UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE CIÊNCIAS E TECNOLOGIA
CURSO DE PÓS-GRADUAÇÃO EM INFORMÁTICA

DISSERTAÇÃO DE MESTRADO

CVSyn – Acoplando um Mecanismo de Notificação Síncrono ao CVS para Otimizar a Comunicação no Desenvolvimento Global de Software

Pasqueline Lacerda Dantas

Francilene Procópio Garcia
(Orientadora)

Campina Grande – PB
Abril - 2004

UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE CIÊNCIAS E TECNOLOGIA
CURSO DE PÓS-GRADUAÇÃO EM INFORMÁTICA

CVSyn – Acoplando um Mecanismo de Notificação Síncrono ao CVS para Otimizar a Comunicação no Desenvolvimento Global de Software

Pasqueline Lacerda Dantas

Francilene Procópio Garcia
(Orientadora)

Dissertação submetida à Coordenação do Curso de Pós-graduação em Informática da Universidade Federal de Campina Grande, como parte dos requisitos necessários para obtenção do grau de mestre em Informática.

Área de concentração: Ciência da Computação.

Linha de pesquisa: Engenharia de Software.

Campina Grande – PB
Abril – 2004

Ficha Catalográfica

DANTAS, Pasqueline Lacerda

D192A

CVSyn – Acoplando um Mecanismo de Notificação Síncrono ao CVS para Otimizar a Comunicação no Desenvolvimento Global de Software

Dissertação de Mestrado, Universidade Federal de Campina Grande, Centro de Ciências e Tecnologia, Coordenação de Pós-graduação em Informática, Campina Grande-PB, abril de 2004.

108 p. Il.

Orientadora: Francilene Procópio Garcia

Palavras-chave:

1. Engenharia de Software
2. Desenvolvimento Global de Software
3. Servidores de Notificação de Eventos
4. CVS

CDU – ~~519.683~~ 004.41(043)

"CVSyn - ACOPLANDO UM MECANISMO DE NOTIFICAÇÃO SÍNCRONO AO CVS PARA OTIMIZAR A COMUNICAÇÃO NO DESENVOLVIMENTO GLOBAL DE SOFTWARE"

PASQUÊLINE LACERDA DANTAS

DISSERTAÇÃO APROVADA EM 26.04.2004

PROFª FRANCILENE PROCÓPIO GARCIA, D.Sc
Orientadora

PROF. JOSÉ ANTÃO BELTRÃO MOURA, Ph.D
Examinador

PROF, ÁLVARO FRANCISCO DE CASTRO MEDEIROS, D.Sc
Examinador

CAMPINA GRANDE – PB

Agradecimentos

Aos meus pais, Vandorli e Didi, meus maiores mestres e a quem tudo devo.

A Wagner, meu irmão, e Carol, minha fiel amiga. Agradeço a paciência, a serenidade e a união nos momentos difíceis. Sempre lembrarei dos momentos alegres que tivemos em Campina Grande.

A Rohit Gheyi, pelos bons momentos juntos e por todo o tempo em que a nossa convivência me tornou uma pessoa melhor. Meus sinceros agradecimentos ao amigo mais companheiro que tive até hoje.

A César, pelo amor, carinho e dedicação.

À minha orientadora, Francilene Garcia, pela confiança e serenidade com que conduziu o nosso trabalho.

Roberta, Flavinha, Giselle, Vanessa e Edemberg, por cada uma das vezes em que estivemos juntos e pude aprender grandes lições.

A Fabrício, pela amizade e pelo carinho que sempre sentimos um pelo outro.

Aos amigos do Rubi, Ricardo Oliveira, Tiago Lamenha e Daniel Paranhos.

A Rodrigo Rebouças, pela amizade.

A Aninha e Vera pela atenção e dedicação.

Resumo

O Desenvolvimento Global de Software é uma abordagem que requer muito investimento, sobretudo com tecnologia. Neste contexto, são as organizações de menor porte aquelas que mais percebem os impactos causados pela separação física dos times e pelas lacunas deixadas na comunicação. Como nem sempre as ferramentas utilizadas facilitam o entendimento das mudanças que ocorrem durante a contrução da aplicação, muitas distorções são geradas, e conseqüentemente, muito re-trabalho.

Este trabalho destaca a importância de ferramentas de controle de versão, assinalando aspectos colaborativos que podem auxiliar o processo de produção de código e integração na presença de colaboradores distribuídos geograficamente. Uma ferramenta foi proposta e um protótipo foi implementado tendo como base o CVS e um servidor de notificação de eventos. O objetivo do CVsyn é permitir que a comunicação (compartilhamento de informação) entre os colaboradores do projeto possa ser melhorada por meio de um mecanismo de notificação.

Abstract

Global Software Development is an approach that requires a lot of investment, over all with technology. In this context, the small and medium sized organizations are those that suffer the impacts caused by physical separation of their teams and the gaps in their communication the most. As not always the used tools facilitate the understanding of the changes that happen during the application evolution, much misunderstanding is generated, and consequently, much re-work.

This work points the importance of tools of version control, marking collaborative aspects that can aid the process of code production and integration in the presence of collaborators geographically distributed. A tool was proposed and a prototype was implemented based in CVS and an event notification server. The objective of CVsyn is to allow that the communication (sharing of information) among the collaborators of the project can be improved through a mechanism of notification.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Quadros

Capítulo 1

Introdução

1.1 Contextualização

Nos últimos tempos, pode-se observar que o panorama do mercado de software tem se modificado, principalmente na maneira com que as organizações concebem seus produtos. Se por um lado, a globalização, estimulada em grande parte pela popularização da Internet, tornou os consumidores muito mais exigentes, por outro, permitiu que as barreiras físicas que separavam os mercados produtores e consumidores desaparecessem.

Novas modalidades de desenvolvimento, a exemplo da abordagem de código-livre (open-source) (JAGIELSKI,1999), Desenvolvimento Baseado em Componentes (BRERETON,2000) e Desenvolvimento Global (BENNATAN,2002), entre outras, surgem para acomodar as mudanças.

A primeira grande adaptação refere-se ao fato de que tais estilos contrariam um modelo de desenvolvimento centralizado, onde as equipes do desenvolvimento necessariamente precisam estar localizadas em uma mesma estrutura física. A possibilidade de construir aplicações trabalhando com times que estão distribuídos em quaisquer partes do país ou até mesmo do mundo é um fato.

Erran Carmel (CARMEL,2001) sugere como aspecto marcante, o caráter altamente cooperativo que rege a realização das tarefas. Por vários motivos (formação de parcerias de negócio, os serviços terceirizados e as novas oportunidades de negócio), os times de desenvolvimento, muitas vezes pertencentes à organizações diferentes, precisam compartilhar experiências e, sobretudo, informações para que as metas do projeto possam ser atingidas.

Muitas estratégias podem ser adotadas para particionar os esforços do desenvolvimento (BENNATAN,2002). A escolha implica um nível maior ou menor de dependência com relação aos artefatos que serão desenvolvidos em separado. O fato é que, como os parceiros do desenvolvimento podem estar trabalhando paralelamente sob o

mesmo módulo do projeto, o resultado das tarefas de um pode interferir no sucesso do trabalho de outros colaboradores do projeto.

Embora o Desenvolvimento Global ofereça muitas vantagens, como o lançamento de um mesmo produto em múltiplos mercados, o usufruto das vantagens competitivas de mercados estrangeiros (mão-de-obra mais barata, contratação de bons especialistas) e os incentivos fiscais, a abordagem apresenta muitos desafios dentre eles, a carência de metodologias de desenvolvimento que consideram as particularidades do desenvolvimento para cenários globais (DANTAS,2003).

O desenvolvimento distribuído[1], em níveis nacionais ou internacionais, produz desafios resultantes da dificuldade em gerenciar a comunicação que é afetada pela distância existente entre os times do projeto, os chamados **times virtuais**. Naturalmente as pessoas tendem a se comunicar mais quando estão localizadas em um mesmo lugar. Segundo James Herbsleb (HERBSLEB,1999), um dos efeitos mais críticos da separação geográfica dos times é que, remotamente a resolução de conflitos[2] se torna mais difícil.

Neste contexto, diferentes tipos de distâncias produzem diferentes tipos de problemas, principalmente quando o projeto envolve pessoas com culturas, valores e idiomas diferentes. Neste âmbito, a utilização de tecnologia colaborativa torna-se fundamental porque permite que os colaboradores mesmo trabalhando localmente possam compartilhar o resultado das suas tarefas com outros colaboradores (mais detalhes sobre tecnologia colaborativa podem ser encontrados no Capítulo 2, Seção 2.1).

Na fase de construção da aplicação, a facilidade de detectar e resolver conflitos está diretamente associada ao fato de como os colaboradores do projeto compartilham as informações. Percebe-se, dessa forma, que melhorar a comunicação é a essência para o sucesso de um projeto distribuído. Isto requer ferramentas, métodos e sobretudo, um entendimento dos aspectos que envolvem a cooperação entre os parceiros envolvidos no desenvolvimento.

[1] No decorrer deste trabalho, o termo desenvolvimento distribuído é utilizado para denotar o fato de que os colaboradores de um projeto encontram-se separados fisicamente. Esta é uma característica do Desenvolvimento Global.

[2] Em um cenário de desenvolvimento distribuído, os conflitos são considerados distorções ou mal-entendimentos causados pela separação física, pelas diferenças culturais ou por falhas no processo de comunicação.

1.2 Motivação

O suporte ferramental usado no projeto deve permitir não apenas a automação das tarefas, tais como a montagem da aplicação, a execução de testes e a integração de código, mas auxiliar a comunicação entre os times que trabalham de forma distribuída. Durante a evolução da aplicação, o apoio tecnológico deve prover a consistência dos artefatos ao mesmo tempo em que deve adequar mecanismos que reduzam as distorções que podem ser geradas pelas distâncias geográficas ou culturais.

Nem sempre as ferramentas utilizadas no projeto conseguem atingir propósitos tão específicos. Os sistemas de gerenciamento de configuração, por exemplo, enquanto fornecem algum suporte automatizado para a coordenação dos artefatos que são produzidos deixam a desejar no que se refere a garantir que um mínimo de conflitos durante a integração aconteça (SARMA,2000). Mecanismos síncronos de notificação, em geral, são raros nas ferramentas, o que obriga os desenvolvedores a utilizarem outros dispositivos para viabilizar a comunicação com o restante do time, que são, em muitas situações ineficazes, como e-mails e inviáveis para o orçamento do projeto, como é o caso dos telefonemas internacionais.

Quando há times distribuídos participando no desenvolvimento, alguns aspectos devem ser considerados durante a etapa de construção de código:

i. a velocidade com que os conflitos são resolvidos é crítica já que do contrário, as distorções causadas nesta etapa podem gerar atrasos no cumprimento do cronograma e o comprometimento da qualidade dos artefatos gerados. É necessário adequar o suporte oferecido para que os colaboradores utilizem mecanismos de comunicação síncrona em situações de dúvida, de conflitos conceituais ou quando necessitam disseminar informações;

ii. a mobilidade de alguns colaboradores é uma característica peculiar em um projeto distribuído. É comum gerentes, líderes de projeto e até mesmo desenvolvedores viajarem entre os locais de desenvolvimento (CARMEL,2001). Dessa forma, em meio às viagens, é necessário manter a sincronia entre os colaboradores, de forma que, mesmo estando desconectados da rede por um longo período, os colaboradores possam

tomar conhecimento do que está acontecendo no projeto ao se conectar novamente à rede;

iii. em uma configuração distribuída, é importante que os resultados produzidos ao longo do projeto possam ser verificados pelos colaboradores, mesmo que estes disponham de recursos limitados para o acesso. Dessa forma, várias modalidades de acesso e compartilhamento das informações devem ser fornecidas: telefones – através de tecnologia WAP (Wireless Access Protocol), notebooks – tecnologia Web, entre outros;

iv. mecanismos de notificação são necessários para permitir que os usuários se cadastrem e recebam notificações das ocorrências naqueles artefatos que possuem algum tipo de associação ou impacto na realização das suas tarefas. As taxas de re-trabalho tendem a ser reduzidas, já que quando notificados no momento apropriado, os desenvolvedores podem acompanhar a construção da aplicação tendo posse da situação em que se encontram os artefatos.

A necessidade de atender esses requisitos deve ser tratada, sobretudo, de forma transparente nas soluções. É necessário oferecer mecanismos tecnológicos que possibilitem os colaboradores a estarem sintonizados com os acontecimentos do projeto sem que para isso novas e desconhecidas aplicações precisem ser utilizadas.

Carmel (CARMEL,2001) sinaliza o fato de que a utilização de novas ferramentas no ambiente do projeto tanto pode causar impacto na curva de aprendizagem dos times quanto resistência ao novo. Isto remete a idéia de que a eficiência de uma solução colaborativa está vinculada à utilização de ferramentas que são de conhecimento dos colaboradores.

1.3 Objetivos da dissertação

O objetivo central desta dissertação é especificar e implementar uma ferramenta que suporte o trabalho dos desenvolvedores. Em particular, o foco do trabalho é fornecer a infra-estrutura necessária para melhorar o compartilhamento de informações entre os times, através de notificações, favorecendo assim, a redução das taxas de re-trabalho, conseqüência dos conflitos, na etapa de construção de código.

O sistema que está sendo proposto complementa um sistema de controle de versão, o CVS (Concurrent Versions System), uma das ferramentas de controle de versão mais utilizadas pela comunidade que produz software atualmente (ASKLUND,2002), acoplando a este um mecanismo de notificação síncrono.

Para alcançar os objetivos traçados, algumas metas devem ser atingidas:

i) identificação e análise de soluções colaborativas disponíveis no mercado que atuem como suporte à fase de produção de código e satisfaçam a uma gama de requisitos pré-definidos;
ii) mapeamento de requisitos essenciais necessários em um ambiente colaborativo que suporte, de forma eficaz, o compartilhamento de informações e a mobilidade a que estão submetidos os colaboradores;
iii) implementação dos modelos gerados e da arquitetura proposta;
iv) geração de um cenário de simulação.

1.4 Relevância

De acordo com (CARMEL,2001), apenas naquele ano cerca de 50 nações estavam envolvidas em projetos de desenvolvimento de software para mercados globais. Muitas delas, relataram suas experiências a fim de repassar, como lição, os processos e estratégias de sucesso, bem como os de falha. Neste sentido, aspectos importantes começam a ser esclarecidos. Um deles é que o Desenvolvimento Global se torna cada vez mais uma tendência no mercado de Tecnologia da Informação, não apenas um "modismo".

De relevante importância também é saber dos investimentos que oneram o desenvolvimento. Produzir software para mercados globais requer o uso pesado de ferramentas colaborativas. O que se percebe é que grandes empresas ou desenvolvem suas soluções, ou as adquirem a altos custos, como pode ser observado nos relatos de (BATTIN, 2001) (EBERT,2001) (AOYAMA,1998) (MOCKUS, 2001).

É necessário viabilizar mecanismos tecnológicos que tornem o Desenvolvimento Gobal uma possibilidade de negócio para organizações de menor porte. É importante que elas possam utilizar soluções que possuam custos mais baixos para que se tornem competitivas em mercados internacionais e possam investir seus recursos em outros serviços que venham a agregar valor ao seu produto, como internacionalização e localização.

A solução oferecida neste trabalho poderá ser utilizada por aqueles que necessitam de um grau de cooperação maior entre os parceiros do desenvolvimento, mas que muitas vezes não dispõem de recursos para aquisição da infra-estrutura necessária para tal. A contribuição do trabalho procura oferecer uma solução de baixo custo para seus usuários.

1.5 Organização da dissertação

O Capítulo 2 apresenta ao leitor conceitos preliminares sobre Desenvolvimento Global de Software, ferramentas colaborativas, bem como a sua importância na fase de produção de código. Ainda neste capítulo, o impacto da otimização da comunicação é mostrado como um requisito para atenuar as distorções que podem ocorrer quando vários times cooperam de maneira distribuída em um mesmo projeto.

Para viabilizar a construção de uma ferramenta que auxilie o compartilhamento de informações em escala de Internet, um servidor de notificação de eventos será utilizado como parte da solução. O Capítulo 3 fornece o respaldo necessário para o entendimento desta tecnologia, apresentando características e serviços oferecidos pelo servidor escolhido para compor a solução.

O Capítulo 4 contém a especificação da ferramenta. Com base nos requisitos apresentados no Capítulo 2, a proposta da solução é delineada. Os aspectos arquiteturais são apresentados nesta parte da dissertação.

O Capítulo 5 apresenta os passos seguidos para conceber um mecanismo de notificação síncrono para o CVS utilizando o CASSIUS, o servidor de notificação escolhido. Também apresenta o cenário de simulação da ferramenta.

O último capítulo finaliza a dissertação com as conclusões obtidas após a realização do trabalho, com a avaliação das contribuições oferecidas pela solução e com a indicação de trabalhos futuros.

Capítulo 2

A Produção de Código no Desenvolvimento Global de Software

Este capítulo fornece uma visão conceitual acerca do Desenvolvimento Global de Software. A Seção 2.1 apresenta tais conceitos. Ainda nesta seção são mencionados aspectos praticados em grandes organizações para viabilizar o desenvolvimento, ao passo que demandas reais de organizações de menor porte são delineadas. É apresentado na Seção 2.2 como o gerenciamento da configuração e a integração de código são importantes na fase de produção de código. A seção seguinte ilustra uma configuração de ambiente composta por soluções de código-livre. Por fim, a Seção 2.4 situa aspectos colaborativos desejáveis nas soluções tecnológicas.

2.1 Introdução

Cada vez mais o desenvolvimento de software se torna uma atividade internacional. Apenas em 2002, cerca de 203 empresas distribuídas em mais de 50 países estavam envolvidas com desenvolvimento colaborativo (BENNATAN,2002). As vantagens competitivas de mercados estrangeiros, a redução dos custos operacionais (mão-de-obra barata), a qualidade dos especialistas encontrados dispersos pelo globo, e a possibilidade em se ter equipes trabalhando 24 horas – o que teoricamente pode reduzir o *time-to-market* de um produto – são alguns dos motivos que despertam o interesse de muitas organizações.

Contudo, a sobrevivência de uma organização em um mercado global depende da sua habilidade em tornar-se mais competitiva e veloz em atender as demandas dos consumidores. Como resultado, muitas organizações têm desenvolvido estruturas virtuais para obter maior flexibilidade.

Cientes da ausência de competência para atuar em áreas que não são as suas, as organizações acabam se aliando através de parcerias estratégicas (KAROLAK,1998), *joint-venture* ou terceirizando parte do trabalho a ser desenvolvido (CARMEL,2001) formando, assim, times que trabalham de forma dispersa e, principalmente, que colaboram para a obtenção de um mesmo objetivo.

Enquanto o mercado mundial se torna cada vez mais "próximo", a adaptação aos novos requisitos não é tão natural, dada à heterogeneidade dos usuários finais e à complexidade do desenvolvimento. Muitas são as diferenças culturais, as barreiras de linguagem, os desafios em coordenar processos diversos e múltiplas equipes, as mudanças nos requisitos e os riscos, bem mais desconhecidos se comparados aos riscos presentes em um modelo de desenvolvimento "convencional".

É preciso adequar as mudanças inerentes aos processos de negócios e ao desenvolvimento de produtos suportando um mercado global e único, vertendo, então, para a produção de documentos que esclareçam as organizações - quanto aos benefícios, riscos e investimentos necessários. Vanessa Farias (DANTAS,2003) trata esta questão de forma clara e concisa.

Também importante é o papel do suporte tecnológico, que permite manter a coordenação dos times, o escalonamento das tarefas, o gerenciamento de mudanças, a integração de código, os testes, a escalabilidade dos sistemas e o gerenciamento dos conflitos, em meio a alta complexidade gerada em grande parte pela sepação geográfica dos times.

O desenvolvimento deste suporte conduz às **tecnologias colaborativas**, as quais são responsáveis por compor uma infra-estrutura mais adequada à construção de aplicações a partir da cooperação entre os times de desenvolvimento que compartilham a construção dos artefatos. Através de soluções colaborativas, o desenvolvimento das partes (componentes) pode ser realizado de maneira isolada, no entanto, as mudanças que ocorrem e o resultado das integrações podem ser usufruídos por todos[3].

Para o Desenvolvimento Global de Software (DGS), tais ambientes são fundamentais porque suportam a troca de mensagens entre os envolvidos e também fornecem uma visão completa do ciclo de vida do projeto, viabilizando atividades do

[3] As ferramentas que oferecem tais funcionalidades são denominadas **CSCW**s (do inglês Computer Supported Cooperative Work). Ferramentas que suportam o processo de gestão do projeto (alocação de pessoas, geração de métricas, etc) são denominadas **PSEE**s (do inglês Process-centered Software Engineering Enviroments).

desenvolvimento e permitindo também que os gerentes possam acompanhar e coordenar mais facilmente o projeto.

2.1.1 Panorama mercadológico

Conforme mencionado na Seção 1.4, um número expressivo de organizações tem aderido ao desenvolvimento global. Grandes empresas como Motorola (BATTIN *et al*, 2001), Alcatel (EBERT,2001), Fujitsu (AOYAMA,1998) (GAO,1999) e Bell Labs (MOCKUS, 2001) podem ser citadas como exemplos.

DGS envolve custos altos, principalmente no tocante ao uso de tecnologias colaborativas que sustentam a coordenação dos esforços despendidos durante o projeto. Adicionalmente, outros aspectos de extrema importância a serem considerados se referem à qualidade e ao nível de competitividade do produto.

Apesar da abordagem de desenvolvimento global onerar os custos de um projeto, percebe-se que organizações de menor porte[4] também têm buscado espaço em mercados globais. De acordo com Eratóstenes Ramalho Araújo (ARAUJO,2003), o Brasil, em especial, começa a dar passos significativos em mercados estrangeiros a partir de pequenas empresas que possuem o apoio de agentes importantes como SOFTEX, CGSoft e GENESS (fornecendo capital de risco já que as empresas possuem mão-de-obra qualificada), que trabalham em parceria com universidades federais, a exemplo da de Campina Grande e de Santa Catarina.

Neste cenário, alguns aspectos de extrema importância a serem considerados se referem à qualidade e ao nível de competitividade do produto, visto que tornar a aplicação competitiva em um mercado mundial não é uma tarefa simples. Um dos requisitos básicos é fornecer o produto de software com suporte à língua nativa do mercado recebedor. Por este motivo, o planejamento da versão de um software para um mercado específico deve reconhecer a importância de duas atividades: a internacionalização e a localização de produtos.

Segundo (LUONG, 1995), (GRIBBONS, 1997) e (COLLINS, 2002), internacionalização é o processo de conceber o software como um *framework* culturalmente neutro, que possa ser adaptado para diferentes mercados mais fácil e eficientemente. Para

[4] É válido mencionar que a classificação quanto ao porte considera o nível de conhecimento e a experiência da empresa com projetos distribuídos, além da margem de recursos financeiros alocável com investimentos em torno de tecnologia colaborativa. Tal classificação é utilizada no trabalho apenas para diferenciar empresas que possuem *know-how* em projetos globais daquelas que ainda estão iniciando projetos nessa área.

isso, o código deve ser independente de plataforma e idioma e preparado para suportar diferentes interfaces e funcionalidades. Todos esses aspectos precisam ser considerados nas fases de projeto e codificação do software. O objetivo da internacionalização é dispensar a necessidade de trocar os componentes centrais durante um segundo processo: a localização.

Localização é o processo de análise e adaptação do software para atender às necessidades e preferências de mercados distintos, através da tradução do produto, e, em alguns casos, do acréscimo de características específicas do mercado. A Localização se aplica a todos os aspectos do software ligados à comunicação com o usuário, como interface, mensagens do sistema, manuais e tutoriais (CORONADO, 2001).

