UNIVERSIDADE FEDERAL DE PELOTAS
INSTITUTO DE FÍSICA E MATEMÁTICA
CURSO DE BACHARELADO EM INFORMÁTICA

Um Estudo das Técnicas Usadas na Construção de Aplicações de Auxílio ao Trabalho Colaborativo

por

FABRÍCIA CARNEIRO ROOS

Monografia do Projeto de Conclusão submetida à avaliação,
como requisito parcial para a obtenção do grau de
Bacharel em Informática – ênfase Sistemas de Computação.

Anal. Sist. Carlos A. M. dos Santos
Orientador

Prof. Maurício Nunes Porto
Co-Orientador

Pelotas, novembro de 2000.

"É melhor tentar e falhar, que preocupar-se e ver a vida passar. É melhor tentar, ainda que em vão; que sentar-se fazendo nada até o final. Prefiro na chuva caminhar que dias tristes em casa me esconder. Prefiro ser feliz embora louco, que em conformidade viver"

Martin Luther King Jr

.

Agradecimentos

Gostaria muito de agradecer primeiramente à minha família pela compreensão e carinho durante os quatro anos que estivemos distantes praticamente o tempo todo. À minha mãe, Valquíria, que sempre teve dedicação incansável para que eu pudesse hoje estar realizando mais uma conquista na minha vida, com muito amor agradeço. Aos meus irmãos Fabiana e Fábio que sempre me apoiaram nas horas mais difíceis com muito carinho e compreensão, agradeço.

Com carinho muito especial agradeço ao meu namorado, Márcio Morales, que sempre contribuiu muito com minha formação pessoal, estando ao meu lado sempre que precisei. Pessoa confiável que me fez mais forte em momentos difíceis, e que sempre admirei pela força e coragem com que assume suas decisões e seus ideais, agradeço com muito amor.

Ao meu orientador Carlos A. M. dos Santos um agradecimento muito especial pela paciência e compreensão que sempre dedicou ao meu trabalho, sempre procurando transmitir da melhor maneira possível os conhecimentos que possui. Além do agradecimento, gostaria de parabenizá-lo pela dedicação com que realiza suas tarefas e pelo profissional competente que é.

Ao meu co-orientador Maurício Nunes Porto pelo grande amigo e professor que sempre foi durante todo o decorrer do curso. Obrigada por todos os ensinamentos.

Aos colegas pela amizade e colaboração que me ofereceram durante os quatro anos sem pedir nada em troca.

Aos amigos que sempre estiveram felizes a cada momento de vitória que conquistei, que sempre me acolheram a cada deslize que cometi e que sempre me compreenderam e me deram força.

Sumário

Lista de Abreviaturas

ACM	*Association for Computing Machinery*
CSCW	*Computer Support for Cooperative Work*
fig.	figura
GUI	*Graphical User Interface*
MVC	*Model-View-Controller*
n.	Número
OSF	*Open Software Foundation*
p.	pagina
PARC	*Palo Alto Research Center*
P	*Personal Computer*
PM	*Presentation Manager*
UIDE	*User Interface Development Environments*
UIMS	*User Interface Management Systems*
v.	volume

Lista de Figuras

Resumo

O trabalho cooperativo tornou-se uma importante área de estudo devido à grande necessidade da existência de aplicações que facilitem a comunicação entre os usuários. Mas existem vários problemas que dificultam o desenvolvimento dessas aplicações, tais como *interface* com o usuário, segurança e outros. Esta monografia apresenta um estudo das técnicas usadas na construção de aplicações de auxílio ao trabalho colaborativo, utilizando como ferramenta de estudo jogos de computador.

A área científica da computação que foi abordada para este estudo é conhecida como Trabalho Colaborativo Auxiliado por Computador, normalmente referenciada na literatura pelo termo em inglês "*Computer Support for Cooperative Work*" (CSCW).

O trabalho apresenta alguns conceitos e abordagens de assuntos necessários para o melhor entendimento dos problemas encontrados no desenvolvimento de aplicações colaborativas, mais especificamente nos jogos de computador.

Para a implementação da *interface* da aplicação exemplo, tomou-se como paradigma a arquitetura *Model-View-Controller* (MVC). As razões que levaram a opção do uso do MVC neste trabalho, foram algumas vantagens que este possui como ser considerado um padrão de projeto, oferecer confiabilidade, permitir modularidade, e outras que serão apresentadas no trabalho.

1. Introdução

1.1 Motivação do Trabalho

Computer Support for Cooperative Work (CSCW) é uma área da Computação que preocupa-se com o estudo das técnicas que facilitem a colaboração entre usuários, possivelmente localizados em lugares diferentes, usando o computador como intermediário. Atualmente existem várias soluções de alta tecnologia para auxílio ao trabalho cooperativo. Por exemplo, tecnologias de comunicação são usadas para atingir o esperado nível de cooperação usando teleconferência [MAN 98].

Uma das principais motivações desse trabalho é a necessidade de ampliar os conhecimentos na área de CSCW, pois é uma área em expansão, graças à disseminação dos computadores e novos dispositivos de *interface*. Outro motivou foi que o tema (CSCW) nunca havia sido abordado em trabalhos de conclusão do curso, o que despertou assim um maior interesse em estudar o assunto.

Atualmente as pesquisas são conduzidas por equipes de trabalho em que diversas pessoas estudam simultaneamente o mesmo assunto, mudam e discutem resultados parciais, podendo estar separados geograficamente e agindo sobre o mesmo modelo [MAN 98]. Como ocorre nestas e em outras aplicações de trabalho colaborativo, muitos são os problemas encontrados para se desenvolver aplicações realmente eficientes onde se possa explorar vários estados da comunicação.

Acreditamos que o estudo das técnicas usadas na construção de jogos pode auxiliar a identificar alguns problemas dessas aplicações e eventualmente as soluções para estes. Considerando que há muitos aspectos comuns nos diferentes usos de CSCW, acreditamos também que as técnicas utilizadas em jogos de computador podem ser usadas na construção de aplicativos muito úteis em outras áreas, como por exemplo a medicina, teleconferência e suporte a decisão.

1.2 Objetivos

O objetivo deste trabalho é adquirir maiores conhecimentos na área de Computação Colaborativa, visando conhecer técnicas que auxiliam na construção de aplicações de auxílio ao trabalho cooperativo. Como forma de encaminhar um estudo dessas técnicas, foi adotada a implementação de um jogo como aplicação exemplo, usando algumas das técnicas estudadas, possibilitando assim obter uma análise sobre esse assunto, permitindo a identificação dos problemas encontrados no desenvolvimento das aplicações de CSCW. Esta aplicação objetiva desenvolver um jogo da velha, onde serão usadas técnicas utilizadas para implementar uma aplicação colaborativa, bem como desenvolvimento de *interface* e paradigmas de desenvolvimento.

Este trabalho visa mostrar as principais dificuldades encontradas para desenvolver aplicações que permitam interação entre os usuários através do computador. Nesta aplicação estima-se que os problemas encontrados sejam resolvidos.

1.3 Organização do Texto

O texto começa no capítulo 2, no qual se faz uma descrição dos Conceitos de CSCW e *Groupware* e mostra uma classificação para as aplicações colaborativas considerando suas características comuns. Este capítulo descreve o tema principal do trabalho, CSCW.

O capítulo 3 aborda conceitos básicos de programação concorrente, como sincronização, ações atômicas e transação. Estes conceitos mostram como uma aplicação colaborativa é coordenada para que haja uma eficiente interação entre os usuários. Quando se faz um estudo de aplicações colaborativas, deve-se levar em consideração qual a arquitetura será utilizada para suportar este tipo de aplicação. Como solução para a aplicação exemplo foi descrito neste capítulo, sistemas cliente/servidor relacionada com o uso em CSCW.

O capítulo 4 faz algumas considerações sobre *interface* com o usuário. Aspectos gerais de *interface* são analisados para caracterizar uma boa *interface*. *Interface*s multiusuário são descritas com o objetivo de definir um tipo especial de *interface* quando se considera aplicações colaborativas. A arquitetura MVC é discutida e apresentada como forma de melhoria no desenvolvimento de aplicações que oferecem

interações entre usuários. Nesse capítulo são consideradas ferramentas de desenvolvimento de *interface*, para se obter uma *interface* gráfica com usuário que permita uma interação mais livre. Arquitetura *X Window* também é abordada neste capítulo para mostrar uma arquitetura cliente/servidor que se presta para suportar aplicações distribuídas.

O capítulo 5 faz uma descrição da aplicação exemplo desenvolvida. Esta aplicação é um jogo da velha que foi implementado usando a Linguagem de Programação *Python* e o *Toolkit TK*.

O capítulo 6, finalmente, apresenta as conclusões do trabalho.

2. Trabalho Cooperativo Auxiliado por Computador - (CSCW)

2.1 Histórico

O estudo na área de auxílio ao trabalho cooperativo começou no início da década de 70. Organizações preocupadas com seus rendimentos, visualizando que a maior parte do trabalho era feita em grupo, sentiram a necessidade de realizar estudos sobre o comportamento dos grupos ao realizarem alguma tarefa. Com este estudo vários profissionais se aliaram para desenvolver melhores tecnologias para auxiliar o trabalho cooperativo. Além de profissionais da área contribuíram também profissionais de ciências humanas, como sociólogos, psicólogos, educadores [ARA 95].

Em 1984 foi organizado um *Workshop* que teve grande efeito. Vinte pessoas de diferentes campos, e com interesses comuns em estudar como as pessoas trabalham em grupo Estas pessoas analisaram o papel da tecnologia num ambiente colaborativo e então criaram o termo "*Computer Suported Cooperative Work*", que descreve esta cooperação entre os usuários. A partir deste momento milhares de pesquisadores e desenvolvedores começaram a acompanhar esta iniciativa. A primeira conferência de CSCW ocorreu nos Estados Unidos e espalhou-se posteriormente pela Europa e Ásia [GRU 94].

A sigla CSCW foi oficialmente lançada em 1986 como título de uma conferência promovida pela ACM. Após esta vieram várias outras conferências [ARA 95].

Atualmente, esta área ainda é considerada recente, pois a maioria das aplicações desenvolvidas ainda trabalham com a relação individual do usuário com o computador. Por exemplo, processadores de texto, editores gráficos e até mesmo pesquisas na área de *interface* homem-máquina continuam explorando a interação individual.

2.2 Conceito de CSCW

CSCW é uma área da computação que descreve como desenvolver aplicações colaborativas que proporcionam maior eficiência nos requisitos comunicação, coordenação e cooperação. Esta disciplina científica tem também um objetivo teórico e prático de estudar como o aplicativo desenvolvido para prestar auxílio ao trabalho cooperativo deve interagir com as pessoas que trabalham em busca de um objetivo comum.

Computer Supported Cooperative Work refere-se a pessoas trabalhando juntas em um produto ou área de pesquisa com ajuda do computador. Esta área é também conhecida por colaboração suportada por computador, *groupware*, *wokflow* e sistemas de suporte a decisão. É portanto uma área muito fértil de pesquisa, desenvolvimento e produção [PAL 94].

Com os avanços da tecnologia, o Trabalho Cooperativo está sendo cada vez mais introduzido em aplicações utilizadas nas mais diversas áreas. As atividades realizadas individualmente com objetivos comuns entre usuários adquirem um melhor rendimento e desempenho com o uso de aplicações colaborativas, onde a interação dos usuários através do computador permite a cooperação e comunicação entre pessoas independentemente da sua localização geográfica e tipo de sistema de computação utilizado. A partir do surgimento de estudos em CSCW, novos requisitos para desenvolvimento de aplicações estão disponíveis. O sistema deixa de ser uma aplicação usuário/computador e passa a ser uma aplicação que enfatiza a interação entre os usuários tendo o computador como intermediário [ARA 95].

Um fator importante para o sucesso de CSCW é o conforto do usuário com o sistema. Este se dá através de *interface* com o usuário. As *Interface*s precisam tornar-se mais intuitivas, para isto espera-se benefícios provindos das pesquisas nas áreas de Interação Homem-Máquina e multimídia [PAL 94].

O termo *groupware* é muito encontrado na literatura quando o assunto é CSCW. Vários autores consideram estes quase como sinônimos, mas existem outros que consideram *groupware* como sendo a aplicação desenvolvida pelo estudo de CSCW, ou seja, a tecnologia resultante de pesquisas nesta área. O objetivo de um *groupware* é apoiar a comunicação, colaboração e coordenação fig. 1 das atividades de um grupo [NIT 2000].

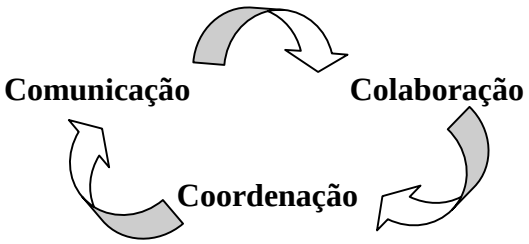

Figura 1: Objetivos de um *Groupware*

Diversas categorias de produtos procuram explorar a nova tecnologia de trabalho cooperativo, geralmente em áreas específicas, como por exemplo Editores e planilhas para uso em grupo, Videoconferência, Correio eletrônico, Sistemas de *Workflow* (sistemas de *workflow* são sistemas que facilitam e coordenam as atividades em grupo como, por exemplo, o processamento de transações financeiras).

