RemotePP – um ambiente cooperativo para programação em pares distribuídos

Robson Machado Borges

UNIVERSIDADE DO VALE DO RIO DOS SINOS
CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS
CURSO DE INFORMÁTICA

RemotePP – um ambiente cooperativo para programação em pares distribuídos

Robson Machado Borges

Prof. D. Sc. Sérgio Crespo
Orientador

Monografia submetida como requisito parcial para a obtenção do título de Bacharel em Informática.

São Leopoldo, novembro de 2006.

Agradecimentos

Agradeço a todos que, de uma forma ou outra, contribuíram para que eu pudesse atingir este que é um dos marcos mais importantes de toda a minha vida.

Aos meus pais, que ensinaram o valor da responsabilidade como forma de alcançar objetivos pessoais e profissionais, de modo harmonioso e respeitoso.

Ao Gilson, que nunca me deixou desanimar nos momentos de dificuldades e incertezas, entendendo o valor que esta etapa representa na minha vida.

Aos professores e à coordenação do curso de Informática da Unisinos, que sempre manifestaram preocupação com o meu engrandecimento acadêmico e humano.

Aos colegas que, por vezes, agüentaram o meu gênio difícil e metódico nos vários momentos desta caminhada.

E, em especial, ao meu orientador, Prof. D. Sc. Sérgio Crespo, pelo acompanhamento e dedicação para a realização deste trabalho, que sintetiza os últimos cinco anos de luta e crescimento.

Resumo

A engenharia de *software* é uma área de conhecimento relativamente recente, por isso, ainda não possuímos uma forma idealmente definida para o desenvolvimento de *software*. Uma das linhas de estudo valoriza as metodologias ágeis, dentre as quais, a *Extreme Programming* (XP) alcança maior sucesso. Uma das práticas mais importantes da XP no sentido de alcançar qualidade e produtividade é a programação em pares. Em paralelo, vemos uma expansão na formação de grupos de trabalho dispersos geograficamente. Assim, este trabalho visa ao desenvolvimento de uma plataforma computacional que viabilize a programação em pares distribuída.

Palavras-chave: programação em pares, programação distribuída, ambiente colaborativo, engenharia de *software*.

Abstract

The software engineering is a knowledge area that is relatively recent and, because o this, we do not have an ideal structured way for the software development yet. One of the software study lines gives importance to the agile methodologies, among which the Extreme Programming (XP) is distinguished for reaching greater success. One of the most important practical features of XP related to reaching quality and productivity is the pair programming. Alongside, we see a spread in the formation of geographically dispersed team groups. Thus, this work aims the development of a software platform that makes possible the distributed pair programming.

Key words: pair programming, distributed programming, groupware, software engineering.

Sumário

Índice de figuras

Índice de tabelas

Introdução

O aumento na oferta de tecnologias de computação e comunicação de alta velocidade torna possível aos indivíduos a realização de boa parte das suas atividades de trabalho fora do escritório, seja durante viagens, em escritórios remotos ou em suas próprias casas. Conforme afirmam Stotts *et al* (2003), é cada vez maior o número de pessoas ou equipes que trabalham de forma geograficamente distribuída. Conseqüentemente, essa tendência traz alguns impactos para a indústria de desenvolvimento de *software*.

Como vantagens, podemos destacar o envolvimento de pessoas com diferentes perfis e culturas e que podem contribuir com experiências próprias. Ou seja, as organizações que fazem uso de times geograficamente dispersos podem obter proveito de talentos altamente qualificados, mas que não estariam dispostos a mudar de cidade ou até mesmo de país.

Por outro lado, desenvolver *software* é uma atividade altamente intelectual e que exige troca de conhecimento constante entre as pessoas envolvidas. Tal necessidade por interação é evidenciada por algumas práticas de desenvolvimento, como a programação em pares. Nesta modalidade, as atividades são desempenhadas sempre por grupos de duas pessoas que dividem a mesa e o computador, construindo, juntos, um único produto de trabalho. É necessário, ainda, que os pares sejam permutados a cada nova atividade, de forma a disseminar o conhecimento adquirido entre toda a equipe.

No entanto, algumas iniciativas tentam implementar a programação em pares de forma distribuída. Para isso, geralmente, faz-se uso de aplicações computacionais diversas com finalidades específicas que ofereçam alguma forma de comunicação. Os exemplos mais comuns de aplicações, segundo Hanks (2005) são: *chat*, mensagem instantânea, compartilhamento de *desktop*, *e-mail*, voz sobre IP, repositórios de arquivos e vídeo-conferência.

Sendo assim, este trabalho tem por objetivo a implementação de um ambiente computacional centralizado que viabilize a programação em pares geograficamente distribuídos – chamado *remotePP* – e foi estruturado da seguinte forma: o primeiro capítulo descreve o problema de pesquisa que o trabalho visa a solucionar, em que será apresentada uma justificativa quanto à sua importância, oportunidade e viabilidade para o meio acadêmico e comercial. Além disso, o capítulo é finalizado com o estabelecimento dos objetivos a serem alcançados e da metodologia empregada para a solução do problema.

Seqüencialmente, o segundo capítulo trata da revisão de literatura, e nele serão abordados os fundamentos da engenharia de *software*, assim como também as metodologias tradicionais e ágeis de desenvolvimento de *software*. Complementarmente, serão apresentados os conceitos e práticas da *Extreme Programming* com ênfase à programação em pares, a qual é objeto deste estudo. E, como este trabalho visa à implementação de um sistema colaborativo, será necessário ainda resgatar os conceitos básicos de *groupware* e *computer supported cooperative work* (CSCW).

Posteriormente, na seção de trabalhos relacionados, serão discutidos artigos recentes que envolvam diretamente o tema discutido deste trabalho, e ainda serão avaliados os resultados, experiências e limitações, a fim de levantar as alternativas já exploradas, seus erros e acertos. Assim, ao final desse capítulo, será possível identificar os conceitos básicos, bem como o "estado da arte" das ferramentas que viabilizem a programação em pares distribuída.

Por sua vez, o terceiro capítulo possui o título de *remotePP* e corresponde a toda documentação produzida para implementação de um ambiente cooperativo para programação em pares distribuídos. Para tanto, foi dividido nas seguintes seções:

- Análise de requisitos – que apresenta a especificação das funcionalidades que devem ser atendidas pela aplicação, casos de uso e prototipação de telas;
- Projeto – que mostra os diagramas de classe, padrões utilizados, diagramas de seqüência e arquitetura proposta;
- Aplicação – que traz o resultado final do sistema na forma de telas funcionais.

Finalmente, o último capítulo apresenta as considerações finais, a contribuição do trabalho para o meio acadêmico, assim como também as limitações encontradas. Além disso, sugestões para trabalhos futuros são propostas como forma de oferecer continuidade ao tema.

1 Definição do problema

A programação por pares é praticada, segundo Beck (2004), quando dois desenvolvedores sentam lado a lado, num único computador, colaborando na construção de um mesmo projeto, algoritmo, código-fonte ou teste. Sendo que esta prática oportuniza a troca de experiências e estilos de programação, além de estimular discussões sobre a abordagem adotada para a solução do problema.

Por sua vez, Willians e Kessler (2000) afirmam que "o olho humano tem uma grande capacidade de não enxergar aquilo que não quer ver". Da mesma forma, um desenvolvedor pode ignorar completamente um erro gritante que seria rapidamente percebido por um observador. Além disso, pesquisas realizadas pelos mesmos autores revelam que dois programadores juntos chegam a uma solução mais rápida e com maior qualidade do que dois programadores trabalhando separadamente na resolução de problemas.

Já conforme Stotts *et al* (2003), é cada vez maior o número de desenvolvedores que trabalham em times geograficamente dispersos. Os autores afirmam ainda que tendências indicam o aumento da oferta por educação à distância e necessidades empresariais, o que faz dos times distribuídos uma realidade em curto prazo.

Complementarmente a essa idéia, a programação distribuída é estimulada, segundo Kircher *et al* (2001), pela preferência por mão de obra mais barata oferecida em países em desenvolvimento ou pela preferência pessoal do desenvolvedor por trabalhar em casa. Entretanto, o autor lembra que os maiores obstáculos da abordagem distribuída estão na comunicação, coordenação, infra-estrutura, disponibilidade e gerenciamento do projeto.

Além desses obstáculos, Hanks (2005) identifica que as iniciativas para viabilizar programação por pares distribuída utilizam aplicações não orientadas para este fim. São utilizados, por exemplo: compartilhamento de *desktop*, *e-mail*, troca instantânea de mensagens, voz sobre IP, repositórios de arquivos compartilhados e vídeo-conferência, mas, no entanto, essas aplicações possuem objetivos específicos e não são orientadas à programação. Já para Stotts *et al* (2003), o desenvolvedor enfrenta dificuldades por não conseguir apontar um trecho de código ou desenhar um diagrama para explicar uma idéia. Além disso, por não serem compatíveis entre si, as aplicações exigem alto esforço de configuração. Podemos, ainda, identificar problemas potenciais ligados à disponibilidade das aplicações e questões de segurança ao utilizar *softwares* diferentes, sendo que, alguns destes não têm suporte adequado pelo fornecedor.

Neste contexto, torna-se importante a existência de um ambiente integrado que viabilize diferentes formas de comunicação orientadas à programação por pares de forma distribuída. Uma vez que, com um ambiente único, o programador não precisará recorrer a diversas aplicações oferecidas por terceiros que, em alguns casos, apresentam restrições e incompatibilidade entre si.

Dessa forma, o desenvolvimento desta aplicação torna-se viável, na medida em que serão utilizados vários recursos de engenharia de *software* como componentização, programação orientada a objetos, linguagens de quarta geração, além do conhecimento

já estabelecido por iniciativas semelhantes que se encontram disponíveis no meio acadêmico.

1.1 Questão de pesquisa

A questão que norteou esse trabalho, portanto, foi: de que forma é possível implementar um ambiente computacional que viabilize a programação em pares à distância mantendo os mesmos benefícios de um encontro presencial?

1.2 Objetivos

1.2.1 Objetivo geral

O objetivo geral desse trabalho foi o de desenvolver um ambiente que viabilize a programação em pares distribuída.

1.2.2 Objetivos específicos

Já os objetivos específicos foram os seguintes:

- Identificar as necessidades a serem atendidas por um ambiente de programação em pares distribuído;
- Elaborar o projeto de um ambiente computacional que viabilize a programação em pares;
- Implementar o ambiente.

1.3 Metodologia

O resumo dos procedimentos metodológicos utilizados para realização desse trabalho é apresentado na Figura 1.

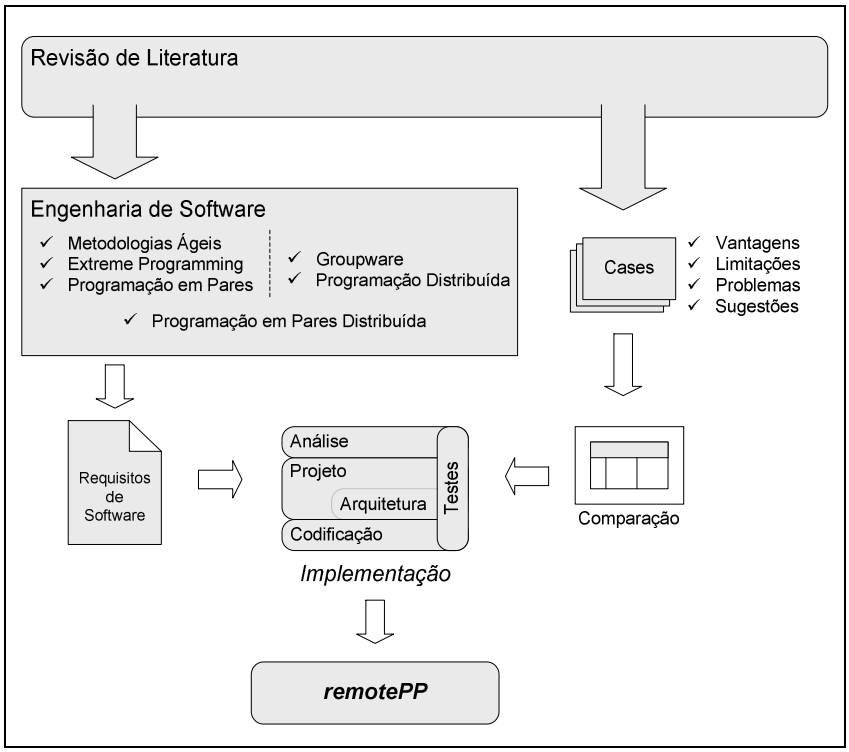

Figura 1 – Conjunto de atividades para atingir os objetivos do trabalho.

Por tratar de um assunto ligado à engenharia de *software*, a revisão de literatura será utilizada para explorar os conceitos relativos ao desenvolvimento de *software* e sistemas colaborativos. Para isso, partiremos das metodologias clássicas até chegarmos às recentes metodologias ágeis de desenvolvimento de *software*, e a *Extreme Programming* será estudada com ênfase na atividade de programação em pares.

Em outra perspectiva, serão resgatadas as definições acerca dos sistemas computacionais de apoio ao trabalho colaborativo (*groupware*): motivações, tendências e necessidades. Além disso, veremos como a utilização de *groupware* tem sido difundida no desenvolvimento distribuído de *software*.

A partir de então, será possível identificar um conjunto de requisitos funcionais e não funcionais para a proposta de um sistema de desenvolvimento de *software* que viabilize a programação em pares em um ambiente distribuído.

Identificaremos, ainda, um conjunto de iniciativas de ferramentas e ambientes que possuam propósito semelhante ao deste trabalho, cuja meta será avaliar as características oferecidas por outros sistemas voltados à programação em pares distribuída. Para tanto, serão avaliados os benefícios, as limitações, os problemas, e as sugestões apresentadas na realização dos trabalhos avaliados, sendo que uma tabulação será a forma utilizada para estabelecer aspectos de complementaridade entre as diferentes propostas estudadas.

Partindo desse panorama, será possível eleger um conjunto de requisitos funcionais e não funcionais para iniciar a implementação de um ambiente colaborativo que viabilize a programação em pares distribuída.

2 Fundamentação teórica

Nesse capítulo, é apresentada a revisão de literatura e trabalhos relacionados, em que são vistos conceitos teóricos e algumas aplicações práticas, e também estudadas diferentes iniciativas de sistemas que visam a oferecer um ambiente computacional de programação de pares distribuída.

2.1 Engenharia de *software*

Para Tonsig (2003), um dos principais objetivos da engenharia de *software* é obter um produto de *software* confiável, funcionando eficientemente e economicamente viável. Já segundo Pressman (2001), a engenharia de *software* envolve a definição dos métodos, ferramentas e procedimentos necessários para o desenvolvimento de *software* de qualidade e com boa produtividade. Tais conceitos são definidos da seguinte maneira:

Métodos – são atividades de planejamento e estimativa de projeto, que envolvem análise de requisitos, projeto, arquitetura, codificação, testes e manutenção. Sommerville (2001) afirma que nenhum processo pode ser considerado perfeito, pois cada organização adapta sua própria forma de desenvolver *software*;

Ferramentas – oferecem apoio automatizado ou semi-automatizado aos métodos;

Procedimentos – representam o encadeamento com os quais os métodos são realizados.

Por sua vez, as abordagens clássicas de engenharia de *software* separam as atividades de análise, projeto, codificação, testes e manutenção como etapas independentes de maior ou menor ciclo e interação (PRESSMAN, 2001). Embora, independentemente da abordagem, as atividades de projeto, codificação e testes caracterizam-se pela definição de como o sistema deve funcionar.

Durante o projeto, os requisitos são expressos na forma de representações gráficas, estruturas, arquiteturas, algoritmos e detalhes de interface. Sendo que a codificação é a tradução das representações produzidas em tempo de projeto para códigos legíveis pelo computador, e a função dos testes é identificar possíveis defeitos de função, lógica e implementação.

No que se refere à codificação, Pressman (2001) estabelece que nem todos os elementos são explicitados pelo projeto. Assim, cabe ao programador oferecer uma boa eficiência ao código e uso racional dos recursos computacionais, além de boa documentação para futuras manutenções.

Por sua vez, Tonsig (2003) afirma que um dos maiores problemas dos processos de desenvolvimento de *software* está ligado a problemas de comunicação interpessoal. E Pressman (2001) estabelece que a revisão de *software* deve ser aplicada em vários pontos, durante o desenvolvimento de *software*, pois, dessa forma, cada defeito pode ser corrigido antes do passo seguinte do processo de engenharia.

2.1.1 Metodologias tradicionais

As metodologias de engenharia de *software* tradicionais estão baseadas no ciclo de vida clássico (ou modelo em cascata). No entanto, Pressman (2001) afirma que esse modelo sugere uma abordagem rígida quando à realização de cada atividade (Figura 2).

Por sua vez, Peters e Pedrycz (2001) apontam, como vantagem do modelo, a possibilidade de gerência de *baseline*: identificar um conjunto fixo de documentos produzidos como resultado de cada fase do ciclo de vida.

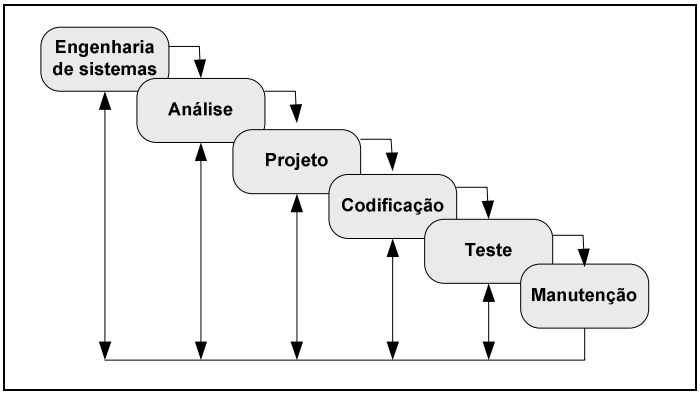

Figura 2 – O ciclo de vida clássico.

O autor lembra ainda que o ciclo de vida clássico é o paradigma mais antigo e utilizado, mas que também é alvo de uma série de críticas, sendo que Pressman (2001) cita as seguintes limitações:

- Na prática, dificilmente os projetos seguem o fluxo seqüencial que o modelo propõe;
- Devido ao caráter rígido, é difícil acomodar alterações requisitadas pelo cliente depois que o projeto é iniciado;
- É necessário aguardar um longo período de tempo até que o cliente possa receber a primeira versão do *software*, o que oferece dificuldades para identificar problemas de interpretação dos requisitos iniciais (PETERS E PEDRYZA, 2001).

Conseqüentemente, para remediar as limitações do ciclo de vida clássico, foram propostos os modelos evolucionários e prototipação (Figura 3), que visam a apresentar um projeto rápido que possa ser avaliado pelo cliente, cujo objetivo é o de reduzir o risco de problemas potenciais antes que atinjam todo o sistema.

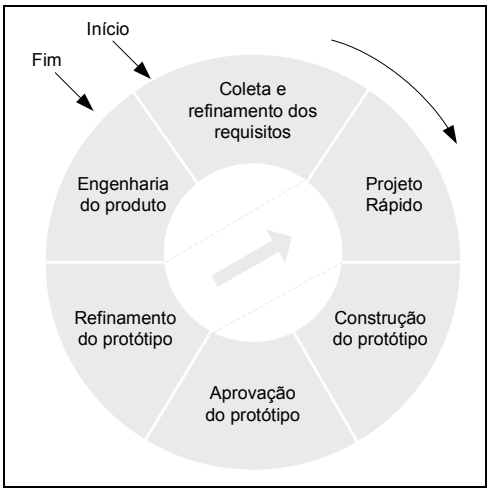

Figura 3 – Prototipação.

Já a abordagem espiral (Figura 4) prega seqüências de interação repetidas e formadas por entregas parciais, cujo modelo abrange as melhores características tanto do ciclo de vida quanto do modelo prototipação. Assim, a cada ciclo completo, o cliente recebe uma versão do sistema, sendo que os primeiros ciclos oferecem apenas protótipos com o objetivo de avaliação.

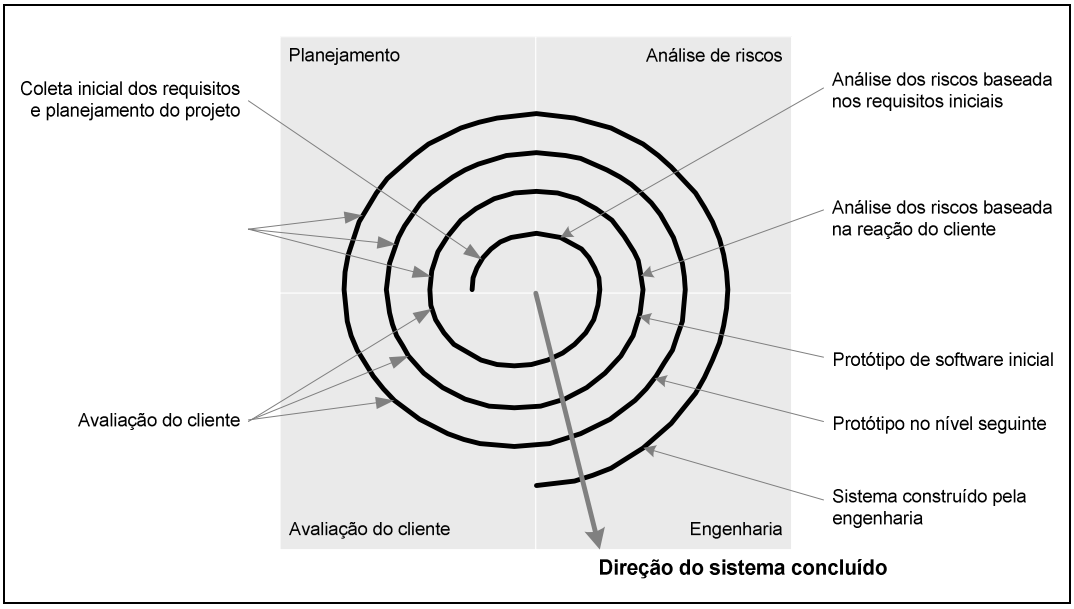

Figura 4 – O modelo espiral.

2.1.2 Metodologias ágeis

Abrahamsson *et al* (2002) explicam que as metodologias baseadas em processos tradicionais tendem a oferecer resistências quando o ambiente do negócio exige rapidez e mudança constante dos requisitos (alta volatilidade de requisitos). Além disso, os autores exemplificam o desenvolvimento de *software* para a Internet e a computação móvel como áreas em franca expansão e que exigem respostas rápidas da indústria de *software*. Já McCauley (2001) afirma que, algumas vezes, a metodologia empregada no desenvolvimento de *software* precisa ser flexível e adaptativa, de modo que mudanças nos requisitos não impactem em alto custo de dinheiro ou tempo para serem implementadas depois que o desenvolvimento foi iniciado.