A utilização de tecnologias colaborativas é um aspecto decisivo para permitir o gerenciamento dos efeitos causados pela distância que afeta, sobretudo, a produção de código e a sua integração. A Tabela 1 apresenta um demonstrativo simples de ferramentas que tentam preencher as lacunas deixadas pela separação física. Pode-se perceber que muitas delas são ferramentas proprietárias, sua aquisição, apesar de muito importante no ambiente do projeto, pode sobrecarregar os recursos disponíveis, o que causa deficiência nos investimentos com os processos relacionados à qualidade, a exemplo dos serviços de localização.

Tabela 1: Ferramentas que suportam o desenvolvimento distribuído

Atividade	Ferramenta
Comunicação	▪ **TeamPortal** ▪ **Microsoft's NetMeeting**
Acompanhamento de problemas	▪ **Rational's ClearQuest Multisite** ▪ **Pragmatic Software's Software Planner** ▪ **Hitachi's Problem Management System**
Controle de mudanças	▪ **Rational ClearQuest Multisite**
Gerenciamento de configuração	▪ **(CVS) Concurrent Versions System** ▪ **PVCS Version Manager** ▪ **Continuus CM Synergy**

No decorrer deste trabalho, o resultado de algumas investigações realizadas na literatura, *in loco*[5] e de buscas na Internet, nos levou a informações importantes:

1. DGS requer funcionalidades adicionais não suportadas por grande parte das ferramentas disponíveis do mercado, porque tais ferramentas utilizam uma abordagem "centrada na tarefa"[6], não estimulando um canal de comunicação efetivo entre os envolvidos no projeto;
2. algumas ferramentas de código-livre encontradas são capazes de automatizar tarefas referentes à produção de código (maiores detalhes podem ser observados na Seção 2.3 e no Anexo B), mas não foi encontrado nenhum ambiente que contemplasse aspectos relacionados à notificação de ocorrências e mobilidade, por exemplo;
3. ferramentas proprietárias mais robustas, como é o caso do Rational´s ClearCase Multisite, um ambiente que satisfaz aos aspectos de solicitação e acompanhamento de mudanças, são inapropriadas para pequenas empresas devido aos custos de aquisição. Outras, de código-livre, a exemplo do CVS, PVCS Version Manager, Continuus CM Synergy, possuem limitações que afetam os aspectos mencionados no item 2;
4. não há vinculação nas ferramentas com a questão de facilitar a comunicação efetiva entre os times, considerando que estes podem estar submetidos a fusos-horários diferentes, por exemplo.

A análise dessas informações indica que a atual disponibilidade de tecnologias de suporte afeta, particularmente, organizações de pequeno e médio porte e projetos desenvolvidos por instituições de P&D.

A grande barreira a ser enfrentada recai na insuficiência de soluções mais acessíveis que tornem possível um controle maior dos aspectos mais críticos do desenvolvimento, como o gerenciamento dos conflitos, por exemplo. Para estabelecer os problemas encontrados na fase da construção da aplicação, a seção seguinte é apresentada.

[5] Algumas empresas contribuíram para o entendimento das lacunas existentes no enfoque tecnológico da questão do DGS. A Motorola e Mobile, ambas localizadas em Recife, foram consultadas. Devido ao sigilo inerente aos projetos, algumas informações não puderam ser explicitamente ilustradas na dissertação.
[6] Termo utilizado por Ulf Asklund no artigo "Configuration Management for Distributed Development in an Integrated Environment".

2.2 A produção de código

Ainda no início do projeto, é necessário caracterizar o cenário onde o desenvolvimento ocorrerá. A formação de parcerias, a organização dos times e a intenção de atingir diferentes mercados são questões que precisam ser consideradas pelas empresas antes de dar início ao desenvolvimento para que os riscos sejam antecipados e as decisões corretas sejam tomadas (DANTAS, 2003).

A necessidade de um aparato tecnológico é percebida nas fases iniciais do projeto. A complexidade em coordenar pessoas à distância é inerente a qualquer projeto distribuído e as dificuldades surgem, sobretudo na fase em que os detalhes do projeto estão sendo negociados. Maiores detalhes podem ser encontrados no Anexo A.

Depois que as tarefas são atribuídas aos times, os encarregados pela codificação precisam seguir uma rotina de trabalho para assegurar a qualidade do código e reduzir a taxa de re-trabalho. Após a codificação, necessariamente os times devem testar os componentes que foram produzidos e integrá-los continuamente a um repositório de artefatos de acesso global.

Se em projetos convencionais (onde os times estão próximos), a tarefa de lidar com as incertezas, a garantia da qualidade dos artefatos e a gestão dos prazos é difícil, em projetos distribuídos, onde múltiplos times compõem o ambiente de desenvolvimento, a complexidade em lidar com atividades cotidianas, como o controle de versões, a sincronização dos artefatos e a notificação de ocorrências, aumenta consideravelmente devido a separação geográfica e as dificuldades provenientes da fragilidade na comunicação. O nível de colaboração[7] nesta etapa é muito grande e a ausência de tecnologia colaborativa aumenta a complexidade de tais atividades.

2.2.1 A gestão da configuração

O gerenciamento da configuração é definido por (BABICH,1986) como sendo a arte de organizar e controlar as modificações que são realizadas por uma equipe de desenvolvimento. Em particular, o gerenciamento da configuração possui um papel muito importante para as etapas de construção e integração de código: além de controlar as várias versões do sistema, o gerenciamento da configuração se preocupa em manter os

[7] O significado do termo colaboração possui uma dimensão muito mais ampla daquela empregada neste trabalho. Quando utilizado, possui a mesma semântica que cooperar, por exemplo. Ambos os termos são tratados indistintamente.

componentes do produto, em armazenar sua história, em oferecer um ambiente estável ao desenvolvimento e em coordenar as mudanças simultâneas que ocorrem na aplicação.

Um ambiente de gerenciamento de configuração oferece várias funcionalidades (ASKLUND,2002), dentre elas:

i. controle das versões: que possibilita armazenar em um repositório diferentes versões de um artefato para recuperação posterior;

ii. gerenciamento de concorrência: que gerencia o acesso múltiplo de vários usuários ao repositório, seja evitando ou suportando o acesso concorrente;

iii. gerenciamento de espaços de trabalho (*workspaces*): evita que o trabalho de alguns desenvolvedores seja afetado pelas alterações de outros;

iv. documentação das mudanças: mantém históricos de requisições, alterações e conflitos que ocorrem durante a construção dos artefatos.

Claramente se não realizada bem, esta atividade se torna um obstáculo frustrante para o atingir os objetivos do projeto. Segundo Bennatan (BENNATAN,2002), esta é uma das áreas principais onde um projeto distribuído pode fracassar.

Usualmente, o gerenciamento de configuração é tratado a partir da perspectiva do gerente de projeto, onde ele direciona e controla o desenvolvimento de um produto através da identificação dos seus componentes e do controle contínuo das mudanças que ocorrerão durante o seu desenvolvimento (ASKLUND, 2002). Contudo, quando lidamos com Desenvolvimento Global o enfoque a partir da visão do desenvolvedor ganha uma dimensão maior dada à necessidade de os times possuírem mecanismos adequados para sincronizar suas tarefas e os resultados produzidos, para que os prazos e, principalmente, a qualidade da aplicação não sejam comprometidos.

A utilização de um repositório central auxilia o controle de versões. No entanto, observa-se que quando os times estão distribuídos, aspectos adicionais devem ser considerados nas ferramentas. Uma propriedade desejável nos sistemas de gerenciamento de configuração é que estes forneçam informações sobre o que está acontecendo no projeto. Segundo Ulf Asklund (ASKLUND,2002), a ausência de mecanismos eficientes de notificação é uma das grandes lacunas existentes. Tais sistemas devem prover melhores estratégias de notificação, permitindo assim, que muitos conflitos gerados durante a codificação sejam rapidamente detectados e resolvidos, conferindo mais tranqüilidade ao processo de integração de código (geração de *releases*).

Um aspecto importante é que as mudanças sejam propagadas no momento em que ocorrem, sem que para isso os desenvolvedores deixem de se concentrar nas suas atribuições técnicas. Isto está diretamente relacionado à boa qualidade com que os desenvolvedores compartilham as informações.

2.2.2 A integração de código

Em um contexto de desenvolvimento distribuído, a rotina de trabalho dos times sofre sensíveis mudanças. Primeiramente, porque todos os artefatos, em especial os de código, devem ser remetidos a um repositório central que é acessado constantemente. A consistência das versões que estão sendo utilizadas por todos num dado momento é um aspecto de extrema importância para uma etapa posterior à codificação: a integração.

O sucesso desta etapa sofre interferências da robustez da ferramenta utilizada no gerenciamento de configuração, isto porque a estratégia utilizada para o acesso aos componentes influencia diretamente o tempo gasto no desenvolvimento das partes, já que muito tempo pode ser gasto com a resolução de conflitos.

Os conflitos mais comuns acontecem no período de preparação para a integração dos componentes, assim como, no momento posterior a ela. Antes de integrar, os problemas recaem no fato de que mudanças nos requisitos muitas vezes não são notificadas a todos os desenvolvedores.

Embora definindo a arquitetura-base do sistema e as interfaces de comunicação entre os componentes de forma centralizada, como sugere (CARMEL,2001), os desenvolvedores possuem autonomia para criar ou modificar classes, bibliotecas ou componentes que estão sob sua responsabilidade ou dos quais fazem uso.

O impacto das alterações no trabalho de outros desenvolvedores muitas vezes é desconhecido em um primeiro momento. O conhecimento das alterações apenas é alcançado quando é preciso fazer uso do artefato em questão, e as alterações são, então, sinalizadas pelo sistema de controle de versão. No entanto, a eficiência das notificações depende da ferramenta que está sendo utilizada. Algumas soluções, como é o caso do CVS, não são capazes de notificar de forma síncrona as alterações realizadas nos artefatos.

Por este motivo, durante a integração dos componentes e posteriormente nos testes de integração, os problemas surgem justamente porque as "inconsistências" não foram detectadas durante a implementação. Sendo assim, evitar que tais conflitos ocorram e,

principalmente identificá-los, quando estes acontecem, não é uma tarefa simples em se tratando de sistemas complexos.

Este fato se torna ainda mais crítico quando determinadas práticas regem o processo de desenvolvimento. A integração contínua, por exemplo, sugerida pelo processo eXtreme Programming (BECK,2000), é fundamental para permitir a evolução incremental da aplicação e evitar que altos custos sejam necessários para a montagem do sistema (compilação, configuração, empacotamento, testes).

Contudo, sua eficácia depende de uma ferramenta adequada para auxiliar a detecção de conflitos, que em muitas das vezes é realizada "manualmente" pelos desenvolvedores. Isto é, a capacidade de permitir que os membros de um time colaborem é um aspecto a ser considerado nas soluções.

2.3 O ambiente de desenvolvimento

Após a definição da arquitetura e a alocação das funcionalidades junto aos times, um ciclo completo deve ser executado para que as versões parciais possam ser apresentadas ao cliente. A gerência deve atentar para os seguintes aspectos:

- as ferramentas a serem utilizadas para a codificação das atividades e na montagem da aplicação;
- as soluções a serem utilizadas para realizar os testes de unidade e, posteriormente, os de integração;
- a manutenção da integração contínua do código;
- o gerenciamento das versões que estão sendo construídas;
- o favorecimento da cooperação no projeto.

Primeiramente, é preciso compor um ambiente de desenvolvimento que se adapte às demandas de um projeto distribuído. A configuração desse ambiente deve considerar a combinação de algumas ferramentas, dentre elas, aquelas responsáveis pela montagem da aplicação; pela automação dos testes; pelo controle de versões, as quais permitam viabilizar a integração contínua, assim como sugere Fowler (FOWLER,2001).

Percebe-se que um conjunto mínimo de ferramentas precisa ser utilizado para conferir qualidade ao código e sua integridade entre os vários locais de desenvolvimento.

Um cenário relativamente simples pode ser constituído a partir de algumas ferramentas de código-livre (Figura 1).

Dada a ampla aceitação da linguagem Java junto ao mercado, algumas soluções orientadas a Java são especificamente mencionadas no cenário. O Ant é sugerido como solução para montagem da aplicação, o JUnit para a automação dos testes de unidade e o AntHill juntamente com o CVS (Concurrent Version System) como suporte à integração contínua. O Eclipse é sugerido como IDE (Integrated Development Environment) para construção de código. Maiores detalhes sobre tais ferramentas podem ser encontrados no Anexo B.

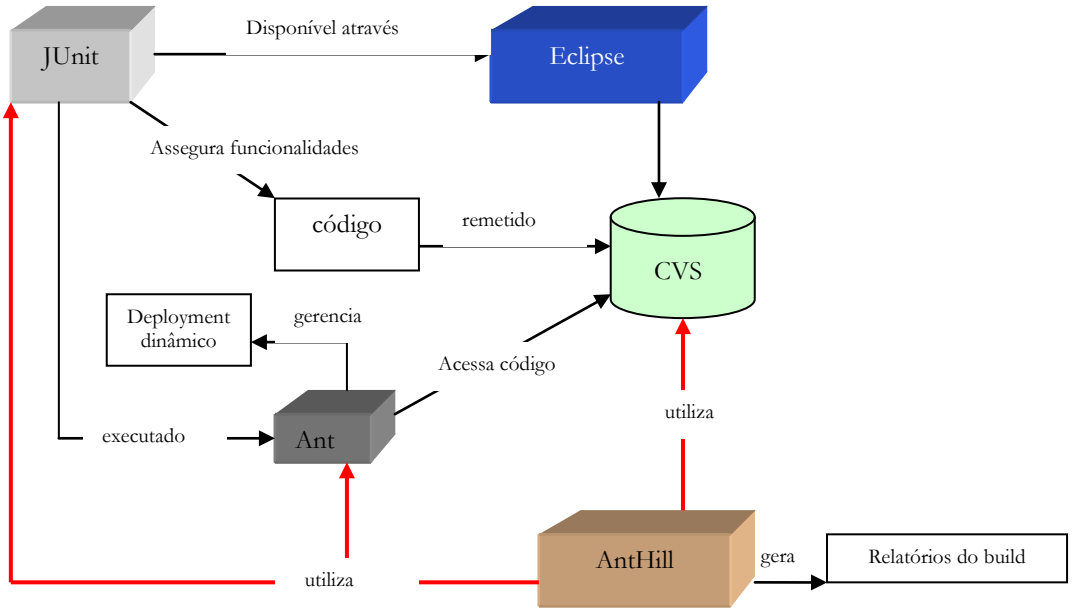


Figura 1: Inter-relacionamento entre as ferramentas JUnit, Eclipse, Ant, CVS e AntHill

Ao configurar a infra-estrutura do ambiente, outra questão deve ser considerada. Além da necessidade de dispor de ferramentas que possam automatizar algumas das tarefas dos desenvolvedores, como é possível mantê-los informados sobre as mudanças que ocorrem nos artefatos quando a utilização de um ambiente colaborativo se torna inviável para o orçamento do projeto?

Na Seção 2.2.1, um aspecto importante é exposto: a deficiência encontrada nos sistemas de gerenciamento da configuração para notificar as mudanças. Uma das principais contribuições da solução é permitir que um mecanismo de notificação possa ser acoplado à ferramenta de controle de versão utilizada no projeto.

Ainda com relação ao cenário apresentado na Figura 1, o mecanismo de notificação fornecido pela ferramenta de controle de versão é ineficaz. O CVS dispõe de um

mecanismo de notificação baseado em e-mails. Tal conduta não confere muita praticidade no que se refere ao compartilhamento de informações. O Ant pode ser utilizado em conjunto com outras ferramentas, a exemplo do JUnit, para automatizar muitas tarefas, embora o seu mecanismo de notificação também funcione de maneira similar ao do CVS. O AntHill segue os anteriores na estratégia de notificação.

Ao fazer uso de um mecanismo eficiente de notificação os benefícios são visíveis. Os custos com comunicação síncrona são reduzidos e há uma tendência para que o impacto das distorções seja menor já que estes são detectados mais cedo no projeto – fato benéfico para a fase de integração de código.

Devido ao fato de que grande parte da complexidade e do desenvolvimento está presente nestas atividades, o escopo de abrangência da solução foi delimitado para atuar nesta etapa do desenvolvimento. A próxima seção discute a importância de embutir aspectos colaborativos nas soluções.

2.4 O impacto da colaboração em cenários distribuídos

Em se tratando de trabalho colaborativo, busca-se muito a cooperação. Especialmente no que se refere ao Desenvolvimento Global, a causa de grande parte dos conflitos que ocorrem durante o desenvolvimento deve-se à falta de uma visão geral do projeto por parte de todos os envolvidos.

Quando o número de times de desenvolvimento (ou de colaboradores) aumenta, a intenção de manter os parceiros sempre atualizados sobre as mudanças e fatos que ocorrem durante o projeto torna-se cada vez mais distante. A Figura 2 ilustra o impacto dos vínculos que são estabelecidos entre os colaboradores quando estes estão fisicamente distantes.

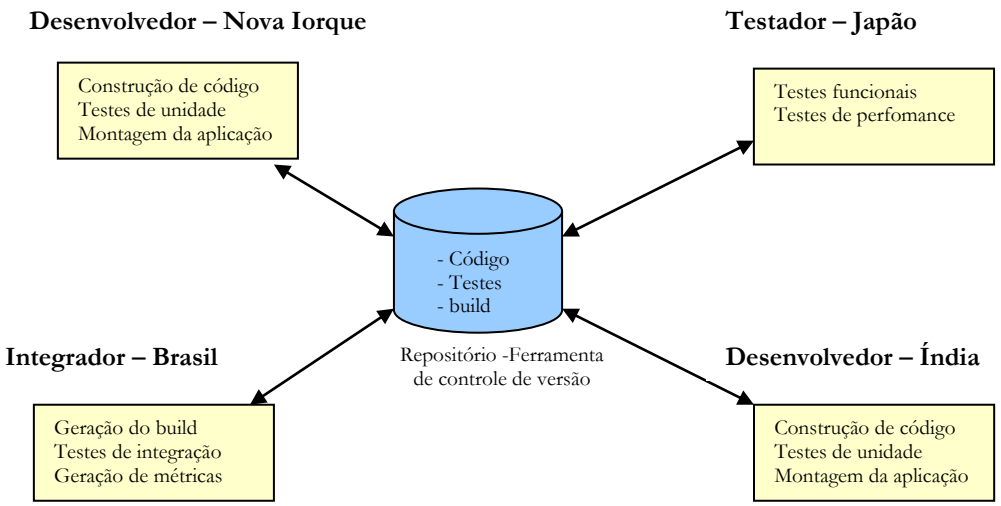


Figura 2: Dependência dos colaboradores em meio à distância física

Observe na figura acima que cada colaborador contribui com o projeto fazendo uso de diversas ferramentas, as quais possuem diferentes propósitos. Todos precisam estar cientes das ocorrências que acontecem nos demais locais de desenvolvimento, caso muito comum aplicado à gerência que precisa monitorar as atividades dos times, mesmo à distância. Uma forma de melhorar a cooperação é permitir que as pessoas sejam notificadas daquelas ocorrências. Para isso, as várias fontes de informação precisam ser monitoradas para que as informações sejam repassadas aos interessados.

Foi mencionado em seções anteriores que parte da responsabilidade das soluções colaborativas é permitir que as tarefas de desenvolvimento dos colaboradores possam ser automatizadas. Entretanto, promover a cooperação de forma transparente no nível requerido por um modelo de desenvolvimento distribuído é a grande questão para o controle, a coordenação e a qualidade do projeto.

O uso de soluções inadequadas pode gerar ruídos na comunicação que atrasam ou complicam a resolução dos problemas. Como as pessoas estão distantes, um simples mal entendimento pode levar dias para ser resolvido. O contexto ilustrado na Figura 2 é importante porque aponta como a utilização de ferramentas pode viabilizar a cooperação.

2.4.1 Incrementando a cooperação nas ferramentas colaborativas

No que diz respeito à qualidade no compartilhamento das informações através de uma ferramenta colaborativa, três aspectos merecem atenção. O primeiro deles é o fato de que muitas fontes de informação precisam ser monitoradas.

Segundo Michael Kantor (KANTOR,2001), o problema-chave das atuais soluções colaborativas se refere ao fato de que não há um equilíbrio entre a quantidade de fontes de informações que podem ser monitoradas e o detalhamento das informações geradas por aquelas fontes – tal desequilíbrio é chamado por Kantor de ***detail-variety trade-off***. Isto significa que, enquanto algumas ferramentas conseguem acessar poucas fontes de informação e fornecem ao usuário muitos detalhes sobre as ocorrências geradas pelas fontes, outras, por sua vez, podem monitorar muitos provedores de informação, mas não conseguem prover ao usuário informações relevantes sobre as mudanças que ocorrem nos objetos que estão sendo monitorados.

Em um projeto distribuído, a qualidade do produto e do processo depende das informações geradas pelas diversas fontes de informação (pessoas e ferramentas) que compõem o ambiente de desenvolvimento. A realização das atividades atribuídas aos vários parceiros do desenvolvimento muitas vezes gera interdependências que apenas podem ser coordenadas, eficientemente, se todas as fontes de informação puderem ser monitoradas e se as ocorrências de cada atividade do projeto puderem ser representadas por qualquer provedor de informação e entendidas pelos interessados.

Um segundo aspecto é o fato de que ter acesso a diferentes fontes de informação pode ser insuficiente se o usuário não tem conhecimento de quais provedores de informação estão disponíveis em um dado momento. Em um ambiente colaborativo eficiente, o usuário deve ser informado (através de mecanismos automatizados que substituam a tarefa dos colegas) de quais fontes de informação estão disponíveis em um dado momento, que objetos (artefatos de código, documentos de texto ou imagem) cada fonte monitora e que tipos de mudanças podem ser notificadas.

Quando o Desenvolvimento Global se configura, muitos especialistas de diferentes países estão envolvidos no projeto. O monitoramento das atividades dos parceiros, sobretudo quando há dependências entre as tarefas, se torna de alta complexidade dado a um número razoável de colaboradores, utilizando ferramentas distintas, com idiomas e fusos-horários diferentes, dever notificar as ocorrências críticas durante a realização das suas atividades.

O último aspecto a ser considerado nas ferramentas é o suporte na escolha da forma com que as notificações são enviadas e representadas para o usuário, assim como, a forma com que o usuário é avisado da chegada de uma nova notificação (através de alertas visuais ou sonoros). De acordo com (KANTOR,2002), um problema comum nas

tecnologias colaborativas é que elas tendem a fixar o estilo de notificação fornecendo poucas alternativas.

As pessoas devem ser capazes de manifestar o seu interesse não apenas no que diz respeito às informações, mas também na forma com que serão notificadas da ocorrência delas. Além do mais, os colaboradores não devem estar limitados a escolher um estilo por vez, dado que a forma com que eles desejam monitorar o ambiente varia de acordo com a sua função dentro do projeto.

É interessante saber o impacto que a ocorrência de uma informação tem para um usuário e se é útil que a ferramenta colaborativa possa ser utilizada por alguém que não necessariamente, possua recursos computacionais e de rede apropriados a todo o momento. De um modo geral, é por meio das ferramentas que o nível de colaboração desejado entre pessoas que estão geograficamente distantes pode ser alcançado. Elas devem ser capazes de obter informação de múltiplas fontes de informação e integrá-las, para fornecer aos usuários um entendimento geral do que acontece dentro do ambiente de desenvolvimento.

Este requisito é importante tanto do ponto de vista gerencial, porque aumenta a capacidade de controle sobre o processo, quanto sob o enfoque de construção de código praticado por analistas, projetistas, desenvolvedores e testadores, já que o esforço com o re-trabalho pode ser minimizado e a taxa de desentendimentos também pode ser reduzida.