2.3 Classificação de um Sistema CSCW

Diversas tentativas de classificação para estudos em CSCW têm sido feitas. Uma aplicação pode auxiliar a interação de usuários que estão face-a-face ou a um grupo que está distribuído em diversos locais. Além disso, pode ampliar a comunicação e a colaboração nas interações síncronas ou assíncronas. A principal finalidade da classificação das aplicações desenvolvidas em CSCW é a fácil identificação de elementos com propriedades comuns. Com isto pode-se classificar os tipos de interações possíveis e suas características. Para entender e solucionar as questões apresentadas pelas aplicações CSCW, é preciso uma conceitualização de como as pessoas trabalham [PAL 94].

DeSanctis e *Gallupe* [DES87], citados por *Grudin* [GRU94] propõem uma classificações de sistemas de CSCW segundo os critérios de espaço e tempo, que considera o local em que os usuários interagem, que pode ser no mesmo local ou em locais diferentes e o tempo em que a interação acontece, que pode ser ao mesmo tempo ou em tempos diferentes. A fig. 2 [NIT 2000] representa a taxionomia de espaço/tempo, que sugere quatro categorias de *groupware*: mesmo tempo e local (salas de reuniões eletrônicas); mesmo tempo e locais diferentes (editor colaborativo); tempos diferentes e mesmo local (quadro de avisos) ou tempos diferentes e locais diferentes (correio eletrônico).

O mesmo *Grundin* [GRU94] propõe uma expansão dessa classificação, considerando também um critério de previsibilidade da comunicação, ou seja, se a atividade ocorrerá em um momento previsível ou não, ou se de alguma maneira pode-se prever o local em que a atividade ocorrerá.

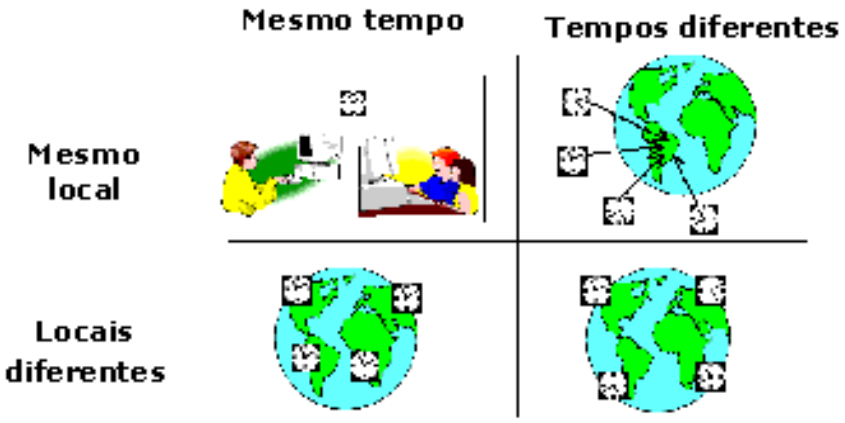

Figura 2: Taxionomia Espaço/Tempo [NIT 2000]

Nunamaker [NUN 91], citado por [ARA 95], apresenta uma outra forma de classificação, cujo critério é o tamanho do grupo. Esta considera que sistemas CSCW podem ainda ser classificados de acordo com o número de pessoas que fazem parte da interação.

Outra classificação apresentada para sistemas CSCW baseia-se no critério de sincronismo. [PAL 94]. Nesta classificação existem quatro modos de trabalho em sistemas colaborativos:

1. **Trabalho síncrono.** Os usuários podem estar no mesmo lugar e ao mesmo tempo trabalhando em diversas atividades de grupo.(ex. videoconferência).
2. **Trabalho síncrono distribuído.** Se as atividades forem realizadas ao mesmo tempo, mas os usuários estiverem em locais diferentes (ex. e-*mail*).
3. **Trabalho assíncrono.** Se estas atividades estiverem acontecendo em tempos diferentes mas em um mesmo local.
4. **Trabalho assíncrono distribuído.** Se estas atividades estão acontecendo em diferentes locais e em diferentes tempos.

Nesta classificação, parece que tudo é muito simples: ou temos usuários separados ou temos usuários trabalhando em um mesmo local e tudo funciona normalmente, mas o problema está justamente na maneira como as coisas funcionam na

prática: existem muitas implicações sociais e técnicas incluídas nesta classificação. No ponto de vista social, o modo como ocorre o encontro físico das pessoas em uma reunião é bem diferente de como ocorre uma simulação dessa comunicação num ambiente virtual, mesmo que tudo ocorra em tempo real. Esta é uma das causas que preocupa a implantação de técnicas de CSCW no que diz respeito aos aspectos sociais.

O problema encontrado no enfoque técnico está relacionado com a necessidade de transmissão de grande volume de dados como, por exemplo, dados multimídia. Outra preocupação está na forma de coordenar as atividades dos cooperantes de forma eficiente e consistente. Um exemplo dessa atividade é como evitar o sobrecarga da *interface* visual do participante no momento em que se torna públicas todas as atividades feitas pelos participantes do sistema.

Algumas pessoas criticam esta classificação pelo fato de os usuários necessitarem de sistemas que satisfaçam as quatro categorias da classificação ao mesmo tempo, ou seja, o sistema deve ter uma flexibilidade que atenda tanto as interações assíncronas como as síncronas e também usuários remotamente distribuídos ou não [ARA 95].

2.4 Implicações práticas das Classificações

As classificações apresentadas pela bibliografia ajudam a separar as aplicações de acordo com características comuns entre elas, e são muito importantes no desenvolvimento de aplicações de apoio ao trabalho colaborativo. Quando se desenvolve uma aplicação onde haverá a interação de mais de um usuário no mesmo modelo, é necessário que se classifique que tipo de interação ocorrerá nesta aplicação. Por exemplo, quando dois jogadores estão jogando algum jogo através do computador, é preciso que existam algumas restrições nas interações dos jogadores. Num jogo onde apenas um jogador pode jogar de cada vez, o trabalho é síncrono. Isto significa que deve haver um sincronismo entre os jogadores na execução das jogadas, o jogo não pode permitir que um mesmo jogador jogue duas vezes seguidas. Mas isto não quer dizer que a comunicação é necessariamente síncrona, os dois jogadores podem tentar jogar de maneira assíncrona, ou seja, sem ordem preestabelecida, mas o próprio modelo do jogo deve impedir que a jogada seja realizada. Portanto esta classificação ajuda a definir

algumas metas a serem seguidas pela aplicação a ser desenvolvida, definindo os tipos de interações possíveis entre os usuários. Estas interações variam nas diferentes aplicações.

De certa forma a classificação quanto à previsibilidade não apresenta tanta importância prática, devido à dificuldade de previsão das interações entre as pessoas. Dificilmente sistemas de uso geral podem oferecer uma real previsibilidade tanto de espaço como de tempo. Com os grandes recursos existentes atualmente na área de sistemas distribuídos, torna-se difícil prever o lugar em que vão ocorrer as interações. Por exemplo no uso do correio eletrônico não se pode prever o lugar onde irá ocorrer uma troca de mensagem, e nem o tempo previsto para a resposta. Porém em alguns sistemas mais específicos esta classificação pode ser observada mais precisamente. Por exemplo, na realização de uma cirurgia utilizando uma aplicação CSCW a resposta a uma interação pode exigir um tempo mínimo previsto, e até mesmo o local deve ser provavelmente adequado para este tipo de trabalho.

3. Programação Concorrente e Distribuída

3.1 Conceitos Básicos

Quando se trata do assunto CSCW, o aspecto mais importante nessa área é a forma de colaboração entre os usuários. Para que se tenha uma aplicação CSCW que permita colaboração entre usuários que usam a mesma aplicação em diferentes lugares, é necessário um sistema que possibilite a distribuição da aplicação através de uma rede. Portanto, para desenvolver programação colaborativa é necessário o uso de sistemas distribuídos [REI 94]. Os sistemas distribuídos são caracterizados pela execução de vários processos que interoperam através de uma rede para resolver um problema computacional comum. Cada processo é um programa que executa uma seqüência de operações.

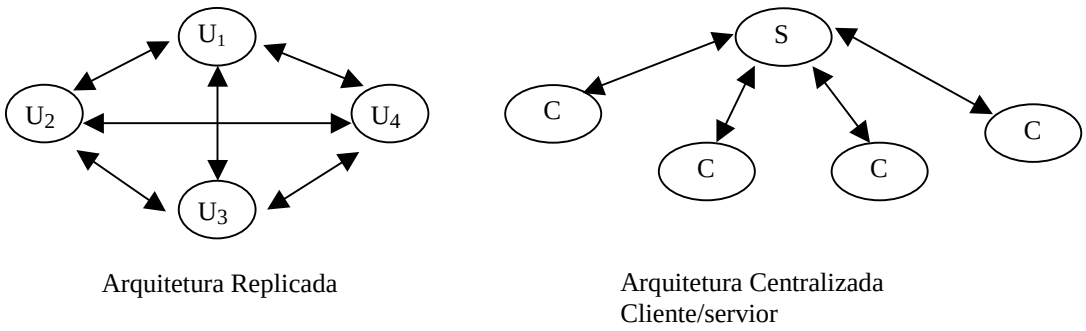

Figura 3: Exemplo de Arquiteturas Distribuídas

Existem diferentes tipos de arquiteturas distribuídas utilizadas em sistemas CSCW. Arquitetura centralizada, arquitetura replicada e arquitetura híbrida [BEN 94]. Uma arquitetura centralizada como a arquitetura Cliente/Servidor, o servidor é responsável pela sincronização dos usuários. Um programa servidor central trata todas as entradas dos usuários e mostra as saídas, as quais são roteadas via programas clientes locais. Isto significa que cada usuário que deseja se comunicar com outro deve requisitar ao servidor. Esta comunicação entre servidor e cliente se dá através da troca

de mensagens, onde o cliente requisita um serviço ao servidor através de uma mensagem e este responde ao cliente através de outra mensagem. Arquitetura replicada mantém cópias idênticas da aplicação em cada estação. Para garantir consistência das informações cada réplica trata do gerenciamento da vista da aplicação, faz a atualização dos dados locais e transmite qualquer mudança nos dados para as outras réplicas. Os usuários se comunicam diretamente uns com os outros, sendo cada cliente responsável pela sincronização. A fig. 3 mostra as arquiteturas extremas centralizada e replicada. A seguir algumas características das arquiteturas distribuídas [BEN 94]:

Arquitetura Centralizada:

1. Fácil implementação.
2. Fácil adição e remoção de *displays*.
3. Fácil de manter a consistência.
4. Dificuldade de fornecer diferentes níveis de compartilhamento.
5. Dificuldade de fornecer rápida atualização das informações.

Arquitetura Replicada:

1. Facilidade de fornecer diferentes níveis de compartilhamento.
2. Facilidade de fornecer rápida atualização das informações.
3. Difícil implementação.
4. Difícil adição e remoção de displ*a*ys.
5. Difícil de manter a consistência.

As arquiteturas centralizada e replicada apresentam benefícios e limitações. Assim, arquiteturas híbridas são necessárias quando os componentes do sistema cooperativo são ou centralizados ou replicados, de acordo com o requerimento da aplicação. A arquitetura é híbrida se a informação compartilhada é mantida consistente em um componente centralizado, enquanto a apresentação e a interação são replicadas nos agentes distribuídos.

3.2 Sincronização

A sincronização é de extrema importância quando se fala em interação entre usuários, para que haja a garantia de coordenação da aplicação [REI 94]. Numa aplicação CSCW existe a necessidade da coordenação das atividades dos usuários. Como o trabalho em grupo envolve múltiplos agentes executando diferentes atividades concorrentemente, tarefas de coordenação tem um impacto decisivo na produtividade do grupo [LIA 94].

Quando os usuários interagem numa aplicação colaborativa, estes são dois ou mais processos que cooperam na realização de uma tarefa através de um sistema distribuído. Os processos comunicam-se através de variáveis compartilhadas ou através de troca de mensagens. Quando a comunicação se dá através de variáveis compartilhadas um processo escreve em uma variável a qual é lida por outro processo. Quando a comunicação se dá através de troca de mensagens um processo manda uma mensagem e esta é recebida por outro processo. Quando existe alguma forma de comunicação entre processos, para que esta funcione normalmente é preciso que exista sincronização [AND 91].

Necessidade de sincronização significa que duas operações pertencentes a processos distintos não devem ser executadas ao mesmo tempo: uma operação deve ser executada após a total realização da outra [AND 91]. Por exemplo numa aplicação colaborativa onde mais de um usuário tem acesso a dados comuns a todos, deve-se utilizar alguma forma de sincronização para que não haja a possibilidade de mais de uma pessoa alterar estes dados ao mesmo tempo. Pode ocorrer que ao mesmo tempo em que alguém está alterando um dado outro usuário esteja utilizando este dado que não possui mais o conteúdo real.

No caso do jogo da velha por exemplo, a classificação em CSCW é trabalho síncrono, ou seja, a comunicação se dá em tempo real. Um jogador só pode jogar novamente quando o outro jogador fizer a sua jogada. Isto significa que as jogadas devem ser sincronizadas. Duas formas de sincronização ocorrem na programação concorrente [AND 91]:

1. **Exclusão mútua**. Se preocupa em garantir que processos que acessam a região crítica não executem ao mesmo tempo nesta região, ou seja, garantia de acesso exclusivo a recursos compartilhados. Um exemplo de exclusão mútua

é a comunicação entre um processo produtor e um processo consumidor, geralmente esta comunicação é implementada usando-se um *buffer* compartilhado, nele a mensagem enviada é escrita pelo produtor e lida dele pelo consumidor. A exclusão mútua garante que a mensagem não seja lida enquanto o processo produtor ainda esta escrevendo no *buffer*.

2. **Condição de sincronização**. Esta relaciona-se com a garantia de que um processo seja bloqueado quando necessário até que a condição seja verdadeira, ou seja, até que ele receba um aviso para que seja desbloqueado. Por exemplo, quando um processo precisa esperar pelo resultado produzido pelo outro.

