UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

INTEGRAÇÃO DE TÉCNICAS DE SISTEMAS DISTRIBUÍDOS APLICADA AO DESENVOLVIMENTO DE UM AMBIENTE PARA O JOGO DE XADREZ

ANTONIO CARLOS BAMBINO FILHO

BLUMENAU
2011

2011/2-03

ANTONIO CARLOS BAMBINO FILHO

INTEGRAÇÃO DE TÉCNICAS DE SISTEMAS DISTRIBUÍDOS APLICADA AO DESENVOLVIMENTO DE UM AMBIENTE PARA O JOGO DE XADREZ.

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciência da Computação — Bacharelado.

Prof. Marcel Hugo, Mestre - Orientador

BLUMENAU
2011

2011/2-03

AMBIENTE PARA O JOGO DE XADREZ UTILIZANDO TÉCNICAS DE SISTEMAS DISTRIBUÍDOS ATRAVÉS DE WEBSERVICES E DO FRAMEWORK JAVA WEB START

Por

ANTONIO CARLOS BAMBINO FILHO

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

	_____
Presidente:	Prof. Marcel Hugo, Mestre – Orientador, FURB
	_____
Membro:	Prof. Jacques Robert Heckmann, Mestre – FURB
	_____
Membro:	Prof. Paulo Fernando da Silva, Mestre – FURB

Blumenau, 14 de dezembro de 2011

Dedico este trabalho a todos que contribuíram para que ele fosse feito e para todos que contribuirão numa possível continuação.

AGRADECIMENTOS

A Deus, por ter mais na vida do que mereço.

À minha família, em especial ao meu pai, por todo o apoio durante a minha formação.

À minha namorada, pela paciência, apoio e compreensão.

Ao meu orientador, professor Marcel Hugo, pela confiança.

Ao amigo, Josimar Cuchi, por compreender que o TCC poderia tomar algum tempo do meu expediente.

Seu tempo é limitado. Por isso, não perca tempo em viver a vida de outra pessoa.

Steve Jobs

RESUMO

Este trabalho descreve o desenvolvimento de um ambiente para o jogo de xadrez e alguns jogos derivados deste, como o xadrez mata-mata e randômico. Para desenvolver a lógica de gerenciamento das partidas foi desenvolvido um *webservice* na linguagem Java, mantido por um servidor *web GlassFish* e por um banco de dados *MySQL*. Para as telas, foi utilizado o Java Web Start, que faz com que o ambiente seja visualmente parecido ao de um sistema *desktop*. Os algoritmos de controle de peças foram todos desenvolvidos em Java para uma melhor integração com as demais partes do sistema.

Palavras-chave: *Webservices*. Java Web Start. *Threads*. Xadrez.

ABSTRACT

This work describes the development of chess game environment and chess variations, as looser and random and chess. To develop the logic and the management of the games, it was developed a Java webservice, maintained by a GlassFish webserver and a MySQL database. For the user interface, it was used the Java Web Start, which makes the application's appearance similar to a desktop system. The control algorithms were all developed in Java for a better integration with the other parts of the system.

Key-words: *Webservices*. Java Web Start. *Threads*. Chess.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

API - *Application Programming Interface*

EA - *Enterprise Architect*

HTML - *HyperText Markup Language*

HTTP - *Hypertext Transfer Protocol*

ICC – *Internet Chess Club*

IDE - *Integrated Development Environment*

IP – *Internet Protocol*

IXC – Internet Xadrez Clube

JAX - *Java API for XML*

JEE - Java *Enterprise Edition*

JNLP - *Java Network Launching Protocol*

JSF - *Java Server Faces*

JSP - *Java Server Pages*

PGN - *Portable Game Notation*

RIA - *Rich Internet Application*

RMI - *Remote Method Invocation*

RPC - *Remote Procedure Call*

SE - *Standard Edition*

SOA – *Service-Oriented Architecture*.

SOAP - *Simple Object Access Protocol*

UML - *Unified Modeling Language*

URL - *Uniform Resource Locator*

WSDL - *Web Service Description Language*

XML - *eXtensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO

Segundo Albuquerque (2006), o Java é uma linguagem que facilita o desenvolvimento de aplicações distribuídas por ser portável e independente de plataforma.

Este projeto visa desenvolver um jogo dinâmico de xadrez via *web* que tenha as características de uma aplicação *desktop*. Para isso, será utilizado o *framework Java Web Start*, que é executado remotamente e possibilita o uso dos recursos da linguagem Java, caracterizando uma *Rich Internet Applications* (RIA). As RIAs vêm ganhando cada vez mais espaço no mercado, pois sistemas que rodam na *web* e tenham uma performance de um sistema desktop são mais performáticos (ADOBE, 2009).

