
# VIEWS: COMBINANDO VISÕES DE SOFTWARE NO DESENVOLVIMENTO DISTRIBUÍDO

Rafael da Silva Viterbo de Cepêda

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientador: Cláudia Maria Lima Werner

Rio de Janeiro
Junho de 2011

VIEWS: COMBINANDO VISÕES DE SOFTWARE NO DESENVOLVIMENTO DISTRIBUÍDO

Rafael da Silva Viterbo de Cepêda

DISSERTAÇÂO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) 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.

Examinada por:

_____

Prof.ª Cláudia Maria Lima Werner, D.Sc.

_____

Prof. Jano Moreira de Souza, Ph.D.

_____

Prof. Eber Assis Schmitz, Ph.D.

RIO DE JANEIRO, RJ – BRASIL

JUNHO DE 2011

Cepeda, Rafael da Silva Viterbo de

VIEWS: Combinando Visões de Software no Desenvolvimento Distribuído / Rafael da Silva Viterbo de Cepêda. – Rio de Janeiro: UFRJ/COPPE, 2011.

XII, 129 p.: il.; 29,7 cm.

Orientador: Cláudia Maria Lima Werner

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2011.

Referências Bibliográficas: p. 100-104.

1. Desenvolvimento de Software. 2. Percepção de Grupo. 3. Desenvolvimento Distribuído. I. Werner, Cláudia Maria Lima. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

# Agradecimentos

Gostaria de agradecer a todos aqueles que, de alguma forma, contribuíram para a realização deste trabalho. Aqui vai uma lista (provavelmente incompleta) daqueles que fizeram parte dessa jornada.

À Deus, primeiramente, por ter me dado forças e me guiado em todos os momentos da minha vida.

À minha família, que sempre me apoiou em todos os momentos. Obrigado por toda dedicação, carinho e investimento em mim e em minha educação.

À minha noiva Danielle, que esteve ao meu lado nas horas em que eu mais precisei. Obrigado pela paciência e compreensão nos momentos em que estive ausente por causa deste trabalho. Mesmo nesses momentos, você sempre esteve comigo. Sua companhia é minha fonte de inspiração.

À Profª Cláudia Werner, que sempre acreditou no meu trabalho e me apoiou em todos os momentos dessa caminhada. Caminhada que apresentou diversas particularidades e obstáculos, dificilmente vencidos se não fosse pela sua orientação, paciência e perseverança em ajudar. Terás para sempre meu carinho e admiração.

Aos professores Leonardo Murta e Aline Vasconcelos, que também muito me ajudaram e aconselharam ao longo de toda a minha vida acadêmica.

Aos membros da banca examinadora docentes do Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ, meu agradecimento pela contribuição à minha formação.

Aos meus colegas do LENS, por me apoiarem e estarem sempre dispostos a contribuir e ajudar no que fosse preciso.

Ao meu coordenador na Petrobras, Carlos Franco, por ter me proporcionado um ambiente favorável na empresa para que eu pudesse finalizar esta dissertação.

"Pois o preceito é uma lâmpada, e a instrução é uma luz",
*—Provérbios 6, 23.*

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

# VIEWS: COMBINANDO VISÕES DE SOFTWARE NO DESENVOLVIMENTO DISTRIBUÍDO

Rafael da Silva Viterbo de Cepêda

Junho/2011

Orientador: Cláudia Maria Lima Werner

Programa: Engenharia de Sistemas e Computação

A crescente complexidade da tecnologia de software tem resultado em grandes desafios na engenharia de sistemas, como a dificuldade de compreensão de suas partes constituintes e a falta de percepção das atividades desempenhadas pela equipe, que são dispostas em formas cada vez mais complexas. É caso do desenvolvimento distribuído de software, onde integrantes de uma equipe estão dispersos geograficamente e, consequentemente, não possuem os mesmos mecanismos de comunicação e percepção como integrantes de equipes co-localizadas possuem. A fim de amenizar esses problemas em um cenário de desenvolvimento distribuído, combinando perspectivas diferentes e complementares do processo de engenharia de software, este trabalho cria uma abordagem, baseada em uma infra-estrutura reutilizável de provimento de dados, capaz de fornecer percepções sobre o software e as pessoas ao longo do desenvolvimento. Para evidenciar o ganho no uso dessas informações quando empregadas em um cenário de desenvolvimento distribuído, a abordagem foi avaliada através de um experimento com estudantes de pós-graduação.

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

VIEWS: COMBINING SOFTWARE VISIONS IN A DISTRIBUTED DEVELOPMENT

Rafael da Silva Viterbo de Cepêda

June/2011

Advisor: Cláudia Maria Lima Werner

Department: Computer Science and Systems Engineering

The increasing complexity of software technology has resulted in major challenges in systems engineering, such as difficulty in understanding its constituent parts and the lack of awareness of the activities performed by the team, which is arranged in increasingly complex ways. It is the case of distributed software development, where members of a team are geographically scattered and thus do not have the same mechanisms of communication and awareness as co-located teams do. In order to mitigate these problems in a distributed development scenario, combining different and complementary perspectives of the software engineering process, this work aims on creating an approach, based on a reusable infrastructure for data provision, able to provide awareness about the software and people throughout the development cycle. In order to demonstrate the advantage of using this aggregated information in a distributed development scenario, the approach was evaluated through an experiment with graduate students.

# SUMÁRIO

# SUMÁRIO DE FIGURAS

# SUMÁRIO DE TABELAS

# Capítulo 1 - Introdução

## 1.1. Preâmbulo

A tecnologia de software e, conseqüentemente, a engenharia de software evoluíram de forma considerável nos últimos 50 anos, passando de programas com algumas centenas de linhas de código fonte e ferramentas rudimentares a complexos sistemas com milhares de linhas de código e avançadas técnicas de apoio ao desenvolvimento de software (Mei et al., 2006). De acordo com os mesmos autores, esta evolução pode ser atribuída principalmente a quatro forças impulsionadoras: a constante necessidade de melhor utilizar a crescente capacidade de hardware; a busca por um modelo computacional cada vez mais expressivo e natural; aumento da heterogeneidade de plataformas e demanda por interoperabilidade; e aumento da necessidade de abstração visando ao reúso e produtividade.

Como conseqüência deste fenômeno, desenvolvedores das mais variadas áreas de aplicação e de diferentes culturas vêm enfrentando em suas atividades de trabalho, ao longo dos anos, novos desafios e dificuldades provenientes deste novo patamar de complexidade. É como se o desenvolvedor de antigamente estivesse acostumado a construir casas de no máximo três andares sem muita infra-estrutura, enquanto que atualmente o mesmo seria requisitado a construir empreendimentos contendo diversos edifícios, ambientalmente responsáveis, contendo área de lazer, infra-estrutura de transporte, segurança, comunicação com outros empreendimentos, dentre outros aspectos. Isto é, problemas que outrora não teriam relevância passam a desempenhar muitas vezes um papel fundamental para o sucesso ou fracasso de uma aplicação.

Além disto, o aumento da complexidade trouxe consigo a necessidade da formação de equipes cada vez maiores e heterogêneas em habilidades individuais dentre os recursos humanos envolvidos. Tornando, desta forma, atividades como a coordenação do desenvolvimento de um sistema um fator chave, apesar de não único, para que um determinado projeto seja bem sucedido (Kraut e Streeter, 1995).

Com o advento da globalização, outro fator de extrema relevância surge neste cenário: a mudança no paradigma de disposição geográfica das equipes. O desenvolvimento distribuído de software (DDS), caracterizado por equipes espalhadas geograficamente, cada vez mais se torna uma realidade nas organizações dominantes do mercado. Mercados nacionais têm se transformado em mercados globais, criando novas formas de cooperação e competição que vão além das fronteiras dos países (Herbsleb e Moitra, 2001). Neste contexto, fatores como dificuldade de comunicação, gerenciamento e a percepção sobre as atividades dos demais membros da equipe são apenas uma pequena amostra dos problemas enfrentados (Herbsleb e Grinter, 1999).

## 1.2. Motivação

Com a finalidade de auxiliar na construção de imagens mentais para uma melhor compreensão de sistemas e fenômenos, a área de visualização emergiu como um importante ramo da ciência. Esta, atualmente, é dividida em duas linhas de aplicação: a visualização científica e a visualização de informação. A primeira processa dados físicos para obter representações visuais que, de certa forma, irão auxiliar cientistas e engenheiros na percepção de informações antes escondidas nos dados obtidos. Por outro lado, a visualização de informação utiliza dados abstratos a fim de criar resultados visuais úteis para um determinado conjunto de usuários. Portanto, pode-se dizer que a visualização de software é um tipo de visualização de informação, onde basicamente tenta-se reproduzir visualmente a estrutura, comportamento e a evolução de sistemas de software (Diehl, 2007).

A estrutura do software é usualmente entendida como aquela parte estática do mesmo, isto é, aquilo que pode ser obtido sem que o programa seja executado. Dentro do universo de possibilidades, alguns exemplos são: o código fonte, as estruturas de dados, a organização da arquitetura e o relacionamento entre sub-sistemas, dentre outros. Já o comportamento do sistema representa os diferentes estados que o mesmo passa ao ser executado, a partir de um determinado conjunto de condições iniciais. Exemplos deste tipo de representação são: fluxo de chamadas de funções, diagramas que mostram a seqüência de execução de objetos e diagramas que apresentam a dinâmica de comunicação entre

objetos, dentre outros. Além disto, existe ainda a visualização de software que se concentra na representação da evolução do sistema. Isto é, procura representar visualmente como seus artefatos foram evoluídos ao longo tempo, como mudanças no código fonte ou na arquitetura do sistema.

Desta forma, os benefícios obtidos por meio da utilização de técnicas de visualização de software têm sido observados tanto por pesquisadores de outras áreas da engenharia de software, como manutenção de software e engenharia reversa (Koschke, 2002), como por participantes da indústria (Bassil e Keller, 2001). Estes últimos acreditam que os principais benefícios trazidos pos estas técnicas são a economia de dinheiro e tempo, a melhor compreensão do software, o aumento da produtividade e qualidade e a maior facilidade obtida para gerenciar a complexidade.

No entanto, conforme observam Mockus e Herbsleb (2001), quando o desenvolvimento do software ocorre de forma distribuída, tais benefícios obtidos a partir da visualização de software são enfraquecidos. Isto porque, alguns fatores que antes não exerciam um papel determinante para o sucesso de um projeto, devido a proximidade das pessoas de uma equipe, passam a desempenhar um lugar de destaque neste novo paradigma de desenvolvimento. Por exemplo, a interdependência entre itens de trabalho (ex.: interdependência entre classes de um sistema) pode gerar grandes dificuldades de coordenação se itens interdependentes forem construídos de forma isolada entre equipes distribuídas. O que ocorre muitas vezes é que ambas as equipes, sem saber, realizam trabalhos conflitantes que podem perdurar durante um bom tempo no projeto até que sejam identificados, acarretando em perda de produtividade e qualidade (Herbsleb e Grinter, 1999).

## 1.3. Problema

Diversas abordagens foram criadas para atender aos objetivos da visualização de software, conforme definido anteriormente. Neste universo, podem ser encontrados trabalhos que representam textualmente ou na forma de diagramas a estrutura de programas, que apresentam visualizações da arquitetura estática de sistemas, que apresentam visualizações da arquitetura dinâmica do software, que forneçam animações

para algoritmos usualmente empregados, além daqueles que apresentam visualmente os diferentes aspectos da evolução de sistemas de software.

Novamente, como pode ser observado, a grande maioria destes trabalhos foca em um ou em um subconjunto das seguintes opções: visualização da estrutura, comportamento ou evolução do software. Além destes, também é importante ressaltar que diferentes trabalhos foram desenvolvidos com a finalidade de apoiar a coordenação de atividades em equipes distribuídas. Dentre estes, cabe ressaltar aqueles relacionados a percepção do espaço de trabalho, a identificação de especialistas e a criação de um modelo mental compartilhado sobre o projeto.

Contudo, poucos trabalhos atentaram para estas duas fontes de complexidade de forma uniforme: a complexidade de entender o software com todas as suas características e peculiaridades e a complexidade do processo de desenvolvimento em si, como o entendimento de quem está desenvolvendo o quê e como. Com isto, os potenciais resultados de uma abordagem unificada ainda não são claros, apesar de algumas evidências de sua eficácia já poderem ser encontradas na literatura. Por exemplo, a ferramenta Augur (Froehlich e Dourish, 2004), que será abordada no Capítulo 2, apresenta de forma unificada uma visualização do software (i.e., foco no resultado final), utilizando uma abordagem baseada em linhas de código (Ball e Eick, 1996), bem como uma visualização das atividades (i.e., foco no meio para atingir o resultado final) realizadas nestas linhas, a partir das informações obtidas do sistema de controle de versão.

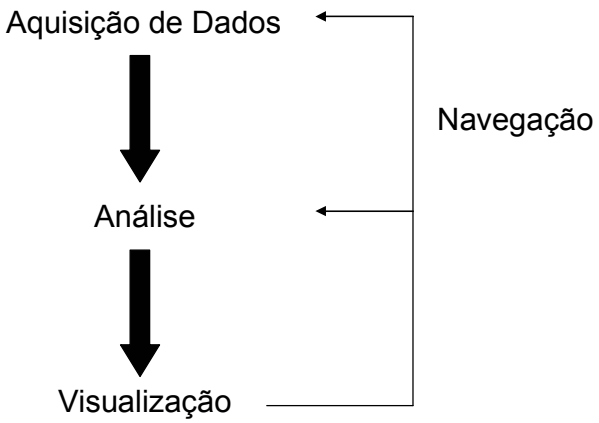



**Figura 1.1. O *pipeline* de visualização (Diehl, 2007).**

Entretanto, com a consideração de diferentes fontes de complexidade, um problema recorrente em diversas abordagens (incluindo a anteriormente citada) relacionadas a área de

visualização parece se tornar cada vez mais evidente: a incapacidade de reúso dos diferentes componentes implementados para a obtenção da visualização final (Storey et al., 2005). Desta forma, é importante observar que a exploração destas diferentes fontes em uma abordagem unificada deverá vir acompanhada de uma preocupação com a separação entre os componentes do *pipeline* de visualização (vide Figura 1.1), potencializando, assim, a capacidade de reutilização posterior do trabalho. Isto é, dada uma separação adequada desses componentes, um trabalho futuro baseado também em visualização poderia reaproveitar parte da implementação já realizada, ao invés de ter que "reinventar a roda".

## 1.4. Objetivo

Diante dos problemas expostos anteriormente, este trabalho tem como objetivo fornecer uma abordagem unificada que faça uso das informações obtidas do desenvolvimento de software para a visualização do mesmo (segundo aquela definição inicial sobre visualização de software) em conjunto com informações sobre o processo de desenvolvimento. Além disto, a abordagem também deve fornecer uma infra-estrutura capaz de ser reutilizada por outros trabalhos, de forma que novos tipos de informação possam ser obtidos e outras formas de representação visual possam ser criadas a partir de uma base em comum. Com isso, espera-se obter uma abordagem capaz de enriquecer o ambiente de equipes distribuídas, fomentando a comunicação entre os membros da equipe e contribuindo para que o desenvolvimento nesse tipo de cenário possa ser realizado de forma mais eficaz (e.g., reduzindo a quantidade de re-trabalho).

Para atingir estes resultados e realizar uma prova de conceito, algumas metas intermediárias devem ser alcançadas: (1) identificar que característica do software deve ser representada inicialmente; (2) identificar que característica do processo de desenvolvimento deve ser representada; (3) desenvolver uma arquitetura onde cada componente do *pipeline* de visualização possa ser tratado de forma independente; (4) elaborar como deve ser a representação visual das características obtidas em (1) e (2); (5) desenvolver cada componente de acordo com as metas aqui estabelecidas e integrá-los para que uma avaliação da abordagem possa ser realizada.

Note que, o resultado deste trabalho não tem como objetivo esgotar as possibilidades de visualização no quesito software (i.e., estrutura, comportamento e evolução) e nem no quesito do processo de desenvolvimento. Espera-se, portanto, apenas evidenciar que sem uma sinergia entre estas duas fontes de complexidade, a visualização de software tende a perder em utilidade, principalmente quando o paradigma de desenvolvimento se torna distribuído. Além disto, espera-se que este trabalho crie uma infra-estrutura comum capaz de ser reutilizada em outros trabalhos relacionados à área, evitando, novamente, que seja "reinventado a roda" da visualização.

## 1.5. Organização

O restante desta dissertação está organizado em outros cinco capítulos, além deste de introdução.

O Capítulo 2 apresenta uma introdução e uma revisão da literatura sobre o tema de visualização de software e sobre trabalhos de percepção em ambientes colaborativos. Como contexto para ambos os tópicos supracitados, são apresentados também as principais características e desafios relacionados com o desenvolvimento distribuído de software e como os diferentes trabalhos daquelas áreas estão sendo aplicados aos problemas deste tipo de desenvolvimento.

O Capítulo 3 apresenta a abordagem proposta por este trabalho de pesquisa, com base nos problemas identificados na literatura e nas abordagens existentes atualmente. Adicionalmente, são discutidas as principais características e funcionalidades propostas para superar estes problemas.

O Capítulo 4 apresenta os detalhes de implementação da abordagem proposta por esse trabalho por meio de um protótipo capaz de visualizar a estrutura de um sistema de software em conjunto com informações de percepção sobre o espaço de trabalho. Além disto, também são apresentados os detalhes da infra-estrutura criada para a obtenção, análise e armazenamento destas informações.

O Capítulo 5 discute o método e apresenta os resultados de uma avaliação inicial da abordagem proposta, com o objetivo de evidenciar os ganhos obtidos pela adoção dessa

estratégia, como aumento no nível de comunicação da equipe e melhor coordenação das atividades em um ambiente distribuído.

O Capítulo 6, finalmente, apresenta algumas conclusões desta dissertação, destacando as suas principais contribuições e relatando as limitações detectadas, além de indicar possíveis trabalhos futuros.

# Capítulo 2 - Revisão da Literatura

## 2.1. Introdução

O desenvolvimento de software é uma atividade complexa e, portanto, convive com desafios das mais variadas formas e naturezas. Entretanto, uma análise cuidadosa do tema nos leva a perceber que tais desafios, usualmente, estão associados basicamente às pessoas, processos ou tecnologia. Os desafios relacionados às pessoas dizem respeito a características que afetam diretamente os recursos humanos envolvidos ao longo do processo de desenvolvimento de software. Já os desafios relacionados ao processo dizem respeito à forma como o projeto será desenvolvido. Por último, os desafios relacionados à tecnologia tratam das diversas ferramentas e aparatos tecnológicos que podem ser utilizados como apoio ou base para o desenvolvimento de software.

Dependendo da disposição geográfica das equipes de um projeto de desenvolvimento de software, tais desafios serão mais ou menos acentuados em algumas perspectivas. Por exemplo, em um ambiente onde as equipes encontram-se dispersas geograficamente, isto é, em um ambiente de desenvolvimento distribuído de software (DDS), desafios relacionados às pessoas, como coordenação e produtividade, emergem naturalmente em função desta configuração. Ainda neste contexto, desafios de cunho tecnológico também aparecem em diversos cenários, como na utilização de ferramentas de apoio e de uma infra-estrutura de comunicação. A fim de solucionar diversos desses problemas, muitos trabalhos foram elaborados nas mais variadas áreas de pesquisa. Apenas a título de ilustração, podemos citar trabalhos das áreas de *Global Software Development* (GSD), *Computer Supported Cooperative Work* (CSCW), Visualização, Compreensão e Manutenção de Software.

Apesar da grande interseção que há entre algumas dessas áreas em diferentes trabalhos de pesquisa, este trabalho concentrou-se em analisar principalmente abordagens das áreas de Visualização de Software e CSCW, uma vez que o resultado obtido assemelha-se àqueles destas áreas estudas. Desta forma, este capítulo apresenta os principais conceitos

e alguns trabalhos relacionados às áreas de visualização de software e CSCW, principalmente sob o aspecto da percepção de grupo pesquisado nesta área. A Seção 2.2 discute brevemente sobre o cenário de aplicação deste trabalho, DDS, apresentando seus principais conceitos e desafios. A Seção 2.3 introduz o conceito de visualização de software e descreve alguns trabalhos da área que podem ser aplicados em diferentes situações do desenvolvimento distribuído. A Seção 2.4 descreve alguns conceitos da área de CSCW e apresenta alguns mecanismos de percepção de grupo utilizados para apoiar o processo de desenvolvimento. A Seção 2.5 discute sobre as abordagens apresentadas, sob a ótica do desenvolvimento distribuído, ressaltando onde cada tipo de abordagem é mais favorável e onde ambas falham quando se tenta obter respostas para certos tipos de questionamento. Na Seção 2.6, são apresentadas as considerações finais do capítulo.

## 2.2. Desenvolvimento Distribuído de Software

Problemas de comunicação e de coordenação de atividades, no contexto do desenvolvimento de software, têm sido, há tempos, importantes desafios enfrentados pelos membros desta comunidade (Brooks, 1995) (Curtis et al., 1988). Além disto, devido a fatores como redução de custos, pressão para um menor *time-to-market*, necessidade de uma corporação mais flexível, dentre outros, fizeram com que nestes últimos anos o desenvolvimento de software se tornasse cada vez mais disperso e multi-cultural, isto é, globalmente distribuído (Herbsleb e Moitra, 2001).

Desta forma, o DDS cada vez mais se torna uma realidade nas grandes organizações de Tecnologia da Informação dominantes do mercado (Herbsleb e Mockus, 2003). Mercados nacionais têm se transformado em mercados globais, criando novas formas de cooperação e competição que vão além das fronteiras dos países. Dentre os diferentes aspectos que tornam isto uma realidade, estão os menores custos no desenvolvimento de software em determinados países, aquisições de novas empresas estrangeiras e outros (Mockus e Herbsleb, 2001). Neste contexto, diferentes fatores representam grandes barreiras quando se trata de desenvolvimento de software. Fatores como dificuldade de comunicação, coordenação, gerenciamento e a dificuldade na identificação de pessoas

necessárias para uma comunicação mais eficaz são apenas uma pequena amostra dos problemas enfrentados neste cenário (Herbsleb e Grinter, 1999).

Segundo Herbsleb e Mockus (2003), existem fortes indícios de que o desenvolvimento de software quando se dá de forma distribuída leva mais tempo para ser realizado do que aquele que não ocorre desta forma. Conforme os mesmos autores apresentam, este resultado está fortemente relacionado com àqueles desafios citados anteriormente, isto é, desafios como a comunicação e coordenação. Isto porque, neste contexto, atrasos são introduzidos pela redução da comunicação, pela dificuldade de se encontrar a pessoa certa e estabelecer o contato, pela ineficiência do processo de coordenação, pela falta de entendimento comum do software sendo construído, bem como pela dificuldade de se estabelecer uma sessão colaborativa eficaz.

O desenvolvimento distribuído pode ser caracterizado dentre diferentes níveis de dispersão. Isto é, uma caracterização baseada na distância física entre os desenvolvedores envolvidos em uma organização. Segundo Prikladnicki e Audy (2008), estes níveis podem ser divididos em quatro categorias:

**Mesma Localização Física**: neste caso, todos os envolvidos encontram-se em um mesmo local. Neste cenário, muitos dos problemas citados anteriormente estão ausentes, ou ao menos, amenizados, uma vez que os membros da organização podem interagir diretamente uns com os outros. Desta forma, não existem problemas com fuso horário e as diferenças culturais raramente envolvem a dimensão nacional.

**Distância Nacional**: este cenário caracteriza-se por ter todos os envolvidos localizados dentro de um mesmo país. Já nesta situação, pode haver alguns problemas de fuso horário e diferenças culturais, usualmente, são mais acentuadas em relação ao cenário anterior.

**Distância Continental**: esta situação ocorre quando os envolvidos encontram-se espalhados em países diferentes, porém, dentro de um mesmo continente obrigatoriamente. Neste caso, reuniões formais presenciais se tornam mais difíceis de serem realizadas e problemas com fuso horário exercem um papel importante no processo.

**Distância Global**: neste caso, os envolvidos encontram-se em países distintos e em continentes distintos. As reuniões face a face ocorrem geralmente no início dos projetos e, entre outros fatores, a comunicação e as diferenças culturais podem ser barreiras para o

trabalho. O fuso horário exerce um papel fundamental, podendo impedir interações entre as equipes.

Conforme apresentado anteriormente, o processo de comunicação no contexto do desenvolvimento distribuído se torna algo de extrema importância em função das barreiras geográficas e culturais, fazendo com que métodos e ferramentas empregados em um cenário de mesma localização física se tornem muitas vezes de alto custo ou até mesmo inviáveis de serem realizados. Por exemplo, reunião face a face é um tipo de interação amplamente utilizado neste cenário, já em um cenário caracterizado por distâncias globais ou até mesmo continentais se torna uma prática altamente custosa de ser empregada.

Note que, as formas de comunicação podem ser caracterizadas pela riqueza de contexto que cada uma proporciona. Aqueles autores ainda apresentam uma lista de meios de comunicação amplamente utilizada em projetos de software. A Figura 2.1 apresenta estas informações. Repare que a figura também indica, respectivamente, o nível de contexto que cada meio proporciona, isto é, se através do mesmo é possível obter uma interação pobre em contexto ou uma interação rica em contexto. A medida que a forma de comunicação se torna mais rica em contexto, potencialmente a mesma se torna mais custosa em uma situação distribuída. Entretanto, dados os diferentes estágios do ciclo de desenvolvimento de software, algumas tarefas necessitam de comunicação mais rica do que outras. O contato com o cliente, por exemplo, deve ser o mais próximo possível do face a face durante o entendimento dos requisitos. Análise e projeto de software necessitam de meios ricos para que a colaboração ocorra.

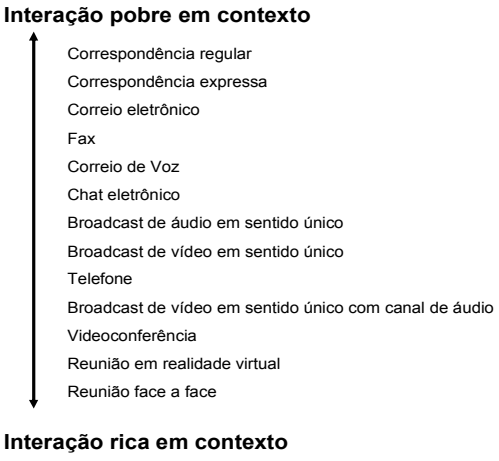



**Figura 2.1 – Formas de comunicação e níveis de interação contextual.**

Técnicas para aumentar o nível de percepção, *awareness*, têm sido estudadas e propostas com o intuito de auxiliar neste processo de comunicação e entendimento do software sendo construído pelas equipes distribuídas. Isto porque este tipo de cenário acaba ocultando informações que em projetos tradicionais estariam presentes no próprio ambiente de uma equipe de trabalho. Neste contexto, Carmel e Tjia (2005) definem pelo menos quatro tipos de percepção em projetos de DDS que visam a auxiliar nas atividades de comunicação e compreensão, em termos de eficácia e eficiência: existe a percepção da atividade, por exemplo, saber quem está trabalhando em qual atividade ou quem foi o responsável por uma determinada tarefa; percepção do processo, ou seja, saber como a tarefa de uma pessoa se encaixa na minha; percepção de presença, isto é, quem está disponível para responder uma determinada dúvida; e percepção do ambiente, em outras palavras, como o ambiente de trabalho de um determinado desenvolvedor afeta o seu trabalho, por exemplo, saber se já é noite ou se aquele desenvolvedor ainda encontra-se em expediente de trabalho.

A Tabela 2.1 resume estes tipos de percepção, apresentando exemplos de informações obtidas em cada caso e também as principais técnicas utilizadas hoje para aumentar o nível de percepção em cada categoria específica.

**Tabela 2.1 – Alguns tipos de percepção**

| **Tipo de Percepção** | **Questionamentos** | **Exemplo** | **Técnicas para aumentar o nível de percepção** |
|---|---|---|---|
| Atividade | Quem está trabalhando em qual atividade? | Aquele desenvolvedor já terminou o módulo de interface | Repositórios comuns<br>Sistemas para controle de projetos e tarefas<br>Reuniões de status de projetos |
| Processo | Como a tarefa daquela pessoa se encaixa na minha? O que eu devo fazer agora? | Acabei de descobrir algo importante sobre o teste deste componente. Agora, quem precisa saber disto? | Sistemas de workflow<br>Ambientes de desenvolvimento integrados<br>Descrição de processos |

| | | | |
|---|---|---|---|
| Presença | Quem está disponível para responder esta questão? | Como o gerente de projeto não veio hoje, quem pode me responder se teremos aquela reunião de equipe? | Cronograma das equipes<br>Cronogramas individuais de disponibilidade<br>Chat eletrônico |
| Ambiente | Como o ambiente de trabalho daquele desenvolvedor (condições climáticas, escritório) afeta o seu trabalho?<br>Já é noite naquela cidade? | Aquele analista de requisitos precisou de outra sala para trabalhar.<br>O temporal de hoje atrasou o expediente. | Troca de fotos e informações sobre o ambiente de trabalho<br>Reuniões por videoconferência<br>Intranet com notícias sobre as equipes e unidades distribuídas |

## 2.3. Visualização de Software

Indubitavelmente, a tecnologia de software está sendo utilizada cada vez mais em uma ampla variedade de áreas de aplicação. Desta forma, seu correto funcionamento torna-se essencial para o sucesso do negócio envolvido e para a segurança do ser humano (ISO/IEC, 2005). Neste contexto, as atividades de manutenção e compreensão de software emergem como duas das principais áreas do ciclo de vida de um sistema computacional.

Entretanto, a natureza intangível e invisível do software, alinhado com seu crescente grau de complexidade, torna tais atividades uma tarefa de difícil gerenciamento e execução (Ball e Eick, 1996), principalmente em cenários onde o desenvolvimento ocorre de forma distribuída. Esta dificuldade decorre, principalmente, daqueles desafios citados anteriormente e que estão associados a este tipo de desenvolvimento. Desta forma, técnicas de visualização de software têm sido utilizadas para proporcionar um meio pelo qual desenvolvedores possam visualizar, identificar e entender os diferentes artefatos de um software, auxiliando, portanto, em uma melhor execução das atividades supracitadas.

Uma das vantagens da visualização é que esta transfere do sistema cognitivo para o sistema perceptivo a carga despendida para uma determinada tarefa, aproveitando, assim, a grande habilidade do ser humano no reconhecimento de padrões e estruturas em informações visuais (Robertson et al., 1993). Portanto, o emprego desta técnica torna mais

fácil e natural a tarefa de compreensão da informação apresentada. Uma vez entendendo a estrutura e funcionamento dos artefatos do software, a execução das atividades de manutenção e coordenação de atividades podem se tornar tarefas de menor custo e complexidade.

A seguir, uma breve apresentação de algumas ferramentas baseadas em visualização é realizada. Tentou-se analisar em cada uma delas, principalmente, os aspectos da forma como a informação é apresentada, o tipo de informação que é disponibilizada, e as fontes de dados utilizadas na obtenção destas informações. Posteriormente, ao final do capítulo, todas as ferramentas pesquisadas serão analisadas e comparadas sob estes aspectos.

### 2.3.1. Augur

A ferramenta Augur (Froehlich e Dourish, 2004) é um sistema de visualização que tem como objetivo prover informações, ao longo do tempo, da estrutura de artefatos de um projeto e de suas atividades relacionadas. Isto é, a ferramenta parte da premissa de que, em um processo de desenvolvimento de software, existem basicamente dois tipos de complexidade que os desenvolvedores costumam lidar. A primeira é inerente aos artefatos gerados ao longo do processo, neste caso, trata-se de código fonte. A segunda complexidade refere-se às atividades por onde tais artefatos são gerados. Desta forma, esta abordagem procura, além de apresentar a evolução dos elementos de um projeto ao longo tempo, correlacionar tais eventos geradores de mudança às atividades estabelecidas ao longo do projeto.

A ferramenta utiliza uma abordagem de visualização baseada em linhas de código fonte (Eick et al., 1992). Desta forma, a ferramenta associa a cada linha de código um conjunto de informações de interesse, como a que atividade esta se refere, a que método ou classe esta se refere e quem criou a linha, dentre outras informações. Fazendo com que estas carreguem em si todas as informações necessárias para a sua compreensão e para a sua correlação com determinadas atividades do projeto.

Adicionalmente, é interessante observar que a ferramenta possui um conjunto muito bem definido de decisões arquiteturais. Por exemplo, a ferramenta foi desenvolvida para trabalhar de modo concorrente ao desenvolvimento do projeto, ao contrário de trabalhar de forma retrospectiva, isto é, baseado em dados históricos do projeto. Em outras palavras, as

informações providas pela ferramenta são atualizadas em tempo de desenvolvimento. Enfatizando o requisito imposto pela proposta de não apenas servir como um meio pelo qual desenvolvedores possam entender um determinado projeto, mas também ser útil nas atividades de coordenação de equipes geograficamente distribuídas.

Além disto, a ferramenta tem como requisito alta interoperabilidade. Desta forma, foi implementada uma arquitetura que permite a utilização de diferentes tipos de sistemas de versionamento como fonte de dados, assim como diferentes tipos de analisadores de código.

No que tange a questão da visualização dos dados, a ferramenta apresenta em uma única tela, porém em painéis distintos, diferentes características do projeto em andamento. Sua parte central, e principal, disponibiliza ao usuário uma visão temporal da evolução das linhas de código fonte do projeto. Níveis de coloração são estabelecidos para identificar o quão recente determinada linha foi modificada. Note que ao selecionar uma linha neste painel, informações associadas a mesma são exibidas nos demais. Por exemplo, no painel esquerdo são exibidos dois tipos de informação: informação do autor que modificou a linha no instante de tempo selecionado e o tipo de estrutura à qual esta linha está associada (p.ex. método, classe, comentário). O painel inferior contém informações adicionais como, informações de check-in e outros metadados do sistema de controle de versão. A Figura 2.2 apresenta a tela da ferramenta e suas funcionalidades.

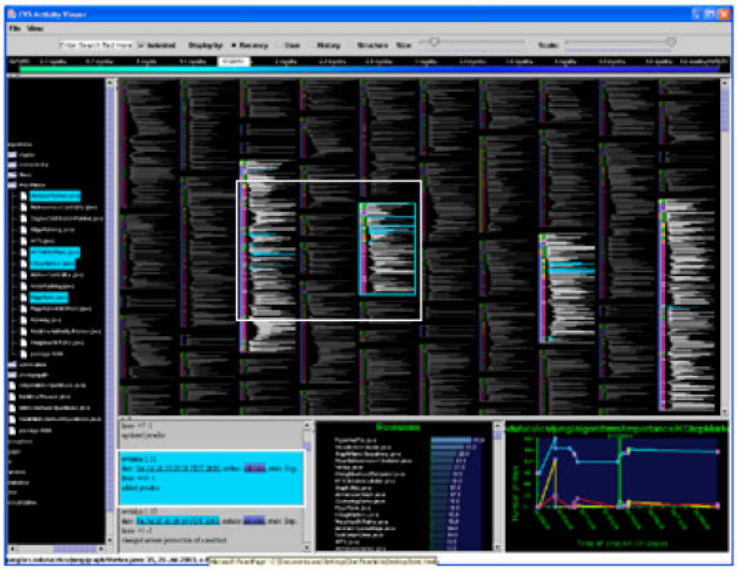


**Figura 2.2 - Tela da ferramenta Augur.**

O Augur utiliza como representação interna para as informações extraídas do projeto uma estrutura de dados baseada em linhas de código. Esta, por sua vez, é incrementada, posteriormente, com informações adicionais, como informações de indentação, data, autor, revisão, um nó de uma árvore sintática abstrata (Aho et al., 2006) (AHO et al., 2006) e informações da estrutura à qual esta linha pertence.

Como a ferramenta utiliza para a tarefa de análise de código fonte o sistema ANTLR, um gerador de *parser* baseado na tecnologia Java, esta pode suportar, em teoria, diversos tipos de linguagens de programação. Entretanto, no contexto desta implementação, foi adotada apenas a análise de código fonte escrito em Java.

