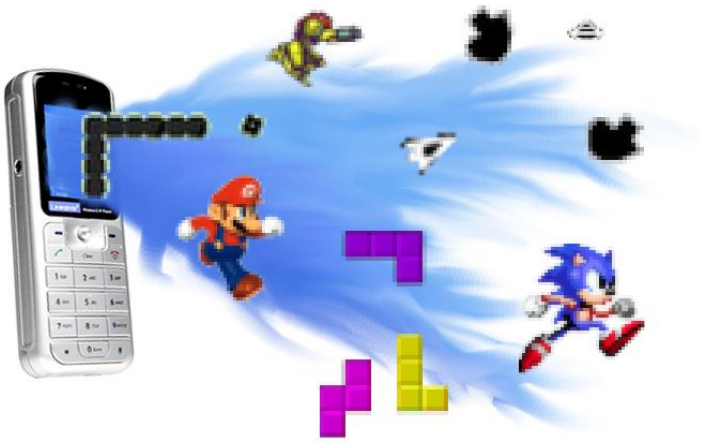
UNIVERSIDADE FEDERAL DE SANTA CATARINA

DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Danilo Machado Delponte

Trabalho de Conclusão de Curso

Desenvolvimento de jogos multi-jogadores na plataforma J2ME



Florianópolis, 13 de julho de 2010

UNIVERSIDADE FEDERAL DE SANTA CATARINA

DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Danilo Machado Delponte

Trabalho de Conclusão de Curso

Desenvolvimento de jogos multi-jogadores na plataforma J2ME

Trabalho de conclusão de curso apresentado como parte dos requisitos para obtenção do grau de Bacharel em Sistemas de Informação.

Florianópolis, 13 de julho de 2010

Danilo Machado Delponte

Desenvolvimento de jogos multi-jogadores na plataforma J2ME

Trabalho de conclusão de curso apresentado como parte dos requisitos para obtenção do grau de Bacharel em Sistemas de Informação.

Professor orientador:

Professor Doutor Frank Siqueira
Universidade Federal de Santa Catarina

Banca examinadora:

Professor Doutor Frank Siqueira
Universidade Federal de Santa Catarina

Professor Doutor Ricardo Pereira e Silva
Universidade Federal de Santa Catarina

Professor Doutor Mario Antonio Ribeiro Dantas
Universidade Federal de Santa Catarina

Florianópolis, 13 de julho de 2010

Resumo

Há muito tempo que os jogos eletrônicos deixaram de ser brinquedo de criança para se tornar uma grande indústria de entretenimento, e a cada ano atingem um número maior de usuários e plataformas. Mas jogos em aparelhos celulares, a não ser pelo iPhone da Apple, nunca chegaram a fazer sucesso. Esse trabalho trata de uma abordagem ainda pouco explorada nessa área: jogos casuais multi-jogadores em J2ME através da internet móvel. Mesmo os aparelhos mais simples de hoje possuem recursos suficientes para esse tipo de jogos, são dispositivos populares e de uso diário, o que os torna uma plataforma em potencial. Nesse trabalho será feita uma análise e revisão do contexto atual de desenvolvimento de jogos para celulares, uma análise e projeções para a internet móvel no Brasil, o projeto e implementação de um *framework* para desenvolvimento de jogos com as características apresentadas e também o projeto e implementação de dois jogos como casos de uso.

Palavras-chave: jogos eletrônicos, J2ME, jogos casuais, jogos sociais, jogos multi-jogadores, aparelhos celulares, internet móvel.

Sumário

Listas

Lista de Figuras

Lista de gráficos

Lista de tabelas

Lista de diagramas

Lista de trechos de código

1. Introdução

1.1 Contexto

Os **telefones celulares** se difundiram no mundo tanto quanto os computadores ou a internet. Possuem hoje mais de 4,6 bilhões de usuários e estima-se que esse número chegará a 5,9 bilhões em 2013 (Teleco, 2010) (Infonetics Research, 2009). O Brasil está entre os 5 países com maior teledensidade, com 91,87 celulares para cada 100 habitantes, 175 milhões no país, o que representa mais que o dobro do número de computadores (ABIN, 2009) (Anatel, 2010).

Esse crescimento gera novas oportunidades e leva ao desenvolvimento de novas tecnologias com o objetivo de fomentar os desejos e suprir as necessidades dos consumidores, fornecendo novidades em aparelhos e serviços, cada vez mais completos e sofisticados.

Paralelamente, os **jogos eletrônicos**, que existiam já antes dos celulares e a internet, também ganham mais adeptos a cada geração e são hoje responsáveis por uma das maiores indústrias de entretenimento do mundo. Seja no computador, no celular, na internet, entre jovens, adultos, homens ou mulheres. Uma pesquisa recente mostra que nos EUA 54% dos adultos com 18 anos ou mais jogam videogame, e dos 12 aos 17 anos, 97% jogam (Games Industry, 2009).

Boa parte desses números deve-se aos jogos *online* **multi-jogadores**, que reúnem até milhões de pessoas em um mesmo jogo. O *World of Warcraft*, um dos maiores do gênero multi-jogadores massivo, já possui mais de 11 milhões de usuários (Blizzard, 2008).

Entretanto, os jogos em aparelhos celulares só ganharam força recentemente, com a chegada de dispositivos mais sofisticados como o *iPhone*. Mas a maioria dos aparelhos ainda possui poucos recursos, principalmente no Brasil onde a incorporação de novas tecnologias é lenta e mais cara. Com aparelhos de baixa capacidade, é difícil criar jogos bem elaborados que atinjam um grande número de usuários, mas não é raro jogos simples fazerem sucesso. Como é o caso de clássicos como *Tetris*, que surgiu como um *minigame*, e o *Snake*, o jogo da "cobrinha" famoso em celulares.

Na internet há jogos simples como esses, também chamados de **jogos casuais** ou **sociais**, que são bastante difundidos. Com várias temáticas e um público alvo abrangente, atraem principalmente pelo fácil acesso e jogabilidade simples e rápida, ideal para pessoas com pouco tempo para dedicar a jogos. Assim acabam conquistando tantos usuários quanto os grandes jogos. Um bom exemplo é o *Farmville*, extremamente mais simples que *World of Warcraft*, mas que conta com cerca de 80 milhões de usuários (Cashmore, 2010).

Dentro desse contexto, podemos dizer que estamos presenciando uma nova evolução, a **internet móvel**. Com a disseminação do serviço e a consequente diminuição dos custos, a tendência é a de um aumento no número de acessos via celular (Cisco, 2009) (Teleco, 2010). O *iPhone* lidera o ranking e isso mostra que com uma gama maior de aplicativos e serviços, muito provavelmente esse uso se tornará tão comum quanto verificar e-mails no computador.

1.2 Motivação

Com base nesse contexto, a motivação para esse projeto vem de um espaço ainda pouco explorado no desenvolvimento de jogos para celulares, os jogos casuais *online* multi-jogadores.

Mesmo para aparelhos de baixa e média capacidades, é possível a criação de jogos simples utilizando o acesso à internet via GPRS. Além do grande apelo, esses jogos utilizam poucos recursos, suficientes em dispositivos desse nível, atingindo a maior parte dos usuários.

Apesar de ainda não serem vistos como plataforma de jogos, os aparelhos celulares englobam cada vez mais funcionalidades, incluindo as de entretenimento. São dispositivos populares, pessoais, móveis e de uso diário, o que os torna uma plataforma em potencial para essa abordagem.

1.3 Objetivos

Os objetivos gerais desse trabalho são: apresentar um estudo e análise do cenário e desenvolvimento de jogos *online* multi-jogador para celulares, e um estudo de caso com a implementação de um jogo desse tipo e do sistema de comunicação necessário para este fim.

1.3.1 Objetivos específicos

- Revisão e análise do contexto atual de jogos multi-jogadores e de jogos para celular.
- Revisão e análise da arquitetura de sistemas distribuídos e dos meios de comunicação móvel.
- Revisão e análise das plataformas disponíveis em dispositivos móveis.
- Análise de *frameworks* existentes para o desenvolvimento de jogos para celular.
- Projeto e implementação de um *framework* para o desenvolvimento de jogos com múltiplos jogadores e do sistema distribuído necessário para seu funcionamento.
- Projeto e implementação de dois jogos como caso de uso de utilização do *framework*.

1.4 Organização do trabalho

Logo no próximo capítulo será apresentado um breve histórico sobre celulares, jogos e a internet móvel.

No capítulo três inicia-se a revisão e análise de jogos para aparelhos celulares e multi-jogadores em geral. Será abordada a internet móvel, arquitetura de sistemas distribuídos, servidores para jogos multi-jogador e motores de jogos, incluindo uma análise dos *frameworks* disponíveis.

No capítulo quatro será apresentado um projeto para a implementação de um *framework* para desenvolvimento de jogos para celular, de um servidor para jogos multi-jogadores e de dois jogos que utilizem o sistema implementado, como caso de uso.

No capítulo cinco será apresentada a implementação do projeto, com a descrição da abordagem utilizada, do processo de desenvolvimento e dos resultados.

Ao final, nos capítulos seis e sete, serão apresentadas as conclusões acerca do trabalho realizado e descritas possibilidades de trabalhos futuros.

2. Histórico

2.1 Celulares e a internet

Os aparelhos celulares funcionam como o rádio ou a TV, a partir de uma rede de ondas eletromagnéticas em uma determinada faixa de freqüência. Até meados de 1980, a tecnologia utilizada era o AMPS, com sinal analógico de baixíssima capacidade, alto gasto de energia e nenhuma segurança, e apenas o serviço de voz era oferecido. Para disponibilizar mais recursos e atender um número maior de usuários, tecnologias com sinal digital foram desenvolvidas, como o TDMA, o CDMA e o GSM (Teleco, 2008).

O TDMA (*Time Division Multiple Access*) permite a vários usuários utilizarem o mesmo canal de freqüência, dividindo o sinal em intervalos de tempo. Foi bastante utilizado, mas tornou-se ultrapassado frente ao CDMA e ao GSM.

O CDMA (*Code Division Multiple Access*) divide o sinal codificando os dados de forma que cada receptor só recebe o sinal enviado pelo seu respectivo emissor. Comporta um número de chamadas maior do que as outras tecnologias, mas acabou perdendo espaço para o GSM.

O GSM é uma evolução do TDMA, mais eficaz, seguro e com mais recursos. Tornou-se o padrão mais utilizado na Europa na década de 80, e é ainda a tecnologia mais usada no mundo, com aproximadamente 80% dos usuários. No Brasil, desde 2004 o GSM é o padrão mais aceito, como mostra o gráfico a seguir.

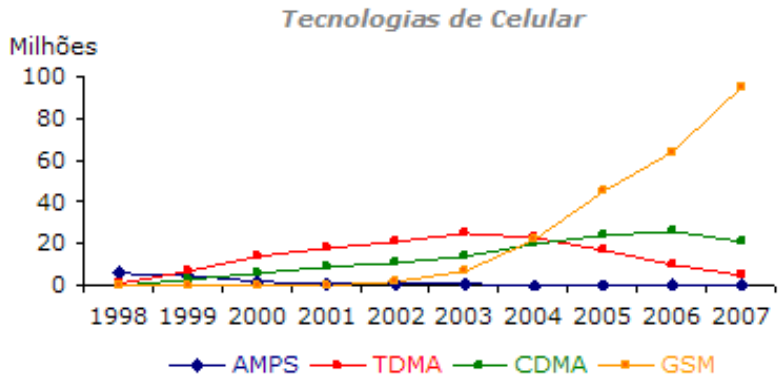


Gráfico 1 - Tecnologias de celular no Brasil
(adaptado de Teleco, 2010)

O serviço de acesso à internet a partir dos telefones celulares só apareceu no final da década de 90, quando várias tecnologias com esse propósito foram criadas. Como exemplo podemos citar o protocolo WAP, padrão para acesso a páginas da internet exclusivas para celular, e o GPRS, protocolo de transferência de dados por pacotes que funciona sobre a rede GSM (Open Mobile Alliance, 2002).

Recentemente chegamos à terceira geração de tecnologias de transmissão de dados, evoluções do GSM e CDMA que permitem conexões consideradas de “banda larga”. A partir de então a internet móvel vem se difundindo e deve tornar-se o padrão em pouco tempo (Huawei/Teleco, 2009).

2.2 Celulares e jogos

Os jogos em aparelhos celulares só começaram a aparecer quando a Nokia lançou o famoso *Snake*. O "jogo da cobrinha" para celulares surgiu em 1997 e, apesar de simples, vendeu aproximadamente 400 milhões de cópias. (Wright, 2008)

No começo, além dos celulares disponibilizarem pouquíssimos recursos, não havia uma plataforma adequada para o desenvolvimento de jogos. Alguns jogos chegaram a ser criados utilizando a tecnologia WAP, como o *Picofun Football*, um gerenciador de times jogado através de páginas e *links* no navegador. Mas o WAP acabou sendo mais utilizado para a distribuição de jogos.

Figuras 1 e 2 – *Nokia Snake* e *Picofun Football*

Em 2000, começaram a aparecer os celulares com tela colorida e a Nokia lançou seu primeiro aparelho com a plataforma Symbian, um sistema potente que mais tarde daria origem ao N-Gage, aposta da Nokia no mercado de jogos em dispositivos móveis.

Em 2002, foram lançados os primeiros aparelhos a suportar a versão para dispositivos móveis da plataforma de Java (J2ME), e no mesmo período a Qualcomm publicou o BREW, para desenvolvimento de aplicativos para celulares em C++. O lançamento dessas plataformas mudou completamente o cenário do desenvolvimento de jogos. O BREW acabou fortemente disseminado em aparelhos CDMA, enquanto o J2ME é encontrado na maioria dos aparelhos GSM. Talvez por esse motivo o J2ME acabou se sobressaindo, apesar do poder do BREW.

Em 2003, ano em que surgiram os primeiros jogos para celulares em 3D, a Nokia lançou o N-Gage, um aparelho poderoso, com capacidade para rodar jogos mais elaborados. Uma aposta que acabou não dando certo, pois o N-Gage, que se aproxima mais de um *videogame* portátil do que um celular, não pôde competir com o PSP da Sony, lançado na mesma época (Melanson, 2006).

Apesar do fracasso do N-Gage, o sistema operacional Symbian foi bastante difundido no mercado de *smartphones*, e incorporado em grande parte dos aparelhos da Nokia. Por ser uma plataforma potente muitos jogos foram desenvolvidos, mas não chegou a se tornar referência.

Figura 3 – *N-Gage* da Nokia

A partir de 2004 muitas empresas investiram pesado em jogos para celular, movimentando muitos milhões de dólares, e grandes empresas começaram a se interessar de fato pelo mercado móvel. A *Electronic Arts*, que já vendia licenças para a produção de jogos de seu portfólio, acabou criando sua própria divisão voltada ao desenvolvimento para celulares.

O período de 2005 a 2007 foi marcado por um grande crescimento no número de jogos e uma grande movimentação de empresas no mercado. As grandes empresas, com a maior fatia do segmento, lamçaram basicamente versões mais enxutas de jogos já disponíveis em outras plataformas, enquanto as pequenas empresas buscavam pequenos nichos de consumidores. Mas até então, aparelhos celulares não tinham desempenhado muita importância no mercado de jogos.

O que fez a diferença foi o iPhone, da Apple, lançado em 2007. Um celular potente, com interface rica, ótimos gráficos e uma tela sensível ao toque. Com ele a Apple conseguiu abocanhar a maior fatia do mercado no segmento de aparelhos mais sofisticados. Um dos motivos, além do próprio aparelho, foi a App Store, a loja virtual de aplicações da Apple, que encurtou muito o espaço entre os desenvolvedores e o público consumidor.

Figuras 4 e 5 – *"Seat Cupra Race" e "Metal Gear Solid"* no iPhone

Em 2008, ainda foi lançada a segunda geração do N-Gage, que deixou de ser um aparelho e tornou-se uma plataforma de software, pré-instalada nos novos celulares da Nokia. Mas também não alcançou as expectativas.

3. Revisão e análise

3.1 Jogos eletrônicos

Os jogos eletrônicos (*videogames*) são programas executados em dispositivos eletrônicos, como computadores ou celulares, que apresentam problemas e desafios com o objetivo de entreter através da interação humano-máquina. Começaram a surgir em experimentos com televisores e projetos acadêmicos em meados do século passado e foram sendo desenvolvidos de acordo com a evolução dos computadores. Logo conquistaram muitos usuários e se tornaram atrativos do ponto de vista comercial. Assim surgiram *fliperamas*, consoles, *minigames*, jogos para computador, etc (Herman, Horwitz, Kent, & Miller).

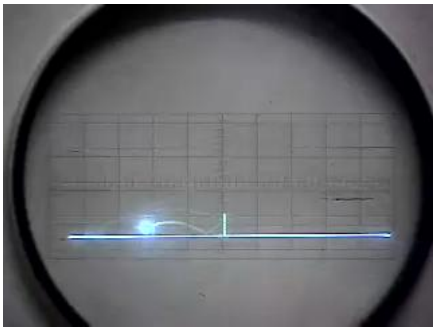


Figura 6 – *Tennis for Two*, criado em 1958 utilizando um osciloscópio e um computador analógico

Durante muito tempo os jogos eletrônicos foram vistos como brinquedos, voltados principalmente para o público masculino jovem, mas com o passar dos anos esse panorama mudou drasticamente, chegando hoje a existir, por exemplo, jogos focados no público de mulheres com mais de quarenta anos (Ingram, 2010).

Com a infinidade de jogos criados e a evolução não só tecnológica, mas também de conteúdo, nas últimas décadas os jogos eletrônicos passaram a ser levados a sério, tornando-se objeto de pesquisa em diversas áreas e uma das principais atividades culturais do século, além de uma indústria bilionária.

Um dos motivos que levaram os jogos eletrônicos a esse patamar são as possibilidades infinitas que o ambiente virtual proporciona, sendo possível criar vários tipos de jogos, de simples *puzzles* a histórias complexas com grandes mundos para exploração.

O sucesso de um jogo está intimamente ligado a algumas características básicas, como os desafios que apresenta e a sua jogabilidade. Um jogo com gráficos de última geração não significa um bom jogo, o objetivo de entreter o jogador pode ser atingido de várias formas.

O tipo do jogo e a elaboração e balanceamento desses aspectos fazem parte do conceito de ***game design***, a concepção do jogo, e todos são desenvolvidos de acordo com o público e plataforma que se deseja atingir. É o que faz a diferença entre um jogo simples, com milhares de jogadores, e um dispendioso, com anos de desenvolvimento, que se torna um fracasso.

Podemos citar um caso recente, a introdução do Wii, console de sétima geração da Nintendo, que não trouxe evoluções na parte gráfica, mas procurou inovar em jogabilidade. Além do controle diferenciado, a maioria dos jogos são simples, com foco na diversão, agradando a um público muito mais abrangente. Tornou-se o console mais bem sucedido dessa geração, mesmo sendo inferior em termos de desempenho (Doolittle, 2007).

Figura 7 – *Wii Sports*: jogos simples, com público alvo abrangente

3.1.1 Jogos casuais e sociais

Jogos simples e rápidos já são bastante difundidos na internet, os chamados jogos casuais, que como o próprio nome diz, são feitos para serem jogados casualmente e não demandam muito tempo de atenção, fornecendo entretenimento quase que instantaneamente.

Na mesma linha há os jogos sociais, encontrados em redes como o *Orkut* ou o *Facebook*. É um tipo de jogo casual, mas para vários jogadores que se encontram dentro de uma rede social. Além de interagir dentro do jogo, compartilham *status* e conquistas utilizando as ferramentas da própria rede social, o que estimula os jogadores. Jogos casuais e sociais ganharam foco nos últimos anos, devido ao seu crescente sucesso (EDGE, 2008) (Cashmore, 2010).

Figura 8 – *Farmville*, construa sua fazenda e faça amigos.

3.2 Jogos multi-jogadores

Jogos multi-jogadores são aqueles em que mais de uma pessoa pode jogar ao mesmo tempo, cooperativamente ou um contra o outro. Apesar de ser uma premissa comum, no meio eletrônico só começou a se difundir no começo dos anos 90, com os clássicos *Doom* e *Neverwinter Nights*.

A maioria dos jogos eletrônicos é apenas para um jogador, com o objetivo de vencer os desafios programados e inimigos controlados por inteligência artificial. Dependendo do tipo de jogo, competir com outras pessoas é muito mais divertido, além de permitir a interação social entre pessoas geograficamente distantes. Jogos multi-jogadores fornecem uma experiência que não poderia acontecer de outra forma senão em um ambiente virtual.

3.2.1 Jogos em rede local e internet

Com relação à rede, jogos multi-jogadores podem ser implantados em redes locais ou utilizar a internet. Jogos em redes locais são muito comuns, como se pode observar nas chamadas "*lan houses"* por exemplo, casas que alugam computadores em rede para jogos com vários jogadores.

Do ponto de vista dos aparelhos celulares, uma rede local pode ser obtida através de tecnologias de transmissão sem fio de curto alcance, como o *bluetooth* ou infra-vermelho. A transmissão de dados é relativamente rápida, mas nem todos os aparelhos disponibilizam esse recurso e o alcance de transmissão é curto, de forma que os dispositivos precisam estar próximos para se comunicar.

Já o acesso à internet em aparelhos celulares pode ser feito de qualquer lugar, desde que dentro da cobertura das operadoras, e está disponível em praticamente todos os aparelhos, com tecnologias como o GPRS. Contudo, a transferência de dados através da rede móvel pode ainda ser cara e apresentar problemas de instabilidade.

Podemos classificar os jogos multi-jogadores também quanto à forma de interação entre os jogadores, como sendo em tempo real, baseada em turnos ou assíncrona (Friedmann, 2009). Essas diferentes formas de interação são descritas a seguir.

3.2.2 Interação em tempo real

Jogos em tempo real são aqueles em que qualquer jogador pode atuar a qualquer momento, alterando o estado do jogo continuamente, como um jogo de futebol. Destacam-se pelo alto grau de interação, mas em termos computacionais esse modelo é o mais complexo, pois, a cada instante, as ações de todos os jogadores devem ser processadas para gerar o estado do jogo no instante seguinte, que consiste em poucos millisegundos. Esse tipo de jogo requer um sistema robusto, tolerante a falhas e com resposta rápida, e é altamente dependente da qualidade da conexão dos jogadores (Cecin & Trinta, 2007).

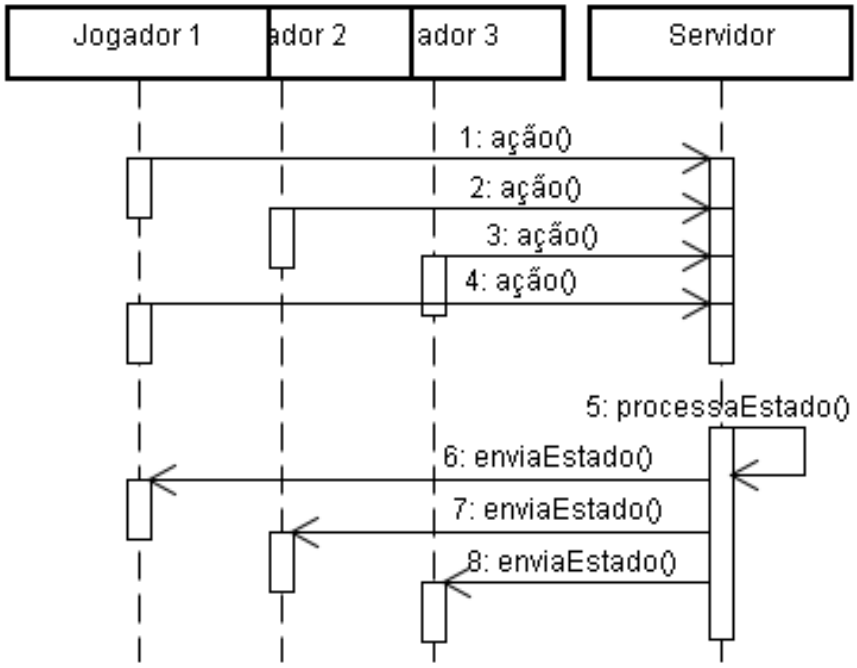


Diagrama 1 – Sequência de interações de um jogo em tempo real

3.2.3 Interação em turnos

Nos jogos baseados em turnos, joga apenas um jogador por vez e há um determinado tempo ou ação que determina o fim do turno, como no jogo de xadrez. Nesse modelo o estado do jogo é alterado apenas uma vez a cada jogada, reduzindo muito a quantidade de informações que devem ser processadas.

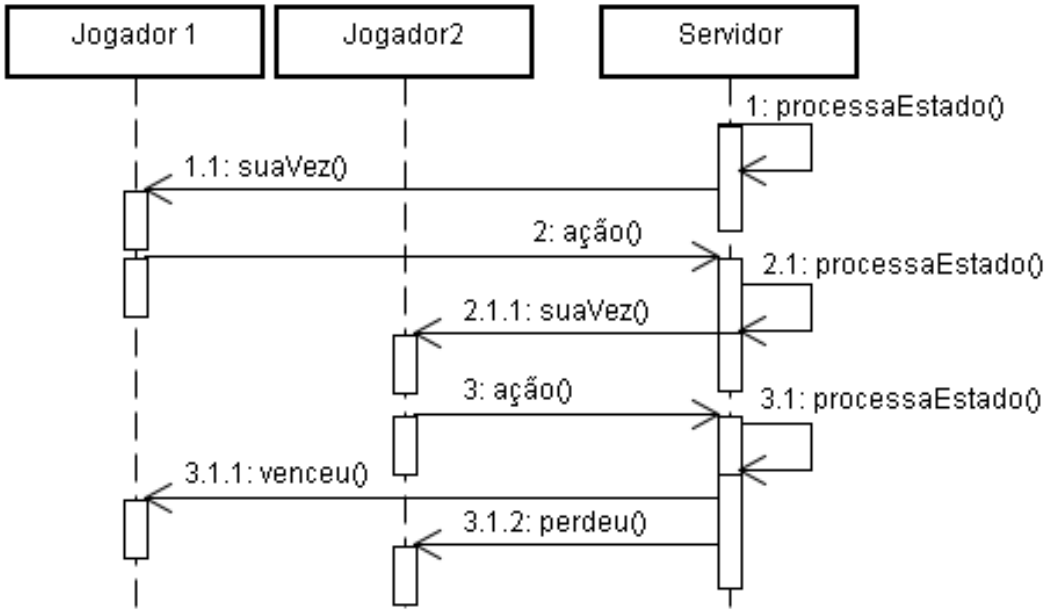


Diagrama 2 – Sequencia das interações de um jogo baseado em turnos

3.2.4 Interação assíncrona

Há ainda a interação de forma indireta ou assíncrona, bastante utilizada em jogos sociais. Cada jogador possui um ambiente próprio dentro do jogo, onde pode atuar sem restrições. Mas a interação entre os jogadores é limitada, de forma que não afeta a continuidade do jogo para os demais. Assim, não há a necessidade de estarem jogando simultaneamente.

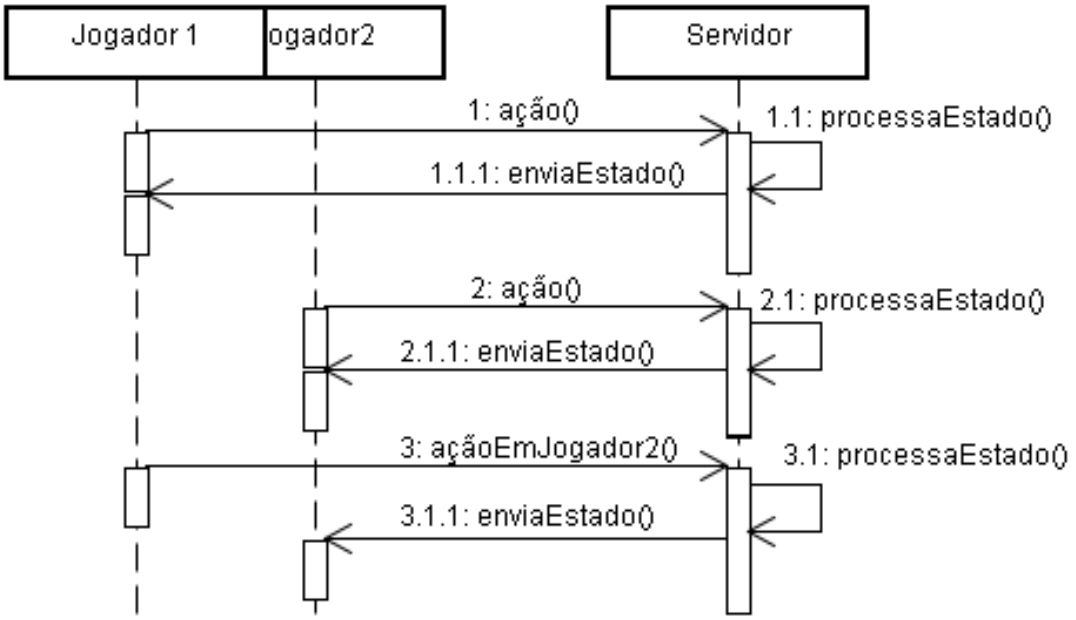


Diagrama 3 – Sequência de interações assíncronas em um jogo

3.3 Arquiteturas de sistemas distribuídos em jogos multi-jogadores

Sistemas distribuídos são aqueles em que dois ou mais pontos trocam informações através de uma rede, como é o caso de um jogo com vários jogadores. Ao menos um dos pontos da rede deve ser responsável pelo processamento das informações e deve haver uma forma de distribuir essas informações entre todos os pontos. Há duas abordagens comumente utilizadas nesse cenário: sistemas ponto-a-ponto e cliente/servidor.

3.3.1 Sistemas ponto-a-ponto

Em sistemas ponto-a-ponto aplicados a jogos, cada participante mantém uma cópia do estado atual do jogo e fica responsável por processar uma parte da informação, geralmente referente às suas próprias ações, e transmiti-la aos demais. Os recursos de todos os jogadores são utilizados para o processamento das informações, que ocorre então de forma descentralizada (Cecin & Trinta, 2007).

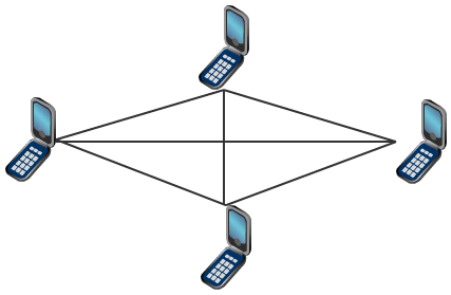


Diagrama 4 – Representação de um sistema ponto-a-ponto

Devido a essas características, nesse modelo é ideal que o número de participantes por jogo seja pequeno, pois é preciso manter a sincronia do estado do jogo entre todos os jogadores. Esse modelo é altamente dependente da velocidade de conexão e do poder de processamento dos dispositivos utilizados pelos jogadores, pois esses aspectos definirão a velocidade do andamento do jogo, geralmente determinada pelo dispositivo de menor capacidade.

Podemos citar o *Age of Empires* como exemplo de jogo que utiliza essa abordagem. Um clássico de estratégia em tempo real para computadores, em que até oito jogadores podem jogar simultaneamente.

O modelo ponto-a-ponto entre aparelhos celulares não é muito interessante, pois todo o processamento da lógica do jogo assim como o tráfego de dados recaem sobre os aparelhos celulares. A maioria dos aparelhos ainda possui capacidade de processamento e quantidade de memória reduzidos, devendo-se evitar ao máximo a utilização desses recursos bem como o excesso de consumo da bateria.

Além disso, sistemas ponto-a-ponto na rede móvel não existem como na rede fixa, sendo necessário uma configuração específica, pois o endereçamento dos dispositivos ocorre de forma diferenciada devido à constante alteração na localização dos aparelhos (Oliveira, 2005).

3.3.2 Sistemas cliente/servidor

No modelo cliente/servidor uma ou mais máquinas, representando o servidor, são dedicadas ao processamento e distribuição dos dados. O estado atual de todo o sistema é centralizado no servidor e a função dos demais pontos, definidos como clientes, é enviar seus dados ao servidor, que irá então processar as informações, atualizar o estado do jogo e transmiti-lo a todos os jogadores.

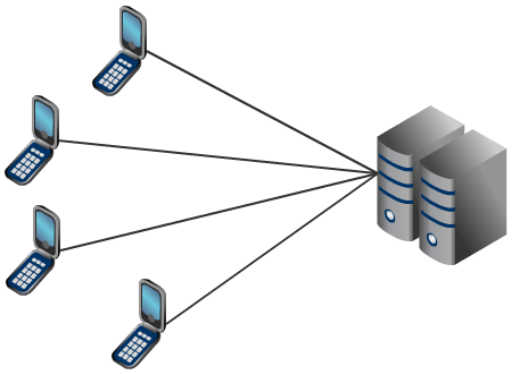


Diagrama 5 – Representação de um sistema cliente/servidor

Um exemplo de jogo que utiliza o modelo cliente-servidor é o *World of Warcraft*, que gerencia mais de 11 milhões de usuários em tempo real, sobre um grande e detalhado mundo virtual.

Em um sistema que envolve aparelhos celulares, podemos utilizar um computador como servidor, dessa forma a complexidade do aplicativo não é limitada pela capacidade dos aparelhos celulares. Essa abordagem também traz a vantagem do controle de acesso ao sistema, já que todos os usuários devem conectar-se ao servidor para ter acesso.

Há ainda outras arquiteturas derivadas dos modelos aqui apresentados, mas estas não serão abordadas nesse trabalho.

3.3.3 Servidores de jogos

O desenvolvimento de servidores para jogos multi-jogadores é tão complexo quanto o volume de dados transferidos e o número de usuários que se deseja suportar. Em servidores de jogos com grande volume de dados trafegados, como jogos massivos em tempo real, é comum a utilização de várias máquinas operando em conjunto com bastante memória e alto poder de processamento, além de técnicas e padrões de projeto criados especialmente para esses casos.

Da mesma forma, os jogos sociais que chegam também a milhares de jogadores acessando e executando ações a todo instante, precisam de um sistema robusto e seguro, principalmente no que diz respeito às transações em bancos de dados.

Dado o escopo e propósito acadêmico desse trabalho, o foco da implementação do servidor será apenas a funcionalidade do sistema, como um exemplo, com o objetivo de suportar o jogo implementado nesse trabalho.

3.4 Internet móvel

A internet móvel evoluiu consideravelmente nos últimos anos, e a expectativa é de que continue evoluindo. Desde as tecnologias GPRS/EDGE o serviço é suficiente para aplicações com baixa demanda, com tráfego de 40kbps a 130kbps. Hoje essas tecnologias são suportadas

em praticamente todos os aparelhos GSM, que representam cerca de 90% do total no Brasil (Teleco, 2010).

Com a chegada da tecnologia 3G, a internet móvel entrou na era ”banda larga”, com transferências de 200 a 700kbps, fornecendo uma experiência semelhante à da internet via cabo.

Geração	**2G**	**2,5 G**	**2,5/3 G**	**3G (UMTS)**	
Tecnologia	GSM	GPRS	EDGE	WCDMA	HSDPA
Taxa de dados máx. teórica (kbit/s)	14,4	171,2	473.6	2.000	14.000
Taxa de dados média (kbit/s)	-	30-40	100-130	200-300	400-700

Tabela 1 – Evolução da tecnologia GSM (adaptado deTeleco, 2008)

O número de acessos à internet a partir de dispositivos móveis vem crescendo gradativamente e calcula-se que o tráfego de dados deve aumentar em torno de sessenta e seis vezes nos próximos cinco anos (Cisco, 2009), já tendo ultrapassado o número de acessos fixos em vários países. No Brasil, a banda larga móvel crescerá mais de 70% até 2014, ultrapassando o número de acessos fixos em meados de 2011, segundo as projeções da Teleco (Huawei/Teleco, 2009).

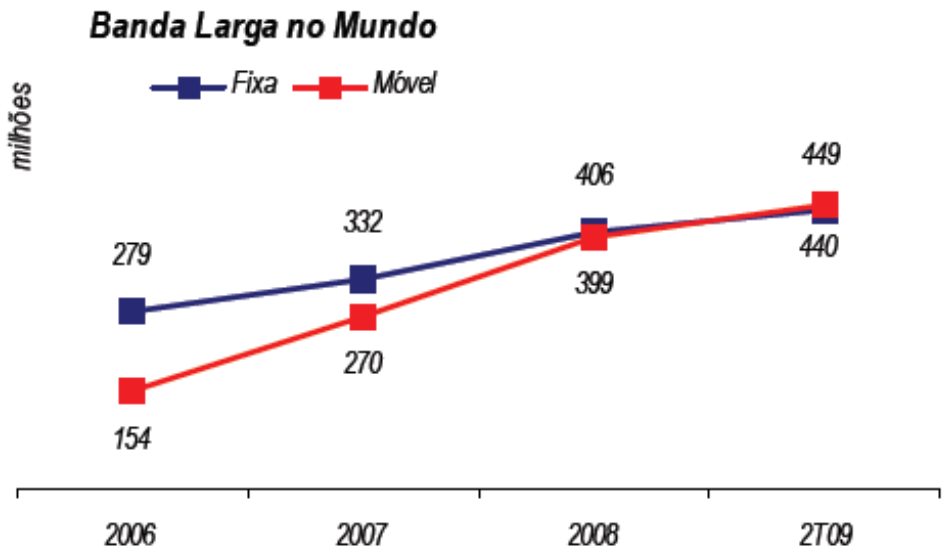


Gráfico 2 – Crescimento da banda larga no mundo
(adaptado de Huawei/Teleco, 2009)

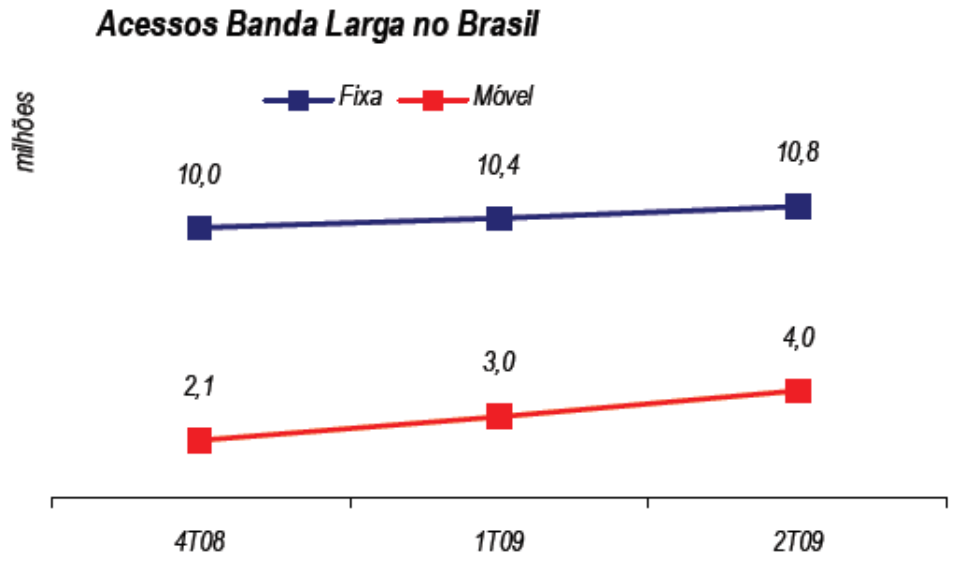


Gráfico 3 – Crescimento da banda larga no Brasil
(adaptado de Huawei/Teleco, 2009)

Mas apesar das projeções, o avanço da internet móvel vem enfrentando dificuldades, principalmente no Brasil. A disponibilidade é baixa para o segmento pré-pago, que representa mais de 70% dos usuários, e o preço é proibitivo. A maioria das pessoas tem medo de acessar e acabar gastando muito. O que pode mesmo acontecer, se considerarmos as tarifas cobradas por algumas operadoras.

Sendo cliente pré-pago na TIM, por exemplo, o acesso é de R$15,73 por MB, o que significa um custo de aproximadamente R$7,50 por página da web visitada, segundo dados da própria TIM (TIM, 2009). Clientes pós-pagos sem um plano de dados pagariam entre R$4 e R$6 por MB. Há ainda planos de dados por limite de velocidade ou volume de dados, com valores mensais de R$119,90 por uma velocidade de até 1MB, e R$69,90 por até 1GB de dados. Mesmo para clientes pós-pagos, esses valores ainda não condizem com a realidade dos brasileiros.

Uma pesquisa recente do Centro de Estudos sobre as Tecnologias da Informação e da Comunicação (CETIC, 2009) mostra o lado pessimista das projeções de acesso à internet móvel. Segundo a pesquisa, nos últimos cinco anos o crescimento percentual de usuários que acessam a internet pelo celular foi de 1%, chegando ao total de apenas 6% em 2009 , como mostra o gráfico 4. A pesquisa também atribui esse resultado ao preço praticado pelas operadoras.

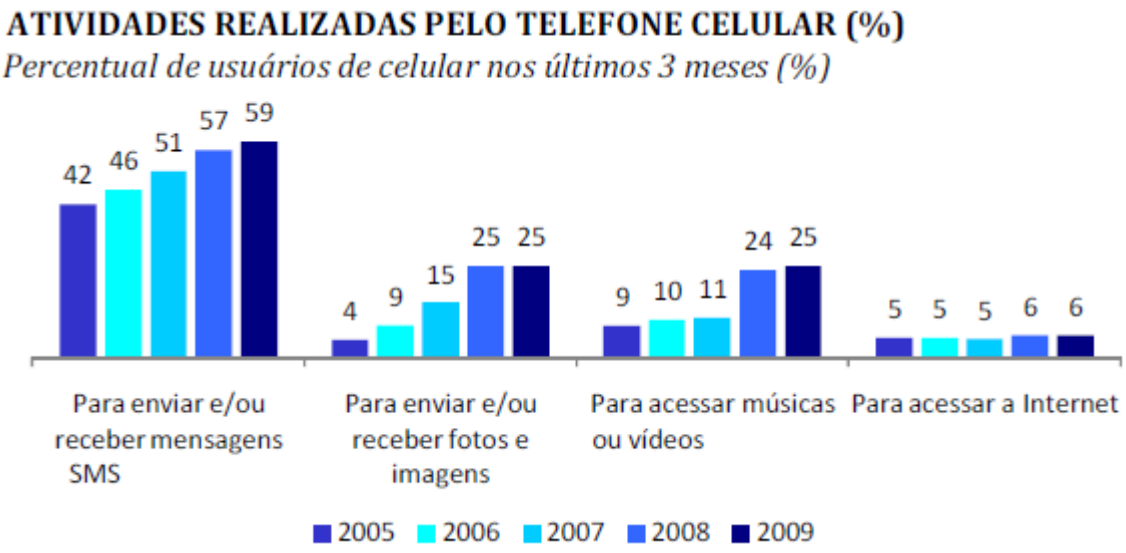


Gráfico 4 – Percentual de atividades realizadas pelo celular nos últimos anos (adaptado de CETIC, 2009)

Nesse cenário, é ideal que uma aplicação para dispositivos móveis que utilize a internet gere o mínimo de tráfego de dados possível. Esse aspecto será abordado na etapa de projeto e implementação deste trabalho.

3.4.1 Problemas da rede móvel

A rede de telefonia móvel é formada por várias antenas responsáveis pela transmissão do sinal de rádio que os dispositivos móveis utilizam. Cada antena cobre uma região delimitada, também chamada de célula, e tem um alcance e largura de banda máximos. Quanto mais longe um aparelho estiver, mais fraco será o seu sinal e maior será a chance de falhas na conexão e transmissão de dados. Caso hajam muitos usuários em uma mesma região, compartilhando a mesma antena, pode ocorrer lentidão na transmissão e até perda de conexão.

A queda da conexão é um dos problemas mais frequentes em aplicativos que utilizam a rede móvel, devido à instabilidade e variação da intensidade do sinal. Porém, carga insuficiente na bateria do dispositivo e até mesmo falta de créditos do usuário com a operadora também podem causar a desconexão do usuário.

Devido a essas situações, a perda de conexão pode ocorrer a qualquer momento durante um jogo, portanto, é indispensável o uso de mecanismos de tolerância a falhas que tratem problemas como perda de pacotes e de conexão. O sistema deve estar preparado para corrigir essas falhas da forma mais consistente e transparente possível, para que a continuidade do jogo não seja prejudicada e o impacto para os usuários seja mínimo.

Em jogos multi-jogadores com interação em tempo real é difícil não prejudicar o andamento do jogo quando ocorrem falhas na transmissão, já em jogos baseados em turnos ou de

interação assíncrona, é possível tratar essas falhas de forma que causem pouco ou nenhum problema ao jogo.

São comuns casos de perda momentânea da conexão, devido à baixa intensidade de sinal ou transição de uma antena para outra. Uma forma de contornar esse problema é estabelecer um tempo de espera para o retorno do jogador e implementar um mecanismo de reentrada no servidor, para que o jogador seja recolocado no estado em que estava antes de perder a conexão.

No caso de um jogador perder a conexão durante uma sessão de jogo e não retornar dentro de um determinado tempo, deve ser considerada a desistência do mesmo e declarada vitória àquele que permanecer conectado. A penalização do jogador que perde a conexão evita que a prática de desconexão seja utilizada de forma intencional, como ferramenta de trapaça.

Uma falha na transmissão pode também fazer com que uma ou mais mensagens não cheguem ao seu destino. Uma maneira de contornar esse problema é incluir no protocolo da aplicação uma confirmação de recebimento de mensagens, para garantir que elas de fato estão chegando ao seu destino.

3.5 Jogos no celular

Apesar do avanço dos jogos eletrônicos, os jogos em aparelhos celulares, com exceção do iPhone, acabaram não chamando muita atenção, mesmo com milhares de títulos disponíveis e gigantes como a Eletronic Arts no mercado.

Aparelhos celulares são dispositivos de uso diário, porém, de recursos limitados, para não mais de alguns minutos de uso constante. Levando em conta esses aspectos de agilidade e simplicidade, a melhor opção para essa plataforma são os jogos casuais, que não são complexos, têm jogabilidade simples e podem ser jogados em curto espaço de tempo. Como exempo de jogo com essas características podemos citar o *Snake*, lançado pela Nokia em 1997.

Um dos problemas dos jogos atuais para celular é que frequentemente são versões reduzidas de jogos de outras plataformas. O público alvo desses jogos em sua maioria é composto por jogadores costumazes, que jogam com frequência e portanto possuem outros aparelhos como um console de mesa ou portátil. Aparelhos celulares não são capazes de fazer frente a essas máquinas e as versões reduzidas de jogos de outras plataformas dificilmente fornecem uma experiência relevante.

Outro problema recorrente é a péssima jogabilidade. O teclado dos aparelhos celulares não é ideal para jogos, o que torna alguns deles frustrantes. Isso ocorre principalmente com títulos de ação ou corrida, quando é necessário pressionar vários botões em curto espaço de tempo.

Figura 9 – *Kung Fu Panda, Fifa Street 3* e *Need For Speed ProStreet*, da EA Mobile

Ainda quanto à jogabilidade, há algumas definições de boas práticas no que diz respeito ao desenvolvimento dos controles de um jogo para celular. Como por exemplo, a utilização de poucos botões para que seja possível jogar apenas com uma das mãos.

Por último, não há um canal simples de distribuição de aplicativos, como a App Store da Apple, para as outras plataformas móveis. Existem portais de operadoras, fabricantes ou terceiros, mas dificilmente possuem um modelo de negócio atrativo para desenvolvedores. Até o método de instalação de aplicativos em aparelhos celulares é desconhecido para a maioria das pessoas, que não estão habituadas a ver as outras possibilidades de uso que esses dispositivos podem oferecer.

3.6 Plataformas de desenvolvimento

Com a grande quantidade de aparelhos diferentes no mercado, são muitas as plataformas disponíveis, variando em linguagem de programação, desempenho, disponibilidade de recursos e abrangência no mercado.

Baseadas nas linguagens C e C++ existem a plataforma BREW, pouco difundida, encontrada em aparelhos CDMA, e a plataforma Symbian, predominante no mercado de *smartphones* em todo o mundo. Ambas possuem alto desempenho, e fornecem acesso a funções nativas dos aparelhos (Qualcomm, 2010) (Symbian Foundation, 2010).

O J2ME, versão de Java para dispositivos móveis, por ser independente de sistema operacional é encontrado em dois terços dos aparelhos, inclusive em outras plataformas como Symbian, Windows Mobile e Blackberry. Porém, perde em desempenho e acesso a recursos nativos (Oracle, 2010).

iPhone e Android são baseados em Linux. O iPhone utiliza Objective C e o Android utiliza Java. São mais recentes, possuem alto desempenho e disponibilizam inúmeros recursos, porém ainda poucas pessoas têm acesso a essas tecnologias, principalmente pelo preço (Apple, 2010) (Google, 2010).

O Blackberry possui sistema operacional próprio e disponibiliza uma API baseada em Java. É mais difundido na América do Norte. Fornece suporte a funções nativas, mas perde em desempenho para outros *smartphones* (Research In Motion, 2010).

O Windows Mobile, sistema operacional da Microsoft para *smartphones*, utiliza as linguagens C e C++. Possui bom desempenho e inúmeros recursos, mas representa apenas uma pequena fatia no mercado (Microsoft Corporation, 2010).

Como o J2ME é ainda o mais abrangente, e fornece recursos suficientes, esta será a plataforma utilizada neste trabalho para implementação dos jogos.

3.6.1 Jogos através do navegador

O navegador de internet dos aparelhos celulares (*microbrowsers*) também aparece como uma possível plataforma para desenvolvimento, atingindo virtualmente todos os modelos de aparelhos. Mas para jogos, os recursos disponíveis são insuficientes, além do alto custo do tráfego gerado para o usuário.

3.7 Desenvolvimento de jogos para celular

O desenvolvimento de jogos não é uma atividade trivial. Possuem lógicas e interfaces complexas, utilizam diversos efeitos visuais e sonoros, gerenciam vários elementos independentes ao mesmo tempo e é fundamental que mantenham o controle de recursos na memória e da velocidade de execução, para uma experiência contínua, rica e consistente. Essas características implicam em métodos e técnicas avançadas para obter o melhor visual e desempenho.

Em se tratando de dispositivos móveis, o desenvolvimento de jogos apresenta ainda outras complicações. Os recursos disponíveis, como tamanho de tela e a quantidade de memória, variam muito devido ao grande número de fabricantes e modelos diferentes. Mesmo aparelhos que na teoria deveriam funcionar da mesma forma, muitas vezes se comportam de modo diferente ao executar um mesmo aplicativo. Essas diferenças trazem a necessidade de adaptações nos aplicativos (*porting*) para cada grupo diferente de aparelhos.

Apesar das dificuldades, existem hoje várias formas de contornar esses problemas, desde a própria concepção do jogo até ferramentas de desenvolvimento, que fazem automaticamente uma adaptação do código para vários modelos diferentes de aparelhos.

Como exemplo podemos citar a variedade de tamanhos de tela diferentes. Na verdade mais da metade dos aparelhos pertence a um grupo de apenas quatro tamanhos, como mostra a tabela de número 2. É possível utilizar essas quatro ocorrências mais comuns como base e implementar ajustes automáticos para os tamanhos restantes.

Common pairs of values		
Screen Width	**Screen Height**	**%**
240	320	21.7
128	160	16.9
176	220	14.8
128	128	8.9

Tabela 2 – Tamanhos de tela mais comuns em aparelhos celulares (adaptado de DeviceAtlas, 2010)

Jogos em geral apresentam muitas funcionalidades em comum, como renderização e animação, por isso também é comum a utilização de *frameworks* para facilitar e acelerar o desenvolvimento. Um *framework* é um conjunto de classes criado com a intenção de reuso. É um sistema incompleto, mas que fornece uma base que pode ser utilizada na criação de outros sistemas.

3.8 Motores de jogos

O termo "motor de jogo" (*game engine*), é utilizado para definir um *framework* no âmbito de desenvolvimento de jogos. Não é difícil encontrar jogos similares na visão do jogador e jogabilidade, com poucas diferenças além da arte gráfica. Esses jogos são criados usando um mesmo "motor". Como exemplos podemos citar os jogos "*Half-Life*", "*Counter-Strike*" e "*Portal*" para PC, e os jogos "*Grand Theft Auto*" e "*Bully*" para PS2.

Um motor de jogos geralmente apresenta suporte para renderização 2D ou 3D, animações, sons, simulação de física, detecção de colisões, suporte para comunicação através de uma rede, entre outras, dependendo do tipo de jogo que se deseja desenvolver.

Embora existam *frameworks* para o desenvolvimento de jogos em muitas plataformas, há relativamente poucos para aparelhos celulares. Como utilizaremos a plataforma J2ME na implementação, dois *frameworks* para desenvolvimento de jogos nessa linguagem foram selecionados para análise: o pacote *javax.microedition.lcdui.game* da própria biblioteca J2ME e o *SNAP Mobile* da Nokia.

3.8.1 J2ME – Pacote javax.microedition.lcdui.game

A versão de Java para dispositivos móveis compreende uma coleção de pacotes e configurações diferentes para vários tipos de dispositivos. No caso dos celulares, todos levam o perfil MIDP (*Mobile Information Device Profile*) e os pacotes disponíveis dependem de qual versão e configuração o aparelho possui (Sun Microsystems, 2006).

A partir da versão 2.0, o MIDP contém o pacote *javax.microedition.lcdui.game* para o desenvolvimento de jogos para celular. Esse pacote possui apenas cinco classes, com funções de renderização, controle de objetos gráficos e interface. Como faz parte da biblioteca MIDP, outros pacotes podem ser utilizados para outras necessidades do jogo, como efeitos sonoros, conexão com a internet, acesso ao sistema de arquivos, etc.

As classes que compõem o pacote são:

- *GameCanvas*: É subclasse de *Canvas* da biblioteca MIDP. Possui métodos para renderização gráfica e de controle de *inputs* do usuário.
- *Layer*: É a base para todos os elementos visuais. Possui atributos como posição, tamanho e visibilidade. Todos os elementos gráficos do *framework* são especializações dessa classe, porém é de acesso restrito ao pacote.
- *TiledLayer*: Possui uma matriz onde cada posição representa um pedaço de cenário em formato retangular, também chamado de *tile*. Cada *tile* contém uma imagem e o conjunto dessas imagens, na ordem em que foram distribuídas na matriz, compõe um cenário completo.
- *Sprite*: Representa um elemento gráfico básico com animação. Também disponibiliza funções de transformação de imagem e detecção de colisão.
- *LayerManager*: É um gerenciador de objetos da classe *Layer*. Fornece controle sobre as camadas de elementos gráficos, a ordem e a posição em que serão renderizados.

Para criar um jogo utilizando esse pacote, é preciso implementar um ciclo de jogo e utilizar a classe *GameCanvas* para renderização e obtenção de *inputs*.

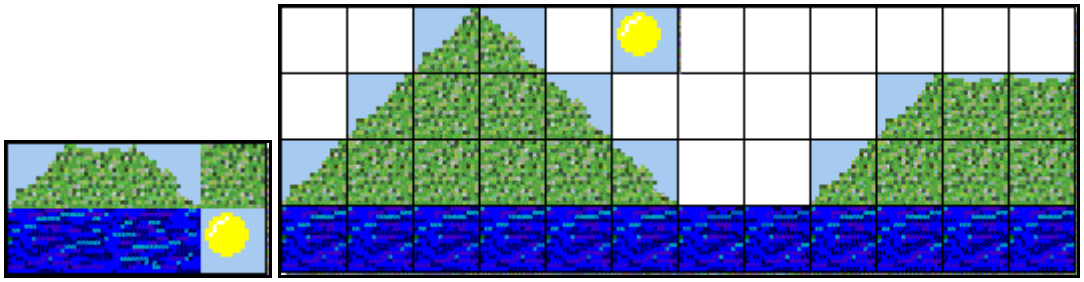
O trecho de código a seguir apresenta um exemplo de uso da classe *GameCanvas*:

```
public class GameExample extends GameCanvas implements Runnable {

  public void run() {
    // Ciclo do jogo.
    while (true) {
      update();
      render();
    }
  }

  public void update() {
    int keyState = getKeyStates();
    // Trata o input e atualiza os objetos.
  }

  public void render() {
    Graphics g = getGraphics();
    // Faz a renderização.
    flushGraphics();
  }
}
```

Trecho de código 1 – Exemplo de uso da classe *GameCanvas*

Para adicionar os elementos do jogo como cenário e personagens, deve-se utilizar as classes provenientes de *Layer*, como veremos a seguir:

Para montar um cenário recomenda-se o uso da classe *TiledLayer*. A partir de uma imagem contendo pequenas partes de uma paisagem, como mostra a figura 10, é possível construir um cenário completo, como mostra a figura 11:



Figuras 10 e 11 – Exemplo de imagem contendo tiles e um cenário criado a partir de um *TiledLayer*

Ao criarmos um objeto da classe *TiledLayer*, passamos como parâmetros uma imagem, o número de linhas, o número de colunas e o tamanho de cada *tile*. E então, setamos a imagem em cada posição passando como parâmetros a posição do *tile* na imagem base e a linha e coluna desejada no cenário.

