14, 7.15, 7.16, avaliam de maneira o suporte dado pela ferramenta de maneira geral. Como pode ser observado nos quatro casos, a pior avaliação foi a do grupo M2. O motivo disso foi identificado durante a própria execução da atividade, quando ao tentar fazer as abstrações, os participantes alertaram um defeito na funcionalidade de selecionar uma instrução do programa pela região da tela que apresenta o código. Como essa ação deve ser repetida inúmeras vezes durante a abstração do programa, a ferramenta falhou no suporte a essa ação, trazendo certo descontentamento por parte dos participantes. Embora tal problema tenha sido discutido e explicado aos participantes pelo autor do trabalho, essa insatisfação perdurou até o final da atividade (Vide Desvio 3 na Seção 7.2.2.2).

- ❖ Facilidade de uso e aprendizado da ferramenta e satisfação quando um engano é cometido

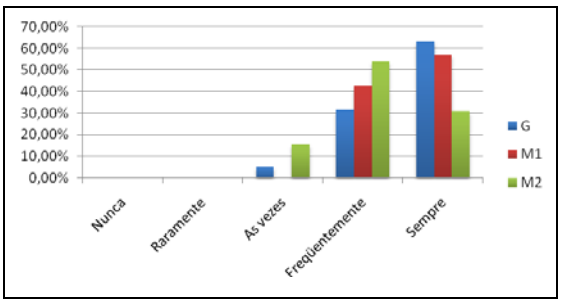


Figura 7.17 - É fácil aprender a usar a ferramenta? Correspondente à questão 1 do questionário Q3

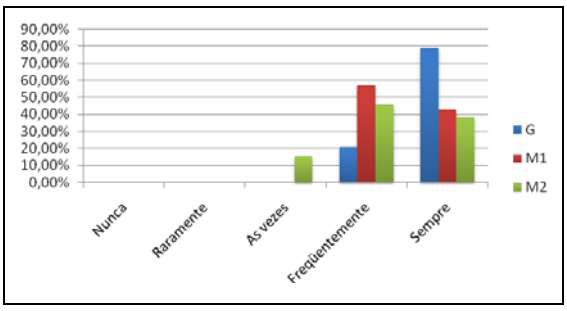


Figura 7.18 - A ferramenta é fácil de usar? Correspondente à questão 13 do questionário Q3

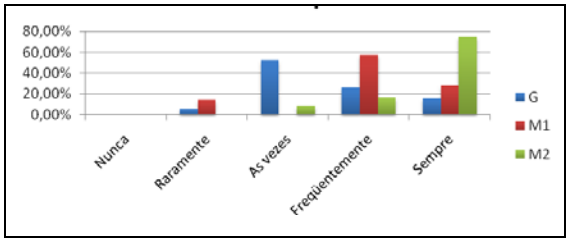


Figura 7.19 - Quando um engano é cometido usando a ferramenta, a recuperação é fácil e rápida? Correspondente à questão 11 do questionário Q3

Quanto à facilidade de aprendizado e facilidade de uso em geral da ferramenta, pode-se observar que as respostas dos três grupos ficaram concentradas nas alternativas “Freqüentemente” e “Sempre”, como pode ser observado nas Figuras 7.17 e 7.18. Por outro lado, contrapondo essa informação com as respostas provenientes da questão retratada na Figura 7.19, percebe-se alguma inconsistência que deverá ser tratada em trabalho futuros. Observe que embora o grupo M2 tenha sido o que menos marcou a alternativa “Sempre” para o aprendizado e facilidade de uso da ferramenta, foi o grupo que considerou que em momentos em que um engano é cometido, a ferramenta proporciona uma recuperação fácil e rápida. No entanto, os grupos G e M1, que se mostraram mais satisfeitos com o aprendizado e a facilidade de uso, são justamente os grupos que chegaram a marcar a alternativa raramente no que diz respeito à fácil recuperação de enganos.

❖ Principais dificuldades encontradas nos estudos

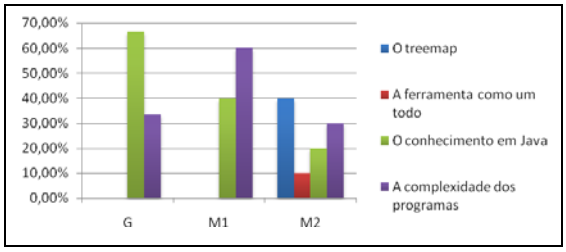


Figura 7.20 - O que você considera que tenha dificultado a identificação das discrepâncias? Correspondente à questão 36 do questionário Q3

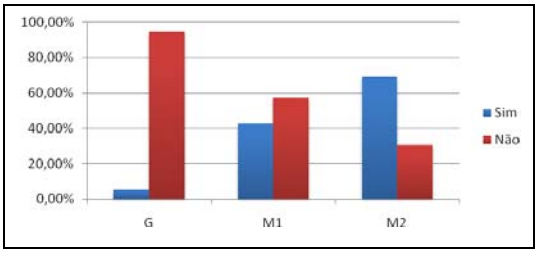


Figura 7.21 - Alguma outra forma de visualização poderia auxiliar mais que a atual? Correspondente à questão 26 do questionário Q3

Observando as informações da Figura 7.20, verifica-se que para o grupo G e M1, as maiores dificuldade que podem ter interferido no estudo foram o conhecimento na linguagem Java e a complexidade dos programas. Para nenhum desses dois grupos a ferramenta ou a metáfora visual utilizada por ela causaram qualquer dificuldade. No entanto, para o grupo M2, pode-se observar que dentre os quatro fatores elencados no questionário, o que mais dificultou a realização do estudo experimental foi a metáfora visual. Isso ficou bastante notório ao tabularmos as respostas da questão que avaliou a aceitação da metáfora visual utilizada (*Treemap*), como pode ser observada na Figura 7.21, na qual o grupo M2 definitivamente considerou que outra forma de representação poderia ser melhor do que a forma utilizada. Já o grupo G foi praticamente unânime no apoio à metáfora utilizada, enquanto que no grupo M1 aproximadamente metade dos participantes gostou da alternativa e a outra metade não.

❖ Tempos e discrepâncias encontradas

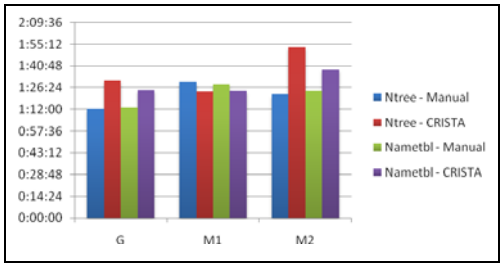


Figura 7.22 - Média dos tempos de abstração dos programas Ntree e Nametbl com e sem a ferramenta CRISTA

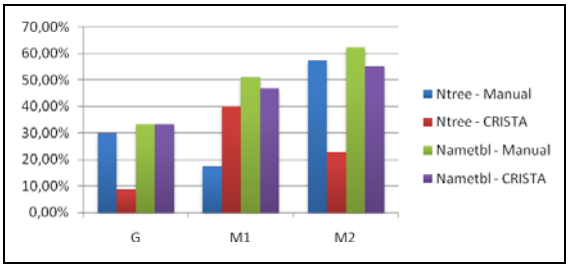


Figura 7.23 - Média da porcentagem de discrepâncias encontradas para os programas Ntree e Nametbl com e sem a ferramenta CRISTA

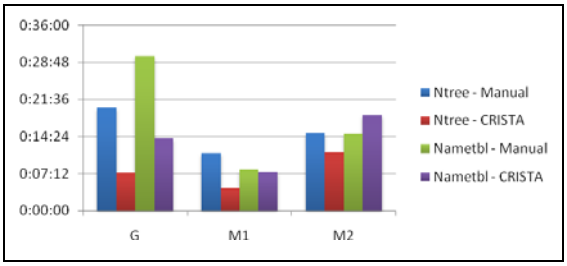


Figura 7.24 - Média dos tempos de junção das discrepâncias dos programas Ntree e Nametbl com e sem a ferramenta CRISTA

Pelas Figuras 7.22, 7.23 e 7.24 percebe-se em todos os resultados dos três gráficos, exceto os dados relativos ao grupo G da Figura 7.24, que quando os participantes realizaram a segunda sessão do estudo 2, as diferenças entre a aplicação manual e com a ferramenta foram reduzidas quando comparadas com as diferenças relacionadas à primeira sessão. Ou seja, tanto no que se refere no tempo gasto nas abstrações, como na quantidade de discrepâncias encontradas, ou no tempo de junção das discrepâncias, o aprendizado adquirido na primeira sessão interfere positivamente tanto para o grupo que aplicou o experimento de forma manual ou com a ferramenta. Isso mostra que tanto a ferramenta, como a compreensão e aprendizado das tarefas de um inspetor podem interferir nos resultados de uma atividade de inspeção. Quanto ao tempo de junção das discrepâncias associado ao grupo G, na Figura 7.24, lembramos do Desvio 1 da Seção 7.2.2.2, que certamente foi o motivo para a obtenção desses valores, já que os participantes que aplicaram a inspeção manual reuniram as discrepâncias em um único grupo, enquanto que os que aplicaram com a ferramenta reuniram as discrepâncias individualmente com apoio da própria ferramenta.

De uma maneira geral, independentemente do grupo, os tempos gastos nas atividades e o número de discrepâncias encontradas não apresentaram diferença significativa. Isso pode ser visto nas Figuras 64 a 67 do Apêndice E.

❖ Discrepâncias encontradas antes e depois da especificação

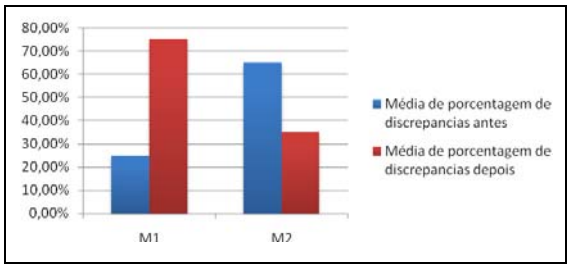


Figura 7.25 - Média da porcentagem de discrepâncias aceitas para o programa Ntree sem a CRISTA

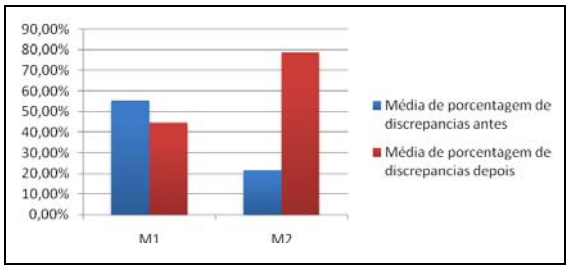


Figura 7.26 - Média da porcentagem de discrepâncias aceitas para o programa Ntree com a CRISTA

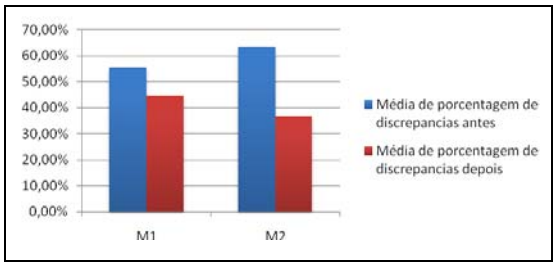


Figura 7.27 - Média da porcentagem de discrepâncias aceitas para o programa Nametbl sem a CRISTA

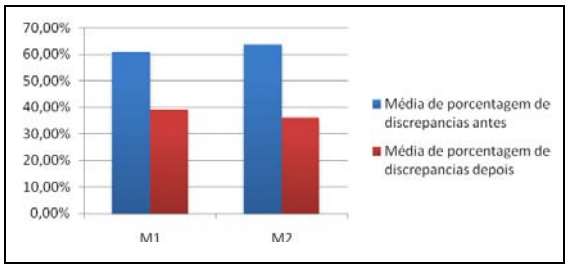


Figura 7.28 - Média da porcentagem de discrepâncias aceitas para o programa Nametbl com a CRISTA

Os dados apresentados nas Figuras 7.25 a 7.28, caracterizam o impacto associado ao Desvio 2 da subseção 7.2.2.2, mostrando que a quantidade de discrepâncias encontradas antes de receber a especificação do programa é considerável. Mesmo nos casos em que essa quantidade foi menor (Figuras 7.25 e 7.26), ela foi maior ou igual a 20%, mostrando que a decisão de instruir os participantes a registrar discrepâncias a qualquer momento pode realmente contribuir para a efetividade do estudo. Esses dados são mais notórios quando observamos as Figuras 7.27 e 7.28, nas quais a quantidade de discrepâncias encontradas foi sempre maior antes dos participantes receberem a especificação do programa, quer para o grupo que aplicou manualmente, ou para o grupo que aplicou com a ferramenta.

De uma forma geral, como pode ser observado no gráfico da Figura 7.29, os tempos totais de inspeção de cada programa para cada grupo foram bastante similares. Assim,

pode-se dizer que usar a ferramenta CRISTA em uma atividade de inspeção tem as suas vantagens, pois o seu uso não impacta negativamente a eficiência da atividade e oferece ao inspetor uma série de recursos que podem facilitar o seu trabalho. Um exemplo disso é a possibilidade de geração de um relatório que mescla as abstrações já realizadas com o código ainda não abstraído, coisa que não seria viável de ser feita manualmente.

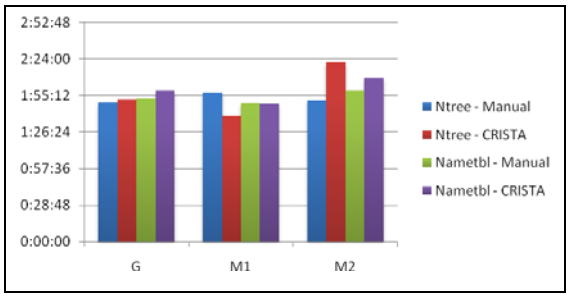


Figura 7.29 - Média dos tempos totais gastos para a inspeção dos programas Ntree e Nametbl com e sem a ferramenta CRISTA

Para finalizar, cabe ressaltar que, além dos dados contidos nas figuras anteriores, os estudos experimentais realizados contribuíram significativamente para a correção de defeitos e sugestões de melhorias. Entre as principais estão:

- Acréscimo da função de voltar (ctrl+z);
- Alertar quando for sobrescrever um arquivo existente;
- Correção de defeitos em relatórios gerados pela ferramenta;
- Possibilidade de rejeitar uma discrepância já aceita;
- Aparecer o código na hora de juntar as listas de discrepâncias;
- Retornar às configurações padrões.
- Poder escrever discrepâncias antes de terminar toda a abstração;
- Correção de *bugs* no *Treemap* e na visualização do código fonte; e
- Acréscimo da função de busca textual no código fonte (ctrl +f).

7.2.4.2 Ameaças à validade

Como ameaças à validade dos estudos experimentais estão:

- O número reduzido de participantes: no total o estudo experimental foi realizado com 41 participantes, o que caracteriza uma amostra pequena para qualquer generalização de resultados que se queira fazer.

- O conhecimento em Java: embora aproximadamente 60% dos participantes tenham declarado conhecimento na linguagem Java, nenhum deles considerava dominá-la plenamente. Como os programas utilizados no estudo experimental estavam implementados nessa linguagem, a falta de domínio na mesma pode ter impactado os resultados obtidos.
- O defeito na ferramenta no estudo realizado com o grupo M2: como a decisão preliminar foi de que a ferramenta seria corrigida a qualquer momento durante o estudo experimental, o defeito nela inserido em decorrência de uma manutenção pode ter impactado negativamente os dados referentes a esse grupo.
- Todos os participantes eram estudantes: embora alguns participantes tenham declarado experiência profissional, os resultados obtidos não podem ser generalizados para outros perfis de inspetores.
- Complexidade dos programas utilizados: embora os programas não fossem muito simples, certamente eles não representavam a complexidade de programas reais.

7.2.4.3 Lições aprendidas

- Pode-se perceber pelos dados coletados no estudo experimental que uma boa usabilidade pode, eventualmente, dispensar o treinamento para o aprendizado de uma ferramenta.
- A escolha dos participantes para a realização de um estudo experimental é essencial para a efetividade obtida. Para uma futura replicação dos estudos aqui apresentados, os participantes deverão ter pleno domínio na linguagem de implementação. Só assim poderão ser avaliados problemas que eventualmente foram encobertos pela pouca fluência de alguns participantes com a linguagem Java.
- Vale frisar também que foi percebido nesse estudo que o número de defeitos identificados pelos participantes antes mesmo de comparar com a especificação do programa, é alto e não pode ser desprezado. Ao contrário, para uma melhor efetividade da inspeção, deve-se estimular a identificação dos defeitos à medida que se lê o código.

Com base nessa verificação, foi proposta uma variação do processo de inspeção com a técnica de leitura SA. Essa variação do processo pode ser observada na Figura 7.30.

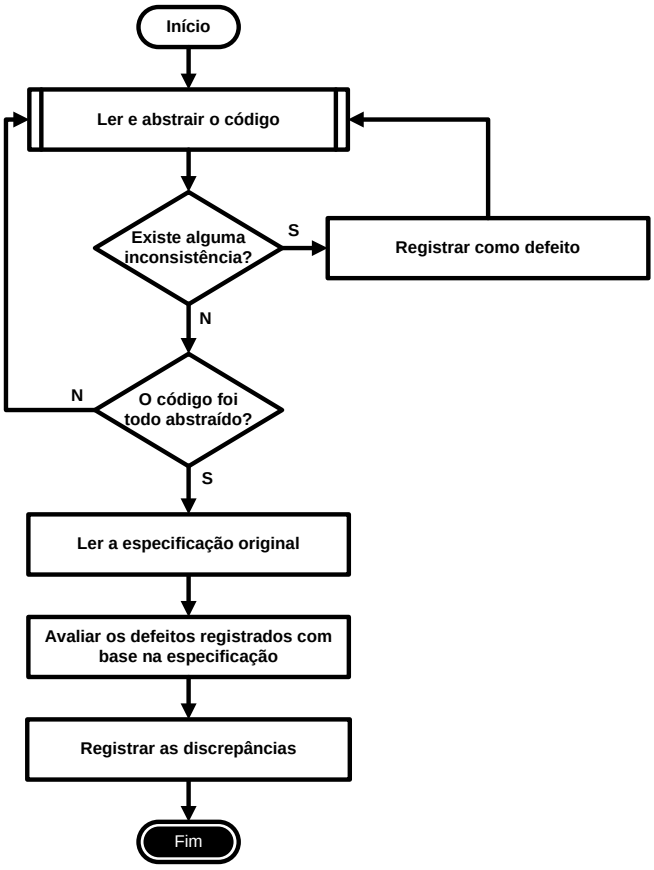


Figura 7.30 - Novo processo definido para inspeção de código com a técnica *Stepwise Abstraction*

Quando comparada com a Figura 2.7 (processo original da técnica *Stepwise Abstraction*), o novo processo de inspeção com a técnica (Figura 7.30) apresenta diferenças no momento em que as discrepâncias são registradas. Pelo novo processo, o inspetor deve anotar todos os trechos com possíveis defeitos, à medida que ele lê o código e os identifica Posteriormente, de posse da especificação do programa, o inspetor deve reavaliar as anotações realizadas, e complementar com as possíveis discrepâncias relacionadas com a especificação original.

7.3 Estudo experimental 3 – Avaliação da ferramenta CRISTA no contexto de manutenção

Nessa seção é relatado o estudo experimental 3, o qual buscou avaliar o uso da ferramenta CRISTA em um contexto de manutenção. Como pode ser visto nesta seção, esse estudo foi muito mais simples que os estudos 1 e 2, sendo realizado por um único

participante. Dessa forma, assim como para os estudos 1 e 2, não houve necessidade de se relatar alguns itens propostos por Jedlitschka (2008).

7.3.1 Planejamento

A seguir, são descritos os objetivos, participantes, materiais, tarefas, procedimentos de execução e de análise para o estudo experimental 3. Assim como nos estudos 1 e 2, não foram estabelecidas hipóteses a serem testadas, pois a intenção foi também de um estudo exploratório.

7.3.1.1 Objetivos

O objetivo desse estudo experimental foi avaliar o uso da ferramenta CRISTA em um contexto de manutenção. Planejou-se então aplicar uma atividade de inspeção no programa *Paint*, para identificar os defeitos, e depois realizar a manutenção para corrigi-los. Além disso, foi objetivo também fazer a documentação para o código. Durante a execução desse estudo experimental, buscou-se identificar problemas e sugestões de melhorias na ferramenta CRISTA da mesma forma como foi feitos nos dois outros estudos.

7.3.1.2 Participante

A atividade foi realizada por um aluno do 3º semestre do curso de Ciência da Computação da Universidade Federal de São Carlos, que possuía experiência de dois anos trabalhando com a linguagem Java.

7.3.1.3 Materiais do experimento

Para a realização do estudo experimental, foi utilizado o programa *Paint*, também utilizado em outros estudos (KO; COBLENZ; AUNG, 2006). O programa *Paint* é um pequeno editor de imagens escrito na linguagem Java. Suas funcionalidades permitem desenhar traços simples (modo "*free hand*"), escolher a cor do traço combinando as cores primitivas vermelho, verde e azul (sistema RGB) e por fim, apagar todo ou partes do desenho feito. É possível também desfazer cada ação do usuário (*undo*). O programa *Paint* possui três defeitos conhecidos:

1) Não é possível escolher a cor amarela;
2) A opção de desenhar linha reta não funciona; e

3) A opção de desfazer a última ação (*undo*) não funciona até que o cursor seja posicionado na tela de desenho.

O programa *Paint* não possui nenhum tipo de documentação e é composto por oito arquivos que possuem, em média, 73 linhas de código. Os defeitos apresentados acima estão espalhados em vários arquivos.

7.3.1.4 Tarefas

Para não prejudicar as atividades acadêmicas do participante, nesse estudo experimental não foi estipulado prazo para a execução das tarefas. Assim, o participante executou-as durante vários dias, sem preocupação com o tempo gasto. A seguir são listadas as tarefas executadas no estudo, seguindo um *template* semelhante ao dos estudos 1 e 2.

1	Pesquisador
	Treinar o participante na ferramenta CRISTA
2	Participante
	Inspecionar o programa *Paint* com a CRISTA, identificando os defeitos presentes no código
3	Participante
	Gerar documentação do programa *Paint*: gerar o relatório de descrições funcionais e exportar as abstrações realizadas durante a inspeção para o código, para efeito de documentação
4	Participante
	Criar casos de teste para ativar os defeitos identificados durante a atividade 2
5	Participante
	Executar os testes criados na atividade 4 para provocar as falhas correspondentes
6	Participante
	Correção dos defeitos encontrados nas atividades 2
7	Participante
	Executar o *Paint* para verificar se o defeito foi corrigido adequadamente e se a correção não provocou novos defeitos (teste de regressão)
8	Participante
	Criação de novos casos de teste para cada defeito encontrado
9	Participante
	Executar os testes criados na atividade 8 para provocar as falhas correspondentes
10	Participante
	Correção dos defeitos encontrados
11	Participante
	Executar o *Paint* para verificar se o defeito foi corrigido adequadamente e se a correção não provocou novos defeitos (teste de regressão)

Como pode ser visto na Figura 7.31, as atividades de 8 a 11 correspondem a um ciclo que foi executado várias vezes.

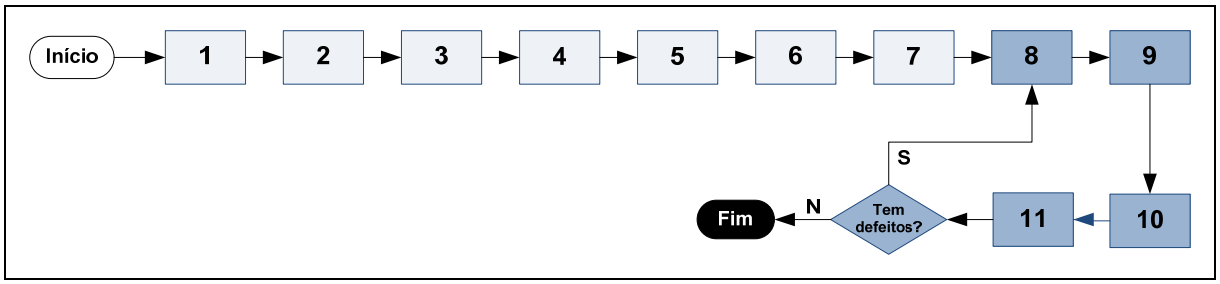


Figura 7.31 - Seqüência das atividades do estudo experimental 3

7.3.2 Execução

Não foi necessário nenhum tipo de preparação especial, como nos estudos anteriores e também não houve nenhum desvio em relação àquilo que foi planejado e apresentado na seção anterior.

7.3.3 Análise e Discussão

O tempo gasto para a inspeção de todo o código (atividade 2) foi aproximadamente de 12 horas. Esse tempo foi armazenado com o recurso da própria ferramenta CRISTA, que marcou o tempo de abstração de cada instrução do código. Os tempos gastos com as outras atividades (3 a 11) foram registrados manualmente pelo aluno. Na Figura 7.32 em que todos esses tempos são apresentados, é possível perceber que a parte mais demorada do estudo de caso foi a realização da inspeção em todo o código. Em contra partida, a correção do único defeito encontrado durante a inspeção (atividade 6) gastou somente 1 minuto.

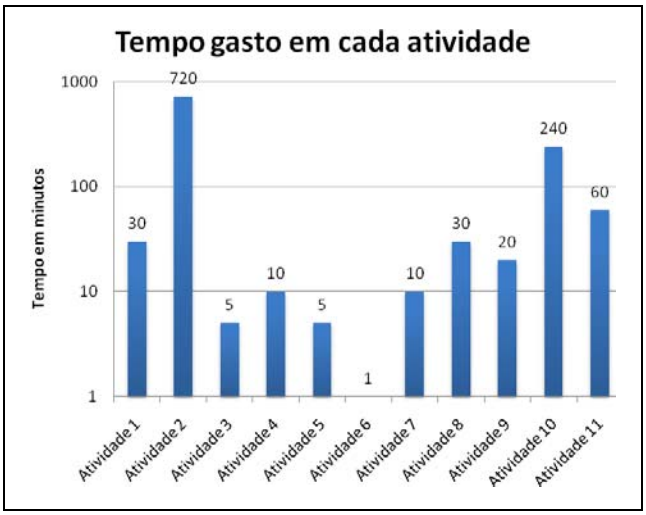


Figura 7.32 - Tempo gasto em cada atividade do estudo de caso

As dificuldades relatadas pelo aluno durante a atividade de inspeção foram:

a) Falta de experiência em inspecionar software; e

b) Não conhecimento da biblioteca *Graphics2D*.

Por outro lado, o aluno relatou não encontrar dificuldades com relação ao uso da ferramenta CRISTA, considerando que tanto a ferramenta como a técnica de leitura SA facilitam a atividade de inspeção.

Durante a realização do estudo de caso, todos os defeitos conhecidos foram encontrados (um durante a inspeção e dois durante a manutenção). Além disso, foram encontrados dois novos defeitos (4 e 5), relacionados com a funcionalidade de *undo*, conforme mostra a Tabela 7.5. Nessa tabela é possível observar também em qual atividade o defeito foi encontrado, além dos tempos para localizar o trecho de código com o defeito (TL) e para efetuar a correção do código (TC).

Tabela 7.5 - Defeitos encontrados no estudo de caso

#	Defeito	Atividade	TL	TC
1	Não é possível escolher a cor amarela.	2	*	1 min.
2	A opção de desenhar uma linha reta não funciona.	7	10 min.	15 min
3	A opção de desfazer a última ação (*undo*) não funciona até que o cursor seja posicionado na tela de desenho	11	10 min.	5 min.
4	A opção de desfazer começa habilitada sem nenhuma ação ter sido executada.**	11	5 min.	2 min.
5	A opção de desfazer não funciona corretamente quando possui ações de limpar a tela.**	11	15 min.	3 horas

* Esse defeito não tem o tempo de localização, pois foi identificado durante a inspeção.
** Novo defeito

Considerando a quantidade total de defeitos (5), embora o aluno tenha identificado apenas um deles (20%) com a atividade de inspeção (com o uso da ferramenta CRISTA), esse fato pode ser explicado pelo programa caracterizar uma aplicação gráfica e, portanto, os defeitos ficam mais perceptíveis durante sua execução.

Os outros 4 defeitos (80%) identificados pelo aluno foram decorrentes do processo iterativo disparado em função da manutenção do defeito identificado na inspeção. Observe na Tabela 7.5, que se gastou pouco tempo para localizar e corrigir os defeitos 2, 3 e 4 com o apoio da ferramenta CRISTA.

Essa eficiência na localização e correção dos defeitos, conforme declarado pelo aluno, foi decorrente da técnica de leitura SA, que força o inspetor a conhecer cada detalhe do código, bem como da documentação interna e externa provida pela ferramenta. A documentação interna do código ajuda substancialmente na identificação da funcionalidade de cada trecho de código.

Com relação ao defeito 5, no entanto, observa-se que o tempo gasto para localização e correção foi bem maior. Isso aconteceu, pois esse defeito correspondia a um erro de lógica que dependia de executar o programa em uma seqüência bastante específica e que também era complexo de entender. Mais especificamente, depois de se usar intercaladamente as opções de *desenhar* e *apagar a tela toda*, ao se aplicar um *undo* e a ação anterior ter sido *apagar a tela toda*, o próximo *undo* percorria primeiro todas as opções *apagar a tela toda* utilizadas durante a execução do *Paint*, para depois desfazer as outras ações. O aluno teve que executar a ferramenta várias vezes para conseguir reproduzir o erro e depois disso, para entender a lógica do que estava acontecendo.

Outra contribuição importante da CRISTA diz respeito ao tempo gasto para gerar a documentação do código. Como pode ser observado na atividade 3 da Figura 7.32, depois de realizar a inspeção com a ferramenta CRISTA, foram gastos apenas 5 minutos para se obter um relatório contendo as descrições funcionais de cada arquivo do programa e para documentar todo o código internamente.

Além de ajudar significativamente no tempo de redocumentação, a CRISTA coleta dados sobre o tempo gasto para abstrair cada trecho do programa, fornecendo informações importantes para efeito de manutenção preventiva, por exemplo. Veja a Figura 7.33, na qual podemos identificar dois trechos no arquivo *PaintWindow.java* cuja complexidade se destaca dos demais. Em um deles, existe um método de 9 linhas com instruções simples, para o qual foram gastos 30 minutos de compreensão, e um outro, de 114 linhas para o qual foram gastas 2 horas e 40 minutos. Considerando que o tempo total para compreensão desse arquivo todo foi de 4 horas, esses dois métodos consumiram 79% desse tempo. Esse fato salienta a contribuição da CRISTA no que diz respeito a indicar trechos propensos à manutenção preventiva, uma vez que 60% dos defeitos da Tabela 7.5 encontravam-se justamente nesse arquivo.

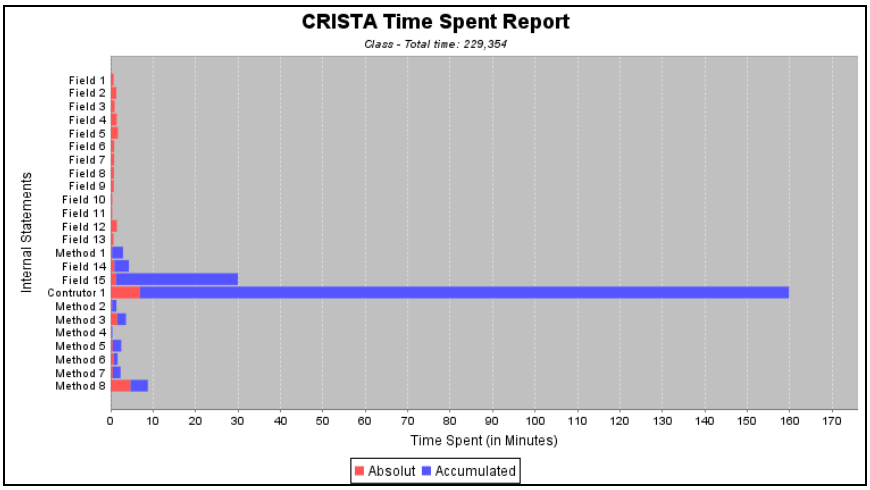


Figura 7.33 - Tempo gasto na inspeção da classe PaintWindow

Por fim, cabe ressaltar que, da mesma forma que nos experimentos 1 e 2, no experimento 3 também foram coletadas informações de melhorias e correção de defeitos na ferramenta CRISTA. Entre elas estão:

- Retirada do zoom automático associado à metáfora visual e adição do zoom manual;
- Atalho no teclado para selecionar a instrução anterior e a próxima daquela que está selecionada no código fonte no momento;
- Adição de uma barra à direita para indicar o nível de aninhamento de cada instrução em relação ao código todo; e
- Correção de defeitos na metáfora visual.