Portanto, a ferramenta Augur representa uma abordagem para tratar do problema da visualização de software ao longo do tempo. A ferramenta utiliza para apresentação uma abordagem baseada em linhas de código fonte. Associados a estas, também são apresentados outros tipos de informação, como a sua estrutura de mais alto nível associada (como método, classe ou pacote) e o autor das diferentes modificações que ocorreram ao longo do tempo. Ao contrário das demais ferramentas, esta apresenta uma solução arquitetural que possibilita sua utilização com diferentes tipos de fontes de dados e analisadores de linguagem, aumentando, desta forma, sua capacidade de utilização.

### 2.3.2. CVSScan

O CVSscan (Voinea et al., 2005) é uma ferramenta de apoio ao processo de manutenção e entendimento de software que, através da técnica de visualização, expõe ao engenheiro diversos aspectos, ao longo do tempo, de uma determinada aplicação sob análise. O CVSscan utiliza uma abordagem baseada em linhas de código para a visualização do software, assim como a ferramenta Augur citada anteriormente.

Nesta abordagem (pixel line), pontos na tela são utilizados para representar, sob algum tipo de perspectiva, linhas de código fonte. Cores distinguem as possíveis variações de uma perspectiva. Por exemplo, na ferramenta CVSscan existem basicamente três perspectivas, ou dimensões, onde as linhas de código são classificadas: status da linha, tipo de construção e autor. A perspectiva de status da linha classifica-a em uma das seguintes categorias: constante (i.e., linha inalterada em relação à versão anterior), a ser inserida, modificada e removida. A perspectiva de tipo de construção classifica funcionalmente a

linha de acordo com a linguagem de programação utilizada. Por exemplo, se a linha for um comentário no programa, será classificada com uma categoria de mesmo nome. Já a perspectiva de autor, classifica a linha de acordo com o autor da linha. Cada autor que realiza alterações no software terá uma cor diferente. A Figura 2.3 ilustra algumas colorações utilizadas para cada perspectiva.

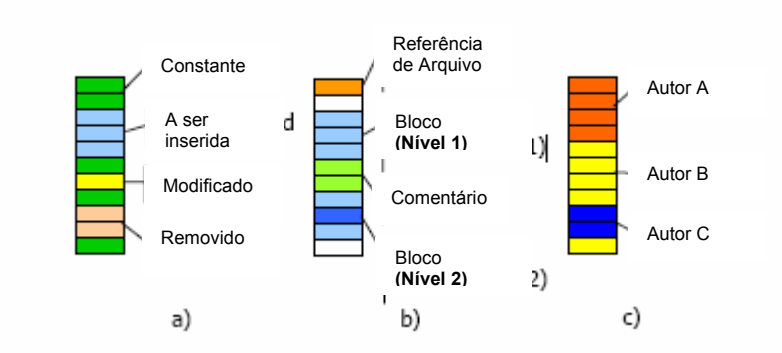



**Figura 2.3 - Colorações utilizadas: status da linha (a), tipo de construção (b) e autor (c).**

Todas estas linhas de código utilizadas, que representam a principal fonte de informação da aplicação, são originadas a partir de sistemas de controle de versão. Atualmente, apenas o sistema CVS é suportado pela ferramenta. Nesta implementação, uma outra ferramenta, chamada CVSgrab, é responsável por extrair as informações do CVS e repassar para o CVSscan para o devido processamento. Desta forma, podemos reparar que houve uma tentativa de se criar uma arquitetura modular para que no futuro outros sistemas de controle de versão pudessem ser utilizados com o CVSscan. Além disto, pôde ser observado que a ferramenta suporta a análise tanto de arquivos fonte escritos em linguagem C como arquivos fonte escritos em linguagem Perl (Wall et al., 2000).

Entretanto, pela simples análise da ferramenta, não fica claro em que tipo de formato estes dados trafegam entre o CVS e o CVSgrab (Voinea e Telea, 2006) e, posteriormente, entre o CVSgrab e o CVSscan, impossibilitando assim inferir o quão independente a ferramenta CVSscan é em relação ao sistema CVS.

Neste contexto, cada ponto discreto (i.e., versão) no ciclo de vida de um arquivo é representado pela ferramenta a partir da tupla: (identificação da versão, autor, data, código). Então, para comparar versões consecutivas de um determinado arquivo, a ferramenta utiliza uma aplicação externa semelhante ao aplicativo diff (Hunt e Mcllroy, 1976) (Hunt e

Szymanski, 1977) do sistema operacional UNIX. Assim, a partir da saída desta aplicação, o CVSscan rotula cada linha de acordo com as categorias de status de linha citadas anteriormente.

A Figura 2.4 ilustra como ocorre o processo de visualização na ferramenta. É interessante notar que a ferramenta não utiliza identação e tamanho da linha para representar uma possível estrutura para o arquivo. Ao contrário, utiliza-se de um tamanho fixo de linhas, maior ou igual ao maior número de linhas atingido pelo arquivo ao longo do tempo, e cores para codificar a estrutura.

Cada linha vertical representa uma versão do arquivo e cada linha horizontal representa uma linha do arquivo, conforme pode ser observado na parte central da ferramenta (marcada como "Visão Geral da Evolução"). Adicionalmente, podem ser observadas, nas bordas, métricas que complementam a visualização. Na borda esquerda, uma barra vertical representa o tamanho da versão em linhas, codificada através de cores. Na borda inferior, uma barra horizontal é utilizada para representar o autor responsável por cada versão.

Uma outra funcionalidade interessante é exemplificada na Figura 2.4. Note que, ao passar o *mouse* sobre uma parte da visualização (marcado como (1) na figura), ocorre respectivamente uma apresentação do código referente a esta passagem (marcado como (2) e (3) na figura). Além disto, a ferramenta oferece alguns recursos adicionais como zoom, alteração do tamanho das fontes exibidas, alteração do tamanho das linhas exibidas, seleção dos intervalos de tempo para exibição (através do "Seletor de Intervalo Esquerdo" e do "Seletor de Intervalo Direito") e outros.

É importante ressaltar que toda análise realizada pela ferramenta, e conseqüentemente todo resultado gerado por esta, tem como único foco arquivos e suas respectivas linhas. Isto é, a ferramenta é capaz apenas de representar, ao longo do tempo, a evolução de um único arquivo por vez, ou seja, representar como suas linhas foram sendo acrescidas, removidas ou alteradas ao longo da execução do projeto.

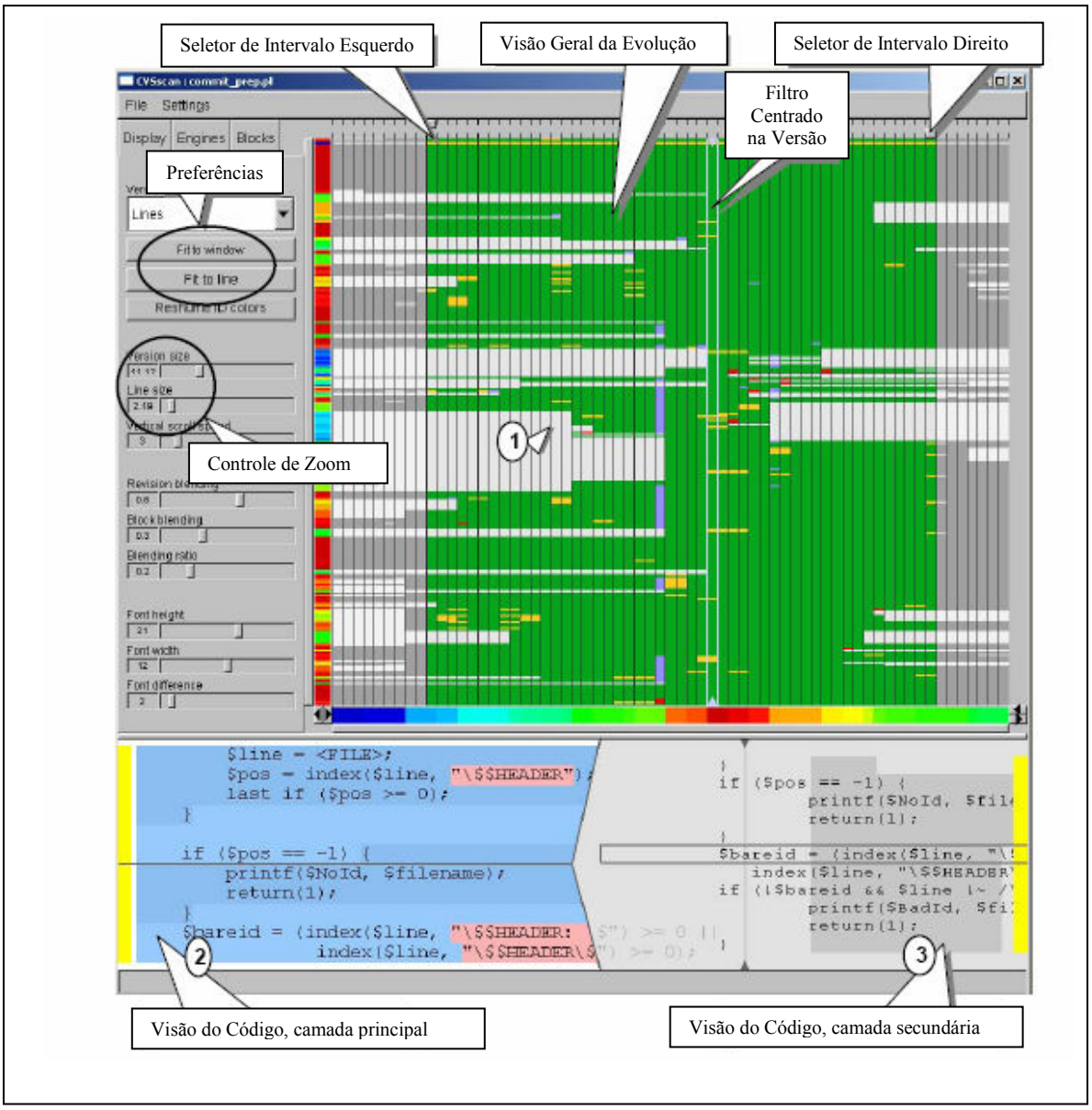



**Figura 2.4 - Visão geral da ferramenta CVSscan.**

## 2.3.3. softChange

Tomando como ponto de partida as limitações estabelecidas pelo sistema de controle de versão CVS, no que tange ao entendimento da evolução de seus projetos sob controle, criou-se a ferramenta softChange (German et al., 2004) (German, 2006). Esta tem por objetivo aproveitar informações disponíveis de projeto e, a partir destas, apresentar diferentes características do software ao longo do tempo.

A abordagem da ferramenta define como rastros de software informações deixadas pelos contribuidores de um determinado projeto ao longo do processo de desenvolvimento, como listas de e-mail, web sites, registros do sistema de controle de versão, releases de software, documentação e código fonte, dentre outros. Então, a idéia da ferramenta é, a partir de repositórios CVS, transformar estes rastros de software, utilizando determinadas heurísticas, em informações de mais alto nível que serão posteriormente apresentadas para o usuário final. Por exemplo, posteriormente, poderia ser apresentado, ao longo da linha do tempo, um reagrupamento do conjunto de revisões dos arquivos do projeto para um dado evento de *check-in* no repositório. Isto se torna útil neste cenário, uma vez que o sistema CVS não é orientado a transação, ou seja, o mesmo não mantém qualquer tipo de rastro ou ligação entre os diferentes arquivos modificados em um dado *check-in*.

A partir deste processo de extração dos rastros de software e da reconstrução de todos os eventos de *check-in* do projeto, uma etapa de análise é realizada. Nesta, classifica-se o evento como sendo uma adição de nova funcionalidade, uma reorganização de código fonte, um simples comentário, e etc. Por fim, após estas atividades de extração e análise, o softChange provê uma representação gráfica destas informações. Para serem visualizadas em navegadores *web*, a ferramenta ainda disponibiliza uma representação dessas informações na forma de documentos que contém hipertextos.

A arquitetura da ferramenta pode ser observada na Figura 2.5. Nesta, nota-se a existência de quatro componentes básicos: repositório de rastros de software, extrator de rastros de software, analisador de rastros de software e um visualizador.

O repositório de rastros de software é um banco de dados relacional responsável por armazenar todas as informações históricas extraídas sobre o projeto. O extrator de rastros de software é o componente responsável por resgatar as informações de projeto das fontes de dados suportadas. Nesta implementação, as fontes de dados suportadas são: o repositório CVS, arquivos do tipo ChangeLog, releases do software (através de arquivos compactados distribuídos pelos membros da equipe) e o sistema Bugzilla (BUGZILLA, 2011). O componente analisador de rastros de software é responsável por aplicar as heurísticas definidas pela abordagem e, a partir das informações previamente coletadas, inferir novas informações e características de software. Por exemplo, este componente possui a

inteligência de correlacionar eventos de *check-in* realizados ao longo do tempo com problemas cadastrados na ferramenta Bugzilla.

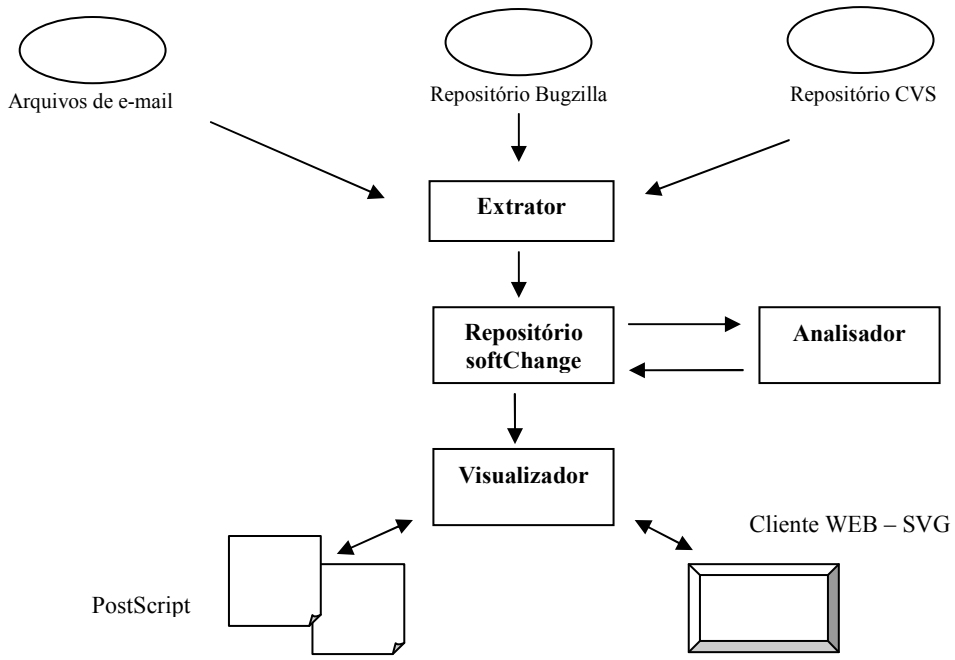



**Figura 2.5 - Arquitetura da ferramenta softChange.**

O último componente é o visualizador. Este é divido em duas partes: um interpretador de hipertexto (i.e., browser) e um visualizador gráfico. O primeiro é utilizado pelo usuário para navegar através dos diversos *check-ins* realizados no projeto ao longo do tempo. Esta navegação pode ser realizada por data, autor ou arquivo. Em uma navegação por data, por exemplo, o softChanges provê informações de que revisões dos arquivos pertencem a cada *check-in*, além de outros metadados associados à modificação.

O visualizador gráfico também pode ser subdividido em duas partes principais. A primeira utiliza arquivos Postscript para gerar gráficos estáticos dos rastros de software. Um exemplo de gráfico gerado em Postscript é ilustrado na Figura 2.6 (neste caso, o autor chama de MR o que está sendo chamado de *check-in* neste texto). A segunda utiliza os recursos de SVG para apresentar informações de forma interativa (Figura 2.7), como a relação de autoria entre os diversos desenvolvedores ao longo de um dado espaço de tempo.

A ferramenta softChange apresenta como principais contribuições a integração com mais de uma fonte de dados para a coleta de informações acerca do projeto, uma arquitetura modular onde papéis e responsabilidades foram bem definidos, aumentando a facilidade de

sua extensão. Além disto, proporciona um método para a visualização gráfica de diferentes propriedades de um dado projeto de software, como: crescimento de linhas de código ao longo do tempo, número de *check-ins* realizados ao longo do tempo, número de arquivos criados ao longo do tempo e número de arquivos por evento de *check-in*, dentre outros.

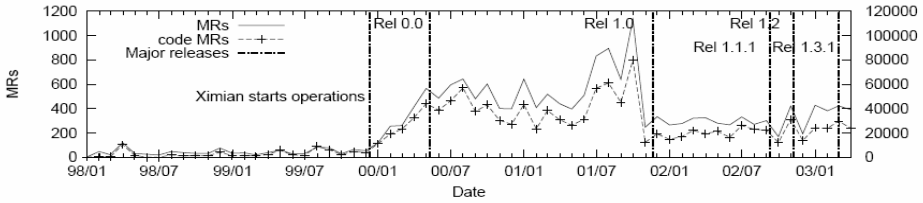



**Figura 2.6 - Exemplo de um gráfico gerado pelo softChange para o projeto open source Ximian.**

Porém, sua principal limitação aparenta ser seu forte vínculo temporal com sistemas de controle de versão. Isto é, por construção, a ferramenta pode apenas apresentar a evolução dada discretamente no tempo, onde cada ponto no espaço temporal representa um evento de *check-in* em um dado repositório de versionamento de código.

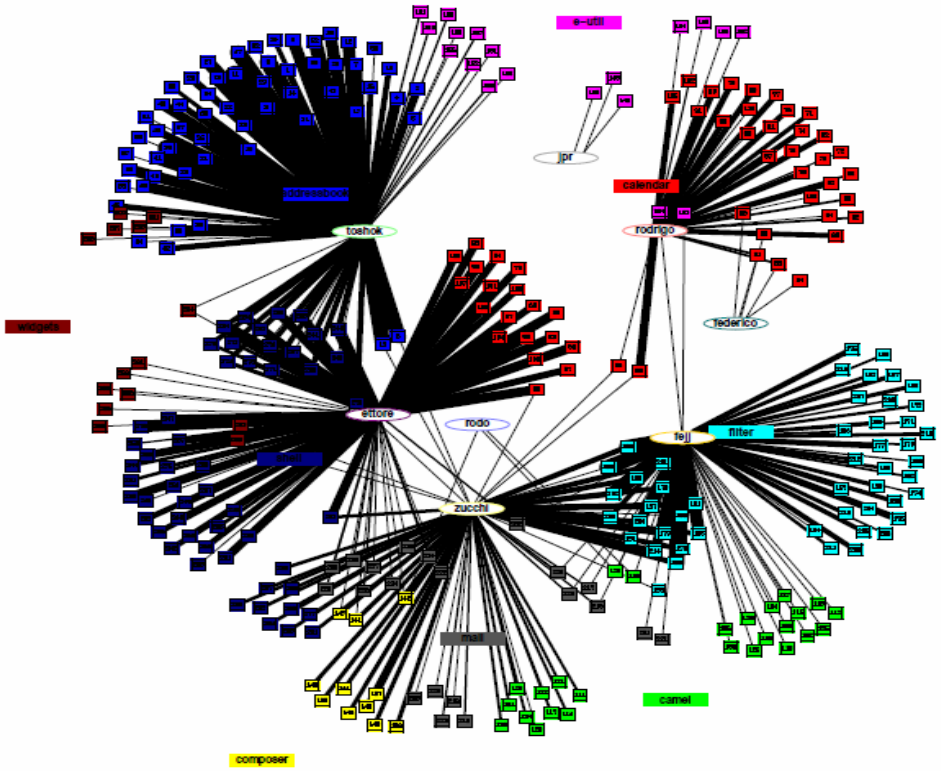



**Figura 2.7 – Gráfico de autoria durante período de manutenção evolutiva.**

### 2.3.4. EvolTrack

A abordagem EvolTrack (CEPEDA et al., 2010), apresentada na Figura 2.8, provê uma infra-estrutura capaz de capturar o ciclo de evolução de um projeto de software e, através da visualização, apresenta de forma temporal este ciclo de evolução previamente identificado. Repare que, além do próprio EvolTrack, existem dois elementos que integram esta visão do mecanismo: Fonte de Dados e Visualizadores. O primeiro, diz respeito a qualquer sistema externo capaz de armazenar e prover informações sobre o histórico de evoluções de um determinado projeto. Neste contexto, podem ser citados como exemplos de fontes de dados sistemas de versionamento de código fonte de projetos de software, como CVS e Subversion (Collins-Sussman et al., 2004), sistemas de versionamento de modelos do projeto de software, como Odyssey-VCS (Murta et al., 2008), ou até mesmo ambientes de desenvolvimento de software como Eclipse.

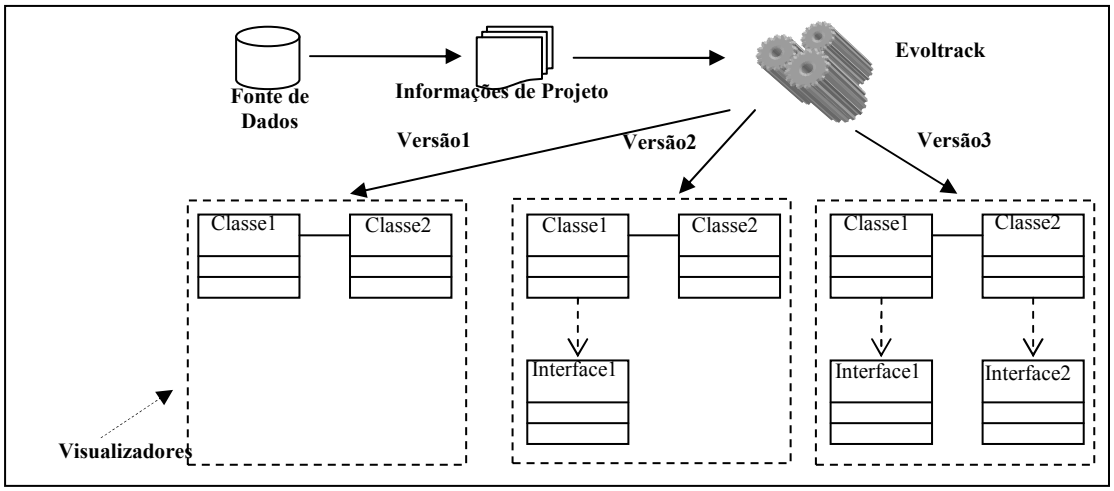



**Figura 2.8 – Uma visão geral da abordagem EvolTrack.**

Desta forma, as informações capturadas a partir de uma dada fonte de dados servirão como base para a criação de toda uma representação do ciclo de evoluções do projeto de software por parte do EvolTrack. Portanto, para viabilizar este esquema de representação, uma implementação do meta-modelo da UML foi adotada na construção do mecanismo, visando refletir um dado instante do projeto. Assim, o ciclo de evolução do projeto é representado internamente como uma seqüência de modelos descritos por este meta-modelo. Em um cenário típico de utilização do mecanismo, pode ser observado que, à medida que o mesmo detecta (ou é informado a partir da fonte de dados) que o projeto sob análise sofreu uma evolução, um novo modelo será criado de forma a representar a nova

configuração para este projeto. Este novo modelo será, então, enviado para o elemento Visualizador para a apresentação visual desta evolução.

Assim, um Visualizador pode ser entendido como um segundo sistema externo capaz de apresentar visualmente as informações do projeto fornecidas pela abordagem EvolTrack. É importante ressaltar que a forma de representação visual adotada pelo visualizador poderá seguir diferentes estilos. Um exemplo de representação visual que poderia ser utilizada neste contexto seria a de diagramas de classe UML (vide Figura 2.9). Porém, qualquer outro tipo de notação visual que consiga representar de maneira satisfatória as informações fornecidas pelo mecanismo também poderia ser adotado.

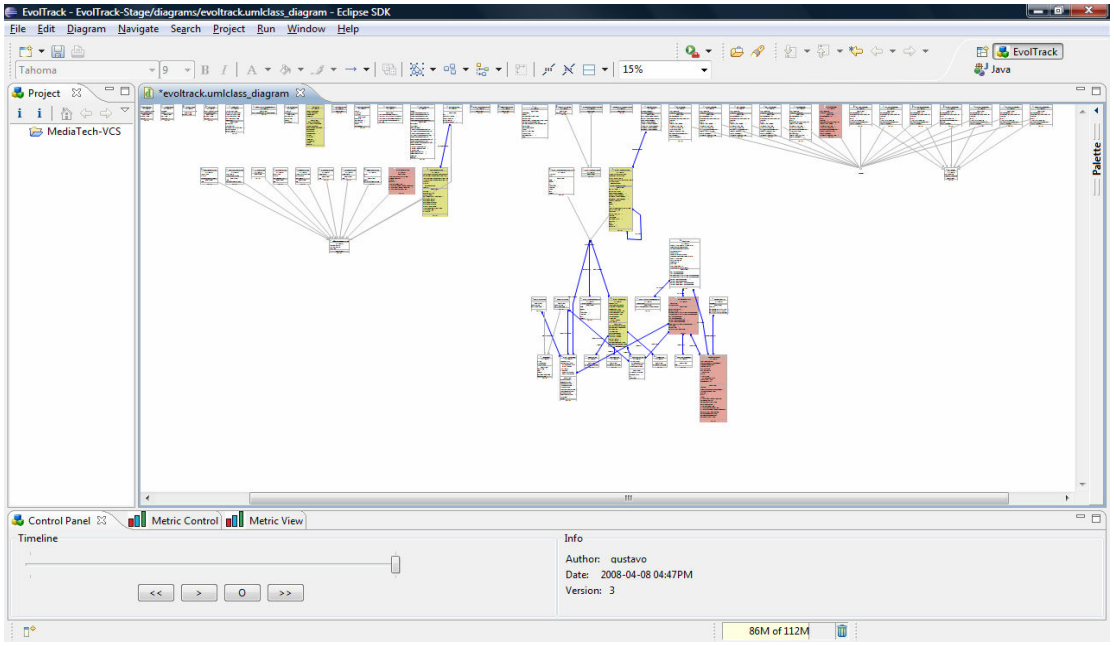



**Figura 2.9 – Ferramenta EvolTrack visualizando um projeto *open source*.**

A visualização disponibilizada pela ferramenta é conhecida como *design emergente*. Isto é, representa uma "foto" atual das entidades e relacionamentos que compõem o sistema em questão. Esta abstração pode ser utilizada para uma melhor compreensão do sistema e de sua evolução. Através de uma análise detalhada, apesar de difícil execução quando o diagrama se torna extenso, é possível obter uma ideia do que está sendo desenvolvido no projeto e por quem, uma vez que a ferramenta apresenta em sua linha do tempo os autores de cada evolução do sistema.

## 2.4. Mecanismos de Percepção de Grupo

A produtividade das equipes distribuídas é prejudicada principalmente pelos problemas de comunicação. Além de ser mais rápido e fácil falar com alguém que está próximo, conversar pessoalmente permite perceber expressões faciais, tons de voz e gestos que proporcionam uma comunicação rica e, conseqüentemente, um melhor entendimento entre as partes. O compartilhamento de um mesmo ambiente físico também torna possível perceber, mesmo involuntariamente, muitos aspectos pessoais dos membros do grupo. Depois de alguns dias de convívio já é possível descobrir padrões de horários de chegada e saída dos colegas, assim como ter noção das suas atividades atuais e suas principais competências. Isto permite, por exemplo, que se consiga inferir quais são os melhores horários para encontrar um desenvolvedor "especialista" em determinado artefato, ou alguém que já tenha trabalhado numa tarefa similar e possa oferecer algum auxílio.

A perda desta percepção em equipes distribuídas contribui para o atraso destas em relação a equipes convencionais (Herbsleb e Mockus, 2003). Assim, a percepção (do inglês, *awareness*) pode ser definida como "o entendimento das atividades dos demais que provê um contexto para a sua própria atividade" e é muito importante para a socialização, sendo um dos principais requisitos para que a colaboração seja efetiva (Dourish e Bellotti, 1992). Outros conceitos importantes são derivados da percepção e, muitas vezes, confundidos com a própria, como informação de percepção e mecanismo de apoio à percepção.

Percepção de grupo é o estado de ciência de um indivíduo em relação ao ambiente e aos demais, tendo um caráter subjetivo. Informação de percepção é a informação que pode ser utilizada para melhorar o estado de percepção de um indivíduo. Por exemplo, o conhecimento sobre as atividades dos colegas e dos seus padrões de horário citado acima. Já mecanismos de apoio à percepção são ferramentas que utilizam informações de percepção para tentar apoiar grupos em atividades colaborativas. A seguir, alguns mecanismos de percepção de grupo são apresentados.

### 2.4.1. MAIS

A ferramenta MAIS (Multi-synchronous, Awareness Infrastructure, Infraestrutura 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.

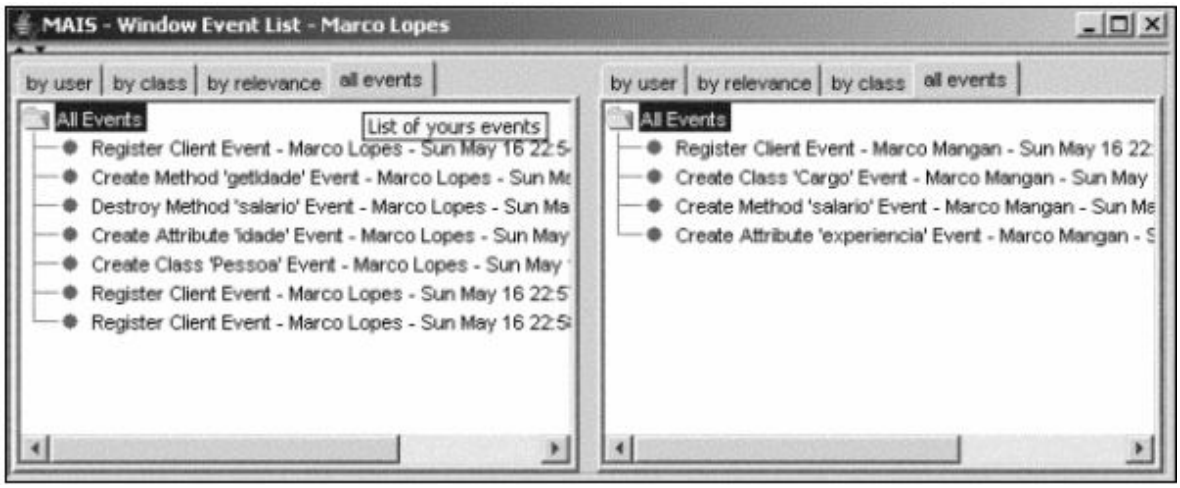



**Figura 2.10 – MAIS apresentando as informações de percepção geradas localmente (esquerda) e remotamente (direita).**

As informações de percepção são apresentadas na interface de usuário da ferramenta em dois painéis (Figura 2.10): o painel 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.

### 2.4.2. GAW

A ferramenta GAW (Mangan et al., 2004) (Da Silva, 2005) explora uma outra forma de representação para as informações coletadas pelo MAIS. Na sua interface de usuário (Figura 2.11), 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 2.11 – GAW apresentando informações coletadas pelo MAIS.**

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.

### 2.4.3. Jazz

O projeto Jazz (Hupfer et al., 2004), da IBM, tem como 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 2.12), 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.

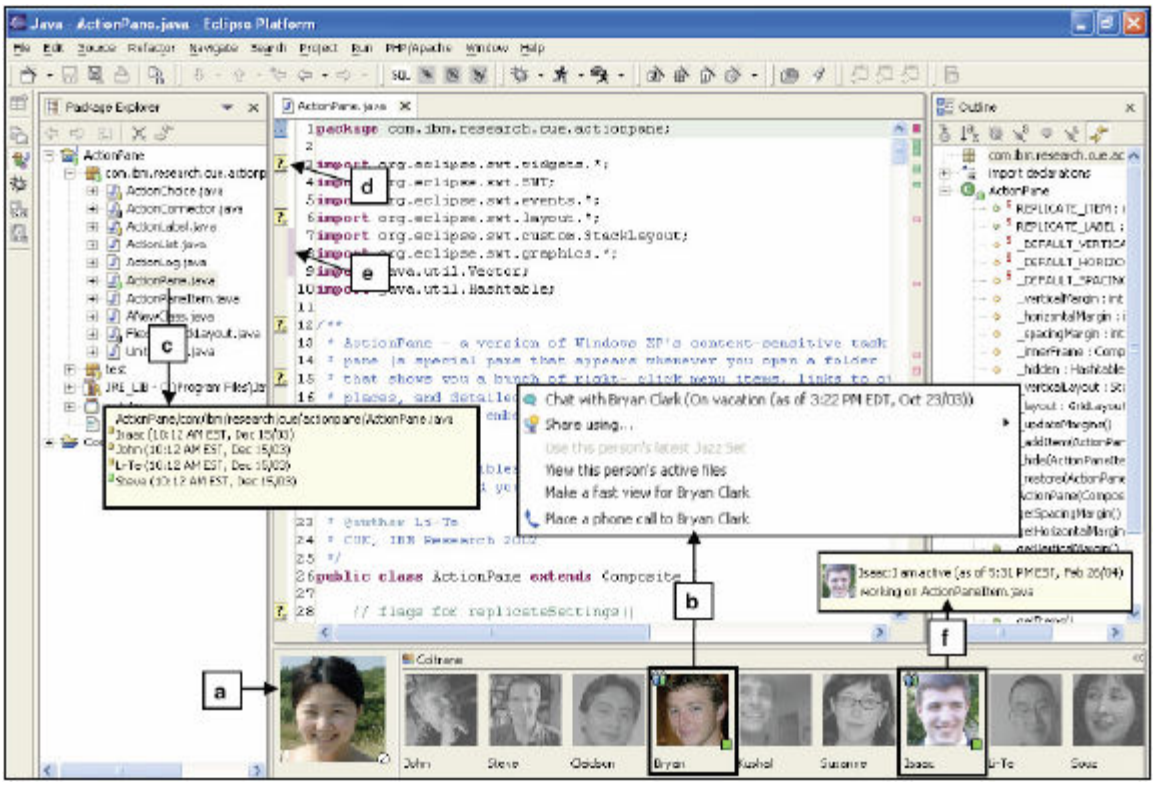



**Figura 2.12 – Jazz no ambiente Eclipse: (a) visualização dos integrantes da equipe; (b) opções de comunicação; (c) decorações na lista de artefatos; (d) e (e) decorações no editor de código-fonte e (f) informações sobre integrante da equipe.**

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.

Atualmente está em desenvolvimento um novo projeto, também denominado Jazz (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 gerência de configuração de software (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.

### 2.4.4. FASTDash

O projeto FASTDash (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 FASTDash tem integração com o ambiente de desenvolvimento Visual Studio com suporte para dois sistemas de gerência de configuração de software, o *Team Foundation Server* 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).