O trecho de código a seguir apresenta um exemplo de uso da classe *TiledLayer*:

```
Image image = null;
try { image = Image.createImage("/tiles.png"); }
catch (IOException ioe) { return null; }

TiledLayer tiledLayer = new TiledLayer(10, 10, image, 16, 16);

// Mapa com o número dos tiles em suas respectivas posições
int[] map = {
    0,  0,  1,  3, 0,  8,  0,  0,  0,  0,
    0,  1,  4,  4,  3,  0,  0,  1,  2,  2,
    1,  4,  4,  4,  4,  3,  0,  0,  0,  0,
    0,  0,  0,  0,  0,  0,  0,  7, 1,  0,
    6,  6,  6,  6,  6,  6,  6,  6,  6,  6,
};

// Seta os tiles no tiledLayer
for (int i = 0; i < map.length; i++) {
    int column = i % 10;
    int row = (i - column) / 10;
    tiledLayer.setCell(column, row, map[i]);
}

// Pinta o tiledLayer
public void paint(Graphics g){
    tiledLayer.paint(g);
}
```

Trecho de código 2 – Exemplo de uso da classe *TyledLayer*

Para adicionar elementos animados ao jogo como personagens ou inimigos utilizamos a classe *Sprite*. Assim como a classe *TiledLayer*, a classe *Sprite* é capaz de gerar frames a partir de uma única imagem, como mostra a figura 12:

Figura 12 – Exemplo de imagem contendo frames para animação de um *Sprite*

Uma vez criado, o *Sprite* fornece controles sobre a animação como: seleção da sequência de frames, avançar para o próximo frame, para o anterior, etc.

O trecho de código a seguir apresenta um exemplo de uso da classe *Sprite*:

```
Image image = null;
try { image = Image.createImage("/sprite.png"); }
catch (IOException ioe) { return null; }

// Cria o Sprite e seta a sequencia de frames
Sprite s = new Sprite(image, 32, 32);
int[] animation = new int[]{0,1,2,1};
s.setFrameSequence(animation);

// Avança para o próximo frame a cada atualização
public void update(){
    s.nextFrame();
}

// Pinta o Sprite
public void paint(Graphics g){
    s.paint(g);
```

Trecho de código 3 – Exemplo de uso da classe *Sprite*

A classe *Sprite* possui ainda métodos para rotação e inversão de imagem e testes de colisão. Testes de colisão são necessários na maioria dos jogos, e sua presença nessa classe é de grande importância, pois dispensa o desenvolvimento de algoritmos adicionais para executar essa tarefa.

Para gerenciar os diversos elementos gráficos dentro de uma cena utiliza-se a classe *LayerManager*, que contém um grupo de objetos do tipo *Layer*. Através dessa classe podemos adicionar ou remover objetos da cena e controlar a posição e a ordem em que são pintados.

O trecho de código a seguir apresenta um exemplo de uso da classe *LayerManager*:

```
// Cria os elementos da cena
Sprite spt = createSprite();
Sprite spt2 = createSprite();
TiledLayer tl = createTiledLayer();

// Cria um LayerManager e adiciona os elementos
LayerManager lm = new LayerManager();
lm.append(tl);
lm.append(spt);
lm.append(spt2);

// Pinta o LayerManager e todos os elementos nele contidos
public void paint(Graphics g){
    lm.paint(g);
```

Trecho de código 4 – Exemplo de uso da classe *LayerManager*

Como resultado dessa análise, conclui-se que o pacote *javax.microedition.lcdui.game* apresenta funcionalidades básicas necessárias para o desenvolvimento de jogos, porém, não suficientes. O desenvolvimento é simples e focado principalmente para jogos baseados em *tiles* e *Sprites*, mas faltam elementos gráficos e funcionalidades para outros tipos de jogos. É pouco extensível, uma vez que a classe *Layer*, base para os elementos gráficos do pacote, é protegida. E também não dá suporte à criação de interfaces personalizadas dentro do jogo, com botões e caixas de texto, por exemplo. Caso forem necessárias é preciso criá-las manualmente ou utilizar uma tela à parte utilizando então os componentes disponíveis no sistema.

3.8.2 Nokia SNAP Mobile

O SNAP Mobile está para os jogos em Java assim como o N-Gage para os jogos em Symbian. É um *framework* para jogos mullti-jogador desenvolvido pela Nokia, sobre a biblioteca MIDP e chegou a ser distribuído junto com o pacote de desenvolvimento de Java para dispositivos móveis (Nokia, 2009).

Figuras 13 e 14 – *SNAP Mobile*: suporte para jogos multi-jogadores em tempo real

O diferencial dessa solução é a inclusão de uma comunidade destinada aos jogadores, na qual podem se cadastrar e, além de jogar uns contra os outros, interagir como uma rede social. O *framework* fornece então suporte para o desenvolvimento de jogos multi-jogador, em tempo real, e também funcionalidades voltadas à interação social entre os jogadores, como adicionar amigos, *status* de presença, troca de mensagens e *rankings* dos jogos.

Na arquitetura da solução SNAP Mobile, a troca de mensagens, tanto dos jogos como das funções sociais, é centralizada nos servidores da Nokia, e restrita a eles. Logo, a lógica dos jogos deve ser tratada apenas nos clientes, assim todos os jogos funcionam de forma independente e integrada com a comunidade de jogadores, como mostra a figura 15.

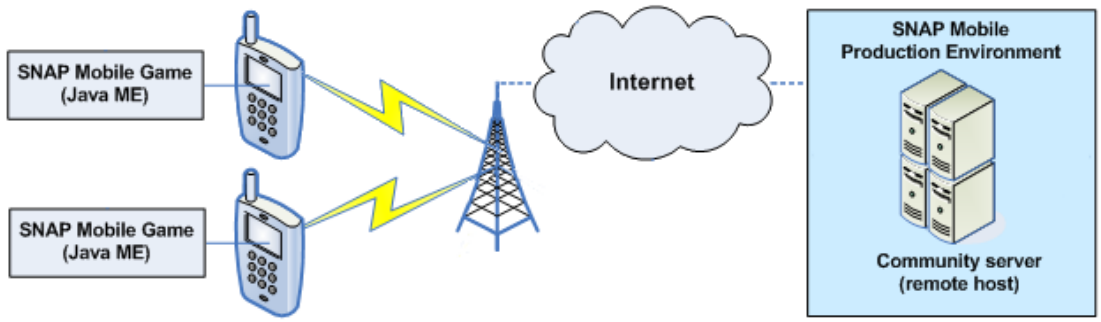


Figura 15 – Arquitetura da solução Nokia SNAP Mobile (adaptado de Nokia, 2009)

Serão apresentados a seguir, os pacotes que compõem o *framework SNAP Mobile*.

Pacote *com.nokia.sm.mui*

É um pacote de componentes de interface, nomeados *widgets*, para criação de interfaces personalizadas. Contém botões, *labels*, caixas de texto, listas, etc. Esses elementos podem ser organizados de várias formas na tela e personalizados através da definição de estilos.

Segue uma descrição das principais classes e interfaces desse pacote:

- *UIWidget*: classe base para todos os componentes do pacote. Possui atributos e funções comuns a todos os componentes, como posição, visibilidade, alinhamento, cor, controle de foco, etc.
- *UIWindow*: representa uma janela, definida por uma região retangular, que atua como um contêiner para outros *widgets*, e possui funções de rolagem de tela.
- *UI*: classe que comporta todos os elementos de interface com o usuário. Fornece funções para manipulação e efeitos como *fade in* e *fade out*.
- *MUI*: classe principal do sistema de interfaceamento com o usuário. Controla funcionalidades de inicialização, visibilidade e seleção de estilos.

- *UIHandler*: interface para tratamento de eventos de interface, como seleção, perda ou ganho de foco, entradas do teclado, etc.

Um aspecto interessante da criação de interfaces utilizando esse *framework* é a montagem de telas a partir de documentos XML, o que pode facilitar bastante o desenvolvimento. Como mostra o trecho de código a seguir:

```
<ui styles="/styles.xml">
   <window id="mainmenu" width="100%" height="100%"
style="transparent">
      <imagebutton id="go_online" y="20%" anchor="top/top/mainmenu"
text="%go_online%" selectoroverrides="/exit//" />

      <imagebutton id="single_player" anchor="top/bottom/go_online"
x="0" y="5" text="%single_player%" />

      <imagebutton id="help" anchor="top/bottom/single_player" x="0"
y="5" text="%help%" />

      <imagebutton id="about" anchor="top/bottom/help" x="0" y="5"
text="%about%" />

      <imagebutton id="settings" anchor="top/bottom/about" x="0" y="5"
text="%settings%" />

      <imagebutton id="exit" anchor="top/bottom/settings" x="0" y="5"
text="%exitGame%" selectoroverrides="///go_online" />
   </window>
</ui>
```

Trecho de código 5 – Exemplo de XML para criação de interfaces com o *SNAP Mobile*

No exemplo podemos verificar a criação de uma janela com o nome “mainmenu”, definida pela *tag “window”*, e a seguir os botões “go online”, “single player”, “help”, “about”, “settings” e “exit”, definidos pela *tag “imagebutton”*. Também podemos verificar a seleção de um estilo logo no início do documento, definido pela *tag “ui”* juntamente com o atributo *“styles”*. O resultado é apresentado na figura 16:

Figura 16 – Exemplo de tela criada através de arquivos XML com o *SNAP Mobile*.

Pacote *com.nokia.sm.client*

É um pacote com funcionalidades para troca de dados através da rede relacionados com a comunidade e os jogos multi-jogadores. Apresenta funções para cadastro na comunidade, conexão, *login*, troca de mensagens de chat, gerenciamento de contatos, *status* de presença, criação de salas e sessões de jogo, envio e recebimento de estatísticas e *ranking* dos usuários, e também o envio de dados durante os jogos, através de pacotes de *bytes*.

A seguir, uma descrição das principais classes e interfaces contidas nesse pacote:

- *SMClient*: classe principal que realiza funções como *login* e cadastro de usuários. Gera automaticamente uma conexão com os servidores da Nokia quando é utilizada.
- *SnapEventListener*: interface para tratamento de eventos recebidos através da rede. Possui métodos como: pedido de jogo, pedido de amigo, mensagem recebida, etc.
- *Session*: classe responsável pela troca de dados quando o usuário está conectado. Possui funções como: criar salas, encontrar usuários, selecionar *status* de presença e enviar e receber convites para sessões de jogo.
- *Room*: base para classes como *Lobby* e *GameRoom*. Representa salas de usuários e fornece métodos para entrar ou sair da sala, enviar mensagens e recuperar o número de usuários.
- *User*: representa qualquer usuário conectado na comunidade SNAP Mobile. Possui métodos para recuperar o nome do usuário, definir e obter atributos quaisquer, que podem ser utilizados na implementação dos jogos.
- *Friend*: extensão de usuário. Um objeto da classe *Friend* possui ainda funções para envio de mensagens diretas e recuperação de *status* de presença.

O trecho de código 6 apresenta uma exemplo de uso das funcionalidades desse pacote.

```
// Trecho da função que efetua o login na comunidade
} else if (currentUiID.equals("/login.xml")) {
    if (widget.id.equals("ok")) {
    String username = (ui.getWidget("username")).text;
    String password = (ui.getWidget("password")).text;
    smController.doLogin(username, password);
(…)

// Trecho da função que envia mensagem de chat
}else if (currentUiID.equals("/chat.xml")) {
    if (widget.id.equals("chat_send_button")) {
        String message = (ui.getWidget("message_field")).text;
        smController.doSendChat(message, friendToChatWith);

// Trecho da função que troca status de presença do usuário
int presenceStatus = 0; // offline
if (choicegroup.selected == 1) {
    presenceStatus = Friend.PRESENCE_STATUS_ONLINE;
    smController.doSetMyPresence(presenceStatus, presenceMessage);
```

```
// Trecho da função que envia posição atual do jogador no mapa
if ((System.currentTimeMillis() - lastPacketSent) >
POSITION_UPDATE_INTERVAL) {
    packInteger(updateBuf, 1, pos);
    log("sending LOC_UPDATE");
    sendGamePacket(updateBuf);
    lastSentPos = pos;
    log("Sending locUpdate" + locUpdateCounter++);
}
(…)
```

Trecho de código 6 – Exemplo de utilização do pacote *com.nokia.sm.client*

Os trechos apresentados nesse exemplo foram retirados do jogo "*MazeRacer*" que acompanha o pacote de distribuição do *framework*, como um exemplo de implementação. A classe *SMController* é uma implementação da interface *SnapEventListener* com acesso a um objeto do tipo *SMClient*.

Pacote *com.nokia.sm.common*

É um pacote com os elementos de controle principais necessários para criação de um jogo. As classes desse pacote foram desenvolvidas para o jogo "*MazeRacer*", disponibilizado como exemplo. Porém, as mesmas classes, com algumas alterações, podem ser utilizadas no desenvolvimento de outros jogos.

As principais classes e interfaces desse pacote são as seguintes:

- *MainMIDlet*: estende *MIDlet*, classe principal de aplicativos MIDP, inicializando o controle de interface do jogo.
- *UIController*: estende *GameCanvas* e implementa *UIHandler*. É responsável pelo ciclo de jogo e pelo gerenciamento da interface e menus.
- SMController: trata o envio e recepção de dados através da rede, implementando *SnapEventListener* e mantendo uma referência a um objeto do tipo *SMClient*.
- *Game*: interface que representa um jogo. Com métodos para inicialização dos controladores e tratamento de pacotes de dados recebidos durante o jogo.

O trecho de código a seguir apresenta um exemplo de uso desse pacote:

```
// Recorte da função que inicializa um jogo multiplayer
public void startMultiPlayerGame(boolean isHost, User[] rivals) {
    maze = new Maze(MAZE_WIDTH, MAZE_HEIGHT);
    (…)
// Se for o host, envia o formato do labirinto para os jogadores
    if (isHost) {
        byte[] mazelayout = maze.getMazeLayout();
        byte[] packet = new byte[mazelayout.length + 1];
        packet[0] = TYPE_LEVEL_SEED;
        System.arraycopy(mazelayout, 0, packet, 1,
mazelayout.length);
        sendGamePacket(packet);
        state = WAITING_FOR_GAME_START_ACK;

// Senao, apenas espera o envio do formato pelo jogador host
    } else {state = WAITING_FOR_SEED;}
}

// Recorte do método de tratamento dos pacotes recebidos
public void gameDataReceived(String from, byte[] data) {
    int id = data[0];
// Recebeu formato do labirinto
    if (state == WAITING_FOR_SEED && id == TYPE_LEVEL_SEED) {
        byte[] mazeLayout = new byte[data.length - 1];
        System.arraycopy(data, 1, mazeLayout, 0, data.length - 1);
        maze.setMazeLayout(mazeLayout);
// Envia aviso de format recebido
        sendGamePacket(new byte[] { TYPE_GAME_START_ACK });
        startGamePlay();
        return;
    }
// recebeu update da posição do outro jogador
    if (id == TYPE_PLAYER_UPDATE) {
        int pos = unpackInteger(data, 1);
        playerList[1].setPosition(pos);
        return;
    }
// mensagem de vitória, reporta os pontos
    if (id == TYPE_YOU_WIN) {
       reportScores(true, true);
       gameStopTimestamp = System.currentTimeMillis();
       state = STATE_LEVEL_COMPLETED;
       return;
    }
```

Trecho de código 7 – Exemplo de uso do pacote *com.nokia.sm.common*

Concluindo a análise, pode-se dizer que o SNAP Mobile é uma plataforma suficiente para o desenvolvimento de jogos multi-jogador para celular. As funções multiplayer são simples, facilitando a implementação. A criação de interfaces via XML economiza código, refletindo positivamente no tamanho final do aplicativo e quantidade de classes carregadas durante o jogo. E ainda possui o diferencial da comunidade de jogadores, que é uma boa oportunidade para desenvolvimento de jogos sociais.

Entretanto, poucos jogos foram desenvolvidos utilizando essa solução e o projeto acabou descontinuado em meados de 2009, após a conclusão dessa análise. O site do projeto (http://snapmobile.nokia.com) foi retirado do ar e foi encontrada uma nota de um funcionário da Nokia no fórum oficial do desenvolvedor, datada de 07/04/2010, notificando que o projeto *SNAP Mobile* foi descontinuado (Nokia, 2010). Alguns dos fatores que acredita-se terem contribuído para o fechamento do projeto são apresentados a seguir.

A arquitetura da solução, restrita aos servidores da Nokia, trazia a necessidade de tratar a lógica dos jogos integralmente nos clientes, limitando a complexidade e lógica dos jogos aos aparelhos celulares.

A falta de um canal de distribuição na época do lançamento (2005), deixando essa tarefa a cargo do desenvolvedor. Ou seja, todos os jogos faziam parte de uma mesma comunidade, mas foram distribuídos de forma descentralizada.

Além disso, a Nokia estipulou um complicado e burocrático processo de pré-aprovação e testes, dificultando o acesso à solução para o desenvolvimento de jogos, principalmente por desenvolvedores independentes. Mesmo hoje, na recém criada loja *online* da Nokia (https://store.ovi.com/), não há jogos produzidos na plataforma *Snap Mobile*.

4. Projeto

De acordo com os objetivos definidos para esse trabalho e com base nas análises realizadas, nesse capítulo serão apresentados os projetos para implementação de um *framework* para desenvolvimento de jogos em J2ME, um *framework* para um servidor de jogos simples multi-jogadores e dois jogos desse tipo para celular, um baseado em turnos e o outro com interação do tipo assíncrona, desenvolvidos a partir dos *frameworks*.

A comunicação deverá ocorrer através da internet. A arquitetura do sistema será do tipo cliente-servidor, utilizando o protocolo TCP, pois o volume de dados trafegados deve ser o menor possível, afim de reduzir o custo e a utilização de recursos dos aparelhos.

4.1 *Framework* para jogos multi-jogadores em J2ME

Jogos possuem uma série de elementos e subsistemas encarregados de executar as diversas funções que os compõem, tais como: controle do ciclo do jogo, renderização, controle de cenas, input, eventos, IA, etc.

O objetivo do *framework* projetado nesse capítulo será fornecer as funcionalidades necessárias para o desenvolvimento dos jogos propostos para esse trabalho e servir de base para o desenvolvimento de jogos futuros. O diagrama 6 ilustra de forma genérica os casos de uso dos jogos aos quais destina-se o *framework*.

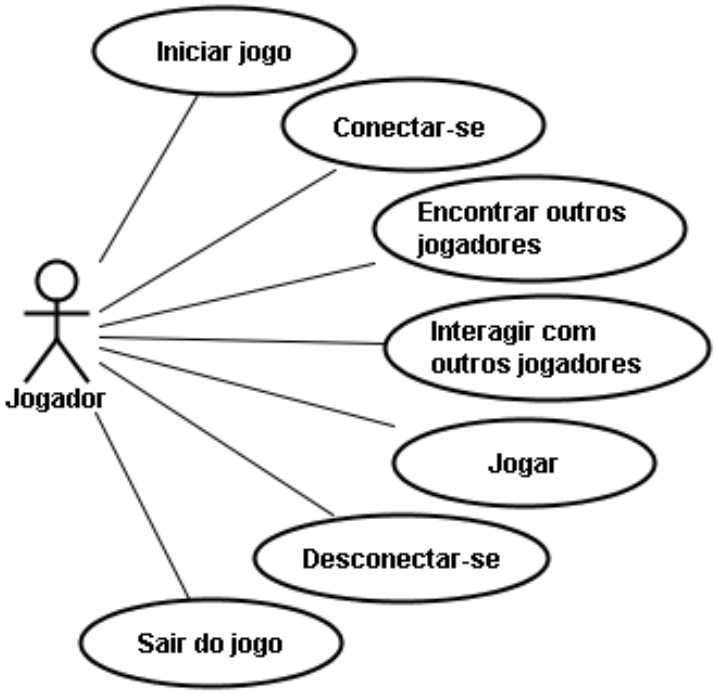


Diagrama 6 – Casos de uso do jogador para o *framework*

Em se tratando de desenvolvimento para dispositivos móveis, deverão ser levadas em conta as limitações existentes quanto à velocidade de processamento e a quantidade de memória

disponível, e também as particularidades resultantes da variedade de modelos existentes no mercado.

A arquitetura do *framework* será composta por módulos, onde cada um implementará uma das funcionalidades que devem ser incluídas nesse *framework*. A seguir serão apresentadas as especificações de cada uma delas.

4.1.1 Ciclo do jogo

Durante o tempo de execução de um jogo há uma sequência de tarefas que devem ser executadas, algumas indefinidamente, como desenhar os objetos na tela, outras a partir da entrada de dados pelo usuário, outras ainda de acordo com o tempo.

Porém, todas essas atividades têm início em um ciclo, que consiste em apenas duas operações básicas:

1- Renderização (desenhar os objetos na tela).

2- Atualização (atualizar o estado dos objetos).

Sobre essas duas operações são implementadas as funções que resultam na experiência visual e interativa que o jogo proporciona.

A seguir, uma descrição detalhada de cada uma delas.

Renderização

A renderização é o processo pelo qual todos os objetos são desenhados na tela. Depende de um objeto gráfico capaz de realizar essa função e é geralmente a tarefa mais dispendiosa em termos de processamento.

Na plataforma J2ME, o objeto gráfico é fornecido por um objeto do tipo *Canvas*, a partir do qual podemos desenhar formas geométricas simples, texto e imagens na tela. Essas funções básicas serão utilizadas para desenhar todos os elementos do jogo.

Atualização

A atualização dos objetos é a etapa em que ocorre a interação com o usuário, é quando os objetos se movem, agem e respondem a ações, iniciadas a partir da entrada de teclas do jogador, de um sistema de inteligência artificial, de um evento disparado, entre outros.

Como exemplo do funcionamento do ciclo do jogo, podemos citar a movimentação de um personagem controlado pelo jogador. Durante a etapa de renderização, a imagem do personagem é desenhada em sua posição inicial, o jogador então pressiona uma tecla para que o personagem se mova. Durante a etapa de atualização, o sistema percebe que a tecla foi pressionada e altera a posição do personagem, de acordo com a dinâmica definida no jogo. Então, durante o próximo ciclo, o personagem será desenhado na nova posição, dando a impressão de movimento. E assim por diante.

Cada ciclo é executado em millisegundos, portanto há várias atualizações a cada segundo. O número de quadros por segundo (*FPS*) representa o número de vezes que a tela é atualizada

em um segundo. Quanto maior o *FPS*, mais suave será a experiência do usuário. É também função do ciclo do jogo manter-se equilibrado para que o número de quadros por segundo seja constante, e a experiência do usuário seja uniforme.

O diagrama 7 apresenta um diagrama de sequência do funcionamento do ciclo do jogo.

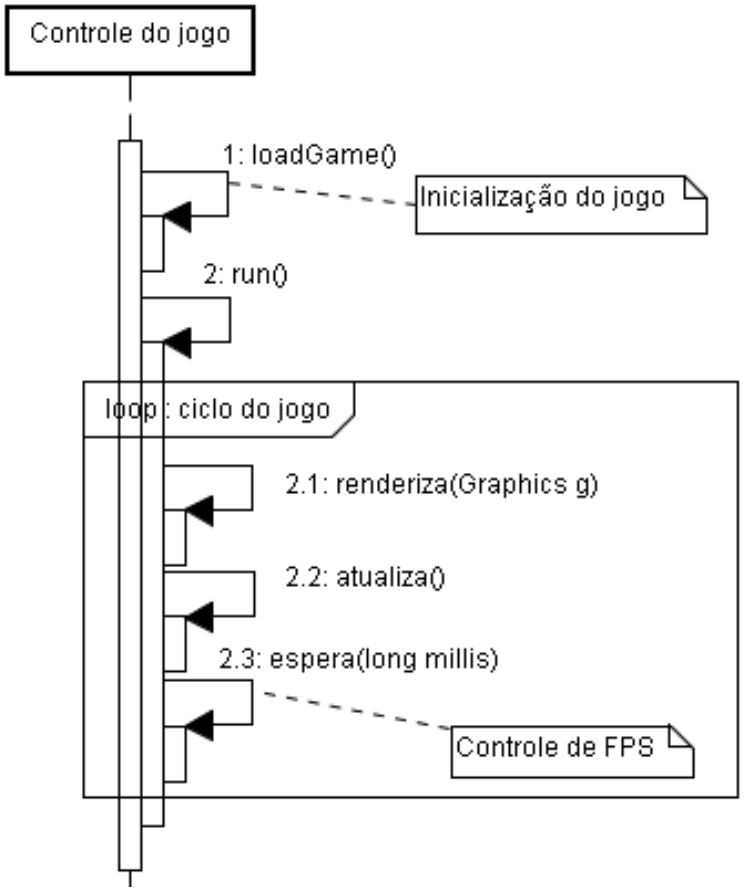


Diagrama 7 – Diagrama de sequência do controle do ciclo do jogo

O módulo responsável pelo controle do ciclo do jogo possuirá apenas uma classe, a qual chamaremos de *GameCore*. Como controle principal do aplicativo, essa classe deverá extender *MIDlet*, uma classe da biblioteca Java, que define os métodos para a inicialização e término da aplicação. Deverá também possuir uma instância da classe *GameCanvas*, a fim de obter o objeto gráfico utilizado para renderização.

O diagrama 8 apresenta o diagrama de classes desse módulo.

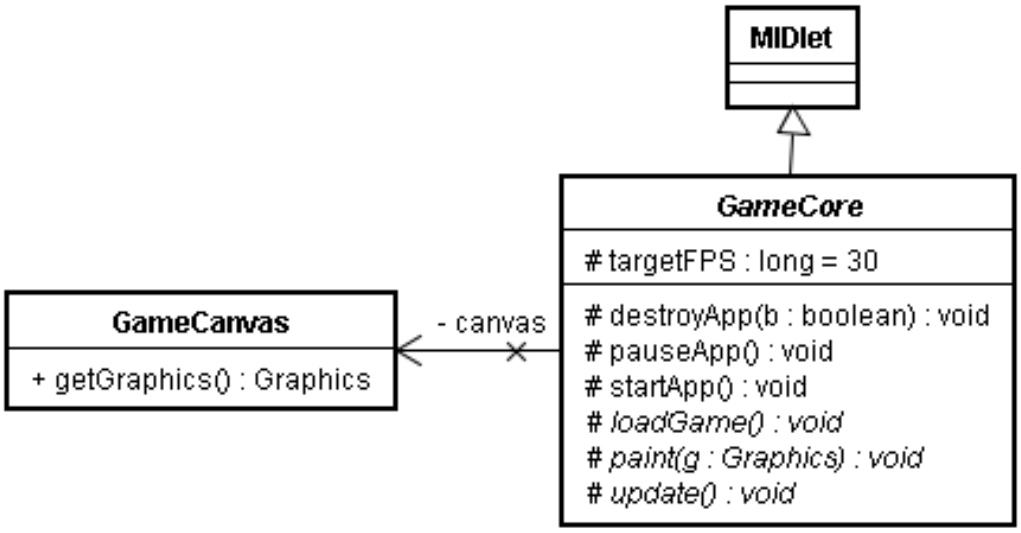


Diagrama 8 – Diagrama de classes do módulo *game.core*

4.1.2 Gerenciamento de cenas

A estrutura de um jogo é composta por cenas, cada qual com sua função, elementos, ações e eventos. Como exemplo, podemos citar o menu principal, que aparece na maioria dos jogos, contendo botões com opções para o jogador, imagem de fundo, etc. Assim como cada fase também é uma cena, que possui cenário, personagens, eventos.

Para controlar o que é mostrado na tela e atualizado a cada momento do jogo, o *framework* deverá implementar um sistema de controle de cenas, de forma que seja possível criar novas cenas, alternar entre elas, adicionar e remover elementos quaisquer.

Estendendo os métodos do ciclo principal, a cena pode ser inserida de modo a fazer uma propagação da renderização e atualização para todos os elementos nela contidos, como demonstra o diagrama de número 9.

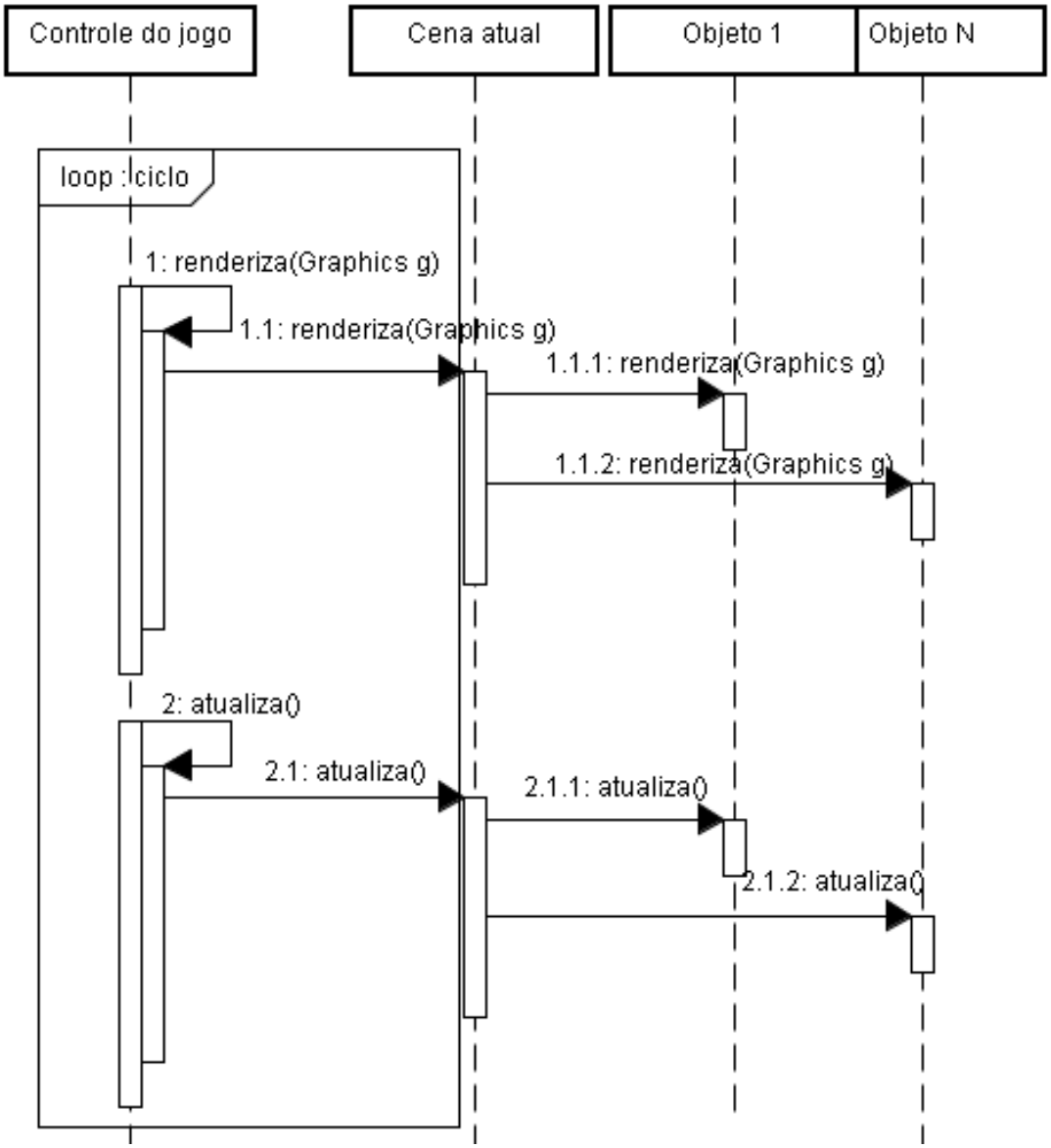


Diagrama 9 – Sequência da propagação das operações de renderização e atualização para os objetos da cena

A partir desse modelo, podemos inferir que o elemento mais básico contido em uma cena deve ao menos implementar os métodos de atualização e renderização. Esse elemento genérico servirá de base para todos os objetos que possam ser inseridos em uma cena.

Deverão ser criadas então, uma classe que represente uma cena, à qual daremos o nome de *Scene*, e uma classe que represente um elemento qualquer da cena, à qual chamaremos de *GameObject*. Ainda, para fazer o controle da troca de cenas, renderização e atualização da cena atual, deverá haver uma classe que estenda *GameCore*, adicionando essa funcionalidade. Chamaremos essa classe de *Game*.

O controle de troca das cenas deverá ser feito sobre uma pilha, sendo a cena atual aquela que estiver no topo. Essa abordagem facilita o controle do fluxo das telas do aplicativo, podendo

acomodar várias cenas subsequentes e retornar às anteriores simplesmente adicionando-as e removendo-as da pilha.

Porém, quando uma cena é adicionada, os objetos de todas as cenas abaixo dela podem permanecer na memória do aparelho, mesmo que não estejam sendo utilizados. Por isso, deverá haver um controle sobre os recursos utilizados por cada cena. Então serão inseridos na classe *Scene* métodos para carregamento, para quando se tornar ativa, e descarregamento de recursos, para quando se tornar inativa.

A esse pacote será dado o nome de *game.scene*. O diagrama 10 mostra o diagrama de classes definido.

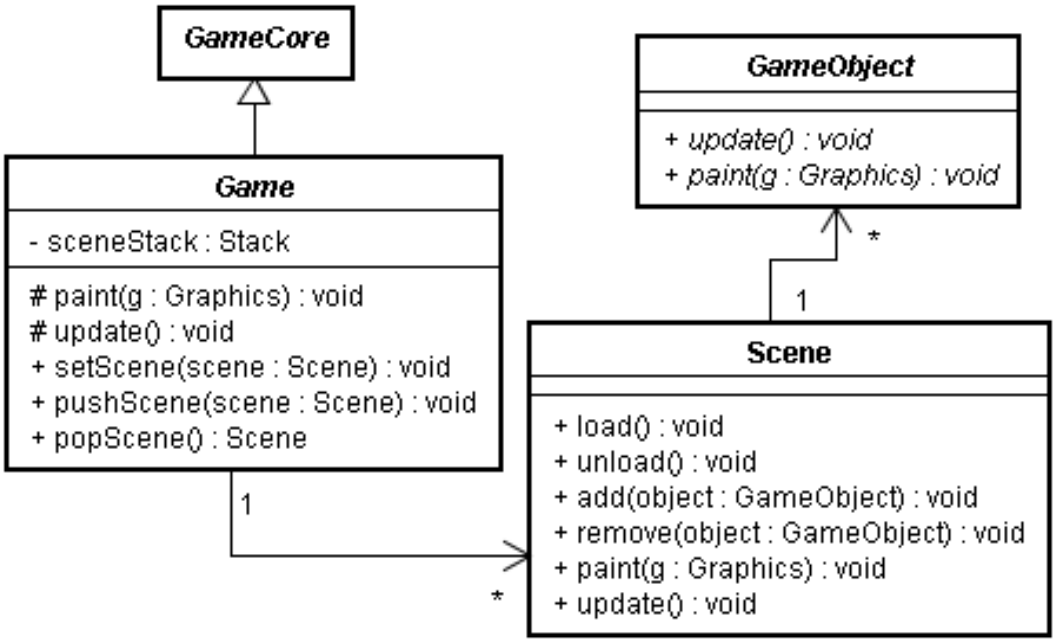


Diagrama 10 – Diagrama de classes do pacote de gerenciamento de cenas

4.1.3 Componentes de interface

Para realizar a interação com os usuários, vários objetos de interface são comumente utilizados, como botões, caixas de texto, listas, etc.

A biblioteca de Java para dispositivos móveis disponibiliza vários desses componentes de interface, porém, de forma não customizável, que adquirem automaticamente a aparência do sistema do aparelho.

Jogos geralmente possuem interface própria personalizada, com menus internos e opções relacionadas ao jogo. Por isso normalmente as desenvolvedoras de jogos acabam por construir também seu próprio *framework*.

Uma arquitetura conhecida para desenvolvimento de interfaces é a que utiliza componentes e contêineres. Um componente seria uma abstração de todos os elementos da interface, com atributos e métodos genéricos comuns. Contêineres são regiões delimitadas que podem

conter componentes, como uma barra de menu com vários botões, ou um painel com informações e imagens descritivas.

As funcionalidades básicas que esses componentes devem implementar são: alinhamento, controle de foco, navegação e seleção.

A função de alinhamento tem como objetivo posicionar os componentes de forma organizada na tela, para apresentar uma interface coerente e intuitiva para o usuário.

A funcionalidade de controle de foco tem como objetivo mostrar ao usuário com qual componente está interagindo no momento e permitir a navegação entre os componentes que possam obter o foco.

As funções de navegação e seleção são as que permitem ao usuário interagir com o sistema, navegando através dos componentes da interface e selecionando-os.

O *framework* deverá então contemplar um pacote de componentes gráficos básicos, como botões, listas, painéis e indicadores, que implementem as funcionalidades aqui definidas e possam ser personalizados de acordo com o jogo.

Para os fins acadêmicos desse trabalho, serão implementados apenas os componentes necessários para a interface dos jogos propostos, são eles: botões, imagens, *labels*, caixas de entrada de texto e contêineres.

A princípio, deverá ser criada uma classe abstrata que estenda *GameObject* e implemente as funcionalidades comuns a todos os componentes da interface. A essa classe daremos o nome de *Component*.

Component

Deverão ser criados três métodos para alinhamento de componentes: um que utilizará como referência a tela do aparelho, outro que utilizará como referência outros componentes da interface e outros que utilizará como referência o contêiner no qual o componente estiver contido. As posições definidas para o alinhamento por referência serão: acima, à esquerda, à direita, embaixo, no centro vertical e no centro horizontal.

A classe *Component* deverá possuir atributos que definam sua posição na tela e também sua largura e altura, que podem ser utilizadas como referência para o alinhamento de outros componentes. Como o desenho do componente será definido apenas em sua especialização, deverão ser definidos métodos abstratos para obter a largura e altura.

Um componente não terá controle sobre o foco de outros componentes, portanto, sua responsabilidade limita-se em alterar sua própria aparência e ações de acordo com o seu foco, e em fornecer métodos para ativar ou desativá-lo. Portanto, deverá ser adicionado à classe *Component* um atributo *boolean* que definirá se o componente possui ou não o foco.

Há ainda componentes que não apresentam interação nenhuma, como é o caso de imagens e *labels*, portanto, será adicionado à classe *Component* um *boolean* que deverá definir se o componente pode receber o foco ou não.

O diagrama 11 mostra um diagrama da classe *Component*.

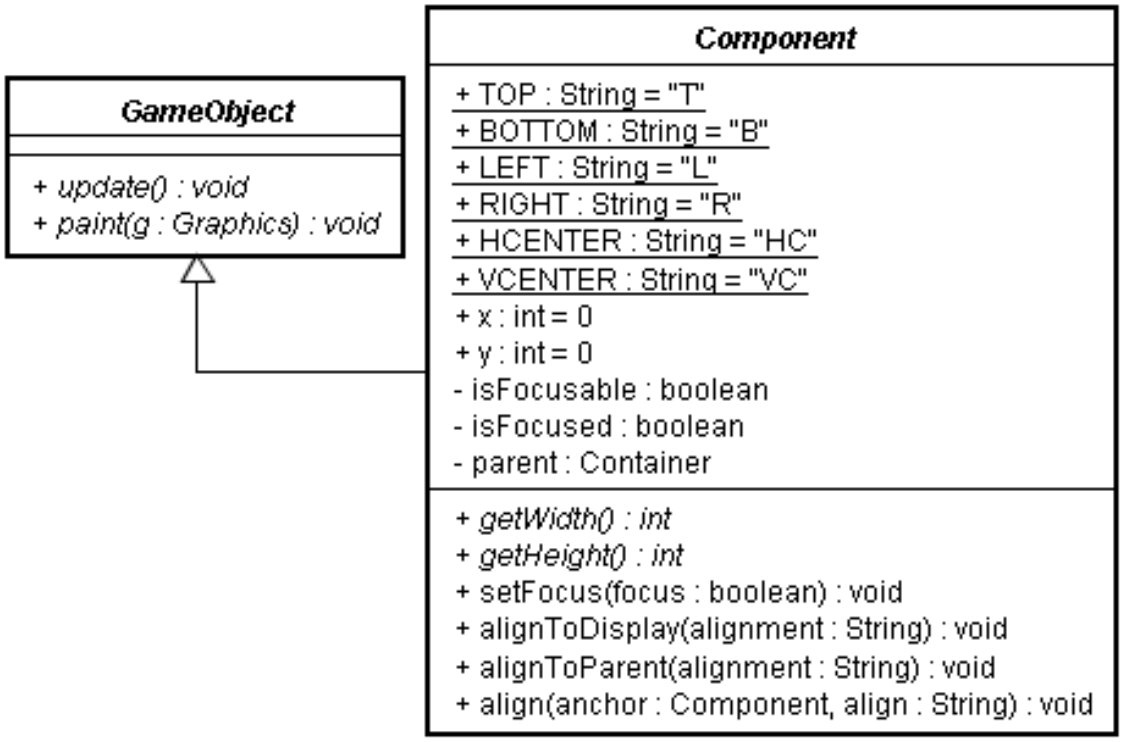


Diagrama 11 – Diagrama da classe *Component*

A responsabilidade sobre o controle de foco e navegação entre componentes deverá ser atribuída aos contêineres, definidos como uma região retangular que pode conter vários componentes. À classe que representa esse objeto daremos o nome de *Container*.

Container

Um contêiner também é um tipo de componente, podendo ganhar foco e fazer alinhamento, portanto, estenderá a classe *Component*.

O contêiner deverá ser capaz de conter objetos do tipo *Component*, e controlar a navegação e o foco dos componentes que contiver. Para isso deverão ser incluídos na classe *Container* métodos para adição e remoção de componentes e alteração de foco, de acordo com os botões de navegação pressionados pelo usuário.

O diagrama 12 mostra um diagrama de sequência, exemplificando a ação de troca de foco durante a navegação do usuário.

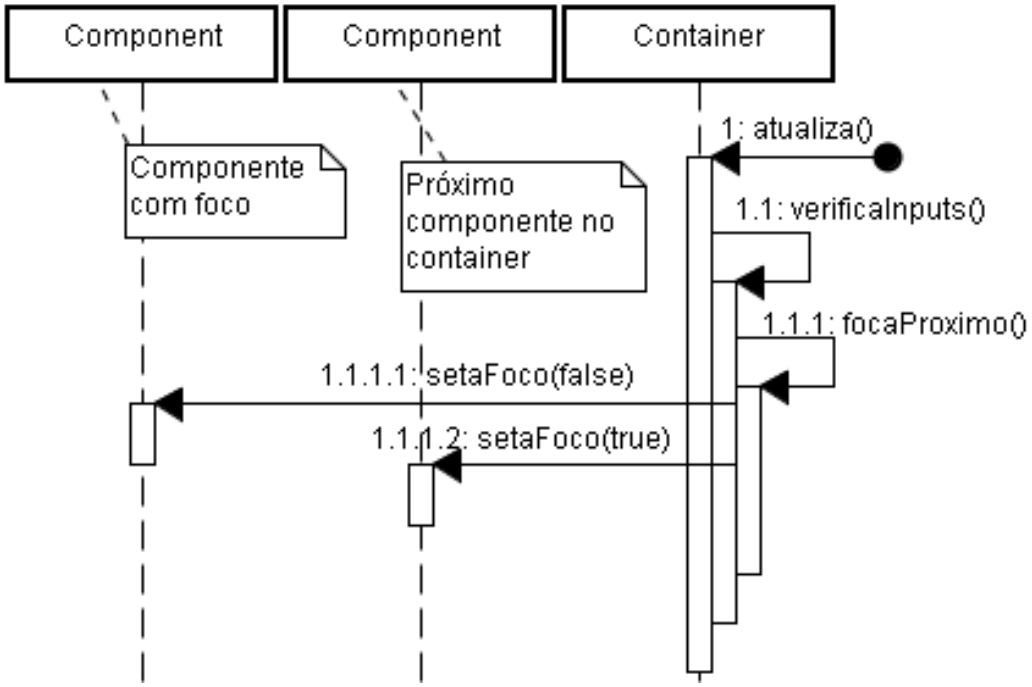


Diagrama 12 – Controle de foco na classe *Container*

O contêiner deverá também repassar as invocações dos métodos de renderização e atualização a todos os componentes que contiver. Porém, a navegação e seleção deverão ser tratadas apenas quando o contêiner detiver o foco.

O diagrama 13 mostra um diagrama da classe *Container*.

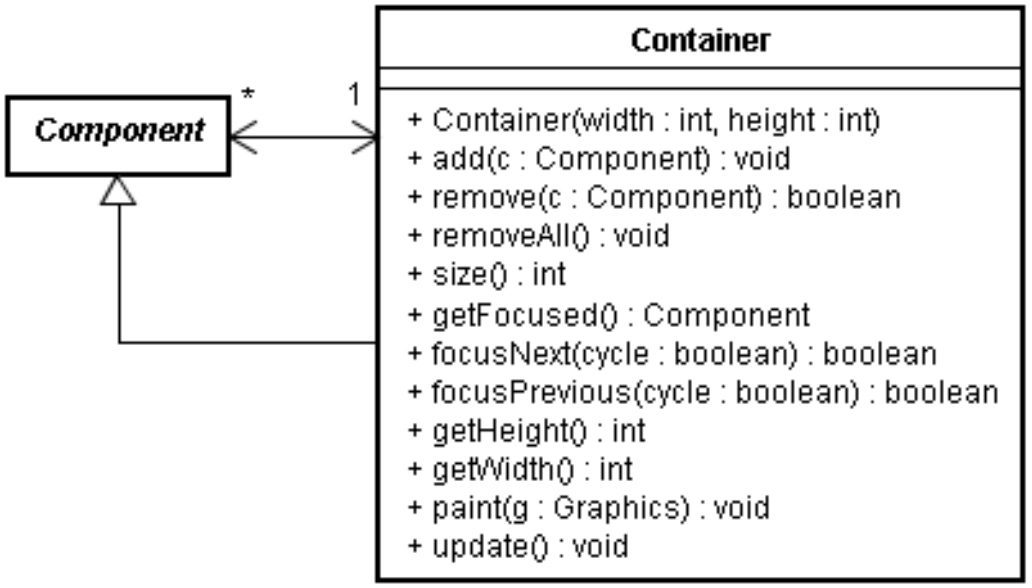


Diagrama 13 – Diagrama da classe *Container*

A seguir serão apresentadas as especificações dos componente restantes, especializações da classe *Component*.

Button, SimpleImage, Label e InputLabel

A classe *Button* deverá representar um botão, cuja função será executar uma ação caso a tecla de seleção seja pressionada enquanto ele possuir o foco. O desenho do botão consistirá apenas em um retângulo e texto, cujas cores se alteram quando o botão tem o foco.

A classe *SimpleImage* permitirá a adição de imagens à interface. Deverá ser utilizada quando for preciso inserir ícones, logos ou imagens de fundo à interface. Não pode obter o foco e não possui nenhuma ação relacionada.

A classe *Label* será utilizada para apresentação de texto, descritivo ou informativo, em mensagens, menus, títulos, etc. Também não pode obter o foco.

A classe *InputLabel* tem como objetivo a inserção de texto pelo usuário. Pode receber o foco, permitindo ao usuário realizar a entrada de texto.

O pacote contendo todos os componentes será chamado de *game.scene.*gui. O diagrama 14 mostra um diagrama de classes simplificado desse pacote.

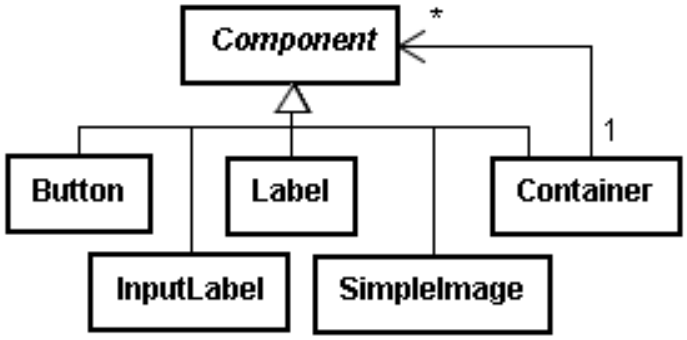


Diagrama 14 – Diagrama de classes do pacote de componentes

4.1.4 Gerenciamento de eventos

Durante um jogo vários elementos atuam ao mesmo tempo, alterando seus estados de acordo com entradas do usuário ou pela própria dinâmica do jogo. O gerenciamento de eventos é uma funcionalidade que visa facilitar o controle sobre as ações que devem definir o fluxo da interação e continuidade do jogo, sem a necessidade de uma relação direta entre o objeto que disparou o evento e o objeto que deverá tratá-lo.

Será apresentada uma proposta de arquitetura simplificada para esse sistema, onde cada evento possui apenas um atributo que definirá o tipo do evento, e qualquer elemento do jogo pode tornar-se ouvinte de um determinado tipo de evento, implementando a interface de ouvinte e registrando-se com o gerenciador de eventos. Qualquer elemento também pode disparar um evento através do gerenciador, e todos os ouvintes daquele tipo de eventos serão notificados.

Deverá então ser criada uma classe que sirva de base para todos os eventos. Possuirá apenas um atributo, do tipo inteiro, que definirá o tipo de evento que o objeto representa. A essa classe daremos o nome de *Event*.

Para o ouvinte de eventos, deverá ser implementada uma *interface* com apenas um método, que receberá o evento disparado como parâmetro. A essa *interface* daremos o nome de *EventListener*.

Como peça principal dessa funcionalidade, será implementado um gerenciador de eventos, através do qual deverá ser possível disparar eventos e registrar-se como ouvinte. À classe criada com esse propósito será dado o nome de *EventManager*.

Para que funcione de forma simples e centralizada, todos os métodos da classe *EventManager* deverão ser disponibilizados de forma estática.

O diagrama 15 exemplifica o funcionamento desse módulo:

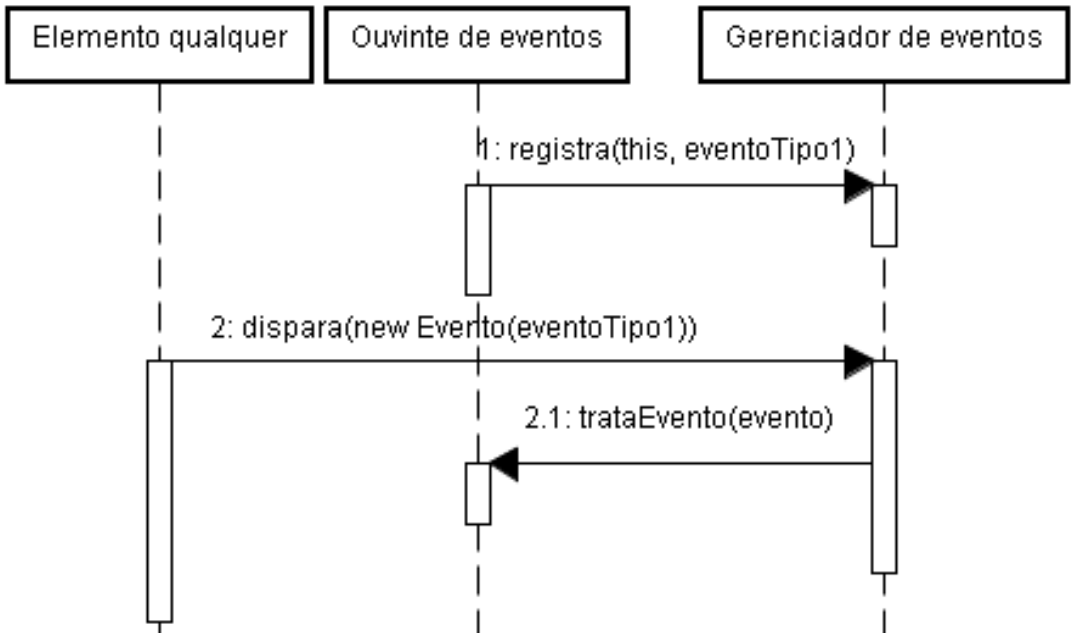


Diagrama 15 – Diagrama de sequência do disparo de um evento

O diagrama 16 mostra um diagrama de classes do pacote de gerenciamento de eventos.

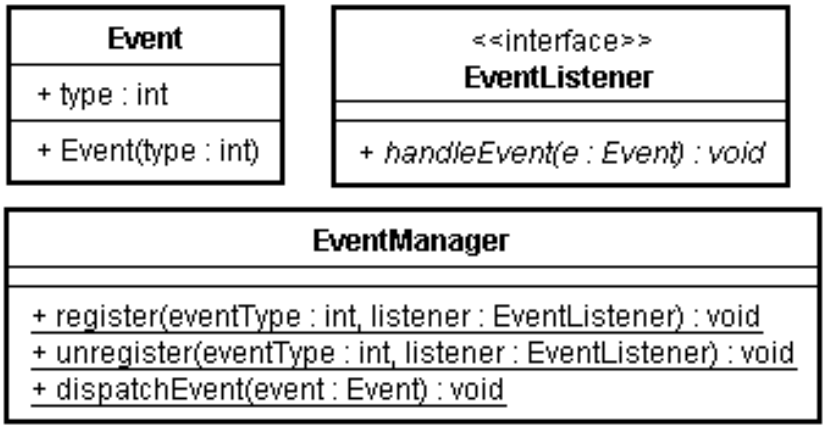


Diagrama 16 –Diagrama de classes do módulo de gerenciamento de eventos

A classe *Event* poderá também ser especializada para prover mais funcionalidades, como será o caso nesse projeto, onde o gerenciamento de eventos será utilizado para a distribuição de mensagens recebidas da rede.

4.1.5 Suporte à comunicação através da rede

Para realizar a comunicação com o servidor o jogo deverá ser capaz de estabelecer uma conexão e, de acordo com o protocolo definido, montar, ler, enviar e receber mensagens.

A biblioteca J2ME fornece suporte à comunicação utilizando os protocolos TCP, HTTP e UDP, através do pacote *javax.microedition.io*.

O protocolo UDP não será utilizado devido à sua baixa confiabilidade, o que, associado à instabilidade da rede móvel, implicaria na necessidade de implementação de funções de controle já existentes nos outros protocolos.

O protocolo HTTP pode ser facilmente implementado, com as vantagens de permitir a integração com outros sitemas que também implementem esse protocolo, como por exemplo servidores *web*.

Porém, para as sessões de jogo desse trabalho o protocolo utilizado será o TCP, devido à sua confiabilidade e tamanho, pois apenas pequenas sequências de *bytes* serão trafegadas.

Essa decisão não impede que o protocolo HTTP seja utilizado para outras funções, como integração com redes sociais e envio de informações, tais como pontuação do jogador, comentários, etc. Porém essas funcionalidades não serão abordadas nesse trabalho.

Será também definido como requisito que as mensagens da rede sejam recebidas através do gerenciador de eventos, de forma que os elementos do jogo possam tratá-las apenas implementando a *interface EventListener*. Assim, durante o jogo serão tratados objetos do tipo *Event* ao invés de pacotes de *bytes*.

Deverá então ser implementada uma especialização da classe *Event*, que inclua atributos básicos que possam ser utilizados nos jogos, como *Strings* ou inteiros. A essa extensão daremos o nome de *Message*. Contudo, para a correta interpretação dos *bytes* será definido um pequeno protocolo, que corresponderá à estrutura dos objetos do tipo *Message*, com o seguinte formato:

[tipo do evento: *int*] [tamanho do pacote de dados: *int*] [pacote de dados: *byte*[]]

Deverá ser criada, também, uma classe que possa realizar uma conexão, envio e recepção de objetos do tipo *Message*, via TCP, através da internet. A essa classe daremos o nome de *MessageClient*.

O diagrama 17 exemplifica a integração entre o módulo de comunicação e o de gerenciamento de eventos.

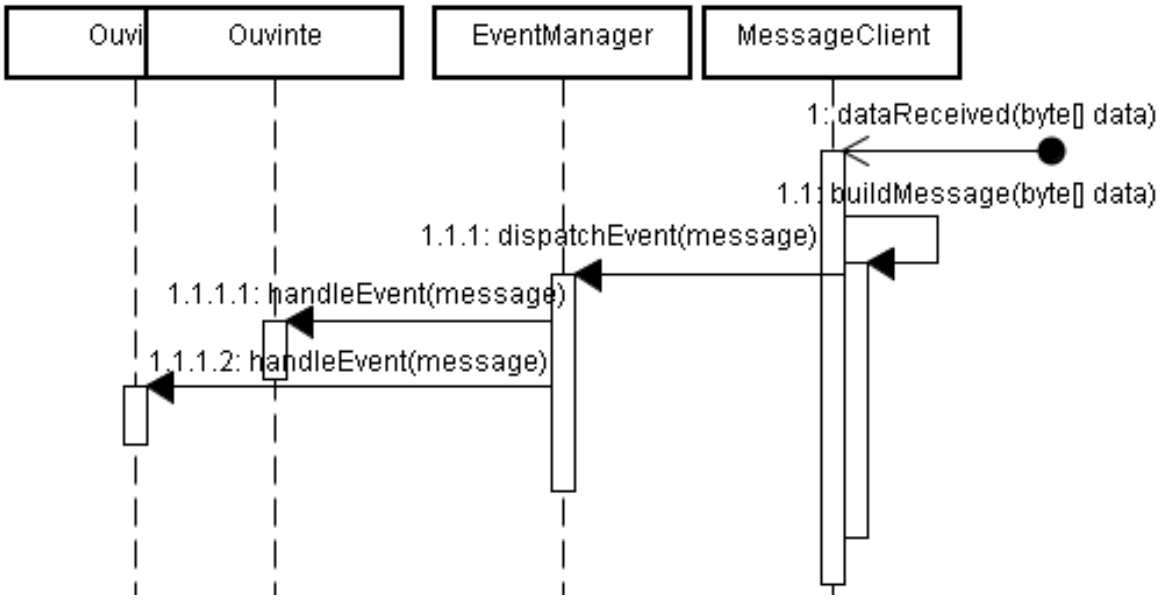


Diagrama 17 – Integração entre os módulos de comunicação e eventos

O diagrama 18 mostra o diagrama de classes do pacote de comunicação.