2.4.2 Propriedades desejáveis em um ambiente colaborativo

Ao configurar os esforços do desenvolvimento em um projeto virtual, os times podem possuir autonomia para decidirem que ferramentas serão utilizadas durante o desenvolvimento do produto de software. Sendo assim, cada time assume tecnologias distintas, seja para a construção de código, para a realização dos testes, assim como para a integração das versões que estão sendo produzidas ao longo do projeto.

Se por um lado, esta configuração permite que a curva de aprendizagem dos times não afete os prazos do projeto, por outro gera uma lacuna na comunicação formal que deve existir entre os parceiros do desenvolvimento. Além disso, as diferenças de idiomas enfraquecem o poder da comunicação informal que é comprometida pela utilização de mecanismos de comunicação assíncrona (CARMEL,2001).

Na seção anterior, foi mencionado que a grande questão envolvida na tarefa de otimizar a cooperação entre os times de desenvolvimento recai na disponibilidade de

ferramentas que consigam atender aos requisitos de uma solução colaborativa robusta. Objetivando aumentar a qualidade no compartilhamento das informações, é possível estabelecer necessidades que, quando atendidas, são capazes de conferir tais características a uma solução colaborativa:

i. o ambiente deve permitir que as ferramentas de desenvolvimento possam participar funcionando como provedores de informação. As ferramentas devem ser monitoradas de forma que determinados eventos ou ocorrências possam ser mapeados em notificações aos interessados numa dada informação;
ii. pouca ou nenhuma modificação deve ser implantada nas ferramentas;
iii. as ferramentas podem estar sendo utilizadas local ou remotamente;
iv. as notificações devem acontecer em escala de Internet e devem ser transparentes aqueles que geram os eventos;
v. o usuário do ambiente colaborativo deve ter acesso às informações referentes a:
    a. quais fontes de informação estão disponíveis no momento;
    b. que objetos podem ser monitorados pelas fontes;
    c. que tipos de eventos podem ser detectados;
vi. o usuário deve manifestar o interesse em receber informação de determinadas ocorrências como forma de selecionar apenas aquelas notificações que podem afetar o seu trabalho;
vii. os usuários podem escolher o dispositivo pelo qual desejam receber as notificações;
viii. as notificações devem ser enviadas aos interessados e ficar armazenadas para consultas posteriores.

2.5 Considerações Finais

Com a construção de uma ferramenta colaborativa a partir das necessidades mencionadas, os benefícios obtidos em uma primeira análise recaem no fato de que a informação gerada por uma fonte é transformada em conhecimento para os seus consumidores. Tal conhecimento pode ajudar na coordenação do grupo, na simplificação da comunicação verbal e na gestão do fluxo das informações entre os membros da equipe.

Muitos ganhos podem ser observados quando a tarefa de notificar as ocorrências passa a ser indiretamente atribuída aos colaboradores. O primeiro impacto ocorre na

melhoria da comunicação, dado que um mecanismo mais adequado para viabilizar a linguagem informal é oferecido. Conseqüentemente, os custos e as conseqüências de determinados mecanismos de comunicação síncrona são reduzidos.

Há uma tendência em reduzir os problemas referentes à integração de código já que, continuamente, os desenvolvedores são notificados das alterações, e assim podem antecipar conflitos e resolvê-los mais facilmente. Outros benefícios também são percebidos para o controle do projeto já que é possível o gerente acompanhar a evolução do projeto à distância.

Como se pode observar, muitas ferramentas podem configurar o ambiente de desenvolvimento. A questão é esta: como criar soluções que possam capacitar a colaboração entre as pessoas sem que, para isso, novas ferramentas sejam adquiridas ou que a curva de aprendizagem do grupo, durante a capacitação do uso da ferramenta, seja afetada? Melhorar a qualidade no mecanismo de compartilhamento de informação é uma solução.

Por este motivo, a tecnologia de servidores de notificação de eventos é apresentada no capítulo a seguir como uma solução eficaz para prover o encaminhamento de informações.

Capítulo 3

Servidores de Notificação de Eventos

Em um projeto distribuído, problemas com a coordenação dos times emergem. Este capítulo aborda características de servidores de notificação de eventos, uma tecnologia útil ao suporte do Desenvolvimento Global, na medida em que facilita a integração e o desenvolvimento de ferramentas colaborativas através da utilização de mecanismos de notificação. Em geral, as vantagens, bem como, os problemas que a tecnologia poderá resolver são ilustrados.

3.1 Introdução - Servidores de notificação de eventos

Em capítulos anteriores vimos que muitos projetos estão distribuídos em diversos centros de desenvolvimento ao redor do globo. A distância entre esses centros gera muitas dificuldades que não existem em projetos centralizados. Neste contexto, muitos problemas técnicos precisam ser resolvidos, entre os quais, permitir que as aplicações que estão distribuídas em diferentes pontos possam se comunicar entre si.

Desde que a Internet não é muito confiável, uma boa idéia de imaginar a comunicação entre as aplicações é tratando-as como objetos autônomos fracamente acoplados (SOUZA,2002a). Para isso, muitas dessas soluções têm utilizado serviços de notificação de eventos (*publish-subscribe*). Em tal abordagem, as mensagens não são enviadas diretamente de uma aplicação para a outra, e sim para um despachante, também conhecido como um Servidor de Notificação de Eventos (SNEs).

Dessa forma, um servidor de notificação distribui os eventos para as aplicações que estão interessadas e nele se subscrevem. Neste estilo de comunicação há dois tipos de componentes: os provedores e os consumidores de informação. Os primeiros são responsáveis por publicar os eventos, enquanto os outros estão interessados em serem notificados de determinadas ocorrências.

Os eventos são enviados pelos provedores[8] ao servidor que garante a entrega desses eventos a todos os interessados. Em geral, nesses sistemas, a interação entre os componentes é modelada como eventos que são transmitidos na forma de notificações.

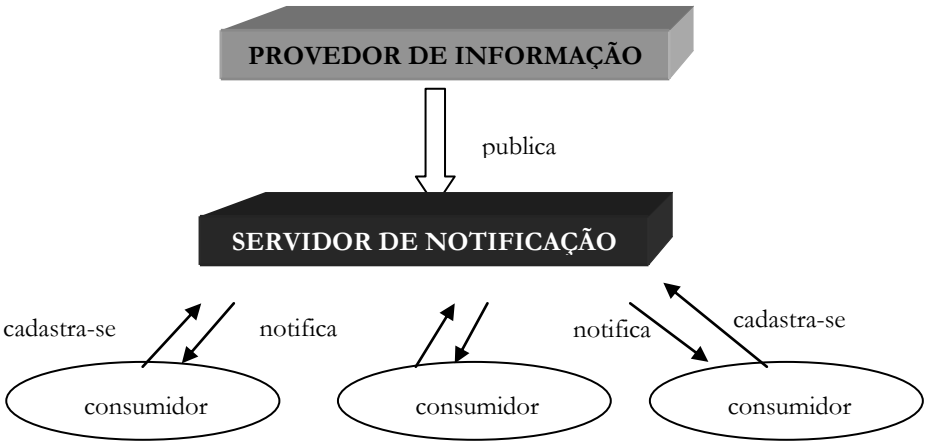


Figura 3: Arquitetura genérica de um servidor de notificação

Usualmente, um servidor fornece um serviço de subscrição que permite que os consumidores se cadastrem em diferentes tipos de eventos, utilizando por exemplo, conectores lógicos, expressões regulares, detecção de seqüência, entre outros (SOUZA,2002a). Neste trabalho, três exemplos representativos de servidores foram estudados: o CASSIUS (KANTOR,2002), o Elvin (FITZPATRICK,1999) e o Siena (CARZANINGA,2001).

No contexto da solução que está sendo tratada, o interesse é o de promover a comunicação entre pessoas que necessitam disseminar informações importantes no projeto. O servidor CASSIUS foi desenvolvido para prover a interação entre grupos de pessoas. Os demais tratam com propósitos mais gerais. A Seção 3.3 apresenta mais detalhes sobre estes servidores[9].

[8] Neste trabalho, os termos provedores, fontes ou origem das informações podem ser tratados indistintamente.

[9] Durante o detalhamento realizado acerca da tecnologia de SNEs, alguns termos em inglês são mantidos para não distanciar a semântica dos mesmos com relação às traduções para o português.

3.2 Servidores de Notificação de Eventos e DGS

Servidores de notificação de eventos (SNEs) podem fornecer muitas vantagens ao desenvolvimento distribuído de software porque conseguem viabilizar a cooperação entre aqueles que produzem as informações e aqueles interessados nela. A cooperação é então, alcançada por meio de ferramentas colaborativas.

O mais importante dos benefícios é o desacoplamento entre as fontes e os consumidores de informação. Quando o acoplamento é reduzido, a flexibilidade do ambiente é melhorada. Uma fonte de informação publica eventos sem estar interessada em saber que componentes estão interessados neles. Do outro lado, os consumidores se cadastram aos eventos para serem notificados sem que haja amarração com os provedores da informação. Assim, se um novo provedor é adicionado ao sistema, nenhuma modificação é necessária. Se um novo consumidor deseja ser notificado sobre um evento, ele simplesmente realiza o cadastro. Nem as fontes nem os consumidores precisam ser alterados. Uma importante vantagem é que as notificações são possíveis em escala de Internet. Embora esta característica não seja implementada em todos os SNEs.

SNEs também podem ser utilizados para a integração de ferramentas de desenvolvimento. Este aspecto é importantíssimo para construir ambientes colaborativos utilizando soluções já existentes. A integração baseada em eventos é significante porque em DGS, cada time de desenvolvimento usa diferentes ferramentas o que requer um mecanismo que integre os artefatos e assim, o suporte à cooperação. Como mencionado no capitulo 2, Seção 2.1.1, a grande dificuldade encontrada por empresas de pequeno porte é a inexistência de ambientes cuja aquisição seja mais acessível e que integrem as várias ferramentas utilizadas nas etapas do desenvolvimento.

Utilizando um servidor, a lacuna existente na comunicação informal pode ser parcialmente resolvida através da construção de ferramentas de comunicação, como sistemas de mensagens instantâneas. Em geral, outras aplicações populares, como sistemas de workflow (COOK,2000), também podem ser construídas a partir dessa tecnologia.

A comunicação entre fontes, servidor e consumidores pode acontecer através dos modelos *push* e *pull* (maiores detalhes são tratados na Seção 3.3.1). No caso de desenvolvimento global, ambas modalidades de comunicação são interessantes: o modelo *push* é adequado para situações em que um gerente precisa ser notificado de uma ocorrência crítica para o projeto assim que ela ocorre. Por outro lado, um modelo *pull* é importante em ambientes onde os consumidores não possuem tanta urgência e fontes de informação

podem ser desconectadas da rede. Neste caso, as notificações poderiam ser perdidas caso não fossem armazenadas para recuperação posterior.

3.2.1 Como as notificações podem ser úteis em um cenário de desenvolvimento distribuído

Neste cenário, três desenvolvedores que estão distribuídos pelo globo trabalham em diferentes partes de um sistema, que são dependentes. O desenvolvedor 1 está construindo um componente chamado "Note.java" em Los Angeles, EUA. Este componente depende de outro componente "Main.java" que está sendo construído por outro desenvolvedor, o desenvolvedor 2, localizado na Índia. A dependência entre esses componentes implica dizer que o desenvolvedor 1 precisa ser notificado quando o componente "Main.java" é modificado. Caso contrário, mudanças em Main.java podem prejudicar o seu código.

Da mesma forma que o desenvolvedor 1 depende dos artefatos do desenvolvedor 2, este também depende de outro componente, o "Connection.java", que está sendo manipulado pelo desenvolvedor 3 em Nova Iorque, EUA. Neste caso, devido à diferença menor de fuso-horário, o desenvolvedor 1 pode estar trabalhando no seu componente ao mesmo tempo em que o desenvolvedor 3 está trabalhando no dele. Assim, o desenvolvedor 1 precisa ser notificado das mudanças assim que elas ocorrem para que inconsistências (conflitos) sejam evitadas.

3.2.2 Aspectos importantes no cenário

Observe que na interação entre os desenvolvedores 1 e 2, a diferença de fuso-horário é grande. Praticamente, os dois trabalham em horários complementares. O que implica dizer que o diálogo entre eles é afetado e, conseqüentemente, a cooperação também já que, se não bem gerenciada, a lacuna existente na comunicação entre os dois pode retardar o trabalho devido aos conflitos que podem acontecer.

Uma alternativa para superar o efeito da diferença de fuso é permitir que os desenvolvedores, ao começarem suas tarefas, possam ser avisados das alterações que foram implementadas pelo seu parceiro de projeto.

No cenário acima, os desenvolvedores precisam utilizar uma ferramenta de controle de versão para gerenciar o código que está sendo desenvolvido, inclusive para permitir a consistência das versões que estão sendo utilizadas em um dado momento.

Contudo, se a ferramenta utilizada não possui mecanismos de notificação, a deficiência na cooperação entre os dois pode acarretar efeitos nocivos ao projeto. Entre os desenvolvedores 1 e 2, algum tipo de infra-estrutura de comunicação precisa ser utilizado. No entanto, a organização pode não possuir recursos para arcar com despesas de telefonemas internacionais, por exemplo. O uso de e-mail muitas vezes não é a alternativa mais adequada para viabilizar a comunicação, isto porque o desenvolvedor pode negligenciar as notificações. Como mencionado na Seção 2.2.2, cabe ao sistema dee gerenciamento de configuração a tarefa de notificar as ocorrências de forma que este passo seja transparente para aqueles que geram as mudanças.

O caso entre o desenvolvedor 1 e o desenvolvedor 3 é mais crítico. Se nenhum mecanismo de notificação é oferecido, o projeto sofrerá as conseqüências das distorções. Como os dois só ficarão sabendo dos conflitos quando acessarem o repositório, a detecção destes poderá ocorrer tardiamente. Se ambos demorarem a cometer suas alterações, provavelmente, a taxa de re-trabalho tenderá a ser maior.

Um servidor de notificação pode dar suporte à ferramenta de gestão da configuração utilizada no projeto para viabilizar tais notificações. Assim, sempre que um artefato, no caso do cenário, código, for adicionado ou atualizado no repositório de artefatos, os colaboradores serão avisados. É válido observar que o mecanismo de comunicação implementado pelo servidor (*pull* e/ou *push*) é quem define a robustez do sistema, já que diante de outros aspectos, como mobilidade, o impacto das notificações no projeto pode ser maior.

3.3 Detalhamento dos servidores CASSIUS, Siena e Elvis

A solução que está sendo proposta parte do princípio que é necessário viabilizar a comunicação entre fontes diversas a partir de um número arbitrário de ferramentas. Tanto as fontes quanto as ferramentas encontram-se distribuídas pela Internet. A meta é, então, suportar a cooperação humana, e não apenas a interoperabilidade de aplicações. Como tal, o servidor de notificação precisa atender a requisitos bastante específicos que estão mencionados no capítulo anterior, Seção 2.4.3.

Portanto, os SNEs disponíveis precisam ser entendidos. Nesta seção, um detalhamento maior é apresentado. Três servidores constituíram o objeto da investigação: o CASSIUS, um dos SNEs estudado, foi escolhido por atender a propósitos bem específicos do aspecto de cooperação humana; o Elvin, por sua vez, foi escolhido devido à sua maturidade e aceitação pela comunidade de engenharia de software; o Siena, o terceiro SNE escolhido, é relativamente recente, mas está ganhando popularidade por enfatizar a escalabilidade em ambientes distribuídos.

Parte das informações descritas nesta seção foi obtida a partir do trabalho de Cleidson R. B. de Souza[10], no artigo "Using Event Notification Servers to Support Application Awareness" (SOUZA,2002b). Através de um cenário de simulação aplicado no ambiente do DARPA (Defense Advanced Research Projects Agency), foi possível concluir que servidores e que serviços por eles fornecidos suportam o nível de cooperação desejado.

As subseções a seguir abordam características operacionais e funcionais daqueles servidores. Para realizar a análise, basicamente, quatro aspectos conduziram a investigação: i. como acontece a comunicação entre o servidor e os consumidores, ii. que estratégia o servidor utiliza para identificar os possíveis interessados nos eventos que são gerados, iii. como o servidor trata a definição com relação ao tipo dos eventos e iv. como acontece a troca de um servidor para outro. Após a apresentação de cada um desses aspectos, considerações práticas acerca do comportamento de cada servidor são apresentadas.

3.3.1 Aspectos sobre a comunicação entre o servidor e os consumidores

Considerações teóricas: no geral, a comunicação em um sistema baseado em eventos ocorre basicamente das fontes de informação para o servidor e do servidor para os consumidores. Há dois tipos de arquitetura para implementar essa comunicação: *pull* e *push*. Na arquitetura *pull*, o consumidor interessado nos eventos é responsável por entrar em contato com o outro componente para checar novos eventos. Já na arquitetura *push*, o provedor de eventos sinaliza ao consumidor as ocorrências, invocando algum dos métodos definidos na sua interface.

Considerações práticas: Elvin, Siena e CASSIUS implementam uma arquitetura *push* com respeito aos provedores de informação. O CASSIUS também implementa uma arquitetura

[10] Cleidson R. B. de Souza é estudante de doutorado pelo departamento de Informação e Ciência da Computação na Universidade da Califórnia, em Irvine, EUA. Atua na área de desenvolvimento de tecnologias CSCW(Computer Supported Cooperative Work).

pull. Neste último caso, se a comunicação entre o servidor e as ferramentas colaborativas utiliza-se do protocolo http para estabilizar a conexão, então a comunicação no CASSIUS é realizada utilizando a arquitetura *pull*. Ainda, se as ferramentas utilizam a API do CASSIUS para abrir conexão entre o servidor e a ferramenta, os eventos são enviados assim que eles chegam ao servidor.

3.3.2 Aspectos sobre o tipo de *matching* que é suportado pelos servidores

Considerações teóricas: uma característica crítica dos servidores de notificação é o mecanismo que modela a relação entre os eventos que chegam e o interesse dos consumidores, chamada de *matching*. Há várias formas para prover isto. Utilizando-se conectores lógicos, detecção de seqüência ou expressões regulares (HILBERT,2000). Um outro aspecto importante é onde o *matching* é realizado (nas fontes, nos consumidores ou no servidor). Na verdade, a escolha depende dos recursos disponíveis no ambiente computacional. Se realizada nos consumidores, significa que estes receberão muitos eventos. Pode acontecer do consumidor não estar interessados na maioria deles. Se acontecer no servidor, o tráfico da rede é reduzido, mas o servidor pode enfrentar problemas de escalabilidade, dependendo do número de eventos que chegam, dos consumidores e provedores. Finalmente, se os provedores processam a combinação, alguns eventos podem não ser introduzidos na rede.

Considerações práticas: A linguagem de subscrição fornecida pelo Elvis suporta conectores lógicos e uma forma simplificada de expressões regulares, não suportando qualquer tipo de detecção de seqüência. O Siena suporta conectores lógicos, expressões regulares e uma forma simplificada de detecção de seqüência. O CASSIUS suporta os três tipos de linguagem.

3.3.3 Aspectos relacionados ao tipo dos eventos

Considerações teóricas: A maioria dos servidores implementa o protocolo *publish-subscribe* para fornecer os serviços básicos de notificação de eventos (notificação e subscrição). No entanto, alguns servidores podem oferecer serviços adicionais. O CASSIUS, por exemplo, fornece suporte para a definição de tipos de eventos bem como o cadastramento dos provedores de informação. O propósito de ter um passo de cadastro possui duas intenções:

primeiro, ele fornece um nível maior de segurança já que o servidor pode validar as fontes de informação que podem enviar eventos a ele. Em segundo lugar, as informações coletadas durante o cadastramento das fontes podem ser usadas para criar uma estrutura hierárquica que permite a navegação e a seleção dos objetos de interesse por parte dos consumidores. Por sua vez, a definição de tipos é interessante para informar os consumidores, em um formato que facilite o seu entendimento, sobre o que aconteceu de fato com um artefato.

Considerações Práticas: CASSIUS adota uma estratégia em que as fontes de informação precisam se cadastrar junto ao servidor antes de mandar os eventos. Também permite que haja a definição de tipos e a associação destes aos eventos, mas esta não é uma atividade necessariamente obrigatória porque o servidor fornece suporte a tipos de eventos genéricos. Em contrapartida, tanto o Siena quanto o Elvis não requer o passo de cadastramento. Eles permitem que arbitrariamente as aplicações enviem eventos para eles. Além disso, os eventos devem obedecer a um formato pré-especificado o que causa danos à flexibilidade da aplicação, já que os desenvolvedores das ferramentas precisam obedecer a um formato padrão de eventos antes de construir as aplicações (neste caso, os consumidores).

3.3.4 Aspectos relacionados à permuta entre servidores

Considerações teóricas: como discutido anteriormente, quando a arquitetura *push* é utilizada os consumidores são invocados pelo servidor de notificação. Assim, os consumidores precisam seguir uma interface padrão.

Considerações práticas: cada servidor define uma única interface com um conjunto diferente de métodos. Por exemplo, a interface do CASSIUS fornece dois diferentes métodos: um que é chamado quando um conjunto de eventos é recebido e outro quando nenhum evento é recebido. A interface do Elvis fornece apenas um método para ser chamado - *public void notificationAction (Notification event)*. Este método será chamado diversas vezes, uma para cada notificação. Já a interface do Siena define dois diferentes métodos, um que é chamado para receber um evento e o outro para receber um conjunto de eventos. Caso seja necessária a mudança de um servidor para outro, os consumidores precisam implementar todas as diferentes interfaces para cada servidor.

3.4 CASSIUS, o servidor de notificação escolhido

Tomando como base os requisitos apresentados no Capítulo 2, Seção 2.4.3, o CASSIUS foi escolhido para fazer parte da especificação e implementação da solução proposta.

O CASSIUS (CASS Information Update Server) é a implementação da abordagem CASS (Cross Application Subscription Service) que possui o propósito maior de melhorar a forma com que as pessoas colaboram entre si no seu ambiente de trabalho, diante da possibilidade de estas estarem utilizando as mais variadas ferramentas de desenvolvimento (KANTOR,2001).

O interesse da abordagem é permitir a integração das pessoas por meio das ferramentas das quais fazem uso. Na verdade, o foco não é fornecer a comunicação entre aplicações, elas são apenas o meio para integrar os membros do projeto. Neste sentido, qualquer ferramenta pode se tornar um provedor de informação, contanto que consiga enviar eventos ao servidor que passa a ser o centro do ambiente colaborativo. O servidor é responsável por distribuir uma ampla quantidade de informações geradas, a partir das fontes diversas, para as pessoas que são afetadas pela informação.

Para implementar a estratégia três passos precisam ser completados:

i. os desenvolvedores precisam construir aplicações capazes de monitorar o estado das informações e enviar as mudanças para o servidor;
ii. o ambiente de desenvolvimento do projeto precisa fornecer um servidor de notificação capaz de informar aos consumidores que tipos de eventos eles podem se subscrever e que objetos são passíveis de monitoramento;
iii. os colaboradores precisam dispor de ferramentas para receber as notificações.

Como mencionado anteriormente, o CASSIUS é a implementação da estratégia CASS. A arquitetura do servidor pode ser vista na Figura 4 a seguir. O fluxo de informação entre as fontes, o servidor e os consumidores acontece através do protocolo http.