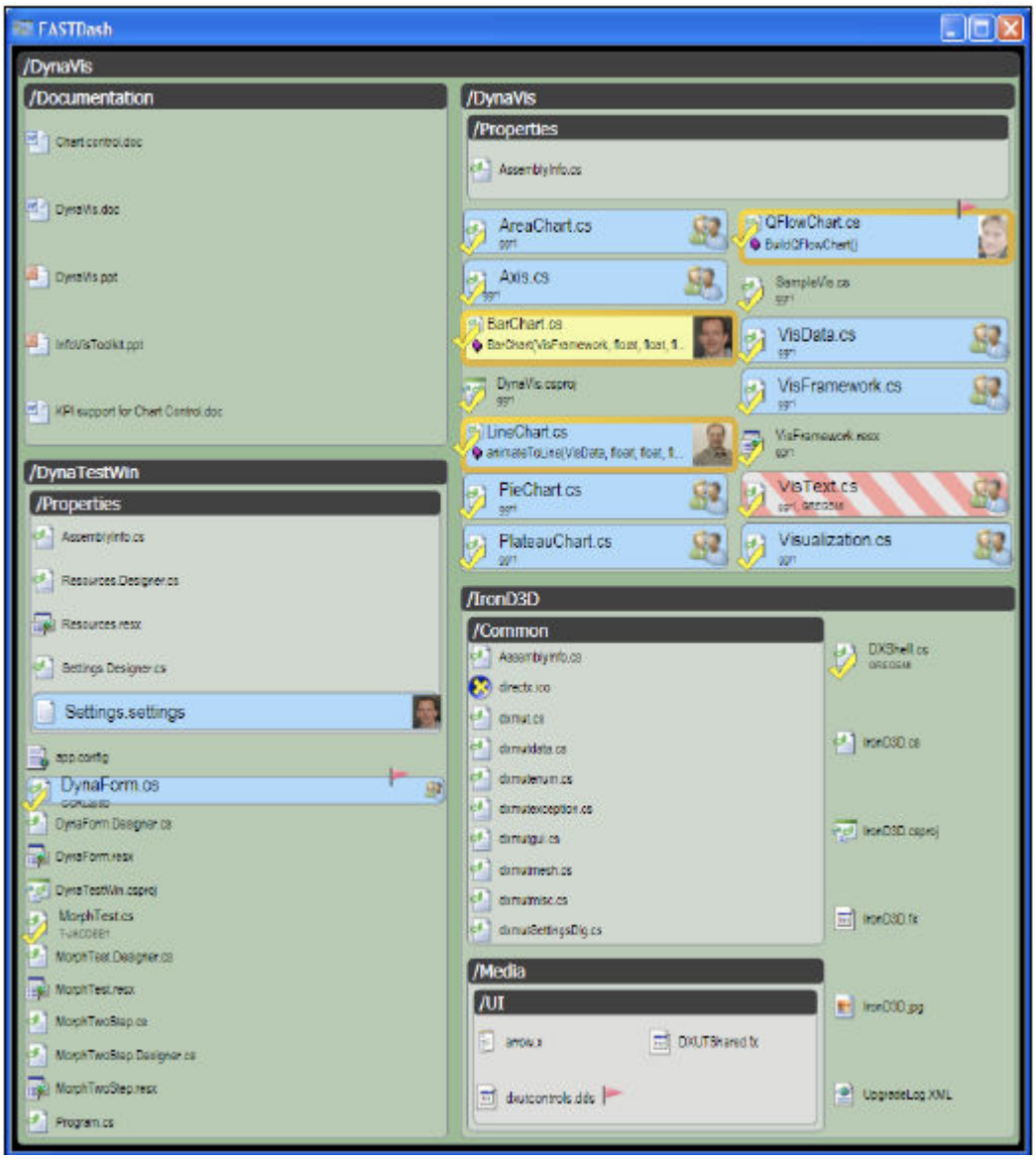



**Figura 2.13 – Visualização dos artefatos compartilhados no Fast Dash.**

Como a ferramenta foi desenvolvida para equipes localizadas, os autores recomendam que essa visualização seja apresentada para a equipe em uma 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.

### 2.4.5. Lighthouse

A ferramenta Lighthouse (Silva et al., 2006) (Da Silva, 2008) se baseia numa abstração da estrutura do sistema de software em desenvolvimento, construída automaticamente a partir do seu código-fonte (Figura 2.14). 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, 2011). 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.

Teoricamente, o *Design Emergente* pode ser decorado com informações de percepção de grupo, coletadas a partir da área de trabalho e do sistema de controle de versão, 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 sistema de controle de versão (se a mudança é local ou se já foi feito *check-in*/*check-out*).

Na abordagem Lighthouse, esse modelo é anotado com informações de percepção que são priorizadas de acordo com o seu potencial de 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.

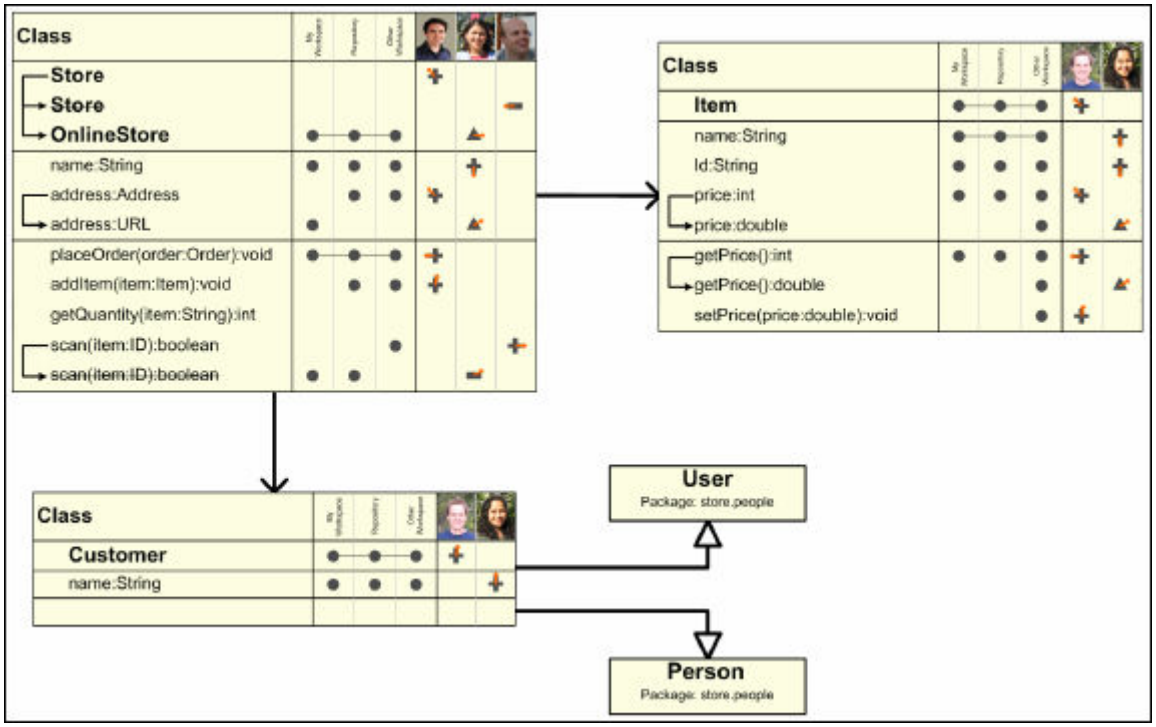



**Figura 2.14 – *Design Emergente* apresentado pela ferramenta Lighthouse.**

Como o Lighthouse é um mecanismo de percepção, este diagrama deve, idealmente, 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*. 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.

## 2.5. Discussão

As abordagens apresentadas na Seção 2.3, focadas principalmente no aspecto da visualização das informações, têm por objetivo principal fornecer uma representação visual a partir de onde seus usuários possam extrair informações acerca do software sendo construído. Como pode ser observado, o tipo de informação disponibilizada quase sempre recai apenas sobre a estrutura do software e sua evolução ao longo do tempo. Isto é, tais abordagens proporcionam um meio pelo qual o software sendo desenvolvido possa ser melhor compreendido e trabalhado. Por exemplo, através dessas abordagens um desenvolvedor pode identificar se determinada parte de um sistema tem sido constantemente modificada ou não.

Entretanto, vale notar que todas as abordagens apresentadas nessa seção utilizam como fonte primária de informação um sistema de controle de versão. Desta forma, o isolamento criado por esse tipo de sistema não é eliminado, isto é, não se sabe ao certo o que cada desenvolvedor faz em seu espaço de trabalho, antes que o mesmo envie suas modificações para o sistema de controle de versão. Isso faz que com que problemas de conflito entre modificações concorrentes, por exemplo, não possam ser detectadas em tempo de desenvolvimento, mas apenas em tempo de *check-in*. Através desse exemplo, observa-se que não há uma percepção das atividades desempenhadas pelo grupo de desenvolvedores e sim uma percepção apenas do software e suas mutações.

Em contrapartida, as abordagens apresentadas na seção 2.4 tentam explorar esta outra faceta do desenvolvimento aos usuários. Esta percepção sobre as atividades dos desenvolvedores pode ser explorada a partir de diferentes tipos de artefatos. Por exemplo, as ferramentas MAIS e GAW utilizam para sua análise os diagramas produzidos ao longo de um desenvolvimento. Já as ferramentas Jazz, FASTDash e Lighthouse utilizam o próprio código-fonte para perspectiva de percepção adotada. Desta forma, através destas últimas, um desenvolvedor pode obter, por exemplo, a informação de quem, além dele mesmo, está

trabalhando em um mesmo código-fonte. Porém, a ausência da informação sobre a estrutura do sistema sendo construído, em algumas dessas abordagens, enfraquece a capacidade de se extrair percepções da mesma. Por exemplo, como identificar visualmente que uma determinada modificação em um artefato tem o potencial de impacto em um outro artefato, se a informação de dependência entre artefatos não é apresentada?

A abordagem Lighthouse resolve este problema fornecendo em conjunto informações referentes as atividades (isto é, o que está acontecendo em um dado artefato) e informações sobre a própria estrutura do sistema. Entretanto, dois pontos principais, evidenciados por um estudo executado com a ferramenta (Da Silva, 2008), revelam ainda algumas deficiências de tal abordagem. Um dos aspectos deficientes dessa abordagem é justamente a utilização de técnicas de visualização adequadas para a apresentação das informações. Por exemplo, observa-se que o uso de cores para diferenciar artefatos alterados, criados ou excluídos não foi explorado pela ferramenta, o que dificulta a percepção dessa essencial informação sobre o projeto. Isto ocorre principalmente quando o sistema sendo construído produz um diagrama relativamente grande, onde uma inspeção detalhada de cada artefato no diagrama se torna custosa.

Outra questão pouco explorada por essas ferramentas da Seção 2.4 é a inferência de informações a partir do estado atual do desenvolvimento. Usualmente, são apresentadas informações vindas diretamente do contexto, ou seja, classes sendo trabalhadas, atributos sendo modificados, arquivos sendo excluídos, ou pessoas se comunicando no projeto, por exemplo. Porém, em muitos casos, uma análise mais elaborada do ambiente ainda poderia ser realizada. Aparentemente, dentre as ferramentas pesquisadas, apenas o FASTDash indica ativamente (colorindo o artefato em questão) uma situação de conflito, quando um determinado arquivo foi alterado por duas pessoas simultaneamente. Outras análises, como impacto de modificações e conflitos indiretos não são exploradas pelas ferramentas investigadas.

Por fim, é importante notar que nenhuma das abordagens supracitadas provê um mecanismo capaz de ser reutilizado em outras abordagens para fins semelhantes. Isto é, ninguém se preocupou em definir e implementar uma infra-estrutura mínima onde abordagens com objetivo de extrair percepções e visualização de projetos de software pudessem utilizar. Desta forma, todas praticamente “reinventam a roda” ao implementar

métodos de recuperação e disponibilização de dados provenientes de fontes de dados comumente utilizadas, como sistemas de controle de versão e ambientes de desenvolvimento de software.

A Tabela 2.1 resume as principais características analisadas em cada ferramenta. Uma adaptação do *framework* desenvolvido por Storey et al. (2005), para a comparação de ferramentas de visualização de software no apoio a percepção de atividades, foi utilizado nesta seção. A tabela é dividida em sei colunas, onde a primeira representa a ferramenta sendo analisada. As demais colunas identificam as características comparadas: tipo de informação, fonte, apresentação, local, e público-alvo. O tipo de informação diz respeito a natureza da informação que é fornecida pela ferramenta. Isto é, que tipo de informação ela provê? A segunda característica está relacionada com a fonte ou as fontes que foram utilizadas na obtenção dessas informações (e.g., sistema de controle de versão (SCV), ambiente desenvolvimento). A seguir, a forma como esta informação é apresentada ao usuário é descrita. O local dessa apresentação é, então, indicado e, posteriormente, a pessoa ou as pessoas a quem a ferramenta se destina é exposto.

**Tabela 2.2 – Comparação entre as ferramentas pesquisadas.**

| Ferramenta | Tipo de Informação | Fonte | Apresentação | Local | Público-alvo |
|---|---|---|---|---|---|
| **Augur** | Código-fonte; Autoria das modificações; Estatísticas sobre a evolução dos artefatos | SCV; Código-fonte | Visualização baseada em linhas de código; Visões com gráficos estatísticos | Aplicação Independente | Desenvolvedores |
| **CVSScan** | Histórico de mudanças dos artefatos; Autoria das modificações | SCV (CVS) | Visualização baseada em linhas de código | Aplicação Independente | Desenvolvedores |
| **softChange** | Histórico de mudanças dos artefatos; Estatísticas sobre a evolução dos artefatos | SCV (CVS); Sistema de rastreamento de defeitos (Bugzilla); E-mail; Código-fonte | Componente de Hipertexto; Histogramas sobre a evolução dos artefatos; Gráfico baseado nas modificações | Aplicação Independente | Desenvolvedores; Pesquisadores |

| | | | | | |
|---|---|---|---|---|---|
| | | | dos artefatos | | |
| **EvolTrack** | Estrutura dos artefatos; Autoria das evoluções no SCV | SCV | Diagrama de classes disposto em uma linha do tempo | Ambiente de desenvolvimento | Desenvolvedores |
| **MAIS** | Histórico de mudanças dos artefatos | Ambiente Odyssey | Lista de eventos | Aplicação Independente | Analistas; Desenvolvedores |
| **GAW** | Histórico de mudanças dos artefatos | Ambiente Odyssey | Linha do tempo colorida baseada para cada indivíduo | Aplicação Independente | Analistas; Desenvolvedores |
| **Jazz** | Histórico de mudanças dos artefatos; Percepção de presença de usuários; Histórico de comunicação | SCV; Ambiente de desenvolvimento; Ferramenta de comunicação | Lista de artefatos decorada; Lista de indivíduos decorada; Decorações no editor de artefatos | Ambiente de desenvolvimento | Desenvolvedores |
| **FastDash** | Histórico de mudanças dos artefatos | SCV; Ambiente de desenvolvimento | Painel Hierárquico de artefatos decorados | Aplicação Independente | Desenvolvedor |
| **Lighthouse** | Histórico de mudanças dos artefatos; Estrutura dos artefatos; | SCV | Diagrama decorado com informações do histórico de modificações | Ambiente de desenvolvimento | Desenvolvedor |

## 2.6. Considerações Finais

Neste capítulo, foi apresentado brevemente o conceito de desenvolvimento distribuído de software, incluindo alguns de seus principais desafios e soluções já adotadas pela comunidade. Neste contexto, foi discutido o importante papel que a falta de percepção sobre um determinado projeto causa ao longo do seu desenvolvimento. Mostrou-se que esta falta de percepção não recai apenas sobre artefatos, mas também sobre as pessoas e

atividades envolvidas. Desta forma, inicialmente foi apresentado como a área de visualização de software tem sido empregada na obtenção de certos tipos de percepção acerca de projetos de software (Storey et al., 2005).

Considerando que tais abordagens não proporcionam uma percepção realmente eficaz no contexto do desenvolvimento distribuído, outras abordagens foram apresentadas. Essas, sem uma grande preocupação com a visualização em si, proporcionam um fenômeno conhecido como percepção de grupo. Através dessas abordagens, é possível obter informações que antes, apenas com a visualização do software, não seriam facilmente obtidas. Informações como quem faz o quê, o que está sendo modificado, criado ou excluído no momento, ou quem são as pessoas integrantes de um dado projeto são as informações principais obtidas nas chamadas ferramentas de percepção de grupo. Através dessas, uma noção sobre o processo humano envolvido no desenvolvimento de um sistema se torna mais evidente e passível de análise.

Porém, o que pôde ser notado é que ambos casos quando tratados de forma isolada carecem de informações pertinentes ao desenvolvimento distribuído. Até mesmo quando foram tratados de forma conjunta, aspectos importantes que poderiam ser melhor evidenciados, utilizando-se técnicas de visualização, por exemplo, não o foram. Assim, considerando as abordagens estudadas nesse capítulo, incluindo seus problemas e seus pontos positivos, foi criada a abordagem VIEWS, que será apresentada no próximo capítulo.

# Capítulo 3 - A Abordagem VIEWS

## 3.1. Introdução

No capítulo anterior foram apresentadas diferentes abordagens que, utilizando ou não técnicas de visualização de software, tinham por objetivo prover percepção no contexto do desenvolvimento de software. Através de uma análise comparativa realizada a partir destes trabalhos, aspectos positivos e negativos foram detectados, bem como pontos comuns e pontos divergentes. Alguns aspectos positivos encontrados na maioria destes trabalhos, como integração com um ambiente de desenvolvimento e a integração com alguma fonte de dados de ampla utilização, como, por exemplo, um sistema de controle de versão (SCV), serviram de pré-requisitos técnicos importantes, uma vez que sua relevância neste tipo trabalho se mostra presente na maioria das ferramentas analisadas no capítulo anterior.

Já os aspectos negativos ou limitantes de algumas destas abordagens serviram como alicerce inicial para a proposta deste trabalho de pesquisa. Um destes aspectos é a falta de uma infra-estrutura comum reutilizável para dados obtidos ao longo do desenvolvimento de software. Nenhum dos trabalhos pesquisados se propôs a definir um modelo de dados, mesmo que inicial, que pudesse ser reutilizado em outras abordagens semelhantes (ou complementares). A falta de tal modelo, desta forma, implica na falta de um componente reutilizável capaz de coletar, armazenar e distribuir informações pertinentes a estes tipos de trabalho, fazendo com que cada um tenha que reinventar este processo comum a todos.

Um outro aspecto detectado foi que poucos trabalhos se preocuparam em correlacionar informações obtidas a partir da estrutura de um sistema com as informações provenientes das atividades de desenvolvimento em si (como por exemplo, quem está desenvolvendo o quê em um dado instante de tempo). Daqueles analisados no capítulo anterior, apenas a ferramenta Lighthouse realiza este tipo de correlacionamento. Com isso, e esta é uma das hipóteses deste trabalho, informações relevantes para o processo de desenvolvimento de software distribuído não podem ser obtidas, muitas vezes, pela simples ausência desse tipo correlação.

Portanto, a abordagem proposta VIEWS visa à atender estes aspectos, fornecendo um componente reutilizável, implementado a partir de um modelo de dados bem definido, capaz de coletar, armazenar e distribuir diferentes tipos de dados obtidos ao longo da implementação de um software. A forma como estes dados são armazenados e distribuídos permite que uma aplicação cliente (que pode ser qualquer outro trabalho interessado em derivar ou representar informações pertinentes ao desenvolvimento do software) possa correlacionar dados obtidos diretamente da estrutura do software com dados obtidos de suas atividades de desenvolvimento, produzindo assim informações que antes não poderiam ser facilmente identificadas.

O restante deste capítulo está dividido da seguinte forma: a Seção 3.2 apresenta uma visão geral da abordagem e seus principais componentes; a Seção 3.3 detalha a etapa de coleta de dados e como estes são repassados para a etapa posterior; a Seção 3.4 apresenta o modelo de dados utilizado para o armazenamento dos dados obtidos na etapa de coleta; a Seção 3.5 apresenta a camada de serviços disponibilizada para a distribuição dos dados; e, por fim, a Seção 3.6 apresenta as considerações finais e discorre sobre algumas das limitações da abordagem proposta.

## 3.2. Visão Geral

Conforme ressaltado na seção anterior, a abordagem proposta deve fornecer uma infra-estrutura (chamado anteriormente de componente reutilizável) capaz de realizar as etapas comuns do processo de criar percepção no desenvolvimento de software. Estas etapas podem ser resumidas em: *coleta*, *armazenamento* e *distribuição*. Além disto, para que o objetivo do trabalho seja plenamente contemplado, na etapa de coleta, tanto os dados referentes a estrutura do sistema, bem como dados referentes ao desenvolvimento do mesmo, deverão ser obtidos e armazenados, de modo que posteriormente possam ser relacionados.

Visto que uma infinidade de tipos de dados poderiam ser obtidos a fim de alcançar este objetivo, a abordagem proposta limitou-se a estabelecer um conjunto inicial de dados, de forma que seus resultados pudessem ser verificados ao final do processo. A natureza destes dados será explicitada na Seção 3.4. Entretanto, vale ressaltar que tanto dados sobre o próprio software (ex.: classes e seus inter-relacionamentos) como sobre as atividades de

desenvolvimento em si (ex.: desenvolvedor que está trabalhando em uma determinada classe) estão sendo contemplados.

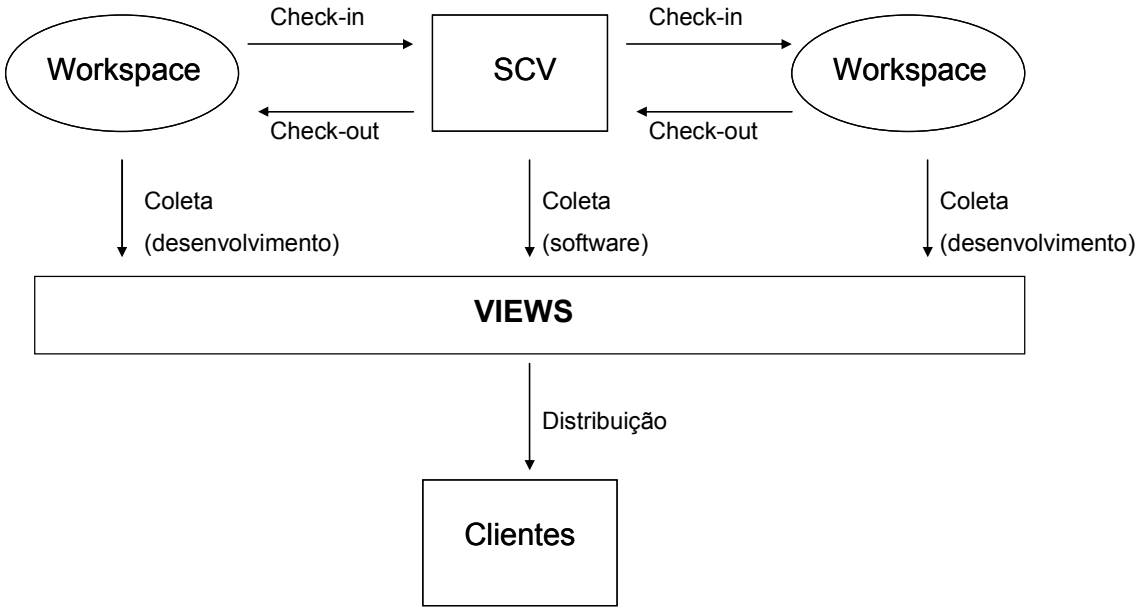



**Figura 3.1. Visão geral da abordagem.**

A Figura 3.1 apresenta uma visão geral da abordagem proposta, ilustrando seus principais elementos. Como pode ser observado na figura, a VIEWS funciona como uma camada intermediária entre clientes interessados em obter (e representar talvez) informações do desenvolvimento de um sistema e a própria realidade do desenvolvimento em si. Esta realidade é representada em conjunto pelos espaços de trabalho de cada desenvolvedor de um projeto (*workspaces*) e o sistema de controle de versão, usualmente utilizado como um repositório central do último estado construído do sistema. Apesar de não ser contemplada por este trabalho, note que conceitualmente nada impede que esta realidade possa ser estendida para conter outros componentes do ciclo de vida de um projeto de software, como sistemas de controle de mudança ou sistemas de rastreamento de erros (*bug tracking systems*).

A partir da realidade determinada, que, no caso deste trabalho, é composta por espaços de trabalho (*workspaces*) e um sistema de controle de versão (SCV), a etapa de coleta é realizada. É nesta etapa que todos os dados do projeto são obtidos e fornecidos para a infra-estrutura. Neste caso, enquanto os dados relacionados ao desenvolvimento são obtidos a partir dos espaços de trabalho em tempo real, os dados do software são obtidos do

sistema de controle de versão sempre que algum desenvolvedor altera o estado do sistema no repositório. Isto é, quando a partir de um espaço de trabalho, detecta-se que um *check-in* fora realizado, a infra-estrutura automaticamente obtém o novo estado do sistema, a partir do sistema de controle de versão. Desta forma, a infra-estrutura se mantém sempre atualizada com a última estrutura construída do software em questão. Exemplos do que pode ser obtido a partir de um espaço de trabalho pode ser a informação de que um determinado desenvolvedor está trabalhando em uma determinada classe no momento, ou até que o mesmo está editando um determinado método neste exato momento.

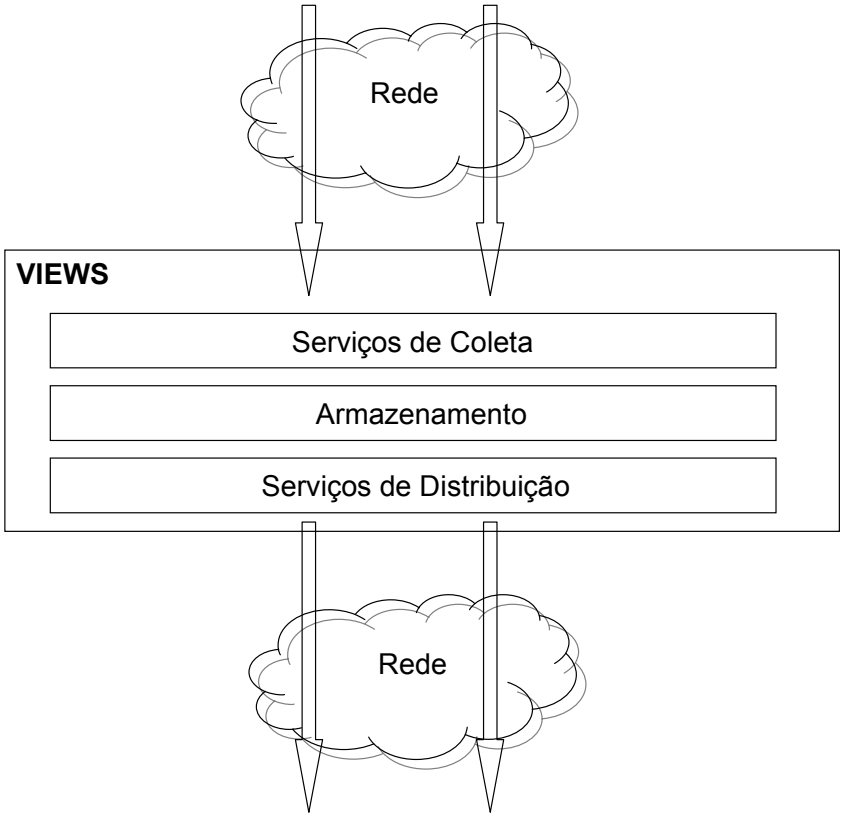



**Figura 3.2. Serviços da infra-estrutura.**

A partir dos dados de entrada, a VIEWS executa um conjunto de atividades de análise que visam a criação das correlações entre os dados. Estas correlações podem ser mapeadas tanto entre elementos do software, como uma dependência funcional ou de herança entre duas classes, assim como também entre os elementos do software e os eventos de desenvolvimento capturados, relacionando, por exemplo, o evento "estar

editando" com a uma classe determinada ou um método. Os diferentes tipos de correlação serão descritos posteriormente neste capítulo.

Com os dados correlacionados e armazenados, a infra-estrutura encontra-se apta para distribuir os mesmos de diferentes formas através de uma interface pública de serviço. A Figura 3.2 ilustra este esquema. Tanto os serviços de coleta como os serviços de distribuição são padronizados em interfaces estáveis e disponibilizados para qualquer um que queira utilizá-las através da rede. Entende-se por serviços de coleta, todos os procedimentos disponibilizados pela infra-estrutura para que fontes de dados quaisquer possam fornecer dados sobre um determinado projeto de software, desde que seja obedecida a interface de comunicação com a infra-estrutura. Da mesma forma, entende-se por serviços de distribuição o conjunto de procedimentos padronizados que são disponibilizados pela infra-estrutura de forma a alimentar aplicações clientes com dados sobre o projeto de software. Qualquer aplicação que respeite a interface de comunicação poderá receber dados de um projeto específico.

## 3.3. Coleta de Dados

A coleta de dados, fase responsável pela obtenção das informações do projeto de software, é contemplada a partir de duas grandes etapas. Em um primeiro momento, eventos gerados dentro do ambiente de desenvolvimento (espaço de trabalho) são monitorados por um componente da abordagem chamado de ***listener*** (*ouvinte*). Obviamente, nem todos os eventos gerados serão úteis para a obtenção de informações sobre o projeto, de modo que muitos deles são simplesmente filtrados. Neste caso, são considerados eventos úteis apenas aqueles eventos referentes aos artefatos e projetos Java, além de operações com o sistema de controle de versão. Estes eventos foram escolhidos porque são as únicas informações que devem ser monitoradas para que o resultado final da abordagem seja obtido. As seções a seguir apresentam uma breve descrição de cada um destes eventos úteis na abordagem.

### 3.3.1. Criação de um espaço de trabalho (*workspace*)

Este evento representa o ponto de partida para todo trabalho da ferramenta. É aqui que um projeto específico é criado na VIEWS e passa a ser monitorado pelo *listener*. Na verdade, este evento pode ou não resultar na criação de um novo projeto na VIEWS. Caso este evento tenha sido gerado no ambiente de desenvolvimento fruto de um *check-out*, o componente *listener* apenas associa na VIEWS o novo espaço de trabalho ao projeto já existente no sistema de controle de versão (i.e., projeto este que fora objeto do comando de *check-out*). Repare que neste caso, considera-se como um espaço de trabalho o projeto local criado no ambiente de desenvolvimento do desenvolvedor. Normalmente, este projeto local (espaço de trabalho) estará associado a um projeto remoto (também denominado neste trabalho como *software*), armazenado no sistema de controle de versão, a não ser que a partir deste espaço de trabalho recém criado saia a primeira versão do projeto em questão. Esta dinâmica fica um pouco mais clara a partir do diagrama de estados representado pela Figura 3.3.

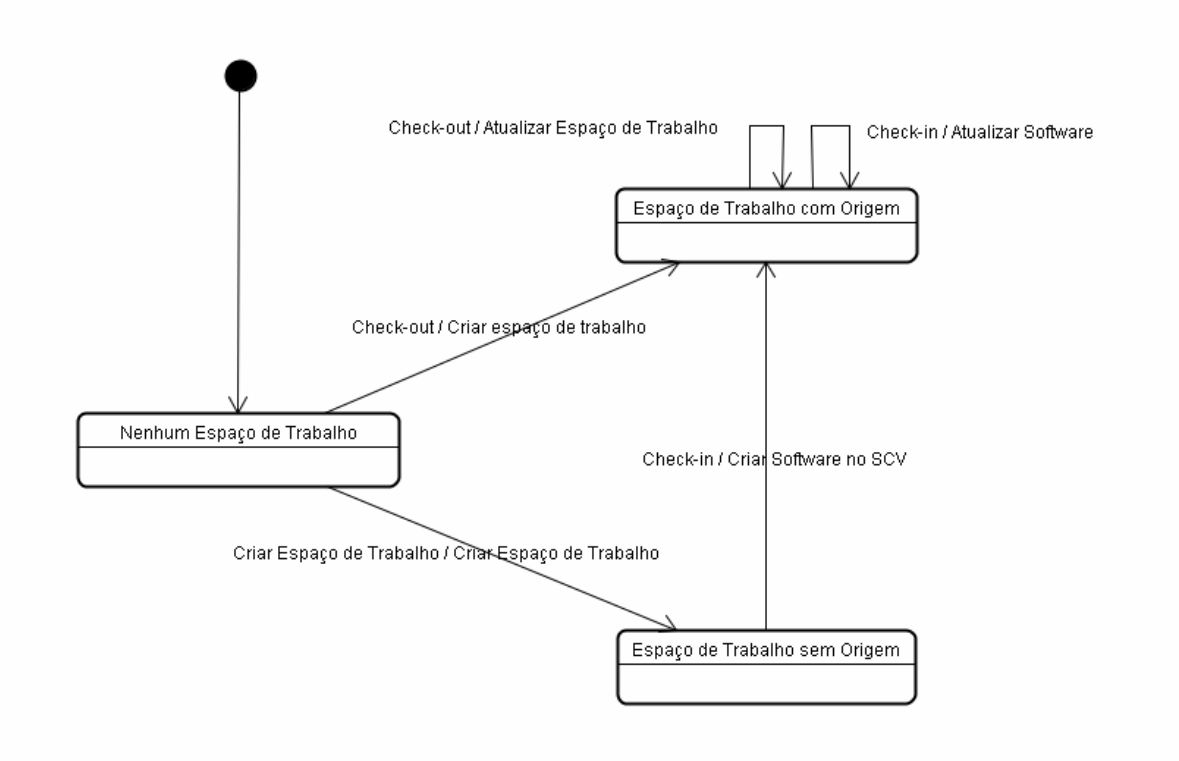



**Figura 3.3. Diagrama de estados do ciclo de vida de um espaço de trabalho.**

Através da observação da Figura 3.3, fica claro que a criação de uma espaço de trabalho pode ser feita a partir de dois eventos distintos. O evento de *check-out* gera um novo espaço de trabalho local, associando o mesmo ao software obtido do repositório. Já um evento de criação de espaço de trabalho normal (i.e., evento gerado a partir de um comando específico executado no próprio ambiente de desenvolvimento) criará um novo espaço de trabalho local, sem associá-lo a qualquer outra fonte. Apenas quando um evento de *check-in* é detectado no projeto local, é que o mesmo é, finalmente, associado ao software (projeto remoto) originado pelo próprio evento de *check-in* no sistema de controle de versão.

Assim, vale ressaltar que, neste trabalho, o conceito de espaço de trabalho está estritamente ligado ao de projeto local e não ao de ambiente de desenvolvimento como um todo. Portanto, se um dado desenvolvedor possui quatro projetos locais criados em seu ambiente de desenvolvimento, este também estará associado a quatro espaços de trabalho distintos na infra-estrutura VIEWS, assumindo que todos os quatros foram colocados em monitoração pelo *listener* da abordagem.

### 3.3.2. Criação / Exclusão / Alteração de arquivos

Estes eventos representam as operações básicas que podem ser detectadas em todos os arquivos de um espaço de trabalho. Através destas, por exemplo, a VIEWS consegue identificar quem está trabalhando em um determinado arquivo (digamos, uma especificação de casos de uso) em um dado momento. Além disto, pelo fato de ser detectado em tempo real, a infra-estrutura consegue antecipar futuros estados do repositório. Por exemplo, suponha que o desenvolvedor A esteja editando o arquivo X em um dado momento. Além disto, suponha também que o desenvolvedor B esteja editando o mesmo arquivo X em seu espaço de trabalho. Quando os dois eventos de edição detectados em separado (um em cada espaço de trabalho) chegam na infra-estrutura VIEWS, esta gera uma **percepção** chamada de "**Situação de Conflito**".

Esta percepção, bem como todas as outras geradas pela ferramenta, nada mais é do que uma correlação entre os dados de desenvolvimento oriundos de diferentes espaços de trabalho. As diversas correlações previstas na ferramenta são chamadas de percepções no

contexto deste trabalho. Outras percepções geradas a partir destes eventos de criação, exclusão ou alteração de arquivos são, por exemplo, "**Situação de Retrabalho**" e "**Exclusão de Arquivo Modificado**". A Seção 3.4 irá detalhar estas e outras percepções geradas pela ferramenta.

### 3.3.3. Criação / Exclusão / Alteração de classes

Esta classe de eventos se assemelha muito com os eventos descritos na seção anterior. Entretanto, ao invés de arquivos sem qualquer semântica para a infra-estrutura, este conjunto de eventos representa as operações básicas que podem ser realizadas em uma classe de software. O evento de criação de classe produz uma nova classe no espaço de trabalho. O evento de exclusão de classe remove uma determinada classe do espaço de trabalho. Já o evento de alteração de classe, caracterizado por um conjunto de operações realizadas dentro do contexto da classe, indica para a infra-estrutura que uma determinada classe, pertencente a um determinado espaço de trabalho, está sendo editada pelo desenvolvedor associado a este espaço de trabalho.

