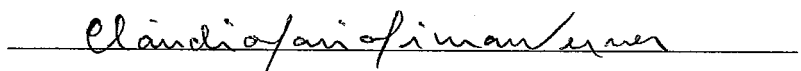
LIGHTHOUSE: UM MECANISMO DE PERCEPÇÃO DE GRUPO BASEADO NO DESIGN EMERGENTE

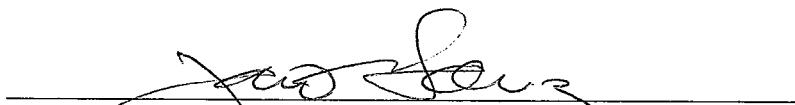
Isabella Almeida da Silva

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

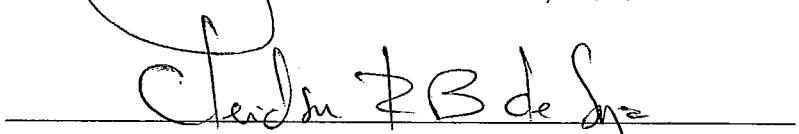
Aprovada por:



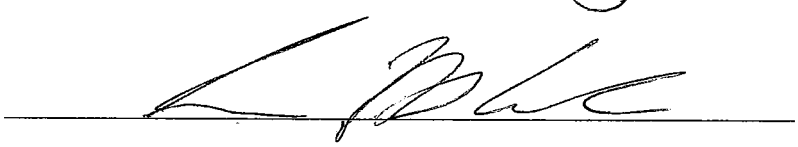
Profa. Cláudia Maria Lima Werner, D.Sc.



Prof. Jano Moreira de Souza, Ph.D.



Prof. Cleidson Ronald Botelho de Souza, Ph.D.



Dr. Leonardo Gresta Paulino Murta , D.Sc.

RIO DE JANEIRO, RJ - BRASIL

ABRIL DE 2008

SILVA, ISABELLA ALMEIDA DA

Lighthouse: Um Mecanismo de Percepção de Grupo Baseado no Design Emergente [Rio de Janeiro] 2008

XIV, 121 p., 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2008)

Dissertação – Universidade Federal do Rio de Janeiro, COPPE

1. Desenvolvimento de Software
2. Trabalho Cooperativo
3. Percepção de Grupo

I. COPPE/UFRJ II. Título (série)

# Agradecimentos

Gostaria de agradecer a todos que, de alguma forma, contribuíram para este trabalho. Aqui vai uma lista (provavelmente incompleta e desordenada) de pessoas que me apoiaram ao longo da minha vida e, especialmente, desses últimos três anos.

À minha família, que sempre acreditou em mim. Obrigada por todo o apoio, o carinho e por todo o investimento em mim e em minha educação. Agradeço especialmente, à minha mãe, que sempre foi um exemplo de pessoa para mim, e ao meu pai, que foi o principal responsável pela minha curiosidade pela computação. Obrigada também por todas as visitas durante o ano e meio em que estive fora, vocês não imaginam o quanto foram importantes. Amo muito vocês.

Ao Marcelo, por ter me acompanhado, de longe e de perto, desde o início da faculdade. Obrigada pelo apoio nos tempos difíceis (e mau-humorados) e nos meus momentos (comuns) de indecisão. Por todas as vezes que teve que acordar de madrugada para conversar no Skype e por ter batalhado tanto para conseguir ficar comigo em Irvine. A sua presença foi, sem dúvida, fundamental para que eu conseguisse fazer tudo o que eu precisava e ainda conseguir me divertir. Você continua sendo minha companhia favorita.

À minha orientadora Cláudia Werner, por ter me aceito no seu grupo de pesquisa (mesmo eu tendo chegado atrasada para a entrevista) e por ter me dado tantas oportunidades de aprendizado, desde o início da minha Iniciação Científica. Obrigada por ter confiado em mim para fazer um raro "mestrado sanduíche" e por ter ainda arranjado um tempinho para me visitar. Não posso deixar de agradecer também pela paciência, por todas as revisões de texto (sempre com prazo bastante apertado) e pelas palavras de conforto quando nem tudo saía como planejado. Obrigada por todas as intervenções que foram necessárias para garantir que esse trabalho realmente chegasse ao fim, com a qualidade desejada.

Ao meu co-orientador André van der Hoek, por ter me dado a inestimável oportunidade de trabalhar na UCI sob sua supervisão. Obrigada por permitir que eu "herdasse" o Lighthouse, um projeto tão interessante, e por ter confiado em mim para levar o projeto adiante. Agradeço por todo o tempo investido no meu aprendizado, as conversas, as reuniões no quadro branco, as revisões detalhadas, as prévias, os *demos*. Obrigada por permitir e até me encorajar a aproveitar a minha estadia também para o lazer, pelos convites a eventos sociais e pelos dias de folga quando chegavam visitas ou quando surgiam oportunidades de viagem. Por último, mas não menos importante, obrigada pela preciosa oportunidade de ter o Marcelo em Irvine.

Aos membros da banca, por terem aceitado avaliar este trabalho, pelas valiosas sugestões e importantes questionamentos levantados.

Aos meus colegas do LENS, por me proporcionarem um agradável ambiente de trabalho e, ao mesmo tempo, um rico ambiente de aprendizagem. Gostaria de agradecer especialmente ao Leo Murta, por todo o apoio dado desde a minha chegada ao LENS. Por ter me ensinado sobre controle de acesso no Linux, por ter "aberto o caminho" e me apoiado na ida à Irvine, pelas duas ótimas visitas e pelos conselhos sobre meu trabalho. Agradeço também ao Marco Mangan, por ter me apresentado à área de CSCW e a muitos aspectos do trabalho de pesquisa em geral, esse conhecimento me serviu como base para o mestrado. Quero agradecer também aos amigos "veteranos" Natanael, Regiane, Beto, Lopes, Dantas, Ana Paula, Aline, Luís Gustavo, Artur e Rafael. Obrigada também aos amigos mais recentes João Gustavo, Paula, Cláudia Susie, Rodrigo, Anderson, Danny e Chessman. Muito obrigada a todos que se voluntariaram para participar dos estudos executados para este trabalho, pela paciência em relação aos muitos adiamentos e dificuldades e pelos valiosos comentários e sugestões dados.

Aos meus colegas de Irvine, por terem tornado a minha estadia muito mais agradável. Aos brasileiros que me acolheram: Leila, obrigada por ter me abrigado em diversas ocasiões (tanto em Irvine quanto em San Matteo), pela companhia dentro e fora do campus, pelas noites de pizza, cerveja e filmes; André e Mirella, por terem me recebido assim que eu cheguei, pela companhia em viagens, passeios e comemorações, além das inúmeras caronas para a *Fry's*; e, finalmente, Roberto, pela companhia no *office*, nas reuniões e eventos, além, também, das várias caronas. Também gostaria de agradecer aos amigos "internacionais": Anita, obrigada por toda ajuda acadêmica e pessoal durante aquele ano e meio, pelas caminhadas no campus, pelas longas e pacientes conversas no *office*, pelos artigos escritos e revisados, pelo planejamento do estudo, pelo abrigo no final da minha estadia e, o mais importante, pela torcida para que tudo desse certo; Gabi e Sam, obrigada pela companhia em todos os programas, desde montanhas russas a video-games, e obrigada por cuidarem tão bem da Martini; Chad, por me aceitar (e depois o Marcelo) como *roomate*, por ter a cadelinha mais linda e educada do mundo, por todas as caronas e por toda a ajuda que você deu. Aos companheiros da UCI Eugen, Scott, Emily, Gerald, Ping & Chris, Alex, Roger, Nick, Chris Jensen, Steve & Eric, Yuzo, Ban, Suzanne, Jungmin e Paul (obrigada por cuidar do Lighthouse).

A todos os meus amigos, pela paciência e compreensão durante meus períodos "anti-sociais" necessários para a finalização deste trabalho. A meus amigos do colégio que sempre torceram por mim: Ana, Adriana e Kito. Obrigada a todos os amigos da faculdade, principalmente Carol, Bernardo, João, Iron, Pedro (valeu pela visita!). Obrigado a esses últimos dois também pelas longas sessões de CoH, que tanto divertiam as noites de Irvine. Agradeço também a meus compadres Rodrigo e Júlia e seus familiares, por terem sempre lembrado meu afilhado de que eu ainda existia.

Ao CNPQ, pelo apoio financeiro provido desde a minha Iniciação Científica.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

# LIGHTHOUSE: UM MECANISMO DE PERCEPÇÃO DE GRUPO BASEADO NO DESIGN EMERGENTE

Isabella Almeida da Silva

Abril / 2008

Orientadores: Cláudia Maria Lima Werner
André van der Hoek

Programa: Engenharia de Sistemas e Computação

A coordenação das atividades dos membros de uma equipe de desenvolvedores de *software* é uma atividade complexa. O suporte ferramental existente apresenta problemas comuns ligados à grande quantidade de informação com as quais essas ferramentas precisam lidar. Este trabalho propõe uma abordagem para o desenvolvimento de um mecanismo de percepção de grupo, denominado Lighthouse, que apresenta potenciais soluções para lidar com esses problemas. O mecanismo se baseia numa abstração dinâmica, denominada *Design* Emergente. Esta abstração reflete a estrutura do código-fonte do sistema em desenvolvimento e permite que a equipe acompanhe a evolução da implementação "ao vivo". O diagrama de *Design* Emergente pode ser ainda enriquecido com informações de percepção adicionais, para facilitar o entendimento sobre "o que está acontecendo" no projeto, o que é fundamental para o sucesso da equipe. Um protótipo do mecanismo foi desenvolvido e dele foi feita uma avaliação preliminar, executada através de um estudo de observação.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

LIGHTHOUSE: A GROUP AWARENESS MECHANISM BASED ON EMERGING DESIGN

Isabella Almeida da Silva

April / 2008

Advisors: Cláudia Maria Lima Werner
André van der Hoek

Department: Computer and Systems Engineering

The coordination of a software development team is a complex task. Existing tool support present common problems, mainly due to the large amount of information available. This work proposes an approach for the development of a group awareness mechanism, named Lighthouse, that presents potential solutions to these problems. The mechanism is based on a dynamic abstraction, called Emerging Design. This abstraction reflects the system's source code and allows the team to watch the implementation evolving "live". The Emerging Design diagram can also be annotated with additional awareness information, in order to help developers in understanding "what is going on" in the project, which is crucial for the team to succeed. A prototype of the mechanism was built and an observation study was performed as a preliminary evaluation.

# Índice

# Índice de Figuras

# Índice de Tabelas

# 1. Introdução

## 1.1. Preâmbulo

O desenvolvimento de sistemas de *software* é uma atividade colaborativa. Uma ou mais equipes de desenvolvedores precisam coordenar seus esforços para construir um sistema. Na prática, observa-se que os desenvolvedores de *software* passam a maior parte do tempo isolados, trabalhando paralelamente em suas tarefas individuais. Periodicamente, as contribuições individuais de cada desenvolvedor são combinadas, e uma nova versão do *software* é gerada. No entanto, as tarefas dos desenvolvedores muitas vezes não são independentes umas das outras. As modificações feitas em um determinado componente do sistema podem impactar componentes relacionados, gerando conflitos no código-fonte.

Os desenvolvedores devem estar cientes dos possíveis impactos que a sua modificação no sistema pode causar, para evitar que ocorram estes conflitos. Além disso, os desenvolvedores também devem procurar saber sobre as modificações inseridas pelos demais membros da equipe, para se assegurar que estas não geram conflitos com a sua tarefa. Em resumo, os desenvolvedores precisam estar cientes sobre "o que está acontecendo" no projeto, para evitar conflitos.

Quanto mais complexo for o sistema, mais difícil se torna detectar os possíveis impactos. Além disso, quanto maior for a quantidade de trabalho concorrente, mais complexa se torna a tarefa de coordenar os desenvolvedores durante a combinação do resultado de suas tarefas individuais.

Sistemas de Controle de Versão (SCVs) são comumente utilizados para auxiliar nesta coordenação dos membros de equipes de desenvolvimento de *software* (GRINTER, 1995). Esses sistemas suportam, entre outras funcionalidades, o compartilhamento dos artefatos de *software* pela equipe a partir de um repositório central; o controle de acesso e modificação dos desenvolvedores a determinados artefatos; a comparação e a combinação de diferentes versões de um mesmo artefato. Com esse suporte ferramental, os desenvolvedores podem trabalhar em paralelo usando cópias locais dos artefatos, não afetando os demais. Somente quando um desenvolvedor termina uma tarefa é que, geralmente, seu código é integrado ao repositório central e compartilhado com o restante da equipe. A freqüência com que o código dos desenvolvedores é integrado ao repositório varia de acordo com o tamanho de cada tarefa, com o processo adotado pela equipe, ou com a experiência dos desenvolvedores do projeto [SOUZA et al., 2003] .

## 1.2. Motivação

Esse isolamento do *trabalho* dos desenvolvedores evita conflitos imediatos no código, que tornariam o desenvolvimento paralelo impraticável. No entanto, ele também incentiva o isolamento das *pessoas*. Se os desenvolvedores ficam isolados do restante da equipe, eles só encontram potenciais conflitos na hora de integrar o seu código, o que pode demorar semanas (SOUZA et al., 2003). A solução destes conflitos geralmente demanda algum tipo de trabalho adicional ou re-trabalho. Esses conflitos muitas vezes poderiam ser resolvidos com menor custo se fossem detectados mais cedo (SARMA e HOEK, 2004).

Ao trabalhar em equipe, cada indivíduo precisa entender o contexto de sua tarefa dentro do projeto como um todo e estar ciente do impacto que suas contribuições podem ter nas tarefas dos demais. Esse contexto é fundamental para que a colaboração seja bem sucedida (DOURISH e BELLOTTI, 1992). Porém, o entendimento deste contexto pode ser oneroso, pois as informações sobre o projeto e as atividades da equipe são abundantes, e nem sempre estão facilmente acessíveis.

Por exemplo, dois desenvolvedores podem passar um bom tempo trabalhando em um mesmo artefato ou em artefatos relacionados, fazendo mudanças conflitantes, sem saber. Se esses desenvolvedores fossem avisados automaticamente sobre esses potenciais conflitos que estão criando, eles poderiam coordenar melhor os seus esforços, evitando problemas. Outro cenário interessante, onde o entendimento sobre a equipe é importante, é a chegada de um novo membro ao projeto. Ainda inexperiente, o novo membro nem sempre sabe a quem contactar em caso de dúvidas sobre uma determinada parte do sistema. Apesar dessa informação estar geralmente disponível de alguma forma (geralmente através do SCV), a sua consulta nem sempre é simples. Cria-se, portanto, a oportunidade de se desenvolver um suporte ferramental para facilitar o acesso às informações sobre o projeto de *software* e sobre a equipe, com intuito de promover o entendimento do contexto de trabalho dos seus membros.

## 1.3. Problema

Existem ferramentas que utilizam informações extraídas do próprio SCVs com o objetivo de aumentar a percepção dos desenvolvedores em relação aos demais (SILVA, 2005). São utilizadas informações sobre a disponibilidade de novas versões de artefatos para que todos possam atualizar suas cópias locais; sobre a autoria dos artefatos para detectar especialistas em determinadas áreas do sistema; análises estatísticas para definir quais são os artefatos que são normalmente modificados em conjunto em uma integração, entre outros. Essas informações são coletadas automaticamente do SCVs e depois são disponibilizadas para a equipe por meio de consultas, visualizações ou alertas no próprio ambiente de programação.

No entanto, essa abordagem é dependente da freqüência com que os desenvolvedores integram seu código, uma vez que essas informações só estão disponíveis nos SCVs quando o desenvolvedor está pronto para compartilhar uma nova versão dos artefatos com os demais. Com isso, as informações disponíveis não necessariamente refletem o contexto mais atual da equipe.

Para superar essa limitação, foram desenvolvidas ferramentas que extraem essas informações diretamente das estações de trabalho dos desenvolvedores para substituir ou complementar as providas pelo SCVs (SARMA et al., 2003; SCHÜMMER, 2001; HUPFER et al., 2004). Desta forma, é possível coletar o mesmo tipo de informação, de forma mais freqüente e sem necessariamente depender de ações específicas dos desenvolvedores. Com essa abordagem, os desenvolvedores têm acesso a informações sempre atualizadas sobre a equipe e o estado do projeto.

No entanto, essa abordagem, por sua vez, também apresenta problemas. Como os desenvolvedores estão constantemente fazendo modificações em diferentes artefatos, a quantidade de informações extraídas de cada estação de trabalho é muito grande, tornando mais difícil a coleta, consulta e visualização das informações. As ferramentas que utilizam essa abordagem, geralmente, enfrentam problemas de escalabilidade, as consultas e visualizações se tornam complexas e alertas constantes no ambiente de desenvolvimento se tornam distrações no trabalho dos desenvolvedores.

## 1.4. Objetivo

Tendo em vista as necessidades dos desenvolvedores e os problemas com o suporte ferramental existente citados anteriormente, o objetivo desse trabalho é prover um mecanismo que auxilie os desenvolvedores a entender o contexto atual da sua equipe, para facilitar a coordenação de suas tarefas com as dos demais. Esse mecanismo deve, no entanto, procurar representar as informações obtidas de forma escalável, e evitar interrupções às tarefas principais dos desenvolvedores, assim como a sobrecarga de informações sobre estes.

Para alcançar esse objetivo, é necessário alcançar as seguintes metas intermediárias: (1) identificar o contexto atual de cada desenvolvedor; (2) analisar as informações disponíveis extraídas das estações de trabalho de toda a equipe; (3) selecionar as informações que são atualmente relevantes para cada desenvolvedor; (4) representar as informações de maneira clara, escalável e não-intrusiva.

## 1.5. Contexto

Este trabalho de pesquisa foi desenvolvido através de uma parceria entre a COPPE-UFRJ e a Universidade da Califórnia, Irvine (UCI). O projeto de pesquisa foi

iniciado na Universidade da Califórnia, Irvine (UCI), no ano de 2005, pelo professor André van der Hoek e dois de seus alunos de doutorado, Ping Chen e Christopher van der Westhuizen. Este projeto foi concebido a partir da experiência anterior do professor van der Hoek com mecanismos de percepção, no projeto Palantír (SARMA et al., 2003). Em junho de 2006, os alunos que originalmente participaram do projeto deixaram a universidade. Na mesma época, a autora desta dissertação foi participar de um estágio de mestrado na UCI, sob supervisão do professor van der Hoek. Devido ao interesse pelo projeto, a UCI passou a desenvolver o projeto em colaboração com a COPPE/UFRJ.

## 1.6. Organização

O restante desta dissertação está organizada da seguinte maneira:

No **Capítulo 2**, é apresentada uma breve revisão da literatura relacionada à colaboração no processo de desenvolvimento de software, apresentando os conceitos que serão utilizados ao longo deste trabalho. São também descritas as algumas abordagens existentes para apoiar as atividades de colaboração.

No **Capítulo 3,** é introduzida a abordagem proposta nesse trabalho, baseada nos problemas identificados na literatura e nas abordagens existentes atualmente. São discutidas as principais características e funcionalidade propostas para superar estes problemas.

No **Capítulo 4,** são discutidos os detalhes de implementação da abordagem num protótipo de mecanismo de percepção.

No **Capítulo 5,** é apresentada uma avaliação inicial da abordagem proposta, que consiste em um estudo de observação com estudantes de pós-graduação utilizando a abordagem apresentada.

No **Capítulo 6**, finalmente, são apresentadas as conclusões deste trabalho, incluindo suas contribuições, limitações e listando os possíveis trabalhos futuros.

# 2. Colaboração e Coordenação

## 2.1. Introdução

O desenvolvimento de sistemas de *software* é uma atividade colaborativa. Uma ou mais equipes de desenvolvedores têm que se coordenar para construir e evoluir um sistema de *software*. Com a crescente complexidade dos sistemas e o tamanho das equipes encontrados atualmente, é cada vez mais comum o uso de processos e ferramentas que apóiem a coordenação dessas equipes (CATALDO et al., 2006). A distribuição geográfica das equipes, prática cada vez mais adotada na indústria, dificulta ainda mais essa tarefa (GRINTER et al., 1999).

Existem duas abordagens principais para a coordenação de equipes de desenvolvimento de *software* (tanto distribuídas, quanto localizadas). A primeira, formal, se baseia em processos bem definidos onde o trabalho é dividido em tarefas independentes que são sincronizadas em determinados pontos. A disciplina de Gerência de Configuração de *Software* (GCS) é bastante utilizada neste tipo de abordagem. Já a segunda abordagem, informal, promove a comunicação e a percepção de grupo. Essa abordagem é, geralmente, associada à área de Trabalho Cooperativo Apoiado por Computador (do inglês *Computer Supported Cooperative Work* - CSCW). Na prática, essas abordagens se complementam e são comumente combinadas (SOUZA et al., 2003; GUTWIN e GREENBERG, 2002; PICKERING e GRINTER, 1994). Recentemente, as ferramentas de apoio à coordenação também passaram a refletir essa combinação para melhor apoiar as equipes.

Nesse capítulo, discutiremos em mais detalhes essas diferentes abordagens relacionadas à coordenação no processo de desenvolvimento de *software* (Seção 2.2), apresentaremos a disciplina de Gerência de Configuração de *Software* (Seção 2.3) e os Mecanismos de Percepção de Grupo (Seção 2.4). Em seguida, descrevemos o suporte ferramental existente para apoiar a colaboração em equipes de desenvolvimento (Seção 2.5) e apresentaremos nossas conclusões (Seção 2.6).

## 2.2. Coordenação do Desenvolvimento de Software

O termo *coordenação* pode ser definido como "o ato de gerenciar interdependências entre atividades executadas para atingir um objetivo" (MALONE e CROWSTON, 1990). Em equipes de desenvolvimento de *software*, o objetivo final da coordenação é a construção dos artefatos de *software* em si. Atividades comuns nesse contexto são: implementação de novas funcionalidades no sistema, conserto de defeitos, escrita de documentação, desenvolvimento e execução testes, etc.

Exemplos de interdependências entre atividades são (MALONE e CROWSTON, 1990): recursos compartilhados (duas atividades necessitam de um mesmo recurso); relações produtor/consumidor (uma tarefa depende do resultado de outra) e limitações de simultaneidade (a necessidade ou impossibilidade de duas tarefas serem feitas ao mesmo tempo). No contexto de desenvolvimento de *software*, no entanto, muitas dessas dependências ganham um sentido diferente por não lidar tanto com recursos físicos e, sim, virtuais. Um recurso físico é alocado a uma atividade por vez. Por exemplo, uma máquina que é necessária para a instalação de duas peças mecânicas distintas durante a montagem de carros. No entanto, artefatos virtuais podem ser facilmente copiados, compartilhados e alterados. Essas propriedades permitem que um artefato seja utilizado simultaneamente por mais de uma pessoa, permitindo a paralelização de muitas atividades. No entanto, essa paralelização pode gerar mudanças conflitantes (BERLINER, 1990), que depois têm que ser resolvidas.

Apesar do crescente suporte ferramental para apoiar a coordenação de equipes de desenvolvimento de *software*, essa ainda é uma tarefa difícil, principalmente no contexto de projetos de grande porte (ESPINOSA, 2002). Problemas de coordenação podem gerar atrasos no projeto, além de piorar a qualidade e aumentar o custo do produto (CATALDO et al., 2006).

### *2.2.1. Coordenação de equipes distribuídas*

A coordenação do desenvolvimento de *software* se torna ainda mais complexa quando os participantes da equipe se encontram distribuídos geograficamente. Diferentes estudos mostram como a distância entre as equipes torna a comunicação e a coordenação mais difíceis (GRINTER et al., 1999; HERBSLEB et al., 2001; KRAUT e STREETER, 1995; OLSON e OLSON, 2000). Os integrantes de equipes localizadas desenvolvem naturalmente um conhecimento sobre os colegas que facilita a coordenação das atividades da equipe. Ao compartilhar o local de trabalho, se torna mais fácil saber, por exemplo, os horários normais de trabalho, as áreas de especialidade, as atividades atuais e passadas de cada um dos integrantes da equipe. Esse tipo de conhecimento não é tão facilmente desenvolvido por equipes distribuídas.

Apesar das dificuldades reportadas por estas equipes, é cada vez mais comum essa distribuição. Atraídas pelo menor custo de salários e impostos, muitas empresas estão contratando equipes em outros países para o desenvolvimento de *software*, um processo denominado Desenvolvimento Global de *Software* (DGS) (HERBSLEB e MOITRA, 2001). Além do custo, outro atrativo é a possibilidade de se usar a diferença de fusos horários para acelerar o desenvolvimento, usando turnos alternados de trabalho com equipes no leste e no oeste do globo. No entanto, de acordo com CARMEL (1999), 58% das companias que usam essa abordagem de desenvolvimento

24 horas acabam observando tempos de desenvolvimento iguais ou até maiores comparados ao uso de equipes tradicionais. Um dos problemas mais comuns apontados por equipes globalizadas é a dificuldade de lidar com diferenças culturais e de idioma (HERBSLEB e MOITRA, 2001; CARMEL, 1999; HEEKS et al., 2001; KRISHNA et al., 2004) entre as diferentes locações.

### 2.2.2. Abordagem formal

Na literatura, existem basicamente dois tipos de abordagens para o apoio à coordenação de equipes de desenvolvimento de *software*: as formais e as informais. A seguir detalhamos cada uma delas. As abordagens formais são aquelas que apresentam um processo definido de coordenação, dividindo o trabalho em atividades independentes que são sincronizadas em pontos específicos (HOEK et al., 2004).

Os Sistemas de Gerência de Configuração de *Software* (SGCS) são normalmente utilizados neste tipo de abordagem, pois apresentam um processo completo para controlar e coordenar as mudanças nos artefatos de *software* durante todo o ciclo de desenvolvimento. Esses sistemas permitem, por exemplo, que os desenvolvedores modifiquem cópias locais dos artefatos de *software* e sincronizem o seu trabalho com os dos demais ao final de cada tarefa. Uma discussão mais detalhada destes sistemas será feita em seguida.

Uma das principais vantagens da abordagem formal é prover uma solução escalável para a coordenação, suportando equipes e projetos de diferentes tamanhos. Essa abordagem também torna controlável a coordenação da equipe, através de um processo bem definido. Além disso, com essa abordagem, as atividades de cada indivíduo são executadas paralelamente de forma independente, evitando assim conflitos imediatos que poderiam atrasar ou até impedir a execução das atividades.

Enquanto esse isolamento do trabalho é interessante por evitar que haja interferência entre as atividades executadas em paralelo, essa abordagem também incentiva o isolamento dos integrantes da equipe. Somente ao sincronizar sua tarefa com as dos demais é que os conflitos podem ser detectados. Essa sincronização pode demorar semanas para acontecer (SOUZA et al., 2003), o que geralmente torna os conflitos mais difíceis de serem resolvidos (SARMA e HOEK, 2004) e implica em re-trabalho.

### 2.2.3. Abordagem informal

A abordagem informal, mais explorada pela área de Trabalho Cooperativo Apoiado por Computador (do inglês *Computer Supported Cooperative Work* - CSCW), foca no aspecto social da coordenação de equipes, através da facilitação da comunicação e da troca de informações entre os integrantes de uma equipe. Essa abordagem se baseia no conceito de *percepção de grupo* (do inglês, *group*

*awareness*), que é definida como "o entendimento das atividades dos demais que provê um contexto para a sua própria atividade" (DOURISH e BELLOTTI, 1992). Esta percepção é útil para coordenar ações, gerenciar acoplamento, discutir tarefas, antecipar ações e encontrar ajuda (GUTWIN e GREENBERG, 2002).

Exemplos comuns de sistemas que suportam essa abordagem são programas de correio eletrônico, mensageiros instantâneos, redes sociais e fóruns de discussão. Esses sistemas provêem um canal de comunicação contínuo para a equipe e incentivam o compartilhamento de informações entre seus integrantes. O correio eletrônico, por exemplo, é bastante utilizado para divulgar eventos, prazos e mudanças no projeto para toda a equipe. Mensageiros instantâneos, por sua vez, trazem informações sobre disponibilidade para comunicação e são utilizados para contactar indivíduos específicos para resolver problemas ou tirar dúvidas, por exemplo. Muitos desses sistemas atualmente provêem, além de mensagens de texto, ligações de áudio, vídeo e compartilhamento de programas, tornando a comunicação muito mais rica. Redes sociais e fóruns de discussão são geralmente utilizados como base de conhecimento, ajudando a identificar especialistas em determinadas áreas e a tirar dúvidas dos menos experientes.

A abordagem informal tem a vantagem de ser extremamente flexível, se adaptando facilmente a diferentes equipes e projetos. Essa abordagem também promove a sinergia da equipe, quebrando o isolamento entre seus integrantes, mesmo quando a equipe se encontra distribuída. Com esse maior compartilhamento de informações, se torna mais fácil desenvolver e manter a percepção de grupo.

Entretanto, essa abordagem também apresenta algumas desvantagens. A principal delas envolve problemas de escalabilidade. Quanto maior o projeto e maior o número de integrantes numa equipe, mais informações ficam disponíveis para serem compartilhadas. Essa quantidade acaba se tornando grande demais para ser absorvida por um indivíduo, que depois de um certo ponto passa a ignorar completamente as informações. Além disso, como essa abordagem é baseada diretamente no compartilhamento de informações entre indivíduos, a intervenção humana é indispensável, gerando constantes interrupções. Os sistemas descritos acima podem acabar gerando muitas distrações e desvios da tarefa principal dos participantes da equipe.

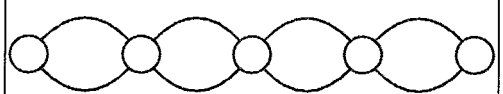
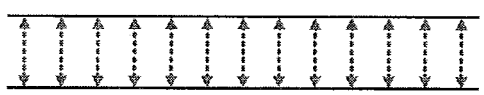
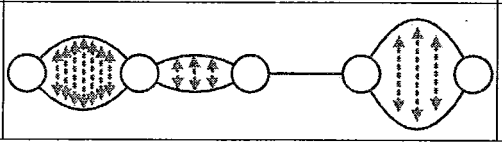
### *2.2.4. Coordenação Contínua*

As abordagens formais e informais por muito tempo foram vistas como opostas (SARMA e HOEK, 2004). No entanto, na prática, muitas vezes se observa a combinação dessas abordagens (SOUZA et al., 2003; GUTWIN e GREENBERG, 2002; PICKERING e GRINTER, 1994). Enquanto as organizações precisam controlar a coordenação do desenvolvimento de *software*, muitas vezes o processo formal

definido é envolvido por práticas informais, mais flexíveis. Por exemplo, SOUZA et al. (2003), descrevem como uma equipe de desenvolvimento de *software* da NASA utilizava o correio eletrônico para avisar quando o código fonte de um desenvolvedor ia ser integrado ao SGCS. A mensagem incluía uma explicação das mudanças que foram feitas e qual o impacto esperado delas no trabalho dos demais.

HOEK et al. (2004) utilizam o termo "coordenação contínua" para definir uma abordagem que combina e apóia explicitamente tanto a coordenação formal quanto a informal. A Tabela 2.1 mostra uma comparação entre qualidades e defeitos dessas três abordagens, assim como uma representação visual de como a coordenação é feita em cada uma delas. A abordagem formal contém as atividades independentes (linhas curvas) e seus pontos de sincronização (círculos). A abordagem informal é apresentada como a comunicação contínua (setas) durante as atividades paralelas (linhas retas). A coordenação contínua, por sua vez, é a combinação das representações anteriores, mantendo os pontos de sincronização e a comunicação contínua ao mesmo tempo.

**Tabela 2.1 - Comparação entre as abordagens de coordenação (HOEK et al., 2004).**

| Abordagem | Representação Visual | Qualidades | Defeitos |
|---|---|---|---|
| **Formal** (Baseada em processos) |  | Escalável; Controlável; Atividades independentes | Problemas de sincronização; Isolamento |
| **Informal** (Baseada em percepção) |  | Flexível; Promove sinergia; Desenvolve percepção | Não escalável; Intermediação humana |
| **Contínua** |  | Espera-se combinar as qualidades das duas abordagens acima | A serem descobertos |

## 2.3. Gerência de Configuração de Software

De acordo com GRINTER (1995), sistemas de *software* desenvolvidos em equipe são difíceis de se gerenciar por três motivos: (1) os desenvolvedores podem facilmente modificar o código fonte; (2) as modificações podem afetar o comportamento do sistema inteiro devido às dependências entre seus módulos internos; e (3) as mudanças feitas por um desenvolvedor podem afetar o trabalho dos demais.

A Gerência de Configuração de *Software* (GCS) é a disciplina que tem como principal responsabilidade controlar essas mudanças nos artefatos de *software* durante todo o processo de desenvolvimento (PRESSMAN, 2004). De acordo com o padrão definido pela IEEE (2005), a GCS inclui cinco funções:

1. *Identificação da Configuração*: consiste na seleção dos Itens de Configuração (IC) (os elementos passíveis de GCS), definição do esquema de nomes e números para identificar unicamente os ICS e a descrição física e funcional dos ICS.
2. *Controle de Configuração*: é o acompanhamento da evolução dos ICS, incluindo atividades como a solicitação de modificação, análise de impacto, implementação e verificação da modificação.
3. *Contabilização da Configuração*: descreve o armazenamento e o acesso a informações geradas pelas demais funções, através do uso de medições, que podem ser futuramente utilizadas para gerar relatórios, melhorar o processo e estimar custos.
4. *Avaliação e Revisão da Configuração*: tem o objetivo de assegurar a corretude e a completude da configuração antes de sua liberação.
5. *Gerenciamento de Liberação e Entrega*: consiste na construção (do inglês, *build*) do sistema a partir dos ICs fonte, identificação das versões dos ICs utilizados e implantação do sistema no ambiente de execução.

Estas funções são apoiadas por três sistemas principais (MURTA, 2006):

1. *Sistema de Controle de Modificações (SCM)*: tem como principal objetivo executar a função de Controle de Configuração, relatando as informações definidas pela função de Contabilização da Configuração.
2. *Sistema de Controle de Versões (SCV)*: é responsável pela Identificação da Configuração e pela evolução paralela e disciplinada dos ICs.
3. *Sistema de Gerenciamento de Construção (SGC)*: apóia as funções de Avaliação e Revisão da Configuração, assim como o Gerenciamento de Liberação e Entrega.

Como é freqüente a utilização de SCVs para apoiar a coordenação de desenvolvedores (SOUZA et al., 2003; GRINTER, 1996), este trabalho explora com mais detalhes esse sistema. Existem diversos SCVs disponíveis, tanto através de licenças de *software* livre (CVS (FREE SOFTWARE FOUNDATION, 2008a) e Subversion (COLLABNET, 2008a)), como por licenças comerciais (Visual Source Safe (MICROSOFT, 2008a) e ClearCase[1] (IBM RATIONAL, 2008a)). Esses sistemas são geralmente utilizados pelas equipes de desenvolvimento para gerenciar a evolução de artefatos como documentos e código fonte.

Para permitir o compartilhamento dos artefatos pela equipe, esses sistemas geralmente provêem um repositório central de armazenamento. O acesso a esse repositório é controlado e cada integrante da equipe é identificado unicamente ao

[1] O ClearCase também provê um SGC.

acessá-lo. Dessa forma, o sistema consegue mapear todas as ações para seus respectivos autores.

Ao acessar o repositório, um desenvolvedor pode fazer uma operação denominada *check-out*, que nada mais é do que baixar uma determinada versão dos artefatos do repositório para sua área local de trabalho (geralmente a versão mais atual). O desenvolvedor faz, então, as modificações necessárias nessa cópia local dos artefatos, sem afetar a cópia compartilhada com os demais. Ao terminar sua tarefa, o desenvolvedor faz a operação inversa, denominada *check-in*, onde sua versão modificada dos artefatos é mandada de volta para o repositório. A partir daí, os demais desenvolvedores passam a poder fazer *check-out* dessa versão modificada do repositório.

É importante notar, no entanto, que esse processo pode levar a conflitos se mais de um desenvolvedor fizer alterações paralelamente em um determinado artefato. Para evitar esses conflitos, as mudanças feitas devem ser adequadamente coordenadas. Do ponto de vista de coordenação, estes sistemas apresentam duas estratégias distintas para a modificação paralela de um artefato: a pessimista e a otimista. A estratégia pessimista simplesmente não permite que mais de um desenvolvedor faça *check-out* de um mesmo artefato. Assim que um desenvolvedor faz *check-out* do artefato, este fica marcado com uma trava (do inglês, *lock*) no repositório, impedindo que outros também o modifiquem. Somente quando o desenvolvedor original faz o *check-in*, é que o artefato é liberado para os demais.

A estratégia pessimista garante que não haverá conflitos diretos ou sobreposição de mudanças em um mesmo artefato, pois somente um desenvolvedor consegue modificar o artefato por vez. Se mais algum desenvolvedor precisar fazer modificações nesse mesmo artefato, ele terá que esperar o primeiro terminar sua tarefa e fazer o *check-in* do artefato de volta para o repositório. No entanto, conflitos indiretos, que impactam outros artefatos relacionados não são evitados por esta abordagem. Além disso, essa trava impede a paralelização das atividades que envolvem um mesmo artefato. Se situações como essa ocorrerem com muita freqüência, o processo de desenvolvimento pode acabar sofrendo atrasos.

A outra estratégia, denominada otimista, não impõe restrições de *check-out* de artefatos. Os desenvolvedores podem modificar quaisquer artefatos em suas cópias locais, simultaneamente, sem qualquer restrição. Porém, antes de fazer o *check-in* de uma nova versão, o desenvolvedor precisa verificar se a versão atual do repositório ainda corresponde à do *check-out* que ele fez antes de iniciar sua tarefa. Se algum outro desenvolvedor fez *check-in* nesse período, ele precisa fazer a junção (do inglês, *merge*) da sua versão com a do repositório. Somente após essa junção, o desenvolvedor faz o *check-in*.

Geralmente, os SCVs provêem algum suporte automatizado para a comparação e a junção de diferentes versões de um projeto, porém esse suporte se limita a fazer a junção de mudanças não-conflitantes (mudanças em artefatos diferentes ou partes diferentes de um mesmo artefato). No caso de conflitos, como, por exemplo, a alteração de uma mesma linha de código por dois desenvolvedores diferentes, a junção tem que ser feita manualmente. Só após completar a junção e se certificar de sua corretude, é que o desenvolvedor finalmente pode fazer o *check-in*. A maioria dos SCVs atualmente utilizam esta estratégia, por ser menos restritiva. Alguns ainda disponibilizam o mecanismo de trava opcionalmente.

Com o apoio desses sistemas, os desenvolvedores podem trabalhar em suas tarefas individuais, modificando os artefatos localmente sem impactos imediatos nas tarefas dos demais. Esse isolamento é benéfico por evitar conflitos diretos entre as modificações sendo feitas por cada desenvolvedor. Por isso, os SCVs (e SGCS em geral) são freqüentemente utilizados como ferramentas de suporte à coordenação pelas equipes de desenvolvimento (GRINTER, 1995). No entanto, esses sistemas têm como objetivo principal gerenciar os artefatos e não a dinâmica da equipe (GRINTER, 1995). O isolamento das tarefas individuais facilitado por estes sistemas pode acabar promovendo também o isolamento dos indivíduos. Como visto anteriormente, isso pode gerar problemas para a equipe, tais como re-trabalho e a detecção tardia de conflitos. Os sistemas do SGCS geralmente provêem algumas funcionalidades, mesmo que limitadas, para lidar com estes problemas, como, por exemplo, mecanismos de notificação de *check-ins* nos SCVs e informações sobre as atividades atuais dos membros das equipes nos SCMs.