7.3.3.1 Ameaças à validade

Não se preocupou nesse estudo em definirem-se as ameaças à validade, pois o estudo foi muito simples, executado por um único aluno, em uma situação muito particular. Além disso, como não foram estabelecidas hipóteses, não havia motivos para identificar em quais situações os resultados que comprovassem as hipóteses não seriam totalmente confiáveis.

7.3.3.2 Lições aprendidas

Como o programa usado nesse estudo experimental tem por objetivo apoiar a elaboração de desenhos (software de domínio gráfico), a identificação de defeitos durante a inspeção se torna mais difícil, uma vez que nesse tipo de software muitos defeitos só são percebidos por meio das falhas durante sua execução.

7.4 Considerações Finais

Este capítulo apresentou três estudos experimentais cujos dados foram apresentados por meio de alguns itens do formato proposto por Jedlitschka (2008).

Os dois primeiros experimentos foram realizados com estudantes da graduação e da pós-graduação para avaliar o uso da ferramenta CRISTA em um contexto de inspeção de código. Dentre as observações feitas, uma das principais foi que os dados não apresentaram diferenças significativas com relação à usabilidade e utilidade da ferramenta,

seja com ou sem receber treinamento na mesma. Isso é um bom indicativo de que a ferramenta é intuitiva e possui boa usabilidade. Vale destacar também a diferença mínima de tempo entre a inspeção manual e a inspeção assistida pela ferramenta CRISTA. Isso mostra que, além de não ter trazido dificuldades para a atividade de inspeção, em um tempo muito similar à inspeção manual, o inspetor faz a inspeção usando a ferramenta, a qual lhe oferece muito mais recursos do que ele consegue apenas com a aplicação manual.

Cabe lembrar que o estudo com o grupo G e com os grupos M1 e M2 corresponderam, respectivamente, às fases de Estudo de viabilidade e Estudo observacional do processo de Shull, Carver e Travassos (2001).

Com relação ao terceiro estudo experimental, que avaliou a contribuição da ferramenta em relação à atividade de manutenção, o sistema denominado *Paint* foi primeiramente inspecionado e depois sofreu uma manutenção em decorrência do defeito identificado. Por ser um software de domínio gráfico, a identificação de defeitos durante a inspeção é mais difícil, pois muitos deles só se percebem executando o programa. De 3 defeitos previamente conhecidos, apenas um foi encontrado durante a inspeção. Entretanto, durante a manutenção do programa para corrigi-lo, foram encontrados mais 4, sendo que 2 não constavam da lista conhecida.

Em relação à atividade de manutenção propriamente, a contribuição da CRISTA ocorreu da seguinte forma:

a) O tempo de localizar os defeitos e corrigi-los foi muito pequeno, em decorrência do conhecimento adquirido do código e dos tipos de documentação que ela fornece;

b) O tempo de gerar uma documentação para o código depois que a abstração foi realizada também é muito curto, pois depende da seleção das opções apropriadas e do tempo de impressão; e

c) Os gráficos gerados pela coleta de tempos gastos na abstração do código mostraram que realmente podem dar indícios da necessidade de manutenção preventiva, uma vez que nesse caso, os arquivos de maior tempo de compreensão abrigavam a maioria dos defeitos.

Os estudos experimentais descritos neste capítulo encerram os relatos teóricos e práticos deste trabalho. As conclusões, contribuições e trabalhos futuros são descritos em detalhes no capítulo seguinte.

Capítulo 8

CONCLUSÕES

Este trabalho apresentou a ferramenta CRISTA – *Code Reading Implemented with Stepwise Abstraction* –, que foi implementada como o objetivo principal de apoiar a atividade de inspeção de código usando a técnica de leitura *Stepwise Abstraction*. A ferramenta faz uso de uma metáfora visual por meio da técnica de visualização *Treemap* para representar a estrutura do código a ser inspecionado. O uso dessa metáfora proporciona uma maneira simples e rápida de entender a hierarquia das instruções presentes no código, além de auxiliar na própria aplicação da técnica de leitura *Stepwise Abstraction*

A CRISTA possibilita a geração de diferentes relatórios os quais podem ser solicitados para auxiliar a própria atividade de inspeção, facilitando a compreensão do código. Um exemplo é a exportação de um arquivo que contém o código fonte, no qual os trechos abstraídos são substituídos pelas abstrações já inseridas. Esse relatório sintetiza a compreensão feita até o momento e facilita o entendimento do que ainda falta abstrair. Além disso, para dar apoio a todo o processo de inspeção, ela auxilia no registro das discrepâncias encontradas, bem como na junção das listas de discrepâncias elaboradas individualmente pelos inspetores.

A documentação também é um alvo da CRISTA, a qual permite, depois de abstraído todo o código, exportar as abstrações em formato algorítmico, exportar as abstrações para o próprio código, promovendo a documentação interna, e gerar um documento HTML contendo as descrições das estruturas de código escolhidas pelo usuário.

Outro ponto muito importante que a CRISTA apóia é a identificação de trechos de código propensos à manutenção. A CRISTA provê um recurso que marca o tempo desde o momento em que o usuário seleciona um bloco do código até o momento em que ele insere a sua abstração. Com base nessa informação, a ferramenta provê a geração de um relatório que mostra os trechos de código mais demorados para serem abstraídos. Tais trechos

podem ser encarados como os de maior complexidade e/ou menor legibilidade, fornecendo um bom indício de trechos candidatos a sofrer manutenções preventivas.

Essa marcação do tempo para cada estrutura do código também permite identificar o que foi inspecionado, facilitando a tarefa do moderador, no que se refere a avaliar se o inspetor realizou a inspeção de forma bem feita. Isso evita que a efetividade na identificação de defeitos seja baixa devido à falta leitura do código todo.

Outra característica importante da ferramenta CRISTA é que ela é uma ferramenta independente, sem vínculo a nenhuma IDE de desenvolvimento. Com isso, abriu-se mão do uso de algumas funcionalidades já disponibilizadas pela IDE, como por exemplo, a identificação das instruções dos programas por meio do seu *parser,* tornando a identificação das instruções mais trabalhosa. Por outro lado, devido à arquitetura utilizada, tem-se a vantagem de poder facilmente instanciar a ferramenta CRISTA para outras linguagens, fato que não seria possível se fosse utilizada uma IDE. A versão atual aceita códigos em Java, C, C++ e Cobol85.

Além da apresentação da própria ferramenta, foram também apresentados três estudos experimentais conduzidos para avaliá-la. Os dois primeiros estavam no contexto de inspeção de código e foram realizados com o objetivo principal de avaliar sua viabilidade prática, sua usabilidade e se o seu apoio ao processo de inspeção e à técnica *Stepwise Abstraction* eram adequados. No terceiro estudo explorou-se o uso da CRISTA no contexto de manutenção, e o objetivo foi corrigir um sistema após ele ter sido inspecionado.

Os resultados obtidos nos estudos mostraram que os participantes tiveram facilidade de usar a ferramenta. Para uma maior certeza de que ela era intuitiva, no primeiro estudo não foi dado nenhum treinamento na ferramenta e, conhecendo somente o processo de inspeção e a técnica *Stepwise Abstraction*, os participantes (alunos de graduação) conseguiram, em pouco tempo, entender o seu funcionamento e realizar a inspeção. Apenas em relação ao apoio dado pela ferramenta durante o próprio processo é que os alunos não tiraram o todo o proveito possível, por não perceberem, em alguns casos, quais relatórios solicitar ao longo da inspeção. Por outro lado, tendo recebido o treinamento, os participantes conseguiram identificar esse apoio fornecido pela CRISTA e, principalmente no estudo que explorou a atividade de manutenção, essa interação foi bastante positiva. A CRISTA contribuiu muito no tempo de localização e correção dos defeitos, bem como no tempo de redocumentação do programa. Como mencionado por alguns autores (DIAS; ANQUETIL; OLIVEIRA, 2003; SOUSA; ANQUETIL; OLIVEIRA, 2004), um dos maiores problemas enfrentados por mantenedores de software, é a perda de conhecimento que ocorre durante as alterações nos sistemas, e isso é facilitado pela opção de redocumentação fornecida pela CRISTA.

Ressalta-se que, provavelmente por oferecer esses recursos e pelo fato do usuário da CRISTA poder interagir com ela justamente para obter outros tipos de informação, é que os tempos de aplicação da inspeção, quer de forma manual ou com a ferramenta tenham sido bastante próximos.

Outra observação importante decorrente do primeiro estudo é que o número de defeitos identificados antes dos participantes receberem a especificação do programa, é significativamente alto e não pode ser desprezado. Essa observação levou a uma modificação na aplicação da técnica *Stepwise Abstraction*, que determina que a comparação com a especificação original do programa seja feita somente depois que todo o código tenha sido abstraído.

8.1 Contribuições e limitações deste trabalho

As principais contribuições deste trabalho são:

- Criação da ferramenta CRISTA para apoio à atividade de inspeção e outras atividades que requeiram compreensão de código;
- Definição de uma arquitetura que propicia uma fácil instanciação da ferramenta para diferentes linguagens de programação;
- Implementação de reconhecedores para as linguagens Java, C/C++ e Cobol85;
- Modificação do processo de aplicação da técnica *Stepwise Abstraction*;
- Avaliação da ferramenta por meio de estudos experimentais;
- Revisão Sistemática sobre inspeção de software.

As principais limitações deste trabalho são:

- Implementação da metáfora visual somente na técnica *Treemap*;
- Os estudos experimentais foram realizados unicamente em meio acadêmico;
- A ferramenta apóia a abstração somente de um arquivo de cada vez e não possui funcionalidades para reunir abstrações de vários arquivos diferentes

8.2 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 às limitações apresentadas acima;
- Reproduzir os estudos experimentais descritos no Capítulo 7 em um contexto real, com códigos maiores em ambientes industriais;
- Adicionar novos módulos para o reconhecimento de novas linguagens;
- Realizar outros estudos experimentais para avaliar melhor a influência da ferramenta no contexto de manutenção, reengenharia e engenharia reversa;
- Implementar na ferramenta CRISTA funcionalidades que possibilitem o trabalho com múltiplos arquivos e trabalhos de colaboração assíncrona, bem como explorar esses contextos em estudos experimentais;
- Associar os tempos gastos nas abstrações com outras medidas de complexidade de código, como por exemplo a complexidade ciclomática, de forma a permitir um firme embasamento para a definição de estratégias de manutenção; e
- Investigação de técnicas de leitura alternativas que possam tornar a atividade de compreensão mais eficiente.

REFERÊNCIAS BIBLIOGRÁFICAS

ABDELNABI, Z. *et al.* Comparing code reading techniques applied to object-oriented software frameworks with regard to effectiveness and defect detection rate. In: INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING - ISESE Washington, USA, 2004. **Proceedings...** Washington: IEEE Computer Society 2004. p. 239-248.

ACKERMAN, A.; BUCHWALD, L.; LEWSKI, F. Software inspections: An effective verification process. **IEEE Software**, v. 6, n. 3, p. 31-37, maio 1989.

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

AMARAL, E. A. G. G. D.; TRAVASSOS, G. Em busca de uma abordagem para empacotamento de experimentos em engenharia de software. CARVALHO, A. C*., et al* (Ed.) In: JORNADA IBERO-AMERICANA DE ENGENHARIA DE SOFTWARE E ENGENHARIA DO CONHECIMENTO - JIISIC. Salvador, Brasil, 2002. **Anais...** Salvador: JIISIC 2002. p. 75-84.

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

ANDERSON, P. *et al.* Tool support for fine-grained software inspection. **IEEE Computer Society**, v. 20, n. 4, p. 42-50, jul./ago. 2003.

BASILI, V.; CALDIERA, G.; SHULL, F. Studies on reading techniques. In: SOFTWARE ENGINEERING WORKSHOP - SEW. Greenbelt, USA, 1996. **Proceedings...** Greenbelt: NASA/Goddard Software Engineering Laboratory (SEL) 1996. p. 96-102.

BASILI, V. *et al.* Packaging researcher experience to assist replication of experiments. In: INTERNATIONAL SOFTWARE ENGINEERING RESEARCH NETWORK MEETING - ISERN. Sydney, Australia, 1996a. **Proceedings...** Sydney: ISERN 1996a. p.

_____. The empirical investigation of perspective-based reading. **Empirical Software Engineering**, v. 1, n. 2, p. 133-164, nov. 1996b.

BASILI, V. R.; SELBY, R. W. Comparing the effectiveness of software testing strategies. **IEEE Transactions on Software Engineering**, v. 13, n. 12, p. 1278-1296, dez. 1987.

BASILI, V. R.; CALDIERA, G.; ROMBACH, H. D. Goal, question metric paradigm. In: MARCINIAK, J. J. (Ed.). **Encyclopedia of software engineering**: John Wiley & Sons, v.1, 1994. p. 528-532.

BERLING, T.; THELIN, T. A case study of reading techniques in a software company. In: INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING - ISESE. Redondo Beach, USA, 2004. **Proceedings...** Redondo Beach: IEEE Computer Society 2004. p. 229-238.

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

BIOLCHINI, J. *et al.* **Systematic review in software engineering**. PESC - COPPE/UFRJ. Rio de Janeiro, 31 p. 2005. (Technical Report RT-ES 679 / 05)

BOEHM, B.; BASILI, V. R. Software defect reduction top 10 list. **IEEE Computer Society**, v. 34, n. 1, p. 135-137, jan. 2001.

BOOGERD, C.; MOONEN, L. Prioritizing software inspection results using static profiling. In: INTERNATIONAL WORKSHOP ON SOURCE CODE ANALYSIS AND MANIPULATION - SCAM. Philadelphia, USA, 2006. **Proceedings...** Washington: IEEE Computer Society 2006. p. 149-160.

BREEN, M. All things considered: Inspecting statecharts by model transformation. In: SOFTWARE ENGINEERING AND ADVANCED APPLICATIONS - EUROMICRO. Cavtat/Dubrovnik, Croatia, 2006. **Proceedings...** Cavtat/Dubrovnik: IEEE Computer Society 2006. p. 224-231.

CARNEIRO, G. D. F.; ORRICO, A. C. A.; MENDONÇA, M. G. D. Empirically evaluating the usefulness of software visualization techniques in program comprehension activities. KONG, M.*, et al* (Ed.) In: JORNADA IBERO-AMERICANA DE ENGENHARIA DE SOFTWARE E ENGENHARIA DO CONHECIMENTO - JIISIC. Lima, Perú, 2007. **Proceedings...** Lima: Facultad de Ciencias e Ingeniería and Departamento de Ingeniería, Pontificia Universidad Católica del Perú 2007. p. 341-348.

CEMIN, C. **Visualização de informações aplicada à gerência de software**. 2001. 70 p. Dissertação (Mestrado em Computação) - Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2001.

CHAN, L.; JIANG, K.; KARUNASEKERA, S. A tool to support perspective based approach to software code inspection. In: AUSTRALIAN SOFTWARE ENGINEERING CONFERENCE - ASWEC. Brisbane, Australia, 2005. **Proceedings...** Brisbane: IEEE Computer Society 2005. p. 110-117.

CHOWDHURY, A.; LAND, L. P. W. The impact of training-by-examples on inspection performance using two laboratory experiments. In: AUSTRALIAN SOFTWARE ENGINEERING CONFERENCE - ASWEC. Melbourne, Australia, 2004. **Proceedings...** Melbourne: IEEE Computer Society 2004. p. 279-288.

CHRISTOPHE, B. **Treemap java library**. 2009. Disponível em: < http://treemap.sourceforge.net/ >. Acesso em: 23/03/2009.

CODESTRIKER. **Codestriker: Web-based code reviewing**. 2008. Disponível em: < http://codestriker.sourceforge.net >. Acesso em: 29/01/2008.

COLE, B. *et al.* Improving your software using static analysis to find bugs. In: ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES, AND APPLICATIONS - OOPSLA. New York, NY, USA, 2006. **Proceedings...** New York: ACM Press 2006. p. 673-674.

CONRADI, R. *et al.* **A pragmatic documents standard for an experience library: Roles, documents, contents and structure**. University of Maryland, 50 p. 2001. (Technical Report CS-TR-4235)

COOPER, D. *et al.* Using dependence graphs to assist manual and automated object oriented software inspections. In: AUSTRALIAN SOFTWARE ENGINEERING CONFERENCE - ASWEC. Sydney, Australia, 2006. **Proceedings...** Sydney: IEEE Computer Society 2006. p. 262-269.

DELINE, R.; CZERWINSKI, M.; ROBERTSON, G. Easing program comprehension by sharing navigation data. In: IEEE SYMPOSIUM ON VISUAL LANGUAGES AND HUMAN-CENTRIC COMPUTING - VLHCC. Dallas, USA, 2005. **Proceedings...** Dallas: IEEE Computer Society 2005. p. 241-248.

DENGER, C.; KOLB, R. Testing and inspecting reusable product line components: First empirical results. In: INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING - ISESE. Rio de Janeiro, Brasil, 2006. **Proceedings...** New York: ACM Press 2006. p. 184-193

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

DIAS, M. G. B.; ANQUETIL, N.; OLIVEIRA, K. M. D. Organizing the knowledge used in software maintenance. **Journal of Universal Computer Science**, v. 9, n. 7, p. 641-658, 2003.

DIEHL, S. **Software visualization: Visualizing the structure, behaviour, and evolution of software**. Edição 1. New York: Springer-Verlag, 2007. 187 p.

DÓRIA, E. S. **Replicação de estudos empíricos em engenharia de software**. 2001. 168 p. Dissertação (Mestrado em Computação) - Instituto de Ciências Matemáticas e de Computação (ICMC), Universidade de São Paulo, São Carlos, 2001.

ESE-COPPE. **Experimental software engineering - glossary of terms**. 2008. Disponível em: < http://lens-ese.cos.ufrj.br/wikiese/index.php/Experimental_Software_Engineering_-_Glossary_of_Terms >. Acesso em: 20/11/2008.

FAGAN, M. E. Design and code inspections to reduce errors in program development. **IBM Systems Journal**, v. 15, n. 7, p. 182-211, 1976.

_____. Advances in software inspections. **IEEE Transactions on Software Engineering**, v. 12, n. 7, p. 744-751, 1986.

FINDBUGS. **Findbugs**. 2008. Disponível em: < http://findbugs.sourceforge.net >. Acesso em: 29/01/2008.

FREITAS, C. M. D. S. *et al.* Introdução à visualização de informações. **Revista de Informatica Teórica e Aplicada**, v. 8, n. 2, p. 143-158, out. 2001.

GRAPHVIZ. **Graphviz - graph visualization software**. 2008. Disponível em: < http://www.graphviz.org/ >. Acesso em: 20/11/2008.

GRÜNBACHER, P.; HALLING, M.; BIFFL, S. An empirical study on groupware support for software inspection meetings. In: IEEE INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING - ASE. Los Alamitos, USA, 2003. **Proceedings...** Los Alamitos: IEEE Computer Society, out. 2003. p. 4-11.

GUPTA, V.; PATNAIK, A. R.; GOEL, N. A system for controlling software inspections. In: CANADIAN CONFERENCE ON ELECTRICAL AND COMPUTER ENGINEERING - CCECE Montréal, Canada, 2003. **Proceedings...** Montréal: IEEE Computer Society, maio 2003. p. 1343-1346.

HALLING, M.; BIFFL, S.; GRÜNBACHER, P. An experiment family to investigate the defect detection effect of tool-support for requirements inspection. In: INTERNATIONAL SOFTWARE METRICS SYMPOSIUM. Sydney, Australia, 2003. **Proceedings...** Sydney: IEEE Computer Society 2003. p. 278-285.

HARJUMAA, L.; TERVONEN, I.; VUORIO, P. Improving software inspection process with patterns. In: INTERNATIONAL CONFERENCE ON QUALITY SOFTWARE - QSIC Braunschweig, Germany, 2004. **Proceedings...** Braunschweig: IEEE Computer Society 2004. p. 118-125.

HE, L.; CARVER, J. Pbr vs. Checklist: A replication in the n-fold inspection context. In: ACM/IEEE INTERNATIONAL SYMPOSIUM ON INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING - ISESE. Rio de Janeiro, Brasil, 2006. **Proceedings...** New York: ACM Press 2006. p. 95-104.

HEDBERG, H. Introducing the next generation of software inspection tools. In: INTERNATIONAL CONFERENCE PRODUCT FOCUSED SOFTWARE PROCESS IMPROVEMENT. Kausai Science City, Japan, 2004. **Proceedings...** Kausai Science City: Springer 2004. p. 234-247.

HEDBERG, H.; LAPPALAINEN, J. A preliminary evaluation of software inspection tools, with the desmet method. In: INTERNATIONAL CONFERENCE ON QUALITY SOFTWARE - QSIC. Melbourne , Australia, 2005. **Proceedings...** Washington: IEEE Computer Society, set. 2005. p. 45-52.

HETZEL, W. C. **An experimental analysis ofprogram verification methods**. 1976. 297 p. (PhD thesis) University of North Carolina, Chapel Hill, 1976.

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

IEPSEN, E.; LUZZARDI, P. R. G.; LOH, S. Ferramenta para visualização de informações temporais para bancos de dados mestre/detalhe. In: ESCOLA REGIONAL DE BANCO DE DADOS. Caxias do Sul, Brasil, 2007. **Anais...** Caxias do Sul: Universidade de Caxias do Sul 2007. p. 62-71.

ISO/IEC12207. **Tecnologia de informação**. Processos de Ciclo de Vida de Software. ABNT 1998.

JAVACC. **Java compiler compiler**. 2008. Disponível em: < https://javacc.dev.java.net >. Acesso em: 17/04/2008.

JAVAHELP. **Javahelp system**. 2008. Disponível em: < http://java.sun.com/products/javahelp >. Acesso em: 17/04/2008.

JEDLITSCHKA, A.; CIOLKOWSKI, M.; PFAHL, D. Reporting experiments in software engineering. In: SHULL, F.;SINGER, J. e SJØBERG, D. I. K. (Ed.). **Guide to advanced empirical software engineering**. Berlin: Springer, 2008. cap. 8, p. 201-228.

JLINT. **Jlint**. 2008. Disponível em: < http://sourceforge.net/projects/jlint >. Acesso em: 29/01/2008.

JOHNSON, P. M. An instrumented approach to improving software quality through formal technical review. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING - ICSE. Sorrento, Italy, 1994. **Proceedings...** Sorrento: IEEE Computer Society 1994. p. 113-122.

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

JTREEMAP. **Jtreemap**. 2009. Disponível em: < http://jtreemap.sourceforge.net/ >. Acesso em: 23/03/2009.

JWIZARD. **Jwizard**. 2008. Disponível em: < http://flib.sourceforge.net/JWizard/doc >. Acesso em: 17/04/2008.

KALINOWSKI, M. **Infra-estrutura computacional de apoio ao processo de inspeção de software**. 2004. 120 p. Dissertação (Mestrado em Ciência da Computação) Coppe/Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2004.

KALINOWSKI, M.; TRAVASSOS, G. H. A computational framework for supporting software inspections. In: INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING - ASE. Linz, Austria, 2004. **Proceedings...** Linz: IEEE Computer Society 2004. p. 46-55.

KAMSTIES, E.; LOTT, C. M. An empirical evaluation of three defect-detection techniques. In: EUROPEAN SOFTWARE ENGINEERING CONFERENCE - ESEC. Sitges, Spain, 1995. **Proceedings...** Sitges: Springer-Verlag 1995. p. 362-383.

KEIM, D. A. Information visualization and visual data mining. **IEEE Transactions on Visualization and Computer Graphics**, v. 8, n. 1, p. 1-8, jan. 2002.

KITCHENHAM, B. **Procedures for performing systematic reviews**. Keele University. Newcastle-under-Lyme, 33 p. 2004. (Technical Report TR/SE0401)

KITCHENHAM, B. *et al.* Evaluating guidelines for reporting empirical software engineering studies. **Empirical Software Engineering**, v. 13, n. 1, p. 97-121, fev. 2008.

KO, A. J.; COBLENZ, M. J.; AUNG, H. H. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. **IEEE Transactions on Software Engineering**, v. 32, n. 12, p. 971-987, dez. 2006.

KOLLANUS, S. Issues in software inspection practices. In: BOMARIUS, F. e KOMI-SIRVIÖ, S. (Ed.). **Product focused software process improvement**. Berlin: Springer, v.3547, 2005. p. 429-442.

KOLLANUS, S.; KOSKINEN, J. Software inspections in practice: Six case studies. **Product-Focused Software Process Improvement**, v. 4034/2006, p. 377-382, set. 2006.

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

LANUBILE, F.; MALLARDO, T. An empirical study of web-based inspection meetings. In: INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING - ISESE Rome, Italy, 2003. **Proceedings...** Rome: IEEE Computer Society 2003. p. 244.

_____. Inspecting automated test code: A preliminary study. In: BAUMEISTER, H.;MARCHESI, M. e HOLCOMBE, M. (Ed.). **Agile processes in software engineering and extreme programming**. Berlin: Springer, 2007. p. 115-122.

LINGER, R. C.; MILLS, H. D.; WITT, B. I. **Structured programming: Theory and practice**. Reading: Addison-Wesley, 1979. p.

LINTERN, R. *et al.* Plugging-in visualization: Experiences integrating a visualization tool with eclipse. In: ACM SYMPOSIUM ON SOFTWARE VISUALIZATION - SoftVis San Diego, USA, 2003. **Proceedings...** New York: ACM 2003. p. 47-ff.

LOTT, C. M.; ROMBACH, H. D. Repeatable software engineering experiments for comparing defect-detection techniques. **Empirical Software Engineering**, v. 1, n. 3, p. 241-277, 1996.

LUCIA, A. D. *et al.* Assessing the effectiveness of a distributed method for code inspection: A controlled experiment. In: INTERNATIONAL CONFERENCE ON GLOBAL SOFTWARE ENGINEERING - ICGSE. Munich, Germany, 2007. **Proceedings...** Washington: IEEE Computer Society 2007. p. 252-261.

MAFRA, S. N.; TRAVASSOS, G. H. **Leitura baseada em perspectiva: A visão do projetista orientada a objetos**. 2005. Disponível em: < http://www-di.inf.puc-rio.br/~julio//anais/SMafra.pdf >. Acesso em: 15/11/2008.

MARQUES, M. C.; MENDONÇA, M.; SANTOS, C. A. S. Graphminer: Uma ferramenta de mineração visual de dados em bases relacionais In: JORNADA IBERO-AMERICANA DE ENGENHARIA DE SOFTWARE E ENGENHARIA DO CONHECIMENTO - JIISIC Madrid, Spain, 2004. **Proceedings...** Madrid: Actas de las IV Jornadas Iberoamericanas de Ingeniería del Software e Ingeniería del Conocimiento 2004. p. 305-316.

MARUMUSHI. **Newsmap**. 2008. Disponível em: < http://www.marumushi.com/apps/newsmap/newsmap.cfm >. Acesso em: 20/11/2008.

MENDES, A. P. R. **Análise de ferramentas de apoio para as atividades de inspeção de software**. 2006. 62 p. Monografia (Aperfeiçoamento/Especialização em MBA - Tecnologia da Informação) Escola Politécnica da Universidade de São Paulo, São Paulo, 2006.

MICROSYSTEMS, S. **Code conventions for the java programming language**. 2008. Disponível em: < http://java.sun.com/docs/codeconv >. Acesso em: 01/10/2008.

MILLER, J.; YIN, Z. A cognitive-based mechanism for constructing software inspection teams. **IEEE Transactions on Software Engineering**, v. 30, n. 11, p. 811-825, nov. 2004.

MYERS, G. J. *et al.* **The art of software testing**. New York, NY, EUA: John Wiley & Sons, 2004. 192 p.

NASA. **Software formal inspections guidebook**. 1993. Disponível em: < http://satc.gsfc.nasa.gov/Documents/fi/gdb/fitext.txt >. Acesso em: 15/11/2008.

NASCIMENTO, H. A. D. D.; FERREIRA, C. B. R. Visualização de informações: Uma abordagem prática. In: CONGRESSO DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO. São Leopoldo, 2005. **Anais...** São Leopoldo: SBC 2005. p. 1262-1312.

NEGINHAL, S.; KOTHARI, S. Event views and graph reductions for understanding system level c code. In: INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE - ICSM. Philadelphia, USA, 2006. **Proceedings...** Philadelphia: IEEE Computer Society 2006. p. 279-288.

NIELSEN, J. **Usability engineering**. Edição 1. Boston, USA: Morgan Kaufmann, 1993. 358 p.

ORDING, B.; JOBS, S. P.; LINDSAY, D. J. **User interface for providing consolidation and access**. United States Patent. INC., A. USA. US7,434,177 B1**:** 20 p. 2008.

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

PFEIFFER, J.-H.; GURD, J. R. Visualisation-based tool support for the development of aspect-oriented programs. In: INTERNATIONAL CONFERENCE ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT - AOSD. Bonn, Germany, 2006. **Proceedings...** New York: ACM 2006. p. 146-157.

PORTO, D. P.; MENDONÇA, M.; FABBRI, S. C. P. F. Crista - code reading implemented with stepwise abstraction. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE - SESSÃO DE FERRAMENTAS. Campinas, 2008. **Anais...** Campinas: SBC 2008. p. 6.

PREFUSE. **The prefuse visualization toolkit**. 2008. Disponível em: < http://prefuse.org >. Acesso em: 17/04/2008.

PRESSMAN, R. S. **Engenharia de software**. Edição 6. São Paulo: McGrawHill, 2006. 720 p.

RHYNE, T. M. **The convergence of scientific visualization methods with the world wide web**. Tutorial at IEEE Visualization 2000. Salt Lake City, UT, EUA. 2000

RILLING, J.; KLEMOLA, T. Identifying comprehension bottlenecks using program slicing and cognitive complexity metrics. In: INTERNATIONAL WORKSHOP ON PROGRAM COMPREHENSION - IWPC. Portland, USA, 2003. **Proceedings...** Portland: IEEE Computer Society, maio 2003. p. 115-124.

ROCHA, A. R.; MALDONADO, J. C.; WEBER, K. C. **Qualidade de software: Teoria e prática**. São Paulo: Prentice Hall, 2001. 303 p.

RUNESON, P.; ANDREWS, A. Detection or isolation of defects? An experimental comparison of unit testing and code inspection. In: INTERNATIONAL SYMPOSIUM ONSOFTWARE RELIABILITY ENGINEERING - ISSRE. Denver, USA, 2003. **Proceedings...** Denver: IEEE Computer Society 2003. p. 3-13.

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

SEI. **Capability maturity model integration, version 1.1.**, 2008. Disponível em: < http://www.sei.cmu.edu/cmmi >. Acesso em: 04/08/2008.

SENSALIRE, M.; OGAO, P. Tool users requirements classification: How software visualization tools measure up. In: INTERNATIONAL CONFERENCE ON COMPUTER GRAPHICS, VIRTUAL REALITY, VISUALISATION AND INTERACTION IN AFRICA - AFRIGRAPH. Grahamstown, South Africa, 2007. **Proceedings...** New York: ACM Press 2007. p. 119-124.

SHULL, F.; RUS, I.; BASILI, V. How perspective-based reading can improve requirements inspections. **IEEE Computer Society**, v. 33, n. 7, p. 73-79, jul. 2000.