### 3.3.4. Criação / Exclusão / Alteração de métodos

Estes eventos refletem a dinâmica das operações realizadas sobre os métodos de uma determinada classe. Através destes eventos, a infra-estrutura é informada, por exemplo, quando um novo método é criado em uma classe antes mesmo que esta classe alterada seja submetida ao sistema de controle de versão. Analogamente, o evento de exclusão de método também informa à infra-estrutura tal ocorrência antes mesmo que esta se materialize ao ser enviado ao repositório. Com isso, diferentes percepções puderam ser criadas a partir destas informações. Uma destas é a "**Situação de Conflito Indireto**". Neste caso, a informação da alteração da assinatura de um método é correlacionada com a informação estrutural do software de dependência entre classes por chamada de método (percepção "**Dependência Funcional**").

### 3.3.5. Criação / Exclusão / Alteração de atributos

Estes eventos refletem a dinâmica das operações realizadas sobre os atributos de uma determinada classe. Através destes eventos, a infra-estrutura é informada, por

exemplo, quando um novo atributo é criado em uma classe antes mesmo que esta classe alterada seja submetida ao sistema de controle de versão. Analogamente, o evento de exclusão de atributo também informa a infra-estrutura tal ocorrência, antes mesmo que esta se materialize ao ser enviada ao repositório. Estas operações mantêm válida a percepção na infra-estrutura de que uma determinada classe está em edição.

### 3.3.6. Realização de *check-in*

Este evento é interceptado sempre que uma operação de *check-in* é realizada no ambiente de desenvolvimento. Ele serve, principalmente, para informar a infra-estrutura que uma nova versão (ou estado) do software foi disponibilizada no sistema de controle de versão. De posse desta informação, a infra-estrutura se encarrega de obter a nova versão diretamente do repositório e atualizar sua representação interna do próprio software em si. A Figura 3.4 ilustra o processo anteriormente citado.

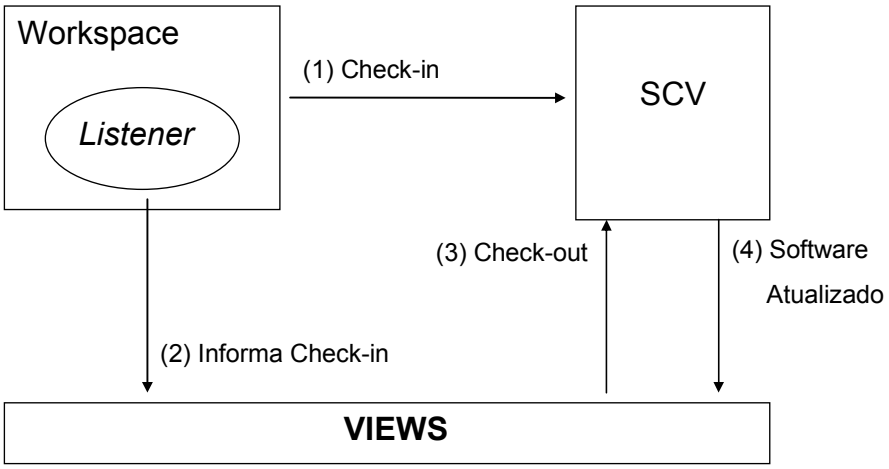



**Figura 3.4. Atividades envolvidas após um *check-in* realizado no espaço de trabalho.**

Esta atualização da representação interna se dá a partir do código fonte obtido do sistema de controle de versão. Este código passa por um processo de engenharia reversa estática (Chikofsky e Cross II, 1990), resultando em uma representação do software (Seção 3.4). Naturalmente, este processo de engenharia reversa é dependente de linguagem de programação. Em função disto, a arquitetura implementada (vide Capítulo 4) está preparada para suportar um número variado de linguagens de programação, de modo que a extensão das funcionalidades da infra-estrutura seja facilitada. No contexto deste trabalho, como será

apresentado no próximo capítulo, apenas uma linguagem de programação é suportada, a saber, Java.

Um outro aspecto relevante neste contexto é sobre o mapeamento entre os projetos locais e o projeto remoto (software) armazenado no repositório. Note que nada garante que os mesmos projetos locais, em cada espaço de trabalho individual, tenham o mesmo nome. Isto é, o desenvolvedor A pode chamar de projeto "Terra" o que o desenvolvedor B chama de projeto "Planeta" e mesmo assim ambos estarem trabalhando sobre o mesmo projeto de software. Desta forma, a abordagem cria um identificador único para cada software criado no sistema de controle de versão e associa este identificador a cada projeto local correspondente. Assim, a infra-estrutura sabe que, apesar de possuírem nomes locais distintos, todos os projetos associados a um mesmo identificador pertencem ao mesmo projeto remoto, isto é, representam o mesmo software em desenvolvimento. É este identificador que é usado para buscar corretamente o projeto no sistema de controle de versão quando um evento de *check-in* é detectado em algum espaço de trabalho.

### 3.3.7. Realização de *check-out*

Este evento é detectado quando um comando de *check-out* é executado no ambiente de desenvolvimento. A partir dele o componente *listener*, que se encontra instalado no próprio ambiente, informa a infra-estrutura que um novo espaço de trabalho deverá ser criado para o desenvolvedor em questão. Caso seja feita apenas uma operação de *check-out* em um arquivo específico (ou update), o *listener* apenas irá registrar o evento na infra-estrutura sem maiores conseqüências.

Repare que neste caso (assim como em vários outros momentos citados), o *listener* deverá de alguma forma saber que o ambiente de desenvolvimento no qual ele se encontra instalado e em execução pertence a um determinado desenvolvedor. Isto é feito de forma prévia através de um simples cadastro feito no próprio ambiente. Ao final deste cadastro, o *listener* informa a infra-estrutura da existência deste novo desenvolvedor que, ao final do processo de registro, receberá um identificador único na VIEWS que servirá para comunicar todas as operações anteriormente citadas.

## 3.4. Dados e Percepções

Os dados obtidos da etapa de coleta serão armazenados em uma estrutura única, capaz de representar de forma integrada tanto a dimensão estrutural (estática) do software como a dimensão da dinâmica do seu desenvolvimento. Esta estrutura pode ser entendida como um modelo canônico capaz de abstrair de aplicações cliente detalhes de representação de dados expostos por cada tipo de fonte de informação.

A Figura 3.5, apresentada a seguir, mostra como este modelo foi idealizado. Inicialmente, quando a infra-estrutura recebe uma solicitação de criação de um desenvolvedor, um elemento *Developer* é criado e unicamente identificado em toda abordagem. A partir deste, todas as outras operações serão realizadas. Isto é, quando um novo projeto é criado (seja via *check-out* ou criação a partir de comando da IDE), o mesmo é associado a um elemento do tipo *Workspace*. Cada artefato (i.e., arquivos) gerado dentro do ambiente de desenvolvimento é, então, mapeado como um *Artifact* pertencente ao *Workspace* previamente criado. Note que, cada artefato estará associado a um tipo de artefato (por exemplo, arquivo Java ou modelo UML) cadastrado na ferramenta, mapeado através dos diferentes *ArtifactType*.

Cada *ArtifactType*, por sua vez, estará diretamente associado a um componente do tipo *ArtifactReader*. Desta forma, para cada tipo de artefato compreendido pela ferramenta, deverá haver ao menos um *ArtifactReader* associado, de modo que artefatos deste determinado tipo possam ser plenamente entendidos. Assim, por exemplo, se a infra-estrutura deseja interpretar corretamente arquivos de código fonte escritos em linguagem Java, a mesma deverá possuir um *ArtifactReader* para Java (e.g., *JavaReader*). Através deste mecanismo de mapeamento de artefatos em tipos de artefatos e seus respectivos interpretadores, a abordagem encontra-se preparada, ao menos arquiteturalmente, para extrair informações de diferentes tipos de artefatos.

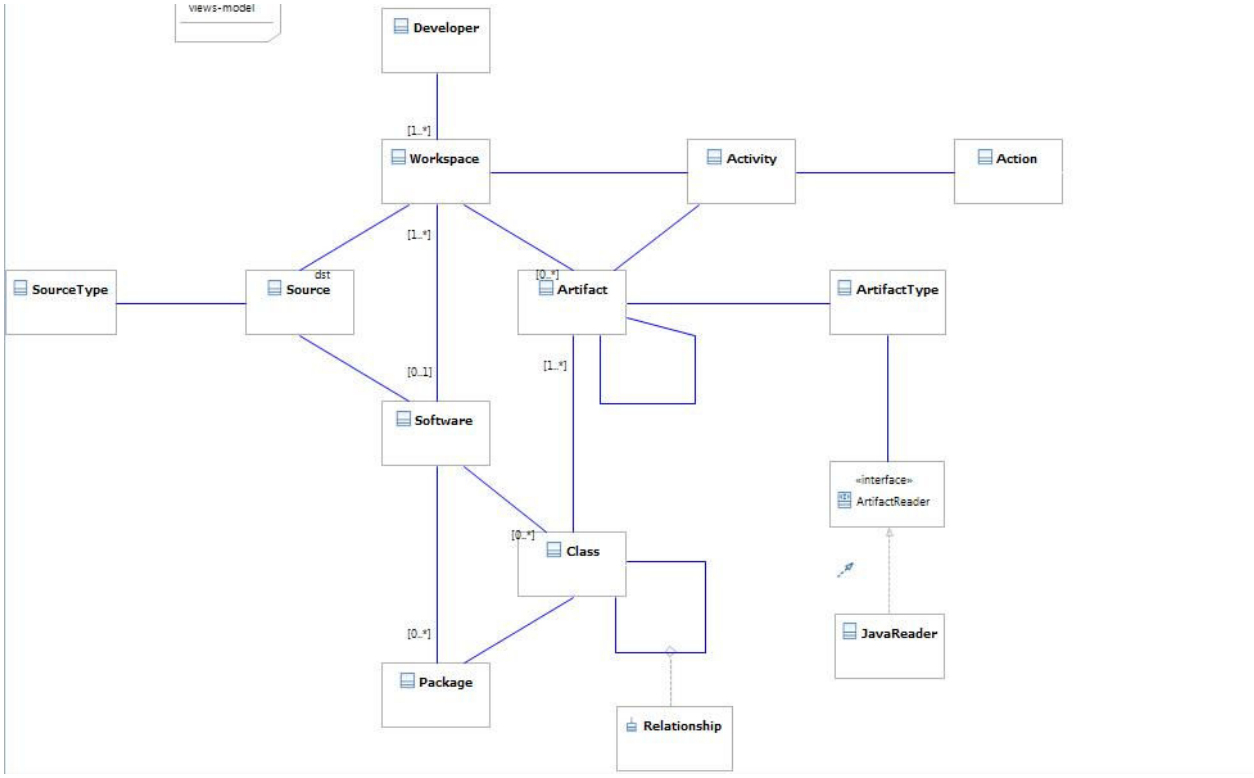



**Figura 3.5. Modelo de dados adotado pela abordagem.**

Estes interpretadores, então, serão responsáveis muitas vezes por realizar os demais mapeamentos do modelo. Ainda no exemplo supracitado, o componente *JavaReader* será responsável pela engenharia reversa que resultará na identificação das classes (*Class)*, pacotes (*Package*) e seus demais atributos e relacionamentos (*Relationship*). Futuramente, outros trabalhos poderão estender a abordagem, de modo que outros tipos de artefatos, como modelos UML, também possam servir como fonte de informação para o modelo de dados unificado da infra-estrutura.

Além disto, cada ação executada no ambiente de desenvolvimento que seja compreensível pela infra-estrutura é mapeada, em tempo de desenvolvimento, em um elemento chamado de *Action*. A principal característica deste tipo de ocorrência é a sua atomicidade. Isto é, caracteriza-se por ser um evento gerado por um simples comando do desenvolvedor, ao contrário do conceito de atividade que será visto a seguir. O conjunto de

ações identificadas e mapeadas por esta abordagem encontra-se previamente descrito nas Seções 3.3.1 à 3.3.7.

Já o conceito de atividade (*Activity*) está associado àqueles eventos que possuem uma duração no tempo. Isto é, são ocorrências que têm inicio, meio e fim bem definidos na escala temporal. Exemplos de atividades são: "Edição de arquivo", "Edição de classe" e "Codificação de método". Cada atividade desta, por sua vez, possui uma data e hora de inicio, data e hora de fim, e outros dados associados. Então, por exemplo, uma simples consulta das atividades que não possuem data e hora de fim, para um determinado *Workspace*, revelam a percepção de todas as atividades sendo executadas no momento por um dado desenvolvedor (em um dado projeto).

Outras percepções podem ser identificadas através do cruzamento de diferentes tipos de informações. A seguir são apresentadas todas as percepções criadas pela abordagem, bem como estas são geradas a partir das informações colidas nas etapas anteriores:

- **Situação de Conflito** – Esta percepção é identificada sempre que dois desenvolvedores estiverem com uma atividade de "Edição de Classe" associada a uma mesma classe do projeto. Note que esta é uma situação **potencial**, isto é, esta percepção não indica que os dois desenvolvedores alteraram a mesma linha do código, apenas que ambos estão trabalhando sobre o mesmo artefato;

- **Situação de Conflito Indireto** – Esta percepção é fruto de uma análise um pouco mais elaborada. Sempre que uma "Codificação de Método" contendo a alteração de sua assinatura é detectada em uma determinada classe, todas as classes que possuem chamadas de método (i.e., "Dependência Funcional por Método") para a mesma serão marcadas com uma situação de conflito indireto. Isto é, apesar da modificação ter sido realizada em uma classe diferente (não configurando assim uma situação de conflito usual), esta modificação poderá ter alterado a forma com que outras classes interagem com a mesma.

Suponha, por exemplo, o seguinte cenário:

**Espaço de Trabalho do Desenvolvedor Rafael**

```
public class Investimento {
    public Double TAXA_DE_JUROS = 0.6;
    public Double INFLACAO = 5.0;
    public Double INVESTIMENTO_INICIAL = 1000.0;
    List<Aplicacao> aplicacoes = new ArrayList<Aplicacao>();

    public Double calculaInvestimento() {
        Double total = 0.0;

        for(Aplicacao aplicacao : aplicacoes) {
            total = total + aplicacao.calcula(TAXA_DE_JUROS, INFLACAO,
                    INVESTIMENTO_INICIAL);
        }

        return total;
    }

    public void adicionaAplicacao(Aplicacao aplicacao) {
        aplicacoes.add(aplicacao);
    }
}
```

**Espaço de Trabalho do Desenvolvedor Marcelo**

```
public class Aplicacao {

    public Double calcula(Double juros, Double inflacao,
            Double investimento) {

        Double inflacaoMes = inflacao/12;
        return investimento * ((1+juros)/(1+inflacaoMes));
    }
}
```

**Figura 3.6. Classes Investimento e Aplicação antes da alteração.**

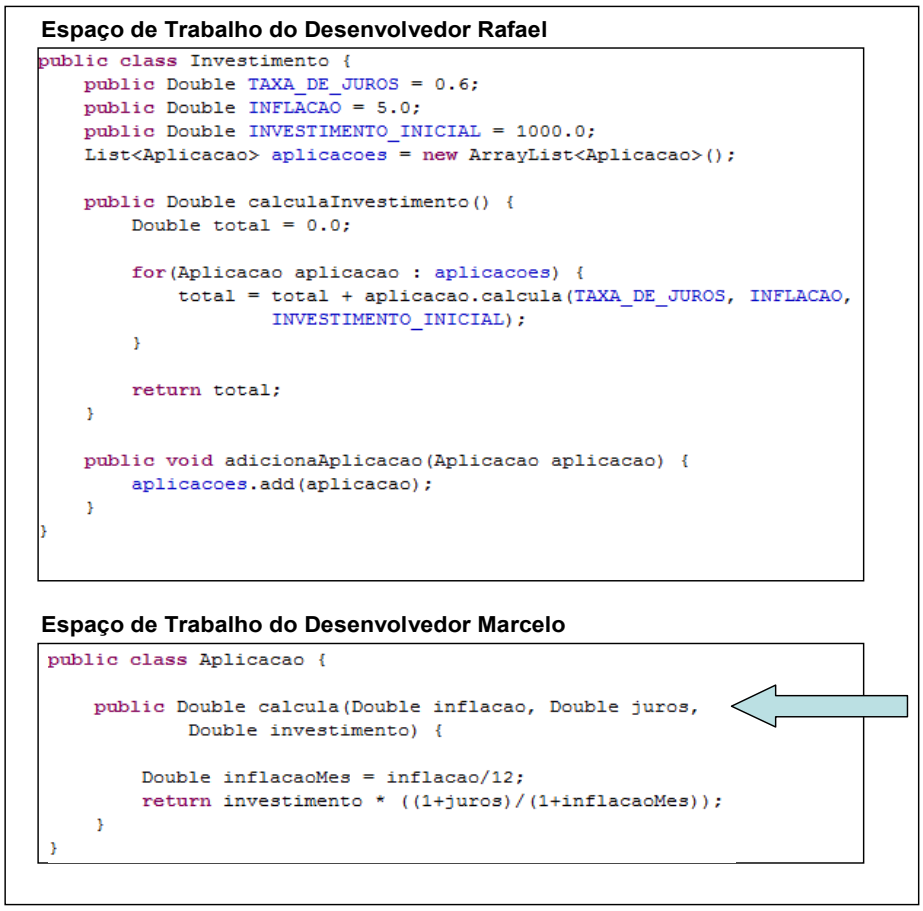



**Figura 3.7. Classes Investimento e Aplicação após a alteração.**

Neste cenário, a classe *Investimento* encontra-se em conflito indireto com a classe *Aplicação*. Note que apesar do exemplo ter apontado apenas dois espaços de trabalho distintos, espaço do Rafael e do Marcelo, todos os espaços que possuem esta classe sofrerão os mesmos efeitos, inclusive o próprio espaço de trabalho do Marcelo. Repare que nesta situação, a assinatura do método *calcula* da classe *Aplicação* foi alterada. Entretanto, como todos os parâmetros deste método são do tipo *Double* a classe *Investimento* (que possui uma "Dependência Funcional por Método") não sofre qualquer tipo de problema sintático, apesar de passar a calcular de forma errada o valor da aplicação financeira.

A falta de um mecanismo capaz de identificar este tipo de problema rapidamente, muitas vezes, acarreta em atrasos no projeto, já que o mesmo é freqüentemente detectado apenas em tempo de testes.

- **Situação de Retrabalho** – Esta percepção indica uma possível fonte de retrabalho. A infra-estrutura cria esta percepção quando dois métodos com a mesma assinatura são criados em uma mesma classe de forma independente em espaços de trabalho diferentes. A Figura 3.8 ilustra esta situação.

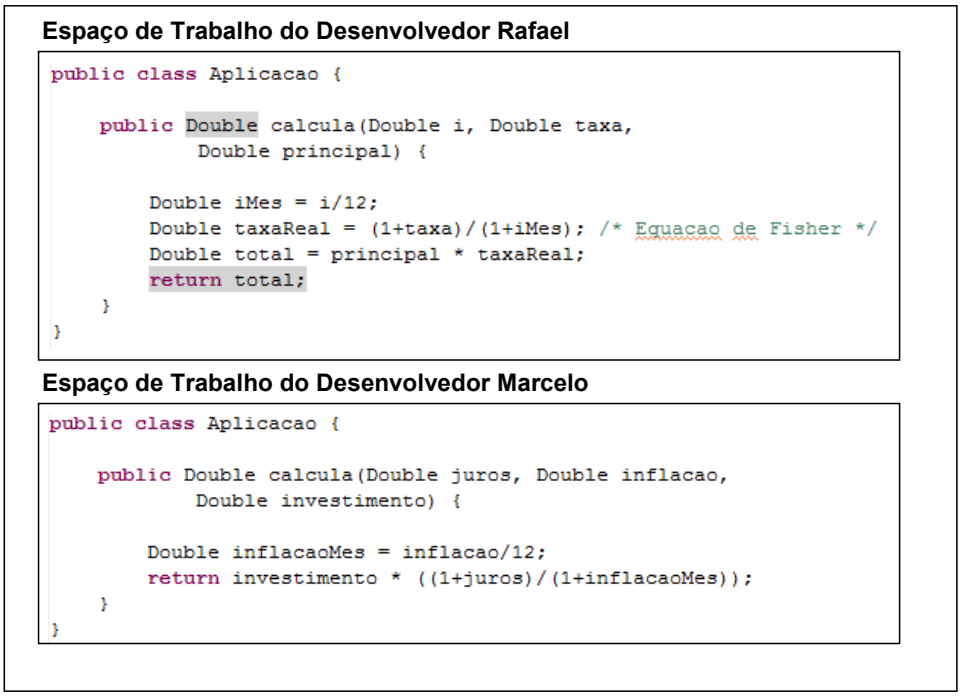



**Figura 3.8. Situação de retrabalho.**

Repare que neste caso, após um dos dois desenvolvedores ter criado o método "calcula", outro desenvolvedor também o criou em seu espaço de trabalho. Apesar de ambos os métodos possuírem conteúdos ligeiramente diferentes, sua assinatura é a mesma, portanto, indicando que potencialmente ambos os desenvolvedores estão trabalhando na mesma coisa.

É importante ressaltar que esta situação apenas ocorrerá na medida em que ambos os desenvolvedores mantenham apenas a sua versão do método em seu espaço de trabalho. A partir do momento em que um dos métodos vai para o repositório (através de um *check-in*) e o outro desenvolvedor obtém este método (através de um *check-out*), a classe passará a ter um erro de compilação, de modo que o desenvolvedor terá que escolher entre uma das

duas implementações, portanto desperdiçando tempo gasto no desenvolvimento.

- **Exclusão de arquivo Modificado –** Esta percepção indica quando um determinado arquivo modificado em um dado espaço de trabalho fora excluído de um outro espaço de trabalho. Para que esta percepção seja criada, a modificação não precisa ter terminado. Desta forma, um possível trabalho que seria posteriormente descartado por outro desenvolvedor (em função de exclusão do arquivo) pode ser interrompido previamente, economizando, assim, tempo de desenvolvimento.

- **Dependência Funcional por Método –** Esta percepção, associada exclusivamente a estrutura do software, indica, para uma dada classe, quais outras a mesma depende através de uma chamada de método. Um cliente desta abordagem, por exemplo, poderia utilizar esta informação para montar um diagrama de classe, onde uma associação entre duas classes pudesse refletir uma dependência funcional por chamada de método.

  Note que esta percepção também é necessária para a obtenção de outras percepções, como aquela que indica uma situação de conflito indireto. Isto porque a mesma, de certa forma, revela características estruturais do próprio software em desenvolvimento, informação esta básica para diversas outras inferências sobre o projeto.

- **Dependência Funcional por Atributo** – Esta percepção, também associada a estrutura do software, indica, para uma dada classe, quais outras a mesma depende através de um atributo. Um cliente desta abordagem, por exemplo, poderia utilizar esta informação para montar um diagrama de classe, onde uma associação entre duas classes pudesse refletir uma dependência funcional por atributo.

- **Atividades em Execução** – Esta percepção apresenta uma lista com todas as atividades sendo executadas por um dado desenvolvedor no momento. Isto

é, mostra quais atividades possuem uma data e hora de inicio, porém ainda não possuem uma data e hora de fim. Através desta percepção, outros desenvolvedores poderão saber o que seus pares estão fazendo no momento, fomentando a comunicação entre a equipe e antecipando possíveis problemas.

- **Atividades Finalizadas** – Esta percepção apresenta uma lista de todas as atividades executadas por um dado desenvolvedor. Isto é, através deste conjunto de informações, uma aplicação cliente poderá, por exemplo, calcular quem trabalhou mais em que ou até mesmo utilizar esta informação para inferir quem possui maior experiência em determinada parte do sistema, identificando, assim, possíveis especialistas do projeto.

Portanto, como pode ser observado, a abordagem trabalha inicialmente com o conceito de ações e atividades. Estas ações e atividades constituem os dados básicos que, quando correlacionados de diferentes maneiras, produzem um conhecimento (percepção) não revelado de forma direta ao longo do processo de desenvolvimento. Como fora demonstrado, este conhecimento poderá ser útil de diversas formas, dependendo da maneira que o mesmo é empregado. E, além disto, novas percepções podem ser elaboradas a partir destas e das atividades fornecidas, não restringindo, desta forma, a utilização desta abordagem apenas nos cenários descritos anteriormente.

## 3.5. Distribuição e Visualização

Todas as percepções e informações coletadas anteriormente são fornecidas a clientes interessados pela camada de distribuição da infra-estrutura. Esta camada está dividida basicamente em três componentes, como mostra a Figura 3.9. O primeiro componente, *WorkspaceServices*, fornece serviços relacionados a um espaço de trabalho qualquer. Isto é, suponha, por exemplo, que uma aplicação cliente precise obter todas as atividades sendo realizadas no momento por um dado desenvolvedor. Esta aplicação fará uso dos serviços fornecidos pelo *WorkspaceServices* para obter estas informações. Além disto, todas as informações referentes ao próprio desenvolvedor também poderão ser obtidas a partir deste componente de serviços.

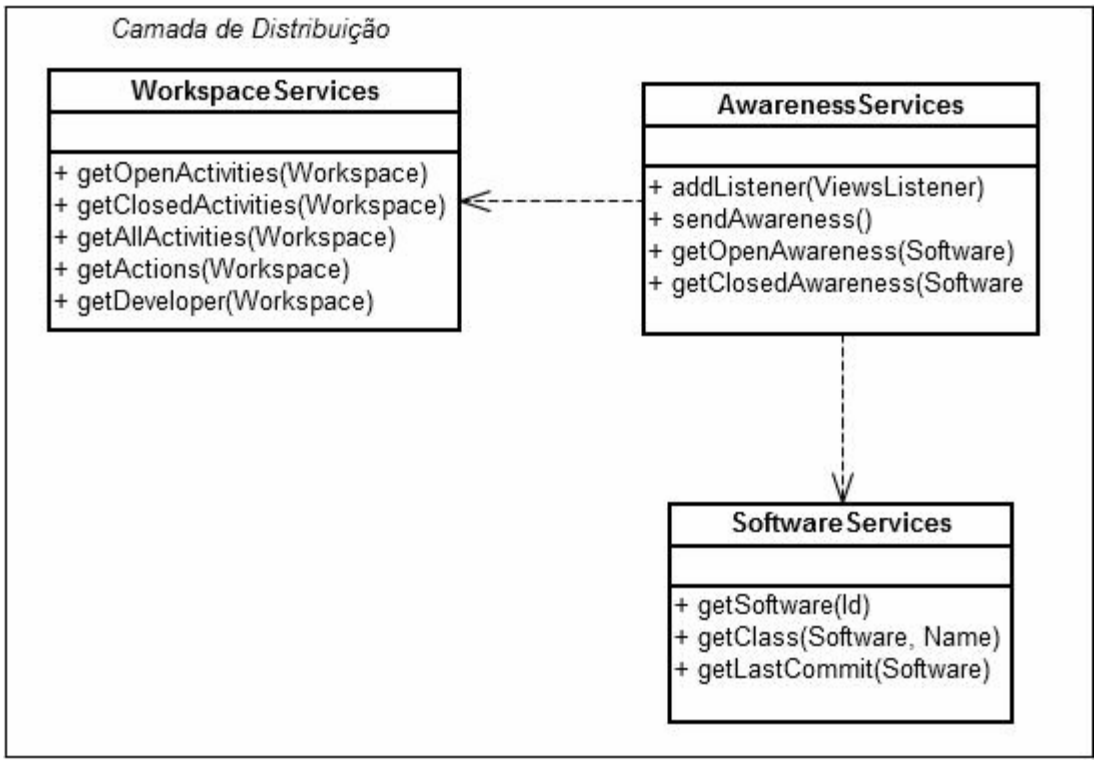



**Figura 3.9. Diagrama resumindo os principais elementos da camada de distribuição.**

O componente *SoftwareServices* fornece serviços relacionados ao próprio software em desenvolvimento. Através deste componente, uma aplicação cliente poderá obter toda árvore de elementos comportados por um software (vide Figura 3.5), incluindo suas classes e seus relacionamentos. De posse desta árvore (i.e., deste elemento chave do modelo de dados), a própria aplicação poderá obter as demais informações referentes ao software e, desta forma, poderá remontar a sua própria estrutura. De qualquer forma, caso a aplicação cliente queira partir de uma classe específica, isto é, esteja interessada apenas nas informações desta classe (como suas dependências e métodos), o componente *SoftwareServices* também fornecerá esta opção. Desta forma, a aplicação não precisará percorrer toda a estrutura do *Software*, a fim de obter esta única informação.

Já o componente *AwarenessServices* é responsável por informar a todas aquelas aplicações interessadas em receber percepções detectadas pela ferramenta. Note que os dois componentes anteriores agiam de forma passiva, isto é, apenas fornecem informações quando são solicitados. Já este, através do padrão de projeto *listener* (Gamma et al., 1994), aciona automaticamente todas aquelas aplicações previamente cadastradas quando uma

nova percepção é detectada para um determinado projeto de software. De posse destas atualizações, aplicações cliente poderão apresentar estas percepções de diferentes maneiras.

Por ser um componente ativo, o *AwarenessServices* precisa de algum tipo de mecanismo capaz de constantemente analisar as informações de entrada frente às informações já existentes e, a partir daí, produzir as percepções que serão disponibilizadas posteriormente. Este papel é desempenhado pela "fábrica de percepções" ou *AwarenessFactory* (nome utilizado internamente na infra-estrutura). Esta fábrica funciona como um observador externo ao desenvolvimento. A partir das informações obtidas pelos agentes de coleta instalados em cada espaço de trabalho e das informações existentes na própria infra-estrutura, esta fábrica é o elemento da infra que irá correlacionar tais informações e gerar as percepções de todo o processo de desenvolvimento.

## 3.6. Considerações Finais

Considerando os problemas destacados no Capítulo 2 sobre os mecanismos de percepção e visualização pesquisados, a abordagem VIEWS, apresentada neste capítulo, criou uma infra-estrutura capaz de recuperar dados das atividades de desenvolvimento sendo executadas de forma isolada em cada espaço de trabalho, bem como capturar informações do próprio produto deste desenvolvimento (o software em si), a partir de um sistema de controle de versão. Além disto, também foi apresentado como esta infra-estrutura é capaz de armazenar e distribuir tais informações de forma única e centralizada. Essa distribuição de informações é realizada a partir dos dados simplesmente coletados das fontes de informação (como os espaços de trabalho) e também realizada a partir de novas informações geradas pela própria infra-estrutura.

Estas novas informações são as chamadas percepções, conclusões derivadas pela ferramenta a partir da correlação entre dados dos diferentes desenvolvedores e dados da própria aplicação em construção. Estas percepções visam à redução do isolamento entre os desenvolvedores, fornecendo-lhes informações sobre o contexto do desenvolvimento em tempo real e de forma unificada.

Entretanto, é importante ressaltar que esta abordagem apenas será útil caso todos os desenvolvedores de um projeto estejam dispostos a compartilhar todas as suas atividades com os demais membros da equipe. A abordagem VIEWS não prevê qualquer tipo de

medida de proteção da privacidade das informações de seus usuários, como, por exemplo, faz a ferramenta Tukan (Schümmer e Haake, 2001). No Tukan, os usuários só podem acessar aquelas informações autorizadas para divulgação por cada desenvolvedor membro da equipe. Apesar desta abordagem estar mais condizente com questões de privacidade, a mesma acaba perdendo muitas informações que poderiam ser relevantes em diferentes cenários.

Como o intuito é o reúso e o compartilhamento destas informações, a abordagem fornece um mecanismo onde diferentes tipos de aplicações clientes poderão utilizar, até mesmo simultaneamente, estas informações para visualizar determinadas características do software e/ou do desenvolvimento do mesmo. Esta flexibilidade será demonstrada pelos protótipos de percepção e visualização implementados a partir desta infra-estrutura. Portanto, a fim de concretizar as ideias propostas neste capítulo, o Capítulo 4 apresentará como estas foram implementadas no protótipo de infra-estrutura VIEWS, bem como apresentará, através da criação de dois protótipos de aplicação cliente (*ViewsAwareness* e *ViewsSocioTechnical*), a forma com que diferentes tipos de aplicação podem fazer uso de suas funcionalidades.

# Capítulo 4 - Implementação

## 4.1. Introdução

Este capítulo apresenta a implementação da infra-estrutura VIEWS, bem como dos protótipos *ViewsAwareness* e *ViewsSocioTechnical*, segundo o direcionamento apontado pela abordagem descrita no Capítulo 3. Todos os protótipos implementados, além da infra-estrutura construída, assumem a utilização da plataforma Eclipse como ambiente de desenvolvimento. Apesar de todas as ideias apresentadas no capítulo não estarem restritas a qualquer tipo de ambiente, devido a sua implementação, seu uso por parte dos desenvolvedores estará exclusivamente restrito a IDE Eclipse (Eclipse, 2011), linguagem de programação Java (ORACLE, 2011) e sistema de controle de versão *Subversion* (Collins-Sussman et al., 2004).

A Seção 4.2 discute os detalhes da arquitetura da abordagem como um todo, incluindo tanto a infra-estrutura como as aplicações cliente criadas, além de apresentar as principais tecnologias e padrões utilizados. A Seção 4.3 apresenta a implementação da infra-estrutura VIEWS, ressaltando a forma como a mesma deve ser utilizada para a coleta e distribuição de informações sobre o desenvolvimento. A Seção 4.4 descreve uma das formas criadas para representar as informações obtidas a partir da infra-estrutura, discutindo o protótipo *ViewsAwareness* tanto no que tange a sua implementação como a sua utilização. A Seção 4.5 discorre sobre uma outra forma de utilizar algumas informações fornecidas pela infra-estrutura através do protótipo *ViewsSocioTechnical*. Por fim, as considerações finais deste capítulo são expostas na Seção 4.6.

## 4.2. Arquitetura

Todos os componentes da arquitetura desenvolvidos neste trabalho podem ser separados em componentes da plataforma Eclipse e componentes de servidor. Os primeiros são aqueles desenvolvidos a partir da API disponibilizada pelo Eclipse e seus *plug-ins*. A IDE Eclipse não corresponde a um programa monolítico, mas sim a um *kernel* reduzido, também conhecido como *plug-in loader*, cercado por centenas de *plug-ins* (Clayberg e

Rubel, 2006). A Figura 4.1 apresenta resumidamente esta arquitetura. Note que, a plataforma corresponde a um pequeno conjunto de componentes que cooperam entre si para formar a funcionalidade básica da mesma e que, adicionalmente, montam um arcabouço estrutural para a conexão de novas funcionalidades na forma de *plug-ins*.

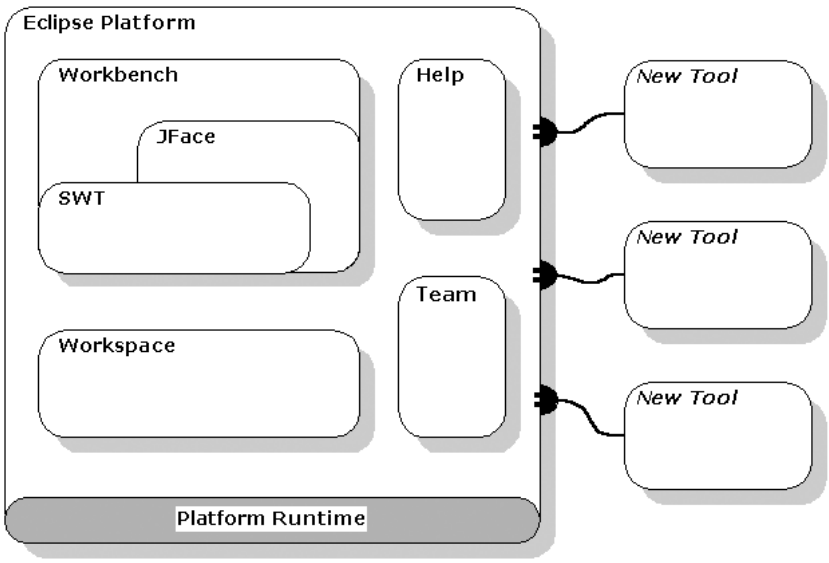



**Figura 4.1. Esquema que representa resumidamente a plataforma Eclipse.**

Portanto, através deste esquema de extensão, baseado em *plug-ins*, os componentes *Agente Coletor*, *ViewsAwareness* e *ViewsSocioTechnical* foram desenvolvidos. Cada um destes estendeu a plataforma em funcionalidades que variam em função do seu uso. Por exemplo, o *Agente Coletor* estendeu funcionalidades de interface com o usuário do próprio Eclipse, através de novas telas adicionadas às opções já existentes, além de funcionalidades fornecidas por outros *plug-ins*, como o *plug-in* que faz a interface com o servidor de gerência de configuração de software (GCS).