Uma maneira de se estabelecer uma sincronização na comunicação dos usuários é o uso de mensagens. No caso de uma aplicação colaborativa que está distribuída numa arquitetura cliente/servidor, os usuários interagem de maneira concorrente de forma que um usuário faz a requisição de uma tarefa através de uma mensagem e aguarda uma mensagem de resposta do servidor, pois o servidor deve estabelecer a sincronização entre os usuários através da troca de mensagens.

Na programação concorrente existe um problema denominado *deadlock*, que ocorre quando programas ficam parados aguardando por respostas que jamais serão enviadas. Por exemplo, um programa P1 está aguardando uma resposta do programa P2, P2 está aguardando resposta de P3, este por sua vez está aguardando o programa P1. Portanto eles ficaram aguardando um ao outro permanentemente. Uma das vantagens da arquitetura centralizada é a prevenção do *deadlock*.

3.3 Ações Atômicas e Transação

O estado de um programa concorrente em qualquer momento da execução é determinado pelo valor das variáveis do programa [AND 91]. Sempre que um programa concorrente inicia sua execução ele possui um estado inicial.

A execução de cada instrução consiste de uma seqüência de uma ou mais ações atômicas, cada uma das quais faz uma transformação indivisível do estado do programa. Isto significa que uma ação atômica é ininterrupta, ela é executada de uma vez só. Um exemplo de ação atômica é o armazenamento de um valor num registro.

Uma seqüência de ações atômicas é gerada quando um programa concorrente é executado. Cada execução produz uma história que pode ser diferente a cada estado inicial diferente. De acordo com *Andrews* [AND 91] para qualquer programa concorrente várias são as histórias possíveis de acontecer no decorrer da execução das ações atômicas, conforme mostra a fig.4.

S_0: estado inicial

α_i: ação atômica i

$$S_0 \xrightarrow[\alpha_1]{} S_1 \xrightarrow[\alpha_2]{} S_2 \xrightarrow[\alpha_i]{} S_i \xrightarrow[\alpha_{i+1}]{} \ldots$$

Figura 4: História resultante da execução das ações atômicas [AND 91]

De acordo com esta visão o papel da sincronização é construir uma possível história de um programa concorrente de maneira a chegar no resultado desejável.

Um outro conceito importante quando se trata de programação concorrente é o de Transação. Transação é uma seqüência de ações atômicas que devem ser executadas de maneira exclusiva e indivisível. Quando acontece uma transação significa que as instruções a serem executadas durante a transação representam uma região crítica, região esta que só pode ser acessada por um processo de cada vez.

Numa aplicação colaborativa onde os usuários têm acesso a dados compartilhados por exemplo, uma alteração feita nesses dados deve ser uma transação, pois apenas um dos usuários de cada vez pode alterar os dados e além disso os dados só podem ser realmente alterados após o termino da transação.

Para que uma aplicação CSCW cumpra seus objetivos de proporcionar uma cooperação entre os usuários através do computador, é necessário que haja consistência das informações [BEN 94]. A transação é um aspecto fundamental na consistência dos dados alterados por diferentes usuários. Sempre que houver alguma alteração na informação todos os componentes que contem esta informação devem ser atualizados antes de qualquer outra manipulação dessa informação.

3.4 Arquitetura Cliente/Servidor

As aplicações CSCW se diferem de aplicações tradicionais porque permitem que pessoas trabalhem juntas eletronicamente, no entanto requer um ambiente de *hardware* distribuído [LIA 94]. A arquitetura cliente/servidor é um modelo conceitual, usado para orientar aplicações que estão funcionalmente separadas em processos distintos, ou seja, as aplicações distribuídas. A arquitetura cliente/servidor facilita o compartilhamento das informações, disseminação, roteamento e interação do usuário.

Aplicações distribuídas são caracterizadas por vários processos que interoperam para resolver um problema computacional comum. Isto requer uma comunicação segura e coordenada. Na arquitetura cliente/servidor, uma aplicação distribuída é conceitualmente modelada para ser composta por dois tipos de processos cooperantes, o processo cliente e o processo servidor. Em sistemas CSCW que utilizam arquitetura cliente/servidor divide as atividades em dois módulos. Um que processa as atividades individuais e o outro que coordena, ou seja, junta as atividades individuais em um conjunto. Numa arquitetura cliente/servidor todos os usuários interagem através do processo cliente e os seus *displays* são tratados centralmente pelo processo servidor [LIA 94].

A comunicação entre o cliente e o servidor se dá através da requisição que o usuário final faz ao processo cliente e este, por sua vez, requisita ao processo servidor que tenta resolver o problema requisitado. Sempre que o processo cliente faz uma requisição bloqueante ao servidor, fica aguardando uma resposta no estado bloqueado. Quando o servidor tiver uma resposta positiva ou negativa este deve comunicar o cliente que passa a ficar desbloqueado, ou seja, fica pronto para fazer uma nova requisição ao servidor. Nos casos em que a requisição não é bloqueante, o cliente simplesmente manda uma mensagem ao servidor quando quer fazer uma requisição.

A principal vantagem do sistema cliente servidor é a simplicidade [LIA 94]. A aplicação e todos os dados são mantidos centralmente, isto torna mais simples o gerenciamento dos acessos a dados compartilhados e facilita o controle da consistência dos dados.

4. Aspectos de *Interface*

4.1 Considerações Gerais sobre *Interface*s

Nos últimos anos as pessoas têm usado cada vez mais o computador não só como ferramenta de trabalho como também uma forma de comunicação e lazer. A maioria delas utilizam o computador com bastante freqüência devido ao crescimento da tecnologia no trabalho diário, por isso é muito importante que o computador se torne uma ferramenta de uso fácil e agradável que proporcione conforto ao usuário.

Tempo é valioso, as pessoas não querem ler manuais, elas querem gastar seu tempo realizando seus objetivos, não aprendendo como operar um sistema baseado em computador [MYE 93]. Um aspecto importante a ser considerado quando se desenvolve a *interface* de uma aplicação, é colocar o problema de forma adequada de maneira que as pessoas a serem beneficiadas por ela estejam aptas a usá-la. Para satisfazer esta característica de *interface* deve-se considerar o fato de os usuários serem dos tipos mais diversos, desde aquele que já tem experiência com o computador até aquele que não usa o computador por vontade própria mas sim por necessidade. Portanto, uma *interface* deve ser bem planejada de forma que a comunicação com o usuário se dê da forma mais próxima possível da comunicação humana.

A *interface* com o usuário se torna quase tão importante para a aplicação quanto a solução do problema computacional, porque é através dela que o usuário consegue alcançar o seu objetivo computacional. Portanto para projetar uma *interface* de boa qualidade e eficiência não basta o programador se colocar no ponto de vista do usuário, esta deve ser criada por especialistas que tenham habilidades para lidar com aspectos sociais e psicológicos.

Segundo *Myers* [MYE 93], existem cinco motivos que dificultam o projeto de uma *interface*, são eles:

1. Dificuldade de entender os tarefas dos usuários. É necessário que se conheça este usuário para saber o que realmente ele deseja.
2. A complexidade das tarefas e aplicações. Quanto mais complexo for o problema mais difícil será desenvolver uma *interface* de fácil aprendizado.
3. A variação de diferentes aspectos e requerimentos, como necessidade de padrões (como *Macintosh*, *Windows* ou *Motif*), representação gráfica dos dados (*layout*, cores, ícones,...), documentação, mensagem e ajuda, internacionalização, desempenho, nível de detalhamento.
4. Teorias e metodologias para que um produto tenha uma boa *interface* com o usuário.
5. Dificuldade em desenvolver um projeto iterativo de *interface*, onde a *interface* será desenvolvida inicialmente com funcionamentos básicos de maneira que possa ser testada quanto à sua funcionalidade.

4.2 *Interface* com o Usuário e Multiusuário

Atualmente, todos os desenvolvedores de sistemas de computação estão interessados em características de *interface* com o usuário que sejam comuns aos diversos produtos e que proporcionem facilidade de uso. Isto se deve à necessidade da redução de treinamento de pessoal quando da aquisição de um novo produto. Estas características comuns são geralmente denominadas na bibliografia de "aparência e sensação" (*look-and-feel*), com isto o grande interesse das pessoas está em produtos que ofereçam o mesmo *look-and-feel* aos seus usuários [TEI 99]. Com a necessidade da existência de uma ferramenta que satisfaça este objetivos foram desenvolvidas as *Interface*s Gráficas de Usuários conhecida pelo termo em inglês *Graphical User Interface* (GUIs).

O surgimento da tecnologia das GUIs aconteceu no laboratório PARC (*Palo Alto Research Center*) da Xerox, nos anos 70, quando foi desenvolvida a estação de trabalho *Star* [TEI 99]. Em 1987, uma parceria entre a IBM e a *Microsoft* possibilitou a criação da GUI *Presentation Manager* (PM) para os microcomputadores compatíveis com o IBM PC, esta foi projetada especificamente para o sistema operacional OS/2. No mundo *Unix*, a GUI *Motif* tornou-se padrão *de facto*, ela também baseou-se na *Presentation Manager*.

A interação humano-humano através do computador é uma nova visão associada aos sistemas classificados como CSCW. Isto significa que a *interface* desse tipo de sistema leva em consideração vários aspectos importantes na comunicação entre usuários de forma sincronizada, sendo então considerada *interface* multiusuário [ROM 2000]. A fig. 5 mostra como funciona uma interação entre o usuário e um sistema de *interface* multiusuário.

Considerando as diversas formas de comunicação possíveis em sistemas CSCW percebe-se que uma *interface* multiusuário requer muitos cuidados na sua implementação devido à complexidade do sistema. Muitos sistemas de computação já suportam interações simultâneas para mais de um usuário [BEN 94].

Há alguns anos atrás os sistemas na maioria das vezes não forneciam um componente de *interface* com o usuário de modo explícito. A implementação da *interface* se dava juntamente com o programa a ser desenvolvido, que utilizava outros componentes de sistema, tais como geradores de telas orientadas a caracteres e também funções que eram incluídas nos códigos dos programas com o objetivo de ativar essas telas [MYE 93].

Para suportar e incentivar cooperação, aplicações cooperativas devem permitir que usuários estejam cientes das atividades dos outros [BEN 94]. A finalidade de uma *interface* multiusuário cooperativa é estabelecer e manter um contexto comum, permitindo que as atividades de um usuário sejam refletidas nos *displays* dos outros usuários [BEN 94]. A *interface* multiusuário deve apresentar a informação em tempo real.

Quando um usuário estabelece uma comunicação com outro usuário, usando o computador ele não faz uma interação com o computador mas através dele. *Interface*s multiusuário usadas em aplicações colaborativas possuem um papel muito importante nesta interação através do computador. Nesta aplicação mais de um usuário pode interagir no mesmo programa através da *interface*. Esta deve sempre manter as informações atualizadas para todos os usuários de maneira transparente para que cada um possa visualizar as interações feitas pelos outros. Uma *interface* monousuário preocupa-se em mostrar os estados de um modelo apenas para um usuário, em uma *interface* multiusuário as atualizações devem ser mostradas igualmente para todos os usuários ao mesmo tempo. Logo, uma *interface* multiusuário precisa fornecer consistência na representação do modelo.

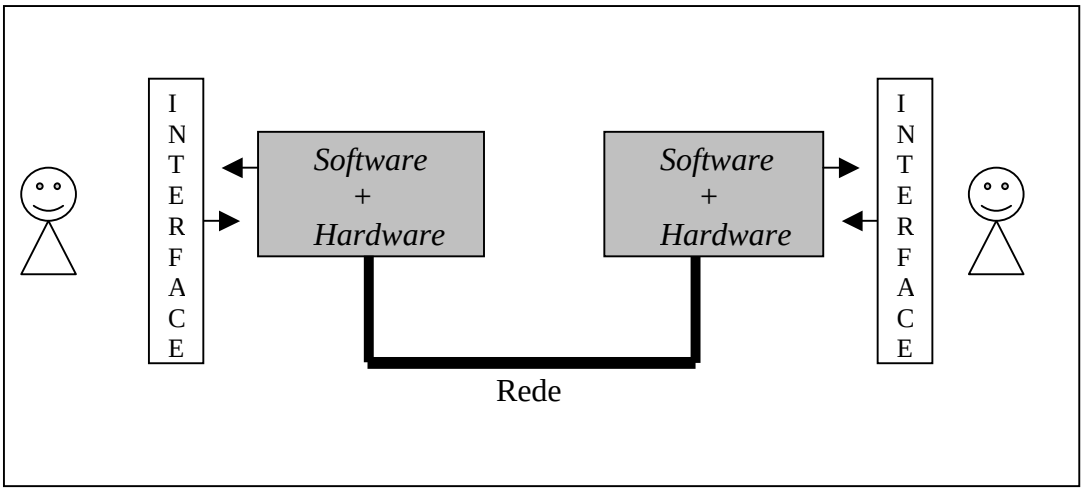

Figura 5: *Interface* Multiusuário (baseado em [PIM2000])

4.3 Arquitetura Modelo-Vista-Controlador

A arquitetura Modelo-Vista-Controle, mais conhecida na literatura pelo termo em inglês *Model-View-Controller* (MVC), foi introduzida como parte da versão Smaltalk-80 da linguagem de programação *Smaltalk*,. Pela necessidade de mecanismos que melhorassem a qualidade e a produtividade do trabalho, desenvolveu-se esta arquitetura que tornou-se um padrão de projeto [SIP 98].