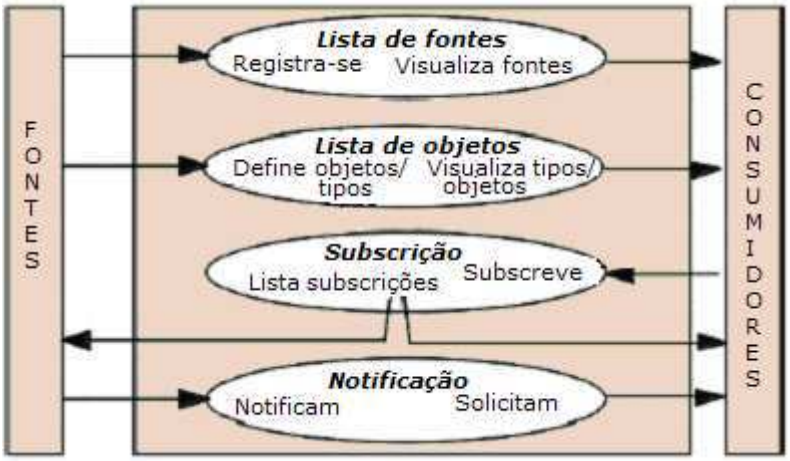


Figura 4: Arquitetura em alto nível do CASSIUS

As fontes precisam se registrar junto ao servidor, fornecendo nomes, senhas e descrições. Os usuários podem acompanhar desta maneira que fontes de informação existem, facilitando a escolha daquelas fontes relevantes que desejam subscrever. As fontes também podem ser categorizadas de acordo com o seu tipo (relatório de testes, código, documentos de projeto, etc). Este mecanismo permite que os consumidores escolham apenas os tipos que afetam seu trabalho.

As fontes de informação mantêm a hierarquia dos objetos que eles podem monitorar e o tipo de eventos que podem afetar aqueles objetos. A definição do tipo fica a critério do desenvolvedor, que pode desejar criar tipos específicos ou pode utilizar os tipos genéricos oferecidos pelo CASSIUS.

O serviço de subscrição implementado pelo servidor permite que os consumidores visualizem os eventos nos quais já estão cadastrados, bem como permite a subscrição para novos eventos e a exclusão de já existentes.

Os principais campos de uma notificação de acordo com o CASSIUS estão ilustrados na Tabela 2. De acordo com o formato da notificação, é possível aos consumidores se cadastrarem não apenas indicando os objetos e ações sofridas por eles, mas também nos eventos gerados por determinados colaboradores.

Tabela 2: Formato de uma notificação no CASSIUS

Summary	Resumo textual da alteração
GenericEvent	Nome de um evento escolhido de uma lista de eventos genéricos
Event	Fontes definem tipos de eventos e definem uma lista de possíveis eventos para cada tipo
Person/Place	(Opcional) pessoa ou local associado ao evento
ObjectID	Identifica o objeto que sofreu alteração
AccountPath	Identifica a que categoria a fonte de informação pertence

3.4.1 O *toolkit* CASSandra

O CASSIUS fornece um *toolkit* chamado CASSandra que facilita muitos aspectos do desenvolvimento, dentre eles:

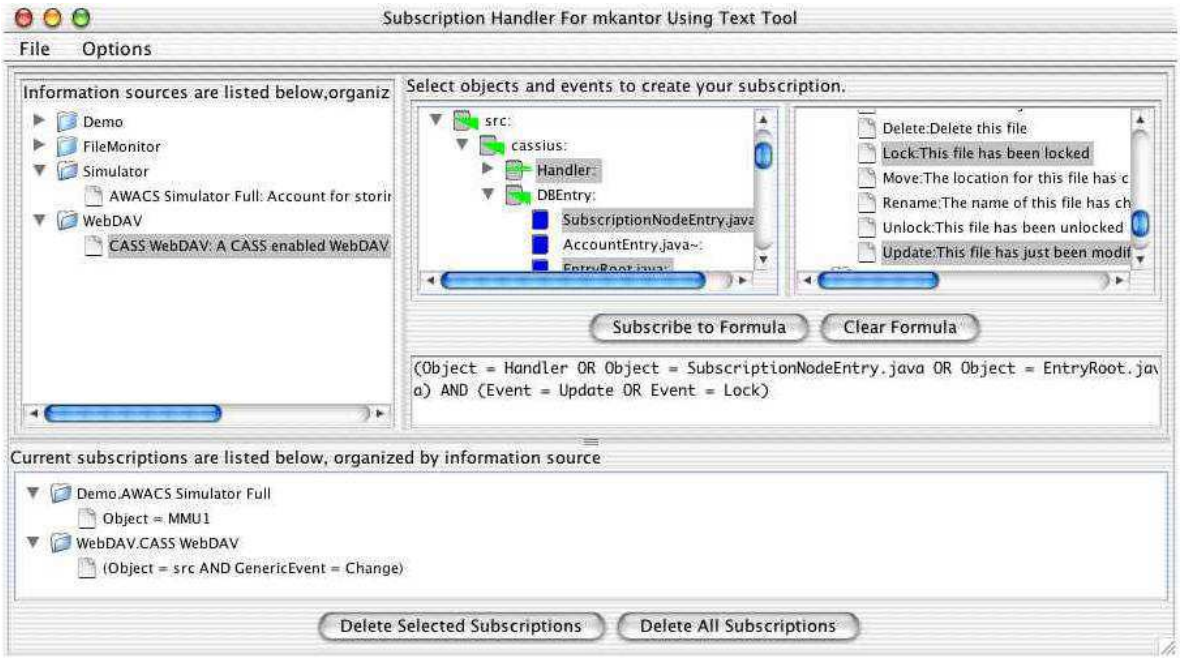


Figura 5: Tela do CASSandra Subscription Editor

É possível não utilizar o CASSandra durante a implementação das ferramentas colaborativas. Isto se aplica quando há a necessidade de construir aplicações em outras linguagens já que o *toolkit* suporta apenas Java. Para tanto, as seguintes tarefas devem ser realizadas:

1. enviar requisições http com o intuito de saber do CASSIUS se há novas notificações;
2. implementar uma ferramenta colaborativa que forneça um estilo de visualização adequado para os usuários;
3. enviar as subscrições dos colaboradores ao CASSIUS.

3.5 Considerações Finais

A estratégia CASS soluciona o problema definido por Kantor como ***"detail-variety trade-off"***, mencionado no capítulo anterior. A falta de um ambiente colaborativo pode ser suprida desde que seja possível viabilizar a troca de informações entre os colaboradores mesmo que estes façam uso de ferramentas heterogêneas.

Como a proposta deste trabalho é justamente oferecer mecanismos de comunicação mais eficientes para a fase de construção de código, em especial solucionando parte dos problemas encontrados durante a gestão de artefatos, o CASSIUS se adequa bem no que

diz respeito à resolução dos problemas em torno do mecanismo de notificação que desejamos utilizar.

Capítulo 4

CVSyn: Projeto e Prototipação da Ferramenta

Este capítulo apresenta a análise e o projeto de uma ferramenta que visa prover suporte à construção de código, fornecendo aos desenvolvedores um mecanismo de notificação síncrono e transparente[11]. O capítulo está organizado da seguinte maneira: a Seção 4.1 ilustra com detalhes os objetivos práticos da solução. A Seção 4.2 situa o leitor com relação aos termos utilizados durante a especificação. Na seção seguinte, os requisitos funcionais e não-funcionais são considerados. O projeto arquitetural da ferramenta é apresentado na Seção 4.4. Na Seção 4.5 a lógica da solução é apresentada. A Seção 4.6 contém o projeto de detalhado da ferramenta e por fim, algumas considerações são mencionadas na Seção 4.7.

4.1 Considerações iniciais: a descrição dos objetivos

Em capítulos anteriores tomamos conhecimento da problemática que afeta o processo de construção de software quando há múltiplos times de desenvolvimento envolvidos no projeto. Embora utilizando ferramentas para automação de tarefas no processo de construção da aplicação, os colaboradores precisam ter meios para compartilhar o resultado de seus trabalhos com os demais colaboradores do projeto que são afetados com as mudanças que ocorrem nos artefatos.

A utilização de soluções que apenas automatizam tarefas mas que não consideram a necessidade de cooperar e compartilhar informações dos times não é suficiente para permitir a qualidade do processo e do produto final. Outra variável ganha relevância neste processo: a qualidade da comunicação. Este é um fator decisivo para o sucesso dos prazos e dos objetivos do projeto visto que quando lidamos com desenvolvimento distribuído as

[11] Transparente porque o desenvolvedor não precisa se preocupar com as notificações.

diferenças culturais, técnicas e tecnológicas podem contribuir para que muitas distorções sejam geradas.

Fundamentalmente, o efeito dos ruídos pode ser reduzido com a utilização de tecnologia colaborativa. A colaboração entre times, ao nível da realização de tarefas de programação, é algo que não se concretiza com a ausência de soluções tecnológicas colaborativas porque elas permitem que as mudanças nos artefatos possam ser propagadas.

Entretanto, em geral, soluções desta natureza oneram os custos do projeto. Uma estratégia bastante utilizada é a utilização de ferramentas de controle de versão que oferecem um repositório central como ponto de apoio aos vários colaboradores que dividem a tarefa de construir partes dependentes da aplicação. No entanto, mesmo utilizando uma ferramenta de controle de versão, o processo de tomar conhecimento das mudanças que ocorrem nos artefatos continua sendo mecânico, o que é crítico, principalmente na fase de integração das partes.

A ferramenta proposta neste trabalho provê, a partir de um mecanismo de notificação, o compartilhamento de informações mesmo na ausência de um ambiente colaborativo. Sabe-se que os times podem utilizar ferramentas distintas mas precisam encaminhar os artefatos a um repositório central. Por este motivo, a ferramenta de controle de versão será o meio pelo qual as notificações serão geradas e encaminhadas aos colaboradores que são afetados diretamente pelas mudanças de estado dos artefatos.

A solução proposta é chamada de CVSyn porque é projetada para atuar junto ao CVS, uma ferramenta de controle de versão, através do acoplamento de um mecanismo de notificação síncrono. A intenção é fornecer a infra-estrutura que permite ao CVS disparar notificações sempre que houver mudança no seu repositório de artefatos.

Para o ambiente do projeto, a concepção do CVSyn "sugere" a utilização de um ambiente colaborativo uma vez que, mesmo utilizando as ferramentas localmente o resultado do trabalho de um desenvolvedor pode ser compartilhado com todos aqueles que estejam interessados nele. Também pode ser utilizada como suporte à gestão do projeto, uma vez que permite aos colaboradores não envolvidos diretamente nessa etapa do desenvolvimento se manterem informados das ocorrências, como é o caso dos gerentes de projeto. Em geral, os usuários em potencial da ferramenta são programadores, testadores, integradores e gerentes que participam de projetos, muitas vezes geograficamente distribuídos.

4.2 A definição dos termos utilizados na construção do CVSyn

No capítulo anterior, quando a tecnologia de servidores de notificação de eventos foi apresentada, alguns termos foram freqüentemente mencionados. Para maior entendimento de aspectos funcionais da solução proposta, apresenta-se nesta seção a definição de alguns desses termos.

levantamento de requisitos foi realizado. As subseções que seguem descrevem os requisitos funcionais e não-funcionais da ferramenta, respectivamente.

4.3.1 Requisitos funcionais

Os requisitos funcionais descrevem as funcionalidades que devem estar presentes na ferramenta. Para comentar tais requisitos, as propriedades listadas no Capítulo 2, Seção 2.4.2, são neste momento retomadas.

RF1. Usuário visualiza as fontes de informação disponíveis: durante a construção da aplicação, muitas ferramentas são utilizadas para a geração de código, testes e documentos, como os relatórios resultantes da integração. Como nem sempre há um mecanismo que permita difundir globalmente as informações geradas pelas ferramentas, como acontece quando uma solução colaborativa é utilizada, o compartilhamento das informações críticas não acontece automaticamente. Além disso, o projeto pode requerer múltiplos repositórios de artefatos. É importante ter conhecimento de todas as fontes geradoras dos eventos por dois motivos: primeiro para permitir que os consumidores saibam que fontes estão disponíveis e também para que o servidor possa gerenciar as solicitações que chegam até ele.

RF2. Usuário visualiza os objetos que estão sendo monitorados: o usuário deve ter acesso aos artefatos que estão sendo monitorados, principalmente para que possa identificar aqueles que afetam a realização das suas tarefas e assim, se subscrever como consumidores de informações.

RF3. Usuário conhece as ações que afetam os artefatos: o usuário deve ter acesso aos eventos ou ocorrências que geram mudanças nos objetos. Este é um requisito crítico para auxiliar a resolução de conflitos. Caso o colaborador tome conhecimento da conclusão de um novo código ou da alteração de um componente no momento em que o evento ocorre, a sincronização do colaborador com as alterações pode acontecer em tempo hábil, minimizando assim, os impactos das distorções ou a perda de trabalho. Ao utilizar o CVS, por exemplo, o colaborador apenas toma conhecimento das mudanças quando este requisita uma operação

chamada *update* ou quando tenta cometer suas alterações. Se o colaborador demorar a acessar o repositório, a resolução de conflitos torna-se muito mais difícil, principalmente se os colaboradores trabalharem com pequenas diferenças de fusos-horários.

RF4. Usuário cadastra-se como um consumidor de informação: é importante que o colaborador possa manifestar o interesse em receber informação de determinadas ocorrências como forma de selecionar apenas aquelas notificações que podem afetar o seu trabalho. O principal benefício desta funcionalidade é permitir que o próprio usuário defina o interesse por novos objetos ou exclua aqueles que não mais lhe interessam.

RF5. Usuário define a ferramenta colaborativa que deseja utilizar: permite que os usuários possam escolher o dispositivo pelo qual desejam receber as notificações. Dependendo da sua função no projeto mais de um dispositivo pode ser necessário.

4.3.2 Requisitos não-funcionais

Os requisitos não-funcionais não estão diretamente ligados às operações do sistema e sim, as questões que envolvem o seu contexto, como plataforma operacional, tempo de resposta, tipo de interface, entre outras (LARMAN,2000).

1. As ferramentas utilizadas para a produção de código devem funcionar como provedores de informação, para isso, pouca ou nenhuma modificação deve ser implantada nas ferramentas.
2. As ferramentas podem estar sendo utilizadas localmente ou remotamente.
3. A inclusão de uma nova fonte ou consumidor de informação não deve causar impacto no ambiente.
4. O sistema deve oferecer uma interface Web que facilite a chamada de métodos para o cadastramento das fontes e para a definição de novos eventos e indicação dos objetos. Do lado dos consumidores, as ferramentas colaborativas devem facilitar a visualização da chegada de novos eventos oferecendo recursos de mídia, a exemplo da utilização de som ou imagem.

5. As notificações devem ser repassadas aos consumidores na mesma ordem que chegam ao servidor de notificação. Não deve haver duplicação.
6. Não há restrição com relação ao número de fontes de informação que se comunicam com o servidor, contudo este aspecto deve ser tratado de forma a evitar que nenhuma notificação deixe de ser entregue a qualquer consumidor.
7. As fontes de informação devem se registrar junto ao servidor para que se tenha controle de quem está gerando os eventos.
8. O ambiente deve ser munido de ferramentas que favoreçam a existência de *thin clients*.

4.4 Projeto arquitetural do CVSyn

Antes de construir a arquitetura-base da aplicação os problemas mais críticos que poderiam interferir no cumprimento dos requisitos foram avaliados. Esta estratégia é conhecida como ***risk confronting***.

No cenário investigado há um número qualquer de ferramentas que precisam "avisar" quando determinadas ações são realizadas. Contudo, nem todas as ferramentas são capazes de gerar eventos, já que muitas delas não foram projetadas com este intuito. O primeiro desafio a ser solucionado relacionava-se com a necessidade de definir como "escutar" um número arbitrário e heterogêneo de ferramentas.

Um obstáculo levantado poderia corromper um dos requisitos não-funcionais: pouca ou nenhuma mudança deveria ser implantada nas ferramentas dos colaboradores. Duas questões foram identificadas: como acoplar um mecanismo de monitoramento sem causar alterações no código da ferramenta e como disponibilizar tais ferramentas aos interessados, sem custos adicionais de aquisição?

Considerando que muitas das ferramentas estariam sendo executadas localmente, a tarefa de monitorar os eventos apenas seria finalizada quando as requisições de notificação chegassem até o servidor de notificação de eventos que as encaminharia aos interessados. Neste caso, uma solução seria construir módulos de software que executassem localmente aguardando a ocorrência de eventos previamente conhecidos. Após uma detecção, o módulo repassaria as informações ao servidor. Esta solução resolveria parte do problema, mas acabaria sendo inviabilizada devido à cultura organizacional e a política de segurança adotada pela organização usuária e suas parceiras. Serviços de FTP (File Transport Protocol), por exemplo, não são bem-vindos, na maioria das vezes.

A solução encontrada centraliza os esforços de notificar as mudanças, antes delegada a cada uma das ferramentas, à ferramenta de controle de versão em uso. Sendo assim, todos os artefatos que precisam ser monitorados devem ser cometidos ao repositório que, por sua vez, será constantemente "espionado". É importante mencionar que a estratégia combina aspectos metodológicos e tecnológicos na tentativa de reduzir a complexidade da solução e os custos de aquisição de ferramentas.

A arquitetura do CVSyn é dividida em camadas. Um modelo habitualmente utilizado que separa a lógica da aplicação em uma camada intermediária, das camadas de apresentação (interface com o usuário) e de armazenamento persistente dos dados, é conhecido como arquitetura em três camadas (LARMAN, 2000).

A camada de apresentação é relativamente livre de processamento ligado à aplicação, ficando geralmente encarregada de repassar solicitações de tarefas para uma camada intermediária ou receber o resultado de um processamento. A camada intermediária (lógica da aplicação ou camada de aplicação) se comunica com a camada de armazenamento internamente, por trás da aplicação. A camada de armazenamento (camada de banco de dados) constitui o mecanismo de armazenamento persistente adotado pela arquitetura.

CVSyn possui arquitetura em 4 camadas. A lógica da aplicação foi decomposta em duas camadas: a Camada de Domínio e a Camada de Serviços. A Camada de Domínio é composta pelos pacotes *comunication*, *rmi* e *listener*. A Camada de Serviços é constituída pelo pacote *cassandra*, que fornece acesso ao armazenamento das notificações.

Utilizando a abordagem de pacotes[13], uma estratégia que favorece o fraco acoplamento entre as camadas e aumenta a coesão entre as partições, a arquitetura foi organizada como pode ser visto através da Figura 6.

O pacote *listener* contém as interfaces e classes responsáveis pelo monitoramento da ferramenta de controle de versão. Já o pacote *rmi* é responsável pelo gerenciamento de informações que são geradas quando um evento acontece, as quais podem ser utilizadas por mais de um processo concomitantemente. Como a comunicação com o servidor de notificação acontece através de requisições http, o pacote *comunication,* juntamente com o pacote c*assandra* (fornecido pelo servidor)*,* são encarregados de prover a interação com o servidor, abrindo as conexões necessárias, realizando o controle das mesmas e mantendo as persistência das notificações que chegam ao servidor. O pacote GUI contém as classes responsáveis pela construção de ferramentas colaborativas.

[13] Pacotes são mecanismos utilizados pela UML (*Unified Modeling Language*) que tem a finalidade de ilustrar o agrupamento de elementos semanticamente interligados.

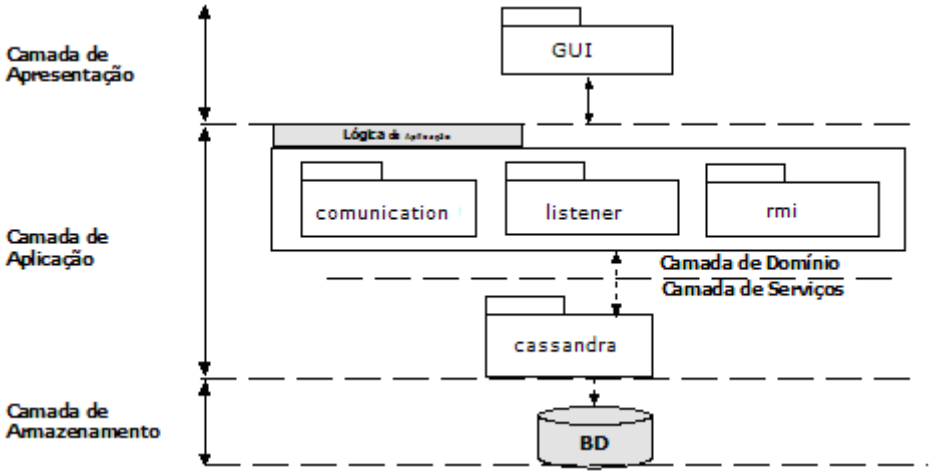


Figura 6: Arquitetura do CVSyn em pacotes

Sob uma outra perspectiva, a arquitetura do CVSyn adota os conceitos sugeridos pelo padrão arquitetural MVC (Model-View-Controller) (GAMMA *et at*, 2000) (Figura 7). O MVC procura diminuir ao máximo o acoplamento direto dos componentes da lógica da aplicação com os objetos da GUI (Graphical User Interface), proporcionando a reutilização da lógica em novas aplicações e minimizando o impacto das mudanças de requisitos de interface sobre a camada de domínio. A utilização do padrão MVC permite definir a separação, de maneira independente, do Model (Modelo) – que são os objetos que constituem a lógica da aplicação – da View (Visão), que compreende os objetos constituintes da interface com o usuário. O Controller (Controlador) define a maneira como a interface do usuário reage às entradas da mesma, ficando responsável pelo controle do fluxo da aplicação.

Um dos requisitos que devem ser atendidos pelo CVSyn é permitir a flexibilidade na forma com que os colaboradores tomam conhecimento das notificações. Logo, é possível que seja necessário utilizar, por exemplo, um navegador de Internet *(browser)* para notificar os desenvolvedores, um *applet* para notificar o integrador e um celular para notificar o gerente de projeto. A utilização do padrão MVC é importante se considerarmos a necessidade de incluir novas ferramentas de visualização.

A camada que contém a lógica da aplicação prevê que a ferramenta de controle de versão é a fonte de informação. A ela está acoplado um módulo denominado **Escutador** que monitora as mudanças que ali ocorrem, para que possa a partir daí identificar que artefatos sofreram alguma ação e que ação foi essa. Após a identificação, o Escutador organiza tais informações na forma de uma notificação (que neste momento ainda não usa

o formato de notificação do CASSIUS) e a encaminha ao servidor, responsável por realizar seu encaminhamento aos interessados, agora no formato descrito no Capítulo 3, Seção 3.4.

Por várias razões, dentre elas, a segurança no acesso às informações e eficiência no sentido de lidar com perdas na rede, o módulo Escutador está fisicamente localizado na máquina em que a ferramenta de controle de versão estará executando, possivelmente em um servidor de aplicação. O CASSIUS, porém, pode ser acessado remotamente. Este, por sua vez, mantém a persistência dos dados (neste caso das notificações) utilizando o SGBD MySQL.