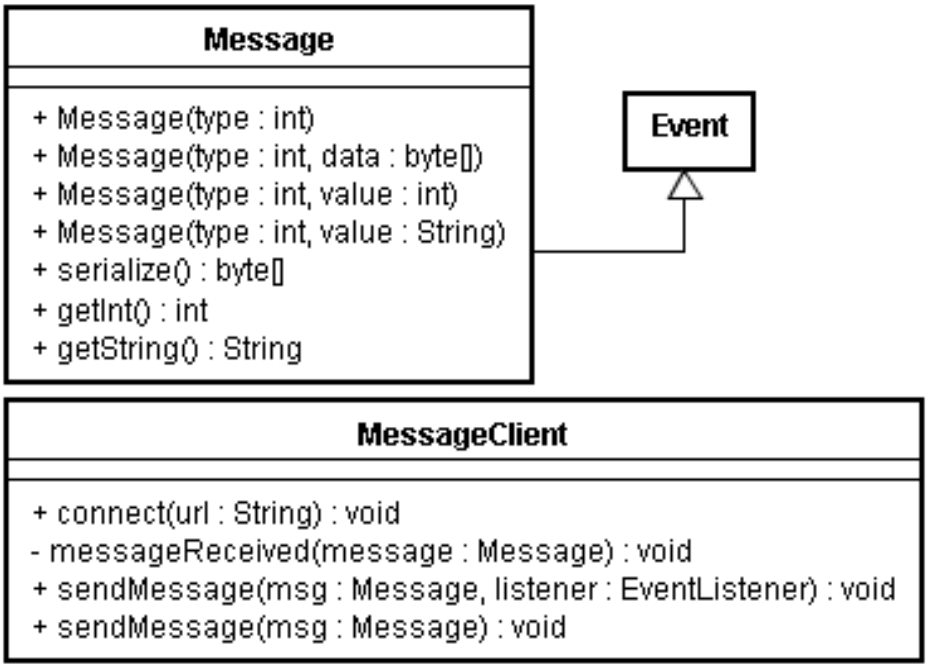


Diagrama 18 – Diagrama de classes do pacote de comunicação

A seguir, o diagrama de classes simplificado de todo o *framework*:

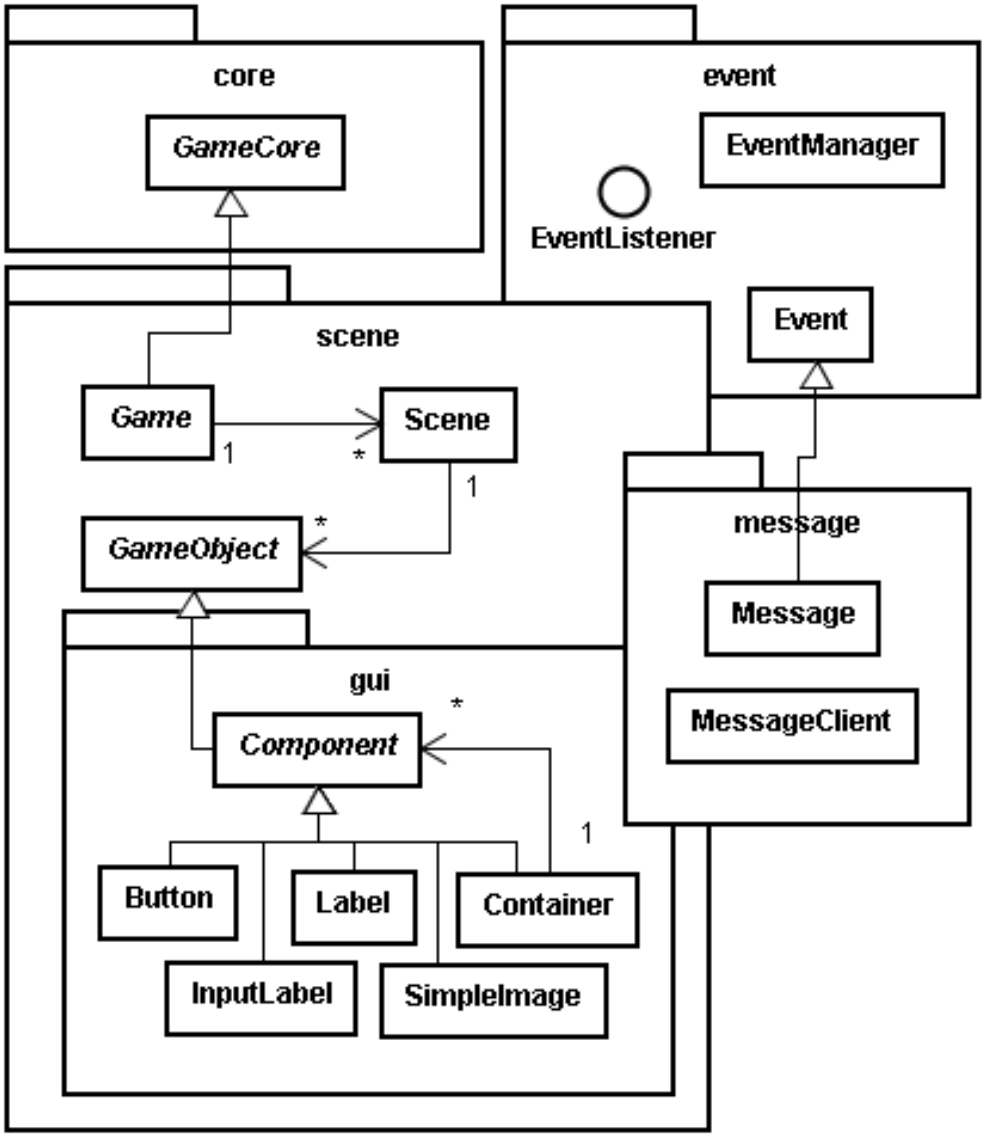


Diagrama 19 – Diagrama de classes simplificado do *framework*.

4.2 *Framework* para o servidor multi-jogadores

Completando o sistema proposto nesse projeto, um *framework* para implementação de servidores para jogos simples multi-jogadores também será implementado.

O servidor será responsável principalmente por criar e gerenciar conexões, trocar mensagens com os clientes e processar a lógica do jogo. E, de acordo com o tipo de jogo, deverá também montar ambientes virtuais para interação entre os jogadores, como salas de encontro e sessões de jogo, além de fazer a persistência de dados quando necessário. O diagrama 20 mostra os casos de uso relacionados ao servidor.

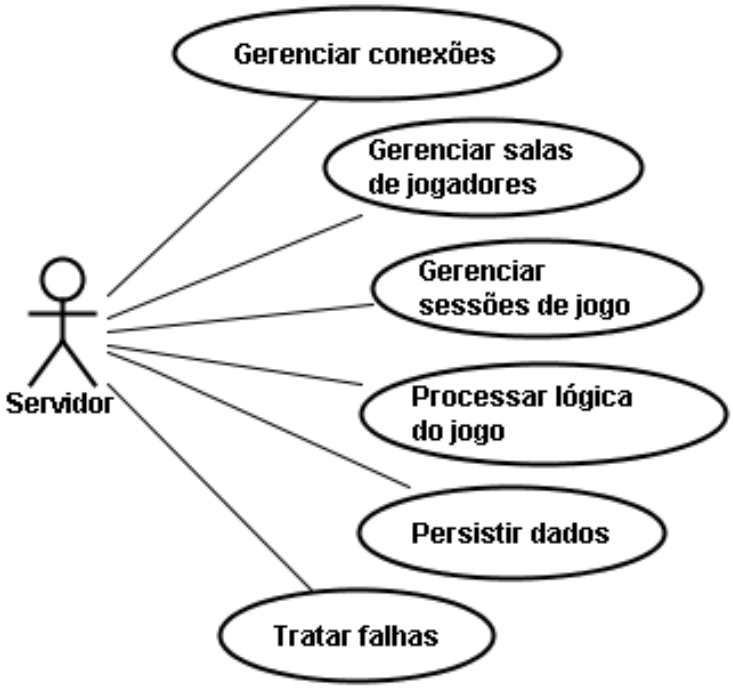


Diagrama 20 – Casos de uso do servidor.

Jogos complexos com muitos jogadores geralmente utilizam servidores de grande capacidade, que processam o estado de todos os jogadores ao mesmo tempo para manter a consistência do jogo.

Já em jogos simples é possível atribuir a lógica como sendo responsabilidade dos clientes, enquanto o servidor faz apenas a interface entre eles, encaminhando as mensagens de um para outro. Essa abordagem traz a vantagem de permitir vários jogos diferentes funcionando através de um mesmo servidor.

Para demonstrar um caso de aplicação da primeira abordagem, nesse trabalho o servidor ficará responsável pela lógica do jogo.

Podemos ainda atender aos requisitos definidos para o servidor de forma independente do protocolo de comunicação, para que o mesmo modelo possa ser implantado utilizando outros protocolos no futuro, como mostra o item a seguir.

4.2.1 Arquitetura do servidor

Dadas as responsabilidades do servidor, será definida uma arquitetura com funcionalidades comuns ao tipo de jogos que esse trabalho propõe, que depois será estendida para atender aos objetivos específicos de cada jogo.

Serão consideradas comuns as funcionalidades de gerenciamento de conexões, ambientes virtuais e troca de mensagens.

Quando uma conexão for estabelecida, o servidor deverá manter um objeto capaz de se comunicar com o cliente. Deverá ser implementada então uma classe abstrata que represente uma conexão com o cliente, provendo métodos para envio e recepção de mensagens. A essa classe daremos o nome de *Connection*.

Deverá ser definida também uma classe que represente uma mensagem, correspondendo diretamente à mensagem utilizada pelo *framework* no lado cliente. A essa classe daremos o nome de *Message*.

O servidor deverá também implementar o mesmo protocolo de aplicação definido para o cliente, que corresponde ao formato das mensagens trocadas através da rede. Essas mensagens serão utilizadas para transmitir as ações do jogo, tais como: pedido de início de jogo, envio de jogada, envio de mensagens de texto, desconexão, entre outros.

O diagrama 21 a seguir mostra um diagrama das classes *Message*, *Connection* e *ConnectionGroup.*

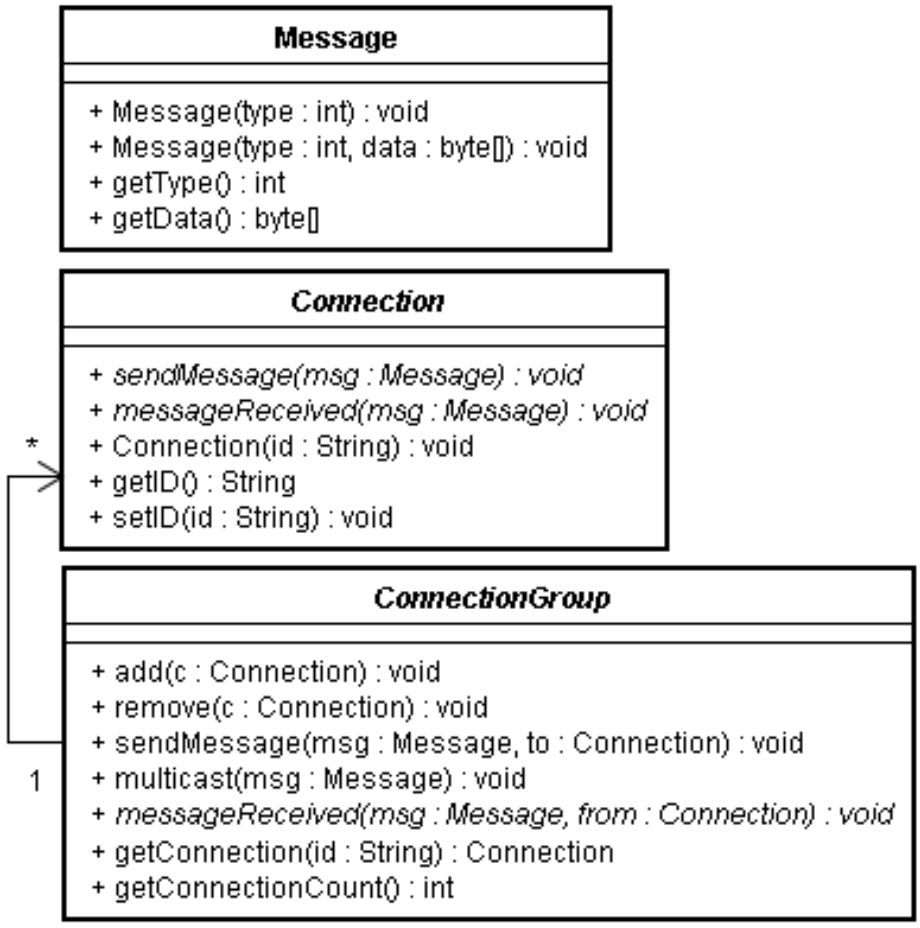


Diagrama 21 – Diagrama das classes genéricas do servidor

Para os jogos em que há mais de um jogador interagindo em um mesmo ambiente, deverá existir uma classe responsável por controlar um grupo de objetos do tipo *Connection* e trocar mensagens com todos eles. Chamaremos essa classe de *ConnectionGroup*. A partir dessa classe podemos conceber os ambientes virtuais previstos, como salas de encontro e sessões ou áreas de jogo, e incluir funções específicas a eles.

Estendendo a classe *ConnectionGroup*, será implementado um ambiente do tipo sala de encontro, onde os jogadores possam desafiar uns aos outros para uma partida. A essa classe daremos o nome de *Lobby*.

A classe *Lobby* deverá tratar automaticamente mensagens de pedido de jogo, pedido de jogo aceito e pedido de jogo recusado, bem como atualizar a lista com os jogadores que entram e saem da sala, notificando a todos da alteração.

A sessão de jogo criada quando um pedido de jogo for aceito, deverá também ser uma especialização da classe *ConnectionGroup*, à qual daremos o nome de *GameSession*. A classe *GameSession* deverá representar uma partida entre dois ou mais jogadores e ficará responsável por tratar a lógica do jogo.

O diagrama 22 mostra o diagrama das classes *Lobby* e *GameSession*.

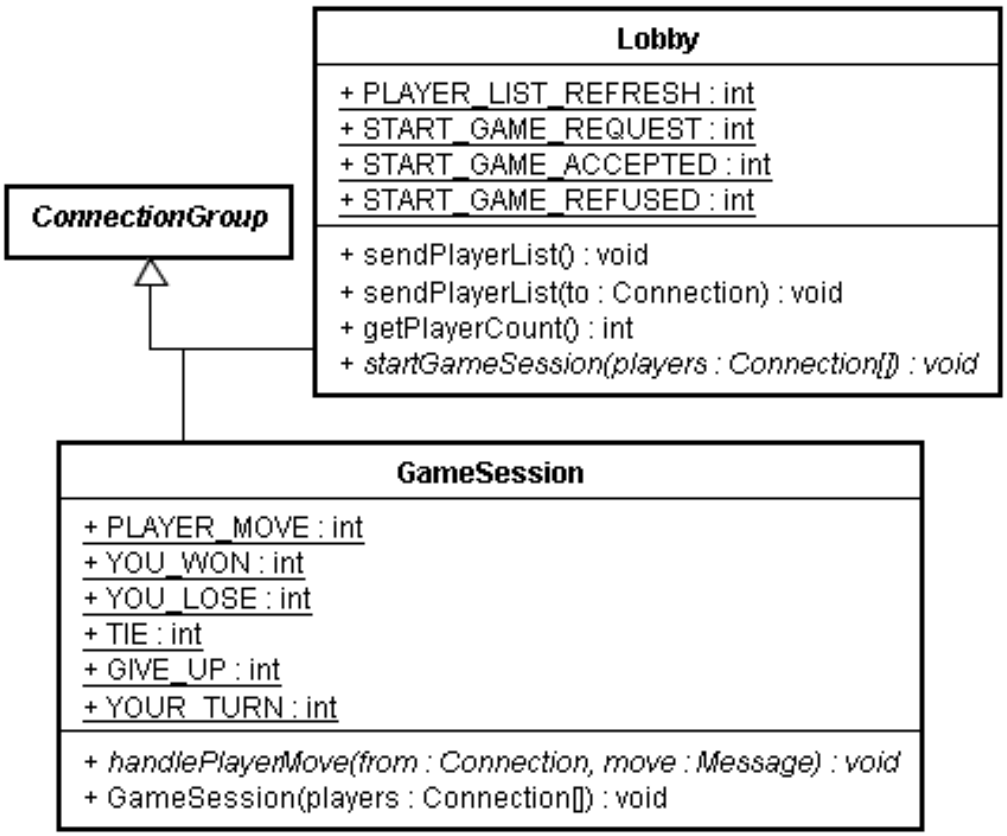


Diagrama 22 – Diagrama de classes do servidor.

4.2.2 Estabelecendo a conexão TCP

A partir dessa arquitetura genérica, o primeiro passo para o estabelecimento da conexão TCP é a criação de uma classe responsável por receber o pedido de conexão via *socket* e direcioná-lo de acordo com a dinâmica do jogo. Essa classe será chamada de *SocketServer*.

Deverá ser criada também uma especialização da classe *Connection*, que receberá o nome de *SocketConnection*, e deverá implementar o envio e recebimento das mensagens através de *sockets*.

O diagrama de classes dessa extensão é apresentado no diagrama 23.

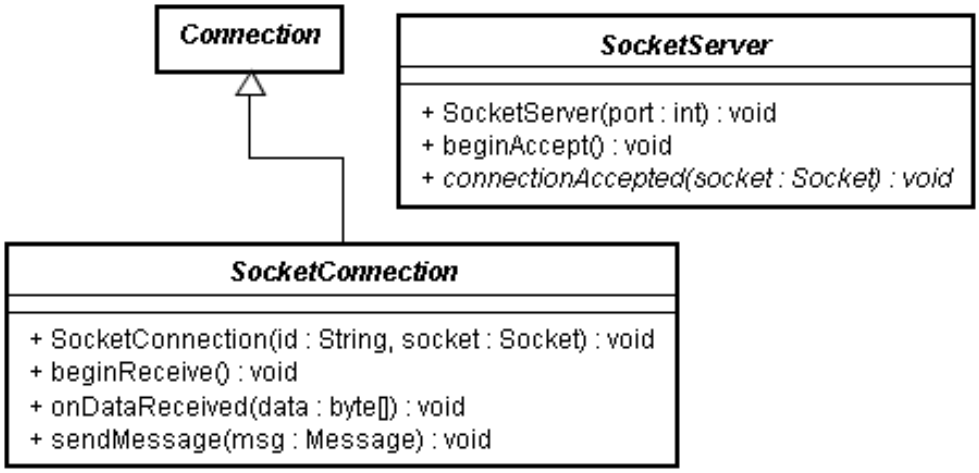


Diagrama 23 – Diagrama de classes da implementação do servidor utilizando *sockets.*

4.3 Jan Ken Po

Como caso de uso de jogo multi-jogadores com interação baseada em turnos, o jogo escolhido foi o "Jan Ken Po", ou como é chamado no Brasil: "papel, tesoura e pedra". A escolha desse jogo baseia-se no fato de que é um jogo simples, porém divertido, com um nível muito baixo de complexidade para um jogo multi-jogadores, pois cada jogada consiste em apenas uma variável para cada jogador.

As regras do jogo são as seguintes: duas pessoas jogam. Com uma das mãos previamente escondida, cada jogador deve escolher um objeto entre papel, pedra ou tesoura. Quando prontos, dá-se um sinal e ao mesmo tempo ambos devem mostrar as mãos no formato do objeto escolhido. Papel ganha de pedra, pedra ganha de tesoura e tesoura ganha de papel, como ilustra a figura 17.

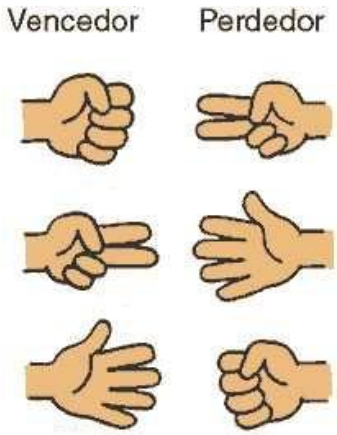


Figura 17 – Exemplificação das regras do *Jan Ken Po.*

A versão *online* para celulares do jogo *Jan Ken Po*, consistirá, primeiramente, em uma sala de jogadores no servidor, onde poderão desafiar uns aos outros para uma partida.

Logo no início do jogo, para efetuar a conexão com o servidor, o jogador deverá informar um apelido, que será utilizado para sua identificação no servidor e incluído na lista da sala de jogadores.

A partir dessa lista os jogadores deverão convidar um dos outros jogadores disponíveis para uma partida. Quando um desafio for aceito, o servidor notificará os dois jogadores de que será iniciada uma sessão de jogo. Os jogadores serão então transferidos para o ambiente responsável pela dinâmica do jogo, onde cada um deverá efetuar uma jogada.

Após o envio das jogadas, o servidor deverá calcular o resultado e comunicar os jogadores, que poderão optar por continuar o jogo ou voltar à sala de jogadores.

Portanto, as telas que o jogo deverá apresentar serão: uma tela para o menu principal, uma tela para seleção de apelido e conexão ao servidor, uma tela para a sala de jogadores, e uma tela para a sessão de jogo.

O servidor deverá implementar uma sala de jogadores e gerenciar as sessões de jogo à medida que forem criadas.

4.3.1 Especificações do cliente

Será preciso estender a classe *Game*, que controla o ciclo do jogo e gerenciamento de cenas, e selecionar a primeira cena do jogo. À essa extensão daremos o nome de *JanKenPo*.

A primeira cena será o menu principal, que deverá conter apenas dois botões, um para iniciar e outro para sair do jogo. Para isso deverão ser utilizados a classe *Scene* e componentes de interface. À classe que representa essa cena será dado o nome de *MainMenu*.

A tela seguinte ao menu principal será a tela de seleção de apelido e conexão, onde o jogador deverá inserir um apelido para acessar a sala de jogadores do servidor. Nessa cena deverá ser utilizado um objeto do tipo *InputLabel*, para que o usuário possa inserir um apelido, e botões para efetuar uma conexão ou voltar ao menu principal. À essa classe daremos o nome de *NickSelection*.

Para conexão e troca de mensagens com o servidor, será criada uma classe *JKPClient*, que encapsulará uma instância de *MessageClient*, disponibilizando métodos estáticos para o envio e recepção de mensagens no cliente.

Após a conexão, espera-se que o servidor envie uma lista com os apelidos dos jogadores disponíveis na sala. Deverá ser implementada uma cena que trate essa mensagem, mostre a lista de jogadores e permita a seleção deles. A cena que representa a sala de jogadores no cliente será também chamada de *Lobby*.

Quando um oponente for selecionado na lista, deverá ser enviada ao servidor uma mensagem de pedido de jogo, informando o apelido do jogador selecionado. A classe *Lobby* deverá também tratar também as mensagens de pedido de jogo recebidas.

Quando um pedido de jogo for aceito, os jogadores deverão entrar na cena de jogo. A cena de jogo deverá inicialmente mostrar as opções de jogada que o jogador pode fazer, no caso, papel pedra ou tesoura. Para mostrar as opções poderá ser criada uma especialização da classe *Button*, incluindo uma imagem no lugar de texto. À classe que representa a sessão de jogo será dado o nome de *GameSession*.

Após a seleção da jogada, o jogador deverá aguardar o processamento do resultado pelo servidor. Quando ambos os jogadores realizarem suas jogadas, o servidor enviará o resultado para cada um deles e uma tela com o resultado deverá aparecer, incluindo as opções de jogar novamente ou retornar à sala de jogadores.

O diagrama 24 a seguir ilustra a proposta do fluxo do jogo:

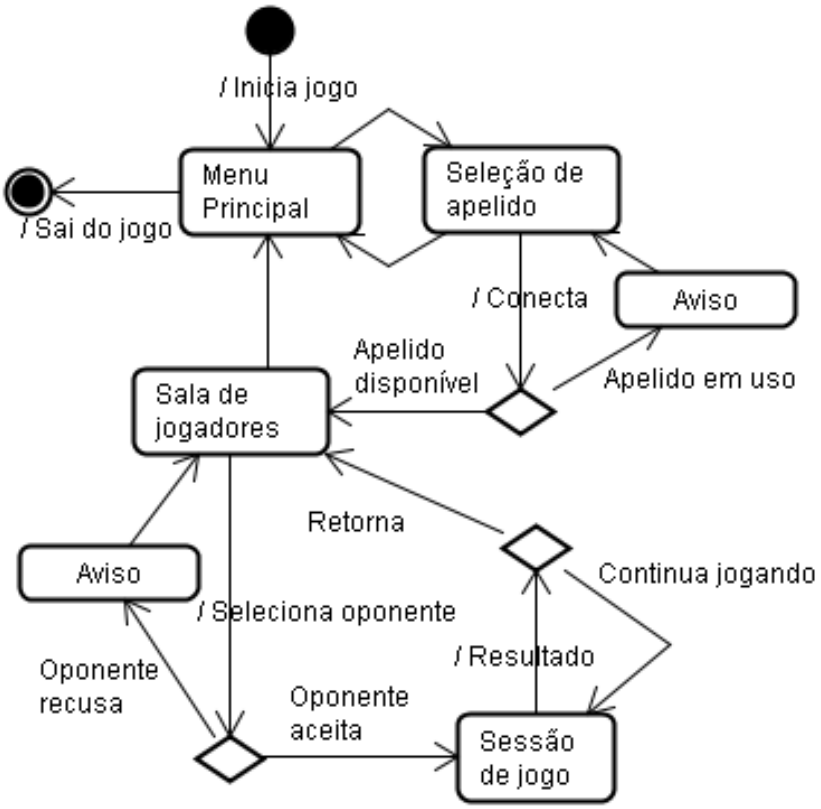


Diagrama 24 – Fluxo proposto para o jogo *Jan Ken Po*

O diagrama 25 mostra um diagrama simplificado da implementação do jogo *Jan Ken Po* a partir do *framework*.

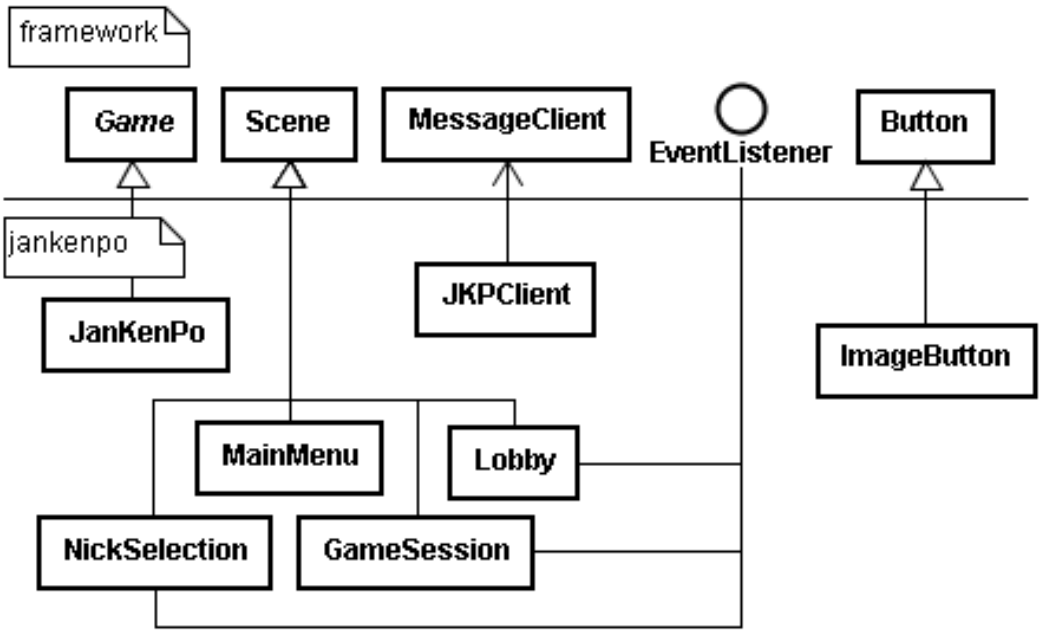


Diagrama 25 – Diagrama simplificado das classes do jogo *Jan Ken Po*

O diagrama 26 representa um diagrama de sequência da comunicação com o servidor esperada durante uma sessão do jogo.

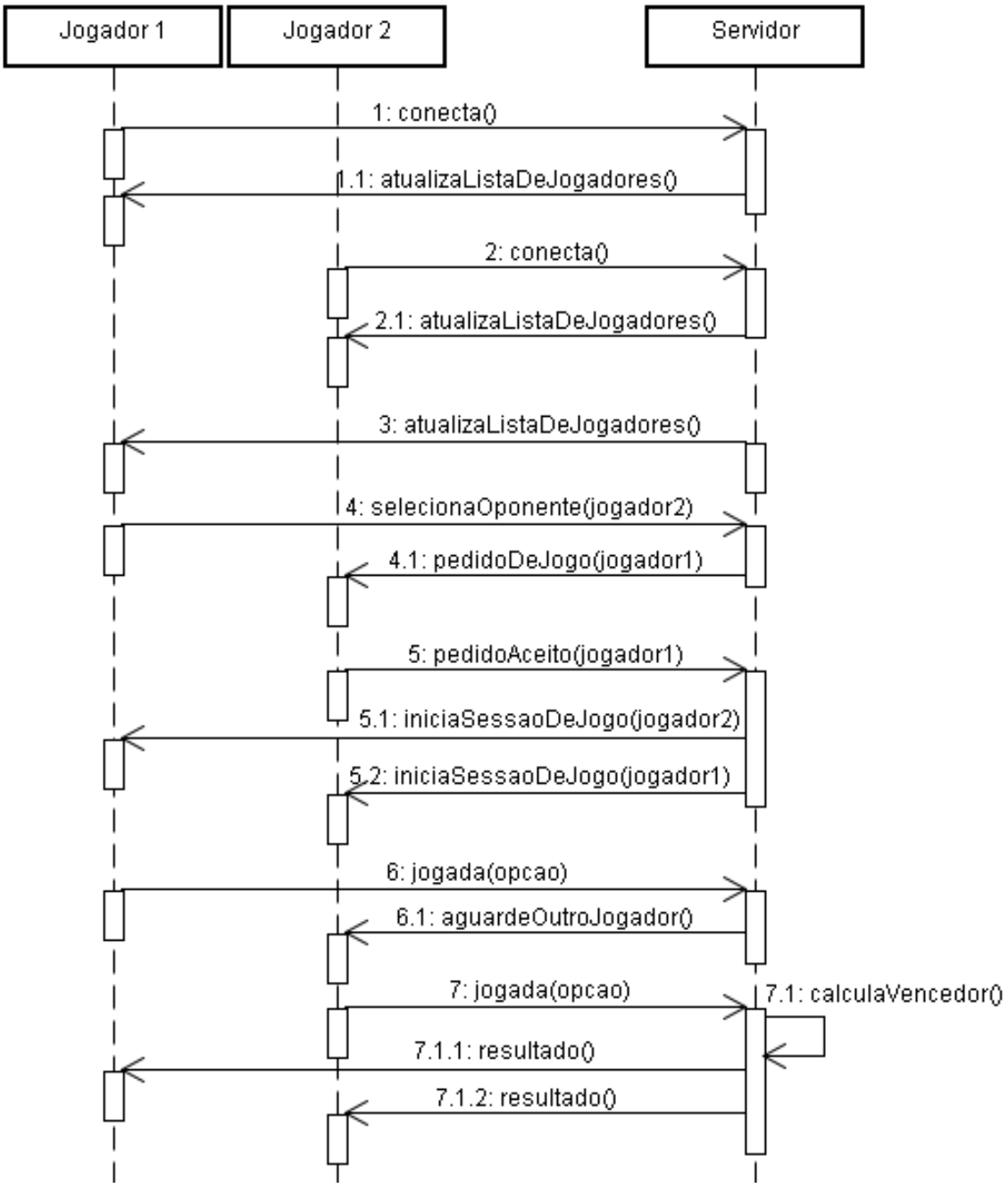


Diagrama 26 – Sequência das interações de uma sessão de jogo *Jan Ken Po.*

4.3.2 Especificações do servidor

O servidor do jogo *Jan Ken Po* deverá manter um objeto da classe *Lobby*, representando a sala de jogadores, implementar uma extensão da classe *SocketServer*, para receber as conexões e encaminha-las ao *Lobby*, e implementar uma especialização da classe *GameSession*, tratando as jogadas efetuadas em acordo com as regras do *Jan Ken Po*.

A especialização da classe *SocketServer*, deverá instanciar o objeto da sala de jogadores, e aguardar pelos pedidos de conexão. A essa classe daremos o nome de *JKPServer*. Uma vez criada a conexão, o servidor deverá esperar por uma mensagem contendo o apelido do jogador, e então verificar se o apelido está disponível na sala de jogadores. Em caso positivo, deverá passar o controle da conexão ao *Lobby*, ou notificar o jogador de que o apelido está em uso.

Como as funções de pedido de jogo já são implementadas na classe *Lobby*, a unica função da sala de jogadores de *Jan Ken Po* será instanciar as sessões de jogo. À essa classe daremos o nome de *JKPLobby*.

A classe que representará uma sessão de jogo será chamada de *JKPSession*. Deverá notificar os jogadores do início do turno assim que for criada, e então aguardar pelas jogadas. Como as jogadas são assíncronas, será preciso marcar cada jogada relacionando-a a seu respectivo jogador. Uma vez que amboss os jogadores tenham feito suas jogadas, o servidor deverá calcular o resultado, com base nas regras do jogo, como ilustra o diagrama de número 27.

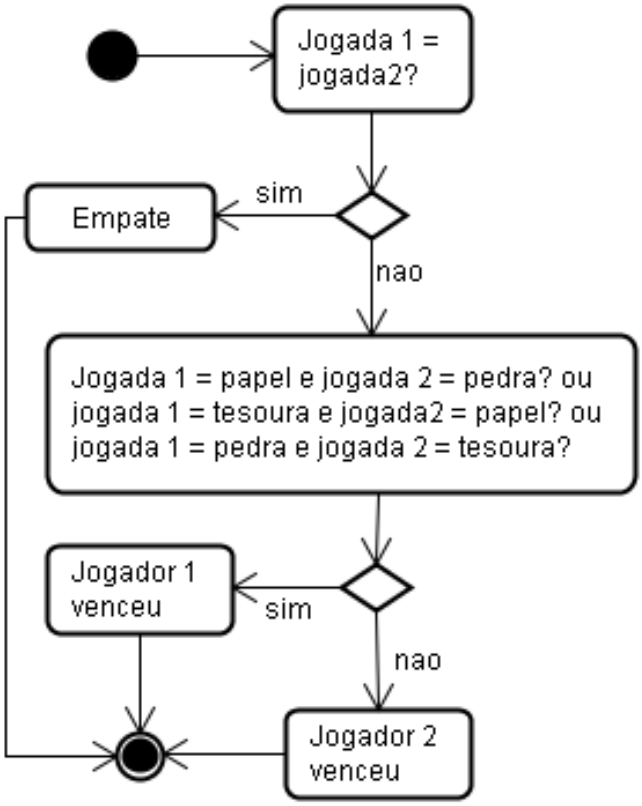


Diagrama 27 – Cálculo do resultado das jogadas na *JKPSession*

Após o envio do resultado haverá a opção de jogar novamente contra o mesmo jogador, ou retornar à sala de jogadores. Caso ambos os jogadores optem por jogar novamente, a sessão deverá enviar novamente a notificação de início de turno, para que os jogadores façam suas jogadas. Caso apenas um jogador escolha continuar o jogo, o servidor deverá notificá-lo da desistência do seu oponente.

4.4 *JobFun*

Como caso de uso de jogo com interação assíncrona será apresentado o *JobFun*, um jogo simples, do tipo social, em que os jogadores poderão escolher uma profissão e produzir objetos a partir de *minigames*.

A interação entre os jogadores se dará na possibilidade de venda e compra de objetos entre eles, uma vez que o jogador poderá produzir apenas objetos relativos à profissão escolhida. Cada jogador possuirá também um "quarto virtual", no qual poderá dispor seus objetos.

Haverá a necessidade da persistência dos dados dos usuários, como os itens disponíveis para venda, itens vendidos, créditos para compra de objetos, etc. Para isso será utilizado um banco de dados, cujas transações serão controladas a partir do servidor.

Um fator interessante nesse modo de jogo é que a transferência de dados é mínima, assim como nos jogos baseados em turnos, pois há comunicação apenas quando o estado do jogo é alterado, que consiste em apenas uma pequena quantidade de informações.

Várias outras funcionalidades poderiam ser adicionadas a esse jogo, como outras profissões, *minigames*, aumento da dificuldade de acordo com o objeto criado e a possibilidade da criação de objetos mais complexos, de acordo com a evolução do jogador dentro do jogo, e outros tipos de interação entre os jogadores. Porém, como caso de uso desse trabalho, o jogo será simplificado.

As profissões disponíveis para o jogo apresentado nesse trabalho serão marceneiro e florista. Escolhendo a profissão de marceneiro será possível criar móveis de madeira como bancos e armários. A profissão de florista trará a possibilidade da criação de vasos e arranjos com flores.

Cada profissão possuirá dois minigames que serão apresentados na construção dos objetos, utilizando tipos de jogabilidade diferentes, como sequência de teclas e *timing*.

Na venda dos objetos o jogador poderá estipular um preço à sua escolha, sendo que esse valor será revertido em créditos para que o jogador possa comprar objetos de outros jogadores. Ou seja, deverá produzir e vender objetos para poder comprar itens de outros jogadores.

O jogo deverá apresentar as seguintes cenas: Menu principal com *login* para validação do usuário, menu de seleção de profissão, cena do quarto do jogador, cena de objetos para venda e compra, cena para construção de objetos, e uma cena para cada *minigame*.

4.4.1 Especificações do cliente

No desenvolvimento desse jogo será preciso estender a classe *Game* e implementar a inicialização, na classe que chamaremos de *JobFun.* A primeira cena deverá ser o menu principal, que conterá as caixas de entrada de texto para o nome e senha do jogador. A essa classe será dado o nome de *MainMenu*.

Caso seja um novo jogador, o servidor deverá enviar uma mensagem de confirmação da criação de um novo cadastro. Caso contrário o servidor deverá validar as credenciais do usuário e enviar as informações correspondentes ao estado do jogo para esse usuário, revelando possíveis alterações no estado do jogo, como venda de objetos e alteração no saldo de créditos disponíveis.

Caso o jogador ainda não tenha selecionado uma profissão, será apresentada uma cena para essa função, à qual daremos o nome de *PickAJob*.

Para conexão e troca de mensagens com o servidor, será criada uma classe *JFClient*, que encapsulará uma instância de *MessageClient*, disponibilizando métodos estáticos para o envio e recepção de mensagens no cliente.

O jogador será então direcionado à cena do seu quarto virtual, o qual chamaremos apenas de *Room*, onde estarão expostos os objetos que o jogador possui. Essa cena será a raiz da navegação do jogo, apresentando um menu para as cenas de compra, venda e construção de objetos.

Na cena de venda será mostrada uma lista dos objetos que o jogador possui, quais deles estão à venda e por qual preço. Selecionando um dos objetos disponíveis, deve-se apresentar uma cena com detalhes e opções para alterar o valor do produto, colocá-lo à venda ou removê-lo. Uma vez colocado à venda ou removido, deve ser feita uma sincronização com os dados do servidor, para que a alteração seja visível a outros usuários.

Na cena de compra deverá haver um campo de texto para pesquisa de produtos. Quando realizada a pesquisa será mostrada uma lista com os produtos disponíveis e seu preço. Ao selecionar um dos produtos da lista, será apresentada uma cena com detalhes do produto e a opção de compra. Quando um produto for comprado, deverá ser confirmada a compra atualizando-se o banco de dados, debitando crédito do comprador, creditando ao vendedor e alterando o estado do objeto para “vendido”.

Na cena de construção de objetos o jogador deverá selecionar de uma lista um dos objetos disponíveis para construção. Será direcionado então para uma tela de *minigame* onde deverá ganhar o desafio para que o objeto seja produzido. Após produzido o objeto ficará armazenado localmente até que o jogador decida vendê-lo ou descartá-lo.

Os *minigames* para a profissão de marceneiro envolverão as atividades de serrar madeira e martelar. Para a profissão de florista, as atividades utilizadas para o minigame serão cavar a terra e podar as plantas.

4.4.2 Especificações do servidor

No servidor para jogos do tipo assíncrono não há a utilização de ambientes virtuais como salas de jogadores ou sessões de jogo. As sessões são inteiramente individuais, havendo a necessidade apenas da uma conexão e integração com banco de dados.

Na própria classe do servidor, serão tratadas as mensagens relacionadas às ações do usuário, fazendo a interação com o banco de dados. Será criada uma classe que facilitará o acesso, mantendo uma conexão com o banco de dados e provendo métodos para execução de transações.

A comunicação entre o cliente e o servidor será realizada do mesmo modo utilizado para o jogo *JanKenPo*, através de mensagens representando certos tipos de eventos. Apenas o servidor terá contato com o banco de dados quando uma ação do cliente o fizer necessário.

As informações contidas no banco de dados serão aquelas que necessitam de persistência para que seja possível a interação entre os jogadores e a manutenção do estado do jogo. Para o jogo *JobFun* essas informações são: as credenciais dos jogadores, quantidade de créditos que possuem e os objetos à venda.

Portanto deverá ser criado um banco de dados com duas tabelas, uma será chamada *Users* e conterá o nome do usuário, sua senha de acesso e quantidade de créditos. A outra será chamada *Itens* e conterá uma referência para o usuário que a gerou, o nome do item, o tipo do item, seu estado e o preço.

O diagrama 28 mostra a arquitetura do banco de dados definida.

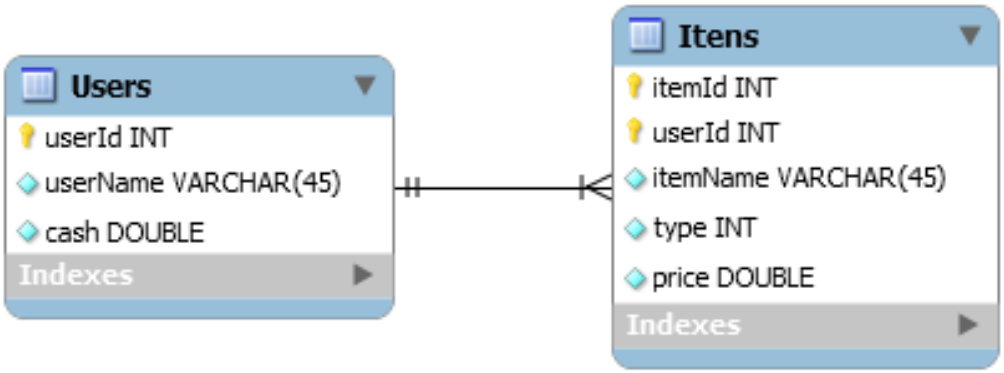


Diagrama 28 – Diagrama do banco de dados para o *JobFun*

O diagrama 29, a seguir, ilustra o comportamento esperado da interação assíncrona entre os jogadores do *JobFun* através do servidor.

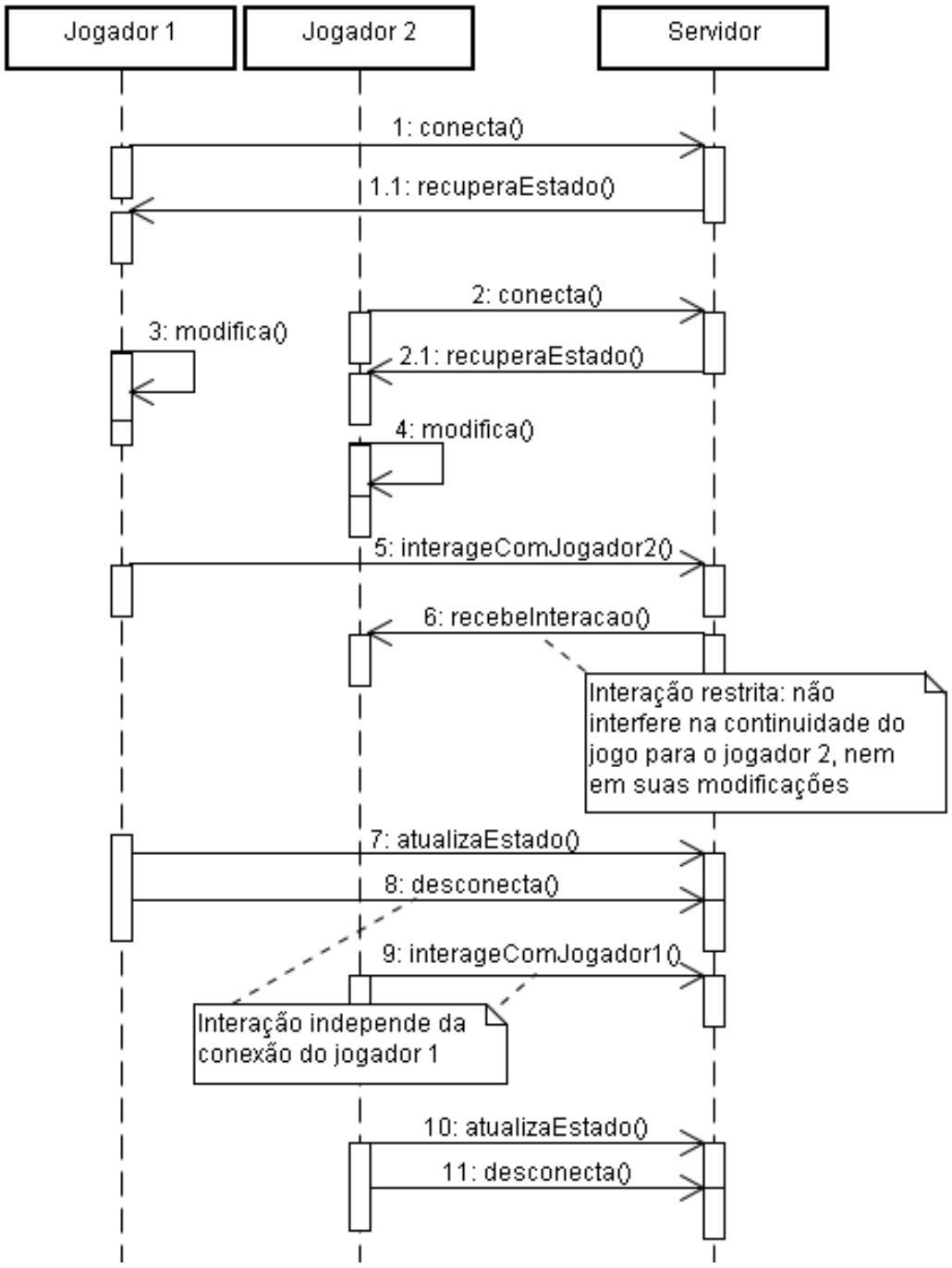


Diagrama 29 – Exemplo de interação assíncrona

5 Implementação

Esse capítulo abordará o processo de implementação do projeto definido para esse trabalho.

Não serão fornecidas definições básicas acerca das linguagens de desenvolvimento utilizadas, assumindo que o leitor já detenha prévio conhecimento sobre o assunto.

Os trechos de código apresentados nesse capítulo foram resumidos para um melhor entendimento. Para visualizar o código completo consulte os anexos.

5.1 *Framework*

Como definido no projeto, a implementação do *framework* começa pelo ciclo do jogo, em seguida será feita a implementação do controle de cenas, dos componentes de interface, gerenciamento de eventos e, por fim, suporte a múltiplos jogadores.

Para o desenvolvimento do *framework* em J2ME as ferramentas utilizadas foram o "*Eclipse IDE for Java Developers"* (The Eclipse Foundation, 2010) com o *plugin* "*EclipseME*" (EclipseME.org, 2005)**,** que fornece suporte ao desenvolvimento para dispositivos móveis. Foi utilizado também o *SDK* para dispositivos móveis da *Sun*, *Java Wireless Toolkit*, versão 2.5.2.

No desenvolvimento em Java para aparelhos celulares, deve-se definir também qual versão da biblioteca MIDP deseja-se atingir. Essa escolha determina quais funções estarão disponíveis e quais aparelhos suportarão a aplicação. Portanto, vamos utilizar a MIDP 2.0, que fornece as funcionalidades necessárias e já é bem disseminada no mercado.

O kit de desenvolvimento de aplicações móveis da *Sun* disponibiliza um simulador de aparelho celular, no qual o aplicativo em desenvolvimento pode ser testado. Porém, esse simulador não deve ser usado como parâmetro de desempenho e robustez, uma vez que reage de maneira genérica aos aplicativos, não correspondendo à realidade e variedade dos aparelhos celulares.

Devido aos fins acadêmicos desse trabalho e à restrição de acesso a diversos modelos de celulares para testes da aplicação, serão feitos testes apenas com o simulador e com o aparelho do próprio autor (*Nokia 6300*), para garantir a fidelidade mínima dos aplicativos desenvolvidos. Todas as figuras apresentadas na implementação foram tiradas do próprio simulador.

5.1.1 Ciclo do jogo

A classe principal dos aplicativos Java para dispositivos móveis é o *MIDlet*, uma classe abstrata a partir da qual temos o controle da execução da aplicação, implementando os métodos *startApp*, *pauseApp* e *destroyApp*. Portanto, a classe *GameCore*, que controla o ciclo do jogo, é implementada como uma extensão da classe *MIDlet*.

O *GameCore* implementa também a *interface Runnable*, para que o ciclo do jogo seja inserido em seu método de execução. Assim tem-se controle sobre a *Thread* que o executa sem afetar a *Thread* principal da aplicação.

Na implementação do ciclo do jogo, devem ser realizadas as operações de atualização e renderização. O método responsável pela atualização é um método vazio, que deve ser implementado no desenvolvimento dos jogos. Na etapa de renderização, é necessário um objeto gráfico capaz de desenhar os objetos do jogo na tela do aparelho. Veremos a seguir como obtê-lo.

Um aplicativo do tipo *MIDlet* pode definir qual tela será mostrada ao usuário a partir da classe *Display*, que possui controle de algumas funções básicas do aparelho. No método para seleção de tela, a classe *Display* aceita apenas objetos do tipo *Displayable*, uma classe abstrata que representa uma tela qualquer.

As classes que estendem diretamente o tipo *Displayable* são as classes *Screen* e *Canvas*. A classe *Screen* é a base para implementação de interfaces em alto nível, utilizando os componentes gráficos do próprio sistema do aparelho. Já a classe *Canvas* é utilizada para renderização de baixo nível, fornecendo uma tela “em branco” e um objeto gráfico capaz de desenhar sobre ela.

A classe *GameCanvas* é uma extensão da classe *Canvas*. Foi introduzida na versão 2.0 da biblioteca MIDP juntamente com o pacote para desenvolvimento de jogos, e por isso fornece funcionalidades já com esse propósito.

As principais funções da classe *GameCanvas* que podem ser utilizadas para renderização são a *getGraphics*, que retorna um objeto do tipo *Graphics*, e a *flushGraphics*, que garante a visualização na tela do que foi desenhado pelo objeto gráfico.

Um objeto do tipo *Graphics* fornece funções básicas de desenho, tais como: *drawImage*, para renderização de imagens, *drawRect* e *fillRect*, para desenho de retângulos, *drawString*, para desenho de texto, *setFont*, para seleção de fonte, *setColor*, para seleção de cor, entre outras.

Para ter acesso a um objeto do tipo *Graphics* a classe *GameCore* possui uma referência a um objeto do tipo *GameCanvas* que é selecionado como tela principal do aplicativo através da classe de controle *Display*. Durante o ciclo do jogo, o objeto *Graphics* é passado como parâmetro para o método de renderização, onde será utilizado para desenhar os objetos do jogo, que são então mostrados na tela do aparelho.

A classe *GameCanvas* fornece ainda métodos para controle de entradas do usuário, através das funções *keyPressed*, *keyReleased* e *keyRepeated*, invocadas quando o usuário pressiona qualquer tecla do aparelho, e até mesmo métodos para controle de toque, em telas sensíveis ao toque, com os métodos *pointerPressed*, *pointerDragged* e *pointerReleased*.

O trecho de código a seguir mostra a implementação da classe *GameCore*.

```
public abstract class GameCore extends MIDlet implements Runnable {

// Inicialização do aplicativo
protected void startApp() throws MIDletStateChangeException {
	if (!paused) {
		// Cria um novo objeto do tipo GameCanvas
		canvas = new GameCanvas();
		// Seta o GameCanvas como tela atual
		Display.getDisplay(this).setCurrent(canvas);
	}
}
(…)

// Ciclo do jogo
public void run() {
	while (!paused) {
		// Atualiza
		update();
		// Recupera o objeto gráfico do canvas
		Graphics g = canvas.getGraphics();
		// Renderiza
		paint(g);
		canvas.flushGraphics();
	}
}

// Método abstrato para renderização
protected abstract void paint(Graphics g);

// Método abstrato para atualização
protected abstract void update();

// Método para inicialização do jogo
protected abstract void loadGame();
```

Trecho de código 8 – Implementação da classe *GameCore*

É possível notar que, dessa forma, o ciclo do jogo acabaria por sobrecarregar o aplicativo, uma vez que é repetido indefinidamente, consumindo todo o tempo e recursos disponíveis da máquina virtual do dispositivo. Logo, é necessária a introdução de um tempo de espera no ciclo, o que pode ser feito através do método *sleep* da própria *Thread*. Porém, essa abordagem não garante a uniformidade da velocidade de execução do jogo, pois o tempo necessário para o processamento, atualização e renderização do jogo varia, de acordo com a quantidade e tipos de objetos na cena.

Foi então adicionado um cálculo para obter a quantidade de millisegundos que a *Thread* deverá esperar até realizar o próximo ciclo, a partir da definição de quantas atualizações por segundo são necessárias em cada ciclo e quanto tempo será gasto para executá-las.

A partir da implementação do *GameCore*, já é possível criar um exemplo de jogo simples, implementando as funções de atualização e renderização (ver anexo I: manual de utilização do *framework*).

5.1.2 Gerenciamento de cenas

No módulo de gerenciamento de cenas, implementamos a classe *Scene,* que representa uma cena qualquer, a classe *GameObject*, que representa objetos quaisquer que possam ser adicionados a uma cena, e a classe *Game*, que estendende *GameCore* adicionando os métodos necessários para o gerenciamento das cenas.

A classe *GameObject*, como um elemento qualquer, deve implementar, no mínimo, os métodos *update* e *paint* do ciclo principal, portanto, esses são os únicos métodos dessa classe, declarados como abstratos, para que as especializações de *GameObject* possam tratá-los da forma que lhes convier.

A classe *Scene* possui métodos para adição e remoção de objetos do tipo *GameObject* e implementa as funções de renderização e atualização, repassando-as a todos os objetos que contém, para que sejam pintados e atualizados. Possui também métodos para carregamento e descarregamento da cena, invocados quando esta se torna ativa ou inativa.

A classe *Game* implementa os métodos do ciclo principal, repassando-os também para a cena atual, que por sua vez repassa a todos os objetos que contiver. Assim, todos os objetos da cena são atualizados e renderizados a partir das invocações do ciclo principal.

A classe *Game* é também a responsável pelo gerenciamento das cenas, mantendo uma pilha e fornecendo métodos para adicionar ou remover cenas a ela. A classe *Game* controla ainda o carregamento e descarregamento das cenas, invocando seus respectivos métodos quando se tornam ou deixam de ser a cena atual.

Os trechos de código 9, 10 e 11 mostram a implementação das classes *GameObject, Scene* e *Game*, respectivamente.

```
public abstract class GameObject {

	// Método para atualização do objeto
	public abstract void update();
	// Método para renderização do objeto
	public abstract void paint(Graphics g);
}
```

Trecho de código 9 – Implementação da classe *GameObject*

```
public abstract class Scene {
	// Método para o carregamento da cena
	public abstract void load();
	// Método para o descarregamento da cena
	public abstract void unload();

	// Adiciona um GameObject à cena
	public void add(GameObject object){…}
	// Remove um GameObject da cena
	public void remove(GameObject object) {…}

	// Pinta todos os objetos que essa cena contém
	public void paint(Graphics g){…}
	// Atualiza todos os objetos que essa cena contém
	public void update() {…}
```

Trecho de código 10 – Implementação da classe *Scene*.

```
public class Game extends GameCore {
	// Pilha de cenas
	private Stack sceneStack = new Stack();
	// A cena atual
	private Scene current;

	// Pinta a cena atual
	protected void paint(Graphics g) {…}

	// Atualiza a cena atual
	protected void update(){…}

	// Substitui a cena atual
	public void setScene(Scene scene) {
		// Carrega a nova cena
		scene.load();
		// Seta a cena atual
		current = scene;

		// Se havia outra cena no topo da pilha, remove,
		// descarrega e chama o garbage collector
		if (!sceneStack.isEmpty())
		{
			Scene s = ((Scene) sceneStack.pop());
			s.unload();
			System.gc();
		}
		// Coloca a cena atual no topo
		sceneStack.push(current);
	}

	// Insere uma cena no topo da pilha
	public void pushScene(Scene scene){…}

	// Remove a cena do topo da pilha
	public Scene popScene() {…}
}
```

Trecho de código 11 – Implementação da classe *Game*

A partir dessa implementação, podemos construir facilmente um exemplo utilizando o conceito de cena com vários objetos (ver anexo I: manual de utilização do *framework*).