A arquitetura MVC se caracteriza por representar os mesmos dados de forma múltipla e sincronizada, e também redução do esforço de programação para este tipo de sistema devido à independência de seus componentes. Além desses aspectos existem várias vantagens a serem observadas neste paradigma, são elas:

1. Oferece grande confiabilidade, permite isolar os componentes facilitando o desenvolvimento.
2. Simplificação da especificação do *software*, é apenas necessário que se defina as *interface*s entre os componentes.
3. A possibilidade de se ter diferentes implementações do mesmo componente, desde que seja preservada a *interface*.
4. Permite desenvolver produtos reutilizáveis.

Os componentes básicos são: Modelo, Vista e Controlador [SUN 96]. O Modelo é o objeto que representa os dados da aplicação, sendo responsável pelas alterações dos dados requisitadas pelo usuário e pela tarefa de disponibilizar as informações sobre os dados ao componente vista de forma consistente e correta.

1. **Modelo** é o principal componente desta arquitetura, pois ele é a peça central do sistema responsável pelas mudanças reais no estado da aplicação, obtendo e fornecendo informações dos outros componentes.
2. **Vista** é a apresentação do objeto na tela. Ela deve assegurar que sua aparência reflita o estado do Modelo. Para que isto aconteça sempre que houver alterações nos dados do modelo, este deve informar a vista para que o usuário possa ver as alterações realizadas.
3. **Controlador** é o objeto que controla as interações do usuário com a *interface*. Ele interpreta as entradas do usuário através do mouse ou teclado e informa a vista ou o modelo as alterações solicitadas.

Cada componente é responsável por uma tarefa específica, pois são desenvolvidas separadamente [SUN 96]. Eles são partes de um programa que se comunicam através de um ajuste feito na *interface* de cada uma das classes.

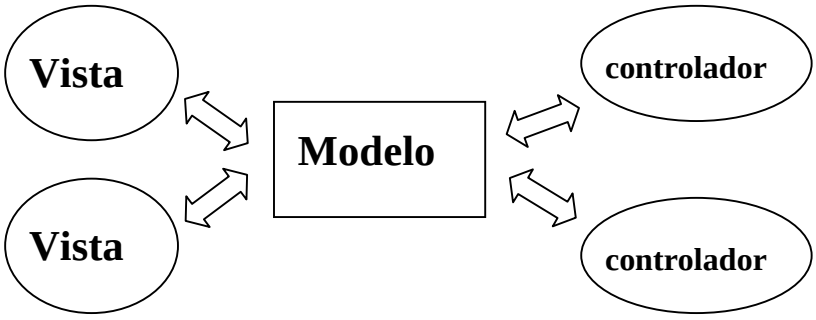

Figura 6: Componentes do MVC (baseado em [SUN 96])

A maneira como são organizados os componentes dentro do paradigma MVC oferece uma grande confiabilidade. Além disso, facilita o desenvolvimento das aplicações, capacitando a detecção de problemas em cada um dos componentes separadamente, possibilitando a substituição do componente sem a necessidade de alterações no sistema como um todo. Por exemplo, se algum componente não funcionar o problema pode ser resolvido mais rapidamente pela substituição de apenas este componente.

Com a divisão dos componentes, as implementações feitas em cada um podem ser diferentes, variadas as plataformas e sistemas, apenas é exigida a definição das *interface*s entre os componentes, e a preservação delas.

4.4 Desenvolvimento de Aplicações

Para se obter uma *interface* amigável e de grande eficiência em sistemas CSCW é necessário uma linguagem de programação que consiga proporcionar uma maior interação entre usuários e permita representar os dados de forma gráfica. Para que exista uma padronização da *interface* das diversas aplicações é necessário que os desenvolvedores de *interface* usem os chamados *toolkits,* conjuntos de bibliotecas que fornecem tipos de dados, e funções comuns entre as aplicações. Além disso também devem usar o *manual de estilo,* o qual possui um conjunto de regras que determina a aparência e o comportamento das aplicações.

Com esta padronização os *toolkits* e manuais de estilo são fazem parte de uma classe de *software* chamada Sistemas de Gerenciamento de *Interface* com o Usuário (*User Interface Management Systems*-UIMS). Um exemplo de UIMS é o *Motif* e o manual de estilo correspondente. Um outro conceito relacionado com construção de *interface* são os ambientes de desenvolvimento de *interface*s com o usuário (*User Interface Development Environments*-UIDE), ferramentas para a construção de *interface*s [BOS 93].

Um modelo de referência para *interface* gráfica de usuário é apresentado a seguir para descrever os componentes dos sistemas que disponibilizam *interface*s gráficas de usuário [TEI 99].

No modelo da fig. 7, a camada superior contem contém programas de aplicação que têm acesso à *interface* gráfica do usuário através de um *toolkit*. Muitos *toolkits* de utilizados no desenvolvimento de *interface*s já foram desenvolvidos, elas oferecem recursos gráficos que facilitam a programação, são responsáveis pela criação e manutenção de objetos como menus e janelas e ainda oferecem recursos que permitem uma redimensionamento e movimentação desses objetos na tela dos terminais. Essas facilidades é que determinam o *look-and-feel*. Na terceira camada o sistema de janelamento implementa as características básicas necessárias para uma *interface* gráfica do usuário (GUI). Na última camada está o sistema operacional.

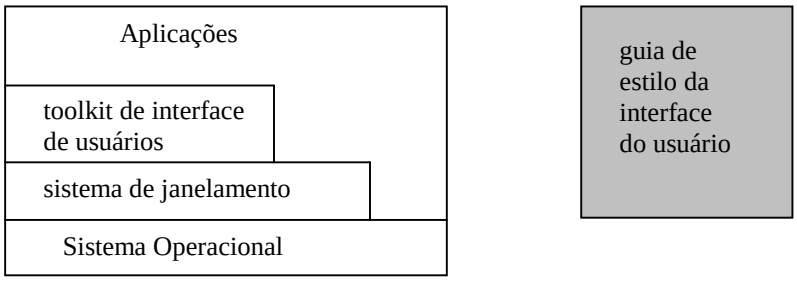

Figura 7: Modelo de referência em camadas para as *interface*s gráficas de usuário.

4.5 Arquitetura *X Window System*

O *X Window System* possui uma arquitetura cliente/servidor, o servidor localiza-se na estação de trabalho do usuário, proporcionando acesso a dispositivos de entrada e saída [KOJ 2000]. Os clientes podem rodar na estação local ou em qualquer outra estação remota que conectam-se, através de uma rede, ao servidor que é responsável por controlar o dispositivo de vídeo que capacitará a visualização da *interface* do cliente. Uma aplicação que desejar saídas gráficas no terminal deve enviar informações ao servidor X que descrevem a tarefa a ser realizada.

Os diferentes tipos de mensagens enviadas ao servidor X são denominadas em conjunto como protocolo X. Qualquer aplicação capaz de exibir as saídas gráficas, pela troca de mensagens através do protocolo X, é denominada cliente X.

O *X Window System* é um sistema independente de *hardware* e sistema operacional, isto determina que as estações de uma rede não precisam ser do mesmo fornecedor ou devam suportar o mesmo sistema operacional.

Na comunicação entre cliente e servidor podem haver três tipos de requisição: *Request*, *Replay* e *Event*. *Request* que faz a requisição do serviço ao servidor, *Replay* que responde as requisições, ou *Event* para eventos como movimentação do mouse, redimensionamento de uma janela.

Os *toolkits* são bibliotecas que implementam *interface*s de programação para o *X Window System* [KOJ 2000]. Eles geralmente fornecem rotinas que constroem menus, botões, controles deslizantes e outros. Existem vários *toolkits*, dentre os quais podemos destacar:

1. ***X Toolkit.*** O *toolkit Xt* consiste de funções básicas que dão suporte a outros *toolkits* mais completos. Pois o *Xt* não possui *widgets* completos, foi desenvolvido para auxiliar no desenvolvimento de *toolkits* gráficos.
2. ***Athena* (*Xaw*).** Foi criado durante a pesquisas do Projeto *Athena*. *Xaw* nunca foi um toolkit completo, por Ter originado de exercício acadêmico carece de muitos recurso existentes em outros *toolkits*. Apresenta muito problemas.
3. ***Motif.*** É um *toolkit* desenvolvido pela antiga OSF e fornece um visual em terceira dimensão, sempre foi considerado uma ferramenta profissional.
4. ***Tcl/TK.*** Tcl é uma linguagem de *scripts* e *TK* é seu *toolkit* gráfico. *TK* pode ser usado com diversas linguagens de programação, além de TCL, e há *interface*s para C *Python*, *Perl*, *Scheme*. Tk foi o toolkit usado na parte prática deste trabalho.

Uma das vantagens do uso do *X window system* é sua arquitetura distribuída, que pode ser usada para facilitar o desenvolvimento de aplicações colaborativas. Desta forma, poupa-se o trabalho que se teria em incluir no *software* desenvolvido todo o suporte necessário a operação em um ambiente com múltiplas estações de trabalho.

Uma aplicação cliente servidor pode ocorrer com vários clientes e um servidor, ou com um cliente e um servidor, como mostra a fig. 8.

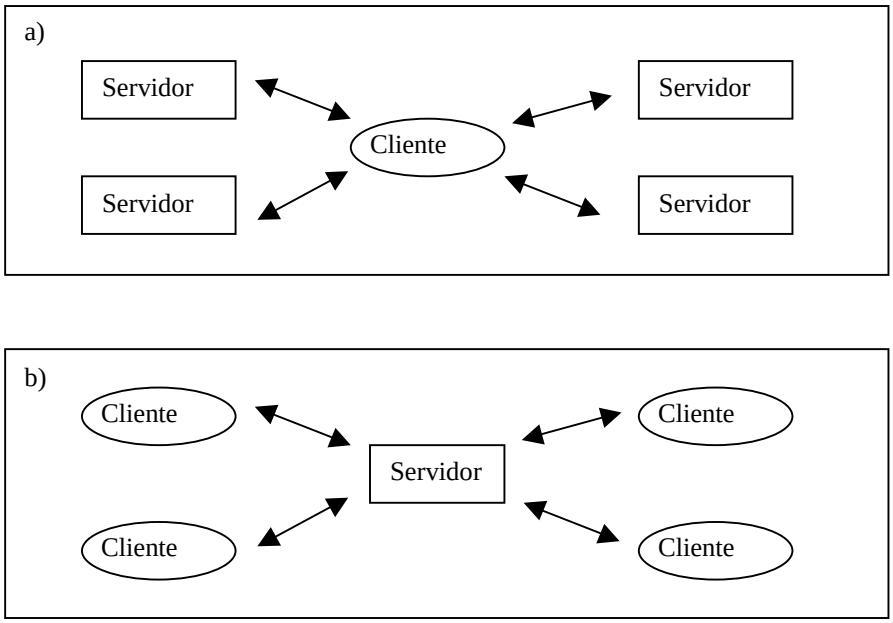

Figura 8: a)Vários servidores e um cliente. b) Vários clientes e um servidor.

5. Desenvolvimento da Aplicação Exemplo

5.1 Descrição da Aplicação

A aplicação exemplo desenvolvida neste trabalho foi um jogo conhecido popularmente como Jogo da Velha. O jogo foi implementado ao longo do trabalho usando técnicas e ferramentas adequadas para a realização do completo funcionamento da aplicação que permite que dois jogadores interajam em um mesmo modelo de maneira sincronizada.

O desenvolvimento deste jogo se deu em etapas diferenciadas, que foram evoluindo de acordo com a implementação. As etapas começaram com algumas definições básicas de funcionamento como possíveis estados do jogo, jogadores, tabuleiro e foram até a necessidade do uso de ferramentas que permitissem uma colaboração entre usuários, oferecendo uma *interface* gráfica de fácil uso.

5.1.1 Funcionamento do Jogo da Velha

O jogo apresenta inicialmente um tabuleiro para cada jogador, com nove casas. É possível apenas a participação de dois jogadores, o jogador "X" e o jogador "O". O tabuleiro é identificado de acordo com o jogador que pode jogar neste tabuleiro, ou seja, o jogador "X" só pode jogar no tabuleiro "X" e o jogador "O" só pode jogar no tabuleiro "O". Os tabuleiros são vistos separadamente como se fossem um para cada jogador, mas internamente na implementação do modelo o tabuleiro é um só, cada jogada feita num tabuleiro é mostrada automaticamente no outro.

O jogo inicia quando algum jogador clica em uma casa do tabuleiro, após isto acontecer esta casa estará ocupada pelo jogador que jogou, o próximo a jogar será o outro jogador que ocupará também uma casa que deve estar desocupada. Posteriormente os jogadores vão fazendo jogadas alternadas, ou seja, uma vez para cada um. As jogadas

ocorrem quando cada jogador clica numa casa ainda não ocupada, e assim passa a ocupá-la.

Existem duas possibilidades de fim de jogo: ou algum jogador ganha ou as casas ficam todas ocupadas e nenhum jogador ganha. Para que algum jogador ganhe é necessário que as casas ocupadas por ele no tabuleiro formem uma linha horizontal, vertical ou diagonal. Quando as nove casas no tabuleiro estão ocupadas e nenhum jogador formou uma linha para ganhar o jogo, não houve ganhador. Após uma ou outra forma de fim de jogo os jogadores podem escolher iniciar um novo jogo ou terminar o jogo.

5.1.2 Definição dos Estados do Jogo

Para que seja iniciada a implementação da aplicação é necessário que sejam definidos os estados do jogo. Quando se fala em estado do jogo isto significa que existem diferentes momentos enquanto a aplicação está sendo executada. De acordo com as ações realizadas pelos usuários a aplicação pode se comportar de diferentes maneiras, cada ação pode gerar diferentes resultados.