Em outra perspectiva, McCauley (2001) lembra que o ato de programar *software* sempre foi considerado "divertido" pelos desenvolvedores de sistemas e que as metodologias tradicionais desestimulam o processo criativo. Nesse contexto, surgiram as metodologias ágeis de desenvolvimento de *software* como uma alternativa viável.

Assim, em *Agile Manifesto*, Fowler e Highsmith (2001) estabelecem um conjunto de características comuns a todas as metodologias ágeis, dentre as quais, os autores citam: iterações curtas; entregas que sejam feitas o mais cedo possível; envolvimento efetivo do cliente; simplicidade nos métodos; e disposição para oferecer respostas rápidas a mudanças nos requisitos.

Abrahamsson *et al* (2002) corroboram Fowler e Highsmith (2001), declarando que, para estar de acordo com uma metodologia ágil, as seguintes características devem estar presentes:

- Incremental – pequenas entregas com rápidos ciclos (entregas semanais, por exemplo);
- Cooperativa – cliente e desenvolvedores com boa comunicação e trabalhando juntos;
- Clareza – a metodologia deve ser documentada e estar disponível para rápida aprendizagem e modificação;
- Adaptativo – permite mudanças de última hora nos requisitos do cliente.

Entretanto, McCauley (2001) alerta que as metodologias ágeis funcionam melhor para determinados projetos, e que não devem, por exemplo, ser utilizadas em projetos que envolvam equipes com mais de cinqüenta pessoas (FOWLER E HIGHSMITH, 2001).

2.1.3 Extreme Programming

A *Extreme Programming* (XP) foi criada por Kent Beck em 1996 (Williams e Kessler, 2000), e, segundo Abrahamsson (2003), esse é um dos mais bem sucedidos e conhecidos exemplos de aplicação dos conceitos de metodologia ágil.

Por sua vez, Abrahamsson *et al* (2002) afirmam ainda que a *Extreme Programming* surgiu como uma alternativa aos problemas causados pelos longos ciclos de desenvolvimento dos modelos tradicionais, e, embora as práticas utilizadas pela XP não sejam novas, são colocadas em prática "ao extremo", conforme Beck (2004).

A Figura 5 apresenta as fases do ciclo de vida de um projeto de *Extreme Programming*.

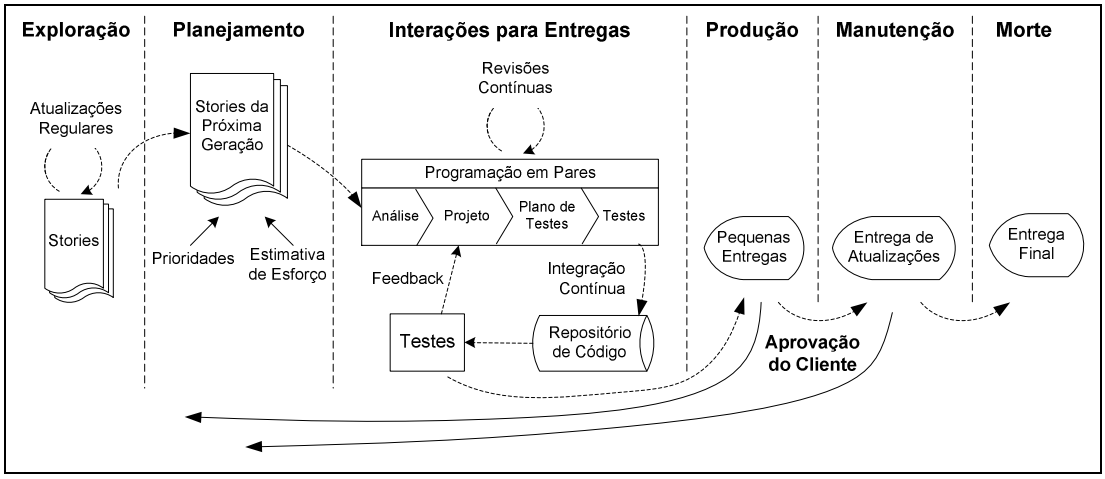

Figura 5 – Ciclo de vida do processo XP.

Abrahamsson (2003) afirma também que a *Extreme Programming* é caracterizada por ciclos pequenos de desenvolvimento, planejamento incremental, projeto evolucionário e, principalmente, pela possibilidade em oferecer respostas a mudanças nas regras de negócio.

Já Beck (2004) apresenta algumas diferenças quanto aos papéis e tarefas entre as pessoas envolvidas no desenvolvimento de *software* XP; dentre as quais pode-se citar, por exemplo, que o programador é incentivado a escrever testes e manter o código o mais simples quanto possível. Além disso, diz que a comunicação e a coordenação com outros programadores é fator crítico para o sucesso do uso da metodologia no projeto.

Tal metodologia estabelece um conjunto de práticas bem definidas (BECK, 2004), as quais são:

- Jogo de planejamento – reunião semanal para estabelecer as prioridades da semana;
- Pequenas versões – liberação constante de versões funcionais de modo que o cliente possa testá-las o mais rápido possível;
- Metáfora – seu uso objetiva estabelecer o entendimento comum entre os termos técnicos do negócio do cliente;
- Projeto simples – usado para evitar ao máximo as soluções complexas, e para restringir o projeto ao problema ou à situação requisitada;
- Time coeso – significa que o cliente e a equipe de desenvolvimento devem estar "jogando no mesmo time", ou seja, deve haver um canal aberto entre ambas as partes para comunicação e esclarecimento de dúvidas;
- Testes de aceitação – são testes que determinam o nível mínimo de aceitação do produto de *software*, e cuja construção deve envolver programadores, cliente e testadores;
- Semana de 40 horas – significa que horas extras somente devem ser feitas quando estas forem produtivas;
- Reuniões em pé – é a prática de estabelecer um assunto e manter o foco nele em reuniões rápidas;
- Posse coletiva – significa não restringir o acesso a determinados artefatos vinculando-os a perfis de usuários, uma vez que todos devem ter acesso ao código-fonte e ao resultado final;
- Refatoração – é estimular a reengenharia constante dos módulos do sistema, de modo a tornar o código mais inteligível e com melhor coesão;
- Padrões de codificação – objetiva assumir um contrato entre os desenvolvedores, a fim de padronizar soluções e implementações;
- Desenvolvimento orientado a testes – é a prática de escrever testes antes do código e executá-los ao final da programação;
- Integração contínua – é o ato de aderir uma nova funcionalidade ao sistema como um todo assim que for implementado;

- Programação em pares – é quando duas pessoas sentam-se lado a lado e trabalham em um único computador, ao mesmo tempo, e visa a descobrir falhas e incentivar discussões quanto à melhor forma de resolver o problema.

Sendo assim, uma vez que, das práticas citadas anteriormente, a programação em pares tornou-se uma das mais conhecidas da XP, Williams (1999) conclui que a programação em pares oferece ao desenvolvedor um aprendizado mais rápido, além de maior confiança e prazer no trabalho. Em seu estudo, o autor verificou, ainda, um significativo aumento da produtividade, atribuído ao fato de os desenvolvedores sentirem-se estimulados pelo seu par a manter o foco no trabalho, pois a complementaridade de experiências formada pelo par ajuda na resolução de problemas.

2.1.4 Programação em pares

Cockburn e Williams (2001) lembram que a inspeção de código é uma técnica baseada na detecção e remoção de defeitos em *software*, e, segundo Pressman (2001), quanto mais cedo os defeitos forem encontrados, menor será o custo para sua remoção.

Entretanto, programadores não sentem satisfação em procurar por defeitos em seu trabalho ou mesmo no produto de outros programadores (COCKBURN E WILLIAMS, 2001). Portanto, a XP estabelece que, na programação em pares, dois desenvolvedores trabalham juntos, dividindo um mesmo computador, *mouse* e teclado, a fim de solucionar um problema. Assim, uma vez que essa prática favorece a identificação de defeitos, antes mesmo que sejam inseridos no código, é importante citar que Williams *et al* (2000) lembram que "o olho humano tem a quase infinita capacidade de não enxergar aquilo que não quer ver".

Além disso, duas pessoas têm condições de encontrar um número maior de soluções para um problema do que uma pessoa que esteja trabalhando sozinha, pois, em pares, podem discutir e decidir mais rapidamente a melhor forma de implementação. Pois, enquanto um dos pares está digitando o código, o outro pode estar pensando em nível mais estratégico, como nos impactos da abordagem em outras partes do sistema. Essa prática favorece conseqüentemente a revisão constante do produto de código, uma vez que o par revisa ao mesmo tempo em que implementa, e a revisão ocorre naturalmente e de forma eficiente.

Do mesmo modo, Cockburn e Williams (2001) constataram que a programação em pares oferece melhor qualidade e melhores soluções de projeto, pois o número de linhas de código tende a ser menor. Os autores afirmam, ainda, que a proximidade entre os desenvolvedores ajuda na troca de experiências, reduzindo a curva de aprendizado dos novos programadores que passam a adquirir o *know-how* dos mais experientes, uma vez que a comunicação não-verbal é estimulada e o conhecimento tácito é transferido com maior facilidade.

Somando-se a isso, uma vez que a cada atividade os pares são trocados, a experiência adquirida é disseminada para todos os membros da equipe. Ainda, sob o ponto de vista de gerência de projeto, Cockburn e Williams (2001), mencionam que a uniformização do conhecimento e a rapidez no aprendizado minimizam a perda do capital intelectual quando da saída de programadores-chave do projeto.

Já Williams e Kessler (2000) descrevem a aplicação prática da programação em pares, conforme Beck (2004), da seguinte forma: uma das pessoas conduz o teclado (*driver)* enquanto o outro acompanha e revisa trabalho (*navigator*). Embora, ao contrário do que possa parecer, o papel do *navigator* não é entediante, pois ele deve ficar atento e participar ativamente da linha de raciocínio da dupla. Além do mais, por estarem

engajados juntos numa única atividade, cada participante estimula o parceiro a manter o foco no trabalho. Dessa maneira, dificilmente, o trabalho é interrompido para ler *e-mails* particulares ou navegar na Internet, por exemplo, o que oferece ganho de produtividade.

Os programadores tendem ainda a desenvolver uma maior autoconfiança quanto ao produto do trabalho, pois a sua solução foi discutida e, de certa forma, avalizada pelo seu parceiro, e, também, os erros são apontados de forma muito mais amigável do que através de uma inspeção formal de código.

No entanto, ao iniciar uma nova dupla, sugere-se que deva existir um período de adaptação, que pode levar horas ou até dias, e cujos objetivos são o de "quebrar o gelo" e o de propiciar o conhecimento mútuo entre os novos parceiros, assim como também seu estilo de programação. Para tanto, ambos devem ter humildade para aceitarem críticas, mas também habilidade de argumentação suficiente para defender seu ponto de vista, quando considerar adequado.

Para finalizar, é importante citar que Williams (2000) refuta a teoria de que a prática de programação em pares dobraria o número de recursos humanos necessários para realizar um projeto, pois, para o autor, passado o período de ajuste inicial, a redução do tempo gasto para realização da tarefa, aliada à redução do tempo para remoção de defeitos, justifica a adoção da programação em pares.

2.2 *Groupware* / Modelo 3C

Greif (1988) *apud* Ellis *et al* (1991) definem *computer supported cooperative work* (CSCW), ou *groupware*, como o estudo dos sistemas que integram o processamento de informações a atividades de comunicação, de forma a identificar como os grupos trabalham e como a tecnologia pode ajudá-los a resolver tarefas em comum.

Já Ellis *et al* (1991) reforçam a importância da comunicação, colaboração e coordenação (Figura 6) a fim de viabilizar o trabalho em grupo e obter melhores resultados, quando comparados ao trabalho realizado individualmente.

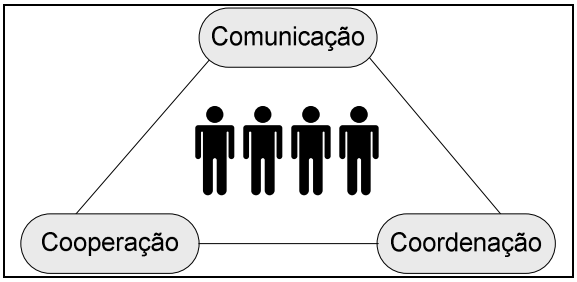

Figura 6 – O modelo 3C.

Segundo Fuks *et al* (2003), a comunicação oportuniza a negociação e o estabelecimento de compromissos entre os participantes do grupo; sendo que, para utilizar o computador como ferramenta de comunicação, é necessário que o sistema ofereça suporte à interação entre as pessoas, devendo haver ainda controle entre os estados, eventos e diálogos de cada participante, conforme apresentado na Figura 7.

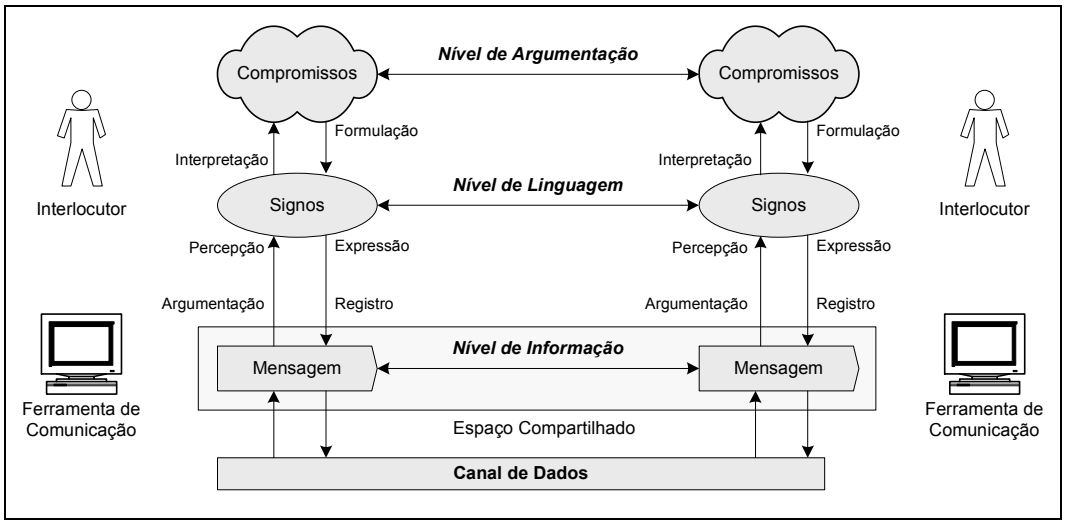

Figura 7 – Comunicação mediada por computador.

Pode-se então perceber que, de acordo com o modelo, o emissor define a sua intenção (no nível mais abstrato de idéias) e formula através de símbolos, depois disso, utiliza o computador para expressá-la numa mensagem e transmiti-la via canal de dados. Finalmente, chegando ao destino, a mensagem é percebida e interpretada de forma a modificar seus argumentos.

Complementarmente, Ellis *et al* (1991) estabelecem dois tipos de classificação para as ferramentas de comunicação: quanto ao tempo de resposta a ao nível de aplicação, as quais são explicadas a seguir.

Quanto ao tempo de resposta:

- Síncronas – as mensagens são transferidas instantaneamente. Desta forma, as respostas são obtidas imediatamente depois de interpretadas, e podem ser utilizadas para construção de conhecimento em conjunto;
- Assíncronas – as mensagens levam um certo tempo para serem transferidas até o destino final. Sendo que Fuks *et al* (2003) afirmam que esse tipo de ferramenta valoriza a reflexão dos participantes, pois oferece tempo para que sejam formuladas respostas mais elaboradas.

Quanto ao nível de aplicação:

- Sistemas de mensagens – suportam a troca de mensagens de texto entre usuários de grupos, como o *e-mail* por exemplo. Além disso, algumas aplicações permitem a tomada de decisões baseadas em regras pré-definidas, sendo que, nesse caso, podem ser utilizados critérios como conteúdo da mensagem ou remetente;
- Editores multiusuários – permitem que documentos sejam alterados por vários usuários; e podem ser assíncronas, quando cada usuário manipula o documento individualmente; ou síncronas, quando as alterações são imediatamente percebidas pelos envolvidos. Algumas aplicações dividem, ainda, o documento em segmentos lógicos, fazendo controle de bloqueio e sincronização destas seções entre os usuários;
- Sistemas para decisão em grupo e salas de conversação – oferecem mecanismos para tomada de decisão em grupo a fim de encontrar soluções para problemas

estruturados ou não estruturados que exigem a participação de várias pessoas. Sendo que algumas aplicações podem implementar salas de conversação como forma de viabilizar a discussão, e outras dispõem de ferramentas para votação, *ranking* ou árvores de decisão;

- Conferências – são, em geral, síncronas e podem ser realizadas com o auxilio de recursos multimídia como áudio (*conference call*) ou vídeo (teleconferência). Atualmente, os computadores pessoais possuem os recursos mínimos necessários para a realização de áudio ou vídeo-conferência;
- Agentes inteligentes – além de pessoas, os grupos podem ser formados por membros não humanos, chamados de agentes inteligentes, os quais são responsáveis por um conjunto de tarefas específicas;
- Sistemas de coordenação – viabilizam o controle e acompanhamento das atividades do grupo.

Os autores estabelecem, ainda, um conjunto de funcionalidades importantes a sistemas de colaboração em tempo real (síncronos), os quais são descritos na seguinte tabela:

Funcionalidade	Definição
Shared Context	É o conjunto de objetos e ações realizadas e percebidas por todos os usuários
Group Window	São as áreas de tela compartilhadas e sincronizadas por todos os participantes instantaneamente, como, por exemplo, o quadro branco
Telepointer	Significa que o movimento do cursor é percebido e disponibilizado a todos
View	É uma representação visual ou multimídia de parte do *shared context*
Synchronous & Asynchronous Interaction	São os tipos de interação, sendo que algumas são percebidas em tempo real, enquanto outras podem ser realizadas em *offline*, com sincronização posterior
Session	É o período de interação síncrona
Role	É o conjunto de privilégios, atribuições ou responsabilidades de participante do grupo, que podem ser formais ou informais

Tabela 1 – Funcionalidades de sistemas de colaboração síncronas.

2.2.1 Desenvolvimento de *software* distribuído

A tendência pelo desenvolvimento de *software* através de grupos geograficamente distribuídos é justificada, segundo Herbsleb e Moitra (2001), pela redução de custos e busca por mão-de-obra qualificada. Sendo que os autores citam fatores como as necessidades competitivas e a globalização das operações corporativas como atenuantes dessas tendências.

No entanto, evidências indicam que o desenvolvimento distribuído costuma consumir tempo maior quando comparado ao realizado por equipes geograficamente centralizadas, sendo que Herbsleb e Moitra (2001) identificam a comunicação e coordenação como principais causas para esse aumento de tempo.

Por sua vez, Carmel (1997) observa que a riqueza na comunicação pode ser medida em relação número de canais sensoriais utilizados pelas pessoas. E, uma vez que apenas

20% da comunicação é verbal, pode-se afirmar que a maior parte da comunicação ocorre de maneira não verbal, ou seja, através de gestos, timbre de voz, expressões faciais ou representações gráficas.

Como conseqüência desse dado, em sua pesquisa sobre desenvolvimento global de *software*, Carmel (1997) verifica que, para lidar com problemas difíceis, é preferível a utilização de interações face-a-face. Observa-se ainda que, em segundo lugar, utiliza-se o telefone, e que já o *e-mail* é utilizado como último recurso de comunicação na resolução de questões com maior complexidade.

Já Carmel *apud* Prikladnicki *et al* (2004) sugere um conjunto de cinco fatores que podem dificultar o sucesso de um projeto distribuído: a comunicação ineficiente, a dispersão geográfica, as diferenças culturais, a perda do espírito de equipe, e a falta de coordenação. Em contrapartida, ele elege seis fatores que contribuem para o sucesso: a arquitetura do produto, a construção de equipe, as técnicas de gerência, a tecnologia de colaboração, a metodologia de desenvolvimento, e a infra-estrutura de comunicação (Figura 8).

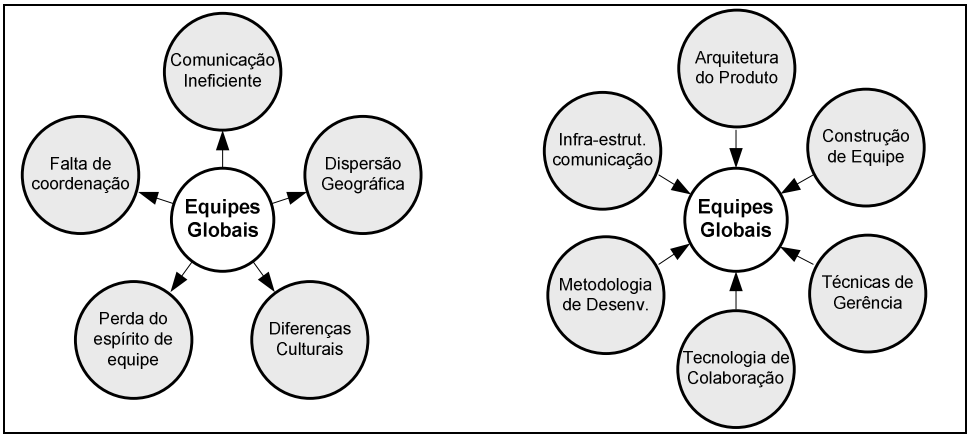

Figura 8 – Forças de equipes globais.

Na Tabela 2, Borges *apud* Reis (1998) estabelece um conjunto de ferramentas de *groupware* que podem auxiliar as etapas do ciclo de vida de desenvolvimento de *software*.

Fase do ciclo de vida	***Groupware***
Análise	Sistemas de apoio à decisão Sistemas de co-autoria Salas eletrônicas
Projeto	Sistemas de apoio à decisão Sistemas de co-autoria Salas eletrônicas
Implementação	Sistemas de co-autoria Sistemas de conferência
Verificação	Sistemas de apoio à decisão Salas eletrônicas Sistemas de conferência
Manutenção	Sistemas de apoio à decisão Sistemas de conferência

Tabela 2 – Ferramentas de *groupware* para desenvolvimento de *software*

2.3 Programação em pares distribuída

Vários estudos têm sido realizados com o objetivo de avaliar as diferenças apresentadas, quanto à produtividade e à qualidade, pela programação em pares distribuída em relação à realizada "lado a lado" (presencial e no mesmo computador). Stotts *et al* (2003) citam como vantagens:

- Em determinadas situações, enquanto um dos programadores edita o código, o outro pode realizar pesquisas em bases de conhecimento ou na *web*;
- Não é necessária alteração no *layout* do escritório a fim de viabilizar que dois programadores sentem lado a lado;
- Há redução da necessidade de viagens;
- Os pares são forçados a manter registros eletrônicos de seus trabalhos e idéias;
- Os membros tendem a diminuir as conversas *off-topic*, permanecendo mais tempo focados na sua tarefa.