5.1.3 Componentes de interface

Os componentes de interface são os elementos com funcionalidades mais complexas do *framework*, incluindo funções de alinhamento, navegação, controle de foco e seleção. Para isso é implementada a classe *Component*, que estende *GameObject* e serve de base para todos os outros elementos de interface.

Component

Em primeiro lugar, foram definidos atributos para o posicionamento do componente, que determinam em que posição da tela ele será desenhado e métodos abstratos para obtenção da largura e altura.

A partir do tamanho e posicionamento do componente, podemos criar as funções de alinhamento, utilizando cálculos simples para encontrar as posições de referência, em cima, embaixo, à esquerda, à direita, no centro vertical e no centro horizontal, em relação à tela, ao *Container* e a outros componentes.

O componente possui também dois atributos do tipo *boolean*, que definem se ele possui o foco e se pode obtê-lo.

A implementação da classe *Component* é apresentada no trecho de código 12.

```
public abstract class Component extends GameObject {

	// Posicionamento
	public int x = 0, y = 0;
	// Controle de foco
	public boolean isFocused = false, isFocusable = true;

	(…)

	// Métodos para obtenção da largura e altura
	public abstract int getWidth();
	public abstract int getHeight();

	// Método para alinhamento que usa a tela como referência
	public void alignToDisplay(String alignment) {
		if (alignment.indexOf(BOTTOM) > -1)
			y = Game.getCanvasHeight() - getHeight();
		else if (alignment.indexOf(TOP) > -1)
			y = 0;
		else if (alignment.indexOf(VCENTER) > -1)
			y = Game.getCanvasHeight() / 2 - getHeight() / 2;

		if (alignment.indexOf(LEFT) > -1)
			x = 0;
		else if (alignment.indexOf(RIGHT) > -1)
			x = Game.getCanvasWidth() - getWidth();
		else if (alignment.indexOf(HCENTER) > -1)
			x = Game.getCanvasWidth() / 2 - getWidth() / 2;
	}
```

```
	// Método para alinhamento que usa o contêiner como
	// referência
	public void alignToParent(String alignment) {…}

	// Método para alinhamento que usa outros componentes como
	// referência
	public void align(Component anchor, String alignment) {…}
```

Trecho de código 12 – Implementação da classe *Component*.

Container

A classe *Container* contém uma lista de componentes, aos quais repassa os métodos de atualização e renderização, e faz o controle do foco e navegação.

No método de renderização, antes de passar o objeto gráfico a seus componentes, o contêiner altera o ponto de origem do objeto gráfico para sua própria posição, de forma que a posição dos componentes tem como referência o ponto de origem do contêiner, e não da tela do aparelho, facilitando seu posicionamento.

Para o controle de foco, o contêiner implementa métodos para seleção do componente com foco e para passar o foco ao próximo componente, ou ao anterior, utilizando os atributos *isFocused* e *isFocusable* da classe *Component*.

A navegação é realizada na etapa de atualização do contêiner, onde é verificado o estado das teclas de navegação do aparelho, definindo para qual direção o foco deverá seguir. Porém a navegação é tratada apenas quando o contêiner também detém o foco.

O trecho de código 13 apresenta a implementação dessa classe.

```
public class Container extends Component {

	// Lista de componentes
	private Vector components = new Vector();

	(…)

	// Construtor
	public Container(int width, int height) {…}

	// Método para adicionar componentes ao contêiner
	public void add(Component c) {
		components.addElement(c);
		c.parent = this;
	}

	// Método para remover componentes do contêiner
	public void remove(Component c) {…}
```

```
// Método para selecionar o componente com foco a partir da
// posição na lista
public boolean setFocused(int index) {
      (…)
            if (toBeFocused.isFocusable) {
                  (…)
                  toBeFocused.focused = true;
                  return true;
            }
      }
      return false;
}

// Método para selecionar o componente com foco
public boolean setFocused(Component c){…}

// Método para passar o foco ao próximo componente
public boolean focusNext() {…}

// Método para passar o foco ao componente anterior
public boolean focusPrevious() {…}
```

Trecho de código 13 – Implementação da classe *Container*

Button, SimpleImage, Label e InputLabel

Os componentes restantes estendem a classe *Component* e implementam os métodos abstratos para criar a visualização e interação especializada de cada um.

O botão, representado pela classe *Button*, possui um texto descritivo e uma ação, definida por um objeto *Runnable* passado como parâmetro na construção do objeto. O método de renderização desenha um retângulo com o texto em seu interior, que mudam de cor de acordo com o foco. Durante sua atualização, caso a tecla de seleção do aparelho seja pressionada enquanto possuir o foco, o objeto *Runnable* é executado. Sua altura e largura são definidas ao executar o construtor da classe.

A classe *SimpleImage* recebe uma imagem como parâmetro em sua construção, e apenas a desenha em seu método de renderização. Esse componente não pode receber o foco e não executa nenhuma ação. Sua largura e altura são definidas de acordo com o tamanho da imagem que contém.

A classe *Label* recebe um texto como parâmetro de construção, e o desenha na etapa de renderização. Também não pode receber o foco. Sua largura e altura são definidas pela altura e largura do texto.

A classe *InputLabel* implementa a função de entrada de texto pelo usuário. Devido à grande variedade de aparelhos, cada qual com um modelo diferente de teclado, a entrada de texto é tratada de maneira diferenciada: quando uma tecla é pressionada com o foco em um objeto do tipo *InputLabel*, uma nova tela é apresentada, com interface e método de inserção de texto próprios do sistema. Quando finalizado, o texto é então repassado para o objeto *InputLabel*, que passa a desenhá-lo em seu método de renderização.

A implementação desses componentes é apresentada nos trechos de código 14, 15, 16 e 17.

```
public class Button extends Component {
	// Texto descritivo
	private String text;
	// Ação de seleção
	public Runnable action;

	// Construtor
	public Button(String text, int width, int height, Runnable
action) {…}

	// Atualização
	public void update() {
		if (isFocused && Key.pressed(Key.K_SELECT))
			action.run();
	}

	(…)

	// Renderização
	public void paint(Graphics g) {
		if(focused)
			g.setColor(focusedColor);
		else
			g.setColor(color);
		(…)
	}
}
```

Trecho de código 14 – Implementação da classe *Button*

```
public class SimpleImage extends Component {

	// Imagem do componente
	private Image img;

	// Construtor
	public SimpleImage(Image img) {
		this.img = img;
		this.width = img.getWidth();
		this.height = img.getHeight();
		this.isFocusable = false;
	}
	(…)

	// Renderização
	public void paint(Graphics g) {
		g.drawImage(img, x, y, 0);
	}
	(…)
`
```

Trecho de código 15 – Implementação da classe *SimpleImage*.

```
public class Label extends Component {

	// Texto da label
	private String text;

	// Construtor
	public Label(String text) {…}

	// Altura
	public int getHeight() {
		return Font.getDefaultFont().getHeight();
	}
	// Largura
	public int getWidth() {
		return Font.getDefaultFont().stringWidth(text);
	}
	(…)
}
```

Trecho de código 16 – Implementação da classe *Label*

```
public class InputLabel extends Component{

	// Texto atual
	private String text = "";

	// Construtor recebe largura do campo de entrada
	public InputLabel(int width) {
		this.width = width;
		height = Font.getDefaultFont().getHeight();
	}

	(…)

	// Se for selecionado enquanto tiver o foco, chama a tela
	// padrão do sistema
	public void update() {
		if(focused && Key.pressed(Key.K_SELECT)){
			Game.getDisplay().setCurrent(
				new InputForm("", text, 0));
		}
	}
	(…)
}
```

Trecho de código 17 – Implementação da classe *InputLabel*

5.1.4 Gerenciamento de eventos

Dando continuidade à implementação do *framework*, foi implementado um gerenciador de eventos, de acordo com a arquitetura apresentada no projeto.

A classe *Event* consiste em um atributo do tipo inteiro, definido como parâmetro no construtor da classe, que servirá para definir o tipo de evento que o objeto representa.

A interface *EventListener* possui apenas um método para o tratamento dos eventos, que recebe então um evento como parâmetro.

Os trechos de código 18 e 19 mostram a implementação da classe *Event* e da *interface EventListener*.

```
public class Event {
	// Atributo inteiro, que definirá o tipo do evento
	public final int type;

	// Construtor
	public Event(int type) {
		this.type = type;
	}
}
```

Trecho de código 18 – Implementação da classe *Event*

```
public interface EventListener {

	// Método que deverá ser implementado para escutar eventos
	public void handleEvent(Event e);

}
```

Trecho de código 19 – Implementação da *interface EventListener*

A classe *EventManager* é a classe principal desse módulo, responsável por manter uma lista de ouvintes de eventos relacionados ao tipo de evento que ouvem, e implementar métodos para registrar e desregistrar ouvintes, e disparar eventos.

Para a implementação da lista de ouvintes, foi criado um *array* de objetos do tipo *Vector*, onde cada posição do *array* corresponde ao tipo de evento de mesmo número, e contém um *Vector* com os ouvintes daquele tipo de evento. Como um *array* tem um tamanho fixo, foi criado um método para definir o número de tipos de eventos suportados. Disparar ou registrar-se para um evento com tipo inferior a zero ou acima do tamanho definido resultará em uma exceção.

O trecho de código 20 mostra a implementação dessa classe.

```
public class EventManager {

	// Lista de ouvintes
	private static Vector[] list = new Vector[50];

	// Método para definir o número de tipos suportados
	public static void reset(int maxNumberOfEventTypes) {
		list = new Vector[maxNumberOfEventTypes];
	}

	// Método para registrar ouvintes de eventos
	public static void register(EventListener listener, int
		eventType) {…}

	// Método para desregistrar ouvintes de eventos
	public static void unregister(EventListener listener, int
		eventType) {…}

	// Método para disparar eventos
	public static void dispatch(Event e){
		Vector listeners = list[e.type];
		for(int i=0;i<listeners.size();i++){
			((EventListener)listeners.elementAt(i))
				.handleEvent(e);
		}
	}
}
```

Trecho de código 20 – Implementação da classe *EventManager*

5.1.5 Suporte à comunicação através da rede

Para transferência de dados, a biblioteca MIDP disponibiliza algumas classes comuns do pacote *java.io*, como *DataInputStream* e *DataOutputStream*, e também classes específicas para dispositivos móveis, no pacote *javax.microedition.io*. Nesse pacote há uma classe chamada *Connector*, responsável por estabelecer todos os tipos de conexão que o aparelho suporta. A partir dessa classe podemos estabelecer uma conexão TCP utilizando *sockets* e obter um objeto do tipo *SocketConnection*.

A classe *SocketConnection*, por sua vez, fornece métodos para obtenção de objetos do tipo *DataInputStream* e *DataOutputStream*, que podem ser utilizados para transferência de *bytes* através da conexão *socket* estabelecida.

A princípio foi criada uma classe que fornece métodos para inicialização de uma conexão TCP e para envio e recepção de *bytes* através dessa conexão. A essa classe foi dado o nome de *SocketClient*, como mostra o trecho de código 21, a seguir.

```
public abstract class SocketClient {

	// Objeto responsável pela conexão
	private SocketConnection c;
	// Objetos resonsáveis pela entrada e saída de dados
	private DataInputStream dis;
	private DataOutputStream dos;

	// Implementação de método para conexão via socket
	public void connect(String url) throws IOException {
		url = "socket://" + url;
		c = (SocketConnection)Connector.open(url);
		dis = c.openDataInputStream();
		dos = c.openDataOutputStream();
		startListening();
	}

	// Método para envio de pacotes de bytes
	public void send(byte[] data) throws IOException {…}

	// Método abstrato, para o tratamento de bytes recebidos
	protected abstract void onDataReceived(DataInputStream dis);

		// Método para inicialização da Thread de escuta
	public void startListening() {
		listening = true;
		Thread listeningThread = new Thread() {
			(…)
				bytesReceived = dis.available();
				if (bytesReceived > 0)
					onDataReceived(dis);
				(…)
	}
```

Trecho de código 21 – Implementação da classe *SocketClient*

Essa classe nos permite criar uma conexão TCP e fazer transferência de dados através de pacotes de *bytes*. A partir dos métodos implementados, estendemos essa classe para trabalhar com mensagens ao invés de *bytes*, criando então a classe *MessageClient.*

A classe *MessageClient* transforma pacotes de *bytes* em objetos do tipo *Message*, no envio e na recepção de dados. Além disso, é responsável pela integração com o gerenciamento de eventos do jogo, o que facilita o tratamento das mensagens recebidas.

O objetivo da classe *Message* é transmitir os eventos do jogo de um ponto a outro na rede, representando ações como jogada, pedido de jogo, mensagem, etc. Para que isso seja possível, a classe *Message* estende a classe *Event*, implementando funções necessárias para representar as ações de jogo e facilitar sua transmissão via rede.

Foi adicionado à classe *Message* um *array* de *bytes* que corresponde à informação que a mensagem carrega, de qualquer tipo, e métodos que facilitam a obtenção dessa informação em forma de texto ou números inteiros. Disponibiliza também construtores que recebem

esses tipos básicos como parâmetro, ficando responsável por transformá-los em *bytes* para o envio.

Para a transmissão, foi implementado um método que transforma o objeto *Message* em um pacote de *bytes*, no formato definido pelo protocolo da aplicação.

Os trechos de código 22 e 23 mostram a implementação das classes *Message* e *MessageClient*.

```
public class Message extends Event {

	// Pacote de dados da mensagem, em forma de bytes
	public final byte[] data;

	// Construtor primário que define apenas o tipo do evento
	public Message(int type) {…}
	// Construtor que recebe um array de bytes como parâmetro
	public Message(int type, byte[] data) {…}

	// Construtor para mensagens que carregam um valor inteiro
	public Message(int type, int value) {
		super(type);
		data = ByteArrayUtil.toByteArray(value);
	}

	// Construtor para mensagens que carregam um texto
	public Message(int type, String value) {
		super(type);
		data = value.getBytes();
	}

	// Método para obtenção do pacote de bytes da mensagem
	// segundo o protocolo de aplicação:
	// [TIPO_DO_EVENTO] [TAMANHO_DO_PACOTE] [PACOTE_DE_DADOS]
	public byte[] serialize() {…}

	// Método que retorna um inteiro a partir do pacote de bytes
	public int getInt() {…}

	// Método que retorna texto a partir do pacote de bytes
	public String getString() {…}
```

Trecho de código 22 – Implementação da classe *Message*.

```
public class MessageClient extends SocketClient {

	// Método para envio de mensagens, registrando um ouvinte
	// para o tipo da mensagem enviada
	public void sendMessage(Message msg, EventListener listener)
		throws IOException {
		sendMessage(msg);
		EventManager.register(msg.type, listener);
	}
```

```
        // Trata os pacotes de bytes recebidos, transformando-os
        // em mensagem e disparando através do EventManager
        protected void onDataReceived(DataInputStream dis) {
              int bytesReceived = dis.available();
              if (bytesReceived > 0) {
                    // Recupera o tipo da mensagem
                    int type = dis.readInt();
                    // Recupera o tamanho do pacote de dados
                    int length = dis.readInt();
                    // Recupera o pacote de dados
                    byte[] data = new byte[length];
                    dis.readFully(data);
                    // Dispara uma mensagem com o tipo e dados
                    // recebidos
                    EventManager.dispatchEvent(new Message(type,
                          data));
              }
              (…)
        }
}
```

Trecho de código 23 – Implementação da classe *MessageClient*.

A partir dessas classes podemos facilmente enviar mensagens no formato definido pelo projeto, como mostra o trecho de código 24:

```
        // Cria um MessageClient
        MessageClient mc = new MessageClient();
        // Conecta via TCP a "meuservidor.com"
        mc.connect("meuservidor.com");
        // Envia uma mensagem do tipo TEXT_MESSAGE, com o
        // texto "Olá servidor"
        mc.sendMessage(new Message(TEXT_MESSAGE, "Olá servidor."));
        // Envia uma mensagem do tipo PONTUACAO, com o
        // número inteiro 1573
        mc.sendMessage(new Message(PONTUACAO, 1573));
```

Trecho de código 24 – Exemplo de uso das classes *Message* e *MessageClient*.

5.2 *Framework* para o servidor dos jogos

A implementação do *framework* para o servidor apresenta uma arquitetura com funções comuns a jogos multi-jogadores sem interação em tempo real: o gerenciamento de conexões e ambientes virtuais e a troca de mensagens e processamento da lógica do jogo.

A linguagem utilizada para o desenvolvimento do servidor foi o C#. A escolha da linguagem foi baseada nas funcionalidades oferecidas e nas necessidades de alto desempenho e robustez em que implicam o gerenciamento e o processamento de jogos multi-jogadores. Adicionalmente, essa escolha mostra que é possível utilizar, na implementação do servidor, uma linguagem diferente daquela utilizada para implementação dos jogos. A ferramenta utilizada para o desenvolvimento do servidor foi o *Microsoft Visual C# 2008 Express Edition*.

Como definido no projeto, foi implementada uma classe chamada *Connection*, responsável pela conexão com o cliente, e que fornece métodos para envio e recepção de mensagens. Para isso foram declarados métodos abstratos, que são implementados em uma especialização, para adequar-se ao protocolo TCP.

Foi incluído na classe *Connection* um atributo do tipo *String* que será o identificador do cliente no servidor, utilizado para localização. Foi implementada também uma lista de ouvintes da conexão, para que outros elementos possam tratar as mensagens recebidas sem a necessidade de estender diretamente a classe *Connection*.

O objeto mensagem utilizado no servidor corresponde ao objeto mensagem utilizado pelo cliente, para que os eventos do jogo sejam interpretados corretamente. Foi criada uma classe *Message*, que inclui um atributo do tipo inteiro, para definição do tipo da mensagem, e um *array* de *bytes*, que carrega a informação adicional da mensagem.

Para a criação de ambientes interativos em que participam dois ou mais jogadores, foi implementada a classe *ConnectionGroup*. Essa classe contém uma lista de objetos do tipo *Connection* e disponibiliza um método para tratar as mensagens deles recebidas.

Os trechos de código 25, 26 e 27 mostram, respectivamente, a implementação das classes *Connection*, *Message*, *ConnectionGroup* e *ConnectionListener.*

```
public abstract class Connection
{
	// Grupo ao qual a conexão pertence
	internal ConnectionGroup group = null;
	// Lista de ouvintes da conexão
	private List<ConnectionListener> listeners =
		new List<ConnectionListener>();
	// Identificador da conexão
	private String id;

	// Construtor
	public Connection(String id){…}

	// Métodos abstratos para envio e recepção de mensagens
	public abstract void SendMessage(Message msg);
	public abstract void MessageReceived(Message msg);

	(…)

	// Método para adição de ouvintes da conexão
	public void AddConnectionListener(ConnectionListener l){…}
	// Método para remoção de ouvintes dessa conexão
	public void RemoveConnectionListener(ConnectionListener l){…}
```

Trecho de código 25 – Implementação da classe *Connection*.

```
public class Message
{
	// Atributo que define o tipo da mensagem
	public readonly int Type;
	// Pacote de dados da mensagem
	private byte[] Data;

	// Construtor para mensagens que possuem apenas um tipo
	public Message(int type){…}
	// Construtor para mensagens com pacotes de dados
	public Message(int type, byte[] data){…}

	// Retorna o pacote de dados
	public byte[] GetData(){…}

	(…)
}
```

Trecho de código 26 – Implementação da classe *Message*.

```
public abstract class ConnectionGroup
{
	// Lista de conexões que o grupo contém
	private List<Connection> list = new List<Connection>();

	// Adiciona uma conexão ao grupo
	public void Add(Connection connection){…}
	// Remove uma conexão do grupo
	public void Remove(Connection connection){…}

	// Obtém uma conexão pelo seu identificador
	public Connection GetConnectionByID(String id){…}

	// Trata mensagens recebidas
	public abstract void MessageReceived(Message msg, Connection
		from);

	// Envia uma mensagem a todas as conexões do grupo
	public void Multicast(Message msg){…}

	(…)
}
```

Trecho de código 27 – Implementação da classe *ConnectionGroup*.

Estendendo a classe *ConnectionGroup*, foram implementadas as classes *Lobby* e *GameSession*, com funções específicas para os objetivos de cada uma.

A classe *Lobby*, representando uma sala de jogadores, trata tipos de mensagens relacionadas ao desafio entre jogadores. A princípio foram definidos os seguintes tipos de mensagens: pedido de jogo, pedido de jogo aceito, pedido de jogo recusado, oponente não disponível e pedido de atualização de lista de jogadores.

Ao receber uma mensagem de pedido de jogo, a classe *Lobby* verifica a qual jogador ela é destinada, localiza o objeto de conexão desse jogador e encaminha a mensagem, junto com o identificador do jogador que fez o pedido.

Ao receber uma mensagem de pedido de jogo aceito, a classe *Lobby* também localiza a conexão do jogador ao qual ela é destinada e então invoca um método abstrato de início de sessão de jogo, que em sua implementação deverá criar uma sessão específica para o jogo em questão.

No recebimento de qualquer das mensagens acima, se o destinatário não for encontrado na lista de jogadores da sala, é enviada ao remetente uma notificação de que o destinatário não está disponível, provavelmente por uma falha na sincronização da lista, que deve ser sincronizada em seguida.

Ao receber uma mensagem de pedido de jogo negado, a classe *Lobby* simplesmente encaminha a mensagem para o destinatário.

A mensagem de pedido de atualização de lista de jogadores é utilizada pelo cliente para solicitar uma sincronização da lista, e utilizada pelo servidor para enviar a lista atual de jogadores. Essa mensagem, quando enviada pelo servidor, contém um texto com os identificadores das conexões do grupo.

O trecho de código 28 mostra a implementação da classe *Lobby*:

```
public abstract class Lobby : ConnectionGroup{
      // Definição dos tipos de mensagem
      public const int PLAYER_LIST_REFRESH = 3,
      START_GAME_REQUEST = 4, START_GAME_ACCEPTED = 5,
      START_GAME_REFUSED = 6, OPPONENT_NOT_AVAILABLE = 7;

      // Tratamento das mensagens recebidas
      public override void MessageReceived(Message msg, Connection
from){
            switch (msg.GetType()){
            case PLAYER_LIST_REFRESH:
                  SendPlayerList(from); break;
            case START_GAME_REQUEST:
                  StartGameRequest(msg, from);break;
            case START_GAME_ACCEPTED:
                        StartGameAccepted(msg, from);break;
            case START_GAME_REFUSED:
                  StartGameRefused(msg, from);break;
            }
        }

      // Método que trata mensagem de pedido de jogo
      protected void StartGameRequest(Message msg, Connection
from){
            String oponentID = Util.GetString(msg.GetData());
            Connection oponent = GetConnectionByID(oponentID);
            if (oponent != null){
                String callerName = from.GetID();
```

```
                oponent.SendMessage(
                        new Message(START_GAME_REQUEST,
                        Util.GetBytes(callerName)));
            }else{
                from.SendMessage(
                        new Message(OPPONENT_NOT_AVAILABLE,
                        msg.GetData());
                SendPlayerList(from);
            }
        }(…)

        // Método abstrato para inicialização da sessão de jogo
        protected abstract void startGameSession(
              Connection[] connection);

        // Envia a lista de jogadores para um cliente
        public void SendPlayerList(Connection to){…}

        // Envia a lista de jogadores para todos
        public void MulticastPlayerList(){…}
```

Trecho de código 28 – Implementação da classe *Lobby*.

A classe *GameSession* foi projetada para servir de base às sessões de jogo. Estendendo *ConnectionGroup*, essa classe define tipos de mensagens para aviso de novo turno do jogador, jogada, desistência, aviso de vitória, derrota e empate. Porém, o único tipo de mensagem que é tratado diretamente pela classe *GameSession* é a mensagem de jogada, invocando um método abstrato que deve ser implementado para tratá-la. Os outros tipos de mensagem devem ser tratados de acordo com a dinâmica do jogo.

O trecho de código 29 mostra a implementação dessa classe.

```
public abstract class GameSession : ConnectionGroup {
        // Definição dos tipos de mensagens
        public const int YOUR_TURN = 30, PLAYER_MOVE = 31,
              GIVE_UP = 32, YOU_WON = 33, YOU_LOSE = 34, TIE = 35;

        // Construtor
        public GameSession(Connection[] players) {…}

        // Tratamento das mensagens recebidas
        public override void MessageReceived(Message msg, Connection
              from){
              if(msg.GetType() == PLAYER_MOVE)
                    HandlePlayerMove(msg, from);
        }

        // Método abstrato para tratamento das jogadas
        protected abstract void HandlePlayerMove(Message move,
Connection from);
}
```

Trecho de código 29 – Implementação da classe *GameSession*.

5.2.1 Estabelecendo a conexão TCP

A partir dessa arquitetura de base foram implementadas as classes responsáveis pelo estabelecimento da conexão TCP, em acordo com o tipo de conexão utilizada pelo cliente.

A classe *SocketConnection* estende a classe *Connection* e implementa os métodos de envio e recepção de mensagens, utilizando *sockets*. É também responsável por montar os objetos do tipo *Message* a partir de pacotes de *bytes* recebidos, utilizando o protocolo definido para a aplicação.

A classe *SocketServer* é responsável pela recepção das conexões. Mantém uma porta de entrada para as conexões via *socket*, e define um método abstrato que recebe como parâmetro o *socket* da conexão recém criada, para direcionamento do cliente de acordo com a dinâmica definida para cada jogo.

Os trechos de código 30 e 31 mostram a implementação das classes *SocketServer* e *SocketConnection*.

```
public class SocketConnection : Connection {

      // Objetos responsáveis pela entrada e saída de dados
      private BinaryReader br;
      private BinaryWriter bw;
      (…)

      // Socket de comunicação com o cliente
      readonly Socket clientSocket;

      // Construtor da classe
      public SocketConnection(String id, Socket socket):base(id){
            (…)
            BeginReceive();
      }
      (…)

      // Método para tratamento dos bytes recebidos e construção
      // da mensagem
      public void OnDataReceived(IAsyncResult asyn){
            int bytesReceived = clientSocket.EndReceive(asyn);
            // Tipo da mensagem
            int type = IPAddress.NetworkToHostOrder(
                        br.ReadInt32());
            // Tamanho do pacote de dados
            int dataLength = IPAddress.NetworkToHostOrder(
                        br.ReadInt32());
            // Recupera o pacote de dados
            byte[] data = new byte[dataLength];
            Array.Copy(InBuffer, (int)br.BaseStream.Position, data,
                  0, dataLength);
            // Cria e dispara a mensagem
            Message msg = new Message(type, data);
            MessageReceived(msg);
            (…)
      }
```

```
        // Método para envio das mensagens
        public override void SendMessage(Message msg){
                (…)
                byte[] data = new byte[bw.BaseStream.Position];
                Array.Copy(OutBuffer, 0, data, 0, data.Length);
                clientSocket.Send(data);
        }
}
```

Trecho de código 30 – Implementação da classe *SocketConnection*.

```
public abstract class SocketServer{
        // Socket para recepção das conexões
        private Socket serverSocket;

        // Construtor
        public SocketServer(int port){
                (…)
                BeginAccept();
        }(…)

        // Método que trata a recepção da conexão
        private void OnClientConnect(IAsyncResult asyn){
                Socket clientSocket = serverSocket.EndAccept(asyn);
                ConnectionAccepted(clientSocket);
                BeginAccept();
        }

        // Método abstrato para encaminhamento da conexão
        abstract protected void ConnectionAccepted(Socket
                clientSocket);
}
```

Trecho de código 31 – Implementação da classe *SocketServer*.

5.2.2 Integração com banco de dados

Para realizar a persistência em banco de dados, foi criada uma classe chamada *MySQLInterface* com o objetivo de facilitar o acesso e transações com banco de dados.

Para tal utilizou-se os componentes disponíveis na própria API do *.NET Framework*, *namespace MySql.Data.MySqlClient*.

O trecho de código 32 mostra a implementação dessa classe.

```
class MySQLInterface {
        private MySqlConnection mConn;

        // Método para conexão com banco
        public void Connect(String server, String database, String
user, String pass) {
                String connectionString = "server="+server+";database="
                +database+";uid="+user+";pwd="+pass+";";
                mConn = new MySqlConnection(connectionString);
```

```
(…)
	// Método para desconexão
	public void Disconnect(){…}

	// Método para executar uma query, sem retorno
	public void Query(String query){
		if (mConn.State == ConnectionState.Open)
			MySqlDataAdapter mAdapter =
				new MySqlDataAdapter(query, mConn);
	}

	// Método que realiza uma query e retorna o conjunto de rows
	public DataRowCollection Select(String query, String table) {
		DataSet mDataSet = new DataSet();
		if (mConn.State == ConnectionState.Open) {
			MySqlDataAdapter mAdapter =
				new MySqlDataAdapter(query, mConn);
			try {
				mAdapter.Fill(mDataSet, table);
			} catch (MySqlException e) {…}
		}
		if (mDataSet.Tables[table] != null)
			return mDataSet.Tables[table].Rows;

		return null;
	}
}
```

Trecho de código 32 – Implementação da classe de integração com banco de dados *MySQL*

Um exemplo de utilização dos *frameworks* para comunicação cliente/servidor pode ser encontrado nos anexos, em “Manual de utilização do *framework*”.

5.3 *Jan Ken Po*

Neste item, será apresentada a implementação do jogo *JanKenPo* utilizando as funcionalidades desenvolvidas nesse trabalho.

As cenas que o jogo apresenta são: menu principal, cena de seleção de apelido, sala de jogadores e sessão de jogo.

Em primeiro lugar foi criada a classe *JanKenPo*, cuja função é inicializar o jogo, selecionando a primeira cena, o menu principal.

Representado pela classe *MainMenu*, o menu principal estende a classe *Scene* e possui um botão para iniciar, que deve levar à cena de seleção de apelido, e um botão para terminar o jogo.

O trecho de código 33 mostra a implementação da classe *MainMenu*.

```
public class MainMenu extends Scene {

	// Método de carregamento da cena
	public void load() {
		// Cria um componente de imagem e o adiciona à tela
		SimpleImage logo =
			new SimpleImage(Image.createImage(
				"/jkp_logo.png"));
		(…)
		// Cria um contêiner para o menu
		Container menu = new Container(50, 40);
		// Cria e adiciona um botão que chama a próxima cena
		Button start = new Button("Iniciar", 50, 20,
			new Runnable() {
				public void run() {
					Game.getGameInstance().pushScene(
						new NickSelection());
				}
		});
		menu.add(start);
		// Cria e adiciona um botão que sai do aplicativo
		Button exit = new Button("Sair", 50, 20, …);
		menu.add(exit);
		// Adiciona o menu à cena
		add(menu);
		(…)
```

Trecho de código 33 – Implementação da cena do menu principal.

O resultado da implementação dessa cena é mostrado na figura 18.

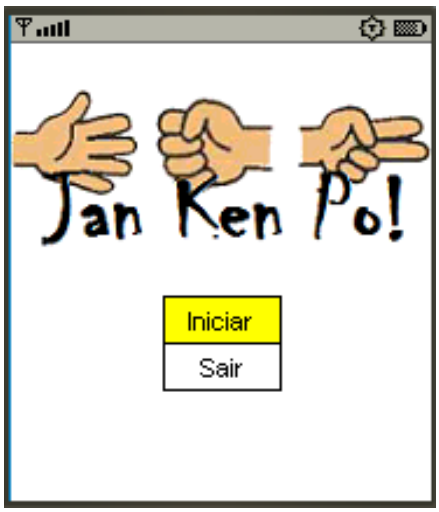


Figura 18 – Menu principal

A seguir, implementamos a cena *NickSelection*, que contém um *InputLabel* para inserção do apelido, um botão para conectar-se ao servidor e outro botão para retornar ao menu principal.

Para tratar da conexão, foi criada uma classe *JKPClient*, que encapsula uma instância estática de *MessageClient* e fornece métodos também estáticos para conexão e envio de mensagens.

O botão com função de conexão faz uma chamada ao método de conexão da classe *JKPClient*, e em seguida envia uma mensagem do tipo *LOGIN*, informando o apelido escolhido pelo usuário.

O trecho de código 34 mostra a implementação da classe *NickSelection*

```
public class NickSelection extends Scene implements EventListener{

	// InputLabel utilizado para inserção do apelido
	InputLabel apelido;

	// Carregamento da cena
	public void load() {
		// Adiciona texto descritivo
		body.add(new Label("Informe um apelido para "));
		body.add(new Label("entrar na sala."));
		// Adiciona a InputLabel para inserção do apelido
		body.add(apelido = new InputLabel(130));
		// Adiciona um botão para conexão
		Button conectar = new Button("Conectar", 60, 20,
			new Runnable() {
			public void run() {
				JKPClient.connect();
				JKPClient.sendMessage(new Message(LOGIN,
					apelido.getText());
			(...)
	}
	// Método para tratamento de eventos
	public void handleEvent(Event e) {
		if (e.type == LOGIN) {
			Message msg = (Message) e;
			if (msg.getString().equals("k")){
				Game.getGameInstance().setScene(
					new Lobby());
				JanKenPo.player = apelido.getText();
			}
		} (...)
}
```

Trecho de código 34 – Implementação da cena de seleção de apelido e conexão.

A figura 19 mostra o resultado da implementação dessa cena.

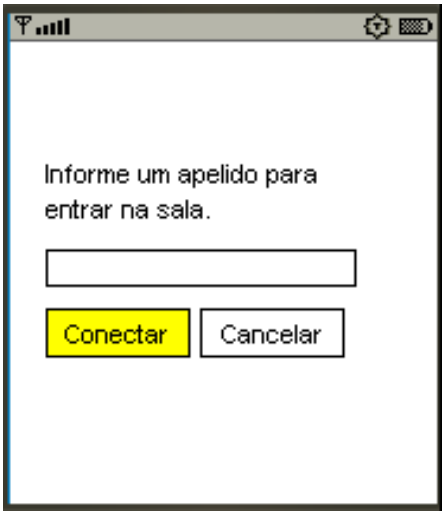


Figura 19 – Tela de seleção de apelido e conexão

Na implementação da sala de jogadores, foi utilizado um contêiner com botões alinhados um abaixo do outro para mostrar a lista de jogadores. Cada botão tem como texto o apelido do jogador que representa, e como ação o envio de um pedido de jogo para esse jogador.

Implementando EventListener, essa classe trata as mensagens de sincronização da lista e os pedidos de jogo dos outros jogadores.

Quando recebe um pedido de jogo a cena apresenta dois botões, um que envia uma mensagem de pedido de jogo recusado, e outro que envia uma mensagem de envio de jogo aceito. Para isso utiliza os mesmos tipos de mensagem definidos no servidor.

Quando um pedido de jogo é aceito, ocorre a transição para a cena da sessão do jogo, que irá esperar por uma mensagem de confirmação do servidor, declarando o início do turno.

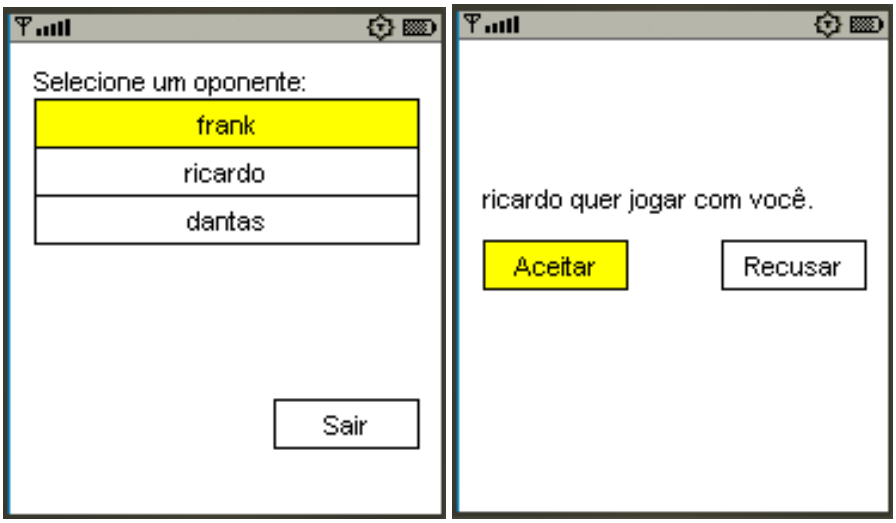
O trecho de código 35 mostra a implementação da cena da sala de jogadores.

```
public class Lobby extends Scene implements EventListener {

	// Método que mostra na tela a lista de jogadores
	private void showList() {
		(…)
		for (int i = 0; i < players.length; i++) {
			body.add(new Button(players[i], body.getWidth(),
				20, new Runnable() {
				public void run() {
					JKPClient.sendMessage(new Message(
					START_GAME_REQUEST, opponent));
					waitForServer();
				}
			}));
		} (…)
	}
```

```
// Método que mostra uma mensagem de espera
private void waitForServer() {
        (…)
        Label aguarde = new Label("Aguarde o outro
                jogador.");
        body.add(aguarde);
        (…)
}
// Método que mostra um pedido de jogo
private void invited(final String by) {
        (…)
        // Cria e adiciona uma label
        Label lbl = new Label(by + " quer jogar com você.");
        // Cria e adiciona o botão para aceitar o pedido
        Button accept = new Button("Aceitar", 60, 20,
                new Runnable() {
                public void run() {
                        JKPClient.sendMessage(new Message(
                                START_GAME_ACCEPTED,by));
                        Game.getGameInstance().pushScene(
                                new GameRoom(by));
                (…)
}

public void handleEvent(Event e) {
        switch (e.type) {
        // Mensagem para atualização de lista
        case PLAYER_LIST_REFRESH:(…)showList();break;
        case START_GAME_REQUEST:(…) invited(resp);break;
        case START_GAME_ACCEPTED:(…)
                Game.getGameInstance().pushScene(
                        new GameRoom(resp)); break;
        (…)
```

Trecho de código 35 – Implementação da sala de jogadores.

O resultado da implementação dessa classe é mostrado nas figuras 20 e 21.



Figuras 20 e 21 – Sala de jogadores e pedido de jogo

A última cena, e também a mais complexa, é a da classe *GameRoom*, que representa uma sessão de jogo.

Essa cena possui três estados bem definidos: aguardando o servidor, fazendo uma jogada e mostrando resultado.

Nos momentos em que o jogador deve aguardar uma mensagem do servidor ou uma interação do oponente, essa cena irá mostrar ao jogador uma mensagem de espera. Como é o caso no começo do jogo em que os jogadores devem aguardar pela notificação de início de turno.

Quando ambos os jogadores estiverem conectados e prontos, o servidor enviará a mensagem de início de turno e a cena do jogo mostrará as opções de jogada que o jogador pode fazer. Para esse caso foi feita uma especialização da classe *Button*, que desenha uma imagem ao invés de texto.

Quando um dos botões de jogada for selecionado, uma mensagem do tipo jogada é enviada para o servidor, informando a opção escolhida pelo usuário.

Após o envio das jogadas e processamento do resultado pelo servidor, ambos os jogadores receberão uma mensagem de resultado, informando se venceram ou não. A cena mostrará as jogadas que cada um fez, um texto descritivo do resultado, um botão para continuar jogando e outro para voltar ao menu principal.

A implementação dessa classe é apresentada no trecho de código 36.

```
public class GameRoom extends Scene implements EventListener {

	(…)
	// Tipos de eventos relacionados às jogadas
	public static final int PAPEL = 0, TESOURA = 1, PEDRA = 2;

	// Construtor
	public GameRoom(String opponentName) {…}

	// Método para mostrar as opções de jogada na tela
	private void makeAMove() {
		// Cria o botão com a imagem do papel
		ImageButton papelbtn = new ImageButton(Image
					.createImage("/papel1.png"),
				new Runnable() {
					public void run() {
						jogada = PAPEL;
						JKPClient.sendMessage(
							new Message(PLAYER_MOVE,
								PAPEL));
						waitForServer();
					}
				});
		// Cria o botão com a imagem da tesoura
		ImageButton tesourabtn = new ImageButton(…);
		// Cria o botão com a imagem da pedra
		ImageButton pedrabtn = new ImageButton(…);
		(…)
	}
```

```
        private void waitForServer() {
                Label aguarde = new Label("Aguarde o outro jogador.");
                (…)
        }

        private void result(int result) {
                Label lbl = new Label(result == 0 ? "Empate!"
                        : result == 1 ? "Você perdeu." : "Você ganhou!");

                Label lbl2 = new Label(JanKenPo.player);
                Label lbl3 = new Label(opponent);
                (…)
                Button cont = new Button("Jogar novamente", 90, 20,…);
                (…)
        }
        (…)

        //Método para tratamento dos eventos
        public void handleEvent(Event e) {
                switch (e.type) {
                case YOUR_TURN:makeAMove();break;
                case GIVE_UP: opponentGaveUp();break;
                case YOU_WON: result(2); break;
                case YOU_LOSE: result(1); break;
                case TIE: result(0); break;
        }
}
```

Trecho de código 36 – Implementação da classe responsável pela sessão de jogo

As figuras 22 e 23 mostram o resultado da implementação da sessão de jogo.

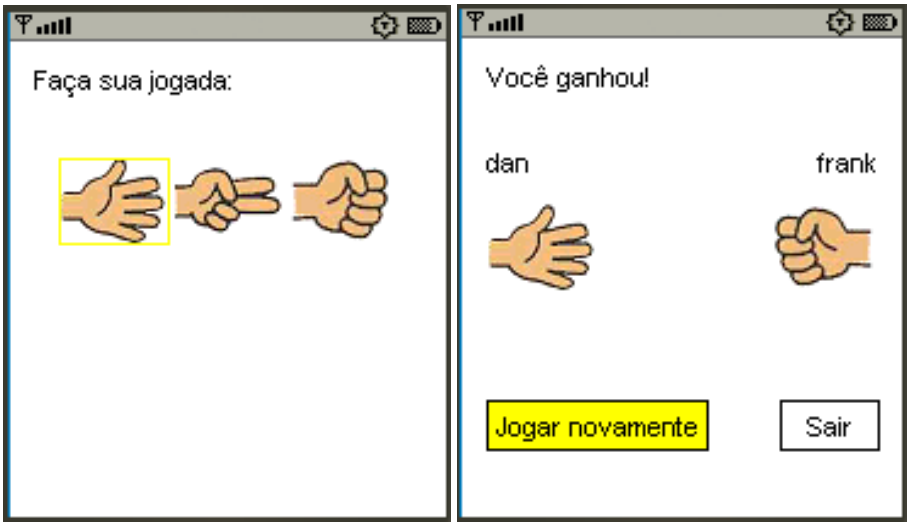


Figura 22 e 23 – Resultado da implementação da interface da sessão de jogo

5.3.1 Implementação do servidor para o *Jan Ken Po*

Após a implementação do jogo, passamos à implementação do servidor. Foi feita uma extensão a partir da arquitetura já criada para atender às necessidades específicas do *JanKenPo*.

Primeiramente, para tratar a recepção das conexões, foi feita uma extensão da classe *SocketServer*, à qual foi dado o nome de *JKPServer*. Esta classe foi preparada para receber conexões via *socket* iniciadas pelo jogador e tratá-las de acordo com o caminho definido.

No caso do *Jan Ken Po*, verifica-se se o apelido escolhido pelo usuário não está em uso por outro jogador presente na sala e, em caso negativo, o usuário é direcionado à sala de jogadores. A classe *JKPServer* implementa a *interface ConnectionListener* a fim de tratar a mensagem do tipo *LOGIN*, e encaminhar a conexão do jogador em seguida.

A classe que controla a sala de jogadores é uma extensão da classe *Lobby*, à qual chamamos *JKPLobby*. Sua única função, além daquelas já implementadas pela classe *Lobby*, é iniciar uma sessão de jogo específica para o *Jan Ken Po*, que também deve ser especializada a fim de tratar a lógica do jogo.

Os trechos de código 37 e 38 apresentam a implementação das classes *JKPServer* e *JKPLobby*.

```
class JKPServer : SocketServer, ConnectionListener{
      // Sala de jogadores
      private JKPLobby lobby = new JKPLobby();
      // Tipo de evento para o login
      public const int LOGIN = 198;

      public JKPServer() : base(3692) {…}

      // Recebe o pedido de conexão e adiciona o servidor
      // como listener para o tratamento da mensagem de login
      protected override void ConnectionAccepted(Socket conn) {
            new SocketConnection("SocketConnection", conn)
                  .AddConnectionListener(this);
      }

      // Trata a mensagem de login recebida
      public void MessageReceived(Message msg, Connection c){
            if (msg.Type == LOGIN){
                  // Caso o apelido esteja em uso, notifica o
                  // cliente
                  if (lobby.GetConnectionByID(nickname) == null){
                        c.SetID(nickname);
                        (…)
                        c.SendMessage(new Message(LOGIN, "k"));
                        lobby.Add(c);
                  }else{
                        c.SendMessage(new Message(LOGIN, "k"));
                  }
      (…)
```

Trecho de código 37 – Implementação da classe *JKPServer.*

```
public class JKPLobby : Lobby
{
	protected override void startGameSession(Connection[] c) {
		// Inicia uma nova sessão de jogo
		new JKPSession(c, this);
	}
```

Trecho de código 38 – Implementação da classe *JKPLobby*.

A classe *JKPSession* representa uma sessão de jogo e portanto estende a classe *GameSession*. Quando se cria uma sessão, é enviada uma mensagem de início de turno para ambos os jogadores, que poderão então enviar suas jogadas.

Foram definidos um tipo de evento para cada jogada do *Jan Ken Po*: um para "papel", um para "tesoura" e um para "pedra".

Ao receber uma jogada, o *JKPLobby* verifica a qual jogador a jogada pertence e a armazena. Quando ambos os jogadores enviarem suas jogadas, ela verifica qual deles venceu e envia-lhes o resultado.

Após o envio dos resultados, os jogadores decidirão entre continuar jogando ou desistir. Para isso, foram definidos também dois outros tipos de eventos. Se receber uma mensagem do tipo "continuar jogando", a classe *JKPSession* verifica se ambos os jogadores optaram por continuar o jogo, e então envia outra mensagem de inicio de turno e o jogo recomeça. Caso receba uma mensagem de desistência, a classe *JKPSession* irá notificar o outro jogador e recolocar ambos na sala de jogadores.

O trecho de código 39 mostra a implementação dessa classe.

```
public class JKPSession : GameSession{

	// Tipos de mensagens referentes ao jogo
	public const int PAPEL = 0, TESOURA = 1, PEDRA = 2,
		PLAY_AGAIN = 29;

	public JKPSession(Connection[] players, JKPLobby lobby)
		:base(players) {(…)
		player1.SendMessage(new Message(YOUR_TURN));
		player2.SendMessage(new Message(YOUR_TURN));
	}

	// Trata as mensagens específicas
	public override void MessageReceived(Message msg, Connection
		from) {
		switch (msg.Type){
		    // Jogador desistiu
		      case GIVE_UP:
		            (…)break;
		      // Jogador quer jogar novamente
		      case PLAY_AGAIN:
		      if (jogada2 == -1 && jogada1 == -1) {
		            player1.SendMessage(new Message(YOUR_TURN));
		            player2.SendMessage(new Message(YOUR_TURN));
		(…)
```

```
        // Método para tratamento das jogadas
        protected override void HandlePlayerMove(Message move,
              Connection from) {
              (…)// Armazena jogadas
              if (from == player1)jogada1 = jogada;
              else jogada2 = jogada;
              // Ambos já jogaram, calcula vitória
              if (jogada1 != -1 && jogada2 != -1)
                   CheckWinner();
        }

        // Método para calcular o resultado
        private void CheckWinner() {
              // Empate
              if (jogada1 == jogada2){
                   player1.SendMessage(new Message(TIE));
                   player2.SendMessage(new Message(TIE));
              }
              // Player 1 ganha
              else if ((jogada1 == PAPEL && jogada2 == PEDRA)|| …) {
                   player1.SendMessage(new Message(YOU_WON));
                   player2.SendMessage(new Message(YOU_LOSE));
              }
              // Player 2 GANHA
              else {
                   player1.SendMessage(new Message(YOU_LOSE));
                   player2.SendMessage(new Message(YOU_WON));
              }
          }
```

Trecho de código 39 – Implementação da classe *JKPSession.*

5.3.2 Resultado da implementação

Após a implementação do jogo e do servidor do jogo *Jan Ken Po*, podemos demonstrar uma partida do início ao fim, em acordo com as especificações do projeto. Na figura a seguir, dois jogadores, João e José, conectaram-se ao servidor e aparecem na sala de jogadores.

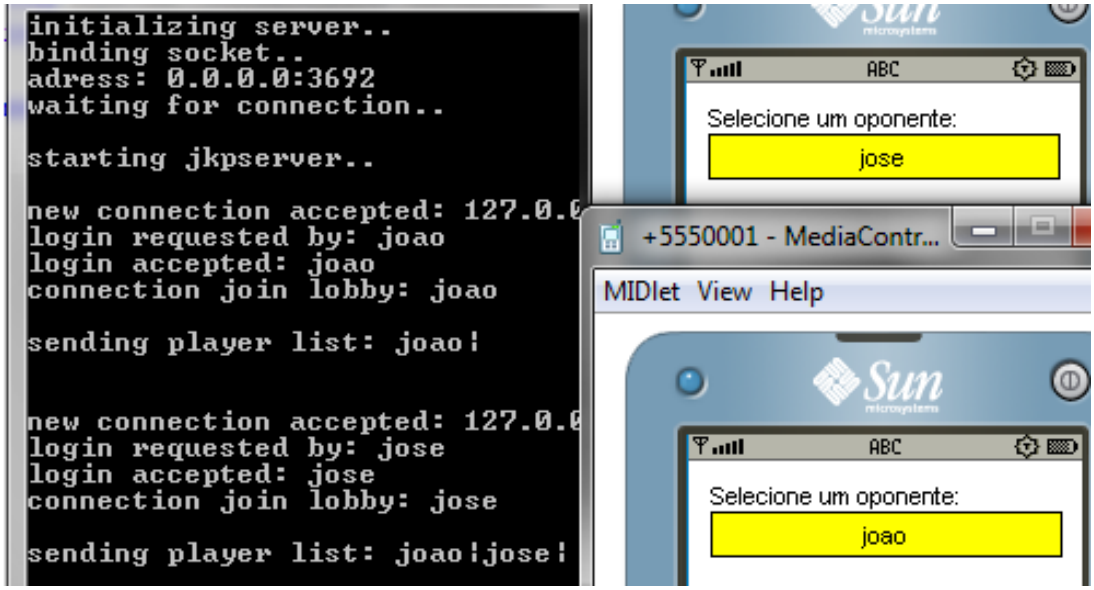


Figura 24 – Resultado do *JanKenPo*: sala de jogadores

Na próxima figura, João, representado pelo simulador do lado esquerdo, selecionou José na lista de oponentes, desafiando-o para uma partida.

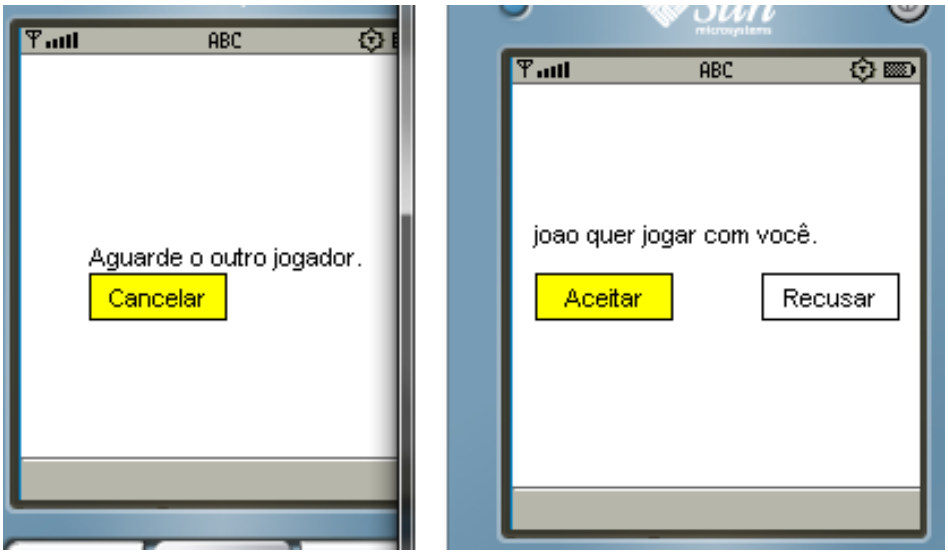


Figura 25 – Resultado do *JanKenPo*: pedido de jogo

Na figura a seguir, José aceitou o desafio de João e os dois foram encaminhados para uma sessão de jogo, onde estão prontos para enviar suas jogadas.

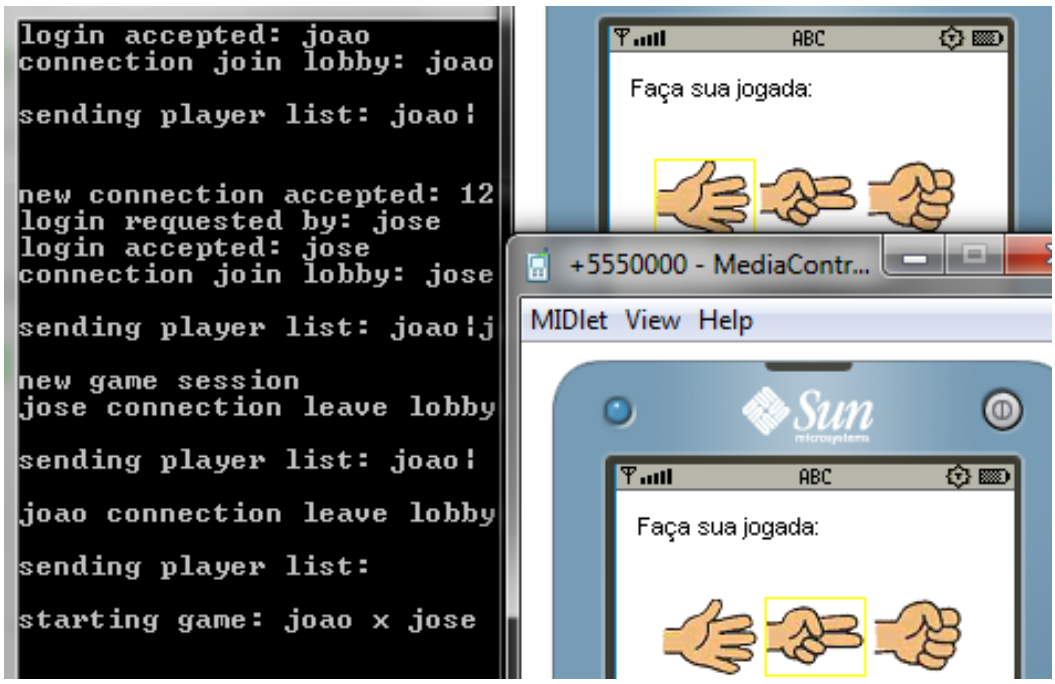


Figura 26 – Resultado do *JanKenPo*: sessão de jogo

Por fim, após o envio das jogadas por ambos os jogadores, o servidor os envia o resultado.

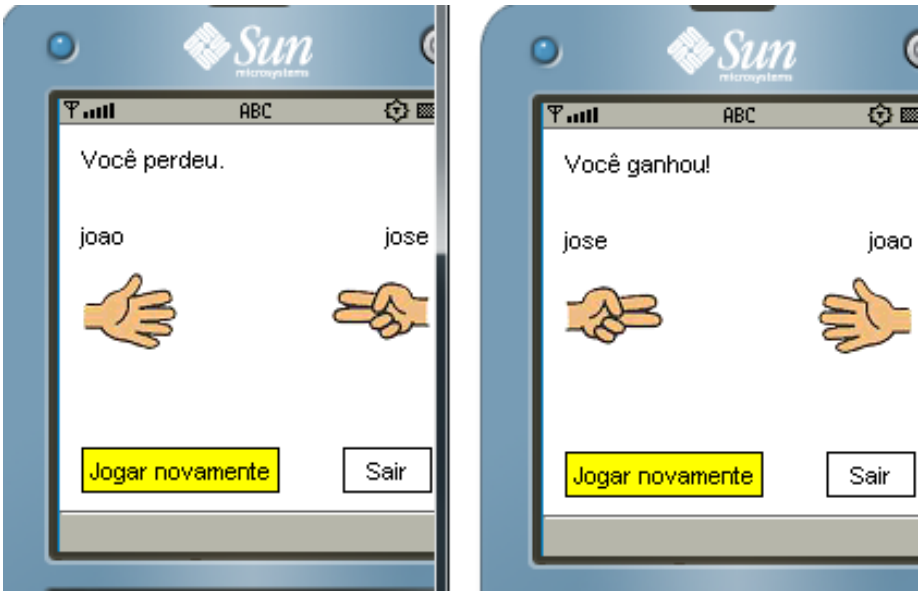


Figura 27 – Resultado do *JanKenPo*: resultado do jogo

5.4 *JobFun*

Nesta seção será apresentada a implementação do jogo *JobFun*. Serão adicionadas funcionalidades que complementam o *framework* para criação de jogos assíncronos e com persistência em banco de dados.