Neste sistema, qualquer jogador poderá se conectar ao *site*, efetuar um cadastro e entrar na sala de jogo, podendo desafiar qualquer jogador que também estiver conectado para uma partida. O jogo será exclusivamente de humanos x humanos.

Para fazer a atualização das telas de jogo, serão utilizados *threads* que trabalharão concorrentemente em um processo de *game-loop*. Neste processo, o cliente requisita todas as alterações ao servidor e atualiza as telas de acordo com as informações obtidas.

O sistema contará também com um banco de dados *MySQL*, que guardará todas as informações de cadastro e a pontuação acumulada de cada jogador.

Para persistir os dados no banco e gerenciar a lógica das partidas, será desenvolvido um *webservice* Java, utilizando a *Application Programming Interface* (API) *Java API for XML* (JAX). Segundo Pamplona (2010), os *webservices* fazem a parte não-visual, levando os dados obtidos através das *interfaces* gráficas do cliente para o servidor de aplicação, em formato XML.

Para manter a aplicação rodando e manter todos os usuários conectados, será utilizado um servidor *GlassFish*. Através do *webservice*, os dados serão levados da *interface* gráfica para o *GlassFish* e persistidos no banco de dados.

1.1 OBJETIVOS DO TRABALHO

O objetivo geral deste trabalho é integrar diferentes técnicas de sistemas distribuídos desenvolvendo um ambiente de jogo via *web* para que pessoas de todos os níveis de

conhecimento de jogo possam jogar xadrez.

Os objetivos específicos do trabalho são:

a) gerenciar os jogos com um *webservice* Java, utilizando a API JAX;
b) utilizar o servidor *GlassFish* para manter as partidas e os usuários conectados;
c) integrar a parte visual com a parte lógica, especificada no *webservice* e no servidor, com uso do *Java Web Start*.

1.2 ESTRUTURA DO TRABALHO

Este trabalho divide-se em fundamentação teórica, desenvolvimento do *software* e conclusões.

No capítulo referente à fundamentação teórica está descrito o jogo de xadrez clássico, randômico e mata-mata. Também está descrito como funcionam os *webservices* e a biblioteca JAX, que será usada para trabalhar com eles. Em seguida, fala-se sobre o funcionamento do *Java Web Start* e dos servidores *web*, com destaque para o *GlassFish*, que é o servidor utilizado no desenvolvimento do *software*. Por fim, fala-se sobre *threads* e sobre três trabalhos correlatos.

No capítulo referente ao desenvolvimento do software, é apresentada a metodologia de desenvolvimento e os algoritmos criados, através de diagramas.

Por fim, a conclusão retrata a opinião do autor sobre o assunto e sugere continuações futuras para este trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

Este tópico foi dividido em seis seções para facilitar o entendimento do assunto. Primeiro apresenta-se as regras do jogo de xadrez e dos seus derivados que serão implementados: xadrez randômico e xadrez mata-mata, além da notação *Portable Game Notation* (PGN). Em seguida explica-se as tecnologias a serem utilizadas e os trabalhos correlatos a este.

2.1 XADREZ

Segundo D'Agostini (1998, p. 17-19), o jogo de xadrez é composto por um tabuleiro de 64 casas, sendo estas negras e brancas de forma alternada. É jogado por dois jogadores, sendo que um conduz 16 peças brancas e o outro conduz 16 peças negras, dispostos simetricamente em cada lado do tabuleiro. Estas 16 peças são: um rei, uma dama, duas torres, dois cavalos, dois bispos e oito peões (Figura 1).

Figura 1- Posição inicial de um tabuleiro de xadrez

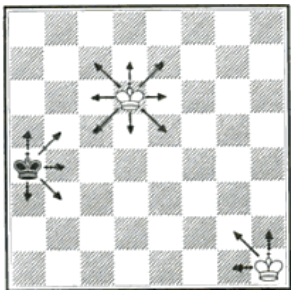
Cada jogador efetua um lance após o outro e o objetivo final é dar xeque-mate, que significa atacar o rei adversário com qualquer peça e este não ter nenhum lance legal para fazer (D'AGOSTINI, 1998, p.23).

2.1.1 Movimentação das peças

Segundo Becker (2002, p. 14), toda peça pode ser jogada em uma casa vazia ou ocupada por uma peça adversária, capturando-a. Nenhuma peça pode mover-se para uma casa

onde já esteja uma peça da mesma cor e nenhuma peça pode pular outras peças com exceção do cavalo.