No entanto, Schummer e Schummer (2001) perceberam que, durante uma sessão de programação em pares distribuída, os programadores devem estar aptos a se comunicar, além do mais, ambos devem acompanhar o mesmo trecho de código, mas que apenas um deve poder alterá-lo. Em relação à comunicação, os autores ressaltam que ela pode ocorrer por voz, sem a necessidade do contato visual.

A Tabela 3 lista o conjunto de ferramentas utilizadas para viabilizar a programação em pares nos estudos de Baheti *et al* (2002), Canfora et al (2003) e Hanks *et al* (2005).

Atividade	Ferramenta
Co-autoria (edição de código)	*Microsoft NetMeeting* *Symantec's PC Anywhere* *VNC*
Comunicação (voz, vídeo, *chat*)	*Yahoo Messenger* *PalTalk* *AOL Messenger*

Tabela 3 – Ferramentas utilizadas para programação em pares distribuída.

Em relação ao *desktop*, a maioria das formas utilizadas para co-autoria restringe ao compartilhamento deste, pois, dessa forma, ambos os programadores enxergam o mesmo ambiente de programação, em que um assume o papel de *driver* e digitava o código, enquanto o outro apenas observa (*navigator*).

Em se tratando da comunicação, ela ocorre por voz, vídeo e *chat*. Assim, quando é necessário mostrar alguma idéia na forma de diagramas, este é desenhado a mão em uma folha de papel e aproximado de uma câmera de vídeo. Além disso, embora, o uso da câmera de vídeo tenha sido mais intenso no início dos estudos, com o passar do tempo, as duplas preferiram apenas utilizar comunicação por voz e *chat*.

Por sua vez, Baheti *et al* (2002) concluem em seu trabalho que a produtividade e a qualidade do produto de *software* não apresentaram significativas diferenças entre a programação em pares realizada de forma presencial e distribuída.

Entretanto, na pesquisa de Canfora *et al* (2003), os autores perceberam que os programadores organizados em pares distribuídos tenderam a trabalhar isoladamente ao realizar suas tarefas, não colocando em prática a programação em pares. Para os autores,

as principais causas desse problema foram: quedas no canal de comunicação (rede de transmissão de dados); falta de diálogo para chegar a um consenso quanto à solução a ser adotada, uma vez que cada programador considerava sua solução melhor que a do parceiro; e o caráter anárquico com o qual as sessões foram conduzidas, ou seja, sem regras pré-estabelecidas.

Finalmente, Hanks (2005) avaliou a programação em pares distribuída no ensino de um curso introdutório de programação, e notou que a experiência foi positiva, pois concluiu que as notas e o nível de confiança no trabalho foram semelhantes àqueles obtidos pelo grupo de controle.

2.4 Trabalhos relacionados

Várias pesquisas apresentam propostas de arquitetura ou implementação de ferramentas específicas com o objetivo de viabilizar a programação em pares distribuída. Assim, o objetivo desta sessão é avaliar um conjunto de ferramentas a fim de identificar aspectos positivos e limitações. Posteriormente, uma tabela permitirá o cruzamento das principais características verificadas nas ferramentas.

2.4.1 Cooper

O *collaborative pair programming editor* (Cooper) foi desenvolvido por Natsu *et al* (2003), possui arquitetura orientada a cliente / servidor para *web*, e é composto por três subsistemas: (1) editor colaborativo; (2) repositório de documentos e usuários (IM&P); e (3) sistema de áudio, cuja comunicação faz parte da arquitetura original, porém, não foi implementada.

O repositório de documentos e usuários é responsável pelo gerenciamento dos documentos, controle de bloqueios e gerenciamento dos usuários dentro das sessões. Para tanto, utilizam-se de servidores *open source Jabber* e *WebDAV* (repositório de arquivos).

O editor de códigos-fonte possui suporte a *highlight*, e permite que dois programadores trabalhem no mesmo código ao mesmo tempo. Além disso, suporta compilação e execução do código digitado, notificando as mensagens de erro a ambos os programadores. Sendo que o código fica armazenado em servidor (*web*), e há controle de bloqueio (conceito de *driver* e *navigator*). Assim, em caso de perda da conexão, o usuário pode continuar editando o documento localmente e realizar a sincronização posterior.

A interface gráfica apresenta um semáforo para identificar o estado de controle do documento na forma de cores, as quais são:

- Vermelho – o usuário encontra-se como *navigator* e só pode observar o código;
- Verde – o usuário tem permissão para alterar o documento (*driver*).
- Vermelho e Amarelo – o usuário é um observador que deseja realizar alterações;
- Verde e Amarelo – o usuário está editando o documento, mas seu parceiro deseja realizar alterações.

Abaixo do semáforo, há um botão com as possíveis ações disponíveis, as quais são: requisitar, conceder ou cancelar.

Durante a avaliação da aplicação, os autores concluíram que um dos maiores problemas apontados foi a comunicação baseada em “mensagens de texto” (*chat*), a qual não

suportava o dinamismo necessário na programação em pares. Além disso, o repositório de documentos (*WebDAV*) não apresentou algumas funcionalidades e robustez satisfatória.

2.4.2 Milos

Maurer e Martel (2002) desenvolveram o Milos (*minimally invasive longterm organizational support*) como um ambiente colaborativo para gerenciamento de projetos XP pela *web*.

A ferramenta oferece mecanismos para o gerenciamento de projetos através de interface *web*, em que é possível criar *user stories* e associá-los a atividades vinculadas a projetos, além de acompanhar o andamento de atividades individuais ou de todo o projeto.

A tela principal da aplicação apresenta ao usuário uma lista das atividades: *user stories*, projetos, estimativas e acompanhamento das tarefas em andamento; e o próprio ambiente trata do versionamento e gerenciamento de arquivos, os quais são anexados diretamente às atividades, o que é realizado pelo servidor HTTP.

A programação em pares não é suportada pela aplicação, sendo necessário o uso de compartilhamento de *desktop* (como o *MS-NetMeeting*). E, em conseqüência desse compartilhamento, as principais limitações percebidas pelo Milos encontram-se no alto consumo de banda de rede, o que inviabiliza sua utilização em conexão *dial-up* de 56Kbps. Além do mais, é necessário estabelecer um protocolo para a interação sobre o código-fonte, pois a ferramenta não oferece mecanismos de bloqueio.

2.4.3 Sangam

O Sangam foi desenvolvido por HO *et al* (2004) na forma de um *plug-in* para o ambiente de desenvolvimento Eclipse da IBM.

A ferramenta oferece sincronização de código-fonte entre mais de dois programadores, ou seja, tudo que é digitado por um programador é acompanhado pelos demais.

A arquitetura do Sangam utiliza três componentes principais, os quais são:

1. Interceptador de eventos – monitora o ambiente a fim de capturar todas as ações que o usuário (*driver*) realiza no editor de código Java da aplicação.
2. Servidor de mensagens – é responsável pelo envio das mensagens (ações) ao *navigator*, sendo que, numa mesma sessão, vários usuários podem acompanhar o mesmo código-fonte. Para isso, devem conectar-se ao mesmo servidor de mensagens, no caso, o Sangam utiliza o servidor *Kizna SyncShare* para esta finalidade.
3. Reprodutor de eventos – realiza as ações recebidas pelo servidor de mensagem de modo a reproduzi-las no computador do *navigator*.

O Sangam permite a execução simultânea e sincronizada de aplicações Java e testes unitários (*JUnit*). No entanto, algumas limitações foram percebidas na ferramenta, principalmente devido à falta de suporte a algumas funcionalidades do Eclipse.

Entre essas limitações, é importante citar que o editor de texto do *navigator* não fica bloqueado durante a sessão, sendo necessário estabelecer um protocolo de interação entre os desenvolvedores, para que os dois não mudem o mesmo código-fonte. Além disso, não há suporte a outros ambientes de programação disponibilizados pelo Eclipse, como CDT (*C++ Development Tools*).

Já as atividades relacionadas à comunicação necessitam de outras ferramentas, como comunicadores instantâneos e repositórios de arquivos, além de haver conflitos com o gerenciador de versionamento CVS. Além do mais, problemas de incompatibilidade com o Eclipse são percebidos em algumas funcionalidades de manipulação de arquivos, como o *refactoring*.

Para finalizar, é importante citar que os autores sugerem como trabalhos futuros, a coleta de todos os dados produzidos durante a sessão de programação em pares.

2.4.4 Tukan

O Tukan foi desenvolvido por Schummer e Schummer (2001) com base numa implementação do ambiente para programação *Smalltalk*, o *Envy / VisualWorks*; e permite edição de código compartilhado; além de gerenciamento de controle de versão e depuração compartilhada baseada em linha de código (a tela de saída é visualizada por ambos os programadores).

Além disso, apresenta uma série de benefícios advindos do *Envy*, sendo voltado à programação orientada a objetos para *Smalltalk*. Entre estas funcionalidades, está a navegação através da relação entre artefatos do sistema, sendo que tais relacionamentos são medidos de acordo com a distância entre eles. Assim, os seguintes relacionamentos são considerados: estrutural, que é a relação de herança ou composição entre métodos e suas classes; o uso, que é a relação de chamada entre dois métodos; e versão, que é a relação entre diferentes versões de um mesmo artefato.

O Tukan agrega uma funcionalidade baseada nos relacionamentos que permite avaliar, em tempo real, o impacto que uma alteração pode oferecer as outras partes do sistema através de um cálculo de proximidade; e utiliza, então, a metáfora de "previsão do tempo" para indicar a possibilidade de conflito (Figura 9).

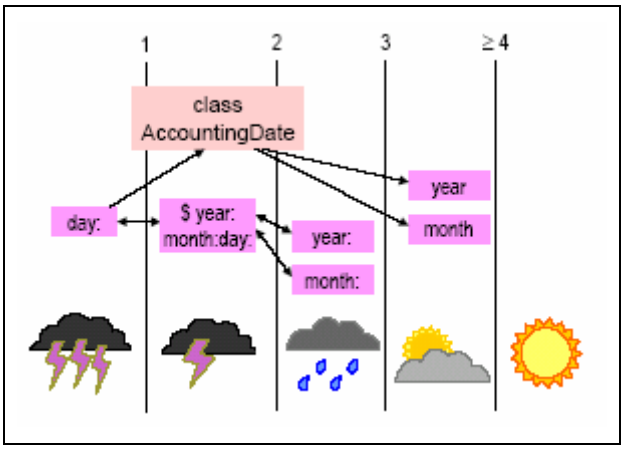

Figura 9 – Conflitos entre métodos.

A programação colaborativa é disponibilizada através de um ambiente chamado de *Collaborative Class Browser*, onde é possível "navegar" entre as classes e seus métodos, ou seja, assim, mais de um usuário pode acessar o mesmo método, ao mesmo tempo. É possível ainda realizar comunicação por *chat* ou voz, sendo que, no topo da tela, aparecem as fotos de todos os participantes e o dispositivo de comunicação utilizado (teclado ou voz).

Além disso, o cursor do outro usuário aparece na forma de um círculo colorido, o que facilita o apontamento de trechos do código; e, como não há bloqueio do teclado, ambos os programadores podem alterar o código ao mesmo tempo. Além do mais, durante uma

sessão de programação, caso os programadores encontrem dificuldades em entender o trecho de código, podem requisitar auxílio do autor do artefato.

Assim, a abordagem utilizada pelo Tukan na detecção de conflitos – que, em geral são percebidos somente nos testes de integração – em tempo de desenvolvimento é bastante útil, e, caso o artefato conflitante esteja sendo alterado, é possível iniciar uma sessão de programação em pares, a fim de resolver o impasse.

Uma outra ferramenta chamada *Activity Explorer*, onde são criados e mantidos todos os *story cards*, suporta o gerenciamento das atividades. Além disso, os programadores podem definir as prioridades de implementação, e é mantido, ainda, o histórico das atividades realizadas por *story cards*.

2.4.5 Flecse

Dewan e Riedl (1993) descrevem o Flecse (*flexible environment for collaborative software engineering*) como um conjunto de ferramentas colaborativas e aplicáveis a todo o ciclo de vida de desenvolvimento de *software*, sendo que a interface de todas as aplicações é baseada em linha de comando e console texto em que:

- O controle de versão é realizado com o *RCSTool*, que permite operações de *check-in* e *check-out* em artefatos depositados em um servidor RCS compartilhado;
- *MShell* é um interpretador de comandos colaborativo que permite a vários usuários executar comandos e acompanhar os resultados;
- *Teleconf* disponibiliza gravação, transmissão e execução de uma conversação por áudio;
- *MDebug* oferece uma interface para a depuração de programas de forma compartilhada;
- CSI (*collaborative software inspection*) apresenta um mecanismo de inspeção assíncrono e síncrono. Para isso, o revisor inspeciona o código individualmente e faz anotações textuais ou registros de áudio com o auxílio do *Teleconf*. Assim, cada registro é ligado a um trecho do código, e, posteriormente, o autor e o revisor iniciam uma discussão *on-line* e revisam as questões apontadas pelo revisor. Além disso, o uso de registro em áudio agiliza o processo de revisão;
- *MEdit* é o editor de código-fonte colaborativo, que, além do bloqueio de todo o arquivo, oferece mecanismos para bloqueio por função ou bloqueio por linha de código. Essa flexibilidade permite que os programadores decidam se desejam alterar o código individualmente sob a supervisão do parceiro ou ao mesmo tempo.

Outras ferramentas que compõem o Flecse são: *SplitTool*, para controle de concorrência; *MAtac*, que identifica as partes do programa que são cobertas pela massa de testes; e o *Nav*, que permite a navegação pelos arquivos descritos no *makefile*.

No entanto, os autores indicam entre as principais limitações no Flesce: a falta de funcionalidades de *undo* e *redo*; a falta de ferramentas que suportem gerenciamento de projeto; e, por terem sido implementadas independentemente, as ferramentas não possuem boa integração entre si ou padronização de interface.

2.4.6 Comparação entre as ferramentas

A Tabela 4 resume as funcionalidades apresentadas pelas cinco ferramentas avaliadas, sendo que elas foram agrupadas de acordo com as seguintes áreas: geral, comunicação, ambiente de programação, arquitetura, e compilação / execução / debug.

Funcionalidade	**Cooper**	**Milos**	**Sangam**	**Tukan**	**Flecse**
Geral					
Repositório de arquivos e versões	✓	✓	-	✓	✓
Troca de arquivos diversos	-	-	-	-	-
Gerenciamento de projeto	-	✓	-	✓	-
Gerenciamento de tarefas	-	✓	-	✓	-
Bookmark compartilhado	-	-	-	-	-
Comunicação					
Chat	✓	-	-	✓	✓
Áudio-conferência	-	-	-	✓	✓
Vídeo-conferência	-	-	-	-	-
Quadro branco	-	-	-	-	-
Coleta de informações da sessão	-	-	-	-	✓
Ambiente de programação					
Possui editor próprio	✓	-	✓	✓	✓
Linguagem	Java	Independente	Java	Smalltalk	C++
Monousuário	✓	✓	✓	✓	✓
Dois usuários	✓	✓	✓	✓	✓
Mais de dois usuários	-	✓	✓	✓	✓
Bloqueio de edição (*navigator*)	✓	-	-	-	✓
Highlight	✓	✓	✓	✓	-
Undo / Redo	-	✓	✓	-	-
Rascunho compartilhado	-	✓	-	-	-
Apontamento de código (telecursor)	-	✓	-	✓	-
Arquitetura					
Cliente / Servidor	✓	✓	-	✓	-
Cliente *web*	-	✓	-	-	-
Ponto-a-ponto	✓	-	✓	✓	✓
Compilação / Execução / Debug					
Suporte à compilação / execução	✓	-	✓	-	✓
Console compartilhado	✓	-	-	✓	✓
Debug	-	-	✓	-	✓
Execução de testes unitários	-	-	✓	-	✓

Tabela 4 – Comparação das funcionalidades oferecidas pelas ferramentas

Conforme apresentado na Tabela 4, as ferramentas oferecem poucos recursos de suporte à comunicação. Em geral, esta atividade utiliza aplicações de terceiros como programas de mensagens instantâneas e voz sobre IP. Além do que, nenhuma das ferramentas avaliadas oferece quadro branco ou vídeo-conferência, as quais são úteis para expor idéias como diagramas de projeto.

Percebe-se ainda que apenas duas ferramentas (Milus e Tukan) apresentam funcionalidades básicas para gerência de atividades e projeto, uma vez que as demais estão focadas apenas na programação.

Além disso, embora todas as ferramentas avaliadas sejam voltadas a *Extreme Programming*, apenas o Cooper e o Flesce implementam o bloqueio de usuário de forma que apenas um programador possa digitar o código por vez (conceito *driver / navigator*). Sob esse aspecto, ambos apresentam soluções complementares: o Cooper oferece bloqueio de todo o arquivo e um "semáforo", e define qual usuário pode editar o código (sinal verde), e qual está apenas acompanhando as alterações (sinal vermelho). Já o Flecse oferece, além do bloqueio de arquivo, o bloqueio em nível de função e o bloqueio em nível linha.

Funcionalidade	**Cooper**	**Milos**	**Sangam**	**Tukan**	**Flecse**	***remotePP***
Geral						
Repositório de arquivos e versões	✓	✓	-	✓	✓	-
Troca de arquivos diversos	-	-	-	-	-	✓
Gerenciamento de projeto	-	✓	-	✓	-	-
Gerenciamento de tarefas	-	✓	-	✓	-	-
Bookmark compartilhado	-	-	-	-	-	✓
Comunicação						
Chat	✓	-	-	✓	✓	✓
Áudio-conferência	-	-	-	✓	✓	✓
Vídeo-conferência	-	-	-	-	-	✓
Quadro branco	-	-	-	-	-	✓
Coleta de informações da sessão	-	-	-	-	✓	✓
Ambiente de programação						
Possui editor próprio	✓	-	✓	✓	✓	✓
Linguagem	Java	Independente	Java	Smalltalk	C++	Independente
Monousuário	✓	✓	✓	✓	✓	✓
Dois usuários	✓	✓	✓	✓	✓	✓
Mais de dois usuários	-	✓	✓	✓	✓	✓
Bloqueio de edição (*navigator*)	✓	-	-	-	✓	✓
Highlight	✓	✓	✓	✓	-	✓
Undo / Redo	-	✓	✓	-	-	-
Rascunho compartilhado	-	✓	-	-	-	-
Apontamento de código (telecursor)	-	✓	-	✓	-	✓
Arquitetura						
Cliente / Servidor	✓	✓	-	✓	-	-
Cliente *web*	-	✓	-	-	-	-
Ponto-a-ponto	✓	-	✓	✓	✓	✓
Compilação / Execução / Debug						
Suporte à compilação / execução	✓	-	✓	-	✓	✓
Console compartilhado	✓	-	-	✓	✓	✓
Debug	-	-	✓	-	✓	-
Execução de testes unitários	-	-	✓	-	✓	-

Tabela 5 – Posicionamento do *remotePP* em relação às ferramentas avaliadas.

A Tabela 5 apresenta o posicionamento do *remotePP* frente às ferramentas avaliadas neste capítulo, e pode-se observar a preocupação com aspectos de comunicação, uma vez que se percebe que este é o principal problema enfrentado por equipes geograficamente distribuídas. Isso ocorre, conforme Kircher *et al* (2001), porque não é possível acompanhar as reações do interlocutor ao receber uma mensagem à distância, o que ocorre naturalmente numa conversa presencial, em que há maior probabilidade da compreensão da mensagem, pois é possível avaliar o comportamento do interlocutor através de gestos, expressões faciais ou mudanças na tonalidade de voz.

O *remotePP* apresenta, ainda, preocupação com recursos do ambiente de programação, em especial, com a necessidade de bloquear um dos usuários para edição do artefato, embora seja importante que ambos possam acompanhar o telecursor do parceiro, a fim de compartilhar seu raciocínio e contribuir para a solução de um problema, funcionalidade essa que também é apresentada pelo *remotePP*.

Após ver-se o posicionamento do *remotePP* em relação às demais ferramentas analisadas, na Tabela 6 vê-se a justificativa das funcionalidades oferecidas por esse ambiente. No quadro, é possível identificar os desafios propostos pelos autores, juntamente com o conjunto de ferramentas desenvolvidas para endereçá-los.

Perspectiva	Desafio	Funcionalidade
Comunicação (ELLIS *et al*, 1991)	Fazer com que interações à distância sejam tão efetivas quando as realizadas face a face.	*Chat*, voz, vídeo, e quadro branco.
Relacionamento interpessoal (MOSCOVICI, 2004) (STOTTS *et al*, 2003)	Promover a compreensão sobre o estilo, preferências pessoais e personalidade do parceiro.	*Bookmark* compartilhado.
Colaboração (BECK, 2004)	Estabelecer um protocolo no qual, enquanto uma pessoa codifica, o outro assume papel mais estratégico.	Bloqueio alternado do comando do editor de código-fonte, telecursor, e rascunho compartilhado.
Infra-estrutura (CROWLEY *et al*, 1990)	Permitir continuidade do trabalho, mesmo em caso de queda da rede.	Ambiente de programação mono e multiusuários. Arquitetura não dependente de servidor centralizado.
Codificação (DEWAN E RIEDL, 1993)	Compartilhar o ambiente de compilação, a fim de obter auxílio para detecção de erros de lógica ou sintaxe.	Compilação com uso de *shell* compartilhado.

Tabela 6 – Justificativa das funcionalidades oferecidas pelo *remotePP*.

3 *RemotePP*

Este capítulo apresenta a análise e projeto do *remotePP* – um ambiente cooperativo para programação em pares distribuídos, sendo que a análise de requisitos identifica as funcionalidades desejáveis à aplicação, em conformidade com a revisão de literatura e análise dos trabalhos relacionados; e o projeto trata da arquitetura necessária para implementação da aplicação.