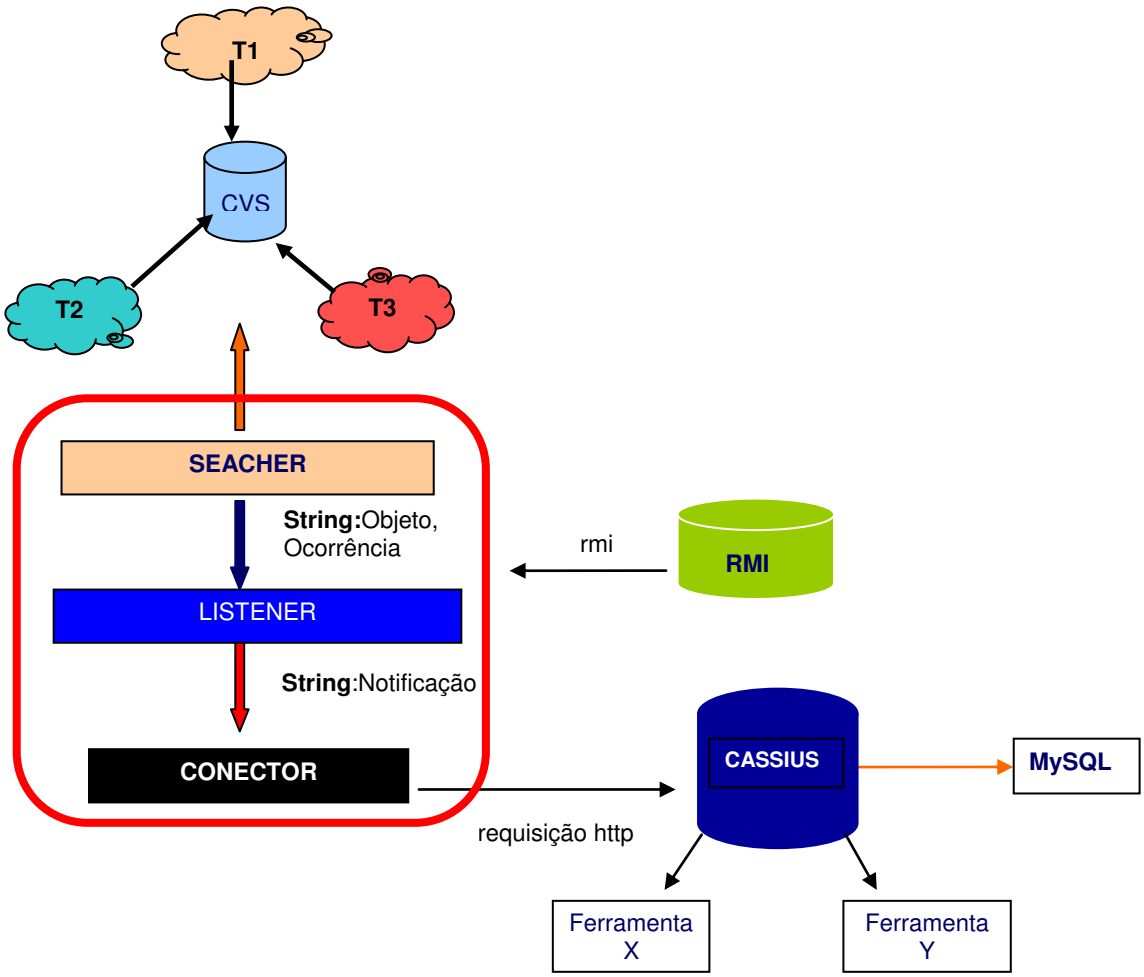


Figura 7: Arquitetura do CVSyn de acordo com o MVC

4.5 A lógica da solução do CVSyn

Como mencionado na Seção 4.2 o CVS foi a ferramenta escolhida para compor a solução. O CVS possui uma política de controle sobre as modificações que ocorrem no seu repositório de artefatos. Sempre que um artefato é atualizado, um arquivo de administração, o CVSROOT/history (Figura 8) também é atualizado. O CVSROOT/history funciona como uma espécie de histórico (arquivo de log).

Para monitorar a ferramenta é necessário ter controle sobre tal arquivo que, numa freqüência relativamente grande é modificado. Por este motivo, o módulo responsável pelo monitoramento do arquivo deve estar atento a dois aspectos importantes: o primeiro se refere justamente a entender quando o arquivo foi alterado, o segundo aspecto é como manter o domínio sobre o ponto em que a última alteração ocorreu, evitando assim que em uma próxima verificação ao arquivo nenhuma informação seja desconsiderada ou lida novamente.

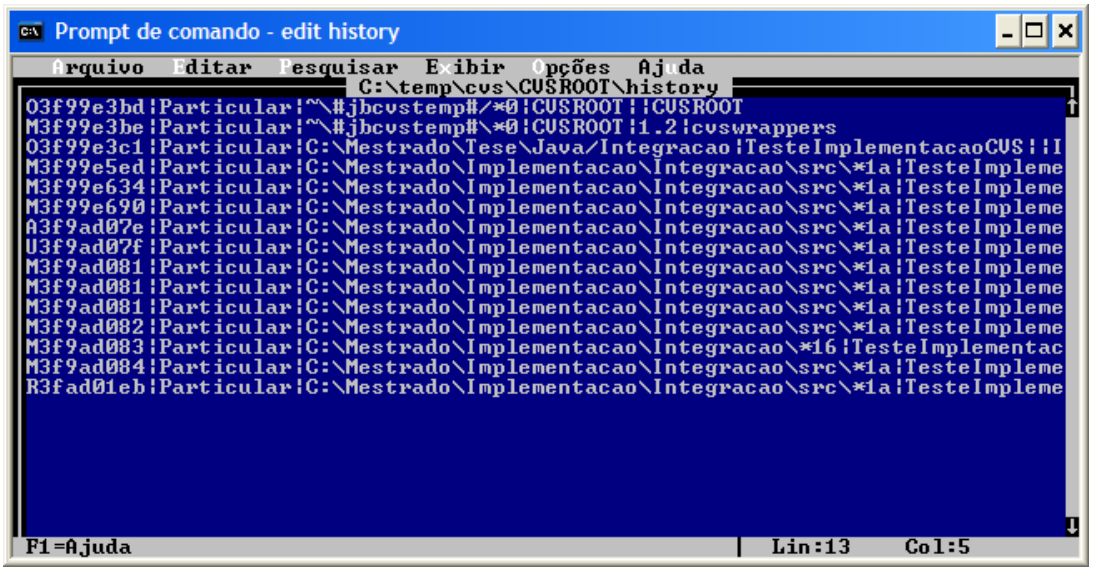


Figura 8: Formato do arquivo CVSROOT/history e seu estado em um dado momento

Para que um artefato seja incluído ou alterado no repositório do CVS, uma operação chamada *commit* deve acontecer (maiores detalhes podem ser encontrados no Anexo B). Um outro arquivo do CVS, chamado loginfo, controla o local para onde as informações de um *commit* devem ser enviadas, permitindo, além disso, que outros programas sejam ativados na ocorrência de um *commit*. Para reduzir o tráfego de verificações que devem acontecer no CVSROOT/history, o Escutador é chamado pelo loginfo sempre que um artefato é cometido ao repositório.

Como há muitos desenvolvedores acessando o repositório, muitos *commits* ocorrerão em pequenos intervalos de tempo, o que significa dizer que o Escutador será chamado nessa mesma proporção. Sempre que chamado, o módulo guarda o ponto da última verificação[14] para que ele possa se orientar na próxima vez em que for chamado. Já que a aplicação é chamada quando um *commit* acontece, mais de uma JVM (Java Virtual Machine) podem estar acessando o mesmo ponto do CVSROOT/history. O que pode acontecer é que a diferença de tempo entre o acesso e gravação do último ponto de verificação, que está sendo realizado por JVMs distintas, gere conflitos.

Isto pode gerar duas implicações: a mesma notificação será enviada mais de uma vez ao servidor ou algumas notificações serão desconsideradas (já que pode acontecer de a linha correspondente àquela notificação não ser lida). Sendo assim, é preciso gerenciar o acesso e gravação daquele ponto de verificação. Depois que o arquivo CVSROOT/history foi lido, as informações devem ser interpretadas para que uma notificação seja montada e repassada ao servidor.

4.6 Projeto de baixo nível

Um sistema orientado a objetos é composto de objetos que enviam mensagens uns aos outros. Escolher como distribuir responsabilidades entre classes é crucial para um bom projeto. Uma má distribuição leva a sistemas e componentes frágeis e difíceis de entender, manter, reusar e estender (LARMAN,2000).

Durante a concepção da solução lógica, um dos maiores cuidados seria justamente projetar uma solução que pudesse ser robusta, reutilizável e de fácil manutenção. A ferramenta foi projetada para atuar sobre o CVS, contudo, a solução não deveria estar amarrada a esta ferramenta de controle de versão, nem ao CASSIUS, o servidor de notificação. Várias seriam as formas de ter acesso às ações que modificariam o repositório de artefatos, dependendo da ferramenta de controle de versão utilizado.

Para isso, padrões de projeto foram utilizados. Alguns dos padrões GRASP (General Resposability Assignment Software Patterns), dentre eles o *creator*, *expert*, alta coesão e baixo acoplamento (LARMAN,2000), e alguns sugeridos pelo GoF (Gang of Four) (GAMMA *et al*,2000), a exemplo do Strategy. A utilização de interfaces (herança de tipo) foi um princípio bastante praticado para prover flexibilidade ao código, evitando assim, que novas alterações causassem grande impacto à aplicação.

[14] O ponto de verificação é na verdade o número da linha do arquivo CVSROOT/history mais recente que foi lido.

Como pode ser observado no diagrama de classes[15] (Figura 9), uma interface Listener foi definida e contém as operações: *search(), configure(), createConector() e createSearcher()*. A finalidade desta interface é permitir que diferentes tipos de Escutadores possam ser manipulados. A intenção é permitir que diante de outras ferramentas de controle de versão, novas estratégias possam ser utilizadas para monitorar as ocorrências dos eventos (assim como sugere o padrão Strategy). Para este caso, onde o CVS é a ferramenta em questão, a implementação dessa interface é a classe CVSListener.

Um objeto do tipo Listener invoca a operação de procura de novos eventos, que é realizada por um outro objeto que implementa, por sua vez, a interface Searcher (que define o método *searchNotification()*), o objeto FileReader. Neste caso, a implementação de um Listener delega responsabilidades a outro objeto, aquele que implementa Searcher.

Quando a aplicação é chamada, um objeto cvsListener faz uso de dois objetos: um fileReader, que possui a responsabilidade de "entender" o que aconteceu no repositório e montar uma notificação, e de um objeto conector, responsável por repassar a notificação ao servidor.

A importância de um objeto do tipo Listener, neste caso o cvsListener, delegar operações a outros objetos é que, tanto a estratégia de monitoramento da ferramenta de controle de versão pode ser trocada quanto a forma de acesso ao servidor de notificação. Se houver necessidade de permutar o servidor, o objeto do tipo Listener através do método *createConector()* instancia aquele objeto que conhece a API do novo servidor e sabe como a ele se conectar.

O cvsListener interpreta um arquivo do tipo Properties (do pacote java.util). A finalidade é configurar algumas informações que serão utilizadas no momento em que os objetos fileReader e conector estão sendo criados. O cvsListener, nesse passo, configura algumas propriedades do objeto fileReader, como por exemplo, o local em que se encontra o CVSROOT/history.

[15] A notação UML (Unified Modeling Language) foi utilizada na construção no diagrama de baixo nível.

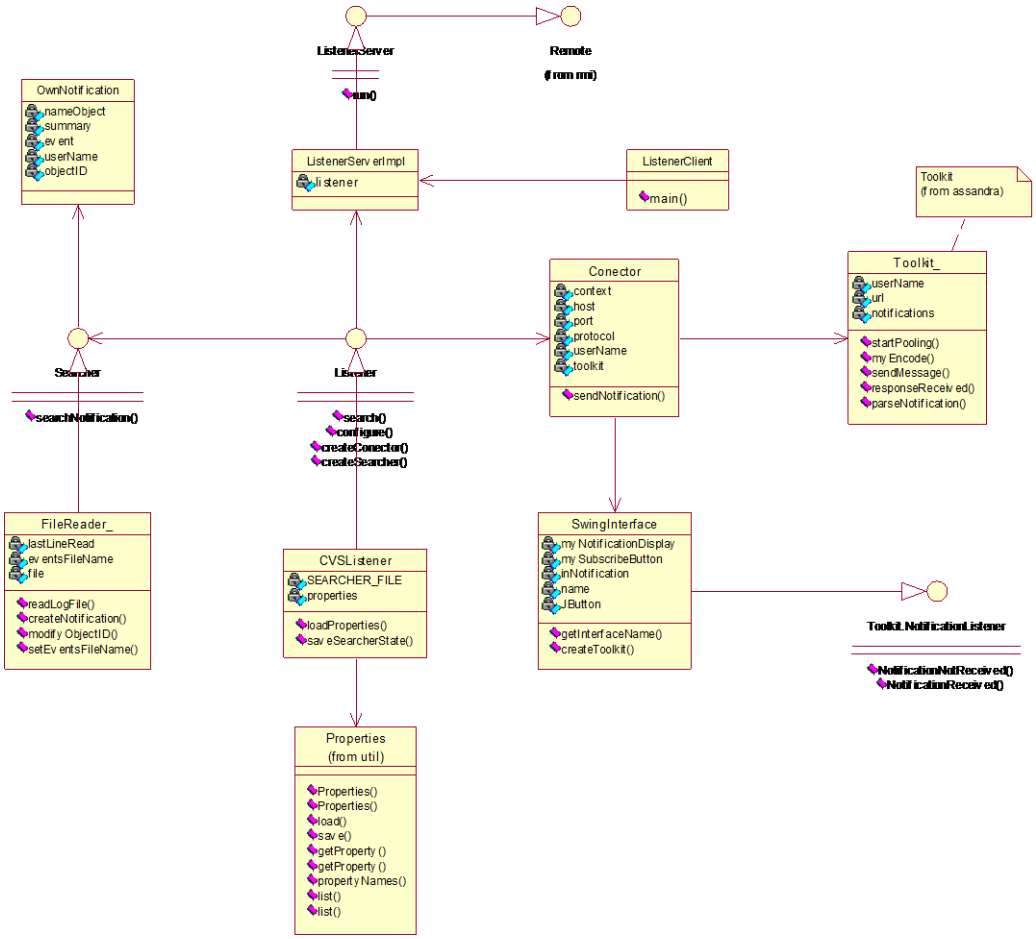


Figura 9: Diagrama de baixo nível

Ainda naquele passo, no momento em que o cvsListener cria uma instância do conector, algumas informações que auxiliam a comunicação com o servidor, a exemplo do tipo de protocolo da conexão e o endereço do servidor, são configuradas. O benefício de fornecer estas informações através de um arquivo Properties é que as alterações não estão vinculadas à lógica da aplicação. Sendo assim, caso o servidor mude, por exemplo, nenhum impacto ocorre. Só é preciso modificar e salvar o arquivo.

O objeto fileReader, de posse das informações de que precisa, lê o arquivo através da implementação do método *searchNotification()*, ou seja, *readLogFile()*, e o interpreta através de *createNotification()* (quadro 1). Este método permite que a informação, que representa o evento no repositório, seja organizada na forma de uma notificação. A classe OwnNotification é um bean que é instanciado por este objeto para prover tal estruturação. É valido mencionar que apenas ser notificado da ocorrência de um *commit* pode não ser

suficiente para aqueles desenvolvedores que são afetados por este evento. Quando um *commit* ocorre, três ações podem ter acontecido, ou objeto foi modificado, ou removido, ou cometido pela primeira vez no repositório. A implementação do método *createNotification()* também é responsável por tratar esta informação para melhor informar aos consumidores.

Um objeto ownNotification permite então, estruturar que versão do objeto sofreu a ação (*getNameObject()*), o que aconteceu com o objeto, visualizar o comentário deixado pelo colaborador no momento da alteração (*getSummary()*), o nome do colaborador (*getUserName()*), e o que realmente aconteceu no artefato (*getEvent()*), além de compor outras informações que serão utilizadas pelo servidor, como é o caso do identificador do objeto.

Como foi mencionado no Capítulo 3, tanto as fontes quanto os objetos precisam ser cadastrados junto ao servidor, o cadastramento dos objetos requer que cada um deles possua um identificador que o diferencie dos demais objetos cadastrados. No momento em que a notificação é solicitada junto ao servidor, o identificador do objeto deve ser repassado para que o CASSIUS valide a solicitação.

```
1   public OwnNotification createNotification(String line){
2-11 …
12
13    while (tokenizer.hasMoreTokens())
14    {
15      event = tokenizer.nextToken("|");
16      username = tokenizer.nextToken("|");
17      changeDir = tokenizer.nextToken("|");
18      localRepository = tokenizer.nextToken("|");
19      version = tokenizer.nextToken("|");
20      artifact = tokenizer.nextToken("|");
21    }
22
23    object = artifact+" v."+version;
24    if (event.charAt(0) == 'A'){summary=object+" was added by first time"; }
25      else if (event.charAt(0) == 'M'){summary=object+" was modified"; }
26           else {summary = artifact+" was removed";}
27
28    event="commit";
29    String oi = this.modifyObjectId(changeDir);
30    idObject=oi+"\\"+artifact;
31    OwnNotification ownNotification = new OwnNotification(object,summary,event,
32                                                          username,idObject);
33   return ownNotification;
34
35 }
```

Quadro 1: Parte da implementação do método createNotification

Quando o objeto fileReader termina a execução do método *searchNotification(),* retorna ao objeto cvsListener um objeto do tipo OwnNotification que possui, então, a notificação que será repassada para o objeto conector. Este por sua vez, executa o método *sendNotification()* que chama o método *notify()* da classe Toolkit, que pertence ao pacote cassandra, para interagir com o CASSIUS. A troca de mensagens entre os objetos pode ser observada no diagrama de seqüência abaixo (Figura 10).

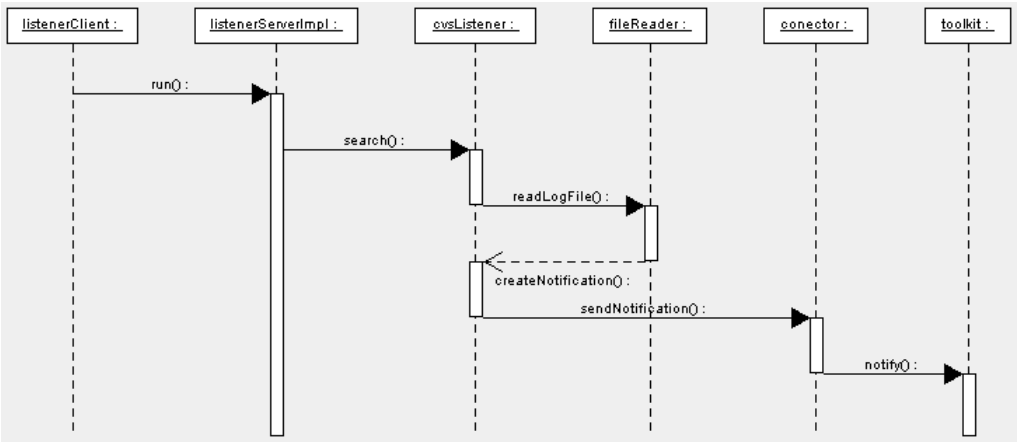


Figura 10: Diagrama de seqüência

Na seção anterior foi mencionado que conflitos podem ocorrer caso os intervalos de tempo em que a aplicação é chamada forem pequenos. É necessário entender como este controle é realizado. Para permitir o domínio do ponto em que o arquivo CVSROOT/history deve ser lido, a tecnologia RMI (Remote Method Invocation) foi utilizada.

RMI permite que objetos Java se comuniquem entre si a partir de diferentes computadores ou processos através da chamada de métodos remotos. Para o programador, a chamada desses métodos acontece de maneira semelhante se esses objetos estivessem sendo operados em um mesmo programa. RMI permite que programas Java possam transferir objetos completos a partir de um mecanismo de serialização (DEITEL,1999). Para construir um sistema distribuído RMI é preciso definir: i. uma interface remota, ii. a implementação de um objeto remoto, iii. uma aplicação cliente que utiliza este objeto remoto.

Para definir uma interface remota é necessário estender a interface java.rmi.Remote. Um objeto de uma classe que implementa a interface Remote direta ou indiretamente é um objeto remoto e pode ser acessado, com as permissões de segurança apropriadas, a partir

de qualquer JVM que possui uma conexão com o computador que executa o objeto remoto.

Todo método remoto precisa ser declarado na interface que estende a classe java.rmi.Remote. O objeto remoto, por sua vez, precisa implementar todos os métodos definidos na interface remota. Os métodos da interface remota devem declarar uma cláusula *throws* que indica que o método pode lançar uma exceção RemoteException, que sinaliza um problema na comunicação com o objeto remoto.

Depois de definidos, os objetos remotos precisam ser registrados junto a um servidor RMI, para que possam aguardar a solicitação de conexão das aplicações clientes. O **rmiregistry** é um programa que gerencia o registro dos objetos remotos junto ao servidor RMI. Através do método *lookup(),* que conecta a aplicação ao **rmiregistry**, a referência para os objetos remotos pode ser obtida.

Uma referência remota refere-se a um objeto *stub* no cliente. Um *stub* permite que clientes invoquem métodos de objetos remotos. Um objeto *stub* recebe cada chamada remota de método e repassa ao sistema RMI, que realiza a comunicação em rede entre clientes e objetos remotos. Finalmente, a camada RMI é responsável pelas conexões de rede aos objetos remotos, permitindo que a comunicação seja transparente para os clientes.

No contexto da solução, a interface remota ListenerServer define o método *run()* (quadro 2).

```
1 package rmi;
2 import java.rmi.*;
3
4 public interface ListenerServer extends Remote {
5 public void run() throws RemoteException;
6 }
```

Quadro 2: Interface ListenerServer

A classe de objetos remotos ListenerServerImpl[16] implementa esta interface (quadro 3). O cliente interage com um objeto da classe ListenerServerImpl invocando o método *run()* da interface ListenerServer para obter instanciar um objeto do tipo Listener e ter acesso ao ponto de verificação do arquivo CVSROOT/history de maneira confiável.

ListenerServerImpl estende a classe UnicastRemoteObject (do pacote java.rmi.server) porque esta oferece as funcionalidades básicas necessárias para todos os

[16] Por convenção, a classe de implementação do objeto remoto possui o mesmo nome da interface remota com o final Impl.

objetos remotos. Em particular, seu construtor exporta o objeto para torná-lo disponível para receber chamadas remotas. O método main() cria o objeto remoto ListenerServerImpl.

A linha 32 define a URL que o cliente pode utilizar para obter uma referência remota para o objeto. A URL normalmente é na forma: **rmi://host:port/remoteObjectName**, onde o **host** representa o computador que está executando o registro dos objetos remotos, **port** representa o número da porta em que o registro está rodando no host e **remoteObjectName** é o nome fornecido para o cliente quando este tenta localizar um objeto remoto no registro.

```
1
2 package rmi;
3 import comunication.Conector;
4 import listener.*;
5 import java.rmi.*;
6 import java.rmi.server.*;
7 import java.rmi.Naming;
8
9 public class ListenerServerImpl extends UnicastRemoteObject
10                              implements ListenerServer{
11
12  private Listener listener;
13
14  public ListenerServerImpl() throws RemoteException{
15    super();
16 }
17
18 public synchronized void run () throws RemoteException{
19    try
20   {
21      listener = new CVSListener();
22      listener.search();
23   }
24    //catch(java.net.ConnectException ce) {ce.printStackTrace();}
25    catch(Exception e){e.printStackTrace();}
26  }
27
28  public static void main(String args()) throws Exception{
29    try{
30      System.out.println("Initializing server...");
31      ListenerServerImpl lsi = new ListenerServerImpl();
32      String serverObjectName="//localhost/ListenerServer";
33      Naming.rebind(serverObjectName,lsi);
34      System.out.println("ListenerServerImpl bounded in registry");
35     }catch (Exception e){
36        System.out.println("ERROR binding ListenerServerImp"+e.getMessage());
37        e.printStackTrace();}
38  }
```

Quadro 3:Classe ListenerServerImpl

A linha 33 invoca o método estático *rebind()* da classe Naming (pacote java.rmi) para ligar o objeto remoto listenerServerImpl ao registro de objetos. O método *bind( )* também poderia ter sido utilizado, no entanto, o método *rebind()* garante que se um objeto já foi registrado com um determinado nome, o novo objeto remoto será substituído pelo objeto previamente registrado.