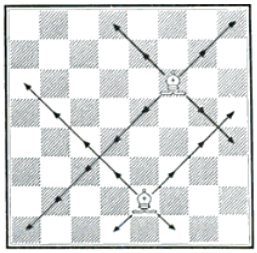
O rei pode mover-se uma casa para qualquer direção, em sentido horizontal, vertical ou diagonal (BECKER, 2002, p. 14). As setas da Figura 2 indicam as possíveis movimentações do rei.



Fonte: Becker (2002, p. 14).

Figura 2 - Movimentação do rei

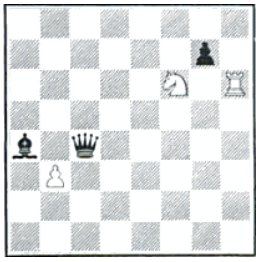
O bispo pode mover-se em sentido diagonal apenas, em qualquer direção (Figura 3).



Fonte: Becker (2002, p. 15).

Figura 3 - Movimentação do bispo

O peão é a única peça que não pode retroceder (BECKER, 2002, p. 17). Em seu primeiro movimento, ele pode ir duas casas pra frente. Depois, movimenta-se apenas uma casa. Para efetuar uma captura o peão possui uma exceção, já que a faz apenas na diagonal (BECKER, 2002, p. 25). Na Figura 4, o peão branco pode capturar tanto o bispo quanto a dama negra. O mesmo vale para o peão negro, que pode capturar a torre ou o cavalo branco.



Fonte: Becker (2002, p. 26).

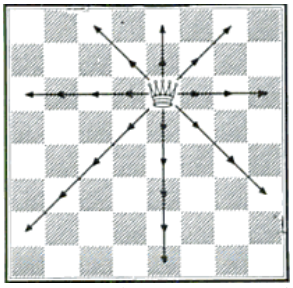
Figura 4 - Movimentação de captura do peão

Quando chega à última casa, o peão pode ser trocado por qualquer outra peça, com exceção do rei.

A torre tem uma movimentação semelhante ao bispo, com a diferença e mover-se na horizontal e na vertical em vez das diagonais.

A dama combina os movimentos da torre e do bispo, podendo mover-se na diagonal,

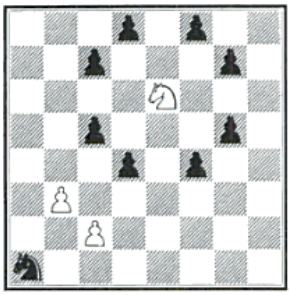
horizontal e vertical, nos dois sentidos (Figura 5).



Fonte: Becker (2002, p. 15).

Figura 5 - Movimentação da dama

O cavalo é a única peça que pode pular as outras no seu caminho (BECKER, 2002, p. 17). Ele pode mover-se uma casa em diagonal e mais uma em frente, formando um "L". Na Figura 6, o cavalo branco pode capturar qualquer um dos oito peões negros, enquanto o cavalo negro pode capturar um dos dois peões brancos, já que estão no seu raio de ação.

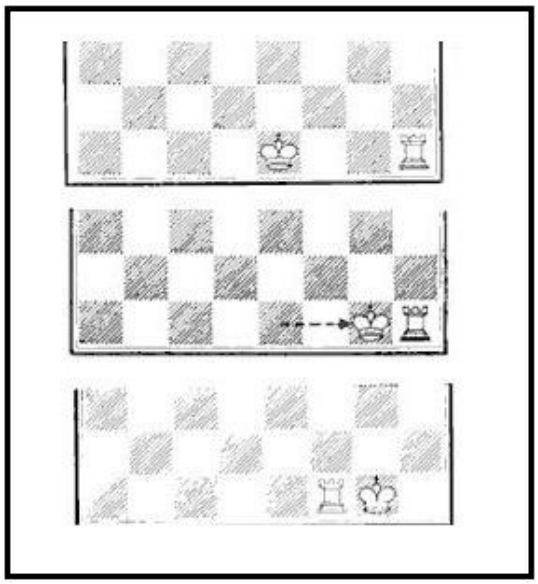


Fonte: Becker (2002, p. 17).

Figura 6 - Movimentação do cavalo

O roque é o único movimento em que duas peças podem ser movidas: o rei e a torre (BECKER, 2002, p. 18). Nele, o rei avança duas casas para um dos lados e a respectiva torre passa para o outro lado do rei (Figura 7).

Para efetuar o roque, é necessário que o rei não esteja atacado e que tanto ele quanto a torre ainda não tenham sido movidos na partida. Além disso, as casas por onde o rei passa não podem estar atacadas por peças adversárias e devem estar livres de quaisquer outras peças (BECKER, 2002, p.21).