A Figura 4.2 apresenta detalhadamente a arquitetura implementada por este trabalho. Nela, é possível observar os principais componentes desenvolvidos e utilizados na construção das ferramentas. Resumidamente, pode-se notar que existem basicamente quatro diferentes tipos de nós físicos nesta arquitetura: *Computador do Desenvolvedor*, *Servidor de GCS*, *Servidor de Aplicação* e *Servidor de Banco de Dados*. Apesar de alguns destes nós estarem tipicamente em um mesmo ambiente físico, optou-se por separá-los nesta descrição para melhor elucidar os conceitos envolvidos. A seguir, uma breve descrição destes ambientes e suas configurações é realizada.

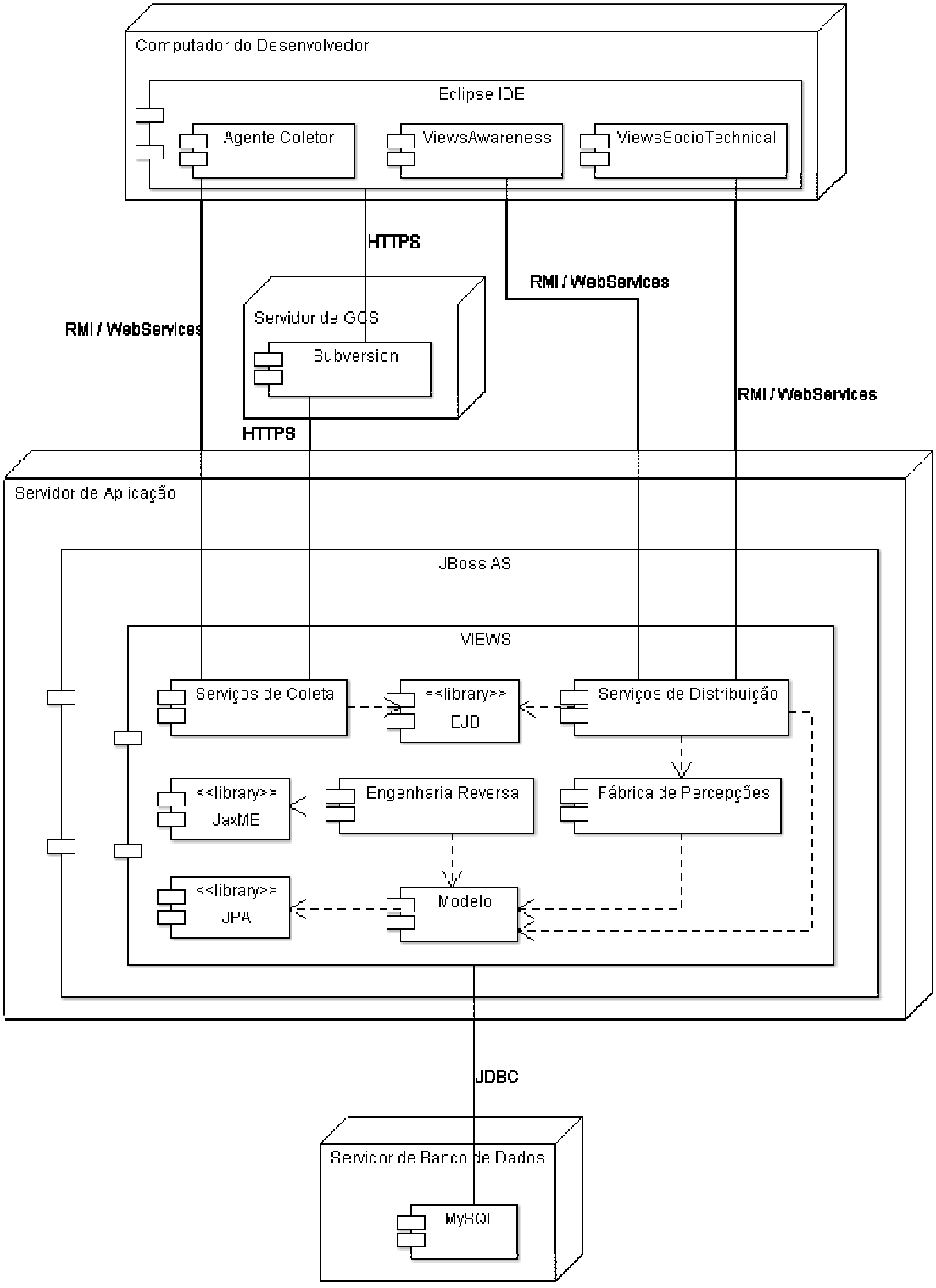



**Figura 4.2. Arquitetura do ferramenta VIEWS.**

O primeiro ambiente a ser descrito é o *Computador do Desenvolvedor*. Neste ambiente, é onde ocorrem todas as atividades de desenvolvimento por parte de uma equipe de software. Obviamente, a cada membro da equipe é atribuído um único *Computador do Desenvolvedor*. Este é o principal ambiente que deverá ser monitorado, a fim de obter as diversas informações referentes ao processo de desenvolvimento. No contexto deste trabalho, estas atividades de desenvolvimento ocorrem na plataforma Eclipse. Portanto, este deverá é o primeiro componente a ser instalado no *Computador do Desenvolvedor*. Posteriormente, já dentro do próprio Eclipse, alguns ajustes ainda deverão ser realizados para o pleno funcionamento do restante da abordagem. Estes ajustes referem-se a instalação do *plug-in Subversive* (Polarion, 2011). Este *plug-in* é responsável por toda interação com o sistema de controle de versão Subversion. Repare que esta é uma típica configuração do ambiente Eclipse para projetos de desenvolvimento de software nesta plataforma, desta forma, não representando apenas uma peculiaridade da abordagem VIEWS. Além disto, para que a infra-estrutura consiga receber dados sobre o desenvolvimento, pelo menos o componente *Agente Coletor* também deverá estar instalado na plataforma Eclipse. Maiores detalhes sobre este componente e sua interação com o Eclipse serão descritos posteriormente neste capítulo.

O segundo ambiente encontrado na arquitetura é o *Servidor de GCS*. É neste servidor que é instalado o sistema de controle de versão Subversion. Este sistema, amplamente utilizado pela comunidade de software, controla arquivos e diretórios, além de gerenciar todas as mudanças efetuadas na árvore de código de fonte ao longo do tempo. É neste sistema que está sempre armazenada a última configuração válida (usualmente assume-se como uma boa prática de desenvolvimento que todo código enviado ao sistema de controle de versão esteja ao menos compilando) do software em construção (ou até mesmo já construído). Também é por causa dele que uma equipe de vários desenvolvedores consegue trabalhar simultaneamente (muitas vezes sobre o mesmo artefato) no mesmo projeto, ganhando assim velocidade de trabalho. Porém, também é por causa do mesmo sistema que os diversos problemas de percepção e coordenação ocorrem, principalmente devido ao isolamento que o mesmo proporciona. É como se existissem vários mini-mundos, um para cada desenvolvedor em seu ambiente físico, separados por uma grande barreira, que representa o sistema de controle de versão. Apenas quando se tenta juntar os

trabalhos realizados separadamente é que diversos eventos ocorridos *a priori* podem ser identificados, como por exemplo o fato de dois desenvolvedores trabalharem no mesmo método em uma mesma classe. De qualquer forma, é através deste sistema que a infra-estrutura VIEWS consegue obter a versão mais atualizada do software já contendo as diversas contribuições individuais.

O terceiro ambiente contém o *core* de toda abordagem proposta e é onde foi aplicada boa parte do tempo de implementação deste trabalho. Este ambiente corresponde fisicamente ao *Servidor de Aplicação*. Como diversos serviços da pilha de protocolos JavaEE (JCP, 2009) foram utilizados, fez-se necessário a instalação neste ambiente do software servidor de aplicação *JBoss AS* (Marchioni, 2010). É a partir deste sistema que os componentes de servidor da infra-estrutura são implantados e acessados externamente. Como será detalhado na Seção 4.3, este acesso pode ocorrer através de duas formas. A primeira utiliza *Remote Method Invocation* – RMI – (Waldo, 1998) para trafegar informações entre o *Computador do Desenvolvedor* e o *Servidor de Aplicação*. Já a segunda forma utiliza *Web Services* (Curbera et al., 2002) para esta tarefa. Ambos os métodos são fornecidos nativamente pela tecnologia utilizada na implementação dos componentes da VIEWS e possibilitam ampla flexibilidade no que tange a integração com outras aplicações.

Por fim, temos o ambiente chamado de *Servidor de Banco de Dados*. É neste ambiente que deve ser instalado o sistema gerenciador de banco de dados (SGBD) utilizado pela abordagem, a saber *MySQL* (DuBois, 2008). Apesar do que possa parecer, este componente é vital para a abordagem. Isto é, sem ele, seria inviável a construção e prática de toda a abordagem. Isto porque a abordagem, como um todo, é intensiva em dados que precisam ser persistidos e mantidos ao longo do tempo. A forma de comunicação desta ferramenta com o restante da abordagem se dá pelo protocolo JDBC. Na verdade, toda essa comunicação ocorre de forma transparente para VIEWS em função da tecnologia adotada para estas questões de persistência.

As seções a seguir descrevem detalhadamente os principais componentes presentes em cada ambiente descrito na arquitetura. Note que, ao passo que o detalhamento ocorre, sempre que for pertinente um exemplo de utilização da abordagem é apresentado, de modo que seu entendimento seja facilitado e sua utilidade seja demonstrada.

## 4.3. Computador do Desenvolvedor

### 4.3.1. Agente Coletor

#### *4.3.1.1. Detectando Eventos*

O *Agente Coletor* para Eclipse é um *plug-in* desenvolvido por este trabalho que tem por objetivo monitorar todos os eventos gerados por um dado *Computador do Desenvolvedor*. Estes eventos podem ser agrupados em três grupos distintos: eventos Java, eventos de recursos e eventos de *Subversion*. Os eventos Java observados e registrados são aqueles referentes a criação, exclusão e alteração de classes, métodos e atributos Java. Para que estes eventos possam ser identificados, mudanças no modelo Java mantido pelo próprio Eclipse devem ser detectadas. Isto é obtido através da interface fornecida pela API do Eclipse *IElementChangedListener*. Desta forma, para que um componente seja capaz de receber notificações de mudança daquele modelo Java, basta para isso que o mesmo implemente esta interface fornecida.

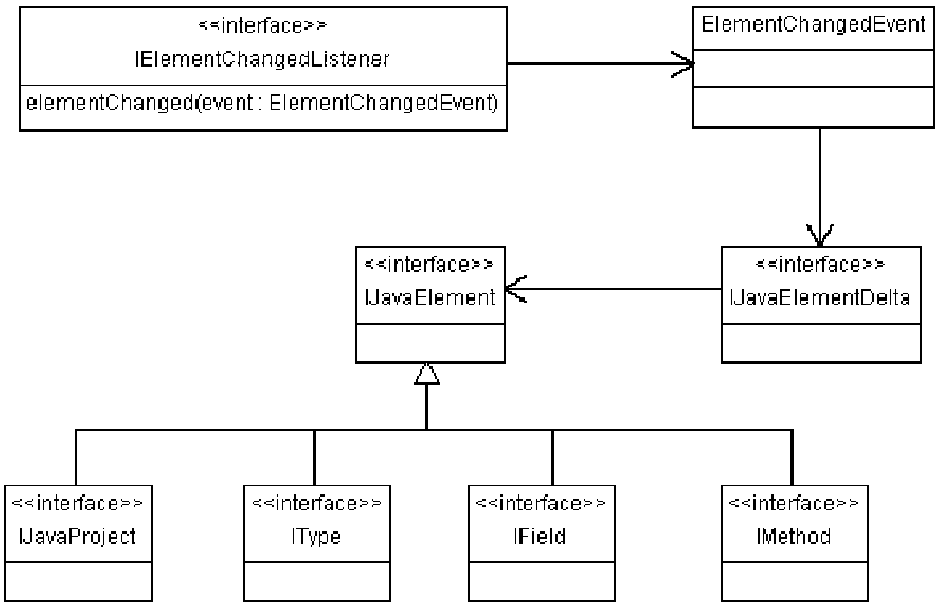



**Figura 4.3. Algumas das interfaces fornecidas pelo Eclipse para a monitoração do seu modelo Java interno.**

A Figura 4.3 apresenta a interface *IElementChangedListener*, além de outras interfaces utilizadas neste processo de monitoração. Como pode ser observado, uma classe cliente que queira receber notificações de alteração dos diferentes tipos de elementos do

modelo Java interno à plataforma Eclipse deve implementar o método *elementChanged*. Este método é invocado pelo Eclipse sempre que uma alteração no modelo é realizada. O próprio Eclipse se encarrega de passar, via parâmetro de entrada, um objeto contendo o tipo de evento ocorrido. Isto é, se for uma criação, exclusão ou simples alteração, por exemplo. Além disto, através deste objeto, também é possível chegar a que tipo de elemento sofreu o evento. Na figura estão representados alguns destes elementos. Por exemplo, caso uma nova classe seja criada na plataforma, o Eclipse irá disparar uma mensagem para a classe que implementa *IElementChangedListener*, contendo a informação de que este é um evento do tipo criação (i.e., *IJavaElementDelta.ADDED*) e que o evento está associado a um novo *IType* (i.e., uma nova classe).

A partir da identificação destes eventos, o Agente Coletor repassa estas informações para o componente responsável por tratar destes eventos. Para cada tipo de evento utilizado pela abordagem, existe um tratador específico. Este tratador é responsável por manipular os dados, se necessário, e posteriormente acionar os serviços de coleta disponibilizados pela infra-estrutura, informando a ocorrência de um novo evento. No exemplo citado anteriormente, que exemplifica a criação de uma nova classe, o tratador responsável por eventos de criação de classe no agente irá acionar o serviço de criação de um novo artefato Java na infra-estrutura. A seguir, uma breve ilustração sobre o passo a passo desta interação, sem todos os detalhes técnicos para sua realização, é apresentado.

**Passo 1: Conectando com o serviço de coleta da VIEWS**

```
try {
    InitialContext ic = new InitialContext();
    collectServices = (CollectServices)ic.lookup("Views/CollectServicesImpl/remote");
} catch (NamingException e) {
    ListenerLogger.printErr(e);
    throw new EclipseListenerException("Unable to connect with Views infra-struture");
}
```

**Passo 2: Informando a criação de um novo artefato Java à VIEWS**

```
collectServices.createArtifact(idworkspace, qname, ReadableType.JAVA, resource);
```

**Figura 4.4. Principais passos na criação de uma nova classe pelo *Agente Coletor*.**

No passo 1, ocorre inicialmente a conexão com os serviços da infra-estrutura VIEWS. Especificamente, a conexão com os serviços de coleta fornecidos pela mesma. Como pode ser observado na Figura 4.4, o componente responsável pelos serviços de coleta

pode ser encontrado, via JNDI, apenas utilizando o nome do serviço: "*Views/CollectServicesImpl/remote*". Observe que, para que a recuperação do serviço seja realizada com sucesso, o arquivo *jndi.properties* deverá estar devidamente configurado para o servidor de aplicação que contém a infra-estrutura VIEWS, estando este em qualquer servidor na Internet.

Já no passo 2, o *Agente Coletor* aciona o método de criação de artefato fornecido pela VIEWS, a partir da conexão obtida no passo anterior. Note que, para que a criação deste artefato possa ser realizada, o *Agente Coletor* deve informar quatro parâmetros de entrada distintos. O primeiro deles refere-se a identificação da *workspace* em que o usuário se encontra. Esta informação deve estar sempre guardada no *Agente Coletor*, visto que a mesma será quase sempre utilizada como chave durante a comunicação entre o *Agente Coletor* e a infra-estrutura. Além disto, também devem ser informados o nome qualificado desta classe, o tipo de artefato que está sendo construído (no caso uma classe Java) e a identificação do arquivo físico associado a este artefato na *workspace* do desenvolvedor.

#### *4.3.1.2. Cadastrando Desenvolvedor*

A primeira atividade a ser desempenhada por um desenvolvedor, ao começar um novo projeto de desenvolvimento, deve ser o seu cadastro na infra-estrutura VIEWS (caso o mesmo ainda não tenha sido cadastrado em qualquer projeto anterior). Para que isso seja realizado, o *Agente Coletor* fornece uma interface na própria IDE do Eclipse. A Figura 4.6 apresenta esta interface. Neste cadastro, apenas dados básicos sobre o desenvolvedor são solicitados, isto é, nome, e-mail e uma foto para identificação.

Esta interface é acessada através da opção *Window->Preferences* do próprio Eclipse, uma vez que a mesma foi implementada como uma extensão de *preferencePages* do próprio Eclipse, vide Figura 4.5.

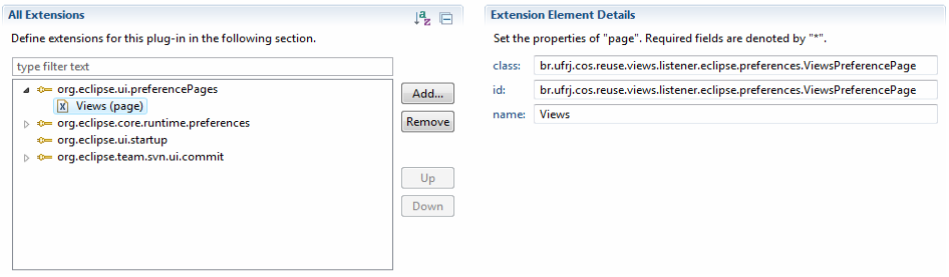



**Figura 4.5. Extensão criada pelo *Agente Coletor* a *preferencePage* do Eclipse.**

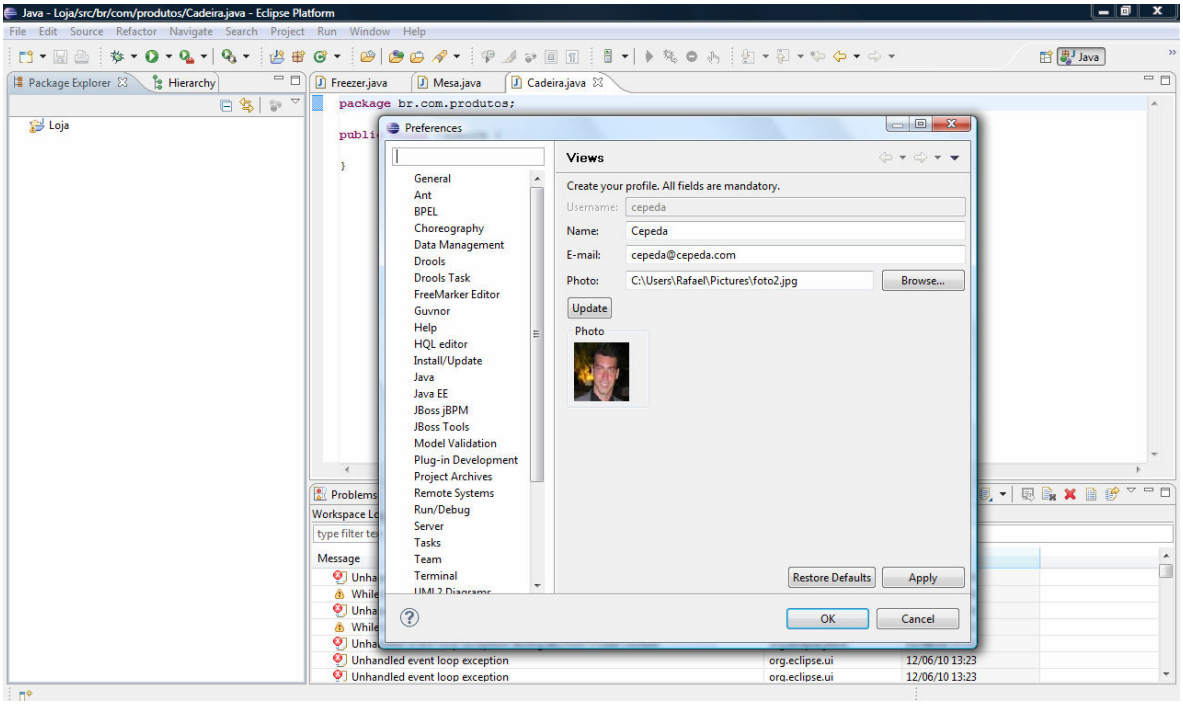



**Figura 4.6. Tela de cadastro de desenvolvedor através do *Agente Coletor* para Eclipse.**

Ao final deste processo, o *Agente Coletor* deve acionar a VIEWS e informar que um novo desenvolvedor será criado na infra-estrutura. Analogamente ao que foi apresentado anteriormente no caso da criação de uma nova classe, o agente deverá inicialmente realizar a conexão com a infra-estrutura através do serviço de coleta (vide Figura 4.6) e posteriormente acionar o devido método de criação, neste caso de um novo desenvolvedor.

### *4.3.1.3. Criando Projeto de Desenvolvimento*

Uma vez criado o cadastro do desenvolvedor na infra-estrutura, o mesmo poderá criar um projeto de desenvolvimento de software quando julgar necessário. Para isto, basta que o mesmo utilize as opções já existentes na IDE Eclipse. A Figura 4.7 mostra como isso pode ser feito no Eclipse. Além disto, quando o desenvolvedor insere o nome do novo projeto e confirma a sua criação, o *Agente Coletor* detecta esta criação e, antes mesmo que o processo de criação do projeto termine no Eclipse, o agente pergunta ao desenvolvedor se o mesmo deseja monitorar (i.e., se a VIEWS deve tomar conhecimento deste projeto) o projeto em questão. Em caso afirmativo, novamente o *Agente Coletor* deve entrar em contato com a infra-estrutura e solicitar a criação de um novo espaço de trabalho (*workspace*) para este desenvolvedor.

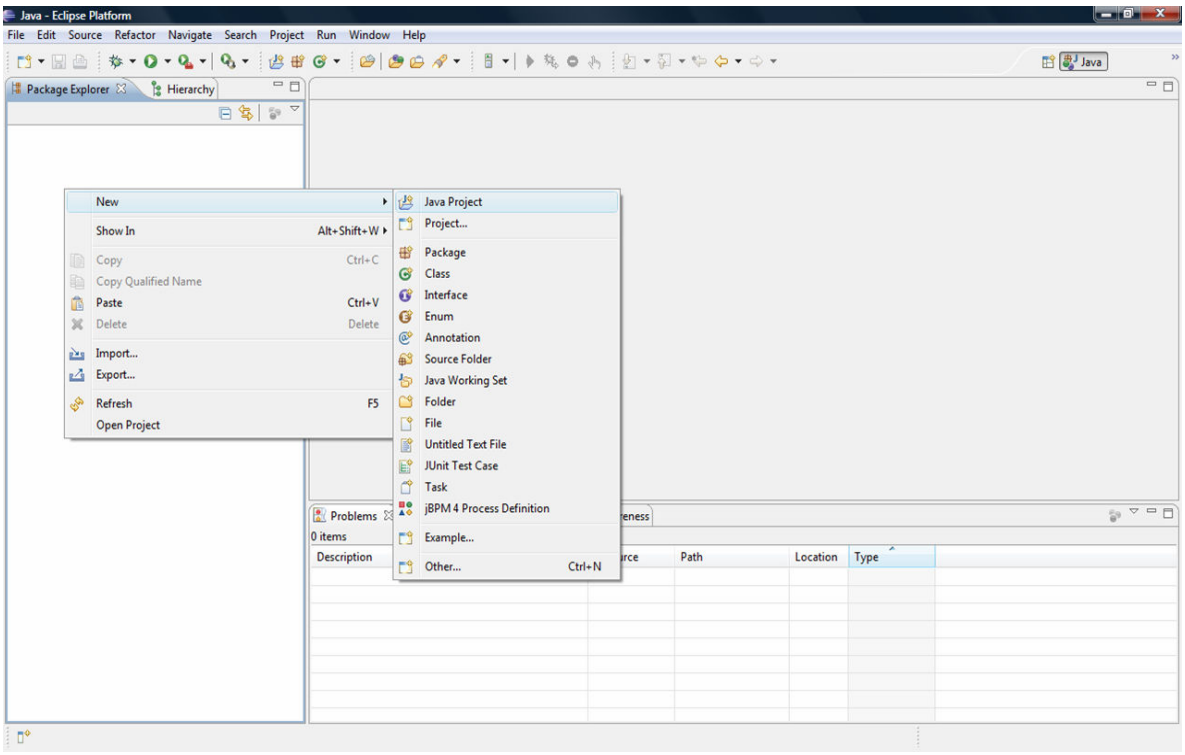



**Figura 4.7. Criando um novo projeto de desenvolvimento no Eclipse.**

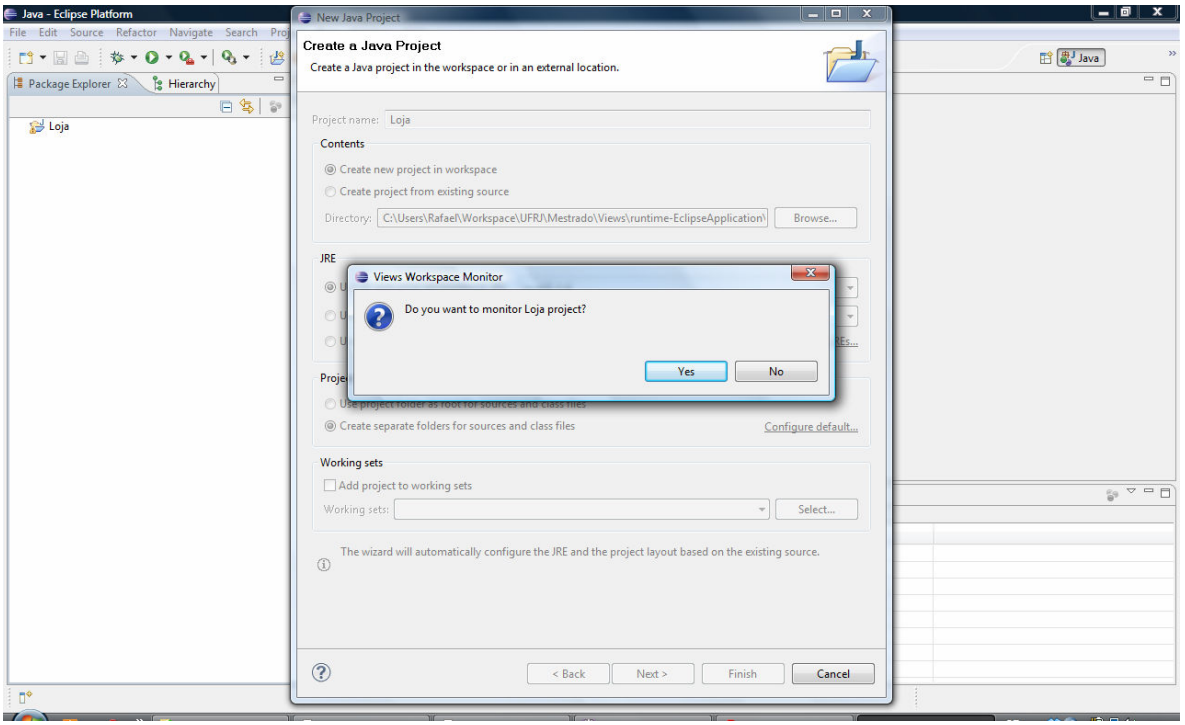



**Figura 4.8. O *Agente Coletor* pergunta se o projeto criado deverá ser monitorado.**

Repare que o mesmo procedimento é executado pelo *Agente Coletor*, no caso de um novo projeto ser criado a partir de um comando de *check-out* feito na IDE. Neste caso, este projeto é considerado um novo projeto apenas no que tange a *workspace* do desenvolvedor, não sendo considerado como um novo software pela VIEWS.

### 4.3.2. Protótipo *ViewsAwareness*

O protótipo *ViewsAwareness* (Figura 4.9) foi criado como prova de conceito para o serviços fornecidos pela infra-estrutura. A ideia era criar uma interface (visão) simples dentro da IDE Eclipse que pudesse ser utilizada para acompanhar em tempo real as percepções geradas pela infra-estrutura Views. Este protótipo também foi desenvolvido como um *plug-in* para o Eclipse. O intuito de o desenvolvimento ter sido realizado desta forma foi de proporcionar um único ambiente onde o desenvolvedor pudesse realizar suas atividades rotineiras (de implementação) e simultaneamente pudesse obter informações atualizadas sobre o andamento do projeto e sobre as atividades dos demais membros da equipe, além, obviamente, de reaproveitar conhecimento e código implementado anteriormente durante o trabalho.

A interface criada representa uma visão no Eclipse como a visão de erros detectados na IDE ou a de problemas do projeto. Ela é composta por uma tabela que é automaticamente atualizada sempre que uma nova informação for enviada pela infra-estrutura. Para isto, foi implementado na infra-estrutura um esquema baseado no padrão de projeto *observer*. Neste esquema, qualquer cliente interessado em receber as percepções criadas pela Views deve se cadastrar como um observador na própria infra-estrutura. Assim, sempre que uma nova percepção é gerada ou uma antiga é atualizada, a Views percorre sua lista de observadores enviando uma mensagem para cada um sobre o evento ocorrido.

Com base nestas informações enviadas, o protótipo *ViewsAwareness* apresenta suas percepções, mostrando o tipo de percepção, a data e hora de sua ocorrência, qual desenvolvedor foi o causador e sobre qual artefato a percepção diz respeito. Porém, o protótipo apenas irá mostrar aquelas percepções que envolvem o desenvolvedor em questão. Isto é, no Eclipse de um dado desenvolvedor, só irá aparecer uma mensagem de conflito quando este estiver em conflito com algum outro. O protótipo não irá apresentar

situações de conflito entre dois desenvolvedores quaisquer. Isto vale para todas as outras percepções que envolvam duas partes.

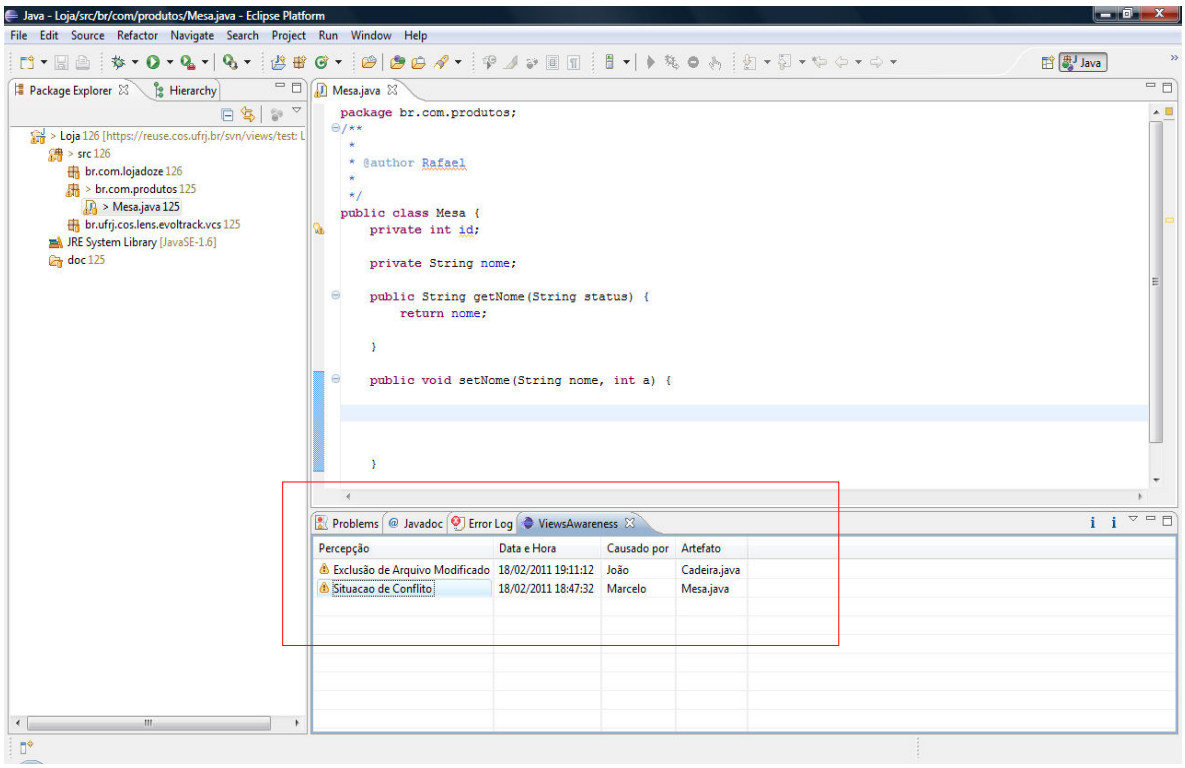



**Figura 4.9. O protótipo ViewsAwareness.**

Repare que esta não é uma abordagem nova ou exclusiva apresentada por este trabalho, conforme discutido no Capítulo 2. Porém, é a única abordagem a receber estas informações sobre percepções e contexto de uma infra-estrutura independente, desenvolvida justamente para este propósito. Portanto, o que nos interessa não é o simples fato da ferramenta apresentar informações úteis durante o desenvolvimento de software, como conflitos com outros desenvolvedores ou criações de artefatos do projeto, e sim mostrar que, através de uma abordagem unificada e centralizada em uma infra-estrutura capaz de prover dados e inferir informações sobre o projeto, é possível construir ferramentas de apoio ao desenvolvimento, como ferramentas de CSCW e visualização de software, sem ter que reinventar boa parte do processo necessário para atingir este objetivo.

### 4.3.3. Protótipo *ViewsSocioTechnical*

O segundo protótipo criado como prova de conceito para os serviços fornecidos pela infra-estrutura explora aspectos totalmente diferentes do protótipo anterior. A intenção é mostrar como a partir da mesma infra-estrutura (Views) é possível obter um tipo de resultado completamente diferente, ainda que na mesma linha das abordagens discutidas no Capítulo 2 que discorrem sobre visualização de software.

O protótipo foi idealizado com o objetivo de visualizar na forma de um diagrama (similar ao diagrama de classe da UML) dados sobre as classes sendo desenvolvidas e sobre as pessoas que as estão desenvolvendo. Note que o intuito não é apresentar quem em algum momento do projeto já **implementou** algum código em uma determinada classe e sim apresentar quem no exato momento da exibição do diagrama **está implementando** em uma determinada classe. Isto é, o diagrama reflete apenas o presente, não sendo possível (*a priori*) visualizar o passado.

A ferramenta nada mais é do que um editor gráfico contendo elementos desenvolvidos para a visualização de classes, pessoas e seus inter-relacionamentos. Como havia uma grande necessidade de customização de diagramas padrões, como o diagrama de classe UML, optou-se por desenvolver um editor totalmente novo para a visualização pretendida. Assim, a ferramenta foi desenvolvida utilizando-se um *framework* conhecido como GEF (ECLIPSE FOUNDATION, 2011). Este *framework* provê um conjunto de tecnologias para que editores gráficos ou visões possam ser construídas na plataforma Eclipse com maior facilidade. O *framework* é dividido em duas partes (que representam um *plug-in* cada): Draw2D e GEF MVC. O primeiro é um conjunto de classes utilitárias desenvolvidas para o *layout* e renderização gráfica. O segundo constitui propriamente o esquema GEF para a criação dos editores e visões. Um esquema resumido de sua arquitetura pode ser observado na Figura 4.10.