3.1 Análise de requisitos

Requisitos funcionais são aqueles que abrangem as atividades que o sistema deve realizar. Desta forma, identificam-se os seguintes requisitos como funcionais:

- Disponibilizar edição de códigos-fonte e construção de diagramas não orientados a uma linguagem específica de programação;
- Oferecer recursos de programação em pares, de forma que dois programadores possam trabalhar, ao mesmo tempo, num único código-fonte a partir de locais geograficamente distribuídos;
- Prover recursos de comunicação através de *chat*, voz e vídeo-conferência entre os programadores;
- Apresentar *console* compartilhado para compilação e execução do artefato que está sendo desenvolvido;
- Dispor de acesso ao *bookmark* compartilhado, ou seja, aos endereços favoritos cadastrados por cada programador;
- Permitir troca síncrona de artefatos (arquivos de natureza diversa);
- Manter um banco de dados com as ações realizadas pelos programadores.

Por sua vez, os requisitos não funcionais são aqueles que identificam as características comportamentais ou restrições que a aplicação deve atender. Assim, são requisitos não funcionais do *remotePP*:

- Ser compatível com redes TCP/IP, de modo a ser permitida sua utilização em ambiente Internet ou Intranet;
- Ser adaptável ao limite de banda disponível na rede;
- Oferecer mecanismos de integridade dos códigos-fonte;
- Implementar padrões de interface;
- Possuir utilização em múltiplas plataformas operacionais;
- Apresentar fácil adaptação para ferramentas que possam ser desenvolvidas futuramente.

3.1.1 Diagrama de casos de uso

O diagrama de casos de uso apresenta os atores (ou usuários do sistema) e suas interações com o sistema numa visão de alto nível. Para melhor visualização, a Figura 10 ilustra o diagrama de casos de uso desenvolvido a partir dos requisitos funcionais.

Dessa forma, no diagrama é possível identificar o ator "Programador" como o principal usuário que a aplicação visa atender, o qual pode pode, ainda, ser especializado na forma de *driver* (que mantém o controle da tarefa que está sendo realizada) ou *navigator* (que acompanha e auxilia o parceiro num nível mais estratégico).

Identifica-se ainda que o ator "Remoto" caracteriza o sistema *remotePP* com o qual o outro programador (local) encontra-se trabalhando. Sob o ponto de vista da aplicação, assume o papel de um sistema externo, e representa o programador localizado em outro local.

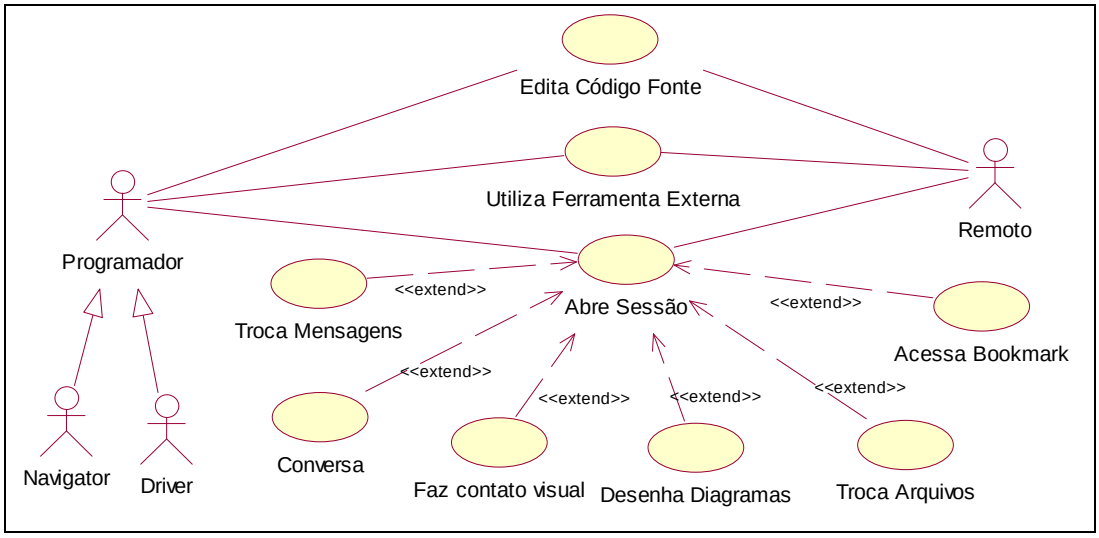

Figura 10 – Diagrama de casos de uso.

Assim, nota-se que os atores podem assumir os seguintes papéis:

- Programador – é o usuário geral do sistema. O termo programador, neste contexto, também pode ser atribuído a projetista, e não deve ser unicamente interpretado como codificador (pessoa responsável exclusivamente pela edição de código-fonte);
- *Navigator* e *driver* – papel assumido pelo programador quando se estabelece um protocolo entre as atividades desempenhadas pelo programador local e seu parceiro remoto. O *navigator* acompanha atentamente o que o *driver* está codificando ou projetando, e pode oferecer alternativas de solução ou realizar pesquisas em bases de conhecimento como a Internet. Os papéis entre d*river* e *navigator* podem ser alternados durante a sessão, sendo que essa convenção é utilizada para evitar conflitos ao editar documentos ao mesmo tempo, embora, tudo que o *driver* realiza é acompanhado, em tempo real, pelo *navigator*;
- Remoto – é a instância longínqua da aplicação, que representa um ambiente idêntico ao apresentado no restante do diagrama.

3.2 Projeto

Segundo Pressman (2001), o projeto de *software* visa a transformar os requisitos levantados em tempo de análise em representações de *software*. Ainda segundo o autor, tal atividade é desenvolvida numa série de interações: da preliminar até a mais detalhada. Sendo assim, os documentos e diagramas apresentados a partir deste capítulo seguem essa orientação, ou seja, inicialmente será apresentado um esboço com traços gerais, e, posteriormente, será apresentado maior detalhamento das funcionalidades da aplicação.

A primeira representação do sistema é apresentada na Figura 11, em que um problema é apresentado a dois programadores localizados em regiões geográficas diferentes, e ambos devem encontrar uma alternativa de solução viável para o problema. Para tanto, recorrem a seus conhecimentos prévios ou experiências anteriores, depois, essas informações são trocadas entre os envolvidos de modo a encontrar uma solução única. Neste contexto, o *remotePP* deve assumir papel de facilitador técnico capaz de permitir uma interação eficaz entre os programadores, e sua aplicação deve oferecer tanto suporte a tarefas em nível estratégico e cognitivo, como a comunicação por exemplo, quanto em nível técnico, através de um ambiente de programação e diagramação de representações gráficas.

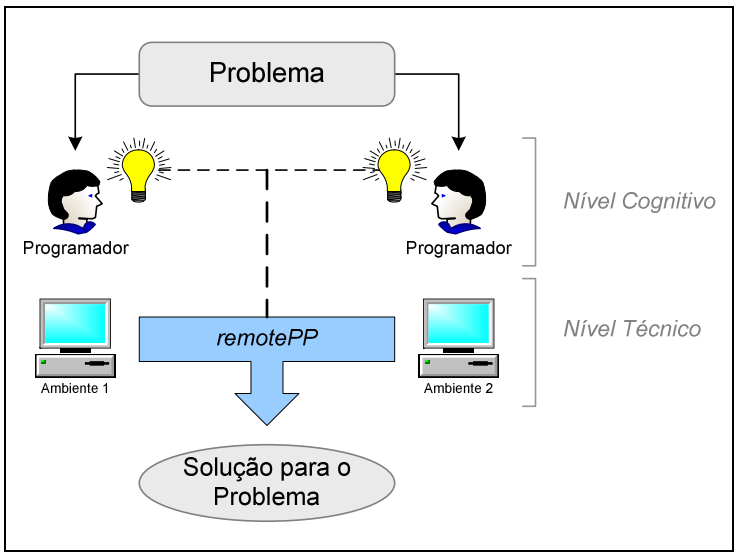

Figura 11 – Visão geral do sistema.

Então, a Figura 12 expande o conceito de comunicação e apresenta os periféricos computacionais utilizados pelo sistema para viabilizar a troca de informações e a construção da solução.

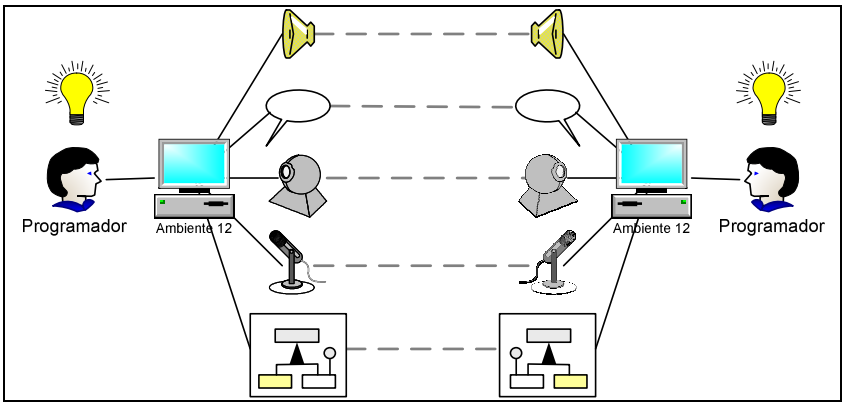

Figura 12 – Interfaces utilizadas pelo sistema.

3.2.1 Arquitetura

Pressman (2001) define a arquitetura de *software* como a estrutura hierárquica dos módulos que compõem um programa de computador e a estrutura dos dados. Além disso, o autor presume que o problema deve sofrer um processo de desagregação, ou seja, cada "subproblema" será endereçado por um módulo específico construído para ser tão auto-suficiente quanto for possível.

Sendo assim, a Figura 13 apresenta a organização dos módulos do *remotePP*, em que cada bloco representa a solução para um problema específico, e a ligação entre os módulos indica o relacionamento entre os mesmos.

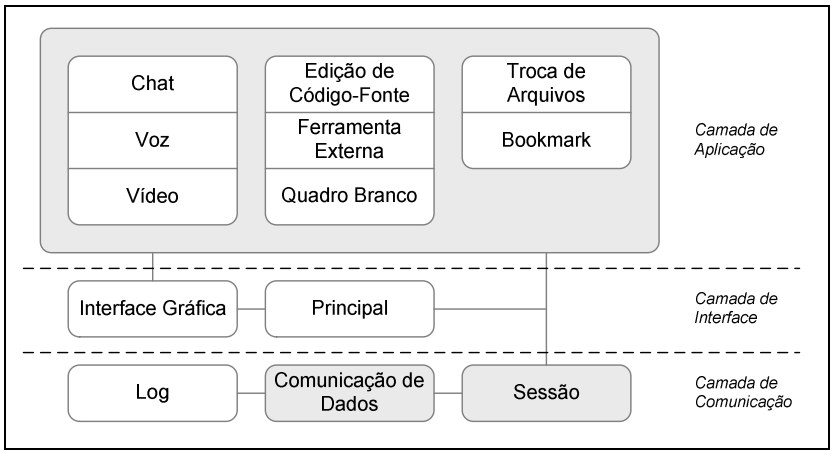

Figura 13 – Arquitetura do sistema.

- Comunicação de dados – é o módulo responsável pela troca de pacotes entre a aplicação local e o sistema que é executado remotamente. Ele implementa aspectos e algoritmos básicos de comunicação de dados de forma comum, independente das informações enviadas ou recebidas.
- *Log* – é acoplado ao módulo de comunicação para oferecer uma forma única de persistência do tráfego gerado durante a comunicação, e é responsável pela formação da base de dados com as mensagens enviadas ou recebidas pelo sistema local e remoto.
- Sessão – uma sessão representa o tempo em que dois computadores são mantidos em comunicação síncrona. Esse módulo é responsável pelo início, gerenciamento, finalização e notificações dos estados da sessão.

- Interface gráfica – mantém *templates* e componentes comuns de interface gráfica com o usuário, tais como: janela, barra de botões, caixas de diálogo, etc. E, ainda, procura oferecer um conjunto de rotinas reutilizáveis e padronizadas para toda a aplicação.
- Principal – é o ponto de entrada da aplicação e oferece os mecanismos para o usuário realizar todas as atividades do sistema. Para tanto, utiliza o módulo de interface gráfica para estabelecer contato visual com o usuário e pode desencadear novas sessões ou atividades de comunicação.

Os módulos a seguir compreendem as atividades de colaboração ente os usuários, sendo que todos utilizam as definições da interface gráfica e dependem da existência das rotinas especificadas pelo módulo de sessão:

- *Chat*, voz e vídeo – compreendem os módulos para troca de mensagens instantâneas de texto, comunicação por voz e comunicação através de *webcam* (vídeo-conferência);
- Edição de código-fonte, ferramenta externa e quadro branco – oferecem recursos para a edição de programas e o controle da execução de ferramentas externas, tais como compiladores por linha de comando. O módulo de quadro branco disponibiliza uma forma de comunicação visual entre os programadores, pois permite que pensamentos possam ser expressos livremente através de representações gráficas, sem que tenhamos o formalismo de uma linguagem específica;
- Troca de arquivos e *bookmark* – o módulo de troca de arquivos permite que arquivos binários alheios à aplicação sejam enviados ou recebidos, sendo que o fluxo e o tamanho dos pacotes são configurados de acordo com a largura da banda disponível. Já o módulo de *bookmark* permite o compartilhamento de endereços de *links* da Internet utilizados freqüentemente pelo usuário, possibilitando uma forma simples de conhecer os gostos pessoais do interlocutor.

As seções a seguir especificam os módulos individualmente e suas interações, sendo que, para cada módulo, podem ser utilizadas as seguintes representações:

- Detalhamento de casos de uso – o detalhamento dos casos de uso apresenta, de forma expandida, os cenários de cada funcionalidade do sistema. Assim, cada caso de uso é composto por:
  - Objetivo – mostra a visão geral da meta que deve ser atingida com o caso de uso;
  - Pré-condições – estabelece as condições que devem ser satisfeitas antes que o caso de uso possa ser iniciado;
  - Fluxo básico – compreende todos os passos que devem ser realizados durante a execução normal do caso de uso, e é apresentado na forma de uma tabela com as ações do ator e as respostas do sistema. Por tratar de uma aplicação ponto-a-ponto, em alguns casos de uso, tanto as ações do ator, como as respostas do sistema, são identificadas como “local” ou “remoto”;
  - Fluxos alternativos – situações de exceção (previsíveis) que podem ser tratadas durante o fluxo básico, mas que não caracterizam o caminho normal de execução do caso de uso;

- Pós-condições – indica as condições que devem ser satisfeitas para que o caso de uso seja considerado concluído com êxito.

- Diagrama de classes – conforme Booch *et al* (2000), os diagramas de classes permitem uma visão estática do projeto de um sistema, e, ainda conforme os autores sugerem, os diagramas de classes deste sistema foram utilizados para a geração dos códigos de construção.
- Diagrama de estados – é uma representação dinâmica que permite identificar os estados transitórios dos objetos ao longo da execução do sistema.
- Protótipo – apresenta um esboço das telas do sistema, em que são sugeridas propostas de solução para interface com o usuário das principais funcionalidades, e cujo objetivo é propor diretrizes de interface que deverão ser seguidas em toda a aplicação e não prototipar toda a aplicação detalhadamente.

3.3 Módulo de interface gráfica

A interface gráfica com o usuário assume um papel importante em qualquer sistema computacional por possuir impacto direto na usabilidade e qualidade do *software*. Então, a Figura 14 apresenta um esboço da tela principal do *remotePP*, em que é possível identificar os principais componentes visuais que apoiarão as atividades do programador.

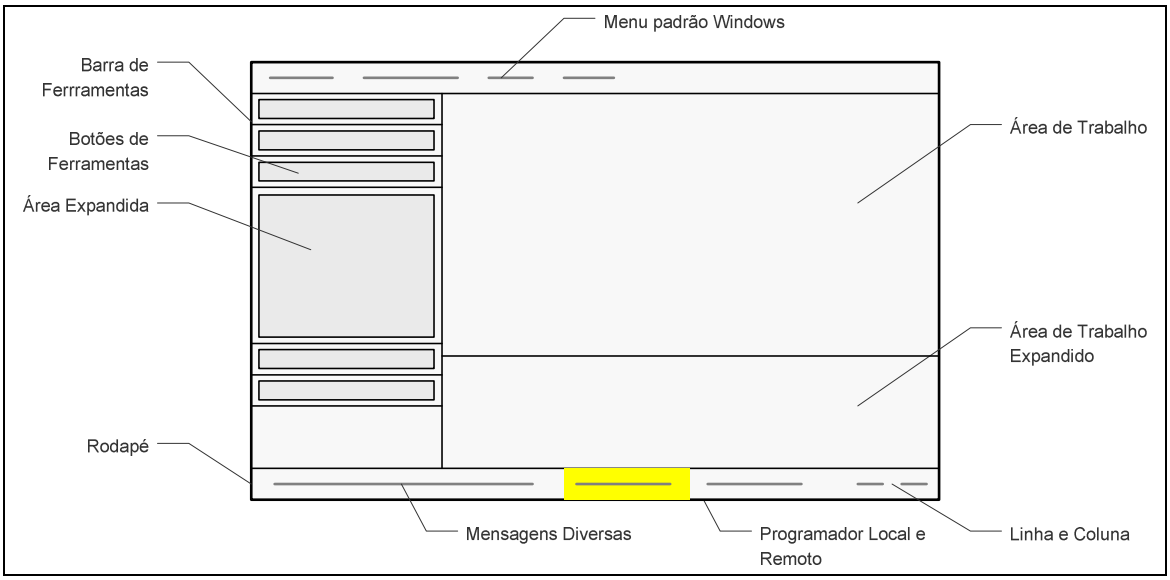

Figura 14 – *Layout* geral de interface do *remotePP*.

Os elementos gráficos possuem os seguintes objetivos:

- Menu padrão Windows – apresentar um menu de acesso a funcionalidades como gerenciamento de arquivos, edição de texto, visualizações e ferramentas;
- Barra de ferramentas – agrupar as funcionalidades de sessão e comunicação, sendo que a barra fica disponível todo tempo e oferece uma forma rápida de acesso às atividades de comunicação;
- Botões de ferramentas – indicar as ferramentas disponíveis no momento, sendo que as funcionalidades não disponíveis ficam desabilitadas;

- Área expandida – recurso cujo objetivo é ser utilizado quando uma determinada funcionalidade requer pouco espaço em tela (como *chat* ou vídeo-conferência). Assim, as diversas áreas expandidas aparecem e desaparecem reorganizando os demais botões de ferramentas;
- Rodapé – mostrar informações úteis como programadores atuais na sessão, linha e coluna do código-fonte, avisos, etc;
- Área de trabalho – é a maior área disponível e seu objetivo é o de ser utilizada para edição de códigos-fonte ou desenho de diagramas;
- Área de trabalho expandida – apresentar as funcionalidades diretamente vinculadas ao artefato apresentado na área de trabalho (posicionado logo acima), como, por exemplo a execução de ferramentas externas como compiladores por linha de comando;
- Mensagens diversas – apresentar avisos e mensagens de erro;
- Programador local e remoto – indicar o *login* do programador local e do programador remoto (quando uma sessão estiver estabelecida);
- Linha e coluna – mostrar o número da linha e da coluna cujo cursor estiver posicionado dentro da área de trabalho.

Conforme visto anteriormente, o módulo de interface gráfica é responsável pelas padronizações do ambiente de interação com o usuário, e oferece componente e rotinas úteis a toda a aplicação.

Barra de ferramentas e menu padrão Windows

A Figura 15 apresenta o diagrama de classes utilizadas para representar a barra de ferramentas, botões de ferramentas e menu padrão Windows.

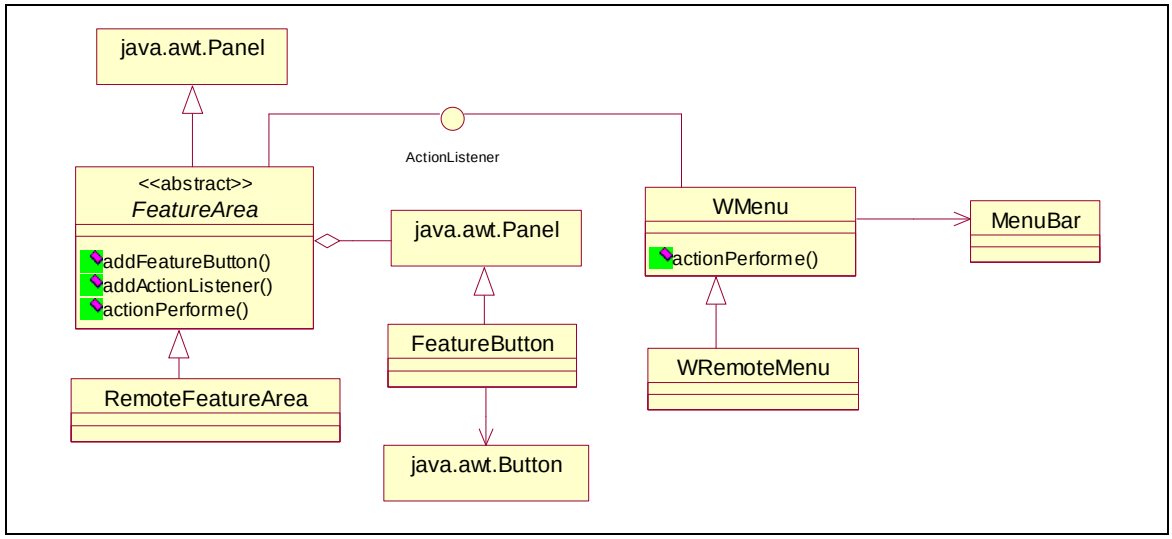

Figura 15 – Diagrama de classes da barra de ferramentas.

A classe abstrata *FeatureArea* deriva de *Panel* a fim de possuir a capacidade de se acomodar como um componente visual na janela principal da aplicação, e implementa a interface *ActionListener* para oferecer um gatilho unificado disparado por todos os seus botões e painéis.

A classe *FeatureButton* também especializa uma *Panel* e possui uma única instância de *Button* para oferecer mecanismo de exibição e tratamento do pressionamento dos botões.

Já o relacionamento de agregação entre *FeatureArea* (todo) e *Panel* (parte) permite que *FeatureButtons* ou *Panels* sejam adicionados à barra de ferramentas.

Por sua vez, a classe concreta *RemoteFeatureArea* acrescenta as instâncias de todos os botões de ferramentas utilizados pelo *remotePP*.

Finalmente, o conjunto formado pelas classes *WMenu*, *MenuBar* e *WRemoteMenu* oferece mecanismos para criação dos menus da janela principal, onde *WMenu* também implementa a interface *ActionListener* a fim de promover o mesmo tratamento unificado quando o usuário selecionar qualquer item do menu. Além disso, todos os menus são instanciados em *WRemoteMenu*.