Fonte: Becker (2002, p. 19).

Figura 7 - Funcionamento do roque

2.1.2 Xadrez randômico

O xadrez randômico (ou *xadrez960*), assim como o xadrez clássico, é jogado por duas pessoas, no mesmo tabuleiro e com o mesmo número de peças, também espelhadas. A diferença é que as casas iniciais de todas as peças, com exceção dos peões, são sorteadas (SCIMIA, 2002).

Segundo Scimia (2002), para que o sorteio seja válido é necessário que um dos bispos inicie numa casa branca e o outro numa casa negra, e o rei deve estar em alguma posição entre as torres. A Figura 8 mostra uma das 960 posições possíveis no sorteio da disposição das peças (MYCHESS, 2003).

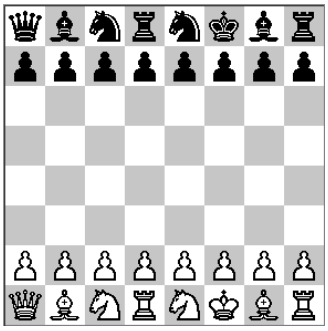


Figura 8 - Posição do xadrez randômico

2.1.3 Xadrez mata-mata

O xadrez mata-mata é jogado nas mesmas condições do xadrez clássico, porém o objetivo é perder a partida (CHESSVARIANTS, 2005).

Sempre que houver alguma captura de peça possível, o jogador é obrigado a executá-la. Quem conseguir entregar todas as suas peças primeiro, vence a partida.

Segundo o Chessvariants (2005), no mata-mata não existe xeque ou xeque-mate, podendo o rei ser entregue também. No caso de o jogador possuir a vez e não ter mais nenhum lance a ser feito, vence o jogador com menos peças no tabuleiro. Na Figura 9, a posição já não possui os reis no tabuleiro. Caso seja a vez das brancas elas vencem, pois não têm nenhum lance possível de ser feito, e, como as negras possuem mais peças em jogo, perdem a partida.

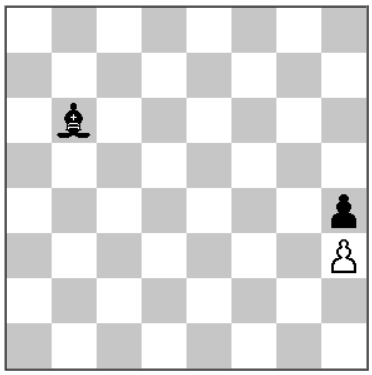


Figura 9 - Posição de xadrez mata-mata

2.1.4 PGN

Segundo Batista (2001), um arquivo PGN serve para anotar uma partida de xadrez.

O cabeçalho do arquivo serve para dar detalhes do jogo, como quem é o jogador de brancas e quem é o de pretas, qual o torneio, ritmo da partida e número de lances.

No corpo, os lances são descritos através da fórmula: número do lance, movimento das brancas, espaço, movimento das pretas, espaço, número do novo lance e assim por diante (Quadro 1).

Através deste arquivo, é possível reproduzir a partida em qualquer programa que possua leitor de PGN, como os softwares *Fritz* ou *Rybka*.

```
[Event "120'/40+60'/20+30'"]
[Site "?"]
[Date "2011.11.16"]
[Round "?"]
[White "Sobrenome1, Jogador1"]
[Black "Sobrenome2, Jogador2"]
[Result "1-0"]
[ECO "C23"]
[Annotator ""]
[PlyCount "7"]
[TimeControl "40/7200:20/3600:1800"]

1. e4 e5 2. Bc4 Nc6 3. Qf3 Bc5 4. Qxf7# 1-0
```

Quadro 1 - Exemplo de arquivo PGN

2.2 WEBSERVICES

Segundo Carneiro (2007), os sistemas distribuídos podes ser implementados de várias formas. Uma delas, o *Service-Oriented Architecture* (SOA), utiliza execução remota de aplicativos em forma de serviços. E a implementação mais utilizada para desenvolver o SOA é o *webservice*.

2.2.1 Conceito

Segundo Deitel e Deitel (2008, p. 676), um *webservice* é um componente de *software*, geralmente representado por uma *Uniform Resource Locator* (URL), que pode ser acessado e executado por outros componentes de *software* por chamadas remotas através de uma rede.

O *webservice* fica no lado do servidor, preparado para receber requisições de qualquer cliente que esteja na rede via *Hypertext Transfer Protocol* (HTTP). Através desta requisição, ele acessa o programa servidor, procura a função requisitada e devolve ao cliente, caracterizando uma *Remote Procedure Call* (RPC) (Figura 10).