## 2.4. Mecanismos de Percepção de Grupo

A percepção pode ser descrita simplesmente como o "entendimento do que está acontecendo" (SARMA e HOEK, 2004). Esse entendimento é fundamental para a eficiência da colaboração (DOURISH e BELLOTTI, 1992). Existem diferentes mecanismos utilizados pelos integrantes de uma equipe para manter a percepção de grupo. Equipes localizadas podem contar com a proximidade para se reunir formalmente ou informalmente, se comunicar diretamente ou perceber padrões de comportamento dos colegas. Essas interações permitem que haja naturalmente uma troca de *informações de percepção* dentro da equipe, ajudando a manter os participantes cientes "do que está acontecendo". No contexto de equipes de *software*, é comum, por exemplo, que se saiba quem é o especialista em um certo assunto técnico ou em um determinado módulo do sistema em desenvolvimento.

No entanto, com o aumento do tamanho da equipe ou do projeto de *software*, se torna cada vez mais difícil manter essa percepção natural. Equipes distribuídas não

desfrutam da convivência diária e, portanto, precisam encontrar outras formas de obter as informações de percepção. Nesse caso, se faz necessário algum outro tipo de mecanismo de percepção.

Muitas dessas informações de percepção estão disponíveis de alguma maneira através das ferramentas utilizadas pela equipe no dia-a-dia. Por exemplo, programas mensageiros indicam quando os integrantes da equipe estão disponíveis para comunicação e SCVs registram os autores de cada mudança no código-fonte. É possível, portanto, automatizar a coleta e a distribuição dessas informações. Com isso, a equipe pode focar nas suas tarefas principais e utilizar algum mecanismo automatizado de percepção para auxiliar a colaboração em si.

Diferentes tipos de mecanismos podem ser utilizados dependendo das informações que estão disponíveis e das necessidades da equipe. É possível classificar esses mecanismos por diferentes aspectos. Em seguida, discutiremos seis desses aspectos: tipo de informações, fonte de informações, análise das informações, representação das informações, local de apresentação das informações e público-alvo. Esses aspectos muitas vezes estão diretamente relacionados, como, por exemplo, o tipo de informações apresentadas pelo mecanismo varia de acordo com o público-alvo.

### *2.4.1. Tipos de Informação*

Existem diversos tipos de informações que os indivíduos utilizam para entender "o que está acontecendo" numa equipe de desenvolvimento. A informação de presença, por exemplo, indica se um indivíduo está no seu ambiente de trabalho e/ou disponível para comunicação. Essa informação pode ser utilizada, por exemplo, para determinar o melhor momento e meio de comunicação para se entrar em contato com uma determinada pessoa. A informação de atividade, por sua vez, descreve uma tarefa sendo executada pela equipe, podendo conter dados como os participantes alocados para a tarefa, a prioridade da tarefa, o progresso feito até o momento, prazos, etc. Esse tipo de informação pode ser usada para atualizar prazos e determinar alocações de tarefas futuras. Já a informação de autoria determina qual integrante da equipe é responsável por produzir um determinado artefato ou uma versão específica do artefato. Através do conhecimento da autoria é possível, entre outras coisas, encontrar potenciais especialistas em determinados módulos do sistema ou determinar o responsável pela introdução de um defeito.

### *2.4.2. Fonte de Informação*

Como visto anteriormente, as ferramentas que compõem o ambiente de trabalho de uma equipe de desenvolvimento são fontes importantes de informação de percepção. Mensageiros instantâneos geralmente provêem algum tipo de informação

de presença, SCMs contêm informações de atividade e SCVs, informações de autoria. Em todos esses exemplos, as informações estão disponíveis em um repositório central, para que seja possível compartilhá-las dentro da equipe. O mecanismo de percepção, nesse caso, faz a coleta desses dados, possivelmente agregando informações de diferentes fontes.

No entanto, existem dados que não são automaticamente compartilhados com a equipe, mas que também podem ser utilizados como informações de percepção. O mecanismo passa, então, a ser o principal responsável pelo compartilhamento dessas informações, tornando-se também uma fonte de informações. Exemplos desse caso serão explorados na Seção 2.5, quando discutirmos o suporte ferramental existente.

### *2.4.3. Análise da Informação*

Integrantes de uma equipe geralmente conseguem, pela convivência, conhecer o horário de chegada e saída de colegas no ambiente de trabalho e determinar qual(is) a(s) pessoa(s) mais experiente(s) em uma determinada área de conhecimento. De forma análoga, um mecanismo automatizado de percepção pode analisar os dados coletados, detectando padrões, relacionamentos e tendências. Como o volume de dados coletados por esses mecanismos é geralmente grande, essa análise também pode ser importante para filtrar ou agrupar as informações, facilitando a sua compreensão. Por exemplo, o histórico de modificações dos artefatos de *software* de um projeto cresce à medida que o mesmo evolui e, para facilitar a sua consulta, pode-se agrupar os dados por equipes ou prover filtros que permitam obter os dados de um determinado período de tempo.

### *2.4.4. Representação da Informação*

Existem diferentes maneiras de se representar as informações de percepção. Alguns mecanismos utilizam uma representação textual como notificações, relatórios e listas, enquanto outros provêem representações gráficas através de visualizações, gráficos e ícones, ou até representações sonoras, como alertas de eventos. É comum também a combinação de diferentes representações, como veremos na Seção 2.5.

Um aspecto importante da representação de informações de percepção é a sua escalabilidade. Com o aumento do volume de informações, se torna difícil para a equipe acompanhar mudanças numa lista muito longa de eventos ou entender um grafo com centenas de nós e arestas. É importante que a representação escolhida seja adaptada para a quantidade de dados esperada. Ao utilizar um grafo como representação, por exemplo, pode-se controlar a quantidade de arestas mostradas simultaneamente através do agrupamento ou expansão dos nós de acordo com o nível de abstração desejado. Já os que utilizam listas muito longas podem prover

mecanismos para ordenar ou buscar itens mais relevantes para um determinado indivíduo.

Outra questão importante em relação à representação de informações é o grau de interrupção causado pelos mecanismos de percepção. Enquanto alguns são passivos, somente mostrando as informações disponíveis, outros utilizam ativamente alertas e notificações para chamar a atenção dos integrantes da equipe para as suas informações. Os mecanismos ativos podem acabar se tornando uma distração constante no trabalho da equipe, o que pode levar seus integrantes a ignorá-los. Por outro lado, algumas informações importantes podem passar desapercebidas nos mecanismos puramente passivos.

A representação da informação ainda tem um papel importante na comunicação entre os integrantes da equipe. Esta pode ser utilizada como um modelo compartilhado por todos, provendo um referencial comum que facilita a comunicação (CURTIS et al., 1988; GRINTER, 1996). Isto é particularmente interessante nos casos de comunicação remota, mais comuns em equipes distribuídas.

### *2.4.5. Local de Apresentação da Informação*

A informação de percepção deve ser apresentada aos integrantes de uma equipe de maneira a facilitar a sua absorção, mas evitando interferir na atividade principal dos indivíduos. Para aumentar a visibilidade dessas informações, alguns mecanismos de percepção são integrados a ferramentas já utilizadas pela equipe, como o ambiente de desenvolvimento. Com isso, é possível ter as informações sempre disponíveis para consulta, sem a necessidade de se trocar o contexto para outras aplicações e aproveitando os recursos das ferramentas pré-existentes e com as quais a equipe já é familiar (HUPFER et al., 2004; MANGAN, 2006).

Outra possível abordagem é utilizar monitores secundários para apresentar as informações de percepção. Estudos recentes (GRUDIN, 2001; COLVIN et al., 2004) indicam as vantagens de se adicionar um ou dois monitores extras à mesa de trabalho, por evitar a troca constante de contexto entre diferentes aplicações. Através desses monitores adicionais, mecanismos de percepção podem apresentar suas informações de forma periférica, não desviando a atenção principal dos indivíduos. Outros trabalhos (BALL e NORTH, 2005; YOST et al., 2007) exploram o uso de múltiplos monitores grandes, de alta resolução, para apresentar grandes quantidades de informação simultaneamente. Mecanismos de percepção podem utilizar essa abordagem para resolver problemas de escalabilidade de suas representações.

### *2.4.6. Público-alvo*

Indivíduos desempenham diferentes papéis numa equipe de desenvolvimento de *software*: analistas, desenvolvedores, gerentes, engenheiros de teste e qualidade,

entre outros. Cada um desses papéis envolve atividades específicas que, por sua vez, necessitam de determinadas informações para serem executadas. O gerente, por exemplo, precisa acompanhar o andamento geral do projeto e, para isso, pode utilizar informações sobre o progresso das tarefas dos desenvolvedores. Os desenvolvedores, por sua vez, precisam obter informações mais específicas sobre o módulo do *software* com que estão trabalhando no momento, como os requisitos daquele módulo ou sobre outros desenvolvedores que já trabalharam nele. O público-alvo de um mecanismo de percepção pode influenciar diretamente outros aspectos citados anteriormente, incluindo o tipo de informação, como e onde ela é apresentada.

## 2.5. Ferramentas de Apoio à Colaboração

Diversas ferramentas já foram desenvolvidas com o objetivo de apoiar a colaboração de equipes de desenvolvimento de *software*. Nesta seção exploramos alguns desses trabalhos, focando nos que, de alguma forma, combinam abordagens formais e informais para a coordenação dessas equipes.

### *2.5.1. Odyssey-Share*

O Odyssey-Share (MANGAN, 2006; WERNER et al., 2003) é um projeto de suporte à colaboração desenvolvido no contexto do ambiente de suporte à reutilização Odyssey (BRAGA et al., 1999; ODYSSEY, 2008). O ambiente apóia os processos de Engenharia de Domínio (a construção de artefatos reutilizáveis para um domínio de conhecimento específico) e de Engenharia de Aplicação (o desenvolvimento de aplicações em um domínio a partir da reutilização dos artefatos existentes). O conhecimento de domínio é representado por extensões de modelos UML (OMG, 2008), que podem ser criados e reutilizados através de editores gráficos disponíveis no ambiente. O Odyssey é um ambiente extensível, que permite, através de um mecanismo de carga dinâmica (MURTA et al., 2004), a incorporação de novas ferramentas. Diversas ferramentas estão disponíveis através desse mecanismo: um sistema de críticas de consistência em modelos (DANTAS, 2001),um sistema de controle de versões para modelos (OLIVEIRA, 2005), um mecanismo de engenharia reversa (VERONESE e NETTO, 2001), entre outros.

O Odyssey-Share foi desenvolvido com o objetivo de prover um conjunto de ferramentas de colaboração ao ambiente Odyssey. Estas ferramentas apóiam a coordenação das atividades dos engenheiros de *software* relacionadas à construção e à manutenção dos modelos de domínio. Em seguida, detalharemos três dessas ferramentas, enfatizando o suporte provido à percepção de grupo.

### 2.5.1.1. Shared Workspaces

A primeira ferramenta, denominada Shared Workspaces (MANGAN, 2006), tem como objetivo apoiar a colaboração síncrona dos engenheiros de *software* no ambiente Odyssey. A ferramenta provê esse apoio através de *collablets*, que são abstrações de programação que têm como objetivo facilitar o desenvolvimento de mecanismos de colaboração de forma transparente (MANGAN, 2006). O Shared Workspaces contém três *collablets*: teleapontador, janela em miniatura e bate-papo (Figura 1). Os *collablets* transmitem via rede informações sobre o espaço de trabalho compartilhado e os indivíduos envolvidos na colaboração.

O teleapontador permite a visualização do cursor do *mouse* de dois indivíduos que estão colaborando remotamente através do ambiente Odyssey. Desta forma, os participantes ficam cientes das áreas do diagrama que seus colegas estão trabalhando naquele momento. A janela em miniatura, contém uma visão geral da área de trabalho compartilhada e também indica o posicionamento de cada indivíduo no diagrama. O *collablet* de bate-papo, por sua vez, permite a troca de mensagens instantâneas entre os participantes, além da indicação da informação de presença. As mensagens podem ser armazenadas como parte da base de conhecimento da equipe e são associadas aos artefatos utilizados no contexto da colaboração.

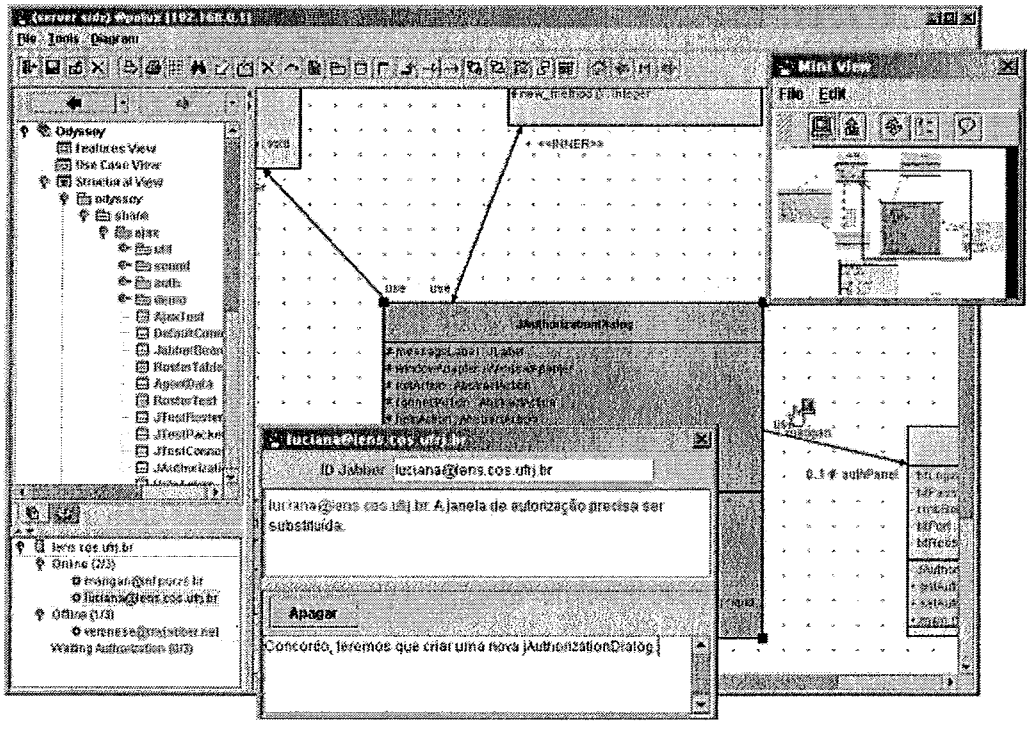



**Figura 2.1 - Ambiente Odyssey estendido com *collablets* do Shared Workspace: teleapontador, janela em miniatura e bate-papo (MANGAN, 2006).**

### 2.5.1.2. MAIS

A ferramenta MAIS (*Multi-synchronous, Awareness Infrastructure*, Infra-estrutura para Percepção Multi-síncrona) (LOPES et al., 2004; LOPES, 2005) tem como foco principal a notificação de mudanças em diagramas desenvolvidos por equipes de engenheiros de *software* no ambiente Odyssey. Informações sobre cada mudança feita (inclusão, edição ou remoção de um elemento) no diagrama são coletadas na estação de trabalho e compartilhadas com a equipe imediatamente. Desta forma, o MAIS provê um mecanismo de percepção sobre as atividades dos integrantes da equipe nos diagramas compartilhados.

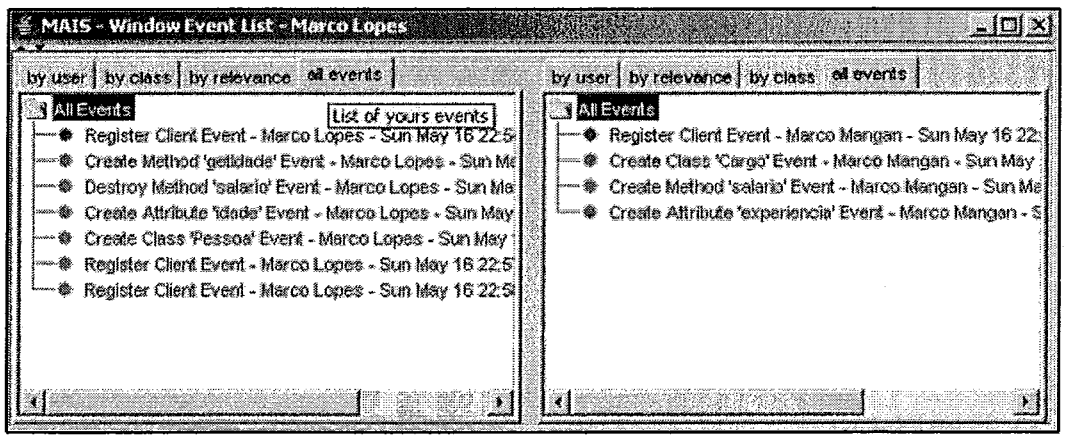



**Figura 2.2 - MAIS mostrando as informações de percepção geradas localmente (esquerda) e remotamente (direita) (LOPES et al., 2004).**

As informações de percepção são apresentadas na interface de usuário da ferramenta em dois painéis (Figura 2): o esquerdo lista informações sobre mudanças feitas localmente e o direito com mudanças feitas remotamente. Para lidar com a grande quantidade de informação disponível, é possível ordenar as informações pelo usuário, por elemento do diagrama, ou pela relevância da informação. A relevância é calculada através da "proximidade semântica" entre os elementos do diagrama modificados localmente e remotamente. Por exemplo, dois indivíduos modificando propriedades de um mesmo elemento num diagrama é algo considerado bastante relevante, enquanto a edição de dois diagramas diferentes não é relevante.

### 2.5.1.3. GAW

A ferramenta GAW (MANGAN et al., 2004; SILVA, 2005) explora uma outra forma de representação para as informações coletadas pelo MAIS. Na sua interface de usuário (Figura 2.3), todos os integrantes da equipe que colaboraram para a construção de um determinado diagrama são listados verticalmente. À direita de cada indivíduo, barras coloridas representam as mudanças feitas no diagrama. As barras são ordenadas em ordem cronológica inversa, ou seja, as mudanças recentes

aparecem mais à esquerda. As barras representando informações antigas se movem para a direita com o passar do tempo, dando lugar para as mais recentes. Informações mais detalhadas sobre a modificação (autor, data e nome do artefato afetado) são apresentadas como dicas ao se posicionar o cursor do mouse sobre as barras coloridas.

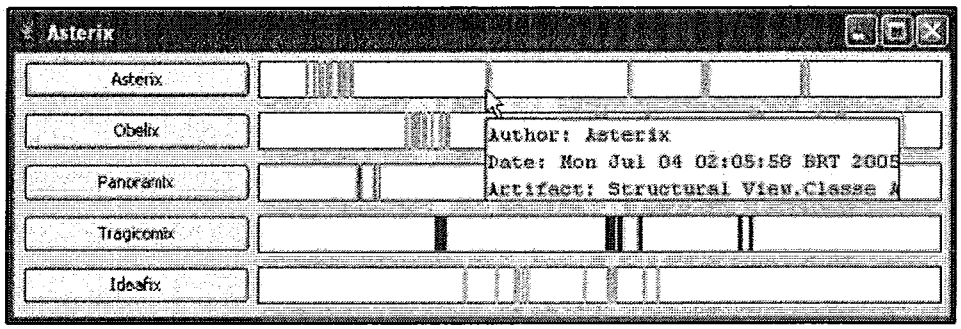



**Figura 2.3 - GAW apresentando informações coletadas pelo MAIS (DA SILVA, 2005).**

Através dessa visualização, os participantes da equipe podem analisar o histórico de modificações dos diagramas e identificar indivíduos que estão trabalhando ou já trabalharam num determinado diagrama. O GAW provê ainda diferentes filtros e escalas de tempo para facilitar a visualização de grandes volumes de informação. É possível, por exemplo, explorar as informações de um determinado intervalo de tempo, de um certo conjunto de autores ou artefatos separadamente.

### *2.5.2. Tukan*

Tukan (SCHÜMMER, 2001) é uma ferramenta de apoio à colaboração de desenvolvedores de *software*. A ferramenta é capaz de prover informações de percepção em diferentes níveis de detalhe, dependendo de quanto cada desenvolvedor esteja disposto a divulgar para os demais. O desenvolvedor pode escolher entre diferentes Modos de Colaboração (SCHÜMMER e HAAKE, 2001):

1. isolado - nenhuma informação é coletada;
2. nível de processo - informada somente a descrição da tarefa atual do desenvolvedor;
3. nível de mudanças - divulgadas as mudanças que o desenvolvedor está fazendo no código-fonte;
4. nível de presença - divulgados os artefatos em que o desenvolvedor está focado no momento (os que ele está visualizando e/ou modificando);
5. comunicação - o desenvolvedor está disponível para comunicação;
6. colaboração de alto acoplamento - o desenvolvedor está disponível para compartilhar sua área de trabalho para executar uma tarefa em conjunto com outros integrantes da equipe.

Ao ir passando de um modo de colaboração para o seguinte, o desenvolvedor vai disponibilizando mais informações sobre seu trabalho. A quantidade de informações que cada desenvolvedor escolhe mostrar é a mesma que ele recebe dos demais, ou seja, quanto mais um desenvolvedor deseja saber sobre um colega, mais terá que divulgar suas próprias informações.

O Tukan utiliza as informações coletadas para alertar desenvolvedores sobre possíveis conflitos ou sobre oportunidades de colaboração. Para isso, a ferramenta faz uma análise do código-fonte em desenvolvimento, identificando relacionamentos entre seus elementos (atributos ou métodos) e sobrepõe as informações dos desenvolvedores, avaliando o quanto as suas tarefas se sobrepõem. O resultado da análise é representado por diferentes decorações, dependendo do modo de colaboração dos desenvolvedores.

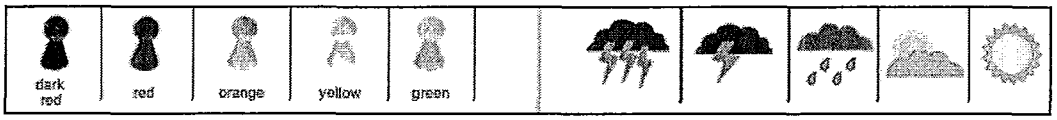



**Figura 2.4 - Ícones de decoração utilizados pelo Tukan : à esquerda, ícones de indicação de presença e à direita, ícones de indicação de conflitos (SCHÜMMER, 2001).**

No nível de mudança, o Tukan identifica os elementos de código que estão sendo modificados no momento para indicar a probabilidade de que haja um conflito. Essa indicação é feita através dos ícones de "clima", mostrados na Figura 2.4. Quanto mais relacionados os elementos modificados, pior o "clima" indicado, ou seja, maior a probabilidade de que haja um conflito. Quando desenvolvedores estão modificando exatamente o mesmo elemento, a probabilidade de conflitos é muito alta e, portanto, é mostrada uma nuvem com raios, enquanto elementos completamente independentes, são decorados por um sol, indicando que não há problemas.

Já no nível de presença, o Tukan analisa o foco do desenvolvedor, representado por todos os elementos de código que ele está visualizando em sua área de trabalho. Quanto mais próximos outros desenvolvedores estão de explorar um determinado elemento de código, mais vermelho fica o indicador de presença (Figura 2.4). Através desses ícones, é possível encontrar desenvolvedores interessados nos mesmos artefatos, identificando possíveis oportunidades de colaboração direta.

Esses ícones são integrados ao navegador de elementos de código provido pelo ambiente de desenvolvimento VisualWorks Smalltalk (CINCOM, 2008) em conjunto com o SCV Envy (previamente denominado Orwell (THOMAS e JOHNSON, 1988)). O resultado é mostrado na Figura 2.5. Além disso, o Tukan utiliza a biblioteca de programação de aplicações colaborativas COAST (SCHUCKMANN et al., 1996) para prover mecanismos de comunicação, percepção e compartilhamento de área de trabalho.

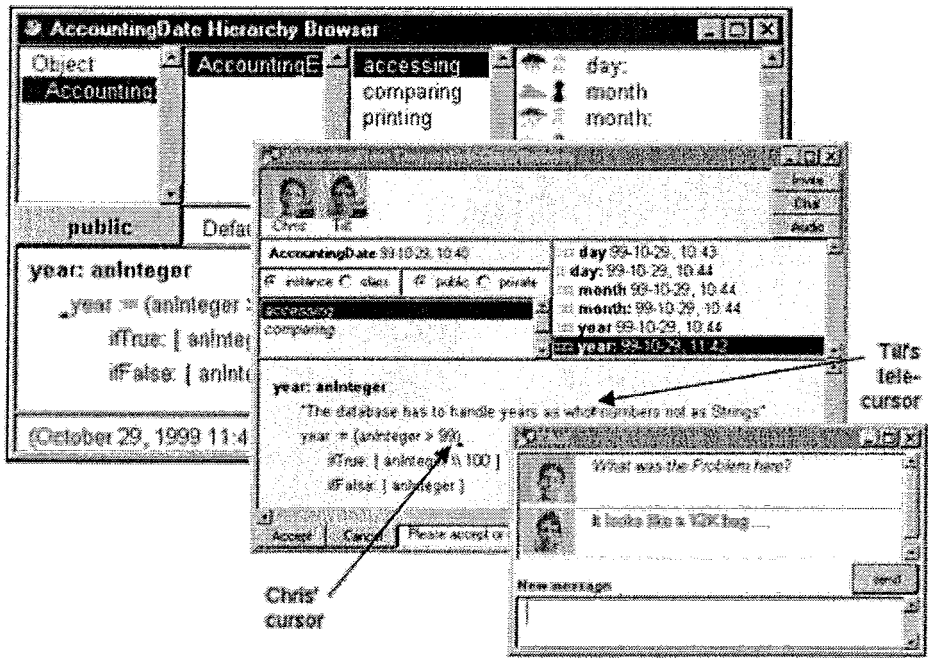



**Figura 2.5 - Ícones de percepção de conflito e presença (no fundo), janela de colaboração direta (no meio) e janela de mensagem de texto (à frente) do Tukan (SCHÜMMER, 2001).**

### *2.5.3. Palantír*

O Palantír (SARMA et al., 2003; SARMA et al., 2007a) é uma ferramenta que visa melhorar a percepção de grupo de equipes distribuídas que usam sistemas de GCS. A ferramenta provê informações sobre as modificações feitas nos artefatos compartilhados pela equipe, indicando a severidade e potencial impacto dessas mudanças. O Palantír contém um sistema de notificação de eventos que permite que mudanças feitas por cada desenvolvedor sejam automaticamente divulgadas para os demais. As informações sobre as mudanças podem ser coletadas de diferentes fontes, através de coletores especializados. Atualmente existem coletores para diferentes Sistemas de Controle de Versão (SCV) como CVS (FREE SOFTWARE FOUNDATION, 2008a), Subversion (COLLABNET, 2008a) e RCS (FREE SOFTWARE FOUNDATION, 2008b). Existe também um coletor para o ambiente de desenvolvimento Eclipse (THE ECLIPSE FOUNDATION, 2008a), que extrai informações diretamente do editor de código-fonte, enquanto as mudanças estão sendo implementadas e antes mesmo de serem compartilhadas através do repositório de artefatos. As informações coletadas podem ser acessadas por diferentes visualizações. A seguir, detalharemos cada uma das visualizações disponíveis.

### 2.5.3.1. Visualização Gráfica

A Visualização Gráfica do Palantír (Figura 2.6) apresenta uma visão hierárquica dos artefatos compartilhados. Cada artefato pode conter outros artefatos, pelos quais os usuários podem navegar, ampliando ou diminuindo o nível de detalhes apresentado. São também apresentadas as diferentes versões de um mesmo artefato que cada desenvolvedor tem na sua área de trabalho, representadas pelas pilhas de janelas. As janelas são coloridas de acordo com a área de trabalho em que a versão do artefato foi gerada e seu respectivo autor.

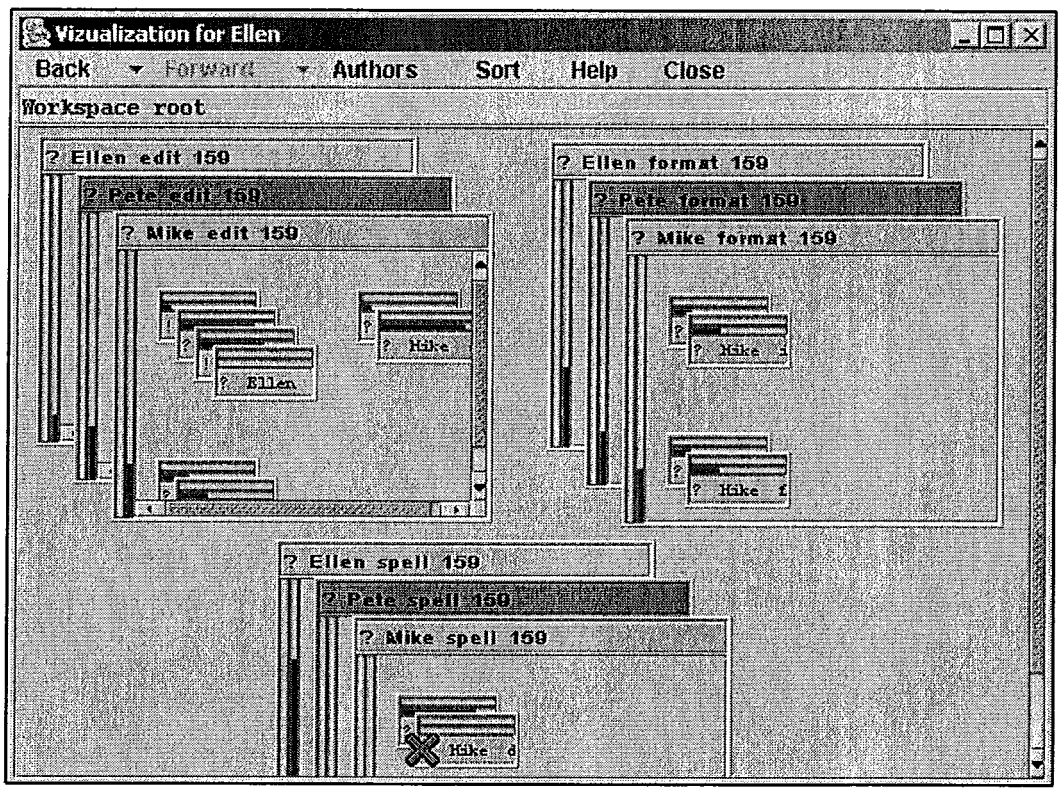



**Figura 2.6 - Visualização Gráfica do Palantír.**

É possível também avaliar a severidade das modificações que foram feitas nos artefatos, representada por uma barra azul em cada janela. Quanto maior o tamanho da barra, maior a mudança feita. Outros símbolos também são utilizados para denotar mudanças em progresso (ponto de interrogação), mudanças já compartilhadas através do repositório (ponto de exclamação), artefatos removidos (ícone vermelho em forma de "x") e recém-adicionados (ícones verdes em forma de "v", não mostrados na Figura 2.6). É possível ainda comparar lado-a-lado o conteúdo da área de trabalho de dois ou mais desenvolvedores, para encontrar possíveis conflitos (também não mostrado na figura).

### 2.5.3.2. *Explorer*

Esta visualização apresenta dois painéis (Figura 2.7) organizados de maneira similar aos gerenciadores de arquivos dos sistemas operacionais modernos. O painel da esquerda contém a estrutura hierárquica dos artefatos, enquanto o painel da direita apresenta os detalhes do artefato selecionado na esquerda. A estrutura é decorada com barras coloridas que indicam a severidade das mudanças feitas na área de trabalho local (verde) e nas áreas de trabalho de outros desenvolvedores (vermelho). O painel de detalhes mostra um histórico de mudanças do artefato, incluindo o nome dos autores, a severidade e o estado atual de cada mudança. É também possível ver a data em que as mudanças foram compartilhadas através do repositório.

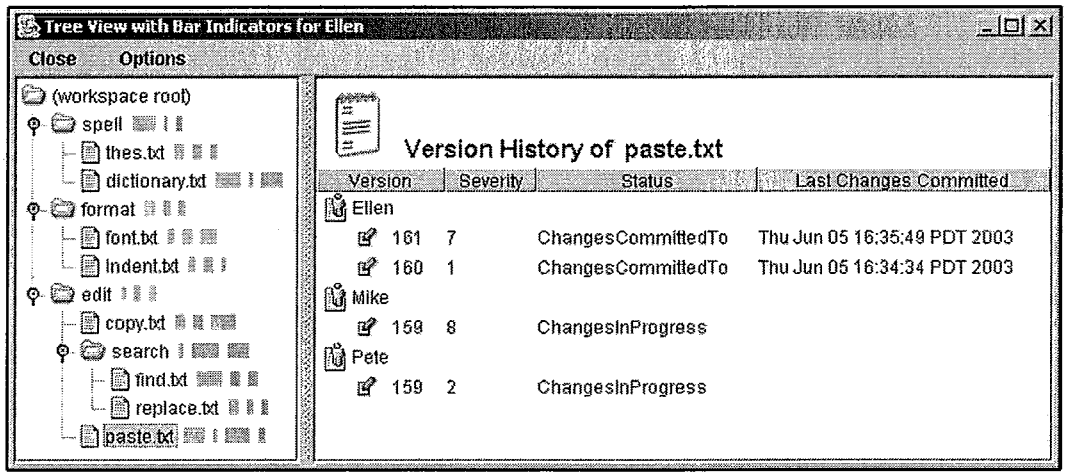



**Figura 2.7 - Visualização *Explorer* do Palantír.**

### 2.5.3.3. *Scrolling Marquee*

A mais simples das visualizações do Palantír, o *Scrolling Marquee* (Figura 2.8), funciona como um notificador de eventos, que informa sobre as mudanças feitas nos artefatos compartilhados pela equipe. Cada usuário pode filtrar os evento que são mostrados por tipo, autor, data, severidade e potencial impacto. Essa visualização foi desenvolvida para servir somente como um alerta dos eventos mais importantes. Caso seja de interesse do desenvolvedor, maiores detalhes devem ser explorados através das outras visualizações.

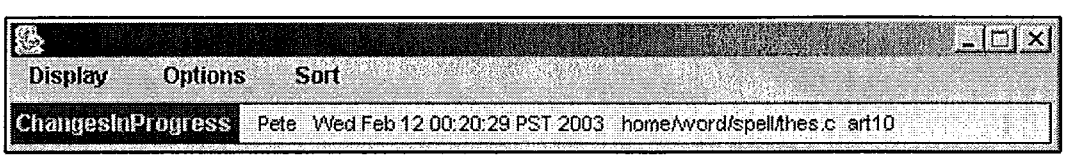



**Figura 2.8 - Scrolling Marquee do Palantír.**

### 2.5.3.4. Integração com Eclipse

As visualizações descritas anteriormente foram implementadas como aplicações independentes. No entanto, é possível também acessar as informações coletadas pelo Palantír através de *plug-ins* do ambiente de desenvolvimento Eclipse. As informações estão disponíveis de duas maneiras (Figura 2.9): através de decorações no painel esquerdo do ambiente (onde é mostrada a lista de artefatos do projeto de *software*) e no painel inferior, em uma tabela. No primeiro, pequenas decorações são adicionadas para indicar os artefatos que estão sendo modificados pelos demais desenvolvedores da equipe, indicando a severidade e o impacto das mudanças. O segundo painel mostra com mais detalhes as informações sobre o impacto das mudanças em cada artefato, incluindo o autor, o estado, a data e razão pela qual a ferramenta detectou um possível impacto.

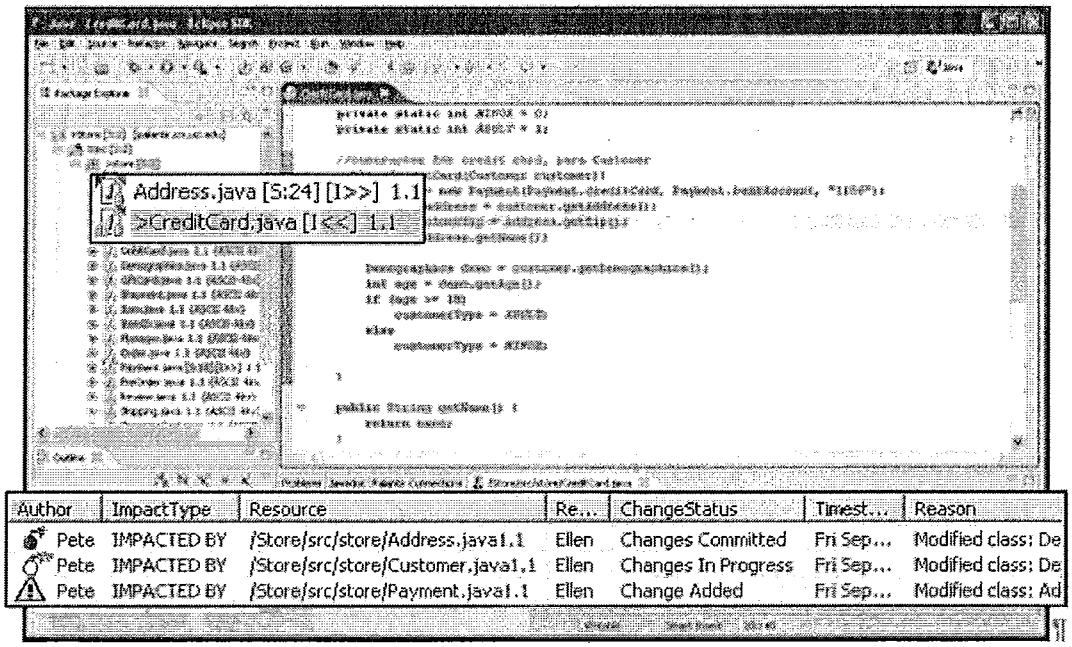



**Figura 2.9 - Integração do Palantír com o ambiente de desenvolvimento Eclipse. Em destaque: (a) na parte superior esquerda, decorações na lista de artefatos indicando severidade e impacto de mudanças feitas; e (b) na parte inferior, painel com detalhes sobre o impacto das mudanças.**

### 2.5.3.5. *Workspace Activity Viewer (WAV)*

Utilizando uma representação tridimensional, o *Workspace Activty Viewer* (RIPLEY et al., 2007), apresenta uma visão geral do histórico de mudanças do projeto (Figura 2.10). No modo de visualização de desenvolvedores, cada pilha de cilindros representa um único desenvolvedor, sendo cada cilindro empilhado um artefato por ele modificado. No modo de visualização de artefatos isso se inverte, as pilhas passam a representar os artefatos e cada cilindro, um desenvolvedor. Nos dois casos, a largura

dos cilindros representa a severidade das modificações. Quanto ao posicionamento dos cilindros, modificações recentes ficam à frente e vão sendo movidas para o fundo com o passar do tempo. É possível configurar as cores e a ordem dos cilindros de acordo com o tipo de artefato, severidade das mudanças, quantidade de modificações paralelas, entre outros parâmetros. Após a configuração da visualização, é possível mostrar o histórico do projeto como uma animação, onde as pilhas vão progressivamente mudando de tamanho e de posição, de acordo com as informações coletadas ao longo do tempo.