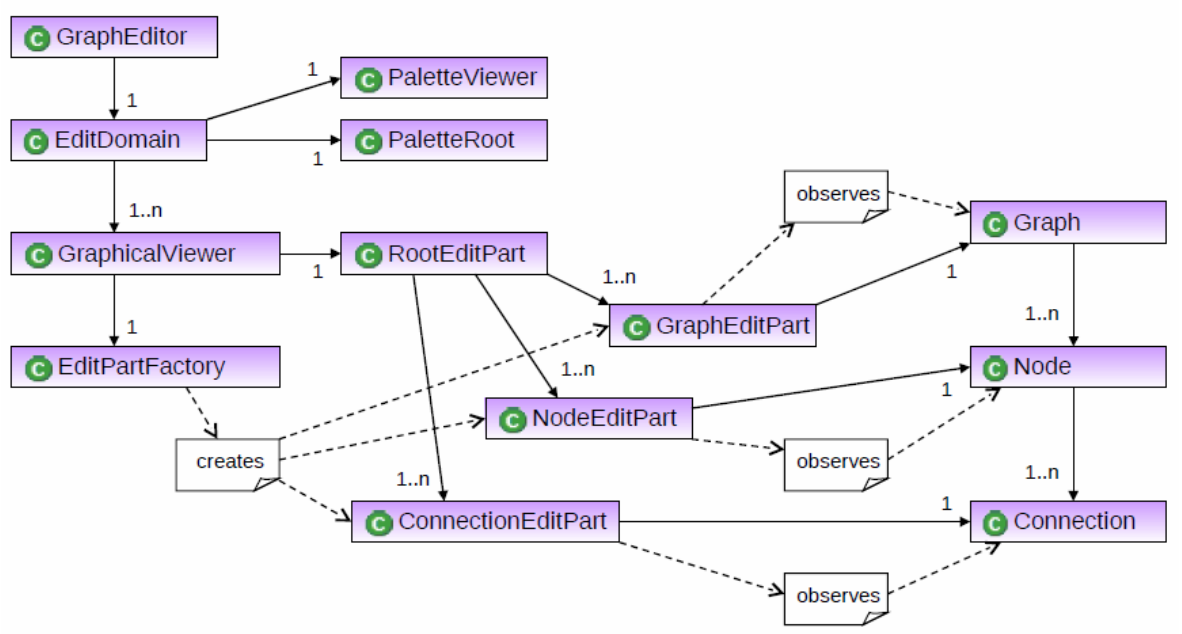



**Figura 4.10. Estrutura de um típico editor GEF (Aers, 2008).**

Resumidamente, um editor para Eclipse baseado no *framework* GEF deve implementar uma estrutura similar a descrita pela figura. Isto é, deve inicialmente conter uma classe que representa um editor (*GraphEditor*) para o Eclipse. Esta classe, além de ser implementada seguindo as regras estipuladas para um editor, também deve ser utilizada na configuração da própria ferramenta através do arquivo *plugin.xml* (vide Figura 4.11). Para que qualquer conteúdo possa ser visualizado no editor, é preciso associar um elemento do tipo *GraphicalViewer* ao mesmo. Este é o elemento que de fato controla o editor. Isto é, é nele que o conteúdo é inserido e as ações do usuário são capturadas. Pode-se pensar neste elemento como aquele responsável por gerenciar todo o ciclo de vida do editor.

O restante da arquitetura pode ser explicado através do modelo *Model-View-Controller* (MVC), esquema utilizado em todo *framework*. O elemento *GraphicalViewer* é responsável pela criação, através de uma fábrica (*EditPartFactory*), de todos os controladores da arquitetura, que neste caso são todos os elementos do tipo *EditPart* (por exemplo, *NodeEditPart*). Estes elementos são a ponte de ligação entre os elementos do modelo e os elementos de visão. Portanto, qualquer que seja a alteração a ser realizada no modelo, esta será realizada através do seu respectivo controlador. No exemplo ilustrado pela Figura 4.10, os elementos monitorados pelo modelo são *Graph, Node* e *Connection*, que por sua vez representam os conceitos de um grafo, de um nó e de uma aresta,

respectivamente. Além disto, para cada elemento do modelo, deverá existir ainda um elemento de visão (que não está representado na figura). Usualmente, quando se trata de GEF, estes elementos são uma extensão da classe *Figure*, provida no *plug-in* Draw2D. Estes constituem a representação gráfica de cada elemento do modelo passível de visualização.

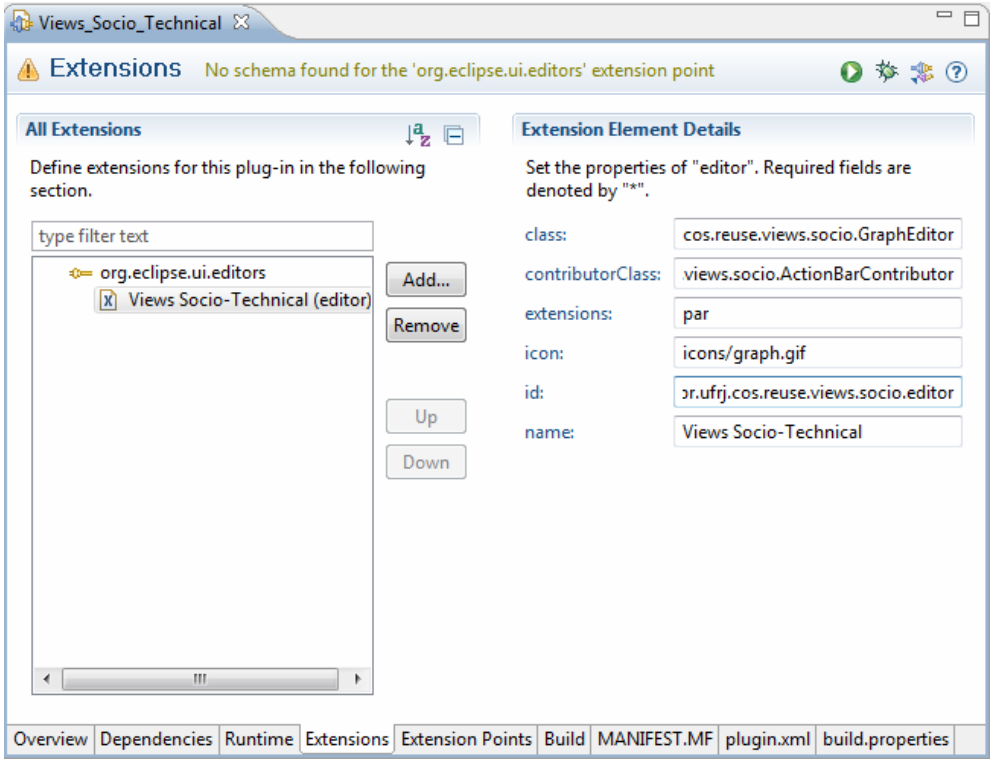



**Figura 4.11. Configuração do arquivo *plugin.xml* no protótipo ViewsSocioTechnical.**

A representação visual de um nó no protótipo *ViewsSocioTechnical* foi projetada para comportar basicamente dois tipos de informação. O nome da classe (quando aplicável) e as informações sobre pessoas (desenvolvedores). A Figura 4.12 ilustra esta representação. Nela pode ser observado que o nó foi divido em duas partes ou dois compartimentos. Na primeira parte, foi colocado um *label* que será populado com o nome da classe a qual está sendo representada pelo nó. Já no segundo compartimento, é exibida uma lista de todos os desenvolvedores que estejam trabalhando nesta classe no momento. Cada linha desta lista é composta por uma imagem, obtida da foto cadastrada pelo desenvolvedor em sua IDE, e um *label* descrevendo o seu nome. Além disto, também foram criadas representações visuais para algumas das conexões básicas encontradas em um diagrama padrão UML,

como uma associação (uma linha reta) e uma generalização (uma linha reta com um triângulo vazado em uma das pontas).

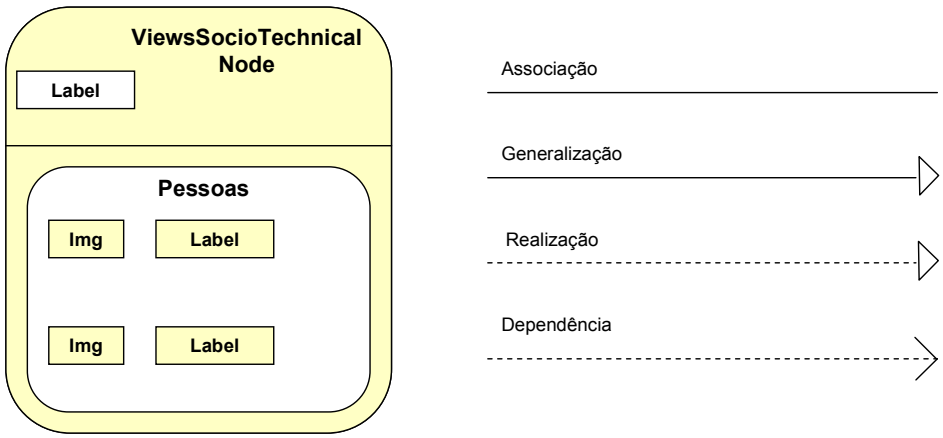



**Figura 4.12. Representações visuais utilizadas no protótipo ViewsSocioTechnical.**

Adicionalmente, vale ressaltar que a ferramenta provê funcionalidades através do próprio editor, para que o diagrama seja modificado com o intuito de, por exemplo, apresentar apenas o nome das classes e seus inter-relacionamentos (diagrama técnico). Neste caso, a ferramenta simplesmente oculta o compartimento das pessoas de sua visualização, resultando em um diagrama de classes UML simplificado. Em contrapartida, caso o usuário tenha interesse em visualizar apenas as pessoas, o que ocorre é que o compartimento superior é omitido e todos os relacionamentos entre os nós são alterados para o tipo dependência. A ideia, neste caso, é mostrar que quando duas pessoas estiverem trabalhando em duas classes distintas, unidas por algum tipo de relacionamento de classes, existe uma boa possibilidade de suas atividades estarem relacionadas, já que seus objetos de trabalho estão.

Trazer a tona esta percepção é justamente o objetivo desta visualização conjunta. Imagine, por exemplo, cada classe como o centro de uma esfera de influência. A medida que nos distanciamos do centro da esfera, ao longo do grafo que representa qualquer sistema, estamos reduzindo a probabilidade de impacto direto relacionado a uma alteração naquela classe. Duas classes em sub-sistemas totalmente distintos (ainda que interligados) usualmente têm um acoplamento mais fraco do que duas classes dentro do mesmo sub-sistema. Desta forma, duas pessoas trabalhando dentro de um mesmo sub-sistema (próximas uma da outra de suas esferas de influência) têm potencialmente uma dependência

entre si e a necessidade de colaboração maior do que duas pessoas trabalhando em sub-sistemas separados. É isto que a visualização discutida apresenta.

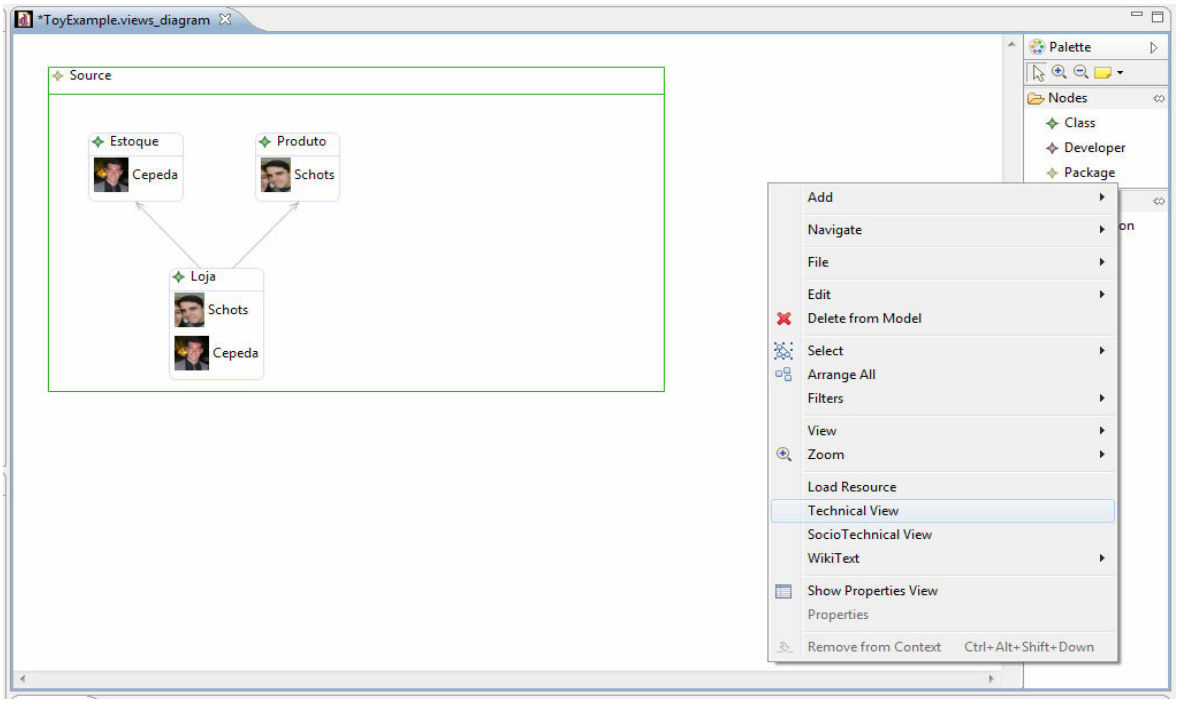



**Figura 4.13 - Protótipo *ViewsSocioTechnical* apresentando informações sócio-técnicas.**

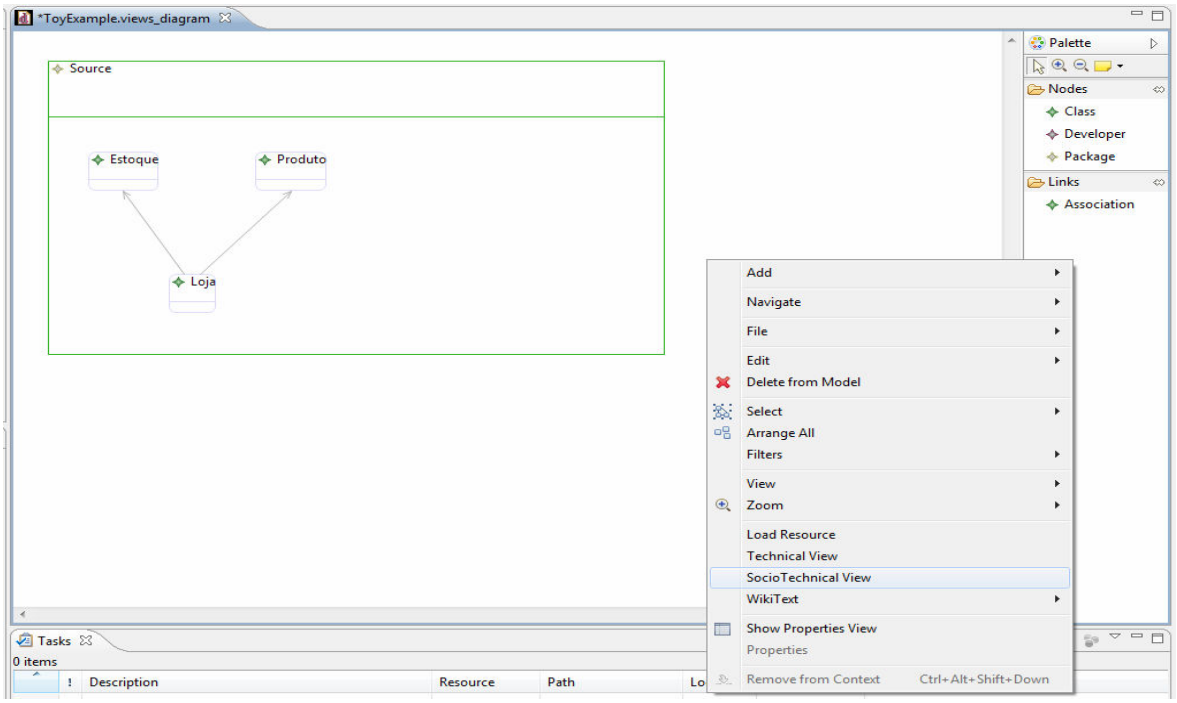



**Figura 4.14 - Protótipo *ViewsSocioTechnical* apresentando apenas informações técnicas.**

A Figura 4.13 apresenta uma visão do protótipo ViewsSocioTechnical, onde um subconjunto de informações obtidas da infra-estrutura VIEWS é utilizado para fornecer uma perspectiva sócio-técnica do software em desenvolvimento. Além disto, através do mesmo protótipo, é possível visualizar uma simples visão técnica da aplicação (Figura 4.14), revelando assim sua estrutura emergente (Westhuizen et al., 2006). Por estrutura emergente, também conhecida como *design emergente*, entende-se o estado atual do software representado na forma de sua estrutura. É como se fosse uma "foto" instantânea do projeto, que evolui ao longo tempo.

## 4.4. Servidor de Aplicação

### 4.4.1. Infra-estrutura VIEWS

A infra-estrutura VIEWS possui um papel central na abordagem proposta. Conforme apresentado na Figura 4.2, esta estrutura chave da abordagem é composta por cinco componentes principais desenvolvidos. Além de outros componentes desenvolvidos externamente. Estes cinco componentes atuam em etapas específicas do processo de recuperação e disponibilização de informação sobre o desenvolvimento de software.

Assim, o componente *"Serviços de Coleta*" é responsável por disponibilizar métodos para a coleta de informações pertinentes a um determinado desenvolvimento de software. Porém, como boa parte destas informações são obtidas a partir de um código fonte de aplicação, a infra-estrutura também possui um componente de *"Engenharia Reversa*" responsável por refazer a estrutura do software sendo desenvolvido. Repare que nada impede que a infra-estrutura possua vários componentes deste tipo, por exemplo, um para engenharia reversa de código Java, outro para código .NET, outro para C++, etc. Este trabalho desenvolveu apenas um para código Java.

Uma atividade adicional desta etapa de análise (i.e., pós-coleta) é a criação das percepções. Estas são oriundas das diferentes informações coletadas, bem como das informações existentes previamente. Para a geração destas percepções, foi desenvolvido na infra-estrutura o componente "*Fábrica de Percepções*". Sempre que uma nova informação é repassada à infra-estrutura, através dos "*Serviços de Coleta*", a "*Fábrica de Percepções*" é acionada. Os dados utilizados para a geração das percepções são obtidos a partir do componente chamado de "*Modelo*". É neste componente que todas as informações ficam

armazenadas e representadas. Este componente, na verdade, representa uma abstração lógica criada para o modelo banco de dados desenvolvido. Isto é, este componente nada mais é do que o conjunto de classes desenvolvidas para o mapeamento objeto-relacional criado para representar as entidades de dados utilizadas neste trabalho.

Para que estes dados pudessem ser disponibilizados para aplicações cliente, o componente "*Serviços de Distribuição*" foi desenvolvido. Este componente disponibiliza diversos métodos, acessados tanto via RMI como via *WebServices*, para a obtenção de diferentes tipos de informação referentes a um projeto de desenvolvimento de software.

Na prática, a infra-estrutura foi dividida em três projetos de implementação. O primeiro deles, chamado de ***views-model***, representa a implementação do componente "Modelo" da arquitetura proposta. Esta implementação faz uso do *framework* de persistência *Hibernate* (vide Figura 4.15) para mapear as classes do domínio da aplicação em elementos do modelo de dados relacional. Alguns dos benefícios da utilização deste *framework* na abordagem são o isolamento quanto ao SGBD utilizado e o acesso a linguagens de consultas orientadas a objetos, como o HQL.

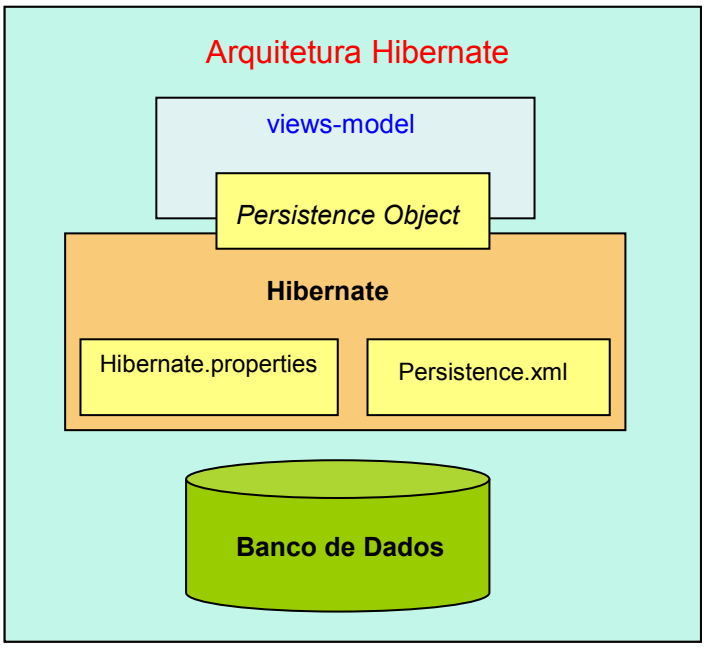



**Figura 4.15. Arquitetura do framework Hibernate.**

O segundo projeto, chamado de ***views-services-api***, possui todas as interfaces de serviços utilizados pelos clientes da infra-estrutura. Desta forma, é através desta API que tanto os "Serviços de Coleta" como os "Serviços de Distribuição" são acessados. Para que

este acesso possa ser realizado, o arquivo ***views-services-api.jar*** disponibilizado deve ser colocado no *classpath* de cada aplicação cliente ou coletora. A partir deste arquivo, todas as classes públicas disponíveis para acesso externo estarão acessíveis.

O terceiro e último projeto, chamado de ***views-services***, representa o *core* de toda a infra-estrutura. Além de possuir a implementação para todas aquelas classes expostas pela API externa (***views-services-api***), possui também a implementação para os demais componentes encontrados na arquitetura, os componentes de "Engenharia Reversa" e "Fábrica de Percepções". O primeiro, conforme já apresentado anteriormente, é responsável pela tradução do código fonte Java em elementos do modelo Views. Para isto, este componente faz uso da biblioteca JaxMe do grupo Apache. Note que, como o processo de engenharia reversa ocorre no ambiente da infra-estrutura de forma independente, nenhum cliente é afetado em relação a este processamento. O que pode ocorrer, no máximo, é uma pequena defasagem momentânea quanto ao que possa estar sendo visualizado e o que de fato já ocorreu em um determinado projeto.

## 4.5. Considerações Finais

Neste capítulo, foi descrita a implementação do projeto Views, com base na abordagem descrita no Capítulo 3. Inicialmente, apresentamos a arquitetura da solução proposta envolvendo tanto a infra-estrutura criada, como os demais componentes do trabalho, como o *Agente Coletor* para Eclipse e as aplicações cliente desenvolvidas: *ViewsAwareness* e *ViewsSocioTechnical*. Além disto, também foi apresentado como os principais componentes da infra-estrutura foram implementados e ainda como estes interagem entre si, com o intuito de prover as funcionalidades discutidas no capítulo anterior.

Através da implementação dos protótipos *ViewsAwareness* e *ViewsSocioTechnical,* ficou clara a flexibilidade da infra-estrutura em prover informações pertinentes ao desenvolvimento de software. Baseada em interfaces de coleta e distribuição e protocolos de comunicação amplamente conhecidos (RMI e *WebServices*), a infra-estrutura mostrou-se como um bom aliado na tarefa de desenvolvimento de ferramentas para o apoio a engenharia de software, principalmente aquelas fundamentadas em visualização de informação, conforme discutido no Capítulo 2.

À exceção da infra-estrutura, todas as demais ferramentas criadas (*Agente Coletor*, *ViewsAwareness* e *ViewsSocioTechnical*) foram desenvolvidas para a plataforma Eclipse, conforme descrito ao longo do capítulo. Optou-se por esta plataforma principalmente porque a mesma representa uma das principais IDEs de desenvolvimento de software utilizadas pela comunidade Java. Além disto, esta escolha também foi realizada com base no conhecimento prévio do autor nesta plataforma, adquirido ao longo do processo de pesquisa para este trabalho (CEPEDA et al., 2010) e de outros trabalhos relacionados.

As ferramentas implementadas, no entanto, possuem algumas limitações. Tanto o Agente Coletor como a própria infra-estrutura só conseguem, atualmente, trabalhar com código desenvolvido em linguagem Java. Além disto, a infra-estrutura também só está apta a recuperar código-fonte do repositório Subversion, não sendo possível, neste momento, a obtenção desta informação de qualquer outro tipo de fonte de dados. Além disto, todo projeto (incluindo as aplicações cliente) presumem que estão trabalhando com um sistema orientado a objetos.

# Capítulo 5 - Avaliação

## 5.1. Introdução

Como fora definido no Capítulo 1, a questão de pesquisa desse trabalho é:

**"A combinação de informações da estrutura do software com informações do seu processo humano de engenharia torna mais eficaz o entendimento global do processo de desenvolvimento de software por parte dos desenvolvedores em ambientes distribuídos?"**

Para responder esta questão de pesquisa, diversos aspectos precisam ser avaliados, dentre eles: aspectos de desempenho; aspectos de usabilidade; aspectos de sua real eficácia, verificando se de fato esta abordagem proporciona *insights* sobre o processo de desenvolvimento distribuído que não seriam indicados por abordagens tradicionais, isto é, abordagens onde apenas uma das perspectivas fosse apresentada (estrutura do software ou percepção de grupo); e aspectos da representação dessas informações de forma visual, identificando seus pontos fortes e possíveis pontos de melhoria. Este capítulo apresenta um estudo avaliando apenas os dois últimos aspectos citados (i.e., eficácia e aspectos da representação), considerando que os mesmos representam algumas das principais características a serem avaliadas neste tipo de trabalho. Outros estudos analisando os demais aspectos deverão ser objetos de trabalhos futuros.

A partir disto, os protótipos descritos no Capítulo 4 foram utilizados na aplicação de um estudo preliminar de observação para avaliar a abordagem *VIEWS*. Foram observados 4 alunos de pós-graduação da *COPPE-UFRJ*, utilizando duas estratégias de entendimento de um processo de desenvolvimento de software distribuído. Em uma das estratégias foi empregada uma abordagem onde apenas as informações da estrutura do software eram fornecidas. Enquanto que na outra, fazendo uso da abordagem *VIEWS*, tanto estas informações da estrutura do software como uma noção de grupo foram apresentados em conjunto. Apesar de restrito, o resultado deste estudo pôde ser utilizado para entender as limitações atuais da abordagem proposta, além de determinar os próximos passos do desenvolvimento do trabalho.

Nas próximas seções deste capítulo, o estudo realizado é detalhado, descrevendo seus objetivos, discutindo seu planejamento e apresentando os resultados obtidos. A Seção 5.2 apresenta os objetivos gerais do estudo; a Seção 5.3 define com mais detalhes o estudo realizado; a Seção 5.4 explica o procedimento de execução do estudo; a Seção 5.5 apresenta os resultados e as observações obtidas sobre a execução do estudo; a Seção 5.6 sumariza a avaliação feita pelos participantes sobre o estudo; a Seção 5.7 discute a validade do estudo; e por fim, a Seção 5.8 apresenta as considerações finais da avaliação realizada.

## 5.2. Objetivo

Visando analisar se o conjunto de informações apresentadas pela abordagem *VIEWS* representa um diferencial na percepção de informações antes não identificadas diretamente por abordagens tradicionais, no contexto do desenvolvimento distribuído de software, foi realizado um estudo de observação. Em estudos desta natureza, o participante realiza uma tarefa enquanto é observado por um experimentador. Este tipo de estudo tem a finalidade de coletar dados sobre como determinada tarefa é realizada (Shull et al., 2001). Por meio destas informações, pode-se obter uma compreensão de como um novo processo é utilizado.

O objetivo deste estudo de observação pode ser apresentado de acordo com o modelo descrito por Wohlin et al. (1999), ilustrado na Tabela 5.1.

**Tabela 5.1 – Definição do objetivo do estudo de observação**

| |
|---|
| **Analisar** o conjunto de informações apresentadas pela abordagem VIEWS |
| **Com o propósito de** evidenciar a capacidade da abordagem VIEWS em responder questões sobre o processo de desenvolvimento de software distribuído em curso |
| **Em relação à** outras abordagens tradicionais utilizadas no entendimento deste processo |
| **Do ponto de vista do** desenvolvedor de software |
| **No contexto de** análise das informações de um processo de desenvolvimento distribuído, a partir da utilização de uma abordagem convencional e da utilização da abordagem proposta |

## 5.3. Definição do Estudo

Este estudo foi realizado em um ambiente acadêmico, no laboratório Lab3D da COPPE/UFRJ. Para participar no estudo, os indivíduos deveriam ter alguma experiência anterior com a linguagem de programação Java, com o ambiente de desenvolvimento Eclipse e com o sistema de controle de versão Subversion, uma vez que este conhecimento é necessário para a correta execução das tarefas sugeridas. Além disto, era necessário que o participante tivesse alguma experiência com o *plugin* que faz a integração entre o ambiente Eclipse e o Subversion. Neste contexto, 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.

Através desse estudo, esperava-se observar como desenvolvedores de software em uma equipe distribuída coordenariam suas atividades e trocariam informações quando a abordagem VIEWS estivesse presente em sua dinâmica de trabalho. Mais especificamente, tentou-se verificar, qualitativamente, se com a introdução das informações providas pela abordagem VIEWS os participantes teriam mais facilidade em responder questionamentos sobre o projeto, quando comparado a um cenário sem essas informações. Desta forma, com o objetivo de realizar esta avaliação, os seguintes pontos principais foram observados:

- É possível identificar as modificações realizadas por todos os desenvolvedores do projeto, mesmo que ainda não enviadas para o repositório de código-fonte?
- As percepções fornecidas são utilizadas de fato na identificação das atividades do próprio desenvolvedor e dos demais?
- É possível identificar conflitos a partir das percepções fornecidas?
- As percepções fornecidas estimulam e servem como referência para a comunicação dentro da equipe?
- As percepções fornecidas estimulam e servem como referência para a reutilização de componentes de software criados?

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 contendo outro desenvolvedor. A equipe trabalhava em um projeto de pequeno porte desenvolvido em Java, para uma empresa de

desenvolvimento de software hoteleiro. Também foi informado que, devido a distância, o outro desenvolvedor da equipe só poderia ser contatado através de mensagens instantâneas de texto.

Como o estudo pretendia comparar tal cenário de desenvolvimento com e sem a utilização da abordagem VIEWS, os quatro participantes foram separados em duas equipes de dois participantes cada. Assim, inicialmente, o estudo foi realizado com a primeira equipe e contou com a presença de modelos fornecidos pela abordagem VIEWS. Já na segunda execução, a outra equipe contou com diagramas de classe convencionais, similar àqueles fornecidos pela abordagem EvolTrack. Em ambos os casos, foi distribuído um conjunto igual de tarefas de programação que envolviam mudanças no código-fonte. Este conjunto era dividido em "Tarefas do Participante A" e "Tarefas do Participante B" (Anexo E).

## 5.4. Procedimento de Execução

Cada sessão do estudo utilizou dois participantes e durou cerca de 2 horas. Foram executadas duas sessões, uma fazendo uso da abordagem VIEWS e outra não. Em cada sessão, inicialmente, cada participante foi informado sobre o experimento através do Formulário de Consentimento (Anexo A). Caso concordasse em participar do experimento, o participante preenchia o Questionário de Caracterização (Anexo B). 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 preenchimento dos formulários iniciais levou cerca de 15 minutos.

Em seguida, o participante recebeu um treinamento de aproximadamente 15 minutos sobre os principais conceitos envolvidos. Além disto, na etapa com a utilização da abordagem VIEWS, um treinamento com as principais características do modelo fornecido pela abordagem também foi realizado. O modelo utilizado nesse treinamento pode ser observado no Anexo G. Ao final do treinamento, foi entregue o documento de Descrição Geral da Tarefa (Anexo D), 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 foram entregues logo depois e podem ser

acessadas no Anexo E. 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 a ser modificado já estava na área de trabalho dos participantes e também já estava compartilhado e sob controle de versão, pelo Subversion. Além disto, todo ambiente de comunicação via mensagem instantânea já estava estabelecido.

Na etapa com a abordagem VIEWS, foram propostas 13 tarefas, divididas em dois conjuntos, um contendo 6 tarefas e outro contendo 7. Um desses conjuntos foi escolhido ao acaso e disponibilizado para o *Participante A*. O conjunto remanescente foi entregue ao *Participante B*. Ao final de cada tarefa, era apresentado a ambos os participantes um modelo, com as informações da VIEWS, representando o estado atual do sistema.

Na etapa sem a abordagem VIEWS, processo idêntico foi adotado para a distribuição dos dois grupos de tarefas aos participantes desta etapa. Note que os participantes dessa etapa não são os mesmos da anterior. Neste caso, inicialmente, um modelo conceitual completo do sistema questão foi entregue a todos participantes. Posteriormente, seguindo a lógica tradicional de obtenção de informações a partir do sistema de controle de versão, um novo modelo emergente era apresentado a cada *check-in* realizado no repositório.

Em ambas as etapas, no final da execução de todas as tarefas por parte dos participantes, dois questionários eram respondidos. O primeiro questionário (Anexo F), avaliava o nível de percepção acerca do projeto e da equipe que os participantes conseguiram obter com a utilização de uma das abordagens. O segundo questionário (Anexo C) referia-se a própria execução do estudo, quanto a alguma dificuldade encontrada, se o treinamento foi adequado, etc. Os questionários foram respondidos, em média, em 20 minutos.

## 5.5. Resultados e observações

O estudo foi realizado com 4 participantes, todos alunos de mestrado do programa de pós-graduação em Engenharia de Sistemas e Computação da COPPE/UFRJ, selecionados por conveniência. Como pode ser observado na Tabela 5.2, todos os participantes possuem um perfil semelhante, a não ser pelo participante “P1”. Esse é o único que já possui uma certa experiência na indústria desenvolvendo sistemas em Java.

Também representa o participante que já trabalhou com a maior equipe, 10 pessoas, e também é quem possui a maior experiência nas tecnologias utilizadas nesse estudo, como o ambiente de desenvolvimento Eclipse e sistemas de controle de versão.

Os demais participantes demonstram características semelhantes. Todos os três já participaram de projetos em pequenas equipes, variando de 1 até 4 pessoas. Eles também possuem experiências semelhantes no que tange as tecnologias utilizadas. Em geral, apresentam uma experiência mediana em relação ao Eclipse (faixas de 2 até 3), modelagem de sistemas de informação e UML. Apresentam uma experiência superior em sistemas de controle de versão e, por fim, praticamente não tem qualquer conhecimento técnico da área de visualização de software.

**Tabela 5.2 – Resultado do Questionário de Caracterização, por participante.**

| | P1 | P2 | P3 | P4 |
|---|---|---|---|---|
| Formação Acadêmica | Mestrando | Mestrando | Mestrando | Mestrando |
| Experiência com Java | Indústria | Curso | Curso | Projeto Pessoal |
| Tempo de Experiência com Java | 4 anos | 1 ano e meio | 2 anos | 4 anos |
| Tamanho Máximo da Equipe | 10 pessoas | 4 pessoas | 3 pessoas | 1 pessoa |
| Experiência com Eclipse (1-4) | 4 | 2 | 3 | 3 |
| Experiência com Modelagem de Sistemas de Informação (1-4) | 4 | 2 | 2 | 3 |
| Experiência com UML (1-4) | 4 | 2 | 2 | 2 |
| Experiência com Sistemas de Controle de Versão (1-4) | 4 | 4 | 4 | 3 |
| Conhecimento de Visualização de Software (1-4) | 2 | 1 | 0 | 0 |

Com base na análise dessas informações sobre a caracterização de cada participante, as equipes foram dividas de modo que houvesse um equilíbrio quanto a experiência de seus integrantes. Desta forma, os participantes “P1” e “P2” foram alocados em uma equipe, doravante denominada “Equipe A”, e os participantes “P3” e “P4” em uma segunda equipe,

doravante denominada “Equipe B”. Separadas as equipes, foi escolhida ao acaso qual equipe executaria o estudo com a VIEWS e qual executaria sem a VIEWS. Neste caso, ocorreu que a “Equipe A” foi selecionada no primeiro caso e a “Equipe B” selecionada no segundo.