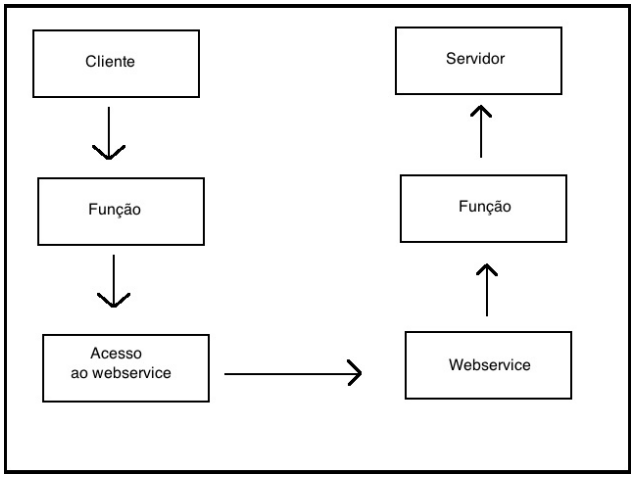


Figura 10 - Fluxo de funcionamento de um *webservice*

Segundo Basha et al. (2002, p. 10), para descrever o *webservice* é utilizado o protocolo *Web Service Description Language* (WSDL).

Segundo W3C (2001), a WSDL define em formato XML toda a estrutura, portas e nomenclatura do *webservice*. Através da WSDL são definidos os métodos com seus tipos de dados e parâmetros de entrada e saída que serão consumidos pelos clientes (Quadro 2).

```
<wsdl:binding name="nmtoken" type="qname">*
    <wsdl:documentation .... />?
    <-- extensibility element --> *
    <wsdl:operation name="nmtoken">*
       <wsdl:documentation .... /> ?
       <-- extensibility element --> *
       <wsdl:input> ?
           <wsdl:documentation .... /> ?
           <-- extensibility element -->
       </wsdl:input>
       <wsdl:output> ?
           <wsdl:documentation .... /> ?
           <-- extensibility element --> *
       </wsdl:output>
       <wsdl:fault name="nmtoken"> *
           <wsdl:documentation .... /> ?
           <-- extensibility element --> *
       </wsdl:fault>
    </wsdl:operation>
</wsdl:binding>

<wsdl:service name="nmtoken"> *
    <wsdl:documentation .... />?
    <wsdl:port name="nmtoken" binding="qname">
       <wsdl:documentation .... /> ?
       <-- extensibility element -->
    </wsdl:port>
    <-- extensibility element -->
</wsdl:service>
```

Fonte: W3C (2001).

Quadro 2 - Trecho de arquivo WSDL

A WSDL possui seis elementos fundamentais (NUNES, 2006):

a) *definitions*: define o nome do *webservice* e declara os *namespaces* usados nele;

b) *types*: define os tipos de dados utilizados pelo *webservice*;

c) *message*: define os dados trocados numa operação. Para cada operação, seus

parâmetros de entrada e valores de retorno;

d) *portType*: define os tipos de mensagens que serão usadas na operação definida;

e) *binding*: define os protocolos utilizados para acessar os métodos do webservice, bem como o formato dos dados para cada um destes protocolos;

f) *service*: define o endereço URL para que o *webservice* seja acessado.

Segundo Topley (2003, p. 7), atualmente o protocolo mais usado pelos *webservices* é o *Simple Object Access Protocol* (SOAP). Ele define um *framework* para a troca de mensagens entre sistemas distribuídos e as diretrizes para chamadas de procedimentos remotos via XML (BASHA ET. AL.; 2002; p. 31).

O SOAP pode ser transportado via HTTP, o que o torna adaptável a diversas estruturas, como servidores *web*, servidores *proxy*, podendo também atravessar *firewalls* com facilidade (BASHA ET. AL.; 2002; p. 31).

Uma mensagem SOAP pode ser dividida em *header* e *body* (Quadro 3). O *header* é opcional, e contém informações para situações específicas como, por exemplo, se a mensagem só deve ser executada em determinado ponto do percurso entre o cliente e o servidor (DANTAS, 2007). O elemento *body* é obrigatório, e contém a mensagem a ser entregue.

```
POST /rpcrouter HTTP/1.1
Host: 127.0.0.1
Content-Type: text/xml; charset=utf-8
Content-Length: 559
SOAPAction: "http://mauricio.com"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
xmlns:xsi="Schema-Instance"
xmlns:xsd="Schema"
xmlns:soap="Envelope">
<soap:Body>
<Converte xmlns="http://conv.com.br">
<Valor>5</Valor>
<De>DEC</De>
<Para>BIN</Para>
</Converte>
</soap:Body>
</soap:Envelope>
```

Fonte: Dantas (2007).