Como pode ser visto no quadro 4, uma aplicação cliente, a classe ListenerClient, precisa ser definida para obter informações do ListenerServerImpl. O construtor acessa o nome do computador em que o objeto remoto está executando. A linha 10 invoca o método estático *lookup()* para obter uma referência ao objeto remoto. Este método se conecta ao registro do RMI (através do **rmiregistry**) e retorna a referência do objeto.

```
1 package rmi;
2 import java.rmi.*;
3
4  public class ListenerClient {
5  public ListenerClient (String server){
6
7   try
8    {
9     String serverObjectName = "rmi:// " + server + "/ListenerServer";
10     ListenerServer server = (ListenerServer)Naming.lookup(serverObjectName);
11     server.run();
12    }catch(java.rmi.ConnectException ce)
13      {System.out.println("Connection to server failed.");}
14      catch(Exception e) {e.printStackTrace();}
15 }
16
17   public static void main(String args()){
18    ListenerClient client = null;
19    client = new ListenerClient("localhost");
20  }
21}
```

Quadro 4: Classe ListenerClient

O cliente pode utilizar esta referência remota como se estivesse referenciando um objeto local que está rodando em uma mesma JVM. Tal referência remota refere-se a um objeto *stub* no cliente. Os *stubs* permitem aos clientes invocarem métodos de objetos remotos. Eles recebem cada chamada de método e repassa essas chamadas para o sistema RMI, que realiza as tarefas de rede necessárias para que o cliente interaja com o objeto remoto. O *stub* ListenerServerImpl conduz a comunicação entre o cliente (ListenerClient) e ListenerServerImpl. A linha 11 invoca o método remoto *run()* que é **synchronized** para

permitir a manipulação confiável do ponto de verificação, já que vários processos podem estar invocando o objeto remoto.

Quando a aplicação é ativada, um objeto ListenerClient é chamado. Na verdade, o que acontece é que a informação passa a ser manipulada de maneira centralizada pelo objeto *stub* ListenerServerImpl, através do método remoto que é *synchronized*. Este mecanismo evita que vários objetos do tipo Listener possam ser instanciados ao mesmo tempo e manipulem o ponto de verificação sem qualquer tipo de controle.

4.7 Considerações Finais

Neste capítulo, aspectos inerentes ao processo de construção de uma aplicação foram apresentados. Para dar suporte ao desenvolvimento da ferramenta o processo de desenvolvimento sugerido por Craig Larman (LARMAN,2000) foi adotado. Por seu caráter iterativo e incremental, a metodologia de desenvolvimento foi útil para consolidar os conceitos na etapa de análise e refinar o entendimento dos requisitos na fase de projeto, o que aconteceu em duas iterações.

Como pode ser visto, o projeto arquitetural foi concebido com a intenção de prover o máximo de flexibilidade possível, tanto no que se refere à ferramenta de controle de versão a ser utilizada, quanto ao servidor de notificação de eventos. Duas fronteiras deveriam ser vencidas: conhecer a mudança que afetavam os artefatos e repassar tais mudanças na forma de notificações, em escala de Internet, para os interessados.

As ferramentas precisariam sinalizar de alguma forma a mudança de estado nos artefatos. Como estamos tratando com ferramentas bastante heterogêneas não seria possível construir módulos genéricos o bastante que pudessem monitorar um número desconhecido de ferramentas, sem excluir uma quantidade considerável de ferramentas. Então, o monitoramento dessas ferramentas se deu através da ferramenta de controle de versão utilizada no projeto.

O fundamental a ser observado nesta parte do trabalho é como se deu a preparação de uma fonte de informação (neste caso o CVS), ou seja, como uma ferramenta que não houvera sido projetada para tal coisa se tornou apta a enviar mensagens a um servidor de notificação de eventos.

O capítulo seguinte detalha outros aspectos da implementação do protótipo e de sua validação. Os passos necessários para configurar a fonte de informação junto ao

CASSIUS são apresentados. Os problemas e dificuldades também são mostrados como forma de conduzir aqueles interessados em utilizar ou melhorar a solução aqui oferecida.

Capítulo 5

A Comunicação com o CASSIUS

No capítulo anterior, o projeto do CVSyn foi ilustrado. Contudo, o relacionamento efetivo da solução com o servidor CASSIUS ainda não fora apresentado. O presente capítulo oferece de maneira objetiva e prática os passos necessários para instrumentar uma fonte de informação junto ao servidor, bem como os requisitos de software que precisam ser atendidos para que a comunicação seja realizada. Ao final deste capítulo, o cenário de simulação montado para validação da ferramenta é descrito.

5.1 Considerações iniciais

Até agora abordamos como uma ferramenta de controle de versão deve ser preparada, ilustrando com o CVS, para que ela se torne o provedor de informação de várias outras ferramentas. O que foi apresentado até o momento descreve como se dá o encaminhamento das informações até o CASSIUS, o servidor de notificação de eventos escolhido. No entanto, ainda não foi mencionado como CVSyn trabalha em conjunto com o servidor de notificação.

No Capítulo 3, quando apresentamos a tecnologia de servidores de notificação, algumas características do CASSIUS foram ilustradas. Uma delas é que as fontes precisam ser identificadas junto ao servidor para que este valide as solicitações que chegam até ele. Um outro motivo da necessidade desse passo de cadastramento é que, de posse das informações das fontes e seus objetos, o servidor consegue prover aos consumidores a oportunidade de conhecer efetivamente os provedores de informação disponíveis a cada momento, bem como os objetos passíveis de monitoramento.

As seções seguintes ilustram as atividades que devem ser realizadas no lado do servidor de notificação para que o CVSyn possa ser finalizado. A seguir, são apresentados

os passos para cadastrar fontes e objetos, bem como, os consumidores de informação e para flexibilizar a maneira de visualizar as notificações diante de várias ferramentas em uso. Este é justamente o atendimento dos requisitos funcionais mencionados no Capítulo 4. Para facilitar o entendimento, alguns detalhes importantes para a instalação do servidor são apresentados.

5.2 A instalação do servidor

Para executar o servidor dois programas são necessários: um servidor Web executando, como o Jakarta-Tomcat[17], e o SGBD MySQL,[18] como servidor de dados. O servidor CASSIUS[19] é um servlet Java e por este motivo, algumas configurações são necessárias. Ao realizar o download do servidor, o arquivo cass.tar.gz deve ser descompactado no diretório **webapps** do servidor tomcat. É neste momento que o contexto do CASSIUS está sendo criado.

Se o tomcat já estiver rodando é necessário reiniciá-lo para que ele reconheça o novo contexto. O diretório WEB-INF, encontrado após a descompactação do cass.tar.gz, deve ser movido para o contexto do cassius. Depois de configurar o CASSIUS, o próximo passo é criar o banco de dados CASS no MySQL. Opcionalmente, o banco de dados TESTCASS pode ser testado com a finalidade de validar a instalação do banco de dados, o que pode ser feito como mostrado abaixo.

```
>[caminho do mysql/bin] / mysql –D CASS <[caminho do diretório WEB-INF]
\InitializeGlobalTables
>[caminho do mysql/bin] / mysql –D TESTCASS <[caminho do diretório WEB-INF]
\InitializeGlobalTables
Ex.:
>c:\mysql\bin>mysql -D CASS <C:\jakarta-tomcat-4.0-b1\webapps\cassius\WEB-
INF\InitializeGlobalTables
```

Um teste muito simples pode determinar se o CASSIUS está executando com sucesso no tomcat e se consegue acessar devidamente o MySQL. Como a interface do CASSIUS não é amigável, todas as solicitações http devem ser solicitadas através da barra de navegação do browser. Para realizar o teste, a linha a seguir deve ser fornecida, porém esta requisição precisa estar numa única linha (um comando copiar-colar pode introduzir novas linhas):

[17] http://jakarta.apache.org/tomcat/
[18] http:// www.mysql.com
[19] http://www.ics.uci.edu/~mkantor/download.html

```
http://localhost:8080/cassius/cass?Operation=Register&GlobalPassword=Spanikopizza&
Account=Test&AccountClass=TestClass&ApplicationPassword=testpass3
```

A resposta deve ser: 0: Account {TestClass}. Test Registered. Se houver problema no teste, uma das causas pode ser a falta de permissão para o acesso ao banco de dados. O comando abaixo pode ser utilizado para prover as permissões necessárias:

```
>[caminho do mysql ] mysql –e "grant all privileges on * to [conta_do_usuario]@localhost"

Ex.:
>c:\mysql > mysql –e "grant all privileges on * to particular@localhost"
```

5.3 Instrumentando uma fonte de informação

Esta seção trata dos passos necessários para configurar o módulo escutador no CVS, que passa a ser nossa fonte de informação, atuando em conjunto com o servidor CASSIUS. Para fins de validação, buscava-se acompanhar as mudanças que ocorrem no repositório de artefatos. Ou seja, sempre que algum artefato de código fosse cometido ao repositório, uma mensagem seria enviada ao CASSIUS.

É relevante mencionar que a realização das etapas a seguir deve acontecer no momento anterior aquele em que as notificações já estão sendo efetivamente recebidas pelo CASSIUS, isto porque, uma fase de configuração se faz necessária antes que o CVSyn seja efetivamente utilizado pelos seus usuários finais.

5.3.1 Registrando uma fonte de informação

Cada fonte de informação que desejamos cadastrar junto ao servidor CASSIUS está associada a uma classe (AccountClass). O nome da classe representa uma espécie de categoria de fontes de informação. Uma classe pode possuir várias contas (Account) a ela associada, que são utilizadas para identificar outras fontes que pertencem àquela categoria.

Para registrar uma fonte de informação é necessário utilizar uma operação *Register*. Como já mencionado, a interface do CASSIUS requer que as requisições sejam repassadas

via browser. Para nossa solução, uma classe de conta CVSListener foi criada. A esta conta foi associada uma conta chamada Code, como pode ser visto a seguir.

```
http://localhost:8080/cassius/cass?Operation=Register&GlobalPassword=Spanikopizza&
Account=Code&AccountClass=CVSListener&ApplicationPassword=testpass3
```

Para validação, apenas artefatos de código foram submetidos ao repositório do CVS, porém outras contas poderiam ter sido criadas para organizar os artefatos referentes aos testes de unidade, de integração, de geração dos builds, relatórios de integração, entre outros.

Registrar uma conta no servidor requer uma senha global e outras informações que devem ser especificadas. Quando a operação de registro é finalizada, a fonte de informação terá sido registrada, uma conta terá sido criada no servidor e todo o acesso futuro a ele deve acontecer por meio desta conta. Maiores detalhes das operações oferecidas pelo CASSIUS podem ser consultados no Anexo C que contém a API do toolkit CASSandra servidor.

5.3.2 Definindo tipos de objetos e eventos

Depois que a fonte foi registrada (CVSListener), é necessário identificar os objetos associados àquela fonte. Antes de registrar os objetos, é necessário definir o tipo desses objetos. O CASSIUS permite a definição de novos tipos de objetos. Este é um aspecto importante, principalmente para que os consumidores possam ter acesso às informações na linguagem praticada no projeto. Isto significa dizer que se o interesse é monitorar código, o tipo código pode ser definido, se o interesse muda, outros tipos de objetos podem ser criados.

Como estamos interessados em monitorar código Java, e definimos apenas a conta Code, apenas os objetos do tipo *Code* e *Package* foram definidos. O evento definido para estes objetos foi *commit*, já que partimos do pressuposto que os consumidores estariam interessados em dois momentos: (1) quando código é alterado no repositório ou (2) quando novo código fosse cometido a ele, que reflete uma operação de commit no CVS.

```
http://localhost:8080/cassius/cass?Account=Code.CVSListener&Password=testpass3&Operation=DefineType&Type=Code&Event=commit&Definition=Java class modified on repository&Operation=DefineType&Type=Package&Event=commit&Definition=Package modified on repository
```

A definição dos tipos e eventos está associada a uma conta em específico. Se uma outra fonte de informação, que está associada à mesma classe de conta, deseja monitorar objetos do tipo código, o tipo precisa ser novamente definido e associado à referida fonte de informação. Além disso, em uma mesma requisição http é possível realizar a definição de vários tipos de objetos e eventos.

5.3.3 Registrando objetos e os associando à fonte

Para efeito de validação, todos os objetos que representam as classes do pacote listener, que no contexto do servidor são do tipo *Code,* foram criadas. No momento do cadastro dos objetos é possível montar a hierarquia desses objetos. No nosso caso, para definir a estrutura de pacotes e classes, a requisição para o registro de objetos (*NewObject*) utilizou o parâmetro **ParentID**, que é opcional, e possui a finalidade de informar ao servidor de notificação quem é o objeto-pai do objeto que está sendo definido. Observe que no momento em que um objeto é cadastrado, um identificador lhe é atribuído. Posteriormente, a manipulação dos objetos requer que o seu identificador seja fornecido.

```
http://localhost:8080/cassius/cass?Account=Code.CVSListener&Password=testpass3&Operation=NewObject&ObjectName=Reader.java&ObjectID=C:\Mestrado\Implementacao\Integracao\src\listener\reader.java&ParentID=C:\Mestrado\Implementacao\Integracao\src\listener&Type=Code&Description=Class Reader.java of listener package
```

O passo de cadastro dos objetos é importante porque informa ao servidor que há um novo objeto sendo monitorado, dando a ele a informação apropriada para ajudá-lo a associar mudanças com aquele objeto e informação adicional para ajudar os usuários a reconhecer e encontrar as informações desejadas, facilitando assim, o seu cadastramento como consumidor de informação.

5.4 Preparação dos consumidores de informação

Como já mencionado no Capítulo 3, a distribuição do CASSIUS inclui o toolkit CASSandra, útil para a construção de ferramentas colaborativas. Três ferramentas Java também são disponibilizadas junto com o servidor: o EventList Tool, o SimpleScroller Tool (Figura 11) e o BiffArray Tool. Dessa forma, se não houver necessidades mais específicas, as ferramentas podem ser utilizadas como o meio de propagação das notificações[20].

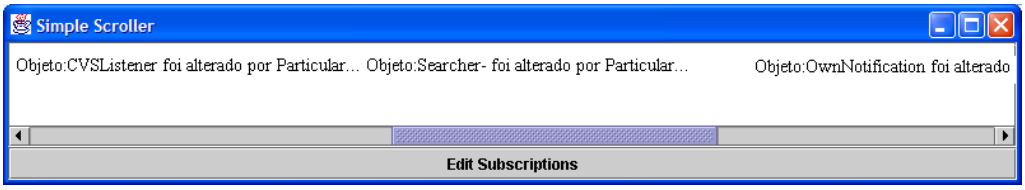


Figura 11: Tela da ferramenta SimpleScroller notificando alterações no repositório

Apesar do CASSIUS disponibilizar tais ferramentas, o desenvolvedor pode construir soluções próprias que se acomodem melhor as necessidades dos colaboradores do projeto. Para construir uma ferramenta colaborativa utilizando o toolkit o programador precisa:

i. implementar a interface Toolkit.NotificationListener;
ii. construir uma instância de Toolkit;
iii. adicionar o NotificationListener a lista de objetos notificationListener do Toolkit;
iv. fornecer, opcionalmente, aos consumidores acesso à janela de subscrição de consumidores, através da criação de um novo objeto *SwingSubscriptionHandler(toolkit,interface_name)* que recebe como parâmetros, a instância do objeto toolkit e o nome daquela interface.

Ao utilizar o CASSandra, toda a interação com o servidor é conduzida pelo *toolkit*, o desenvolvedor apenas se preocupa com a interface do usuário. Também é possível

[20] Para utilizá-las, basta adicionar o arquivo cassandra.jar ao seu classpath.

adicionar novos componentes de interface ao toolkit, um aspecto importante quando há necessidade de trabalhar em determinadas situações que requerem alternativas à tecnologia swing, como J2ME, por exemplo.

Para efeito de validação, criou-se uma ferramenta com interface swing. A solução prevê que o usuário da ferramenta pode receber notificações que afetam até três objetos. Cada botão da ferramenta sinaliza a notificação pertinente a um objeto em particular. Sempre que alguma operação de *commit* acontece nos objetos CVSListener e OwnNotification, uma notificação é enviada ao servidor que permite que o consumidor tome conhecimento do que aconteceu com aquele objeto.

As figuras abaixo representam dois momentos em que diferentes notificações acontecem. No primeiro deles, apenas o objeto CVSListener foi modificado no repositório. Em um momento posterior, outra notificação avisa ao consumidor que o objeto OwnNotification também foi alterado.

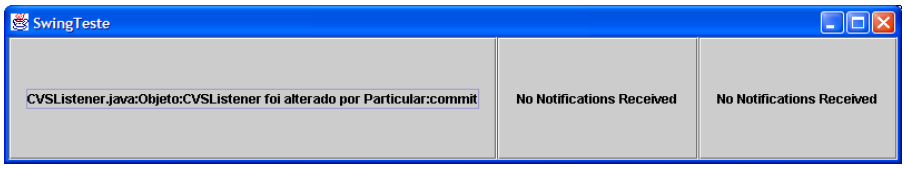


Figura 12: Instante em que o objeto sofreu alteração no repositório

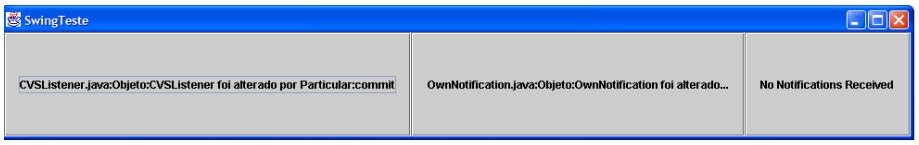


Figura 13: Instante em que o objeto OwnNotification sofreu alteração no repositório

Para tornar-se um consumidor de informação é necessário que o colaborador (desenvolvedor) especifique o interesse nos objetos que são passíveis de monitoramento e nos eventos que afetam aqueles objetos. Além disso, é necessário especificar as ferramentas nas quais deseja receber as notificações. Um consumidor pode utilizar diferentes tipos de ferramentas colaborativas ao mesmo tempo. Como observado nas Figuras 11 e 12, respectivamente, duas ferramentas foram utilizadas para receber notificações.

Para cadastrar-se como um consumidor de informação, o colaborador pode utilizar dois mecanismos. Um deles é utilizar uma ferramenta disponibilizada pelo CASSandra, chamada CASSandra Subscription Editor, mostrada no Capítulo 3, Figura 5. Uma

alternativa é realizar a operação de subscrição através do browser. O que pode ser feito da seguinte maneira:

```
http://localhost:8080/cassius/cass?Operation=Subscribe&UserName=Pasqueline&In
terface=SwingTool&Method=Replace&Formula=[AND Tese.CVSListener.ObjectID
%3D C:\Mestrado\Implementacao\Integracao\src\listener\Listener.java,
Code.CVSListener.Event %3D commit]
```

O primeiro aspecto a ser observado é o sinal de igualdade dentro da definição do parâmetro **Formula**. O sinal "=" deve ser substituído pela cadeia de caracteres "%3D", de forma a evitar ambigüidades com o sinal de igualdade existente fora da fórmula. Neste caso, a fórmula especifica que o consumidor está interessado nas notificações da fonte Code que pertence a classe CVSListener; nas alterações que ocorrem no objeto cujo identificador é C:\Mestrado\Implementacao\Integracao\src\listener\Listener.java e no evento *commit* que pode sofrer aquele objeto. O parâmetro **Method** configurado como Replace instrui o servidor a substituir qualquer subscrição existente para este usuário.

Porém, apesar de permitir que o usuário visualize as fontes disponíveis e os objetos, facilitando assim a seleção dos itens que são de interesse do futuro consumidor, a ferramenta possui alguns problemas. Durante a sua utilização neste trabalho, a ferramenta apresentou algumas falhas. Dependendo da versão do JSDK (Java Software Development Kit) que estiver sendo utilizada, problemas ocorrem no momento de visualizar e editar as subscrições através do CASSandra Subscription Editor. A versão indicada é o JDK 1.3[21]. O J2SDK 1.4 apresentou os problemas anteriormente mencionados.

Para visualizar as subscrições de um consumidor via *browser*, a solicitação a seguir deve ser utilizada:

```
http://localhost:8080/cassius/cass?Operation=ListSubscriptions&Password=testpass3&A
ccount=Code.CVSListener&Username=Pasqueline
```

Observando a Figura 14, é possível acompanhar o formato de resposta fornecido pelo servidor. A mesma figura também demonstra que um consumidor pode estar cadastrado em diferentes ferramentas. Cada uma delas comporta necessidades diferentes, ou seja, se um dado artefato causa um impacto maior, a ferramenta utilizada para prover as

[21] Segundo Roberto Silveira, aluno de doutorado pelo departamento de Informação e Ciência da Computação da Universidade de Irvine e membro da equipe de desenvolvimento do CASSIUS.

notificações pode fazer uso de recursos adicionais como som e alertas visuais, bem como, se adequar a situações em que o consumidor nem sempre pode acessar a ferramenta de um computador de mesa.

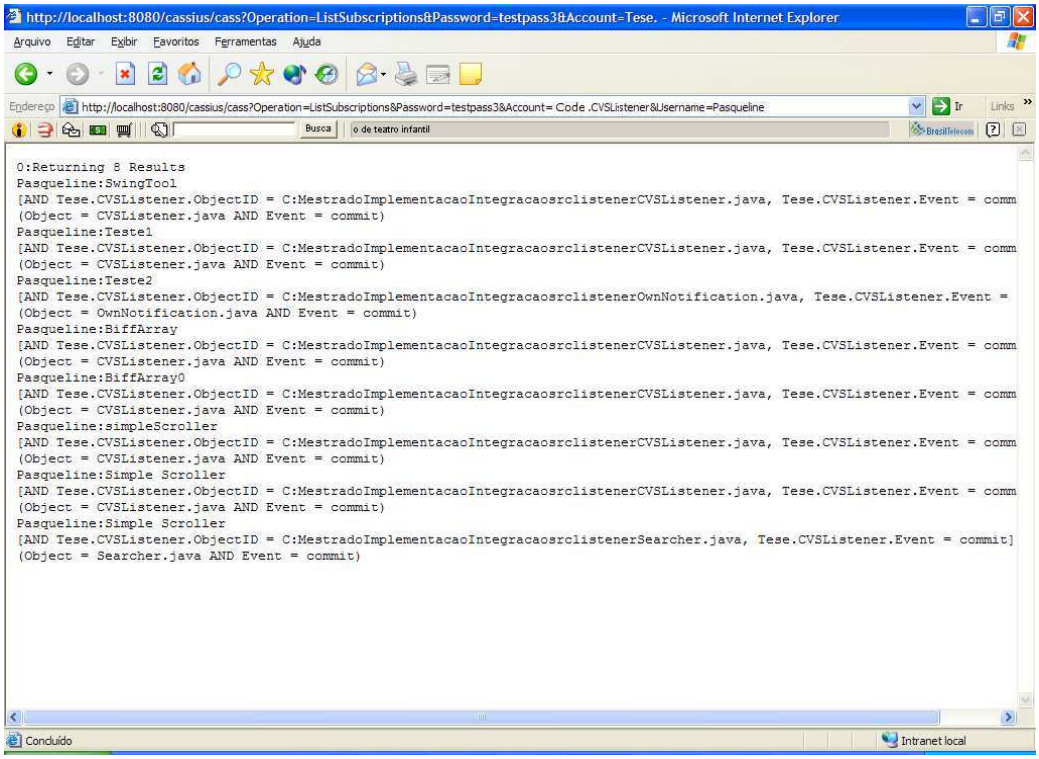


Figura 14: Lista de subscrições do consumidor Pasqueline

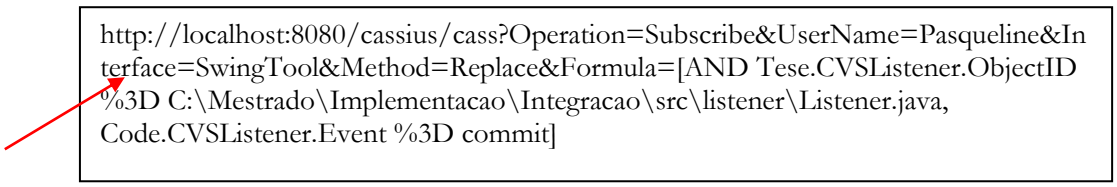
5.4.1 Indicando ao servidor sobre as ferramentas colaborativas dos consumidores

Uma vez que os consumidores foram cadastrados, ainda falta cadastrar no servidor novas ferramentas colaborativas. Apenas ao criar uma ferramenta própria, é necessário realizar a operação *Poll* junto ao servidor. Se o usuário deseja utilizar uma das ferramentas disponibilizadas pelo CASSandra este passo não é necessário. Para cadastrar uma nova ferramenta basta solicitar a seguinte operação:

```
http://localhost:8080/cassius/cass?Operation=Pool&Username=Pasqueline&Interface
=SwingTool
```

Isto equivale dizer ao servidor que há uma nova ferramenta disponível. Para cada subscrição é necessário dizer através de qual ferramenta a notificação acontecerá. O que

equivale dizer que um cliente pode utilizar vários estilos de visualização. Retomando a operação de subscrição mostrada na seção anterior, tem-se que o usuário Pasqueline se subscreveu como consumidor de ocorrências *commits* no objeto Listener.java, e deseja receber notificações através da ferramenta SwingTool.