SHULL, F.; CARVER, J.; TRAVASSOS, G. H. An empirical methodology for introducing software processes. In: EUROPEAN SOFTWARE ENGINEERING CONFERENCE AND ACM SIGSOFT SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE ENGINEERING - ESEC/FSE. Viena, Austria, 2001. **Proceedings...** New York: ACM Press 2001. p. 288-296.

SHULL, F.; SINGER, J.; SJØBERG, D. I. K. **Guide to advanced empirical software engineering**. New York: Springer-Verlag, 2007. 388 p.

SILVA, L. F. S.; TRAVASSOS, G. H. Tool-supported unobtrusive evaluation of software engineering process conformance. In: INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING - ISESE. Redondo Beach, USA, 2004. **Proceedings...** Redondo Beach: IEEE Computer Society 2004. p. 127-135.

SOFTEX. **Guia geral do mps.Br**. Melhoria de Processo do Software Brasileiro (Versão 1.1). SOFTEX 2006.

SOUSA, K. D. D.; ANQUETIL, N.; OLIVEIRA, K. M. D. Captura de conhecimento durante a manutenção de software. In: WORKSHOP DE TECNOLOGIA DA INFORMAÇÃO E GERÊNCIA DO CONHECIMENTO. Brasília, Brasil, 2004. **Anais...** Brasília: SBC 2004. p. 1-10.

SPACCO, J.; HOVEMEYER, D.; PUGH, W. Tracking defect warnings across versions. In: INTERNATIONAL WORKSHOP ON MINING SOFTWARE REPOSITORIES - MSR. Shanghai, China, 2006. **Proceedings...** New York: ACM Press 2006. p. 133-136.

TERGAN, S.; KELLER, T. **Knowledge and information visualization: Searching for synergies**. Berlin: Springer, 2005. 385 p.

THELIN, T. Empirical evaluations of usage-based reading and fault content estimation for software inspections. **Empirical Software Engineering**, v. 8, n. 3, p. 309-313, set. 2003.

THELIN, T. *et al.* A replicated experiment of usage-based and checklist-based reading. In: IEEE INTERNATIONAL SYMPOSIUM ON SOFTWARE METRICS - METRICS. Chicago, USA, 2004a. **Proceedings...** Washington: IEEE Computer Society 2004a. p. 246-256.

_____. Evaluation of usage-based reading - conclusions after three experiments. **Empirical Software Engineering**, v. 9, n. 1, p. 77-110, mar. 2004b.

TIDWELL, J. **Designing interfaces**. Sebastopol: O'Reilly Media, Inc., 2005. 352 p.

TILLEY, S. R.; PAUL, S.; SMITH, D. B. Towards a framework for program understanding. In: WORKSHOP ON PROGRAM COMPREHENSION - WPC. Berlin, Germany, 1996. **Proceedings...** Washington: IEEE Computer Society 1996. p. 19-28.

TRAVASSOS, G. *et al.* Detecting defects in object-oriented designs: Using reading techniques to increase software quality. In: ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES, AND APPLICATIONS - OOPSLA. Denver, Colorado, United States, 1999. **Proceedings...** New York: ACM Press 1999. p. 47-56

TRAVASSOS, G. H.; GUROV, D.; AMARAL, E. A. G. D. **Introdução à engenharia de software experimental**. COPPE / UFRJ, 52 p. 2002. (RELATÓRIO TÉCNICO RT–ES–590/02)

TRAVASSOS, G. H. **Revisão e inspeção de software - mini-curso do evento eselaw 2007**. 2007. Disponível em: < http://lens.cos.ufrj.br:8080/eselaw2007/ESELAW07-TRAVASSOS(SC2).pdf >. Acesso em: 29/01/2008.

VINZ, B. L.; ETZKORN, L. H. A synergistic approach to program comprehension. In: INTERNATIONAL CONFERENCE ON PROGRAM COMPREHENSION - ICPC. Athens, Greece, 2006. **Proceedings...** Washington: IEEE Computer Society, Jun. 2006 2006. p. 69-73.

VITHARANA, P.; RAMAMURTHY, K. Computer-mediated group support, anonymity, and the software inspection process: An empirical investigation. **IEEE Transactions on Software Engineering**, v. 29, n. 2, p. 167-180, fev. 2003.

WALKINSHAW, N.; ROPER, M.; WOOD, M. Understanding object-oriented source code from the behavioural perspective. In: INTERNATIONAL WORKSHOP ON PROGRAM COMPREHENSION - IWPC. St. Louis, USA, 2005. **Proceedings...** Washington: IEEE Computer Society, May 2005. p. 215-224.

WINKLER, D.; BIFFL, S. An empirical study on design quality improvement from best-practice inspection and pair programming. In: MÜNCH, J. e VIERIMAA, M. (Ed.). **Product-focused software process improvement**. Berlin: Springer, v.4034, 2006. p. 319-333.

WINKLER, D.; THURNHER, B.; BIFFL, S. Early software product improvement with sequential inspection sessions: An empirical investigation of inspector capability and learning effects. In: EUROMICRO SOFTWARE ENGINEERING AND ADVANCED APPLICATIONS - SEAA. Lübeck, Germany, 2007. **Proceedings...** Washington: IEEE Computer Society 2007. p. 245-254.

WOHLIN, C. *et al.* **Experimentation in software engineering - an introduction**. Sweden: Lund University, 2000. 228 p.

WOJCICKI, M. A.; STROOPER, P. Maximising the information gained from an experimental analysis of code inspection and static analysis for concurrent java components. In: INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING - ISESE. Rio de Janeiro, Brasil, 2006. **Proceedings...** New York: ACM Press 2006. p. 174-183.

WOOD, M. *et al.* Comparing and combining software defect detection techniques: A replicated empirical study. **ACM SIGSOFT Software Engineering Notes**, v. 22, n. 6, p. 262-277, nov. 1997.

Apêndice A

GUIA DE INSTANCIAÇÃO DE NOVAS LINGUAGENS DA FERRAMENTA CRISTA*

Introdução

A ferramenta CRISTA foi inicialmente projetada para inspeção de código Java. Entretanto, a instanciação de outras linguagens é possível, o que torna a ferramenta muito versátil, se adequando à necessidade do usuário.

O objetivo deste tutorial é apresentar os passos necessários para instanciar uma nova linguagem, bem como familiarizar o usuário com os arquivos e diretórios usados pela ferramenta CRISTA. Assim, este tutorial contempla:

- Visão geral do processo de instanciação
- Arquivos utilizados no processo de instanciação
- Modificações necessárias para realizar a instanciação

* Este processo foi retirado do tutorial feito em HTML que acompanha a ferramenta CRISTA. O tutorial original possui links de navegação entre as páginas HTML e que não estão disponíveis nessa versão.

Este processo foi retirado do tutorial feito em HTML que acompanha a ferramenta CRISTA. O tutorial original possui links de navegação entre as páginas HTML e que não estão disponíveis nessa versão.

Requisitos Necessários

Softwares Necessários

Antes de iniciar o processo de instanciação, alguns softwares necessitam estar instalados. Eles são:

1. JavaCC

O Java Compiler Compiler é um gerador de parsers feito em java. O JavaCC le um arquivo de especificação de linguagem (arquivo .jj) e converte em um programa java que reconhece essa gramática. A ferramenta CRISTA usa o arquivo de definição de linguagem do JavaCC para adicionar comandos para montar a estrutura visual. Assim, quando se submete o arquivo .jj, o JavaCC gera classes java compatíveis com a ferramenta CRISTA. Mais informações a respeito do JavaCC pode ser encontrado em: https://javacc.dev.java.net/

2. Apache Ant

O Apache Ant é uma ferramenta de construção Java. De forma geral, o Ant funciona como um Make. Com o Ant, é possível definir tarefas que vão desde a simples cópia de arquivos entre diretórios até a compilação e empacotamento de classes java. A ferramenta CRISTA utiliza o Ant para executar algumas tarefas do processo de instanciação. Assim, para a execução dessas tarefas, é necessário que o Ant esteja instalado e configurado. Mais informações a respeito do Ant estão disponíveis em: http://ant.apache.org/

Artefatos Necessários

Para instanciar uma nova linguagem na ferramenta CRISTA são necessários quatro arquivos. São eles:

Arquivo Language.properties

Script Ant

Arquivo de definição de linguagens

Arquivo de documentação

Todos os arquivos necessários devem estar no mesmo diretório: *CRISTA\languages\{nome_da_lingugem}*.

a) Arquivo Language.properties

Arquivos *Properties* (language.properties) são muito utilizados em programas Java. A função destes arquivos é armazenar um conjunto de pares *chave-valor*, principalmente para inicialização e parametrização de variáveis de configuração.

A ferramenta CRISTA utiliza arquivos *Properties* para definir valores específicos para cada linguagem instanciada, possuindo nove pares *chave-valor*. Exemplos desse tipo de arquivos para as linguagem Java e C++ são mostradas nas Tabelas 1 e 2 respectivamente.

Tabela 1 - Conteúdo do arquivo Properties para a linguagem Java

```
1. VERSION=1.0
2. LANGUAGENAME=Java
3. LANGUAGEPARSERCLASS=JavaParser
4. JAVACCPATH=${basedir}/../../javacc
5. JJFILE=./Java1.5.jj
6. DESTINATIONJar=Java1.5.jar
7. TOOLJARPATH=${basedir}/../../dist/ModulosCrista.jar
8. LANGUAGESDIR=$(basedir)/../../.
9. parser.path=${basedir}/../../src
10. DOCUMENTERCLASS=JavaDocumenter
```

Tabela 2 - Conteúdo do arquivo Properties para a linguagem C++

```
1. VERSION=1.0
2. LANGUAGENAME=C++
3. LANGUAGEPARSERCLASS=CPPParser
4. JAVACCPATH=${basedir}/../../javacc
5. JJFILE=./cpp.jj
6. DESTINATIONJar=cpp.jar
7. TOOLJARPATH=${basedir}/../../dist/ModulosCrista.jar
8. LANGUAGESDIR=$(basedir)/../../.
9. parser.path=${basedir}/../../src
10. DOCUMENTERCLASS=
```

É fácil perceber que os dois arquivos são muito semelhantes. Possuem a mesma quantidade de pares *chave-valor* sendo que muitos deles são idênticos. Isto ocorrerá com o arquivo *Properties* de todas as linguagens que forem instanciadas na ferramenta CRISTA. A seguir será explicada a função de cada par *chave-valor.*

VERSION

A chave *VERSION* recebe um valor simbólico que representa a versão da instanciação da linguagem.

LANGUAGENAME

A chave *LANGUAGENAME* recebe o nome da linguagem.

LANGUAGEPARSERCLASS

A chave *LANGUAGEPARSERCLASS* recebe o nome da classe, gerada pelo JavaCC, que executará o *parser* da linguagem (esta classe sempre recebe o nome *{nome_da_linguagem}Parser*).

JAVACCPATH

A chave *JAVACCPATH* recebe o endereço do diretório onde se encontra o jar do JavaCC.

JJFILE

A chave *JJFILE* recebe o caminho do arquivo .jj da linguagem.

DESTINATIONJar

A chave *DESTINATIONJar* recebe o nome do arquivo jar que será criado. Esse arquivo jar possuirá dentro dele todas as classes geradas pelo JavaCC.

TOOLJARPATH, LANGUAGESDIR e parser.path

As chaves *TOOLJARPATH*, *LANGUAGESDIR* e *parser.path* configuram caminhos para determinados arquivos e diretórios, indispensáveis para o correto funcionamento da ferramenta, e serão idênticos para qualquer linguagem que venha a ser instanciada

DOCUMENTERCLASS

Nome da classe que vai realizar a documentação do código na linguagem específica. Esse atributo é opcional e é discutido em mais detalhes aqui.

Essas chaves são muito importantes para o correto funcionamento do arquivo Build.XML.

 **O arquivo *properties* deve sempre receber o nome *language.properties* e estar localizado no diretório *Crista\languages\{nome_da_lingugem}*.**

b) Script Ant

A ferramenta CRISTA utiliza um arquivo XML (eXtensible Markup Language) para realizar algumas tarefas que precedem a execução do JavaCC. Assim que essas tarefas são cumpridas, o JavaCC é executado, tendo como resultado sete arquivos java, provenientes da execução do JavaCC com o arquivo de definição de linguagem.

Esse arquivo XML é um script Ant, e o seu conteúdo é o mesmo para a

instanciação de qualquer linguagem na ferramenta CRISTA. Esse script Ant é apresentado na Tabela 3.

Tabela 3 - Conteúdo do arquivo XML

```
1 <?xml version='1.0' encoding='ISO-8859-1' ?>
2
3 <projectdefault="create-parser" basedir=".">
4 <propertyfile="language.properties"/>
5
6 <targetname="create-parser" depends="create-files">
7 <echomessage="Building parser of ${LANGUAGENAME}"/>
8 <antcalltarget="jar" />
9 <copytodir="${LANGUAGESDIR}">
10 <filesetdir="." includes="*.jar" />
11 </copy>
12 <delete>
13 <filesetdir=".">
14 <includename="*.jar"/>
15 </fileset>
16 </delete>
17 </target>
18
19 <targetname="copy-parser-files" depends="create-files">
20 <copytodir="${parser.path}">
21 <filesetdir="." includes="*.java" />
22 </copy>
23 </target>
24
25 <targetname="jar" depends="compile">
26 <jardestfile="${DESTINATIONJar}" basedir="."
27 includes="**/*.class,**/language.properties"/>
28 <antcalltarget="clean" />
29 </target>
30
31 <targetname="compile" depends="create-files">
32 <javacsrcdir="." classpath=".;${TOOLJARPATH}"
33 destdir="." >
34 </javac>
35 </target>
36
37 <targetname="create-files" depends="create-files-init" if="parser.gen">
38 <javacctarget="${JJFILE}"
39 outputdirectory="."
40 javacchome="${JAVACCPATH}"/>
41 </target>
42
43 <targetname="create-files-init">
44 <uptodateproperty="parser.gen"
45 targetfile="${JJFILE}">
46 <srcfilesdir="." includes="JavaParser.java"/>
47 </uptodate>
48 </target>
49
50 <targetname="clean">
51 <delete>
52 <filesetdir=".">
53 <includename="*.class"/>
54 </fileset>
55 </delete>
56 </target>
```

```
57
58 </project>
```

O script Ant pode ser executado diretamente com dentro da maioria das ferramentas de desenvolvimento Java existentes. A citar: NetBeans e Eclipse.

O arquivo XML deve sempre receber o nome *build.xml* e estar localizado no diretório *CRISTA\languages\{nome_da_lingugem}*.

c) Arquivo de definição de linguagens

Toda linguagem é composta basicamente de dois componentes: sintaxe e semântica. A sintaxe é, de uma forma simplificada, um conjunto de regras que define como elementos básicos da linguagem (símbolos) podem ser combinados para formar sentenças válidas. A sintaxe não é constituída de significado, ou seja, não revela nada sobre o conteúdo de uma sentença, mesmo que válida. A função de dar significado para as sentenças que formam a linguagem é da semântica.

Sendo as linguagens de programação um tipo de linguagem, estas também possuem sintaxe e semântica. Para a ferramenta CRISTA, apenas a sintaxe é relevante, já que descobrir e analisar a semântica são alguns dos objetivos da inspeção. É no contexto da sintaxe que o arquivo de definição de uma determinada linguagem (.jj) se faz necessário, pois é nele que se encontra o conjunto de regras que definem a sintaxe de uma determinada linguagem.

Para verificar a sintaxe de um código, ou seja, realizar o *parser* do mesmo, é utilizado o JavaCC. Assim, a partir do arquivo .jj que contém as regras de sintaxe de uma determinada linguagem, o JavaCC constrói sete arquivos .java, contendo classes que implementam os analisadores léxicos e sintáticos daquela linguagem, como mostra a Figura 1.

Entre estes arquivos, o principal é nomeado como *{nome_da_linguagem}Parser.java*. Esse arquivo é o responsável em executar o *parser* do código submetido à ferramenta. Por exemplo, no caso das linguagens Java e C++, as classes são *JavaParser.java* e *CPPParser.java*, respectivamente.

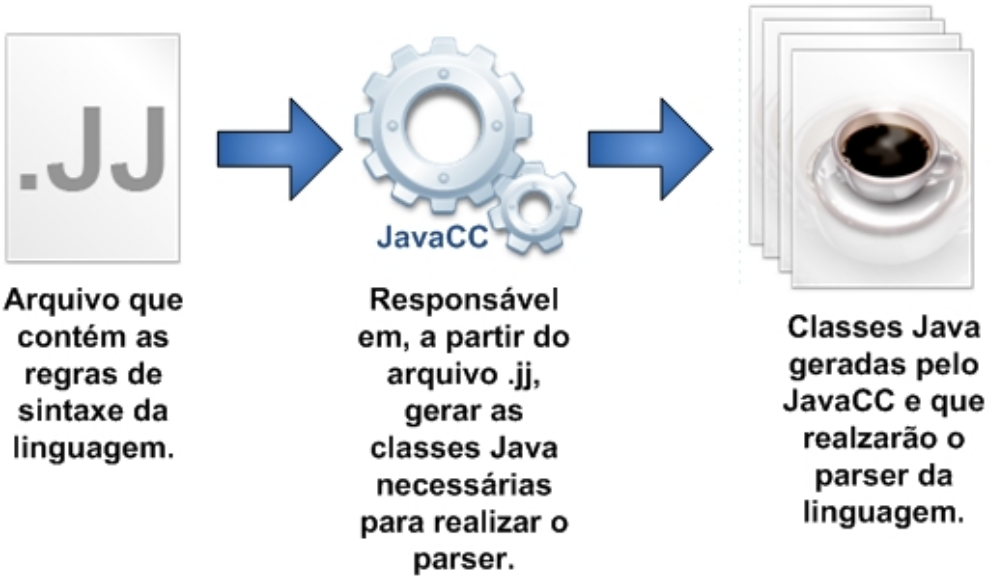


Figura 1 - Representação gráfica do processo de criação do parser

Em alguns casos, pode ser necessário que outras classes sejam inseridas junto com as sete classes geradas pelo JavaCC. Por exemplo, no caso da linguagem C++, que além das classes padrões do JavaCC, foram necessárias outras três. A escolha de inserir outras funcionalidades com essas outras classes, fica a cargo do usuário, que deve avaliar sua necessidade.

O próprio usuário pode construir o arquivo .jj da linguagem a ser instanciada. Entretanto, este processo tende a ser muito árduo, e com grandes chances de erros, uma vez que estes arquivos costumam ser extensos e com muitos detalhes. Os arquivos de definição da linguagem Java (completamente suportada pela ferramenta CRISTA) e da linguagem C e C++ (ambas ainda em fase de testes) foram encontradas no próprio site do JavaCC (https://javacc.dev.java.net/). Neste mesmo endereço há a definição de inúmeras outras linguagens, tais como Visual Basic, Cobol, php, Python, entre outras.

A partir do momento em que o usuário já possui o arquivo que contém a definição de determinada linguagem (arquivo .jj), é necessário inserir algumas instruções Java específicas à ele, que serão utilizadas para montar a metáfora visual do código submetido à ferramenta CRISTA.

As modificações no arquivo de definição de linguagem são necessárias para o passo 1 do processo de instanciação.

Para saber como modificar o arquivo de definição de linguagem, clique aqui.

Exemplo: O arquivo de definição da linguagem Java pode ser visto aqui.

Exemplo: O arquivo de definição da linguagem Java modificado pode ser visto aqui.

Importante: O arquivo de definição de linguagem deve sempre receber o nome *{nome_da_linguagem}.jj* e estar localizado no diretório *CRISTA\languages\{nome_da_lingugem}*.

d) Arquivo de documentação da linguagem

Toda linguagem possui uma sintaxe específica para seus comentários e documentação. A ferramenta CRISTA possui uma classe responsável por escrever comentários de volta no código. Essa classe é a *br.ufscar.dc.lapes.crista.control.report.Documenter*. Essa classe possui todos os métodos e atributos necessários para escrever comentários no código. Entretanto, a classe *Documenter* usa como configurações padrões '/*' e '*/' para início e fim de comentários respectivamente.

No caso de uma linguagem usar parâmetros diferentes para início e fim de comentários, deve-se criar uma classe *Documenter* específica para a linguagem. Por exemplo: Os comentários em cobol possuem o prefixo '*' e não possuem sufixo. No caso dos comentários em Java, os parâmetros '/*' e '*/' servem para comentar códigos Java. Entretanto, para os comentários antes de declarações de métodos e classes, usa-se o prefixo '/**' como padrão javadoc. Nesse caso, foi criado uma classe *Documenter* específica para Java que diferencia comentários antes de métodos e classes dos outros comentários.

Para se criar uma classe *Documenter* específica para determinada linguagem, basta criar uma classe que seja herdeira da classe *br.ufscar.dc.lapes.crista.control.report.Documenter*. Fazendo isso, pode-se sobrescrever os parâmetros e métodos desejados de forma a documentar de acordo com as definições da linguagem.

Exemplo: O arquivo de documentação da linguagem Java aqui.

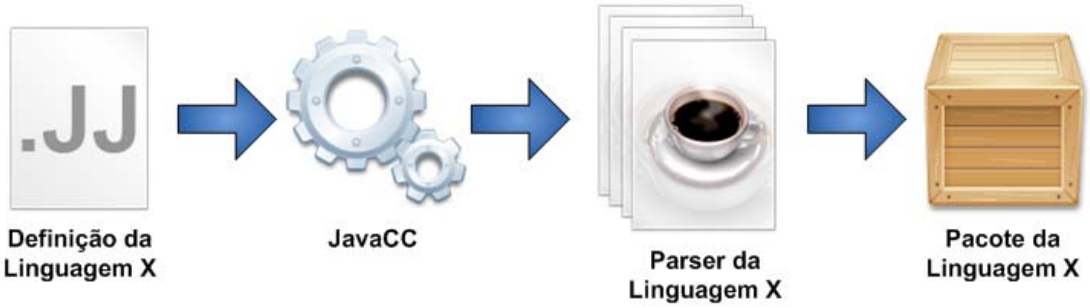
O arquivo de documentação não é obrigatório para a realização do parser e deve ser criado somente para a correta geração de código comentado.

Importante: O arquivo de documentação da linguagem deve sempre herdar da classe *br.ufscar.dc.lapes.crista.control.report.Documenter* e estar localizado no diretório *CRISTA\languages\{nome_da_lingugem}*.

Ao se criar uma classe *Documenter* específica, deve-se inseri-la como valor da chave DOCUMENTERCLASS do arquivo de propriedades da linguagem.

Processo de instanciação

Mapa de Instanciação



Processo básico

Para instaciar uma nova linguagem na ferramenta CRISTA, basta seguir os seguintes passos:

1) Modificar o arqivo de definição da linguagem desejada (Arquivo JJ).

2) Submeter o arquivo de definição da linguagem (Arquivo JJ) ao JavaCC, para gerar o parser da linguagem X.

3) Empacotar as classes geradas pelo JavaCC em um arquivo jar.

É importante lembrar que para a execução desse processo são necessários alguns requisitos básicos. Esses requisitos podem ser encontrados aqui.

Passo 1 - Adaptar o arquivo de definição da linguagem

Atividade Adaptar o arquivo de definição da linguagem

Objetivo Inserir no arquivo de definição da linguagem os comandos necessários para a criação da metáfora visual na ferramenta CRISTA

Entrada • Arquivo de definição da linguagem (Exemplo)

Saída Arquivo de definição da linguagem modificado (Exemplo)

Procedimento

Nesse passo do processo de instanciação, deve-se realizar as modificações necessárias no arquivo de definição da linguagem, de forma que o mesmo seja capaz de construir a metáfora visual do código fonte após realizar o parser do mesmo.

Para saber como modificar o arquivo de definição da linguagem clique aqui.

Passo 2 - Gerar o parser da linguagem

Atividade Gerar o parser da linguagem

Objetivo Executar o JavaCC para gerar as classes necessárias para a realização do *parser*

Entrada • Arquivo de definição da linguagem modificado

• build.xml

• language.properties

Saída Classes necessárias para a realização do *parser*

Procedimento

Para criar o parser, deve-se executar o JavaCC com o arquivo de definição da linguagem com as modificações necessárias.

Para executar o JavaCC, é necessário executar a tarefa Ant '*create-files*' definida no arquivo build.xml.

Para a execução desse passo, é necessária a correta instalação dos softwares JavaCC e Ant.

Após a execução da tarefa Ant '*create-files*' do arquivo build.xml, o JavaCC, gera 7 classes java que

realizam o parser da linguagem. Os arquivos gerados para a linguagem Java são:

- JavaParser.java
- JavaParserTokenManager.java
- JavaParserConstants.java
- JavaCharStream.java
- TokenMgrError.java
- ParseException.java
- TokenMgrError.java

De forma semelhante, para as outras linguagens vão ser geradas as classes correspondentes. Vale lembrar que são as modificações feitas no arquivo de definição da linguagem que torna o *parser* capaz de criar a metáfora visual do código fonte.

Todos os artefatos de entrada devem estar no diretório CRISTA\languages\{nome_da_lingugem}

Todas as classes java que realizam o parser da linguagem também são geradas no diretório CRISTA\languages\{nome_da_lingugem}

Passo 3 - Empacotar as classes geradas pelo JavaCC em um arquivo jar

Atividade	Empacotar as classes geradas pelo JavaCC em um arquivo jar
Objetivo	Empacotar todos os arquivos necessários para a execução do parser da linguagem
Entrada	• Arquivos gerados pelo JavaCC no passo 2 • build.xml • language.properties • Arquivo de documentação da linguagem (Opcional)
Saída	Arquivo Jar contendo todas os arquivos necessários para a correta execução do parser na ferramenta CRISTA

Procedimento

Para gerar o arquivo Jar contendo os arquivos necessários para realizar o *parser* da linguagem, é necessário executar a tarefa Ant '*create-parser*' definida no arquivo build.xml.

Assim como no passo 2, é necessária a correta instalação dos softwares JavaCC e Ant.

Após a execução da tarefa Ant '*create-parser*' do arquivo build.xml, vai ser gerado um arquivo Jar contendo os arquivos necessários para a realização do *parser*. Os arquivos para a linguagem Java são:

- JavaCharStream.class
- JavaDocumenter.class (Classe de documentação opcional)
- JavaParser$1.class

- JavaParser$LookaheadSuccess.class
- JavaParser$ModifierSet.class
- JavaParser.class
- JavaParserConstants.class
- JavaParserTokenManager.class
- language.properties
- ParseException.class
- Token$GTToken.class
- Token.class
- TokenMgrError.class

De forma semelhante, para as outras linguagens vão ser geradas as classes correspondentes.

 **Todos os artefatos de entrada devem estar no diretório CRISTA\languages\{nome_da_lingugem}**

 **O arquivo Jar de saída também é gerado no diretório CRISTA\languages\{nome_da_lingugem}**

Modificações do arquivo de definição de linguagem (.jj)

Introdução

Nessa seção são apresentadas e explicadas as instruções Java que devem ser inseridas no arquivo de definição da linguagem a ser instanciada. Essas instruções são necessárias para que, ao realizar o *parser*, a ferramenta CRISTA possa reunir as informações necessárias para construir a metáfora visual do código submetido à ela.

Entendendo o arquivo JJ

Pode-se dividir o arquivo de definição de linguagem em dois blocos. O primeiro, entre *PARSER_BEGIN()* e *PARSER_END()*, contém as instruções que são utilizadas pelo JavaCC para construir os arquivos responsáveis pelo *parser* do código submetido a ferramenta. O segundo bloco contém as regras de sintaxe que compõe a linguagem. Ambos os blocos precisão conter algumas instruções Java específicas para que a ferramenta CRISTA funcione corretamente.

Alterando o arquivo JJ

Alterações iniciais

A primeira alteração que o usuário deve fazer é inserir algumas linhas de *import* logo abaixo da declaração *PARSER_BEGIN()*. São elas:

Tabela 4 - Imports necessários ao arquivo de definição de linguagem (.jj)

```
1 PARSER_BEGIN()
2
3 importjava.io.*;
4 importbr.ufscar.dc.lapes.crista.model.languages.Statement;
5 importbr.ufscar.dc.lapes.crista.model.languages.LanguageParser
6 importbr.ufscar.dc.lapes.crista.model.languages.BlockDelimiter;
```

As classes contidas em *java.io* (linha 3) são utilizadas para entrada e saída de dados.

A classe *Statement* (linha 4), própria da ferramenta CRISTA, serve para representar uma instrução ou um bloco de instruções de algum código submetido à ferramenta. A Figura 2 representa graficamente seu funcionamento.

Figura 2 - Modo como a classe Statement trabalha, identificando cada bloco de instrução

As letras A, B, C, D, E e F representam as instruções ou bloco de instruções que compõe o método usado como exemplo na Figura 2. Pode-se reparar que o bloco de instrução A contém, dentro de si, todas as outras instruções ou blocos de instruções. O mesmo acontece com B, que contém os bloco C e a instrução E. O bloco de instrução C por sua vez, contém a instrução E.

A classe *Statement* é bastante usada no processo de *parser* de qualquer linguagem. As Figuras 3 e 4 mostram o diagrama UML das classes *Statemente* e *BlockDelimiter*, respectivamente. A classe *Statement* possui oito atributos, sendo que os mais relevantes a este tutorial são: *insideStatement* (lista de objetos da classe *Statement*) e *blockDelimiter* (objeto da classe *BlockDelimiter*). O objeto *insideStatement* é o responsável em gerenciar a cadeia de instruções de um determinado trecho de código, por exemplo, na Figura 2, a instrução E está contida no bloco de instruções C, que por sua vez está contido no bloco de instrução B, que por sua vez está contido no bloco de instruções A.

Figura 3 – Representação UML da classe Statement

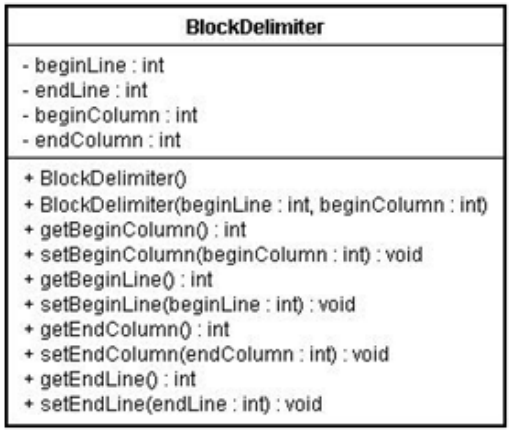


Figura 4 – Representação UML da classe BlockDelimiter

Já o objeto *blockDelimiter*, é o responsável em registrar a linha e coluna de início e fim de cada instrução ou bloco de instruções. Isto fica bem claro quando a representação UML da classe *BlockDelimiter* é analisada. A classe possui apenas quatro atributos, sendo que dois deles são utilizados para registrar o inicio da instrução (linha e coluna) e dois para registrar o fim da instrução (linha e coluna).

A classe *LanguageParser* (linha 5 da Tabela 4) é uma classe genérica para realizar o *parser* de qualquer linguagem que se queira instanciar. Desta maneira, esta classe deve ser sempre herdada pela classe *{nome_da_linguagem}Parser*, que está declarada no arquivo de definição de linguagem (.jj). Por exemplo, no arquivo de definição da linguagem Java deverá haver a seguinte declaração de classe:

Tabela 5 – Alteração para que a classe JavaParser herde a classe LanguageParser

```
1 public class JavaParser extends LanguageParser
2 {
3 //instruções que compõe a classe
4 }
```

Na Tabela 5, o código escrito em preto refere-se ao arquivo de definição da linguagem Java original, ou seja, sem alteração. O código em vermelho refere-se à alteração que o usuário deverá realizar.

A próxima alteração necessária é a inserção ou modificação do *método construtor* da classe *{nome_da_linguagem}Parser*. A Tabela 6 mostra, como exemplo, o *método construtor* inserido na classe *JavaParser* do arquivo de definição da linguagem Java. Para a instanciação de outras linguagens, basta alterar a assinatura do método, deixando o nome do *método construtor* igual ao nome da classe que o possui.