Quadro 3 - Mensagem SOAP via HTTP

Para o desenvolvimento de *webservices*, uma das bibliotecas mais utilizadas é o *Java API for XML* (JAX) (SPÍNOLA, 2010).

2.2.2 JAX

O JAX é uma biblioteca que visa dar velocidade e simplificar a construção de um

webservice Java (SPÍNOLA, 2010).

Segundo Topley (2003, p. 18), o JAX possui técnicas muito parecidas com o *Remote Method Invocation* (RMI), com chamadas de funções remotas com objetos locais, caracterizando uma aplicação distribuída.

Para desenvolver um *webservice*, o JAX traz como principal vantagem uma abstração no XML, no SOAP e no HTTP, permitindo que o desenvolvedor preocupe-se mais com o desenvolvimento das classes lógicas de negócio e com as *interfaces* de chamada de funções remotas (TOPLEY, 2003, p. 19).

Sua maior limitação é que alguns tipos de dados não são suportados nas chamadas dos métodos (TOPLEY, 2003, p. 44). Neste caso é necessário criar uma classe que serialize o objeto, para transformá-lo num tipo passível de ser transmitido e assim seja enviado via XML.

2.3 JAVA WEB START

O *Java Web Start* faz parte do pacote da plataforma *Standard Edition* (SE) do Java. Seu principal objetivo é permitir uma *interface* gráfica mais interativa sem a necessidade de um navegador, e, portanto, independente de linguagens de internet como *HyperText Markup Language* (HTML) ou o *JavaScript* (SOL, 2001).

Segundo Oracle (2006), o *Java Web Start* é iniciado automaticamente quando é feito o primeiro *download* de um aplicativo que faça uso desta tecnologia. O aplicativo fica armazenado localmente, em memória *cache*, fazendo com que todas as inicializações posteriores sejam praticamente instantâneas. Ao executá-lo, o *Java Web Start* verifica se o cliente possui a versão mais atualizada do aplicativo em memória, comparando com a versão do servidor, e verifica se o Java está instalado corretamente (MARCELINO, 2003). Caso não esteja, ele primeiro inicia o *download* do Java para depois fazer o *download* da aplicação.

O acesso a um aplicativo *Java Web Start* pode ser feito por um *link* num *site*, um ícone local na própria máquina ou pelo visualizador de *cache* de aplicativos Java (ORACLE, 2006). Como é armazenado em um servidor, é possível garantir que o cliente executará sempre a versão mais atualizada disponível.

Segundo Oliveira (2005), a tecnologia *Java Network Launching Protocol* (JNLP) é a responsável pelas diretrizes de execução de uma aplicação que utilize o *Java Web Start*, funcionando como um *eXtensible Markup Language* (XML). Isto inclui uma verificação se o

cliente possui os recursos necessários para a execução além de alguns pacotes adicionais.

Apesar de o arquivo ter a estrutura de um XML, como mostrado no Quadro 4, é imprescindível que a sua extensão seja *.jnlp* (MARCELINO, 2003).

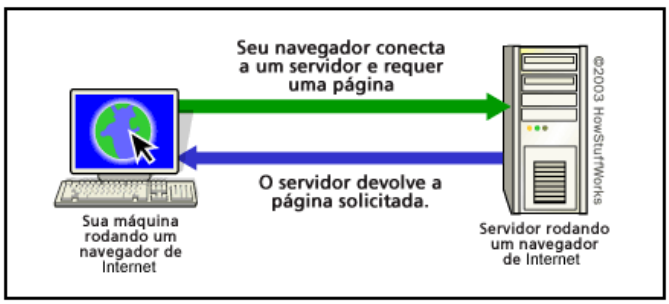
```
<?xml version="1.0" encoding="utf-8"?>
<jnlp
  spec="1.0+"
  codebase="http://localhost:8080/tutorialjws"
  href="TutorialJWS.jnlp">

    <information>
        <title>Tutorial Java Web Start</title>
        <vendor>GUJ - Grupo de Usuarios Java</vendor>
        <homepage href="http://localhost:8080/tutorialjws/index.html"/>
        <description>Tutorial Java Web Start</description>
        <description kind="short">Tutorial JWS</description>
        <icon href="images/logo.jpg"/>
        <offline-allowed/>
    </information>

    <resources>
        <j2se version="1.3+" href="http://java.sun.com/products/autodl/j2se"/>

        <jar href="TutorialJWS.jar"/>

        <property name="myProperty" value="Isso é um exemplo de propriedade"/>
    </resources>

    <application-desc main-class="br.com.guj.tutorial.jws.HelloJWS"/>
</jnlp>
```

Fonte: Oracle (2004).