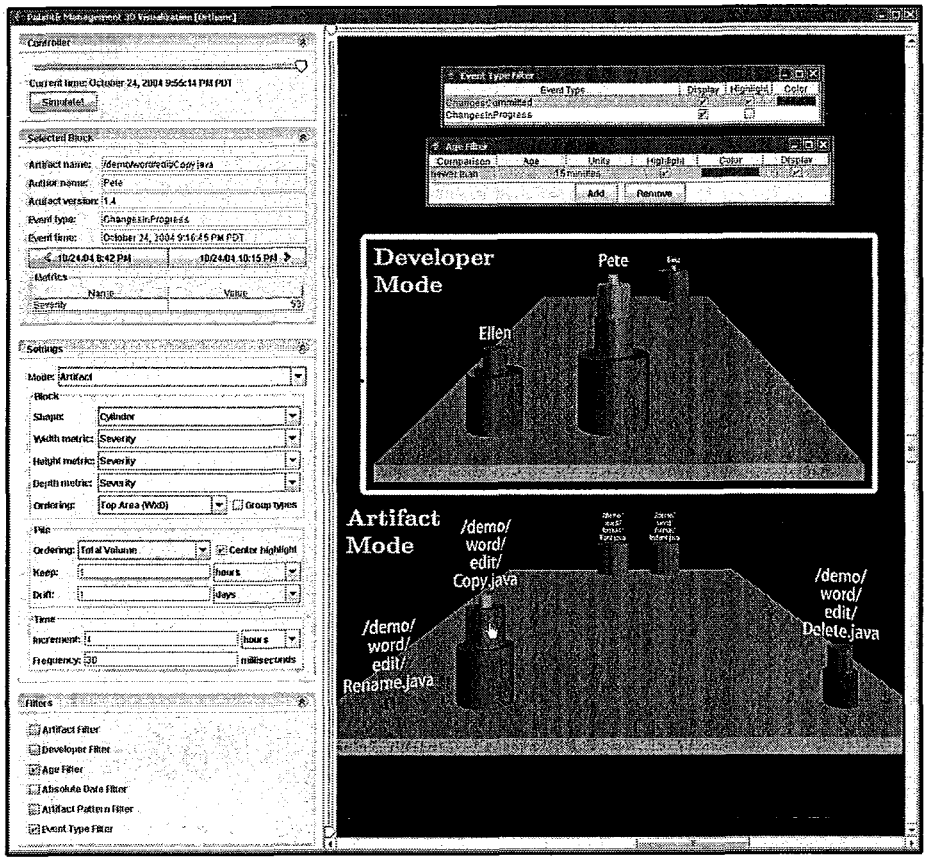



**Figura 2.10 - *Workspace Activity Viewer*: visualização 3D do Palantír (RIPLEY et al., 2007).**

### *2.5.4. Jazz*

O projeto Jazz (HUPFER et al., 2004), da IBM, tem com objetivo melhorar o suporte à colaboração de equipes de *software* no ambiente de desenvolvimento Eclipse. Através de diferentes extensões do ambiente, o Jazz provê mecanismos de percepção e ferramentas de comunicação para a equipe. Em sua principal visualização (Figura 2.11), são mostrados cada um dos desenvolvedores do projeto, representado por seu nome, sua foto, um ícone indicando a sua disponibilidade para comunicação e uma descrição textual para informações mais detalhadas. Essa mensagem pode ser configurada para automaticamente apresentar informações como

o artefato no qual o desenvolvedor está trabalhando, ou que parte do ambiente de desenvolvimento ele está utilizando no momento.

Para facilitar a comunicação entre os integrantes da equipe, é possível mandar mensagens de texto (que podem ser relacionadas ao código-fonte para futuras consultas), fazer chamadas de voz ou iniciar uma sessão de compartilhamento de tela, onde se consegue ver e interagir remotamente com o ambiente de desenvolvimento de um desenvolvedor. A equipe também conta com uma área de discussão, onde os desenvolvedores podem debater tópicos relacionados ao projeto e se informar sobre eventos relacionados ao SCV, como *check-in* e *check-out* do código-fonte compartilhado. O Jazz também adiciona decorações na lista de artefatos do projeto, indicando se existe uma versão mais nova do artefato no SCV, se alguém está modificando o artefato nesse exato momento ou se o artefato foi modificado, mas não foi feito o *check-in* ainda. O editor de código-fonte também é decorado para indicar trechos de código que foram modificados por outro desenvolvedor.

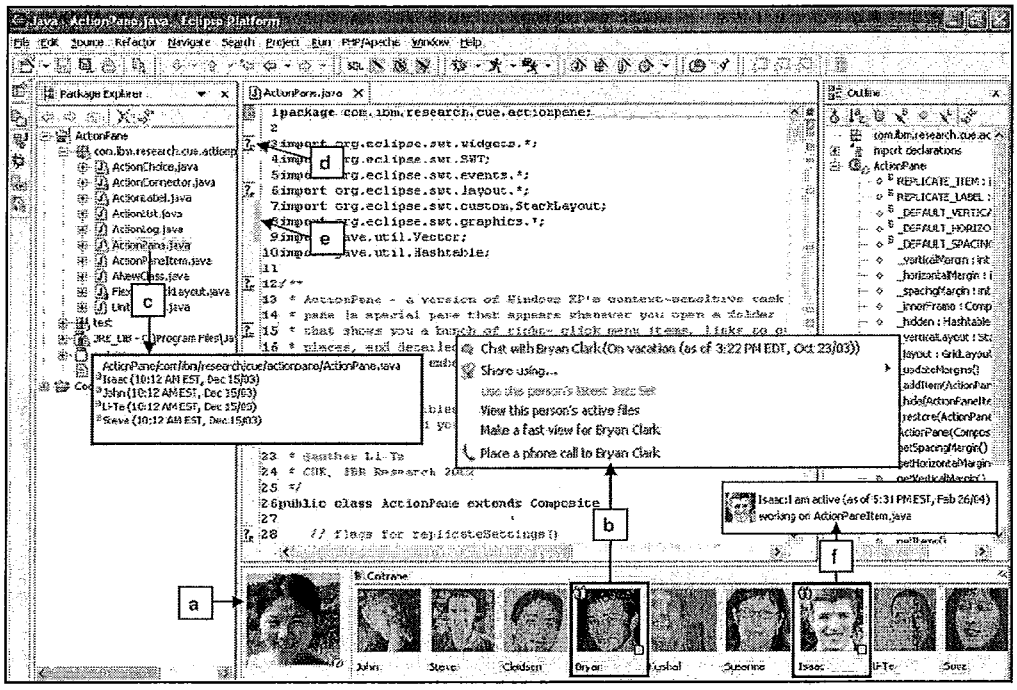



**Figura 2.11 - Jazz no ambiente Eclipse: (a) visualização dos integrantes da equipe; (b) opções de comunicação; (c) decorações na lista de artefatos; (d) e (e) decorações no editor de código-fonte e (f) informações sobre integrante da equipe (HUPFER et al., 2004).**

Atualmente está em desenvolvimento um novo projeto, também denominado Jazz (IBM RATIONAL, 2008b; FROST, 2007), com objetivos similares ao descrito anteriormente. O novo projeto, porém, é mais ambicioso, envolvendo uma grande infra-estrutura de colaboração no ambiente Eclipse. Essa infra-estrutura conta com uma máquina de processos, um repositório para o compartilhamento de artefatos e

informações em geral e um sistema de GCS (incluindo controle de versão, modificação e construção). O projeto conta ainda com ferramentas de comunicação, de definição de requisitos, de geração automática de relatórios e sítios, além de prover serviços *web* de consulta às informações do projeto. A filosofia desse novo projeto é promover um processo de desenvolvimento mais transparente, onde os interessados possam consultar as informações sobre um determinado projeto de *software* sem ter que perguntar diretamente às pessoas envolvidas.

### *2.5.5. Flecha*

O editor Flecha (CAMARGO e SOUZA, 1992), foi desenvolvido com objetivo de prover funcionalidades de edição colaborativa de diagramas de objetos para o ambiente de desenvolvimento TABA (TRAVASSOS e ROCHA, 1992). O editor provê diferentes mecanismos para suportar a comunicação e a coordenação das modificações feitas pelos membros da equipe. Além disso, o histórico do diagrama e de comunicação da equipe é armazenado para consultas futuras. A tela principal do editor pode ser vista na Figura 2.12.

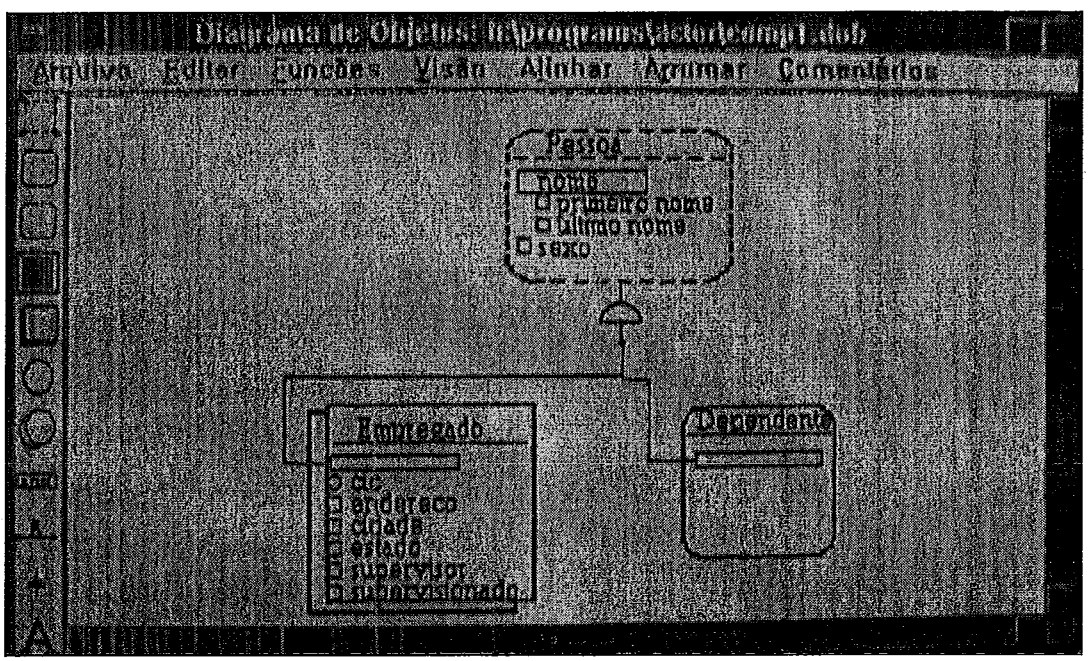



**Figura 2.12 - O editor colaborativo Flecha** (CAMARGO e SOUZA, 1992).

Para lidar com a coordenação durante a edição concorrente, um diagrama pode ser dividido em diferentes "contextos", que são associados a cada um dos membros da equipe. Um elemento do diagrama pode pertencer a mais de um contexto simultaneamente. Nesse caso, o primeiro usuário que acessar um contexto ao qual o elemento pertence tem a preferência para a edição, bloqueando-o para os demais usuários. No entanto, mesmo quando o elemento está bloqueado, as mudanças que

são feitas sobre ele podem ser vistas em tempo real por todos. Além disso, os usuários podem se comunicar através de mensagens diretas (pessoais), privadas (somente para os criadores do diagrama) ou públicas (para todos os usuários).

Os usuários podem exercer diferentes papéis durante a edição do diagrama. O papel de gerente permite a criação de novos diagramas, a divisão em contextos, a associação de usuários aos contextos e demais atividades relacionadas ao controle de acesso ao diagrama. Os co-modeladores são os usuários responsáveis pela edição de um ou mais contextos do diagrama. Um co-modelador pode controlar o acesso aos elementos dentro dos seus contextos. Finalmente, o usuário leitor é aquele que só pode observar o diagrama e deixar mensagens, sem poder alterar os elementos nele contidos.

### *2.5.6. Fast Dash*

O projeto Fast Dash (BIEHL et al., 2007) foi desenvolvido pela Microsoft, a partir de uma série de entrevistas feitas com empregados da empresa sobre percepção de grupo. O resultado da pesquisa foi, então, utilizado para desenvolver um protótipo focado em informações de percepção sobre os artefatos compartilhados pela equipe. A ferramenta apresenta essas informações através de uma visualização espacial hierárquica dos diretórios e arquivos do projeto, que é decorada com informações coletadas a partir da área de trabalho dos integrantes da equipe (Figura 2.13).

O Fast Dash tem integração com o ambiente de desenvolvimento Visual Studio (MICROSOFT, 2008b) com suporte para dois SCMs, o Team Foundation Server (MICROSOFT, 2008c) e um outro cujo nome não foi citado pelos autores do trabalho. Através dessa integração, as seguintes informações são coletadas e apresentadas para a equipe:

- Quais arquivos estão abertos e por quem (com fundo colorido de azul e decorados com a foto da pessoa que o abriu ou por um ícone especial no caso de múltiplas pessoas)
- Quais arquivos estão sendo editados (fundo colorido de amarelo);
- Em quais arquivos os programadores estão focados (com borda dourada);
- Se o desenvolvedor está depurando o sistema através do ambiente de desenvolvimento (com borda vermelha);
- Se aplicável, qual classe, método ou atributo o desenvolvedor está visualizando/implementando/depurando;
- De quais arquivos foram feitos *check-outs* e quem os fez (ícone amarelo em forma de V);
- Quais desses arquivos estão diferentes da atual versão no repositório (nome do arquivo em amarelo);

•• Onde estão os potenciais conflitos (por exemplo, quais arquivos estão sendo editados em paralelo) (listrado em vermelho).

Como a ferramenta foi desenvolvida para equipes localizadas, os autores recomendam que essa visualização seja apresentada para a equipe num tela grande, compartilhada pela equipe, ou em monitores secundários na mesa de cada indivíduo. No entanto, o conteúdo e organização da visualização são os mesmos para toda a equipe, independente do local de apresentação. O protótipo implementado também não persiste as informações coletadas, impossibilitando o seu uso como fonte de dados históricos do projeto.

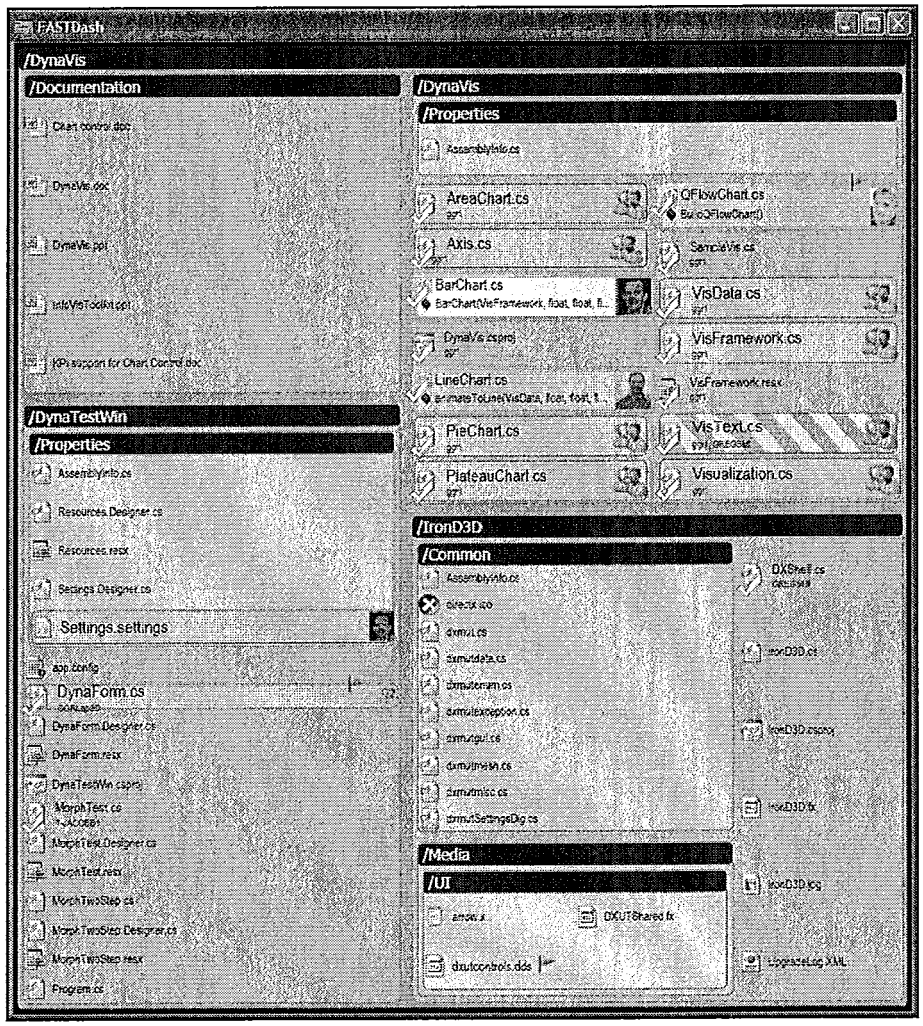



**Figura 2.13 - Visualização dos artefatos compartilhados no Fast Dash (BIEHL et al., 2007).**

### *2.5.7. Collide*

Collide (SCHNEIDER et al., 2004) é um mecanismo de percepção de grupo para o ambiente de desenvolvimento Eclipse, que provê uma visão geral do que está acontecendo com os artefatos compartilhados de um projeto de *software*. A ferramenta

coleta informações sobre os artefatos de um repositório "sombra", que vai sendo preenchido automaticamente, enquanto os desenvolvedores vão fazendo suas mudanças localmente. Toda vez que um artefato é salvo no ambiente, o Collide faz um *check-in* para este repositório intermediário, gerando um histórico de versões mais detalhado do que o repositório final. As informações coletadas são então analisadas para identificar diferentes elementos de código, as relações entre eles e entre diferentes versões de um mesmo elemento. São também identificados os autores das diferentes versões.

A ferramenta provê duas maneiras de visualizar essas informações. A primeira é uma visão compacta do projeto, chamada ProjectWatcher (Figura 2.14, à esquerda). Nela, os arquivos compartilhados são representados por pequenos retângulos, agrupados por diretórios. Dentro de cada retângulo, pequenas barras verticais representam as mudanças feitas naquele arquivo. Os retângulos podem ser coloridos de acordo com filtros que indiquem, por exemplo, o autor das mudanças mais recentes de cada um.

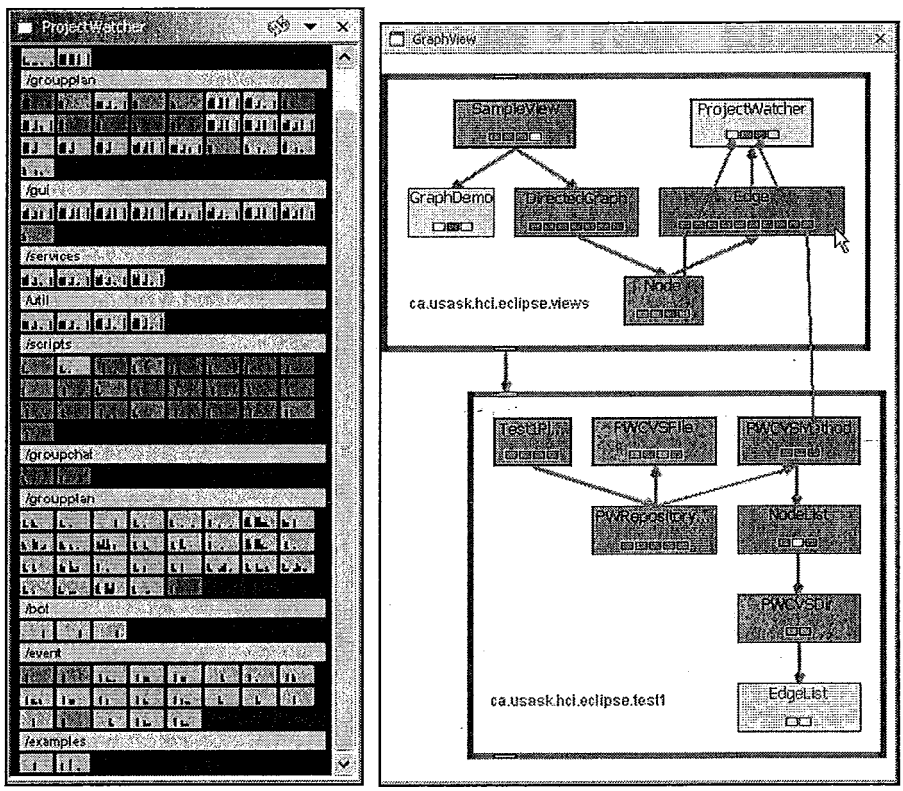



**Figure 2.14 - Visualizações da ferramenta Collide: Project Watcher (à esquerda) e GraphView (à direita) (SCHNEIDER et al., 2004).**

A segunda visualização, denominada GraphView (Figura 2.14, à direita), reflete a estrutura do código, onde os elementos são representados por retângulos e seus relacionamentos são representados por setas. Mais um vez os elementos podem ser coloridos de acordo com seu autor. Essa visualização tem como objetivo auxiliar o

desenvolvedor a entender quem está trabalhando "próximo" a ele, em relação aos artefatos do projeto.

### 2.5.8. Discussão

Analisando as ferramentas apresentadas, podemos perceber algumas tendências, como o uso de SCVs como fontes de informação e a integração com ambientes de desenvolvimento. É possível notar também a variedade de maneiras de se representar as mesmas informações. Para facilitar a comparação entre as ferramentas apresentadas, a Tabela 2.2 mostra, de forma resumida, a classificação de cada uma delas em relação aos aspectos identificados na Seção 2.4.

Observa-se que o tipo e a fonte de informações são aspectos fortemente relacionados, sendo a combinação mais comum o uso do histórico de mudanças do projeto coletado dos repositórios dos SCVs. Algumas ferramentas como o Palantír e o Collide, no entanto, utilizam históricos mais completos, por complementarem as informações dos SCVs com o monitoramento das modificações diretamente nas áreas de trabalho dos integrantes da equipe. Como o intervalo de tempo entre o *check-out* e o *check-in* de um artefato pode ser longo, a área de trabalho se torna uma fonte de informação mais detalhada e atualizada sobre eles.

A área de trabalho também provê outras informações importantes, como o foco de um determinado desenvolvedor no ambiente de desenvolvimento, incluindo os artefatos nos quais está trabalhando ou visualizando no momento e que tipo de ferramentas está utilizando. Outras fontes comumente utilizadas são ferramentas de comunicação, principalmente as baseadas em mensagem de textos, que são facilmente armazenadas e podem ser utilizadas como base de conhecimento.

Em relação à análise das informações coletadas, nem todas as ferramentas se aplicam. O tipo de análise mais comum é a simples filtragem da informação, por vezes utilizadas para diminuir a quantidade de informações mostradas (como no caso do GAW) ou para classificar elementos por um determinado critério (MAIS, Collide). Apenas o Tukan e o Palantír fazem uma análise mais avançada das informações, para fazer inferências sobre elas.

O aspecto de representação das informações, como visto anteriormente, foi o mais diversificado. Listas, gráficos, painéis e decorações estão entre os elementos mais utilizados pelas ferramentas, talvez por influência dos próprios ambientes de desenvolvimento, que utilizam com freqüência essas representações. Na maioria dos casos, no entanto, as representações sofrem com problemas de escalabilidade, e pode se tornar difícil acompanhar as informações de projetos e equipes grandes.

O local de apresentação das informações, ao contrário da representação, foi bastante uniforme entre as ferramentas descritas. A maioria das ferramentas explora a integração com ambiente de desenvolvimento para manter as informações de

percepção sempre ao alcance dos indivíduos. Somente o projeto Fast Dash, provavelmente por ser mais recente, explora o uso de telas grandes e compartilhadas pela equipe ou monitores adicionais. Mesmo assim, o projeto não explora as particularidades de cada uma dessas opções, mostrando as mesmas informações e as representando da mesma forma, em ambos os casos.

O público-alvo dessas ferramentas é, em grande maioria, o desenvolvedor. De acordo com diferentes estudos (LATOZA et al., 2006; OLSON et al., 1992; PERRY et al., 1994), o desenvolvedor gasta boa parte do seu tempo para manter a percepção do estado atual do projeto. LATOZA et al. (2006), inclusive, indicam que até 40% de um dia de trabalho de um desenvolvedor é utilizado para comunicação sobre o código-fonte. Desta forma, não é surpreendente que existam tantas ferramentas com o objetivo de auxiliá-los nestas tarefas.

No entanto, muitas dessas ferramentas podem ser utilizadas por pessoas de diferentes cargos e tarefas numa equipe de desenvolvimento de *software,* como, por exemplo, o editor colaborativo Flecha. Algumas informações podem ser úteis tanto para desenvolvedores quanto gerentes, como, por exemplo, a autoria dos artefatos do sistema. Enquanto um desenvolvedor pode utilizar essa informação para encontrar o responsável por uma mudança conflitante, ou para tirar dúvidas sobre o funcionamento de uma função, o gerente pode utilizar essa informação de forma mais abrangente, para entender quem é o principal responsável por quais partes do sistema, antes de dividir e atribuir as tarefas para a equipe.

**Tabela 2.2 - Comparação entre as ferramentas de apoio à colaboração**

| Ferramenta | | Tipo de Informação | Fonte | Análise | Representação | Local | Público-alvo |
|---|---|---|---|---|---|---|---|
| OdysseyShare | Shared Workspaces | Presença no espaço compartilhado, disponibilidade para comunicação, histórico de comunicação | Ambiente de desenvolvimento, ferramenta de comunicação | - | Teleapontador, visão miniatura, lista de indivíduos decorada | Ambiente de desenvolvimento | Analista, Desenvolvedor |
| | MAIS | Histórico de mudanças dos artefatos | Ambiente de desenvolvimento | Relevância, filtros | Listas de eventos | | |
| | GAW | | | Filtros | Lista de indivíduos com barras coloridas numa linha de tempo | | |
| Tukan | | Estrutura dos artefatos, histórico de mudanças dos artefatos, disponibilidade para comunicação | SCV, ambiente de desenvolvimento, ferramentas de comunicação | Foco, conflitos | Listas de artefatos decorada | Ambiente de desenvolvimento | Desenvolvedor |
| Palantír | Visualização Gráfica | Estrutura dos artefatos, histórico de mudanças dos artefatos | SCV, ambiente de desenvolvimento | Relevância, conflitos | Diagrama hierárquico dos artefatos | Aplicação independente | Desenvolvedor |
| | Explorer | | | | Listas de artefatos decorada | | |
| | Scrolling Marquee | | | | Lista animada de eventos | | |
| | Eclipse | | | | Listas de artefatos decorada, lista de conflitos | Ambiente de desenvolvimento | |
| | WAV | | | | Cilindros num plano 3D animado | Aplicação independente | Gerente de projeto |
| Flecha | | Histórico de comunicação, histórico de mudanças | Ambiente de desenvolvimento, ferramenta de comunicação | - | Lista de artefatos compartilhados, anotações no diagrama | Ambiente de desenvolvimento | Analista, Desenvolvedor |
| Jazz | | Histórico de mudanças dos artefatos, disponibilidade para comunicação, histórico de comunicação | SCV, ambiente de desenvolvimento, ferramentas de comunicação | - | Listas de artefatos decorada, lista de indivíduos decorada, decorações no editor de artefatos | Ambiente de desenvolvimento | Desenvolvedor |
| FashDASH | | Histórico de mudanças dos artefatos | SCV, ambiente de desenvolvimento | - | Painel hierárquico de artefatos decorado | Tela grande, monitor secundário | Desenvolvedor |
| Collide | | Histórico de mudanças dos artefatos, estrutura dos artefatos | SCV, ambiente de desenvolvimento | Filtros | Painel hierárquico de artefatos decorado, diagrama de estrutura do código-fonte decorado | Ambiente de desenvolvimento | Desenvolvedor |

## 2.6. Considerações Finais

Neste capítulo, foram apresentadas as diferentes abordagens para a coordenação de equipes de desenvolvimento de *software*. Comparamos as abordagens formal e informal e discutimos como elas são usadas de forma complementar na prática, resultando em uma terceira abordagem: a coordenação contínua. Apresentamos, também, a disciplina de Gerência de Configuração de *Software*, que segue a abordagem formal e que provê sistemas que são fontes ricas de informações sobre o projeto de *software*. Em seguida, descrevemos os Mecanismos de Percepção de Grupo, que seguem a abordagem informal para facilitar a troca de informações dentro da equipe. Identificamos, também, diferentes aspectos pelos quais esses mecanismos podem ser classificados, como seu público-alvo e a forma de representação das informações. Finalmente, indicamos algumas das ferramentas que combinam abordagens formais e informais e as comparamos a partir dos aspectos previamente identificados.

Através desta análise da literatura e dos trabalhos relacionados, conseguimos identificar alguns pontos importantes, tanto em relação à abordagem geral utilizada atualmente quanto aos diferentes aspectos identificados:

- *Abordagem geral*: a colaboração e a coordenação em equipes de desenvolvimento de *software* é uma tarefa complexa e, na prática, envolve atividades formais e informais. A combinação de abordagens formais e informais no suporte ferramental provido para estas equipes oferece tanto o controle definido pelos processos, quanto a flexibilidade da percepção de grupo. Essa combinação é cada vez mais comum nas ferramentas de apoio à coordenação.
- *Fontes de informação*: O código de um sistema de *software* é uma fonte rica de informações sobre o projeto e é bastante explorada pelas ferramentas de apoio à colaboração, refletindo o estado atual de implementação do projeto. Além disso, o código fonte é geralmente de fácil acesso. Os SCVs são os principais meios de obtenção do código e provêem informações adicionais como autoria e dados históricos. No entanto, é cada vez mais comum o monitoramento da área de trabalho dos desenvolvedores, por esta conter informações mais atualizadas e prover um histórico mais detalhado dos dados.
- *Análise da Informação*: Como a quantidade de informações de percepção é grande, é interessante que se faça algum tipo de análise antes da apresentação para os integrantes da equipe. Esta análise pode ser utilizada para filtrar as informações de acordo com algum critério definido explicitamente, para detectar padrões e tendências ou para inferir quais são as informações mais relevantes para um determinado indivíduo. Esta análise, no entanto, deve ser leve o suficiente para não prejudicar a atualização das informações apresentadas aos usuários.

- *Representação da Informação*: Existem inúmeras maneiras de representar as informações de percepção. É interessante prover uma representação que sirva de modelo comum para a equipe, facilitando a comunicação. É importante, no entanto, prover soluções para possíveis problemas como a escalabilidade da representação e para a sobrecarga de informações, além de tentar minimizar interrupções às tarefas principais dos integrantes da equipe.
- *Apresentação da informação*: É importante apresentar as informações em um local facilmente acessível aos usuários. A solução mais comum é a integração das ferramentas com o ambiente de desenvolvimento já utilizado pela equipe. No entanto, com o barateamento recente dos equipamentos de vídeo, é também interessante explorar outros locais de apresentação, como monitores secundários ou telas centrais compartilhadas por toda equipe.
- *Público-alvo*: O principal público-alvo das ferramentas apresentadas é o desenvolvedor. Isso pode se dar ao fato de estes trabalharem diretamente com o código-fonte, pela necessidade de trabalhar em paralelo com os demais e, portanto, terem que coordenar cuidadosamente seus esforços e por passar boa parte do tempo se comunicando sobre o código. No entanto, as informações de percepção também são úteis para pessoas que desempenham outros cargos na equipe, como analistas e gerentes.

# 3. A Abordagem Lighthouse

## 3.1. Introdução

No capítulo anterior foram apresentados diferentes aspectos pelos quais mecanismos de percepção podem ser classificados. Essa classificação foi utilizada, então, para descrever os trabalhos relacionados encontrados na literatura. Através da comparação desses trabalhos, foi possível destacar algumas características comuns às abordagens. Algumas destas semelhanças são positivas, como, por exemplo, a integração com o ambiente de desenvolvimento. No entanto, a maioria das ferramentas também apresenta problemas em comum, como a falta de escalabilidade da representação das informações de percepção.

Neste capítulo, são utilizados os pontos destacados no capítulo anterior como base para a proposta deste trabalho de pesquisa: o mecanismo de percepção Lighthouse (SILVA et al., 2006). Ao longo deste capítulo, a abordagem proposta é apresentada em progressivos níveis de detalhe. Na Seção 3.2, é apresentada a proposta geral desta abordagem; na Seção 3.3, as etapas previstas de execução do Lighthouse são definidas; e, na Seção 3.4, estas etapas são detalhadas através da descrição dos requisitos de cada uma. Finalmente, as considerações finais deste capítulo são apresentadas na Seção 3.5.

## 3.2. Proposta

Através da revisão da literatura e da comparação entre diferentes trabalhos de pesquisa feitas no Capítulo 2, foram encontradas abordagens interessantes na área de colaboração e coordenação em equipes de desenvolvimento de *software*. Foram observadas algumas similaridades em relação às características das ferramentas encontradas na literatura. Algumas das similaridades identificadas são positivas e devem ser incorporadas na proposta deste trabalho. No entanto, existem também problemas em comum nas abordagens encontradas, que devem ser tratados neste trabalho.

A abordagem proposta nesta dissertação é um novo mecanismo de percepção para equipes de desenvolvimento de *software*, denominado Lighthouse. Este mecanismo se baseia numa abstração da estrutura do sistema de *software* em desenvolvimento, construída automaticamente a partir do seu código-fonte. Essa abstração, denominada *Design* Emergente, representa os elementos que compõem a estrutura do código utilizando uma notação similar à do modelo de classes da UML (OMG, 2008). No entanto, o *Design* Emergente é uma abstração dinâmica, sendo

atualizada constantemente para refletir a evolução da implementação enquanto a equipe desenvolve colaborativamente o sistema. Desta forma, esta abstração apresenta um modelo compartilhado do sistema para toda a equipe.

O *Design* Emergente pode ser decorado com informações de percepção de grupo, coletadas a partir da área de trabalho e do SCV, com o intuito de dar suporte à coordenação da equipe. Através do *Design* Emergente, os integrantes da equipe podem se informar sobre a estrutura atual do sistema; as mudanças (no código-fonte) que afetaram esta estrutura; quem foi o responsável por cada mudança feita; e como as mudanças estão progredindo em relação ao SCV (se a mudança é local ou se já foi feito *check-in/check-out*).

Ao analisar o diagrama de *Design* Emergente, os usuários podem identificar contribuições recentes e antigas dos demais para o sistema; detectar conflitos na implementação assim que ocorram (por exemplo, métodos duplicados ou mudanças conflitantes); saber da existência de mudanças das quais não foi feito *check-in* ainda; saber do *check-in* de mudanças assim que este ocorrer; identificar especialistas em determinadas partes do sistema, entre outros.

Como todo mecanismo de percepção, o Lighthouse tem que lidar com problemas associados à escalabilidade da sua representação, à sobrecarga de informação e à interrupção dos seus usuários. A quantidade de elementos no *Design* Emergente cresce rapidamente enquanto a implementação do *software* evolui. Mesmo em projetos pequenos, se torna difícil visualizar todos os elementos de uma só vez. Para entender o que está acontecendo no projeto, pode ser necessário focar em partes específicas da estrutura do sistema. No entanto, mesmo que se tente ver todo o diagrama de uma só vez, é provável que muitas informações não sejam relevantes para um determinado indivíduo a todo momento.

Muitas das ferramentas apresentadas no capítulo anterior se baseiam na premissa de que o usuário deve procurar pelas informações relevantes ao seu trabalho. Enquanto essa abordagem dá aos usuários uma grande flexibilidade para explorar as informações como queiram, algumas informações importantes podem não receber a devida atenção. Além disso, a quantidade de informações é muito grande e não é desejável que os usuários se distraiam demais das suas tarefas principais para separar as informações relevantes dentre as disponíveis.

O Lighthouse deve, portanto, analisar as informações disponíveis e selecionar aquelas que sejam mais relevantes para o contexto atual de trabalho do usuário. Desta forma, as informações mais importantes são automaticamente priorizadas no diagrama de *Design* Emergente, com o intuito de poupar o usuário, deixando que ele se concentre nas suas tarefas principais. Além disso, o Lighthouse deve evitar tirar o foco do trabalho de implementação de seus usuários, minimizando interrupções.

Em linhas gerais, a abordagem proposta é representada pelo mecanismo de percepção Lighthouse, que provê à equipe um modelo auto-atualizável do sistema, denominado *Design* Emergente. Este modelo é anotado com informações de percepção que são priorizadas de acordo com sua potencial relevância para o contexto de trabalho de cada indivíduo (o contexto, nesta abordagem, é determinado pelo conjunto de artefatos no qual o desenvolvedor está trabalhando). Essa ordenação tem como objetivo dar maior ênfase às informações que têm potencial impacto no trabalho de cada indivíduo, para que ele possa agir o mais rápido possível quando oportunidades de colaboração são identificadas ou problemas são detectados. A priorização também ajuda a filtrar informações menos interessantes naquele momento, diminuindo a interferência com as demais tarefas dos indivíduos.

## 3.3. Etapas de Execução

Um mecanismo de percepção passa por diferentes etapas durante a sua execução, desde a coleta das informações necessárias até a apresentação destas para o usuário final. Nesta seção, são apresentadas as etapas propostas para a execução do mecanismo de percepção Lighthouse. O fluxo de execução dessas etapas está representado na Figura 3.1.

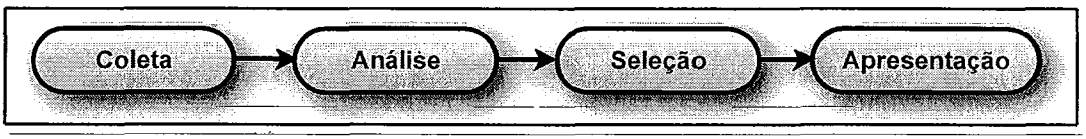



**Figura 3.1 - Etapas de execução propostas para o mecanismo Lighthouse.**

A primeira etapa é a coleta das informações de percepção. Como visto no capítulo anterior, existem diferentes fontes de dados e diferentes tipos de informação disponíveis. O código-fonte é uma boa fonte de informações sobre o projeto e que deve servir de base para o diagrama de *Design* Emergente. No entanto, para construir o diagrama, é necessário que a coleta seja feita nas áreas de trabalho dos integrantes da equipe. É importante que esta coleta seja feita constante e automaticamente, para que a equipe sempre tenha acesso rápido às informações mais atuais sobre o projeto.

Na segunda etapa, a análise, os diferentes tipos de informação coletados são combinados para construir e manter uma base de dados sobre o sistema sendo desenvolvido. Nesta etapa, os dados não são simplesmente armazenados e, sim, analisados para agregar informações dispersas, facilitando o seu entendimento. Esta base pode ser utilizada, então, nas demais etapas como fonte de informações. Como um mecanismo de percepção, é importante que as informações cheguem em tempo hábil para seus usuários. A análise, portanto, precisa ser um processo leve, para não atrasar a apresentação das informações de percepção.

A quantidade de informações coletadas e, consequentemente, analisadas pelos mecanismos de percepção, é enorme, como visto anteriormente. A etapa de seleção é responsável por determinar quais informações devem ser apresentadas para cada usuário. Idealmente, esta seleção escolheria somente as informações que são relevantes para um determinado usuário num determinado momento. No entanto, como nem sempre é possível inferir o que é relevante diretamente das informações disponíveis, também é necessário permitir que o próprio usuário participe desta seleção.

Finalmente, na última etapa, as informações consideradas relevantes são apresentadas para os usuários finais. A representação das informações deve servir como um modelo comum da estrutura do sistema em desenvolvimento. Este modelo deve ser anotado com as demais informações de percepção disponíveis.

## 3.4. Requisitos

Nesta seção, são detalhados os passos apresentados na Seção 3.3, apresentando os requisitos propostos para desenvolvê-los.

### *3.4.1. Coleta*

A etapa de coleta dos dados representa a busca por informações sobre o sistema de *software* em desenvolvimento pela equipe. Como visto anteriormente, o próprio código do sistema é uma boa fonte de informações e pode ser facilmente obtido através do SCV. No entanto, o SCV pode demorar a ser atualizado, não contendo necessariamente a versão mais recente do sistema. Conforme dito anteriormente, isso acontece porque os integrantes da equipe fazem as modificações no código nas suas respectivas áreas de trabalho e só depois as compartilham com os demais através do SCV. Para obter as informações mais atualizadas, o Lighthouse deve monitorar também a área de trabalho dos seus usuários, coletando dados sobre as mudanças em andamento.

O Lighthouse deve coletar diferentes tipos de informação de percepção. O principal, no entanto, é capturar o estado atual de implementação do sistema, assim como o histórico de modificações do mesmo. Estas informações são úteis para o entendimento da estrutura corrente do sistema e para o acompanhamento da evolução do mesmo. Essa estrutura serve de base para a representação dos elementos de *software* na etapa de apresentação.