Janelas e caixas de diálogo

As classes responsáveis pela apresentação das janelas e caixas de diálogo do sistema são apresentadas no diagrama de classes da Figura 16.

A classes *WSplash* é apresentada durante a inicialização do sistema, enquanto os módulos são carregados em memória, e possui uma instância de *java.awt.Image* a fim de carregar uma imagem com os dados de identificação do projeto.

Já a classe *WMain* deriva de *java.awt.Frame* e implementa interfaces de controle de janelas e ativação de ações (*WindowListener e ActionListener*). Além disso, ela será utilizada pela tela principal da aplicação (módulo principal).

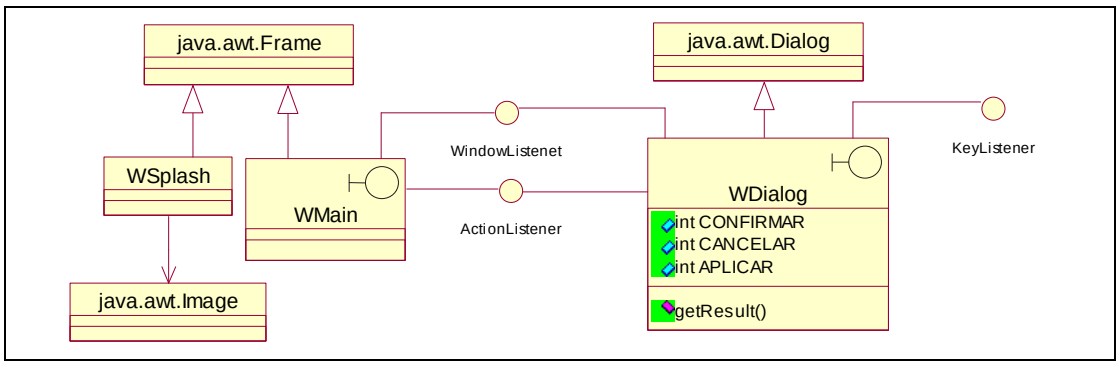

Figura 16 − Diagrama de classes de janelas e caixas de diálogo.

A classe *WDialog* (especialização de *java.awt.Dialog*) é responsável pela padronização de todas as caixas de diálogo do tipo “confirmar/cancelar”, conforme esboçado pela Figura 17, onde pode-se observar que o estereótipo de *boundary* foi aplicado às classes *WMain* e *WDialog* para reforçar o caráter de ligação direta junto ao usuário final.

Figura 17 − Caixa de diálogo padrão.

3.4 Módulo de comunicação de dados

Assim como o módulo de interface gráfica é importante ao usuário final, o módulo de comunicação assume papel de extrema importância para o bom funcionamento da aplicação, pois ele estabelece arcabouço de classes e interações utilizadas por todas as ferramentas de comunicação.

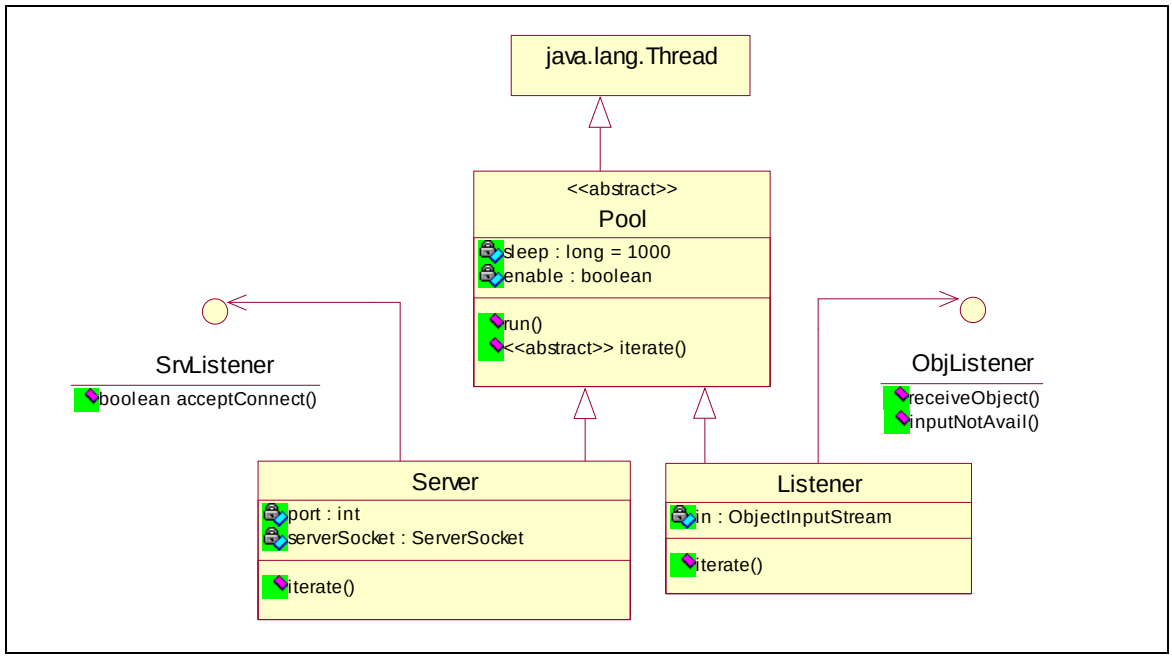

Figura 18 − Diagrama de classes básicas de comunicação.

Server* e *Listener

A classe abstrata *Pool* (Figura 18) está no topo das classes básicas de comunicação,e especializando uma *Thread*, evoca o método *iterate* a cada *sleep* milisegundos, desde que esteja habilitada (*enable = true*).

Já a classe concreta *Server* (derivada de *Pool*) é responsável por tratar os pedidos de conexão advindos de um computador remoto. Assim, sempre que uma nova conexão é recebida, o método *acceptConnect* (da interface *SrvListener*) é executado a fim de avaliar se é permitido estabelecer a nova conexão, e o atributo *port* identifica a porta TCP/IP utilizada para comunicação.

Por outro lado, a classe *Listener*, que também deriva de *Pool*, é responsável por monitorar o *stream* de entrada e receber todos os objetos recebidos pela conexão remota. Assim, cada vez que um objeto é recebido, o método *receiveObject* é evocado a partir da interface *ObjListener*, e, durante esse monitoramento, se a conexão for perdida, o método *inputNotAvail* (da interface *ObjListener*) é executado a fim de promover à classe que a implementa os tratamentos pertinentes.

Communicator

A classe abstrata *AbstractCommunicator* (Figura 19) faz uso de uma instância de *Server* e outra de *Listener*, e define os atributos de estado (ST_), que são responsáveis pela identificação do estado de uma comunicação, os quais podem ser:

- ST_NA (*state not available*) – estado que indica que não há comunicação ativa e nenhuma nova conexão recebida por computador remoto será aceita.

- ST_WAIT_CONNECT – esse estado indica que não há conexão ativa, porém, o *Server* está monitorando e tratando novas conexões.
- ST_WAIT_ID – indica que uma nova conexão foi estabelecida, e a classe de comunicação está aguardando as credenciais do computador remoto. Caso as credenciais não sejam recebidas, a comunicação cai por *time-out*.

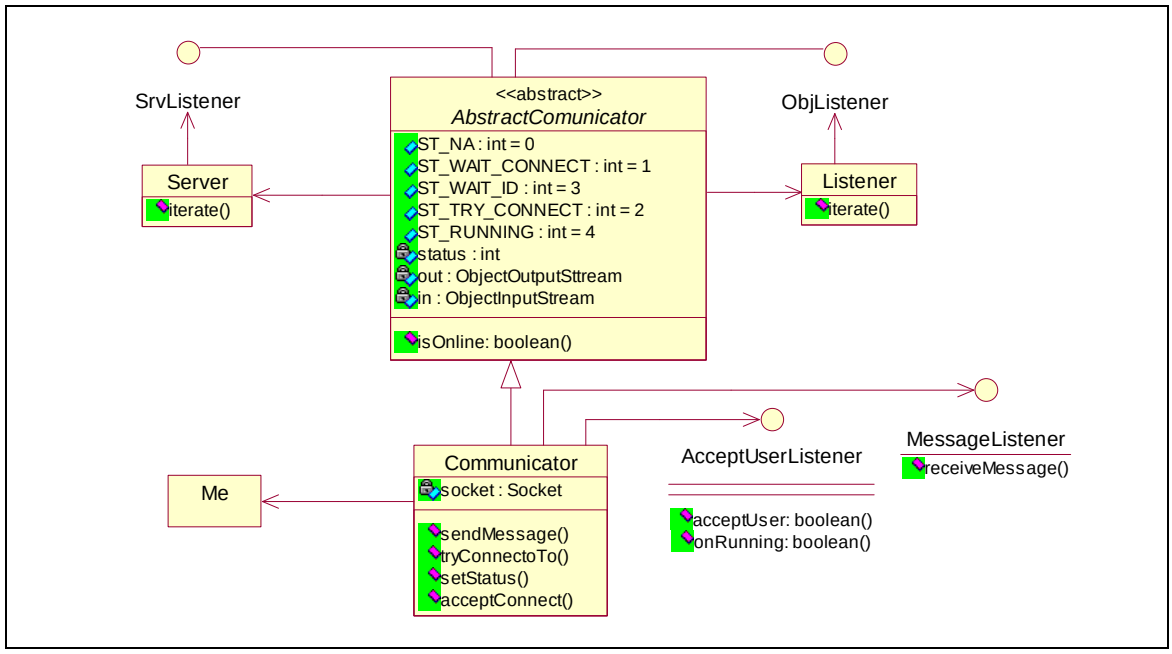

Figura 19 − Diagramas de classes de comunicação.

- ST_TRY_CONNECT – indica que uma nova comunicação foi solicitada a um computador remoto, porém, este ainda não o aceitou.
- ST_RUNNING – indica que há uma conexão ativa com o computador remoto.

A máquina de estados (diagrama de estados) que um objeto da classe de comunicação pode assumir é mostrada pela Figura 20.

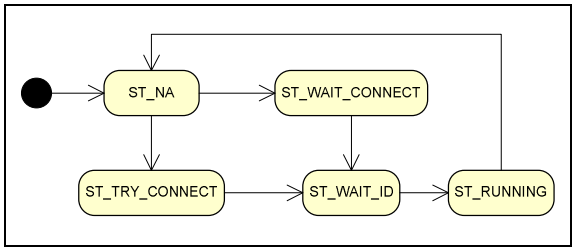

Figura 20 − Diagrama de estados de comunicação.

O módulo de comunicação define, através do diagrama de classes da Figura 21, as mensagens de infra-estrutura, sendo que a classe *AbstractMessage* implementa a interface *Serializable* de modo a permitir a serialização de objetos derivados, e possui apenas o atributo *forceClose* utilizado quando uma mensagem deve forçar o término do canal de comunicação. Portanto, esse recurso é útil quando é necessário transferir artefatos de tamanho fixo.

Por sua vez, a classe *UserID* representa as credenciais de um usuário (local ou remoto), e a classe *singleton Me* mantém uma única instância disponível que identifica apenas o usuário local.

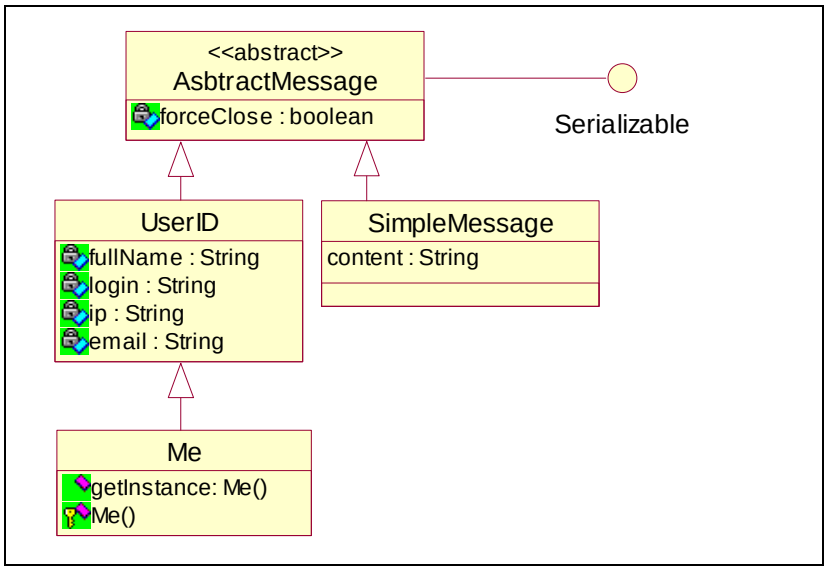

Figura 21 − Classes básicas de mensagens.

3.5 Módulo de *log*

O módulo de *log* é responsável pela persistência das mensagens trocadas durante a sessão em um banco de dados baseado em XML. Por questões de limitação deste trabalho, não é realizado nenhum gerenciamento ou extração de conhecimento a partir das bases geradas.

A Figura 22 apresenta o diagrama de classes do módulo, em que pode-se notar que as mensagens são persistidas através de acoplamento à classe *Communicator* (método *sendMessage*) e interface *MessageListener* (método *receiveMessage*).

É importante salientar que nem todas as mensagens trocadas durante a sessão deverão ser persistidas, como, por exemplo, o reposicionamento do cursor na tela, que não deve ser persistido. Para realizar tal distinção, foi criada a interface *Logable* (sem nenhum método), e somente as mensagens que implementam esta interface serão salvas na base de dados.

No entanto, para evitar sobrecarga no acesso a disco, as mensagens são enfileiradas (atributo *queue*) e descarregadas (método *flush*) somente quando a fila fica cheia ou é passado algum tempo pré-estabelecido.

Assim, as mensagens são transformadas de instâncias de objetos em código XML através da classe *LogLayout*, que identifica o remetente (*source*), o destinatário (*receiver*), a ferramenta de transmissão/recepção (*feature*), e o conjunto de atributos enviados/recebidos (*data*).

O *layout* completo da mensagem no formato XML é apresentado na Figura 23.

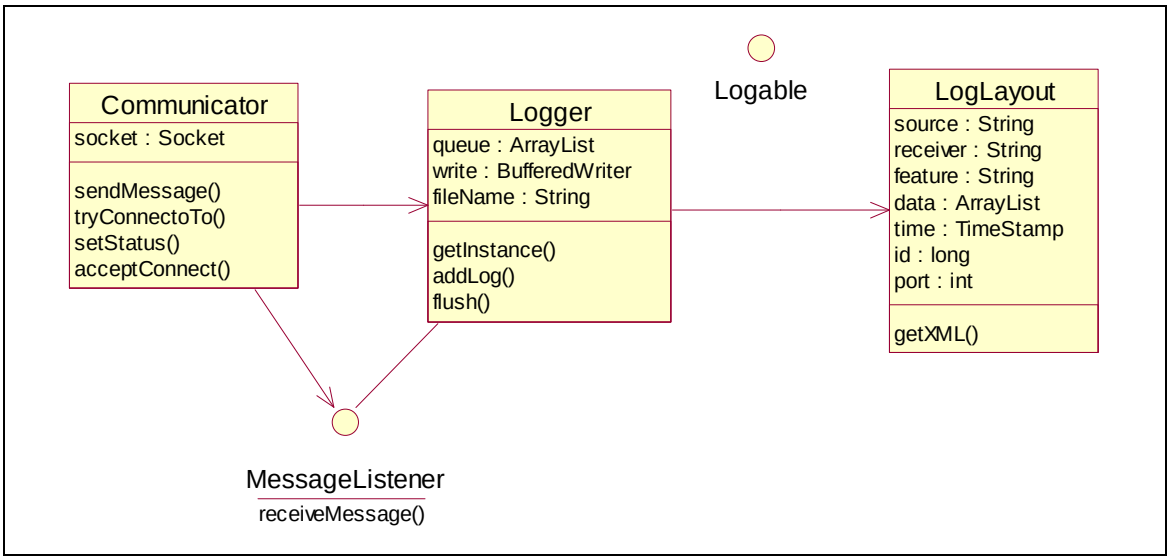

Figura 22 − Diagrama de classes do módulo de *log*.

```
<RemotePP_Message
  ID=1255448,
  Time=”08/10/2006 16:18:01.015”
  Feature="Chat",
  Port=1001,
  Direction=”Send”
  Source="byrobson, 192.168.0.10",
  Receiver="gilsonz, 192.168.0.20",
  field0="content",
  data0="utilize o operador instanceof para identificar se o objeto
implementa a interface!!! ;)"
/>
```

Figura 23 − *Layout* da mensagem em formato XML.

3.6 Módulo de sessão

O módulo de sessão é responsável pela conexão inicial entre os programadores (local e remoto).

3.6.1 Caso de uso: abre sessão

Objetivo

Iniciar uma nova sessão através da rede com um parceiro localizado em computador remoto.

Pré-condições

O programador deve ter registrado seu nome completo, *login* e *e-mail* através do menu: “Exibir > Perfil do Usuário”. O *login* do usuário local aparece no canto inferior direito da tela.

Fluxo básico

Ações do ator	Respostas do sistema
1. (local) Acessa botão de ferramenta: "Iniciar sessão"	2. (local) Apresenta uma tela a fim de obter o *login* do programador convidado e seu endereço IP.
3. (local) Informa *login* e endereço IP do parceiro.	4. (local) Aguarda pela resposta da aplicação remota. (remoto) Apresenta tela com identificação do programador que deseja iniciar a sessão.
5. (remoto) Aceita o início da sessão.	6. (remoto) Fecha a tela de identificação e habilita as ferramentas de comunicação. (local) Habilita as ferramentas de comunicação.

Fluxo alternativo

Em (4), caso o computado não responda ao convite, a mensagem "Falha na conexão" é apresentada e o caso de uso é encerrado.

Pós-condições

A conexão é estabelecida com êxito e o *login* do parceiro aparece no canto inferior direito da tela, junto ao nome do programador local.

Pontos de extensão

Os seguintes casos de uso são disponibilizados a partir deste:

- Troca mensagens
- Conversa
- Faz contato visual
- Desenha diagrama
- Troca arquivos

3.6.2 Diagrama de classes

O módulo de sessão estabelece os controles das interações entre a interface com o usuário e as classes de comunicação (Figura 24). Sendo que as classes de controle de sessão implementam o padrão de desenvolvimento *façade* com o objetivo de permitir a expansão do sistema sem aumentar a sua complexidade de compreensão e manutenção (COOPER, 1998). Desta forma, instâncias de novas classes de comunicação devem ser aderidas apenas à classe de controle e ao acesso.

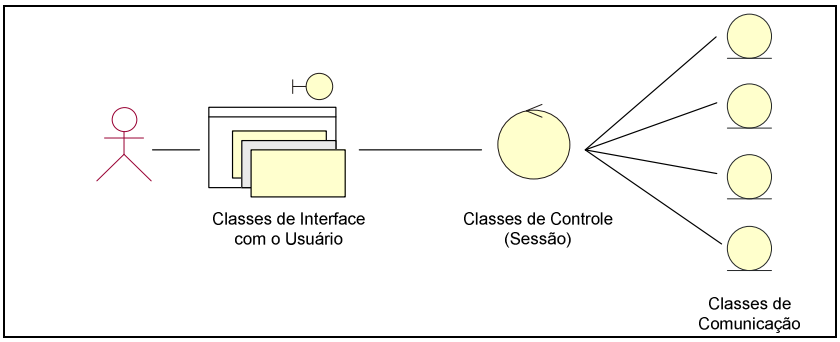

Figura 24 − Relacionamentos do módulo de sessão.

A Figura 25 apresenta as classes do módulo de sessão, sendo que, no diagrama, é possível identificar a interação com o módulo de comunicação (já visto anteriormente).

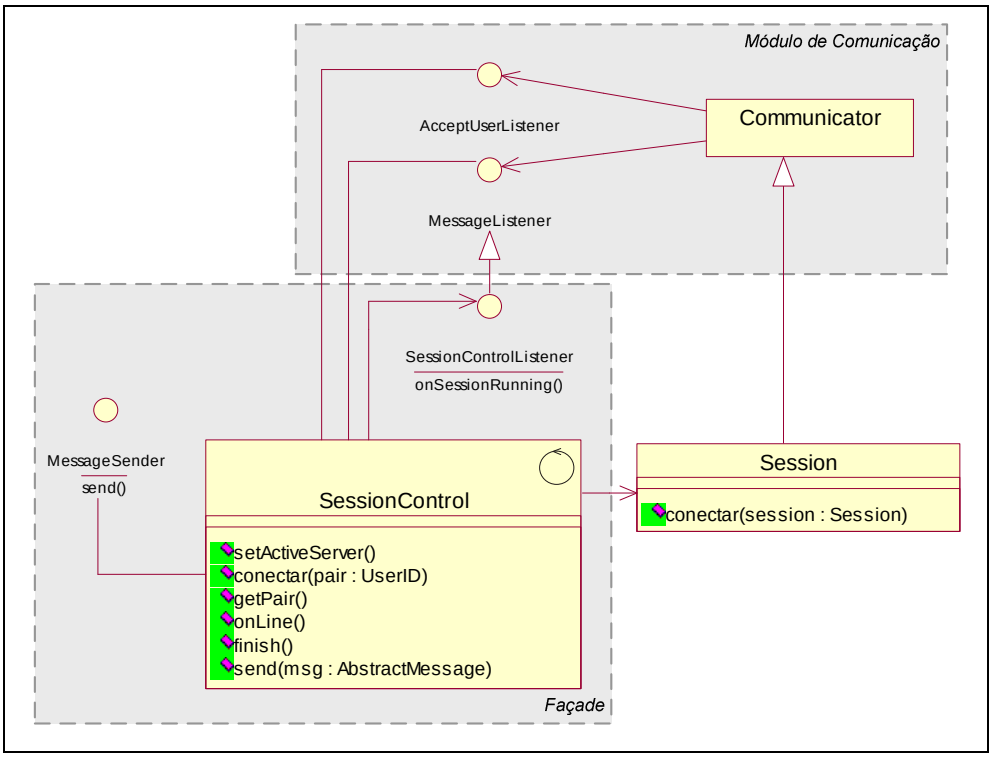

Figura 25 − Diagrama de classes do módulo de sessão.

A classe S*ession* deriva de *Communicator* e implementa um novo método *conectar*, que será utilizada posteriormente pelos módulos de ferramentas específicas (*chat*, *editor*, *etc*.).