Existem quatro estados possíveis: o estado inicial, onde nenhum jogador está jogando e o tabuleiro está vazio, o segundo é o estado em que o jogador X está jogando, o terceiro é o estado em que o jogador O está jogando, e o último é o estado de fim de jogo.

No primeiro estado deve estar garantido que qualquer um dos jogadores pode jogar e que o tabuleiro deve estar vazio. No estado em que um jogador está jogando não deve ser permitido que o outro possa jogar antes dele. No estado final, nenhum jogador pode jogar enquanto o jogo não for reiniciado.

5.1.3 Definição dos Jogadores

Quando se desenvolve uma aplicação colaborativa, ou seja, mais de um usuário interage no mesmo modelo, é preciso uma devida atenção na definição dos possíveis usuários da aplicação. Saber quantas pessoas podem interagir neste modelo e de que forma irão fazer esta interação é um dos primeiros passos a serem seguidos.

No caso desta aplicação existem apenas dois usuários o jogador "X" e o jogador "O". Eles interagem no modelo através da escolha da casa no tabuleiro.

5.1.4 Definição das Possíveis Jogadas

Para saber se uma jogada é válida é preciso saber primeiramente quem é o jogador que está tentando jogar e qual a casa que ele escolheu no tabuleiro. Quando um jogador tenta jogar a aplicação deve saber quem é este jogador e definir se é a sua vez de jogar ou não. Se o jogador está na sua vez de jogar, a aplicação tem que verificar se a posição em que tentou jogar é válida ou não. Uma posição é inválida se já está ocupada por um dos jogadores. Depois que a aplicação determinar que o jogador que está tentando jogar é realmente o próximo a jogar e que a posição é válida, a jogada deve ser então realizada e o estado do jogo automaticamente será alterado. Isto significa que o jogador que acabou de jogar deve ser bloqueado, ou seja, impossibilitado de jogar novamente antes que o outro jogador faça sua jogada.

5.2 A Linguagem de Programação *Python*

Python é considerada uma poderosa linguagem de programação e de fácil aprendizado[VAN 98]. É uma linguagem de programação orientada a objetos que possui uma eficiente estrutura de dados de alto nível. É uma linguagem interpretada, e considerada ideal para *scripts* e desenvolvimento rápido de aplicações em muitas áreas e em várias plataformas[VAN 98].

O interpretador *python* é facilmente estendido com novas funções e tipos de dados implementados em C ou C++.

Python é simples de usar, mas é uma linguagem de programação que oferece poderosas estruturas de dados e suporta grandes programas. Ele permite que o programa seja dividido em módulos que podem ser *reusados* em outros programas *Python*. Ele possui uma grande coleção de módulos padrões para que possam ser usados por outros programas.

Por ser uma linguagem interpretada, pode ajudar a aproveitar melhor o tempo de programação, pois não é necessário compilação e ligação. O interpretador pode ser usado interativamente, o que oferece facilidade para experimentar as características da linguagem, testar funções durante o desenvolvimento do programa.

Python permite escrever programas compactos e legíveis, Os programas escritos em *Python* geralmente são mais curtos do que programas equivalentes escritos em C, porque permite expressar operações complexas em uma única linha de programação, as instruções são agrupadas pela indentação ao invés de *begin/end*,. Não há necessidade de declaração das variáveis.

5.3 Aplicação do MVC

Depois de definidos os jogadores, as possíveis jogadas e os estados do jogo, a aplicação inicia a sua implementação usando a arquitetura MVC, onde o estado do jogo fará parte do modelo, a vista será responsável pela visualização do estados do jogo e o controle permitirá a interação do usuário com o estado do modelo.

A fig. 9 mostra como o modelo, a vista e o controle interagem durante a execução da aplicação:

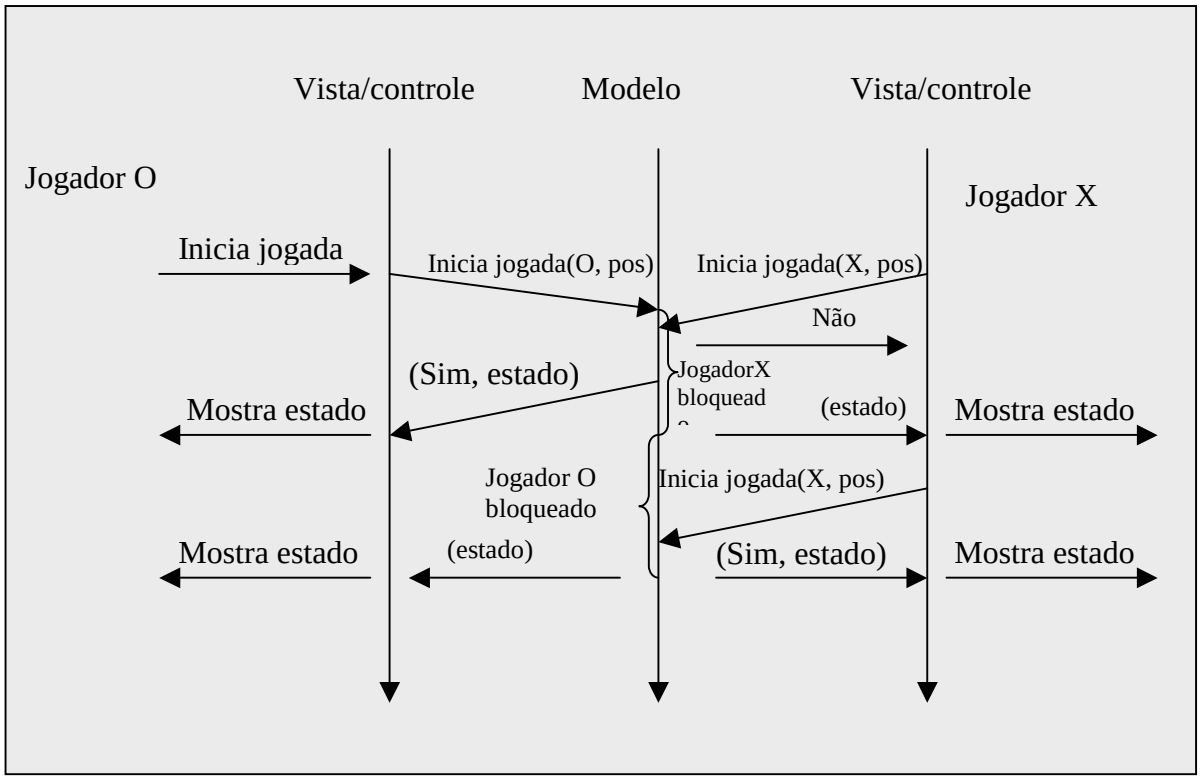

Figura 9: Comunicação entre os componentes da aplicação

A linguagem de programação utilizada para implementar a aplicação foi a linguagem *Python* descrita no item anterior. Esta foi escolhida principalmente por ser uma linguagem orientada a objeto o que facilita o desenvolvimento em MVC, onde os

componentes modelo, vista e controle serão objetos que possuem instâncias e métodos. Além disso é uma linguagem que permite aprendizado rápido e oferece facilidades de programação que tornam o programa simples mas eficiente. Na aplicação exemplo a vista e o controle formam um só componente e o modelo outro componente. A vista e o controle são considerados um só componente porque o objeto que os constitui é o mesmo, mas continuam sendo componentes diferentes na função que desempenham. Isto significa que são dois componentes(duas funções) num mesmo objeto.

5.4 Usando o *Toolkit TK*

O *toolkit* utilizado na aplicação exemplo foi o *Tk* [OUS 94] desenvolvido inicialmente para ser usado com a linguagem de *scripts Tcl* ou com programas em C. Posteriormente foram criadas *interface*s para diversas outras linguagens, entre elas *Python*, que se usou neste trabalho. *Tk* pode ser usado tanto em sistemas UNIX como *Microsoft Windows*. Ele oferece uma *interface* de alto nível que permite implementar em poucas linhas uma *interface* básica para uma aplicação[LUN 99].

Um *widget* é um elemento de *interface* gráfica, como botões, *labels*, menus, caixas de texto, etc. Janelas também são *widgets*.. Uma implementação básica de *interface* construída pelo TK necessita que seja criada uma *widget* principal, ou seja, um elemento de *interface* gráfica que funciona como janela principal da aplicação que pode possuir botões, menus, e outros elementos gráficos proporcionados pelo *window manager*.

```
Aplicação básica do TK:
from Tkinter import *
root = TK()
root.mainloop()
```

Figura 10: Aplicação Básica TK

O módulo *Tkinter* (*TK interface*) é a *interface* padrão do *Python* para o *toolkit TK*[LUN 99]. O *Tkinter* possui todas as classes, funções e outras coisas necessárias para trabalhar com o *toolkit TK*.

Como pode ser visto na aplicação básica TK na fig. 10, a lógica da aplicação está na última linha `root.mainloop()`. Isto significa que a aplicação fica rodando permanentemente até que um evento ocorra. A cada evento ocorrido o TK responde de acordo com a função determinada pelo evento. Portanto os jogadores interagem com o modelo através de eventos enviados através da *interface*, esta comunicação é totalmente transparente à aplicação. Desta forma o TK permite que os jogadores vejam o estado do jogo através de uma janela gráfica, que mostra o tabuleiro em seu estado atual e as mensagens necessárias para que haja a sincronização.

5.4.1 Implementando o Tabuleiro do Jogo

No início do desenvolvimento da aplicação exemplo, a linguagem usada para implementar o jogo foi a linguagem C. Após a escolha da arquitetura MVC para a construção do jogo, foi adotado o uso do *Python* para desenvolver objetos que representam modelo, vista e controle.

Inicialmente o jogo foi desenvolvido sem o uso do *toolkit*. Isto dificultava muito a interação do usuário que ficava limitado a ações preestabelecidas pelo modelo. Além disso a visualização do estado do jogo se torna mais complicada de ser desenvolvida, porque a *interface* deve representar todas as alterações feitas no modelo.

Na primeira implementação não existia um tabuleiro colaborativo onde o jogador pudesse jogar através de um clique na casa do tabuleiro. Apenas um texto era mostrado, indicando quem deveria jogar.

Uma forma de implementar um tabuleiro que oferece maior liberdade de interação, foi a utilização do TK. Foram implementados botões que respondem a eventos produzidos pelos jogadores. A cada clique do jogador no tabuleiro acontece uma resposta ao evento acionado pelo botão.

Quando um botão é clicado por um dos jogadores o *toolkit* é responsável por identificar na *interface* qual o jogador está tentando jogar e qual a posição que o jogador escolheu.

O seguinte trecho de código, desenha um botão b11 que chama o método jogada11 a cada vez que é clicado pelo jogador.

```
Self.b11= Button(frame,width=2, height=1, text="  ", command= self.jogada11)
self.b11.pack(side=LEFT)
```

Figura 11: Desenha um botão.

O método jogada11 é encarregado de desenvolver alguma ação no modelo a cada vez que o botão b11 é clicado. Nesta aplicação a comunicação entre o modelo e a vista/controle se dá através da troca de mensagens. Estas mensagens são os métodos usados para realizar alguma tarefa. Por exemplo, quando a vista/controle deseja fazer alguma alteração no modelo, ela envia uma mensagem, que é um método pertencente ao modelo.

Com a implementação dos botões do tabuleiro resultou na seguinte *interface*:

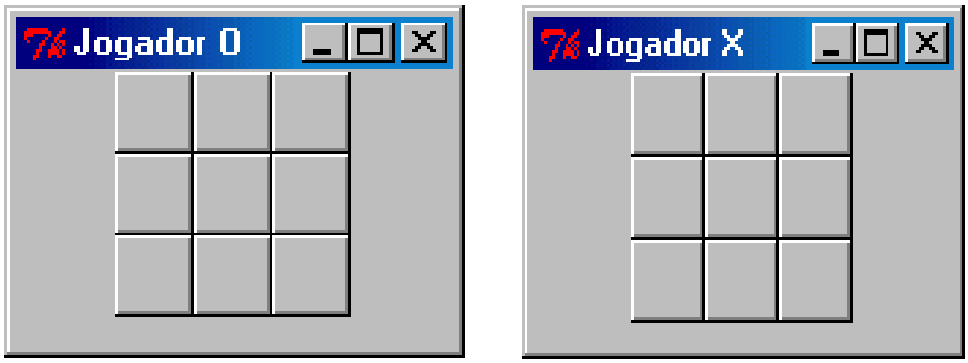

Figura 12: Tabuleiros do Jogo

5.4.2 Comunicação entre Jogadores

A forma utilizada nesta aplicação, para implementar a comunicação entre dois jogadores que interagem no mesmo modelo garantindo a sincronização entre eles é a Troca de Mensagens.

O mecanismo de troca de mensagem garante que cada jogador só irá jogar quando chegar a sua vez, e que dois jogadores não podem jogar ao mesmo tempo. Um aspecto fundamental para que o jogo funcione corretamente é a sincronização, pois existem momentos no jogo em que um jogador deve ficar bloqueado enquanto o outro jogador ainda não jogou. Para que esta sincronização aconteça quando um jogador joga ele manda um evento para o modelo, o qual envia uma mensagem de resposta a este

jogador para que ele saiba qual é o estado atual do jogo, enquanto isto o outro jogador fica aguardando bloqueado até que ele possa jogar.

Quando o jogo inicia os dois jogadores estão aptos a jogar, isto significa que nenhum jogador está bloqueado. Qualquer jogador que tentar jogar no tabuleiro vazio receberá uma mensagem de jogada válida e o estado do modelo mudará. O jogador que jogar primeiro fica bloqueado até que o outro jogador faça uma jogada válida.