É importante obter também informações sobre a autoria de cada um dos elementos e das modificações feitas sobre eles, para que se saiba quem se deve contactar no caso de dúvidas ou problemas. Além disso, é interessante prover dados sobre a progressão dessas modificações em relação ao SCV, ou seja, identificar se as

mudanças são locais ou se já estão disponíveis no repositório para *check-out*. Esse tipo de informação é interessante para que se saiba diferenciar entre mudanças ainda em progresso e as já mais consolidadas.

Para que o Lighthouse possa posteriormente selecionar as informações que são relevantes para cada um dos usuários, é necessário coletar informações adicionais sobre eles. A seleção automática das informações deve ser baseada no contexto de trabalho do usuário, para que se possa inferir sobre o que é importante para o usuário no momento. Este contexto é representado pelos artefatos de *software* que ele está editando ou simplesmente visualizando em sua área de trabalho. Provavelmente estes artefatos são relacionados com a tarefa atual do usuário, e, portanto, informações de percepção sobre eles são relevantes para o usuário.

### *3.4.2. Análise*

O principal objetivo da análise de dados proposta é combinar as informações coletadas e priorizar as mais relevantes para o contexto de trabalho de um usuário específico. Essa priorização é importante, pois permite que o mecanismo escolha melhor as informações que serão mostradas mais à frente ao usuário, evitando sobrecarga e facilitando a sua representação.

Para fazer esta análise, o Lighthouse precisa analisar o sistema de *software* em si e entender como os diferentes elementos do *software* se relacionam. Em seguida, é necessário encontrar os elementos de *software* mais relevantes para o usuário no momento e, finalmente, priorizar a apresentação das informações sobre estes elementos.

Existem várias maneiras de realizar a análise do sistema. É desejável, no entanto, que esta análise seja leve, pois ela precisa ser executada continuamente sem comprometer o desempenho da ferramenta. Como mecanismo de percepção, é importante que o Lighthouse consiga atualizar suas informações rapidamente.

A ordenação das informações sobre os elementos de software deve ser feita de acordo com o quão fortemente relacionados eles são aos elementos em que o usuário está trabalhando. Entre os elementos disponíveis, é preciso destacar aqueles que podem impactar ou ser impactados pela tarefa atual do usuário. Essa classificação o ajudaria a entender quais partes do sistema (e, consequentemente, que outros usuários), podem ser influenciados pela suas recentes contribuições ao código. Ao mesmo tempo, ele poderá encontrar as contribuições de outros que possam afetar a sua própria tarefa.

No restante desta seção, vamos discutir as opções de análise consideradas e as escolhas feitas para esta abordagem.

#### 3.4.2.1. Análise estática x Análise dinâmica

Para realizar a análise do sistema de *software* sendo implementado, o Lighthouse poderia obter seus dados a partir de uma representação estática do sistema (como o código-fonte), ou a partir da execução do sistema em si (análise dinâmica). A análise estática pode ser facilmente automatizada, utilizando um interpretador de código, por exemplo. Todas as chamadas de um método para outro podem ser armazenadas, construindo um grafo de dependências entre eles. Para códigos-fonte escritos em linguagens estruturadas como C, este tipo de análise é suficiente para a grande maioria dos casos (com exceção das chamadas feitas através de ponteiros para funções). No entanto, no paradigma Orientado a Objetos (OO), chamadas a métodos só são perfeitamente rastreáveis em tempo de execução, devido à utilização de recursos como interfaces ou polimorfismo. Para linguagens OO, portanto, uma análise dinâmica pode ser mais precisa.

A análise dinâmica, porém, não pode ser totalmente automatizada, pois exige que o *software* seja executado, enquanto o analisador recolhe os traços de execução dos seus elementos (ORSO et al., 2004). Isso não é desejável, uma vez que exigiria tirar o foco dos usuários de suas tarefas principais para executar esta análise periodicamente. Assim, optamos por usar apenas a análise estática na nossa abordagem. Para tentar amenizar as limitações deste tipo de análise, deve-se levar em conta as características específicas das linguagens na análise, quando possível.

#### 3.4.2.2. Executável x Código-fonte x Modelo

Pode-se analisar um sistema de *software* a partir de diferentes artefatos: o arquivo binário executável, o código-fonte, ou um modelo de maior nível de abstração (como o modelo de classes da UML). Enquanto executáveis são mais comumente usados em abordagens que utilizam análise dinâmica, informações de arquivos binários também podem ser estaticamente exploradas (Por exemplo, ver (NAEEM e HENDREN, 2006)). A análise do executável é especialmente útil quando o código-fonte do *software* não está disponível (BINKLEY, 2007) (o que não é o caso do Lighthouse). O principal problema desta abordagem é que o *software* tem que primeiro ser compilado. Como o Lighthouse foi desenvolvido para uso em tempo real, enquanto a equipe está implementando o sistema, é razoável esperar que o seu código-fonte não compile corretamente durante grande parte do tempo. Assim, o uso deste tipo de análise restringiria nossa abordagem.

As duas abordagem restantes, análises baseadas em código-fonte e modelo também estão descritas na literatura. O uso do código-fonte é normalmente preferido, porque modelos podem ficar rapidamente ultrapassados e sua atualização é geralmente cara (BINKLEY, 2007). No entanto, uma vez que o Lighthouse mantém código e modelo sincronizados, qualquer uma dessas opções está disponível. Análise

do código-fonte pode ser mais precisa, por conter informações mais detalhadas sobre o *software*. Por outro lado, a análise do modelo pode ser mais leve e, assim, realizada com mais freqüência. Como o *Lighthouse* já extrai suas informações do código-fonte, usar este tipo de análise não compromete o desempenho da ferramenta.

#### 3.4.2.3. Análise estrutural x Análise semântica

O Lighthouse precisa determinar como os elementos de *software* estão relacionados uns aos outros através da análise do seu código. Embora a própria estrutura do código já evidencie muitos desses relacionamentos (por exemplo, chamadas a métodos, hierarquia de classes), alguns deles não são explicitamente declarados (por exemplo, métodos que fazem parte da implementação de uma mesma funcionalidade). É por isso que algumas abordagens utilizam uma análise semântica para aumentar a precisão de seus resultados. Por exemplo, em linguagens OO, podemos aproveitar o uso geralmente consistente de substantivos para nomear objetos de dados e o uso de verbos em nomes de métodos para encontrar aqueles que têm alguma relação semântica (BOHNER, 2002). Por exemplo, se uma classe denominada *Cliente* é alterada, outras classes como *RelatorioCliente* ou *ClienteEspecial* poderiam ser priorizadas em relação a outros resultados.

Embora essa análise semântica possa ajudar a ordenar os resultados da análise estrutural, esta seria uma computação adicional, que poderia retardar o nosso processo. Como priorizamos abordagens leves, não pensamos em executar análises semânticas inicialmente. Acreditamos, no entanto, que este tipo de análise possa ser incorporado no futuro, caso se mostre necessário. Uma opção para se incorporar essa abordagem sem retardar tanto o processo como um todo, seria manter a análise estática (mais superficial) constante e executar, em um intervalo maior de tempo, a análise semântica (mais profunda), combinando o resultado das duas periodicamente.

#### 3.4.2.4. Versão atual x versões anteriores

A análise de código-fonte pode levar em conta apenas a versão atual do sistema, indicando, assim, a forma como os elementos de *software* estão relacionados neste momento. Outra opção é analisar a evolução desses elementos, para identificar aqueles que se relacionam através de seus históricos de modificação. Com este tipo de análise, por exemplo, o Lighthouse poderia inferir relacionamentos do tipo: "quando uma mudança é feita no elemento A, o elemento B geralmente é modificado também". Esta é mais uma maneira interessante de se detectar relacionamentos lógicos entre elementos, que não estão definidos no código-fonte. No entanto, para esta análise, precisaríamos usar alguma técnica de mineração de dados (KAGDI et al., 2005), que também seria uma computação pesada para nossa análise constante. Mais uma vez, priorizamos a leveza, mas não descartamos essa possibilidade para o futuro.

### 3.4.2.5. Relacionamento direto x Relacionamento Indireto

Um elemento de código pode ter um relacionamento direto ou indireto com outro elemento. A relação direta ocorre quando um elemento está relacionado através de uma das dependência que chegam/saem diretamente de/para o outro elemento. Por exemplo, na Figura 3, o elemento A está diretamente relacionado com o elemento B. Se A e B forem classes em uma linguagem OO, esse relacionamento pode ser, por exemplo, uma chamada que um método em A faz a um método em B.

Relacionamentos indiretos ocorrem quando os elementos estão relacionados através de um conjunto de dependências que representam um caminho entre elas. Por exemplo, o elemento A é indiretamente relacionado com os elementos C, D, E e F através do seu relacionamento com B (Figura 3.2). Mais uma vez, se pensarmos nesses elementos como classes, A chama diretamente um método em B, que por sua vez chama métodos em C e D, e assim por diante.

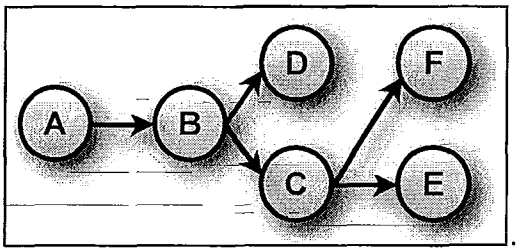



**Figura 3.2 - Exemplo de dependências entre elementos de código.**

Relacionamentos indiretos são mais difíceis de se detectar e, muitas vezes, não são planejados. Além disso, a quantidade de itens relacionados aumenta consideravelmente, quando cresce a distância considerada entre os elementos. Portanto, num caso extremo, uma análise dos relacionamentos indiretos poderia resultar em todos os elementos que compõem o sistema estarem relacionados. É necessário, então, priorizar ou limitar os resultados de alguma forma, como descrito na próxima seção. Em nossa análise, levamos em conta esses relacionamentos indiretos, por eles serem mais dificilmente detectados, especialmente por alguém novo no projeto.

### 3.4.2.6. Métricas de Acoplamento

Como visto na subseção anterior, não basta determinar quais são as relações existentes entre elementos do código, pois todos os elementos podem estar relacionados de alguma forma. O Lighthouse precisa utilizar algum tipo de métrica para priorizar as informações sobre os elementos que estão mais fortemente relacionados com o contexto de trabalho de cada indivíduo. Nessa abordagem, este contexto é determinado pelos elementos que estão sendo editados ou visualizados pelo usuário na sua área de trabalho. Portanto, para inferir quais elementos são

relevantes naquele contexto, é preciso determinar quais elementos são mais fortemente acoplados àqueles sendo editados ou simplesmente visualizados.

Há uma grande variedade de métricas de acoplamento para linguagens OO na literatura (BRIAND et al., 1999a). Das que podem ser calculadas utilizando as escolhas anteriores (com base numa única versão da estrutura estática do código-fonte), é especialmente interessante o trabalho de BRIAND et al. (1999b). Neste artigo, trinta métricas de acoplamento OO são empiricamente comparadas no contexto de uma análise de impacto. Três dessas métricas foram consideradas pelos autores as mais precisas na indicação de pares de classes que têm grande probabilidade de terem efeitos colaterais uma na outra. Como este é o foco do Lighthouse, utilizamos essas três métricas na nossa análise. As métricas escolhidas são:

- *Acoplamento entre classes* (do inglês, *Coupling between Object Classes* - CBO') (CHIDAMBER e KEMERER, 1991): Um indicador binário de acoplamento. Duas classes A e B são consideradas acopladas se algum método de A referencia algum método ou atributo de B, ou vice-versa. Porém, para esta métrica, as classes não são consideradas acopladas se houver um relacionamento de herança entre elas.

- *Métodos Chamados Polimorficamente* (do inglês, *Polymorphically Invoked Methods* - PIM) (BRIAND et al., 1999b): O número de chamadas, na classe A, a métodos na classe B, levando em consideração as chamadas a métodos polimórficos.

- *Acoplamento através de Agregação Indireta* (do inglês, *Indirect Aggregation coupling* - INAG) (BRIAND et al., 1999b): Outro indicador binário de acoplamento. Considera que há acoplamento se uma classe A tem um atributo do tipo B. Além disso, se B tem um atributo do tipo C, A e C são indiretamente consideradas acopladas.

As três métricas parecem se complementar na determinação do acoplamento entre classes. A CBO' é a única a considerar a existência (ou a não-existência) de relacionamentos de herança entre duas classes. Já a PIM é a única não binária, contando o número de referências de uma classe para outra, e ainda leva em consideração o polimorfismo dos métodos. Finalmente, a INAG é a única a considerar relacionamentos indiretos entre as classes.

Estas métricas são calculadas para cada par de classes de um sistema de *software*. Embora isso possa parecer uma computação pesada, o cálculo completo só será realizado uma única vez. Depois de se obter os valores iniciais, ele será relativamente simples de atualizar para cada mudança feita no sistema. Dado que a maior parte das métricas são binárias, apenas alguns tipos específicos de alterações terão impacto nos seus valores e, muitas vezes, apenas algumas classes serão afetadas, simplificando a manutenção dos valores.

Ao final do cálculo, os valores dessas três métricas são combinados para cada par de elementos. Dois elementos são considerados acoplados se qualquer uma dessas três métricas indica o acoplamento. Essa combinação tem pode gerar um número alto de acoplamentos, mas foi considerada satisfatória para esta abordagem. Outras combinações podem ser testadas no futuro, como, por exemplo, a soma dos valores das métricas.

### *3.4.3. Seleção*

Embora a análise dos dados resultantes tenha um papel importante na orientação do foco do *Design* Emergente e das informações nele representadas, não é possível inferir o que exatamente cada usuário precisa saber em cada momento. Assim, além do resultado da análise, o Lighthouse também deve permitir que os usuários escolham o que deve ser mostrado no diagrama de acordo com suas necessidades.

#### 3.4.3.1. Vigia de Elementos

Os usuários do Lighthouse podem ter interesse em determinados elementos do *software* que não necessariamente são relacionados com sua tarefa atual: pode ser uma classe, cuja implementação é complexa e que não deve ser modificada por programadores inexperientes, ou um método que precisa manter um comportamento específico para que seu código funcione. O objetivo desta funcionalidade é permitir que os usuários indiquem um conjunto de "elementos de interesse", que serão "vigiados". Cada modificação neles feita será notificada, independente da prioridade que o elemento recebeu no resultado da análise. Com a ajuda desses "vigias", os usuários não perderiam as informações relevantes para eles, especialmente aquelas que não podem ser inferidas automaticamente pela análise feita pelo Lighthouse.

#### 3.4.3.2. Filtros

Com o crescimento do sistema de *software* ou da própria equipe, se torna mais difícil absorver todas as informações disponíveis no diagrama do Lighthouse. Por vezes, os usuários só precisam ter uma idéia geral do que está acontecendo no projeto, e seria mais fácil fazê-lo se pudessem ocultar muitos dos detalhes naquele momento, evitando uma sobrecarga de informações. Desta forma, a capacidade de filtrar o diagrama é uma forma de lidar com essa grande quantidade de informação, ajudando o usuário a concentrar-se no que é mais importante. Diferentes tipos de filtros devem ser providos:

- *Filtro de Elementos*: Usuários podem ter a necessidade de se concentrar em grupos específicos de classes (por exemplo, a partir de um pacote específico), filtrando o restante do diagrama.

- *Filtro de Relacionamentos*: Ao exibir todas as relações entre os elementos, o diagrama pode ficar bastante difícil de se acompanhar. Assim, seria útil ocultar certos tipos de relacionamento (associações, generalizações ou realizações) para melhorar a legibilidade do diagrama.
- *Filtro de Novidades*: Este filtro tem como objetivo alertar os usuários sobre os elementos que foram alterados recentemente, podendo ser utilizado para descobrir quais partes do sistema ainda estão sendo implementadas e quais estão mais estáveis, não tendo sido modificadas durante um longo período.
- *Filtro de Informações do SCV*: Todos os elementos mostrados no diagrama do Lighthouse apresentam informações adicionais coletadas do SCV. Embora estas informações sejam importantes do ponto de vista da coordenação da equipe, raramente há necessidade de tê-las visíveis em todos os elementos do diagrama ao mesmo tempo. Assim, a capacidade de esconder essas informações é desejada.
- *Filtro de Histórico*: O Lighthouse acompanha todas as mudanças feitas no código e as apresenta como um histórico completo de evolução dos elementos do diagrama. No entanto, com o passar do tempo, esse histórico cresce e acaba tornando difícil o entendimento do estado atual do sistema. Por esse motivo, e também por questões de escalabilidade do diagrama, o Lighthouse deve permitir que os dados históricos sejam ocultados ou, pelo menos, limitados.

### *3.4.4. Apresentação*

A representação visual dos dados desempenha um papel importante neste trabalho, já que nosso foco é apresentar informações relevantes para os usuários. Nesta seção, serão discutidas algumas maneiras de visualizar os dados previamente selecionados. As questões de escalabilidade, sobrecarga de informações e interrupções no trabalho dos usuários são levadas em consideração na escolha dessas representações.

#### 3.4.4.1. Diagrama de Design Emergente

O Lighthouse utiliza uma abstração denominada *Design* Emergente (do inglês, *Emerging Design*) (WESTHUIZEN et al., 2006) para representar as informações coletadas. Nessa abstração, são representadas as entidades e relacionamentos existentes no código-fonte. Essa representação é atualizada constantemente para refletir as mudanças que vão sendo introduzidas pelos integrantes da equipe. O *Design* Emergente é compartilhado por toda a equipe, permitindo que a evolução da implementação possa ser acompanhada "ao vivo" por todos.

A Figura 3.3 apresenta um exemplo de como o *Design* Emergente evolui, a partir de um código-fonte escrito em Java e utilizando uma notação baseada no diagrama de classes da UML (OMG, 2008). A Figura 3.3a apresenta o código

responsável por criar uma classe denominada *Store* e o *Design* Emergente representando essa classe.

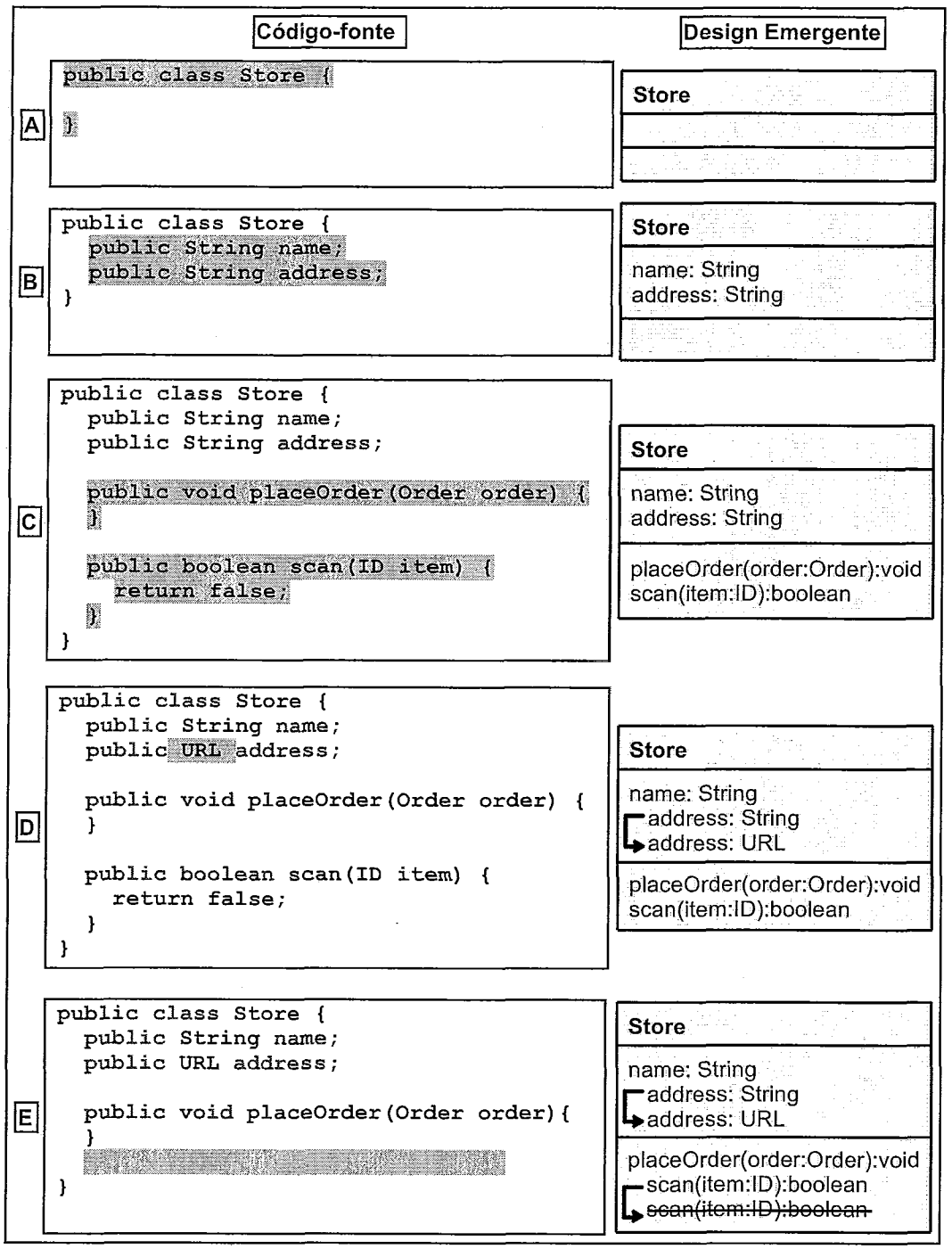



**Figura 3.3 - Evolução do Design Emergente a partir de mudanças no código-fonte (marcados em cinza): (A) adição de classe; (B) adição de atributos; (C) adição de métodos; (D) renomeação de atributo e (E) remoção de método.**

Quando são adicionados atributos à classe (Figura 3.3b), o *Design* Emergente é automaticamente atualizado para refletir essas mudanças. O mesmo ocorre quando

são adicionados métodos (Figura 3.3c). Quando algum elemento do código-fonte é alterado como, por exemplo, um atributo tem seu tipo trocado (Figura 3.3d), o *Design* Emergente não só representa a nova versão do elemento, como indica qual era a versão anterior.

Esse histórico é mostrado com o objetivo de facilitar o entendimento de como o código foi alterado. Uma seta ligando as duas versões do elemento indica que houve a mudança. No caso da remoção de um elemento (na Figura 3.3e, o método *scan*), mais uma vez é utilizada a seta indicativa de mudança, mas nesse caso o elemento aparece riscado no *Design* Emergente para indicar que ele não mais existe no código.

Os participantes da equipe podem utilizar o *Design* Emergente para se informar sobre "o que está acontecendo" no projeto no momento, através do acompanhamento das mudanças no diagrama. O histórico de evolução do projeto também pode ser acompanhado, facilitando o entendimento de como o software chegou ao estágio atual. O histórico também pode ser utilizado para se entender o que mudou no projeto durante um período de ausência, ou detectar mudanças problemáticas como, por exemplo, a duplicação de funções ou a remoção imprópria de um atributo.

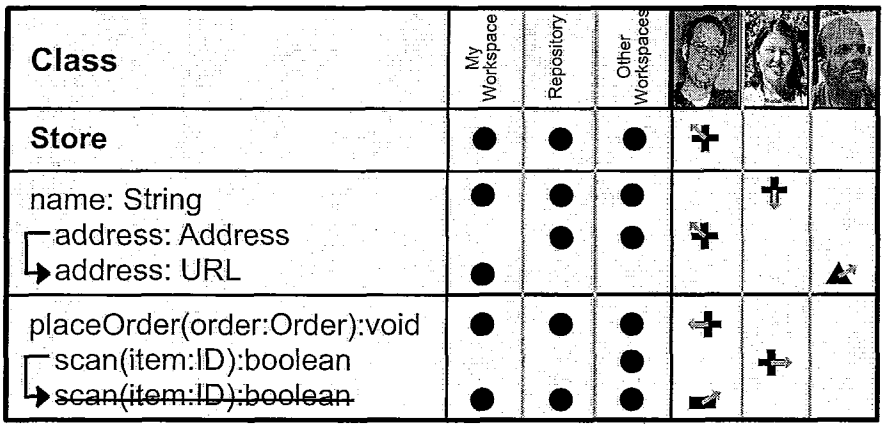



**Figura 3.4 - *Design* Emergente decorado com informações adicionais sobre cada versão dos elementos.**

No entanto, para melhor auxiliar a coordenação dentro da equipe, é possível ainda adicionar decorações a esse diagrama que dêem maiores informações sobre as modificações feitas. Alguns exemplos de decorações podem ser vistos na Figura 3.4, onde a representação da classe *Store* apresenta colunas adicionais. A três primeiras colunas extras apresentam o local onde os elementos de código-fonte (e suas diferentes versões) podem ser encontrados no momento. A primeira dessas colunas indica se aquela versão do elemento está disponível localmente na área de trabalho do usuário. Se afirmativo, um círculo preto aparece nesta coluna, ao lado do elemento em questão. A segunda coluna indica se aquela versão está no repositório do SCV, ou

seja, se já foi feito *check-in* dessa versão. A terceira coluna indica se a versão está na área de trabalho de outros integrantes da equipe.

A combinação dessas três primeiras colunas dá indicações da progressão das mudanças em relação ao repositório. Por exemplo, no caso do atributo *address* que foi modificado para o tipo *URL*, apenas a primeira coluna está marcada, indicando que esta foi uma mudança local da qual ainda não foi feito o *check-in* para o repositório e, portanto, nenhum outro usuário pôde fazer *check-out* dela ainda. O atributo *name*, por sua vez, já está no repositório e já foi feito o *check-out* dele tanto na área de trabalho local quanto na de outros usuários. Esse tipo de informação permite a distinção entre o que está sendo modificado nas áreas de trabalho do que foi realmente feito *check-in* para o repositório. Essa distinção é interessante, pois permite que os usuários saibam quais modificações são locais, e possivelmente "exploratórias", e, ao mesmo tempo, sejam avisados de novas versões mais "definitivas" já disponíveis para *check-out* no repositório.

As últimas colunas (nesse caso, três) identificam os autores das diferentes versões dos elementos, assim como o tipo de modificação que eles fizeram e quando. Cada autor é representado por sua foto e abaixo dela são marcados os elementos que sofreram modificações por ele. As modificações são representadas por símbolos que indicam qual ação foi feita sobre o elemento: o símbolo de adição significa que o autor criou o elemento, o triângulo significa que ele o editou (renomeou, mudou o tipo, etc.) e o símbolo de subtração indica que ele removeu o elemento. Somente as mudanças que impactam diretamente o design do sistema são representadas, mudanças na implementação interna de métodos são detectadas pelo Lighthouse, mas não são representadas no diagrama de *Design* Emergente.

Através dessas informações, a equipe fica ciente não só de como o sistema está evoluindo, como também quem é responsável por cada modificação feita. Essa informação é útil, por exemplo, para determinar quem contactar no caso de dúvidas sobre um determinado elemento do sistema. O Lighthouse provê as informações necessárias, mas fica a critério do indivíduo escolher quem melhor pode ajudá-lo, seja o criador do elemento, quem mais recentemente o modificou ou até, entre os usuários que já modificaram o elemento, o que se encontrar mais facilmente disponível para comunicação.

As setas sobrepostas aos símbolos de mudança indicam há quanto tempo a mudança foi feita. Mudanças recentes são representadas pela seta em pé e, ao passar do tempo, a seta vai girando no sentido horário, para indicar mudanças mais antigas. Essa indicação de tempo ajuda a entender a ordem cronológica das mudanças e a distinguir autores que estão trabalhando neste momento no artefato dos que contribuíram de alguma forma no passado.

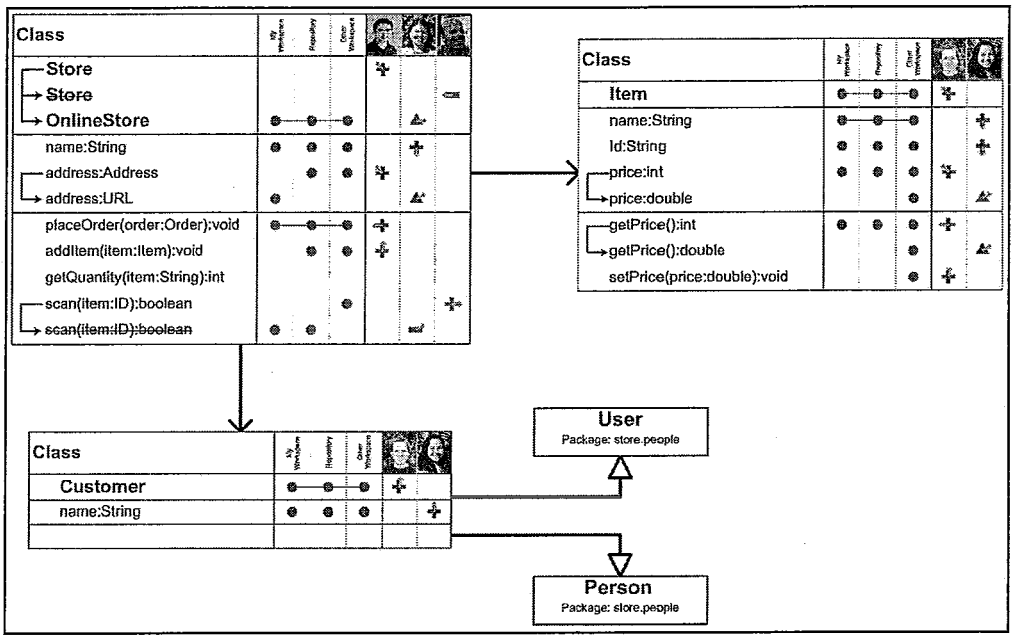



**Figura 3.5 - Diagrama do *Design* Emergente**

O *Design* Emergente mostra ainda os relacionamentos entre os elementos de código, o que ajuda a entender a estrutura do sistema. Um exemplo do diagrama com várias classes é mostrado na Figura 3.5. No diagrama, é possível ver as classes decoradas com as informações de percepção e os relacionamentos entre elas. Através dessa visão geral, é possível ter idéia de quais elementos estão sendo alterados e quem é responsável por estas alterações.

### 3.4.4.2. Mini-Diagrama

Uma maneira simples de auxiliar a navegação em diagramas grandes é ter uma versão compacta e simplificada do diagrama que provê um panorama geral dos elementos. Esse panorama provê uma maneira de se lidar com o diagrama de *Design* Emergente, de forma mais escalável. Dessa forma, como pode ser visto na Figura 3.6, este Mini-Diagrama deve estar visível próximo do *Design* Emergente.

Para melhor ajudar os usuários a navegar pelas diferentes partes do diagrama, o Mini-Diagrama deve ainda indicar explicitamente em qual parte o indivíduo se encontra no momento. Na Figura 3.6, esse indicação é feita pelo retângulo tracejado, que representa a janela de visão (do inglês, *viewport*) do usuário. Com o auxílio do Mini-Diagrama, portanto, é possível ver na figura que existem dois elementos no diagrama que não estão sendo mostrados nessa janela de visão.

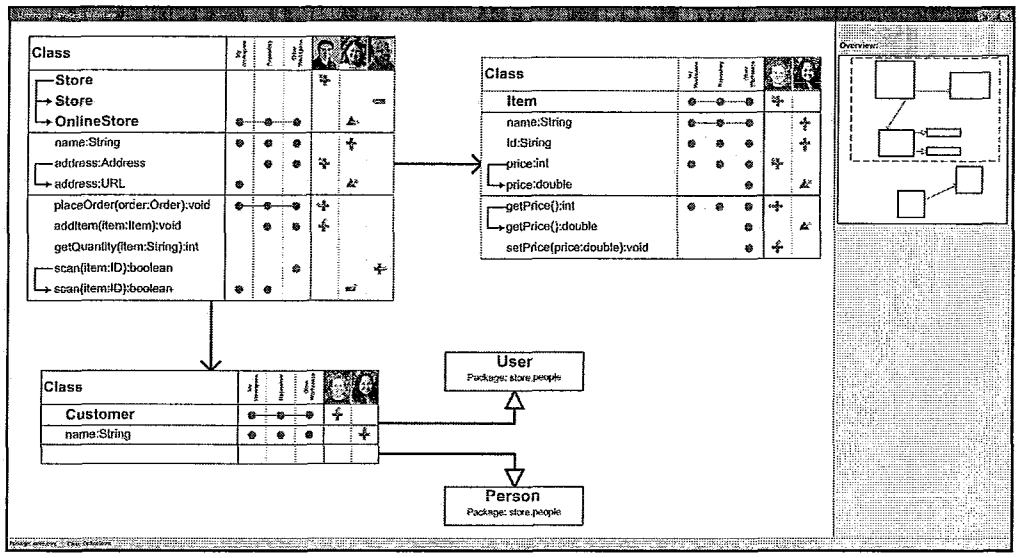



**Figura 3.6 - Mini-Diagrama no canto superior direito do *Design* Emergente.**

### 3.4.4.3. Elos

Uma maneira simples de fazer o usuário focar nas partes mais relevantes do *Design* Emergente é manter uma ligação entre a janela de visão atual do diagrama e os artefatos de *software* sendo editados no ambiente de desenvolvimento. A idéia é que o diagrama mude a janela de visão automaticamente de acordo com elementos que estão sendo editados ou somente visualizados, acompanhando o contexto atual do usuário. Ao abrir o código de um elemento no editor, o Lighthouse automaticamente passaria a mostrar a parte do diagrama a qual esse elemento pertence. No entanto, esse elo com o editor deve poder ser desligado, caso o usuário queira manter a sua janela de visão manualmente.

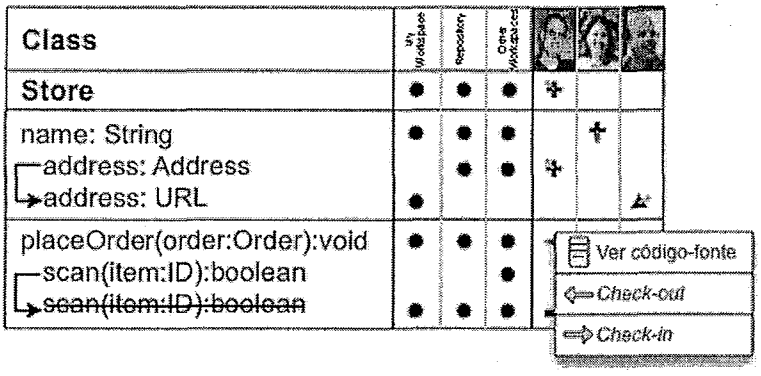



**Figura 3.7 - Alguns dos elos entre elementos do Design Emergente e o editor do ambiente de desenvolvimento.**

O elo também pode funcionar na outra direção: o usuário pode abrir, através do diagrama, um editor com o código de um elemento selecionado, quando possível[2]. Isso permitirá a fácil associação dos elementos no diagrama com código-fonte e vice-versa, o que deve facilitar ainda mais a navegação no diagrama. Além de abrir o código, o usuário também deve ter a opção de fazer *check-in* e *check-out* dos elementos representados no diagrama, quando possível[3]. Estes elos representam um primeiro passo para uma visualização mais interativa (Figura 3.7), além de também servirem como mecanismos para se lidar com a escalabilidade do diagrama de *Design* Emergente.

#### 3.4.4.4. Arranjo Automático de Elementos

Para facilitar o seu entendimento, o usuário deve poder posicionar manualmente os elementos no diagrama onde preferir. Esse posicionamento deve ser individual, ou seja, cada usuário pode escolher o melhor arranjo para si, não afetando nem sendo afetado pelo arranjo escolhido pelos demais. Além disso, esse arranjo deve ser mantido da forma mais consistente possível durante diferentes seções de trabalho, mesmo que existam mudanças nos elementos, ou que o diagrama seja fechado e reaberto posteriormente.

No entanto, como já discutido anteriormente, o diagrama de *Design* Emergente cresce rapidamente com a evolução do sistema. O arranjo manual de elementos com o tempo pode se tornar um processo lento. Esse arranjo deve poder ser feito automaticamente pelo próprio Lighthouse, como mais um mecanismo para se lidar com a escalabilidade do diagrama. Diferentes algoritmos podem ser utilizados para este fim, dependendo do tipo de organização que se procura. Seria interessante explorar algoritmos que facilitassem o entendimento do diagrama. De qualquer forma, os usuários devem continuar podendo ajustar manualmente o arranjo feito pelo Lighthouse.

É importante notar que a forma como os elementos são organizados pode influenciar o foco dos usuários no diagrama. O Lighthouse pode, portanto, utilizar o arranjo do diagrama como uma forma de destacar os elementos mais relevantes para o usuário. A análise feita pelo Lighthouse já provê a priorização dos elementos de acordo com os artefatos que estão sendo editados pelo usuário. O algoritmo de organização do diagrama pode, então, utilizar o resultado desta análise para determinar como os elementos devem ser organizados.

---

[2] O *Design* Emergente apresenta elementos que foram criados remotamente, por outros usuários, cujo código pode não estar na área de trabalho local, não permitindo sua visualização.

[3] Os elementos criados remotamente podem ainda não estar no repositório. Neste caso, não é possível efetuar operações de *check-in*/*check-out*.

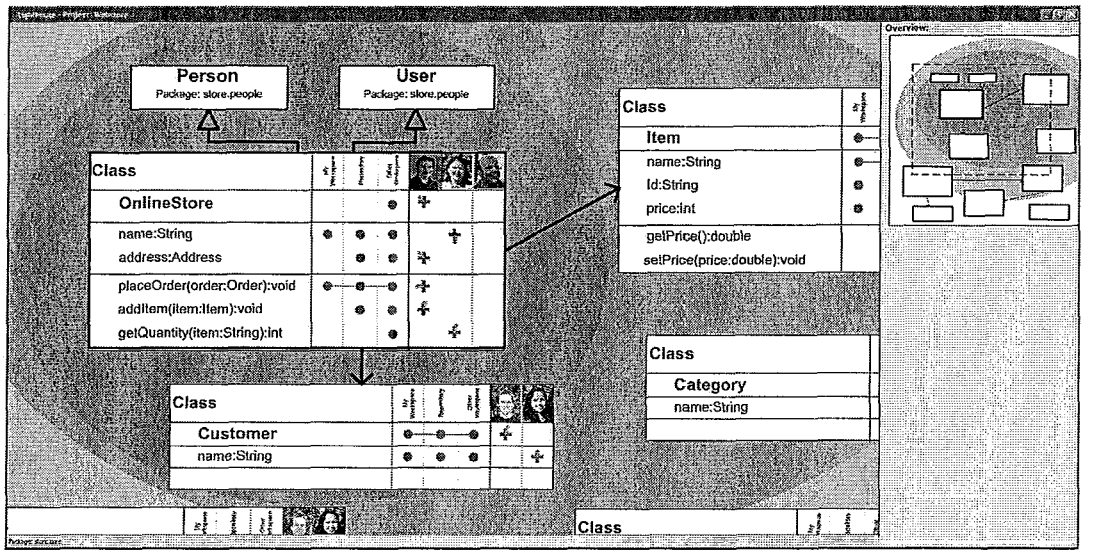



**Figura 3.8 - Esferas de influência**

A Figura 3.8 mostra um tipo de arranjo automático no diagrama de *Design* Emergente, denominado "esferas de influência". Com esse arranjo, o elemento que está sendo editado ou visualizado pelo usuário no momento é posicionado no centro do diagrama. Os elementos mais fortemente relacionados a este são então posicionados em volta do centro, chamando atenção do usuário para os elementos que irão potencialmente influenciar ou ser influenciados pelas modificações que ele está fazendo. Quanto mais próximo ao centro, mais forte é o acoplamento com o elemento central. Esse arranjo facilita o entendimento das relações entre os elementos de código, o acompanhamento dos elementos mais fortemente relacionados com o contexto de trabalho atual e a identificação dos usuários que estão trabalhando em elementos próximos. Esse arranjo é especialmente interessante para diagramas grandes (como um mecanismo para lidar com a escalabilidade) e para membros mais novos da equipe (como forma de aprendizado sobre quais as partes do sistema são mais fortemente relacionadas).