Por sua vez, a classe *SessionControl* é a espinha dorsal (*façade*) da aplicação, pois transfere as solicitações da interface do usuário para o módulo de comunicação (através de instâncias de *Session*), além de realizar o caminho inverso, implementando suas interfaces (*AcceptUserListener*, *MessageListener*).

Os principais métodos de *SessionControl* são descritos a seguir:

- *setActiveServer* – prepara a sessão para atuar como servidor (passa para o estado *ST_WAIT_CONNECT*), aguardando e respondendo a pedidos de conexão advindos de um computador remoto (agindo passivamente);

- *conectar* – toma a iniciativa de estabelecer uma nova conexão (ou *Session*) com um computador remoto (agindo ativamente). Para tanto, recebe como argumento uma instância de *UserID*, o qual indica o endereço IP do computador de destino;
- *getPair* – depois de estabelecida a conexão (seja ativa ou passivamente), esse método pode ser utilizado para obter a identificação do computador remoto;
- *onLine* – avalia o estado da comunicação (ativa ou inativa, *on-line* ou *off-line*);
- *finish* – termina todas as conexões estabelecidas;
- *send* – envia mensagens (*AbstractMessage*) para o computador remoto.

A classe ainda implementa as seguintes interfaces:

- *AcceptUserListener* – responde ao módulo de comunicação sempre que um computador remoto solicitar abrir uma nova sessão. Naturalmente, será necessário solicitar permissão do usuário através de caixas de diálogo;
- *MessageListener* – trata as mensagens recebidas pelo módulo de comunicação, encaminhando-as à sessão correspondente;
- *SessionControlListener* – monitora o estado das sessões, de forma a tratar queda ou necessidades de ajuste como redução na banda utilizada;
- *MessageSender* – distribui as mensagens oriundas da interface para a sessão correspondente.

Embora o diagrama apresentado na Figura 25 apresente apenas uma classe *Session*, esta será composta por uma série de instâncias, de acordo com o número de ferramentas disponibilizadas (*chat*, editor de código, etc.). Isso garante que diferentes *features* possuam canais próprios de comunicação, mas sejam tratados pela mesma *SessionControl*.

Além disso, é importante que cada módulo de comunicação possua suas próprias classes de mensagem (derivadas de *AbstractMessage*). Isso é necessário, pois a classe de controle *SessionControl* distribui as mensagens para as classes de comunicação de acordo com o objeto instanciado, como pode ser observado no trecho de código do método *send* (Figura 26).

```
public void send(AbstractMessage msg) {
  if (chat.isOnline() && msg instanceof ChatMessage)
     chat.sendMessage(msg);
  else if (console.isOnline() && msg instanceof ConsoleMessage)
     console.sendMessage(msg);
  else if (editor.isOnline() && msg instanceof EditorMessage)
     editor.sendMessage(msg);
  else if (transfer.isOnline() && msg instanceof TransferMessage)
     transfer.sendMessage(msg);
  else if (bookmark.isOnline() && msg instanceof BookmarkMessage)
     bookmark.sendMessage(msg);
  else ...
}
```

Figura 26 − Trecho de código do método *SessionControl.send()*.

3.7 Módulo principal

O módulo principal oferece o ambiente de trabalho do sistema na forma de um contêiner onde os componentes visuais da aplicação são dispostos. O protótipo da tela principal é apresentado na Figura 27, e sua interface foi projetada de forma a permitir que o usuário possua acesso direto a todas as principais funcionalidades de comunicação (à esquerda) e edição de código-fonte ou diagramas (à direita), sendo que, a medida que as funcionalidades são ativadas, os botões são habilitados.

Na parte inferior, o usuário acompanha o *status* da sessão atual ou recebe avisos do sistema. Na medida do possível, as mensagens são apresentadas nesta barra de modo a não desviar a atenção dos artefatos que estão sendo editados.

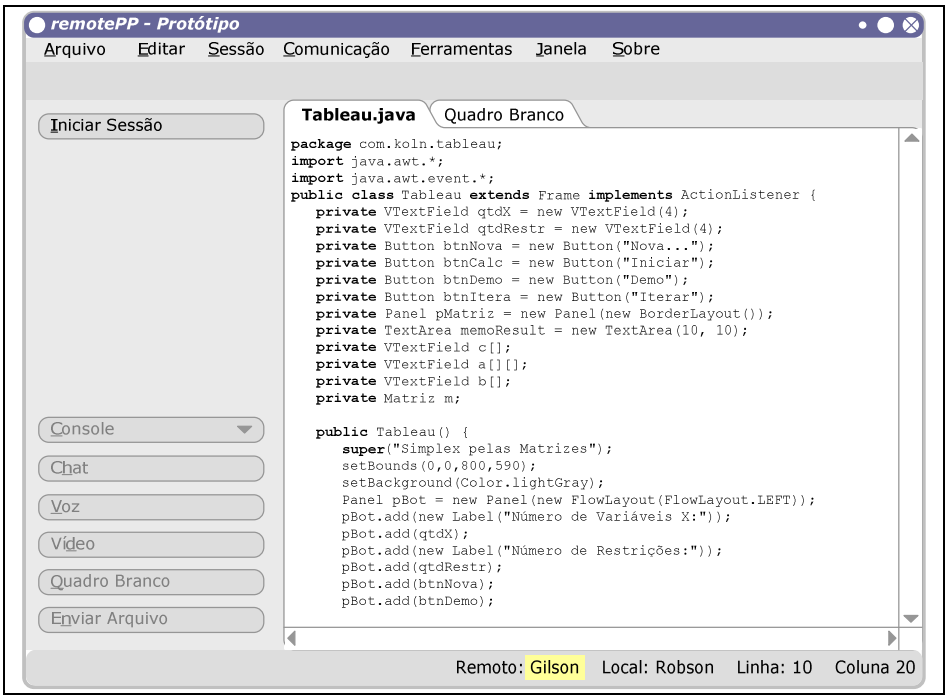

Figura 27 − Protótipo da tela principal.

3.7.1 Diagrama de classes

A Figura 28 apresenta o diagrama de classes do módulo principal, que é responsável pela apresentação da aplicação ao usuário. No diagrama, é possível identificar a classe *WRemotePP*, derivada de *WMain*, apresentando atributos para instâncias de:

- *WRemoteMenu* – menu principal da aplicação (padrão Windows);
- *RemoteFeatureArea* – contêiner com os botões para as ferramentas disponibilizadas;
- *UserDialog* – caixa de diálogo para que o usuário cadastre sua identificação (nome, *login* e *e-mail*);
- *SessionDialog* – caixa de diálogo para entrada dos dados do computador remoto (*login* e IP);
- *BookmarkDialog* – caixa de diálogo para cadastramento de *links* de endereços favoritos na Internet;

- *PanelBar* – barra de rodapé que apresenta informações úteis como *login* do usuário local e remoto, linha e coluna do editor de textos;
- *SessionControl* – instância para a classe de controle de comunicação (já vista no módulo anterior).

A classe *WRemotePP* implementa a interface *SessionControlListener*, de forma a atender aos eventos advindos das sessões (via *SessionControl*).

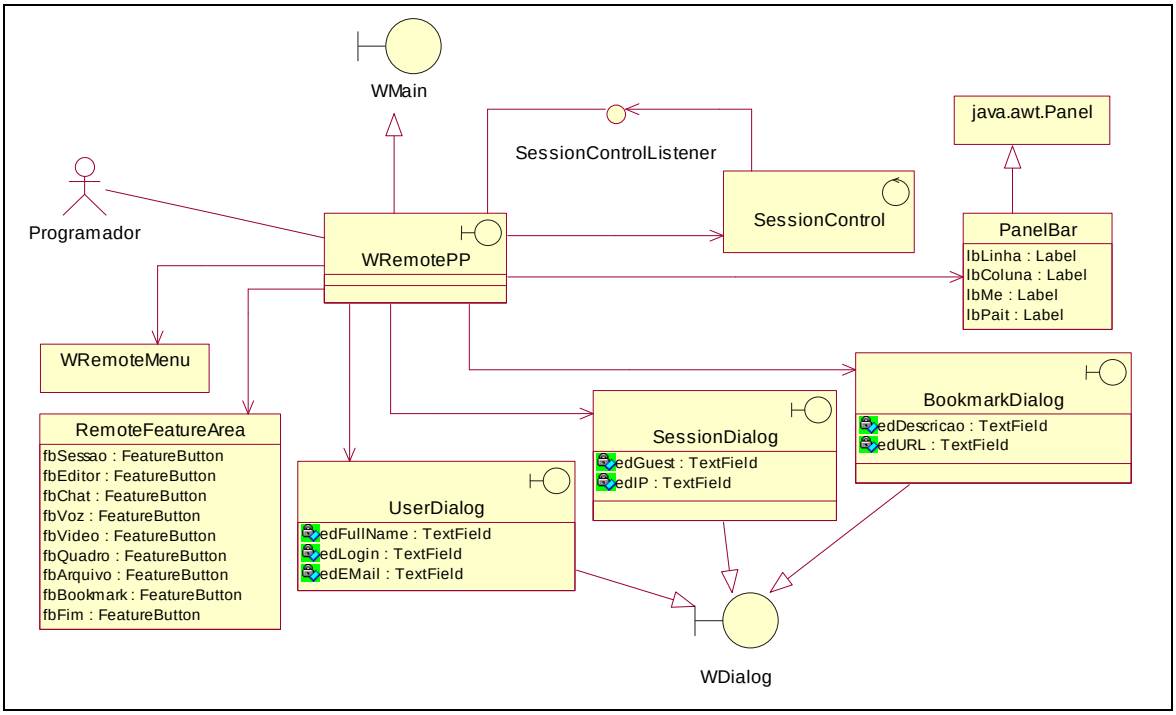

Figura 28 − Diagrama de classes do módulo principal.

3.8 Módulo de *chat*

O módulo de *chat* representa a ferramenta de comunicação mais simples da aplicação: um comunicador de mensagens instantâneas por linha de texto.

3.8.1 Caso de uso: troca mensagens

Objetivo

Descrever o funcionamento da funcionalidade que permite a comunicação entre os programadores (local e remoto) através de mensagens instantâneas.

Pré-condições

Uma sessão deve ser estabelecida entre o programador local e remoto.

Fluxo básico

Ações do ator	Respostas do sistema
1. (local) Acessa botão de ferramenta: "*Chat*"	2. (local) Estabelece uma nova conexão com o sistema remoto.
	3. (local / remoto) Expande a área abaixo do botão "*chat*" na barra de ferramentas e apresenta o painel para envio e recebimento das mensagens (Figura 29).
	4. (local) Posiciona o cursor no campo para digitação de mensagens.
5. (local) Digita a mensagem e pressiona <*Enter*> ou clica no botão <Enviar>.	6. (local) Envia a mensagem ao sistema remoto e a inclui na caixa de texto das mensagens enviadas e recebidas
	7. (remoto) Recebe a mensagem do sistema local e a apresenta na caixa de texto das mensagens enviadas e recebidas. As mensagens recebidas são realçadas para diferenciar das mensagens enviadas.
8. (remoto) Posiciona o cursor no campo de digitação das mensagens. Escreve a resposta. Pressiona <Enter> ou clica no botão <Enviar>.	9. (remoto) Envia a mensagem ao sistema local e a inclui na caixa de texto das mensagens enviadas e recebidas
	10. (local) Recebe a mensagem do sistema remoto e a apresenta na caixa de texto das mensagens enviadas e recebidas. As mensagens recebidas são realçadas para diferenciar das mensagens enviadas.
11. Retorna ao passo 5 (local) ou 8 (remoto) enquanto houver interesse na troca de mensagens.	
12. (local / remoto) Acessa o botão de ferramenta para encerrar a comunicação: "*Chat*"	13. (local / remoto) Finaliza a conexão e fecha a área de troca de mensagens instantâneas, reorganizando os botões de ferramentas. Encerra o caso de uso.

Fluxo alternativo

- Em (2), caso a conexão com o sistema remoto não seja estabelecida, apresenta a mensagem: "Falha de comunicação com o sistema remoto" e encerra-se o caso de uso.
- A qualquer momento: em caso de queda da conexão, pula para o passo (13).

Pós-condições

Não se aplica.

Condições especiais

As mensagens enviadas e recebidas devem ser persistidas conforme módulo de *log*.

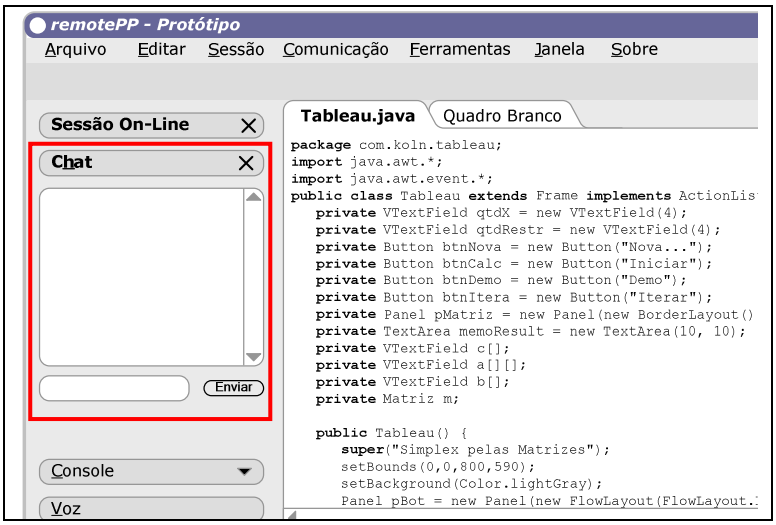

Figura 29 − Protótipo da ferramenta de *chat*.

3.8.2 Diagrama de classes

A Figura 30 apresenta o diagrama de classes do módulo de *chat*.

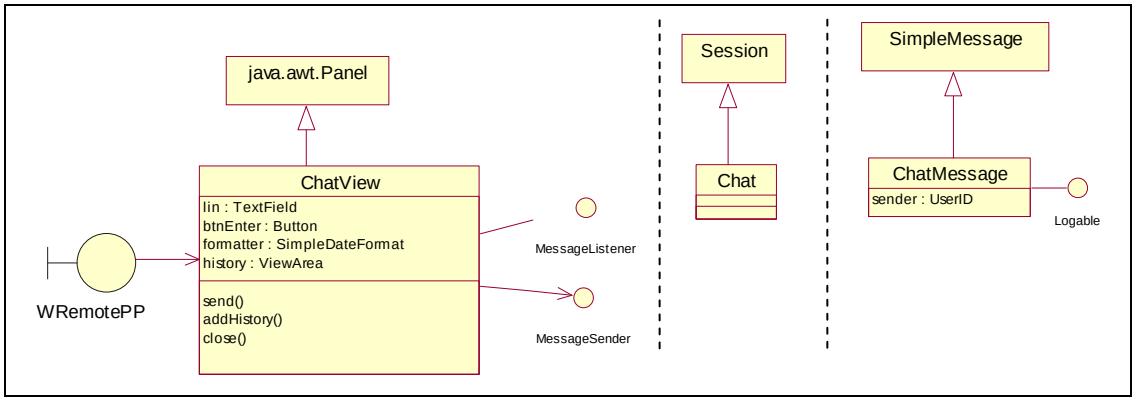

Figura 30 − Diagrama de classes do módulo de *chat*.

O módulo de *chat*, assim como as demais ferramentas de comunicação, é composto por três componentes básicos:

- *ChatView* – representa o contêiner aderido à tela principal da aplicação (*WRemotePP*), e é o ponto de interação com o usuário;
- *ChatMessage* – mensagem que será enviada ao computador remoto;
- *Chat* – classe de comunicação (derivada de *Session*).

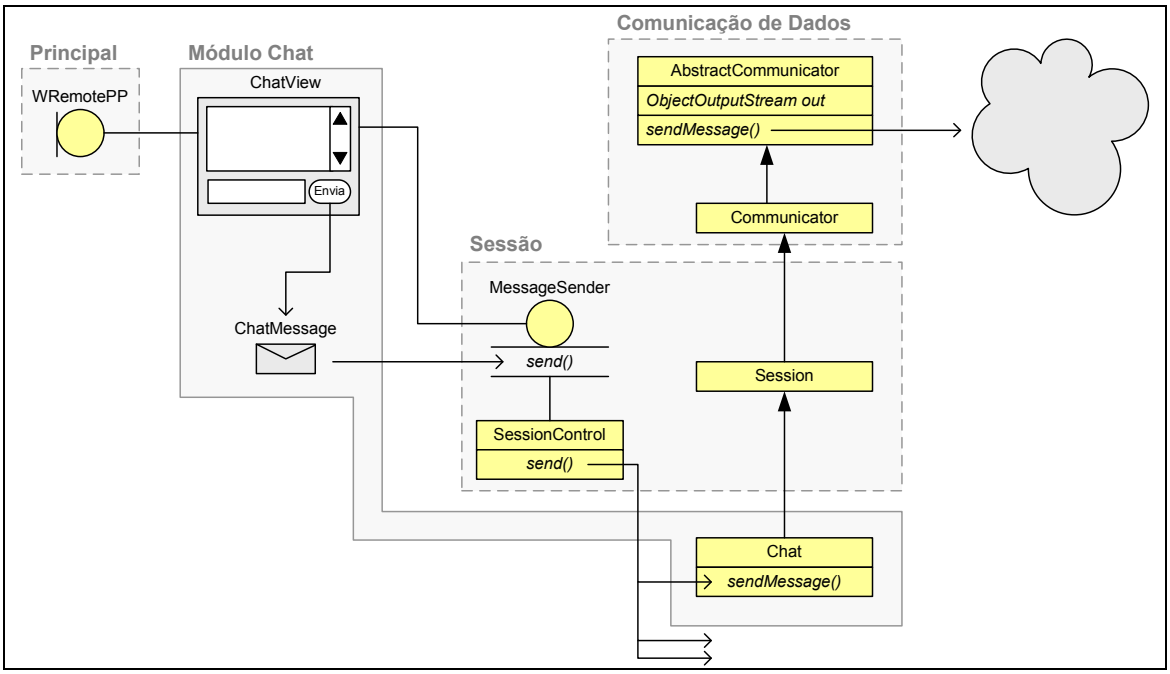

Figura 31 − Diagrama de atividades para envio de mensagens via *Chat*.

O fluxo apresentado pela Figura 31 indica o caminho percorrido pela mensagem desde o momento em que o usuário pressiona o botão “Envia” no contêiner *ChatView* até sua efetiva transmissão pela classe abstrata *AbstractCommunicator*, em que fica ilustrada a interdependência entre os módulos: principal, de *chat*, de sessão, e de comunicação de dados.

3.9 Módulo de edição de código-fonte

O módulo de edição de código-fonte deve ser utilizado durante uma sessão para edição de programas de forma colaborativa. No entanto, se não houver nenhuma conexão estabelecida, a edição ocorre apenas localmente e o caso de uso abaixo não se aplica.

Além disso, no caso de uso a seguir, não são descritas as funcionalidades básicas do editor de textos, mas assume-se que o editor deve oferecer recursos comuns como “Abrir arquivo”, “Salvar”, “Salvar como”, “Fechar”, “Copiar”, “Colar”, etc.

O editor deve apresentar em destaque as palavras-chave da linguagem que está sendo utilizada. Para tanto, deve ser utilizado um arquivo com o dicionário de palavras específicas ou reservadas para cada linguagem.

3.9.1 Caso de uso

Objetivos

Oferecer um ambiente para edição de código-fonte de forma colaborativa entre o ator local e o ator remoto.

Pré-condições

Uma sessão deve ser estabelecida entre o programador local e o remoto.

Fluxo básico – sem sessão

Ações do ator	Respostas do sistema
1. (local) Acessa o botão de ferramenta: "Editor"	2. (local) Estabelece uma nova conexão com o sistema remoto. Expande o painel abaixo do botão "Editor" para indicar o programador "*Driver*" e "*Navigator*". A opção "*Driver*" inicia marcada.
	3. (remoto) Expande o painel abaixo do botão "Editor" para indicar o programador "*Driver*" e "*Navigator*". A opção "*Navigator*" inicia marcada..
4. (local) Edita o código-fonte, pois está com o *status* de "*Driver*".	5. (local / remoto) As palavras reservadas da linguagem utilizada são apresentadas de forma diferenciada. Indicam a posição do cursor ou seleção do texto em ambos os sistemas.
6. (remoto) Clica na opção "*Driver*" para solicitar o controle do editor.	7. (remoto) Envia uma solicitação ao sistema local.
	8. (local) Recebe a solicitação e pergunta ao ator local se ele deseja alternar o controle do editor (Sim / Não).
9. (local) Responde "Sim"	10. (local) Responde positivamente à solicitação do sistema remoto. Muda a opção do painel para "*Navigator*".
	11. (remoto) Recebe a resposta do sistema local. Muda a opção do painel para "*Driver*".
12. (remoto) Passa a editar o código-fonte.	13. (local / remoto) Volta ao passo 5.

Fluxo alternativo

- Em (2), caso não seja possível estabelecer uma nova conexão, a mensagem "Falha de comunicação" é apresentada ao ator local.
- Em (9), caso o ator local responder "não", uma resposta negativa é enviada ao sistema remoto e apresentada ao respectivo ator.
- Em (9), caso o ator não responder à solicitação em 10 segundos, nada ocorre.

Pós-condições

O arquivo em formato texto é gerado e salvo em disco rígido de ambos os programadores (local e remoto).

A Figura 32 apresenta o protótipo da edição colaborativa de código-fonte.

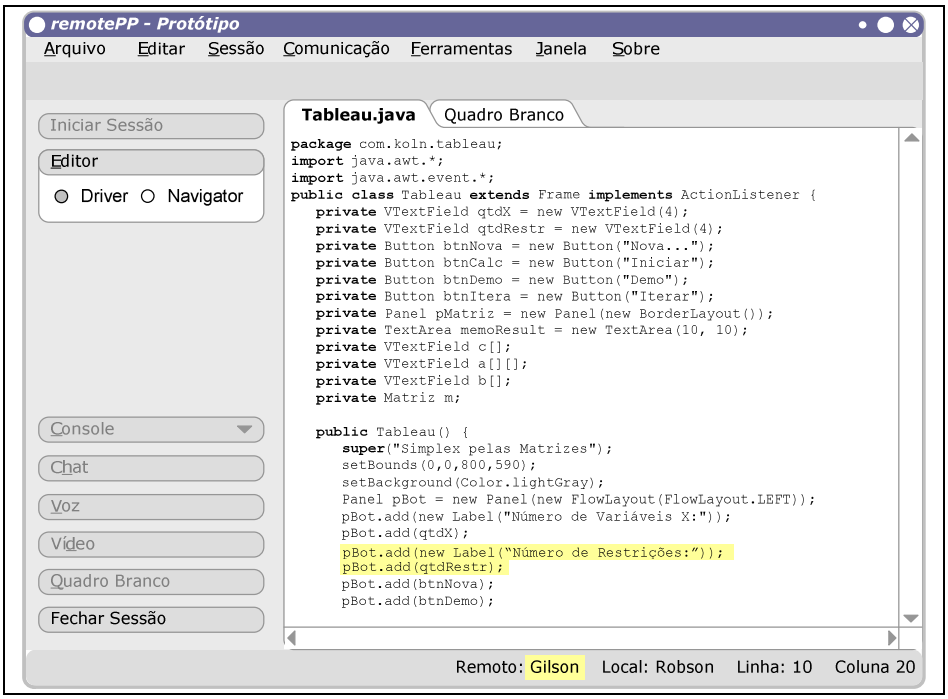

Figura 32 − Protótipo da edição de código-fonte.