Se este mesmo consumidor deseja receber notificações através de uma outra ferramenta, a operação de subscrição deve ocorrer novamente, indicando agora a ferramenta no qual ele deseja receber as notificações. Para executar qualquer uma das ferramentas fornecidas pelo CASSandra, basta executar a ferramenta da seguinte maneira:

```
>cd cassius/cassandra
java <nome_da_ferramenta> <UserName> <URL do servidor>
Ex.:
java simpleScroller Pasqueline http://localhost:8080/cassius/cass
```

5.5 O cenário da simulação

Depois que o processo de comunicação do CVS com o servidor e este para com os consumidores foi completado, um cenário de simulação foi montado para validar a solução. Durante a simulação, um computador foi utilizado fazendo este o papel de servidor e máquina-cliente da aplicação.

Do lado servidor as seguintes aplicações estavam executando: um servidor web, o jakarta, para permitir a execução do CASSIUS, o MySQL, o CVS e o módulo Escutador, compondo assim o CVSyn. No lado cliente o CVS e duas ferramentas colaborativas.

A simulação aconteceu da seguinte maneira: várias atualizações foram realizadas no repositório do CVS o que refletia várias operações *commit*. A cada atualização um processo era ativado para chamar o módulo escutador. A intenção de tantas atualizações consecutivas era validar se alguma notificação estava sendo ignorada ou sendo enviada mais de uma vez.

Ao realizar quatro operações de *commit* sucessivamente, o que se pôde perceber é que o tempo em que as notificações chegavam às ferramentas colaborativas pertencia à ordem de segundos, mesmo que os processos tivessem que esperar que o objeto remoto liberasse o próximo ponto de verificação do arquivo CVSROOT/history.

Com a finalidade de verificar se um mesmo consumidor poderia receber as notificações a partir de diferentes ferramentas colaborativas, duas ferramentas foram utilizadas: o SwingTool, nossa ferramenta construída a partir do toolkit CASSandra, e o simpleScroller, fornecida juntamente com o CASSIUS. Quanto a este aspecto, as notificações foram enviadas corretamente.

Todos os requisitos não-funcionais foram atendidos. O que é relevante já que não seria interessante para o ambiente do projeto que a inclusão e/ou remoção de fontes e consumidores causasse impacto na ferramenta. Sob o ponto de vista do lado cliente da aplicação, pôde-se observar que a responsabilidade de notificar outros colaboradores foi transferida para a ferramenta.

5.6 Considerações Finais

Neste capítulo alguns aspectos foram abordados: o primeiro deles se refere aos aspectos de validação da solução. Tomando-se os requisitos funcionais e não-funcionais apresentados no Capítulo 4, a implementação da arquitetura e dos modelos construídos comprovou a viabilidade e o desempenho da solução.

Com relação a falhas, a ferramenta mostrou-se bastante eficiente dado que o maior problema encontrado seria o gerenciamento de algumas informações. A utilização da tecnologia RMI conferiu maior confiabilidade à aplicação, evitando que a redundância de informações ou falhas no envio destas ao servidor acontecessem.

Levando em consideração o momento em que uma ocorrência é gerada no repositório e uma notificação é enviada aos consumidores por meio das ferramentas colaborativas, observou-se o consumo de tempo: cerca de 1 segundo.

Um outro aspecto observado refere-se ao desempenho do servidor de notificação de eventos em atender as necessidades do ambiente. O CASSIUS é uma solução poderosa e de muita utilidade quando há necessidade de notificar consumidores que não são máquinas. Contudo, considerando este contexto, o servidor ainda oferece um fraco suporte no que se refere à etapa de configuração das fontes e dos consumidores, isto porque a forma com que a interação é oferecida – dado que a interface não é amigável – conduz o

colaborador responsável pela configuração do CVSyn a cometer um número maior de erros.

Capítulo 6

Conclusões e Trabalhos Futuros

Neste capítulo, as contribuições oferecidas como resultado deste trabalho são apresentadas. A importância do CVSyn, bem como, o impacto da mesma para a comunidade que produz software também são discutidas. Ainda neste capítulo, sugestões para trabalhos futuros são indicadas como forma de melhorar este trabalho e motivar novas pesquisas.

6.1 Contribuições

Nesta dissertação, uma ferramenta chamada CVSyn foi especificada e implementada como auxílio ao Desenvolvimento Global de Software. As dificuldades encontradas por algumas empresas de menor porte, no que se refere ao apoio ferramental, motivaram o desenvolvimento de uma solução capaz de fornecer suporte a alguns aspectos críticos de uma abordagem de desenvolvimento distribuída, dentre eles o compartilhamento de informações através de notificações.

No decorrer do trabalho, a importância do suporte tecnológico foi evidenciada nas etapas de construção da aplicação. Apresentou-se também que, primariamente, as dificuldades surgem porque a comunicação é afetada pela separação geográfica e que muitos conflitos são gerados sobretudo, nas etapas que circundam o momento de integração das partes, principalmente quando ferramentas colaborativas não são utilizadas como suporte à comunicação do projeto.

Neste contexto, as soluções que prezam por aspectos que contemplam a necessidade de notificar ocorrências são pouco encontradas na forma de ferramentas de código-livre. Observa-se que mesmo as ferramentas de gerenciamento da configuração, tão úteis no suporte ao desenvolvimento, possuem mecanismos pouco eficientes para ajudar a identificação de situações que podem causar re-trabalho para os desenvolvedores.

Dos resultados obtidos com este trabalho destaca-se um incremento com relação às funcionalidades já oferecidas pelo CVS, cuja aceitação é ampla no âmbito da comunidade produtora de software. A solução oferecida acoplou ao CVS um mecanismo de notificação síncrono – um melhoramento claramente necessário, sobretudo, quando a utilização de uma ferramenta de gerenciamento da configuração é contextualizada em um cenário como o DGS.

A integração das diversas informações envolvidas no projeto, a partir da ferramenta de controle de versão, possibilita benefícios práticos importantes:

i. redução de conflitos nos momentos que antecedem à integração, principalmente nos casos onde existem diferenças de fusos-horários e os colaboradores cooperam entre si compartilhando a construção dos artefatos;

ii. redução das taxas de re-trabalho;

iii. redução de custos com comunicação síncrona, telefonemas e videoconferência;

iv. redução de custos de aquisição de soluções proprietárias. Não há necessidade de aquisição de um ambiente colaborativo já que a infra-estrutura para tal está sendo oferecida;

v. redução das curvas de aprendizagem, uma vez que os desenvolvedores poderão continuar fazendo uso dos mesmos ambientes de apoio;

vi. desacoplamento entre provedores e consumidores de informação, originando uma conduta de gestão mais eficiente;

vii. notificações em escala de Internet a partir de diversas ferramentas.

Atualmente o CVSyn é adequado para cenários nos quais grupos de desenvolvedores, distribuídos fisicamente, precisam compartilhar as alterações nos artefatos assim que estas ocorrem.

Pelo fato de um servidor de notificação ter sido utilizado na solução, o que gera ganhos consideráveis do ponto de vista do encaminhamento das mensagens, outros contextos podem ser beneficiados com o mecanismo de notificação já que a necessidade de propagar informações não é um problema específico da produção de software.

Os resultados obtidos com a revisão bibliográfica também agregam valor ao trabalho, mapeando e documentando particularidades e problemas que desafiavam o cumprimento dos objetivos traçados para o projeto. Adicionalmente, uma outra

contribuição é o fato da concepção do CVSyn estar baseada em software livre, a exemplo da tecnologia de servidores de notificação de eventos sugerida.

6.2 Limitações do CVSyn

Apesar do CVSyn ser uma ferramenta útil para o processo de construção da aplicação, é necessário mencionar suas limitações:

- CVSyn, como o próprio nome sugere, é uma solução fundamentada em um sistema de controle de versão específico. Isto não significa dizer que outros sistemas de versão não poderão ser incluídos como parte da solução já o projeto do CVSyn foi realizado para prover flexibilidade no que se refere a ferramenta de controle de versão escolhida. No entanto, é necessário implementar a estratégia de monitoramento dos eventos. O CVSyn não permite que a troca entre sistemas de versão aconteça de maneira automática, ele fornece apenas uma solução finalizada aqueles interessados em trabalhar com o CVS;
- a configuração das fontes de informação, bem como a definição dos objetos e dos eventos, junto ao servidor de notificação não pode ser realizada pela interface do CVSyn que é composta apenas por uma ferramenta capaz de receber notificações. Ainda é necessário acoplar ao CVSyn outras ferramentas que permitam a visualização das notificações através de dispositivos móveis como telefones móveis, por exemplo;
- para que os consumidores recebam as informações que permitem o entendimento do que está acontecendo no repositório de artefatos, há uma dependência com relação à forma com que os objetos e eventos são definidos junto ao servidor de notificação, ou seja, a eficiência com que os colaboradores compreendem as informações que estão sendo compartilhadas depende daquele que configura tais variantes junto ao servidor.

6.3 Trabalhos futuros

Alguns aspectos não contemplados no escopo deste trabalho demandam outros esforços no sentido de otimizar o CVSyn. A primeira sugestão é o melhoramento da

interface do CVSyn para permitir que o cadastro das fontes, objetos e consumidores de informação possa ser realizado através da interface desta ferramenta. Não foi possível atender este requisito nesta versão da ferramenta.

O CVSyn consegue atender às demandas em se tratando de contextos em que a ferramenta de controle de versão adotada é o CVS. Dependendo do cenário em questão, pode haver necessidade de múltiplos repositórios de artefatos, sendo que estes não necessariamente precisam ser do mesmo tipo. Um trabalho futuro poderia ser orientado à construção de um *framework* que possibilite o monitoramento de outras ferramentas de controle de versão.

Atualmente, a solução permite que a estratégia de monitoramento dos eventos possa ser modificada, contudo, é necessário que o cliente implemente a estratégia de escuta. Um trabalho importante seria na direção de generalizar a tarefa de monitorar os eventos, independente da ferramenta de controle de versão em questão. Este fator retiraria do cliente a responsabilidade de qualquer implementação.

Também é possível extrapolar a utilização de um mecanismo de notificação para outros cenários. É importante considerar a possibilidade de que regras de negócio possam ser definidas, permitindo assim que a ocorrência das notificações automatize determinados aspectos do processamento.

Para horizontalizar a solução é necessário melhorar o formato das notificações que são propagadas, atualmente, apenas na forma de mensagens de texto, cujos formatos são pré-determinados pelo servidor de notificação. É interessante definir o formato da notificação de acordo com o cenário considerado, já que em algumas situações poderia ser útil notificar os consumidores utilizando outros meios além de texto, como vídeo ou áudio.

A utilização do ambiente em um projeto real que envolva usuários distribuídos, os quais possuem diferentes idiomas, poderia ser de grande valia para o melhoramento da solução.

O CVSyn é uma ferramenta adequada para propagar informações. Embora projetada para isso, diversos componentes poderiam ser acoplados para que outros aspectos referentes ao ambiente do projeto pudessem ser auxiliados, a exemplo da análise de métricas e riscos, mediante as informações que chegam ao servidor e podem ser transformadas em conhecimento para a gerência do projeto.

A inclusão de outras ferramentas para visualização das notificações que contemplem a mobilidade dos colaboradores é um outro aspecto a ser visto em trabalhos futuros, ampliando a abrangência da validação da solução.

Anexo A - A IMPORTÂNCIA DE TECNOLOGIA COLABORATIVA NO CONTEXTO DE DESENVOLVIMENTO

A.1. Formação de Parcerias

Ao atuar globalmente, muitas organizações se percebem incapazes de realizar projetos em outros domínios ou alocar recursos extras, como tempo ou novas pessoas. Ao atuar em projetos de grande porte, é comum o envolvimento de um maior número de empresas no desenvolvimento, tendo uma organização como líder do projeto e unidades virtuais representam outras organizações com direitos iguais (KÖTTING,2001).

Ao optar pelo desenvolvimento baseado em parceiras, a dificuldade em encontrar as pessoas certas, responsáveis por desenvolver um conjunto de tarefas ou até mesmo, gerenciar a colaboração dos times pode comprometer o sucesso do projeto. A etapa de negociação, nem sempre uma atividade onde os parceiros podem estar presentes fisicamente, envolve interesses diversos incluindo desde a forma de gestão da propriedade intelectual, o cumprimento de prazos, até o modelo de participação nos lucros do projeto. Na maioria das vezes, os custos com viagens são altos e os parceiros podem possuir compromissos em períodos complementares.

A.2. Modelo de desenvolvimento

Na instalação de um novo projeto é comum a sua segmentação em partes, de forma que as várias atividades sejam realizadas por empresas (ou grupos) diferentes. A negociação de aspectos dos projetos, muitas vezes, requer mecanismos automatizados que possam reduzir a conflitos gerados durante as negociações, assim como, permitir a reusabilidade de contratos (Statement of Work), de forma que haja maior agilidade quanto à atribuição das responsabilidades de cada parte.

As organizações também precisam fazer uso de soluções que confiram mais segurança no momento em que optam por realizar *outsourcing*[22], de forma que possam balancear a carga de trabalho (*workload,* concentrado anteriormente em apenas uma organização); estruturar as informações das tarefas permitindo que os times decidam mais rápido se podem realmente realizar tais atividades; e se concentrar nas competências

[22] Outsourcing acontece quando uma organização contrata uma outra empresa para realizar partes do projeto. Este processo também é conhecido como terceirização.

principais (*core competencies*) das parceiras, fundamentalmente, permitindo escalonar os times certos para as tarefas certas.

A.3. Análise do Problema

Um aspecto marcante no Desenvolvimento Global de Software é o alto nível de mudanças de requisitos. Devido ao dinamismo do mercado que receberá o produto de software, os colaboradores devem ter acesso às informações sempre que alguma mudança seja requisitada pelo cliente ou pelo próprio mercado.

A grande contribuição desta fase para a construção do produto é o modelo arquitetural. No entanto, sabendo da possibilidade dos requisitos serem alterados com uma freqüência bem maior que uma aplicação destinada a um único mercado, é imprescindível que haja um local de acesso central onde os colaboradores possam acessar as informações, assim como, serem notificados de qualquer mudança.

É importante que outras informações também sejam compartilhadas. A disponibilização de métricas adequadas permite que a evolução do trabalho seja acompanhada. A divulgação dos resultados dos testes de aceitação estimula que valores sejam mantidos entre os times, como coesão, confiança e o espírito de equipe. É através de suporte tecnológico que informações técnicas e de propósito mais específico aproximam os times de um ambiente de trabalho mais integrado e produtivo.

A.4. Decisões de projeto

Uma vez estabelecida a arquitetura inicial da aplicação, o próximo passo é decidir como o trabalho será alocado entre os times. A escolha da estratégia de escalonamento dependerá, principalmente, do arranjo de negócios feito entre as empresas e da localização das competências necessárias (DANTAS,2003).

A. 4.1 Formação dos times

Existindo ou não a criação de parcerias entre organizações distintas, um projeto pode demandar a alocação de especialistas e técnicos que não estão geograficamente próximos. Sendo assim, um encontro *face-to-face* nem sempre é possível. A formação dos times virtuais é um aspecto que impõe desafios devido à ampla e global disponibilidade de

técnicos. O gerente do projeto ou mediador deve ser cuidadoso em escolher o perfil adequado da equipe de forma que se evite desentendimentos gerados pela divergência entre fusos-horários, *background* técnico, culturas e idiomas (CARMEL,2001).

Nesta etapa, a utilização de ferramentas de comunicação e de suporte ao processo de desenvolvimento são de extrema importância para manter o espírito de equipe entre os membros dos times e auxiliar o gerente de projeto na tomada de decisões, respectivamente.

A.4.2 Alocação de recursos

De posse do modelo arquitetural, os componentes da aplicação são definidos e as atividades podem ser alocadas aos times. Essa tarefa, além da alocação de recursos físicos e tecnológicos, é complexa dada a grande quantidade de pessoas que estão envolvidas no projeto.

Sendo assim, um gerente de projeto ou mediador precisa fazer uso de ferramentas para que estas auxiliem à tomada de decisão e permitam a reutilização do conhecimento adquirido de projetos anteriores, a exemplo de condutas técnicas, perfil dos times, sugestões de utilização das tecnologias mais adequadas ao domínio do problema, ou até mesmo, plataformas de desenvolvimento.

No que se refere ao enfoque gerencial, os ambientes de suporte ao processo são essenciais para permitir que os gerentes atinjam o nível de coordenação desejado para que tanto o processo adotado quanto o projeto alcancem seus objetivos. O processo de tomada de decisão é o mais importante para o gerente de projeto que necessita que o maior número de informações possam estar presentes e sempre atualizadas para que as decisões sejam eficientes.

A.5. Produção de código e integração

Assim que as tarefas são atribuídas aos times, aqueles encarregados pela codificação precisam seguir uma rotina de tarefas para assegurar a qualidade do código e reduzir a taxa de re-trabalho.

Após a codificação, necessariamente, os times devem testar os componentes que foram produzidos e integrá-los continuamente a um repositório de artefatos de acesso global. Contudo, quando múltiplos times compõem o ambiente de desenvolvimento, muitos conflitos desafiam o cumprimento dos objetivos, assim como, a gerência do

processo já que atividades cotidianas como o controle de versões, sincronização dos artefatos e notificação de eventos, tornam-se de alta complexidade. O nível de colaboração nesta etapa é muito grande e a ausência de ambientes colaborativos tornam tais atividades impraticáveis.

A.5.1 A gestão da configuração de artefatos

Durante a fase de produção de código, a utilização de ambientes de desenvolvimento é mais freqüente já que envolve várias atividades para que um ***release*** seja obtido. A automação de tarefas cotidianas como integração e acesso aos componentes, geração de documentação, implantação de serviços Web, execução da bateria de testes e o *deployment* da aplicação, garante rapidez e confiabilidade ao desenvolvimento.

O gerenciamento de configuração é definido por (BABICH,1986) como sendo a arte de identificar, organizar e controlar as modificações que são realizadas por uma equipe de desenvolvimento. Em particular, a gestão de artefatos possui um papel muito importante para as etapas de construção e integração de código. Em um projeto distribuído a eficiência dessa atividade se torna de alto impacto para o projeto.

Sua responsabilidade é controlar a construção das partes individuais do sistema em meio às alterações nelas sofridas, acompanhar a situação da implementação da aplicação e verificar a conformidade dos requisitos especificados no projeto. Claramente se não realizada bem, esta atividade se torna um obstáculo frustrante para o atingir os objetivos do projeto. Segundo Bennatan (BENNATAN,2002), esta é uma das áreas principais onde um projeto distribuído pode fracassar.

O gerenciamento de configuração se torna um desafio na medida em que as mudanças devem ser aprovadas e notificadas aos times. Em se tratando de desenvolvimento distribuído, gerenciar as alterações que ocorrem no código e nos artefatos se torna uma atividade complexa porque acentua os problemas provenientes de atualizações simultâneas e notificação das mudanças.

Um ambiente de gestão de configuração oferece várias funcionalidades (ASKLUND,2002), dentre elas:

i. controle das versões: que possibilita armazenar diferentes versões de um artefato para recuperação posterior;

ii. gerenciamento de concorrência: gerenciando o acesso múltiplo de vários usuários ao repositório, seja evitando ou suportando o acesso concorrente;

iii. gerenciamento de espaços de trabalho (*workspaces*): evitando que o trabalho dos desenvolvedores seja afetado pelas alterações dos outros desenvolvedores;

iv. documentação das mudanças: que mantém históricos de requisições, alterações e conflitos que ocorrem durante a construção dos artefatos.

Entretanto, alguns aspectos devem estar presentes na ferramenta de gestão de configuração utilizada no projeto. Um fator importante é que apenas um repositório central exista para que as sincronizações ocorram apenas nos espaços de trabalho dos desenvolvedores.

É interessante que a ferramenta seja fácil de usar e administrar, e não adote um modelo transacional rígido para coordenar as alterações. Um modelo transacional indica a política de atualização dos artefatos, sendo responsável por definir, por exemplo, que todos os arquivos serão cometidos ao repositório com sucesso ou nenhum o será quando os espaços de trabalho não estiverem sincronizados com o repositório.

Uma ferramenta que suporta este modelo e facilita a atualização dos espaços de trabalho, incluindo a recuperação dos arquivos que precisam ser atualizados, é ideal para o desenvolvimento distribuído, especialmente quando os clientes estão off-line a maior parte do tempo. Uma outra propriedade desejável nas ferramentas é que esta forneça informações sobre o que está acontecendo no projeto. Esta é uma das grandes lacunas existentes nos sistemas de gestão de configuração (ASKLUND,2002): mecanismos efetivos de notificação.

A. 5.2 A integração de código

Em um contexto de desenvolvimento distribuído, a rotina de trabalho dos times sofre sensíveis mudanças. Todos os artefatos devem ser remetidos a um repositório central que é acessado constantemente por outros times. A integração pode demandar tempo porque, a depender da robustez da ferramenta que esteja sendo utilizada, ou os componentes acessados do repositório são bloqueados ou por outro lado, quando cópias do componente são requisitadas existe a possibilidade de que muito tempo seja gasto na resolução de conflitos.

Os conflitos mais comuns acontecem no período de preparação para a integração dos componentes, assim como, no momento posterior à integração. Antes de integrar os problemas recaem no fato de que mudanças de requisitos muitas vezes não são notificadas a todos os desenvolvedores. Embora definindo a arquitetura base do sistema e as interfaces de comunicação entre os componentes de forma centralizada, como sugere (CARMEL,2001), os desenvolvedores ao realizarem suas atividades possuem autonomia para criar ou modificar classes, bibliotecas ou componentes que estão sob sua responsabilidade ou do qual fazem uso.

O impacto das alterações no trabalho de outros desenvolvedores muitas vezes é desconhecido em um primeiro momento. A ciência das alterações apenas é alcançada quando é necessário fazer uso do artefato em questão, e as alterações são então, sinalizadas pelo sistema de controle de versões. No entanto, a eficiência das notificações depende da ferramenta que está sendo utilizada. Algumas soluções, como é o caso do CVS, não são capazes de notificar de forma síncrona as alterações realizadas nos artefatos.

Por este motivo, durante a integração dos componentes e posteriormente, nos testes de integração, os problemas surgem justamente porque as "inconsistências" não foram detectadas durante a implementação. Sendo assim, evitar que tais conflitos ocorram e principalmente, identificá-los quando estes acontecem não é uma tarefa simples em se tratando de sistemas complexos.