#### 3.4.4.5. Notas

Em projetos de *software*, os integrantes de uma equipe também se comunicam informalmente através do código-fonte. Por exemplo, são deixados comentários ao longo do código com explicações sobre a lógica de um algoritmo ou advertências sobre trechos críticos que devem ser alterados com cautela. Como muitas vezes a documentação do sistema é desatualizada ou inconsistente, o código também passa a ser o principal meio que se tem de entender as funcionalidades implementadas (SINGER, 1998).

O Lighthouse deve prover algo semelhante em seu diagrama, possibilitando que seus usuários deixem pequenas notas textuais para os demais, com informações

que desejam compartilhar sobre os elementos mostrados no *Design* Emergente. Essas notas podem ser utilizadas para documentar mudanças, explicar decisões de *design*, alertar sobre potenciais problemas, etc. Um exemplo de uso pode ser visto na Figura 3.9.

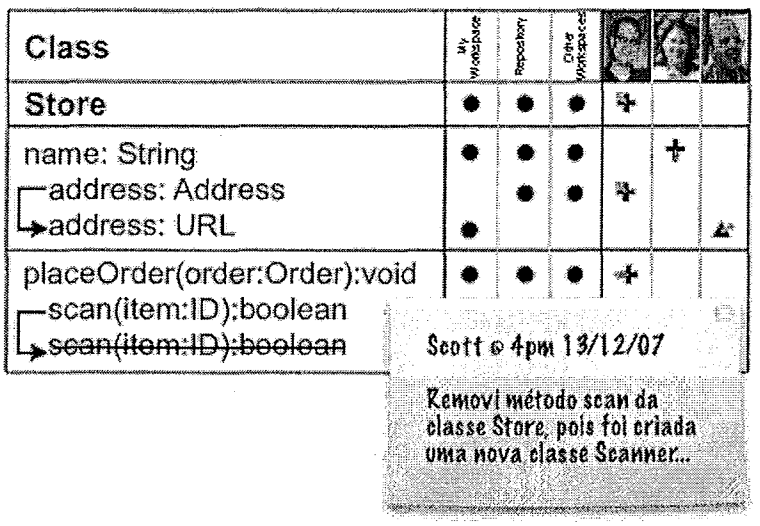



**Figura 3.9 - Notas nos elementos do diagrama de *Design* Emergente.**

### 3.4.4.6. Local de Apresentação

Como o Lighthouse é um mecanismo de percepção, este diagrama idealmente deve estar sempre visível para os integrantes da equipe. No entanto, sistemas de *software*, mesmo quando pequenos, contêm pelo menos algumas dezenas de classes, o que faz com que esse diagrama necessite de todo o espaço da tela do usuário para ser legível. Além disso, as atualizações constantes do diagrama podem se tornar uma distração constante no trabalho dos desenvolvedores.

Por isso, é recomendável o uso de um monitor adicional que possa ser dedicado ao *Design* Emergente (Figura 3.10). Dessa forma, o monitor principal continua sendo utilizado para as suas aplicações (como, por exemplo, o ambiente de desenvolvimento), enquanto o mecanismo de percepção fica no monitor periférico. Mesmo que o diagrama seja atualizado constantemente, as suas mudanças serão percebidas através deste monitor adicional, o que permite que o desenvolvedor continue seu trabalho normalmente, sem ter sua atenção desviada ou seu trabalho interrompido.

Outras configurações podem ser exploradas, como, por exemplo, a utilização de monitores grandes e de alta resolução, para visualização do diagrama inteiro por gerentes ou analistas, que precisam de uma visão mais geral da estrutura do *software*. O uso de monitores interativos, que aceitem vários usuários simultaneamente, também seria uma boa opção, pois facilitaria a navegação e a exploração do diagrama durante reuniões ou discussões de *design*.

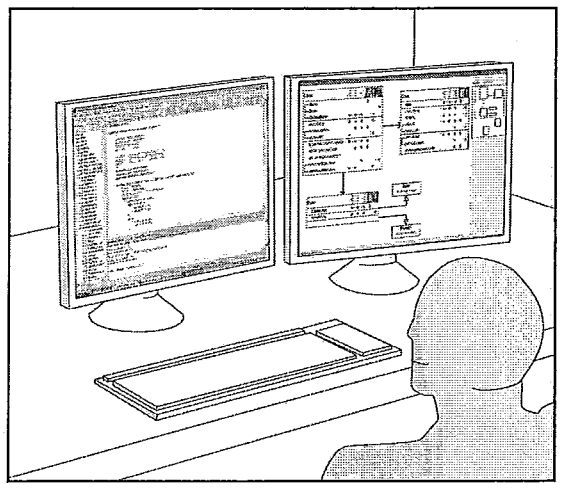


**Figura 10 - Ambiente de trabalho com dois monitores: um (principal) para o ambiente de desenvolvimento e outro (periférico) para o diagrama de *Design* Emergente.**

## 3.5. Considerações Finais

Neste capítulo, foi apresentada a abordagem proposta para um mecanismo de percepção, denominado Lighthouse, que provê informações sobre os elementos estruturais do código-fonte de um sistema de software de acordo com o contexto de trabalho e as preferências de cada usuário. Essa proposta tem como objetivo prover um modelo comum do sistema em desenvolvimento para a equipe, provendo potenciais soluções para problemas comuns ligados a mecanismos de percepção, como a baixa escalabilidade das representações das informações, a sobrecarga de informações dos seus usuários e a alta freqüência de interrupções e distrações provocadas. Para atingir este objetivo, foram propostas quatro etapas no fluxo de execução da ferramenta (Coleta, Análise, Seleção e Apresentação). Cada uma das etapas foi detalhada a partir dos requisitos propostos para implementá-las.

Retomando a classificação apresentada no Capítulo 2, podemos descrever essa abordagem a partir dos aspectos identificados:

- *Fonte de Informação*: O Lighthouse extrai informações sobre o sistema de *software* e sua evolução a partir do código-fonte do sistema, através do monitoramento da área de trabalho do usuário. Informações adicionais são coletadas do SCV, para indicar a progressão das mudanças feitas pelos desenvolvedores.
- *Análise da Informação*: As informações coletadas são combinadas para determinar a relevância destas no contexto de trabalho atual do usuário. Além disso, o próprio usuário pode configurar diferentes tipos de filtros ou designar "elementos de interesse" .
- *Representação da Informação*: O Lighthouse provê um modelo compartilhado da estrutura do sistema em desenvolvimento, denominada *Design* Emergente, que

pode ser decorado com informações de percepção. Este modelo não é só uma visualização, apresentando algumas opções de interação com o usuário.

- *Apresentação da Informação*: O Lighthouse tem integração com o ambiente de desenvolvimento, mas deve ser utilizado num monitor secundário, para facilitar sua leitura e não distrair seus usuários. Também pode ser utilizado em um monitor central compartilhado pela equipe.
- *Público-alvo*: O público-alvo principal desta abordagem é o desenvolvedor de *software*, pelo seu contato direto com o código-fonte. No entanto, analistas, gerentes, engenheiros de teste e qualidade podem também podem utilizar as informações disponíveis no Lighthouse em suas tarefas.

Esta abordagem combina aspectos já avaliados e considerados positivos em mecanismos de percepção para desenvolvedores de *software*, como a combinação de informações de percepção do repositório e da área de trabalho e a integração com o ambiente de desenvolvimento. No entanto, esta abordagem também explora oportunidades de inovação, como a utilização de uma abstração da estrutura do código-fonte, que evolui junto com a implementação, e que serve de base para diferentes tipos de informação de percepção. Esta abstração ainda serve como um modelo compartilhado pela equipe, o que pode facilitar a comunicação.

Através desta abstração, apresentamos também possíveis soluções para problemas recorrentes em mecanismos de percepção, como a escalabilidade da representação e a sobrecarga de informações, além da interrupção excessiva da tarefas principais de seus usuários. Para minimizar estes problemas, a abordagem proposta provê diversas opções para a seleção automática e manual das informações mais relevantes para o contexto de trabalho atual do usuário.

Além disso, exploramos nessa abordagem a utilização de diferentes locais de apresentação das informações de percepção. Isso permite que existam diferentes visões simultâneas sobre um mesmo modelo, dependendo do interesse do usuário. Os desenvolvedores teriam um monitor secundário, particular, focado nas suas tarefas atuais, enquanto um gerente poderia ver a estrutura completa do sistema num monitor de maior resolução.

Sabemos, no entanto, que esta abordagem tem suas limitações. Para manter o diagrama de *Design* Emergente sempre atualizado e com informações relevantes para a equipe, é necessário que todos os desenvolvedores estejam dispostos a divulgar suas atividades em relação ao código-fonte. O Lighthouse não prevê medidas para proteção da privacidade dos seus usuários, como, por exemplo, faz a ferramenta Tukan (SCHÜMMER, 2001). No Tukan, os usuários só têm acesso aos tipos de informações sobre os demais que eles mesmos aceitam divulgar. Enquanto essa abordagem parece ser justa, muitas informações relevantes podem ser perdidas. No

pior caso, nenhum tipo de informação é compartilhado pela equipe, o que inutiliza o mecanismo de percepção.

Para a utilização do Lighthouse, no entanto, é necessário que os gerentes e a equipe como um todo tenham um certo grau maturidade para lidar com as informações apresentadas. O diagrama de *Design* Emergente deve ser utilizado somente como fonte de informações de percepção para ajudar a sua coordenação e comunicação e não como uma maneira de vigiar o trabalho dos desenvolvedores. Apesar de muitas das informações apresentadas estarem disponíveis de outras maneiras (por exemplo, histórico de modificações dos artefatos no SCV), o Lighthouse facilita o acesso a estas, o que pode incentivar a sua utilização como forma de avaliação dos desenvolvedores.

Outra limitação da abordagem é o suporte voltado para linguagens de programação OO. Seria possível desenvolver uma ferramenta similar para sistemas desenvolvidos em tipos diferentes de linguagem, mas, no entanto, outro tipo de representação (não um diagrama de classes) e outras métricas de acoplamento (não as descritas na Seção 3.4.2.6) deveriam ser utilizadas, nesse caso. O importante é que a representação seja conhecida ou facilmente aprendida pelos usuários e que as métricas escolhidas levem em consideração as características específicas da linguagem.

# 4. O Protótipo Lighthouse

## 4.1. Introdução

A abordagem descrita no capítulo anterior foi implementada através de um protótipo do mecanismo de percepção Lighthouse. O protótipo foi desenvolvido de acordo com os requisitos descritos anteriormente, sendo implementado como uma extensão do ambiente de desenvolvimento Eclipse. Além disso, são utilizadas outras extensões do próprio ambiente para coletar informações necessárias da área de trabalho dos usuários e do SCV.

Neste capítulo, a implementação do protótipo é descrita inicialmente através de sua arquitetura de alto nível (Seção 4.2) e, em seguida, a partir do detalhamento de cada um de seus elementos arquiteturais (Seção 4.3). Por fim, as considerações finais deste capítulo são apresentadas (Seção 4.4).

## 4.2. Arquitetura

O protótipo do mecanismo de percepção Lighthouse foi implementado na linguagem Java, como uma extensão do ambiente de desenvolvimento Eclipse. O Eclipse é um ambiente de código livre bastante popular, principalmente entre desenvolvedores que utilizam Java. O ambiente é bastante extensível e, por isso, conta com um número grande de ferramentas desenvolvidas por terceiros. O próprio Eclipse foi desenvolvido a partir de extensões, que são adicionadas ao seu *kernel*[4].

O Lighthouse utiliza algumas das extensões do Eclipse em sua implementação para coletar as informações de que necessita sobre o sistema de *software* em desenvolvimento e sobre seus usuários. As informações são coletadas em forma de eventos, fazendo com que o Lighthouse receba notificações sobre determinadas ações do usuário no ambiente. Esses eventos são utilizados para construir o histórico de ações de cada usuário.

No entanto, os eventos coletados localmente precisam ser compartilhados com o restante da equipe, uma vez que o *Design Emergente* é criado e atualizado a partir das contribuições de todos os seus integrantes. Para isso, o Lighthouse utiliza uma arquitetura onde cada cliente (que corresponde a cada área de trabalho da equipe) coleta localmente os eventos gerados, que são enviados para armazenamento no

[4] O *kernel* é o núcleo do ambiente, que contém apenas as suas funcionalidades básicas. As demais funcionalidades são adicionadas através de extensões ao *kernel* ou à outras extensões, recursivamente.

servidor, um repositório central de informações da equipe. Desta forma, os eventos podem ser consultados pelos demais clientes.

A arquitetura de alto nível do Lighthouse está representada na Figura 4.1. Os elementos brancos na arquitetura representam os componentes implementados do cliente Lighthouse: Coletor, Processador, Replicador, Analisador, Seletor, e Visualizador. A arquitetura conta ainda com o Repositório Local e o Repositório Central para o armazenamento dos eventos, além do Repositório do SCV, onde fica o código-fonte compartilhado pela equipe. No entanto, este último não é acessado diretamente pelo Lighthouse.

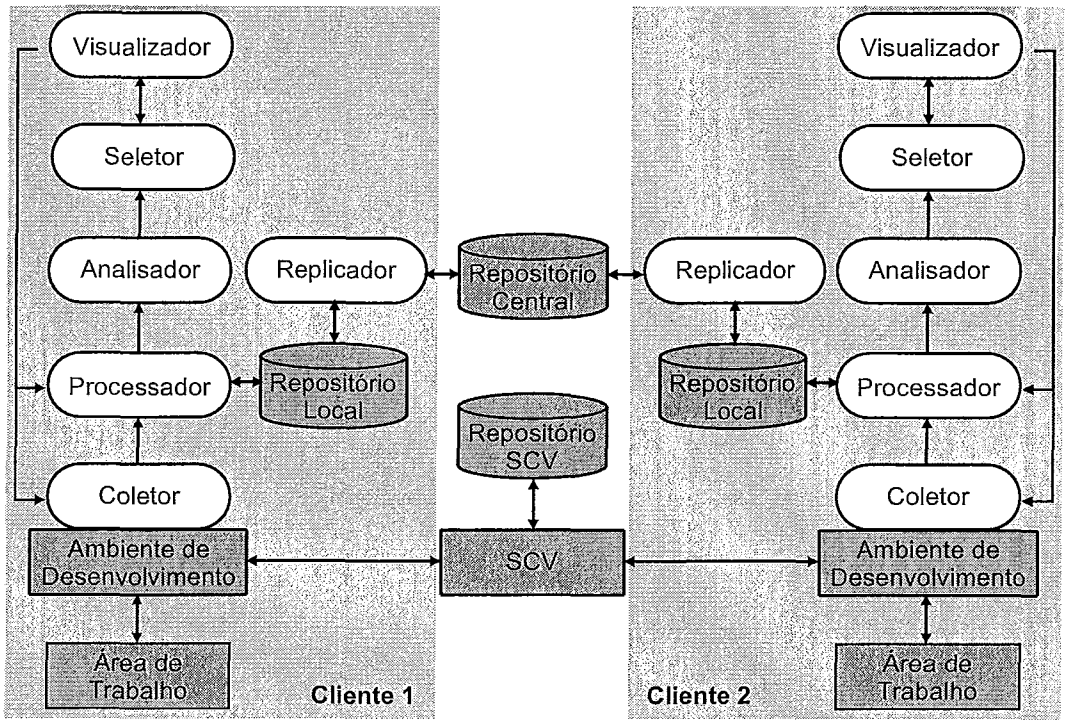



**Figura 4.1 - Arquitetura do Lighthouse**

O Coletor é o responsável por receber todos os eventos necessários para o Lighthouse. Estes eventos são gerados no Eclipse por extensões do ambiente que monitoram o código-fonte, o SCV e algumas ações do usuário no ambiente. Os eventos são, então, colocados em uma fila para análise.

O Processador, por sua vez, os retira da fila para que sejam transformados em eventos que possam ser entendidos pelo restante do Lighthouse. Essa separação permite que o Lighthouse seja menos dependente da implementação dos eventos provenientes do ambiente externo. Caso haja mudanças no ambiente, basta atualizar este processamento que o restante do cliente deverá funcionar normalmente. Depois desta tradução, os eventos são propagados tanto para o Analisador quanto para o Repositório Local.

O Analisador utiliza os eventos traduzidos para construir e atualizar o modelo interno da estrutura do código-fonte, que é utilizado como base para o *Design* Emergente. O histórico de versões de cada elemento do código, assim como as demais informações de percepção, também são armazenados neste modelo. O restante do cliente Lighthouse utiliza este modelo como fonte de informações sobre o sistema e sua equipe.

O Seletor, por sua vez, é responsável por determinar quais elementos e informações devem ser apresentados para o usuário. O Seletor leva em conta o contexto atual de trabalho do usuário na sua escolha, além de suas preferências (por exemplo, os filtros e vigias de elementos descritos no capítulo anterior). Ao receber uma notificação de mudança no modelo interno, o Seletor reavalia suas escolhas.

As informações selecionadas são então utilizadas para atualizar o diagrama de *Design* Emergente, representado pelo Visualizador na Figura 4.1. O Visualizador também pode gerar eventos a partir das interações dos usuários com a interface gráfica do protótipo. Estes eventos são recebidos e processados de forma similar aos que são gerados pelo ambiente.

Para manter a comunicação de eventos entre diferentes instâncias do Lighthouse, o Replicador de Eventos periodicamente envia para o Repositório Central os eventos gerados localmente, que foram guardados no Repositório Local. Ao mesmo tempo, os eventos gerados em outros clientes do Lighthouse, que estão guardados no Repositório Central, são trazidos para o Repositório Local. Desta forma, os repositórios são mantidos em sincronia. O Repositório Central guarda o histórico completo dos eventos, o que permite que novos usuários do Lighthouse tenham acesso a todas as informações disponíveis. Essa arquitetura também permite que um cliente que seja desconectado não perca de vez os eventos que ocorreram enquanto esteve fora. Assim que a conexão é retomada, o Replicador busca os eventos que ainda não foram recebidos ou enviados pelo cliente e re-sincroniza os repositórios. É importante notar que podem haver inúmeros clientes Lighthouse (proporcional ao número de integrantes da equipe), mas somente um repositório central para cada projeto.

## 4.3. Implementação do Protótipo

Nesta seção, detalhamos a implementação de cada um dos elementos arquiteturais descritos na seção anterior.

### *4.3.1. Coletor*

O Lighthouse utiliza algumas extensões do ambiente Eclipse para coletar as informações que necessita. Para monitorar as mudanças locais feitas no código-fonte

por um usuário, o protótipo utiliza a extensão JDT (THE ECLIPSE FOUNDATION, 2008b), que provê suporte à linguagem Java no Eclipse. Estas informações são coletadas em forma de eventos, ou seja, o Lighthouse recebe uma notificação cada vez que uma mudança é feita no código. Cada tipo de mudança corresponde a um evento específico. Por exemplo, a criação de uma nova classe Java no código gera um evento do tipo "Adição de Classe". Já a renomeação de um método é notificada com um evento de "Edição de Método". Para receber estes eventos, é necessário implementar a interface *IElementChangedListener* disponível no JDT.

Para coletar eventos sobre *check-ins* e *check-outs*, é utilizada uma outra extensão do Eclipse, o Subclipse (COLLABNET, 2008b), que permite a comunicação com o SCV Subversion. Para receber estes eventos, o protótipo do Lighthouse implementa a interface *IConsoleListener* do Subclipse. Como o Subversion armazena o código-fonte como um arquivo de texto comum, os eventos recebidos neste caso são menos refinados. São feitos somente *check-ins/check-outs* dos arquivos inteiros e, ao ser notificado, o Lighthouse precisa determinar quais elementos de software foram impactados. Um único arquivo de código-fonte pode conter diversas classes com seus métodos e atributos, que precisam ser avaliados separadamente.

O Lighthouse coleta ainda informações sobre o contexto de trabalho e as preferências do usuário, que são posteriormente utilizadas na seleção das informações prioritárias. O contexto de trabalho de um usuário é determinado pelo conjunto de artefatos (nesse caso, arquivos de código-fonte) abertos no seu editor do Eclipse. No caso de múltiplos artefatos abertos, aquele que tem o foco da janela é considerado o principal. O Lighthouse é notificado a cada vez que o usuário abre um novo artefato no editor ou troca seu foco. Além disso, as preferências do usuário em relação à seleção de informações também são coletadas, através das configurações de filtros e vigias de elementos.

### *4.3.2. Processador*

Os eventos coletados pelo Lighthouse são gerados por diferentes extensões do ambiente Eclipse. O Processador é responsável por traduzir estes eventos para o padrão interno de notificação do Lighthouse. Esta tradução permite que o protótipo seja pouco dependente do formato utilizado na descrição de cada tipo de evento. Desta forma, em caso de mudanças na implementação das extensões utilizadas, somente a coleta e o processamento destes eventos deverão sofrer atualizações.

Enquanto os eventos de modificação e de *check-in/check-out* precisam ser compartilhados com o restante da equipe, os que se referem ao contexto de trabalho e às preferências do usuário são apenas utilizados localmente. Por isso, estes últimos não são passados para o Repositório Local, sendo enviados diretamente para o

Analisador. A Figura 4.2 apresenta os principais elementos da hierarquia de eventos internos do Lighthouse.

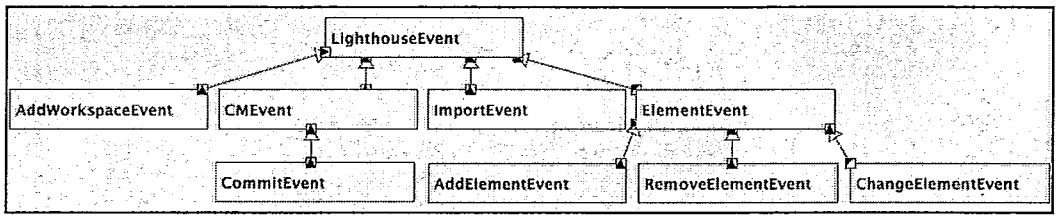



**Figura 4.2 - Hierarquia de eventos do Lighthouse**

No topo da hierarquia, se encontra o *LighthouseEvent*, que contém as informações comuns a todos os eventos do Lighthouse, incluindo um identificador, a data em que o evento foi gerado, o usuário responsável pelo evento, entre outros. Derivam deste, quatro tipos de evento:

- *AddWorkspaceEvent:* representa o registro de um novo usuário no Lighthouse;
- *CMEvent:* superclasse de eventos do SCV, como o *CommitEvent*, que notifica *check-ins*);
- *ImportEvent:* evento específico para notificar a importação de projetos pré-existentes para o Lighthouse;
- *ElementEvent;* superclasse dos eventos sobre o código-fonte em si, como de adição (*AddElement*), remoção (*RemoveElement*) e modificação (*ChangeElement*) de elementos do código.

### *4.3.3. Replicador*

O Replicador é responsável pela troca de informações entre os diferentes clientes do Lighthouse, tornando possível a construção do *Design* Emergente do sistema a partir das contribuições locais de cada usuário. Os eventos coletados localmente que devem ser compartilhados são armazenados no Repositório Local. O Replicador copia estes eventos locais para o Repositório Central, para que os demais clientes tenham acesso. O Replicador, então, copia para o Repositório Local os eventos dos demais clientes que estão armazenados no Repositório Central. Desta forma, o os repositórios Central e Local ficam sincronizados, agregando os eventos coletados de todos os clientes Lighthouse. Esta sincronização é feita periodicamente pelo Replicador, com intervalo de tempo configurável (o intervalo padrão é de 10 segundos).

Através desta arquitetura, é possível manter a consistência dos dados mesmo no caso do cliente não conseguir acessar o Repositório Central por algum motivo (por exemplo, por falta de conexão de rede ou falha do servidor). Nestas situações, os eventos locais continuam sendo armazenados no Repositório Local, mas o usuário

fica "isolado" do restante da equipe, sem conseguir transmitir nem receber eventos. O Replicador, no entanto, continua tentando executar a sincronização até que a conexão com o Repositório Central seja restabelecida. Assim que a sincronização for efetuada, os eventos que foram acumulados no Repositório Local são enviados e os novos eventos do Repositório Central são recebidos normalmente, como se simplesmente o intervalo de sincronização fosse maior. De forma análoga, novos clientes que são adicionados ao longo do projeto também conseguem obter o histórico completo de eventos automaticamente, na primeira execução da sincronização.

O Repositório Local nada mais é do que um banco de dados de eventos, que são unicamente identificados. Para facilitar o armazenamento dos eventos, foi utilizada a biblioteca *Hibernate* (RED HAT MIDDLEWARE, 2008), que faz o mapeamento do modelo orientado à objetos para o modelo relacional utilizado por bancos de dados. Esse mapeamento é feito através de anotações no código-fonte que descreve os objetos que devem ser persistidos. O *Hibernate* provê uma abstração da camada de persistência, fazendo a criação, a atualização e a busca no banco de dados transparente para o programador. O uso desta biblioteca simplificou a implementação do Replicador e ainda facilitou a evolução da hierarquia de eventos interna do Lighthouse, pois o banco de dados é automaticamente redefinido para refletir as novas estruturas dos eventos.

```
package lighthouse.com;

import java.io.Serializable;

@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public abstract class LighthouseEvent implements Serializable
{
    @EmbeddedId
    protected EventID eventID;

    @Temporal(TemporalType.TIMESTAMP)
    protected Date time;

    @Embedded
    @AttributeOverride(name = "name", column = @Column(name = "authorName"))
    protected LighthouseAuthor author;
```

**Figura 4.3 - Anotações no código-fonte que permitem o mapeamento objeto-relacional pelo *Hibernate*.**

A Figura 4.3 apresenta um trecho de código da classe *LighthouseEvent*, que é persistida automaticamente pelo *Hibernate*. As anotações no código (expressões que começam com o símbolo "@") definem como os objetos da classe devem ser armazenados no banco de dados. A anotação *@Entity*, por exemplo, determina que esta classe representa uma entidade no modelo relacional, e, portanto, deve ser

representada por uma tabela no banco de dados. Já a anotação *@Temporal* indica que o atributo deve ser armazenado como um campo de data na tabela.

O Repositório Central também é um banco de dados, que deve estar em um servidor acessível a todos os integrantes da equipe. Embora deva existir um cliente do Lighthouse para cada usuário, todos os integrantes de uma mesma equipe devem utilizar um único Repositório Central, para que os eventos sejam corretamente compartilhados.

### *4.3.4. Analisador*

O Analisador é um dos principais componentes da arquitetura do Lighthouse. Ele é responsável pela construção e atualização do modelo interno que serve de base para o *Design* Emergente. Este modelo é gerado a partir dos eventos coletados localmente ou remotamente (trazidos pelo Replicador).

O *Design* Emergente é uma representação da estrutura do sistema em desenvolvimento, que combina as contribuições de todos os integrantes da equipe. Para conseguir gerar esta representação, o Lighthouse precisa conhecer a estrutura local do sistema em cada área de trabalho da equipe. A estrutura mostrada no *Design* Emergente nada mais é que a combinação destas estruturas locais.

O Analisador armazena a estrutura do código-fonte de cada área de trabalho como a representação mostrada na Figura 4.4. Todos os elementos são derivados de *ILighthouseElement* e são unicamente identificados, para que sejam consistentemente referenciados nas diferentes áreas de trabalho. O elemento *ILighthouseLocalModel* representa a *área de trabalho* em si. Em uma área de trabalho, podem existir diferentes *projetos* (*ILighthouseProject*) que estejam sendo desenvolvidos pela equipe. Dentro de cada *projeto*, existem diferentes *pacotes* (*ILighthousePackage*), que correspondem aos diferentes diretórios utilizados para organizar os arquivos de código-fonte. Cada *pacote*, por sua vez, contém diferentes *unidades de compilação* (*ILighthouseCompilationUnit*), que correspondem aos arquivos de código-fonte em si. Uma *unidade de compilação* pode conter vários *tipos* (*ILighthouseType*), que podem ser *classes* (*ILighthouseClass*), *interfaces* (*ILighthouseInterface*) ou *enumerações* (*ILighthouseEnumeration*). Esses *tipos* podem conter *membros* (*ILighthouseMember*), que são *métodos* (*ILighthouseMethod*) ou *atributos* (*ILighthouseField*), além de *relacionamentos* (*ILighthouseRelationship*) com outros *tipos.* Um relacionamento pode representar uma *associação* (*ILighthouseAssociation*, quando um *tipo* tem como atributo o outro *tipo*), uma *generalização* (*ILighthouseGeneralization*, quando um *tipo* herda de outro *tipo*), uma *realização* (*ILighthouseRealization*, quando um *tipo* implementa uma *interface*), ou uma *dependência* (*ILighthouseDependency*, quando um *tipo* referencia um outro *tipo* de alguma outra forma: chamando um método, passando um parâmetro, instanciando uma variável do outro *tipo*, etc.).

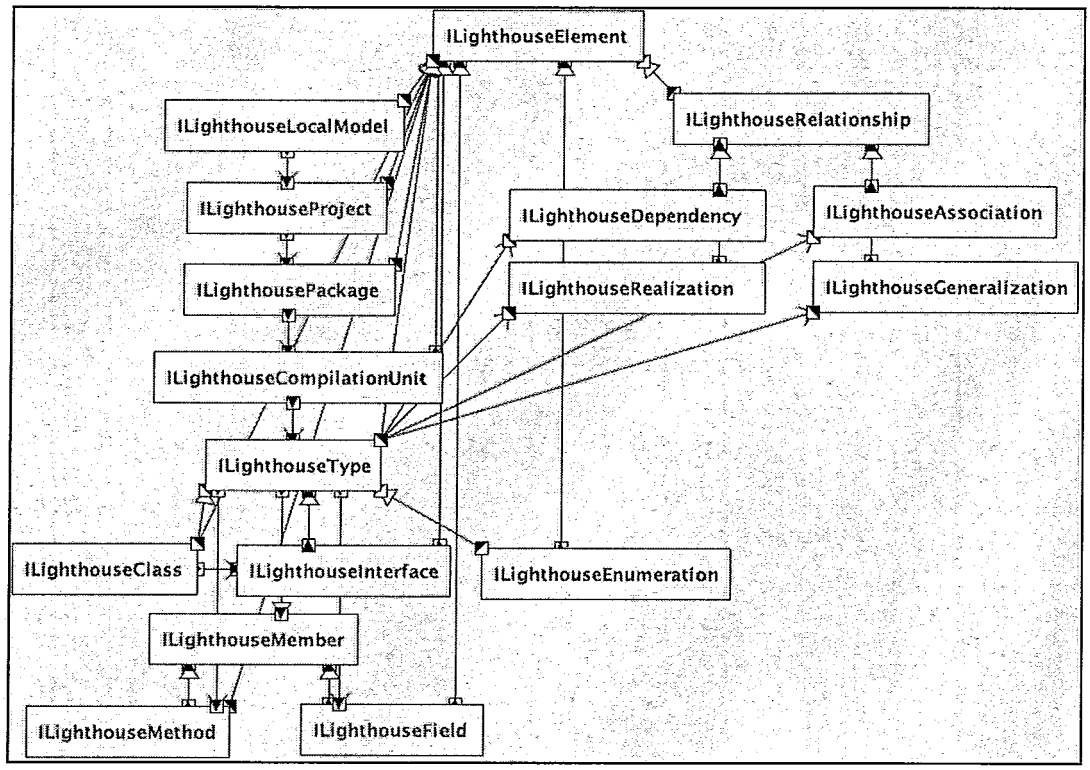



**Figura 4.4 - Estrutura interna de representação do código-fonte utilizado pelo protótipo.**

É importante ressaltar que o Analisador em cada cliente de Lighthouse armazena a estrutura do sistema da sua área de trabalho, como também a de todas as demais áreas de trabalho da equipe. Dessa forma, cada cliente conhece a estrutura do sistema da área de trabalho de todos os outros. Essa redundância é necessária para que o cliente tenha acesso local às informações dos demais, podendo funcionar desconectado dos demais mantendo a consistência das informações.

Além da estrutura atual do sistema, o *Design* Emergente também apresenta o seu histórico de evolução. O histórico de cada elemento é armazenado como um conjunto de *snapshots,* que descrevem suas diferentes versões ao longo do tempo. Cada *snapshot* contém informações adicionais como: o seu autor; a data em que foi gerado; e sua progressão em relação ao SCV.

O Analisador também é responsável pelo cálculo e o armazenamento das métricas de acoplamento descritas no Capítulo 3. Estas métricas são utilizadas para determinar quais elementos do código são mais fortemente relacionados com o contexto atual de trabalho do usuário. O valor das métricas é calculado para cada par de *tipos* do sistema e armazenadas em uma matriz, que é atualizada quando os *tipos* sofrem modificações.

O protótipo do Lighthouse utiliza as métricas escolhidas no estudo de BRIAND et al. (1999). No entanto, a sua implementação permite a incorporação de outras métricas para se determinar o acoplamento entre os *tipos* que compõem o sistema.

Para isso, a interface *ICouplingMetric* deve ser implementada, provendo o método *areCoupled(Type t1, Type t2)*, que determina se dois tipos *t1* e *t2* são acoplados ou não. A Figura 4.5 mostra a hierarquia das métricas no protótipo do Lighthouse, com a interface *ICouplingMetric* no topo. A classe *SymmetricCouplingMetric* foi implementada como superclasse para métricas simétricas de acoplamento, ou seja, para aquelas que consideram que se um tipo *t1* é acoplado a *t2*, então *t2* também é considerado acoplado a *t1*. As três métricas implementadas, CBO', INAG e PIM, são simétricas.

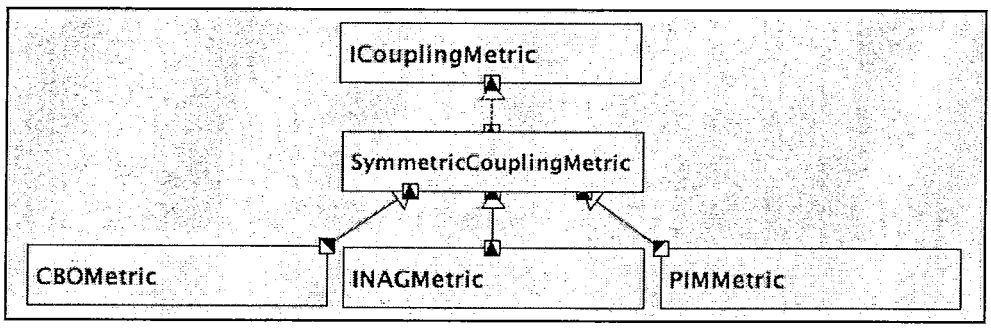



**Figura 4.5 - Hierarquia das métricas implementadas no Lighthouse.**

### *4.3.5. Seletor*

O Seletor é responsável por determinar quais elementos do sistema e informações de percepção disponíveis devem ser apresentadas para o usuário. Existem diferentes tipos de seleção que podem ser feitas. O Seletor é capaz de inferir quais elementos devem ser mostrados em um determinado momento a partir das métricas de acoplamento providas pelo Analisador. Neste caso, são escolhidos os elementos que fazem parte do contexto de trabalho atual do usuário e os que são mais fortemente acoplados à eles.

O usuário pode preferir, no entanto, selecionar os elementos de acordo com outros critérios. Para isso, foram implementados os filtros e os vigias de elementos, descritos na Seção 3.4.3. Os filtros determinam critérios configuráveis para a seleção de elementos ou informações do sistema. Já os elementos "vigiados" são sempre adicionados à seleção de elementos, mesmo que sejam utilizados filtros, pois o usuário indicou interesse especial por eles.

Foram implementados os filtros descritos no Capítulo 3. No entanto, novos filtros podem ser facilmente incorporados ao Seletor. Para isto, basta implementar a interface *IDiagramFilter* definida no Lighthouse, provendo o método *filter()* que avalia cada elemento do diagrama para determinar se este deve ser mostrado ou não. Os filtros devem ser adicionados à classe *FilterManager* que determina quais filtros estão ativos ou não, dependendo das configurações do usuário.

A Figura 4.6 ilustra a hierarquia de filtros implementados no protótipo do Lighthouse. A interface *IDiagramFilter* se encontra no topo desta hierarquia, que

também conta como a interface *IEmergingDesignFilter* que deve ser implementada por todos os filtros do diagrama de *Design* Emergente. A classe *AbstractFilter* contém as funcionalidades básicas de um filtro, como o acesso ao modelo interno de representação do código-fonte no Lighthouse, e, por isso, é superclasse de todos os filtros implementados.

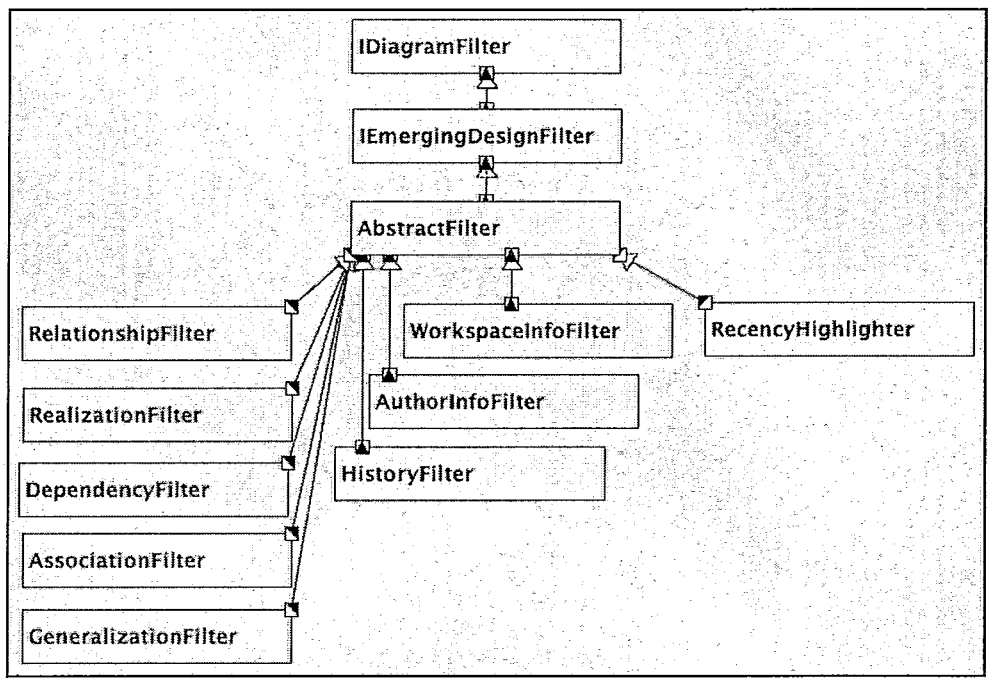



**Figura 4.6 - Hierarquia de filtros implementados no Lighthouse.**

### *4.3.6. Visualizador*

O Visualizador é um componente muito importante na arquitetura do Lighthouse, pois é o responsável pela interface com o usuário. O seu principal elemento é o diagrama de *Design* Emergente, que apresenta as informações selecionadas pelos componentes anteriores para os usuários. A implementação deste diagrama é discutida em seguida, na Seção 4.4.5.1. Outro aspecto importante da implementação do Visualizador foram os algoritmos de arranjo automático do diagrama, que são apresentados na Seção 4.4.5.2.

#### 4.3.6.1. Diagrama *de Design* Emergente

Para a implementação do diagrama de *Design* Emergente, foi utilizada a biblioteca gráfica BNA (2008a), também desenvolvida como uma extensão do ambiente Eclipse em Java e de código-fonte aberto. A biblioteca provê uma infra-estrutura para o desenvolvimento de diagramas, com elementos gráficos básicos como caixas e conectores, além de suporte a funcionalidades como movimento e redimensionamento de elementos, diferentes níveis de *zoom* e barras de rolagem para

diagramas grandes. A biblioteca BNA é extremamente flexível e extensível, suportando o desenvolvimento de novos elementos gráficos, assim como novos comportamentos e funcionalidades.