3.9.2 Diagrama de classes

O módulo de edição de código-fonte (Figura 33) é responsável pelas funcionalidades que permitem aos usuários (local e remoto) editarem um mesmo código-fonte simultaneamente. De forma análoga ao módulo de *chat*, possui classes de visualização (*EditorView*), sessão (*Editor*) e mensagem. Nesse módulo, há duas classes para troca de mensagem e sincronização:

- *EditorMessage* – derivada diretamente de *AbstractMessage* (módulo de comunicação), trata-se de uma classe simples, que inclui apenas atributos para controle da posição do cursor e da seleção dentro do código-fonte entre os usuários;
- *EditorCompleteMessage* – especialização de *EditorMessage*, inclui todo o código-fonte que está sendo editado. Por ser uma classe mais "pesada", é enviada apenas: (a) quando o código é alterado pelo usuário que está editando o documento, e (b) depois de um determinado tempo decorrido (*timeslice*).

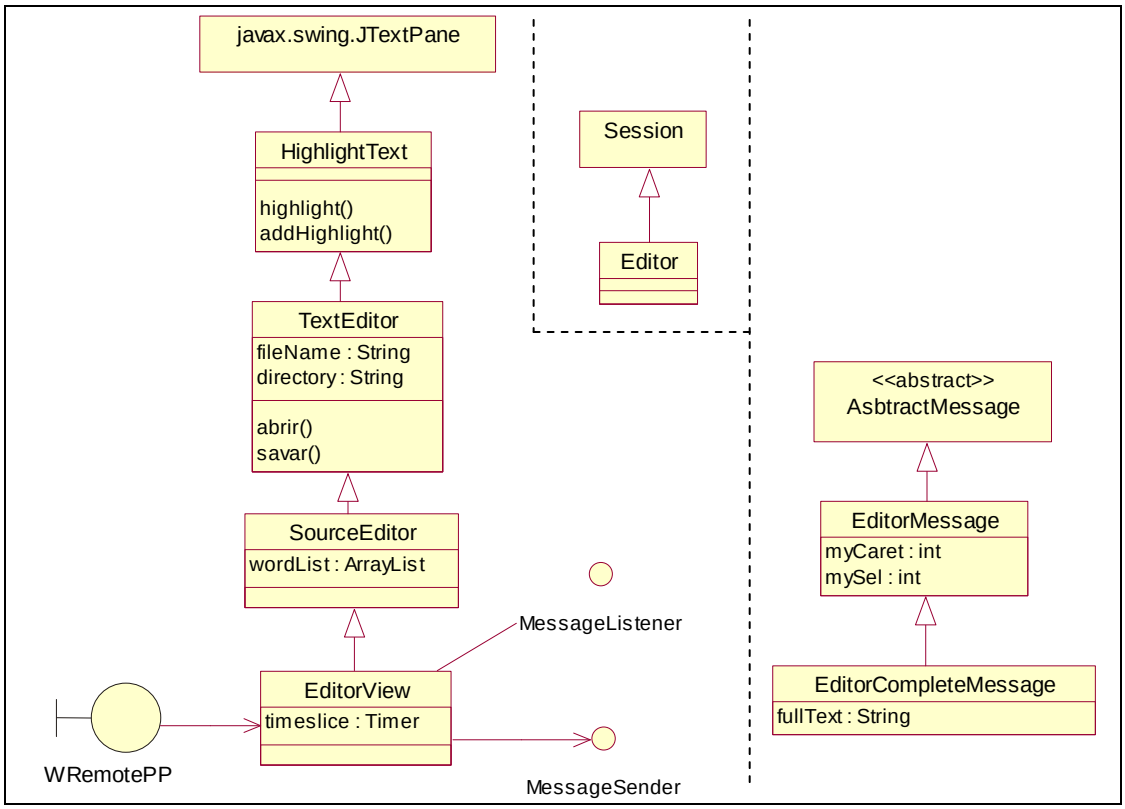

Figura 33 − Classes do módulo de edição de código-fonte.

As classes *HighlightText* e *TextEditor* fazem parte do pacote de interface gráfica e são utilizadas por outros módulos como o de c*hat* e o de ferramenta externa.

3.10 Módulo de troca de arquivos

O módulo de troca de arquivos oferece aos programadores, durante a sessão, a possibilidade de enviar e receber arquivos diversos, como especificações ou qualquer tipo de documentação eletrônica disponível para a realização da atividade em comum. A recepção do arquivo deve ocorrer de forma imediata pelo sistema remoto, de forma que ambos possam avaliar seu conteúdo tão logo quanto possível.

3.10.1 Caso de uso

Objetivo

Permitir ao programador local enviar um arquivo de formato binário ao programador remoto.

Pré-condições

Uma sessão deve estar ativa entre o programador local e o remoto.

Fluxo básico

Ações do ator	Respostas do sistema
1. (local) Acessa botão de ferramenta: "Enviar arquivo"	2. (local) Solicita uma nova conexão com o sistema remoto a partir da sessão atual.
	3. (remoto) Aceita a nova conexão.
	4. (local) Apresenta ao ator local a tela para seleção do arquivo que será enviado.
5. (local) Seleciona, em seu computador, o arquivo que deseja enviar.	6. (local) Envia uma solicitação ao sistema remoto com o nome e tamanho do arquivo que será enviado.
	7. (remoto) Apresenta ao ator local o nome e tamanho do arquivo a ser recebido. Solicita confirmação.
8. (remoto) Aceita o arquivo.	9. (remoto) Notifica o sistema local que a transferência pode ser iniciada.
	10. (local) Divide o arquivo que será transmitido em pacotes de tamanho pré-definido. Envia os pacotes ao sistema remoto.
	11. (remoto) Recebe os pacotes e monta o arquivo no computador remoto.
	12. (remoto) Informa ao ator remoto que o arquivo foi recebido com êxito e disponibiliza que seja aberto a partir da aplicação associada. (local) Avisa que o arquivo foi enviado com sucesso.
13. (remoto) Solicita que o arquivo seja aberto.	14. (remoto) Abre o arquivo a partir da aplicação associada.

Fluxo alternativo

Em (2), caso não seja possível estabelecer uma conexão com o sistema remoto, é exibida a mensagem: "Não foi possível iniciar conexão com o sistema remoto". Então, o caso de uso é encerrado.

Em (8), caso o ator remoto não aceite o recebimento do arquivo, o sistema remoto notifica o sistema local, que, por sua vez, avisa que o arquivo não foi aceito, e o caso de uso é encerrado.

Em (13), o ator local não necessita abrir o arquivo imediatamente. Neste caso, pode acessá-lo posteriormente via sistema operacional.

Em (14), caso o arquivo recebido não esteja associado a nenhum aplicativo, o sistema informa que não foi possível abrir o arquivo.

Pós-condições

O arquivo é recebido pelo ator remoto.

3.10.2 Diagrama de classes

A seleção do arquivo é representada na Figura 34 pela classe *TransferView*, que é uma especialização das janelas de diálogo padrão da aplicação (*WDialog*). Já a classe básica para tratamento das mensagens é identificada por *TransferMessage*, que possui o atributo *tamPacote* em comum com as demais mensagens.

Primeiramente, ao solicitar o início da transferência, a classe *TransferDialogMessage* é utilizada, e, posteriormente, uma instância de *TransferDataMessage* encarrega-se do envio dos pacotes do arquivo (atributo *byte[] data*).

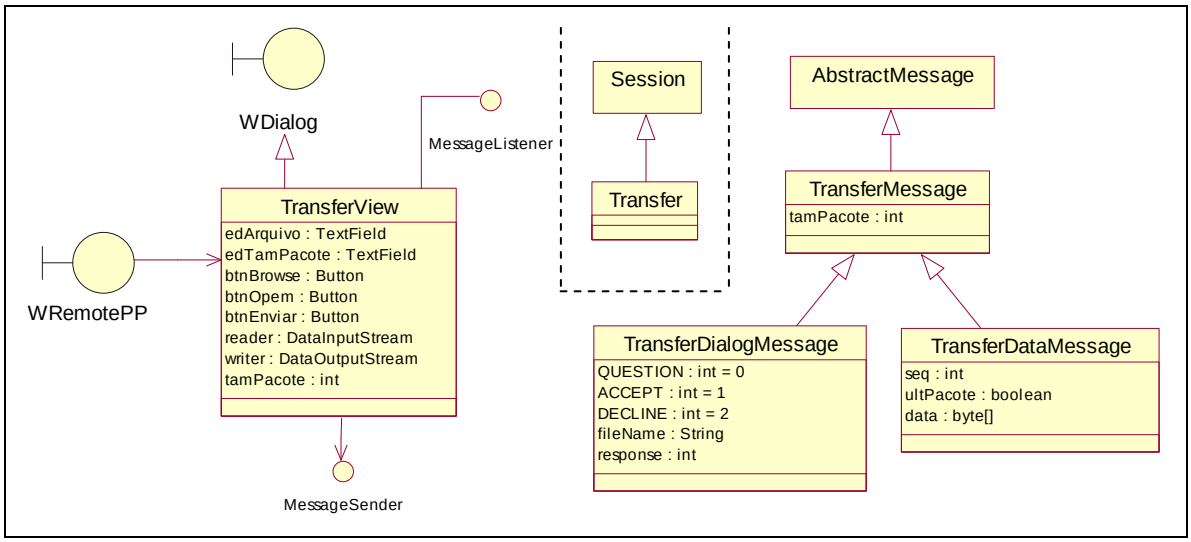

Figura 34 − Diagrama de classes do módulo de transferência de arquivos.

3.11 Módulo de ferramenta externa

O módulo de ferramenta externa permite expandir as funcionalidades do ambiente de programação ao possibilitar que seja evocado um compilador ou mesmo a execução do programa que está sendo editado. Assim, quando uma sessão estiver ativa, todas as mensagens de erro ou de saída oriundas da ferramenta externa são compartilhadas entre o ator local e o remoto.

3.11.1 Caso de uso

Objetivos

Oferecer ao programador a execução de ferramentas externas ao *remotePP*, tais como compilação ou execução por linha de comando, sendo que até quatro ferramentas externas configuráveis pelo próprio programador devem ser suportadas.

Pré-condições

O programador deve estar editando um código-fonte.

Fluxo básico

Ações do ator	Respostas do sistema
1. Acessa menu: Exibir > Console	2. Apresenta uma tela abaixo da área de edição de código com a execução da ferramenta.
3. Acessa menu: Ferramentas > F5, F6, F7 ou F8	4. Salva o código-fonte atual. Inicia a execução da ferramenta.
5. Durante a execução da ferramenta externa não é possível editar o código-fonte.	6. Conclui a execução da ferramenta externa, apresentando as possíveis mensagens de erro retornadas. Libera a o código-fonte para edição.
7. Volta a editar o código-fonte.	

Fluxo alternativo

- Em (4), caso a ferramenta externa não esteja disponível ou não tenha sido configurada previamente pelo programador, a mensagem "Ferramenta não disponível" deve ser apresentada.

Pós-condições

O programador volta a editar o código-fonte.

Condições especiais

Se uma sessão estiver estabelecida, o console é exibido em ambos os sistemas (local e remoto).

A Figura 35 apresenta o protótipo da chamada a uma ferramenta externa, ilustrando a chamada do compilador do código que está sendo editado.

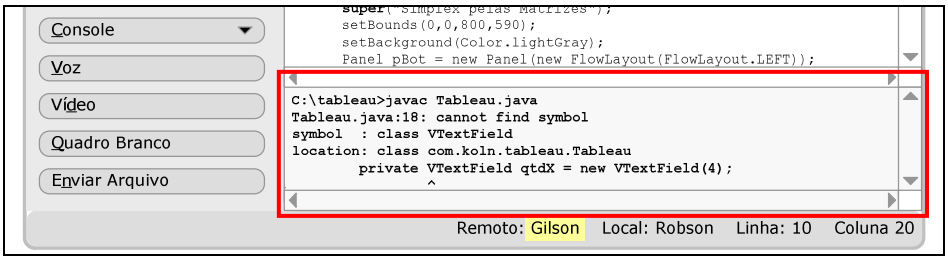

Figura 35 − Execução da ferramenta externa para compilação.

3.11.2 Diagrama de classes

Segundo o padrão das demais ferramentas de comunicação, o módulo para execução de ferramentas externas é dividido em três partes (Figura 36):

1. *ConsoleView* – derivando de *ViewArea*, é acoplado como atributo da tela principal (*WRemotePP*). Para que a execução da ferramenta ocorra de modo assíncrono em relação ao sistema, implementa a interface *Runnable*, podendo ser iniciado como uma *Thread*.
2. *Console* – apenas para especializar a classe *Session*.

3. *ConsoleMessage* – classe de mensagens de console persistente (implementa *Logable*).

Além dos componentes básicos, este módulo implementa uma outra *Thread* (*StreamCollector*), responsável pelo monitoramento do "console" da execução externa. Desta forma, toda e qualquer saída (mensagens ou erros) são direcionadas a *ConsoleView* através da interface *CollectorInterface*.

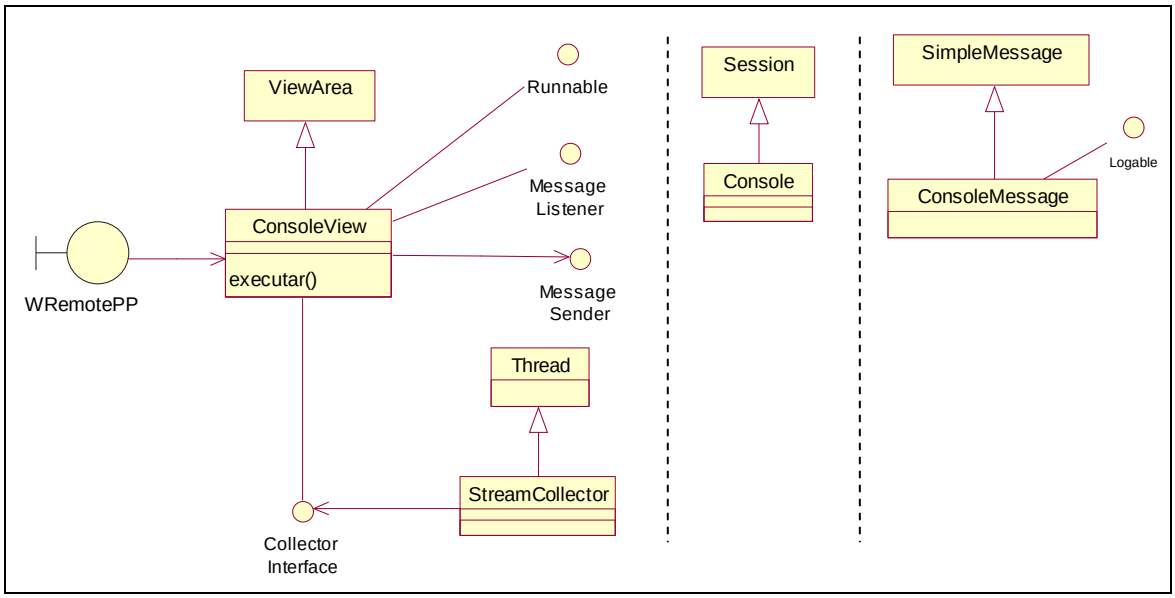

Figura 36 − Diagrama de classes do módulo de ferramenta externa.

3.12 Módulo de *bookmark*

O módulo de *bookmark* é responsável por sincronizar os endereços favoritos da Internet entre os programadores (local e remoto). Esta é uma ferramenta de interação social que permite a um ator conhecer um pouco mais sobre seu interlocutor, desde as bases de conhecimento na Internet até de interesse em comum, o que facilita a empatia e aproxima um pouco mais os programadores.

3.12.1 Caso de uso

Objetivo

Enviar e receber os endereços favoritos da Internet, cadastrados sob o menu "Meu *bookmark*".

Pré-condições

É necessário que ambos estejam com uma sessão ativa.

Fluxo básico

Ações do ator	Respostas do sistema
1. (local) Acessa botão de ferramenta: *"Bookmark"*	2. (local) Estabelece uma conexão com o sistema remoto. Envia seus próprios endereços favoritos.
	3. (remoto) Recebe os endereços favoritos do sistema local. Envia os endereços favoritos.
	4. (local) Recebe os endereços favoritos do sistema remoto. Apresenta a lista de *links* da Internet oriundas do sistema remoto.
5. (local) Clica em um dos endereços recebidos.	6. (local) Abre o *browser* padrão do sistema e apresenta o *site* correspondente.

Fluxo alternativo

- Em (2), se não for possível estabelecer uma conexão, a mensagem "Falha de comunicação" é apresentada.

Pós-condições

A lista de endereços favoritos (*bookmark*) da Internet é sincronizada entre os programadores.

3.12.2 Diagrama de classes

A visualização dos endereços favoritos é feita através de um menu do tipo *popup* (classe *BookmarkView*), já a conexão é estabelecida utilizando a classe especializada *Bookmark* (subclasse de *Session*), e as mensagens são trafegadas por instâncias da classe *BookmarkMessage*.

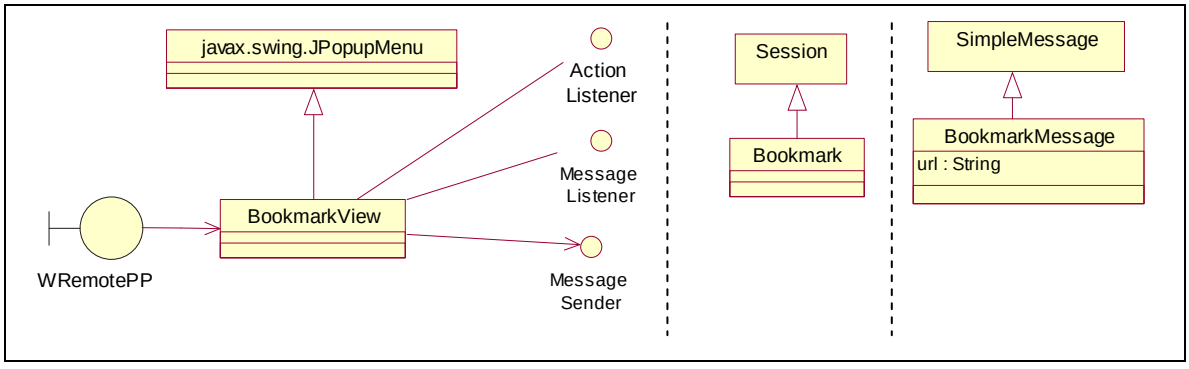

Figura 37 − Diagrama de classes do módulo de *bookmark*.

3.13 Módulo de quadro branco

O funcionamento do quadro branco é semelhante ao editor de código-fonte, com a diferença de que, neste caso, ambos os programadores (local e remoto) podem alterar o artefato simultaneamente.

3.13.1 Caso de uso

Objetivo

O objetivo da ferramenta de quadro branco é oferecer um ambiente para realização de desenhos de projeto de forma colaborativa.

Pré-condições

Deve haver uma sessão estabelecida entre o sistema local e o remoto.

Fluxo básico

Ações do ator	Respostas do sistema
1. (local) Acessa botão de ferramenta: "Quadro branco"	2. (local) Estabelece uma conexão com o sistema remoto. Apresenta a janela para edição do desenho.
3. (local) Seleciona uma ferramenta de edição (retângulo, elipse, texto, linha ou desenho a mão livre).	4. (local) Realça o objeto gráfico selecionado na barra de ferramentas.
5. (local) Traça os objetos gráficos com o auxílio do *mouse*.	6. (local) Atualiza o quadro branco com a figura desenhada e transmite ao sistema remoto.
	7. (remoto) Recebe as atualizações do sistema local e reflete os objetos desenhados pelo ator local no quadro branco do sistema remoto.
8. (remoto) Traça objetos com o auxílio do *mouse*.	9. (remoto) Atualiza o quadro branco do sistema remoto com os desenhos e transmite ao sistema local.
	10. (local) Recebe as atualizações do sistema remoto e reflete os objetos desenhados pelo ator remoto no quadro branco do sistema local.

Fluxo alternativo

Em (2), caso não seja possível estabelecer uma conexão com o sistema remoto, exibe a mensagem: "Não foi possível iniciar conexão com o sistema remoto".

Pós-condições

O desenho é atualizado, simultaneamente, em ambos os quadros brancos (local e remoto).

A Figura 38 ilustra o protótipo do quadro branco em funcionamento.

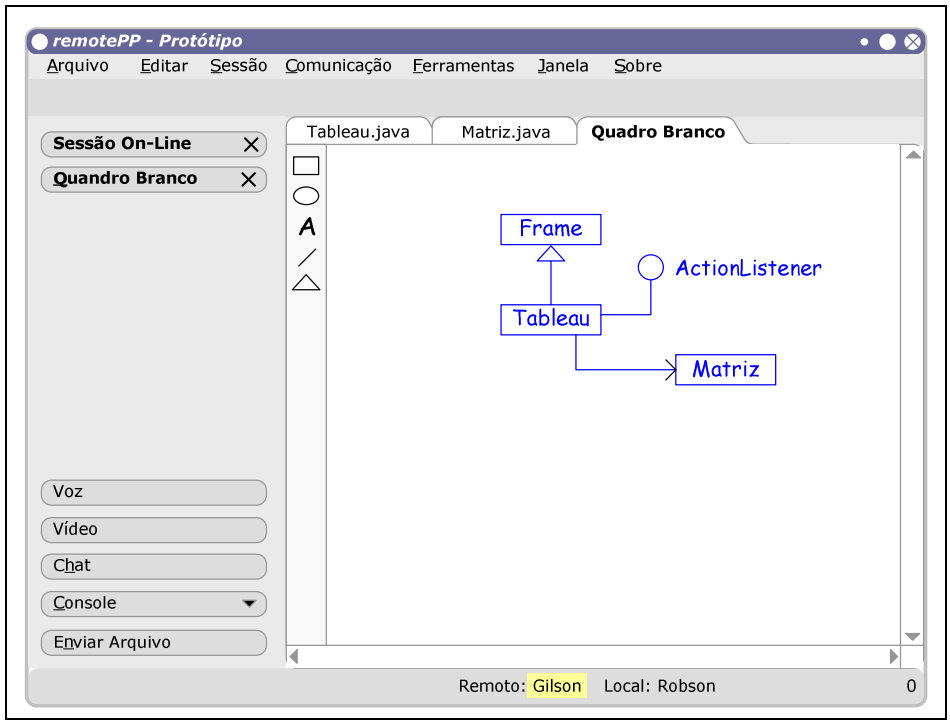

Figura 38 − Protótipo do quadro branco.