Quadro 4 - Exemplo de arquivo JNLP

2.4 SERVIDOR WEB

Segundo Alecrim (2006), um servidor *web* é um computador capaz de processar requisições HTTP, dando sustentação a um sistema. Ele deve estar rodando a todo o momento, já que qualquer pessoa pode solicitar uma conexão, seja via navegador ou por meio de algum *software* (Figura 11).



Fonte: Brain (1999).

Figura 11 - Funcionamento de um servidor *web*

A principal função de um servidor *web* é dar suporte a sites e sistemas *web* que usem dados dinâmicos. Ou seja, dados extraídos de um banco de dados, *scripts* ou de outros

elementos (SOBRAL, 2001).

Todas as requisições processadas e respondidas por um servidor *web* são essencialmente texto, que são interpretados pelo cliente e pelo servidor como vários cabeçalhos seguidos por um corpo (CRANE; JAMES; PASCARELLO, 2007, p. 40).

O servidor ainda tem a opção do envio de *cookies*, que são identificadores das ações do cliente no servidor, para que numa próxima vez que o mesmo cliente volte a visitá-lo, o servidor possa disponibilizar uma informação mais direcionada (CRANE; JAMES; PASCARELLO, 2007, p. 40).

Alguns servidores *web* têm opções adicionais, como registro de estatísticas, segurança de manipulação e criptografia de dados (AURÉLIO, 2005).

Desenvolvido pela *Sun Microsystems*, hoje Oracle, o *GlassFish* é um dos principais servidores existentes no mercado (MOUSSINE; PELEGRI; YOSHIDA, 2007).

2.4.1 GlassFish

Criado em 2005, o *GlassFish* possui compatibilidade com diversos tipos de tecnologia, como *Java Enterprise Edition* (JEE), *Java Server Faces* (JSF) e *Java Server Pages* (JSP), além de ter compatibilidade com diversos tipos de *webservices*, incluindo o JAX (MOUSSINE; PELEGRI; YOSHIDA, 2007).

Além de possuir várias APIs que facilitam o desenvolvimento e de ter extensa documentação disponível, o *GlassFish* permite múltiplas versões e compartilhamento de bibliotecas entre projetos, além de realizar um *deploy* automaticamente assim que alguma estrutura do projeto for modificada (LONGO, 2009).

Segundo Longo (2009), a respeito do banco de dados, o *GlassFish* facilita a criação e gerenciamento do *pool* de conexões. Ou seja, reduz o tempo das conexões estabelecidas criando uma conexão já na inicialização do sistema.

Segundo Pouchkine (2007), o *GlassFish* ainda pode rodar tanto num *desktop* quanto num celular, além de ser totalmente modular e consumir pouca memória.

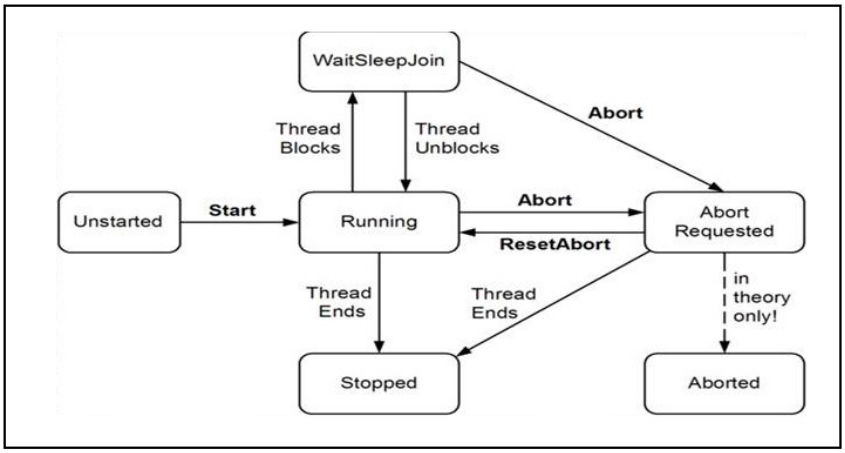
2.5 THREADS

Threads são sub-processos no sistema operacional, que tornam o processamento do programa menos custoso (GUEDES, 2004).

Em Java, os *threads* não são programas, mas sim rodam dentro dos programas e permitem com que várias tarefas sejam executadas ao mesmo tempo, independentemente umas das outras (ALCÂNTARA, 1996).

Segundo Morimoto (2005), um programa dividido em vários *threads* pode rodar mais rápido do que um programa monolítico, pois várias tarefas podem ser executadas simultaneamente, além de trocar dados entre si e compartilhar áreas de memória.