O diagrama de *Design* Emergente implementado com a biblioteca pode ser visto na Figura 4.7. É possível perceber algumas diferenças entre o diagrama do protótipo e o que foi proposto anteriormente no Capítulo 3. Por exemplo, as setas que ligavam diferentes versões de um mesmo elemento foram substituídas, por limitações de escalabilidade desta representação. Como a quantidade de modificações a serem representadas no diagrama é bastante grande, principalmente no início de um projeto, o uso de muitas destas setas dificultaria o entendimento do diagrama. A representação de diferentes versões dos elementos passou a se dar através de um esquema simples de tons de cinza: a versão atual do elemento é descrita em preto, enquanto versões anteriores ficavam em cinza. Desta forma, a versão atual de cada elemento também ganha maior destaque. Elementos removidos, também, são apresentados em cinza e seus nomes aparecem entre parêntesis, ao invés de riscados, para facilitar a leitura.

Outra diferença é o uso de ícones, ao invés de texto, para indicar as colunas com informações sobre o local onde o elemento se encontra no momento (na área local de trabalho, no repositório e em outros locais de trabalho). Essa mudança também foi feita com objetivo de melhorar a escalabilidade da representação. Os ícones podem ser vistos na Figura 4.7.

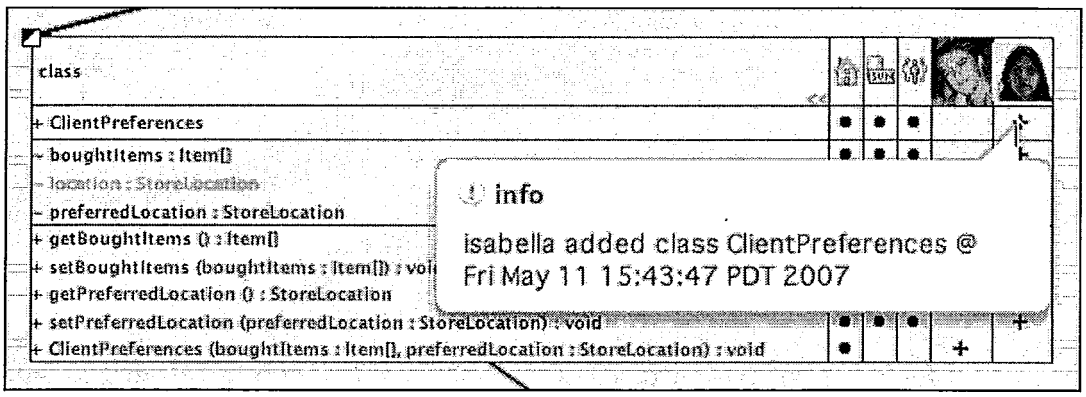



**Figura 4.7 - Diferenças de representação na implementação do diagrama de Design Emergente: texto em tons de cinza para indicar versões anteriores; ícones ao invés de texto no cabeçalho das colunas; e utilização de dicas para indicar as datas de modificação dos elementos.**

A última mudança foi a remoção das setas sobrepostas ao ícones de adição, remoção e edição na colunas de autores, que representavam há quanto tempo cada uma destas ações foi feita. Estas setas eram de difícil leitura pelo seu tamanho reduzido e, por isso, foram substituídas por dicas que apresentam textualmente a data e a hora correspondentes às ações. Um exemplo destas dicas pode ser visto na Figura 4.7.

#### 4.3.6.2. Arranjo Automático de Elementos

A biblioteca BNA provê somente um suporte básico para o arranjo automático dos elementos no diagrama. Para facilitar a utilização de algoritmos mais complexos de arranjo do diagrama, foi utilizada a biblioteca JUNG (2008). Esta biblioteca, também desenvolvida em Java e de código-livre, provê algoritmos para o arranjo de grafos, árvores e outras estruturas similares. Como o diagrama de *Design* Emergente é formado por nós (*caixas*) e arestas (*conectores*), os algoritmos providos pela biblioteca puderam ser utilizados nesta implementação.

Dentre os algoritmos disponíveis na biblioteca, foram escolhidos dois para o arranjo automático do diagrama de *Design* Emergente: o algoritmo FR (FRUCHTERMAN e REINGOLD, 1991) e o Radial. Exemplos de arranjos feitos por estes algoritmos podem ser vistos na Figura 4.8. O algoritmo FR é utilizado para distribuir os elementos uniformemente pelo diagrama, minimizando o cruzamento entre arestas e mantendo o tamanho destas parecido. Já o algoritmo Radial é utilizado como base para a implementação das "esferas de influência" descritas no Capítulo 3. O algoritmo posiciona as raízes da estrutura no centro do diagrama e os demais vértices são posicionados em volta deste, em círculos concêntricos. No protótipo, são levadas em conta as métricas de acoplamento calculadas pelo componente Analisador na execução do algoritmo.

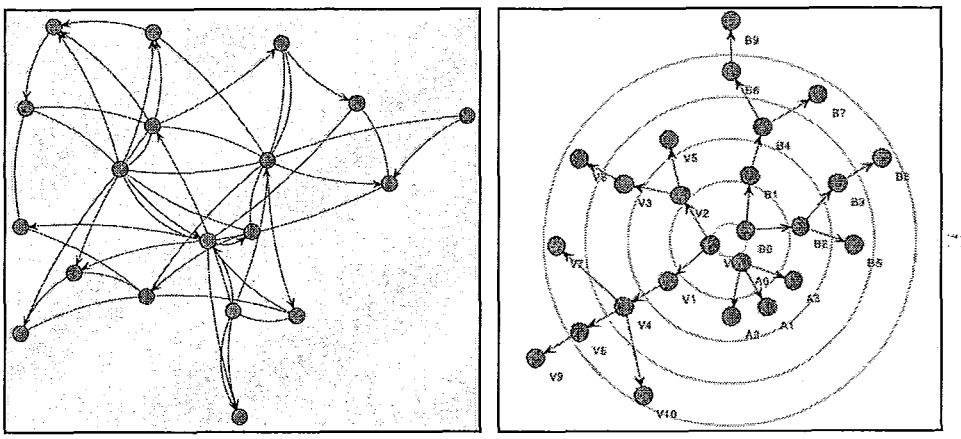


**Figura 4.8 - Arranjos automáticos utilizados pelo protótipo: FR (à esquerda) e Radial (à direita) (JUNG, 2008).**

Os algoritmos providos pela biblioteca JUNG têm uma limitação de não considerar o tamanho dos nós da estrutura no cálculo do seu posicionamento. Isso se deve ao fato da biblioteca ter como foco o arranjo de grafos e estruturas similares, cujos nós geralmente têm tamanho pequeno e constante. Nesses casos, o algoritmo pode desconsiderar o tamanho e o formato dos nós. No entanto, no diagrama de *Design* Emergente, os elementos têm tamanhos maiores e bastante variados. Nesse

caso, os algoritmos do JUNG posicionavam os elementos muito próximos uns dos outros, muitas vezes até sobrepostos.

Para solucionar este problema, após a execução de algoritmos de arranjo do JUNG, é possível executar um outro algoritmo para eliminação da sobreposição dos elementos. Esse algoritmo, desenvolvido por DWYER et. al (2005), determina quais elementos em um diagrama estão sobrepostos e os move minimamente para que não mais se toquem, tentando manter suas posições relativas aos demais elementos (Figura 4.9). O algoritmo foi escolhido exatamente por quase não interferir com o arranjo original do diagrama, preservando, neste caso, o resultado dos algoritmos providos pela biblioteca JUNG. Uma implementação em Java do algoritmo está disponível na página de um dos autores (DWYER, 2008) e é utilizada no protótipo do Lighthouse.

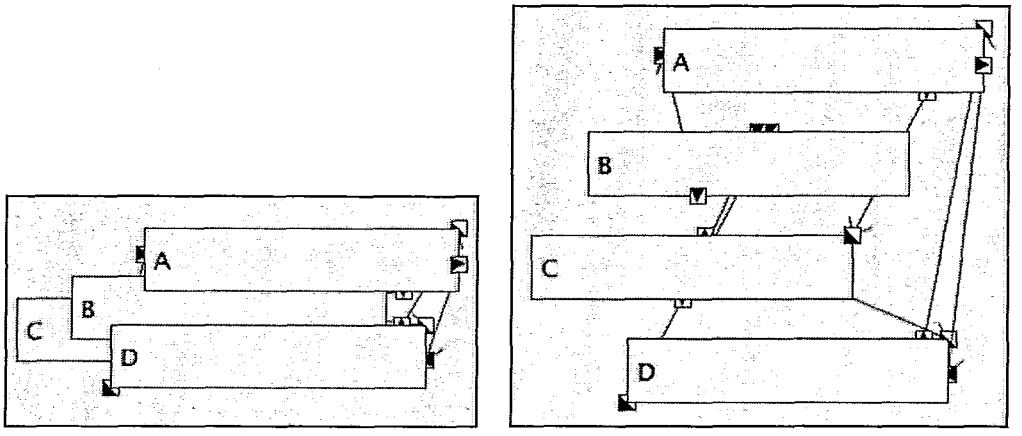



**Figura 4.9 - Exemplo do algoritmo de solução de sobreposição de elementos: à esquerda, elementos sobrepostos, à direita, os elementos re-arranjados pelo algoritmo.**

Foi utilizado ainda um último algoritmo no arranjo automático do diagrama de *Design* Emergente, mas dessa vez com foco nas arestas e não nos seus nós. Como a quantidade de relacionamentos entre os elementos do sistema apresentados é bastante grande, é desejável que se posicione os relacionamentos de forma a facilitar o entendimento do diagrama. O algoritmo utilizado tem como motivação posicionar os relacionamentos de maneira que representem o menor caminho entre os nós conectados, priorizando o uso de linhas retas, quando possível (Figura 4.10).

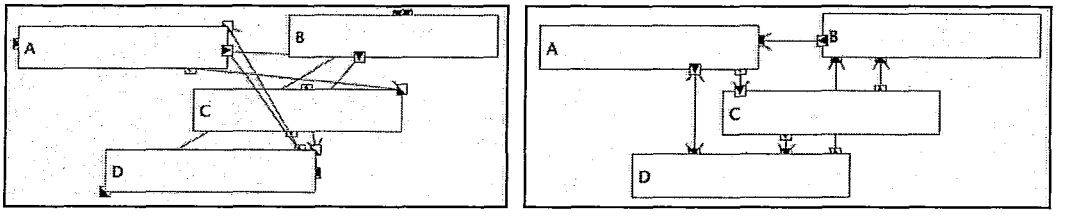


**Figura 4.10 - Exemplo do algoritmo arranjo de relacionamentos: à esquerda, elementos com relacionamentos desorganizados, à direita, os relacionamentos re-arranjados pelo algoritmo.**

Outros algoritmos de arranjo automático (da biblioteca JUNG ou não) podem ser adicionados ao protótipo. Para isso, é necessário implementar a interface *IDiagramLayout* definida no protótipo, provendo o método *layout()* que é chamado quando o usuário escolhe aplicar aquele arranjo no diagrama. Além disso, o novo arranjo deve ser adicionado à classe *LayoutManager*, que armazena todos os arranjos disponíveis e os disponibiliza através de um menu na interface de usuário do Lighthouse.

A Figura 4.11 apresenta a hierarquia de arranjos automáticos implementados no protótipo do Lighthouse. A interface *IDiagramLayout*, como já dito, se encontra no topo desta hierarquia. A interface *IEmergingDesignLayout* deve ser implementada por todos os filtros para elementos do diagrama de *Design* Emergente. A classe *AbstractLayout* contém as funcionalidades comuns aos arranjos e é superclasse de todos os filtros implementados. A classe *JUNGDiagramLayout* implementa métodos comuns aos algoritmos da biblioteca JUNG. Os algoritmos de solução de sobreposição e de arranjo de relacionamentos descritos nesta seção são implementados pelas classe *OverlapSolver* e *RelationshipSolver*, respectivamente.

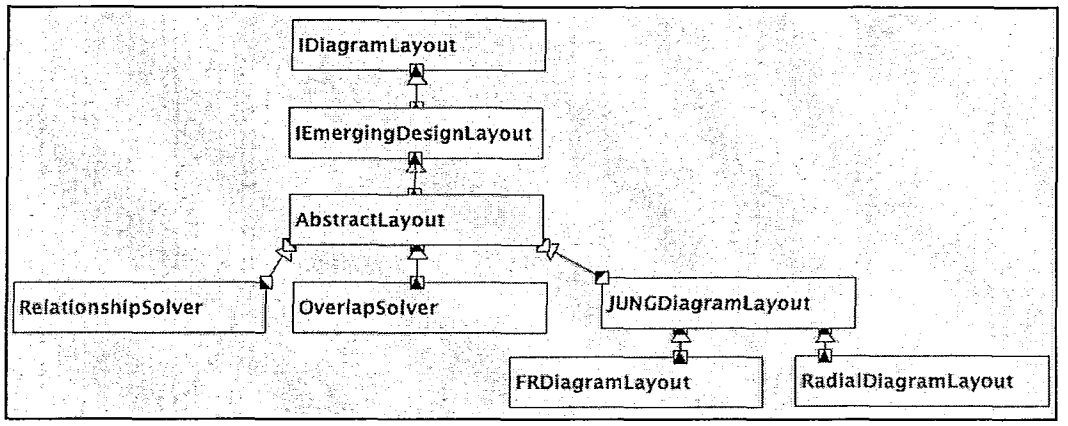



**Figura 4.11 - Hierarquia de arranjos automáticos implementados no Lighthouse.**

## 4.4. Utilização do Protótipo

Nesta seção, é demonstrada a utilização do protótipo, desde a sua configuração até a escolha do seu local de apresentação.

### *4.4.1. Instalação*

Como visto anteriormente, o protótipo foi desenvolvido como uma extensão do ambiente Eclipse e utiliza outras extensões na sua implementação. Para executar o

protótipo é necessário, portanto, instalar o ambiente na área de trabalho de cada usuário. O Eclipse pode ser baixado na sua página (THE ECLIPSE FOUNDATION, 2008a) e a sua instalação é simples, bastando descompactar o arquivo baixado em qualquer diretório do sistema de arquivos.

Depois de instalar o ambiente, é necessário também instalar as extensões necessárias para o Lighthouse. Extensões do Eclipse, geralmente, podem ser baixadas e instaladas diretamente do próprio ambiente, através dos chamados "sítios de atualização", que nada mais são do que endereços que indicam ao ambiente onde encontrar as novas extensões (localmente ou pela Internet). As extensões Subclipse (COLLABNET, 2008c) e BNA (2008b) podem ser instaladas desta forma. Instruções detalhadas para a instalação das extensões podem ser encontradas na página de Ajuda do Eclipse (THE ECLIPSE FOUNDATION, 2008c).

Com todas as extensões necessárias instaladas, deve-se instalar, então, o próprio Lighthouse. Para isso, basta copiar o arquivo compactado do protótipo para o diretório *plugins* no local de instalação do ambiente Eclipse. Ao executar o ambiente, o Lighthouse e as outras extensões instaladas são carregadas automaticamente no Eclipse.

O último passo da instalação é a criação do banco de dados que irá servir de Repositório Central para a equipe. Atualmente, o Lighthouse suporta bancos de dados MySQL (2008) e PostgreSQL(2008), por serem ambos de código aberto e de fácil instalação. O protótipo se encarrega de criar as tabelas que utiliza, só é necessário criar o banco de dados vazio e configurar as permissões de acesso para os integrantes da equipe. O banco precisa estar disponível para acesso remoto, para que todos consigam utilizá-lo.

### *4.4.2. Configuração*

Antes de começar a utilizar o Lighthouse, cada integrante da equipe deve configurá-lo na sua área de trabalho, através do menu de *Preferências* do ambiente Eclipse. A tela de configuração do Lighthouse pode ser vista na Figura 4.12. Os primeiros campos da tela dizem respeito ao Repositório Central, sendo necessário indicar as informações de acesso ao banco: o endereço (campo *Database URL*), nome de usuário (*User Name*) e senha (*Password*). O endereço deve ser o mesmo para toda a equipe. Para que essas configurações tenham efeito, é necessário reiniciar o ambiente Eclipse.

Em seguida, de volta à tela de configuração, o usuário deve se identificar, para que o protótipo saiba quem é o responsável por aquela área de trabalho. O usuário deve prover um identificar único dentro da sua equipe (*Author ID*) e seu nome (*Author name*). Um identificador para a área de trabalho é gerado automaticamente (*Workspace ID*). Esse identificador é utilizado para diferenciar diferentes áreas de

trabalho de um mesmo usuário como, por exemplo, no caso do usuário utilizar dois computadores diferentes, um de casa e outro no escritório, para trabalhar no projeto.

O usuário pode, ainda, definir seu *avatar*, ou seja, uma foto ou figura que vai ser utilizada para representá-lo na interface gráfica do Lighthouse. Para isso, é necessário escolher um arquivo de imagem através do campo *Image Path* da tela de configuração. O *avatar* dos usuários é compartilhado com a equipe.

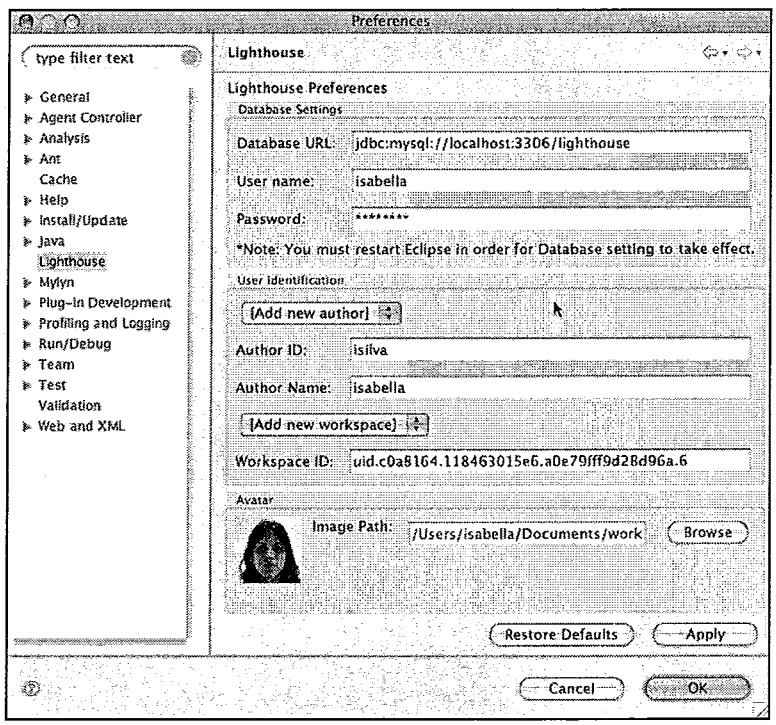



**Figura 4.12 - Janela de configuração do Lighthouse**

### *4.4.3. Execução*

A partir do momento que o protótipo foi corretamente configurado, o Lighthouse passa a monitorar a área de trabalho dos integrantes da equipe. Todo o código desenvolvido a partir deste momento passa a ser analisado pela ferramenta e apresentado no diagrama de *Design* Emergente no Eclipse (Figura 4.13). O diagrama está disponível através do menu *Window* → *Show View* → *Other...* → *Lighthouse* → *Emerging Design View*.

O protótipo do Lighthouse pode ser utilizado tanto em projetos novos, quanto naqueles já em andamento. No caso de um novo projeto, basta começar a desenvolver o código-fonte normalmente no Eclipse, que os elementos criados ou alterados são detectados e suas informações compartilhadas com o restante da equipe. Já no caso de projetos pré-existentes, é necessário primeiro fazer *check-out* de seu código-fonte para a área de trabalho do Eclipse, para que o Lighthouse consiga importar o projeto. No entanto, o histórico do projeto até aquele momento não é

atualmente recuperado pelo protótipo. Somente as modificações feitas a partir da importação serão coletadas e representadas no diagrama de *Design* Emergente. Está é uma limitação do protótipo que pode ser resolvida no futuro através da utilização do histórico de modificações do próprio SCV.

Na Figura 4.13, foram destacados alguns dos elementos da interface gráfica do protótipo. Nela, é possível ver o menu do *Design* Emergente (Figura 4.13a), que contém a opção de ligar ou desligar o Elo com o editor (Figura 4.13a1), as opções de arranjo automático (Figura 4.13a2) e de filtros (Figura 4.13a3).

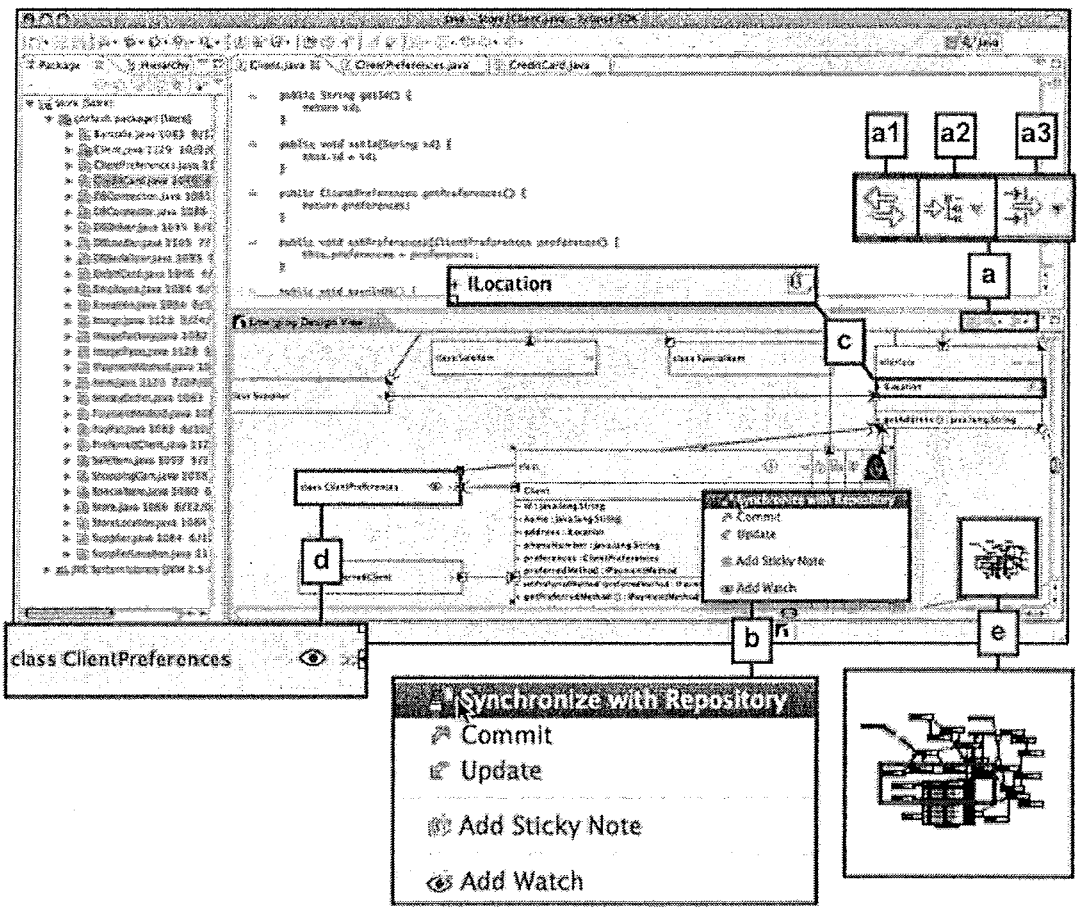



**Figura 4.13 - Protótipo do Lighthouse no ambiente Eclipse: (a) menu de opções com (a1) elo com editor, (a2) arranjo automático e (a3) filtros; (b) menu de contexto com opções do SCV, notas e vigias; (c) elemento com nota; (d) elemento com vigia; e (e) mini-diagrama.**

Ao pressionar o botão do Elo com o editor, toda vez que um arquivo de código-fonte é aberto no editor do Eclipse, a janela de visão do diagrama centraliza o elemento que o representa no *Design* Emergente. De forma análoga, ao se clicar duas vezes em um elemento do diagrama, o seu código-fonte é aberto no editor, caso este esteja disponível localmente. Para desativar o Elo com o editor, basta clicar no botão novamente.

As diferentes opções de algoritmo de arranjo automático são mostradas ao se clicar no botão mostrado na Figura 4.13a2. O menu com as opções pode ser visto na Figura 4.14 (à esquerda), contendo os algoritmos descritos previamente na Seção 4.3.6, neste capítulo. Analogamente, as opções de filtro podem ser vistas com um clique no botão mostrado na Figura 4.13a3, e seu menu é apresentado na Figura 4.14 (à direita), contendo os filtros descritos na Seção 4.3.5.

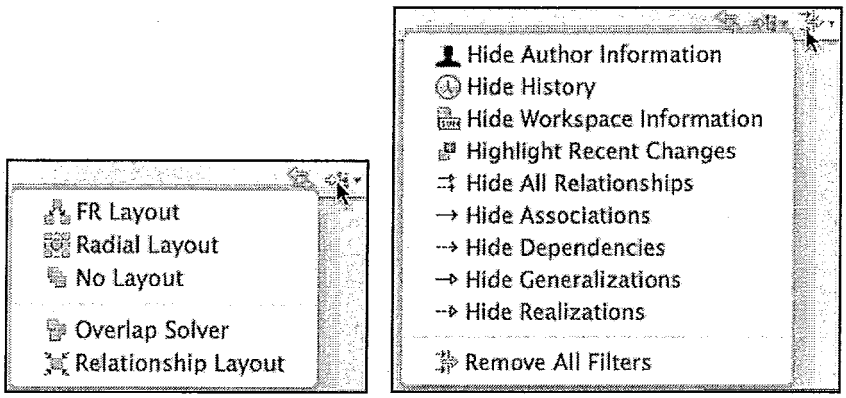



**Figura 4.14 - Menus de opção de arranjo automático (à esquerda) e de filtros (à direita).**

Algumas funcionalidades, no entanto, só são acessíveis através do menu de contexto aberto ao se clicar com botão direito em um dos elementos do digrama (Figura 4.13b). Através deste menu, é possível acessar as opções de interação com o SCV (sincronizar com o repositório[5], fazer *check-in* e *check-out*), além de adicionar notas e vigias ao elemento. Algumas opções de interação com o SCV não se aplicam a elementos cujo código-fonte não estão na área local do usuário ou no repositório.

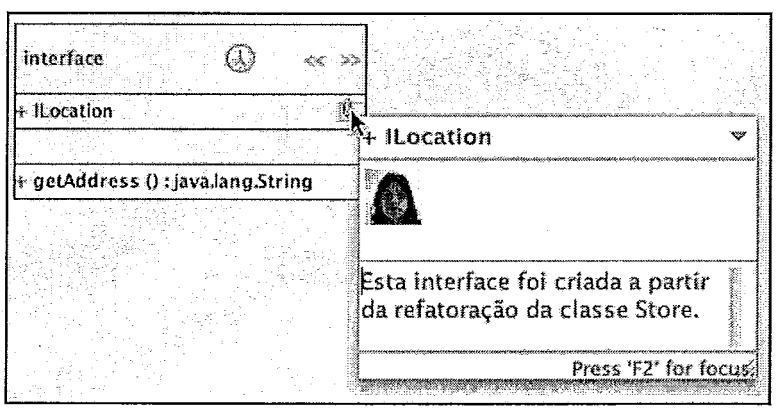



**Figura 4.15 - Exemplo de nota deixada em elemento do diagrama de *Design* Emergente.**

A Figura 4.13c mostra um elemento ao qual foi adicionado uma nota (representada pelo ícone amarelo em forma de nota). O conteúdo da nota pode ser lido ao posicionar o cursor em cima do ícone (Figura 4.15). Já a Figura 4.13d,

[5] A sincronização consiste em mostrar em um painel à parte quais elementos de que pode ser feito *check-in* e *check-out*. Essa funcionalidade é provida pela extensão Subclipse.

apresenta um elemento que está sendo vigiado pelo usuário (marcado com o ícone de olho). Esse elemento fica sempre visível para o usuário no diagrama, mesmo que ele esteja utilizando algum tipo de filtro.

O Mini-Diagrama proposto no capítulo anterior pode ser visto na Figura 4.13e, mostrando, através do retângulo azul, a janela de visão do usuário em relação ao diagrama completo. O retângulo pode ser arrastado para outras partes do Mini-Diagrama, fazendo com que a janela de visão do diagrama de *Design* Emergente seja trocada. Isso permite que o usuário navegue pelo diagrama com mais facilidade.

### *4.4.4. Local de apresentação*

O Lighthouse foi desenvolvido como um mecanismo de percepção periférico, ou seja, as informações apresentadas por ele devem estar facilmente acessíveis para consulta, mas não devem ser o foco principal de atenção dos seus usuários. Exploramos diferentes locais de apresentação para o protótipo implementado, desde monitores adicionais na mesa de trabalho dos usuários, quanto telas maiores que podem ser compartilhadas por uma equipe em sua sala.

A Figura 4.16, demonstra uma configuração com três monitores para um usuário do Lighthouse. No monitor mais à esquerda, está sendo utilizado o navegador e um *software* de troca de mensagens instantâneas; no monitor central está o ambiente de desenvolvimento Eclipse, que é o foco central de atenção do usuário; e à direita, o monitor com o diagrama de *Design* Emergente do Lighthouse. Esta configuração permite que o usuário consiga visualizar ao mesmo tempo as ferramentas que necessita para seu trabalho, evitando uma constante troca de contexto entre as janelas.

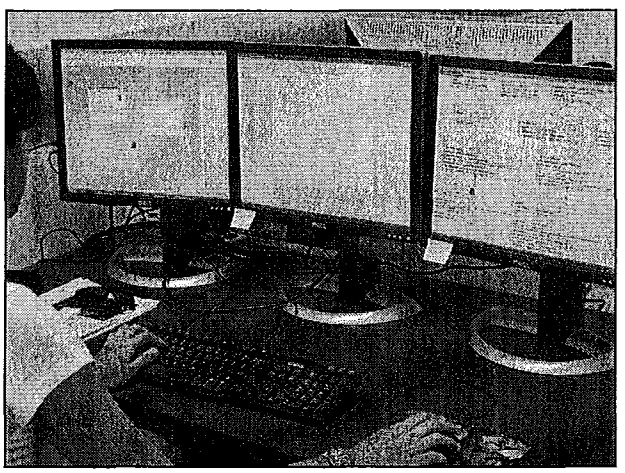


**Figura 4.16 - Configuração com três monitores: um com navegador e mensageiro instantâneo (à esquerda), outro com ambiente de desenvolvimento (no centro) e, o terceiro, com o Lighthouse (à direita).**

Outra opção é mostrada na Figura 4.17, com o uso de múltiplos monitores, mostrando diferentes visualizações de *software*. O diagrama de *Design* Emergente pode ser visto nos monitores inferiores do centro e da direita. Essa configuração pode ser utilizada como um painel de controle do projeto (SILVA et al., 2007), podendo ser compartilhada pela equipe num local acessível a todos. Esta opção, também, é interessante do ponto de vista de gerentes que precisam ter uma visão geral de seus projetos.

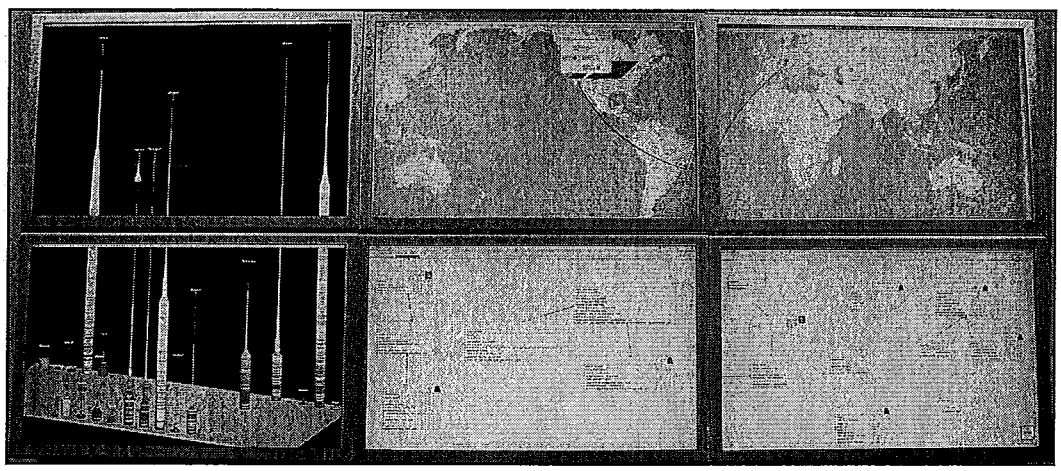


**Figura 4.17 - Configuração de seis monitores apresentando o *Design* Emergente (nos monitores inferiores do centro e da direita) em conjunto com outras visualizações de *software*.**

## 4.5. Considerações Finais

Neste capítulo, foi descrita a implementação do protótipo Lighthouse, com base na abordagem descrita no Capítulo 3. Inicialmente, apresentamos a arquitetura do protótipo que inclui seis componentes: Coletor, Processador, Replicador, Analisador, Seletor e Visualizador. Em seguida, foi detalhada a implementação de cada um desses componentes e foi descrita a utilização do protótipo.

Este protótipo foi desenvolvido como uma extensão do ambiente Eclipse, utilizando a linguagem Java, que permite a sua execução em diversas plataformas. O seu código-fonte é livre, e está disponível em um repositório aberto (LIGHTHOUSE, 2008). Instruções detalhadas de instalação e configuração podem ser encontradas na pasta *docs* do mesmo repositório.

O protótipo implementado, no entanto, tem algumas limitações. Como já dito anteriormente, apesar do protótipo poder ser utilizado em projetos pré-existentes, não é possível recuperar o seu histórico até aquele momento. Mesmo que o projeto esteja sob controle de versão, o Lighthouse atualmente só consegue construir o histórico de mudanças feitas após o projeto ser importado na ferramenta. Isso ocorre porque, atualmente, o Lighthouse não faz nenhum tipo de engenharia reversa para coletar as

informações sobre as mudanças feitas no código-fonte. Essas informações são coletadas somente através dos eventos providos pelo Eclipse. O SCV, nesse caso, é utilizado apenas para determinar quais elementos ou versões dos elementos já estão no repositório. Por isso, o protótipo assume que toda a equipe está trabalhando num mesmo ramo do projeto ou a partir de uma mesma versão inicial.

A escolha de não utilizar engenharia reversa leva ainda à outra limitação: caso o código-fonte seja alterado fora do ambiente de desenvolvimento, o protótipo não é capaz de detectar as mudanças feitas. Nesse caso, o diagrama de *Design* Emergente ficaria fora de sincronia com o código-fonte. Por isso, a equipe precisa utilizar o Eclipse para todas as alterações no código do sistema ao utilizar o protótipo. Apesar dessas limitações, esta abordagem foi escolhida por sua facilidade de implementação e de manutenção e por ser mais leve, pois o próprio Eclipse se encarrega da interpretação do código-fonte.

O protótipo só pode ser utilizado em projetos desenvolvidos com a linguagem de programação Java. No entanto, é possível adaptá-lo futuramente para aumentar a diversidade de projetos suportados. O próprio ambiente Eclipse possui suporte a outras linguagens de programação. Nesse caso, o componente Coletor deveria ser alterado para receber eventos sobre a modificação de elementos de código-fonte em outras linguagens. De qualquer forma, a representação interna do projeto quanto a do diagrama de *Design* Emergente foram desenvolvidas para linguagens OO. A adaptação do protótipo para outros tipos de linguagens pode, no pior caso, envolver a modificação de todos os seus componentes.

A implementação do protótipo é dependente do ambiente Eclipse e do SCV Subversion. Equipes que utilizam outras ferramentas de desenvolvimento não podem utilizá-lo atualmente. O componente mais dependente do ambiente é o Coletor, que pode ser substituído no caso de uma adaptação do protótipo para outro ambiente. O mesmo pode ser dito no caso de utilização de outro SCV.

# 5. Avaliação

## 5.1. Introdução

O protótipo do Lighthouse, descrito no capítulo anterior, foi utilizado durante um estudo preliminar de observação. Esse estudo representa um primeiro passo para avaliar a abordagem apresentada neste trabalho. Foram observados 4 alunos de pós-graduação da COPPE-UFRJ utilizando o protótipo do Lighthouse, durante a execução de tarefas de programação, numa organização que simulava uma equipe de desenvolvimento distribuída. Apesar de restrito, o resultado deste estudo pôde ser utilizado para entender as limitações atuais do protótipo implementado e para determinar os próximos passos para a utilização do Lighthouse no contexto de um projeto de *software* real.

Este capítulo está organizado da seguinte forma: inicialmente, é apresentado o objetivo geral do estudo (Seção 5.2) e a definição do mesmo é detalhada (Seção 5.3). Em seguida, são listados os passos do procedimento de execução do estudo (Seção 5.4), apresentadas as observações feitas durante a sua execução (Seção 5.5) e a avaliação feita pelos participantes (Seção 5.6). A validade do estudo é discutida em seguida (Seção 5.7) e, por fim, as considerações finais deste capítulo são apresentadas (Seção 5.8).

## 5.2. Objetivo

Com o objetivo de caracterizar a viabilidade do uso do Lighthouse no suporte à coordenação em equipes de desenvolvedores de software, foi realizado um estudo observacional. O termo "observacional" é usado para definir estudos onde o participante realiza alguma tarefa enquanto é observado por um experimentador (SHULL et al., 2001). Este tipo de estudo tem como finalidade coletar dados sobre como determinada tarefa é realizada. Através destas observações, pode-se obter uma compreensão de como uma ferramenta ou um processo novo é utilizado. O objetivo do estudo pode ser apresentado de acordo com o modelo descrito em (WOHLIN et al., 2000), e apresentado na Tabela 5.1.

**Tabela 5.1 - Definição do objetivo do estudo**

| |
|---|
| **Analisar** o protótipo do mecanismo de percepção Lighthouse |
| **Com o propósito de** caracterizar a viabilidade do seu uso |
| **Em relação ao** suporte à coordenação de desenvolvedores de *software* |
| **Do ponto de vista do** desenvolvedor de *software* |
| **No contexto de** equipes de desenvolvimento de *software* |

É importante destacar que o estudo foi executado como uma avaliação preliminar deste trabalho. O protótipo Lighthouse foi avaliado unicamente em relação à sua viabilidade como mecanismo de percepção de grupo. Experimentos mais avançados ainda precisam ser feitos para determinar se o Lighthouse atende, na prática, a todos os objetivos apresentados no início deste trabalho.

## 5.3. Definição Do Estudo

Este estudo foi realizado num ambiente acadêmico, no Laboratório de Engenharia de Software (LENS) da COPPE/UFRJ. Para participar no estudo, os indivíduos deveriam ter algum experiência anterior com a linguagem de programação Java, com o ambiente de desenvolvimento Eclipse e com algum SCV (de preferência, o Subversion), uma vez que este conhecimento é necessário para o uso correto da ferramenta. Quatro estudantes da pós-graduação da linha de Engenharia de Software da COPPE/UFRJ se voluntariaram para participar deste estudo. Não houve compensação (monetária ou de qualquer outro tipo) para os participantes.

Com este estudo, se pretendia observar como desenvolvedores de *software* de uma equipe coordenam suas tarefas, com o suporte da ferramenta Lighthouse. Mais especificamente, o estudo teve como foco a utilização da ferramenta como suporte para a coordenação entre os integrantes de uma equipe, enquanto estes executam paralelamente suas tarefas de programação. Quatro pontos principais foram observados:

1. O uso do mecanismo Lighthouse permite a detecção de conflitos na implementação do código-fonte do *software?*
2. O diagrama de *Design* Emergente é utilizado como referência para o entendimento da estrutura do *software?*
3. O diagrama de *Design* Emergente é utilizado como referência sobre "o que está acontecendo" no projeto de *software*?
4. O diagrama de *Design* Emergente estimula e serve como referência para a comunicação dentro da equipe?