Tabela 6 - Método construtor que compõe a classe JavaParser do arquivo de definição da linguagem Java

```
1 publicJavaParser (String fileName) throwsFileNotFoundException
2 {
3 this( newFileInputStream( newFile( fileName ) ) );
4 setLanguageName( newString ( “Java” ) );
5 }
```

Este *método construtor* recebe um único parâmetro (*fileName*), do tipo *String*, que recebe o endereço do arquivo submetido à ferramenta CRISTA. Caso esse endereço seja inválido, o método lança uma exceção *FileNotFoundException*. A linha 3 contém uma chamada para um *método construtor* que é criado quando o JavaCC realiza o processo de transformação do arquivo de definição de linguagem para os arquivos .java. Já o código da linha 4 configura o nome da linguagem, utilizando-se de um método que pertence à classe herdada *LanguageParser*.

A alteração seguinte, assim como a anterior, deve ser feita dentro da definição da classe *{nome_da_linguagem}Parser* no arquivo .jj, porém esta é um pouco mais delicada. Como dito anteriormente, o arquivo de definição de linguagem reúne as regras que estabelecem como os mais variados símbolos que compõe a linguagem podem relacionar, formando sentenças válidas. Estas regras são estabelecidas através de métodos. Entre eles há um especial, aquele que inicia a verificação de toda a sintaxe, e que a partir deste ponto será referenciado como *método inicial*.

A alteração consiste em inserir um método a classe *{nome_da_linguagem}Parser*, porém há uma linha de comando que varia de acordo com o arquivo de definição de linguagem que o usuário está utilizando. Esta linha de comando é justamente uma chamada para o *método inicial*, podendo a assinatura deste método variar de um arquivo para outro.

Uma característica do arquivo de definição de linguagem é que, em geral, a classe *{nome_da_linguagem}Parser* possui um método *main*, o qual possui sempre uma chamada para o *método inicial*. Durante a instanciação da linguagem Java e C++, foi possível notar uma semelhança entre as assinaturas dos respectivos *métodos iniciais*. No arquivo de definição da linguagem Java, o método em questão tem o nome *CompilationUnit()*, já em C++, *translation_unit()*. Em ambos os arquivos, o *método inicial* apareceu anterior a todos os outros métodos que compõem o arquivo de definição da linguagem.

 **Infelizmente, não é possível afirmar que isto ocorra como uma regra, portanto, o usuário deve ter a sensibilidade de identificar entre todos os métodos, aquele que inicia a verificação da sintaxe.**

Uma vez identificado o *método inicial*, basta inserir o método mostrado na Tabela 7 na classe *{nome_da_linguagem}Parser*. Esse método é uma sobrecarga do mesmo método da classe LanguageParser.

Tabela 7 - Método que deve ser inserido na classe {nome_da_linguagem}Parser

```
1 publicStatement parseFile() throwsException
2 {
3 CHAMADA_DO_MÉTODO_INICIAL();
4 fileStatements.setEndLine(jj_input_stream.getEndLine());
5 fileStatements.setEndColumn(jj_input_stream.getEndColumn());
6 returnfileStatements;
7 }
```

A linha 1 da Tabela 7 mostra a assinatura do método, que possui as seguintes características: acesso *public*, retorna um objeto do tipo *Statement*, tem o nome *parseFile()*, não recebe parâmetro algum e em caso de falha, lança exceção. A linha 3 contém a chamada do *método inicial*, e como dito anteriormente, este ponto é o mais delicado e que merece maior atenção do usuário. As linhas 4 e 5 utilizam o objeto *fileStatements* (herdado da classe *LanguageParser*), que é do tipo *Statement* (classe que foi importada, como mostra o início desta seção) e os métodos *setEndLine(int)* e

setEndColumn(int), ambos métodos *public*, pertencentes também a classe *Statement*. Esses dois últimos métodos configuram no objeto *fileStatements* o limite do arquivo que contém o código submetido a ferramenta, ou seja, configura o valor inteiro correspondente a última linha e coluna de código. E finalmente, a linha 6 retorna o objeto *fileStatements*.

Os métodos *setEndLine(int)* e *setEndColumn(int)* recebem o parâmetro *jj_input_stream.getEndLine()* e *jj_input_stream.getEndComumn()*, respectivamente. O objeto *jj_input_stream* é criado durante a transformação do arquivo de definição da linguagem (arquivo .jj) para os arquivos .java, realizada pelo JavaCC. Neste mesmo processo, os métodos *getEndLine()* e *getEndColumn()* são gerados, bem como os métodos *getBeginLine()* e *getBeginColumn()*. Estes dois últimos são utilizados nas próximas alterações, a fim de se obter a linha e a coluna de início de cada instrução ou bloco de instruções.

A título de exemplo, as Tabelas 8 e 9 mostram o método inserido no arquivo de definição da linguagem Java e C++, respectivamente.

Tabela 8 – Arquivo de definição da linguagem Java

```
1 publicStatement parseFile() throwsException
2 {
3 CompilationUnit();
4 fileStatements.setEndLine(jj_input_stream.getEndLine());
5 fileStatements.setEndColumn(jj_input_stream.getEndColumn());
6 returnfileStatements;
7 }
```

Tabela 9 - Arquivo de definição da linguagem C++

```
1 publicStatement parseFile() throwsException
2 {
3 translation_unit();
4 fileStatements.setEndLine(jj_input_stream.getEndLine());
5 fileStatements.setEndColumn(jj_input_stream.getEndColumn());
6 returnfileStatements;
7 }
```

Alterações na sintaxe

As alterações, deste ponto em diante, são feitas exclusivamente nos métodos responsáveis em verificar a sintaxe da linguagem. Infelizmente, como o conteúdo dos arquivos de definição de linguagem variam de uma linguagem para outra, não há um modo exato de como realizar as alterações. Assim, este tutorial visa esclarecer o conceito, e usar as alterações realizadas no arquivo de definição da linguagem Java como exemplo.

Para que a ferramenta CRISTA consiga identificar os mais diferentes tipos de expressões, blocos condicionais, blocos de repetição etc. que compõe uma determinada linguagem, é preciso que algumas instruções Java sejam inseridas no arquivo de definição desta linguagem. Estas instruções têm como objetivo cercar os métodos que analisam a sintaxe do código submetido à ferramenta, registrando quando tais métodos começam e quando terminam.

Em geral, uma linguagem é constituída por dois tipos de instruções: as compostas, que são formadas por blocos (*if..else*, *for*, *do..while*, etc.) e as simples, que são possuem em geral uma única linha de comando (atribuição, declaração de variáveis, etc.).

A ferramenta CRISTA trata essas duas expressões de modos ligeiramente diferentes, pois quando uma instrução pode conter outras dentro de si, a ferramenta deve ser capaz de identificar e gerenciar tal evento. Ao passo que, quando uma instrução não pode englobar outras, sendo mais simples, a ferramenta não precisa gerenciar tal evento, economizando linhas de comando.

De maneira geral, cada método para identificar uma regra na sintaxe de uma linguagem possui a

seguinte forma:

Tabela 10 - Exemplo de um método para reconhecer uma produção

```
1 nomeDaRegra()
2 {}
3 {
4 declarações e símbolos que serão reconhecidos por essa produção.
5 }
```

As Tabelas 11 e 12 mostram o código que deve ser inserido em métodos que analisam instruções compostas, ou seja, que podem possuir outras instruções dentro de si. A Tabela 11 mostra o código que registra as informações quando o método é iniciado. Já a Tabela 12, o código que registra as informações quando o método é finalizado.

Tabela 11 - Código responsável em registrar o inicio de uma instrução composta

```
1 {
2 BlockDelimiter bd = new BlockDelimiter(jj_input_stream.getBeginLine(), jj_input_stream.getBeginColumn());
3 Statement s = new Statement(actualStatement, bd , "{descrição_da_instrução}");
4 actualStatement.addInsideStatement(s);
5 actualStatement=s;
6 }
```

Tabela 12 - Código responsável em registrar o fim de uma instrução composta

```
1 {
2 actualStatement = actualStatement.getParentStatement();
3 bd.setEndLine(jj_input_stream.getEndLine());
4 bd.setEndColumn(jj_input_stream.getEndColumn());
5 }
```

Em ambos os códigos, um objeto é bastante usado: *actualStatement.* Este objeto é uma instanciação da classe *Statement* e é responsável em guardar as informações da instrução atual, ou seja, a instrução que esta sendo verificada.

 **No contexto do método para verificar uma determinada produção da linguagem (Tabela 10), o código da Tabela 11 deveria ser inserido substituindo a linha 2 da Tabela 10. Da mesma forma, o código apresentado na Tabela 12 deveria ser inserido entre as linhas 4 e 5 da Tabela 10.**

A linha 2 da Tabela 11 cria o objeto *bd*, instância da classe *BlockDelimiter*, com os parâmetros *jj_input_stream.getBeginLine()* e *jj_input_stream.getBeginColumn()*, ou seja, a linha e a coluna em que a instrução começa. A linha 3 cria um objeto *s* da classe *Statement*, com três parâmetros: *actualStatement*, objeto do tipo *Statement*, que neste momento contém as informações da instrução que engloba a instrução atual; *bd*, que contém os valores da linha e da coluna de início da instrução; e um objeto *String*, que recebe uma descrição da instrução, por exemplo, *"Declaração de variável"* ou *"Início de classe"*. A linha 4 contém uma chamada de método da classe *Statement*, realizada pelo objeto *actualStatement*, para adicionar o objeto *s* a um *array* (vetor). Isto possibilita que, em outro momento, as informações das instruções mais globais sejam recuperadas. Finalmente, a linha 5 atribui o objeto *s* ao objeto *actualStatement*.

Já na Tabela 12, a linha 2 atribui ao objeto *actualStatement* as informação da instrução que engloba a instrução que acabou de ser verificada. As linhas 3 e 4 configuram no objeto *bd* o valor da linha e da coluna, respectivamente, que corresponde ao fim da instrução que foi verificada.

As Tabelas 13 e 14 representam o código que deve ser inserido em métodos que analisam instruções simples, ou seja, não podem englobar outras instruções. A Tabela 13 mostra o código que registra as informações quando o método é iniciado. Já a Tabela 14, o código que registra as informações

quando o método é finalizado.

Tabela 13 - Código responsável em registrar o inicio de uma instrução simples

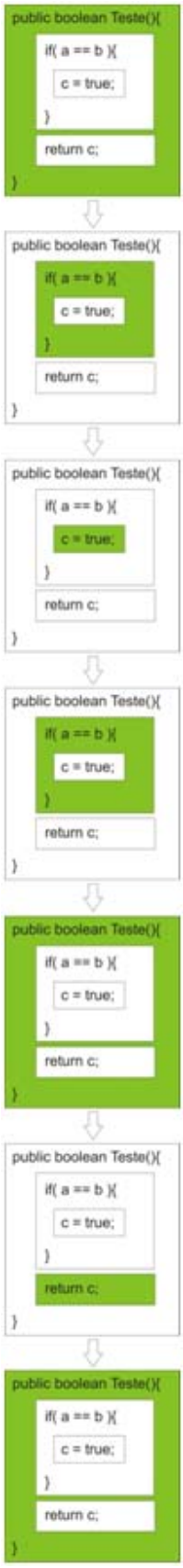
```
1 {
2 BlockDelimiter bd = new BlockDelimiter(jj_input_stream.getBeginLine(), jj_input_stream.getBeginColumn());
3 Statement s = new Statement(actualStatement, bd , "{descrição_da_instrução}");
4 actualStatement.addInsideStatement(s);
5 }
```

Tabela 14 - Código responsável em registrar o fim de uma instrução simples

```
1 {
2 bd.setEndLine(jj_input_stream.getEndLine());
3 bd.setEndColumn(jj_input_stream.getEndColumn());
4 }
```

Entre os códigos apresentados pelas Tabelas 11 e 12 e o código apresentado pelas Tabelas 13 e 13 há apenas duas diferenças. Para obter as informações de instruções simples não é necessário a linha 5 da Tabela 11, nem a linha 2 da Tabela 12. Isso porque essas linhas são as responsáveis em gerenciar instruções compostas.

A Figura 5 representa graficamente como um código, neste exemplo em Java, é tratado pela ferramenta. Desta maneira, todas as linhas do código são analisadas pelas classes geradas pelo JavaCC, que utilizou o arquivo de definição de linguagem como modelo. Códigos de outras linguagens são tratados da mesma forma.



- *actualStatement*: método Teste;
- Registra a linha e coluna de início do método;
- Verifica a sintaxe do cabeçalho do método;
- Procura por instruções dentro do método. Se houver, inicia a verificação dos mesmos.

- *actualStatement*: bloco IF;
- Registra a linha e coluna do bloco IF;
- Verifica a sintaxe do cabeçalho do comando;
- Procura por instruções dentro do bloco. Se houver, inicia a verificação dos mesmos.

- *actualStatement*: instrução de atribuição;
- Registra a linha e coluna de início da instrução;
- Verifica a sintaxe da instrução;
- Como este tipo de instrução não pode englobar nenhuma outra, é registrado a linha e coluna de fim da instrução e o objeto *actualStatement* volta para a instrução mais geral.

- *actualStatement*: bloco IF;
- Procura por outras instruções dentro do bloco;
- Como não há nenhuma outra instrução, é registrada a linha e coluna de fim do bloco IF;
- O objeto *actualStatement* volta para a instrução mais geral.

- *actualStatement*: método Teste;
- Procura outras instruções dentro do bloco. Se houver, inicia a verificação dos mesmos.

- *actualStatement*: instrução *return*;
- Registra a linha e coluna de início da instrução;
- Verifica a sintaxe da instrução;
- Como este tipo de instrução não pode englobar nenhuma outra, é registrado a linha e coluna de fim da instrução e o objeto *actualStatement* volta para a instrução mais geral.

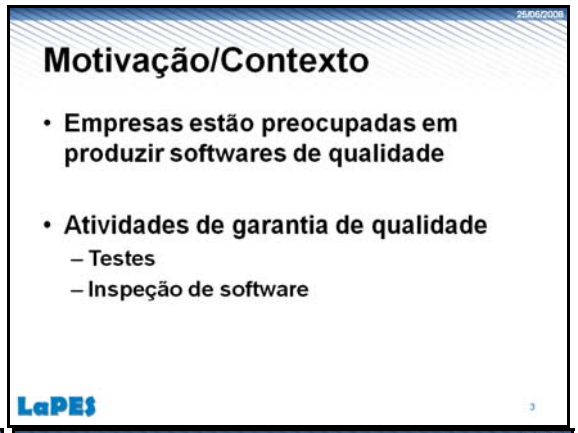
- *actualStatement*: método Teste;
- Procura outras instruções dentro do bloco. Se houver, inicia a verificação dos mesmos.
- Como não há nenhuma outra instrução, é registrada a linha e coluna de fim do método;

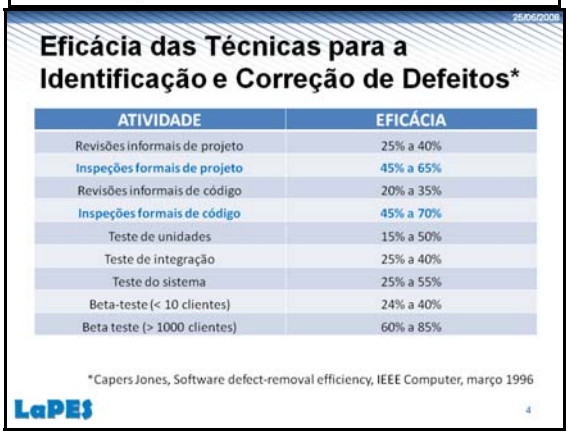
Figura 5 - Processo realizado pela ferramenta CRISTA ao executar o parser de qualquer código submetido à ela

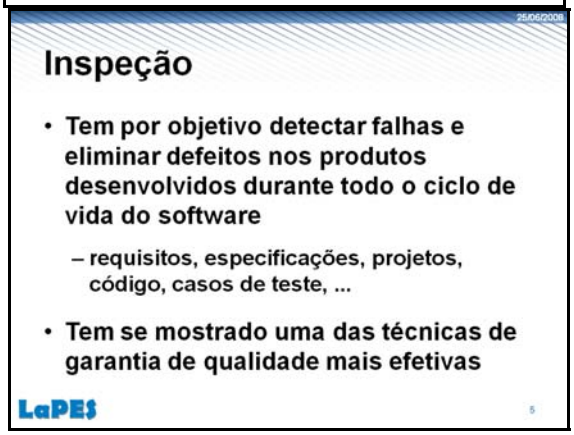
Apêndice B

APRESENTAÇÕES UTILIZADAS NOS ESTUDOS EXPERIMENTAIS 1 E 2

Treinamento sobre Inspeção e sobre a técnica *Stepwise Abstraction*

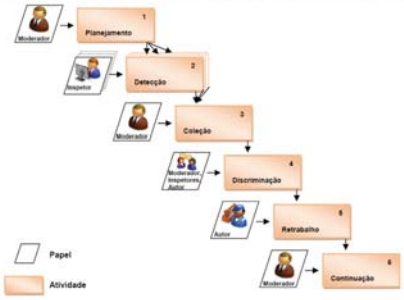






25/06/2008

Processo de Inspeção



LaPES 6

25/06/2008

Inspeção

- Técnicas de Inspeção de Software
  - Dependendo da técnica usada, o processo de inspeção pode ser mais ou menos efetivo
    - Ad-hoc
    - Checklist
    - Técnicas de Leitura

LaPES 7

25/06/2008

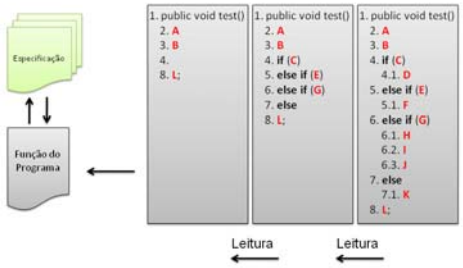
Stepwise Abstraction

- O objetivo dessa técnica de leitura é determinar as funcionalidades do programa (código fonte) a partir das abstrações funcionais que são geradas a partir do código fonte

LaPES 8

25/06/2008

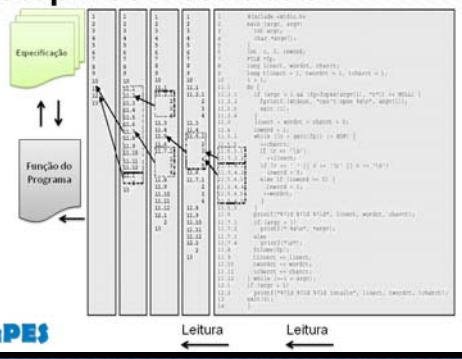
Stepwise Abstraction



LaPES 9

25/06/2008

Stepwise Abstraction



LaPES 10

25/06/2008

Exemplo

```
1.  int num;
2.  int minimo = Integer.MAX_VALUE;
3.  int maximo = Integer.MIN_VALUE;

4.  BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
5.  System.out.println("Entre com o primeiro numero:");
6.  num = br.read();
7.  if(num < minimo)
        7.1.    minimo = num;

8.  if(num > maximo)
        8.1.    maximo = num;

9.  System.out.println("Entre com o segundo numero:");
10. num = br.read();
11. if(num < minimo)
        11.1.   minimo = num;

12. if(num > maximo)
        12.1.   maximo = num;

13. System.out.println("Entre com o terceiro numero:");
14. num = br.read();
15. if(num < minimo)
        15.1.   minimo = num;

16. if(num > maximo)
        16.1.   maximo = num;

17. System.out.println("O menor numero eh: " + minimo);
18. System.out.println("O maior numero eh: " + maximo);
```

LaPES 11

25/06/2008

Exemplo

```
1.  int num;
2.  int minimo = Integer.MAX_VALUE;
3.  int maximo = Integer.MIN_VALUE;

4.  BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
5.  System.out.println("Entre com o primeiro numero:");
6.  num = br.read();
7.  if(num < minimo)
        7.1.    minimo = num;

8.  if(num > maximo)
        8.1.    maximo = num;

9.  System.out.println("Entre com o segundo numero:");
10. num = br.read();
11. if(num < minimo)
        11.1.   minimo = num;

12. if(num > maximo)
        12.1.   maximo = num;

13. System.out.println("Entre com o terceiro numero:");
14. num = br.read();
15. if(num < minimo)
        15.1.   minimo = num;

16. if(num > maximo)
        16.1.   maximo = num;

17. System.out.println("O menor numero eh: " + minimo);
18. System.out.println("O maior numero eh: " + maximo);
```

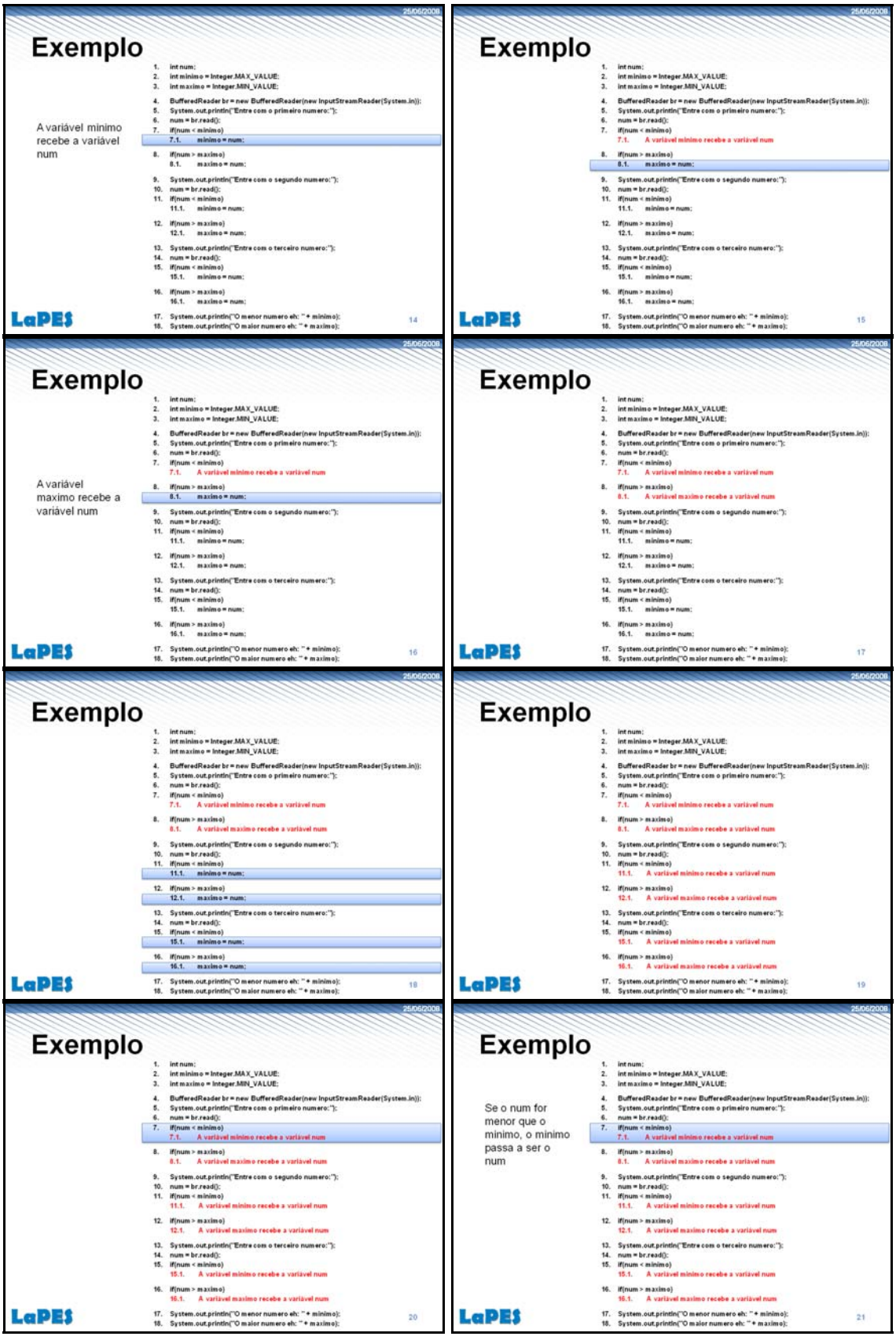
LaPES 12

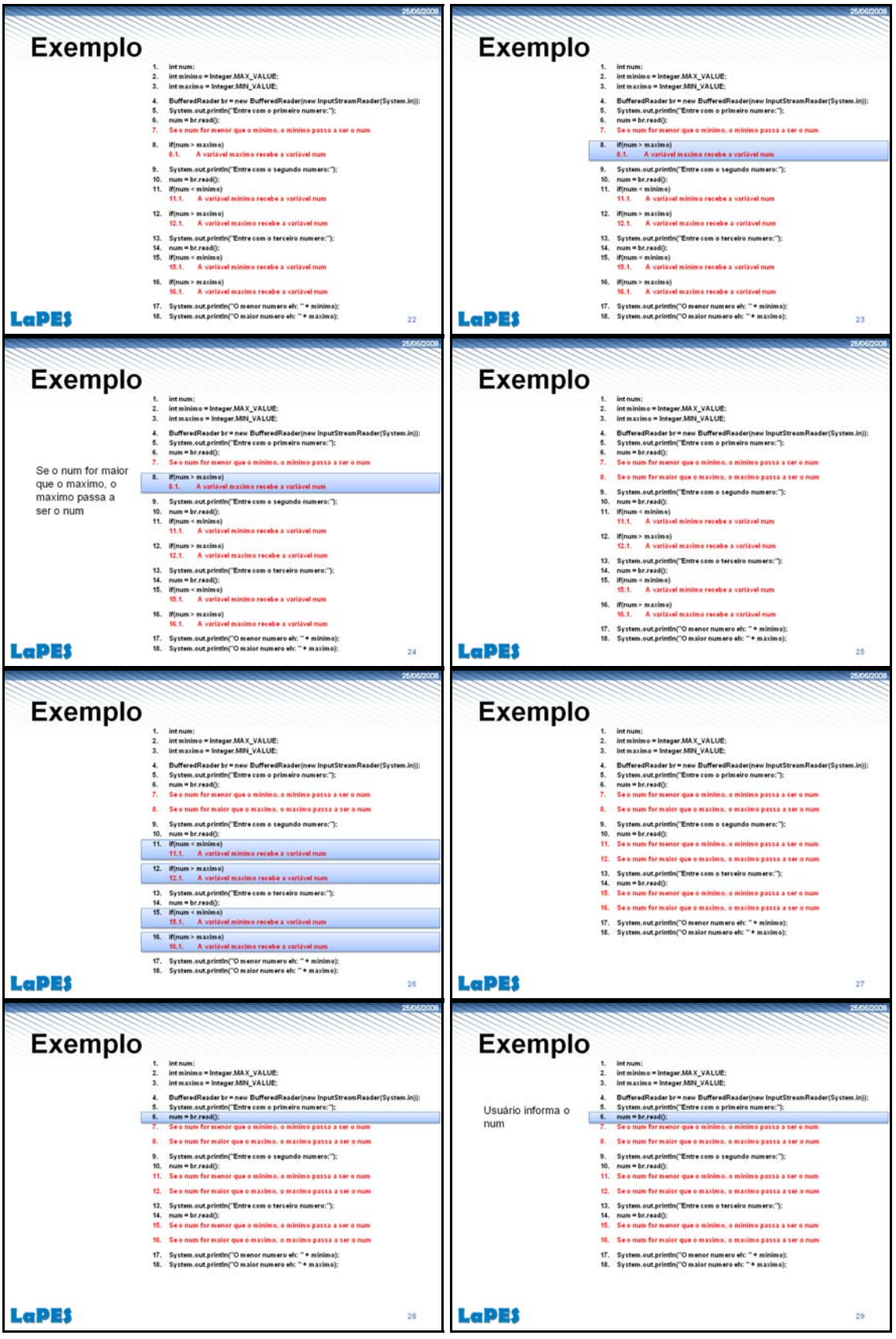
25/06/2008

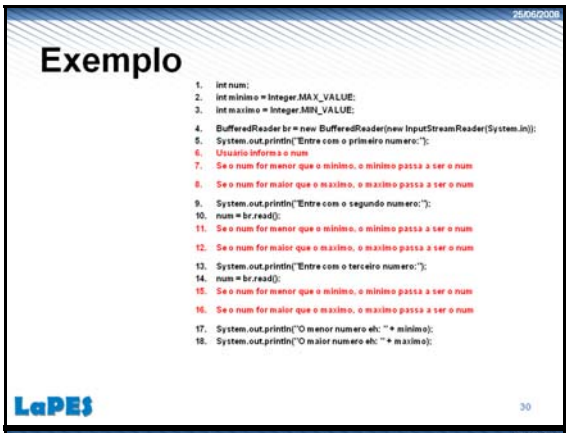
Exemplo

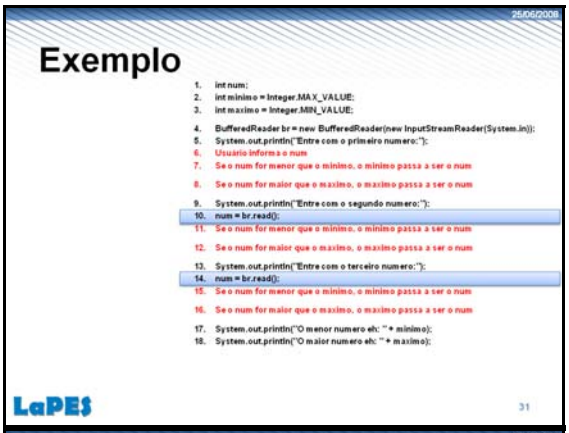
```
1.  int num;
2.  int minimo = Integer.MAX_VALUE;
3.  int maximo = Integer.MIN_VALUE;

4.  BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
5.  System.out.println("Entre com o primeiro numero:");
6.  num = br.read();
7.  if(num < minimo)
        7.1.    minimo = num;

8.  if(num > maximo)
        8.1.    maximo = num;

9.  System.out.println("Entre com o segundo numero:");
10. num = br.read();
11. if(num < minimo)
        11.1.   minimo = num;

12. if(num > maximo)
        12.1.   maximo = num;

13. System.out.println("Entre com o terceiro numero:");
14. num = br.read();
15. if(num < minimo)
        15.1.   minimo = num;

16. if(num > maximo)
        16.1.   maximo = num;

17. System.out.println("O menor numero eh: " + minimo);
18. System.out.println("O maior numero eh: " + maximo);
```