3.13.2 Diagrama de classes

Conforme pode ser verificado na Figura 39, a classe *SimpleWhiteboard* implementa um contêiner para a edição *standalone* de elementos gráficos. Já a sua especialização (*WhiteBoardView*) agrega funcionalidades para edição colaborativa dos desenhos através das interfaces *MessageListener* e *MessageSender*, as quais são responsáveis, respectivamente, pelo recebimento e envio de notificações.

As mensagens trocadas entre o sistema local e remoto são derivações da classe *WhiteMessage*:

- *LineWhiteMessage* – desenha uma linha traçada do ponto (x0, y0) até (x1, y1);
- *OvalWhiteMessage* – descreve os limites de uma elipse;
- *RectangleWhiteMessage* – retângulo limitado pelos pontos (x0, y0) e (x1, y1);
- *TextWhiteMessage* – elementos de texto posicionados a partir de (x0, y0);
- *PencilWhiteMessage* – desenho poligonal a mão livre (representado pelo *array* de pontos *poly*).

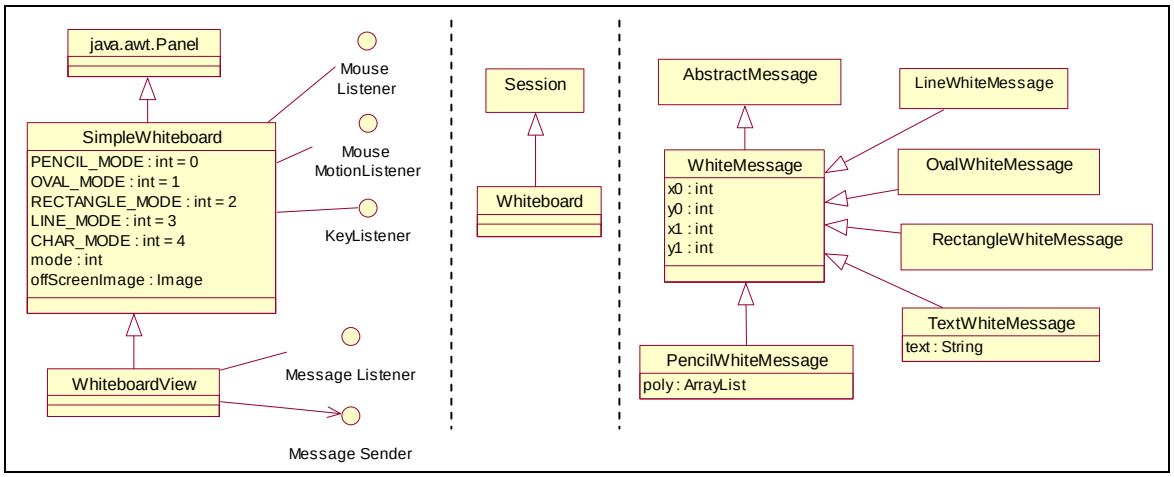

Figura 39 − Diagrama de classes do módulo de quadro branco.

3.14 Módulo de vídeo

O módulo de vídeo é responsável pela possibilidade de realizar vídeo-conferência entre os programadores, sendo que, para tanto, é necessário acoplar uma *webcam* ao sistema que deseja enviar o sinal de vídeo. O objetivo dessa ferramenta é oferecer contato visual e permitir uma comunicação mais eficaz entre os programadores na discussão de problemas e alternativas de solução. Além disso, ela torna a sessão mais próxima de uma programação por pares presencial.

3.14.1 Caso de uso

Objetivo

Permitir comunicação visual entre os programadores através de periférico computacional de captura de vídeo como *webcam*.

Pré-condições

É necessário que uma sessão esteja estabelecida entre os programadores.

Fluxo básico

Ações do ator	Respostas do sistema
1. (local) Acessa botão de ferramenta: "Vídeo"	2. (local) Requisita uma nova conexão com o sistema remoto a partir da sessão atual.
	3. (remoto) Aceita e estabelece a conexão.
	4. (local / remoto) Expandem a área abaixo do botão de vídeo (conforme Figura 40).
	5. (local) Apresenta, no quadro menor, a própria imagem e a transmite ao sistema remoto. Apresenta, no quadro maior, a imagem recebida a partir do sistema remoto.
	6. (remoto) Apresenta, no quadro menor, a própria imagem e a transmite ao sistema local. Apresenta, no quadro maior, a imagem recebida a partir do sistema local.
7. (local) Clica no botão "Vídeo" novamente.	8. Encerra a captura e transmissão de vídeo. Finaliza a conexão com o sistema remoto.

Fluxo Alternativo

Em (2), caso a conexão não seja estabelecida, apresenta a mensagem: "Não foi possível iniciar conexão".

Em (5) ou (6), caso o programador não possua periférico para captura de vídeo, será apresentada somente a imagem maior, correspondente ao interlocutor. Caso nenhum dos dois possua tal periférico, nenhuma imagem é apresentada.

Pós-condições

Não se aplica.

O modelo de tela do módulo de vídeo é apresentado na Figura 40, onde, baixo do botão "Vídeo" é possível identificar dois painéis: um amplo, com a imagem do programador remoto; e outro menor, com a própria imagem do programador local. Além disso, esse recurso permite que sejam realizados ajustes no periférico de transmissão.

Caso o programador não possua periférico para transmissão de vídeo (*webcam*) apenas a imagem maior (remota) será apresentada.

Figura 40 − Protótipo da transmissão de vídeo.

3.14.2 Diagrama de classes

Para implementação desse módulo, será necessário utilizar a biblioteca de captura de vídeo (*dsj*), disponível em <http://www.humatic.de/htools/dsj.htm>.

As classes *SingleVideo* e *SingleVideoView* fazem uso da classe de captura *DSCapture* e possuem os seguintes propósitos:

- *SingleVideo* – apresentar a imagem obtida a partir do periférico local e disponibilizá-lo para controle;
- *SingleVideoView* – exibir a imagem obtida através da conexão com o computador remoto.

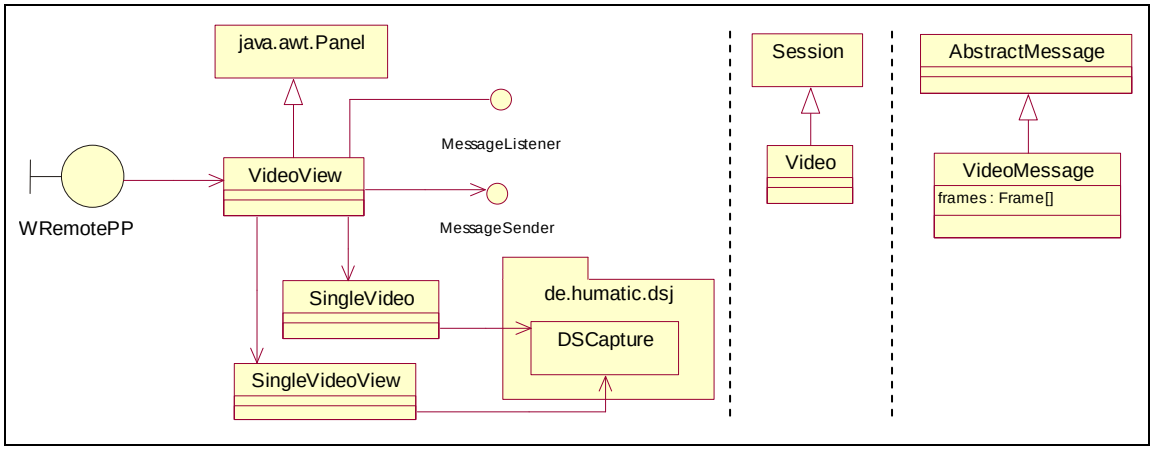

Figura 41 − Diagrama de classes do módulo de vídeo.

3.15 Módulo de voz

A comunicação por voz possibilita aos programadores discutir verbalmente as questões relativas à implementação que está sendo realizada, e possui vantagens em relação à comunicação por *chat* na medida que os programadores não precisam deslocar o cursor do código-fonte. Já em relação à comunicação por vídeo, a comunicação por voz permite manter a concentração no artefato que está sendo construído. Além disso, exige menor recurso computacional e menor largura de banda do que a vídeo-conferência. Por esse motivo, a comunicação por áudio não necessita ser realizada simultaneamente com o vídeo. No entanto, ambos podem ser utilizados de forma complementar, na medida em que um problema exigir um esforço cognitivo mais elevado entre os participantes.

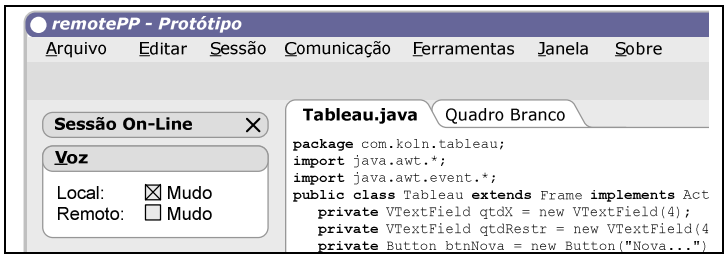

Figura 42 − Protótipo do módulo de voz.

A interface sugerida na Figura 42 apresenta duas opções para inibir temporariamente o envio ou recepção de áudio através das caixas de checagem “Mudo”: local ou remoto.

3.15.1 Diagrama de classes

Seguindo a arquitetura vista nos módulos anteriores, o módulo de voz implementa as classes *VoiceView*, *Voice* e *VoiceMessage* (Figura 43). Para captura de áudio foi utilizada a classe *Downstream* (da biblioteca *dsj*, semelhante à solução para captura de vídeo). Já a classe *VoicePlayer* é utilizada para reproduzir o *streaming* de áudio recebido do sistema remoto.

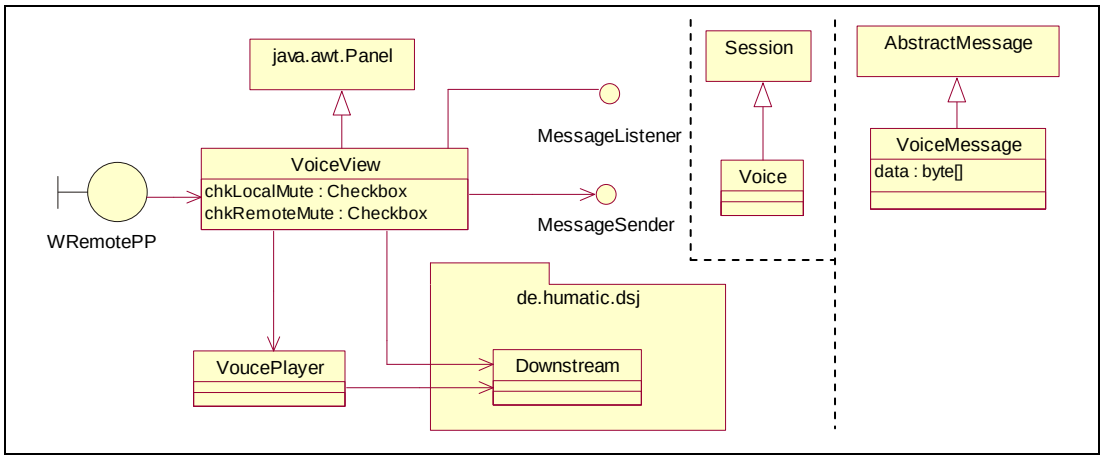

Figura 43 − Diagrama de classes do módulo de voz.

A seleção dos periféricos utilizados para captura e reprodução (*driver* de áudio) deve ser realizada previamente à utilização do módulo de voz.

Desta maneira, o módulo de voz encerra o capítulo de análise e projeto do *remotePP*, em que foram apresentados os requisitos de funcionamento, justificativas, arquitetura e projeto de implementação do ambiente de programação em pares distribuídos. Portanto, conforme pôde ser observado, futuros módulos de comunicação ou ferramentas de programação podem ser facilmente desenvolvidas e acopladas ao sistema atual, respeitando regras simples de derivação e utilização das classes de exibição, comunicação e mensagens.

4 Considerações finais

O primeiro capítulo deste trabalho identificou o objetivo de viabilizar um ambiente computacional que permitisse a programação por pares em ambientes geograficamente distribuídos, sendo que os procedimentos metodológicos nortearam e possibilitaram a execução deste trabalho de forma coerente.

No segundo capítulo, a revisão de literatura foi necessária para estudar as características de engenharia de *software* aplicáveis a esse tipo de metodologia de desenvolvimento de sistemas, em que foram avaliados os benefícios e limitações que a programação realizada por times distribuídos apresenta.

Sendo assim, ao estudar os recursos técnico-científicos, sociais e comerciais atualmente disponíveis, foi possível identificar a contribuição acadêmica e comercial para a aplicação prática deste trabalho. Já a análise de cinco trabalhos acadêmicos voltados à programação distribuída permitiu identificar um conjunto de ferramentas que, ao serem complementadas entre si, ajudou a montar o cenário a ser atendido por este trabalho.

Finalmente, no terceiro capítulo, foi possível estruturar o *remotePP* como uma solução computacional para atender ao objetivo primário deste trabalho: um ambiente para programação em pares geograficamente distribuídos. Para tanto, identificando os requisitos funcionais e não funcionais, a arquitetura foi projetada e desenvolvida de modo a endereçar cada uma das necessidades avaliadas, e o projeto utilizou-se não apenas da UML de representação da solução de desenvolvimento, mas também de diagramas livres que pudessem expressar os relacionamentos entre os diversos módulos do sistema.

Assim, a implementação do trabalho permitiu disponibilizar um ambiente computacional suficiente para colocar em prática ferramentas de programação e comunicação unificadas e capazes de tirar o melhor proveito das técnicas ágeis de desenvolvimento de *software*, em especial a programação por pares, utilizada para implementar a *Extreme Programming (XP)*. E, além disso, conclui-se que o controle de fluxo e a formação de uma semântica (XML) para persistência das mensagens de comunicação podem oferecer recursos para extração de conhecimento a respeito de estilos de programação ou identificação de problemas recorrentes no desenvolvimento de sistemas.

4.1 Trabalhos futuros

Os temas abordados neste trabalho estão longe de se esgotarem e podem servir de inspiração para realização de pesquisas e trabalhos futuros. Dessa maneira, espera-se que o conhecimento adquirido até aqui possa contribuir para o progresso da Ciência e de novas técnicas para solução de problemas computacionais e sociais. Neste sentido, sugerem-se os seguintes trabalhos:

- Construção de uma base de conhecimento a partir dos dados de *log* persistidos com o uso do *remotePP* a fim de identificar estilos de programação e auxiliar os programadores a utilizar as soluções mais bem sucedidas na resolução dos seus problemas;

- Evolução da ferramenta de quadro branco de forma a integrá-la como uma ferramenta *case* para projeto e construção de códigos-fonte de forma otimizada;
- Integração com sistemas de gerenciamento e controle de versões dos artefatos construídos colaborativamente;
- Inclusão da figura do mediador como um consultor com alto conhecimento técnico e de negócio durante as sessões, cujo intuito da mediação seria oferecer uma forma de resolver possíveis situações de impasse entre os programadores quando não houver consenso entre eles;
- Realizar um estudo de caso qualitativo com o objetivo de medir a qualidade e a velocidade da programação em pares distribuída em relação à técnica presencial.

Bibliografia

Abrahamsson, P. Extreme programming: first results from a controlled case study. Euromicro Conference, 2003. Proceedings. 29th , vol., no.pp. 259- 266, 1-6 Sept. 2003

Abrahamsson, P., Salo, O., Ronkainen, J., Warsta, J. Agile Software Development Methods: Review and Analysis. 2002. VTT Publications 478.

Baheti, P., Gehringer, E., Stotts D. Exploring the efficacy of distributed pair programming. XP Agile Universe 2002, August 2002

Beck, K. Programação extrema (xp) explicada : acolha as mudanças. Porto Alegre: Bookman, 2004. 182 p.

BOOCH, Grady; RUMBAUGH, James; JACONSON, Ivar. Uml : guia do usuário. Rio de Janeiro: Campus, 2000. 472 p.

Canfora, G.; Cimitile, A.; Visaggio, C.A., Lessons learned about distributed pair programming: what are the knowledge needs to address?. Enabling Technologies: Infrastructure for Collaborative Enterprises, 2003. WET ICE 2003. Proceedings. Twelfth IEEE International Workshops on , vol., no.pp. 314- 319, 9-11 June 2003

Carmel, E. Thirteen assertions for globally dispersed software development research. System Sciences, 1997, Proceedings of the Thirtieth Hawaii International Conference on , vol.3, no.pp.445-452 vol.3, 7-10 Jan 1997

Cockburn, A., Williams, L The costs and benefits of pair programming, Extreme Programming Examined, Succi, G., Marchesi, M. eds., pp. 223-248, Boston, MA: Addison Wesley, 2001

Cooper, J. The Design Pattern: Java Companion, Addison-Wesley, 1998. 218 p.

Crowley, T., Milazzo, P., Baker, E., Forsdick, H., Tomlinson, R. MMConf: an infrastructure for building shared multimedia applications. In Proceedings of the 1990 ACM conference on Computer-supported cooperative work, 329-342, Los Angeles, California, United States, ACM Press. 1990.

Dewan, P.; Riedl, J., Toward computer-supported concurrent software engineering. Computer , vol.26, no.1pp.17-27, Jan 1993

Ellis, C. A., Gibbs, S. J., and Rein, G. 1991. Groupware: some issues and experiences. Commun. ACM 34, 1 (Jan. 1991), 39-58.

Fowler, M., Highsmith, J. The Agile Manifesto. Software Development. 28-31. 2001.

Fuks, H., Raposo, A, Gerosa, M. Do Modelo de Colaboração 3C à Engenharia de Groupware. WEBMIDIA 2003 - Simpósio Brasileiro de Sistemas Multimídia e Web, Trilha especial de Trabalho Cooperativo Assistido por Computador, 03 a 06 de Novembro de 2003, Salvador-BA, pp. 445-452

Hanks, B. 2005. Student performance in CS1 with distributed pair programming. In Proceedings of the 10th Annual SIGCSE Conference on innovation and Technology

in Computer Science Education (Caparica, Portugal, June 27 - 29, 2005). ITiCSE '05. ACM Press, New York, NY, 316-320.

Herbsleb, J.D.; Moitra, D., "Global software development," Software, IEEE , vol.18, no.2pp.16-20, Mar/Apr 2001

Ho, C., Raha, S., Gehringer, E., and Williams, L. 2004. Sangam: a distributed pair programming plug-in for Eclipse. In Proceedings of the 2004 OOPSLA Workshop on Eclipse Technology Exchange (Vancouver, British Columbia, Canada, October 24 - 24, 2004). eclipse '04. ACM Press, New York, NY, 73-77.

Humatic Hypermedia – Interactive Screens. Disponível em <http://www.humatic.de/htools/dsj.htm>. Acesso em 02 de nov. 2006.

Kircher, M., Jain, P., Corsaro, A., Levine, D. Distributed Extreme Programming. In Marchesi and Succi [35], pages 66-71

L. Williams, R. Kessler, W. Cunningham, and R. Jeffries. Strengthening the case for pair-programming, IEEE Software 17:4, July/Aug 2000, pp. 19-25.

Maurer, F., Mertel, S.: Process support for distributed Extreme Programming teams. In: Proceedings of the International Workshop on Global Software Development (ICSE'02 GSD), Orlando, FL (2002).

McCauley, R. 2001. Agile development methods poised to upset status quo. SIGCSE Bull. 33, 4 (Dec. 2001), 14-15.

Natsu, H., Favela, J., Moran, A., Decouchant, D., Martinez-Enriquez, A., Distributed pair programming on the Web. Computer Science, 2003. ENC 2003. Proceedings of the Fourth Mexican International Conference on , vol., no.pp. 81- 88, 8-12 Sept. 2003.

Peters, J., Pedrycz W. Engenharia de software : teoria e prática. Rio de Janeiro : Campus, 2001. 602 p.

Pressman, R. Software engineering : a practitioner's approach. 5. ed. Boston: McGraw-Hill, 2001. 860 p.

Prikladnicki, R., Audy, J. MuNDDoS: Um modelo de referência para desenvolvimento distribuído de software. In: 18 SBES - Simpósio Brasileiro De Engenharia De Software, 2004, Brasília. 2004.

Roesch, Sylvia Maria Azevedo. Projetos de estágio e de pesquisa em administração: guia para estágios, trabalhos de conclusão, dissertações e estudos de caso. 2. ed. São Paulo: Atlas, 1999. 301 p. ISBN 85-224-2338-5

Schummer, T., Schummer J. Support for distributed teams in Extreme Programming. Extreme Programming examined, Succi, G., Marchesi, M. eds., p. 355–377, Boston, MA: Addison Wesley, 2001

Sommerville, I. Software engineering. 6. ed. Harlow: Addison-Wesley, 2001. 693 p.

Stotts, David, Laurie Williams, Nachiappan Nagappan, Prashant Beheti, Dennis Jen, and Anne Jackson, Virtual Teaming: Experiments and Experiences with Distributed Pair Programming, Proceedings of the Third XP Agile Universe Conference (Springer LNCS 2753), pages 129 - 141, August 2003.

Tonsig, Sérgio Luiz. Engenharia de software : análise e projeto de sistemas. São Paulo: Futura, 2003. 351 p.

Williams, L. A., Kessler, R. R. 2000. All I really need to know about pair programming I learned in kindergarten. Commun. ACM 43, 5 (May. 2000), 108-114.

Williams, L., But, isn't that cheating? [collaborative programming]. Frontiers in Education Conference, 1999. FIE '99. 29th Annual , vol.2, no.pp.12B9/26-12B9/27 vol.2, 1999