Para poder fazer essas observações, o participante deveria se encontrar no contexto de uma equipe de desenvolvimento de *software* e executar tarefas de programação. Para facilitar a imersão do participante no contexto do experimento, lhe foi informado que ele estaria integrando uma equipe pré-existente, para substituir um dos desenvolvedores, que deixou o projeto. A equipe trabalhava em um projeto de pequeno porte desenvolvido em Java, para uma loja virtual. Além dele, mais dois integrantes faziam parte atualmente da equipe, mas estes estavam localizados em outra cidade, somente disponíveis para comunicação via mensagens instantâneas de texto.

O participante recebia, então, algumas tarefas de programação que envolviam mudanças no código-fonte do projeto. Foram dadas instruções detalhadas em relação às alterações necessárias para executar as tarefas propostas. Todos os participantes utilizaram o protótipo do Lighthouse em um monitor secundário durante a execução das tarefas, além do ambiente Eclipse, do SCV Subversion e do programa mensageiro *iChat* (APPLE, 2008), no monitor principal. O Lighthouse, neste caso, proveria um suporte para que o participante entendesse a estrutura atual do projeto, assim como o seu histórico, e permitiria que ele detectasse as mudanças feitas pelos demais membros da equipe.

Na realidade, o participante estava trabalhando sozinho e os demais membros da equipe eram entidades virtuais, denominados confederados, controlados pelo experimentador. As tarefas de programação dos confederados foram simuladas através de *scripts* automatizados, que introduziam as alterações necessários ao código fonte do projeto em desenvolvimento em intervalos pré-determinados de tempo. Estas tarefas simulavam as atividades do restante da equipe, gerando oportunidades de coordenação entre os confederados e o participante. Algumas destas tarefas introduziam conflitos entre o código-fonte do participante e o do confederado. Cabia ao participante detectar e lidar com esses conflitos. O experimentador também respondia pelos confederados, quando o participante tentava se comunicar via mensagens de texto.

A introdução destes confederados teve como objetivo facilitar o controle da execução do estudo e a comparação dos resultados dos diferentes participantes. Esta organização garantiu que um determinado número de oportunidades de coordenação fossem introduzidas durante o estudo e se mantivesse constante em toda sessão. Desta forma, todos os participantes teriam uma experiência similar. A organização geral, as tarefas e o projeto utilizado durante este estudo foram fortemente baseados no experimento descrito por SARMA et al. (2007b), para avaliação do mecanismo de percepção Palantír.

## 5.4. Procedimento de Execução

Cada sessão do estudo utilizou um único participante e durou cerca de 2 horas. Inicialmente, cada participante foi informado sobre o experimento através do Formulário de Consentimento (Apêndice I). Caso concordasse em participar do experimento, o participante preenchia o Questionário de Caracterização (Apêndice II). Este questionário avalia o nível de conhecimento e experiência do participante em diferentes temas relacionados ao estudo. Esses dados foram utilizados para garantir que os participantes estavam aptos a executar o estudo e também para interpretar os resultados obtidos por cada um dos participantes. O resultado deste questionário está

resumido no Apêndice III. O preenchimento dos formulários iniciais levou cerca de 15 minutos.

Em seguida, o participante recebeu um treinamento de aproximadamente 15 minutos sobre os principais conceitos e funcionalidades do protótipo do Lighthouse. Ao final do treinamento, foi entregue o documento de Descrição Geral da Tarefa (Apêndice IV), que descreve o contexto em que ele estará inserido durante a execução das suas tarefas no estudo. A descrição detalhada das tarefas de programação que deveriam ser executadas durante o estudo estavam disponíveis como páginas HTML no navegador embutido no próprio ambiente Eclipse. A descrição das tarefas (tanto dos participantes quanto dos confederados) pode ser vista no Apêndice V. Durante a execução das tarefas, os participantes também tinham acesso livre à Internet para consultas técnicas (manuais, APIs, etc.). O código-fonte do projeto de *software* (Apêndice VI) a ser modificado já estava na área de trabalho dos participantes e também já estava compartilhado e sob controle de versão, pelo Subversion. O diagrama de *Design* Emergente do Lighthouse estava aberto no monitor secundário e apresentava o histórico de evolução do projeto até aquele ponto (Apêndice VI).

Os participantes dispunham de uma hora para executar as tarefas. Foram propostas, no total, 12 tarefas. Os participantes deveriam tentar completar a maior quantidade possível de tarefas, não sendo obrigados a terminar todas. No final do tempo previsto para as tarefas, os participantes responderam a um Questionário de Avaliação (Anexo VII), que diz respeito à experiência com as tarefas executadas, à utilização do protótipo e ao treinamento recebido. O preenchimento deste questionário durava de 20 a 30 minutos.

## 5.5. Observações

Para descrever e comparar os resultados das quatro sessões do estudo, foi construída uma linha do tempo (Figura 5.1) com os principais eventos observados durante a execução das tarefas de programação pelos participantes. No eixo Y, representamos cada um dos participantes, numerados de 1 a 4, e sobre o eixo X, temos o tempo de execução, em minutos (de 0 a 70). Diferentes símbolos foram utilizados para representar os principais eventos observados, que foram ordenados cronologicamente.

O início de cada tarefa do participante é representado por um triângulo normal, enquanto as tarefas dos confederados, por um triângulo de cabeça para baixo. A cruz representa uma tentativa de fazer *check-in* das modificações feitas no código-fonte (relativas a uma tarefa) para o repositório. Este evento marca uma mudança de foco do participante, da codificação para o compartilhamento das suas alterações com a

equipe. Com freqüência, esse evento envolvia a comparação e a combinação do código modificado localmente e o disponível no repositório. Como visto anteriormente, o participante podia contactar os confederados via mensagens de texto. Esses contatos são representados na linha do tempo como quadrados. Um dos eventos mais importantes a se observar neste estudo foi, no entanto, a detecção das alterações introduzidas pelos confederados, marcadas com X na linha do tempo. Finalmente, o fim do tempo reservado para a execução das tarefas é representado por um losango.

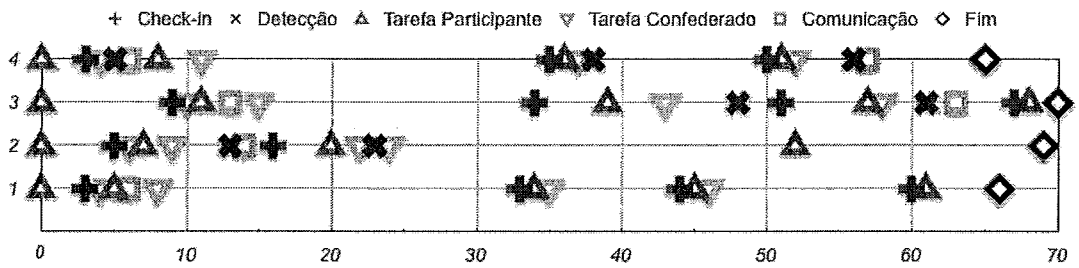



**Figura 5.1 - Linha de tempo dos principais eventos observados durante o experimento.**

Seguindo a linha do tempo, desde o início, podemos ver que os participantes executaram a primeira tarefa sem qualquer interferência dos confederados. A primeira tarefa de um confederado só foi executada após a conclusão da primeira tarefa do participante. Estamos considerando que uma tarefa foi concluída quando é feito o check-in do código relativo a ela. A primeira tarefa de um confederado envolvia fazer alterações na classe da qual o participante havia feito *check-in*. As mudanças introduzidas não afetavam a tarefa anterior, nem a seguinte, do participante. Estas somente provocavam um conflito para o próprio confederado, que devia fazer a combinação das versões. Os participantes podiam ignorar essas mudanças com segurança. Esta tarefa foi introduzida para que os indivíduos pudessem se acostumar à idéia de que estavam ocorrendo alterações feitas pelos confederados em paralelo. Apenas o participante 4 detectou as alterações feitas e contactou o confederado. O restante dos participantes passaram para a próxima tarefa sem notar as mudanças e sem quaisquer conseqüências.

Em seguida, a mesma tarefa foi designada tanto para o participante, quanto para um confederado, ao mesmo tempo. Ao criar esta situação, queríamos observar se o participante iria detectar e lidar corretamente com esta duplicação de trabalho. Durante esta tarefa, a maioria dos indivíduos contactou os confederados, mas apenas um o fez por causa da duplicação. O participante 2, contactou o confederado correto sobre a questão, pedindo-lhe para fazer check-in de suas mudanças. A conversa pode ser vista na Figura 5.2, à esquerda.

As mudanças completavam a segunda tarefa e, portanto, o participante 2 passou rapidamente para a próxima tarefa. Os demais apenas consultaram os confederados sobre uma construção da linguagem de programação, mas continuaram

a implementar as mudanças duplicadas. Estes participantes encontraram dificuldades na hora de fazer *check-in* das mudanças, por causa dos conflitos gerados entre as mudanças dos confederados e as suas próprias.

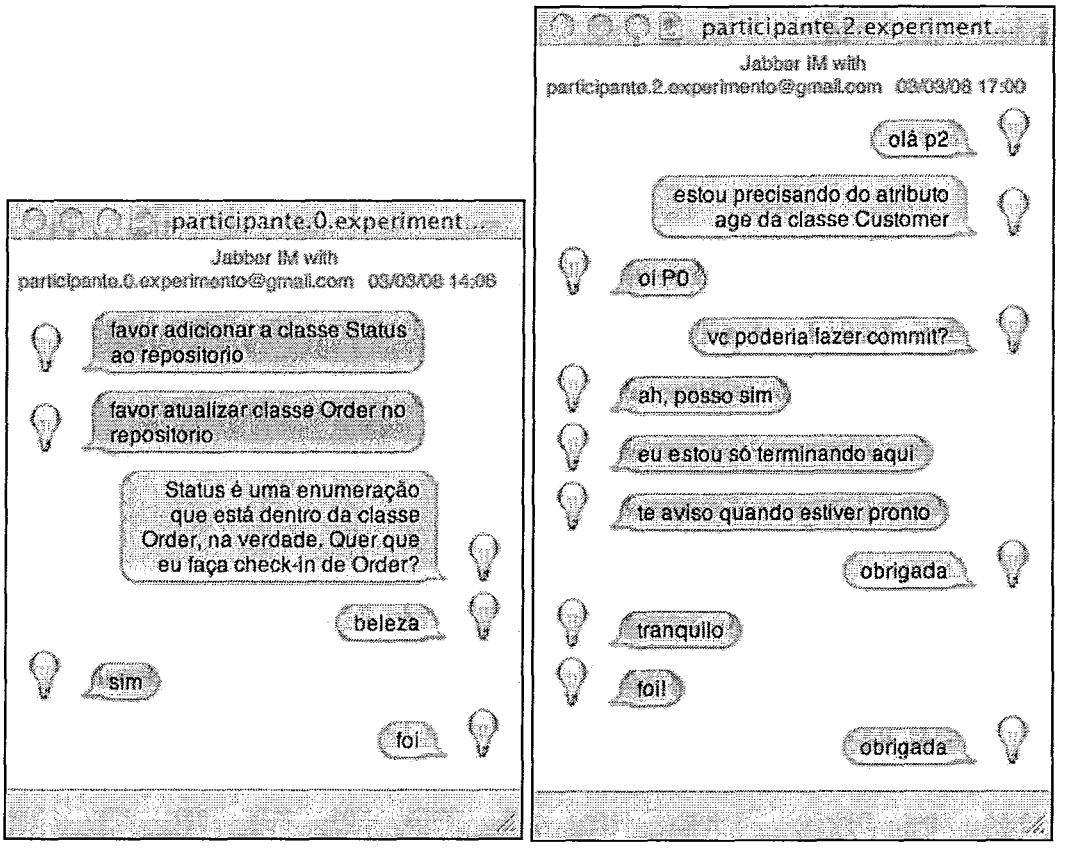



**Figura 5.2 - Comunicação entre participantes e confederados: à esquerda, janela no participante 2 contactando o confederado sobre a segunda tarefa; e, à direita, janela de um dos confederados (P2), sendo contactado sobre a quarta tarefa pelo participante (P0).**

A terceira tarefa dos participantes era completamente independente da próxima tarefa do confederado. Ambos precisavam criar novas classes, mas em partes diferentes e desconexas do projeto. Estas tarefas foram introduzidas para que se pudesse observar se os participantes seriam capazes de identificar tarefas não-relacionadas e se eles entrariam em contato com o confederado sobre esse assunto. Todos os participantes que detectaram as mudanças (três dos quatro) a ignoraram corretamente. Nenhum participante contactou o confederado responsável pelas mudanças. O participante 2, no entanto, não executou a terceira tarefa, por nenhum motivo aparente, provavelmente por engano. De qualquer maneira, ele detectou as mudanças feitas pelo confederado durante a execução da quarta tarefa e as ignorou adequadamente.

A próxima tarefa levava a um conflito direto entre o participante e um dos confederados. A tarefa do confederado envolvia a remoção de uma classe e a transferência de seu conteúdo para uma outra, já existente. O participante necessitava deste conteúdo para completar a sua tarefa. Metade dos participantes detectou o conflito e contactou o confederado responsável, resolvendo o problema. Uma conversa entre um dos participantes que detectou o problema e o confederado é mostrada na Figura 5.2, à direita. O participante 2 encontrou o conflito quando tentou fazer check-in das mudanças e passou algum tempo tentando entender o que havia acontecido e combinando as alterações manualmente. Em um determinado momento, no entanto, o participante desistiu da resolução do conflito e passou para a próxima tarefa sem fazer *check-in* das mudanças. O participante 1 não detectou as mudanças e seu tempo terminou antes que pudesse completar a tarefa.

Apenas dois dos participantes (2 e 3) começaram a executar a quinta tarefa, e nenhum deles chegou a completá-la antes do tempo disponível. No entanto, para o participante 2 esta foi, na verdade, a sua quarta tarefa, pois este havia ignorado uma das tarefas anteriormente. Foram dados aos participantes entre 60 e 70 minutos para a conclusão das tarefas.

Pudemos observar que o tempo foi um fator decisivo na detecção das modificações dos confederados. Os participantes que iniciaram a codificação de uma tarefa antes da introdução de mudanças conflitantes pelos confederados, foram os que, geralmente, deixaram de detectá-las. Provavelmente, isto ocorreu devido a uma interpretação errada das informações mostradas no diagrama do Lighthouse. Todos os participantes pareciam estar se baseando apenas no filtro que indica quais elementos haviam sido modificados recentemente para detectar alterações feitas pelos confederados. No entanto, este filtro não distingue entre modificações feitas pelo participante localmente e mudanças feitas por demais integrantes da equipe, (neste caso, os confederados). O filtro destaca todas as alterações recentes da mesma forma, o que provavelmente levou os participantes a acreditarem que o elemento em que estavam trabalhando estava destacado somente por causa de suas próprias alterações. Provavelmente, o uso de cores diferentes para destacar mudanças locais e remotas teria aumentado a taxa de detecções das mudanças introduzidas pelos confederados.

Foi observado também que as alterações ou foram detectadas logo que inseridas, ou não foram detectadas até o final da tarefa, quando os participantes encontravam conflitos na hora de fazer o *check-in*. Todas as alterações detectadas a tempo, porém, foram rapidamente e adequadamente abordadas: as mudanças independentes eram corretamente ignoradas, sem a ajuda dos confederados; e as conflitantes foram resolvidas ao se contactar o confederado responsável pelo conflito em questão. É importante destacar que, em todos os casos, os participantes

contactaram o confederado mais adequado. Foi observado o uso das informações de autoria disponíveis no diagrama de *Design* Emergente para determinar qual confederado deveria ser contactado.

## 5.6. Avaliação dos Participantes

Depois de concluir as tarefas de programação, os participantes preencheram um questionário de avaliação do protótipo e do experimento em si. A avaliação geral foi positiva. Sobre o uso do Lighthouse, os participantes identificaram diferentes aspectos como sendo positivos:

- "Dá uma boa visão do sistema como um todo; permite chegar rapidamente nos autores dos elementos para tirar dúvidas e resolver problemas; facilita a sincronização no desenvolvimento, evitando, quando possível, que as pessoas mexam no mesmo elemento ao mesmo tempo, evitando problemas de conflito; permite ficar ciente de toda alteração no sistema, permitindo uma maior colaboração entre os desenvolvedores."
- "É uma boa referência para saber 'o que está acontecendo' nas outras áreas de trabalho. É integrada ao ambiente de programação, o que permite ter uma uniformidade nos conceitos envolvidos."
- "Alterações são exibidas assim que ocorrem; exibição de um modelo universal (comum a todos); visão periférica realmente consegui detectar alterações no modelo; recurso de foco na classe que você está desenvolvendo (*Link with Editor*)."
- "Facilita o entendimento do sistema; elimina possível re-trabalho, aumentando a produtividade; incentiva a comunicação entre os membros da equipe; permite o conhecimento em tempo real da estrutura do sistema."

A avaliação dos participantes foi considerada positiva por destacar muitos dos conceitos e funcionalidades propostos pela abordagem descrita neste trabalho. Isto nos traz uma boa indicação de que o protótipo conseguiu implementar adequadamente a abordagem proposta, além de que esta abordagem foi bem aceita entre os participantes. A avaliação geral da utilização do segundo monitor durante a execução das tarefas também foi positiva:

- "O segundo monitor facilita a visualização do modelo, além da percepção das modificações no sistema. Além disso, acaba facilitando o acesso e consulta ao modelo."
- "Extremamente positiva, além de aumentar minha área de trabalho, alterações que acontecem no segundo monitor são detectadas pela visão periférica, o que me fez dar atenção ao segundo monitor."

- "O segundo monitor funcionou de fato como um mecanismo auxiliar na programação. Ao recorrer a visualização neste dispositivo, podem ser resolvidos alguns problemas de consistência entre elementos ao passo que estes ocorrem."
- "Sim, pois assim que "chegava" uma alteração, eu percebia rapidamente e verificava se geraria algum conflito com o que eu estava fazendo."

Os participantes também foram questionados sobre os aspectos negativos da ferramenta, sendo que um deles não respondeu este item no questionário. O desempenho do protótipo foi apontado como principal problema:

- "Destaco apenas o problema na demora do *refresh* do lighthouse quando as alterações ocorrem, fazendo com que o desenvolvedor fique esperando esta ação acontecer, causando uma perda no foco do trabalho."
- "As vezes, um pouco de lentidão; notação do diagrama um pouco confusa para representar relacionamentos; não consigo salvar o modelo."
- "Problemas de desempenho (lentidão e *travamento*); não se recupera muito bem de falhas."

Entre os aspectos negativos da ferramenta, destacam-se as limitações do protótipo atual, não da nossa abordagem. Em uma outra questão do questionário, todos os participantes responderam que utilizariam a ferramenta em um projeto real, caso os problemas de desempenho e os demais pontos negativos fossem devidamente consertados.

Embora os participantes tenham se mostrado satisfeitos com a duração e o conteúdo do treinamento recebido para o experimento, sabemos que os indivíduos necessitam de mais tempo para amadurecer os conceitos aprendidos e se familiarizar com a ferramenta. Alguns recursos do Lighthouse, como, por exemplo, o uso de notas e vigias, não foram utilizados talvez por falta de um treinamento mais prático ou por falta de um maior incentivo do experimentador e seus confederados. O uso prolongado da ferramenta e uma situação mais realista (por exemplo, com uma equipe maior e um projeto de *software* mais complexo) seria o ideal neste caso.

## 5.7. Validade

Este estudo preliminar foi executado a fim de observar o uso controlado do protótipo desenvolvido. Os resultados obtidos a partir das observações e da avaliação dos participantes nos ajudaram a entender as limitações do protótipo e os pontos que precisam ser melhorados. No entanto, devido às restrições deste estudo, os resultados obtidos não podem ser generalizados.

A seleção dos participantes foi feita através da solicitação de voluntários dentro de um grupo de alunos que compartilham um mesmo laboratório de pesquisa do qual também fazem parte os experimentadores. Isto foi necessário devido a restrições de

tempo e de pessoal disponível. Como conseqüência, o grupo escolhido pode não ser representativo para a população que se deseja testar e pode ser influenciado pela sua relação com os experimentadores. Além disso, houve somente um número reduzido de participantes neste estudo. É possível que os resultados sejam influenciados pelo tamanho e pelas características específicas do grupo. O uso de alunos de pós-graduação, não tão experientes quanto profissionais da indústria, também restringe a generalização das observações obtidas.

As tarefas de programação e código do projeto de *software* utilizados durante o estudo eram pequenos e simples em comparação a projetos reais. Isto foi necessário devido a limitações de tempo para cada sessão do estudo e por problemas de desempenho do protótipo, como discutido anteriormente. A simplicidade desses artefatos de *software* também pode ter influenciado as observações obtidas. Acreditamos, porém, que o Lighthouse tem o potencial de ser ainda mais útil em projetos maiores. Neste estudo, os participantes poderiam facilmente e rapidamente navegar por todo código-fonte, para compreender o sistema. Embora alguns participantes tenham efetivamente utilizado o diagrama de *Design* Emergente para compreender a estrutura do sistema, estes não tinham tanta necessidade de uma abstração do sistema, como têm os desenvolvedores de sistemas complexos.

Outra limitação deste estudo é o fato de que a ferramenta foi utilizada logo depois do treinamento, sem que os participantes tenham tido tempo para amadurecer e assimilar todos os conceitos apresentados. O resultado do estudo pode ser influenciado pela falta de tempo de maturação do conhecimento necessário para realizar as tarefas propostas corretamente. Acreditamos que o Lighthouse possa ser melhor utilizado por desenvolvedores com um maior entendimento sobre a ferramenta.

## 5.8. Considerações Finais

Neste capítulo, foi descrito o estudo de observação desenvolvido para avaliar preliminarmente o Lighthouse. Neste estudo, cada participante executou algumas tarefas pré-determinadas de programação utilizando nosso protótipo, enquanto participava de uma equipe distribuída, simulada pelo experimentador. Na verdade, os demais membros da equipe, os confederados, eram controlados pelo experimentador, assim como as mudanças que eles introduziam no código-fonte do projeto e a comunicação entre eles e o participante. As mudanças introduzidas pelos confederados criavam oportunidades de coordenação, que podiam ou não ser aproveitadas pelo participante.

Ao observar os participantes utilizando o Lighthouse, pudemos perceber que ainda é necessário refinar nosso protótipo para o uso em projetos reais, principalmente no que diz respeito ao seu desempenho. No entanto, os resultados

foram considerados promissores, pois mesmo com as limitações atuais, os participantes conseguiram utilizar as informações do diagrama de *Design* Emergente para detectar e evitar conflitos durante o desenvolvimento de *software* em equipe. Os participantes se mostraram bastante receptivos em relação à abordagem proposta através da ferramenta e ao uso de um segundo monitor como fonte de informações de percepção.

Em relação aos quatro pontos apresentados no início deste capítulo, obtivemos relativo sucesso na observação do uso do protótipo:

1. O uso do mecanismo Lighthouse permite a detecção de conflitos na implementação do código-fonte do *software?* Apesar de nem todos os conflitos terem sido detectados, os participantes conseguiram evitar parte dos conflitos introduzidos. Conseguimos também identificar uma melhoria que poderia levar a um índice maior de detecção: a diferenciação de elementos modificados localmente e remotamente.

2. O diagrama de *Design* Emergente é utilizado como referência para o entendimento da estrutura do *software?* Todos os participantes citaram no questionário de avaliação como ponto positivo, a utilização do diagrama como um modelo que representa o *software* em desenvolvimento, de alguma forma. Além disso, o experimentador pôde observar a exploração do diagrama pelos participantes durante o estudo através dos elementos de software e seus relacionamentos ali apresentados.

3. O diagrama de *Design* Emergente é utilizado como referência sobre "o que está acontecendo" no projeto de *software*? Todos os participantes utilizaram o filtro que destaca as mudanças recentes no diagrama durante o estudo. Foi observado que as modificações detectadas pelos participantes foram descobertas principalmente através deste filtro. No questionário de avaliação, todos os participantes citaram de alguma forma esta funcionalidade do Lighthouse como ponto positivo.

4. O diagrama de *Design* Emergente estimula e serve como referência para a comunicação dentro da equipe? Um dos participantes citou no questionário de avaliação, o estímulo à comunicação como fator positivo do uso da ferramenta. No entanto, o resultado mais importante em relação a este ponto, foi obtido através da observação dos participantes. Os participantes, quando pretendiam contactar um membro da equipe para resolver algum conflito, sempre consultavam o diagrama de *Design* Emergente, para determinar o autor das mudanças. Como dito anteriormente, em todos os casos, os participantes contactaram o confederado adequado.

Outra observação interessante relacionada a estes pontos, foi o fato de que os participantes só iniciaram comunicação com os confederados quando era realmente

necessário. As mudanças introduzidas que não eram relacionadas às suas tarefas, eram corretamente ignoradas, sem consultar os confederados. Isso é uma indicação de que o Lighthouse estimula a "boa" comunicação, potencialmente evitando interrupções desnecessárias.

Este estudo foi um primeiro passo para a avaliação da abordagem apresentada neste trabalho. O estudo é limitado em diferentes aspectos, restringindo a generalização dos resultados obtidos, como discutido na seção anterior. Serão necessários, ainda, estudos adicionais para avaliar plenamente o que foi proposto.

# 6. Conclusão

## 6.1. Epílogo

Uma equipe de desenvolvedores que colabora para construir um sistema de *software* precisa coordenar suas tarefas. Quanto maior o número de integrantes e mais complexo for o sistema em desenvolvimento, mais difícil se torna esta tarefa. Além disso, quanto maior a necessidade de se trabalhar em paralelo, maior é também a probabilidade de se ter conflitos entre as atividades, o que geralmente gera re-trabalho. O suporte ferramental utilizado para apoiar a coordenação dentro destas equipes varia desde o simples correio eletrônico a completos Sistemas de Gerência de Configuração.

Um dos pontos críticos para o sucesso da colaboração, no entanto, é que os integrantes mantenham a ciência do que "está acontecendo" no projeto (DOURISH e BELLOTTI, 1992). Mecanismos de percepção são ferramentas que tem como objetivo principal levantar esse conhecimento do projeto e notificar os interessados. Os mecanismos descritos na literatura utilizam diferentes tipos de informação e meios para apresentá-las à equipe. No entanto, esses mecanismos apresentam problemas como representações pobres e pouco escaláveis das informações, a sobrecarga de informações e um alto nível de interrupção ao usuário. Esses problemas limitam a utilização destes mecanismos na prática.

Neste trabalho, foi apresentado o mecanismo Lighthouse, um mecanismo de percepção baseado numa abstração dinâmica da estrutura do código-fonte, que permite que os desenvolvedores acompanhem a evolução do sistema. Essa abstração ainda pode ser decorada com diferentes informações de percepção, para auxiliar os desenvolvedores a entender em detalhes "o que está acontecendo" no seu projeto. No entanto, como a quantidade de informações disponíveis é considerável, o Lighthouse apresenta diferentes mecanismos, automáticos e manuais, para ajudar o desenvolvedor a encontrar o que é relevante para ele no momento.

Um protótipo do mecanismo foi implementado como um módulo do ambiente de desenvolvimento Eclipse. Deste protótipo foi feita uma avaliação preliminar, através de um estudo de observação com estudantes de pós-graduação. Apesar de restrito, o estudo indicou algumas oportunidades de melhoria do protótipo e a avaliação dos participantes foi positiva.

Neste último capítulo, são apresentadas as principais contribuições (Seção 6.2) e limitações (Seção 6.3) deste trabalho e são discutidos os potenciais trabalhos futuros (Seção 6.4).

## 6.2. Contribuições

Esta dissertação apresentou os resultados de um trabalho de pesquisa que visou propor um mecanismo de percepção para suportar a coordenação de desenvolvedores de software, apresentando potenciais soluções para os problemas mais comuns a esses mecanismos. Entre as principais contribuições deste trabalho, podemos destacar:

- O estudo e a comparação de diferentes mecanismos de percepção descritos na literatura, destacando os pontos positivos e negativos mais comuns entre essas ferramentas.
- A proposta do mecanismo de percepção, Lighthouse, que se baseia numa abstração do código-fonte, denominada *Design* Emergente, que permite que os desenvolvedores de uma equipe acompanhem "ao vivo" a evolução da implementação do sistema de *software*. Essa abstração ainda pode ser enriquecida com diferentes informações de percepção.
- A proposta de potenciais soluções para lidar com os problemas comuns aos mecanismos de percepção, destacando-se a combinação de arranjos automáticos do diagrama com o uso de métricas de acoplamento para destacar os elementos de código potencialmente mais relevantes para a tarefa atual do desenvolvedor e esconder os menos importantes, amenizando a quantidade de informações apresentadas.
- A implementação de um protótipo do mecanismo proposto e a disponibilização de seu código-fonte, para futuras extensões. Além disso, a implementação foi feita em uma linguagem multi-plataforma e o mecanismo foi integrado ao popular ambiente de desenvolvimento Eclipse, facilitando a sua adoção.
- A avaliação preliminar do protótipo, através de um estudo de observação, com estudantes de pós-graduação, utilizando o mecanismo Lighthouse enquanto executavam tarefas de programação, em uma equipe simulada pelo experimentador. Os resultados obtidos no estudo poderão ser utilizados para melhorar o protótipo e como base para uma avaliação futura mais completa da abordagem.

## 6.3. Limitações

Fazendo uma análise crítica da abordagem proposta e do protótipo implementado, é possível perceber as limitações deste trabalho. As principais são listadas a seguir:

- A abordagem proposta não lida com a questão de privacidade dos seus usuários. Alguns desenvolvedores podem não se sentir à vontade com a divulgação das modificações que ocorrem na sua área de trabalho. No paradigma atual, os

desenvolvedores compartilham as suas contribuições através do repositório, mas isso geralmente só ocorre quando o código já está mais "maduro" e considerado pronto para ser divulgado. Entendemos que pode haver resistência à utilização da ferramenta ou mesmo que o seu uso, inicialmente, iniba seus usuários. A ferramenta, também, demanda um certo grau de maturidade da gerência, para não utilizar as informações apresentadas com o objetivo de vigiar as atividades dos desenvolvedores. Apesar das informações apresentadas estarem disponíveis de outras maneiras, o Lighthouse facilita o acesso contínuo a estas, o que pode influenciar no seu uso.

- O mecanismo Lighthouse foi desenvolvido especialmente para auxiliar desenvolvedores que utilizam linguagens de programação OO. Apesar da abordagem geral apresentada ser aplicável a outros tipos de linguagens, aspectos como o tipo de representação da informação (atualmente baseado em um diagrama de classes), por exemplo, deverão ser adaptados para refletir as características específicas da outra linguagem.
- O Lighthouse não suporta a manipulação simultânea de diferentes ramos de um mesmo projeto de *software*. Atualmente, o protótipo assume que todos os membros da equipe estão trabalhando sobre um mesmo ramo principal do repositório. Para suportar essa opção, a representação das informações no diagrama de *Design* Emergente, também, deveria ser modificada para levar em conta este aspecto.
- O protótipo implementado depende do ambiente Eclipse para receber as notificações sobre as mudanças feitas no código-fonte. Enquanto esta opção foi escolhida por ser mais leve do que a execução contínua de engenharia reversa do código-fonte, ela limita o protótipo em alguns aspectos. Se o código for editado fora do Eclipse, o Lighthouse atualmente não detecta a mudança e, consequentemente, seu diagrama não é atualizado. Uma vez que o diagrama perde a sincronia com o código, não existe atualmente uma forma de fazer essa sincronização.

## 6.4. Trabalhos Futuros

A partir do que foi proposto e implementado ao longo deste trabalho, é possível identificar possibilidades de melhoria e novas opções para se expandir a abordagem e o protótipo aqui apresentados. Entre esses possíveis trabalhos futuros, destacamos:

- A utilização do resultado do estudo de observação executado para melhoria do protótipo, levando em consideração a avaliação dos participantes. Os principais pontos de melhoria seriam o desempenho do protótipo e a diferenciação da notificação de mudanças locais em relação às remotas.

• O planejamento e a execução de um novo estudo, mais completo, para avaliação da abordagem proposta.

• A implementação da recuperação do histórico de um projeto através do SCV. O Lighthouse, atualmente, só consegue mostrar o histórico de modificações de um projeto a partir do momento que se começa a utilizar o mecanismo. As alterações passadas, mesmo que disponíveis no SCV, não são levadas em consideração atualmente.

• A comparação do *Design* Emergente com o *design* conceitual. Caso a equipe disponha de um diagrama de classes construído antes da implementação do sistema, seria interessante comparar o que foi proposto inicialmente para o projeto com o que está sendo implementado na prática. Os diagramas poderiam ser sobrepostos, diferenciando os elementos que se encontram em ambos os diagramas dos que se encontram só em um ou outro. Essa comparação poderia dar aos desenvolvedores uma melhor idéia do progresso da implementação, indicando quais partes do sistema já estão prontas e o quanto falta para o seu término. Além disso, os elementos que fossem implementados de forma diferente em relação ao que foi planejado, poderiam ser marcados como desvios do *design* original.

• O diagrama de *Design* Emergente pode ser utilizado para mostrar diferentes tipos de informação sobre o projeto. Enquanto o foco deste trabalho é a informação de percepção sobre a tarefa de implementação do projeto, é possível utilizar o diagrama como base para informações sobre outras tarefas do ciclo de vida do *software.* Por exemplo, a cobertura de testes do projeto poderia ser projetada no diagrama: os elementos seriam coloridos de acordo com a quantidade de testes existentes para cada um deles. Os elementos com 100% de cobertura de testes poderiam ser coloridos de verde, enquanto os de baixa cobertura, de vermelho. Ao analisar o diagrama, os integrantes da equipe poderiam encontrar as partes do sistema que precisam de mais atenção neste sentido. Uma opção análoga poderia ser utilizada para se visualizar o resultado desses testes ou a quantidade de *bugs* reportados sobre cada um dos elementos. Outros artefatos, como modelos, requisições de mudanças e requisitos do sistema também poderiam ser atrelados ao Lighthouse, através da integração com outras ferramentas utilizadas pela equipe.

• A exploração de diferentes níveis de abstração para o *Design* Emergente. Enquanto a utilização de um diagrama de classes parece apropriado para que os desenvolvedores visualizem a estrutura do *software*, seria interessante explorar outros níveis de abstração para se acompanhar a evolução do sistema. Para gerentes ou outros indivíduos que necessitem ter uma visão mais geral do

projeto, um diagrama menos detalhado, como um que só apresente pacotes ou componentes pode ser útil.

• Poderia ser feito algum tipo de análise sobre mudanças conflitantes no código-fonte, para chamar a atenção dos desenvolvedores para potenciais problemas através do diagrama de *Design* Emergente. Seria possível, por exemplo, detectar quando mais de um desenvolvedor está trabalhando em um mesmo artefato ou em artefatos muito acoplados. Nesse caso, os elementos ou o fundo do diagrama poderiam ser coloridos para chamar a atenção para o problema detectado.

# Referências Bibliográficas


APPLE, 2008, Ichat. http://www.apple.com/support/ichat/ (Acessado em Janeiro de 2008).

BALL, R. and NORTH, C., 2005, "Effects of tiled high-resolution display on basic visualization and navigation tasks". In: *Chi '05 extended abstracts on human factors in computing systems*, pp. 1196-1199, Portland, OR, USA, April.

BERLINER, B., 1990, "CVS II: parallelizing software development". In: *Proceedings of the usenix technical conference*, pp. 341-352, Washington, D.C., USA, January.

BIEHL, J. T., CZERWINSKI, M., SMITH, G., et al., 2007, "Fast Dash: a visual dashboard for fostering awareness in software teams". In: *Proceedings of the SIGCHI conference on human factors in computing systems*, pp. 1313-1322, San Jose, California, USA, April/May.

BINKLEY, D., 2007, "Source code analysis: a road map". In: *Future of software engineering*, pp. 104-119, Minneapolis, MN, USA, May.

BNA, 2008, BNA update site. http://www.isr.uci.edu/projects/archstudio-update (Acessado em Janeiro de 2008).

BNA, 2008, Boxes and arrows graphical editing framework. http://tps.ics.uci.edu/trac/projects/browser/bna4 (Acessado em Janeiro de 2008).

BOHNER, S., 2002, "Software change impacts - an evolving perspective". In: *Proceedings of the international conference on software maintenance (icsm'02)*, pp. 263-271, Montréal, Canada, October.

BRAGA, R., WERNER, C. M. L., and MATTOSO, M., 1999, "Odyssey: a reuse environment based on domain models". In: *2Nd IEEE symposium on application-specific systems and software engineering technology*, pp. 50-57, Tel Aviv, Israel, October.

BRIAND, L. C., DALY, J. W., and WÜST, J. K., 1999, "A unified framework for coupling measurement in object-oriented systems". *IEEE Transactions of Software Engineering*, v. 25, n. 1(January) , pp. 91-121.

BRIAND, L. C., WUEST, J., and LOUNIS, H., 1999, "Using coupling measurement for impact analysis in object-oriented systems". In: *Proceedings of the IEEE international conference on software maintenance*, pp. 475-482, Oxford, England, UK, September.

CAMARGO, C. S. P. and SOUZA, J. M., 1992, "Flecha: a cooperative graphic editor". In: *XII international conference of the chilean computer science society*, pp. 195-206, Santigo, Chile.

CARMEL, E., 1999, *Global software teams: collaborating across borders and time zones*, 1 ed. Prentice Hall PTR, Upper Saddle River, NJ, USA.

CATALDO, M., WAGSTROM, P. A., HERBSLEB, J. D., et al., 2006, "Identification of coordination requirements: implications for the design of collaboration and awareness tools". In: *Proceedings of the 20th conference on computer supported cooperative work*, pp. 353-362, Banff, Alberta, Canada, November.

CHIDAMBER, S. R. and KEMERER, C. F., 1991, "Towards a metrics suite for object oriented design". *ACM SIGPLAN Notices*, v. 26, n. 11(November) , pp. 197-211.

CINCOM, 2008, Cincom smalltalk. http://www.cincomsmalltalk.com/ (Acessado em Janeiro de 2008).

COLLABNET, 2008, Subclipse update site. http://subclipse.tigris.org/update_1.2.x (Acessado em Janeiro de 2008).

COLLABNET, 2008, Subclipse. http://subclipse.tigris.org/ (Acessado em Janeiro de 2008).

COLLABNET, 2008, Subversion. http://subversion.tigris.org (Acessado em Janeiro de 2008).

COLVIN, J., TOBLER, N., and ANDERSON, J. A., 2004, "Productivity and multi-screen displays". *Rocky Mountain Communication Review*, v. 2, pp. 31-53.

CURTIS, B., KRASNER, H., and ISCOE, N., 1988, "A field study of the software design process for large systems". *Communications of the ACM*, v. 31, n. 11, pp. 1268-1287.

CURTIS, B., SOLOWAY, E. M., BROOKS, R. E., et al., 1986, "Software psychology: the need for an interdisciplinary program". *Proceedings of the IEEE*, v. 74, n. 8, pp. 1092-1106.

DANTAS, A. R., 2001, *Oráculo: um sistema de críticas para a uml*. Trabalho de Conclusão de Graduação, IM/UFRJ, Rio de Janeiro, RJ, Brasil.

DOURISH, P. and BELLOTTI, V., 1992, "Awareness and coordination in shared workspaces". In: *Proceedings of the 1992 ACM conference on computer-supported cooperative work*, pp. 107-114, Toronto, Ontario, Canada, November.

DWYER, T., 2008, Fast node overlap removal algorithm implementation. http://www.csse.monash.edu.au/~tdwyer/ (Acessado em January 2008).

DWYER, T., MARRIOTT, K., and STUCKEY, P. J., 2005. "Fast node overlap removal". In: *Proceeding of the 13th International Symposium of Graph Drawing*, pp. 153-164, Limerick, Ireland, September.

ESPINOSA, J. A., 2002, *Shared mental models and coordination in large-scale, distributed software development*. Tese de Doutorado, Carnegie Mellon University, Pittsburgh, PA, USA.

FREE SOFTWARE FOUNDATION, 2008, Concurrent Versions System. http://www.nongnu.org/cvs (Acessado em Janeiro de 2008).