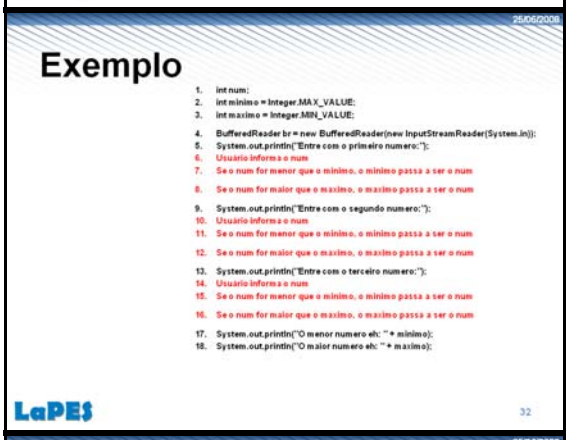
LaPES 13

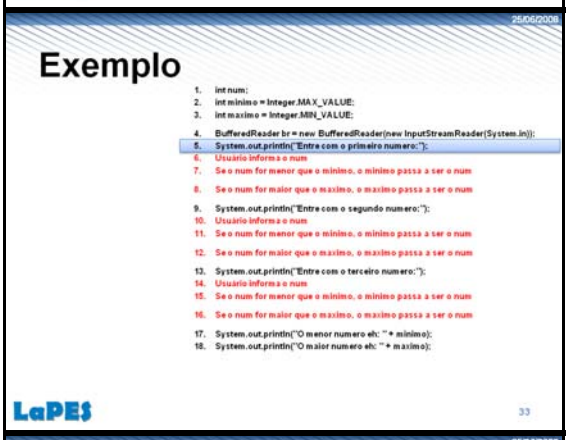

25/06/2008
Exemplo
1. int num;
2. int minimo = Integer.MAX_VALUE;
3. int maximo = Integer.MIN_VALUE;
4. BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
5. System.out.println("Entre com o primeiro numero:");
6. num = br.read();
7. if(num < minimo)
7.1. minimo = num;
8. if(num > maximo)
8.1. maximo = num;
9. System.out.println("Entre com o segundo numero:");
10. num = br.read();
11. if(num < minimo)
11.1. minimo = num;
12. if(num > maximo)
12.1. maximo = num;
13. System.out.println("Entre com o terceiro numero:");
14. num = br.read();
15. if(num < minimo)
15.1. minimo = num;
16. if(num > maximo)
16.1. maximo = num;
17. System.out.println("O menor numero eh: " + minimo);
18. System.out.println("O maior numero eh: " + maximo);
A variável minimo recebe a variável num
LaPES
14
Exemplo
7.1. A variável minimo recebe a variável num
LaPES
15
Exemplo
A variável maximo recebe a variável num
8.1. maximo = num;
LaPES
16
Exemplo
8.1. A variável maximo recebe a variável num
LaPES
17
Exemplo
11.1. minimo = num;
12.1. maximo = num;
LaPES
18
Exemplo
11.1. A variável minimo recebe a variável num
12.1. A variável maximo recebe a variável num
15.1. A variável minimo recebe a variável num
16.1. A variável maximo recebe a variável num
LaPES
19
Exemplo
7.1. A variável minimo recebe a variável num
8.1. A variável maximo recebe a variável num
11.1. A variável minimo recebe a variável num
12.1. A variável maximo recebe a variável num
15.1. A variável minimo recebe a variável num
16.1. A variável maximo recebe a variável num
LaPES
20
Exemplo
Se o num for menor que o minimo, o minimo passa a ser o num
LaPES
21

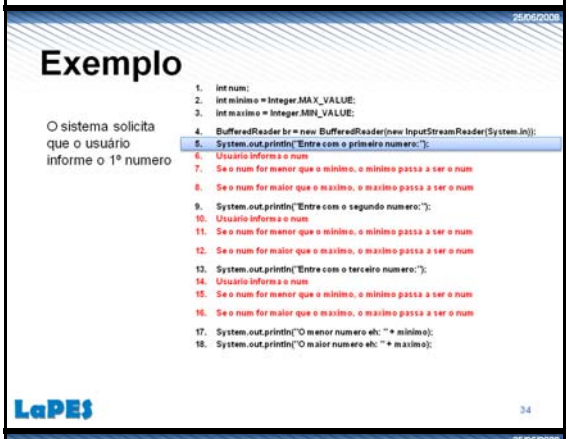

25/06/2008
Exemplo
1. int num;
2. int minimo = Integer.MAX_VALUE;
3. int maximo = Integer.MIN_VALUE;
4. BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
5. System.out.println("Entre com o primeiro numero:");
6. num = br.read();
7. Se o num for menor que o minimo, o minimo passa a ser o num
8. if(num > maximo)
8.1. A variável maximo recebe a variável num
9. System.out.println("Entre com o segundo numero:");
10. num = br.read();
11. if(num < minimo)
11.1. A variável minimo recebe a variável num
12. if(num > maximo)
12.1. A variável maximo recebe a variável num
13. System.out.println("Entre com o terceiro numero:");
14. num = br.read();
15. if(num < minimo)
15.1. A variável minimo recebe a variável num
16. if(num > maximo)
16.1. A variável maximo recebe a variável num
17. System.out.println("O menor numero eh: " + minimo);
18. System.out.println("O maior numero eh: " + maximo);
LaPES
22
Exemplo
Se o num for maior que o maximo, o maximo passa a ser o num
LaPES
24
Exemplo
Se o num for menor que o minimo, o minimo passa a ser o num
Se o num for maior que o maximo, o maximo passa a ser o num
LaPES
26
Exemplo
Usuário informa o num
LaPES
28

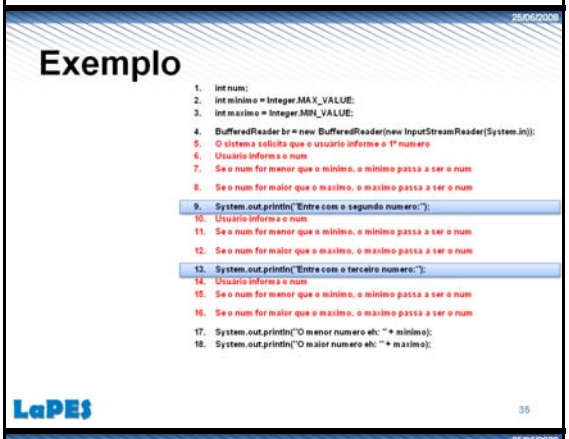

25/06/2008
Exemplo
1. int num;
2. int minimo = Integer.MAX_VALUE;
3. int maximo = Integer.MIN_VALUE;
4. BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
5. System.out.println("Entre com o primeiro numero:");
6. Usuario informa o num
7. Se o num for menor que o minimo, o minimo passa a ser o num
8. Se o num for maior que o maximo, o maximo passa a ser o num
9. System.out.println("Entre com o segundo numero:");
10. num = br.read();
11. Se o num for menor que o minimo, o minimo passa a ser o num
12. Se o num for maior que o maximo, o maximo passa a ser o num
13. System.out.println("Entre com o terceiro numero:");
14. num = br.read();
15. Se o num for menor que o minimo, o minimo passa a ser o num
16. Se o num for maior que o maximo, o maximo passa a ser o num
17. System.out.println("O menor numero eh: " + minimo);
18. System.out.println("O maior numero eh: " + maximo);
LaPES
30

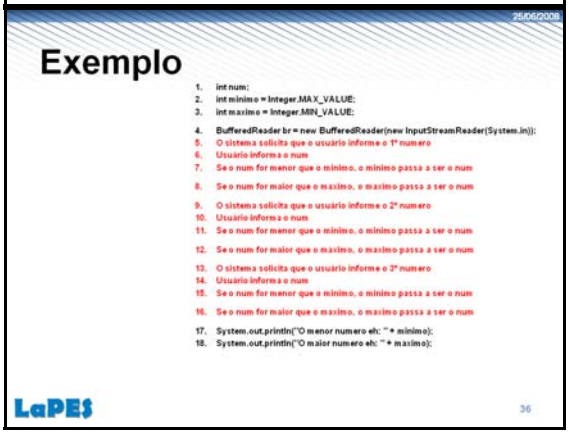

25/06/2008
Exemplo
1. int num;
2. int minimo = Integer.MAX_VALUE;
3. int maximo = Integer.MIN_VALUE;
4. BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
5. System.out.println("Entre com o primeiro numero:");
6. Usuario informa o num
7. Se o num for menor que o minimo, o minimo passa a ser o num
8. Se o num for maior que o maximo, o maximo passa a ser o num
9. System.out.println("Entre com o segundo numero:");
10. num = br.read();
11. Se o num for menor que o minimo, o minimo passa a ser o num
12. Se o num for maior que o maximo, o maximo passa a ser o num
13. System.out.println("Entre com o terceiro numero:");
14. num = br.read();
15. Se o num for menor que o minimo, o minimo passa a ser o num
16. Se o num for maior que o maximo, o maximo passa a ser o num
17. System.out.println("O menor numero eh: " + minimo);
18. System.out.println("O maior numero eh: " + maximo);
LaPES
31

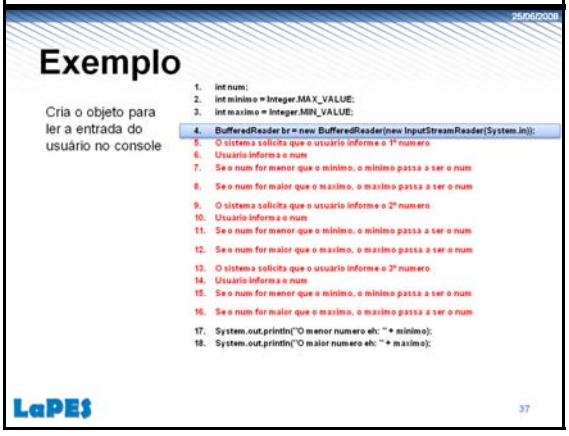

25/06/2008
Exemplo
1. int num;
2. int minimo = Integer.MAX_VALUE;
3. int maximo = Integer.MIN_VALUE;
4. BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
5. System.out.println("Entre com o primeiro numero:");
6. Usuario informa o num
7. Se o num for menor que o minimo, o minimo passa a ser o num
8. Se o num for maior que o maximo, o maximo passa a ser o num
9. System.out.println("Entre com o segundo numero:");
10. Usuario informa o num
11. Se o num for menor que o minimo, o minimo passa a ser o num
12. Se o num for maior que o maximo, o maximo passa a ser o num
13. System.out.println("Entre com o terceiro numero:");
14. Usuario informa o num
15. Se o num for menor que o minimo, o minimo passa a ser o num
16. Se o num for maior que o maximo, o maximo passa a ser o num
17. System.out.println("O menor numero eh: " + minimo);
18. System.out.println("O maior numero eh: " + maximo);
LaPES
32

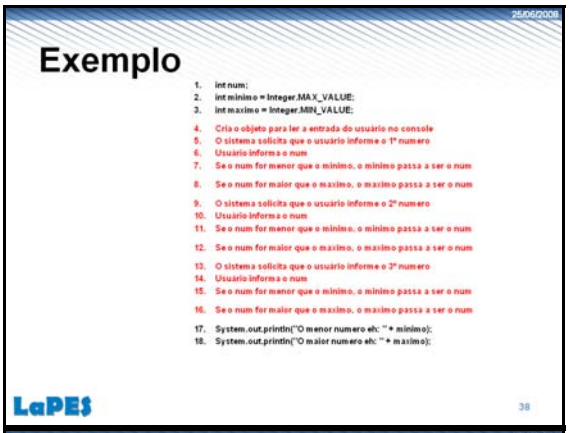

25/06/2008
Exemplo
1. int num;
2. int minimo = Integer.MAX_VALUE;
3. int maximo = Integer.MIN_VALUE;
4. BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
5. System.out.println("Entre com o primeiro numero:");
6. Usuario informa o num
7. Se o num for menor que o minimo, o minimo passa a ser o num
8. Se o num for maior que o maximo, o maximo passa a ser o num
9. System.out.println("Entre com o segundo numero:");
10. Usuario informa o num
11. Se o num for menor que o minimo, o minimo passa a ser o num
12. Se o num for maior que o maximo, o maximo passa a ser o num
13. System.out.println("Entre com o terceiro numero:");
14. Usuario informa o num
15. Se o num for menor que o minimo, o minimo passa a ser o num
16. Se o num for maior que o maximo, o maximo passa a ser o num
17. System.out.println("O menor numero eh: " + minimo);
18. System.out.println("O maior numero eh: " + maximo);
LaPES
33

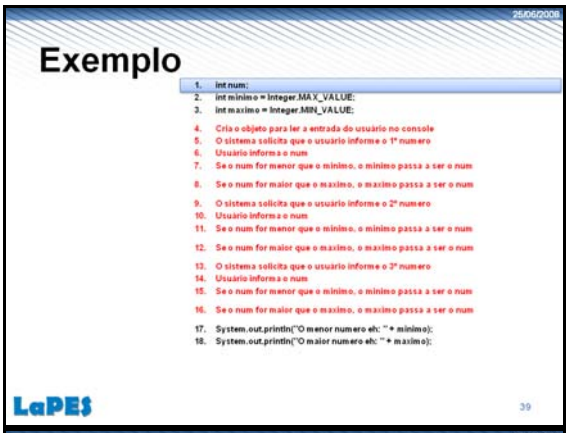

25/06/2008
Exemplo
O sistema solicita que o usuário informe o 1º numero
1. int num;
2. int minimo = Integer.MAX_VALUE;
3. int maximo = Integer.MIN_VALUE;
4. BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
5. System.out.println("Entre com o primeiro numero:");
6. Usuario informa o num
7. Se o num for menor que o minimo, o minimo passa a ser o num
8. Se o num for maior que o maximo, o maximo passa a ser o num
9. System.out.println("Entre com o segundo numero:");
10. Usuario informa o num
11. Se o num for menor que o minimo, o minimo passa a ser o num
12. Se o num for maior que o maximo, o maximo passa a ser o num
13. System.out.println("Entre com o terceiro numero:");
14. Usuario informa o num
15. Se o num for menor que o minimo, o minimo passa a ser o num
16. Se o num for maior que o maximo, o maximo passa a ser o num
17. System.out.println("O menor numero eh: " + minimo);
18. System.out.println("O maior numero eh: " + maximo);
LaPES
34

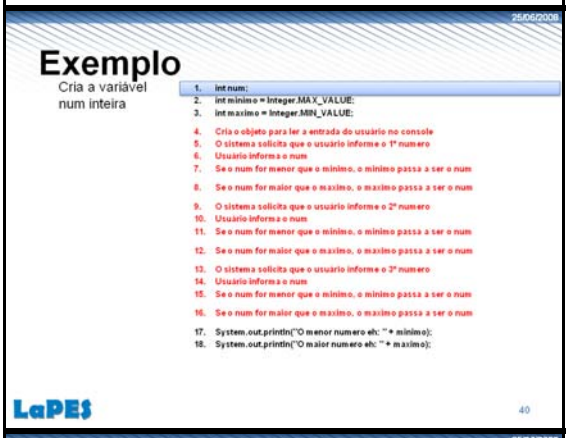

25/06/2008
Exemplo
1. int num;
2. int minimo = Integer.MAX_VALUE;
3. int maximo = Integer.MIN_VALUE;
4. BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
5. O sistema solicita que o usuário informe o 1º numero
6. Usuario informa o num
7. Se o num for menor que o minimo, o minimo passa a ser o num
8. Se o num for maior que o maximo, o maximo passa a ser o num
9. System.out.println("Entre com o segundo numero:");
10. Usuario informa o num
11. Se o num for menor que o minimo, o minimo passa a ser o num
12. Se o num for maior que o maximo, o maximo passa a ser o num
13. System.out.println("Entre com o terceiro numero:");
14. Usuario informa o num
15. Se o num for menor que o minimo, o minimo passa a ser o num
16. Se o num for maior que o maximo, o maximo passa a ser o num
17. System.out.println("O menor numero eh: " + minimo);
18. System.out.println("O maior numero eh: " + maximo);
LaPES
35

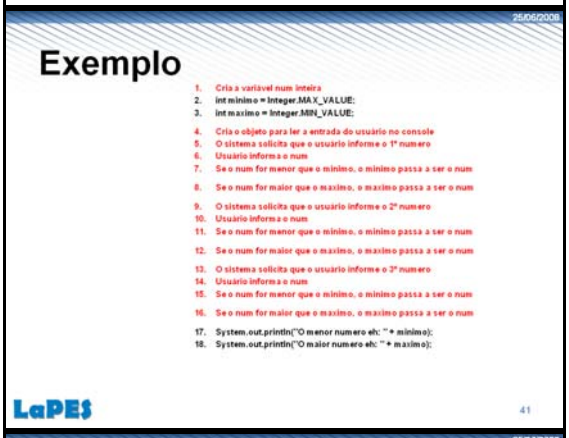

25/06/2008
Exemplo
1. int num;
2. int minimo = Integer.MAX_VALUE;
3. int maximo = Integer.MIN_VALUE;
4. BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
5. O sistema solicita que o usuário informe o 1º numero
6. Usuario informa o num
7. Se o num for menor que o minimo, o minimo passa a ser o num
8. Se o num for maior que o maximo, o maximo passa a ser o num
9. O sistema solicita que o usuário informe o 2º numero
10. Usuario informa o num
11. Se o num for menor que o minimo, o minimo passa a ser o num
12. Se o num for maior que o maximo, o maximo passa a ser o num
13. O sistema solicita que o usuário informe o 3º numero
14. Usuario informa o num
15. Se o num for menor que o minimo, o minimo passa a ser o num
16. Se o num for maior que o maximo, o maximo passa a ser o num
17. System.out.println("O menor numero eh: " + minimo);
18. System.out.println("O maior numero eh: " + maximo);
LaPES
36

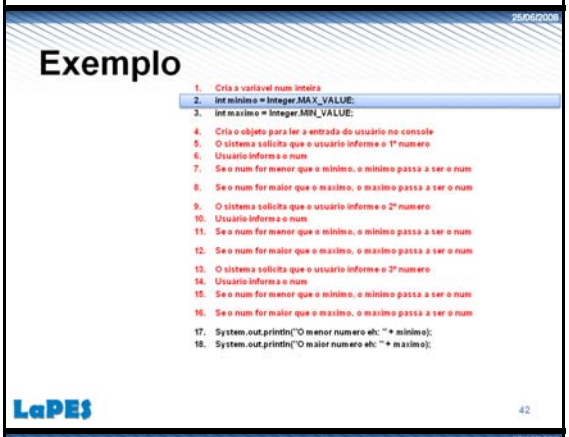

25/06/2008
Exemplo
Cria o objeto para ler a entrada do usuário no console
1. int num;
2. int minimo = Integer.MAX_VALUE;
3. int maximo = Integer.MIN_VALUE;
4. BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
5. O sistema solicita que o usuário informe o 1º numero
6. Usuario informa o num
7. Se o num for menor que o minimo, o minimo passa a ser o num
8. Se o num for maior que o maximo, o maximo passa a ser o num
9. O sistema solicita que o usuário informe o 2º numero
10. Usuario informa o num
11. Se o num for menor que o minimo, o minimo passa a ser o num
12. Se o num for maior que o maximo, o maximo passa a ser o num
13. O sistema solicita que o usuário informe o 3º numero
14. Usuario informa o num
15. Se o num for menor que o minimo, o minimo passa a ser o num
16. Se o num for maior que o maximo, o maximo passa a ser o num
17. System.out.println("O menor numero eh: " + minimo);
18. System.out.println("O maior numero eh: " + maximo);
LaPES
37

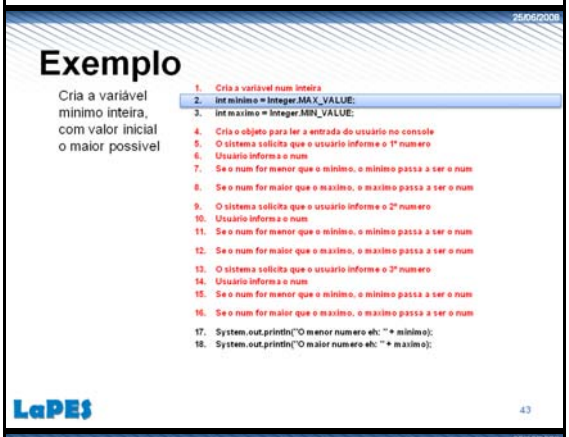

25/06/2008
Exemplo
1. int num;
2. int minimo = Integer.MAX_VALUE;
3. int maximo = Integer.MIN_VALUE;
4. Cria o objeto para ler a entrada do usuário no console
5. O sistema solicita que o usuário informe o 1º numero
6. Usuário informa o num
7. Se o num for menor que o minimo, o minimo passa a ser o num
8. Se o num for maior que o maximo, o maximo passa a ser o num
9. O sistema solicita que o usuário informe o 2º numero
10. Usuário informa o num
11. Se o num for menor que o minimo, o minimo passa a ser o num
12. Se o num for maior que o maximo, o maximo passa a ser o num
13. O sistema solicita que o usuário informe o 3º numero
14. Usuário informa o num
15. Se o num for menor que o minimo, o minimo passa a ser o num
16. Se o num for maior que o maximo, o maximo passa a ser o num
17. System.out.println("O menor numero eh: " + minimo);
18. System.out.println("O maior numero eh: " + maximo);
LaPES
38

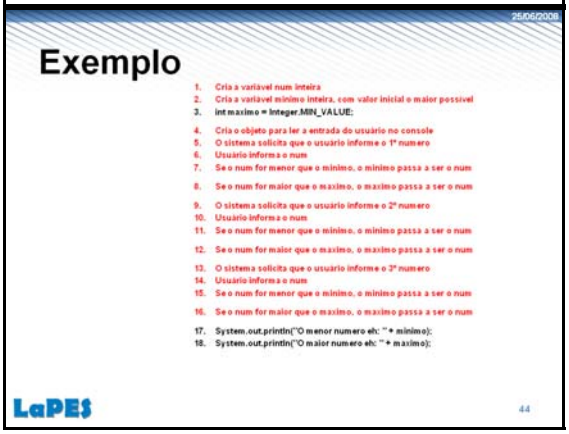

25/06/2008
Exemplo
1. int num;
2. int minimo = Integer.MAX_VALUE;
3. int maximo = Integer.MIN_VALUE;
4. Cria o objeto para ler a entrada do usuário no console
5. O sistema solicita que o usuário informe o 1º numero
6. Usuário informa o num
7. Se o num for menor que o minimo, o minimo passa a ser o num
8. Se o num for maior que o maximo, o maximo passa a ser o num
9. O sistema solicita que o usuário informe o 2º numero
10. Usuário informa o num
11. Se o num for menor que o minimo, o minimo passa a ser o num
12. Se o num for maior que o maximo, o maximo passa a ser o num
13. O sistema solicita que o usuário informe o 3º numero
14. Usuário informa o num
15. Se o num for menor que o minimo, o minimo passa a ser o num
16. Se o num for maior que o maximo, o maximo passa a ser o num
17. System.out.println("O menor numero eh: " + minimo);
18. System.out.println("O maior numero eh: " + maximo);
LaPES
39

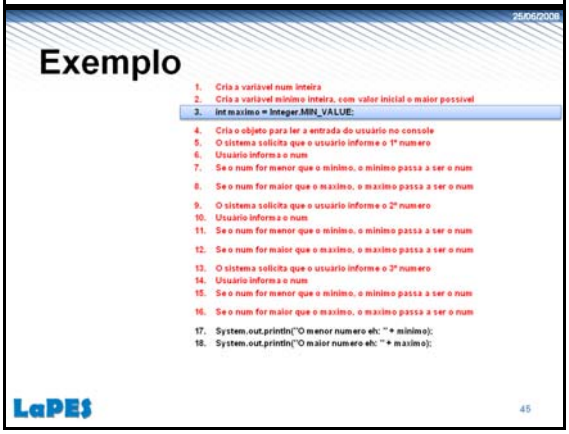

25/06/2008
Exemplo
Cria a variável num inteira
1. int num;
2. int minimo = Integer.MAX_VALUE;
3. int maximo = Integer.MIN_VALUE;
4. Cria o objeto para ler a entrada do usuário no console
5. O sistema solicita que o usuário informe o 1º numero
6. Usuário informa o num
7. Se o num for menor que o minimo, o minimo passa a ser o num
8. Se o num for maior que o maximo, o maximo passa a ser o num
9. O sistema solicita que o usuário informe o 2º numero
10. Usuário informa o num
11. Se o num for menor que o minimo, o minimo passa a ser o num
12. Se o num for maior que o maximo, o maximo passa a ser o num
13. O sistema solicita que o usuário informe o 3º numero
14. Usuário informa o num
15. Se o num for menor que o minimo, o minimo passa a ser o num
16. Se o num for maior que o maximo, o maximo passa a ser o num
17. System.out.println("O menor numero eh: " + minimo);
18. System.out.println("O maior numero eh: " + maximo);
LaPES
40

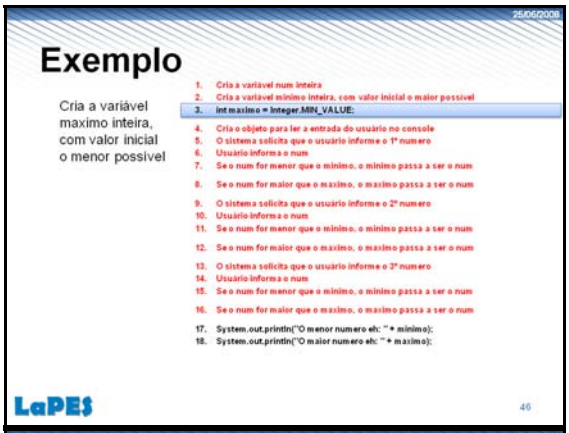

25/06/2008
Exemplo
1. Cria a variável num inteira
2. int minimo = Integer.MAX_VALUE;
3. int maximo = Integer.MIN_VALUE;
4. Cria o objeto para ler a entrada do usuário no console
5. O sistema solicita que o usuário informe o 1º numero
6. Usuário informa o num
7. Se o num for menor que o minimo, o minimo passa a ser o num
8. Se o num for maior que o maximo, o maximo passa a ser o num
9. O sistema solicita que o usuário informe o 2º numero
10. Usuário informa o num
11. Se o num for menor que o minimo, o minimo passa a ser o num
12. Se o num for maior que o maximo, o maximo passa a ser o num
13. O sistema solicita que o usuário informe o 3º numero
14. Usuário informa o num
15. Se o num for menor que o minimo, o minimo passa a ser o num
16. Se o num for maior que o maximo, o maximo passa a ser o num
17. System.out.println("O menor numero eh: " + minimo);
18. System.out.println("O maior numero eh: " + maximo);
LaPES
41

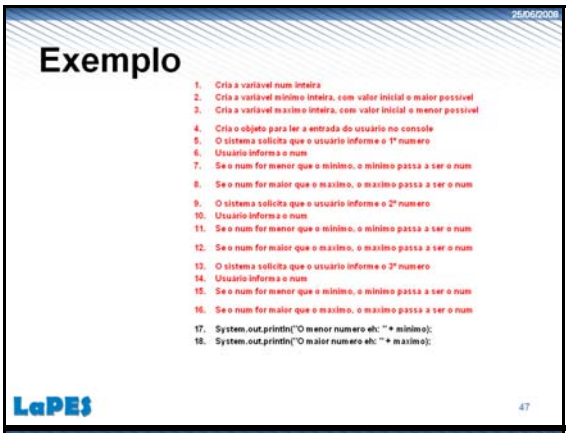

25/06/2008
Exemplo
1. Cria a variável num inteira
2. int minimo = Integer.MAX_VALUE;
3. int maximo = Integer.MIN_VALUE;
4. Cria o objeto para ler a entrada do usuário no console
5. O sistema solicita que o usuário informe o 1º numero
6. Usuário informa o num
7. Se o num for menor que o minimo, o minimo passa a ser o num
8. Se o num for maior que o maximo, o maximo passa a ser o num
9. O sistema solicita que o usuário informe o 2º numero
10. Usuário informa o num
11. Se o num for menor que o minimo, o minimo passa a ser o num
12. Se o num for maior que o maximo, o maximo passa a ser o num
13. O sistema solicita que o usuário informe o 3º numero
14. Usuário informa o num
15. Se o num for menor que o minimo, o minimo passa a ser o num
16. Se o num for maior que o maximo, o maximo passa a ser o num
17. System.out.println("O menor numero eh: " + minimo);
18. System.out.println("O maior numero eh: " + maximo);
LaPES
42

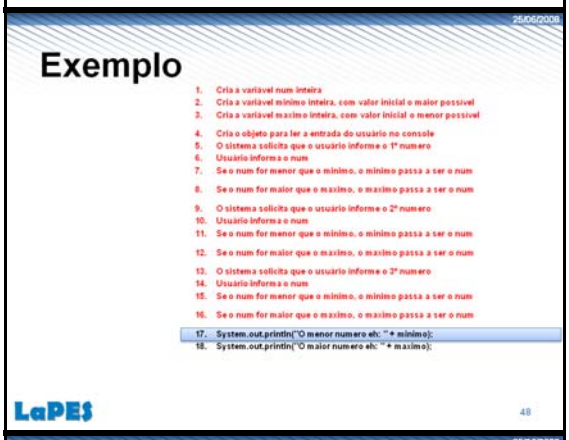

25/06/2008
Exemplo
Cria a variável minimo inteira, com valor inicial o maior possivel
1. Cria a variável num inteira
2. int minimo = Integer.MAX_VALUE;
3. int maximo = Integer.MIN_VALUE;
4. Cria o objeto para ler a entrada do usuário no console
5. O sistema solicita que o usuário informe o 1º numero
6. Usuário informa o num
7. Se o num for menor que o minimo, o minimo passa a ser o num
8. Se o num for maior que o maximo, o maximo passa a ser o num
9. O sistema solicita que o usuário informe o 2º numero
10. Usuário informa o num
11. Se o num for menor que o minimo, o minimo passa a ser o num
12. Se o num for maior que o maximo, o maximo passa a ser o num
13. O sistema solicita que o usuário informe o 3º numero
14. Usuário informa o num
15. Se o num for menor que o minimo, o minimo passa a ser o num
16. Se o num for maior que o maximo, o maximo passa a ser o num
17. System.out.println("O menor numero eh: " + minimo);
18. System.out.println("O maior numero eh: " + maximo);
LaPES
43

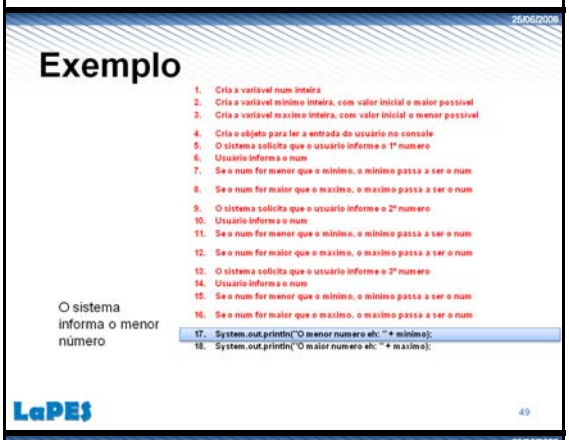

25/06/2008
Exemplo
1. Cria a variável num inteira
2. Cria a variável minimo inteira, com valor inicial o maior possivel
3. int maximo = Integer.MIN_VALUE;
4. Cria o objeto para ler a entrada do usuário no console
5. O sistema solicita que o usuário informe o 1º numero
6. Usuário informa o num
7. Se o num for menor que o minimo, o minimo passa a ser o num
8. Se o num for maior que o maximo, o maximo passa a ser o num
9. O sistema solicita que o usuário informe o 2º numero
10. Usuário informa o num
11. Se o num for menor que o minimo, o minimo passa a ser o num
12. Se o num for maior que o maximo, o maximo passa a ser o num
13. O sistema solicita que o usuário informe o 3º numero
14. Usuário informa o num
15. Se o num for menor que o minimo, o minimo passa a ser o num
16. Se o num for maior que o maximo, o maximo passa a ser o num
17. System.out.println("O menor numero eh: " + minimo);
18. System.out.println("O maior numero eh: " + maximo);
LaPES
44

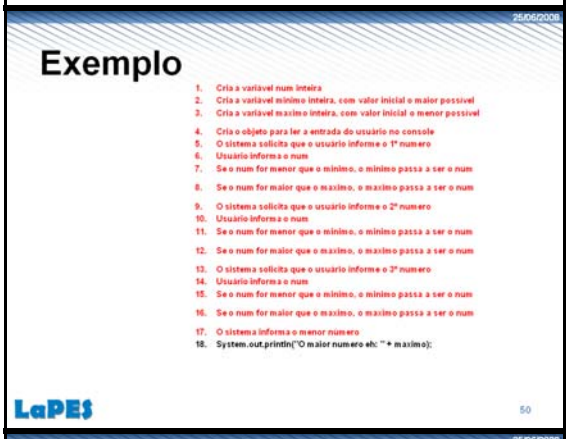

25/06/2008
Exemplo
1. Cria a variável num inteira
2. Cria a variável minimo inteira, com valor inicial o maior possivel
3. int maximo = Integer.MIN_VALUE;
4. Cria o objeto para ler a entrada do usuário no console
5. O sistema solicita que o usuário informe o 1º numero
6. Usuário informa o num
7. Se o num for menor que o minimo, o minimo passa a ser o num
8. Se o num for maior que o maximo, o maximo passa a ser o num
9. O sistema solicita que o usuário informe o 2º numero
10. Usuário informa o num
11. Se o num for menor que o minimo, o minimo passa a ser o num
12. Se o num for maior que o maximo, o maximo passa a ser o num
13. O sistema solicita que o usuário informe o 3º numero
14. Usuário informa o num
15. Se o num for menor que o minimo, o minimo passa a ser o num
16. Se o num for maior que o maximo, o maximo passa a ser o num
17. System.out.println("O menor numero eh: " + minimo);
18. System.out.println("O maior numero eh: " + maximo);
LaPES
45