Na parte inferior do tabuleiro uma mensagem de texto é mostrada ao jogador para que ele saiba quais os passos a seguir no decorrer do jogo. Quando o jogo inicia os tabuleiros aparecem aos jogadores da seguinte maneira.

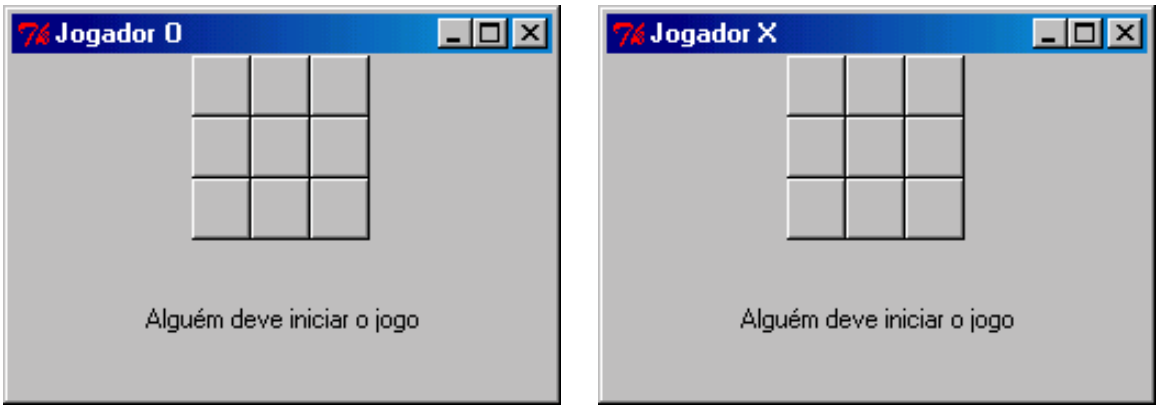

Figura 13: Aparência do tabuleiro quando o jogo inicia

O primeiro que tentar fazer uma jogada ocupará uma casa escolhida no tabuleiro. O estado inicial do modelo era número de jogadas igual a zero, tabuleiro vazio, último que jogou também igual a vazio, nenhum jogador bloqueado. A partir da primeira jogada o estado do jogo é alterado e o último que jogou passa a ficar bloqueado.

A fig. 14 mostra uma jogada exemplo. Nesta jogada o jogador "O" clicou na casa central do tabuleiro que estava vazia e assim um evento associado a esta casa manda para o modelo a informação de quem está jogando e a posição que escolheu, então uma mensagem é enviada, ou seja, um método é executado. Este método testa se a jogada é válida e se o jogador está bloqueado, como a resposta é positiva a jogada é realizada, isto é, ocorre uma alteração no modelo. A partir deste momento o jogador "O" está bloqueado até que o jogador " X" faça uma jogada válida.

Como o jogador "O" está bloqueado a mensagem "Jogador X deve jogar " é enviada para que os jogadores saibam de quem é a vez de jogar.

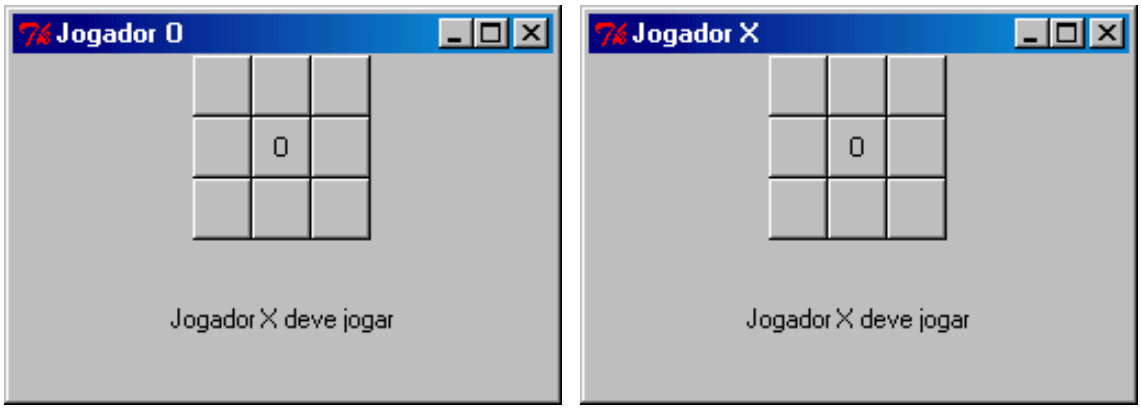

Figura 14: Uma jogada exemplo

Suponha que o jogador "X" faça alguma jogada.

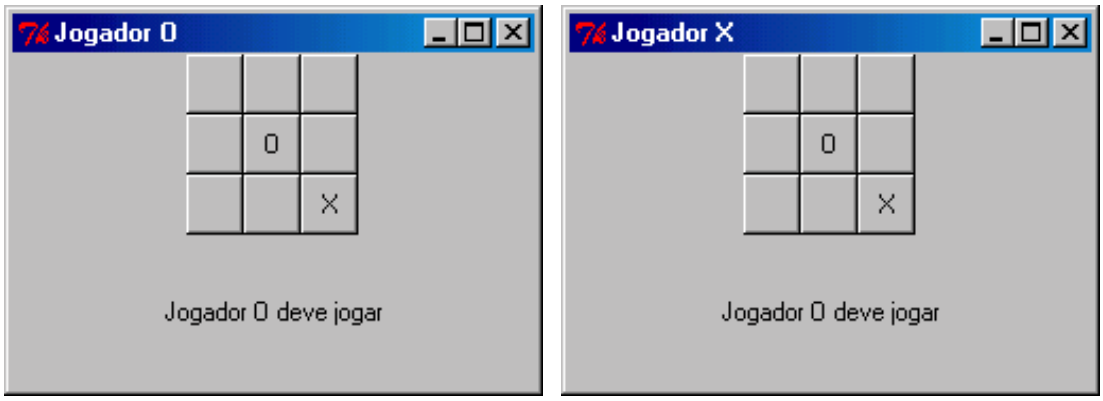

Figura 15: Mensagem que avisa quem deve jogar

A seguinte jogada feita pelo jogador "X" obteve o mesmo efeito da jogada anterior, passando então o jogador "X" para o estado bloqueado e o jogador "O" para o estado desbloqueado, produzindo a mensagem " Jogador O deve jogar".

5.4.3 Quando uma Jogada é Inválida

Existem diferentes tipos de jogadas inválidas. Quando um jogador bloqueado tenta jogar, a jogada é inválida, quando um jogador tenta jogar numa casa já ocupada, a jogada também é inválida. A cada jogada inválida uma mensagem de erro é mostrada para que o jogador saiba porque a jogada é inválida.

Supondo que após a primeira jogada feita no exemplo anterior pelo jogador "O", o jogador "X" tente jogar na casa já ocupada, o evento produzido pelo clique na determinada posição do tabuleiro fornece para o modelo qual o jogador que está jogando e a casa que ele escolheu. O modelo testa se a posição é válida e verifica que a posição já está ocupada. Então a jogada é inválida.

A seguir é mostrada a mensagem:

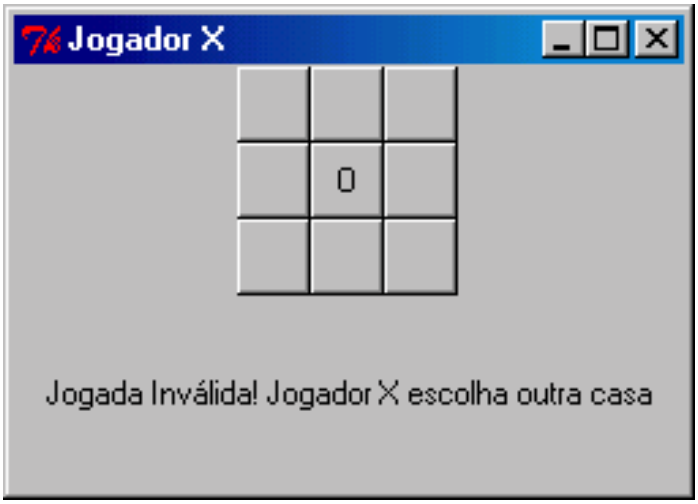

Figura 16: Jogada inválida: jogador tenta jogar em posição ocupada.

A mensagem de erro é mostrada apenas para o jogador que fez a jogada inválida para que ele saiba que a jogada é inválida e possa tentar fazer uma nova jogada. Enquanto ele não fizer uma jogada válida a mensagem de erro será mostrada e o estado do jogo continua inalterado.

Para o outro jogador a mensagem continua a mesma, pois ele apenas deve saber quem é próximo a jogar. Então o tabuleiro do jogador "O" fica conforme fig. 17.

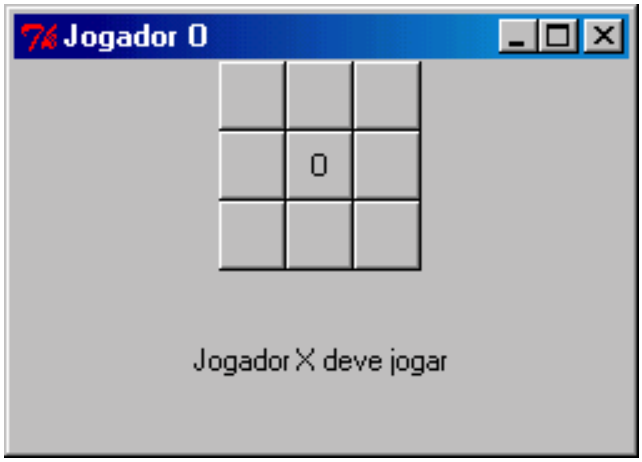

Figura 17: Tabuleiro do usuário que está aguardando que o outro jogue

Supondo que agora após a primeira jogada feita no exemplo anterior pelo jogador "O", o mesmo jogador "O" tenta fazer uma nova jogada. O modelo testa se a posição é válida e se o jogador está bloqueado e verifica que o jogador "O" está bloqueado e que a vez de jogar é do jogador "X". Então a jogada é inválida. Assim uma nova mensagem de erro é mostrada para que o jogador saiba que não é a sua vez de jogar. Quando esta jogada é feita o tabuleiro do jogador "O" mostra o seguinte estado do jogo.

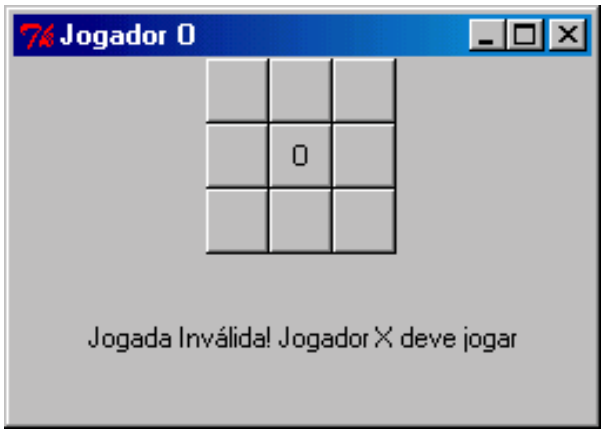

Figura 18: Jogada Inválida: O jogador bloqueado tenta jogar

Como na jogada inválida anterior, a mensagem de erro é mostrada apenas no tabuleiro do jogador que fez a jogada inválida, assim o tabuleiro do jogador "X" continua mostrando quem é o próximo jogador a jogar.

Figura 19: Jogador recebe mensagem que indica que é sua vez de jogar

Um aspecto importante considerando a classificação desta aplicação na área CSCW, é a forma de comunicação entre os usuários e o modelo. A primeira aplicação que foi desenvolvida em C funcionava como trabalho síncrono e comunicação síncrona, pois a *interface* gráfica com o usuário era textual. Como mostrado a seguir:

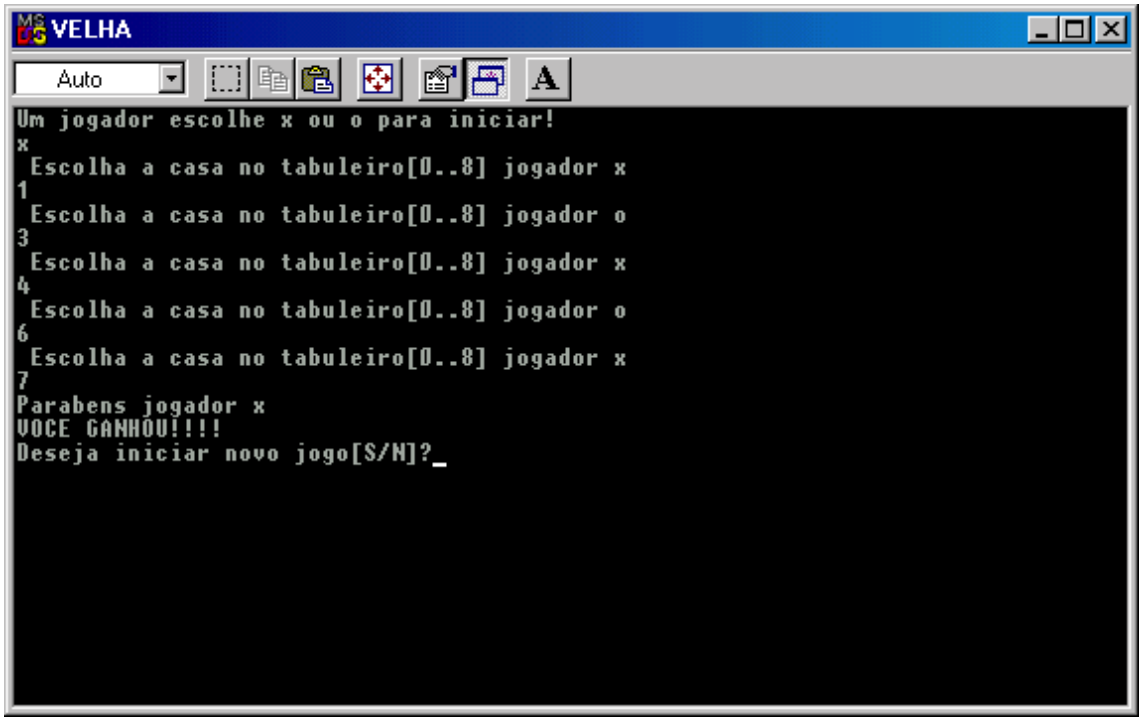

Figura 20: *Interface* da aplicação em C

A seguir está o Código em C da função que define quem joga:

```
/*----------------function quem joga-------------------*/
char quem_joga(char quem)

      {
           char x;
           if (quem == '0')
            {
                do {
                  printf("Escolha um jogador (x ou o) para iniciar: ");
                  x = getchar();
                } while (x != 'o' && x != 'x');
                return(x);
            }
           if (quem == 'o')
            {
            return 'x';
            }
           else
            {
            return 'o';
            }

      }
```

Figura 21: Código em linguagem C da Função que define quem joga.