Para modelagem, criação e execução do banco de dados foram utilizados os programas *MySQL Workbench* da *Oracle* (MySQL Workbench, 2010) e *XAMPP* da *Apache Friends* (XAMPP, 2010).

Dada a quantidade de cenas e funcionalidades necessárias a esse jogo, nem todas serão implementadas e apresentadas nesse trabalho, dando-se ênfase apenas às mais relevantes.

As cenas principais que o jogo possui são: cena de *login* do usuário, cena de seleção de profissão, quarto virtual do jogador, cenas de produção, compra e venda de itens.

A primeira cena do jogo é a cena *MainMenu*, onde será realizada a conexão e validação do usuário. Essa cena possui dois campos de texto, um para o nome de usuário do jogador e outro para a senha, e também um botão para conexão, como mostra a figura 28.

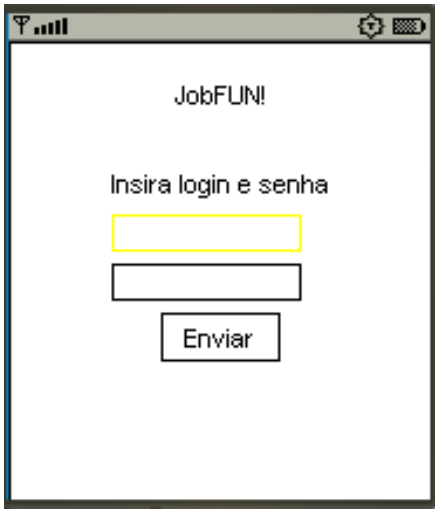


Figura 28 – Cena de entrada do JobFun

Quando o botão de conexão é pressionado, verifica-se o preenchimento dos campos de texto e é enviada uma mensagem do tipo "*login*" para o servidor contendo o nome de usuário e senha apresentados. Para esse trabalhao nenhum tipo de segurança quanto à senha do usuário foi implementada.

Caso o nome de usuário fornecido não esteja presente no banco de dados é apresentada uma mensagem com a opção de criar um novo usuário. Ao optar por criar um novo usuário, o jogador é levado à cena de seleção de profissão, que será sua única profissão e influenciará no resto do jogo. Essa informação é armazenada no aparelho e o jogo passa à cena do quarto virtual do jogador, que começa com um saldo de 50 créditos.

A figura 29 mostra a cena de seleção de profissão.

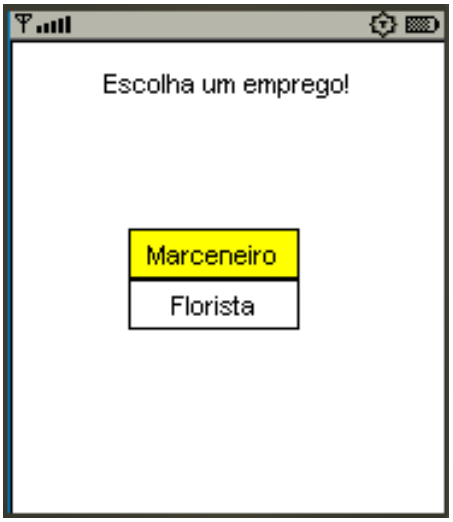


Figura 29 – Cena de seleção de profissão

O trecho de código a seguir mostra o tratamento das conexões e mensagens do cliente para a validação do *login* no servidor, interagindo com o banco de dados.

```
class JFServer : SocketServer, ConnectionListener     {
      // Tipo de eventos para o login
      public const int LOGIN_MESSAGE = 0, NEW_USER = 2;
      // Classe de integração com o banco de dados
      MySQLInterface msql;

      public JFServer() : base(3692) {
            msql = new MySQLInterface();
            msql.Connect("localhost", "jobfun", "jobfun",
                  "jobfun");
      }
      (…)
      // Trata a mensagem de login recebida
      public void MessageReceived(Message msg, Connection c) {
            switch (msg.Type) {
                  case LOGIN_MESSAGE:
                        ValidateUser(msg, c);
                        break;
                  case NEW_USER:
                        NewUser(msg, c);
                        break;
            }
      }(…)

      private void ValidateUser(Message msg, Connection c) {
            String data = Util.GetString(msg.GetData());
            int spt = data.IndexOf('#');
            String username = data.Substring(0, spt);
            String pass = data.Substring(spt + 1);

            String query = "SELECT `pass` FROM `user` WHERE
                  `name`=\"" + username + "\"";

            DataRowCollection ds = msql.Select(query, "user");
```

```
            // Verifica se o usuario existe
            if (ds != null && ds.Count > 0) {
                  String userpass = (String)ds[0].ItemArray[0];

                  // Verifica senha
                  if (pass.Equals(userpass)) {
                        c.SendMessage(new Message(LOGIN_MESSAGE,
                              Util.GetBytes("success")));
                        c.SetID(username);
                  } else {
                        c.SendMessage(new Message(LOGIN_MESSAGE,
                              Util.GetBytes("pass_error")));
                  }
            } else {
                  // Caso o usuário não exista, envia uma
                  // mensagem de cadastro
                  c.SendMessage(new Message(LOGIN_MESSAGE,
                        Util.GetBytes("user_error")));
            }
      }

      private void NewUser(Message msg, Connection c) {
            String data = Util.GetString(msg.GetData());

            int spt = data.IndexOf('#');
            String username = data.Substring(0, spt);
            data = data.Substring(spt + 1);
            spt = data.IndexOf('#');
            String pass = data.Substring(0, spt);
            String userJob = data.Substring(spt + 1);

            String query = "INSERT INTO `Users` (`name`, `pass`,
                  `cash`, `job`) VALUES ('" + username + "', '" +
                  pass + "', '50', '" + userJob + "');";

            msql.Query(query);
            c.SendMessage(new Message(NEW_USER,
                        Util.GetBytes("success")));
      }
}
```

Trecho de código 40 – Tratamento de *login* no servidor com banco de dados para o *JobFun*

Quando o usuário já está cadastrado, após receber a mensagem de confirmação de *login*, o jogo também passa à cena da casa virtual do jogador. Essa cena mostra as imagens de alguns dos objetos que o usuário possui, utilizando componentes do tipo *SimpleImage*, *Container* e *Label*. A partir do menu contido nessa cena, atribuído ao botão de menu esquerdo (*softkey 1*), pode-se ir à cena de compra, venda ou produção de itens.

A figura 30 mostra a casa virtual do jogador, alguns dos itens que possui e o menu de opções.

Figura 30 – Cena da casa virtual do jogador no *Jobfun*

Ao entrar na cena do quarto virtual, o jogo faz uma sincronização com o servidor, a fim de atualizar o saldo de créditos e o estado dos objetos que o jogador possui disponíveis para venda. Como mostram os trechos de código 41 e 42.

```
public class HouseScene extends JobfunBaseScene implements
EventListener {
	public static final int UPDATE_STATE = 1;

	public void load() {
		title = "Sua casa";
		super.load();
		JFClient.sendMessage(new Message(UPDATE_STATE), this);
		(…)
	}
	(…)
	public void handleEvent(Event e) {
		Message m = (Message) e;
		switch (m.type) {
		case UPDATE_STATE:
			String data = m.getString();
			// Formato da mensagem:
			// saldo#nome$tipo$estado$preco¥nome$tipo$...
			String[] itens;
			if (data != null && data.length() > 0) {
				itens = StringUtil.split(data, "#");
				double saldo = Double.parseDouble(itens[0]);
				// Salva o saldo de creditos localmente
				Util.saveRSString("JF-CREDITS", saldo+"");
				// Salva itens à venda localmente
				if (itens.length > 1 && itens[1].length()>0)
					Util.saveRSString("JF-SELL-ITENS",
						itens[1]);
			}
			break;
		} (…)
```

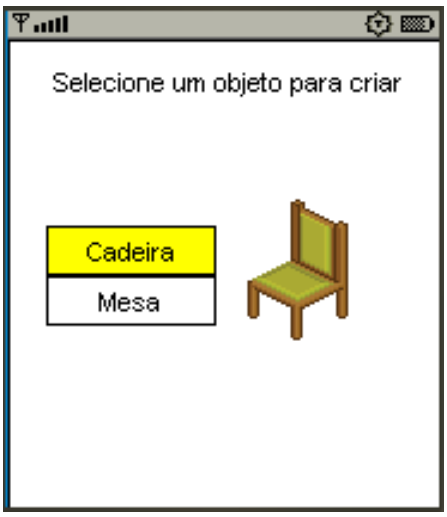
Trecho de código 41 – Sincronização do estado do jogo com o servidor

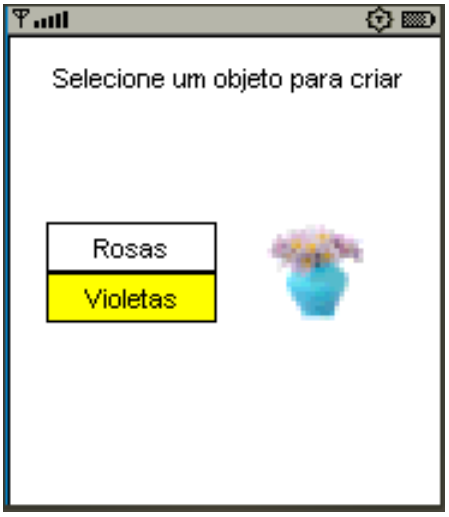
```
class JFServer : SocketServer, ConnectionListener {
(…)
private void UpdateUser(Connection c) {
	double cash = 0;
	String query = "SELECT `cash` FROM `Users` WHERE `name`=\"" +
		c.GetID() + "\"";
	DataRowCollection ds = msql.Select(query, "Users");
	if (ds != null && ds.Count > 0) {
		cash = (Double)ds[0].ItemArray[0];
	}

	query = "SELECT * FROM `Itens` WHERE `username`=\"" +
		c.GetID() + "\"";
	ds = msql.Select(query, "Itens");

	String itens = "";
	if (ds != null && ds.Count > 0) {
		object[] columns;
		for (int i = 0; i < ds.Count; i++) {
			columns = ds[i].ItemArray;
			String itemname = (String)columns[2];
			int itemtype = (Int32)columns[3];
			int status = (Int32)columns[4];
			double price = (Double)columns[5];

			if (i > 0)
				itens += "¥";
			itens += itemname + "§" + itemtype + "§" + status
				+ "§" + price;
		}
	}
	// Envia mensagem formatada para o usuário
	String response = cash + "#" + itens;
	c.SendMessage(new Message(UPDATE_STATE,
		Util.GetBytes(response)));
} (…)
```

Trecho de código 42 – Envio de mensagem de sincronização do servidor para o cliente

Os dados salvos localmente após a sincronização serão então consultados quando necessário.

Navegando para a tela de produção de itens, opção “Fazer” do menu, é apresentada ao usuário uma lista de objetos que ele poderá produzir, baseado na sua escolha de profissão. No caso do marceneiro, como exemplo, foram disponibilizados a produção de cadeiras e mesas. No caso do florista, um vaso de rosas e um de violetas, como mostram as figuras 31 e 32.





Figuras 31 e 32 – Seleção de objetos para produção de acordo com a profissão

Ao selecionar uma das opções de objetos para produção, o jogador é direcionado para um minigame, o qual deverá ser vencido para gerar o item, simulando a produção. No desenvolvimento dos minigames foram utilizadas extensões do componente *SimpleImage*, de forma a montar o cenário do jogo no formato desejado e de acordo com a lógica criada para cada um.

No *minigame* para a profissão de marceneiro, por exemplo, o jogador deve primeiro serrar uma tábua, apertando botões em sequência para mover o serrote. No segundo *minigame* o jogador deverá acompanhar o *timing* do martelo, que se move aleatoriamente na tela, para acertar o prego algumas vezes até alcançar o objetivo.

As figuras 33, 34, 35 e 36 mostram os minigames implementados para ambas as profissões.

Figuras 33 e 34 – *Minigames* para a profissão de marceneiro

Figuras 35 e 36 – *Minigames* para a profissão de florista

Ao produzir um objeto, o mesmo estará disponível na lista de objetos da cena de venda, onde o jogador poderá escolher o destino do objeto.

Para acessar a cena de vendas o jogador deverá selecionar a opção “Vender” do menu no quarto virtual. Essa cena mostra uma lista de botões customizados, criados a partir de uma extensão da classe *Container*, que mostram o nome do objeto, seu estado e valor. A figura 37 mostra a cena de venda de itens.

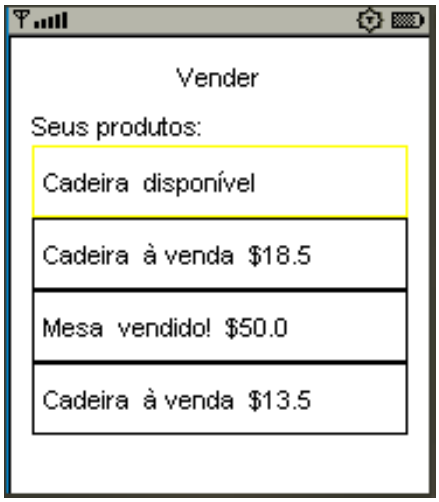


Figura 37 – Cena de venda de itens

Ao selecionar um dos objetos disponíveis para venda, o jogador é direcionado a uma tela com os detalhes do objeto e opções para colocá-lo à venda e definir um preço. Assim que um objeto é posto à venda, essa informação é enviada para o servidor, que a persistirá no banco de dados, para quando outros jogadores desejarem comprá-lo. Então, o estado do objeto é alterado para “à venda”, o que pode ser visto na lista de objetos mostrada na cena. Por fim, a

cena de vendas mostra na lista também os objetos que já foram vendidos e por qual valor, sem as opções para alterar o estado ou preço. A figura 38 mostra a cena de detalhes do objeto.

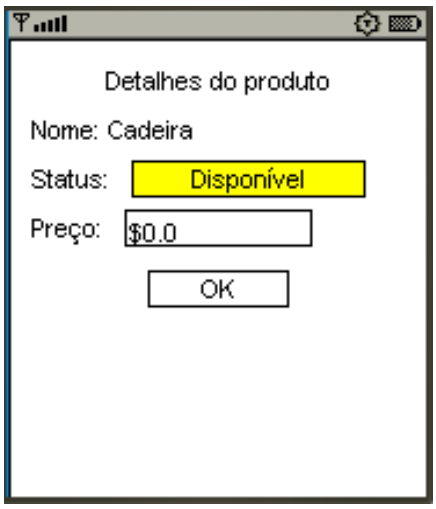


Figura 38 – Detalhes de item disponível para venda

O trecho de código 43 mostra a implementação, no servidor, do método que adiciona um item do usuário para venda.

```
class JFServer : SocketServer, ConnectionListener {
(…)
private void AddForSale(Message m, Connection c) {
	// Informações do item
	String msg = Util.GetString(m.GetData());
	String[] item = msg.Split('$');

	// Recupera o id do usuario
	String query = "SELECT `userId` FROM `Users` WHERE `name`=\""
		+ c.GetID() + "\"";
	DataRowCollection ds = msql.Select(query, "user");
	int userid;
	if (ds != null && ds.Count > 0)
		userid = (int)ds[0].ItemArray[0];

	// Insere os dados no banco
	query = "INSERT INTO `Itens` (`userId`, `itemName`, `type`,
		`status`, `price`) VALUES ("+userid+", "+item[0]+","+
		 item[1]+", "+item[2]+", "+item[3]+")";
	msql.Query(query);

	// Responde o cliente
	c.SendMessage(new Message(ADD_FOR_SALE,
		Util.GetBytes("success")));
}
```

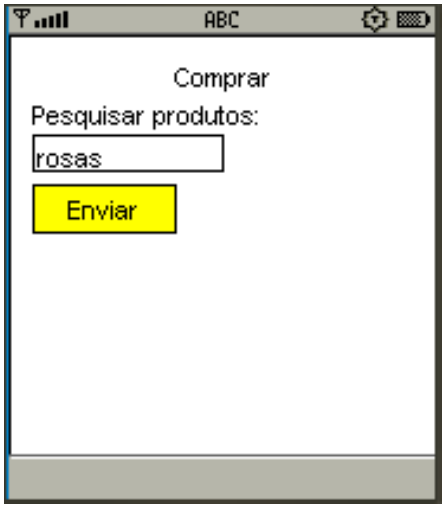
Trecho de código 43 – Adicionando itens à venda no banco de dados

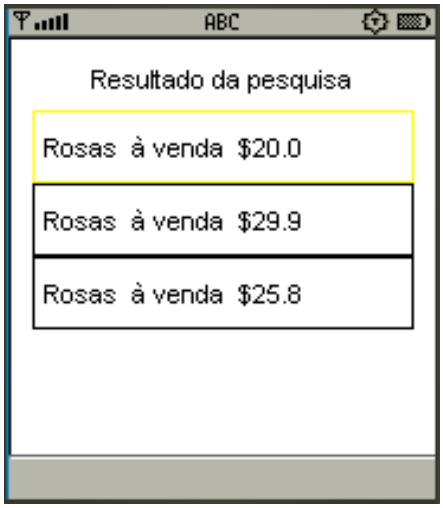
Entrando na cena de compra, há uma caixa de texto para pesquisa pelo nome do item e um botão para efetuar a pesquisa. Quando recebe a mensagem, o servidor faz uma busca no banco de dados e retorna o resultado formatado em uma mensagem de texto, como mostra o trecho de código 44.

```
class JFServer : SocketServer, ConnectionListener {
(…)
private void Search(Message m, Connection c) {
	String keyword = Util.GetString(m.GetData());

	// Faz a pesquisa no banco de dados
	String query = "SELECT * FROM `Itens` WHERE `name` LIKE '%" +
		keyword + "%'";
	DataRowCollection ds = msql.Select(query, "Itens");

	String itens = "";
	if (ds != null && ds.Count > 0) {
		object[] columns;
		for (int i = 0; i < ds.Count; i++) {
			columns = ds[i].ItemArray;
			int sellerId = (int)columns[1];
			String itemname = (String)columns[2];
			int itemtype = (int)columns[3];
			int status = (int)columns[4];
			double price = (int)columns[5];

			if (i > 0)
				itens += "¥";
			itens += sellerId + "§" + itemname + "§" +
				itemtype + "§" + status + "§" + price;
		}
	}
	// Envia mensagem de texto formatada com o resultado
	c.SendMessage(new Message(SEARCH, Util.GetBytes(itens)));
}
(…)
```

Trecho de código 44 – Método do servidor para pesquisa de itens à venda

Ao receber a resposta, o jogo mostra uma lista com os itens disponíveis para compra. As figuras 39 e 40 mostram as cenas de pesquisa e resultados para compra, respectivamente.





Figuras 39 e 40 – Cenas de pesquisa e resultados para compra de itens

Ao selecionar um deles é apresentada uma cena com os detalhes do item e um botão com a opção de comprar. Quando um item é comprado o servidor altera o estado do item no banco de dados, debita o valor do saldo do comprador, e credita ao vendedor. Após receber a confirmação da compra, o objeto é adicionado à lista de itens do comprador.

A figura 41 mostra a cena com detalhes da compra.

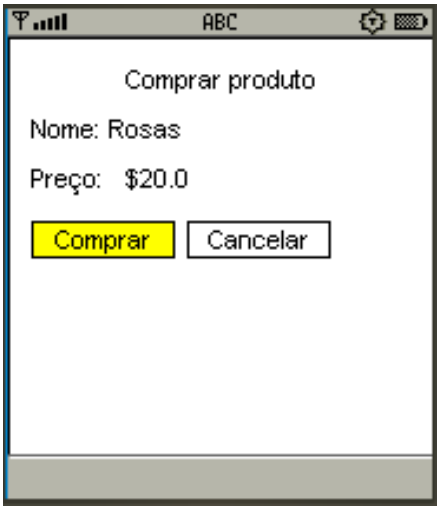


Figura 41 – Cena de detalhes para compra de itens

O trecho de código 45 mostra o algoritmo de compra de itens no servidor.

```
class JFServer : SocketServer, ConnectionListener {
(…)
private void Buy(Message m, Connection c) {
      String msg = Util.GetString(m.GetData());
      String[] item = msg.Split('§');

      int itemId = Int32.Parse(item[0]);
      double price = Double.Parse(item[5]);
      int sellerId = Int32.Parse(item[1]);
      int buyerid = -1;
      double buyerCash = -1;

      // Recupera o id do comprador
      String query = "SELECT `userId`, `cash` FROM `Users` WHERE
            `name`=\"" + c.GetID() + "\"";
      DataRowCollection ds = msql.Select(query, "user");

      if (ds != null && ds.Count > 0) {
            buyerid = (int)ds[0].ItemArray[0];
            buyerCash = (double)ds[0].ItemArray[1];
      }
      if (buyerid >= 0) {
            if (buyerCash > price) {
                  // Credita o valor no saldo do vendedor
                  query = "UPDATE `Users` SET `cash` =
                        (`cash`+" +price+ ") WHERE `Users`.`user_id`
                              = " +sellerId;
                  msql.Query(query);

                  // Devita o valor no saldo do comprador
                  query = "UPDATE `Users` SET `cash` = (`cash`-"
                        +price+ ") WHERE `Users`.`user_id` = " +
                              buyerid;
                  msql.Query(query);

                  // Altera o status do item para vendido
                  query = "UPDATE `Itens` SET `status` = '2' WHERE
                        `Itens`.`item_id` = " + itemId;
                  msql.Query(query);

                  // Notifica o cliente do sucesso da compra
                  c.SendMessage(new Message(BUY,
                        Util.GetBytes("success")));
            } else {
                  // Notifica o cliente de que nao possui
                  // saldo suficiente
                  c.SendMessage(new Message(BUY,
                        Util.GetBytes("nocash")));
            }
      } else {
            //Notifica o cliente de que houve problema na transacao
            c.SendMessage(new Message(BUY,
                  Util.GetBytes("failed")));
      }
} (…)
```

Trecho de código 45 – Algoritmo de compra de itens no servidor

6 Conclusão

Como resultado desse trabalho, foi apresentado o projeto e implementação de um *framework* para desenvolvimento de jogos com múltiplos jogadores para celular através da internet, um *framework* para servidores que possam suportar esses jogos, com as funções de comunicação e persistência utilizadas e, por fim, o desenvolvimento de dois jogos desse tipo.

Os objetivos do trabalho foram alcançados, abordando o contexto atual do desenvolvimento de jogos para celular e alguns dos problemas e dificuldades encontradas nessa área. Foram apresentados alguns fatos que dificultam a aceitação desses aparelhos como plataformas para jogos, principalmente problemas de usabilidade, complexidade e distribuição. Foi feita uma análise das projeções acerca da internet móvel, visando sua utilização como meio para o desenvolvimente de jogos multi-jogadores para celular e também uma análise de ferramentas existentes para o desenvolvimento de jogos com esssas características.

Assim, pudemos observar a possibilidade de levar aos aparelhos celulares que disponibilizam a plataforma J2ME, um modelo de jogos bastante difundido na internet. Devido à simplicidade e abrangência desse tipo de jogos acredita-se que poderiam facilmente ser adaptados para esses dispositivos e atingir um grande público.

O *framework* desenvolvido apresenta funcionalidades básicas de jogos implementadas para a plataforma J2ME, tais como controle do ciclo de renderização e atualização do jogo, controle de cenas, componentes de interface, controle de eventos e comunicação através da internet. Podendo ser utilizado para implementação de jogos com essas funcionalidades e facilmente estendido para servir a vários tipos de jogos.

O *framework* desenvolvido para o servidor apresenta funcionalidades comuns a jogos multi-jogadores, com ambientes virtuais de interação, que podem ser especializados de acordo com a dinâmica de cada jogo, e integração com banco de dados, para persistência de dados do usuário possibilitando inúmeras aplicações.

Ao final, nem todos os aspectos apresentados pela análise foram abordados de forma prática, tais como tolerância a falhas na rede móvel e distribuição comercial de jogos para celular. Esses aspectos não devem ser ignorados, pelo contrário, definem de forma decisiva a viabilidade comercial de um projeto como esse, porém, para os fins acadêmicos a que se destina esse trabalho, o resultado foi considerado satisfatório.

6.1 Trabalhos futuros

A partir da arquitetura projetada para o *framework*, é possível estendê-lo para atender aos requisitos de vários tipos de jogos, incluindo aqueles abordados pelos *frameworks* analisados no capítulo quatro. Pode-se também adicionar funcionalidades que não foram implementadas como tolerância a falhas, comunicação via HTTP e integração com redes sociais.

Futuramente é possível que a internet móvel e os aparelhos celulares evoluam a ponto de possibilitar o desenvolvimento de jogos multi-jogadores em tempo real. O *framework* apresentado também pode ser utilizado para criar jogos desse tipo. Ou ainda, pode-se adicionar suporte a conexões *bluetooth*, para jogos em que seja interessante a proximidade entre os jogadores.

Outro tema interessante é a implementação de jogos *cross-platform*, que significa jogadores participando de um mesmo jogo a partir de plataformas diferentes. Como por exemplo, um jogo que apresenta uma versão *web*, acessada através de um navegador de internet, e também uma versão para celulares, integradas para que os jogadores possam interagir dentro de um mesmo jogo indepentemente da plataforma.

O desenvolvimento de jogos multi-jogadores para celular poderia ser utilizado também no âmbito acadêmico, pois envolve diversas tecnologias e conceitos abordados em sala de aula como engenharia de software, redes de computadores e computação distribuída, de forma que poderiam enriquecer e estimular o aprendizado.

7 Referências Bibliográficas

ABIN. (27 de maio de 2009). *Brasil supera a média mundial de computadores por habitante*. Acesso em novembro de 2009, disponível em Abin: http://www.abin.gov.br/modules/articles/article.php?id=4407

Anatel. (janeiro de 2010). *Quantidade de Acessos/Plano de Serviço/Unidade da Federação - Janeiro/2010*. Acesso em março de 2010, disponível em http://sistemas.anatel.gov.br/SMP/Administracao/Consulta/AcessosPrePosUF/tela.asp

Apple. (2010). *iPhone Dev Center*. Acesso em maio de 2010, disponível em Apple Developer: http://developer.apple.com/iphone/index.action

Barth, P. (2009, outubro 30). Colóquios Cyclops. *Mobile Applications are (yet again?) taking off* . UFSC; Florianópolis.

Blizzard. (21 de novembro de 2008). *WORLD OF WARCRAFT® SUBSCRIBER BASE REACHES 11.5 MILLION WORLDWIDE*. Acesso em outubro de 2009, disponível em http://us.blizzard.com/en-us/company/press/pressreleases.html?081121

Bombonatti, P. (18 de agosto de 2009). *Mercado de mobile games terá crescimento significante*. Acesso em abril de 2010, disponível em MobilePedia: http://www.mobilepedia.com.br/noticias/mercado-de-mobile-games-tera-crescimento-significante

Cashmore, P. (março de 2010). *FarmVille Surpasses 80 Million Users*. Acesso em abril de 2010, disponível em Mashable: http://mashable.com/2010/02/20/farmville-80-million-users

Cecin, F., & Trinta, F. (9 de novembro de 2007). *Jogos Multiusuário Distribuídos.* Acesso em novembro de 2009, disponível em VI Brasilian Synposium on Computer Games and Digital Entertainment: http://www.inf.unisinos.br/~sbgames/anais/tutorials/Tutorial1.pdf

CETIC. (2009). *TIC DOMICÍLIOS e USUÁRIOS - Pesquisa sobre o Uso das Tecnologias da Informação e da Comunicação no Brasil.* Acesso em abril de 2010, disponível em Cetic.br: http://www.cetic.br/usuarios/tic/2009/analise-tic-domicilios2009.pdf

Chone, J. (8 de junho de 2009). *Myths about JavaFX, Android, and J2ME*. Acesso em abril de 2010, disponível em Bits & Buzz: http://www.bitsandbuzz.com/article/myths-about-javafx-android-and-j2me/

Cisco. (10 de fevereiro de 2009). *Newest Cisco Visual Networking Index Forecast Highlights Growth in Mobile Traffic.* Acesso em 2009 de novembro, disponível em Cisco: http://newsroom.cisco.com/dlls/2009/prod_021009d.html

DeviceAtlas. (2010). *Data Explorer*. Acesso em maio de 2010, disponível em DeviceAtlas: http://deviceatlas.com/explorer#_/

Doolittle, M. (26 de agosto de 2007). *Wii and the casual gaming boom*. Acesso em março de 2010, disponível em Game Critics: http://www.gamecritics.com/wii-and-the-casual-gaming-boom

EA Mobile. (s.d.). *Mobile Games, iPhone Games, Cell Phone Games*. Acesso em novembro de 2009, disponível em EA Mobile: http://www.eamobile.com/home/

EclipseME.org. (2005). Acesso em maio de 2010, disponível em http://eclipseme.org/

EDGE. (5 de março de 2008). *New Stats Show Casual Explosion*. Acesso em novembro de 2009, disponível em EDGE: http://www.edge-online.com/features/new-stats-show-casual-explosion

Ferrari, B. (22 de fevereiro de 2010). *O engarrafamento celular*. Acesso em março de 2010, disponível em Época: http://revistaepoca.globo.com/Revista/Epoca/0,,EMI122764-15224,00-O+ENGARRAFAMENTO+CELULAR.html

Friedmann, D. (8 de novembro de 2009). *Multiplayer Game Models*. Acesso em dezembro de 2009, disponível em http://proj-mgt.com/wordpress/

Games Industry. (21 de maio de 2009). *NPD: More Americans play games than go to movies*. (D. Jenkis, Editor) Acesso em novembro de 2009, disponível em Games Industry: http://www.gamesindustry.biz/articles/npd-more-americans-play-games-than-go-to-movies

Gandra, A. (3 de março de 2009). *Mercado de jogos eletrônicos cresce no Brasil apesar da crise*. Acesso em novembro de 2009, disponível em Inovação tecnológica: http://www.inovacaotecnologica.com.br/noticias/noticia.php?artigo=mercado-de-jogos-eletronicos-cresce-no-brasil-apesar-da-crise&id=

GetJar. (2010). Acesso em abril de 2010, disponível em GetJar: http://www.getjar.com/

Gettler, J. (s.d.). *The first video game?* Acesso em março de 2010, disponível em Brookhaven National Laboratory: http://www.bnl.gov/bnlweb/history/higinbotham.asp

Google. (2010). *Android Developers*. Acesso em maio de 2010, disponível em Android Developers: http://developer.android.com/index.html

GSM Association. (s.d.). *About Us*. Acesso em novembro de 2009, disponível em GSM World: http://www.gsmworld.com/about-us/index.htm

GSM Association. (s.d.). *GSM History*. Acesso em novembro de 2009, disponível em GSM World: http://www.gsmworld.com/about-us/history.htm

Handango. (2010). Acesso em abril de 2010, disponível em Handango: http://www.handango.com/homepage/Homepage.jsp?storeId=2218

Herman, L., Horwitz, J., Kent, S., & Miller, S. (s.d.). *The history of video games*. Acesso em maio de 2010, disponível em GameSpot: http://www.gamespot.com/gamespot/features/video/hov/index.html

Huawei/Teleco. (dezembro de 2009). *Balanço Huawei da Banda Larga Móvel.* Acesso em março de 2010, disponível em Huawei: http://www.huawei.com/pt/catalog.do?id=1779

Infonetics Research. (3 de novembro de 2009). *Mobile subscribers to hit 5.9 billion in 2013, driven by China, India, Africa*. Acesso em novembro de 2009, disponível em Infonetics: http://www.infonetics.com/pr/2009/Fixed-and-Mobile-Subscribers-Market-Highlights.asp

Ingram, M. (17 de fevereiro de 2010). *Average Social Gamer Is a 43-Year-Old Woman*. Acesso em março de 2010, disponível em Gigaom: http://gigaom.com/2010/02/17/average-social-gamer-is-a-43-year-old-woman

Knowles, J. (12 de setembro de 2007). *Thought on Mobile Social Gaming*. Acesso em março de 2010, disponível em Auscillate: http://www.auscillate.com/post/118

Long, K. (6 de março de 2007). *GDC Mobile - The Future of Mobile Games Will be Social*. Acesso em abril de 2010, disponível em Future Lab: http://www.futurelab.net/blogs/marketing-strategy-innovation/2007/03/gdc_mobile_the_future_of_mobil.html

MacInnes, F. (s.d.). *Ten reasons why mobile games are completely awesome*. Acesso em abril de 2010, disponível em PocketGamer: http://www.pocketgamer.co.uk/r/Mobile/Top+10+mobile+charts/feature.asp?c=5402

Melanson, D. (3 de março de 2006). *A Brief History of Handheld Video Games*. Acesso em maio de 2010, disponível em Engadget: http://www.engadget.com/2006/03/03/a-brief-history-of-handheld-video-games/

Microsoft Corporation. (2010). *Windows Mobile Developer Center*. Acesso em maio de 2010, disponível em http://msdn.microsoft.com/en-us/windowsmobile/default.aspx

MySQL Workbench. (2010). Acesso em junho de 2010, disponível em MySQL Developer Zone: http://dev.mysql.com/

Nokia. (2009). *SNAP Mobile Javadocs*. Acesso em novembro de 2009, disponível em Nokia Forum: http://americas.forum.nokia.com/info/sw.nokia.com/id/a3cc82ac-d774-43ea-84fb-d0fa326b5c45/SNAP_Mobile_Client_SDK_Javadocs_v2_0_en.zip.html

Nokia. (7 de abril de 2010). *SNAP mobile sdk and its Game*. (M. Lumivuori, Editor) Acesso em abril de 2010, disponível em Nokia Forum: http://www.forum.nokia.com/forum/showthread.php?p=722653

O Estado de S. Paulo. (5 de maio de 2008). *'GTA IV' leva games ao topo da cultura pop*. (A. Matias, & J. Auricchio, Editores) Acesso em novembro de 2009, disponível em Estadão: http://www.estadao.com.br/noticias/tecnologia+link,gta-iv-leva-games-ao-topo-da-cultura-pop,1923,0.shtm

O Estado de S. Paulo. (8 de junho de 2009). *Todo mundo joga videogame*. (J. Auricchio, R. Martins, & H. Lupin, Editores) Acesso em novembro de 2009, disponível em Estadão: http://www.estadao.com.br/noticias/tecnologia+link,todo-mundo-joga-videogame,2597,0.shtm

Oliveira, L. P. (agosto de 2005). *Os dispositivos móveis e as redes peer-to-peer (P2P).* Acesso em maio de 2010, disponível em Universidade Federal de Pernambuco: http://www.cin.ufpe.br/~tg/2005-1/lpo.pdf

Open Mobile Alliance. (2002). *What is WAP?* Acesso em novembro de 2009, disponível em Open Mobile Alliance: http://www.wapforum.org/what/index.htm

Oracle. (2010). *Java ME: the Most Ubiquitous Application Platform for Mobile Devices*. Acesso em maio de 2010, disponível em Sun Developer Network (SDN): http://java.sun.com/javame/index.jsp

Powers, M. (novembro de 2006). *Mobile Multiplayer Gaming, Part 1: Real-Time Constraints*. Acesso em outubro de 2009, disponível em Sun Developer Network (SDN): http://developers.sun.com/mobility/midp/articles/gamepart1/

Qualcomm. (2010). *Qualcomm Brew*. Acesso em maio de 2010, disponível em Qualcomm Brew: http://brew.qualcomm.com/brew/en/

Research In Motion. (2010). *Blackberry Developer Zone*. Acesso em maio de 2010, disponível em Blackberry Developer Zone: http://na.blackberry.com/eng/developers/

Rieger, B. (fevereiro de 2009). *Effective Design for Multiple Screen Sizes*. Acesso em abril de 2010, disponível em mobiForge: http://mobiforge.com/designing/story/effective-design-multiple-screen-sizes

Sun Microsystems. (2006). *Overview (MIDProfile)*. Acesso em outubro de 2009, disponível em http://java.sun.com/javame/reference/apis/jsr118/

Symbian Foundation. (2010). *The Symbian Foundation Community*. Acesso em maio de 2010, disponível em The Symbian Foundation Community: http://www.symbian.org/

Teleco. (15 de março de 2010). *Estatísticas de celular no mundo*. Acesso em novembro de 2009, disponível em Teleco: http://www.teleco.com.br/pais/celular.asp

Teleco. (25 de abril de 2010). *Estatísticas de celulares e redes 3G no mundo (WCDMA/HSDPA e EVDO)*. Acesso em abril de 2010, disponível em Teleco: http://www.teleco.com.br/pais/3g.asp

Teleco. (15 de março de 2010). *Estatísticas de celulares no mundo, ranking de países, operadores e tecnologias*. Acesso em março de 2010, disponível em Teleco: http://www.teleco.com.br/pais/celular.asp

Teleco. (27 de janeiro de 2008). *Seção: Telefonia Celular*. Acesso em novembro de 2009, disponível em Teleco: http://www.teleco.com.br/tecnocel.asp

The Eclipse Foundation. (2010). *Eclipse*. Acesso em maio de 2010, disponível em http://www.eclipse.org/

TIM. (outubro de 2009). *Serviços > Internet > Internet no Celular > Quanto custa*. Acesso em outubro de 2009, disponível em TIM: http://www.tim.com.br/portal/site/PortalWeb/

Todo mundo quer ter sua própria App Store. (9 de março de 2010). Acesso em maio de 2010, disponível em ArchTec Blog: http://archtec.wordpress.com/2010/03/09/todo-mundo-quer-ter-sua-propria-app-store/

Tsirulnik, G. (19 de abril de 2010). *Smartphones shift US mobile gaming market landscape*. Acesso em maio de 2010, disponível em Mobile Marketer: http://www.mobilemarketer.com/cms/news/research/6006.html

What future for J2ME? (17 de dezembro de 2008). Acesso em março de 2010, disponível em TomSoft Blog: http://blog.landspurg.net/what-future-for-j2me/

Wilson, M. (27 de março de 2002). *The Realities of Deploying Wireless J2ME Solutions Over Unreliable Networks*. Acesso em abril de 2010, disponível em O'Reilly: http://tim.oreilly.com/pub/a/onjava/2002/03/27/j2me.html

Wright, C. (22 de dezembro de 2008). *A Brief History of Mobile Games*. Acesso em agosto de 2009, disponível em Pocket Gamer: http://www.pocketgamer.biz/r/PG.Biz/A+Brief+History+of+Mobile+Games/feature.asp?c=10618

XAMPP. (Maio de 2010). Acesso em junho de 2010, disponível em Apache Friends: http://www.apachefriends.org/

Anexo I – Manual de utilização do *Framework*

Esse manual tem como objetivo apresentar exemplos da utilização do *framework* desenvolvido nesse trabalho.

Serão abordados todos os módulos implementados para desenvolvimento de jogos em J2ME e também para um servidor que suporte esses jogos e a comunicação entre eles.

I.1 GameCore

A classe *GameCore* é o centro do jogo, onde ocorre o ciclo principal. É responsável pela execução dos métodos *paint* e *update* que devem ser implementados de acordo com as necessidades do jogo.

O trecho de código I.1 mostra um exemplo da utilização do *GameCore*.

```
public class HelloWorld extends GameCore{

	// Implementação do método de renderização
	protected void paint(Graphics g) {
		// Desenha o texto "Olá Mundo" nas posições definidas
		g.drawString("Olá Mundo", x, y, 0);
	}

	// Implementação do método de atualização
	protected void update() {
		// Atualiza as posições x e y de acordo com as
		// teclas de navegação pressionadas
		if(Key.pressed(Key.K_UP)) y-=2;
		else if(Key.pressed(Key.K_DOWN)) y+=2;
		if(Key.pressed(Key.K_LEFT)) x-=2;
		else if(Key.pressed(Key.K_RIGHT)) x+=2;
	}
}
```

Trecho de código I.1 – Exemplo da utilização da classe *GameCore*

Esse exemplo mostra na tela a frase "Olá Mundo", que se move de acordo com o pressionamento das teclas para cima, para baixo e para os lados, como mostra a figura I.1.

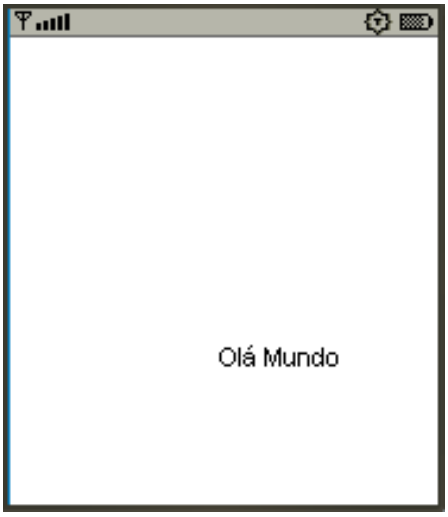


Figura I.1 – Resultado do exemplo da utilização da classe *GameCore*

I.2 O controle de cenas

O controle de cenas foi implementado para facilitar a criação e gerenciamento de várias cenas e objetos que possam vir a ser criados para os jogos. Para isso foram criadas a classe *Scene*, correspondendo a uma cena qualquer do jogo, como menu principal ou tela de jogo, e a classe *GameObject*, representando quaisquer objetos que possam aparecer em uma cena do jogo. Para criar os objetos do jogo é necessário então estender a classe *GameObject*, implementando os métodos *paint* e *update*.

Como exemplo será feita uma especialização da classe *GameObject*, representando um círculo colorido com movimentação aleatória. Será então criada uma cena que carrega vários desses objetos, com velocidades e cores variadas e, por fim, uma extensão da classe *Game* que seleciona essa cena em sua inicialização. Essa implementação é mostrada no trecho de código a seguir.

```
public class BallsGame extends Game {

	// Método de inicialização do jogo
	protected void loadGame() {
		// Seta uma nova BallScene como cena atual
		pushScene(new BallScene());
	}

	// Implementação da cena BallScene
	public class BallScene extends Scene {
		// Método para carregamento da cena
		public void load() {
			// Gera 100 objetos do tipo Ball com
			// posição, cor e velocidades aleatórias
			// e os adiciona à cena
			for (int i = 0; i < 100; i++) {
				(...)
				add(new Ball(startx, starty, accx, accy,
						ballColor));
			}
		}
```

```
            // Método de descarregamento da cena
            public void unload() {
                  // Remove todos os objetos que a cena contém
                  removeAll();
            }
      }

      // Implementação da classe Ball
      public class Ball extends GameObject {
            // Atributos de posição, aceleração e cor
            private int x, y, accx, accy, color;

            // Construtor da classe
            public Ball(int startx, int starty, int accx, int accy,
                        int color) {…}

            // Seleciona a cor e desenha um círculo
            public void paint(Graphics g) {
                  g.setColor(color);
                  g.fillArc(x, y, 20, 20, 0, 360);
            }

            // Atualiza posição de acordo com a aceleração
            public void update() {
                  x += accx;
                  y += accy;
                  (…)
            }
      }
}
```

Trecho de código I.2 – Exemplo de utilização do pacote *game.scene*

O resultado desse exemplo é mostrado na figura I.2.

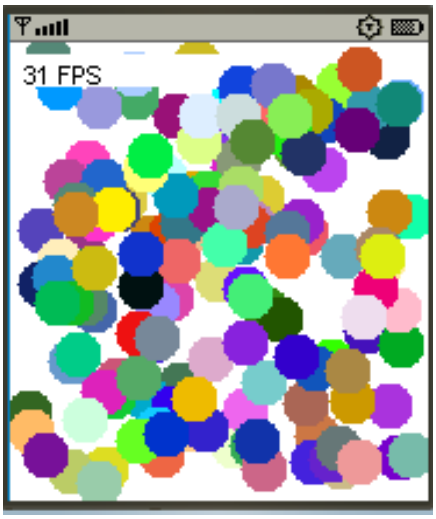


Figura I.2 – Imagem do exemplo que utiliza o pacote *game.scene*

I.3 Os componentes de interface

A partir da classe *GameObject* foram criados componentes de interface, comuns em qualquer jogo ou aplicação, como botões, caixas de texto, etc.

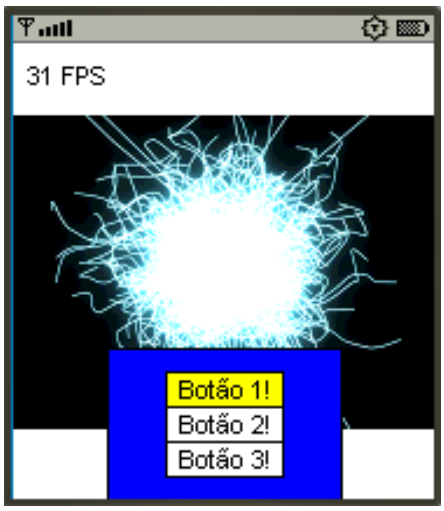
Será apresentado um exemplo de uso do pacote de componentes, onde é criada uma cena contendo uma imagem de fundo e um contêiner, representando um menu com três botões. O primeiro botão tem como ação adicionar outro contêiner à cena, o qual contém um componente do tipo *Label* e um *Inputlabel*. Selecionando o segundo botão do menu o segundo contêiner é removido, e selecionando o terceiro botão, o jogo passa à cena dos círculos, apresentada na seção de implementação do gerenciamento de cenas.

O trecho de código I.3 mostra a implementação desse exemplo.

```
// Implementação da cena
public class GUISceneExample extends Scene {

	// Containers utilizados
	SimpleContainer c1, c2;

	// Carregamento da cena
	public void load() {
		// Imagem de fundo usando SimpleImage
		SimpleImage back = new SimpleImage(…);
		// Alinha imagem ao centro
		back.alignToDisplay(Component.VCENTER +
			Component.HCENTER);
		// Instancia o segundo container
		c2 = new SimpleContainer(120, 50, Color.GREY);
		// Cria uma Label e seta sua posição
		Label lbl = new Label("botão 1 pressionado!");
		lbl.x = 10;
		// Adiciona ao contêiner
		c2.add(lbl);
		// Cria uma InputLabel adiciona ao contêiner
		c2.add(new InputLabel(80));

		// Instancia o primeiro container
		c1 = new SimpleContainer(100, 65, Color.BLUE);
		// Cria um botão com o texto "Botão 1!"
		Button b1 = new Button("Botão 1!", 50, 15,
			new Runnable() {
				public void run() {
					add(c2);
				}
		});

		(…)
		c1.add(new Button("Botão 2!", 50, 15, …);
		c1.add(new Button("Botão 3!", 50, 15, …);

		add(c1);
		c1.setFocused(b1);
	}
	(…)
```

Trecho de código I.3 – Exemplo de uso do pacote de componentes

O resultado é mostrado nas figuras a seguir.

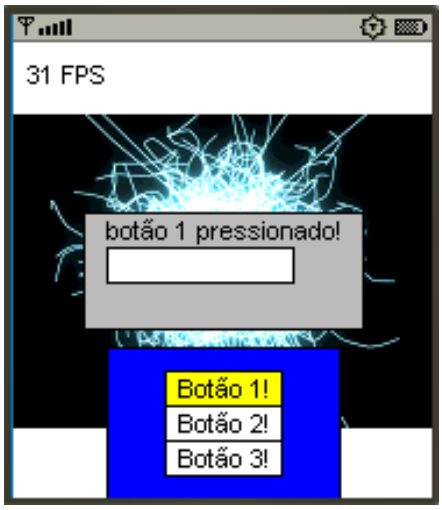


Ao iniciar o aplicativo, podemos ver o componente *SimpleImage* carregado ao fundo.

O contêiner azul (c1), representando um menu, com três botões, sendo que o primeiro botão possui o foco.

Podemos verificar o alinhamento do contêiner, embaixo e centralizado horizontalmente em relação à tela, e dos botões, automaticamente um embaixo do outro.

Figura I.3 – Resultado do exemplo de uso do pacote componentes



Ao selecionarmos o botão 1, um novo contêiner de cor cinza (c2) é adicionado. Dentro dele há uma *Label* com o texto "botão 1 pressionado!" e uma *InputLabel* para inserção de texto.

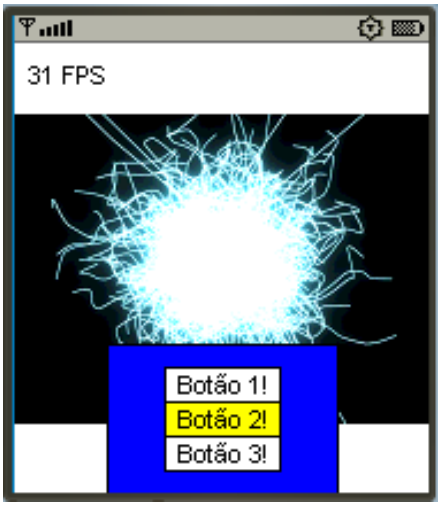
Figura I.4 – Resultado do exemplo de uso do pacote componentes

Ao selecionarmos a *InputLabel*, o aplicativo mostra uma tela padrão para inserção de texto do próprio sistema do aparelho.

Figura I.5 – Resultado do exemplo de uso do pacote componentes

Após a inserção do texto, o aplicativo nos leva de volta à sua interface, já com o texto inserido na *InputLabel*.

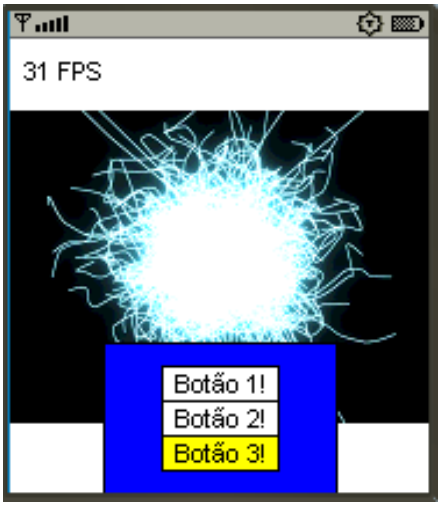
Figura I.6 – Resultado do exemplo de uso do pacote componentes



Ao pressionarmos a tecla de navegação para baixo, podemos verificar que o foco passa para o botão 2, que muda de cor, assim como o botão 1, que perde o foco.

Ao selecionarmos o botão 2 removemos o contêiner cinza do centro da tela.

Figura I.7 – Resultado do exemplo de uso do pacote componentes



Podemos então navegar para o botão de número 3.

Figura I.8 – Resultado do exemplo de uso do pacote componentes

E ao pressionarmos a tecla de seleção com o foco no botão 3 ele nos leva à cena com círculos coloridos, apresentada anteriormente.

I.4 Gerenciamento de eventos

A seguir, será feita uma demonstração utilizando as classes implementadas para o gerenciador de eventos.

Foi criada uma cena na qual o jogador é representado por um círculo colorido que se movimenta de acordo com as teclas de navegação. Há também três alvos, representados por quadrados coloridos, que interagem com o jogador, atribuindo a ele suas cores. Para isso os alvos verificam, durante o método de atualização, se o personagem do jogador está colidindo com eles e em caso positivo, disparam um evento previamente definido, através da classe *EventManager*. O jogador, por sua vez, implementa a *interface EventListener* e trata os eventos recebidos, alterando a sua cor de acordo com o alvo que atinge.

O trecho de código I.4 mostra a implementação desse exemplo.

```
public class EventExample extends Game {

	//Tipos de eventos
	public final int BLUE_GOAL = 10, RED_GOAL = 11,
		GREEN_GOAL = 12;
	(…)
	// Implementação da cena
	public class EventExampleScene extends Scene {
		// Inicialização da cena
		public void load() {
			add(player);
			add(new Goal(50, 120, RED_GOAL));
			add(new Goal(150, 80, BLUE_GOAL));
			add(new Goal(80, 30, GREEN_GOAL));
		}
	}

	// Classe do personagem controlado pelo jogador
	public class Player extends GameObject implements
		EventListener {

		// Construtor
		public Player() {
			// Registrando-se como ouvinte
			EventManager.register(this, BLUE_GOAL);
			EventManager.register(this, RED_GOAL);
			EventManager.register(this, GREEN_GOAL);
		}
		(…)

		// Método para o tratamento dos eventos, alterando sua
		// cor de acordo com o tipo de evento
		public void handleEvent(Event e) {
			switch (e.type) {
			case BLUE_GOAL:   color = Color.BLUE;break;
			case GREEN_GOAL:  color = Color.GREEN;break;
			case RED_GOAL:    color = Color.RED;break;
			}
		}
```

```
        }

        public class Goal extends GameObject {
                (…)
                public Goal(int x, int y, int goaltype) {…}

                public void update() {
                        if (player.getBounds().intersects(
                                this.getBounds()))
                                EventManager.dispatch(new Event(goaltype));
                }
        }
}
```

Trecho de código I.4 – Exemplo de uso do gerenciamento de eventos

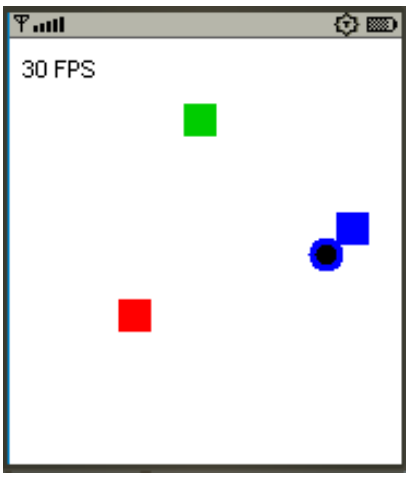
O resultado é mostrado nas figuras a seguir.

Podemos verificar o personagem do jogador, círculo de cor amarela, em sua posição inicial. E os alvos, quadrados, nas cores vermelha, verde e azul.

Figura I.9 – Resultado do exemplo de uso do gerenciamento de eventos

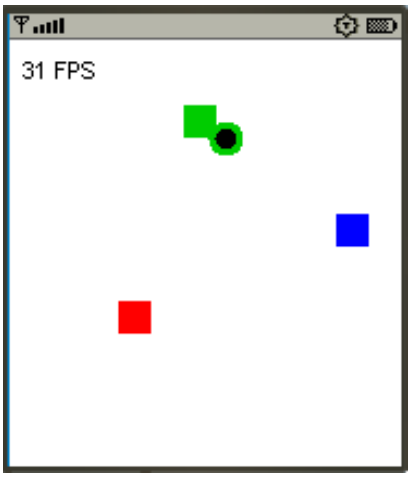
Nessa figura, o personagem do jogador foi movimentado. Ao encostar no alvo vermelho podemos perceber a ação do evento disparado, fazendo com que o jogador adquira a cor do alvo atingido.`

Figura I.10 – Resultado do exemplo de uso do gerenciamento de eventos



A seguir, o jogador foi movimentado até o próximo alvo, adquirindo a cor azul.

Figura I.11 – Resultado do exemplo de uso do gerenciamento de eventos



E por último, a cor verde.

Figura I.12 – Resultado do exemplo de uso do gerenciamento de eventos

É claro que esse exemplo poderia ser implementado de outra forma, sem o auxílio do gerenciador de eventos, mas seu objetivo é apenas demonstrar a funcionalidade do sistema. Em jogos complexos, uma forma centralizada e simples de notificar vários elementos sem que haja uma relação direta entre eles pode ser muito útil.

I.5 Comunicação cliente/servidor

Nesse tópico será apresentado um exemplo de implementação da comunicação entre o servidor e cliente, utilizando os *frameworks* desenvolvidos. O exemplo consiste em um pedido de conexão feito pelo cliente, que ao ser recebido pelo servidor é encaminhado para uma sala, extensão da classe *Lobby*. Essa sala então envia uma mensagem de boas vindas ao cliente, que responde em seguida.

O trecho de código I.5 mostra a implementação desse exemplo no lado cliente.

```
public class ClientExample extends GameCore implements
      EventListener {
      // Tipo da mensagem
      public static final int TEXT_MESSAGE = 0;

      protected void loadGame() {
            client = new MessageClient();
            EventManager.register(TEXT_MESSAGE, this);
            client.connect("127.0.0.1:12345");
      }

      // Trata as mensagens e eventos recebidas
      public void handleEvent(Event e) {
            if (e.type == TEXT_MESSAGE) {
                  // Grava a mensagem recebida e responde
                  servermsg = "Mensagem do servidor: "
                        +((Message)e).getString();
                  client.sendMessage(new Message(TEXT_MESSAGE,
                              "Ola servidor."), this);
            }
```

Trecho de código I.5 –Implementação do lado cliente do exemplo.

O trecho de código a seguir mostra a implementação do lado servidor do exemplo.

```
class ServerExample : SocketServer{
      // Sala de exemplo
      private LobbyExample lobby = new LobbyExample();

      // Inicialização do servidor
      public ServerExample() : base(12345){}

      // Cria uma SocketConnection e adiciona à sala
      protected override void ConnectionAccepted(Socket socket){
            client = new SocketConnection("cid", socket);
            lobby.Add(client);
      }
}

class LobbyExample : Lobby{

      // Tipo da mensagem utilizada
      public const int TEXT = 0;

      // Tratamento da mensagem recebida
      public override void MessageReceived(Message msg,
            Connection from){
            // Mostra a mensagem do cliente no console
            if(msg.GetType() == TEXT)
                  Console.WriteLine("Mensagem do cliente: "
                        + Util.GetString(msg.GetData()));
      }
      // Trata a entrada de novas conexões
      public override void ConnectionJoinCallback(Connection c){
            c.SendMessage(new Message(TEXT, "Ola cliente."));
      }( )
```

Trecho de código I.6 –Implementação do lado servidor do exemplo.

As figuras 54, 55 e 56 mostram o resultado da implementação desse exemplo.