Este fato se torna ainda mais crítico quando determinadas práticas regem o processo de desenvolvimento. A integração contínua, por exemplo, sugerida pelo processo eXtreme Programming (BECK,2000), é fundamental para permitir a evolução incremental da aplicação e evitar que altos custos sejam necessários para a montagem do sistema (compilação, configuração, empacotamento, testes).

Contudo, sua eficácia depende de uma ferramenta adequada para auxiliar a detecção de conflitos, que em muitas das vezes é realizada "manualmente" pelos desenvolvedores. Isto é, a capacidade de permitir que os membros de um time colaborem é um aspecto a ser considerado nas soluções.

A.6. Inspeções virtuais

Em qualquer modalidade de desenvolvimento, é necessário revisar os artefatos que estão sendo produzidos pelos times durante o ciclo de vida do produto. Documentos de requisitos, do projeto e a documentação na forma de textos e imagens precisam ser

constantemente revistos para que se possam detectar falhas, manter a qualidade do código que está sendo gerado e possibilitar a geração de métricas do projeto. Em tais inspeções, de alguma forma é necessário o encontro presencial de alguns colaboradores, *pair programming* é um exemplo típico de estratégia de inspeção.

Em projetos distribuídos tais encontros são muitas vezes inviáveis e o fato de que os times utilizam ferramentas e processos diferentes podem dificultar a análise global de tais artefatos. Para esta atividade, é importante um auxílio ferramental, de forma que os revisores possam utilizar padrões e templates que permitirão a captura (através de um repositório de artefatos, por exemplo) e representação das informações produzidas por cada time colaborador.

A.7. Construção de releases

Em projetos com vários times distribuídos, após a integração e a realização dos testes de cada funcionalidade separadamente, os gerentes de teste e os mediadores precisam ainda integrar as funcionalidades produzidas e testar o funcionamento do sistema por completo. Para a gerência, a conexão das informações obtidas a partir dos testes e do resultado das integrações permite muito mais que a simples automação dos processos de geração de releases, permite o acompanhamento da evolução de cada time para o projeto.

Por mais que os desenvolvedores façam uso de ferramentas que automatizem determinadas atividades, a principal contribuição da utilização de ambientes colaborativos integrados é a disponibilização de informações que conduzam a um controle maior dos times. A gerência do processo precisa estar capacitada tecnologicamente para acompanhar os times à distância.

Anexo B – ALGUMAS FERRAMENTAS DE CÓDIGO-LIVRE: Eclipse, JUnit, Ant, CVS e AntHill

B.1 Eclipse

O Eclipse[23] é um IDE (Integrated Development Environment) para construção de código. A plataforma suporta a construção de uma variedade de ferramentas para o desenvolvimento das aplicações; suporta ferramentas capazes de manipular categorias arbitrárias de conteúdo, como HTML, Java, JSP, EJB, XM; facilita a integração de ferramentas; além de rodar em vários sistemas operacionais, incluindo Windows e Linux. Embora o Eclipse tenha sido mencionado, nada impede que outras soluções sejam utilizadas no projeto.

B.2 JUnit

O Junit[24] é um framework que facilita o desenvolvimento e a execução dos testes de unidade em código Java. O framework possui uma API para a construção dos testes e fornece uma aplicação, o TestRunner, para execução dos testes. O JUnit é capaz de verificar se cada método de uma classe funciona da forma esperada, permitindo executar vários testes ao mesmo tempo. Além disso, também permite que cenários de falhas sejam testados. O DBUnit e o JunitEE são extensões do JUnit para testar aplicações JDBC e J2EE, respectivamente.

Assim como em projetos co-localizados, os testes de unidade são muito importantes porque ao integrar o processo de testar com o desenvolvimento, o progresso pode ser medido. Além disso, os testes de unidade serão úteis futuramente para os testes de regressão da aplicação (este tipo de teste valida as partes que foram modificadas do software, garantindo que nenhum novo erro foi introduzido no código que havia sido testado previamente).

[23] http://www.eclipse.org/articles/index.html
[24] http://www.junit.org

B.3 Ant

O Ant[25] é um framework para a construção e o *deployment* de aplicações Java. O Ant é capaz de montar praticamente qualquer aplicação mesmo que esta esteja distribuída em pacotes, que requeira a definição de *classepaths* locais e precise vincular código a bibliotecas (.jars) ou cuja criação ou vinculação dependam de mais que uma chamada ao javac, como aplicações que utilizam RMI, CORBA, JSP, EJB.

O framework também auxilia na automação de tarefas freqüentes tais como: gerar Javadoc, implantar serviços Web e J2EE, acessar um repositório de artefatos tipo CVS, empacotar e comprimir (.jar, .zip), executar programas (java, ant, sql, etc), testar unidades de código, montar componentes, expandir, copiar e instalar, além de outras tarefas (ver figura abaixo).

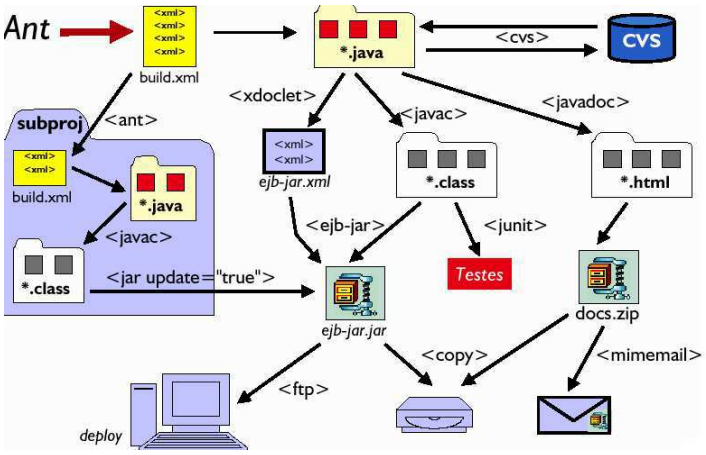


Figura 15: Funcionamento do ANT

O Ant executa roteiros escritos em XML, chamados *buildfiles*. Cada projeto do Ant possui um roteiro. Subprojetos podem ter, opcionalmente, *buildfiles* adicionais que são chamados durante a execução do primeiro. Cada projeto possui uma coleção de alvos e cada alvo consiste de uma seqüência de tarefas.

[25] http://jakarta.apache.org/ant

```
<project name="example" default="compile">

  <property name="source" location="src" />
  <property name="output" location="classes" />

  <target name="init">
    <mkdir dir="${output}" />
  </target>

  <target name="compile" depends="init">
    <javac srcdir="${source}" destdir="${output}" />
  </target>

</project>
```

Figura 16: Exemplo de um buildfile

A importância de utilizar o framework é que ele oferece muito mais recursos que qualquer comando *build* dos IDEs hoje existentes. Além disso, o Ant provoca vários eventos que podem ser capturados por outras aplicações, o que é útil para implementar a integração.

B.4 CVS (Concurrent Version System)

O CVS[26] é um sistema de controle de versões de código-livre que se baseia em um repositório central onde usuários podem fazer atualizações e recuperações dos artefatos do projeto, através de comandos *commit* e *checkout,* respectivamente. O repositório CVS mantém uma cópia de todos os arquivos e diretórios sob controle de versões e guarda um histórico das modificações que ocorrem. Também permite a recuperação de quaisquer versões anteriormente armazenadas de arquivos texto.

Cada desenvolvedor cria seu diretório de trabalho que contém uma cópia local dos arquivos baixados do repositório (através de um *checkout*). Quando o desenvolvedor deseja trabalhar com um código é necessário "baixar" a última versão do repositório. Assim pode manipular o artefato localmente no seu diretório de trabalho protegendo-o, assim, de ser manipulado por outros desenvolvedores naquele momento. Assim que terminar a atualização do artefato o desenvolvedor o atualiza no repositório com uma espécie de upload (*commit*) e as alterações são mescladas em uma nova revisão.

Contudo, atualizar o repositório com a nova versão apenas é possível se não houver conflitos. Um conflito ocorre quando dois usuários alteram a mesma área do código. O

[26] http://www.cvshome.org

primeiro que fizer o *commit* grava as alterações. O outro usuário apenas pode cometer suas mudanças depois que atualizar sua cópia de trabalho e resolver o conflito (Figura 2).

Em projetos distribuídos tais conflitos são mais comuns e sua resolução torna-se mais complicada, principalmente, se os desenvolvedores demorarem algum tempo para acessar o repositório ou estiverem em paises que compartilham fusos-horários diferentes.

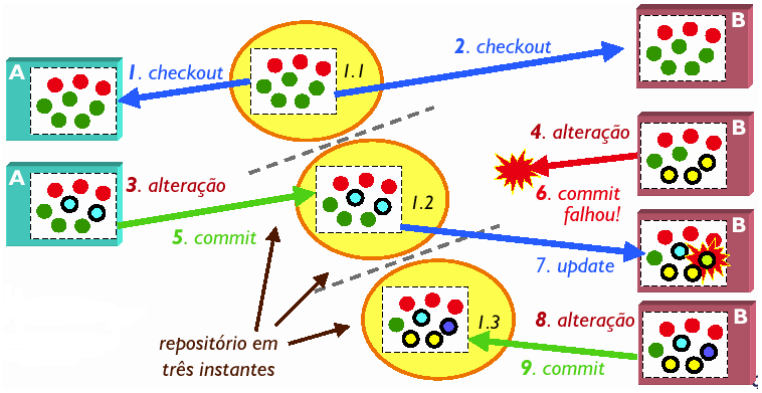


Figura 17: Cenário de conflito no CVS

Quando o CVS está sendo utilizado no projeto, o ciclo de desenvolvimento pode ser resumido como se segue :

1. O desenvolvedor inicia o ambiente através do *checkout* do código para o seu diretório de trabalho;
2. O ciclo geralmente envolve alterações no código-fonte, criação de novos arquivos e sua adição no repositório CVS, bem como a remoção;
3. Localmente, após a alteração, o desenvolvedor compila, monta e executa os testes;
4. Em seguida fazer um *update* para incorporar eventuais mudanças feitas por outros desenvolvedores;
5. Se um conflito for detectado, este deve ser resolvido. Nova compilação e testes são refeitos;
6. O último passo é cometer todas as mudanças através de um *commit*.

B.5 AntHill

O AntHill[27] é uma aplicação Web que fornece suporte à integração contínua e automática. É ideal para integrar software desenvolvido em equipe. É baseada na ferramenta Ant, através da qual opera um sistema de controle de versões – o CVS.

A aplicação roda em um servidor onde periodicamente: monta toda a aplicação, roda todos os testes e gera o *build*. Cada projeto está associado a um agendamento. No momento de realizar o build, o AntHill consulta o repositório. Se houver novos arquivos, o build ocorre. Em caso de sucesso o arquivo contendo o número da versão é incrementado, em caso de falha, os participantes do projeto são notificados via e-mail. Relatórios são gerados sobre o sucesso ou falha na geração do build.

[27] http:// www.urbancode.com/projects/anthill

Anexo C - API do CASSandra

C. 1 Criação de novos objetos

Tabela 3: Criação de novos objetos

Parâmetro	Descrição
Operation	NewObject
Account	Account path; either AccountClass.AccountName (where AccountClass can be any of the account's classes), or if there is no class then just AccountName
Password	Password for accessing this account
ObjectName	Name of the object to create. Used to enable the server to present readable lists of objects to the user. This supports the goal of allowing users to browse for information that they may subscribe to.
ObjectID	A unique identifier that remains constant even if the name of the object should change.
ParentID	*Optional* A unique identifier that informs the notification server of the object's parent or enclosing object. If not used or if used to pass an empty value, then no parent object exists. Objects with no parent are the highest level objects in the hierarchy.
Position	*Optional* In cases where there are many objects contained by a parent object, the information source may feel that the order of these objects is important, in which case it will specify each object's position in the list of objects that are contained by the parent. If Position is not used, then the new object will be the last object in the list. Position=0 places the object as the first item in the list. An example of where this is useful is for representing the sections of a document. The order of the sections communicates their meaning, if Introduction comes after conclusion, users may assume that it is not the kind of introduction that they are looking for.
Type	*Optional* type name. Type specifies a set of event names for the object, if left unspecified, object uses generic events only.
Description	*Optional* description of the object's purpose and meaning.

Respostas

Tabela 4: Remoção de objetos

Parâmetro	Descrição
Operation	RemoveObject
Account	Account path; either AccountClass.AccountName (where AccountClass can be any of the account's classes), or if there is no class then just AccountName
Password	Password for accessing this account
ObjectID	Identify the object to be deleted using its unique identifier.

Respostas

Respostas

Password	Password for accessing this account
Type	Name of the type being defined
Event	Name of the event which objects of this type may receive
Definition	Description of the event.

Respostas

Respostas

	specify which Type-Specific EventName describing what occurred to this object Note that there is currently no type and event checking. You can send a notification of changes to a File with the event field set to *"I am a goose"*, and it will be accepted. The only side-effect of this is that if *"I am a goose"* is not an event defined for File objects, no one will know to subscribe to it.
GenericEvent	Even if Event is used, always specify one of the GenericEvents so that applications that are either to0:Account {TestClass}.Test Registered simple to analyze the Events and type definitions or come designed with knowledge of only one set of types can still have a limited understanding of what happened to the object. The current list of GenericEvents is: 1. **Active**: Light is on, application is now running, object has just been created, etc... 2. **Inactive**: Light is off, process or information source is terminating, object has been deleted, etc... 3. **Increase**: Data has been added to the object, value has increased, etc... 4. **Decrease**: Data has been removed from the object, value has decreased, etc... 5. **Change**: Complex changes took place that can be represented with increases and decreases.
Person	*Optional* Specifies who made the change. An important part of awareness is knowing who is doing what. This provides the who part.
Place	*Optional* Specifies where the change were made. It may be an office, a building, a city, some identifying information of use to the users of the information source. Being aware of one's environment can involve knowing where things are happening.
Summary	A textual summary of changes, designed to fit on a single line, such as an email subject line, a string of text that can scroll, a headline to an article, details that may be revealed when the user points to a representation of an event, etc...
Value	A numerical representation of the event, quantifying it in some manner, and usable by awareness tools that deal in numbers, such as graphing tools, tools that utilize intensity (through use of sound, light, vibration, disruption, etc...) to communicate the intensity of the event.
Data	*Optional & Currently Unimplemented* Contains a file that represents the details of the change. This file can be of any type. It can be a text file, an HTML file, a gif, whatever the application chooses to use to communicate changes. Note that the choice of format must use something that the client awareness tools will know how to interpret. Aan illustrator file for example would restrict the types of people who can monitor changes, and eliminates any likelihood that the pager, tickertape, or other UI will be able to directly display it. However, that is why there are other fields -- so that each awareness tool can find information that they can best work with. Two suggestions on how this data field might be used: 1. Representation of change. A gif file, viewable by any awareness

	tool that bothers supporting gifs can be used to present a before and after snapshot of how a document, system, architecture, etc... looked before and after the changes were made. 2. Systems like portholes which are based upon pictorial information can include the photo as part of its notification.
URL	*Optional* Rather than tying up bandwidth by sending files with the details of the change to the CASSIUS server (currently implemented), one can include a URL to the file containing the details of the change. This URL can be used by awareness tools to either automatically download further information for its representation of the notification, or can be used if the user requests further information. This is optional, but potentially quite valuable to users. News headline on a scrolling tickertape or a page listing news headlines isn't enough information: users need to be able to click on it to see the full article.
FileType	*Optional, not yet implemented* Provide an extension to specify the document type ("txt", "html","doc"...) of the Data field.

Respostas

	which would require that each have a different interface name.
Since	*Optional* Check for all new notifications to arrive since this time in miliseconds after the standard 1970s date used by unix. If not specified, notification server uses the last time this user/interface polled (something it needs to track). When a tool quits, and is restarted 24 hours later, it may not want every event to happen in the last 24 hours (i.e. every light switch, printer error, etc...), and may instead want the last 10 minutes only. Alternately, it may want to get the last week's worth of events to so that its style of awareness and visualization will have some context to work with.
Include	*Optional, not implemented* This parameter is used to determine when the Data field of the notification should be returned with the notification. The format is a comma separated list of file types, including the '*' wildcard. Any file type that matches the types in the Include list is sent with the notification.

Referências Bibliográficas

(AOYAMA,1998) Aoyama, M. *Web-based Agile Software Development*, IEEE Software, Vol. 15, No. 6, November-December 1998.

(ARAUJO,2003) Araújo, E. E. R., A. *Internacionalização e a Localização de Produtos e Serviços: a sua importância na indústria de software*. Revista T&C Amazônia, Ano 1, nº 2, p. 44-48, Junho de 2003.

(ASKLUND, 2002) Asklund U., Bendix L. *A Study of Configuration Management in Open Source Software Projects*. In IEE Proceedings - Software Engineering. Open Source Software Engineering: Special Issue of IEE Proceedings-Software, 2002.

(BABICH,1986) Babich, A. W. *Software Configuration Management - Coordination for Team Productivity*, Addison- Wesley Publishing Company, 1986.

(BATTIN *et al*, 2001) Battin, R.D., Crocker, R.; Kreidler, J., Subramanian, K. *Leveraging Resources in Global Software Development*, IEEE Software, Vol. 18, No. 2, March-April 2001.

(BECK,2000) Beck, K. *Extreme Programming Explained: Embrance Change*, Addison-Wesley, May 2000.

(BENNATAN, 2002) Bennatan, E.M. "*What is Happening to the Global Software Village? Is There Still a Case for Distributed Software Development?"* Agile Project Management Advisory Service, Cutter Consortium, Executive Report, Vol. 3, No. 1, 2002.

(BRERETON,2000) Brereton, P., Budgen. D. *Component-based Systems: A classification of Issues*, IEEE Computer, Vol. 33, No. 11, November 2000.

(CARMEL,2001) Carmel E., Agarwal R. *"Tactical Approaches for Alleviating Distance in Global Software Devel0opment"* IEEE Software, Vol. 18, Issue 2. March/April 2001.

(CARZANINGA,2001) Carzaniga, A., Rosenblum, D. and Wolf. *Design and Evaluation of a Wide-Area Notification Service.* ACM Transactions on Computer Systems, 19(3),332-383,2001

(COOLINS, 2002) Collins, R. W. *Software Localization for Internet Software: Issues and Methods*, IEEE, December 2001.

(COOK,2000) Cook, J. *Internet-based software Engineering Enables and Requires Event-based Management tools*. Proceedings of the 3rd workshop on Software Engineering over the Internet, At International Conference on Software Engineering, Limerick, Ireland, June 2000.

(CORONADO, 2001) Coronado, J., Livermore, C. *Going Global with the Product Design Process*, IEEE, December, 2001.

(DANTAS,2003) Dantas, V. F. *WideWorkWeb – Uma Metodologia para o Desenvolvimento de Aplicações Web num Cenário Global.* Tese de Dissertação de Mestrado submetida à COPIN, Universidade Federal de Campina Grande, Julho de 2003.

(DEITEL, 1999) Deitel, H. M. *Java: how to program*. 3rd .Prentice-Hall, 1999.

(EBERT,2001) Ebert, C., De Neve, P. *Surviving Global Software Development*, IEEE Software, Vol. 18, No. 2, March-April 2001.

(FITZPATRICK,1999) Fitzpatrick, G., Mansfield, T. *et al. Augmenting the workday world with Elvis*, In Proceedings of 6th European Conference on Computer Supported Cooperative Work, pp.431-450. Copenhagen, Denmark, Kluwer, September, 1999.

(FOWLER,2001) Fowler, M., Foemmel, M. *Continuous Integration*. Disponível em:http://www.martinfowler.com/articles/continuousIntegration.html

(GAMA *et al*,2000) Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Padrões de Projeto: Soluções Reutilizáveis de Software Orientado a Objetos*. Editora Bookman, Porto Alegre, RS, 2000.

(GAO,1999) Gao, J.Z., Chen, C., Toyoshima, Y., Leung, D.K. *Engineering on the Internet for Global Software Production*, IEEE Computer, Vol. 32, No. 5, May 1999.

(GRIBBONS, 1997) Gribbons, W.M. *Designing for the Global Community*. Professional Communication Conference, 1997. IPCC'97 Proceedings. Crossroads in Communication, 1997 IEEE International, 1997.

(HERBSLEB, 1999) Herbsleb, J. D., Grinter, R. E. *"Architectures, Coordination, and Distance: Conway's Law and Beyond"* IEEE Software, September/October 1999.

(HILBERT,2000) Hilbert, D., Redmiles, D. *Extracting Usability Information form User Interface Events*. ACM Computing Surveys, 32(4), December,2000.

(JAGIELSKI,1999) J. Jagielski, "*The Apache Success Story – Exploring an Open Source Development Process"*. Cutter Consortium, Out 1999. Disponível por:

http://www.newarchitectmag.com/archives/1999/10/jagielski - Último acesso: 10/10/2002

(KANTOR,2001) Kantor, M., Redmiles, D. *Creating an Infrastructure for Ubiquitous Awareness*, Eight IFIP TC 13 Conference on Human-Computer Interaction (INTERACT 2001Tokyo, Japan), July 2001b, pp. 431-438.

(KANTOR,2002) Kantor, M., Redmiles, D. *CASSIUS: Designing Dynamic Subscription and Awareness Services*. Disponível em: http://www.ics.uci.edu/~redmiles/publications/C042-KR02.pdf - Último acesso: 24/05/2003

(KAROLAK,1998) Karolak, D. W. *Global Software Development*. IEEE Computer Society Press, California, 1998.

(KÖTTING,2001) Kötting, B., Maurer, F. *Approaching Support for Internet-based Negotiation on Software Projects*. Enabling Technologies: Infrastructure for Collaborative Enterprises, 2001. WETICE 2001. Proceedings. Tenth IEEE International Workshops on, 2001, p. 25-30.

(LARMAN,2000) Larman, G. *Utilizando UML e padrões: uma introdução à análise e ao projeto orientado a objetos*. Editora Bookman, Porto Alegre, RS, 2000.

(LOPES, 2003) Lopes, P. G., Skarmeta, A. F. G. *ANTS Framework for Cooperative Work Environments*. IEEE Internet Computing, March 2003.

(LUONG,1995) Luong, T. V., Lok, J.S.H., Taylor D. J., Driscoll K. *Internalization-Developing Software for Global Markets*, New York: John Wilwy, 1995.

(MOCKUS,2000) Mockus, A., Fielding R. T., Herbsleb J. *A Case Study of Open Source Software Development: The Apache Server*, in Proceedings of the International Conference on Software Engineering, Limerick, Ireland, June 4-11, 2000.

(MOCKUS,2001) Mockus, A., Weiss, D. M. *Globalization by Chunking: A Quantitative Approach*, IEEE Software, Vol. 18, No. 2, March-April 2001.

(TJORTJIS *et al*, 2002) Tjortjis, C., Dafoulas G., Layzell P., Macaulay L.. *A Model for Selecting CSCW Technologies for Distributed Software Maintenance Teams in Virtual Organizations*. In Proceedings of the 26 th Annual International Computer Software and Applications Conference (COMPSAC´02). IEEE Internet Computing, 2002.

(SARMA,2002) Sarma, A., Hoek, A. *Palantír: Increasing Awareness in Distributed Software Development*. Disponível em:

http://citeseer.nj.nec.com/sarma02palantiacuter.html - Último acesso: 15/07/2003

(SOUZA,2002a) Souza, C. R. B., Basaveswara, S., Redmiles, D. R. *Supporting Global Software Development with Event Notification Servers*. In Proceedings of International Workshop on Global Software Development: Proceedings, ICSE'02, May, 2002 Orlando, Florida.

(SOUZA, 2002b) Souza, C. R. B., Basaveswara, S., Redmiles. *Using Event Notification Servers to Support Application Awareness*. In Proceedings of ISR (Institute Software Research) – NASA Ames Collaborative Software Engineering Tools Workshop, August, 2002.