Nesta aplicação as jogadas são puramente síncronas apenas um jogador previamente estabelecido conseguirá jogar. Além disso, o primeiro jogador deve ser escolhido digitando-se um caracter. Deve existir um tratamento para este caracter que reconheça se o caracter é válido e qual é. Isto permite que um jogador digite qualquer caracter inválido. No caso da aplicação atual uma jogada só pode ser feita através do clique no botão.

O código da função que define quem joga, mostrado anteriormente define o primeiro jogador e quem é o próximo jogador a jogar.

Como pode ser comparado com a jogada mostrada anteriormente, o uso do *toolkit* estabelece uma maneira mais fácil de identificar que jogador está jogando porque o próprio *toolkit* se encarrega de tratar os eventos dos usuários. Além disso embora o trabalho seja síncrono, na aplicação usando o *toolkit* a comunicação passa ser assíncrona, pois qualquer usuário pode tentar jogar. O modelo é quem define se a jogada está correta. Quando um jogador faz uma jogada inválida apenas recebe uma mensagem. No caso da primeira implementação isto não é possível pois os jogadores se comunicam de maneira sincronizada, ou seja, numa ordem preestabelecida pelo modelo.

5.4.4 Quando há um Ganhador

Quando os jogadores fazem suas jogadas seqüencialmente e algum deles completam uma linha diagonal, horizontal ou vertical no tabuleiro, este jogador é o ganhador. Uma mensagem de felicitação ao ganhador é mostrada.

Existem oito possibilidades de cada jogador completar uma vitória, a seguir é apresentada uma jogada em que o jogador "O" foi o vencedor:

Quando o jogador "O" completa a linha horizontal superior recebe uma mensagem " Parabéns jogador O! VOCÊ GANHOU!!!!

A fig. 22 mostra a *interface* do jogo quando um jogador ganha.

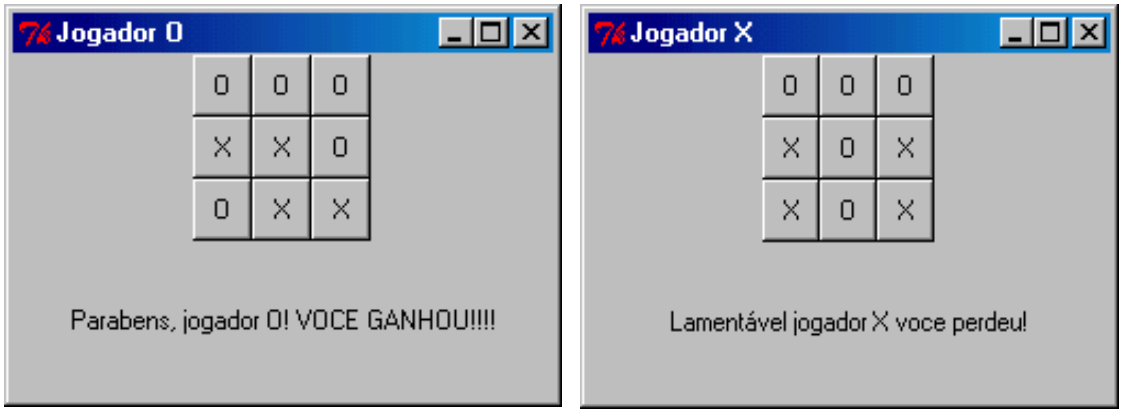

Figura 22: Estado do tabuleiro quando o jogador ganha

5.4.5 Quando não há Ganhador

Quando os jogadores fazem suas jogadas válidas e vão mudando o estado do jogo a cada jogada e finalmente o tabuleiro fica todo completo e nenhum jogador ganhou, uma mensagem de fim de jogo é mostrada. A mensagem " Não houve ganhador" significa que o jogo acabou e nenhum jogador ganhou.

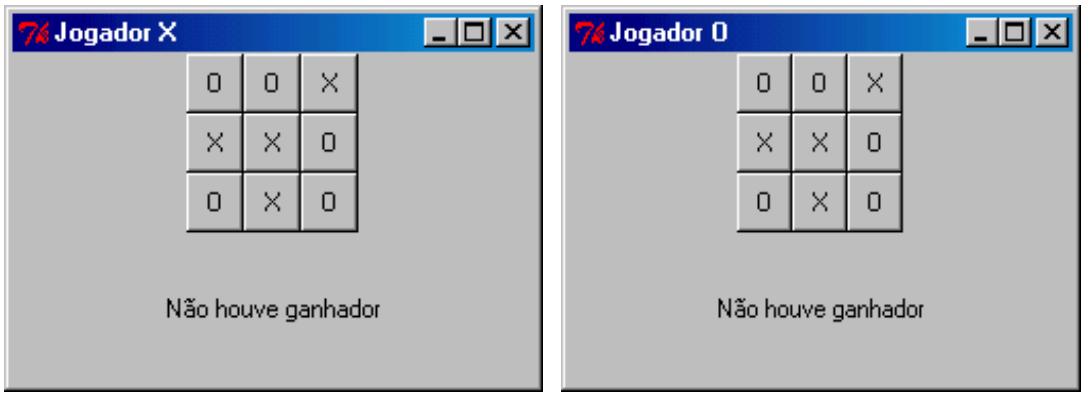

Figura 23: Estado do tabuleiro quando não há jogador

5.4.6 Como Iniciar um Novo Jogo ou Terminar um Jogo

Além dos eventos produzidos pelos jogadores através das jogadas no tabuleiro, existem outras formas de interagir no jogo. O jogador pode iniciar um novo jogo quando desejar e também acabar um jogo. Existe uma outra janela que mostra dois botões adicionais. Um botão permite que um novo jogo seja iniciado e o outro permite que a aplicação seja encerrada.

Para que um novo jogo seja iniciado é preciso que o estado do jogo volte a ser o estado inicial, onde nenhum jogador estará bloqueado e o tabuleiro deve ficar todo vazio. Então quando um novo jogo é iniciado as vistas, ou seja, os tabuleiros dos jogadores apareceram vazios e qualquer um dos jogadores poderá jogar.

A qualquer momento um jogador pode decidir iniciar um novo jogo, mas existe também o caso em que o jogo acaba ou com uma vitória de algum jogador, ou sem um ganhador, mas a mensagem "Inicie um novo jogo " será mostrada para que um novo jogo possa começar.

A seguir é mostrada toda a *interface* da aplicação Jogo da Velha.

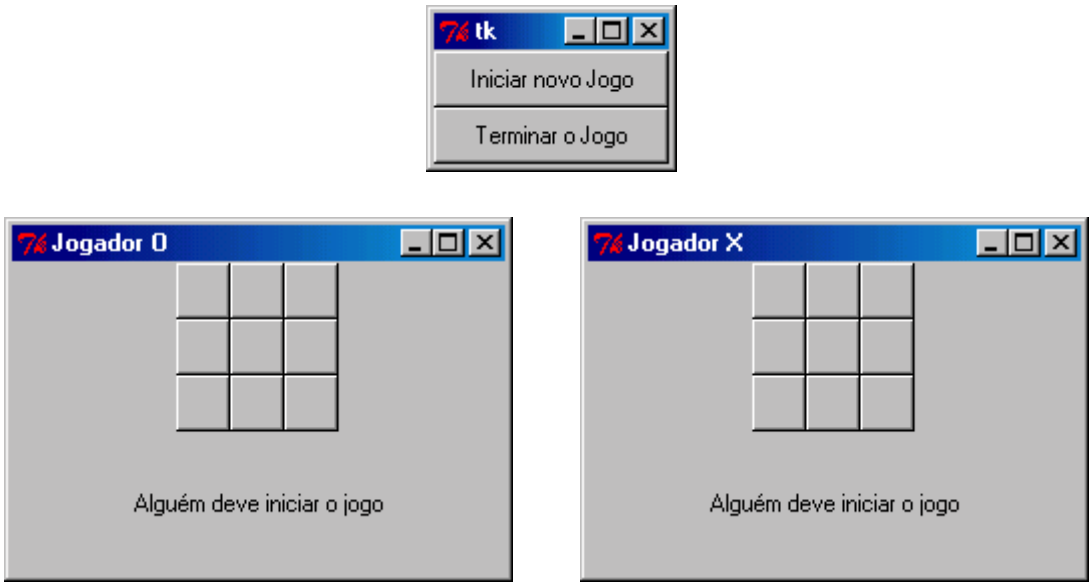

Figura 24: *Interface* da aplicação jogo da velha

Na parte superior estão os botões que podem iniciar um novo jogo ou terminar a aplicação.

Quando um jogo chegar ao fim a seguinte mensagem aparecerá para os jogadores para que eles iniciem um novo jogo: " Inicie um novo jogo".

Figura 25: Estado do tabuleiro quando o usuário inicia um novo jogo

5.4.7 Aplicação na Forma Distribuída

A aplicação foi dividida em modelo e vista/controle para que o modelo fosse responsável pelo estado do jogo e a vista/controle tratasse dos eventos do usuário e da visualização do estado. Quando um usuário interage na aplicação o modelo recebe uma mensagem da vista/controle para tratar a tarefa solicitada, quando o modelo possui algum resultado, manda uma informação para a vista /controle para que seja feita a atualização da visão do modelo. Até o momento a aplicação não permite interação entre usuários localizados em máquinas diferentes. Apenas há uma divisão dos componentes do programa de acordo com suas tarefas. O próximo passo seria a implementação do jogo usando uma arquitetura que permitisse uma comunicação distribuída.

A arquitetura cliente/servidor é uma arquitetura que fornece uma maneira mais fácil de coordenar as ações dos jogadores. O controle de todas as alterações no modelo fica a cargo do servidor, e os jogadores são os clientes que comunicam-se com o modelo através do servidor. Para que o jogo seja distribuído, de maneira que o modelo localize-se no servidor e este modelo seja compartilhado por dois clientes localizados em máquinas diferentes, é preciso utilizar um sistema que suporte esta aplicação. Portanto para implementar esta distribuição da aplicação posteriormente, o sistema X *Window* é o mais adequado a estas exigências, pois permite que a aplicação crie uma janela (tabuleiro) no *display* de cada jogador, conforme mostrado anteriormente na fig. 8.

A aplicação é dividida em duas partes. A parte da aplicação vista/controle é o cliente e o modelo é o servidor. Quando esta aplicação for executada no sistema X *window*, ela será o cliente que fornecerá as tarefas aos usuários através dos *displays* nos servidores. Portanto, esta situação é interessante considerando o conceito de cliente/servidor, pois os componentes da aplicação se dividem em vários clientes que se comunicam com um servidor, o modelo. Caracterizando assim um sistema cliente servidor. Mas a aplicação, quando distribuída em um sistema que permite que esta seja visualizada em vários *displays*, o sistema *X Window*, se comporta como um cliente que fornece serviços para vários servidores através dos seus *displays*. Portanto, uma variação nos componentes da arquitetura cliente/servidor é aceitável e funcional dentro do conceito dessa arquitetura, nada impede que sejam vários servidores e um cliente, ou vários clientes um servidor.

6. Conclusão

O trabalho abordou vários conceitos importantes na área de CSCW, contribuindo para a identificação dos aspectos cruciais na implementação de aplicações colaborativas.

A classificação de CSCW é importante para que se possa projetar uma aplicação de acordo com suas características de espaço/tempo. A aplicação exemplo apresentou um aspecto muito interessante a respeito da relação síncrono x assíncrono. Nesta aplicação o trabalho é tratado de maneira síncrona, cada jogador só pode jogar após o outro ter jogado. Mas ao mesmo tempo que existe este sincronismo nas jogadas, as interações feitas pelos usuários são totalmente assíncronas. As interações não têm ordem preestabelecidas para ocorrerem, um jogador pode tentar jogar a qualquer momento, tendo assim uma maior liberdade de interação.

De acordo com a arquitetura MVC, utilizada na implementação, ocorreu uma fusão dos componentes vista e controle. Pois o mesmo objeto que implementa a vista é responsável por controlar as entradas do usuário. O mesmo objeto que mostra os dados do modelo na *interface* controla as interações dos usuários.