Após a execução das tarefas de programação solicitadas, cada participante teve que responder a um questionário de percepção acerca do projeto utilizado no estudo. Os questionamentos realizados podem ser consultados no Anexo F. A seguir, uma breve descrição de cada pergunta realizada e o resultado obtido em ambas as etapas é descrito.

A questão 1 pergunta quais foram as classes alteradas ou criadas pelo participante. A intenção desta pergunta era verificar que tipos de recursos seriam utilizados para a compreensão das próprias atividades do participante. Na etapa com a abordagem VIEWS, ambos os participantes fizeram uso dos modelos fornecidos para a resolução do problema. Apesar de não ser a única forma de obtenção dessas informações, o que pôde ser observado é que a facilidade de capturar quem fez o quê através desta abordagem proporcionou a sua utilização direta neste questionamento.

Já na abordagem sem a VIEWS, também ambos os participantes conseguiram responder corretamente a pergunta. Porém, neste caso, em nenhum dos casos os modelos fornecidos foram consultados. Ambos, para responder este questionamento, consultaram a folha de tarefas para fazê-lo. O fato do modelo fornecido não incorporar informações referentes as pessoas dificultou a sua utilização neste caso, pelo que foi observado.

A questão 2 indagou sobre as classes alteradas, criadas ou excluídas pelo outro membro da equipe. O intuito dessa pergunta era avaliar até que ponto o participante conseguiria obter uma percepção do trabalho do outro participante em um ambiente distribuído. Novamente, e pelos mesmos motivos, os participantes da “Equipe A”, com a VIEWS, conseguiram responder corretamente a este questionamento.

A “Equipe B”, por sua vez, teve grandes dificuldades em responder a esta pergunta. Por exemplo, o participante “P3” conseguiu identificar visualmente apenas uma alteração não realizada por ele através do *design emergente* fornecido. As demais alterações não foram identificadas por ele. Já o participante “P4”, apesar de não ter listado as classes alteradas, indicou que possivelmente através de “uma inferência totalmente pessoal a partir dos atributos *Last Change At* e *Last Change By*” fornecidos pelo sistema de versionamento,

talvez conseguisse obter esta resposta. Entretanto, vale notar que mesmo que o fizesse, algumas classes ainda poderiam ficar de fora de sua análise, uma vez que, em um projeto de software, não há garantias de que todos enviaram suas alterações por completo para o repositório.

As questões 3, 4 e 5 tentaram avaliar a eficácia da abordagem em uso em auxiliar a identificação e antecipação de conflitos de código, antes que os mesmos de fato ocorressem em tempo de *check-in*. Neste caso, a ideia também era avaliar até que ponto a abordagem em questão forneceria subsídios para uma possível comunicação entre os membros da equipe. Neste contexto, a "Equipe A" teve sucesso na identificação e resolução antecipada de todos os conflitos provocados pelas tarefas, antes que os mesmos ocorressem. Através das indicações de modificações realizadas pela abordagem VIEWS, os participantes não tiveram qualquer dificuldade em detectar futuros conflitos que viriam a ocorrer em decorrência de suas atividades conflitantes. Além disso, com as informações sobre quem desenvolve o quê, a rápida percepção de quem procurar, neste caso, foi notória.

Assim, através das informações fornecidas pela VIEWS, foi possível que um participante se coordenasse com o outro de forma que o trabalho não fosse refeito, em alguns casos. Isto pode ser evidenciado através de um trecho da conversa entre os participantes "P1" e "P2", apresentado na Figura 5.1. Neste caso, o participante "P1" verificou através do modelo que algumas alterações solicitadas a ele já haviam sido realizadas na classe *Reserva* pelo participante "P2". Como pode ser observado na figura, antes que o participante "P1" realize as suas alterações, ele solicita a classe *Reserva* para o particpante "P2". Desta forma, evitando um conflito de código que certamente ocorreria e, possivelmente, traria atrasos ao projeto, em função do esforço de *merge* necessário.

**P1:** "P2?"
**P2:** "diga"
**P1:** "você comitou a classe Reserva?"
**P2:** "não. vc irá precisar?"
**P1:** "pode fazer isso pra mim,por favor exatamente"
**P2:** "commit feito"

**Figura 5.1 – Trecho de uma conversa entre o participante "P1" e o participante "P2".**

Em contrapartida, como era de se imaginar, os integrantes da "Equipe B" não conseguiram antecipar qualquer conflito entre as tarefas realizadas. Apenas em tempo de

*check-in* que cada um dos participantes dessa equipe tomou conhecimento do trabalho conflitante e, por vezes, redundante realizado pelo outro colega de equipe. A dificuldade na resolução deste problema, neste caso, se deve ao isolamento mantido pela abordagem baseada em *design* emergente que utiliza como fonte de informação o sistema de controle de versão. Além disso, é muito importante ressaltar que através desta abordagem tradicional, durante esse estudo, não houve qualquer tipo de comunicação entre os membros da equipe. Isto se deve, principalmente, pela pobre percepção de grupo obtida através dessa abordagem. Desta forma, aparentemente não houve qualquer estímulo para que se estabelecesse uma comunicação, ao contrário do cenário com a abordagem VIEWS.

Por fim, a questão 6 perguntou se em algum momento o participante conseguiu reutilizar alguma classe previamente criada pelo outro. O intuito, neste caso, era avaliar se os recursos disponibilizados pela abordagem em questão eram úteis no que tange a identificação de oportunidades de reutilização. O cenário para a obtenção dessa informação foi construído da seguinte forma. Uma tarefa de criação de uma classe específica chamada *DataHora* foi designada para os participantes "P1" e "P3". Esta classe ainda contaria com métodos de conversão do tipo de dados *date* para *string* e vice e versa. Então, foi solicitado, de forma genérica, aos participantes "P2" e "P4" a criação de uma classe que deveria ser responsável pelo tratamento de datas, com possíveis métodos de conversão.

Como resultado, na etapa realizada com a VIEWS, o participante "P2" conseguiu identificar previamente, através dos modelos fornecidos, que uma classe chamada *DataHora* já havia sido criada pelo participante "P1". Neste caso, confirmando a intenção classe, foi possível reutilizá-la na execução de sua tarefa. Provavelmente, isso só foi possível porque, **antes** de começar a tarefa, o participante consultou o modelo e verificou a criação desta nova classe. A fácil identificação de classes criadas, através do mecanismo de cores, também facilitou o processo. Já no caso da "Equipe B", o mesmo não pôde ser realizado, principalmente por causa de todos os fatores já citados anteriormente, como o isolamento e dificuldade de compreensão do modelo.

A Tabela 5.3 sumariza o resultado apresentado. O símbolo (  ) indica que o questionamento foi atendido utilizando a abordagem fornecida. O símbolo (  ), em contrapartida, sinaliza que, apesar do questionamento ter sido atendido corretamente, a abordagem fornecida não fora utilizada em sua resolução. Já o símbolo (  ) chama a

atenção para o fato de que o questionamento não fora respondido corretamente. Questões que não possuíam uma resposta do tipo sim ou não tiveram suas respostas resumidamente apresentadas na tabela.

**Tabela 5.3 – Resultado dos participantes na resolução do questionário de percepção.**

| | Equipe A – Com VIEWS | | Equipe B – Sem VIEWS | |
|---|---|---|---|---|
| | P1 | P2 | P3 | P4 |
| Questão 1 |  |  |  |  |
| Questão 2 |  |  |  |  |
| Questão 3 |  |  |  |  |
| Questão 4 | • Percepção de modificações através das cores;<br>• Percepção dos desenvolvedores | • Percepção de modificações através das cores;<br>• Percepção do conflito;<br>• Percepção dos desenvolvedores | N/A | N/A |
| Questão 5 | N/A | N/A | Não houve resposta | • Informação distribuída apenas após processo de *check-in* |
| Questão 6 |  |  |  |  |

## 5.6. Avaliação realizada pelos participantes

Depois de concluir as duas etapas do estudo, os participantes preencheram um questionário de avaliação com perguntas referentes a execução do estudo e questões genéricas em relação as duas abordagens adotadas. Com relação a abordagem VIEWS os participantes se mostraram bastante satisfeitos com os resultados obtidos e com a facilidade

em responder perguntas que, tomando como base suas experiências de desenvolvimento, não seriam facilmente respondidas. Alguns comentários desses participantes foram:

- Em relação as tarefas distribuídas, "faltaram breves descrições sobre o que os métodos conteriam e quando não precisariam ter seu conteúdo implementado. Isso não prejudica o estudo em nada, mas a execução poderia ter sido mais rápida com essa informação";
- "Aumentou a comunicação entre a equipe e o trabalho fluiu de forma que não tivemos nenhum conflito acontecido. Previmos todos!";
- "A abordagem foi adequada para responder a todos os questionamentos feitos";
- Como sugestão de melhoria, a abordagem poderia contar com "uma visão geral do modelo poderia ser bem interessante";
- "As cores identificando as modificações nas classes e a identificação dos desenvolvedores trabalhando em uma determinada classe ajudaram na antecipação dos conflitos".

Já os participantes que executaram o estudo sem a abordagem VIEWS, relataram os mesmos problemas encontrados por desenvolvedores em equipes distribuídas sem a devida infra-estrutura de apoio. Alguns comentários desses participantes foram:

- "As questões sobre a percepção do trabalho são difíceis de serem respondidas";
- "Não fiquei satisfeito com o resultado final nesta etapa, pois não foi possível resolver os conflitos em tempo de *commit*";
- "A coordenação do trabalho distribuído fica a cargo pura e simplesmente da experiência dos membros da equipe";
- "Não soube identificar se houve realmente uma prévia atualização do projeto por parte de outro membro da equipe";
- "Algumas questões, como a autoria, perguntavam sobre as atualizações de outros participantes do projeto. Com essa abordagem, não acredito que tenha respondido bem a esses questionamentos".

## 5.7. Validade

Este estudo preliminar foi executado a fim de observar o uso controlado da abordagem desenvolvida e compará-la com uma abordagem tradicional encontrada no

contexto de desenvolvimento de software. Os resultados obtidos a partir das observações e da avaliação dos participantes nos ajudaram a entender as limitações da abordagem 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.

O fato do participante "P1" ter sido selecionado, mesmo que ao acaso, na etapa de execução com a abordagem VIEWS pode ter influenciado o resultado do estudo, uma vez que o mesmo, notoriamente, possuía maior experiência frente aos demais.

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 etapa do estudo. A simplicidade desses artefatos de software também pode ter influenciado as observações obtidas.

Para realizar algumas justificativas na análise dos dados, as informações de caracterização que cada participante forneceu de si mesmo foram utilizadas. Entretanto, não é possível confirmar que tais informações fornecidas estejam corretas.

A disposição dos elementos (layout) dos diagramas pode exercer influência sobre os resultados (Sun e Wong, 2005) (Knodel et al., 2006). No contexto deste estudo, os diagramas foram reorganizados de forma a não ultrapassarem as margens de impressão e não haver sobreposição de elementos.

O entendimento dos participantes sobre as questões dos formulários é diretamente influenciado pela forma como as questões foram elaboradas; se a questão tiver sido mal formulada, o estudo pode ser afetado negativamente (Wohlin et al., 1999). A análise dos

instrumentos utilizados no estudo (inclusive os formulários) por especialistas visou reduzir esta interferência. No entanto, a partir das análises das respostas, pode-se observar que algumas perguntas foram interpretadas de maneiras diferentes pelos participantes, enquanto outras foram consideradas ambíguas, principalmente no questionário de avaliação.

## 5.8. Considerações Finais

Neste capítulo, foi descrito o estudo desenvolvido para avaliar preliminarmente uma parte da abordagem VIEWS. Neste estudo, cada participante executou algumas tarefas pré-determinadas de programação utilizando modelos idênticos àqueles fornecidos pela nossa abordagem, enquanto participava de uma equipe distribuída, simulada pelo experimentador. Para conseguir simular este cenário, os participantes foram dispostos isoladamente em um ambiente onde apenas poderiam se comunicar entre si e com o experimentador virtualmente, através da utilização de mensagens instantâneas. As mudanças introduzidas pelos participantes criavam oportunidades de coordenação e necessidade de entendimento do projeto como um todo, que podiam ou não ser aproveitadas a partir da abordagem proposta.

A partir desse estudo, foi possível ter indícios de que a abordagem VIEWS é uma alternativa real para auxiliar desenvolvedores no processo de desenvolvimento de sistemas em equipes distribuídas. Além disto, com as respostas obtidas e através da própria observação dos participantes ao longo do estudo, foi possível identificar diversos pontos de melhoria e algumas limitações da abordagem.

Em relação aos quatro pontos apresentados no inicio deste capítulo, obtivemos relativo sucesso na observação do uso das informações fornecidas pela abordagem VIEWS, tanto em conteúdo como em sua forma visual de apresentação:

1. É possível identificar as modificações realizadas por todos os desenvolvedores do projeto, mesmo que ainda não enviadas para o repositório de código-fonte? Apesar de não contar com um número realístico de desenvolvedores e de modificações no projeto, obtivemos uma avaliação positiva quanto aos mecanismos fornecidos pela abordagem.

2. As percepções fornecidas são utilizadas de fato na identificação das atividades do próprio desenvolvedor e dos demais? Ambos os participantes que utilizaram a abordagem, ressaltaram a facilidade de obtenção deste tipo de informação dos modelos. Desta forma, as percepções fornecidas realmente puderam ser utilizadas na identificação das atividades. O experimentador pôde observar que, de fato, sempre antes da execução de uma determinada tarefa o modelo era consultado.
3. É possível identificar conflitos a partir das percepções fornecidas? O estudo mostrou que, quando os modelos são consultados previamente, a identificação de conflitos é feita de forma direta. Todos os conflitos esperados pelas tarefas criadas no experimento foram antecipadamente detectados. Acredita-se que nem mesmo a possível complexidade de um modelo traria muita dificuldade neste caso, já que o desenvolvedor irá observar apenas aquela classe na qual sua tarefa está atrelada, não precisando analisar *a priori* as demais classes e suas relações.
4. As percepções fornecidas estimulam e servem como referência para a comunicação dentro da equipe? Isto pôde ser notado pela inspeção da conversa entre os participantes da etapa realizada com a VIEWS. Neste caso, ficou claro que as percepções obtidas dos modelos fomentaram significativamente a comunicação entre a equipe, o que é evidenciado pela ausência de comunicação entre a outra equipe, quando a abordagem foi retirada.
5. As percepções fornecidas estimulam e servem como referência para a reutilização de componentes de software criados? Apesar do estudo não ser conclusivo quanto a este ponto, foi possível observar que quando os modelos fornecidos pela VIEWS são incorporados de fato na dinâmica do desenvolvimento, e dependendo da perspicácia do desenvolvedor, oportunidades de reutilização podem ser aproveitadas. Em todas as oportunidades de reutilização que se mostraram presentes ao longo do estudo, ambos os participantes conseguiram fazer uso das informações fornecidas pela VIEWS para tirar proveito da situação. Conseqüentemente, a equipe que trabalhou com a VIEWS não sofreu qualquer tipo de re-trabalho.

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.

# Capítulo 6 - Conclusões

## 6.1. Epílogo

O desenvolvimento de software, por sua própria natureza, é uma atividade colaborativa. Praticamente não há projeto de software onde apenas um único desenvolvedor é o encarregado de todas as atividades de sua construção. Conseqüentemente, desenvolvedores são organizados em equipes para a engenharia desse tipo de empreendimento. Estas equipes, todavia, podem estar dispostas de diferentes maneiras, dependendo da complexidade do projeto, tamanho da empresa e necessidades da área em questão. Em geral, essas disposições são classificadas em duas categorias: em equipes locais ou equipes distribuídas. Conforme apresentado ao longo do texto, equipes distribuídas sofrem de diversos problemas em função da distância, como a falta de conhecimento amplo e comum sobre o projeto e a dificuldade de comunicação entre os membros da equipe.

Em relação a comunicação, não apenas há uma precariedade dos meios, em geral pobres em contexto. Estudos (Herbsleb e Mockus, 2003) (Mockus e Herbsleb, 2001) mostram que, apesar de existirem meios de comunicação operacionais neste tipo de cenário, a comunicação ocorre de forma muito deficitária entre membros separados geograficamente. Não há, por exemplo, conversas informais, onde informações são trocadas constantemente entre os membros de uma equipe, como acontece em equipes locais. Desta forma, os problemas de falta de percepção e precariedade das comunicações emergem como principais fatores de sucesso no desenvolvimento distribuído de software.

Como foi observado, inúmeras abordagens foram criadas com o objetivo de enriquecer o ambiente de desenvolvedores de software, distribuídos ou não, para fins diversos. Porém, muitas dessas abordagens focalizam exclusivamente um único aspecto desse universo. Por exemplo, algumas tentam recuperar e representar de forma mais intuitiva e representativa a estrutura do sistema, ou mesmo sua dinâmica de evolução. Outras concentram-se apenas nas pessoas e suas interações, como é caso, principalmente,

dos trabalhos da área de CSCW. Existem ainda aquelas que tentam unir estas duas perspectivas, como é o caso das ferramentas Augur e Lighthouse.

A abordagem apresentada por este trabalho, VIEWS, recai nessa última categoria de trabalhos, isto é, tenta unir sob uma única forma de representação informações provenientes da estrutura do software com informações provenientes do seu processo humano de desenvolvimento. A forma como essa união é realizada e representada é o que diferencia este trabalho daqueles. Ao apresentar tais informações combinando técnicas de visualização pouco exploradas por aqueles trabalhos, a abordagem VIEWS se mostrou capaz de responder de forma intuitiva alguns questionamentos sobre as pessoas e atividades em um projeto de desenvolvimento distribuído de software que antes não conseguiriam ser respondidos facilmente.

A seguir, são apresentadas as principais contribuições (Seção 6.2) e limitações (Seção 6.3) deste trabalho e são também discutidos os principais trabalhos futuros (Seção 6.4) vislumbrados para este projeto.

## 6.2. Contribuições

Esta dissertação apresentou os resultados de um trabalho de pesquisa que visou propor um mecanismo de percepção e uma infra-estrutura reutilizável para apoiar desenvolvedores de software em equipes distribuídas geograficamente, 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 e visualização de software descritos na literatura, destacando os pontos positivos e negativos mais comuns entre essas ferramentas;
- A definição de uma arquitetura padrão capaz de ser reutilizada em mecanismos de percepção semelhantes;
- A implementação de uma infra-estrutura protótipo, baseada na arquitetura definida, capaz de ser reutilizada em mecanismos de percepção e visualização de software semelhante. A implementação foi realizada em linguagem multi-plataforma e disponibiliza duas formas de acesso, incluindo uma interface padronizada e independente de linguagem baseada em *web service*;

- A implementação de um módulo coletor de informação de projeto baseado no amplamente utilizado ambiente Eclipse, totalmente integrado já a infra-estrutura criada. Desta forma, caso outro trabalho queira, por exemplo, visualizar um diferente aspecto não explorado aqui, ou até mesmo apresentar as mesmas informações de uma maneira diferente, tudo que deverá fazer é programar a parte visual e fazer uso da infra-estrutura e coleta de dados fornecida pela VIEWS;
- O conjunto de protótipos de visualização implementados, incluindo um mecanismo de percepção que faz uso tanto das informações do software como das pessoas envolvidas. O mecanismo faz uso de colorações para diferenciar percepções sobre o projeto visualizado, incluindo percepções de conflito direto e indireto;
- A avaliação preliminar da abordagem, através de um estudo controlado, com alunos de pós-graduação, comparando a execução de determinadas tarefas de programação em um ambiente de desenvolvimento distribuído munido de informações fornecidas pela VIEWS e posteriormente sem a mesma.

## 6.3. Limitações

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

- Toda a abordagem foi elaborada para trabalhar exclusivamente com sistemas orientados a objetos (OO). Além disso, todos os testes e a avaliação do trabalho foram feitos a partir de projetos desenvolvidos em linguagem de programação Java. O protótipo de coleta de dados baseado no Eclipse é totalmente dependente desta plataforma, não sendo possível sua utilização com qualquer outro tipo de linguagem. Já a infra-estrutura, apesar de só entender projetos escritos em linguagem Java, já possui em sua arquitetura pontos de extensão para outros tipos de linguagem OO;
- Todos os protótipos são dependentes do ambiente de desenvolvimento Eclipse, a menos a infra-estrutura desenvolvida. Além disso, cabe ressaltar que

modificações realizadas fora do Eclipse e que não tenham sido enviadas, via *web service*, para a infra-estrutura não serão identificadas pelo projeto. Desta forma, ocorrerá um erro de sincronia no modelo;

- O conjunto de informações detectadas e armazenadas pela VIEWS representa um conjunto muito limitado em relação a riqueza de informações disponíveis em um cenário de desenvolvimento distribuído. Apesar da escolha desse conjunto ter sido embasada em outros trabalhos e pesquisas sobre o tema, ainda assim representa de certa forma uma escolha subjetiva por parte deste trabalho. Só um estudo experimental com uma população relevante poderia validar definitivamente o que pode ser considerado pertinente no contexto do desenvolvimento distribuído;
- A avaliação da abordagem foi limitada, sendo restrita a apenas um estudo de uma equipe de duas pessoas distribuídas. Para que exista uma evidência significativa do real ganho no uso desse tipo de abordagem e se mesma possui escalabilidade para tratar de projetos reais, estudos adicionais deverão ser elaborados.

## 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:

- Um estudo experimental com uma população relevante para validar que tipo de informação pode ser considerado pertinente no contexto do desenvolvimento distribuído de software;
- A extensão da infra-estrutura para contemplar outras linguagens de programação e outros tipos de artefatos, além do código fonte. Por exemplo, seria interessante obter informações sobre que está alterando um determinado arquivo de caso de uso. A partir desse tipo de informação, um mecanismo de percepção poderia apresentar quem está desenvolvendo o quê, e até mesmo quais casos de uso (casos de testes, especificações, etc.) estão sendo modificados no momento;

- A captura do histórico do projeto poderia ser implementada na infra-estrutura. Atualmente, apenas quando um projeto começa a trabalhar com a abordagem é que o mesmo passa a ser monitorado e tem suas informações armazenadas. Uma leitura inicial do sistema de controle de versões, para projetos já existentes, poderia resgatar estas informações. Além disso, a forma como essas informações seriam disponibilizadas para aplicações cliente também deverá ser objeto de estudo;
- Implementar mais interatividade no protótipo de visualização implementado traria mais recursos aos desenvolvedores. Por exemplo, seria interessante que fosse implementado um sistema de filtros, onde os usuários pudessem escolher visualizar apenas aqueles artefatos que atendem a algum tipo de critério, como artefatos modificados na última hora, artefatos em conflito, artefatos não modificados em um determinado período;
- No protótipo de visualização, poderiam ser explorados, também, diferentes níveis de abstração, e não apenas o nível de classes, como atualmente encontra-se implementado;
- O planejamento e a execução de um novo estudo, mais completo, e com uma população significativa que possa trazer evidência estatística acerca dos ganhos pretendidos com a abordagem;
- Avaliar a abordagem proposta em um ambiente de aprendizado, como, por exemplo, em uma disciplina em que trabalhos de programação fossem realizados por grupos de pessoas distribuídas;
- Evoluir a abordagem de modo que artefatos gerenciais e aspectos do planejamento das atividades possam ser incorporados nas percepções fornecidas.

# Referências Bibliográficas


AERS, K.ERRO. Developing an Editor for Directed Graphs with the Eclipse Graphical Editing Framework.

AHO, A. V.; LAM, M. S.; SETHI, R.; et al., 2006, *Compilers: Principles, Techniques, and Tools*. 2 ed. Prentice Hall.

BALL, T.; EICK, S. G., 1996, "Software visualization in the large", *Computer*, v. 29, n. 4, pp. 33-43.

BASSIL, S.; KELLER, R. K., 2001, "Software visualization tools: survey and analysis". In: *Program Comprehension, 2001. IWPC 2001. Proceedings. 9th International Workshop onProgram Comprehension, 2001. IWPC 2001. Proceedings. 9th International Workshop on*, pp. 7-17

BIEHL, J. T.; CZERWINSKI, M.; SMITH, G.; et al., 2007, "FASTDash: a visual dashboard for fostering awareness in software teams". In: *Proceedings of the SIGCHI conference on Human factors in computing systems*, p. 1313–1322, New York, NY, USA.

BROOKS, F. P.ERRO. The Mythical Man-Month : Essays on Software Engineering.

BUGZILLA, 2011. Bugzilla. Disponível em: http://www.bugzilla.org/. Acesso em: 16 maio 2011.

CARMEL, E.; TJIA, P., 2005, *Offshoring information technology: sourcing and outsourcing to a global workforce*. Cambridge University Press.

CEPEDA, R. DA S. V.; MAGDALENO, A. M.; MURTA, L. G. P.; et al., 2010, "EvolTrack: improving design evolution awareness in software development", *Journal of the Brazilian Computer Society*, v. 16, n. 2, pp. 117-131.

CHIKOFSKY, E. J.; CROSS II, J. H., 1990, "Reverse Engineering and Design Recovery: A Taxonomy", *IEEE Software*, v. 7, n. 1, pp. 13-17.

CLAYBERG, E.; RUBEL, D., 2006, *Eclipse: Building Commercial-Quality Plug-ins*. 2 ed. Addison-Wesley Professional.

COLLINS-SUSSMAN, B.; FITZPATRICK, B. W.; PILATO, C. M., 2004, *Version control with Subversion*. O'Reilly Media, Inc.

CURBERA, F.; DUFTLER, M.; KHALAF, R.; et al., 2002, "Unraveling the Web services Web: An introduction to SOAP, WSDL, and UDDI", *IEEE Internet Computing*, v. 6, n. 2, pp. 86-93.

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

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

DA SILVA, I. A., 2008, *LIGHTHOUSE: UM MECANISMO DE PERCEPÇÃO DE GRUPO BASEADO NODESIGN EMERGENTE*. Dissertação de Mestrado, COPPE/UFRJ, Rio de Janeiro, RJ, Brasil.

DIEHL, S., 2007, *Software visualization: visualizing the structure, behaviour, and evolution of software*. Springer.

DOURISH, P.; BELLOTTI, V., 1992, "Awareness and coordination in shared workspaces". In: *Proceedings of the 1992 ACM conference on Computer-supported cooperative work*, p. 107–114, New York, NY, USA.

DUBOIS, P., 2008, *MySQL*. 4 ed. Addison-Wesley Professional.

ECLIPSE FOUNDATION, 2011. GEF. Disponível em: http://www.eclipse.org/gef/. Acesso em: 31 mar 2011.

EICK, S. G.; STEFFEN, J. L.; SUMNER, J., 1992, "Seesoft-A Tool for Visualizing Line Oriented Software Statistics", *IEEE Transactions on Software Engineering*, v. 18 (nov.), p. 957–968.

Eclipse, 2011. Eclipse. Disponível em: http://www.eclipse.org/. Acesso em: 21 fev 2011.

FROEHLICH, J.; DOURISH, P., 2004, "Unifying Artifacts and Activities in a Visual Tool for Distributed Software Development Teams". In: *Proceedings of the 26th International Conference on Software Engineering*, pp. 387-396

FROST, R., 2007, "Jazz and the Eclipse Way of Collaboration", *IEEE Software*, v. 24, n. 6, pp. 114-117.

GAMMA, E.; HELM, R.; JOHNSON, R.; et al., 1994, *Design Patterns: Elements of Reusable Object-Oriented Software*. illustrated edition ed. Addison-Wesley Professional.

GERMAN, D. M., 2006, "An empirical study of fine-grained software modifications", *Empirical Software Engineering*, v. 11, pp. 369-393.

GERMAN, D. M.; HINDLE, A.; JORDAN, N., 2004, "Visualizing the evolution of software using softchange". In: *Proceedings 16th International Conference on Software Enginneering and Knowledge Engineering(SEKE 2004)*, pp. 336-341, Banff, Alberta, Canada.

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

HERBSLEB, J. D.; GRINTER, R. E., 1999, "Splitting the organization and integrating the code: Conway's law revisited". In: *Software Engineering, International Conference on*, p. 85, Los Alamitos, CA, USA.

HERBSLEB, J. D.; MOCKUS, A., 2003, "An Empirical Study of Speed and Communication in Globally Distributed Software Development", *IEEE Transactions on Software Engineering*, v. 29, n. 6, pp. 481-494.

HUNT, J. W.; SZYMANSKI, T. G., 1977, "A fast algorithm for computing longest common subsequences", *Communications of the ACM*, v. 20 (maio.), p. 350–353.

HUNT, J.; MCLLROY, M., 1976, *An algorithm for differential file comparison*, Relatório Técnico 41, AT&T Bell Laboratories.

HUPFER, S.; CHENG, L.-T.; ROSS, S.; et al., 2004, "Introducing collaboration into an application development environment". In: *Proceedings of the 2004 ACM conference on Computer supported cooperative work*, p. 21–24, New York, NY, USA.

ISO/IEC, 2005, *ISO/IEC 25000:2005 - Guide to SQuaRE*

JCP, 2009. JSR-000316 Java Platform, Enterprise Edition 6 - Final Release. Disponível em: http://jcp.org/aboutJava/communityprocess/final/jsr316/index.html. Acesso em: 22 fev 2011.

KNODEL, J.; MUTHIG, D.; NAAB, M., 2006, "Understanding software architectures by visualization - an experiment with graphical elements".

KOSCHKE, R., 2002, "Software Visualization for Reverse Engineering". In: *Revised Lectures on Software Visualization, International Seminar*, pp. 138-150

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

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.; 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(SBES - Sessão de Ferramentas)*, pp. 61-66, Brasília, DF, Brasil.

MANGAN, M. A. S.; DA SILVA, I. A.; WERNER, C. M. L., 2004, "GAW: uma ferramenta de percepção de grupo aplicada o desenvolvimento de software". In:

*Anais do XVIII Simpósio Brasileiro de Engenharia de Software (SBES)*, pp. 25-30, Brasília, DF, Brasil.

MARCHIONI, F., 2010, *JBoss AS 5 Development*. Packt Publishing.

MEI, H.; CAO, D.-G.; YANG, F.-Q., 2006, "Development of Software Engineering: A Research Perspective", *Journal of Computer Science and Technology*, v. 21, n. 5, pp. 682-696.

MOCKUS, A.; HERBSLEB, J., 2001, "Challenges of Global Software Development". In: *Software Metrics, IEEE International Symposium on*, p. 182, Los Alamitos, CA, USA.

MURTA, L.; CORRÊA, C.; PRUDÊNCIO, J. G.; et al., 2008, "Towards odyssey-VCS 2: improvements over a UML-based version control system". In: *Proceedings of the 2008 international workshop on Comparison and versioning of software models*, pp. 25-30, Leipzig, Germany.

OMG, 2011. Object Management Group - UML. Disponível em: http://www.uml.org/. Acesso em: 2 jun 2011.

ORACLE, 2011. Java. Disponível em: http://www.oracle.com/technetwork/java/index.html. Acesso em: 21 fev 2011.

PRIKLADNICKI, R.; AUDY, J., 2008, *Desenvolvimento Distribuído de Software*. 1 ed. Campus.

Polarion, 2011. Subversive. Disponível em: http://www.eclipse.org/subversive/. Acesso em: 22 fev 2011.

ROBERTSON, G. G.; CARD, S. K.; MACKINLAY, J. D., 1993, "Information visualization using 3D interactive animation", *Communications of the ACM*, v. 36 (abr.), p. 57–71.

SCHÜMMER, T.; HAAKE, J. M., 2001, "Supporting distributed software development by modes of collaboration". In: *Proceedings of the seventh conference on European Conference on Computer Supported Cooperative Work*, p. 79–98, Norwell, MA, USA.

SHULL, F.; CARVER, J.; TRAVASSOS, G. H., 2001, "An empirical methodology for introducing software processes". In: *ACM SIGSOFT Software Engineering Notes*, p. 288–296, New York, NY, USA.

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

STOREY, M.-A. D.; ČUBRANIĆ, D.; GERMAN, D. M., 2005, "On the use of visualization to support awareness of human activities in software development: a survey and a framework". In: *Proceedings of the 2005 ACM symposium on Software visualization*, pp. 193-202, St. Louis, Missouri.

SUN, D.; WONG, K., 2005, "On Evaluating the Layout of UML Class Diagrams for Program Comprehension". In: *International Conference on Program Comprehension*, pp. 317-326, Los Alamitos, CA, USA.

VOINEA, L.; TELEA, A.; VAN WIJK, J. J., 2005, "CVSscan: visualization of code evolution". In: *Proceedings of the 2005 ACM symposium on Software visualization*, p. 47–56, New York, NY, USA.

VOINEA, L.; TELEA, A., 2006, "Mining software repositories with CVSgrab". In: *Proceedings of the 2006 international workshop on Mining software repositories*, p. 167–168, New York, NY, USA.

WALDO, J., 1998, "Remote procedure calls and Java Remote Method Invocation", *Concurrency, IEEE*, v. 6, n. 3, pp. 5-7.

WALL, L.; CHRISTIANSEN, T.; ORWANT, J., 2000, *Programming Perl*. 3 ed. O'Reilly Media, Inc.

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

WOHLIN, C.; RUNESON, P.; HÖST, M., 1999, *Experimentation in Software Engineering: An Introduction*. 1 ed. Springer.

# ANEXO A– FORMULÁRIO DE CONSENTIMENTO

## Formulário de Consentimento

### Estudo

Este estudo visa à caracterizar a capacidade da abordagem *VIEWS* em responder questões pertinentes ao processo de desenvolvimento de software distribuído.

### Idade

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

### Procedimento

Este estudo acontecerá em uma única sessão, composta de duas etapas. Em ambas as etapas os participantes deverão executar um conjunto de tarefas e responder a uma série de questionamentos relacionados as atividades realizadas. Eu entendo que, uma vez o experimento tenha terminado, os trabalhos que desenvolvi serão estudados visando a entender a eficiência dos procedimentos e as técnicas propostas.

### 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. O histórico de todas as comunicações que por ventura sejam registradas será armazenado apenas para fins de estudos.

### Benefícios e liberdade de desistência

Eu entendo que os benefícios que receberei deste estudo são limitados ao aprendizado do material que é distribuído e apresentado. 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 visualização de software e o desenvolvimento distribuído.