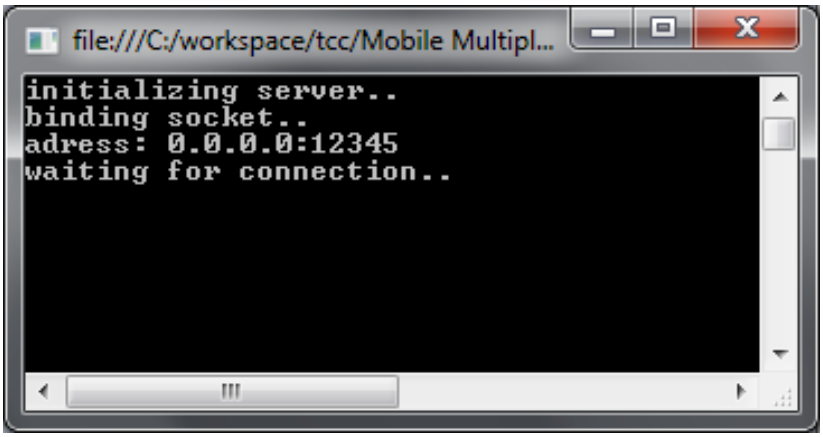


Figura I.12 – Inicialização do servidor

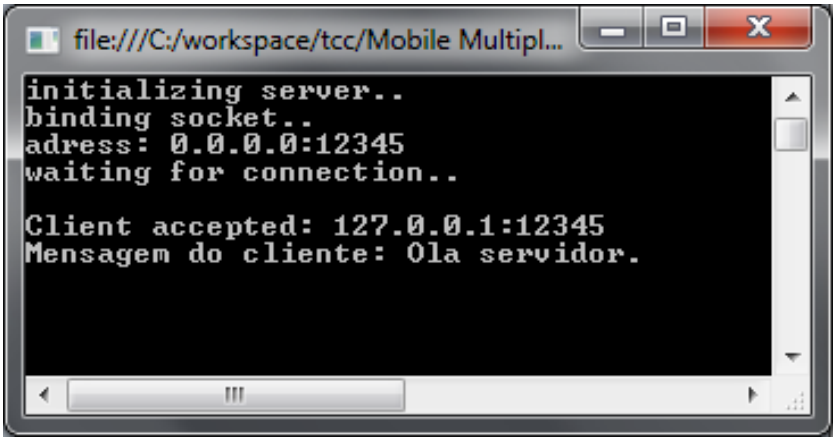


Figura I.13 – Mensagem do cliente, recebida pelo servidor

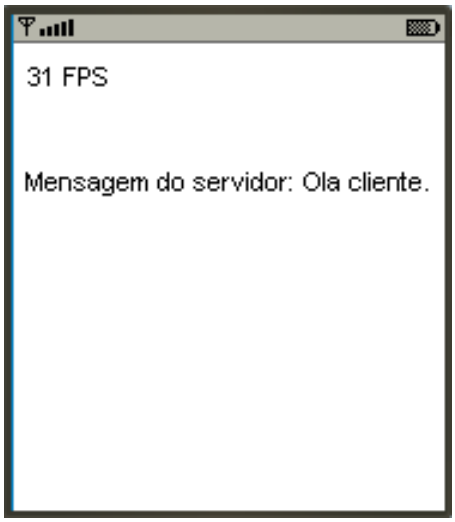


Figura I.14 – Mensagem do servidor, recebida pelo cliente

Anexo II – Artigo

Desenvolvimento de jogos multi-jogadores na plataforma J2ME

Danilo Machado Delponte

Departamente de informática e Estatística
Universidade Federal de Santa Catarina
Florianópolis, Brasil; 88040900
Email: dan@inf.ufsc.br

Resumo

Há alguns anos a indústria de jogos eletrônicos se tornou uma das maiores indústrias de entretenimento do mundo, e a cada ano atinge um número maior de usuários e plataformas. Mas jogos em aparelhos celulares, a não ser pelo iPhone da Apple, nunca chegaram a fazer sucesso. Esse artigo trata de uma abordagem ainda pouco explorada nessa área: jogos casuais multi-jogadores em J2ME através da internet móvel. Mesmo os aparelhos mais simples de hoje possuem recursos suficientes para esse tipo de jogos, são dispositivos populares e de uso diário, o que os torna uma plataforma em potencial. Nesse artigo será apresentada uma análise e revisão do contexto atual de jogos para celulares, uma análise e projeções para a internet móvel no Brasil e um framework para desenvolvimentode jogos multi-jogadores em J2ME.

Palavras-chave: J2ME, jogos eletrônicos, jogos casuais, jogos sociais, jogos multi-jogadores, aparelhos celulares, internet móvel.

Abstract

Some years ago the electronic games industry had become one of the biggest entertainment industries of the world, and each year reachs a higher number of users and platforms. But games on mobile cellphones, except the iPhone by Apple, never had a substantial success.This paper is about an approach yet underexplored on this matter: casual J2ME multiplayer games through mobile internet. Even the low-end mobile phones today have enough resources for this kind of games. They are popular, daily used devices which makes them a potential platform for that approach. This paper will present an analisys and revision of the current scenario of mobile games, analisys and projections for the mobile internet in Brazil and a framework for development of J2ME multi-player games.

Keywords: J2ME, electronic games, casual games, social games, multiplayer games, mobile games, cellphones, mobile internet.

Introdução

Há hoje mais de 4,6 bilhões de usuários de **telefones celulares** e estima-se que esse número passará de 5,9 bilhões em 2013 (Teleco, 2010) (Infonetics Research, 2009). O Brasil está entre os 5 países com maior teledensidade, com 91,87 celulares para cada 100 habitantes, 175 milhões no país (Anatel, 2010).

Esse crescimento leva ao desenvolvimento de novas tecnologias com o objetivo de fomentar e suprir os desejos dos consumidores, fornecendo novidades em aparelhos e serviços, cada vez mais completos e sofisticados.

Paralelamente a isso os **jogos eletrônicos** também ganham mais adeptos a cada geração. Seja no computador, no celular, na internet, entre jovens, adultos, homens ou mulheres. Uma pesquisa recente mostra que nos EUA 54% dos adultos com 18 anos ou mais jogam videogame, e dos 12 aos 17 anos, 97% jogam (Games Industry, 2009).

Entretanto, os jogos em aparelhos celulares só ganharam força recentemente, com a chegada de dispositivos mais sofisticados como o iPhone. A maioria dos aparelhos ainda possui poucos recursos, principalmente no Brasil onde a incorporação de novas tecnologias é lenta e mais cara.

Entretanto não é raro jogos simples fazerem sucesso. Como é o caso de clássicos como *Tetris*, que surgiu como um *minigame*, e o *Snake*, o jogo da "cobrinha" famoso em celulares. Na internet há jogos simples como esses, também chamados de **jogos casuais** ou **sociais**, que são bastante difundidos. Com várias temáticas e um público alvo abrangente, atraem principalmente pelo fácil acesso e jogabilidade simples e rápida, ideal para pessoas com pouco tempo para dedicar a jogos. Assim acabam conquistando tantos usuários quanto grandes jogos. Um bom exemplo é o *Farmville* que conta com cerca de 80 milhões de usuários (Cashmore, 2010).

Dentro desse contexto podemos dizer que estamos presenciando uma nova evolução, a da **internet móvel**. Com a disseminação do serviço e a consequente diminuição dos custos, espera-se um aumento no número de acessos via celular (Cisco, 2009) (Teleco, 2010). O *iPhone* lidera o ranking e isso mostra que com uma gama maior de aplicativos e serviços, provavelmente esse uso se tornará tão comum quanto verificar e-mails no computador.

Aparelhos celulares e jogos

Há uma grande quantidade de aparelhos diferentes no mercado, sendo muitas as plataformas disponíveis para o desenvolvimento de jogos, variando em linguagem de programação, desempenho, disponibilidade de recursos e abrangência no mercado.

A plataforma mais relevante atualmente no mercado de games é o iPhone, por seu desempenho, recursos e modelo de distribuição.

O J2ME perde em desempenho e acesso a recursos nativos, mas possui recursos suficientes para jogos casuais e é encontrado em dois terços dos aparelhos celulares, inclusive em outras plataformas como Symbian, Windows Mobile e Blackberry. Por isso apresenta-se como uma plataforma em potencial para aplicação desse tipo de jogos.

Jogos casuais já são bastante difundidos na internet. Como o próprio nome diz, são feitos para serem jogados casualmente e

não demandam muito tempo de atenção, fornecendo entretenimento quase que instantaneamente. Os jogos são um tipo de jogo casual, mas para vários jogadores que se encontram dentro de uma rede social como o *Orkut* ou o *Facebook*. Além de interagir dentro do jogo, compartilham *status* e conquistas utilizando as ferramentas da própria rede social, o que estimula os jogadores. Jogos casuais e sociais ganharam foco nos últimos anos, devido ao seu crescente sucesso (EDGE, 2008).

Aparelhos celulares são dispositivos de uso diário, porém, para não mais de alguns minutos de uso constante. Um dos problemas dos jogos atuais em J2ME é que frequentemente são versões reduzidas de jogos de outras plataformas. O público alvo desses jogos em sua maioria é composto por jogadores que jogam com frequência e portanto possuem outros aparelhos como um console de mesa ou portátil. Aparelhos celulares não são capazes de fazer frente a essas máquinas e as versões reduzidas de jogos de outras plataformas dificilmente fornecem uma experiência relevante em um aparelho celular.

Outro problema recorrente é a péssima jogabilidade. O teclado dos aparelhos celulares não é ideal para jogos complexos, o que torna alguns deles frustrantes. Isso ocorre principalmente com títulos de ação ou corrida, quando é necessário pressionar vários botões em curto espaço de tempo.

Levando em conta esses aspectos de agilidade e simplicidade, a melhor opção para essa plataforma são os jogos casuais, que têm jogabilidade simples e podem ser jogados em curto espaço de tempo.

Outro problema que afeta o desenvolvimento de jogos em J2ME é que não há um canal simples de distribuição de aplicativos, como a App Store da Apple. Existem portais de operadoras, fabricantes ou terceiros, mas que dificilmente possuem um modelo de negócio atrativo para desenvolvedores.

Jogos multi-jogadores através da internet

A maioria dos jogos eletrônicos é apenas para um jogador, com o objetivo de vencer os desafios programados e inimigos controlados por inteligência artificial. Dependendo do tipo de jogo, competir com outras pessoas é muito mais divertido, além de permitir a interação social entre pessoas geograficamente distantes. Jogos multi-jogadores fornecem uma experiência que não poderia acontecer de outra forma senão em um ambiente virtual. Utilizando aparelhos celulares como plataforma de acesso, estamos levando essa experiência para o bolso de bilhões de pessoas.

Jogos multi-jogadores podem ser implantados em redes locais ou utilizar a internet. Do ponto de vista dos aparelhos celulares, uma rede local pode ser obtida através de tecnologias de transmissão sem fio de curto alcance, como o *bluetooth* ou infra-vermelho. A transmissão de dados é relativamente rápida, mas nem todos os aparelhos disponibilizam esse recurso e o alcance de transmissão é curto, de forma que os dispositivos precisam estar próximos para se comunicar.

Já o acesso à internet em aparelhos celulares pode ser feito de qualquer lugar, desde que dentro da cobertura das operadoras, e está disponível em praticamente todos os aparelhos, com tecnologias como o GPRS. A transferência de dados através da rede móvel pode ser cara e apresentar problemas de instabilidade, contudo, espera-se que os

avanços recentes nessa área contribuam para uma melhora no serviço.

Os jogos multi-jogadores podem variar quanto à forma de interação entre os jogadores, sendo em tempo real, baseada em turnos ou assíncrona (Friedmann, 2009). Jogos em tempo real são aqueles em que qualquer jogador pode atuar a qualquer momento, alterando o estado do jogo continuamente. Destacam-se pelo alto grau de interação, mas requerem um sistema robusto, tolerante a falhas e com resposta rápida, e é altamente dependente da qualidade da conexão dos jogadores não sendo ideal para aplicação em aparelhos celulares (Cecin & Trinta, 2007).

Nos jogos baseados em turnos, o estado do jogo é alterado apenas uma vez a cada jogada, reduzindo muito a quantidade de informações que devem ser processadas. Há ainda a interação de forma indireta ou assíncrona, onde cada jogador possui um ambiente próprio dentro do jogo, no qual pode atuar sem restrições. A interação entre os jogadores é limitada, de forma que não afeta a continuidade do jogo para os demais e o estado do jogo é definido por uma pequena quantidade de informações. Portanto, para aplicação em aparelhos celulares são ideais os jogos baseados em turnos ou de interação assíncrona.

Quanto à arquitetura do sistema distribuído há duas abordagens utilizadas normalmente: sistemas ponto-a-ponto e cliente/servidor.

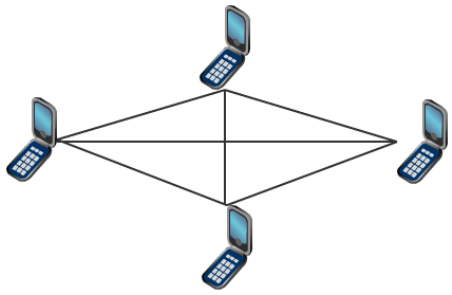


Diagrama 1 – Representação de um sistema ponto-a-ponto

O modelo ponto-a-ponto entre aparelhos celulares não é muito interessante, pois todo o processamento da lógica do jogo assim como o tráfego de dados recaem sobre os aparelhos celulares. A maioria dos aparelhos ainda possui capacidade de processamento e quantidade de memória reduzidos, devendo-se evitar ao máximo a utilização desses recursos bem como o excesso de consumo da bateria.

Além disso, sistemas ponto-a-ponto na rede móvel não existem como na rede fixa, sendo necessário uma configuração específica, pois o endereçamento dos dispositivos ocorre de forma diferenciada devido à constante alteração na localização dos aparelhos (Oliveira, 2005).

No modelo cliente/servidor uma ou mais máquinas, representando o servidor, são dedicadas ao processamento e distribuição dos dados. Em um sistema que envolve aparelhos celulares, podemos ainda utilizar um computador como servidor, dessa forma a complexidade do aplicativo não é limitada pela capacidade dos aparelhos celulares. Essa abordagem também traz a vantagem do controle de acesso ao sistema, já que todos os usuários devem conectar-se ao servidor para ter acesso.

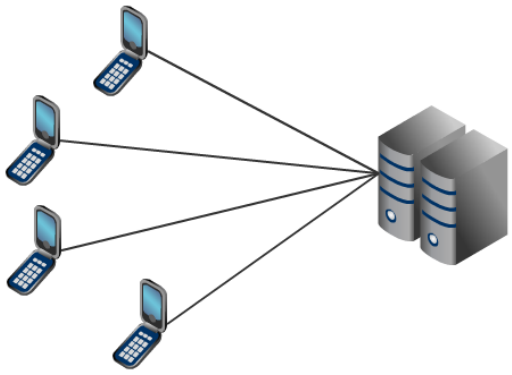


Diagrama 2 – Representação de um sistema cliente/servidor

Internet móvel

A internet móvel evoluiu consideravelmente nos últimos anos, e a expectativa é de que continue evoluindo. Desde as tecnologias GPRS/EDGE o serviço é suficiente para aplicações com baixa demanda, com tráfego de 40kbps a 130kbps. Hoje essas tecnologias são suportadas em praticamente todos os aparelhos GSM, que representam cerca de 90% do total no Brasil (Teleco, 2010).

Com a chegada da tecnologia 3G, a internet móvel entrou na era ”banda larga”, com transferências de 200 a 700kbps, fornecendo uma experiência semelhante à da internet via cabo.

O número de acessos à internet a partir de dispositivos móveis vem crescendo gradativamente e calcula-se que o tráfego de dados deve aumentar em torno de sessenta e seis vezes nos próximos cinco anos (Cisco, 2009), já tendo ultrapassado o número de acessos fixos em vários países. No Brasil, a banda larga móvel crescerá mais de 70% até 2014, ultrapassando o número de acessos fixos em meados de 2011, segundo as projeções da Teleco (Huawei/Teleco, 2009).

Mas apesar das projeções, o avanço da internet móvel vem enfrentando dificuldades, principalmente no Brasil. A disponibilidade é baixa para o segmento pré-pago, que representa mais de 70% dos usuários, e o preço é proibitivo. A maioria das pessoas tem medo de acessar e acabar gastando muito.

Uma pesquisa recente do Centro de Estudos sobre as Tecnologias da Informação e da Comunicação (CETIC, 2009) mostra o lado pessimista das projeções de acesso à internet móvel. Segundo a pesquisa, nos últimos cinco anos o crescimento percentual de usuários que acessam a internet pelo celular foi de 1%, chegando ao total de apenas 6% em 2009 , como mostra o gráfico 4. A pesquisa também atribui esse resultado ao preço praticado pelas operadoras.

Nesse cenário, é ideal que uma aplicação para dispositivos móveis que utilize a internet gere o mínimo de tráfego de dados possível.

Outros problemas relacionados à estrutura da rede móvel também afetam sua popularidade. A queda da conexão é um dos problemas mais frequentes, devido à instabilidade e variação da intensidade do sinal. Uma falha na transmissão pode também fazer com que uma ou mais mensagens não cheguem ao seu destino.

Mas carga insuficiente na bateria do dispositivo e até mesmo falta de créditos do usuário com a operadora também podem causar a desconexão do usuário. Devido a essas situações, a perda de conexão pode ocorrer a qualquer momento durante um jogo, portanto, é indispensável o uso de mecanismos de tolerância a falhas que tratem problemas como perda de pacotes e de conexão. O

sistema deve estar preparado para corrigir essas falhas da forma mais consistente e transparente possível, para que a continuidade do jogo não seja prejudicada e o impacto para os usuários seja mínimo.

Uma forma de contornar esses problemas é estabelecer um tempo de espera para o retorno do jogador e implementar um mecanismo de reentrada no servidor, para que o jogador seja recolocado no estado em que estava antes de perder a conexão, e também incluir no protocolo da aplicação uma confirmação de recebimento de mensagens, para garantir que elas de fato estão chegando ao seu destino.

Desenvolvimento de jogos

O desenvolvimento de jogos não é uma atividade trivial. Possuem lógicas e interfaces complexas, utilizam diversos efeitos visuais e sonoros, gerenciam vários elementos independentes ao mesmo tempo e é fundamental que mantenham o controle de recursos na memória e da velocidade de execução, para uma experiência contínua, rica e consistente. Essas características implicam em métodos e técnicas avançadas para obter o melhor visual e desempenho.

Em se tratando de dispositivos móveis, o desenvolvimento de jogos apresenta ainda outras complicações. Os recursos disponíveis, como tamanho de tela e a quantidade de memória, variam muito devido ao grande número de fabricantes e modelos diferentes. Mesmo aparelhos que na teoria deveriam funcionar da mesma forma, muitas vezes se comportam de modo diferente ao executar um mesmo aplicativo. Essas diferenças trazem a necessidade de adaptações nos aplicativos (*porting*) para cada grupo diferente de aparelhos.

Existem hoje várias formas de contornar esses problemas, desde a própria concepção do jogo até ferramentas de desenvolvimento, que fazem automaticamente uma adaptação do código para vários modelos diferentes de aparelhos.

Jogos em geral apresentam muitas funcionalidades em comum, como renderização e animação, por isso também é comum a utilização de *frameworks* para facilitar e acelerar o desenvolvimento. O termo “motor de jogo” (*game engine*), é mais utilizado para definir um *framework* no âmbito de desenvolvimento de jogos. Não é difícil encontrar jogos similares na visão do jogador e jogabilidade, com poucas diferenças além da arte gráfica. Esses jogos são criados usando um mesmo “motor”.

Um motor de jogos geralmente apresenta suporte para renderização 2D ou 3D, animações, sons, simulação de física, detecção de colisões, suporte para comunicação através de uma rede, entre outras, dependendo do tipo de jogo que se deseja desenvolver.

Embora existam *frameworks* para o desenvolvimento de jogos em muitas plataformas, há relativamente poucos para aparelhos celulares.

Com base nas análises realizadas, foram realizados o projeto e implementação de um *framework* para desenvolvimento de jogos em J2ME e um *framework* para um servidor de jogos multi-jogadores a fim de suportar essa funcionalidade.

Framework para jogos multi-jogadores em J2ME

Jogos possuem uma série de elementos e subsistemas encarregados de executar as diversas funções que os compõem, tais

como: controle do ciclo do jogo, renderização, controle de cenas, input, eventos, IA, etc. O *framework* desenvolvido busca suprir algumas necessidades básicas encontradas em jogos, que serão utilizadas no desenvolvimento dos dois jogos apresentados como caso de uso.

Em se tratando de desenvolvimento para dispositivos móveis, foram levadas em conta as limitações existentes quanto à velocidade de processamento e a quantidade de memória disponível, e também as particularidades resultantes da variedade de modelos existentes no mercado.

A arquitetura do *framework* é composta por módulos, onde cada um implementa uma das funcionalidades incluídas. A seguir serão apresentadas as especificações de cada uma delas.

Ciclo do jogo

Durante o tempo de execução de um jogo há uma sequência de tarefas que devem ser executadas, algumas indefinidamente, como desenhar os objetos na tela, outras a partir da entrada de dados pelo usuário, outras ainda de acordo com o tempo.

Todas essas atividades têm início em um ciclo que consiste em apenas duas operações básicas: renderização (desenhar os objetos na tela) e atualização (atualizar o estado dos objetos).

Sobre essas duas operações são implementadas as funções que resultam na experiência visual e interativa que o jogo proporciona.

Gerenciamento de cenas

A estrutura de um jogo é composta por cenas, cada qual com sua função, elementos, ações e eventos. Como exemplo, podemos citar o menu principal, que aparece na maioria dos jogos, contendo botões com opções para o jogador, imagem de fundo, etc. Assim como cada fase também é uma cena, que possui cenário, personagens, eventos.

Para controlar o que é mostrado na tela e atualizado a cada momento do jogo, o *framework* possui um sistema de controle de cenas, de forma que é possível criar novas cenas, alternar entre elas, adicionar e remover elementos quaisquer.

A partir da implementação dos métodos do ciclo principal, a cena é inserida de modo a fazer uma propagação da renderização e atualização para todos os elementos nela contidos.

Componentes de interface

Para realizar a interação com os usuários, vários objetos de interface são comumente utilizados, como botões, caixas de texto, listas, etc.

A biblioteca de Java para dispositivos móveis disponibiliza vários desses componentes de interface, porém, de forma não customizável, que adquirem automaticamente a aparência do sistema do aparelho.

Jogos geralmente possuem interface personalizada, com menus internos e opções relacionadas ao jogo. Por isso normalmente as desenvolvedoras de jogos acabam por construir também seu próprio *framework*.

Uma arquitetura conhecida para desenvolvimento de interfaces é a que utiliza componentes e contêineres. Um componente é uma abstração de todos os elementos da interface, com atributos e métodos genéricos comuns. Contêineres são regiões delimitadas que podem conter componentes, como uma barra de menu

com vários botões, ou um painel com informações e imagens descritivas.

As funcionalidades básicas implementadas para componentes no *framework* foram: alinhamento, controle de foco, navegação e seleção. A função de alinhamento tem como objetivo posicionar os componentes de forma organizada na tela, para apresentar uma interface coerente e intuitiva para o usuário. A funcionalidade de controle de foco tem como objetivo mostrar ao usuário com qual componente está interagindo no momento e permitir a navegação entre os componentes que possam obter o foco. As funções de navegação e seleção são as que permitem ao usuário interagir com o sistema, navegando através dos componentes da interface e selecionando-os.

O *framework* então dispõe de um pacote de componentes gráficos básicos, como botões, imagens, *labels*, caixas de entrada de texto e contêineres, que implementam essas funcionalidades e podem ser personalizados de acordo com o jogo.

Gerenciamento de eventos

Durante um jogo vários elementos atuam ao mesmo tempo, alterando seus estados de acordo com entradas do usuário ou pela própria dinâmica do jogo. O gerenciamento de eventos é uma funcionalidade que visa facilitar o controle sobre as ações que devem definir o fluxo da interação e continuidade do jogo, sem a necessidade de uma relação direta entre o objeto que disparou o evento e o objeto que deverá tratá-lo.

O *framework* implementa uma proposta de arquitetura simplificada para o gerenciamento de eventos, onde cada evento possui apenas um atributo que define o tipo do evento. Qualquer elemento do jogo pode tornar-se ouvinte de um determinado tipo de evento, implementando a interface de ouvinte e registrando-se com o gerenciador de eventos. E qualquer elemento também pode disparar um evento através do gerenciador, e todos os ouvintes daquele tipo de eventos serão notificados.

Suporte à comunicação

Para realizar a comunicação com o servidor o jogo deve ser capaz de estabelecer uma conexão e, de acordo com um protocolo definido, montar, ler, enviar e receber mensagens.

A biblioteca J2ME fornece suporte à comunicação utilizando os protocolos TCP, HTTP e UDP, através do pacote *javax.microedition.io*. O protocolo utilizado foi o TCP, devido à sua confiabilidade e tamanho, pois apenas pequenas sequências de *bytes* são trafegadas, a fim de reduzir o custo da transmissão. Porém, essa decisão não impede que o protocolo HTTP seja utilizado para outras funções, como integração com redes sociais e envio de informações, tais como pontuação do jogador, comentários, etc.

O *framework* implementado recebe as mensagens da rede através do gerenciador de eventos, de forma que os elementos do jogo podem tratá-las como eventos ao invés de pacotes de *bytes*, facilitando o tratamento das mensagens. Foi criada também uma classe responsável por esse protocolo, criando, enviando e recepcionando os pacotes de *bytes* e transformando-os em eventos.

Framework para servidores multi-jogadores

Completando o sistema proposto, foi desenvolvido também um *framework* para

implementação de servidores para jogos simples multi-jogadores.

A linguagem utilizada para o desenvolvimento do servidor foi o C#. A escolha da linguagem foi baseada nas funcionalidades oferecidas e nas necessidades de alto desempenho e robustez em que implicam o gerenciamento e o processamento de jogos multi-jogadores.

O servidor dispõe de classes responsáveis por criar e gerenciar conexões, trocar mensagens com os clientes, criar ambientes virtuais para interação entre os jogadores, como salas de encontro e sessões de jogo, e também classes de apoio à persistência de dados. A partir dessas classes pode-se então montar a estrutura do jogo, tratar a interação entre os jogadores e processar a lógica do jogo.

O *framework* também implementa o mesmo protocolo de aplicação definido para o cliente, que corresponde ao formato das mensagens trocadas através da rede. Essas mensagens são utilizadas para transmitir as ações do jogo, tais como: pedido de início de jogo, envio de jogada, envio de mensagens de texto, desconexão, entre outros.

Casos de uso

Como casos de uso da utilização dos *frameworks* implementados foram desenvolvidos dois jogos de exemplo, um baseado em turnos e outro com interação do tipo assíncrona.

Como caso de uso de jogo multi-jogadores com interação baseada em turnos, o jogo escolhido foi o “Jan Ken Po”, ou como é chamado no Brasil: “papel, tesoura e pedra”. A escolha desse jogo baseia-se no fato de que é um jogo simples, porém divertido, com um nível baixo de complexidade para um jogo multi-jogadores, pois cada jogada consiste em apenas uma variável para cada jogador.

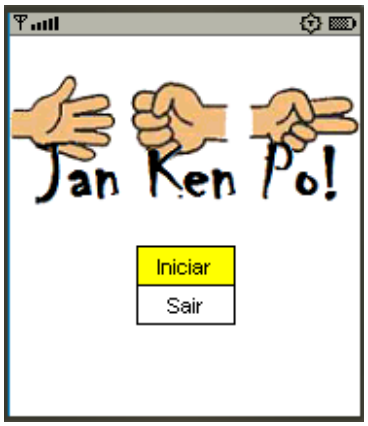


Figura 1 – Menu principal do *Jan Ken Po*

A versão *online* para celulares do jogo *Jan Ken Po*, consiste em uma sala de jogadores, implementada no servidor, onde podem desafiar uns aos outros para uma partida. Logo no início do jogo o jogador deve informar um apelido, que será utilizado para sua identificação no servidor e incluído na lista da sala de jogadores. A partir dessa lista os jogadores devem convidar um dos outros jogadores disponíveis para uma partida. Quando um desafio é aceito, o servidor notifica os dois jogadores de que será iniciada uma sessão de jogo. Os jogadores então são transferidos no servidor, para o ambiente responsável pela dinâmica do jogo, onde cada um deve efetuar uma jogada. Após o envio das jogadas, o servidor calcula o resultado e comunica os jogadores, que podem optar por continuar o jogo ou voltar à sala de jogadores.

Para o desenvolvimento desse jogo foram utilizados todos os módulos implementados no *framework* em J2ME, para o gerenciamento das cenas, criação da interface baseada nos componentes e troca de mensagens através da rede. O *framework* para implementação do servidor foi utilizado para a criação de uma

sala de jogadores e gerenciamento das sessões de jogo.

Figura 2 – Resultado do *Jan Ken Po*

Como caso de uso de jogo com interação assíncrona foi implementado o *JobFun*, um jogo simples, do tipo social, em que os jogadores podem escolher uma profissão e produzir objetos a partir de *minigames*.

A interação entre os jogadores se dá na possibilidade de venda e compra de objetos entre eles, uma vez que cada jogador pode produzir apenas objetos relativos à profissão que escolheu. Cada jogador possui também um "quarto virtual", no qual pode dispor seus objetos.

Figura 4 – Quarto virtual no *JobFun*

Há a necessidade da persistência dos dados dos usuários, como os itens disponíveis para venda, itens vendidos, créditos para compra de objetos, etc. Para isso foi utilizado um banco de dados, cujas transações serão controladas a partir do servidor.

As profissões disponíveis para o jogo desenvolvido, a título de demonstração, foram: marceneiro e florista. Escolhendo a profissão de marceneiro é possível criar móveis de madeira como bancos e mesas. A profissão de florista traz a possibilidade da criação de vasos e arranjos com flores. Cada profissão possui dois minigames que serão apresentados na construção dos objetos, utilizando tipos de jogabilidade diferentes, como sequência de teclas e *timing*.

Figura 5 – *Minigame* para o marceneiro

Na venda dos objetos o jogador pode estipular um preço à sua escolha, sendo que esse valor é revertido em créditos para que o jogador possa comprar objetos de outros jogadores. Ou seja, deve produzir e vender objetos para poder comprar itens de outros jogadores.

No desenvolvimento desse jogo foram utilizados os mesmos módulos do *framework*, para o gerenciamento das cenas, criação da interface baseada nos componentes e troca de mensagens através da rede. O *framework* para o servidor foi utilizado no tratamento das mensagens disparadas por eventos dos usuários e na integração com o banco de

dados no qual os dados dos jogadores foram persistidos.

Conclusão

Como resultado desse trabalho, foi apresentado o projeto e implementação de um *framework* para desenvolvimento de jogos com múltiplos jogadores para celular através da internet, um *framework* para servidores que possam suportar esses jogos, com as funções de comunicação e persistência utilizadas e, por fim, o desenvolvimento de dois jogos desse tipo como casos de uso.

Os objetivos do trabalho foram alcançados, abordando o contexto atual do desenvolvimento de jogos para celular e alguns dos problemas e dificuldades encontradas nessa área. Foram apresentados alguns fatos que dificultam a aceitação desses aparelhos como plataformas para jogos, principalmente problemas de usabilidade, complexidade e distribuição e foi feita uma análise das projeções acerca da internet móvel, visando sua utilização como meio para o desenvolvimente de jogos multi-jogadores.

Assim, pudemos observar a possibilidade de levar aos aparelhos celulares que disponibilizam a plataforma J2ME, um modelo de jogos bastante difundido na internet. Devido à simplicidade e abrangência desse tipo de jogos acredita-se que poderiam facilmente ser adaptados para esses dispositivos e atingir um grande público.

Ao final, nem todos os aspectos apresentados pela análise foram abordados de forma prática, tais como tolerância a falhas na rede móvel e distribuição comercial de jogos para celular. Esses aspectos não devem ser ignorados, pelo contrário, definem de forma decisiva a viabilidade comercial de um projeto como esse, porém, para os fins acadêmicos a que se destina esse trabalho, o resultado foi considerado satisfatório.

Trabalhos futuros

A partir da arquitetura projetada para o *framework*, é possível estendê-lo para atender aos requisitos de vários tipos de jogos. Pode-se também adicionar funcionalidades que não foram implementadas como tolerância a falhas, comunicação via HTTP e integração com redes sociais.

Futuramente é possível que a internet móvel e os aparelhos celulares evoluam a ponto de possibilitar o desenvolvimento de jogos multi-jogadores em tempo real. O *framework* apresentado também pode ser utilizado para criar jogos desse tipo. Ou ainda, pode-se adicionar suporte a conexões *bluetooth*, para jogos em que seja interessante a proximidade entre os jogadores.

Outro tema interessante é a implementação de jogos *cross-platform*, que significa jogadores participando de um mesmo jogo a partir de plataformas diferentes. Como por exemplo, um jogo que apresenta uma versão *web*, acessada através de um navegador de internet, e também uma versão para celulares, integradas para que os jogadores possam interagir dentro de um mesmo jogo indepentemente da plataforma.

Referências bibliográficas

Anatel. (janeiro de 2010). *Quantidade de Acessos/Plano de Serviço/Unidade da Federação - Janeiro/2010*. Acesso em março de 2010, disponível em http://sistemas.anatel.gov.br/SMP/Administracao/Consulta/AcessosPrePosUF/tela.asp

Barth, P. (2009, outubro 30). Colóquios Cyclops. *Mobile Applications are (yet again?) taking off* . UFSC; Florianópolis.

Bombonatti, P. (18 de agosto de 2009). *Mercado de mobile games terá crescimento significante*. Acesso em abril de 2010, disponível em MobilePedia: http://www.mobilepedia.com.br/noticias/mercado-de-mobile-games-tera-crescimento-significante

Cecin, F., & Trinta, F. (9 de novembro de 2007). *Jogos Multiusuário Distribuídos.* Acesso em novembro de 2009, disponível em VI Brasilian Synposium on Computer Games and Digital Entertainment: http://www.inf.unisinos.br/~sbgames/anais/tutorials/Tutorial1.pdf

CETIC. (2009). *TIC DOMICÍLIOS e USUÁRIOS - Pesquisa sobre o Uso das Tecnologias da Informação e da Comunicação no Brasil.* Acesso em abril de 2010, disponível em Cetic.br: http://www.cetic.br/usuarios/tic/2009/analise-tic-domicilios2009.pdf

Cisco. (10 de fevereiro de 2009). *Newest Cisco Visual Networking Index Forecast Highlights Growth in Mobile Traffic.* Acesso em 2009 de novembro, disponível em Cisco: http://newsroom.cisco.com/dlls/2009/prod_021009d.html

EA Mobile. (s.d.). *Mobile Games, iPhone Games, Cell Phone Games*. Acesso em novembro de 2009, disponível em EA Mobile: http://www.eamobile.com/home/

EDGE. (5 de março de 2008). *New Stats Show Casual Explosion*. Acesso em novembro de 2009, disponível em EDGE: http://www.edge-online.com/features/new-stats-show-casual-explosion

Friedmann, D. (8 de novembro de 2009). *Multiplayer Game Models*. Acesso em dezembro de 2009, disponível em http://proj-mgt.com/wordpress/

Games Industry. (21 de maio de 2009). *NPD: More Americans play games than go to movies*. (D. Jenkis, Editor) Acesso em novembro de 2009, disponível em Games Industry: http://www.gamesindustry.biz/articles/npd-more-americans-play-games-than-go-to-movies

Gandra, A. (3 de março de 2009). *Mercado de jogos eletrônicos cresce no Brasil apesar da crise*. Acesso em novembro de 2009, disponível em Inovação tecnológica: http://www.inovacaotecnologica.com.br/noticias/noticia.php?artigo=mercado-de-jogos-eletronicos-cresce-no-brasil-apesar-da-crise&id=

Huawei/Teleco. (dezembro de 2009). *Balanço Huawei da Banda Larga Móvel.* Acesso em março de 2010, disponível em Huawei: http://www.huawei.com/pt/catalog.do?id=1779

Infonetics Research. (3 de novembro de 2009). *Mobile subscribers to hit 5.9 billion in 2013, driven by China, India, Africa*. Acesso em novembro de 2009, disponível em Infonetics: http://www.infonetics.com/pr/2009/Fixed-and-Mobile-Subscribers-Market-Highlights.asp

Knowles, J. (12 de setembro de 2007). *Thought on Mobile Social Gaming*. Acesso em março de 2010, disponível em Auscillate: http://www.auscillate.com/post/118

Long, K. (6 de março de 2007). *GDC Mobile - The Future of Mobile Games Will be*

Social. Acesso em abril de 2010, disponível em Future Lab: http://www.futurelab.net/blogs/marketing-strategy-innovation/2007/03/gdc_mobile_the_future_of_mobil.html

O Estado de S. Paulo. (8 de junho de 2009). *Todo mundo joga videogame*. (J. Auricchio, R. Martins, & H. Lupin, Editores) Acesso em novembro de 2009, disponível em Estadão: http://www.estadao.com.br/noticias/tecnologia+link,todo-mundo-joga-videogame,2597,0.shtm

Oliveira, L. P. (agosto de 2005). *Os dispositivos móveis e as redes peer-to-peer (P2P).* Acesso em maio de 2010, disponível em Universidade Federal de Pernambuco: http://www.cin.ufpe.br/~tg/2005-1/lpo.pdf

Oracle. (2010). *Java ME: the Most Ubiquitous Application Platform for Mobile Devices*. Acesso em maio de 2010, disponível em Sun Developer Network (SDN): http://java.sun.com/javame/index.jsp

Powers, M. (novembro de 2006). *Mobile Multiplayer Gaming, Part 1: Real-Time Constraints*. Acesso em outubro de 2009, disponível em Sun Developer Network (SDN): http://developers.sun.com/mobility/midp/articles/gamepart1/

Teleco. (15 de março de 2010). *Estatísticas de celular no mundo*. Acesso em novembro de 2009, disponível em Teleco: http://www.teleco.com.br/pais/celular.asp

Teleco. (25 de abril de 2010). *Estatísticas de celulares e redes 3G no mundo (WCDMA/HSDPA e EVDO)*. Acesso em abril de 2010, disponível em Teleco: http://www.teleco.com.br/pais/3g.asp

Teleco. (15 de março de 2010). *Estatísticas de celulares no mundo, ranking de países, operadores e tecnologias*. Acesso em março de 2010, disponível em Teleco: http://www.teleco.com.br/pais/celular.asp

Teleco. (27 de janeiro de 2008). *Seção: Telefonia Celular*. Acesso em novembro de 2009, disponível em Teleco: http://www.teleco.com.br/tecnocel.asp

Todo mundo quer ter sua própria App Store. (9 de março de 2010). Acesso em maio de 2010, disponível em ArchTec Blog: http://archtec.wordpress.com/2010/03/09/todo-mundo-quer-ter-sua-propria-app-store/

Tsirulnik, G. (19 de abril de 2010). *Smartphones shift US mobile gaming market landscape*. Acesso em maio de 2010, disponível em Mobile Marketer: http://www.mobilemarketer.com/cms/news/research/6006.html

Anexo III – Código fonte

III.1 Framework para desenvolvimento de jogos em J2ME

III.1.1 *game.core.GameCore*

```
package game.core;

import javax.microedition.lcdui.Display;
import javax.microedition.lcdui.Graphics;
import javax.microedition.midlet.MIDlet;
import javax.microedition.midlet.MIDletStateChangeException;

public abstract class GameCore extends MIDlet implements Runnable {

	private XCanvas xcanvas;
	private boolean paused;
	private long targetFPS = 30;
	private long period = 1000 / targetFPS;
	private long startTime = 0, excess = 0, endTime = 0, overSleep =
0,
			diffTime = 0, sleepTime = 0;

	private static GameCore instance;

	protected void destroyApp(boolean arg0) throws
MIDletStateChangeException {
		System.out.println("destroyApp()");
		notifyDestroyed();
		// TODO handle
	}

	protected void pauseApp() {
		System.out.println("pauseApp()");
		paused = true;
	}

	protected void exit() {
		System.out.println("exit()");
		try {
			destroyApp(true);
		} catch (MIDletStateChangeException e) {
			e.printStackTrace();
		}
	}

	protected void startApp() throws MIDletStateChangeException {
		System.out.println("startApp()");
		instance = this;
		if (!paused) {
			xcanvas = new XCanvas();
			Display.getDisplay(this).setCurrent(xcanvas);
		}
		loadGame();
		start();
	}
```

```
protected void start() {
	System.out.println("start()");
	paused = false;
	new Thread(this, "gameloop").start();
}

public void run() {
	System.out.println("starting game loop..");
	endTime = System.currentTimeMillis();
	while (!paused) {

		if(!xcanvas.isShown())
		{
			try {
				Thread.sleep(50);
				continue;
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}


		startTime = System.currentTimeMillis();
		overSleep = (startTime - endTime) - sleepTime;

		// updates
		update();

		// paints
		Graphics g = xcanvas.getGraphics();
		xcanvas.clearBackground();
		paint(g);
		xcanvas.flushGraphics();

		endTime = System.currentTimeMillis();
		diffTime = endTime - startTime;

		sleepTime = (period - diffTime) - overSleep;

		if (sleepTime <= 0) {
			excess -= sleepTime;
			sleepTime = 2;
		}

		while (excess >= period) {
			update();
			excess -= period;
		}

		try {
			Thread.sleep(sleepTime);
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
	}
	System.out.println("loop thread ended..");
}

/**
 * Load application before the game loop thread starts.
 */
```

```
	protected abstract void loadGame();

	/**
	 * Paint method called on game loop thread.
	 *
	 * @param g
	 *            - the XCanvas graphics object.
	 */
	protected abstract void paint(Graphics g);

	/**
	 * Update method called on game loop thread.
	 */
	protected abstract void update();

	/**
	 * Returns the current GameCore instance.
	 *
	 * @return the current GameCore instance.
	 */
	public static GameCore getGameCoreInstance() {
		return instance;
	}

	/**
	 * The display associated to the current GameCore instance.
	 *
	 * @return the display associated to the current GameCore
instance.
	 */
	public static Display getDisplay() {
		return Display.getDisplay(instance);
	}

	/**
	 * The {@link XCanvas} from the current GameCore instance.
	 *
	 * @return the {@link XCanvas} from the current GameCore
instance.
	 */
	public static XCanvas getCanvas() {
		return instance.xcanvas;
	}

	/**
	 * The width from the {@link XCanvas}.
	 *
	 * @return the width from the {@link XCanvas}.
	 */
	public static int getCanvasWidth() {
		return instance.xcanvas.getWidth();
	}

	/**
	 * The height from the {@link XCanvas}.
	 *
	 * @return the height from the {@link XCanvas}.
	 */
	public static int getCanvasHeight() {
		return instance.xcanvas.getHeight();
	}
```

```
}
```

III.1.2 *game.core.XCanvas*

```
package game.core;

import javax.microedition.lcdui.Font;
import javax.microedition.lcdui.Graphics;
import javax.microedition.lcdui.Image;
import javax.microedition.lcdui.game.GameCanvas;

public class XCanvas extends GameCanvas {

	private boolean nativeDoubleBuffer = false;
	private Graphics bufferedGraphics, graphics;
	private Image buffer;
	private Key key = new Key();

	// FPS stuff
	private long FPSTimer, currentTime;
	private int FPSCounter, lastCount;
	private String FPSLabel;
	private Font FPSfont = Font.getFont(Font.FACE_MONOSPACE,
Font.STYLE_PLAIN,
			Font.SIZE_SMALL);
	private Image FPSImage = Image.createImage(FPSfont
			.stringWidth("XXXXXXXXXXXXXXXX"),
FPSfont.getHeight() + 2);
	private Graphics FPSGraphics = FPSImage.getGraphics();

	public XCanvas() {
		super(false);
		System.out.println("XCanvas.XCanvas()");

		setFullScreenMode(true);

		nativeDoubleBuffer = isDoubleBuffered();
		if (nativeDoubleBuffer) {
			bufferedGraphics = super.getGraphics();
		} else {
			buffer = Image.createImage(getWidth(), getHeight());
			bufferedGraphics = buffer.getGraphics();
			graphics = super.getGraphics();
		}
		System.out.println("XCanvas initiated." + "\n\t Size: " +
getWidth()
				+ "x" + getHeight() + "." + "\n\t
DoubleBuffered: "
				+ isDoubleBuffered());
	}

	protected void sizeChanged(int w, int h) {
		super.sizeChanged(w, h);
		System.out.println("CanvasX.sizeChanged(): " + w + ", " +
h);
	}

	protected void keyPressed(int keyCode) {
		key.keyPressed(keyCode);
	}
```

```
	protected void keyRepeated(int keyCode) {
	}

	protected void keyReleased(int keyCode) {
		key.keyReleased(keyCode);
	}

	public void flushGraphics() {
		FPSCounter++;
		if (!nativeDoubleBuffer)
			graphics.drawImage(buffer, 0, 0, 0);
		super.flushGraphics();
	}

	private long free = 0, total = 0;

	public void drawMemory() {
		free = Runtime.getRuntime().freeMemory();
		total = Runtime.getRuntime().totalMemory();
		currentTime = System.currentTimeMillis();
		if (currentTime - FPSTimer > 1000) {
			FPSTimer = System.currentTimeMillis();

			if (FPSCounter != lastCount) {
				FPSLabel = free + "-" + total;
				FPSGraphics.setFont(FPSfont);
				FPSGraphics.setColor(0xFFFFFF);
				FPSGraphics.fillRect(0, 0, FPSImage.getWidth(),
FPSImage
							.getHeight());
				FPSGraphics.setColor(0);
				FPSGraphics.drawString(FPSLabel, 1, 1, 0);
				lastCount = FPSCounter;
			}
			FPSCounter = 0;
		}
		bufferedGraphics.drawImage(FPSImage, 0, 0, 0);

	}

	public void drawFPS() {
		currentTime = System.currentTimeMillis();
		if (currentTime - FPSTimer > 1000) {
			FPSTimer = System.currentTimeMillis();
			if (FPSCounter != lastCount) {
				FPSLabel = FPSCounter + " FPS";
				FPSGraphics.setColor(0xFFFFFFFF);
				FPSGraphics.fillRect(0, 0, FPSImage.getWidth(),
FPSImage
							.getHeight());
				FPSGraphics.setColor(0);
				FPSGraphics.drawString(FPSLabel, 1, 1, 0);
				lastCount = FPSCounter;
			}
			FPSCounter = 0;
		}
		bufferedGraphics.drawImage(FPSImage, 5, 5, 0);
	}

	// private int colorTemp;
```

```
	public void clearBackground() {
		// colorTemp = bufferedGraphics.getColor();
		bufferedGraphics.setColor(0xFFFFFF);
		bufferedGraphics.fillRect(0, 0, getWidth(), getHeight());
		// bufferedGraphics.setColor(colorTemp);
	}

	public Graphics getGraphics() {
		return bufferedGraphics;
	}

}
```

III.1.3 *game.core.Key*

```
package game.core;

import java.util.Enumeration;
import java.util.Vector;

public class Key {

	public static final short PRESSED = 0, RELEASED = 1, CLICKED =
2,
			IDLE = -1;

	private static int[] keymap = new int[256];
	private static int pos = 0, index = 0;
	private static Vector listeners = new Vector();
	private static Enumeration enn;

	public static int K_UP = -1, K_DOWN = -2, K_LEFT = -3, K_RIGHT =
-4,
			K_SELECT = -5, K_SOFT1 = -6, K_SOFT2 = -7;

	private static long lastClick, lastPress, lastRelease,
pressControl,
			releaseControl;
	private static int lastKeyClicked, lastKeyPressed,
lastKeyReleased;

	Key() {
	}

	public static void clearState() {
		keymap = new int[256];
	}

	protected void keyPressed(int keyCode) {
		put(keyCode, PRESSED);

		pressControl = System.currentTimeMillis();

		enn = listeners.elements();
		while (enn.hasMoreElements())
			((KeyListener)
enn.nextElement()).keyPressed(keyCode);
	}

	protected void keyReleased(int keyCode) {
```

```
            put(keyCode, RELEASED);

            releaseControl = System.currentTimeMillis();

            enn = listeners.elements();
            while (enn.hasMoreElements())
                  ((KeyListener)
enn.nextElement()).keyReleased(keyCode);
      }

      public static final void addKeyListener(KeyListener l) {
            listeners.addElement(l);
      }

      public static final void removeKeyListener(KeyListener l) {
            listeners.removeElement(l);
      }

      public static final void removeAllListeners() {
            listeners.removeAllElements();
      }

      public static final boolean pressed(int key) {
            return get(key) == PRESSED;
      }

      public static final boolean pressed(int key, long delay) {
            if (lastKeyPressed != key
                        || System.currentTimeMillis() - lastPress >
delay) {
                  if (pressed(key)) {
                        lastPress = System.currentTimeMillis();
                        lastKeyPressed = key;
                        return true;
                  }
            }
            return false;
      }

      public static final boolean clicked(int key) {
            return get(key) == CLICKED;
      }

      public static final boolean clicked(int key, long delay) {
            if (lastKeyClicked != key
                        || System.currentTimeMillis() - lastClick >
delay) {
                  if (clicked(key)) {
                        lastClick = System.currentTimeMillis();
                        lastKeyClicked = key;
                        return true;
                  }
            }
            return false;
      }

      public static final boolean released(int key) {
            return get(key) == RELEASED;
      }

      public static final boolean released(int key, long delay) {
```

```
            if (lastKeyReleased != key
                        || System.currentTimeMillis() - lastRelease >
delay) {
                  if (released(key)) {
                        lastRelease = System.currentTimeMillis();
                        lastKeyReleased = key;
                        return true;
                  }
            }
            return false;
      }

      private static final void put(int key, int pressed) {
            index = contains(key);
            if (index > -1) {
                  index++;
                  keymap[index] = pressed;
            } else {
                  keymap[pos] = key;
                  pos++;
                  keymap[pos] = pressed;
                  pos++;
            }
      }

      static final int contains(int key) {
            index = 0;
            while (index < pos) {
                  if (keymap[index] == key)
                        return index;
                  index += 2;
            }
            return -1;
      }

      static final int get(int key) {
            index = contains(key);
            if (index > -1) {
                  index++;
                  updateKeyState(index);
                  return keymap[index];
            } else
                  return IDLE;
      }

      static final void updateKeyState(int keyStateIndex) {
            if (keymap[index] == IDLE)
                  return;

            long releaseDelay = System.currentTimeMillis() -
releaseControl;

            if (keymap[index] == CLICKED) {
                  if (releaseDelay > 200)
                        keymap[index] = IDLE;

            } else if (keymap[index] == RELEASED) {
                  long pressDelay = releaseControl - pressControl;

                  if (releaseDelay > 100)
                        keymap[index] = IDLE;
```

```
                else if (releaseDelay > 50 && pressDelay < 100)
                        keymap[index] = CLICKED;
            }
        }

}
```

III.1.4 *game.core.KeyListener*

```
package game.core;

public interface KeyListener {
        public void keyPressed(int keyCode);

        public void keyReleased(int keyCode);
}
```

III.1.5 *game.core.Color*

```
package game.core;

public interface Color {

        public static final int WHITE = 0xFFFFFFFF;
        public static final int BLACK = 0xFF000000;

        public static final int RED = 0xFFFF0000;
        public static final int GREEN = 0xFF00C000;
        public static final int BLUE = 0xFF0000FF;
        public static final int YELLOW = 0xFFFFFF00;
        public static final int GREY = 0xFFB0B0B0;

        public static final int LIGHT_RED = 0xFFFFAAAA;
        public static final int LIGHT_GREEN = 0xFFAAFFAA;
        public static final int LIGHT_BLUE = 0xFFAAAAFF;
        public static final int LIGHT_YELLOW = 0xFFFFFFAA;
}
```

III.1.6 *game.scene.GameObject*

```
package game.scene;

import javax.microedition.lcdui.Graphics;

/**
 * Base object on the game environment. It updates and paints itself.
 *
 */
public abstract class GameObject {
        /**
         * Handles all logic and behavior of this object on it`s
context.
         */
        public abstract void update();

        /**
         * Paints this game object content.
         *
         * @param g
         *             - the {@link Graphics} object.
```

```
         * @see Game#getCanvas()
         */
        public abstract void paint(Graphics g);
}
```

III.1.7 *game.scene.Scene*

```
package game.scene;

import java.util.Vector;

import javax.microedition.lcdui.Graphics;

public abstract class Scene {

        // Lista de objetos da cena
        private Vector objects = new Vector();

        // Método para o carregamento da cena
        public abstract void load();

        // Método para o descarregamento da cena
        public abstract void unload();

        // Adiciona um GameObject à cena
        public void add(GameObject object) {
                objects.addElement(object);
        }

        // Remove um GameObject da cena
        public void remove(GameObject object) {
                objects.removeElement(object);
        }

        // Pinta todos os objetos que essa cena contém
        public void paint(Graphics g) {
                int index = 0;
                while (index < objects.size()) {
                        ((GameObject) objects.elementAt(index)).paint(g);
                        index++;
                }
        }

        // Atualiza todos os objetos que essa cena contém
        public void update() {
                int index = 0;
                while (index < objects.size()) {
                        ((GameObject) objects.elementAt(index)).update();
                        index++;
                }
        }

        public int getWidth()
        {
                return Game.getCanvasWidth();
        }

        public int getHeight()
        {
                return Game.getCanvasHeight();
        }
```

```
	public void removeAll()
	{
		objects = new Vector();
	}

	public void pushSelf()
	{
		Game.getGameInstance().pushScene(this);
	}

	public void popSelf()
	{
		Game.getGameInstance().popScene(this);
	}
}
```

III.1.8 *game.scene.Game*

```
package game.scene;

import game.core.GameCore;
import game.util.Stack;

import javax.microedition.lcdui.Graphics;

public abstract class Game extends GameCore {

	private Stack sceneStack = new Stack();
	private Scene current = new Scene() {
		public void paint(Graphics g) {

		}

		public void update() {

		}

		public void load() {

		}

		public void unload() {

		}
	};

	/**
	 * From the game loop thread, gets the lock over the current
scene and
	 * paints it
	 */
	protected void paint(Graphics g) {
		// from the game loop thread, gets the lock over the
current scene and
		// paints it
		synchronized (current) {
			current.paint(g);
		}
	}
```

```
	/**
	 * From the game loop thread, gets the lock over the current
scene and
	 * updates it.
	 */
	protected void update() {
		// from the game loop thread, gets the lock over the
current scene and
		// updates it
		synchronized (current) {
			current.update();
		}
	}

	/**
	 * Replaces the current scene.
	 *
	 * @param scene
	 *            - the new scene
	 */
	public void setScene(Scene scene) {
		System.out.println("Game.setScene()");

		if (scene == null)
			throw new IllegalArgumentException(
					"Game.setScene(): scene can`t be null");

		// build the scene
		scene.load();

		// set it as the active
		synchronized (current) {
			current = scene;

			// destroy the last scene and removes it from the
stack
			if (!sceneStack.isEmpty()) {
				Scene s = ((Scene) sceneStack.pop());
				s.unload();
				System.gc();
			}
		}
		// pushes the current to the stack
		sceneStack.push(scene);
	}

	/**
	 * Pushes a new scene to the stack.
	 *
	 * @param scene
	 *            - the new scene
	 */
	public void pushScene(Scene scene) {
		System.out.println("Game.pushScene()");

		if (scene == null)
			throw new IllegalArgumentException(
					"Game.setScene(): scene can`t be null");

		// build the scene
		scene.load();
```

```
			// set it as the active
			synchronized (current) {
				current = scene;

				// destroy the last scene
				if (!sceneStack.isEmpty()) {
					((Scene) sceneStack.lastElement()).unload();
				}
			}
			// push the current to the stack
			sceneStack.push(scene);
		}

		/**
		 * Pops the scene from the stack.
		 * <p>
		 * If that scene is the last one on the stack, the program
terminates.
		 * </p>
		 *
		 * @param scene
		 *            - the scene to be popped
		 */
		public void popScene(Scene scene) {
			System.out.println("Game.popScene()");

			if (scene == null)
				throw new IllegalArgumentException(
						"Game.setScene(): scene can`t be null");

			// if it is the current scene
			if (scene == current) {
				popScene();
			}
			// otherwise just remove from the stack
			else
				sceneStack.removeElement(scene);

		}

		/**
		 * Pops the current scene from the stack.
		 * <p>
		 * If the current scene was the last one on the stack, the
program
		 * terminates.
		 * </p>
		 *
		 * @return the popped scene
		 */
		public Scene popScene() {
			System.out.println("Game.popScene()");

			// remove the current scene from the stack
			Scene poped = (Scene) sceneStack.pop();
			synchronized (current) {
				// if there is another scene behind the current,
activate it
				if (!sceneStack.isEmpty()) {
					Scene s = ((Scene) sceneStack.lastElement());
```

```
                        s.load();
                        current = s;
                    }
                    // otherwise exit application
                    else {
                        notifyDestroyed();
                        return null;
                    }
                    // then destroy the old one
                    poped.unload();
                }
                // and returns it
                return poped;
            }

            /**
             * Returns the current Game instance
             *
             * @return the current Game instance
             */
            public static Game getGameInstance() {
                return (Game) getGameCoreInstance();
            }

}
```