A arquitetura cliente/servidor possui a vantagem de facilitar o controle dos dados compartilhados entre os clientes através da estrutura centralizada no servidor. Nesta arquitetura o modelo se comporta como um servidor e vista/controle como clientes. As duas vistas interagem no mesmo modelo de forma coordenada pelo servidor.

O *X window* é um sistema cliente /servidor adequado para o desenvolvimento de aplicações colaborativas por permitir que uma aplicação possua múltiplos *displays*. Vários usuários podem interagir em uma aplicação que roda no servidor através da visualização da aplicação em terminais remotos.

7. Anexo 1

Código da Aplicação Exemplo

```
#File: TwoWindow.py

from Tkinter import*

class App:

    def __init__(self, quem, modelo):
       self.quem = quem
       self.modelo = modelo
       top = Toplevel()
       top.title("Jogador " + quem)
       self.final= 0

        frame = Frame(top)
        self.b11 = Button(frame, width=2, height=1, text="    ", command =
self.jogada11)
        self.b11.pack(side=LEFT)
        self.b12 = Button(frame, width=2, height=1, text="    ", command =
self.jogada12)
        self.b12.pack(side=LEFT)
        self.b13 = Button(frame, width=2, height=1, text="    ", command =
self.jogada13)
        self.b13.pack(side=LEFT)
        frame.pack()

        frame = Frame(top)
        self.b21 = Button(frame, width=2, height=1, text="    ", command =
self.jogada21)
        self.b21.pack(side=LEFT)
        self.b22 = Button(frame, width=2, height=1, text="    ", command =
self.jogada22)
        self.b22.pack(side=LEFT)
        self.b23 = Button(frame, width=2, height=1, text="    ", command =
self.jogada23)
        self.b23.pack(side=LEFT)
        frame.pack()

        frame = Frame(top)
        self.b31 = Button(frame, width=2, height=1, text="    ", command =
self.jogada31)
        self.b31.pack(side=LEFT)
        self.b32 = Button(frame, width=2, height=1, text="    ", command =
self.jogada32)
        self.b32.pack(side=LEFT)
        self.b33 = Button(frame, width=2, height=1, text="    ", command =
self.jogada33)
        self.b33.pack(side=LEFT)
```

```
    frame.pack()

    frame = Frame(top)
    self.w = Label(frame, width=40, height=5, text="Alguém deve iniciar o jogo")
   self.w.pack()
   frame.pack()

def jogada11(self):
   self.inicia_jogada(0)

def jogada12(self):
   self.inicia_jogada(1)

def jogada13(self):
   self.inicia_jogada(2)

def jogada21(self):
   self.inicia_jogada(3)

def jogada22(self):
   self.inicia_jogada(4)

def jogada23(self):
   self.inicia_jogada(5)

def jogada31(self):
   self.inicia_jogada(6)

def jogada32(self):
   self.inicia_jogada(7)

def jogada33(self):
   self.inicia_jogada(8)

def inicia_jogada(self, onde):
    if self.final == 0:
       joga= self.modelo.jogada(self.quem, onde)
       if joga == 1:
          fim= self.modelo.testa_final()
          self.mensagem_fim(fim, self.quem)
       else:
          self.mensagem(self.quem, joga)
   else:
       appX.w["text"] = "Inicie um novo jogo "
       appO.w["text"] = "Inicie um novo jogo"

def mensagem(self, quem, joga):
    if joga == 2:
       self.w["text"] = "Jogada Inválida! Jogador "+ quem +" escolha outra casa"
   else:
       if quem == 'X':
          self.w["text"] = "Jogada Inválida! Jogador O deve jogar"
       else:
          self.w["text"] = "Jogada Inválida! Jogador X deve jogar"

def mensagem_fim(self, fim, quem):
    if fim == 2:
       if quem == 'X':
          appX.w["text"] = 'Parabens, jogador '+ quem +'! VOCE GANHOU!!!!'
          appO.w["text"] = 'Lamentável jogador O voce perdeu!'
       else:
```

```
            appO.w["text"] = 'Parabens, jogador '+ quem +'! VOCE GANHOU!!!!'
            appX.w["text"] = 'Lamentável jogador X voce perdeu!'
        appX.final= 1
        appO.final= 1
    elif fim == 1:
        appX.w["text"] = 'Não houve ganhador'
        appO.w["text"] = 'Não houve ganhador'
        appX.final= 1
        appO.final= 1
    else:
        if quem == 'X':
            appX.w["text"] = "Jogador O deve jogar"
            appO.w["text"] = "Jogador O deve jogar"
        else:
            appX.w["text"] = "Jogador X deve jogar"
            appO.w["text"] = "Jogador X deve jogar"

    def mostra(self):

        self.b11["text"] = self.modelo.tab[0]
        self.b12["text"] = self.modelo.tab[1]
        self.b13["text"] = self.modelo.tab[2]

        self.b21["text"] = self.modelo.tab[3]
        self.b22["text"] = self.modelo.tab[4]
        self.b23["text"] = self.modelo.tab[5]

        self.b31["text"] = self.modelo.tab[6]
        self.b32["text"] = self.modelo.tab[7]
        self.b33["text"] = self.modelo.tab[8]


class tabuleiro:
        tab = [' ',' ',' ',' ',' ',' ',' ',' ',' ']
        jogadas = 0
        ultimo = ' '
        notifica = []
        def limpa(self):
            self.tab= [' ',' ',' ',' ',' ',' ',' ',' ',' ']
            self.jogadas = 0
            self.ultimo = ' '

        def jogada(self, quem, onde):
            if (quem != self.ultimo):
                if self.tab[onde] == ' ':
                    self.tab[onde]= quem
                    self.jogadas= self.jogadas + 1
                    self.ultimo = quem
                    i=0
                    while i<=1 :
                      self.notifica[i].mostra()
                      i=i+1
                    return 1
                else:
                    return 2
            else:
                return 3

        def testa_final(self):
            if (self.jogadas >= 5) and \
               (((self.tab[0] != ' ') and
```

```
                    (self.tab[0] == self.tab[1]) and
                    (self.tab[0] == self.tab[2])) or
                  ((self.tab[3] != ' ') and
                    (self.tab[3] == self.tab[4]) and
                    (self.tab[3] == self.tab[5])) or
                  ((self.tab[6] != ' ') and
                    (self.tab[6] == self.tab[7]) and
                    (self.tab[6] == self.tab[8])) or
                  ((self.tab[0] != ' ') and
                    (self.tab[0] == self.tab[3]) and
                    (self.tab[0] == self.tab[6])) or
                  ((self.tab[1] != ' ') and
                    (self.tab[1] == self.tab[4]) and
                    (self.tab[1] == self.tab[7])) or
                  ((self.tab[2] != ' ') and
                    (self.tab[2] == self.tab[5]) and
                    (self.tab[2] == self.tab[8])) or
                  ((self.tab[0] != ' ') and
                    (self.tab[0] == self.tab[4]) and
                    (self.tab[0] == self.tab[8])) or
                  ((self.tab[2] != ' ') and
                    (self.tab[2] == self.tab[4]) and
                    (self.tab[2] == self.tab[6]))):
                            return 2
            elif self.jogadas == 9:
                            return 1
            else:
                    return 0

        def novo_jogo(self):
            self.limpa()
            i=0
            while i<=1 :
               self.notifica[i].mostra()
               self.notifica[i].w["text"] = "Alguém deve iniciar o jogo"
               self.notifica[i].final= 0
               i=i+1

foo = Tk()

modelo = tabuleiro()

appX = App("X", modelo)
appO = App("O", modelo)

tabuleiro.notifica = [appX,appO]

frame= Frame(foo)
B1 = Button(frame, width=17, text="Iniciar novo Jogo", command = modelo.novo_jogo)
B1.pack()
B2 = Button(frame, width=17, text="Terminar o Jogo", command = frame.quit)
B2.pack()
frame.pack()


appX.mostra()
appO.mostra()

foo.mainloop()
```

8. Referências Bibliográficas

[AND 91] ANDREWS, Gregory R. **Concurrent Programming** . University of Arizona : Ed. Addson- Wesley Publishing Company, 1991.

[ARA 95] ARAUJO, R. **CSCW, Groupware & Internet**. Disponível por WWW em http://www.cos.ufrj.br/~renata/cscw/ (09/08/2000).

[BEN 94] BENTLEY, Richard; RODDEN, Tom; SAWYER, Pete; SOMMERVILLLE, Ian. Architectural Support for Cooperative Multiuser Interfaces. **Revista Computer,** v.27, n.5, p. 37-46, maio.1994.

[BOS93] BOS, Gÿsbert. Rapid user interface development with the scripting language Gist. Groningen, Universiteitsdrukkerÿ, 1993. (Tese de Doutorado). Disponível via WWW em http://www.let.rug.nl/~bert/dissertation.ps.gz

[DES87] DESANCTIS, G. & GALLUPE, B. A Foundation of the Study of Group Decision Support Systems. Management Science, v.33, n.5. May 1987. p.589-609.

[GRU 94] GRUDIN, Jonathan. Histoty and Focus. **Revista Computer,** v.27, n.5, p. 19-26, maio.1994.

[KOJ 2000] KOJIMA, Alfredo K.; ANDRADE, Paulo C. P.; SANTOS, Carlos A. M. Desenvolvimento de Aplicações Gráficas para o X Window System. In Mini-curso apresentado no *SOFTWARE* LIVRE 2000; Porto Alegre, 5 de maio de 2000.

[LIA 94] LIANG, Ting-Pen; LAI, Hsiangchu; CHEN, Nian-Shing; WEI, Hungshiung; CHEN, Meng Chang. When Client/Server Isn't Enough: Coordinating Multiple Distributed Tasks. **Revista Computer,** v.27, n.5, p. 73-79, maio.1994.

[LUN 99] LUNDH, Fredrik. **An Introduction to Tkinter**. 1999. 166 p.

[MAN 98] MANSSOUR, I. H.; FREITAS, C. M. D. S. An Introduction to Collaborative Visualization. In: SEMANA ACADÊMICA DO CPGCC, 3., 1998, Porto Alegre, RS. **Proceedings**. Porto Alegre: CPGCC da UFRGS, 1998.

[MYE 93] MYERS, Brad A. **Why are Human-Computer Interfaces Difficult to Design and Implement?** Computer Science Department Carnegie Mellon University Pittsburgh. July, 1993.

[NIT 2000] NITZKE **,** Julio Alberto; CARNEIRO, Mara Lúcia Fernandes; GELLER, Marlise; SANTAROSA, Lucila Costi. Criação de Ambientes de Aprendizagem Colaborativa**.** In: VII CONGRESO INTERNACIONAL DE INFORMÁTICA EN LA EDUCACIÓN; Havana, Cuba, maio 2000.

[NUN91] Nunamaker, J.F. Electronic meeting systems to support group work, Communications of the ACM, 34 (7), Jul 1991, pp. 40-61.

[OUS 94] OUSTERHOUT, John K.. **Tcl and the Tk toolkit**. Reading, Addison-Wesley, 1994. 458p.

[PAL 94] PALMER, James D; FIELDS, Ann. Computer-Supported Cooperative Work. **Revista Computer**, v.27, n.5, p.15-17, maio.1994.

[RAP 2000] RAPOSO, Alberto B.; CRUZ, Adailton J. A.; BICHO,Alessandro L.; KOJIMA Alfredo K.; SANTOS, Carlos A. M.; SILVA, Isla C: F.; MAGALHÃES, Léo P.; ANDRADE, Paulo C. P. **Ferramentas de Programação Livres para Computação Gráfica e Animação por Computador.** Setembro 2000.

[REI 94] REINHARD Walter; SCHWEITZER, Jean; VÖLKSEN, Gerd; WEBER, Michael. CSCW Tools: Concepts and Architectures. **Revista Computer,** v.27, n.5, p.28-36, maio.1994.

[ROM 2000] ROMANI, L. a F.; ROCHA, H.V.; SILVA, C.G. Ambientes para Educação à distância baseados na Web. Onde estão as pessoas? In: PIMENTA, Marcelo; VIEIRA, Renata(Ed.). **Workshop sobre Fatores Humanos em Sistemas Computacionais.** 3.Gramado. Porto Alegre: Instituto de Informática da UFRGS, 2000.p.12-21.

[SIP 98] SIPERT, Lisiane Volpi; MARTINS, Tecnologia de Objetos Edição 75/98. **Design Patterns-Composite**. Disponível por WWW em http://www.pr.gov.br/celepar/celepar/batebyte/bb75/design.htm (06/09/2000).

[SUN 96] SUNDSTED, Todd**. An introduction to the Observer interface and Observable class using the Model/View/Controller architecture as a guide**. Disponível por WWW em http://www.javaworld.com/javaworld/jw-10-1996/jw-10-howto.html.

[TEI 99] TEIXEIRA JR.; José H.;SAUVÉ, Jacques P.; MOURA, José A.B.; TEIXEIRA, Suzana de Q. R.. **Redes de Computadores - Serviços, administração e segurança.** São Paulo, 1999.

[VAN 98] VAN ROSSUM, Guido. **Python tutorial**. Amsterdam, Stichting Mathematitisch Centrum, 1998. 60p. **Python reference manual**. Amsterdam, Stichting Mathematitisch Centrum, 1998. 60p.

Estudos das Técnicas Usadas na Construção de Aplicações de Suporte ao Trabalho Colaborativo

por

Fabrícia Carneiro Roos

Monografia apresentada aos Senhores:

_____

Prof°. Gil Carlos Medeiros

_____

Profª. Eliane da Silva Alcoforado Diniz

Vista e permitida a impressão.

Pelotas, ___/___/___.

_____

Analista de Sistemas, Carlos Augusto Moreira dos Santos

Orientador.