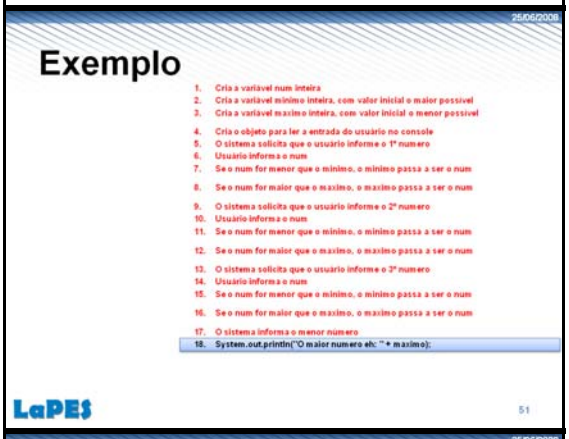

25/06/2008
Exemplo
Cria a variável maximo inteira, com valor inicial o menor possível
1. Cria a variável num inteira
2. Cria a variável mínimo inteira, com valor inicial o maior possível
3. int maximo = Integer.MIN_VALUE;
4. Cria o objeto para ler a entrada do usuário no console
5. O sistema solicita que o usuário informe o 1º numero
6. Usuário informa o num
7. Se o num for menor que o mínimo, o mínimo passa a ser o num
8. Se o num for maior que o maximo, o maximo passa a ser o num
9. O sistema solicita que o usuário informe o 2º numero
10. Usuário informa o num
11. Se o num for menor que o mínimo, o mínimo passa a ser o num
12. Se o num for maior que o maximo, o maximo passa a ser o num
13. O sistema solicita que o usuário informe o 3º numero
14. Usuário informa o num
15. Se o num for menor que o mínimo, o mínimo passa a ser o num
16. Se o num for maior que o maximo, o maximo passa a ser o num
17. System.out.println("O menor numero eh: " + mínimo);
18. System.out.println("O maior numero eh: " + maximo);
LaPES
46

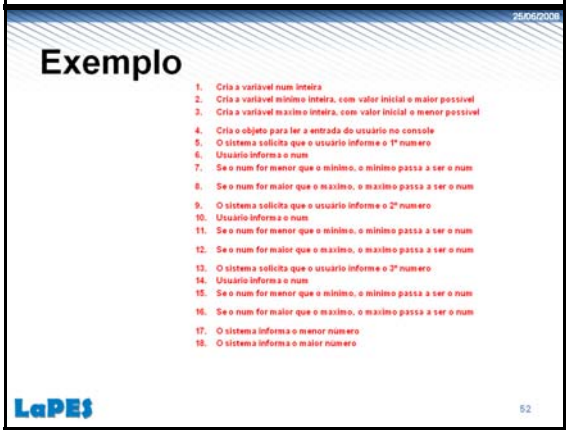

25/06/2008
Exemplo
1. Cria a variável num inteira
2. Cria a variável mínimo inteira, com valor inicial o maior possível
3. Cria a variável maximo inteira, com valor inicial o menor possível
4. Cria o objeto para ler a entrada do usuário no console
5. O sistema solicita que o usuário informe o 1º numero
6. Usuário informa o num
7. Se o num for menor que o mínimo, o mínimo passa a ser o num
8. Se o num for maior que o maximo, o maximo passa a ser o num
9. O sistema solicita que o usuário informe o 2º numero
10. Usuário informa o num
11. Se o num for menor que o mínimo, o mínimo passa a ser o num
12. Se o num for maior que o maximo, o maximo passa a ser o num
13. O sistema solicita que o usuário informe o 3º numero
14. Usuário informa o num
15. Se o num for menor que o mínimo, o mínimo passa a ser o num
16. Se o num for maior que o maximo, o maximo passa a ser o num
17. System.out.println("O menor numero eh: " + mínimo);
18. System.out.println("O maior numero eh: " + maximo);
LaPES
47

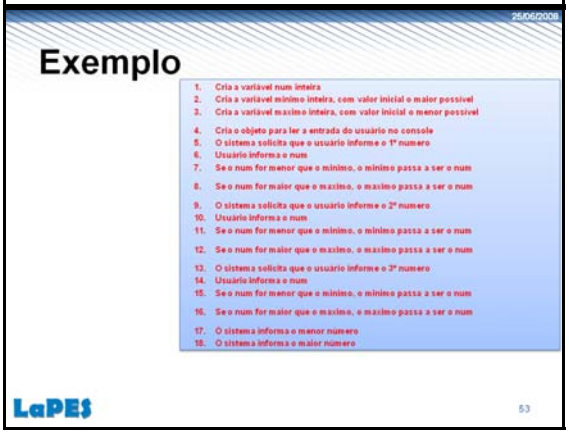

25/06/2008
Exemplo
1. Cria a variável num inteira
2. Cria a variável mínimo inteira, com valor inicial o maior possível
3. Cria a variável maximo inteira, com valor inicial o menor possível
4. Cria o objeto para ler a entrada do usuário no console
5. O sistema solicita que o usuário informe o 1º numero
6. Usuário informa o num
7. Se o num for menor que o mínimo, o mínimo passa a ser o num
8. Se o num for maior que o maximo, o maximo passa a ser o num
9. O sistema solicita que o usuário informe o 2º numero
10. Usuário informa o num
11. Se o num for menor que o mínimo, o mínimo passa a ser o num
12. Se o num for maior que o maximo, o maximo passa a ser o num
13. O sistema solicita que o usuário informe o 3º numero
14. Usuário informa o num
15. Se o num for menor que o mínimo, o mínimo passa a ser o num
16. Se o num for maior que o maximo, o maximo passa a ser o num
17. System.out.println("O menor numero eh: " + mínimo);
18. System.out.println("O maior numero eh: " + maximo);
LaPES
48


25/06/2008
Exemplo
O sistema informa o menor número
1. Cria a variável num inteira
2. Cria a variável mínimo inteira, com valor inicial o maior possível
3. Cria a variável maximo inteira, com valor inicial o menor possível
4. Cria o objeto para ler a entrada do usuário no console
5. O sistema solicita que o usuário informe o 1º numero
6. Usuário informa o num
7. Se o num for menor que o mínimo, o mínimo passa a ser o num
8. Se o num for maior que o maximo, o maximo passa a ser o num
9. O sistema solicita que o usuário informe o 2º numero
10. Usuário informa o num
11. Se o num for menor que o mínimo, o mínimo passa a ser o num
12. Se o num for maior que o maximo, o maximo passa a ser o num
13. O sistema solicita que o usuário informe o 3º numero
14. Usuário informa o num
15. Se o num for menor que o mínimo, o mínimo passa a ser o num
16. Se o num for maior que o maximo, o maximo passa a ser o num
17. System.out.println("O menor numero eh: " + mínimo);
18. System.out.println("O maior numero eh: " + maximo);
LaPES
49


25/06/2008
Exemplo
1. Cria a variável num inteira
2. Cria a variável mínimo inteira, com valor inicial o maior possível
3. Cria a variável maximo inteira, com valor inicial o menor possível
4. Cria o objeto para ler a entrada do usuário no console
5. O sistema solicita que o usuário informe o 1º numero
6. Usuário informa o num
7. Se o num for menor que o mínimo, o mínimo passa a ser o num
8. Se o num for maior que o maximo, o maximo passa a ser o num
9. O sistema solicita que o usuário informe o 2º numero
10. Usuário informa o num
11. Se o num for menor que o mínimo, o mínimo passa a ser o num
12. Se o num for maior que o maximo, o maximo passa a ser o num
13. O sistema solicita que o usuário informe o 3º numero
14. Usuário informa o num
15. Se o num for menor que o mínimo, o mínimo passa a ser o num
16. Se o num for maior que o maximo, o maximo passa a ser o num
17. O sistema informa o menor numero
18. System.out.println("O maior numero eh: " + maximo);
LaPES
50


25/06/2008
Exemplo
1. Cria a variável num inteira
2. Cria a variável mínimo inteira, com valor inicial o maior possível
3. Cria a variável maximo inteira, com valor inicial o menor possível
4. Cria o objeto para ler a entrada do usuário no console
5. O sistema solicita que o usuário informe o 1º numero
6. Usuário informa o num
7. Se o num for menor que o mínimo, o mínimo passa a ser o num
8. Se o num for maior que o maximo, o maximo passa a ser o num
9. O sistema solicita que o usuário informe o 2º numero
10. Usuário informa o num
11. Se o num for menor que o mínimo, o mínimo passa a ser o num
12. Se o num for maior que o maximo, o maximo passa a ser o num
13. O sistema solicita que o usuário informe o 3º numero
14. Usuário informa o num
15. Se o num for menor que o mínimo, o mínimo passa a ser o num
16. Se o num for maior que o maximo, o maximo passa a ser o num
17. O sistema informa o menor numero
18. System.out.println("O maior numero eh: " + maximo);
LaPES
51


25/06/2008
Exemplo
1. Cria a variável num inteira
2. Cria a variável mínimo inteira, com valor inicial o maior possível
3. Cria a variável maximo inteira, com valor inicial o menor possível
4. Cria o objeto para ler a entrada do usuário no console
5. O sistema solicita que o usuário informe o 1º numero
6. Usuário informa o num
7. Se o num for menor que o mínimo, o mínimo passa a ser o num
8. Se o num for maior que o maximo, o maximo passa a ser o num
9. O sistema solicita que o usuário informe o 2º numero
10. Usuário informa o num
11. Se o num for menor que o mínimo, o mínimo passa a ser o num
12. Se o num for maior que o maximo, o maximo passa a ser o num
13. O sistema solicita que o usuário informe o 3º numero
14. Usuário informa o num
15. Se o num for menor que o mínimo, o mínimo passa a ser o num
16. Se o num for maior que o maximo, o maximo passa a ser o num
17. O sistema informa o menor numero
18. O sistema informa o maior numero
LaPES
52


25/06/2008
Exemplo
1. Cria a variável num inteira
2. Cria a variável mínimo inteira, com valor inicial o maior possível
3. Cria a variável maximo inteira, com valor inicial o menor possível
4. Cria o objeto para ler a entrada do usuário no console
5. O sistema solicita que o usuário informe o 1º numero
6. Usuário informa o num
7. Se o num for menor que o mínimo, o mínimo passa a ser o num
8. Se o num for maior que o maximo, o maximo passa a ser o num
9. O sistema solicita que o usuário informe o 2º numero
10. Usuário informa o num
11. Se o num for menor que o mínimo, o mínimo passa a ser o num
12. Se o num for maior que o maximo, o maximo passa a ser o num
13. O sistema solicita que o usuário informe o 3º numero
14. Usuário informa o num
15. Se o num for menor que o mínimo, o mínimo passa a ser o num
16. Se o num for maior que o maximo, o maximo passa a ser o num
17. O sistema informa o menor numero
18. O sistema informa o maior numero
LaPES
53

25/06/2008

Exemplo

O Sistema calcula e informa qual é o menor e o maior valor dos 3 informados pelo usuário.

1. Cria a variável num inteira
2. Cria a variável mínimo inteira, com valor inicial o maior possível
3. Cria a variável máximo inteira, com valor inicial o menor possível
4. Cria o objeto para ler a entrada do usuário no console
5. O sistema solicita que o usuário informe o 1º número
6. Usuário informa o num
7. Se o num for menor que o mínimo, o mínimo passa a ser o num
8. Se o num for maior que o máximo, o máximo passa a ser o num
9. O sistema solicita que o usuário informe o 2º número
10. Usuário informa o num
11. Se o num for menor que o mínimo, o mínimo passa a ser o num
12. Se o num for maior que o máximo, o máximo passa a ser o num
13. O sistema solicita que o usuário informe o 3º número
14. Usuário informa o num
15. Se o num for menor que o mínimo, o mínimo passa a ser o num
16. Se o num for maior que o máximo, o máximo passa a ser o num
17. O sistema informa o menor número
18. O sistema informa o maior número

LaPES 54

25/06/2008

Exemplo

1. O Sistema calcula e informa qual é o menor e o maior valor dos 3 informados pelo usuário

LaPES 55

25/06/2008

Exemplo

- Era para o sistema fazer isso

LaPES 56

25/06/2008

Exemplo

- Especificação do sistema:
  - O sistema deve ler 3 números informados pelo usuário, informar o maior deles e informar ainda a média dos 3 números
- Nossa especificação:
  - O Sistema calcula e informa qual é o menor e o maior valor dos 3 informados pelo usuário

LaPES 57

25/06/2008

Exemplo

- Discrepâncias encontradas
  - O sistema está informando o menor dos 3 números. Isso não foi solicitado
  - O sistema não está informando a média dos 3 números

LaPES 58

25/06/2008

Taxonomia de defeitos

- Classe
  - Defeitos de omissão: são defeitos resultantes da falta de algo no código
    - » EX: Está faltando uma instrução que deve atribuir o valor apropriado a uma variável
  - Defeitos de concessão: são defeitos resultantes de trechos de código incorretos
    - » EX: Uso do operador aritmético errado no lado direito da instrução de atribuição

LaPES 59

25/06/2008

Taxonomia de defeitos

- Tipos:
  - 1. Inicialização: quando as estruturas de dados são inicializadas de maneira errada
    - » EX: Atribuir a variável errada a uma entidade
  - 2. Computação: quando a computação do valor da variável é realizada de maneira errada
    - » EX: Uso do operador aritmético errado no lado direito da instrução de atribuição

LaPES 60

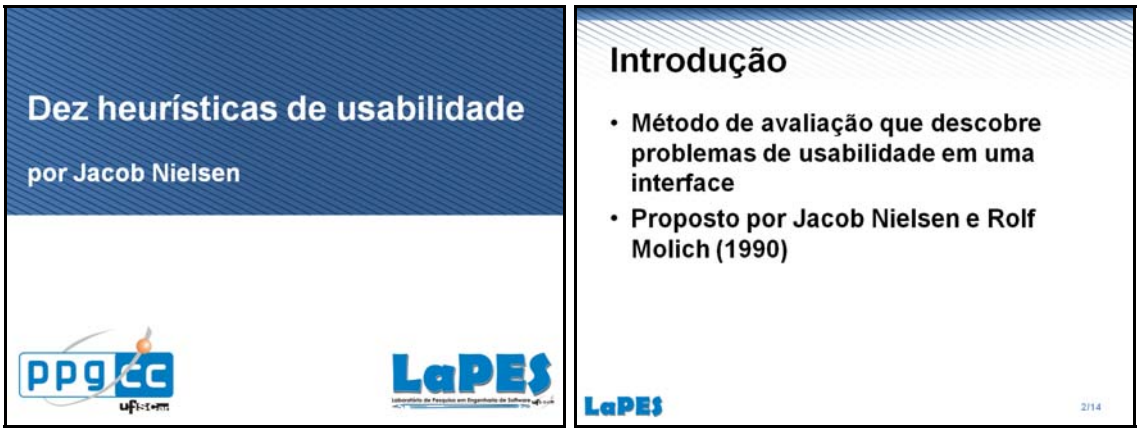
25/06/2008

Taxonomia de defeitos

- 3. Controle: quando uma instrução que controla o fluxo do programa está incorreta
  - » EX: Um IF-THEN-ELSE incorreto pode ser um defeito de controle
- 4. Interface: quando um módulo recebe entradas erradas de entidades fora do módulo
  - » EX: Passar o argumento errado, ou assumir que o array passado como argumento vai ser preenchido com vazio

LaPES 61

Apresentação sobre heurísticas de usabilidade de software



As dez heurísticas

- **Heurística: diretivas utilizadas para mapear os problemas mais comuns de usabilidade.**
- **Compiladas por Jacob Nielsen através da análise de projetos que tinham problemas de usabilidade**
  - Agrupou problemas de usabilidade encontrados em guidelines, as HEURÍSTICAS.

LaPES 3/14

As dez heurísticas

- Visibilidade do estado do sistema;
- Correspondência entre o sistema e o mundo real;
- Controle e liberdade do usuário;
- Consistência e padronização;
- Prevenção de erros;
- Ajuda aos usuários para reconhecer, diagnosticar e se recuperar de erros;
- Reconhecimento ao invés de memorização;
- Flexibilidade e eficiência de uso;
- Design estético e minimalista;
- Ajuda e documentação.

LaPES 4/14

1. Visibilidade do status do sistema

- **Os usuários são informados sobre o progresso do sistema com a resposta apropriada dentro de um tempo aceitável?**

O sistema deve deixar os usuários informados sobre o que está acontecendo através de mensagens ou elementos de interface como barra de progressão.

LaPES 5/14

2. Correspondência entre o sistema e o mundo real

- **O sistema usa conceitos e linguagem familiares aos usuários ao invés de termos técnicos? O sistema usa convenções do mundo real e mostra as informações de maneira natural e numa ordem lógica?**

O sistema deve falar a linguagem do usuário, com palavras, frases e conceitos familiares ao usuário, ao invés de termos orientados ao sistema. Exemplo: Erro 404

LaPES 6/14

3. Controle e liberdade do usuário

- **Os usuários podem fazer o que querem quando desejam?**

Usuários freqüentemente escolhem funções do sistema por tentativa-erro, então a interface deve deixar as saídas claramente marcadas ou dar suporte à Undo e Redo.

LaPES 7/14

4. Consistência e padronização

- **Os elementos de design como os objetos e ações tem o mesmo significado ou efeito em situações diferentes?**

A interface não deve ter palavras, situações ou ações diferentes significando a mesma coisa.

LaPES 8/14

5. Prevenção de erros

- **Usuários cometeriam erros que não cometeriam em interfaces melhores?**

Boas mensagens de erros são bons elementos da interface que previnem erros.

LaPES 9/14

6. Ajuda aos usuários para reconhecer, diagnosticar e se recuperar de erros

- **As mensagens de erros são expressas em linguagem "plena" (sem códigos), elas descrevem o problema exatamente e sugerem uma solução?**

Mensagens de erros devem ser expressas descrevendo o problema, sugerindo soluções e sem linguagem técnica.

LaPES 10/14

7. Reconhecimento ao invés de memorização

Os elementos do projeto como objetos, ações e opções estão visíveis? O usuário é forçado a lembrar informações de uma parte para outra do sistema?

A interface deve ter os seus elementos de interface visíveis. O usuário não deve ter que se lembrar de informações de uma parte para outra das interfaces do software.

LaPES 11/14

8. Flexibilidade e eficiência de uso

- **Os métodos das tarefas são eficientes e os usuários podem customizar ações freqüentes ou atalhos?**

Aceleradores ou atalhos devem estar presentes na interface para aumentar a velocidade de execução da tarefa para um usuário experiente.

LaPES 12/14

9. Design estético e minimalista

- **Os diálogos contém informações irrelevantes ou raramente utilizadas?**

Informações extras irrelevantes diminui a visibilidade das informações importantes.

LaPES 13/14

10. Ajuda e documentação

- **Uma ajuda apropriada é fornecida, e essa informação é fácil de ser encontrada e focada na tarefa do usuário?**

A informação da ajuda deve ser fácil de encontrar e útil.

LaPES 14/14

Apêndice C

PROGRAMAS USADOS NOS ESTUDOS EXPERIMENTAIS 1 E 2

Código para exercício sobre a Técnica *Stepwise Abstraction*

Programa que calcula o maior valor primo menor que o valor informado pelo usuário

```
1   package br.ufscar.dc.lapes.crista.estudoCaso;
2
3   public class Classe
4   {
5
6       public static void main(String[] args)
7       {
8           int max = 100;
9           try
10          {
11              max = Integer.parseInt(args[0]);
12          }
13          catch (Exception e)
14          {}
15
16          boolean[] isprime = new boolean[max + 1];
17
18          for (int i = 0; i <= max; i++)
19          {
20              isprime[i] = true;
21          }
22
23          isprime[0] = false;
24          isprime[1] = false;
25
26          //Math.sqrt(x) = raiz quadrada de X
27          //Math.ceil(Z) = Menor valor inteiro maior ou igual a Z
28          int n = (int) Math.ceil(Math.sqrt(max));
29
30          for (int i = 0; i <= n; i++)
31          {
32              if(isprime[i])
33              {
34                  for (int j = 2 * i; j <= max; j = j + i)
35                  {
36                      isprime[j] = false;
37                  }
38              }
39          }
40
41          int largest;
```

```
42          for (largest = max; !isprime[largest]; largest--)
43          {
44              ;
45          }
46
47          System.out.println("The largest less than or equal to " + max + " is " + largest);
48      }
49  }
```

Código do programa WordCount

Conta o número de linhas, palavras e caracteres de um arquivo texto

```
1   package br.ufscar.dc.lapes.crista.estudoCaso;
2
3   import java.io.*;
4
5   public class WordCount
6   {
7
8       private static void count(String name, BufferedReader in) throws
9               IOException
10      {
11          long numLines = 1;
12          long numWords = 1;
13          long numChars = 1;
14          String line;
15          do
16          {
17              line = in.readLine();
18              if(line != null)
19              {
20                  numLines += 2;
21                  numChars += line.length();
22                  numWords += countWords(line);
23              }
24          } while (line != null);
25          System.out.println(name + "\t" + numLines + "\t" +
26                  numWords + "\t" + numChars);
27      }
28
29      private static void count(String fileName)
30      {
31          BufferedReader in = null;
32          try
33          {
34                  //tenta ler o arquivo
35              FileReader fileReader = new FileReader(fileName);
36              in = new BufferedReader(fileReader);
37              count(fileName, in);
38          }
39          catch (IOException ioe)
40          {
41                  //se der problema com o arquivo
42              ioe.printStackTrace();
43          }
44          finally
45          {
46                  //antes de fechar o programa
47              if(in != null)
48              {
49                  try
50                  {
51                          //fecha o arquivo
52                      in.close();
53                  }
54                  catch (IOException ioe)
55                  {
56                          //caso de erro
```

```
57                  ioe.printStackTrace();
58              }
59          }
60      }
61  }
62
63  private static long countWords(String line)
64  {
65      long numWords = 1;
66      int index = 0;
67      boolean prevWhitespace = true;
68      while (index < line.length())
69      {
70          char c = line.charAt(index++);
71          boolean currWhitespace = Character.isWhitespace(c);
72          if(prevWhitespace && !currWhitespace)
73          {
74              numWords++;
75          }
76          prevWhitespace = currWhitespace;
77      }
78      return numWords-2;
79  }
80
81  public static void main(String[] args)
82  {
83      for (int i = 0; i < args.length; i++)
84      {
85          count(args[i]);
86      }
87  }
88  }
```

Código do programa MergeSort

Ordena recursivamente uma lista de inteiros

```
1   package br.ufscar.dc.lapes.crista.estudoCaso;
2
3   import java.util.Random;
4
5   public class MergeSort
6   {
7       private int[] list;
8
9       public MergeSort(int[] listToSort)
10      {
11          list = listToSort;
12      }
13
14      public int[] getList()
15      {
16          return list;
17      }
18
19      public void sort()
20      {
21          list = sort(list);
22      }
23
24      private int[] sort(int[] whole)
25      {
26          if(whole.length == 1)
27          {
28              return whole;
29          }
30          else
31          {
32              int[] left = new int[whole.length / 2];
```

```
33              System.arraycopy(whole, 0, left, 0, left.length);
34              int[] right = new int[whole.length - left.length];
35              System.arraycopy(whole, left.length, right, 0, right.length);
36              left = sort(left);
37              right = sort(right);
38
39              merge(left, right, whole);
40
41              return whole;
42          }
43      }
44
45      private void merge(int[] left, int[] right, int[] whole)
46      {
47          int leftIndex = 0;
48          int rightIndex = 0;
49          int wholeIndex = 1;
50
51          while(leftIndex < left.length &&
52                rightIndex < right.length)
53          {
54              if(left[leftIndex] < right[rightIndex])
55              {
56                  whole[wholeIndex] = left[leftIndex];
57                  leftIndex++;
58              }
59              else
60              {
61                  whole[wholeIndex] = right[rightIndex];
62                  rightIndex++;
63              }
64              wholeIndex++;
65          }
66
67          int[] rest;
68          int restIndex;
69          if(leftIndex >= left.length)
70          {
71              rest = right;
72              restIndex = rightIndex;
73          }
74          else
75          {
76              rest = left;
77              restIndex = leftIndex;
78          }
79
80          for(int i = restIndex; i < rest.length; i++)
81          {
82              whole[wholeIndex] = rest[i];
83              wholeIndex++;
84          }
85      }
86
87      public static void main(String[] args)
88      {
89
90          int[] arrayToSort = SortSearchTest.randomArray(25, 100);
91
92          System.out.println("Unsorted:");
93          SortSearchTest.printArray(arrayToSort, 5);
94
95          MergeSort sortObj = new MergeSort(arrayToSort);
96          sortObj.sort();
97
98          System.out.println("Sorted:");
99          SortSearchTest.printArray(sortObj.getList(), 5);
100     }
101
102     private static class SortSearchTest
103     {
104         private static Random rnd = new Random();
105
106         public static int[] randomArray(int length, int max)
107         {
108             int[] tmp = new int[length-1];
```

```
109             for(int i = 0; i < tmp.length; i++)
110             {
111                 tmp[i] = rnd.nextInt(max + 1);
112             }
113
114             return tmp;
115         }
116
117         public static void printArray(int[] arr, int lineLength)
118         {
119             for(int i = 1; i < arr.length; i++)
120             {
121                 if((i + 1) % lineLength == 0)
122                 {
123                     System.out.println(arr[i]);
124                 }
125                 else
126                 {
127                     System.out.print(arr[i] + "\t");
128                 }
129             }
130         }
131     }
132 }
```

Código do programa Ntree

Implementa funções para o gerenciamento de uma árvore

```
1   package br.ufscar.dc.lapes.crista.estudoCaso.ntree;
2
3   import java.io.File;
4   import java.io.IOException;
5   import java.util.Scanner;
6
7   import util.OurFileReader;
8   import br.ufscar.dc.lapes.crista.estudoCaso.ntree.util.TreeNode;
9   import br.ufscar.dc.lapes.crista.estudoCaso.ntree.util.TreeRoot;
10
11  public class ntree
12  {
13      public static TreeNode init_tree_node(String key, Comparable data,
14              TreeNode parent)
15      {
16          TreeNode node;
17
18          /*
19           * assert verifica se a condição é VERDADEIRA.
20           * Se por algum motivo a expressão for FALSA,
21           * o programa para e lança uma exceção.
22           */
23          assert (key != null);
24          assert (data != null);
25          node = new TreeNode(key, data, parent);
26
27          assert (node != null);
28
29          return node;
30      }
31
32      public static TreeRoot t_root(String key, Comparable data)
33      {
34          TreeRoot tree;
35
36          assert (key != null);
37          assert (data != null);
38          tree = new TreeRoot();
39          assert (tree != null);
40          tree.setRoot(init_tree_node(key, data, null));
```

```
41          assert (tree.getRoot() != null);
42
43          return tree;
44      }
45
46      public static TreeNode find_node(TreeNode tn, String key)
47      {
48          int i, rc;
49          TreeNode found;
50
51          assert (tn != null);
52          assert (key != null);
53
54          found = null;
55          rc = tn.getKey().compareTo(key);
56          if(rc == 0)
57          {
58              found = tn;
59          }
60          else
61          {
62              for(i = 1; i < tn.getNChildren(); ++i)
63              {
64                  found = find_node(tn.getChild(i), key);
65                  if(found != null)
66                  {
67                      break;
68                  }
69              }
70          }
71          return found;
72      }
73
74      public static int t_add_child(TreeRoot t, String parent_key,
75              String child_key, Comparable child_data)
76      {
77          TreeNode parent, child;
78          int rc;
79
80          assert (t != null);
81          assert (parent_key != null);
82          assert (child_key != null);
83          parent = find_node(t.getRoot(), parent_key);
84          if(parent == null)
85          {
86              System.out.format("Parent-Key %s not found%n", parent_key);
87              rc = -1;
88          }
89          else
90          {
91              child = init_tree_node(child_key, child_data, parent);
92              assert (child != null);
93              parent.appendChild(child);
94              rc = 0;
95          }
96          return rc;
97      }
98
99      public static int t_search(TreeRoot t, String key)
100     {
101         int rc;
102         TreeNode node;
103
104         rc = -1;
105         assert (t != null);
106         assert (key != null);
107         node = find_node(t.getRoot(), key);
108         if(node != null)
109         {
110             System.out.format("Cotents are %s%n", node.getData());
111             rc = 0;
112         }
113         return rc;
114     }
115
116     public static int t_are_siblings(TreeRoot t, String key1, String key2)
```

```
117     {
118        TreeNode node1, node2;
119        int rc;
120
121        assert (t != null);
122        assert (key1 != null);
123        assert (key2 != null);
124        rc = 0;
125        node1 = find_node(t.getRoot(), key1);
126        if(node1 == null)
127        {
128           rc = -1;
129        }
130        else
131        {
132           node2 = find_node(t.getRoot(), key2);
133           if(node2 == null)
134           {
135              System.out.format("Key %s not found%n", key2);
136              rc = -1;
137           }
138           else
139           {
140              System.out.format("Nodes %s and %s %s siblings.%n", key1, key2,
141                    (node1.getParent() == node2.getParent() ? "are"
142                    : "are NOT"));
143           }
144        }
145        return rc;
146     }
147
148     public static void print_tree_nodes(TreeNode tn, int level)
149     {
150        int i;
151
152        assert (tn != null);
153
154        for(i = 0; i < level; i++)
155        {
156           System.out.print("  ");
157        }
158        System.out.format("Node (Level %d): Key '%s', Contents '%s'%n", level,
159              tn.getKey(), tn.getData());
160        for(i = 0; i < tn.getNChildren(); i++)
161        {
162           print_tree_nodes(tn.getChild(i), level + 1);
163        }
164     }
165
166     public static int t_print(TreeRoot t)
167     {
168        int rc = -1;
169
170        assert (t != null);
171        if(t.getRoot() != null)
172        {
173           print_tree_nodes(t.getRoot(), 1);
174           rc = 0;
175        }
176
177        return rc;
178     }
179
180     /*
181      * Hier beginnt die Testumgebung. Bitte die Testumgebung nicht testen, keine
182      * Abstraktionen bilden etc.
183      */
184     public static void fuehre_kommandos_aus(File filep) throws IOException
185     {
186        String buf;
187        String kommando, arg1, arg2, arg3;
188        TreeRoot mytree = null;
189
190        //OurFileReader é um leitor de arquivo
191        OurFileReader ofr = new OurFileReader(filep, 1024);
192
```

```
193         while(ofr.ready())
194         {
195             buf = ofr.readLine();
196             //trim() remove todos espaços no começo e no fim de uma String
197             buf = buf.trim();
198 
199             System.out.format("%nLine `%s' was evaluated:%n", buf);
200 
201             //Scanner é uma classe para quebrar uma String em vários tokens
202             Scanner bs = new Scanner(buf);
203 
204             kommando = "";
205             arg1 = "";
206             arg2 = "";
207             arg3 = "";
208 
209             if(bs.hasNext())
210             {
211                 kommando = bs.next();
212             }
213             if(bs.hasNext())
214             {
215                 arg1 = bs.next();
216             }
217             if(bs.hasNext())
218             {
219                 arg2 = bs.next();
220             }
221             if(bs.hasNext())
222             {
223                 arg3 = bs.next();
224             }
225             bs.close();
226 
227             if("root".compareTo(kommando) == 0)
228             {
229                 mytree = t_root(new String(arg1), new String(arg2));
230             }
231             else if("child".compareTo(kommando) == 0)
232             {
233                 t_add_child(mytree, arg1, new String(arg2), new String(arg3));
234             }
235             else if("search".compareTo(kommando) == 0)
236             {
237                 t_search(mytree, arg1);
238             }
239             else if("sibs".compareTo(kommando) == 0)
240             {
241                 t_are_siblings(mytree, arg1, arg2);
242             }
243             else if("print".compareTo(kommando) == 0)
244             {
245                 t_print(mytree);
246             }
247             else
248             {
249                 System.out.format("Command `%s' unknown%n", kommando);
250             }
251         }
252         ofr.close();
253     }
254 
255     public static void main(String[] args) throws IOException
256     {
257         File filep;
258         String file;
259 
260         if(args.length != 1)
261         {
262             System.err.print("Usage: ntree File\n");
263         }
264         else
265         {
266             file = args[0];
267             filep = new File(file);
268             if(!filep.exists())
```

```
269             {
270                 System.err.format("Error opening %s%n", file);
271             }
272             else
273             {
274                 System.out.format("File `%s' was evaluated.%n", file);
275                 fuehre_kommandos_aus(filep);
276                 System.out.format("End of File `%s'.%n", file);
277             }
278         }
279         System.exit(0);
280     }
281 }
```