### Pesquisador responsável

Rafael da Silva Viterbo de Cepêda
Programa de Engenharia de Sistemas e Computação - COPPE/UFRJ

### Professor responsável

Profª. Cláudia Maria Lima Werner
Programa de Engenharia de Sistemas e Computação - COPPE/UFRJ

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

# ANEXO B- QUESTIONÁRIO DE CARACTERIZAÇÃO

## Questionário de Caracterização

### 1) Formação acadêmica

( ) Doutorado
( ) Doutorando
( ) Mestrado
( ) Mestrando
( ) Graduação

Ano de ingresso: _____ Ano de conclusão (ou previsão de conclusão): _____

### 2) Formação geral

2.1) Qual é a sua experiência com desenvolvimento de *software* em Java?
(marque aquele item que melhor se aplica)
( ) já li material sobre desenvolvimento de *software* 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, relacionada 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. (Ex.: "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 já participou? (Ex.: "Eu trabalhei com equipes distribuídas globalmente. A maior equipe que participei tinha 29 pessoas")
_____
_____
_____
_____

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 |
| 2.4.5) Visualização de Software | 0 | 1 | 2 | 3 | 4 |

**Obrigado por sua colaboração!**

# ANEXO C- QUESTIONÁRIO DE AVALIAÇÃO

## Questionário de Avaliação

### Etapa do Estudo que não utiliza a abordagem VIEWS

| | |
|---|---|
| 1) Você sentiu dificuldades em responder as questões nesta etapa? Especifique. | ☐ Sim ☐ Não ☐ Parcialmente |
| | |
| | |
| | |

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

| | |
|---|---|
| 3) Você acha a abordagem usada nesta etapa adequada para responder aos questionamentos realizados? Explique o porquê da sua resposta. | ☐ Sim ☐ Não ☐ Parcialmente |
| | |
| | |
| | |

## Etapa do Estudo que usa a abordagem VIEWS

| 4) Você sentiu dificuldades em responder as questões nesta etapa? Especifique. | ☐ Sim ☐ Não ☐ Parcialmente |
|---|---|
| | |
| | |
| | |

| 5) Você ficou satisfeito com o resultado final das tarefas nesta etapa? Especifique, se necessário. | ☐ Sim ☐ Não ☐ Parcialmente |
|---|---|
| | |
| | |
| | |

| 6) Você acha a estratégia usada nesta etapa adequada para responder aos questionamentos realizados? Explique o porquê da sua resposta. | ☐ Sim ☐ Não ☐ Parcialmente |
|---|---|
| | |
| | |
| | |

| 7) Você tem alguma sugestão para melhorar a abordagem? Especifique. | ☐ Sim ☐ Não |
|---|---|
| | |
| | |

**Obrigado por sua colaboração!**

# ANEXO D- DESCRIÇÃO GERAL DA TAREFA

## Descrição Geral da Tarefa

### Instruções

- Antes da execução de cada tarefa, aguarde autorização para o seu inicio;
- Após cada tarefa, indique se o resultado foi obtido a partir da lista de atividades solicitadas ou se foi utilizado algum outro recurso. Neste caso, descreva qual;
- Novas informações sobre o projeto serão fornecidas ao longo do estudo;
- Registre o **horário de inicio** e o **horário de fim** das tarefas sempre que solicitado na lista de tarefas;
- Para este estudo, sempre que o valor da cardinalidade de um relacionamento estiver ausente, tal valor deve ser considerado [1..1]. Vale lembrar que associações do tipo generalização (herança) não possuem cardinalidade;
- Nos modelos apresentados, as palavras *src* e *dst* indicam respectivamente a origem e o destino de uma determinada associação;
- Antes da execução de cada tarefa, leia-a inteiramente e consulte o último modelo disponível;
- Responda as perguntas do questionário **na ordem em que elas são apresentadas**;
- Pergunte e comente tudo que achar necessário.

### Contextualização

Você foi contratado(a) como engenheiro de software pela empresa de desenvolvimento YesWeCan Hotels Software S/A, sendo alocado inicialmente em atividades de manuteção evolutiva de sistemas. O sistema que você irá inicialmente trabalhar é o *Emirates.* Este sistema é um dos mais bem sucedidos da empresa. Ele é responsável por toda a gestão da cadeia de hotéis *Emirates Palace*, perfazendo todos os seus processos de reservas, relacionamento com os clientes e fornecedores. É importante notar que o sistema conta com uma integração de toda a rede *Emirates Palace*, incluindo cidades como Rio de Janeiro, Nova York e Paris. A sua equipe, hoje, contém mais um integrante que se encontra situado, atualmente, em Angola. Apesar da distância, este integrante poderá ser sempre contatado via mensagens instantâneas de texto. A equipe utiliza o ambiente Eclipse com um sistema de controle de versões Subversion no desenvolvimento do projeto. Em breve, você receberá suas primeiras tarefas de implementação no projeto.

# ANEXO E- TAREFAS DO ESTUDO

## Tarefas do Participante A

**Tarefa 1: Modificar: "Cliente.java" no pacote "entidades"**

1. Adicionar atributo privado do tipo String: *login*.
2. Adicionar atributo privado do tipo String: *senha*.
3. Adicionar métodos públicos: *setLogin(String login)* e *getLogin()* para acesso ao atributo *login.*
4. Adicionar métodos públicos: *setSenha(String senha)* e *getSenha()* para acesso ao atributo *senha.*

**Tarefa 1: Modificar: "Cliente.java" no pacote "entidades"**

Hora de Inicio: _ _ : _ _

Hora de Término: _ _ : _ _

_____
_____
_____
_____
_____
_____
_____
_____

**Tarefa 2: Criar: "DataHora.java" no pacote "utilitarios"**

1. Adicionar classe *DataHora* pacote *utilitários*, que fará conversões de data e hora.
2. Adicionar método estático e público:
   *public static Date stringParaData(String data)*;
3. Adicionar método estático e público:
   *public static String dataParaString(Date data)*;

**Tarefa 2: Criar: "DataHora.java" no pacote "utilitarios"**

Hora de Inicio: _ _ : _ _

Hora de Término: _ _ : _ _

_____
_____
_____
_____
_____
_____
_____
_____

**Tarefa 3: Modificar: "Reserva.java" no pacote "entidades"**

1. Adicionar atributo privado do tipo String: *dataHoraString*.
2. Adicionar método público: *setDataHoraString(String dataHoraString)*.
3. Adicionar método público: *getDataHoraString()* para acesso ao atributo *dataHoraString*.
4. Importe a classe *DataHora: import utilitarios.DataHora;*
5. Implementar na última linha do método público *setDataHora* o seguinte código:
   *this.dataHoraString = DataHora.dataParaString(dataHora);*

**Tarefa 3: Modificar: "Reserva.java" no pacote "entidades"**

Hora de Inicio: _ _ : _ _
Hora de Término: _ _ : _ _

_____
_____
_____
_____
_____
_____
_____
_____

**Tarefa 4: Modificar: "ClienteDAO.java" no pacote "daos"**

1. Adicionar método público: *limpar()*.
2. Adicionar métodos públicos: *setCliente(Cliente cliente)* e *getCliente* para acesso ao atributo *cliente*.
3. Implementar dentro do método *limpar()* um conjunto de

operações para limpar cada atributo do objeto *Cliente* associado: Exemplo: "*cliente.setId(“”); cliente.setCpf(“”);* ..."

4. Adicionar o método público *alterarCliente()* com a seguinte operação: "*throw new UnsupportedOperationException(“Not yet supported”);*"
5. Implementar dentro do método *deletar()* a seguinte operação: "*throw new UnsupportedOperationException(“Not yet supported”);*"

**Tarefa 4: Modificar: “ClienteDAO.java” no pacote “daos”**
Hora de Inicio: _ _ : _ _
Hora de Término: _ _ : _ _

_____
_____
_____
_____
_____
_____
_____
_____

**Tarefa 5: Modificar: “ControleAutenticacao.java” no pacote “seguranca”**

1. Adicionar no inicio do método público *invoke o seguinte comando: “System.out.println(“Chamando invoke...”);”*.

**Tarefa 5: Modificar: “ControleAutenticacao.java” no pacote “seguranca”**
Hora de Inicio: _ _ : _ _
Hora de Término: _ _ : _ _

_____
_____
_____
_____
_____
_____
_____
_____

**Tarefa 6: Realizar *Commit***

1. Executar comando de *commit* para todo o projeto.

**Tarefa 6: Realizar *Commit***

Hora de Inicio: _ _ : _ _

Hora de Término: _ _ : _ _

_____
_____
_____
_____
_____
_____
_____
_____

# Tarefas do Participante B

**Tarefa 1: Modificar: "Reserva.java" no pacote "entidades"**

1. Adicionar atributo privado do tipo String: *dataHoraString*.
2. Adicionar método público: *setDataHoraString(String dataHoraString)*.
3. Adicionar método público: *getDataHoraString()* para acesso ao atributo *dataHoraString*.
4. Adicionar atributo privado do tipo String: *situacao*.
5. Adicionar método público: *setSituacao(String situacao)*.
6. Adicionar método público: *getSituacao()* para acesso ao atributo *situacao*.

**Tarefa 1: Modificar: "Reserva.java" no pacote "entidades"**

Hora de Inicio: _ _ : _ _

Hora de Término: _ _ : _ _

_____
_____
_____
_____
_____
_____
_____
_____

**Tarefa 2: Modificar: "Quarto.java" no pacote "entidades"**

1. Adicionar atributo privado do tipo String: *situacao*.
2. Adicionar atributo privado do tipo String: *nPessoas*.
3. Adicionar métodos públicos: *setSituacao(String situacao)* e *getSituacao()* para acesso ao atributo *situacao.*
4. Adicionar métodos públicos: *setNPessoas(String nPessoas)* e *getNPessoas()* para acesso ao atributo *nPessoas.*

**Tarefa 2: Modificar: "Quarto.java" no pacote "entidades"**

_____

Hora de Inicio: _ _ : _ _
Hora de Término: _ _ : _ _
_____
_____
_____
_____
_____
_____
_____

**Tarefa 3: Modificar: "Perfil.java" no pacote "seguranca"**

1. Alterar a assinatura do método público *temAcesso(String classe, String metodo)* para: *temAcesso(String metodoDeAcesso, String classeDeAcesso)*
2. Alterar no corpo do método *temAcesso* referência para os novos parâmetros de entrada.

**Tarefa 3: Modificar: "Perfil.java" no pacote "seguranca"**
Hora de Inicio: _ _ : _ _
Hora de Término: _ _ : _ _
_____
_____
_____
_____
_____
_____
_____
_____

**Tarefa 4: Realizar *Commit***

1. Executar comando de *commit* para todo o projeto.

**Tarefa 4: Realizar *Commit***
Hora de Inicio: _ _ : _ _
Hora de Término: _ _ : _ _
_____
_____

_____
_____
_____
_____
_____
_____

**Tarefa 5: Modificar: “ClienteDAO.java” no pacote “daos”**

1. Adicionar método público: *limpar()*.
2. Adicionar métodos públicos: *setCliente(Cliente cliente)* e *getCliente* para acesso ao atributo *cliente*.
3. Implementar dentro do método *limpar()* um conjunto de operações para limpar cada atributo do objeto *Cliente* associado: Exemplo: “*cliente.setId(“”); cliente.setCpf(“”);* ...”
4. Adicionar o método público *alterarCliente()* com a seguinte operação: “*throw new UnsupportedOperationException(“Not yet supported”);*”

**Tarefa 5: Modificar: “ClienteDAO.java” no pacote “daos”**

Hora de Inicio: _ _ : _ _

Hora de Término: _ _ : _ _

_____
_____
_____
_____
_____
_____
_____
_____

**Tarefa 6: Criar classe utilitária no pacote “utilitarios”**

1. Adicionar classe no pacote *utilitarios* com o objetivo de tratar datas e horas. A classe poderá conter o nome que você desejar e deverá conter métodos para a conversão do tipo *Date* em *String* e vice-versa: Exemplo: *String convertDateString(Date data)*;

**Tarefa 6: Criar classe utilitária no pacote “utilitarios”**

Hora de Inicio: _ _ : _ _
Hora de Término: _ _ : _ _

_____
_____
_____
_____
_____
_____
_____
_____

**Tarefa 7: Realizar *Commit***

1. Executar comando de *commit* para todo o projeto.

**Tarefa 7: Realizar *Commit***

Hora de Inicio: _ _ : _ _
Hora de Término: _ _ : _ _

_____
_____
_____
_____
_____
_____
_____
_____

# ANEXO F- QUESTIONÁRIO DE PERCEPÇÕES

## Questões com as Percepções obtidas no Estudo de Desenvolvimento Distribuído de Software

1) Quais foram as classes que você alterou ou criou ao longo do estudo? Indique como você obteve a resposta da pergunta anterior.

_____
_____
_____

2) Você conseguiu identificar, ao longo do estudo, quais foram as classes que o seu colega de equipe alterou? Quais foram? Indique como você obteve a resposta da pergunta anterior.

_____
_____
_____

3) Você conseguiu identificar algum conflito antes que o mesmo ocorresse em tempo de *commit*? Indique quais foram estas situações.

_____
_____
_____
_____
_____
_____

4) Na sua opinião, que tipo de recurso lhe ajudou a identificar o(s) conflito(s) citado(s) na pergunta anterior? (*responder caso tenha identificado algum*)

_____
_____
_____
_____

_____
_____

5) Na sua opinião, o que dificultou a identificação do(s) conflito(s) citado(s) na questão 3? (*responder caso não tenha identificado nenhum*)

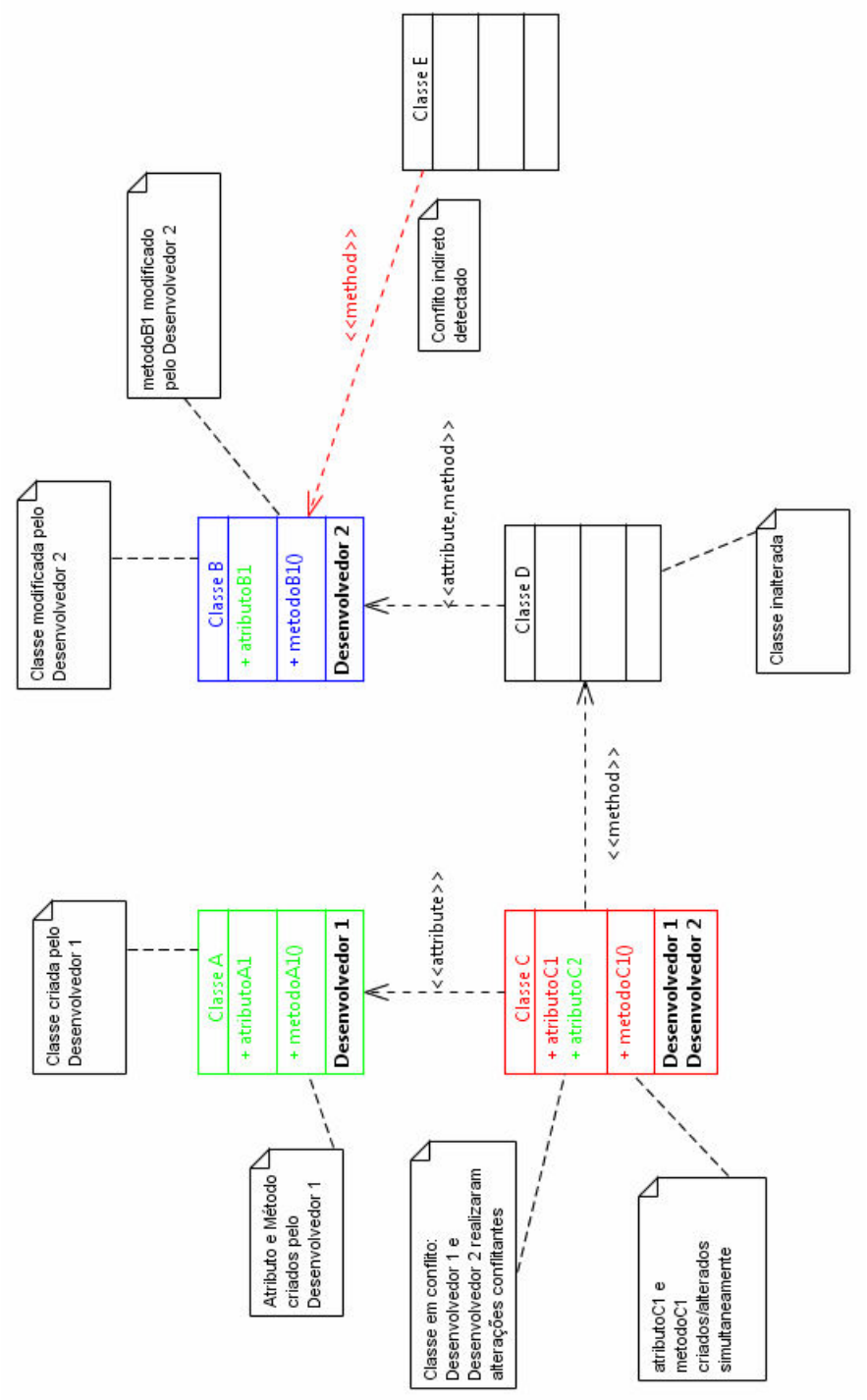
_____
_____
_____
_____
_____
_____

6) Você foi capaz de reutilizar alguma classe previamente criada pelo outro membro da equipe? Em caso negativo, cite um possível motivo para a não identificação desta oportunidade.

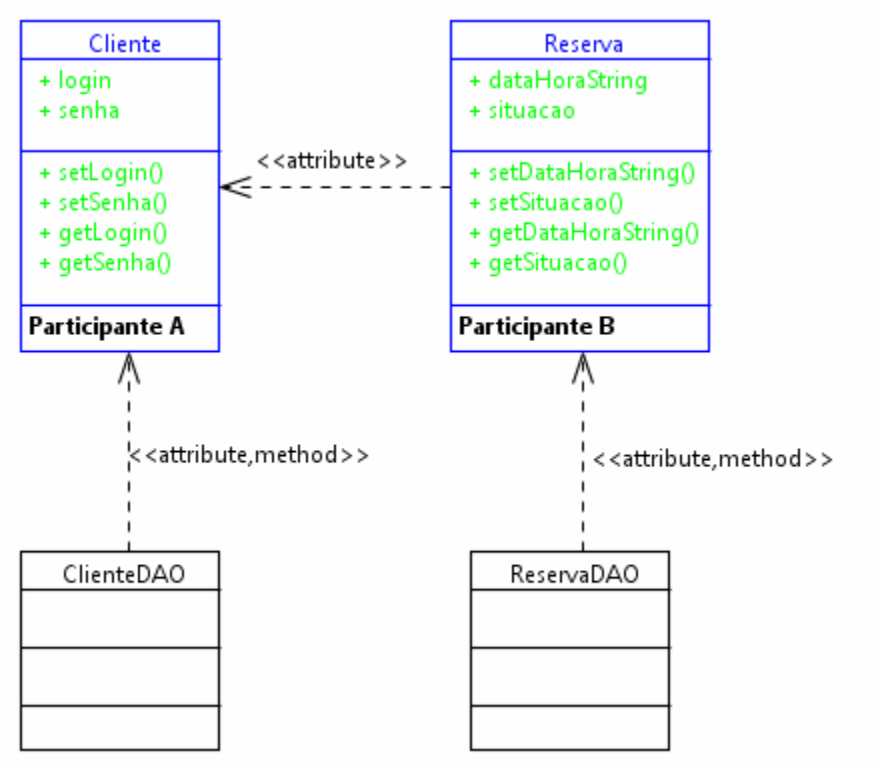
_____
_____
_____
_____
_____
_____

# ANEXO G- MODELOS UTILIZADOS NO ESTUDO

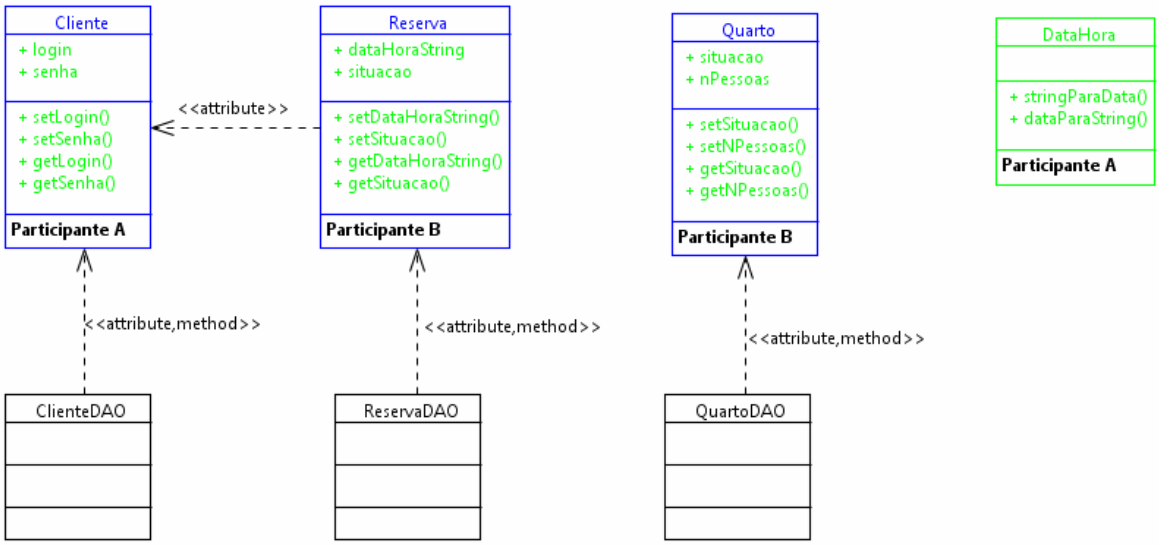
## Etapa com VIEWS – Modelo de Treinamento



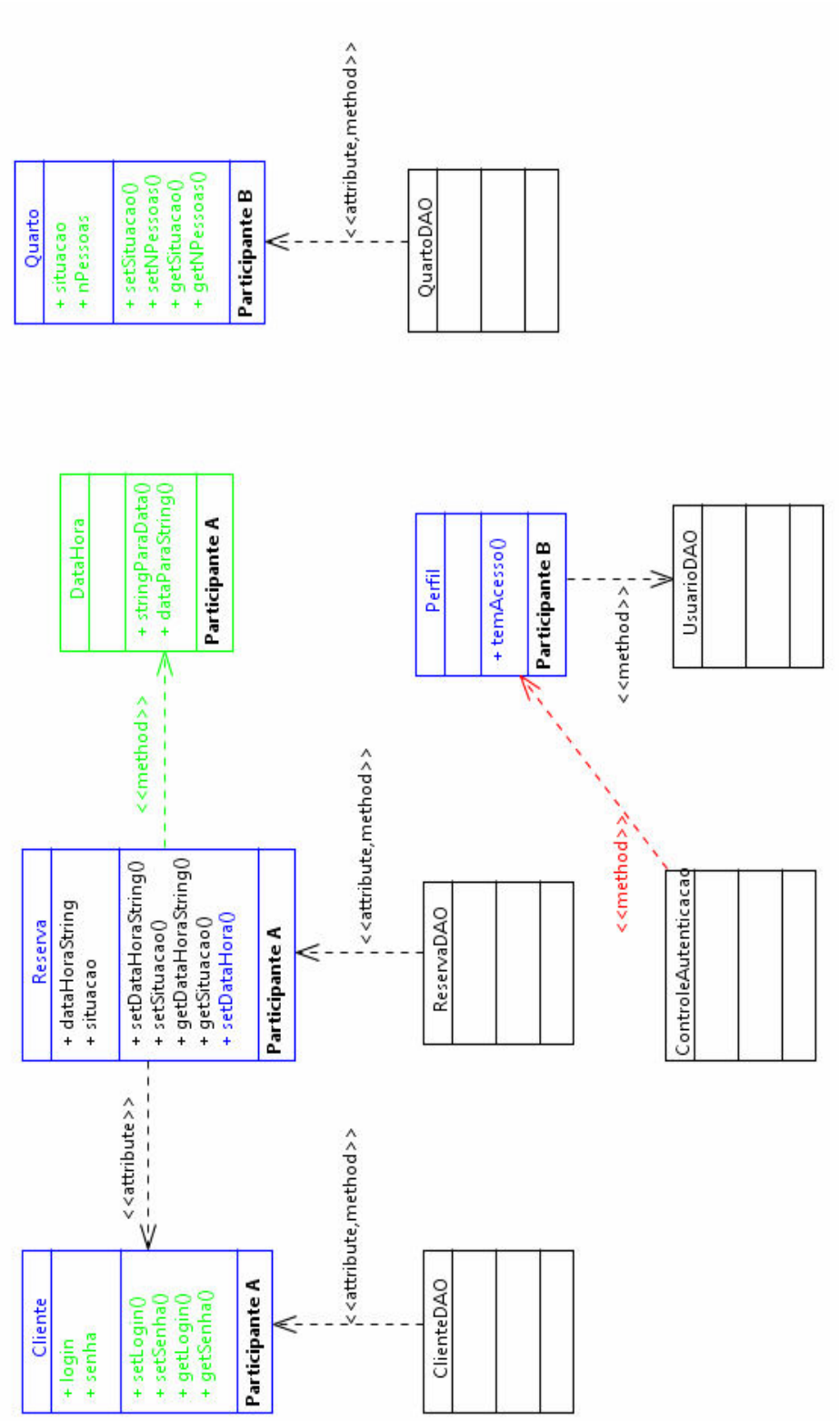
## Etapa com VIEWS – Modelo 1



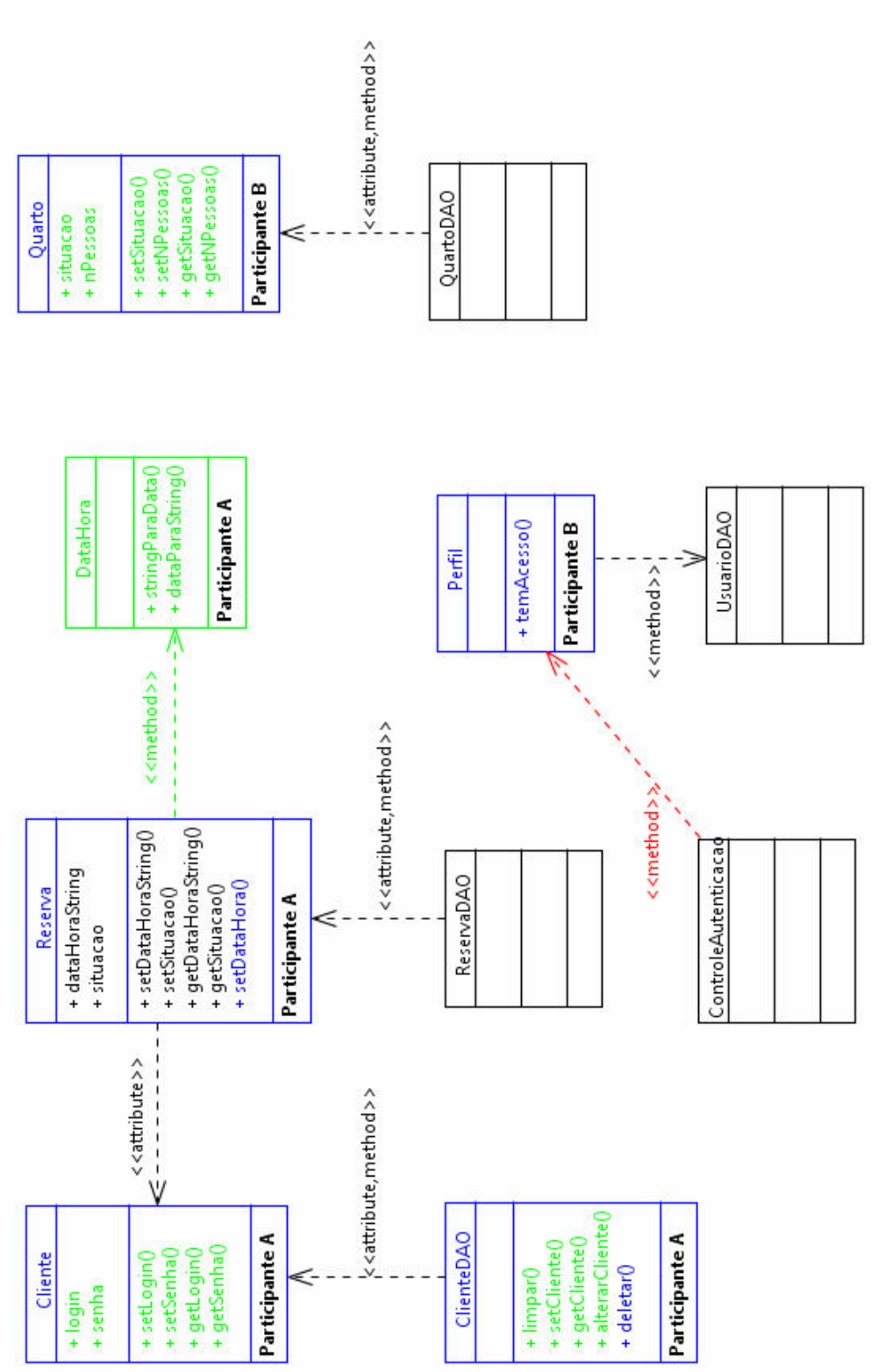

## Etapa com VIEWS – Modelo 2



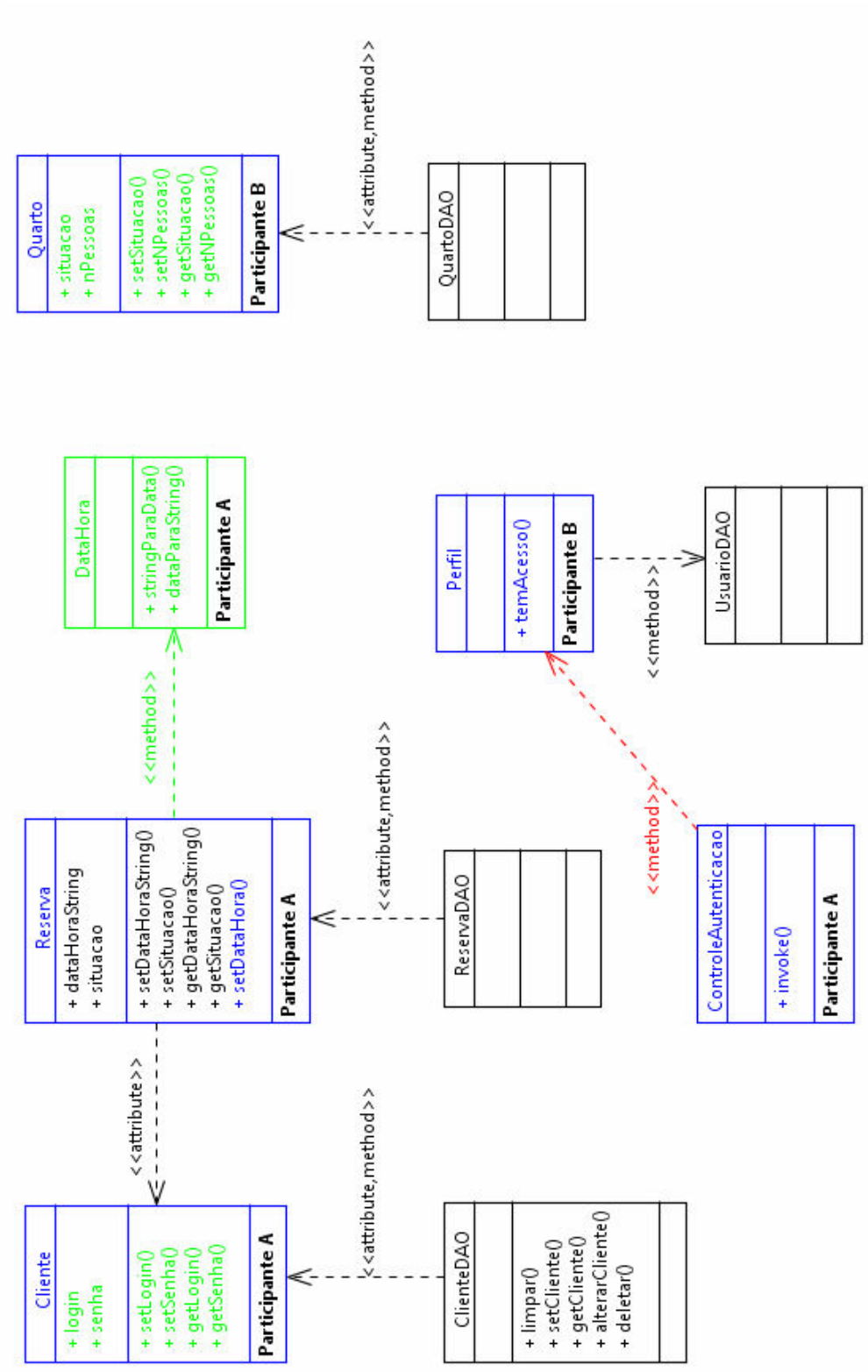
# Etapa com VIEWS – Modelo 3



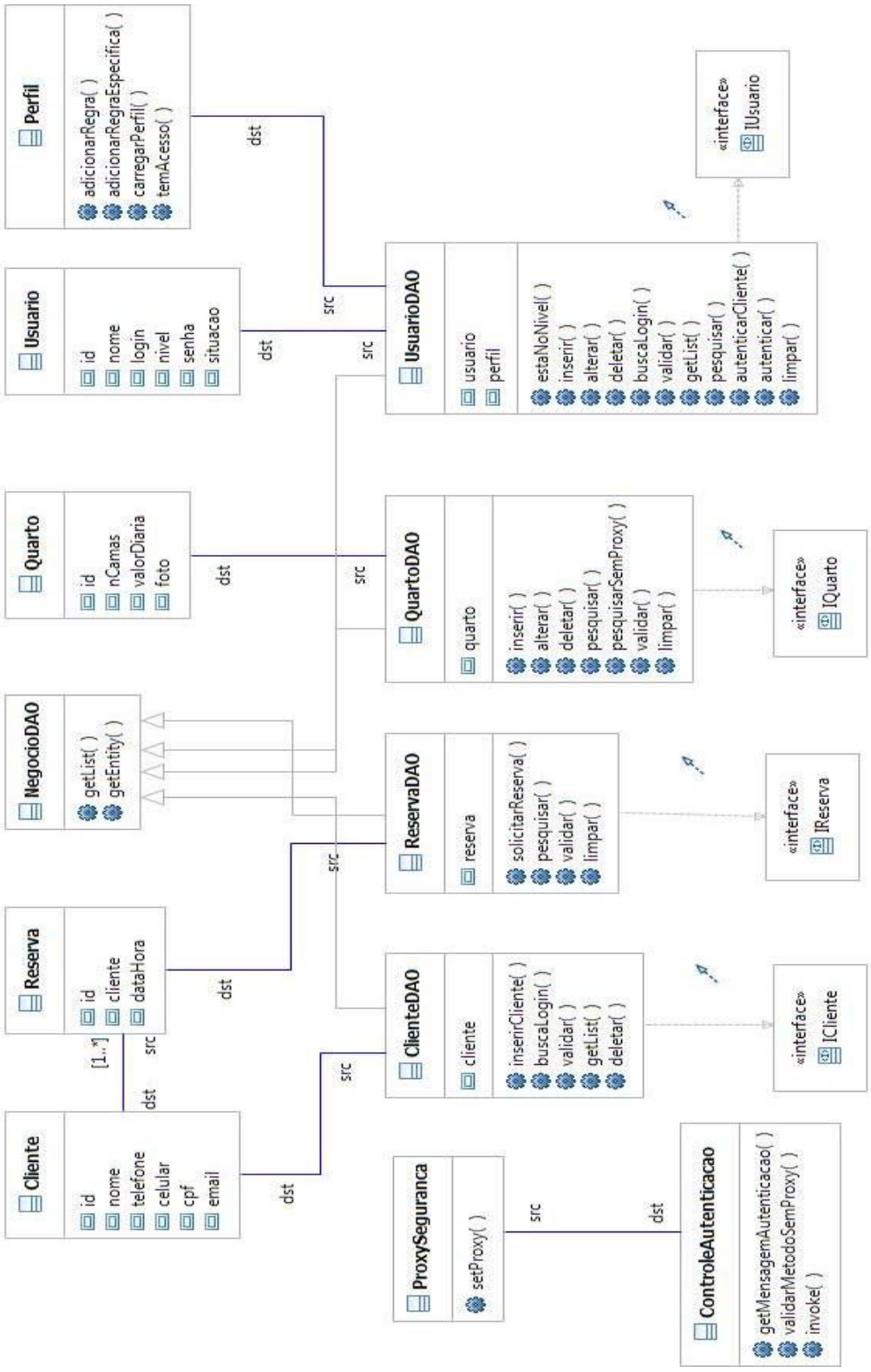
# Etapa com VIEWS – Modelo 4



# Etapa com VIEWS – Modelo 5



# Etapa sem VIEWS - Modelo 1



# Etapa sem VIEWS - Modelo 2