III.1.9 *game.gui.Component*

```
package game.gui;

import game.scene.Game;
import game.scene.GameObject;

public abstract class Component extends GameObject {

    public int x = 0, y = 0;
    public boolean focused = false;
    public boolean isFocusable = true;
    public Container parent;
    public static final String TOP = "T", BOTTOM = "B", LEFT = "L",
            RIGHT = "R", VCENTER = "VC", HCENTER = "HC";

    public abstract int getWidth();

    public abstract int getHeight();

    public void alignToDisplay(String alignment) {
        if (alignment.indexOf(BOTTOM) > -1)
            y = Game.getCanvasHeight() - getHeight();
        else if (alignment.indexOf(TOP) > -1)
            y = 0;
        else if (alignment.indexOf(VCENTER) > -1)
            y = Game.getCanvasHeight() / 2 - getHeight() / 2;

        if (alignment.indexOf(LEFT) > -1)
            x = 0;
        else if (alignment.indexOf(RIGHT) > -1)
            x = Game.getCanvasWidth() - getWidth();
        else if (alignment.indexOf(HCENTER) > -1)
            x = Game.getCanvasWidth() / 2 - getWidth() / 2;
    }
```

```
	public void alignToParent(String alignment) {
		if (parent!=null)
		{
			if (alignment.indexOf(BOTTOM) > -1)
				y = parent.y + parent.getHeight() -
getHeight();
			else if (alignment.indexOf(TOP) > -1)
				y = 0;
			else if (alignment.indexOf(VCENTER) > -1)
				y = parent.getHeight() / 2 - getHeight() / 2;

			if (alignment.indexOf(LEFT) > -1)
				x = 0;
			else if (alignment.indexOf(RIGHT) > -1)
				x = parent.x + parent.getWidth() - getWidth();
			else if (alignment.indexOf(HCENTER) > -1)
				x = parent.getWidth() / 2 - getWidth() / 2;
		}
	}

	public void align(Component anchor, String alignment) {
		if (alignment.indexOf(BOTTOM) > -1)
			y = anchor.y + anchor.getHeight()+1;
		else if (alignment.indexOf(TOP) > -1)
			y = anchor.y - (getHeight() + 1);
		else if (alignment.indexOf(VCENTER) > -1)
			y = anchor.y;

		if (alignment.indexOf(LEFT) > -1)
			x = anchor.x - (getWidth() + 1);
		else if (alignment.indexOf(RIGHT) > -1)
			x = anchor.x + anchor.getWidth() + 1;
		else if (alignment.indexOf(HCENTER) > -1)
			x = anchor.x;
	}

}
```

III.1.10 *game.gui.Container*

```
package game.gui;

import game.core.Color;
import game.core.Key;

import java.util.Vector;

import javax.microedition.lcdui.Graphics;

public class Container extends Component {

	private Vector components = new Vector();
	public int width = 0, height = 0;
	public int focusedIndex = 0;
	private Component lastFocused;
	public int color = Color.WHITE, borderColor = 0;

	public Container(int width, int height) {
		this.width = width;
		this.height = height;
	}
```

```
	public void add(Component c) {
		if (!components.isEmpty())
			c.align((Component) components.lastElement(),
Component.BOTTOM
					+ Component.HCENTER);
		components.addElement(c);
		c.parent = this;
	}

	public void remove(Component c) {
		if (components.indexOf(c) == focusedIndex)
			focusedIndex = -1;
		if (lastFocused != null && components.indexOf(lastFocused)
>= 0)
			setFocused(lastFocused);

		lastFocused = null;

		components.removeElement(c);
		c.parent = null;
	}

	public void removeAll() {
		for (int i = components.size() - 1; i >= 0; i--) {
			remove((Component) components.elementAt(i));
		}
	}

	public int getHeight() {
		return height;
	}

	public int getWidth() {
		return width;
	}

	public void paint(Graphics g) {
		g.translate(x, y);

		g.setColor(color);
		g.fillRect(0, 0, width, height);
		g.setColor(borderColor);
		g.drawRect(0, 0, width, height);

		for (int i = 0; i < components.size(); i++)
			((Component) components.elementAt(i)).paint(g);
		g.translate(-x, -y);
	}

	public void update() {
		if (focused) {

			if (Key.pressed(Key.K_RIGHT) ||
Key.pressed(Key.K_DOWN))
				focusNext();
			else if (Key.pressed(Key.K_LEFT) ||
Key.pressed(Key.K_UP))
				focusPrevious();

			if (focusedIndex >= 0) {
```

```
                        Component c = ((Component)
components.elementAt(focusedIndex));
                        c.update();
                        c.focused = true;
                  }

            } else if (focusedIndex >= 0 && focusedIndex <
components.size())
                  ((Component)
components.elementAt(focusedIndex)).focused = false;
      }

      public boolean focusNext() {
            return setFocused(focusedIndex + 1);
      }

      public boolean focusPrevious() {
            return setFocused(focusedIndex - 1);
      }

      public boolean setFocused(int index) {
            if (index >= 0 && index < components.size()) {
                  Component toBeFocused = ((Component)
components.elementAt(index));
                  if (toBeFocused.isFocusable) {
                        if (focusedIndex >= 0 && focusedIndex <
components.size()) {
                              lastFocused = ((Component) components
                                          .elementAt(focusedIndex));
                              lastFocused.focused = false;

                        }

                        focusedIndex = index;
                        toBeFocused.focused = true;
                        this.focused = true;
                  }
            }
            return false;
      }

      public boolean setFocused(Component c) {
            return setFocused(components.indexOf(c));
      }

      public Component getFocused() {
            if (focusedIndex >= 0 && focusedIndex < components.size())
                  return (Component)
components.elementAt(focusedIndex);
            return null;
      }

      public int size() {
            return components.size();
      }
}
```

III.1.11 *game.gui.Button*

```
package game.gui;

import game.core.Color;
import game.core.Key;

import javax.microedition.lcdui.Font;
import javax.microedition.lcdui.Graphics;

public class Button extends Component {

	private String text;
	private int width, height;
	private int textOffsetX, textOffsetY;
	public int color = Color.WHITE, focusedColor = Color.YELLOW;
	public Runnable action;

	public Button(String text, int width, int height, Runnable
action) {
		this.text = text;
		this.width = width;
		this.height = height;
		this.action = action;
		int textWidth = Font.getDefaultFont().stringWidth(text);
		int textHeight = Font.getDefaultFont().getHeight();
		textOffsetX = width / 2 - textWidth / 2;
		textOffsetY = height / 2 - textHeight / 2;
	}

	public void setText(String text)
	{
		this.text = text;
		int textWidth = Font.getDefaultFont().stringWidth(text);
		int textHeight = Font.getDefaultFont().getHeight();
		textOffsetX = width / 2 - textWidth / 2;
		textOffsetY = height / 2 - textHeight / 2;
	}

	public void update() {
		if (Key.pressed(Key.K_SELECT, 500)) {
			if (action != null)
				action.run();
		}
	}

	public int getHeight() {
		return height;
	}

	public int getWidth() {
		return width;
	}

	public void paint(Graphics g) {
		g.setColor(focused ? focusedColor : color);
		g.fillRect(x, y, width, height);

		g.setColor(0);
		g.drawRect(x, y, width, height);
		g.drawString(text, x + textOffsetX, y + textOffsetY, 0);
	}
```

```
}
```

III.1.12 *game.gui.Label*

```
package game.gui;

import javax.microedition.lcdui.Font;
import javax.microedition.lcdui.Graphics;

public class Label extends Component {

	private String text;
	public int color = 0;

	public Label(String text) {
		this.text = text;
		this.isFocusable = false;
	}

	public int getHeight() {
		return Font.getDefaultFont().getHeight();
	}

	public int getWidth() {
		return Font.getDefaultFont().stringWidth(text);
	}

	public void paint(Graphics g) {
		g.setColor(color);
		g.drawString(text, x, y, 0);
	}

	public void update() {
	}

}
```

III.1.13 *game.gui.InputLabel*

```
package game.gui;

import game.core.Color;
import game.core.Key;
import game.scene.Game;

import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.CommandListener;
import javax.microedition.lcdui.Displayable;
import javax.microedition.lcdui.Font;
import javax.microedition.lcdui.Form;
import javax.microedition.lcdui.Graphics;
import javax.microedition.lcdui.TextField;

public class InputLabel extends Component{
	private int width, height;
	private String text = "";
	private String textThatsActuallyPainted = "";
	public int focusedColor = Color.YELLOW;

	public InputLabel(int width) {
```

```
            this.width = width;
            height = Font.getDefaultFont().getHeight();
      }

      public int getHeight() {
            return height;
      }

      public int getWidth() {
            return width;
      }

      public String getText()
      {
            return text;
      }

      public void paint(Graphics g) {
            g.setColor(Color.WHITE);
            g.fillRect(x, y, width, height);
            g.setColor(focused?focusedColor:0);
            g.drawRect(x, y, width, height);
            g.setColor(0);
            g.drawString(textThatsActuallyPainted, x+2, y+2, 0);
      }

      public void update() {
            if(Key.pressed(Key.K_SELECT, 100))
                  Game.getDisplay().setCurrent(
                              new InputForm("", text, 0));
      }

      public void setText(String txt){
            text = txt;
            textThatsActuallyPainted = text;
            for(int i=text.length()-2;i>0;i--)
            {

      if(Font.getDefaultFont().stringWidth(textThatsActuallyPainted)>w
idth)
                        textThatsActuallyPainted = text.substring(0,
i);
                  else
                        break;
            }
      }

      private class InputForm extends Form implements CommandListener
{
            Command ok = new Command("OK", Command.OK, 0);
            Command cancel = new Command("Cancel", Command.CANCEL, 0);
            TextField textField;

            public InputForm(String title, String text, int
constraints) {
                  super(title);
                  this.setCommandListener(this);
                  textField = new TextField("", text, 100,
constraints);
                  this.append(textField);
                  this.addCommand(ok);
```

```
                this.addCommand(cancel);
        }

        public void commandAction(Command c, Displayable d) {
                this.removeCommand(ok);
                this.removeCommand(cancel);
                this.deleteAll();

                if (c.equals(ok)) {
                        setText(textField.getString());
                }

                Key.clearState();
                Game.getDisplay().setCurrent(Game.getCanvas());
        }

    }
}
```

III.1.14 *game.gui.SimpleImage*

```
package game.gui;

import javax.microedition.lcdui.Graphics;
import javax.microedition.lcdui.Image;

public class SimpleImage extends Component {

    private Image img;
    private int width, height;

    public SimpleImage(Image img) {
        this.img = img;
        this.width = img.getWidth();
        this.height = img.getHeight();
        isFocusable = false;
    }

    public void setImage(Image img)
    {
        this.img = img;
        this.width = img.getWidth();
        this.height = img.getHeight();
    }

    public int getHeight() {
        return height;
    }

    public int getWidth() {
        return width;
    }

    public void paint(Graphics g) {
        g.drawImage(img, x, y, 0);
    }

    public void update() {

    }
}
```

III.1.15 *game.event.Event*

```
package game.event;

public class Event {

	public final int type;

	/**
	 * Constructs a new event with the given type.
	 *
	 * @param type
	 *             - the type of the event
	 */
	public Event(int type) {
		this.type = type;
	}

	/**
	 * Dispatches this event through the {@link EventManager}
	 */
	public void dispatch() {
		EventManager.dispatchEvent(this);
	}

}
```

III.1.16 *game.event.EventListener*

```
package game.event;

public interface EventListener {

	/**
	 * This method is called every time an event is dispatched with
the types
	 * this EventListener is registered for listening.
	 *
	 * @param e
	 *             - the event dispatched.
	 * @see {@link EventManager#register(int, EventListener)}
	 * @see {@link EventManager#dispatchEvent(Event)}
	 */
	public void handleEvent(Event e);

}
```

III.1.17 *game.event.EventManager*

```
package game.event;

import java.util.Vector;

public class EventManager {

	private static ListenerBox[] list = new ListenerBox[256];
	private static ListenerBox allListener;
	private static Vector eventQueue = new Vector();
	private static EventDispatcher eventDispatcher;

	/**
```

```
	 * Be aware that using this method will unregister all listener
that may be
	 * already registered.
	 *
	 * @param numberOfEvents
	 *            - the maximum number of events that will be
supported.
	 */
	public static void reset(int numberOfEvents) {
		list = new ListenerBox[numberOfEvents];
	}

	/**
	 * Register the EventListener to listen for the the given types
of events
	 *
	 * @param eventTypes
	 *            - the types of events the listener wants to be
notified of
	 * @param listener
	 *            - the listener
	 */
	public static void register(int[] eventTypes, EventListener
listener) {
		for (int i = 0; i < eventTypes.length; i++)
			register(eventTypes[i], listener);
	}

	/**
	 * Register the EventListener to listen for the given type of
events
	 *
	 * @param eventType
	 *            - the type of events the listener wants to be
notified of
	 * @param listener
	 *            - the listener
	 */
	public static void register(int eventType, EventListener
listener) {
		ListenerBox box = list[eventType];

		if (box == null) {
			list[eventType] = new ListenerBox(listener, null);
		} else if (box.listener.equals(listener)) {
			return;
		} else {
			while (true) {
				if (box.next != null &&
box.next.listener.equals(listener)) {
					return;
				} else
					box = box.next;

				if (box == null) {
					box = new ListenerBox(listener,
list[eventType]);
					list[eventType] = box;
					return;
				}
			}
```

```
            }
      }

      /**
       * Register the listener to listen for all types of events
       * <p>
       * Note that after being registered for all types of events, the
only way to
       * unregister this listener is through the method
       * {@link #unregister(EventListener)}.
       * </p>
       *
       * @param listener
       *            - the listener
       */
      public static void register(EventListener listener) {
            if (allListener == null) {
                  allListener = new ListenerBox(listener, null);
            } else if (allListener.listener.equals(listener)) {
                  return;
            } else {
                  ListenerBox box = allListener;

                  while (true) {
                        if (box.next != null &&
box.next.listener.equals(listener)) {
                              return;
                        } else
                              box = box.next;

                        if (box == null) {
                              box = new ListenerBox(listener,
allListener);
                              allListener = box;
                              return;
                        }
                  }
            }
      }

      /**
       * Unregister the listener for the given type of events
       *
       * <p>
       * Note that if the listener was previously registered for all
types of
       * events by the method {@link #register(EventListener)}, the
method
       * {@link #unregister(EventListener)} must be used instead.
       * </p>
       *
       * @param eventType
       *            - the type of events the listener no longer wants
to listen
       * @param listener
       *            - the listener
       * @return <code>true</code> if the listener was really
registered for any
       *         type of events
       */
```

```
	public static boolean unregister(int eventType, EventListener
listener) {
		ListenerBox box = list[eventType];

		if (box == null) {
			return false;
		} else if (box.listener.equals(listener)) {
			list[eventType] = box.next;
			return true;
		} else {
			while (true) {
				if (box.next != null &&
box.next.listener.equals(listener)) {
					box.next = box.next.next;
					return true;
				} else
					box = box.next;

				if (box == null)
					return false;
			}
		}
	}

	/**
	 * Unregister the listener for any types of events it may be
registered.
	 *
	 * @param listener
	 *            - the listener
	 * @return <code>true</code> if the listener was really
registered for any
	 *         type of events
	 */
	public static boolean unregister(EventListener listener) {
		boolean unregistered = false;

		for (int i = 0; i < list.length; i++) {
			if (unregister(i, listener))
				unregistered = true;
		}
		if (listener.equals(allListener)) {
			allListener = allListener.next;
			unregistered = true;
		} else {
			ListenerBox box = allListener;
			while (true) {
				if (box == null)
					break;

				if (box.next != null
						&&
box.next.listener.equals(listener)) {
					box.next = box.next.next;
					unregistered = true;
				} else
					box = box.next;
			}
		}
		return unregistered;
	}
```

```
	/**
	 * Unregister the listener for the given types of events
	 * <p>
	 * Note that if the listener was previously registered for all
types of
	 * events by the method {@link #register(EventListener)}, the
method
	 * {@link #unregister(EventListener)} must be used instead.
	 * </p>
	 *
	 * @param eventTypes
	 *            - the types of events the listener no longer wants
to listen
	 * @param listener
	 *            - the listener
	 * @return <code>true</code> if the listener was really
registered for any
	 *         type of events
	 */
	public static boolean unregister(int[] eventTypes, EventListener
listener) {
		boolean unregistered = false;
		for (int i = 0; i < eventTypes.length; i++)
			if (unregister(eventTypes[i], listener))
				unregistered = true;
		return unregistered;
	}

	/**
	 * Dispatches an event
	 *
	 * @param event
	 *            - the event to be dispatched
	 */
	public static void dispatchEvent(Event event) {
		synchronized (eventQueue) {
			eventQueue.addElement(event);
		}
		if (eventDispatcher == null)
			eventDispatcher = new EventDispatcher();
	}

	// internal class for listing listeners
	private static class ListenerBox {
		EventListener listener;
		ListenerBox next;

		ListenerBox(EventListener listener, ListenerBox next) {
			this.listener = listener;
			this.next = next;
		}

		void dispatch(Event event) {
			this.listener.handleEvent(event);
			if (this.next != null)
				this.next.dispatch(event);
		}
	}

	// event dispatcher thread
```

```
	private static class EventDispatcher extends Thread {
		public EventDispatcher() {
			super("EventDispatcher");
			start();
		}

		public void run() {
			while (true) {
				synchronized (eventQueue) {
					while (!eventQueue.isEmpty()) {
						threadEventDispatch((Event)
eventQueue.firstElement());
						eventQueue.removeElementAt(0);
					}
				}
				try {
					sleep(50);
				} catch (InterruptedException e) {
					e.printStackTrace();
				}
			}
		}
	}

	// event thread dispatching method
	private static void threadEventDispatch(Event event) {
		if (list[event.type] != null)
			list[event.type].dispatch(event);
		if (allListener != null)
			allListener.dispatch(event);
	}
}
```

III.1.18 *game.net.SocketClient*

```
package game.net;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;

import javax.microedition.io.Connector;
import javax.microedition.io.SocketConnection;

/**
 * Provides basic socket connection capabilities.
 *
 * @author Danilo D
 *
 */
public abstract class SocketClient {

	private SocketConnection c;
	private DataOutputStream dos;
	private DataInputStream dis;
	private boolean receiving = false;

	/**
	 * Connects through socket to the specified url, and start
listening for
	 * data.
```

```
         *
         * @param url
         *            - the url to connect to.
         * @throws IOException
         */
        public void connect(String url) throws IOException {
              System.out.println("SocketClient.connect()");
              url = "socket://" + url;
              System.out.println("openning socket connection: " + url);
              SocketConnection c = (SocketConnection)
Connector.open(url);

              dis = c.openDataInputStream();
              dos = c.openDataOutputStream();

              System.out.println("connected!");

              onConnection();
              startListening();
        }

        /**
         * Returns <code>true</code> if the connection is opened.
         *
         * @return <code>true</code> if the connection is opened,
<code>false</code>
         *         otherwise.
         */
        public boolean isConnected() {
              if (c == null || dos == null || dis == null)
                    return false;
              try {
                    c.getAddress();
                    dis.available();
                    return true;
              } catch (IOException e) {
                    return false;
              }
        }

        /**
         * Stops the listening thread.
         */
        public void stopListening() {
              System.out.println("SocketClient.stopListening()");
              receiving = false;
        }

        /**
         * Starts listening for data.
         *
         * @see #stopListening
         * @see #onDataReceived
         */
        public void startListening() {
              receiving = true;
              Thread receivingThread = new Thread() {
                    public void run() {
                          while (receiving) {
                                int bytesReceived;
                                try {
```

```
                                    bytesReceived = dis.available();

                                    if (bytesReceived > 0) {
                                          onDataReceived(dis);
                                          System.gc();
                                          received+=bytesReceived;
                                    }
                                    try {
                                          Thread.sleep(50);
                                    } catch (InterruptedException e1) {
                                          e1.printStackTrace();
                                    }
                              } catch (IOException e1) {
                                    listeningException(e1);
                              }
                        }
                  }
            };
            receivingThread.start();
      }

      public abstract void listeningException(Exception e);


      /**
       * This method is called when the connection is opened.
       */
      protected void onConnection() {
      }

      /**
       * This method is called when there is available data on the
input stream.
       *
       * @param dis
       *            - the data input stream
       * @see {@link DataInputStream}
       */
      protected abstract void onDataReceived(DataInputStream dis);

      static int sent = 0;
      static int received = 0;

      /**
       * Sends a packet of bytes through the open connection.
       *
       * @param data
       *            - the array of bytes to send.
       * @throws IOException
       */
      public void send(byte[] data) throws IOException {
            sent+=data.length;
            dos.write(data);
            dos.flush();
            System.gc();
      }

      public void send(byte[] data, ExceptionHandler e) {
            try {
                  send(data);
            } catch (IOException e1) {
```

```
                    e1.printStackTrace();
                    e.onException(e1);
                }
        }

}
```

III.1.19 *game.message.Message*

```
package game.message;

import game.event.Event;
import game.util.ByteArrayUtil;

public class Message extends Event {

        public final byte[] data;

        /**
         * Constructs a new Message object with the given type and no
data.
         *
         * @param type
         *             - the type of the message
         */
        public Message(int type) {
                super(type);
                this.data = new byte[0];
        }

        /**
         * Constructs a new Message object with the given type and data.
         *
         * @param type
         *             - the type of the message.
         * @param data
         *             - the data of the message.
         */
        public Message(int type, byte[] data) {
                super(type);
                this.data = data;
        }

        /**
         * Constructs a new Message object with the given type and an
integer value
         * as data.
         *
         * @param type
         *             - the type of the message.
         * @param value
         *             - the integer value.
         */
        public Message(int type, int value) {
                super(type);
                data = ByteArrayUtil.toByteArray(value);
        }

        /**
         * Constructs a new Message object with the given type and an
integers array
```

```
         * as data.
         *
         * @param type
         *            - the type of the message.
         * @param value
         *            - the integers array
         */
        public Message(int type, int[] value) {
              super(type);
              data = ByteArrayUtil.toByteArray(value);
        }

        /**
         * Constructs a new Message object with the given type and an
String value
         * as data.
         *
         * @param type
         *            - the type of the message.
         * @param value
         *            - the String value
         */
        public Message(int type, String value) {
              super(type);
              data = value.getBytes();
        }

        /**
         * Get the network byte packet format of this message.
         *
         * @return the byte array for this.
         */
        public byte[] serialize() {
              byte[] ba1 = ByteArrayUtil.toByteArray(type);
              byte[] ba2 = ByteArrayUtil.toByteArray(length());
              return ByteArrayUtil.merge(ByteArrayUtil.merge(ba1, ba2),
data);
        }

        /**
         * Returns the length of the data.
         *
         * @return the length of the data.
         */
        public int length() {
              return data.length;
        }

        public int getInt() {
              return ByteArrayUtil.toInt(data);
        }

        public int[] getIntArray() {
              return ByteArrayUtil.toIntArray(data);
        }

        public String getString() {
              return new String(data);
        }

}
```

III.1.20 *game.message.MessageClient*

```
package game.message;

import game.event.EventListener;
import game.event.EventManager;
import game.net.SocketClient;

import java.io.DataInputStream;
import java.io.IOException;

public class MessageClient extends SocketClient {

	public final int TEXT = 0;

	private void messageReceived(Message message) {
		System.out.println("MessageClient.messageReceived(): type="
				+ message.type);
		EventManager.dispatchEvent(message);
	}

	/**
	 * Sends a message to the network and registers the given
listener to that
	 * message type.
	 *
	 * <p>
	 * If you are already registered on {@link EventManager} for
this type of
	 * message, consider the use of the {@link
#sendMessage(Message)} method.
	 * </p>
	 *
	 * @param msg
	 *            - the message to send.
	 * @throws IOException
	 * @see {@link EventManager#register(int, EventListener)}
	 * @see #sendMessage(Message)
	 */
	public void sendMessage(Message msg, EventListener listener)
			throws IOException {
		sendMessage(msg);
		EventManager.register(msg.type, listener);
	}

	/**
	 * Sends a message to the network.
	 * <p>
	 * If you are expecting to handle the response to this message,
make sure
	 * that you are registered for that type of message on {@link
EventManager},
	 * or use {@link #sendMessage(Message, EventListener)} instead.
	 * </p>
	 *
	 * @param msg
	 * @throws IOException
	 * @see {@link EventManager#register(int, EventListener)}
	 * @see #sendMessage(Message, EventListener)
	 */
	public void sendMessage(Message msg) throws IOException {
		send(msg.serialize());
```

```
	}

	protected void onDataReceived(DataInputStream dis) {
		int bytesReceived;
		try {
			bytesReceived = dis.available();
			if (bytesReceived > 0) {

				int type = dis.readInt();

				int length = dis.readInt();

				byte[] data = new byte[length];
				dis.readFully(data);

				messageReceived(new Message(type, data));
			}
		} catch (IOException e) {
			e.printStackTrace();
		}
	}

	public void listeningException(Exception e) {

	}
}
```

III.2 Exemplos

III.2.1 *game.examples.HelloWorld*

```
package game.examples;

import game.core.GameCore;
import game.core.Key;

import javax.microedition.lcdui.Graphics;

public class HelloWorld extends GameCore{

	private int x, y;

	protected void loadGame() {
		x=50;
		y=50;
	}

	protected void paint(Graphics g) {
		g.setColor(0);
		g.drawString("Olá Mundo", x, y, 0);
	}

	protected void update() {
		if(Key.pressed(Key.K_UP))
			y-=2;
		else if(Key.pressed(Key.K_DOWN))
			y+=2;
		if(Key.pressed(Key.K_LEFT))
			x-=2;
		else if(Key.pressed(Key.K_RIGHT))
```

```
                x+=2;
        }
}
```

III.2.2 *game.examples.BallScene*

```
package game.examples;

import game.scene.Game;
import game.scene.GameObject;
import game.scene.Scene;

import java.util.Random;

import javax.microedition.lcdui.Font;
import javax.microedition.lcdui.Graphics;

public class BallScene extends Scene {
        Random r = new Random();
        Font f = Font.getFont(Font.FACE_MONOSPACE, Font.STYLE_PLAIN,
                        Font.SIZE_SMALL);
        int numberOfBalls;

        public void load() {
                for (int i = 0; i < 50; i++) {
                        int startx = (int) (r.nextDouble() * (getWidth() -
20));
                        int starty = (int) (r.nextDouble() * (getHeight() -
20));
                        int velx = r.nextInt(10) - 5;
                        int vely = r.nextInt(10) - 5;
                        int ballColor = r.nextInt();
                        add(new Ball(startx, starty, velx, vely, ballColor));
                }
                numberOfBalls = 50;
        }

        public void addMoreBalls() {
                for (int i = 0; i < 10; i++) {
                        int startx = (int) (r.nextDouble() * (getWidth() -
20));
                        int starty = (int) (r.nextDouble() * (getHeight() -
20));
                        int velx = r.nextInt(10) - 5;
                        int vely = r.nextInt(10) - 5;
                        int ballColor = r.nextInt();
                        add(new Ball(startx, starty, velx, vely, ballColor));
                }
                numberOfBalls += 10;
        }

        public void unload() {
                removeAll();
        }

        public void paint(Graphics g) {
                super.paint(g);
                g.setFont(f);
                g.setColor(0xFFFFFFFF);
                g.fillRect(getWidth() - 80, 0, 80, 20);
                g.setColor(0);
```

```
            g.drawString(numberOfBalls + " balls", getWidth() - 70, 5,
0);
      }

      public class Ball extends GameObject {
            private int x, y;
            private int velx, vely;
            private int color;

            public Ball(int startx, int starty, int velx, int vely, int
color) {
                  this.x = startx;
                  this.y = starty;
                  this.velx = velx;
                  this.vely = vely;
                  this.color = color;
            }

            public void paint(Graphics g) {
                  g.setColor(color);
                  g.fillArc(x, y, 20, 20, 0, 360);
            }

            public void update() {
                  x += velx;
                  y += vely;
                  if (x < 0 || x + 20 > Game.getCanvasWidth())
                        velx = -velx;
                  if (y < 0 || y + 20 > Game.getCanvasHeight())
                        vely = -vely;
            }
      }
}
```

III.2.3 *game.examples.BallsGame*

```
package game.examples;

import game.core.Key;
import game.scene.Game;
import game.scene.GameObject;
import game.scene.Scene;

import java.util.Random;

import javax.microedition.lcdui.Font;
import javax.microedition.lcdui.Graphics;

public class BallsGame extends Game {

      private BallScene ballScene;

      protected void loadGame() {
            ballScene = new BallScene();
            pushScene(ballScene);
      }

      protected void update() {
            super.update();
            if (Key.pressed(Key.K_SELECT)) {
                  ballScene.addMoreBalls();
```

```
		}
	}

	public class BallScene extends Scene {
		Random r = new Random();
		Font f = Font.getFont(Font.FACE_MONOSPACE,
Font.STYLE_PLAIN,
				Font.SIZE_SMALL);
		int numberOfBalls;

		public void load() {
			for (int i = 0; i < 50; i++) {
				int startx = (int) (r.nextDouble() *
(getWidth() - 20));
				int starty = (int) (r.nextDouble() *
(getHeight() - 20));
				int velx = r.nextInt(10) - 5;
				int vely = r.nextInt(10) - 5;
				int ballColor = r.nextInt();
				add(new Ball(startx, starty, velx, vely,
ballColor));
			}
			numberOfBalls = 50;
		}

		public void addMoreBalls() {
			for (int i = 0; i < 10; i++) {
				int startx = (int) (r.nextDouble() *
(getWidth() - 20));
				int starty = (int) (r.nextDouble() *
(getHeight() - 20));
				int velx = r.nextInt(10) - 5;
				int vely = r.nextInt(10) - 5;
				int ballColor = r.nextInt();
				add(new Ball(startx, starty, velx, vely,
ballColor));
			}
			numberOfBalls += 10;
		}

		public void unload() {
			removeAll();
		}

		public void paint(Graphics g) {
			super.paint(g);
			g.setFont(f);
			g.setColor(0xFFFFFFFF);
			g.fillRect(getWidth() - 80, 0, 80, 20);
			g.setColor(0);
			g.drawString(numberOfBalls + " balls", getWidth() -
70, 5, 0);
		}
	}

	public class Ball extends GameObject {
		private int x, y;
		private int velx, vely;
		private int color;
```

```
		public Ball(int startx, int starty, int velx, int vely, int
color) {
			this.x = startx;
			this.y = starty;
			this.velx = velx;
			this.vely = vely;
			this.color = color;
		}

		public void paint(Graphics g) {
			g.setColor(color);
			g.fillArc(x, y, 20, 20, 0, 360);
		}

		public void update() {
			x += velx;
			y += vely;
			if (x < 0 || x + 20 > Game.getCanvasWidth())
				velx = -velx;
			if (y < 0 || y + 20 > Game.getCanvasHeight())
				vely = -vely;
		}
	}
}
```

III.2.4 *game.examples.GUIExample*

```
package game.examples;

import game.core.Color;
import game.gui.Button;
import game.gui.Component;
import game.gui.Container;
import game.gui.InputLabel;
import game.gui.Label;
import game.gui.SimpleImage;
import game.scene.Game;
import game.scene.Scene;
import game.util.ImageUtil;

import javax.microedition.lcdui.Graphics;

public class GUIExample extends Game {

	protected void loadGame() {
		pushScene(new GUISceneExample());
	}

	public class GUISceneExample extends Scene {

		SimpleContainer c1, c2;

		public void load() {
			SimpleImage back = new
SimpleImage(ImageUtil.loadImage(
					"background.png", false));
			add(back);
			back.alignToDisplay(Component.VCENTER +
Component.HCENTER);
			c1 = new SimpleContainer(100, 65, Color.BLUE);
```

```
                  c2 = new SimpleContainer(120, 50, Color.GREY);
                  c2.alignToDisplay(Component.VCENTER +
Component.HCENTER);
                  Label l = new Label("botão 1 pressionado!");
                  l.x = 10;
                  c2.add(l);
                  c2.add(new InputLabel(80));

                  Button b = new Button("Botão 1!", 50, 15, new
Runnable() {
                        public void run() {
                              add(c2);
                        }
                  });
                  b.x = 50 - b.getWidth() / 2;
                  b.y = 10;
                  c1.add(b);
                  c1.add(new Button("Botão 2!", 50, 15, new Runnable()
{
                        public void run() {
                              remove(c2);
                        }
                  }));
                  c1.add(new Button("Botão 3!", 50, 15, new Runnable()
{
                        public void run() {
                              pushScene(new BallScene());
                        }
                  }));

                  add(c1);
                  c1.alignToDisplay(Component.HCENTER +
Component.BOTTOM);
                  c1.setFocused(0);
                  c1.focused = true;
            }

            public void unload() {

            }
      }

      public class SimpleContainer extends Container {
            public int color;

            public SimpleContainer(int w, int h, int color) {
                  super(w, h);
                  this.color = color;
            }

            public void paint(Graphics g) {
                  g.setColor(color);
                  g.fillRect(x, y, width, height);
                  g.setColor(0);
                  g.drawRect(x, y, width, height);
                  super.paint(g);
            }
      }

}
```

III.2.5 *game.examples.EventExample*

```
package game.examples;

import game.core.Color;
import game.core.Key;
import game.event.Event;
import game.event.EventListener;
import game.event.EventManager;
import game.geom.Rectangle;
import game.scene.Game;
import game.scene.GameObject;
import game.scene.Scene;

import javax.microedition.lcdui.Graphics;

public class EventExample extends Game {
	Player player = new Player();
	public final int BLUE_GOAL = 10, RED_GOAL = 11, GREEN_GOAL = 12;

	protected void loadGame() {
		pushScene(new EventExampleScene());
	}

	public class EventExampleScene extends Scene {
		public void load() {
			add(player);
			add(new Goal(50, 120, RED_GOAL));
			add(new Goal(150, 80, BLUE_GOAL));
			add(new Goal(80, 30, GREEN_GOAL));
		}

		public void unload() {
			removeAll();
		}
	}

	public class Player extends GameObject implements EventListener
{
		public int x = 50, y = 190, accx, accy, color =
Color.YELLOW;

		public Player() {
			EventManager.register(BLUE_GOAL, this);
			EventManager.register(RED_GOAL, this);
			EventManager.register(GREEN_GOAL, this);
		}

		public void paint(Graphics g) {
			g.setColor(color);
			g.fillArc((x - 3) + accx * 2, (y - 3) + accy * 2, 16,
16, 0, 360);
			g.setColor(0);
			g.fillArc(x, y, 10, 10, 0, 360);
		}

		public void update() {
			accx = 0;
			accy = 0;

			if (Key.pressed(Key.K_UP))
				accy = -2;
```

```
            else if (Key.pressed(Key.K_DOWN))
                  accy = 2;

            if (Key.pressed(Key.K_RIGHT))
                  accx = 2;
            else if (Key.pressed(Key.K_LEFT))
                  accx = -2;

            x += accx;
            y += accy;
      }

      public Rectangle getBounds() {
            return new Rectangle(x, y, 16, 16);
      }

      public void handleEvent(Event e) {
            switch (e.type) {
            case BLUE_GOAL:
                  color = Color.BLUE;
                  break;
            case GREEN_GOAL:
                  color = Color.GREEN;
                  break;
            case RED_GOAL:
                  color = Color.RED;
                  break;
            }
      }
}

public class Goal extends GameObject {
      private int x, y, goaltype, color;

      public Goal(int x, int y, int goaltype) {
            this.x = x;
            this.y = y;
            this.goaltype = goaltype;
            switch (goaltype) {
            case BLUE_GOAL:
                  color = Color.BLUE;
                  break;
            case GREEN_GOAL:
                  color = Color.GREEN;
                  break;
            case RED_GOAL:
                  color = Color.RED;
                  break;
            }
      }

      public void paint(Graphics g) {
            g.setColor(color);
            g.fillRect(x, y, 15, 15);
      }

      public Rectangle getBounds() {
            return new Rectangle(x, y, 15, 15);
      }

      public void update() {
```

```
                if (player.getBounds().intersects(this.getBounds()))
                      EventManager.dispatchEvent(new
Event(goaltype));
            }
      }
}
```

III.2.6 *game.examples.ClientExample*

```
package game.examples;

import game.core.GameCore;
import game.event.Event;
import game.event.EventListener;
import game.event.EventManager;
import game.message.Message;
import game.message.MessageClient;

import java.io.IOException;

import javax.microedition.lcdui.Graphics;

public class ClientExample extends GameCore implements EventListener {

      public static final int TEXT_MESSAGE = 0;
      private MessageClient client;
      private String servermsg = "";

      protected void loadGame() {
            client = new MessageClient();
            EventManager.register(TEXT_MESSAGE, this);
            try {
                  client.connect("127.0.0.1:12345");
            } catch (IOException e) {
                  e.printStackTrace();
            }
      }

      protected void paint(Graphics g) {
            g.setColor(0);
            g.drawString(servermsg, 5, 50, 0);
      }

      public void handleEvent(Event e) {
            if (e.type == TEXT_MESSAGE) {
                  Message msg = (Message) e;

                  servermsg = "Mensagem do servidor: "
+msg.getString();

                  try {
                        client.sendMessage(new Message(TEXT_MESSAGE,
"Ola servidor."),
                                          this);
                  } catch (IOException e1) {
                        e1.printStackTrace();
                  }
            }
      }

      protected void update() {
```

```
        }
}
```

III.3 Framework para servidores de jogos multi-jogadores

III.3.1 *Game.Net.Server.Connection*

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Game.Net.Server
{
    public abstract class Connection
    {
        internal ConnectionGroup group = null;
        private List<ConnectionListener> listeners = new
List<ConnectionListener>();

        private String id;

        public Connection(String id)
        {
            this.id = id;
        }

        public string GetID()
        {
            return id;
        }

        public void SetID(String id)
        {
            this.id = id;
        }

        public abstract void SendMessage(Message msg);

        public abstract void MessageReceived(Message msg);

        public void dispatchMessage(Message msg)
        {
            if (group != null)
                group.MessageReceived(msg, this);

            ConnectionListener[] list = new
ConnectionListener[listeners.Count];
            listeners.CopyTo(list, 0);

            foreach (ConnectionListener l in list)
            {
                l.MessageReceived(msg, this);
            }
        }

        public void AddConnectionListener(ConnectionListener l)
        {
            lock (listeners)
```

```
            {
                listeners.Add(l);
            }
        }

        public void RemoveConnectionListener(ConnectionListener l)
        {
            lock (listeners)
            {
                listeners.Remove(l);
            }
        }

        public void RemoveAllListeners()
        {
            lock (listeners)
            {
                listeners = new List<ConnectionListener>();
            }
        }
    }
}
```

III.3.2 *Game.Net.Server.ConnectionGroup*

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Collections;

namespace Game.Net.Server
{
    public abstract class ConnectionGroup
    {
        private List<Connection> list = new List<Connection>();

        public List<Connection> GetConnectionList()
        {
            return list;
        }

        public int GetConnectionCount()
        {
            return list.Count;
        }

        public void Add(Connection connection)
        {
            lock (list)
            {
                if(!list.Contains(connection))
                {
                    list.Add(connection);
                    if (connection.group != null)
                        connection.group.Remove(connection);
                    connection.group = this;
                    ConnectionJoinCallback(connection);
                }
            }
        }
```

```
        public void Remove(Connection connection)
        {
            lock (list)
            {
                if (list.Contains(connection))
                {
                    list.Remove(connection);
                    connection.group = null;
                    ConnectionLeaveCallback(connection);
                }
            }
        }

        public Connection GetConnectionByID(String id)
        {
            lock (list)
            {
                int i = 0;
                int count = list.Count;
                while (i < count)
                {
                    if (list[i].GetID() == id)
                    {
                        return list[i];
                    }
                    i++;
                }
            }
            return null;
        }

        public virtual void ConnectionJoinCallback(Connection c){}
        public virtual void ConnectionLeaveCallback(Connection c){}

        public Connection GetConnectionAt(int index)
        {
            lock (list)
            {
                return list[index];
            }
        }

        abstract public void MessageReceived(Message msg, Connection
from);

        public void Multicast(Message msg)
        {
            lock (list)
            {
                int i = 0;
                int count = list.Count;
                while (i < count)
                {
                    list[i].SendMessage(msg);
                    i++;
                }
            }
        }
    }
}
```

III.3.3 *Game.Net.Server.ConnectionListener*

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Game.Net.Server
{
    public interface ConnectionListener
    {
        void MessageReceived(Message msg, Connection c);
    }
}
```

III.3.4 *Game.Net.Server.Lobby*

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Game.Net.Server
{
    public abstract class Lobby : ConnectionGroup
    {
        public const int PLAYER_LIST_REFRESH = 3, START_GAME_REQUEST =
4, START_GAME_ACCEPT = 5, START_GAME_REFUSE = 6,
OPPONENT_NOT_AVAILABLE = 7;

        public override void MessageReceived(Message msg, Connection
from)
        {
            switch (msg.Type)
            {
                case PLAYER_LIST_REFRESH:
                    SendPlayerList(from);
                    break;

                //recebe pedido de inicio de jogo
                case START_GAME_REQUEST:
                    StartGameRequest(msg, from);
                    break;

                //caso chegue uma resposta positiva de pedido de
inicio de jogo
                case START_GAME_ACCEPT:
                    StartGameAccept(msg, from);
                    break;

                case START_GAME_REFUSE:
                    StartGameRefuse(msg, from);
                    break;
            }
        }

        protected void StartGameRefuse(Message msg, Connection from)
        {
            //pega o nome do desafiador
             String oponentID =
Encoding.UTF8.GetString(msg.GetData());
             Connection oponent =
(Connection)GetConnectionByID(oponentID);
```

```
            //se ele ainda estiver disponível
             if (oponent != null)
            {
                String callerName = from.GetID();
                //envia mensagem de que o pedido foi negadp
                 oponent.SendMessage(new Message(START_GAME_REFUSE,
Encoding.UTF8.GetBytes(callerName)));
                 Console.WriteLine("game refused by " + callerName +
"\n");
            }
        }

        protected void StartGameAccept(Message msg, Connection from)
        {
            //pega o nome do desafiador
            String oponentID = Util.GetString(msg.GetData());
            Connection oponent = GetConnectionByID(oponentID);

            //se ele ainda estiver disponível
            if (oponent != null)
            {
                String callerName = from.GetID();
                //envia mensagem de que o pedido foi aceito
                oponent.SendMessage(new Message(START_GAME_ACCEPT,
Util.GetBytes(callerName)));
                //inicia sessao de jogo
                startGameSession(new Connection[] { from, oponent});
                Console.WriteLine("starting game: " + oponent.GetID()
+ " x " + callerName + "\n");
            }
            //caso nao esteja mais disponível
            else
            {
                from.SendMessage(new Message(OPPONENT_NOT_AVAILABLE,
Util.GetBytes(oponentID)));
                SendPlayerList(from);
            }
        }

        protected abstract void startGameSession(Connection[]
connection);

        protected void StartGameRequest(Message msg, Connection from)
        {
            //pega o nome do oponente selecionado
            String oponentID = Util.GetString(msg.GetData());
            Connection oponent = GetConnectionByID(oponentID);

            //se ele ainda estiver na lista e nao estiver ocupado
            if (oponent != null)
            {
                String callerName = from.GetID();
                //envia uma mensagem de pedido de jogo com o nome do
desafiador
                oponent.SendMessage(new Message(START_GAME_REQUEST,
Util.GetBytes(callerName)));
                //notifica o desafiador que a mensagem foi enviada e
seta seu status para "aguardando"
                from.SendMessage(new Message(28));
            }
```

```
            //se o oponente nao estiver mais na lista ou estiver
ocupado
            else
            {
                //envia mensagem de aviso e atualiza a lista do
desafiador
                from.SendMessage(new Message(OPPONENT_NOT_AVAILABLE,
Util.GetBytes(oponentID)));
                SendPlayerList(from);
            }
        }

        public override void ConnectionJoinCallback(Connection c)
        {
            Console.WriteLine("connection join lobby: " + c.GetID() +
"\n");
            SendPlayerList();
        }

        public override void ConnectionLeaveCallback(Connection c)
        {
            Console.WriteLine("connection leave lobby: " + c.GetID() +
"\n");
            SendPlayerList();
        }

        public void SendPlayerList(Connection to)
        {
            String data =
GetFormattedList(GetConnectionList().ToArray());
            to.SendMessage(new Message(PLAYER_LIST_REFRESH,
Util.GetBytes(data)));
        }

        public void SendPlayerList()
        {
            String data =
GetFormattedList(GetConnectionList().ToArray());
            Console.WriteLine("sending player list: " + data + "\n");
            Multicast(new Message(PLAYER_LIST_REFRESH,
Util.GetBytes(data)));
        }

        protected virtual String GetFormattedList(Connection[] list)
        {
            String data = "";
            for (int i = 0; i < list.Length; i++)
                data += list[i].GetID() + "|";

            return data;
        }
    }
}
```

III.3.5 *Game.Net.Server.GameSession*

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Net;

namespace Game.Net.Server
{
    public abstract class GameSession : ConnectionGroup
    {
        public const int PLAYER_READY = 29, YOUR_TURN = 30,
PLAYER_MOVE = 31, GIVE_UP = 32, YOU_WON = 33, YOU_LOSE = 34, TIE = 35;

        public GameSession(Connection[] players)
        {
            Console.WriteLine("new game session");

            for (int i = 0; i < players.Length; i++)
            {
                Console.Write(players[i].GetID() +" ");
                Add(players[i]);
            }
        }

        public override void MessageReceived(Message msg, Connection
from)
        {
            switch (msg.Type)
            {
                case PLAYER_MOVE:
                    HandlePlayerMove(msg, from);
                    break;
            }
        }

        protected abstract void HandlePlayerMove(Message move,
Connection from);
    }
}
```

III.3.5 *Game.Net.Server.Message*

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Game.Net.Server
{
    public class Message
    {
        public readonly int Type;
        private byte[] Data;

        public Message(int type)
        {
            this.Type = type;
            Data = new byte[0];
        }

        public Message(int type, byte[] data)
```

```
        {
            this.Type = type;
            this.Data = data;
        }

        public int Length()
        {
            if (Data == null)
                return 0;
            return Data.Length;
        }

        public byte[] GetData()
        {
            return Data;
        }
    }
}
```

III.3.6 *Game.Net.Server.SocketConnection*

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Net.Sockets;
using System.IO;
using System.Net;

namespace Game.Net.Server
{
    public class SocketConnection : Connection
    {
        private byte[] InBuffer;
        private byte[] OutBuffer;
        private BinaryReader br;
        private BinaryWriter bw;
        private AsyncCallback receiveCallback;
        readonly Socket clientSocket;

        public SocketConnection(String id, Socket socket):base(id)
        {
            InBuffer = new byte[1024];
            br = new BinaryReader(new MemoryStream(InBuffer));

            OutBuffer = new byte[1024];
            bw = new BinaryWriter(new MemoryStream(OutBuffer));

            clientSocket = socket;
            receiveCallback = new AsyncCallback(OnDataReceived);
            BeginReceive();
        }

        public void BeginReceive()
        {
            try
            {
                clientSocket.BeginReceive(InBuffer, 0,
InBuffer.Length, SocketFlags.None,
                                          receiveCallback,
clientSocket);
            }
```

```
            catch (SocketException e)
            {
                Console.WriteLine("conexão fechada: id=" + GetID());
                this.RemoveAllListeners();
                if (group != null)
                    group.Remove(this);
                clientSocket.Shutdown(SocketShutdown.Both);
                //notificar queda
            }
        }

        public override void MessageReceived(Message msg)
        {

        }

        public void OnDataReceived(IAsyncResult asyn)
        {
            try
            {
                int bytesReceived = clientSocket.EndReceive(asyn);
            }
            catch (Exception e)
            {
                Console.WriteLine("conexão fechada: id=" + GetID());
                this.RemoveAllListeners();
                if (group != null)
                    group.Remove(this);
                clientSocket.Shutdown(SocketShutdown.Both);
                //notificar queda
            }

            int type = IPAddress.NetworkToHostOrder(br.ReadInt32());

            int dataLength =
IPAddress.NetworkToHostOrder(br.ReadInt32());

            if (dataLength > 1024)
            {
                Console.WriteLine("WARNING: size exceeded " +
dataLength);
                //Begin receiving again
                BeginReceive();
                br.BaseStream.Position = 0;
                return;
            }

            byte[] data = new byte[dataLength];

            Array.Copy(InBuffer, (int)br.BaseStream.Position, data, 0,
dataLength);

            Message msg = new Message(type, data);
            dispatchMessage(msg);
            MessageReceived(msg);

            //Begin receiving again
            BeginReceive();

            br.BaseStream.Position = 0;
        }
```

```
        public override void SendMessage(Message msg)
        {
            bw.Write(IPAddress.HostToNetworkOrder(msg.Type));
            bw.Write(IPAddress.HostToNetworkOrder(msg.Length()));
            bw.Write(msg.GetData());
            bw.Flush();

            try
            {
                byte[] data = new byte[bw.BaseStream.Position];
                Array.Copy(OutBuffer, 0, data, 0, data.Length);
                clientSocket.Send(data);
                bw.BaseStream.Position = 0;
            }
            catch (Exception e)
            {
                Console.WriteLine("conexão fechada: id=" + GetID());
                this.RemoveAllListeners();
                if (group != null)
                    group.Remove(this);
                clientSocket.Shutdown(SocketShutdown.Both);
            }
        }
    }
}
```

III.3.7 *Game.Net.Server.SocketServer*

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Net.Sockets;
using System.Net;
using System.Collections;

namespace Game.Net.Server
{

    public abstract class SocketServer
    {
        Socket serverSocket;
        private AsyncCallback clientConnectCallback;

        public SocketServer(int port)
        {
            Console.WriteLine("initializing server..");

            clientConnectCallback = new
AsyncCallback(OnClientConnect);

            //Creates new server socket
            serverSocket = new Socket(AddressFamily.InterNetwork,
SocketType.Stream, ProtocolType.Tcp);

            //Gets local address
            IPEndPoint localEndPoint = new IPEndPoint(IPAddress.Any,
port);

            //Binds serverSocket to local address
            Console.WriteLine("binding socket..");
```

```
            serverSocket.Bind(localEndPoint);

            Console.WriteLine("adress: " +
serverSocket.LocalEndPoint);

            //Additional options
            serverSocket.Blocking = false;
            serverSocket.UseOnlyOverlappedIO = false;

            //Start listening
            serverSocket.Listen(1);

            //Start accepting connections
            beginAccept();

            Console.WriteLine("waiting for connection..\n");
        }

        public void beginAccept()
        {
            //When connection is accepted calls OnClientConnect
            serverSocket.BeginAccept(clientConnectCallback, null);

        }

        private void OnClientConnect(IAsyncResult asyn)
        {
            try
            {
                lock (this)
                {
                    //Accepts the connection and creates a working
socket for the client
                    Socket clientSocket =
serverSocket.EndAccept(asyn);

                    ConnectionAccepted(clientSocket);

                    Console.WriteLine("\nnew connection accepted: " +
clientSocket.LocalEndPoint);
                }

                //Start accepting again
                beginAccept();
            }
            catch (SocketException se)
            {
                Console.WriteLine(se.ToString());
            }

        }

        abstract protected void ConnectionAccepted(Socket
clientSocket);

    }
}
```

III.3.8 *Game.Net.Server.MySQLInterface*

```
using System;
using System.Collections.Generic;
using System.Text;
using MySql.Data.MySqlClient;
using System.Data;

namespace Game.Net.Server
{
    class MySQLInterface
    {
        private MySqlConnection mConn;

        public void Connect(String server, String database, String
user, String pass)
        {
            Console.WriteLine("connecting to database..");
            Console.WriteLine("server=" + server + "  database=" +
database + "  uid=" + user);
            String connectionString = "server=" + server +
";database=" + database + ";uid=" + user + ";pwd=" + pass + ";";
            mConn = new MySqlConnection(connectionString);
            mConn.Open();
        }

        public void Connect(String connectionString)
        {
            mConn = new MySqlConnection(connectionString);
            mConn.Open();
        }

        public void Disconnect()
        {
            mConn.Close();
        }

        public void Query(String query)
        {
            Console.WriteLine("quering: " + query);
            if (mConn.State == ConnectionState.Open)
            {
                MySqlDataAdapter mAdapter = new
MySqlDataAdapter(query, mConn);
            }
        }

        public void Insert(String table, String[] values)
        {
            if (mConn.State == ConnectionState.Open)
            {
                String query = "INSERT INTO '" + table + "' VALUES (";
                for (int i = 0; i < values.Length; i++)
                {
                    if (i > 0)
                        query += ", ";
                    query += values[i];
                }
                query += ")";
                Console.WriteLine("quering: " + query);
                new MySqlDataAdapter(query, mConn);
            }
```

```
        }

        public DataRowCollection Select(String query, String table)
        {
            Console.WriteLine("quering: " + query);

            DataSet mDataSet = new DataSet();
            if (mConn.State == ConnectionState.Open)
            {
                MySqlDataAdapter mAdapter = new
MySqlDataAdapter(query, mConn);
                try
                {
                    mAdapter.Fill(mDataSet, table);
                }
                catch (MySqlException e)
                {
                    Console.WriteLine("exc: " + e.Message + " n" +
e.ErrorCode);
                }
            }
            if (mDataSet.Tables[table] != null)
                return mDataSet.Tables[table].Rows;
            return null;
        }
    }
}
```

III.3.9 *Game.Net.Server.ServerExample*

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Net.Sockets;

namespace Game.Net.Server
{
    class ServerExample : SocketServer
    {
        private LobbyExample lobby;
        public ServerExample() : base(12345)
        {
            lobby = new LobbyExample();
        }

        protected override void ConnectionAccepted(Socket
clientSocket)
        {
            SocketConnection client = new SocketConnection("clientID",
clientSocket);
            lobby.Add(client);
        }
    }

    class LobbyExample : Lobby
    {
        public const int TEXT = 0;

        public override void MessageReceived(Message msg, Connection
from)
        {
```

```
            if(msg.Type == 0)
            {
                Console.WriteLine("Mensagem do cliente: "
+Util.GetString(msg.GetData()));
            }
            else
                base.MessageReceived(msg, from);
        }

        public override void ConnectionJoinCallback(Connection c)
        {
            System.Threading.Thread.Sleep(5000);
            c.SendMessage(new Message(TEXT, Util.GetBytes("Ola
cliente.")));
        }

        protected override void startGameSession(Connection[]
connection)
        {

        }
    }

}
```

III.4 *Jan Ken Po*

III.4.1 *game.jankenpo.MainMenu*

```
package game.jankenpo;

import game.gui.Button;
import game.gui.Component;
import game.gui.Container;
import game.gui.SimpleImage;
import game.scene.Game;
import game.scene.Scene;

import java.io.IOException;

import javax.microedition.lcdui.Image;

public class MainMenu extends Scene {

      public void load() {
            try {
                  SimpleImage logo = new SimpleImage(Image
                              .createImage("/jkp_logo.png"));
                  add(logo);
                  logo.alignToDisplay(Component.HCENTER);
                  logo.y += 20;
            } catch (IOException e) {
                  e.printStackTrace();
            }
            Container menu = new Container(50, 40);
            Button start = new Button("Iniciar", 50, 20, new Runnable()
{
                  public void run() {
```

```
                        Game.getGameInstance().pushScene(new
NickSelection());
                  }
            });
            menu.add(start);
            Button exit = new Button("Sair", 50, 20, new Runnable() {
                  public void run() {
                        Game.getGameInstance().notifyDestroyed();
                  }
            });
            menu.add(exit);
            add(menu);
            menu.alignToDisplay(Component.HCENTER + Component.VCENTER);
            menu.y += 30;
            menu.setFocused(0);
      }

      public void unload() {

      }
}
```

III.4.2 *game.jankenpo.NickSelection*

```
package game.jankenpo;

import game.event.Event;
import game.event.EventListener;
import game.gui.Button;
import game.gui.Component;
import game.gui.Container;
import game.gui.InputLabel;
import game.gui.Label;
import game.message.Message;
import game.scene.Game;
import game.scene.Scene;

public class NickSelection extends Scene implements EventListener {

      public static final int LOGIN = 198;
      private InputLabel apelido;
      private Container body;

      public void load() {
            insertNickname();
      }

      public void insertNickname() {
            removeAll();
            body = new Container(Game.getCanvasWidth() - 30, 100);
            body.alignToDisplay(Component.HCENTER + Component.VCENTER);
            body.add(new Label("Informe um apelido para "));
            body.add(new Label("entrar na sala."));

            apelido = new InputLabel(130);
            body.add(apelido);
            apelido.y += 10;

            Button conectar = new Button("Conectar", 60, 20, new
Runnable() {
                  public void run() {
```

```
                        if (apelido.getText().length() < 1) {
                              alert("Apelido invalido");
                              return;
                        }

                        try {
                              JKPClient.connect();
                              JKPClient.sendMessage(
                                          new Message(LOGIN,
apelido.getText()),
                                          NickSelection.this);
                        } catch (Exception e) {
                              alert("Houve um erro na conexão!");
                        }
                  }
            });
            body.add(conectar);
            conectar.y += 10;

            Button cancelar = new Button("Cancelar", 60, 20, new
Runnable() {
                  public void run() {
                        Game.getGameInstance().popScene();
                  }
            });
            body.add(cancelar);
            cancelar.align(conectar, Component.RIGHT +
Component.VCENTER);
            cancelar.x += 5;

            add(body);
            body.setFocused(2);
      }

      public void alert(String text) {
            removeAll();

            body = new Container(Game.getCanvasWidth() - 30, 100);