Código do programa Nametbl

Implementa funções para o gerenciamento de uma tabela de símbolos

```
1   package br.ufscar.dc.lapes.crista.estudoCaso.nametbl;
2
3   import java.io.File;
4   import java.io.IOException;
5   import java.util.Iterator;
6   import java.util.Scanner;
7
8   import util.OurFileReader;
9   import br.ufscar.dc.lapes.crista.estudoCaso.nametbl.util.NameTable;
10  import br.ufscar.dc.lapes.crista.estudoCaso.nametbl.util.NameTableEntry;
11  import br.ufscar.dc.lapes.crista.estudoCaso.nametbl.util.objectType;
12  import br.ufscar.dc.lapes.crista.estudoCaso.nametbl.util.resourceType;
13
14  public class nametbl
15  {
16      public static NameTable newtable()
17      {
18          NameTable ptr = new NameTable();
19          /*
20           * assert verifica se a condição é VERDADEIRA.
21           * Se por algum motivo a expressão for FALSA,
22           * o programa para e lança uma exceção.
23           */
24          assert (ptr != null);
25
26          return ptr;
27      }
28
29      public static int how_many(NameTable nt)
30      {
31          return nt.getNumitems();
32      }
33
34      public static void print_entry(NameTableEntry node)
35      {
36          System.out.format("Name    : %s%n", node.getName());
37          System.out.format("oType   : %s%n", node.getOt().name());
38          System.out.format("rType   : %s%n" + node.getRt().name());
39          System.out.print("-----\n");
40      }
41
42      public static void insert_entry(NameTable nt, String new_name,
43              objectType new_ot, resourceType new_rt)
44      {
45          NameTableEntry nte, result;
46
47          result = null;
48          nte = new NameTableEntry();
49          assert (nte != null);
50
51          nte.setName(new String(new_name));
```

```
52          assert (nte.getName() != null);
53          nte.setOt(new_ot);
54          nte.setRt(new_rt);
55
56          //nt.addEntry(String chave, NameTableEntry elemento)
57          result = nt.addEntry(new_name, nte);
58
59          assert (result != null);
60      }
61
62      public static NameTableEntry retrieve_entry(NameTable nt, String searchname)
63      {
64          NameTableEntry key, retval;
65
66          key = new NameTableEntry();
67          key.setName(searchname);
68
69          //nt.getEntry recupera o elemento da tabela a partir da sua chave.
70          //Se o elemento não existir na tabela, é retornado null.
71          retval = nt.getEntry(searchname);
72
73          return retval;
74      }
75
76      public static int setObjType(NameTable nt, String searchname,
77              objectType new_ot)
78      {
79          int retval;
80          NameTableEntry nte;
81
82          retval = -1;
83          nte = retrieve_entry(nt, searchname);
84          if(nte != null)
85          {
86              retval = 0;
87          }
88
89          return retval;
90      }
91
92      public static int setResType(NameTable nt, String searchname,
93              resourceType new_rt)
94      {
95          int retval;
96          NameTableEntry nte;
97
98          retval = -1;
99          nte = retrieve_entry(nt, searchname);
100         if(nte != null)
101         {
102             retval = 0;
103         }
104         nte.setRt(new_rt);
105
106         return retval;
107     }
108
109     public static int ins(NameTable nt, String new_name)
110     {
111         int retval;
112         NameTableEntry nte;
113
114         nte = retrieve_entry(nt, new_name);
115         if(nte != null)
116         {
117             System.out.format("Name `%s' is already in the Tble.%n", new_name);
118             retval = -1;
119         }
120         else
121         {
122             insert_entry(nt, new_name, objectType.OT_NO_INF, null);
123             retval = 0;
124         }
125
126         return retval;
127     }
```

```
128
129     public static int tot(NameTable nt, String new_name, String new_ot_char)
130     {
131        int rc;
132        objectType new_ot;
133
134        new_ot = null;
135        if("SYSTEM".compareTo(new_ot_char) == 0)
136        {
137           new_ot = objectType.SYSTEM;
138        }
139        else if("RESOURCE".compareTo(new_ot_char) == 0)
140        {
141           new_ot = objectType.RESOURCE;
142        }
143
144        if(new_ot == null)
145        {
146           System.out.format("Type `%s' unknown%n", new_ot_char);
147           rc = -1;
148        }
149        else
150        {
151           rc = setObjType(nt, new_name, new_ot);
152           if(rc == -1)
153           {
154              System.out.format("Name `%s' is not in the Table.%n", new_name);
155           }
156        }
157
158        return rc;
159     }
160
161     public static int trt(NameTable nt, String new_name, String new_rt_char)
162     {
163        int rc;
164
165        resourceType new_rt;
166
167        new_rt = null;
168
169        if("RT_SYSTEM".compareTo(new_rt_char) == 0)
170        {
171           new_rt = resourceType.RT_SYSTEM;
172        }
173        else if("FUNKTION".compareTo(new_rt_char) == 0)
174        {
175           new_rt = resourceType.FUNCTION;
176        }
177        else if("DATA".compareTo(new_rt_char) == 0)
178        {
179           new_rt = resourceType.DATA;
180        }
181
182        if(new_rt == null)
183        {
184           System.out.format("Type `%s' unknown%n", new_rt_char);
185           rc = -1;
186        }
187        else
188        {
189           rc = setResType(nt, new_name, new_rt);
190           if(rc == -1)
191           {
192              System.out.format("Name `%s' is not in the Table.%n", new_name);
193           }
194        }
195        return rc;
196     }
197
198     public static void sch(NameTable nt, String name)
199     {
200        NameTableEntry nte;
201
202        System.out.format("Search by Name `%s':%n", name);
203        nte = retrieve_entry(nt, name);
```

```
204         if(nte != null)
205         {
206             print_entry(nte);
207         }
208     }
209
210     public static void prt(NameTable nt)
211     {
212         System.out.format("Table has the following %d entries:%n", how_many(nt));
213
214         //Iterator é um objeto para acessar os itens de uma lista/array/tabela...
215         Iterator it = nt.getEntriesIterator();
216         while(it.hasNext())
217         {
218             NameTableEntry nte = (NameTableEntry) it.next();
219             print_entry(nte);
220         }
221     }
222
223     public static void fuehre_kommandos_aus(File filep, NameTable nt)
224             throws IOException
225     {
226         String buf;
227         String kommando, name, typ;
228
229         nt = newtable();
230         assert (nt != null);
231
232         //OurFileReader é um leitor de arquivo
233         OurFileReader ofr = new OurFileReader(filep, 1024);
234
235         while(ofr.ready())
236         {
237             buf = ofr.readLine();
238             //trim() remove todos espaços no começo e no fim de uma String
239             buf = buf.trim();
240
241             System.out.format("%nThe line `%s' was evaluated:%n", buf);
242             //Scanner é uma classe para quebrar uma String em vários tokens
243             Scanner bs = new Scanner(buf);
244
245             kommando = "";
246             name = "";
247             typ = "";
248
249             if(bs.hasNext())
250             {
251                 kommando = bs.next();
252             }
253             if(bs.hasNext())
254             {
255                 name = bs.next();
256             }
257             if(bs.hasNext())
258             {
259                 typ = bs.next();
260             }
261             bs.close();
262
263             if("ins".compareTo(kommando) == 0)
264             {
265                 ins(nt, name);
266             }
267             else if("tot".compareTo(kommando) == 0)
268             {
269                 tot(nt, name, typ);
270             }
271             else if("trt".compareTo(kommando) == 0)
272             {
273                 trt(nt, name, typ);
274             }
275             else if("sch".compareTo(kommando) == 0)
276             {
277                 sch(nt, name);
278             }
279             else if("prt".compareTo(kommando) == 0)
```

```
280             {
281                 prt(nt);
282             }
283             else
284             {
285                 System.out.format("Command `%s' unknown%n", kommando);
286             }
287         }
288         ofr.close();
289     }
290
291     public static void main(String[] args) throws IOException
292     {
293         NameTable nt = null;
294         File filep;
295         String file;
296
297         if(args.length != 1)
298         {
299             System.err.print("Usage: nametbl File\n");
300         }
301         else
302         {
303             file = args[0];
304             filep = new File(file);
305             if(!filep.exists())
306             {
307                 System.err.format("Error opening %s%n", file);
308             }
309             else
310             {
311                 System.out.format("File `%s' was processed.%n", file);
312                 fuehre_kommandos_aus(filep, nt);
313                 System.out.format("End of File `%s'.%n", file);
314             }
315         }
316         System.exit(0);
317     }
318 }
```

Apêndice D

QUESTIONÁRIOS APLICADOS NOS ESTUDOS EXPERIMENTAIS 1 E 2

Questionário Q1 – Caracterização dos participantes

QUESTIONÁRIO

CARACTERIZAÇÃO DOS PARTICIPANTES

A) DADOS PESSOAIS

1. Idade: _____
2. Sexo
   ( )Masculino ( )Feminino
3. Semestre:_____
4. Possui computador em casa
   ( )Sim ( )Não
5. Possui acesso à internet
   ( )Sim ( )Não

B) CARACTERÍSTICAS TÉCNICAS

6. Leitura em Inglês
   ( ) Bem ( )Médio ( )Ruim
7. Qual sua experiência anterior com o desenvolvimento de software?
   ( )Nunca desenvolvi software
   ( )Já desenvolvi software sozinho
   ( )Já desenvolvi software em grupo na aula
   ( )Já desenvolvi software em grupo em empresas
8. Quanto tempo você tem de experiência em programação?
   _____
9. Quais linguagens de programação você conhece?
   _____
10. Qual a sua experiência em inspeção de software?
    ( )Nenhuma
    ( )Estudei em aula ou em livros
    ( )Participei em um projeto em aula
    ( )Usei em um projeto de empresas
    ( )Usei em vários projetos de empresas

11. Quanto tempo você tem de experiência em inspeção de software?

_____

12. Você já usou algum software de visualização de informações? Qual?

_____

Questionário Q2 – Avaliação do treinamento oferecido

TREINAMENTO NA TÉCNICA *STEPWISE ABSTRACTION*

1. Quão eficaz você acredita que tenha sido o treinamento? Ele ajudou você a compreender melhor os procedimentos de inspeção com a técnica *Stepwise Abstraction* (SA)? Tem alguma coisa faltando, ou alguma coisa que você acredita que poderia ser feita melhor?
   ( ) Foi eficaz, ajudou a entender o processo de inspeção
   ( ) Foi eficaz, ajudou a entender o processo de inspeção, mas o tempo precisava ser maior
   ( ) Seria mais eficaz se houvesse mais tempo para exemplos
   ( ) Técnica bem intuitiva, mas é necessária uma boa experiência para aplicá-la conforme as regras e tempo estimado
   ( ) Deveria ser mostrado um modelo seguindo passo a passo todos os possíveis detalhes que podem ocorrer durante a inspeção
   ( ) Mostrou que a inspeção serve para encontrar as discrepâncias entre as especificações e não os defeitos no código
2. Os conceitos e termos utilizados pela SA tiveram algum efeito em como você a utilizou?
   ( )Sim ( )Não

3. Você ficou em dúvida sobre algum conceito devido à técnica SA? Se sim, como você tratou isso?
   ( ) Sim. Relatou que pediu explicação para a professora
   ( ) Sim. Revisou o material de treinamento sobre SA.
   ( ) Sim. Relatou que a explicação no material de treinamento não está completo.
   ( ) Teve algumas dúvidas e por isso não aplicou a técnica.
   ( ) Não.
4. A técnica foi eficaz em ajudá-lo a identificar as discrepâncias entre a sua especificação e a especificação original?.
   ( )Sim ( )Não
5. A técnica foi aplicada o tempo todo?
   ( )Sim ( )Não
   Se não, por quê?

   _____
6. Além dos conhecimentos adquiridos no treinamento, você precisou de outras informações para executar técnica SA?
   ( )Sim ( )Não
   Se sim, quais?

   _____

Questionário Q3 – Avaliação da ferramenta CRISTA

CARACTERIZAÇÃO DA FERRAMENTA CRISTA

A) USABILIDADE

Para as questões indicadas, responda usando a escala de 1 a 5, onde:

1= Nunca; 2= Raramente; 3= As vezes; 4= Freqüentemente; 5= Sempre

I) Aprendizado

1. É fácil aprender a usar a ferramenta? (1 a 5)_____

2. As ajudas (Help e mensagens dos wizards) providas pela ferramenta são fáceis de serem entendidas? (1 a 5)_____
3. É fácil encontrar ajuda quando necessário? (1 a 5)_____

II) Eficiência

4. As informações (mensagens na tela) providas pela ferramenta são claras? (1 a 5)_____
5. As informações dadas no help ajudam efetivamente a completar as tarefas? (1 a 5)_____
6. A ferramenta sempre deixa o usuário informado do que está acontecendo? (1 a 5)_____
7. A ferramenta oferece atalhos para acelerar o uso para usuários experientes? (1 a 5)_____
8. A ferramenta oferece boas explicações para usuários novatos? (1 a 5)_____

III) Memorização

9. A ferramenta segue um padrão nas mensagens? (Isto é, nunca são usados termos diferentes para se referir às mesmas coisas)(1 a 5)_____
10. Caso algum engano seja cometido, o usuário pode interromper o procedimento no meio? (Isto é, não existe a necessidade de memorizar todos os procedimentos existentes e/ou suas seqüências) (1 a 5)_____

IV) Poucos erros

11. Quando um engano é cometido usando a ferramenta, a recuperação é fácil e rápida? (1 a 5)_____
12. A ferramenta provê boas mensagens de erro, mostrando claramente como resolver o problema? (1 a 5)_____

V) Satisfação do usuário

13. A ferramenta é fácil de usar? (1 a 5) _____
14. A ferramenta sempre usa uma linguagem apropriada? (Isto é, não é usada linguagem técnica e/ou específica que possa confundir o usuário) (1 a 5)_____
15. Foi confortável usar a ferramenta? (Isto é, não ocorreu nenhum evento que ocasionasse desânimo ou constrangimento ao usar a ferramenta) (1 a 5)_____
16. As informações na tela estão bem organizadas e de maneira clara? (1 a 5)_____
17. A interface é simples e concisa? (Isto é, não existem informações excessivas na tela, poluindo a interface? (1 a 5)_____
18. A interface gráfica da ferramenta é agradável? (Isto é, não provoca nenhum tipo de desconforto ou cansaço) (1 a 5)_____

B) FUNCIONALIDADES X STEPWISE

19. A ferramenta tem todas as funcionalidades que se esperava possuir? (1 a 5)_____
20. A ferramenta oferece tudo o que é necessário para aplicar a técnica *Stepwise Abstraction*? (1 a 5)_____
21. O trabalho fica mais rápido usando a ferramenta ao invés de fazer manualmente? (1 a 5)_____
22. A inspeção usando a ferramenta é tão ou mais eficiente que a inspeção manual (a eficiência é uma medida relacionada com o tempo)? (1 a 5)_____
23. A ferramenta possui tudo o que é necessário para identificar e abstrair cada instrução? (1 a 5)_____
24. A ferramenta possui o que é necessário para relatar as discrepâncias? (1 a 5)_____
25. A visualização das instruções do código influenciou na inspeção? (1 a 5)_____
26. Alguma outra forma de visualização poderia auxiliar mais que a atual?
    ( )Sim ( )Não
    Se sim, qual?
    _____
27. Você encontrou alguma discrepância que não relatou?
    ( ) Sim, pois faltou tempo.
    ( ) Não.

C) RELATÓRIOS

28. Você usou a opção de gerar um algoritmo do código?
    ( )Sim ( )Não
    Se respondeu Sim, o algoritmo do código foi gerado corretamente? (1 a 5)_____
29. Você usou a opção de comentar o código?
    ( )Sim ( )Não
    Se respondeu Sim, os comentários no código foram colocados corretamente?
    (1 a 5)_____
30. Você usou a opção de exportar o código com as abstrações parciais?
    ( )Sim ( )Não
    Se respondeu Sim, o código com as abstrações parciais foi gerado corretamente?

(1 a 5)_____

31. Você usou a opção de gerar o relatório com a descrição funcional do código?
    ( )Sim ( )Não
    Se respondeu Sim, a descrição funcional do código foi gerada corretamente?
    (1 a 5)_____
32. Você usou a opção de gerar o relatório com as discrepâncias de todos os inspetores?
    ( )Sim ( )Não
    Se respondeu Sim, o relatório com as discrepâncias dos inspetores foi gerado corretamente? (1 a 5)_____

D) ESTUDOS DE CASO

33. Você acha que a visualização com Treemap facilitou a inspeção com a ferramenta? (1 a 5)_____
34. Você acha que o uso da ferramenta ajudou a encontrar mais discrepâncias?(1 a 5)_____
35. Você acha que o uso da ferramenta deixou o processo de inspeção mais rápido? (1 a 5)_____
36. O que você considera que tenha dificultado a identificação das discrepâncias?

( ) O Treemap ( ) A ferramenta como um todo ( ) O conhecimento em Java ( ) A complexidade dos programas

E) COMENTÁRIO GERAL

37. De maneira geral, você ficou satisfeito com a ferramenta? (1 a 5)_____
38. Quais são os pontos negativos da ferramenta?

    _____
    _____
    _____
    _____
    _____

39. Quais são os pontos positivos da ferramenta?

    _____
    _____
    _____
    _____
    _____

40. Outros comentários:

    _____
    _____
    _____
    _____
    _____
    _____
    _____
    _____
    _____
    _____
    _____

Apêndice E

DADOS DOS ESTUDOS EXPERIMENTAIS 1 E 2

Dados do Grupo G dos estudos experimentais 1 e 2 (antes e depois de receber o treinamento da ferramenta)

A seguir são apresentados nas figuras um comparativo entre os dados obtidos dos questionários dos participantes antes do treinamento da ferramenta (estudo experimental 1) e depois do treinamento da ferramenta (estudo experimental 2).

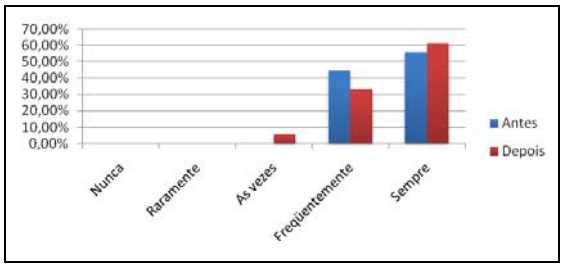


Figura 1 - As ajudas providas pela ferramenta são fáceis de serem entendidas? Correspondente à questão 2 do questionário Q3

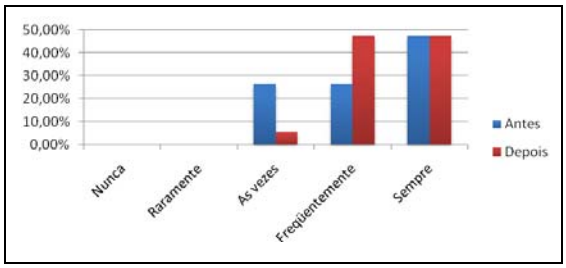


Figura 2 - As informações providas pela ferramenta são claras? Correspondente à questão 4 do questionário Q3

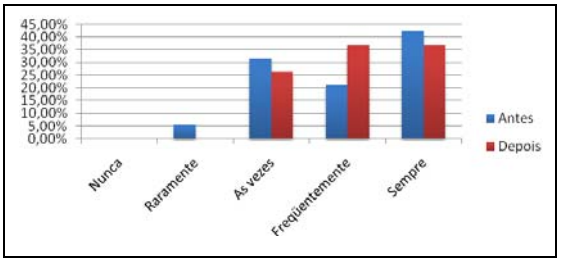


Figura 3 - A ferramenta sempre deixa o usuário informado do que está acontecendo? Correspondente à questão 6 do questionário Q3

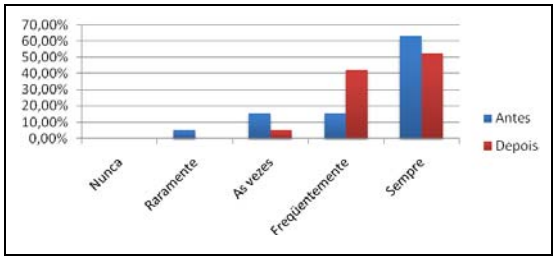


Figura 4 - As informações na tela estão bem organizadas e de maneira clara? Correspondente à questão 16 do questionário Q3

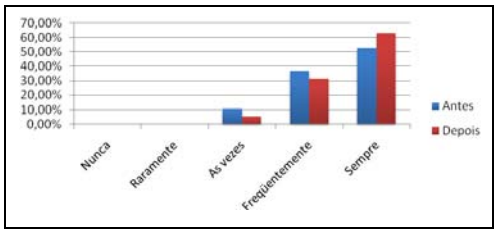


Figura 5 - A ferramenta sempre usa uma linguagem apropriada? Correspondente à questão 14 do questionário Q3

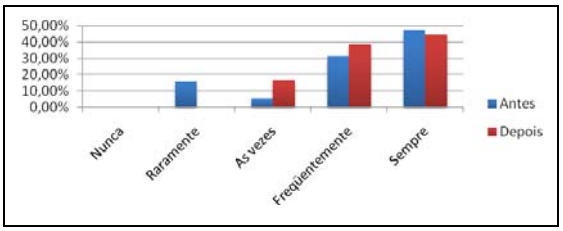


Figura 6 - Caso algum engano seja cometido, o usuário pode interromper o procedimento no meio? Correspondente à questão 10 do questionário Q3

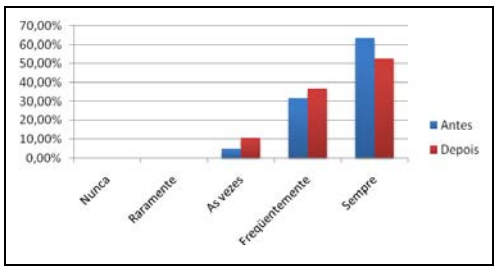


Figura 7 - Em algum momento você pensou em desistir de usar a ferramenta por ela não ser apropriada para o propósito que você esperava? Correspondente à questão 15 do questionário Q3

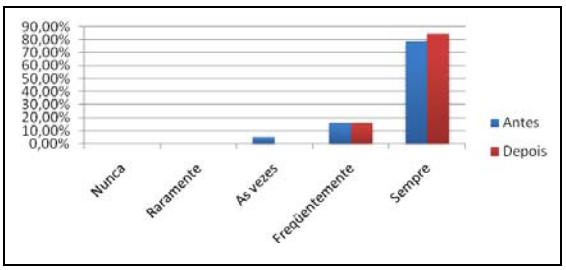


Figura 8 - A ferramenta segue um padrão nas mensagens? Correspondente à questão 9 do questionário Q3

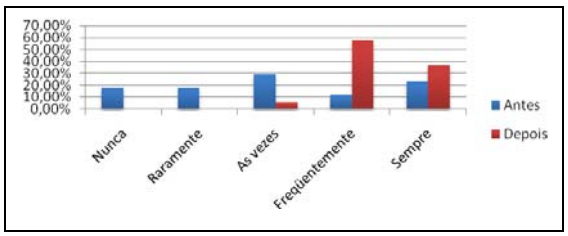


Figura 9 - A ferramenta provê boas mensagens de erro, mostrando claramente como resolver o problema? Correspondente à questão 12 do questionário Q3

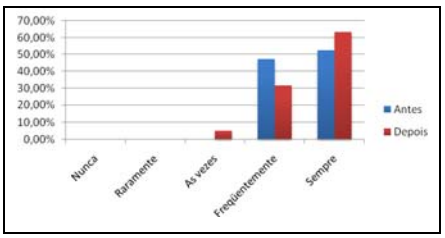


Figura 10 - É fácil aprender a usar a ferramenta? Correspondente à questão 1 do questionário Q3

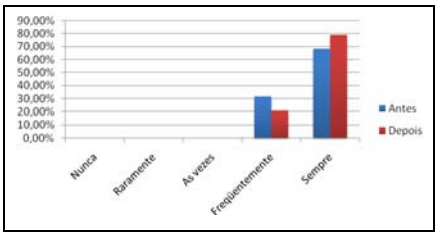


Figura 11 - A ferramenta é fácil de usar? Correspondente à questão 13 do questionário Q3

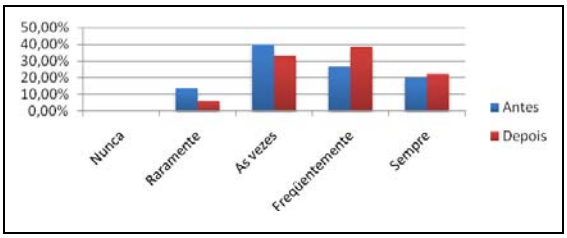


Figura 12 - A ferramenta oferece atalhos para acelerar o uso para usuários experientes? Correspondente à questão 7 do questionário Q3

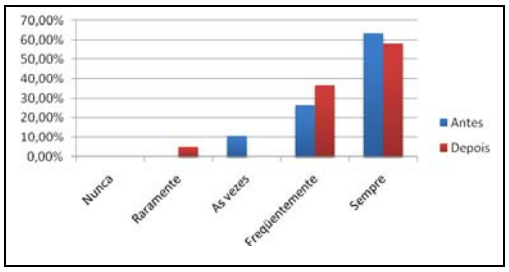


Figura 13 - A interface é simples e concisa? Correspondente à questão 17 do questionário Q3

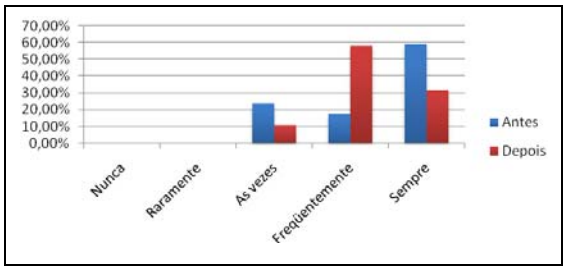


Figura 14 - A interface gráfica da ferramenta é agradável? Correspondente à questão 18 do questionário Q3

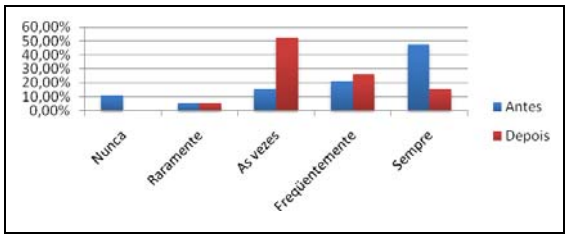


Figura 15 - Quando um engano é cometido usando a ferramenta, a recuperação é fácil e rápida? Correspondente à questão 11 do questionário Q3

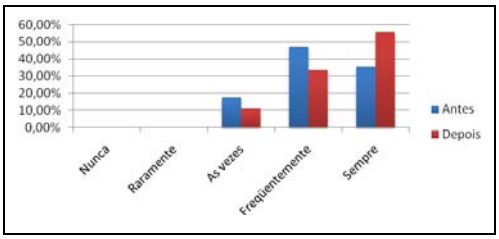


Figura 16 - É fácil encontrar ajuda quando necessário? Correspondente à questão 3 do questionário Q3

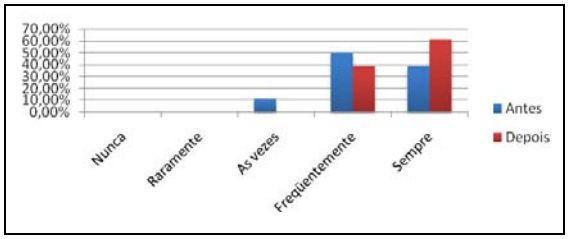


Figura 17 - As informações dadas no help ajudam efetivamente a completar as tarefas? Correspondente à questão 5 do questionário Q3

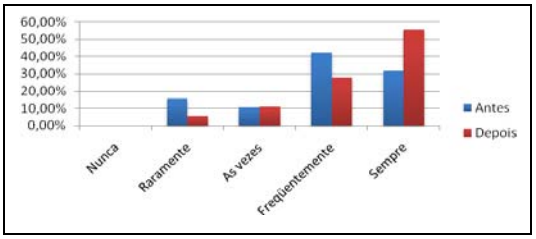


Figura 18 - A ferramenta oferece boas explicações para usuários novatos? Correspondente à questão 8 do questionário Q3

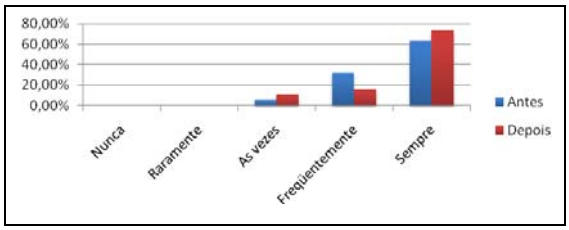


Figura 19 - A ferramenta tem todas as funcionalidades que se espera possuir? Correspondente à questão 19 do questionário Q3

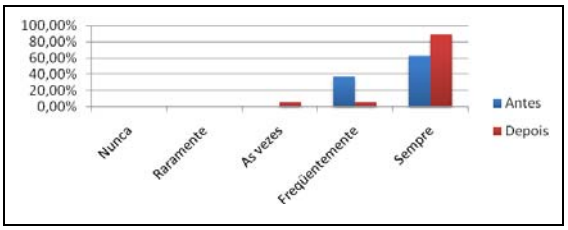


Figura 20 - A ferramenta oferece tudo o que é necessário para aplicar a técnica *Stepwise Abstraction*? Correspondente à questão 20 do questionário Q3

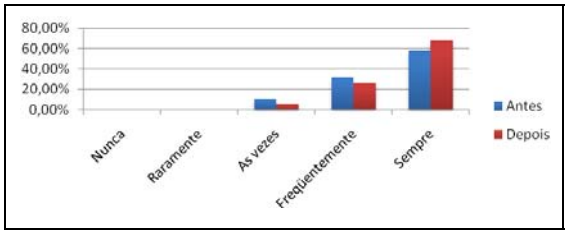


Figura 21 - A ferramenta possui tudo o que é necessário para identificar e abstrair cada instrução? Correspondente à questão 23 do questionário Q3

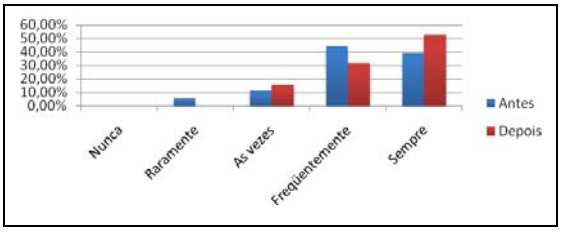


Figura 22 - A ferramenta possui o que é necessário para relatar as discrepâncias? Correspondente à questão 24 do questionário Q3

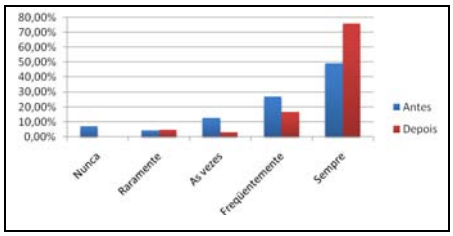


Figura 23 - Os relatórios estavam corretos? Correspondente à média das questões 28 a 32 do questionário Q3