FREE SOFTWARE FOUNDATION, 2008, Revision Control System. http://www.gnu.org/software/rcs/ (Acessado em Janeiro de 2008).

FROST, R., 2007, "Jazz and the eclipse way of collaboration". *IEEE Software*, v. 24, n. 6, pp. 114-117.

FRUCHTERMAN, T. M. J. and REINGOLD, E. M., 1991, "Graph drawing by force-directed placement". *Software: Practice and Experience*, v. 21, n. 11, pp. 1129-1164.

GRINTER, R. E., 1995, "Using a configuration management tool to coordinate software development". In: *Proceedings of conference on organizational computing systems*, pp. 168-177, Milpitas, California, United States, August.

GRINTER, R. E., 1996, "Supporting articulation work using software configuration management systems". *Computer Supported Cooperative Work*, v. 5, n. 4, pp. 447-465.

GRINTER, R. E., HERBSLEB, J. D., and PERRY, D. E., 1999, "The geography of coordination: dealing with distance in R&D work". In: *Proceedings of the international ACM siggroup conference on supporting group work*, pp. 306-315, Phoenix, Arizona, United States, November.

GRUDIN, J., 2001, "Partitioning digital worlds: focal and peripheral awareness in multiple monitor use". In: *Proceedings of the SIGCHI conference on human factors in computing systems*, pp. 458-465, Seattle, Washington, USA, April.

GUTWIN, C. and GREENBERG, S., 2002, "A descriptive framework of workspace awareness for real-time groupware". *Computer Supported Cooperative Work*, v. 11, n. 3, pp. 411-446.

HEEKS, R., KRISHNA, S., NICHOLSON, B., et al., 2001, "Synching or sinking: global software outsourcing relationships". *IEEE Software*, v. 18, n. 2, pp. 54-60.

HERBSLEB, J. D. and MOITRA, D., 2001, "Global software development". *IEEE Software*, v. 18, n. 2, pp. 16-20.

HERBSLEB, J. D., MOCKUS, A., FINHOLT, T. A., et al., 2001, "An empirical study of global software development: distance and speed". In: *Proceedings of the 23rd international conference on software engineering*, pp. 81-90, Toronto, Ontario, Canada, May.

HOEK, A., REDMILES, D., DOURISH, P., et al., 2004, "Continuous coordination: a new paradigm for collaborative software engineering tools". In: *Proceedings of the workshop on directions in software engineering environments - 26th international conference on software engineering*, v. 2004, pp. 29-36, Edinburgh, United Kingdom, May.

HUPFER, S., CHENG, L., ROSS, S., et al., 2004, "Introducing collaboration into an application development environment". In: *Proceedings of the 2004 ACM conference on computer supported cooperative work*, pp. 21-24, Chicago, Illinois, USA, November.

IBM RATIONAL, 2008, Clearcase. http://www-306.ibm.com/software/awdtools/clearcase (Acessado em Janeiro de 2008).

IBM RATIONAL, 2008, Jazz. http://jazz.net (Acessado em Janeiro de 2008).

IEEE, 2005, *Std 828 - IEEE standard for software configuration management plans.*

JUNG FRAMEWORK DEVELOPMENT TEAM, 2008, Java Universal Network/Graph framework. http://jung.sourceforge.net/ (Acessado em Janeiro de 2008).

KAGDI, H., COLLARD, M. L., and MALETIC, J. I., 2005, "Towards a taxonomy of approaches for mining of source code repositories". *ACM SIGSOFT Software Engineering Notes*, v. 30, n. 4(July) , pp. 1-5.

KRAUT, R. E. and STREETER, L. A., 1995, "Coordination in software development". *Communications of the ACM*, v. 38, n. 3, pp. 69-81.

KRISHNA, S., SAHAY, S., and WALSHAM, G., 2004, "Managing cross-cultural issues in global software outsourcing". *Communications of the ACM*, v. 47, n. 4, pp. 62-66.

LATOZA, T. D., VENOLIA, G., and DELINE, R., 2006, "Maintaining mental models: a study of developer work habits". In: *Proceeding of the 28th international conference on software engineering*, pp. 492-501, Shanghai, China, May.

LIGHTHOUSE, 2008, Lighthouse source code. http://tps.ics.uci.edu/svn/projects/lighthouse/ (Acessado em Janeiro de 2008).

LOPES, M. A. M., 2005, *MAIS: um mecanismo para apoio à percepção aplicado a modelos de software compartilhados.* Dissertação de Mestrado, COPPE/UFRJ, Rio de Janeiro, RJ, Brasil.

LOPES, M. A. M., MANGAN, M. A. S., and WERNER, C. M. L., 2004, "MAIS: uma ferramenta de percepção para apoiar a edição concorrente de modelos de análise e projeto". In: *Anais do XVII simpósio brasileiro de engenharia de software - sessão de ferramentas*, pp. 61-66, Brasília, DF, Brasil, Outubro.

MALONE, T. W. and CROWSTON, K., 1990, "What is coordination theory and how can it help design cooperative work systems?". In: *Proceedings of the ACM conference on computer-supported cooperative work*, pp. 357-370, Los Angeles, California, USA, October.

MANGAN, M. A. S., 2006, *Uma abordagem para o desenvolvimento de apoio à percepção em ambientes colaborativos de desenvolvimento de software.* Tese de Doutorado, COPPE/UFRJ, Rio de Janeiro, RJ, Brasil.

MANGAN, M. A. S., DA SILVA, I. A., and WERNER, C. M. L., 2004, "GAW: uma ferramenta de percepção de grupo aplicada no desenvolvimento de software". In: *Anais do XVIII simpósio brasileiro de engenharia de software*, pp. 25-30, Brasília, DF, Brasil, October.

MICROSOFT, 2008, Team Foundation Server. http://msdn2.microsoft.com/teamsystem/ (Acessado em Janeiro de 2008).

MICROSOFT, 2008, Visual Source Safe. http://msdn2.microsoft.com/en-us/vstudio/aa700905.aspx (Acessado em Janeiro de 2008).

MICROSOFT, 2008, Visual Studio. http://msdn2.microsoft.com/vstudio/ (Acessado em Janeiro de 2008).

MURTA, L. G. P., 2006, *Gerência de configuração no desenvolvimento baseado em componentes*. Tese de Doutorado, COPPE/UFRJ, Rio de Janeiro, RJ, Brasil.

MURTA, L. G. P., VASCONCELOS, A. P. V., BLOIS, A. P. T. B., et al., 2004, "Run-time variability through component dynamic loading". In: *Anais do XVIII simpósio brasileiro de engenharia de software*, pp. 67-72, Brasília, DF, Brasil, Outubro.

MYSQL AB, 2008, MySQL database. http://www.mysql.com/ (Acessado em Janeiro de 2008).

NAEEM, N. A. and HENDREN, L., 2006, "Programmer-friendly decompiled java". In: *Proceedings of the 14th IEEE international conference on program comprehension* , pp. 327-336, Athens, Greece, June.

ODYSSEY, 2008, Odyssey. http://reuse.cos.ufrj.br/odyssey/ (Acessado em Janeiro de 2008).

OLIVEIRA, H., 2005, *Odyssey-VCS: Uma abordagem de controle de versões para elementos da UML*. Dissertação de Mestrado, COPPE/UFRJ, Rio de Janeiro, RJ, Brasil.

OLSON, G. M. and OLSON, J. S., 2000, "Distance matters". *Human-Computer Interaction*, v. 15, n. 2&3, pp. 139-178.

OLSON, G. M., OLSON, J. S., CARTER, M. R., et al., 1992, "Small group design meetings: an analysis of collaboration". *Human-Computer Interaction*, v. 7, n. 4, pp. 347-374.

OMG, 2008, Unified Modeling Language specification, version 2.1.1. http://www.omg.org/technology/documents/formal/uml.htm (Acessado em Janeiro de 2008).

ORSO, A., APIWATTANAPONG, T., LAW, J., et al., 2004, "An empirical comparison of dynamic impact analysis algorithms". In: *Proceedings of the 26th international conference on software engineering*, pp. 491-500, Scotland, UK, May.

PERRY, D. E., STAUDENMAYER, N., and VOTTA, L. G., 1994, "People, organizations, and process improvement". *IEEE Software*, v. 11, n. 4(July) , pp. 36-45.

PICKERING, J. M. and GRINTER, R. E., 1994, "Software engineering and CSCW: a common research ground". In: *Proceedings of the workshop on software engineering and human-computer interaction at ICSE*, pp. 241-250, Sorrento, Italy, May.

POSTGRESQL GLOBAL DEVELOPMENT GROUP, 2008, PostgreSQL database. http://www.postgresql.org/ (Acessado em Janeiro de 2008).

PRESSMAN, R. S., 2004, *Software engineering: a practitioner's approach*, 6 ed. McGraw-Hill.

RED HAT MIDDLEWARE, 2008, Hibernate. http://www.hibernate.org/ (Acessado em Janeiro de 2008).

RIPLEY, R. M., SARMA, A., and VAN DER HOEK, A., 2007, "A visualization for software project awareness and evolution". In: *4Th IEEE international workshop on visualizing software for understanding and analysis*, pp. 137-144, Alberta, Canada, June.

SARMA, A. and VAN DER HOEK, A., 2004, "A conflict detected earlier is a conflict resolved easier". In: *In collaboration, conflict, and control: the proceedings of the 4th workshop on open source software engineering*, pp. 82-86, Edinburgh, United Kingdom, May.

SARMA, A., BORTIS, G., and VAN DER HOEK, A., 2007, "Towards supporting awareness of indirect conflicts across software configuration management workspaces". In: *Proceedings of the 22nd IEEE/ACM international conference on automated software engineering*, pp. 94-103, Atlanta, Georgia, USA, November.

SARMA, A., NOROOZI, Z., and VAN DER HOEK, A., 2003, "Palantír: Raising awareness among configuration management workspaces". In: *Proceedings of the 25th international conference on software engineering*, pp. 444-454, Portland, Oregon, May.

SARMA, A., VAN DER HOEK, A., and REDMILES, D. F., 2007, "A comprehensive evaluation of workspace awareness in software configuration management systems". In: *IEEE symposium on visual languages and human-centric computing*, pp. 23-26, Idaho, USA, September.

SCHNEIDER, K., GUTWIN, C., PENNER, R., et al., 2004, "Mining a software developer's local interaction history". In: *Proceedings of the 2004 international workshop on mining software repositories at the IEEE international conference on software engineering*, pp. 106-110, Scotland, UK, May.

SCHUCKMANN, C., KIRCHNER, L., SCHÜMMER, J., et al., 1996, "Designing object-oriented synchronous groupware with coast". In: *Proceedings of the 1996 ACM conference on computer supported cooperative work*, pp. 30-38, Boston, Massachusetts, United States, November.

SCHÜMMER, T. and HAAKE, J. M., 2001, "Supporting distributed software development by modes of collaboration". In: *Proceedings of the 7th european conference on computer supported cooperative work*, pp. 79-98, Bonn, Germany, September.

SCHÜMMER, T., 2001, "Lost and found in software space". In: *Proceedings of the 34th annual hawaii international conference on system sciences*, v. 9, pp. 9066, Maui, Hawaii, January.

SHULL, F., CARVER, J., and TRAVASSOS, G. H., 2001, "An empirical methodology for introducing software processes". *SIGSOFT Software Engineering Notes*, v. 26, n. 5, pp. 288-296.

SILVA, I. A., 2005, *GAW: um mecanismo visual de percepção de grupo aplicado ao desenvolvimento distribuído de software*. Trabalho de Conclusão de Graduação, IM/UFRJ, Rio de Janeiro, RJ, Brasil.

SILVA, I. A., ALVIM, M., RIPLEY, R., et al., 2007, "Designing software cockpits for coordinating distributed software development". In: *Proceedings of the 1st workshop on measurement-based cockpits for distributed software and systems engineering projects, in conjunction with the IEEE international conference on global software engineering*, pp. 14-18, Munich, Germany, August.

SILVA, I. A., CHEN, P. H., WESTHUIZEN, C., et al., 2006, "Lighthouse: coordination through emerging design". In: *Proceedings of the workshop on eclipse technology exchange at OOPSLA*, pp. 11-15, Portland, Oregon, October.

SINGER, J., 1998, "Practices of software maintenance". In: *Proceedings of the international conference on software maintenance*, pp. 139-145, Bethesda, Maryland, USA, November.

SOUZA, C. R. B., REDMILES, D., MARK, G., et al., 2003, "Management of interdependencies in collaborative software development". In: *Proceedings of the 2003 international symposium on empirical software engineering*, pp. 294, Rome, Italy, September.

THE ECLIPSE FOUNDATION, 2008, Eclipse documentation - installing new features with the update manager. http://help.eclipse.org/stable/index.jsp?topic=/org.eclipse.platform.doc.user/tasks/tasks-34.htm (Acessado em Janeiro de 2008).

THE ECLIPSE FOUNDATION, 2008, Eclipse Java Development Tools subproject. http://www.eclipse.org/jdt/ (Acessado em Janeiro de 2008).

THE ECLIPSE FOUNDATION, 2008, Eclipse. http://www.eclipse.org/ (Acessado em Janeiro de 2008).

THOMAS, D. and JOHNSON, K., 1988, "Orwell: a configuration management system for team programming". *SIGPLAN Not.*, v. 23, n. 11, pp. 135-141.

TRAVASSOS, G. H. and ROCHA, A. R., 1992, "O gerador de ferramentas da estação TABA". In: *XV Taller de ingenieria de sistemas*, Santigo, Chile.

VERONESE, G. O. and NETTO, F. J., 2001, *Uma ferramenta de apoio a recuperação de projetos no ambiente Odyssey*. Trabalho de Conclusão de Graduação, IM/UFRJ, Rio de Janeiro, RJ, Brasil.

WERNER, C. M. L., MANGAN, M., MURTA, L. G. P., et al., 2003, "Odysseyshare: an environment for collaborative component-based development". In: *IEEE international conference on information reuse and integration*, pp. 61-68, Las Vegas, NV, USA, Outubro.

WESTHUIZEN, C., CHEN, P. H., and HOEK, A., 2006, "Emerging design: new roles and uses for abstraction". In: *Proceedings of the 2006 international workshop on role of abstraction in software engineering - International conference on software engineering*, pp. 23-28, Shanghai, China, May.

WOHLIN, C., RUNESON, P., HÖST, M., et al., 2000, *Experimentation in software engineering: an introduction*, 1 ed. Kluwer Academic Publishers, Norwell, MA, USA.

YOST, B., HACIAHMETOGLU, Y., and NORTH, C., 2007, "Beyond visual acuity: the perceptual scalability of information visualizations for large displays". In: *Proceedings of the SIGCHI conference on human factors in computing systems*, pp. 101-110, San Jose, California, USA, May.

# Apêndice I

## F1 Formulário de Consentimento

### LIGHTHOUSE

Este estudo visa observar a utilização da ferramenta *Lighthouse* no contexto de desenvolvimento *software* em equipes distribuídas.

### IDADE

Eu declaro ter mais de 18 anos de idade e concordar em participar de um estudo conduzido por Isabella Almeida da Silva na Universidade Federal do Rio de Janeiro.

### PROCEDIMENTO

Este estudo em uma única sessão, que incluirá um treinamento sobre a ferramenta *Lighthouse* e o desenvolvimento em equipe de um jogo utilizando a linguagem Java. Eu entendo que, uma vez o experimento tenha terminado, os trabalhos que desenvolvi, serão estudados visando entender a eficiência dos procedimentos e as técnicas que me foram ensinadas.

### CONFIDENCIALIDADE

Toda informação coletada neste estudo é confidencial, e meu nome não será divulgado. Da mesma forma, me comprometo a não comunicar os meus resultados enquanto não terminar o estudo, bem como manter sigilo das técnicas e documentos apresentados e que fazem parte do experimento.

### BENEFÍCIOS, LIBERDADE DE DESISTÊNCIA

Eu entendo que os benefícios que receberei deste estudo são limitados ao aprendizado do material que é distribuído e ensinado. Eu entendo que sou livre para realizar perguntas a qualquer momento ou solicitar que qualquer informação relacionada a minha pessoa não seja incluída no estudo. Eu entendo que participo de livre e espontânea vontade com o único intuito de contribuir para o avanço e desenvolvimento de técnicas e processos para a Engenharia de *Software*.

### PESQUISADOR RESPONSÁVEL

Isabella Almeida da Silva
Programa de Engenharia de Sistemas e Computação - COPPE/UFRJ

### PROFESSOR RESPONSÁVEL

Profa. Cláudia M.L. Werner
Programa de Engenharia de Sistemas e Computação - COPPE/UFRJ

**Nome (em letra de forma):**_____

**Assinatura:** _____ **Data:**_____

# Apêndice II

| F2 | Caracterização do Participante | ID |
|---|---|---|

Este formulário contém algumas perguntas sobre sua experiência acadêmica e profissional.

**1) Formação Acadêmica:**

( ) Doutorado
( ) Mestrado
( ) Especialização
( ) Graduação
( ) Técnico
( ) Outra: _____

Ano de ingresso: _____ Ano de conclusão: _____

**2) Formação Geral:**

2.1) Qual é sua experiência com desenvolvimento de *software* em Java? (marque aqueles itens que melhor se aplicam)

( ) já li material sobre desenvolvimento de *software* em Java.
( ) já participei de um curso sobre desenvolvimento de *software* em Java.
( ) nunca desenvolvi *software* em Java.
( ) tenho desenvolvido *software* em Java para uso próprio.
( ) tenho desenvolvido *software* em Java como parte de uma equipe, relacionado a um curso.
( ) tenho desenvolvido *software* em Java como parte de uma equipe, na indústria.

2.2) Por favor, explique sua resposta. Inclua o número de semestres ou número de anos de experiência relevante em desenvolvimento de *software* em Java. (E.g. "Eu trabalhei por 2 anos como programador de *software* em Java na indústria")

_____
_____
_____

2.3) Qual é sua experiência com desenvolvimento de *software* em equipes? Qual a maior equipe que você participou?

_____
_____
_____
_____

2.4) Por favor, indique o grau de sua experiência nesta seção seguindo a escala de 5 pontos abaixo:

*0 = nenhum*
*1 = estudei em aula ou em livro*
*2 = pratiquei em projetos em sala de aula*

*3 = usei em projetos pessoais*
*4 = usei em projetos na indústria*

| | | | | | |
|---|---|---|---|---|---|
| 2.4.1) Ambiente de Desenvolvimento Eclipse | 0 | 1 | 2 | 3 | 4 |
| 2.4.2) Modelagem de sistemas de informação | 0 | 1 | 2 | 3 | 4 |
| 2.4.3) UML (Unified Modeling Language) | 0 | 1 | 2 | 3 | 4 |
| 2.4.4) Sistemas de Controle de Versão (CVS, Subversion, SourceSafe, etc.) | 0 | 1 | 2 | 3 | 4 |

2.5) Qual a sua familiaridade com a língua inglesa?
*0 = nenhuma*
*1 = pouca*
*2 = razoável*
*3 = boa*
*4 = muito boa*

| | | | | | |
|---|---|---|---|---|---|
| 2.5.1) Leitura | 0 | 1 | 2 | 3 | 4 |
| 2.5.2) Escrita | 0 | 1 | 2 | 3 | 4 |
| 2.5.3) Compreensão Oral | 0 | 1 | 2 | 3 | 4 |
| 2.5.4) Fala | 0 | 1 | 2 | 3 | 4 |

## 3) Experiência em Contextos Diferentes:

Esta seção será utilizada para compreender quão familiar você está com o domínio que será utilizado para as atividades durante o experimento. Por favor, indique o grau de experiência nesta seção seguindo a escala de 3 pontos abaixo:

*0 = Eu não tenho familiriadade com este domíno. Eu nunca fiz isto.*
*2 = Eu utilizo isto algumas vezes, mas não sou um especialista.*
*4 = Eu sou muito familiar com este domíno. Eu me sentiria confortável fazendo isto.*

| | | | | | |
|---|---|---|---|---|---|
| Domínio de Lojas Virtuais | 0 | 1 | 2 | 3 | 4 |

**Obrigado por sua colaboração!**

# Apêndice III - Resultados da Caracterização

**Tabela III.1 - Resultado do Questionário de Caracterização, por participante.**

| | | | Pergunta | P1 | P2 | P3 | P4 |
|---|---|---|---|---|---|---|---|
| 1 | | | Formação acadêmica | Graduação | Mestrado | Graduação | Graduação |
| 2 | 2.1 | | Experiência em Java | Indústria | Indústria | Curso | Indústria |
| | 2.2 | | Tempo de experiência | 2 anos e meio | 7 anos | 3 anos | 4 anos e meio |
| | 2.3 | | Tamanho máximo da equipe | 6 membros | 10 membros | 4 membros | 20 membros |
| | 2.4 | 2.4.1 | Experiência em Eclipse (1-4) | 4 | 4 | 3 | 4 |
| | | 2.4.2 | Experiência em modelagem (1-4) | 4 | 4 | 3 | 4 |
| | | 2.4.3 | Experiência em UML (1-4) | 4 | 4 | 3 | 4 |
| | | 2.4.4 | Experiência em SCV (1-4) | 4 | 4 | 4 | 4 |
| | 2.5 | 2.5.1 | Leitura em inglês (1-4) | 3 | 3 | 3 | 4 |
| | | 2.5.2 | Escrita em inglês (1-4) | 2 | 3 | 3 | 3 |
| | | 2.5.3 | Compreensão oral em inglês (1-4) | 3 | 3 | 2 | 3 |
| | | 2.5.4 | Fala em inglês (1-4) | 1 | 2 | 2 | 2 |
| 3 | | | Experiência com domínio de loja virtual(1-4) | 4 | 4 | 2 | 3 |

# Apêndice IV

| F3 | Descrição da Tarefa | ID |
|---|---|---|

**Instruções**

Este estudo é gravado através da captura da tela do computador e da gravação de som através de um microfone. Sempre que possível, verbalize seus pensamentos, para que o experimentador possa melhor avaliar os resultados obtidos. Pergunte e comente tudo que achar necessário.

**Contextualização**

Você está participando de uma equipe de desenvolvimento de *software* da empresa *Javali*. Esta empresa trabalha com o desenvolvimento de aplicações em Java. A sua equipe é responsável pelo projeto *OnlineStore*, uma loja virtual. Um dos integrantes da equipe do projeto deixou a empresa recentemente e você acabou de ser contratado para substituí-lo. A sua equipe hoje contém mais dois integrantes, que se encontram na filial da empresa em outra cidade. Os dois integrantes podem ser contactados via mensagens instantâneas de texto. A equipe utiliza o ambiente Eclipse com o sistema de controle de versões Subclipse, além da ferramenta Lighthouse, para o desenvolvimento do projeto. Você receberá em breve suas primeiras tarefas de implementação no projeto.

# Apêndice V

## Tarefas

### Participante

**Tarefa 1: Modificar: "ShoppingCart.java"**

1. Adicionar método público: *clearCart()*, que remove todos os itens do *ShoppingCart*.
2. Adicionar método público: *getItemCount()*, que retorna o número de itens no *ShoppingCart*.

**Tarefa 2: Modificar: "Order.java"**

1. Adicionar enumeração: *Status*, que contém os possíveis estados de um pedido. Os valores devem ser: *IN_PROGRESS*, *CANCELLED* e *SHIPPED*.
2. Adicionar atributo: *status* do tipo *Status*, que representa o estado do pedido.
3. Adicionar métodos públicos: *getStatus()* e *setStatus(Status status)*, para acesso ao atributo *status*.
4. Adicionar métodos públicos: *startOrder()*, *shipOrder()* e *cancelOrder()*, que atribuem o status correto ao pedido.

**Tarefa 3: Modificar: "Item.java" e Criar: "Recommender.java"**

1. Adicionar atributo: *category*, do tipo *String*, na classe Item, que especifica a categoria do *Item*.
2. Adicionar métodos públicos: *getCategory()* e *setCategory(String category)*, para acesso ao atributo *category*.
3. Adicionar classe: *Recommender*, que fará a recomendação de itens para o cliente.
4. Adicionar métodos público: *findSimilarItems(List<Item> items, Item item)*, que itera pela lista de itens (passada como primeiro parâmetro) e retorna uma outra lista com os itens que pertencem à mesma categoria do item (passado como segundo parâmetro).

**Tarefa 4: Criar: "CreditCardValidator.java"**

1. Adicionar classe: *CreditCardValidator*, que fará a recomendação de itens para o cliente.
2. Adicionar método público: *isCardValid()*, que retorna verdadeiro se a idade do cliente for maior do que 18 anos e se o número do cartão de crédito tiver exatos 16 dígitos.

**Tarefa 5: Modificar: "Item.java"**

1. Adicionar método público: *getReviewsByRating(int rating)*, que retorna uma lista de *Reviews* que são maiores do que a nota (*rating*) passada por parâmetro.

As demais tarefas planejadas (6 a 12) não foram utilizadas durante o estudo, por limites de tempo.

## Confederados

**Tarefa 1: Confederado 1 : Modificar: "ShoppingCart.java"**

1. Adicionar método público: *clearCart()*, que remove todos os itens adicionados ao *ShoppingCart*.
2. Adicionar método público: *getItemCount()*, que retorna o número atual de itens no *ShoppingCart*.

**Tarefa 2: Confederado 1: Modificar: "Order.java"**

1. Adicionar enumeração: *Status*, que contém os possíveis estados de um pedido. Os valores devem ser: *IN_PROGRESS*, *CANCELLED* e *SHIPPED*.
2. Adicionar atributo: *status* do tipo *Status*, que representa o estado do pedido.
3. Adicionar métodos públicos: *getStatus()* e *setStatus(Status status)*, para acesso ao atributo *status*.
4. Adicionar métodos públicos: *startOrder()*, *shipOrder()* e *cancelOrder()*, que atribuem o status correto ao pedido.

**Tarefa 3: Confederado 2: Criar: "CustomerValidator.java"**

1. Adicionar classe: *CustomerValidator*, responsável pela validação de clientes.
2. Adicionar método público: *validateCustomer(Customer customer)*, que recebe um cliente como parâmetro e retorna verdadeiro se ele é válido ou falso, caso contrário.

**Tarefa 4: Confederado 2: Modificar: "Customer.java" e Remover: "Demographics.java"**

1. Modificar classe: *Customer*, mover atributos *age* e *gender* e os métodos *getAge()*, *setAge()*, *getGender()* e *setGender()* da classe *Demographics* para a classe *Customer*.
2. Remover classe: *Demographics*.

As demais tarefas planejadas (5 a 12) não foram utilizadas durante o estudo, por limites de tempo.

# Apêndice VI

## Projeto utilizado no estudo

### Diagramas

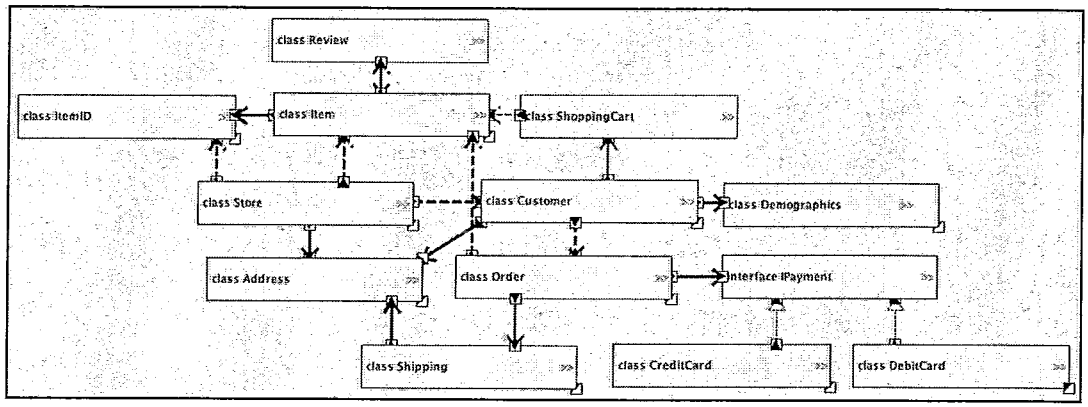



**Figura VI.1 - Diagrama de Design Emergente inicial do projeto de loja virtual.**

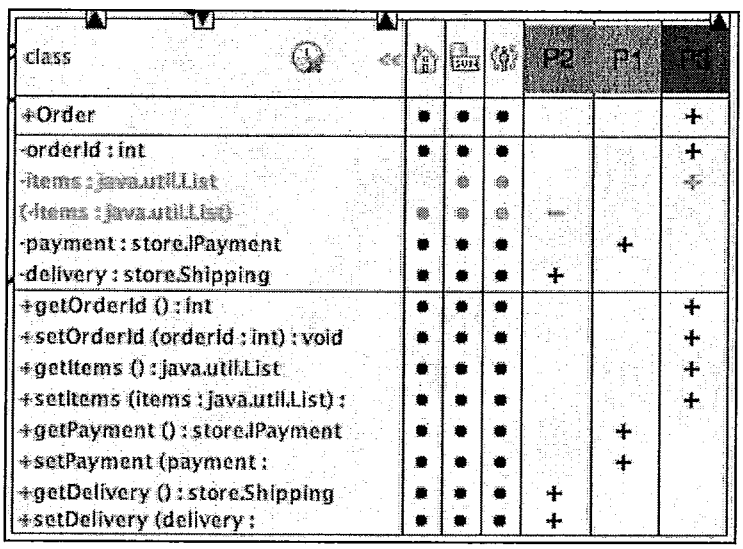



**Figura VI.2 - Classe *Order* do projeto de loja virtual, com histórico previamente construído para a execução do estudo.**

### Código-fonte

#### Address.java

```
package store;

public class Address {
     private String name;
     private String line1;
     private String line2;
     private String city;
     private String state;
     private String zip;
     private String phoneNumber;
```

```
    public Address(String name, String line1, String line2, String city, String state,
String zip, String phoneNumber) {
            this.name = name;
            this.line1 = line1;
            this.line2 = line2;
            this.city = city;
            this.state = state;
            this.zip = zip;
            this.phoneNumber = phoneNumber;
    }

    public String getCity() {
            return city;
    }

    public void setCity(String city) {
            this.city = city;
    }

    public String getLine1() {
            return line1;
    }

    public void setLine1(String line1) {
            this.line1 = line1;
    }

    public String getLine2() {
            return line2;
    }

    public void setLine2(String line2) {
            this.line2 = line2;
    }

    public String getName() {
            return name;
    }

    public void setName(String name) {
            this.name = name;
    }

    public String getState() {
            return state;
    }

    public void setState(String state) {
            this.state = state;
    }

    public String getZip() {
            return zip;
    }

    public void setZip(String zip) {
            this.zip = zip;
    }

    public String getPhoneNumber() {
            return phoneNumber;
    }
```

```
    public void setPhoneNumber(String phoneNumber) {
            this.phoneNumber = phoneNumber;
    }
}
```

**CreditCard.java**

```
package store;
import java.util.Date;

public class CreditCard implements IPayment {
    private String accountNumber;
    private Date expirationDate;
    private String name;

    public String getName() {
            return name;
    }

    public void setName(String name) {
            this.name = name;
    }

    public String getAccountNumber() {
            return accountNumber;
    }

    public void setAccountNumber(String accountNumber) {
            this.accountNumber = accountNumber;
    }

    public Date getExpirationDate() {
            return expirationDate;
    }

    public void setExpirationDate(Date expirationDate) {
            this.expirationDate = expirationDate;
    }
}
```

**Customer.java**

```
package store;
import java.util.List;

public class Customer {
    private ShoppingCart shoppingCart;
    private Address address;
    private Demographics demographics;
    private List<Order> orders;

    public Customer(Address address, Demographics demographics) {
            this.address = address;
            this.demographics = demographics;
    }

    public Address getAddress() {
            return address;
    }

    public void setAddress(Address address) {
```

```
                this.address = address;
        }

        public Demographics getDemographics() {
                return demographics;
        }

        public void setDemographics(Demographics demographics) {
                this.demographics = demographics;
        }

        public ShoppingCart getShoppingCart() {
                return shoppingCart;
        }

        public void setShoppingCart(ShoppingCart shoppingCart) {
                this.shoppingCart = shoppingCart;
        }
}
```

**DebitCard.java**

```
package store;

public class DebitCard implements IPayment {
        private String accountNumber;
        private String name;

        public String getName() {
                return name;
        }

        public void setName(String name) {
                this.name = name;
        }

        public String getAccountNumber() {
                return accountNumber;
        }

        public void setAccountNumber(String accountNumber) {
                this.accountNumber = accountNumber;
        }
}
```

**Demographics.java**

```
package store;

public class Demographics {
        private int age;
        private char gender;

        public int getAge() {
                return age;
        }

        public void setAge(int age) {
                this.age = age;
        }
```

```
    public char getGender() {
            return gender;
    }

    public void setGender(char gender) {
            this.gender = gender;
    }
}
```

### IPayment.java

```
package store;

public interface IPayment {
    public String getName();

    public String getAccountNumber();
}
```

### Item.java

```
package store;
import java.util.List;

public class Item {
    private ItemID itemId;
    private String name;
    private String description;
    private double price;
    private List<Review> reviews;

    public Item(ItemID itemId, String name, String description, double price) {
            this.itemId = itemId;
            this.name = name;
            this.description = description;
            this.price = price;
    }

    public String getDescription() {
            return description;
    }

    public void setDescription(String description) {
            this.description = description;
    }

    public ItemID getItemId() {
            return itemId;
    }

    public void setId(ItemID itemId) {
            this.itemId = itemId;
    }

    public String getName() {
            return name;
    }

    public void setName(String name) {
            this.name = name;
    }
```

```
    public double getPrice() {
            return price;
    }

    public void setPrice(double price) {
            this.price = price;
    }

    public List<Review> getReviews() {
            return reviews;
    }

    public void setReviews(List<Review> reviews) {
            this.reviews = reviews;
    }
}
```

**ItemID.java**

```
package store;

public class ItemID {
    private int upc;

    public int getUpc() {
            return upc;
    }

    public void setUpc(int upc) {
            this.upc = upc;
    }
}
```

**Order.java**

```
package store;
import java.util.List;

public class Order {
    private int orderId;
    private IPayment payment;
    private List<Item> items;
    private Shipping delivery;

    public Shipping getDelivery() {
            return delivery;
    }

    public void setDelivery(Shipping delivery) {
            this.delivery = delivery;
    }

    public int getOrderId() {
            return orderId;
    }

    public void setOrderId(int orderId) {
            this.orderId = orderId;
    }

    public List<Item> getItems() {
```

```
            return items;
    }

    public void setItems(List<Item> items) {
            this.items = items;
    }

    public IPayment getPayment() {
            return payment;
    }

    public void setPayment(IPayment payment) {
            this.payment = payment;
    }
}
```

**Review.java**

```
package store;

public class Review {
    public int rating;

    public String comments;

    public Item myItem;
}
```

**Shipping.java**

```
package store;
import java.util.Date;

public class Shipping {
    private Double weight;
    private Date estimatedDeliveryDate;
    private Address shippingAddress;

    public Date getEstimatedDeliveryDate() {
            return estimatedDeliveryDate;
    }

    public void setEstimatedDeliveryDate(Date estimatedDeliveryDate) {
            this.estimatedDeliveryDate = estimatedDeliveryDate;
    }

    public Double getWeight() {
            return weight;
    }

    public void setWeight(Double weight) {
            this.weight = weight;
    }

    public void setShippingAddress(Address shippingAddress) {
            this.shippingAddress = shippingAddress;
    }

    public Address getShippingAddress() {
            return this.shippingAddress;
    }
```

```
    public void printShippingAddress() {
            // Use the shippingAddress field.
    }
}
```

**ShoppingCart.java**

```
package store;
import java.util.List;

public class ShoppingCart {
    private List<Item> items;
}
```

**Store.java**

```
package store;
import java.util.List;

public class Store {
    private String name;
    private List<Item> items;
    private List<Customer> customers;
    private Address address;

    public void addItem(ItemID itemId, String name, String description, double price) {
            items.add(new Item(itemId, name, description, price));
    }

    public Address getAddress() {
            return address;
    }

    public void setAddress(Address address) {
            this.address = address;
    }

    public String getName() {
            return name;
    }

    public void setName(String name) {
            this.name = name;
    }

    public List<Customer> getCustomers() {
            return this.customers;
    }

    public List<Item> getItems() {
            return this.items;
    }
}
```

# Apêndice VII

| F4 | Questionário de Avaliação | ID |
|---|---|---|

## Realização da tarefa

| 1) Você conseguiu efetivamente realizar todas as tarefas propostas? Especifique, se necessário. | ☐ Sim ☐ Não ☐ Parcialmente |
|---|---|
| | |
| | |
| | |

| 2) Você ficou satisfeito com o resultado final das tarefas? Especifique, se necessário. | ☐ Sim ☐ Não ☐ Parcialmente |
|---|---|
| | |
| | |
| | |

| 3) Você sentiu dificuldades para trabalhar em equipe? Especifique. | ☐ Sim ☐ Não ☐ Parcialmente |
|---|---|
| | |
| | |
| | |

| 4) Você encontrou algum conflito entre sua tarefa e as do restante da equipe? Especifique. | ☐ Sim ☐ Não ☐ Parcialmente |
|---|---|
| | |
| | |
| | |

## Uso da Ferramenta

| 5) Você sentiu dificuldades na utilização do Lighthouse? Especifique. | ☐ Sim ☐ Não ☐ Parcialmente |
|---|---|
| | |
| | |
| | |

| 6) Você utilizou a ferramenta para determinar o autor de elementos do *software*? Se sim, como? Se não, por quê? | ☐ Sim ☐ Não ☐ Parcialmente |
|---|---|
| | |
| | |
| | |

| 7) Você utilizou a ferramenta para determinar se uma mudança estava no repositório ou na área de trabalho de outro desenvolvedor? Se sim, como? Se não, por quê? | ☐ Sim ☐ Não<br>☐ Parcialmente |
|---|---|
| | |
| | |
| | |

| 8) Você utilizou a ferramenta para se coordenar com o restante da equipe? Se sim, como? Se não, por quê? | ☐ Sim ☒ Não<br>☐ Parcialmente |
|---|---|
| | |
| | |
| | |

| 9) Você utilizou a ferramenta para entender a estrutura do sistema? Se sim, como? Se não, por quê? | ☐ Sim ☒ Não<br>☐ Parcialmente |
|---|---|
| | |
| | |
| | |

| 10) Você utilizou a ferramenta para entender o que os demais integrantes da equipe estavam fazendo? Se sim, como? Se não, por quê? | ☐ Sim ☐ Não<br>☐ Parcialmente |
|---|---|
| | |
| | |
| | |

| 11) Você utilizaria a ferramenta em uma equipe na prática? Por quê? | ☒ Sim ☐ Não<br>☐ Parcialmente |
|---|---|
| | |
| | |
| | |

| 12) Liste os aspectos positivos da ferramenta |
|---|
| |
| |
| |
| |

| 13) Liste os aspectos negativos da ferramenta |
|---|
| |
| |

| |
|---|
| |
| |

| 14) Você tem alguma sugestão para melhorar a ferramenta? Especifique. | ☐ Sim ☐ Não |
|---|---|
| | |
| | |
| | |
| | |
| | |

| 15) Você achou a experiência de ter um segundo monitor positiva? Por quê? | ☐ Sim ☐ Não |
|---|---|
| | |
| | |
| | |
| | |
| | |

## Treinamento

| 16) Você considera que o treinamento aplicado para o uso da ferramenta e para a realização da tarefa foi suficiente? Você teria sugestões? | ☐ Sim ☐ Não<br>☐ Parcialmente |
|---|---|
| | |
| | |
| | |

| 17) Você considera que existiu alguma dificuldade na interpretação das informações apresentadas sobre a ferramenta? Se sim, por favor, exemplifique. | ☐ Sim ☐ Não<br>☐ Parcialmente |
|---|---|
| | |
| | |
| | |

**Obrigado por sua colaboração!**