            Label lbl = new Label(text);
            body.add(lbl);
            lbl.alignToParent(Component.HCENTER + Component.VCENTER);
            lbl.y -= 10;

            Button ok = new Button("OK", 40, 20, new Runnable() {
                  public void run() {
                        insertNickname();
                  }
            });

            body.add(ok);
            ok.alignToParent(Component.HCENTER);
            ok.y += 5;

            add(body);
            body.alignToDisplay(Component.VCENTER + Component.HCENTER);
            body.setFocused(ok);
      }

      public void unload() {
            removeAll();
```

```
	}

	public void handleEvent(Event e) {
		if (e.type == LOGIN) {
			Message m = (Message) e;
			if (m.getString().equals("k")) {
				Game.getGameInstance().setScene(new Lobby());
				JanKenPo.player = apelido.getText();
			} else
				alert("Apelido em uso!");
		}
	}

}
```

III.4.3 *game.jankenpo.Lobby*

```
package game.jankenpo;

import game.event.Event;
import game.event.EventListener;
import game.event.EventManager;
import game.gui.Button;
import game.gui.Component;
import game.gui.Container;
import game.gui.Label;
import game.message.Message;
import game.scene.Game;
import game.scene.Scene;
import game.util.StringUtil;

import java.io.IOException;

public class Lobby extends Scene implements EventListener {

	private String[] players = new String[] {};
	public static final int TEXT = 0, PLAYER_LIST_REFRESH = 3,
			START_GAME_REQUEST = 4, START_GAME_ACCEPT = 5,
			START_GAME_REFUSE = 6;

	public void load() {
		showList();
		EventManager.register(PLAYER_LIST_REFRESH, this);
		EventManager.register(START_GAME_REQUEST, this);
		EventManager.register(START_GAME_ACCEPT, this);
		EventManager.register(START_GAME_REFUSE, this);
	}

	private void showList() {
		removeAll();

		Container body = new Container(Game.getCanvasWidth() - 20,
160);

		Label lbl = new Label("Selecione um oponente:");
		body.add(lbl);

		if (players.length == 0) {
			Label lbl2 = new Label("Não há oponentes
disponíveis.");
			body.add(lbl2);
```

```
                  lbl2.alignToParent(Component.VCENTER +
Component.HCENTER);
            } else
                  for (int i = 0; i < players.length; i++) {
                        final String opponent = players[i];
                        body.add(new Button(players[i],
body.getWidth(), 20,
                                    new Runnable() {
                                          public void run() {
                                                try {

      JKPClient.sendMessage(new Message(

      START_GAME_REQUEST, opponent),

      Lobby.this);
                                                      waitForOthers();
                                                } catch (IOException e)
{

      e.printStackTrace();
                                                }
                                          }
                                    }));
                  }

            Button exit = new Button("Sair", 60, 20, new Runnable() {
                  public void run() {
                        Game.getGameInstance().popScene();
                  }
            });
            body.add(exit);
            exit.alignToParent(Component.BOTTOM + Component.RIGHT);

            add(body);
            body.setFocused(1);
            body.alignToDisplay(Component.HCENTER + Component.VCENTER);
      }

      private void waitForOthers() {
            removeAll();
            Container body = new Container(Game.getCanvasWidth() - 20,
160);

            Label aguarde = new Label("Aguarde o outro jogador.");
            body.add(aguarde);
            aguarde.alignToDisplay(Component.HCENTER +
Component.VCENTER);

            Button cancel = new Button("Cancelar", 60, 20, new
Runnable() {
                  public void run() {
                        showList();
                  }
            });
            body.add(cancel);
            body.setFocused(cancel);

            add(body);
      }
```

```
	private void invited(final String by) {
		removeAll();
		Container body = new Container(Game.getCanvasWidth() - 20,
60);

		Label lbl = new Label(by + " quer jogar com você.");
		body.add(lbl);

		Button accept = new Button("Aceitar", 60, 20, new
Runnable() {
			public void run() {
				try {
					JKPClient.sendMessage(new
Message(START_GAME_ACCEPT, by));
					Game.getGameInstance().pushScene(new
GameRoom(by));
				} catch (IOException e) {
					e.printStackTrace();
				}
			}
		});
		body.add(accept);
		accept.y += 10;

		Button refuse = new Button("Recusar", 60, 20, new
Runnable() {
			public void run() {
				try {
					JKPClient.sendMessage(new
Message(START_GAME_REFUSE, by));
					showList();
				} catch (IOException e) {
					e.printStackTrace();
				}
			}
		});
		body.add(refuse);
		refuse.align(accept, Component.VCENTER);
		refuse.alignToParent(Component.RIGHT);

		add(body);
		body.setFocused(accept);
		body.alignToDisplay(Component.HCENTER + Component.VCENTER);
	}

	public void opponentRefused(String opponent) {
		removeAll();
		Container body = new Container(Game.getCanvasWidth() - 20,
160);

		Label aguarde = new Label(opponent + " não quis jogar com
você.");
		body.add(aguarde);
		aguarde.alignToDisplay(Component.HCENTER +
Component.VCENTER);

		Button ok = new Button("OK", 40, 20, new Runnable() {
			public void run() {
				showList();
			}
		});
```

```
            body.add(ok);
            body.setFocused(ok);

            add(body);
      }

      public void unload() {
            removeAll();
            EventManager.unregister(PLAYER_LIST_REFRESH, this);
            EventManager.unregister(START_GAME_REQUEST, this);
            EventManager.unregister(START_GAME_ACCEPT, this);
            EventManager.unregister(START_GAME_REFUSE, this);
      }

      // Método para tratamento de eventos
      public void handleEvent(Event e) {
            switch (e.type) {
            // Mensagem para atualização de lista
            case PLAYER_LIST_REFRESH:
                  Message m = (Message) e;
                  String resp = m.getString();
                  resp = removeSelf(resp);

                  if (resp.length() > 1)
                        players = StringUtil.split(resp, "|");

                  showList();
                  break;
            // Mensagem
            case START_GAME_REQUEST:
                  m = (Message) e;
                  resp = m.getString();
                  System.out.println("received game request from " +
resp);
                  invited(resp);
                  break;
            case START_GAME_ACCEPT:
                  m = (Message) e;
                  resp = m.getString();
                  System.out.println("received game accept from " +
resp);
                  Game.getGameInstance().pushScene(new GameRoom(resp));
                  break;
            case START_GAME_REFUSE:
                  m = (Message) e;
                  resp = m.getString();
                  System.out.println("received game refuse from " +
resp);
                  opponentRefused(resp);
                  break;
            }
      }

      private String removeSelf(String response) {
            String pname = JanKenPo.player + "|";
            int index = response.indexOf(pname);

            response = response.substring(0, index)
                        + response.substring(index + pname.length());

            if (response.length() > 0)
```

```
                  response = response.substring(0, response.length() -
1);

            return response;
      }
}
```

III.4.4 *game.jankenpo.GameRoom*

```
package game.jankenpo;

import game.event.Event;
import game.event.EventListener;
import game.event.EventManager;
import game.gui.Button;
import game.gui.Component;
import game.gui.Container;
import game.gui.Label;
import game.gui.SimpleImage;
import game.message.Message;
import game.scene.Game;
import game.scene.Scene;

import java.io.IOException;

import javax.microedition.lcdui.Graphics;
import javax.microedition.lcdui.Image;

public class GameRoom extends Scene implements EventListener {

      public static final int PLAY_AGAIN = 29, YOUR_TURN = 30,
PLAYER_MOVE = 31,
                  GIVE_UP = 32, YOU_WON = 33, YOU_LOSE = 34, TIE = 35;

      public static final int PAPEL = 0, TESOURA = 1, PEDRA = 2;

      String opponent;
      int jogada;
      String[] jogadas = { "papel", "tesoura", "pedra" };

      public GameRoom(String opponentName) {
            this.opponent = opponentName;
            EventManager.register(YOUR_TURN, this);
            EventManager.register(GIVE_UP, this);
            EventManager.register(YOU_WON, this);
            EventManager.register(YOU_LOSE, this);
            EventManager.register(TIE, this);
      }

      public void load() {
            waitForOthers();
      }

      private void makeAMove() {
            removeAll();
            Container body = new Container(Game.getCanvasWidth() - 20,
160);

            Label lbl = new Label("Faça sua jogada:");
            body.add(lbl);
```

```
            try {
                  ImageButton papelbtn = new ImageButton(Image
                              .createImage("/papel1.png"), new
Runnable() {
                        public void run() {
                              jogada = PAPEL;
                              try {
                                    JKPClient.sendMessage(new
Message(PLAYER_MOVE, PAPEL));
                                    waitForOthers();
                              } catch (IOException e) {
                                    e.printStackTrace();
                              }
                        }
                  });
                  ImageButton tesourabtn = new ImageButton(Image
                              .createImage("/tesoura1.png"), new
Runnable() {
                        public void run() {
                              jogada = TESOURA;
                              try {
                                    JKPClient
                                                .sendMessage(new
Message(PLAYER_MOVE, TESOURA));
                                    waitForOthers();
                              } catch (IOException e) {
                                    e.printStackTrace();
                              }
                        }
                  });
                  ImageButton pedrabtn = new ImageButton(Image
                              .createImage("/pedra1.png"), new
Runnable() {
                        public void run() {
                              jogada = PEDRA;
                              try {
                                    JKPClient.sendMessage(new
Message(PLAYER_MOVE, PEDRA));
                                    waitForOthers();
                              } catch (IOException e) {
                                    e.printStackTrace();
                              }
                        }
                  });

                  body.add(papelbtn);
                  papelbtn.y += 25;
                  papelbtn.x += 10;
                  body.add(tesourabtn);
                  tesourabtn.align(papelbtn, Component.RIGHT +
Component.VCENTER);
                  tesourabtn.x += 2;
                  body.add(pedrabtn);
                  pedrabtn.align(tesourabtn, Component.RIGHT +
Component.VCENTER);
                  pedrabtn.x += 2;
            } catch (IOException e) {
                  e.printStackTrace();
            }
            body.setFocused(1);
```

```
            add(body);
            body.alignToDisplay(Component.HCENTER + Component.VCENTER);
      }

      private void waitForOthers() {
            removeAll();
            Container body = new Container(Game.getCanvasWidth() - 20,
160);

            Label aguarde = new Label("Aguarde o outro jogador.");
            body.add(aguarde);
            aguarde.alignToDisplay(Component.HCENTER +
Component.VCENTER);

            Button cancel = new Button("Cancelar", 60, 20, new
Runnable() {
                  public void run() {
                        Game.getGameInstance().popScene();
                  }
            });
            body.add(cancel);
            body.setFocused(cancel);

            add(body);
      }

      private void result(int result) {
            removeAll();
            Container body = new Container(Game.getCanvasWidth() - 20,
160);

            Label lbl = new Label(result == 0 ? "Empate!"
                        : result == 1 ? "Você perdeu." : "Você
ganhou!");
            body.add(lbl);

            Label lbl2 = new Label(JanKenPo.player);
            body.add(lbl2);
            lbl2.y += 20;

            Label lbl3 = new Label(opponent);
            body.add(lbl3);
            lbl3.align(lbl2, Component.VCENTER);
            lbl3.alignToParent(Component.RIGHT);

            try {
                  SimpleImage img = new
SimpleImage(Image.createImage("/"
                              + jogadas[jogada] + "1.png"));
                  SimpleImage img2 = new
SimpleImage(Image.createImage("/"
                              + jogadas[(jogada + result) % 3] +
"2.png"));
                  body.add(img);
                  img.align(lbl2, Component.BOTTOM +
Component.HCENTER);
                  img.y += 10;
                  body.add(img2);
                  img2.align(lbl3, Component.BOTTOM);
                  img2.alignToParent(Component.RIGHT);
                  img2.y += 10;
```

```
            } catch (IOException e) {
                  e.printStackTrace();
            }

            Button cont = new Button("Jogar novamente", 90, 20, new
Runnable() {
                  public void run() {
                        try {
                              JKPClient.sendMessage(new
Message(PLAY_AGAIN),
                                          GameRoom.this);
                        } catch (IOException e) {
                              e.printStackTrace();
                        }
                        waitForOthers();
                  }
            });
            body.add(cont);
            cont.alignToParent(Component.BOTTOM + Component.LEFT);

            Button leave = new Button("Sair", 40, 20, new Runnable() {
                  public void run() {
                        try {
                              JKPClient.sendMessage(new
Message(GIVE_UP), GameRoom.this);
                        } catch (IOException e) {
                              e.printStackTrace();
                        }
                        Game.getGameInstance().popScene();
                  }
            });
            body.add(leave);
            leave.alignToParent(Component.RIGHT);
            leave.align(cont, Component.VCENTER);

            body.setFocused(cont);

            add(body);
            body.alignToDisplay(Component.HCENTER + Component.VCENTER);
      }

      public void opponentGaveUp() {
            removeAll();
            Container body = new Container(Game.getCanvasWidth() - 20,
160);

            Label aguarde = new Label(opponent + " desistiu.");
            body.add(aguarde);
            aguarde.alignToDisplay(Component.HCENTER +
Component.VCENTER);

            Button ok = new Button("OK", 40, 20, new Runnable() {
                  public void run() {
                        Game.getGameInstance().popScene();
                  }
            });
            body.add(ok);
            body.setFocused(ok);

            add(body);
      }
```

```
	public void unload() {
		removeAll();
	}

	public class ImageButton extends Button {

		private Image img;

		public ImageButton(Image img, Runnable action) {
			super("", img.getWidth(), img.getHeight(), action);
			this.img = img;
		}

		public void paint(Graphics g) {
			g.drawImage(img, x, y, 0);
			g.setColor(focused ? focusedColor : color);
			g.drawRect(x, y, getWidth(), getHeight());
		}
	}

	// Método para tratamento dos eventos
	public void handleEvent(Event e) {
		switch (e.type) {
		case YOUR_TURN:
			makeAMove();
			break;
		case GIVE_UP:
			opponentGaveUp();
			break;
		case YOU_WON:
			result(2);
			break;
		case YOU_LOSE:
			result(1);
			break;
		case TIE:
			result(0);
			break;
		}
	}
}
```

III.4.5 *game.jankenpo.JKPClient*

```
package game.jankenpo;

import game.event.EventListener;
import game.message.Message;
import game.message.MessageClient;

import java.io.IOException;


public class JKPClient {

	// Instância da classe MessageClient
	private static final MessageClient client = new MessageClient();

	private JKPClient(){}
```

```
	// Método estático para conexão
	public static void connect() throws IOException {
		client.connect("127.0.0.1:3692");
	}

	// Métodos estáticos para envio de mensagens
	public static void sendMessage(Message m, EventListener
listener)
			throws IOException {
		client.sendMessage(m, listener);
	}

	public static void sendMessage(Message m) throws IOException {
		client.sendMessage(m);
	}
}
```

III.4.6 *Game.JanKenPo.JKPServer*

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Net.Sockets;
using Game.Net.Server;

namespace Game.JanKenPo
{
    class JKPServer : SocketServer, ConnectionListener
    {
        // Sala de jogadores
        private JKPLobby lobby;

        // Tipo de evento para o login
        public const int LOGIN_MESSAGE = 198;

        public JKPServer() : base(3692)
        {
            Console.WriteLine("starting jkpserver..");
            lobby = new JKPLobby();
        }

        // Recebe o pedido de conexão e adiciona o servidor como
listener
        // para o tratamento da mensagem de login
        protected override void ConnectionAccepted(Socket conn)
        {
            new SocketConnection("SocketConnection",
conn).AddConnectionListener(this);
        }

        // Trata a mensagem de login recebida
        public void MessageReceived(Message msg, Connection c)
        {
            switch (msg.Type)
            {
                case LOGIN_MESSAGE:
                    byte[] bytes;

                    String nickname = Util.GetString(msg.GetData());
```

```
                    Console.WriteLine("login requested by: " +
nickname);

                    // Caso o apelido esteja em uso, notifica o
cliente
                    if (lobby.GetConnectionByID(nickname) == null)
                    {
                        bytes = Util.GetBytes("k");
                        c.SetID(nickname);

                        Console.WriteLine("login accepted: " +
nickname);
                        c.RemoveConnectionListener(this);

                        c.SendMessage(new Message(LOGIN_MESSAGE,
bytes));
                        lobby.Add(c);
                    }
                    else
                    {
                        bytes = Util.GetBytes("n");
                        Console.WriteLine("login denied: " + nickname
+"already in use");
                        c.SendMessage(new Message(LOGIN_MESSAGE,
bytes));
                    }
                    break;
            }
        }
    }

}
```

III.4.7 *Game.JanKenPo.JKPLobby*

```
using System;
using System.Collections.Generic;
using System.Text;
using Game.Net.Server;

namespace Game.JanKenPo
{
    public class JKPLobby : Lobby
    {
        protected override void startGameSession(Connection[]
connection)
        {
            // Inicia uma nova sessão de jogo
            new JKPSession(connection, this);
        }
    }
}
```

III.4.8 *Game.JanKenPo.JKPSession*

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Net;
using Game.Net.Server;
```

```
namespace Game.JanKenPo
{
    public class JKPSession : GameSession
    {
        Connection player1, player2;

        // Atributos para armazenagem das jogadas e pontuação
        private int jogada1 = -1, jogada2 = -1, score1 = 0, score2 =
0;

        // Tipos de mensagens referentes ao jogo
        public const int PAPEL = 0, TESOURA = 1, PEDRA = 2, PLAY_AGAIN
= 29;

        // Mantém referência à sala para que os jogadores possam
        // retornar após a sessão
        private JKPLobby lobby;

        public JKPSession(Connection[] players, JKPLobby
lobby):base(players)
        {
            this.lobby = lobby;
            player1 = players[0];
            player2 = players[1];

            // Sessão está pronta para receber jogadas
            player1.SendMessage(new Message(YOUR_TURN));
            player2.SendMessage(new Message(YOUR_TURN));
        }

        // Trata as mensagens específicas
        public override void MessageReceived(Message msg, Connection
from)
        {
            switch (msg.Type)
            {
                // Jogador desistiu
                case GIVE_UP:
                    if (from == player1)
                        player2.SendMessage(new Message(GIVE_UP,
Util.GetBytes(player1.GetID())));
                    else
                        player1.SendMessage(new Message(GIVE_UP,
Util.GetBytes(player2.GetID())));

                    Remove(player1);
                    Remove(player2);
                    lobby.Add(player1);
                    lobby.Add(player2);

                    break;

                // Jogador quer jogar novamente
                case PLAY_AGAIN:
                    if (from == player1)
                        jogada1 = -1;
                    else
                        jogada2 = -1;

                    if (jogada2 == -1 && jogada1 == -1)
```

```
                    {
                        player1.SendMessage(new Message(YOUR_TURN));
                        player2.SendMessage(new Message(YOUR_TURN));
                    }
                    break;
            }
            base.MessageReceived(msg, from);
        }

        // Método para tratamento das jogadas
        protected override void HandlePlayerMove(Message move,
Connection from)
        {
            int jogada =
IPAddress.NetworkToHostOrder(BitConverter.ToInt32(move.GetData(),0));

            if (from == player1)
            {
                Console.WriteLine("player 1 move: " + jogada);
                jogada1 = jogada;
            }
            else
            {
                Console.WriteLine("player 2 move: " + jogada+"\n");
                jogada2 = jogada;
            }

            //Ambos já jogaram, calcula vitória
            if (jogada1 != -1 && jogada2 != -1)
            {
                CheckWinner();
            }
        }


        private void CheckWinner()
        {
            // Empate
            if (jogada1 == jogada2)
            {
                player1.SendMessage(new Message(TIE));
                player2.SendMessage(new Message(TIE));
            }
            // Player 1 ganha
            else if ((jogada1 == PAPEL && jogada2 == PEDRA)
                    || (jogada1 == PEDRA && jogada2 == TESOURA)
                    || (jogada1 == TESOURA && jogada2 == PAPEL)) {
                score1++;
                player1.SendMessage(new Message(YOU_WON));
                player2.SendMessage(new Message(YOU_LOSE));
            }
            // Player 2 GANHA
            else {

                score2++;
                player1.SendMessage(new Message(YOU_LOSE));
                player2.SendMessage(new Message(YOU_WON));
            }
        }
    }
}
```

III.5 *JobFun*

III.5.1 *game.jobfun.MainMenu*

```
package game.jobfun;

import game.event.Event;
import game.event.EventListener;
import game.gui.Button;
import game.gui.Container;
import game.gui.InputLabel;
import game.gui.Label;
import game.message.Message;
import game.scene.Scene;

import java.io.IOException;

public class MainMenu extends Scene implements EventListener {

	public static final int LOGIN_MESSAGE = 0, UPDATE_STATE = 1,
NEW_USER = 2;

	InputLabel nick, pass;
	Container body;

	public void load() {

		body = new Container(getWidth(), getHeight());
		add(body);
		body.focused = true;

		Label title = new Label("JobFUN!");
		add(title);
		title.alignToDisplay("HC");
		title.y = 15;

		Container menu = new Container(120, 90);
		menu.borderColor = 0xFFFFFF;
		body.add(menu);
		menu.alignToDisplay("HCVC");
		body.setFocused(menu);

		Label nicklbl = new Label("Insira login e senha");
		menu.add(nicklbl);
		nicklbl.alignToParent("HC");

		nick = new InputLabel(80);
		menu.add(nick);
		menu.setFocused(nick);
		nick.y += 5;

		pass = new InputLabel(80);
		menu.add(pass);
		pass.y += 5;

		Button btn = new Button("Enviar", 50, 20, new Runnable() {
			public void run() {
				enviar();
			}
		});
```

```
			menu.add(btn);
			btn.alignToParent("HC");
			btn.y += 5;
		}

		boolean sent = false;

		protected void enviar() {

			if (nick.getText().length() < 1 || pass.getText().length()
< 1) {
				showInfo("Preencha os campos!");
				return;
			}

			if (sent)
				return;

			sent = true;

			String msg = nick.getText() + "#" + pass.getText();

			try {
				if (!JFClient.isConnected())
					JFClient.connect();
				JFClient.sendMessage(new Message(LOGIN_MESSAGE, msg),
this);
			} catch (IOException e) {
				e.printStackTrace();
			}
		}

		public void unload() {

		}

		public void handleEvent(Event e) {
			switch (e.type) {
			case LOGIN_MESSAGE:
				Message m = (Message) e;
				String response = m.getString();
				if ("k".equals(response)) {
					new HouseScene().pushSelf();
				} else if ("i".equals(response)) {
					showInfo("Senha invalida, tente novamente.",
new Runnable() {
						public void run() {
							sent = false;
						}
					});
				} else if ("e".equals(response)) {
					showAsk("Deseja criar novo usuario?", "Sim",
new Runnable() {
						public void run() {
							try {
								if (!JFClient.isConnected())
									JFClient.connect();
								String msg = nick.getText() +
"#" + pass.getText();
```

```
                                        JFClient.sendMessage(new
Message(NEW_USER, msg),
                                                MainMenu.this);
                                    } catch (IOException e) {
                                        e.printStackTrace();
                                    }
                                }
                            }, "Não", new Runnable() {
                                public void run() {
                                    sent = false;
                                }
                            });
                        }
                        break;
                case NEW_USER:
                        m = (Message) e;
                        response = m.getString();
                        if ("k".equals(response)) {
                            new PickAJob().pushSelf();
                        } else if ("n".equals(response)) {
                            showInfo("Dados inválidos, tente novamente.");
                            sent = false;
                        }
                        break;
                }
        }

        public void showInfo(String text) {
                final Container info = new Container(getWidth() - 20,
getHeight() - 40);
                Label lbl = new Label(text);
                info.add(lbl);
                lbl.alignToParent("HCVC");
                lbl.y -= 10;

                Button ok = new Button("OK", 50, 20, new Runnable() {
                        public void run() {
                            remove(info);
                        }
                });

                info.add(ok);
                ok.y += 10;
                ok.alignToParent("HC");

                add(info);
                info.alignToDisplay("HCVC");
                info.setFocused(ok);
                info.focused = true;
                body.setFocused(info);
        }

        public void showInfo(String text, final Runnable r) {
                final Container info = new Container(getWidth() - 20,
getHeight() - 40);
                Label lbl = new Label(text);
                info.add(lbl);
                lbl.alignToParent("HCVC");
                lbl.y -= 10;

                Button ok = new Button("OK", 50, 20, new Runnable() {
```

```
                public void run() {
                        remove(info);
                        r.run();
                }
        });

        info.add(ok);
        ok.y += 10;
        ok.alignToParent("HC");

        add(info);
        info.alignToDisplay("HCVC");
        info.setFocused(ok);
    }

    public void showAsk(String text, String c1, final Runnable r1,
String c2,
                final Runnable r2) {
        final Container info = new Container(getWidth() - 20,
getHeight() - 40);
        Label lbl = new Label(text);
        info.add(lbl);
        lbl.alignToParent("HCVC");
        lbl.y -= 10;

        Button b1 = new Button(c1, 50, 20, new Runnable() {
                public void run() {
                        remove(info);
                        if (r1 != null)
                                r1.run();
                }
        });

        info.add(b1);
        b1.y += 10;
        b1.alignToParent("HC");

        Button b2 = new Button(c2, 50, 20, new Runnable() {
                public void run() {
                        remove(info);
                        if (r2 != null)
                                r2.run();
                }
        });

        info.add(b2);
        b2.y += 2;
        b2.alignToParent("HC");

        add(info);
        info.alignToDisplay("HCVC");
        info.setFocused(b1);
        body.setFocused(info);
    }

}
```

III.5.2 *game.jobfun.PickAJob*

```
package game.jobfun;
```

```
import game.gui.Button;
import game.gui.Container;
import game.util.Util;

public class PickAJob extends JobfunBaseScene {

	Container list;

	public void load() {
		title = "Escolha um emprego!";
		super.load();

		list = new Container(70, 40);

		Button marc = new Button("Marceneiro", 70, 20, new
Runnable() {
			public void run() {
				Util.saveRSString("JF-USERJOB", "m");
				System.out.println("Marceneiro.run()");
			}
		});

		Button jard = new Button("Florista", 70, 20, new Runnable()
{
			public void run() {
				Util.saveRSString("JF-USERJOB", "f");
				System.out.println("Florista.run()");
			}
		});

		list.add(marc);
		list.add(jard);
		add(list);
		list.alignToDisplay("VC");
		list.x += 10;
		list.setFocused(marc);

		makeMenu();
	}

	public void update() {
		super.update();
		list.focused = !menuAdded;
	}

}
```

III.5.3 *game.jobfun.HouseScene*

```
package game.jobfun;

import game.core.Color;
import game.event.Event;
import game.event.EventListener;
import game.gui.Button;
import game.gui.Container;
import game.gui.SimpleImage;
import game.message.Message;
import game.util.ImageUtil;
import game.util.StringUtil;
import game.util.Util;
```

```
import java.io.IOException;

import javax.microedition.lcdui.Image;

public class HouseScene extends JobfunBaseScene implements
EventListener {

	public static final int UPDATE_STATE = 1;

	private HouseObject[] hobjs = new HouseObject[5];

	public void load() {

		title = "Sua casa";

		super.load();

		Container room = new Container(getWidth() - 10, getHeight()
- 60);
		add(room);
		room.alignToDisplay("HCVC");

		Image chair = null, table = null, violet = null;

		chair = ImageUtil.loadImage("/chair.png", true);
		table = ImageUtil.loadImage("/table.png", true);
		violet = ImageUtil.loadImage("/violet.png", true);

		SimpleImage c = new SimpleImage(chair);
		SimpleImage t = new SimpleImage(table);
		SimpleImage v = new SimpleImage(violet);

		room.add(c);
		c.alignToParent("R");

		room.add(t);
		t.alignToParent("TR");
		t.y += 15;
		t.x -= 15;

		room.add(v);
		v.alignToParent("TR");
		v.y+=10;
		v.x-=55;

		Button buy = new Button("Comprar", 70, 20, new Runnable() {
			public void run() {
				new ComprarScene().pushSelf();
			}
		});
//		buy.color = Color.LIGHT_BLUE;

		Button make = new Button("Fazer", 70, 20, new Runnable() {
			public void run() {
				new MakeScene().pushSelf();
			}
		});
//		make.color = Color.LIGHT_BLUE;

		Button sell = new Button("Vender", 70, 20, new Runnable() {
```

```
                  public void run() {
                        new VenderScene().pushSelf();
                  }
            });
//          sell.color = Color.LIGHT_BLUE;

            soft1Menu.add(buy);
            soft1Menu.add(make);
            soft1Menu.add(sell);

            makeMenu();

            double cash = 50;
            String data = Util.getRSString("JF-CASH");
            if (data != null)
                  cash = Double.parseDouble(data);

            data = Util.getRSString("JF-HOBJS");
            if (data != null && data.length() > 0) {
                  //
itemname§type§itemx§itemy¥itemname§type§itemx§itemy
                  String[] itens = StringUtil.split(data, "¥");
                  String[] item;
                  HouseObject ho;
                  int type, itemx, itemy;

                  for (int i = 0; i < itens.length; i++) {
                        item = StringUtil.split(itens[i], "§");
                        type = Integer.parseInt(item[1]);
                        itemx = Integer.parseInt(item[2]);
                        itemy = Integer.parseInt(item[3]);

                        try {
                              ho = new
HouseObject(Image.createImage("item" + type
                                          + ".png"));
                              ho.x = itemx;
                              ho.y = itemy;
                              hobjs[i] = ho;
                        } catch (IOException e) {
                              e.printStackTrace();
                        }
                  }
            } else {
                  showInfo("Voce pode adicionar itens ao seu quarto na
opção itens do menu");
            }
      }

      public void handleEvent(Event e) {
            Message m = (Message) e;
            switch (m.type) {
            case UPDATE_STATE:
                  String data = m.getString();
                  // Formato da mensagem:
                  //
saldo#nome§tipo§estado§preco¥nome§tipo§estado§preco
                  if (data != null && data.length() > 0) {
                        String[] itens = StringUtil.split(data, "#");
                        double saldo = Double.parseDouble(itens[0]);
                        // Salva o saldo de creditos localmente
```

```
                        Util.saveRSString("JF-CREDITS", saldo + "");
                        // Caso haja itens à venda, também os salva
localmente
                        if (itens.length > 1 && itens[1].length() > 0)
                              Util.saveRSString("JF-SELL-ITENS",
itens[1]);
                  }
                  break;
            }
      }

}
```

III.5.3 *game.jobfun.ComprarScene*

```
package game.jobfun;

import game.gui.Button;
import game.gui.InputLabel;
import game.gui.Label;

public class ComprarScene extends JobfunBaseScene {

      public void load() {
            title = "Comprar";
            super.load();

            Label lbl = new Label("Pesquisar produtos:");
            add(lbl);
            lbl.x = 10;

            InputLabel ipt = new InputLabel(80);
            add(ipt);

            body.setFocused(ipt);

            Button sch = new Button("Enviar", 60, 20, new Runnable() {
                  public void run() {
                        new PesquisaCompra().pushSelf();
                  }
            });
            add(sch);
            sch.y += 5;
      }

}
```

III.5.4 *game.jobfun.VenderScene*

```
package game.jobfun;

import game.event.Event;
import game.event.EventListener;
import game.gui.Component;
import game.gui.Label;
import game.message.Message;
import game.util.StringUtil;
import game.util.Util;
```

```
public class VenderScene extends JobfunBaseScene implements
EventListener {

	public static final int UPDATE_STATE = 1;

	public void load() {
		title = "Vender";
		super.load();

		Label lbl = new Label("Seus produtos:");
		add(lbl);
		lbl.x = 10;
		lbl.y += 5;

		String data = Util.getRSString("JF-OBJS");
		if (data != null && data.length() > 0) {
			// nome§tipo§status§valor¥
			String[] objs = StringUtil.split(data, "¥");
			String[] obj;
			for (int i = 0; i < objs.length; i++) {
				obj = StringUtil.split(objs[i], "§");

				add(new ItemButton(new Produto(obj[0],
						Integer.parseInt(obj[1]),
Integer.parseInt(obj[2]),
						Double.parseDouble(obj[3]))));
			}

		}

		Component c = new ItemButton(new Produto("Cadeira", 0, 0,
0));
		add(c);
		body.setFocused(c);
		add(new ItemButton(new Produto("Cadeira", 0, 1, 18.50)));
		add(new ItemButton(new Produto("Mesa", 0, 2, 50)));
		add(new ItemButton(new Produto("Cadeira", 0, 1, 13.50)));

		makeMenu();

	}

	public void handleEvent(Event e) {
		Message m = (Message) e;
		switch (m.type) {
		case UPDATE_STATE:

			break;
		}
	}

}
```

III.5.5 *game.jobfun.MakeScene*

```
package game.jobfun;

import game.gui.Button;
import game.gui.Container;
import game.gui.SimpleImage;
import game.scene.Game;
```

```
import game.util.Util;

import java.io.IOException;

import javax.microedition.lcdui.Image;

public class MakeScene extends JobfunBaseScene {

	Image chair, table;
	private Container list;
	private static int objectType = -1;

	public void load() {
		title = " Selecione um objeto para criar";
		super.load();
		makeMenu();

		list = new Container(70, 40);

		String objs[] = new String[2];
		String job = Util.getRSString("JF-USERJOB");
		job = "f";
		if ("m".equals(job)) {
			objs[0] = "Cadeira";
			objs[1] = "Mesa";
		} else if ("f".equals(job)) {
			objs[0] = "Rosas";
			objs[1] = "Violetas";
		}

		try {
			chair = Image.createImage("/chair.png");
			table = Image.createImage("/violet.png");
		} catch (IOException e) {
			e.printStackTrace();
		}

		final SimpleImage img = new SimpleImage(chair);
		Button obj1 = new Button(objs[0], 70, 20, new Runnable() {
			public void run() {
				Game.getGameInstance().pushScene(new
Lumbering());
				objectType = 0;
			}
		}) {
			public void update() {
				if (focused)
				{
					img.setImage(chair);
					img.align(list, "R");
					img.alignToDisplay("VC");
					img.x += 5;
				}
				super.update();
			}
		};

		Button ojb2 = new Button(objs[1], 70, 20, new Runnable() {
			public void run() {
				Game.getGameInstance().pushScene(new
Lumbering());
```

```
                        objectType = 1;
                }
        }) {
                public void update() {
                        if (focused)
                        {
                                img.setImage(table);
                                img.align(list, "R");
                                img.alignToDisplay("VC");
                                img.x += 10;
                        }
                        super.update();
                }
        };

        list.add(obj1);
        list.add(ojb2);

        add(list);
        list.alignToDisplay("VC");

        add(img);
        img.align(list, "R");
        img.alignToDisplay("VC");
        img.x += 5;

        list.setFocused(obj1);
    }

    static void objectDone() {
        switch (objectType) {

        }
    }

    public void update() {
        super.update();
        list.focused = !menuAdded;
    }

}
```

III.5.6 *game.jobfun.Lumbering*

```
package game.jobfun;

import game.core.Key;
import game.gui.Button;
import game.gui.Container;
import game.gui.Label;
import game.gui.SimpleImage;
import game.scene.Game;
import game.util.ImageUtil;
import game.util.SpriteLoader;

import javax.microedition.lcdui.Image;

public class Lumbering extends JobfunBaseScene {

    private SimpleImage serrote, corte;
    private SpriteLoader cortes;
```

```
private boolean playing = true;
private int initY, tamCorte = 0, corteAtual = 0;

public void load() {
	title = "Fazendo cadeira!";
	super.load();

	Image m = ImageUtil.loadImage("madeira.png", false);
	SimpleImage madeira = new SimpleImage(m);
	add(madeira);
	madeira.alignToDisplay("HCVC");

	Image c = ImageUtil.loadImage("cortes.png", true);
	cortes = new SpriteLoader(c, 25, 39, true);

	corte = new SimpleImage(cortes.getImage(0));
	add(corte);
	corte.alignToDisplay("HCVC");
	corte.x += 5;

	Image s = ImageUtil.loadImage("serrote.png", true);
	serrote = new SimpleImage(s);
	add(serrote);
	serrote.alignToDisplay("HCVC");
	serrote.x -= 20;
	serrote.y += 10;
	initY = serrote.y;
}

public void update() {
	super.update();

	if (playing) {
		if (Key.pressed(Key.K_UP, 100)) {
			serrote.x += 15;
			serrote.y -= 30;
			if (initY - serrote.y > 20)
				tamCorte++;
		} else if (Key.pressed(Key.K_DOWN, 100)) {
			serrote.x -= 15;
			serrote.y += 30;
			if (serrote.y - initY > 20)
				tamCorte++;
		}

		if (tamCorte > 2) {
			corteAtual++;
			if (corteAtual > 4) {
				win();
				playing = false;
			} else {

corte.setImage(cortes.getImage(corteAtual));
			}
			tamCorte = 0;
		}
	}
}

public void win() {
	Container c = new Container(100, 50);
```

```
            Label lbl = new Label("OK!");
            Button btn = new Button("Continuar", 70, 20, new Runnable()
{
                  public void run() {
                        Game.getGameInstance().setScene(new
Hammering());
                  }
            });

            c.add(lbl);
            lbl.alignToParent("HC");
            lbl.y += 5;
            c.add(btn);
            btn.alignToParent("HC");

            add(c);
            c.alignToDisplay("HCVC");
            c.setFocused(btn);
      }
}
```

III.5.7 *game.jobfun.Hammering*

```
package game.jobfun;

import game.core.Key;
import game.gui.Button;
import game.gui.Container;
import game.gui.Label;
import game.gui.SimpleImage;
import game.scene.Game;
import game.util.ImageUtil;
import game.util.SpriteLoader;

import javax.microedition.lcdui.Image;

public class Hammering extends JobfunBaseScene {

      SpriteLoader pregos;
      SimpleImage prego, martelo;
      int initX, initY, accX = 2, accY = 1;
      int currentPrego = 0;
      Image m1, m2;
      boolean ism1 = true, ended = false;
      long hitTime;

      public void load() {
            title = "Fazendo cadeira!";
            super.load();

            Image m = ImageUtil.loadImage("madeira.png", false);
            SimpleImage madeira = new SimpleImage(m);
            add(madeira);
            madeira.alignToDisplay("HCB");
            madeira.y -= 5;

            Image c = ImageUtil.loadImage("pregos.png", true);
            pregos = new SpriteLoader(c, 14, 39, true);

            prego = new SimpleImage(pregos.getImage(currentPrego));
            add(prego);
```

```
		prego.alignToDisplay("HCB");
		prego.x -= 10;
		prego.y -= 35;

		m1 = ImageUtil.loadImage("martelo1.png", true);
		m2 = ImageUtil.loadImage("martelo2.png", true);
		martelo = new SimpleImage(m1);
		add(martelo);
		martelo.alignToDisplay("HCVC");
		martelo.x += 30;
		martelo.y -= 20;
		initX = martelo.x;
		initY = martelo.y;
	}

	public void update() {
		super.update();

		if (ended || System.currentTimeMillis() - hitTime < 400)
			return;

		else if (Key.pressed(Key.K_SELECT, 200)) {
			if (ism1) {
				martelo.setImage(m2);
				martelo.y += 45;
				ism1 = false;

				if (martelo.x <= prego.x && prego.x < martelo.x
+ 38) {
					currentPrego++;
					if (currentPrego == 4) {

	prego.setImage(pregos.getImage(currentPrego));
						win();
					} else

	prego.setImage(pregos.getImage(currentPrego));
				}
			}
			hitTime = System.currentTimeMillis();
		} else {
			if (!ism1) {
				martelo.setImage(m1);
				ism1 = true;
				martelo.y -= 45;
			}

			if (Math.abs(martelo.x - initX) >= 40)
				accX *= -1;

			if (Math.abs(martelo.y - initY) >= 5)
				accY *= -1;

			martelo.x += accX;
			martelo.y += accY;
		}
	}

	public void win() {
		ended = true;
		Container c = new Container(100, 50);
```

```
            Label lbl = new Label("Cadeira criada!");
            Button btn = new Button("Continuar", 70, 20, new Runnable()
{
                  public void run() {
                        Game.getGameInstance().popScene();
                        MakeScene.objectDone();
                        //adiciona novo objeto e sync com server
                  }
            });

            c.add(lbl);
            lbl.alignToParent("HC");
            lbl.y += 5;
            c.add(btn);
            btn.alignToParent("HC");

            add(c);
            c.alignToDisplay("HCVC");
            c.setFocused(btn);
      }
}
```

III.5.7 *game.jobfun.Digging*

```
package game.jobfun;

import game.core.Key;
import game.gui.Button;
import game.gui.Container;
import game.gui.Label;
import game.gui.SimpleImage;
import game.scene.Game;
import game.util.ImageUtil;
import game.util.SpriteLoader;

import javax.microedition.lcdui.Image;

public class Digging extends JobfunBaseScene {

      SpriteLoader buracos, pas;
      SimpleImage buraco, pa;
      int initX, initY, accX = 2, accY = 1;
      int currentBuraco = 0, cavada = 0;
      boolean pa0 = true, ended = false;
      long hitTime;

      public void load() {
            title = "Fazendo violetas!";
            super.load();

            Image m = ImageUtil.loadImage("terra.png", false);
            SimpleImage terra = new SimpleImage(m);
            add(terra);
            terra.alignToDisplay("HCB");

            Image c = ImageUtil.loadImage("buracos.png", true);
            buracos = new SpriteLoader(c, 79, 47, true);

            buraco = new SimpleImage(buracos.getImage(currentBuraco));
            add(buraco);
            buraco.alignToDisplay("HCB");
```

```
		buraco.y -= 15;

		pas = new SpriteLoader("pa.png", 65, 122, true);

		pa = new SimpleImage(pas.getImage(0));
		add(pa);
		pa.alignToDisplay("HCVC");
		pa.x += 20;
		pa.y -= 25;
		initX = pa.x;
		initY = pa.y;
	}

	public void update() {
		super.update();

		if (ended || System.currentTimeMillis() - hitTime < 400)
			return;

		else if (Key.pressed(Key.K_DOWN, 200)) {
			if (pa0) {
				pa.setImage(pas.getImage(1));
				pa.y += 25;
				pa.x -= 20;
				pa0 = false;
				cavada++;
				if (cavada == 4) {
					cavada = 0;
					currentBuraco++;
					if (currentBuraco == 5)
						win();
					else

	buraco.setImage(buracos.getImage(currentBuraco));
					}

				}
				hitTime = System.currentTimeMillis();
			} else {
				if (!pa0) {
					pa.setImage(pas.getImage(0));
					pa0 = true;
					pa.y -= 25;
					pa.x += 20;
				}
			}
	}

	public void win() {
		ended = true;
		Container c = new Container(100, 50);
		Label lbl = new Label("OK!");
		Button btn = new Button("Continuar", 70, 20, new Runnable()
{
			public void run() {
				Game.getGameInstance().popScene();
				MakeScene.objectDone();
				// adiciona novo objeto e sync com server
			}
		});
```

```
            c.add(lbl);
            lbl.alignToParent("HC");
            lbl.y += 5;
            c.add(btn);
            btn.alignToParent("HC");

            add(c);
            c.alignToDisplay("HCVC");
            c.setFocused(btn);
      }
}
```

III.5.8 *game.jobfun.PesquisaCompra*

```
package game.jobfun;

import game.core.Key;

public class PesquisaCompra extends JobfunBaseScene {

      public void load() {
            title = "Resultado da pesquisa";
            super.load();

            ItemButton ib;
            add(ib = new ItemButton(new Produto("Rosas", 0, 1, 20)));
            ib.x = 10;
            ib.y += 5;
            add(new ItemButton(new Produto("Rosas", 0, 1, 29.90)));
            add(new ItemButton(new Produto("Rosas", 0, 1, 25.80)));

            body.setFocused(ib);

            makeMenu();
      }

      public void update() {
            if (Key.pressed(Key.K_SELECT, 100)) {
                  if (body.getFocused() instanceof ItemButton) {
                        ItemButton b = (ItemButton) body.getFocused();
                        new DetalhesCompra(b.p).pushSelf();
                  }
            } else
                  super.update();
      }
}
```

III.5.9 *game.jobfun.DetalhesCompra*

```
package game.jobfun;

import game.gui.Button;
import game.gui.Label;

public class DetalhesCompra extends JobfunBaseScene {

      Produto p;
      Button state;

      public DetalhesCompra(Produto p) {
```

```
            this.p = p;
      }

      public void load() {
            title = "Comprar produto";
            super.load();

            Label lbl = new Label("Nome: " + p.nome);
            add(lbl);
            lbl.x = 10;
            lbl.y += 5;

            Label lbl3 = new Label("Preço: ");
            add(lbl3);
            lbl3.x = 10;
            lbl3.y += 5;

            Label ipt = new Label("$" + p.valor);
            add(ipt);
            ipt.align(lbl3, "VCR");
            ipt.x += 5;

            Button ok = new Button("Comprar", 60, 15, new Runnable() {
                  public void run() {
                        popSelf();
                  }
            });

            Button cancel = new Button("Cancelar", 60, 15, new
Runnable() {
                  public void run() {
                        popSelf();
                  }
            });

            add(ok);
            ok.y += 10;
            ok.x = 10;

            add(cancel);
            cancel.align(ok, "VCR");
            cancel.x += 5;

            body.setFocused(ok);

      }
}
```

III.5.9 *game.jobfun.JobFunBaseScene*

```
package game.jobfun;

import game.core.Key;
import game.gui.Button;
import game.gui.Component;
import game.gui.Container;
import game.gui.Label;
import game.scene.Game;
import game.scene.GameObject;
import game.scene.Scene;
```

```
public class JobfunBaseScene extends Scene {

	protected Container soft1Menu;
	protected boolean menuAdded = false;
	protected String title = "";
	protected Container body;

	public void load() {
		body = new Container(getWidth(), getHeight()) {
			public void update() {
				super.update();
			}
		};
		super.add(body);

		Label lbltitle = new Label(title);
		add(lbltitle);
		lbltitle.alignToDisplay("HC");
		lbltitle.y += 10;

		soft1Menu = new Container(70, 0);
		soft1Menu.x += 10;
	}

	public void add(GameObject object) {
		if (object instanceof Component)
			body.add((Component) object);
		super.add(object);
	}

	public void remove(GameObject object) {
		if (object instanceof Component)
			body.remove((Component) object);
		super.remove(object);
	}

	protected void makeMenu() {
		Button back = new Button("Voltar", 70, 20, new Runnable() {
			public void run() {
				Game.getGameInstance().popScene();
			}
		});
		// back.color = Color.LIGHT_BLUE;
		soft1Menu.add(back);
	}

	public void unload() {
		remove(soft1Menu);
		menuAdded = false;
	}

	public void update() {
		super.update();
		if (Key.clicked(Key.K_SOFT1, 150)) {
			if (menuAdded) {
				remove(soft1Menu);
				menuAdded = false;
			} else {
				add(soft1Menu);
				menuAdded = true;
				soft1Menu.setFocused(0);
```

```
                        soft1Menu.height = soft1Menu.size() * 20;
                        soft1Menu.alignToDisplay("B");
                  }
            }
      }

      public void showInfo(String text) {
            final Container info = new Container(getWidth() - 20,
getHeight() - 40);
            Label lbl = new Label(text);
            info.add(lbl);
            lbl.alignToParent("HCVC");
            lbl.y -= 10;

            Button ok = new Button("OK", 50, 20, new Runnable() {
                  public void run() {
                        body.remove(info);
                  }
            });

            info.add(ok);
            ok.y += 10;
            ok.alignToParent("HC");

            body.add(info);
            info.alignToDisplay("HCVC");
            info.setFocused(ok);
            body.setFocused(info);
      }

      public void showInfo(String text, final Runnable r) {
            final Container info = new Container(getWidth() - 20,
getHeight() - 40);
            Label lbl = new Label(text);
            info.add(lbl);
            lbl.alignToParent("HCVC");
            lbl.y -= 10;

            Button ok = new Button("OK", 50, 20, new Runnable() {
                  public void run() {
                        body.remove(info);
                        r.run();
                  }
            });

            info.add(ok);
            ok.y += 10;
            ok.alignToParent("HC");

            body.add(info);
            info.alignToDisplay("HCVC");
            info.setFocused(ok);
      }

      public void showAsk(String text, String c1, final Runnable r1,
String c2,
                  final Runnable r2) {
            final Container info = new Container(getWidth() - 20,
getHeight() - 40);
            Label lbl = new Label(text);
            info.add(lbl);
```

```
            lbl.alignToParent("HCVC");
            lbl.y -= 10;

            Button b1 = new Button(c1, 50, 20, new Runnable() {
                  public void run() {
                        body.remove(info);
                        if (r1 != null)
                              r1.run();
                  }
            });

            info.add(b1);
            b1.y += 10;
            b1.alignToParent("HC");

            Button b2 = new Button(c2, 50, 20, new Runnable() {
                  public void run() {
                        body.remove(info);
                        if (r2 != null)
                              r2.run();
                  }
            });

            info.add(b2);
            b2.y += 2;
            b2.alignToParent("HC");

            body.add(info);
            info.alignToDisplay("HCVC");
            info.setFocused(b1);
      }

}
```

III.5.10 *game.jobfun.JFClient*

```
package game.jobfun;

import game.event.EventListener;
import game.message.Message;
import game.message.MessageClient;

import java.io.IOException;

public class JFClient {

      // Instância da classe MessageClient
      private static final MessageClient client = new MessageClient();

      private JFClient() {
      }

      // Método estático para conexão
      public static boolean isConnected(){
            return client.isConnected();
      }

      // Método estático para conexão
      public static void connect() throws IOException {
            client.connect("127.0.0.1:3692");
      }
```

```
	// Métodos estáticos para envio de mensagens
	public static void sendMessage(Message m, EventListener
listener)
			throws IOException {
		client.sendMessage(m, listener);
	}

	public static void sendMessage(Message m) throws IOException {
		client.sendMessage(m);
	}

}
```

III.5.11 *Game.Jobfun.JFServer*

```
using System;
using System.Collections.Generic;
using System.Text;
using Game.Net.Server;
using System.Net.Sockets;
using System.Data;

namespace Game.Jobfun
{
    class JFServer : SocketServer, ConnectionListener
    {
        // Tipo de evento para o login
        public const int LOGIN_MESSAGE = 0, UPDATE_STATE = 1, NEW_USER
= 2, SEARCH = 3, ADD_FOR_SALE = 4, BUY = 5;

        MySQLInterface msql;

        public JFServer()
            : base(3692)
        {
            msql = new MySQLInterface();
            msql.Connect("localhost", "jobfun", "jobfun", "jobfun");
        }

        protected override void ConnectionAccepted(Socket conn)
        {
            new SocketConnection("sc",
conn).AddConnectionListener(this);
        }

        // Trata a mensagem de login recebida
        public void MessageReceived(Message msg, Connection c)
        {
            switch (msg.Type)
            {
                case LOGIN_MESSAGE:
                    validateUser(msg, c);
                    break;
                case UPDATE_STATE:
                    UpdateUser(c);
                    break;
                case NEW_USER:
                    newUser(msg, c);
                    break;
            }
```

```
        }

        private void UpdateUser(Connection c)
        {
            double cash = 0;
            String query = "SELECT `cash` FROM `Users` WHERE
`name`=\"" + c.GetID() + "\"";
            DataRowCollection ds = msql.Select(query, "Users");
            if (ds != null && ds.Count > 0)
            {
                cash = (Double)ds[0].ItemArray[0];
            }

            query = "SELECT * FROM `Itens` WHERE `username`=\"" +
c.GetID() + "\"";
            ds = msql.Select(query, "Itens");

            String itens = "";
            if (ds != null && ds.Count > 0)
            {
                object[] columns;
                for (int i = 0; i < ds.Count; i++)
                {
                    columns = ds[i].ItemArray;
                    String itemname = (String)columns[2];
                    int itemtype = (Int32)columns[3];
                    int status = (Int32)columns[4];
                    double price = (Double)columns[5];

                    if (i > 0)
                        itens += "¥";
                    itens += itemname + "§" + itemtype + "§" + status
+ "§" + price;
                }
            }
            String response = cash + "#" + itens;
            c.SendMessage(new Message(UPDATE_STATE,
Util.GetBytes(response)));
        }

        private void validateUser(Message msg, Connection c)
        {
            String data = Util.GetString(msg.GetData());
            int spt = data.IndexOf('#');
            String username = data.Substring(0, spt);
            String pass = data.Substring(spt + 1);

            String query = "SELECT `pass` FROM `Users` WHERE
`name`=\"" + username + "\"";

            DataRowCollection ds = msql.Select(query, "Users");

            if (ds != null && ds.Count > 0)
            {
                String userpass = (String)ds[0].ItemArray[0];
                if (pass.Equals(userpass))
                {
                    Console.WriteLine("VALID PASS");
                    c.SendMessage(new Message(LOGIN_MESSAGE,
Util.GetBytes("success")));
                    c.SetID(username);
```

```
                }
                else
                {
                    Console.WriteLine("INVALID PASS");
                    c.SendMessage(new Message(LOGIN_MESSAGE,
Util.GetBytes("pass_error")));
                }
            }
            else
            {
                Console.WriteLine("USER DOESNT EXIST");
                c.SendMessage(new Message(LOGIN_MESSAGE,
Util.GetBytes("user_error")));
            }
        }

        private void newUser(Message msg, Connection c)
        {
            String data = Util.GetString(msg.GetData());
            int spt = data.IndexOf('#');
            String username = data.Substring(0, spt);
            data = data.Substring(spt + 1);
            spt = data.IndexOf('#');
            String pass = data.Substring(0, spt);
            String userJob = data.Substring(spt + 1);

            String query = "INSERT INTO `Users` (`name`, `pass`,
`cash`, `job`) " +
                           "VALUES ('" + username + "', '" + pass +
"', '50', '" + userJob + "');";

            msql.Query(query);
        }

        private void search(Message m, Connection c)
        {
            String keyword = Util.GetString(m.GetData());

            // Faz a pesquisa no banco de dados
            String query = "SELECT * FROM `Itens` WHERE `name` LIKE
'%" + c.GetID() + "%'";
            DataRowCollection ds = msql.Select(query, "Users");

            String itens = "";
            if (ds != null && ds.Count > 0)
            {
                object[] columns;
                for (int i = 0; i < ds.Count; i++)
                {
                    columns = ds[i].ItemArray;
                    int sellerId = (int)columns[1];
                    String itemname = (String)columns[2];
                    int itemtype = (int)columns[3];
                    int status = (int)columns[4];
                    double price = (int)columns[5];

                    if (i > 0)
                        itens += "¥";
                    itens += sellerId + "§" + itemname + "§" +
itemtype + "§" + status + "§" + price;
                }
```

```
            }
            c.SendMessage(new Message(SEARCH, Util.GetBytes(itens)));
        }

        private void AddForSale(Message m, Connection c)
        {
            String msg = Util.GetString(m.GetData());
            String[] item = msg.Split('$');

            // Recupera o id do usuario
            String query = "SELECT `userId` FROM `Users` WHERE
`name`=\"" + c.GetID() + "\"";
            DataRowCollection ds = msql.Select(query, "user");
            int userid = -1;
            if (ds != null && ds.Count > 0)
                userid = (int)ds[0].ItemArray[0];

            // Insere os dados no banco
            query = "INSERT INTO `Itens` (`userId`, `itemName`,
`type`, `status`, `price`) " +
                    "VALUES (" + userid + ", '" + item[0] + "', '" +
item[1] + "', '" +
                    item[2] + "', '" + item[3] + "')";

            msql.Query(query);

            // Responde o cliente
            c.SendMessage(new Message(ADD_FOR_SALE,
Util.GetBytes("k")));
        }

        private void Buy(Message m, Connection c)
        {
            String msg = Util.GetString(m.GetData());
            String[] item = msg.Split('$');

            int itemId = Int32.Parse(item[0]);
            double price = Double.Parse(item[5]);
            int sellerId = Int32.Parse(item[1]);

            int buyerid = -1;
            double buyerCash = -1;

            // Recupera o id do comprador
            String query = "SELECT `userId`, `cash` FROM `Users` WHERE
`name`=\"" + c.GetID() + "\"";
            DataRowCollection ds = msql.Select(query, "user");

            if (ds != null && ds.Count > 0)
            {
                buyerid = (int)ds[0].ItemArray[0];
                buyerCash = (double)ds[0].ItemArray[1];
            }
            if (buyerid >= 0)
            {
                if (buyerCash > price)
                {
                    // Credita o valor no saldo do vendedor
                    query = "UPDATE `Users` SET `cash` = (`cash`+" +
                        price + ") WHERE `Users`.`user_id` = " +
sellerId;
```

```
                    msql.Query(query);

                    // Devita o valor no saldo do comprador
                    query = "UPDATE `Users` SET `cash` = (`cash`-" +
                        price + ") WHERE `Users`.`user_id` = " +
buyerid;
                    msql.Query(query);

                    // Altera o status do item para vendido
                    query = "UPDATE `Itens` SET `status` = '2' WHERE
`Itens`.`item_id` = " + itemId;
                    msql.Query(query);

                    // Notifica o cliente do sucesso da compra
                    c.SendMessage(new Message(BUY,
Util.GetBytes("success")));
                }
                else
                {
                    // Notifica o cliente de que nao possui saldo
suficiente
                    c.SendMessage(new Message(BUY,
Util.GetBytes("nocash")));
                }
            }
            else
            {
                //Notifica o cliente de que houve problema na
transacao
                c.SendMessage(new Message(BUY,
Util.GetBytes("failed")));
            }
        }
    }
}
```