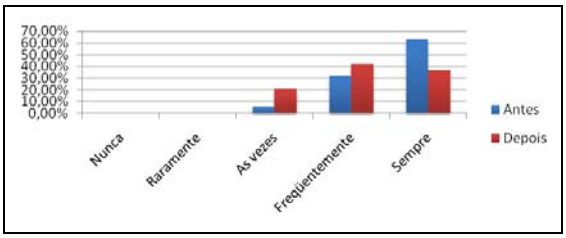


Figura 24 - O trabalho fica mais rápido usando a ferramenta ao invés de fazer manualmente? Correspondente à questão 21 do questionário Q3

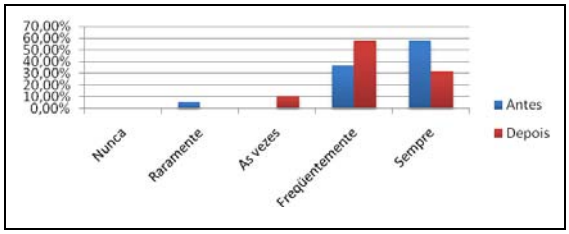


Figura 26 - A inspeção usando a ferramenta é tão ou mais eficiente que a inspeção manual? Correspondente à questão 22 do questionário Q3

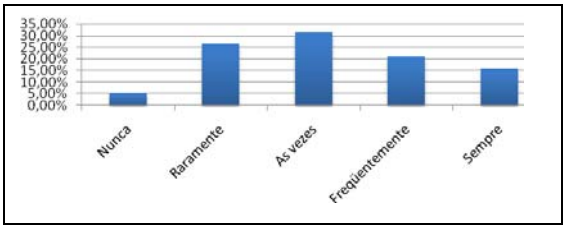


Figura 27 - Você acha que o uso da ferramenta ajudou a encontrar mais discrepâncias? Correspondente à questão 34 do questionário Q3

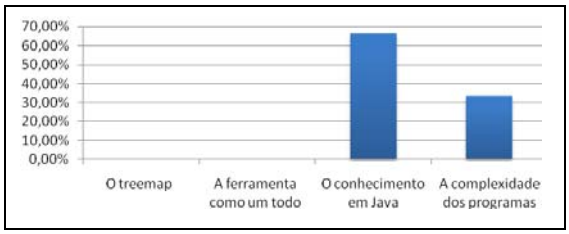


Figura 28 - O que você considera que tenha dificultado a identificação das discrepâncias? Correspondente à questão 36 do questionário Q3

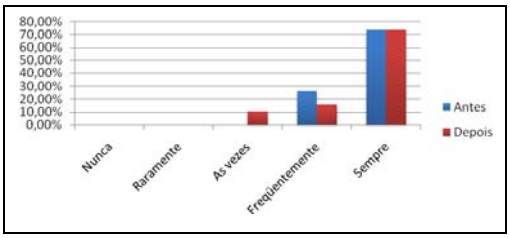


Figura 29 - A visualização das instruções do código influenciou na inspeção? Correspondente à questão 33 do questionário Q3

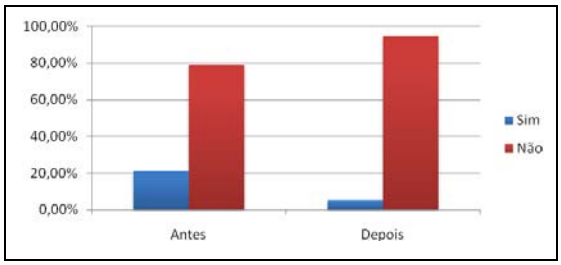


Figura 30 - Alguma outra forma de visualização poderia auxiliar mais que a atual? Correspondente à questão 26 do questionário Q3

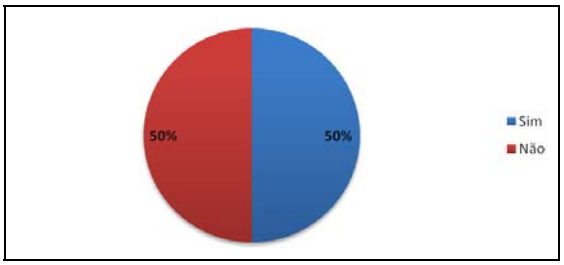


Figura 31 - Porcentagem de alunos que começaram a aprender a usar a ferramenta lendo o help

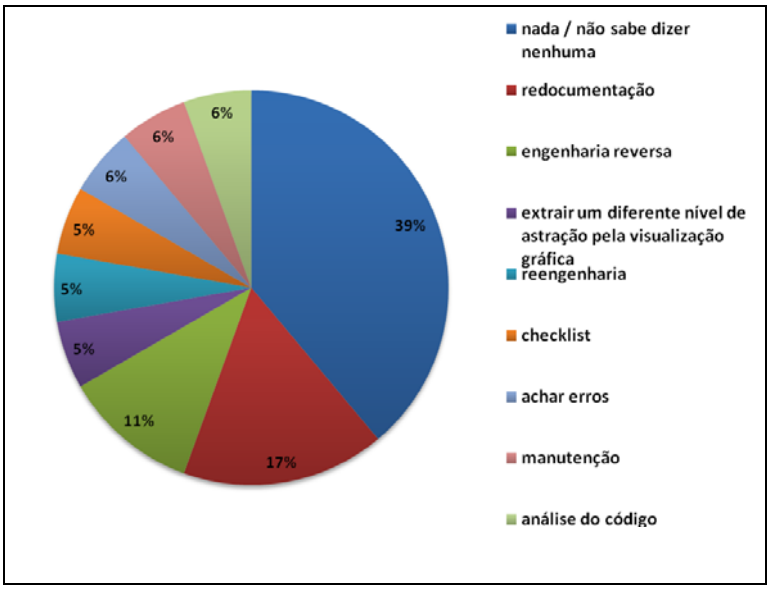


Figura 32 - Outros usos para a ferramenta CRISTA

Dados dos grupos G, M1 e M2 do estudo experimental 2

Nas figuras a seguir serão apresentados os dados dos questionários respondidos ao final do estudo experimental 2 pelos dos grupos G, M1 e M2.

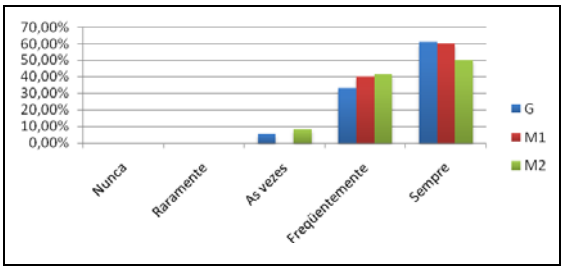


Figura 33 - As ajudas providas pela ferramenta são fáceis de serem entendidas? Correspondente à questão 2 do questionário Q3

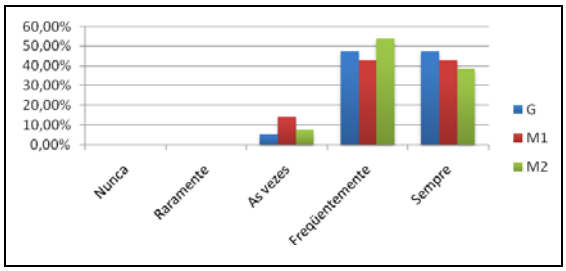


Figura 34 - As informações providas pela ferramenta são claras? Correspondente à questão 4 do questionário Q3

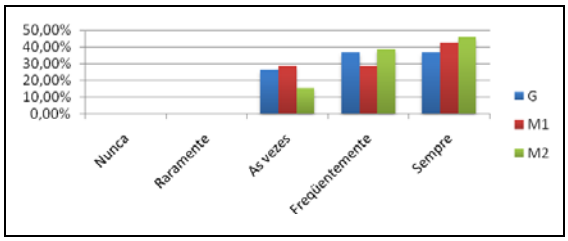


Figura 35 - A ferramenta sempre deixa o usuário informado do que está acontecendo? Correspondente à questão 6 do questionário Q3

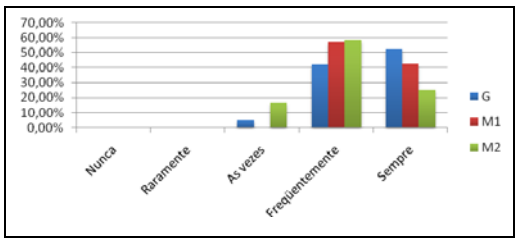


Figura 36 - As informações na tela estão bem organizadas e de maneira clara? Correspondente à questão 16 do questionário Q3

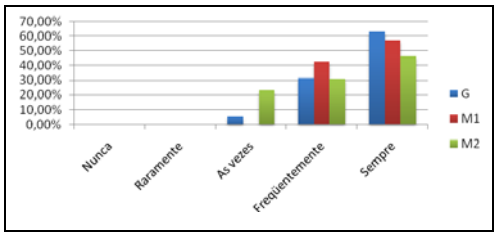


Figura 37 - A ferramenta sempre usa uma linguagem apropriada? Correspondente à questão 14 do questionário Q3

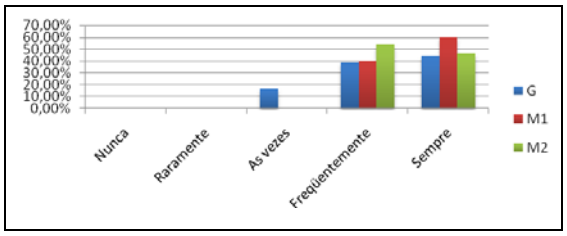


Figura 38 - Caso algum engano seja cometido, o usuário pode interromper o procedimento no meio? Correspondente à questão 10 do questionário Q3

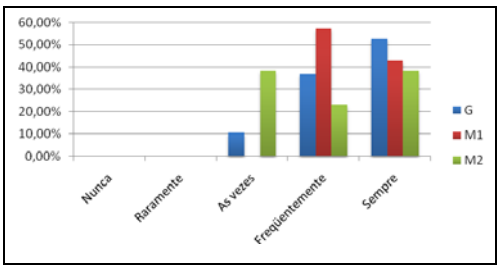


Figura 39 - Em algum momento você pensou em desistir de usar a ferramenta por ela não ser apropriada para o propósito que você esperava? Correspondente à questão 15 do questionário Q3

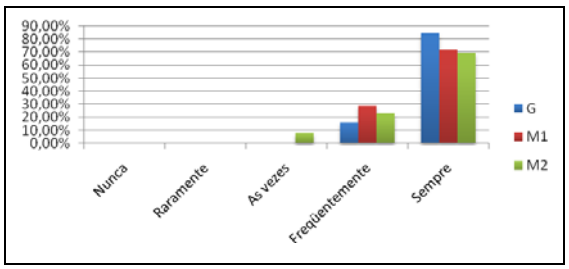


Figura 40 - A ferramenta segue um padrão nas mensagens? Correspondente à questão 9 do questionário Q3

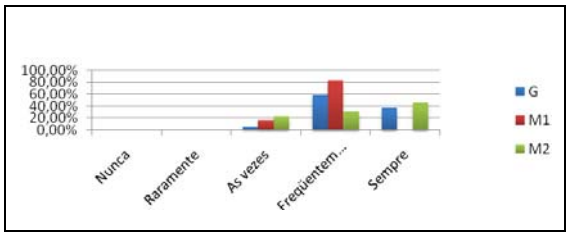


Figura 41 - A ferramenta provê boas mensagens de erro, mostrando claramente como resolver o problema? Correspondente à questão 12 do questionário Q3

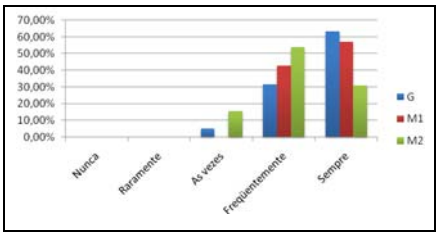


Figura 42 - É fácil aprender a usar a ferramenta? Correspondente à questão 1 do questionário Q3

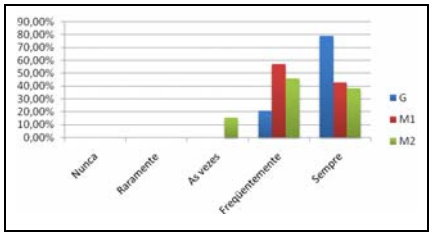


Figura 43 - A ferramenta é fácil de usar? Correspondente à questão 13 do questionário Q3

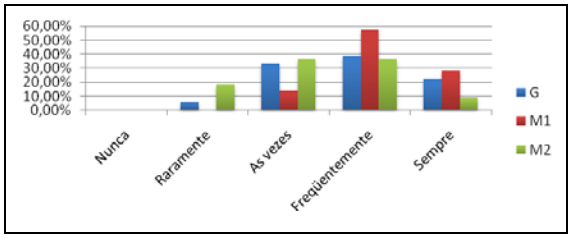


Figura 44 - A ferramenta oferece atalhos para acelerar o uso para usuários experientes? Correspondente à questão 7 do questionário Q3

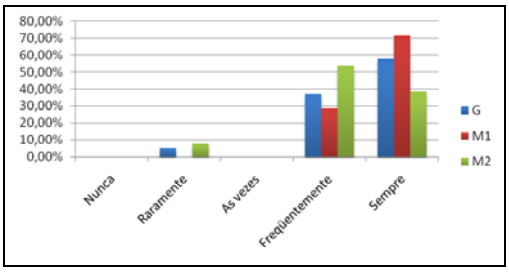


Figura 45 - A interface é simples e concisa? Correspondente à questão 17 do questionário Q3

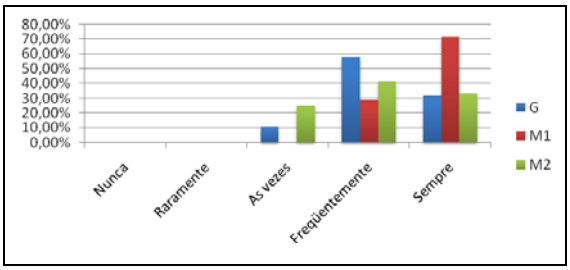


Figura 46 - A interface gráfica da ferramenta é agradável? Correspondente à questão 18 do questionário Q3

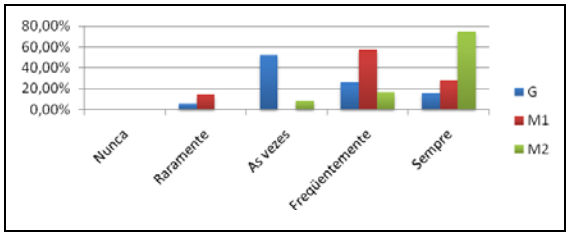


Figura 47 - Quando um engano é cometido usando a ferramenta, a recuperação é fácil e rápida? Correspondente à questão 11 do questionário Q3

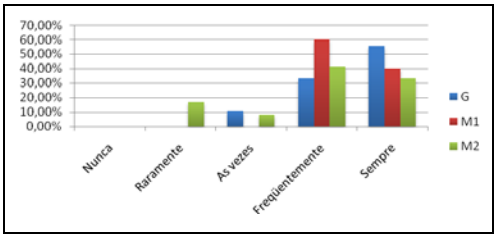


Figura 48 - É fácil encontrar ajuda quando necessário? Correspondente à questão 3 do questionário Q3

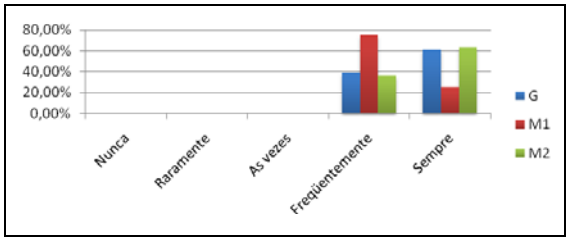


Figura 49 - As informações dadas no help ajudam efetivamente a completar as tarefas? Correspondente à questão 5 do questionário Q3

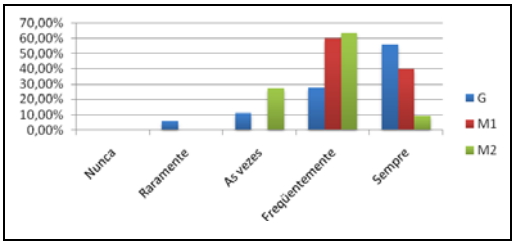


Figura 50 - A ferramenta oferece boas explicações para usuários novatos? Correspondente à questão 8 do questionário Q3

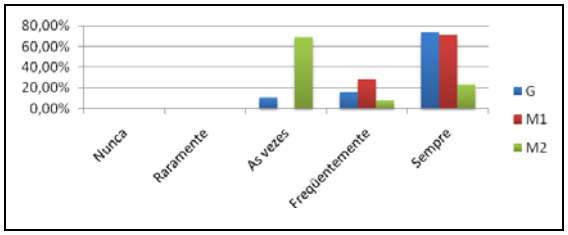


Figura 51 - A ferramenta tem todas as funcionalidades que se espera possuir? Correspondente à questão 19 do questionário Q3

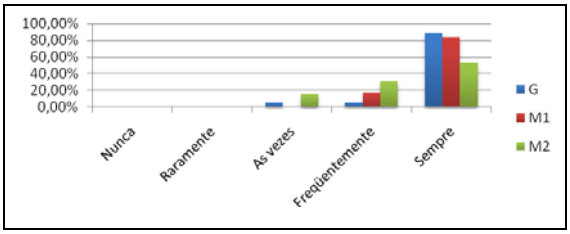


Figura 52 - A ferramenta oferece tudo o que é necessário para aplicar a técnica *Stepwise Abstraction*? Correspondente à questão 20 do questionário Q3

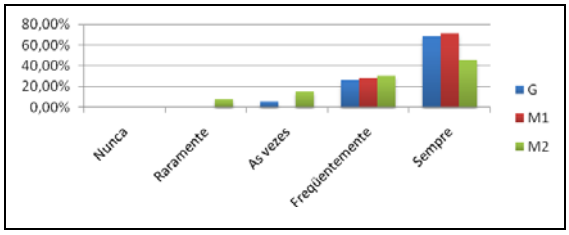


Figura 53 - A ferramenta possui tudo o que é necessário para identificar e abstrair cada instrução? Correspondente à questão 23 do questionário Q3

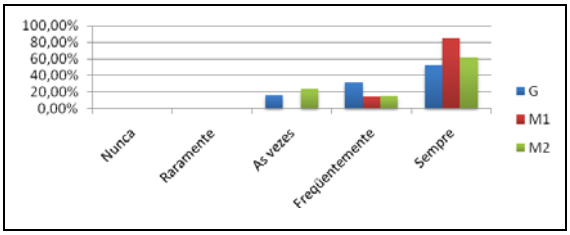


Figura 54 - A ferramenta possui o que é necessário para relatar as discrepâncias? Correspondente à questão 24 do questionário Q3

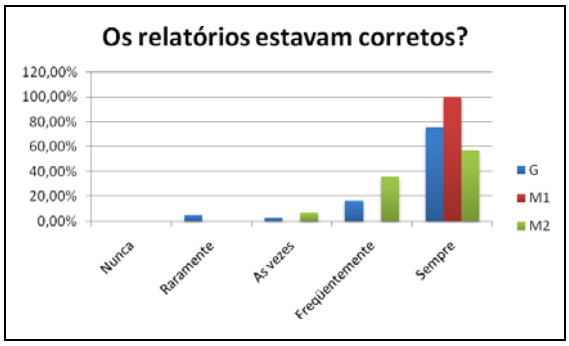


Figura 55 - Os relatórios estavam corretos? Correspondente à média das questões 28 a 32 do questionário Q3

Figura 56 - O trabalho fica mais rápido usando a ferramenta ao invés de fazer manualmente? Correspondente à questão 21 do questionário Q3

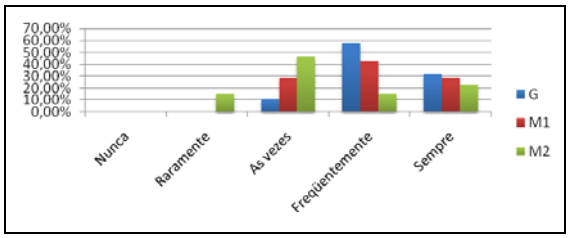


Figura 58 - A inspeção usando a ferramenta é tão ou mais eficiente que a inspeção manual? Correspondente à questão 22 do questionário Q3

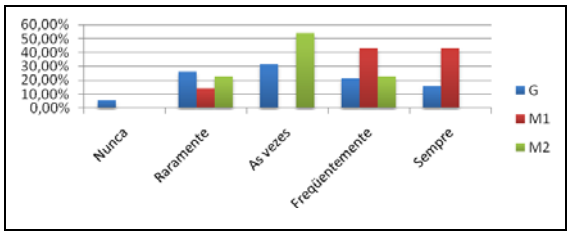


Figura 59 - Você acha que o uso da ferramenta ajudou a encontrar mais discrepâncias? Correspondente à questão 34 do questionário Q3

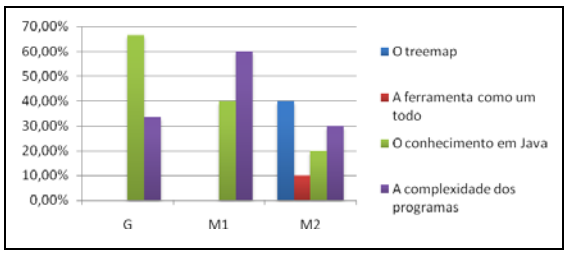


Figura 60 - O que você considera que tenha dificultado a identificação das discrepâncias? Correspondente à questão 36 do questionário Q3

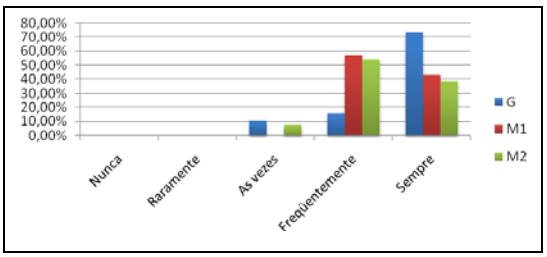


Figura 61 - A visualização das instruções do código influenciou na inspeção? Correspondente à questão 33 do questionário Q3

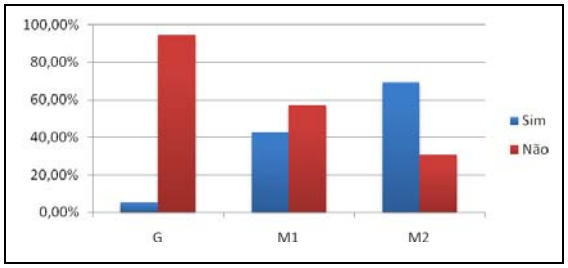


Figura 62 - Alguma outra forma de visualização poderia auxiliar mais que a atual? Correspondente à questão 26 do questionário Q3

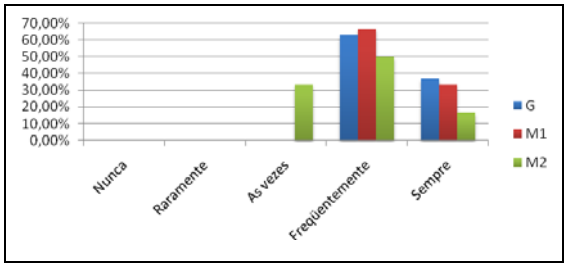


Figura 63 - De maneira geral você ficou satisfeito com a ferramenta? Correspondente à questão 37 do questionário Q3

Tempos gastos e número de discrepâncias encontradas pelos grupos G, M1 e M2 no estudo experimental 2

A seguir serão apresentados os dados referentes aos tempos gasto nas atividades do estudo experimental 2 pelos grupos G, M1 e M2. Serão apresentados também os dados referentes ao número de discrepâncias encontradas durante esse estudo.

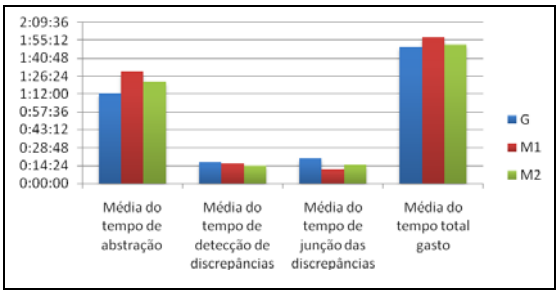


Figura 64 - Média de tempos gastos na inspeção manual do programa Ntree

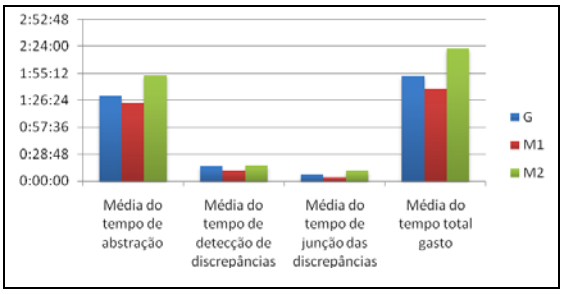


Figura 65 - Média de tempos gastos na inspeção com a CRISTA do programa Ntree

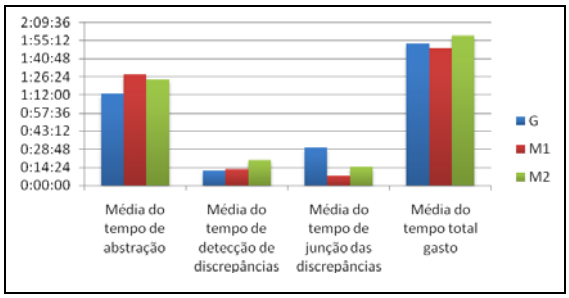


Figura 66 - Média de tempos gastos na inspeção manual do programa Nametbl

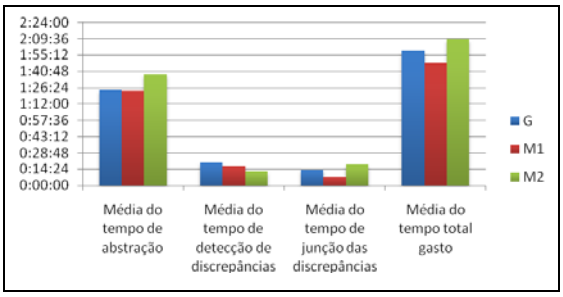


Figura 67 - Média de tempos gastos na inspeção com a CRISTA do programa Nametbl

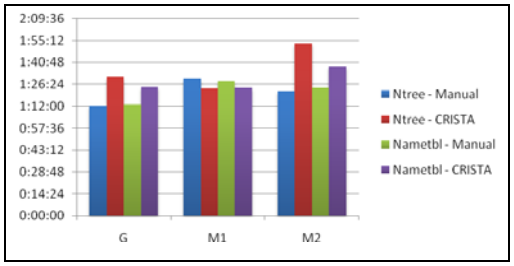


Figura 68 - Média dos tempos de abstração dos programas Ntree e Nametbl com e sem a ferramenta CRISTA

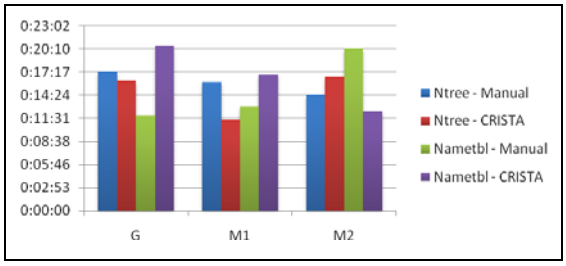


Figura 69 - Média dos tempos de detecção de discrepâncias dos programas Ntree e Nametbl com e sem a ferramenta CRISTA

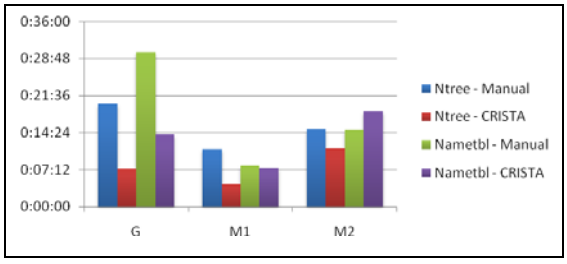


Figura 71 - Média dos tempos de junção das discrepâncias dos programas Ntree e Nametbl com e sem a ferramenta CRISTA

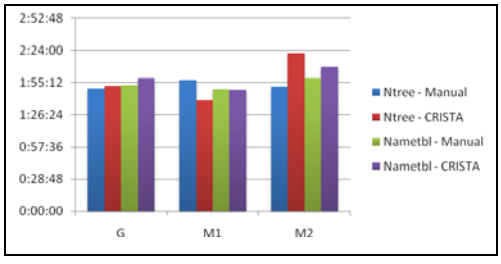


Figura 70 - Média dos tempos totais gastos para a inspeção dos programas Ntree e Nametbl com e sem a ferramenta CRISTA

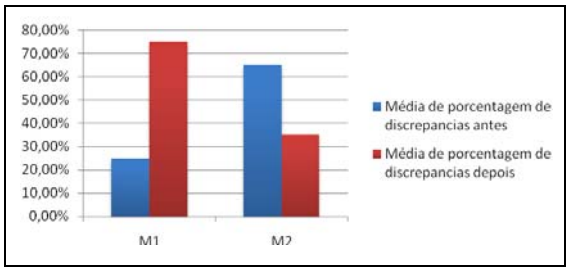


Figura 72 - Média da porcentagem de discrepâncias aceitas para o programa Ntree sem a CRISTA

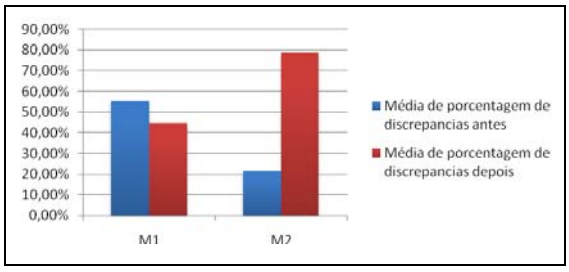


Figura 73 - Média da porcentagem de discrepâncias aceitas para o programa Ntree com a CRISTA

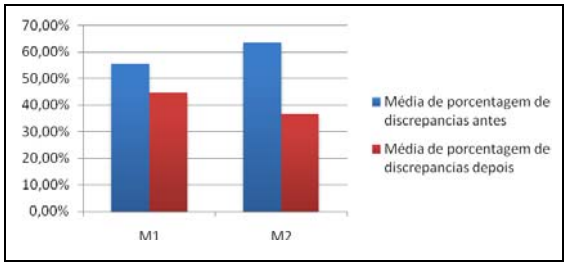


Figura 74 - Média da porcentagem de discrepâncias aceitas para o programa Nametbl sem a CRISTA

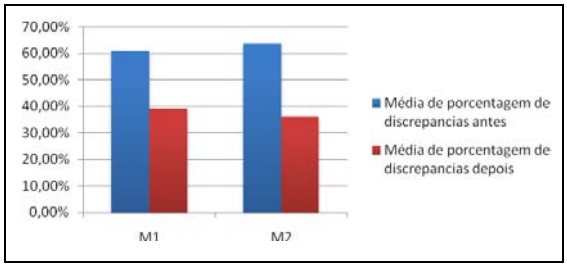


Figura 75 - Média da porcentagem de discrepâncias aceitas para o programa Nametbl com a CRISTA

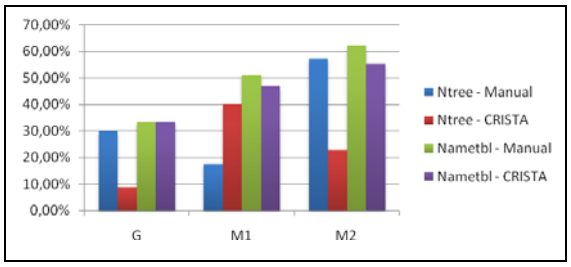


Figura 76 - Média da porcentagem de discrepâncias encontradas para os programas Ntree e Nametbl com e sem a ferramenta CRISTA