De acordo com Hisha e Snesha (2009), um *thread* possui diversos estados durante a execução do programa (Figura 12).



Fonte: Hisha e Snesha (2009).

Figura 12 - Estados de um *thread*

Segundo Silva (2007), os *threads* podem ser usados para implementar o *game-loop*. O *game-loop* é um conceito muito utilizado em programação de jogos, onde um ciclo "infinito" fica executando e atualizando a tela, independentemente de haver novos dados ou não.

2.6 TRABALHOS CORRELATOS

Aqui estão descritos três sistemas *online* de jogo de xadrez: o *Internet Chess Club* (ICC, 1996), o *Buho21* (2001) e o Internet Xadrez Clube (IXC, 1999).

2.6.1 ICC

O ICC é o sistema mais utilizado no mundo para se jogar xadrez. Apesar de ser pago (gratuito apenas para teste), o ICC possui uma interface bem simples e a maioria das ações são feitas numa espécie de "linha de comando" própria.

O ICC também disponibiliza diversos tipos de tabuleiros e peças. Permite que um desafio seja feito de um jogador para outro, ou que o desafio seja lançado num gráfico para que o primeiro que clicar sobre ele aceite o desafio e disponibiliza um *chat* para que todos os jogadores possam interagir (ICC, 1996).

Ainda, permite que qualquer jogador entre em uma partida em andamento e veja os lances executados. Dispõe de um relógio para cada jogador para marcar o tempo da partida e armazena as partidas de cada jogador num banco de dados para futuras consultas.

O ICC disponibiliza ainda uma opção para jogar contra uma *engine* programada, que possui vários níveis de jogo, caso o usuário não queira enfrentar outro jogador.

A Figura 13 mostra o ambiente de jogo que o jogador visualiza ao iniciar uma partida no ICC.

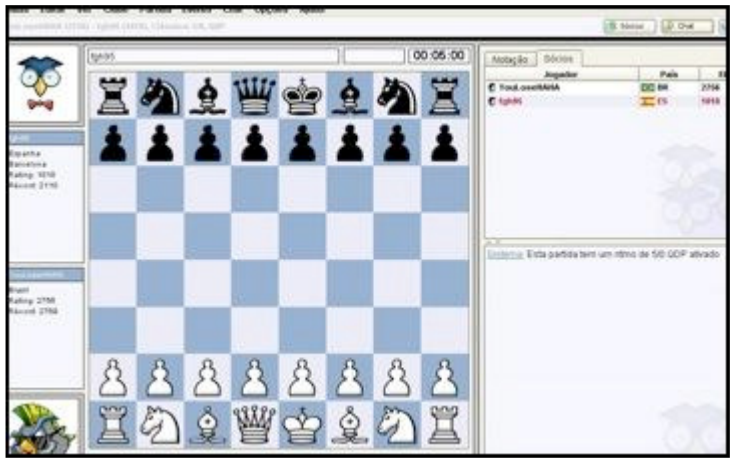
Fonte: ICC (1996).

Figura 13 - Ambiente de jogo do ICC

2.6.2 Buho21

O *Buho21* é um ambiente feito em Java e gratuito. Apesar de possuir também uma espécie de linha de comando, nenhuma interação com o sistema necessita dela, tendo botões e menus para substituí-la (MARLBORO, 2009).

Além do xadrez, o *Buho21* permite jogos derivados desse, como mata-mata, xadrez australiano e xadrez randômico, além de mostrar num quadro todas as pessoas que estão na sala de jogo no momento (BUHO21, 2001). A Figura 14 mostra o ambiente de jogo visto pelo usuário durante a partida, com as telas de *chat*, jogo e participantes da partida.



Fonte: Buho21 (2001).

Figura 14 - Ambiente de jogo do *Buho21*

2.6.3 IXC

O IXC (1999) é um sistema de jogo de xadrez brasileiro, desenvolvido por programadores de Porto Alegre.

Neste sistema, é possível jogar xadrez exclusivamente, sem qualquer jogo derivado.

Cada jogador possui uma pontuação, que varia de acordo com as vitórias e derrotas que consegue. As partidas possuem controle de tempo fixo e incremental. No controle incremental, um bônus, em segundos, é acrescido ao tempo do jogador depois que ele efetua um lance.

O IXC disponibiliza também *engines* programadas em vários níveis para que o jogador treine caso não queira enfrentar nenhum dos jogadores conectados. Possui uma biblioteca com várias partidas de jogadores importantes, banco de dados de aberturas e diagramas com posições de desafios para treino.

3 DESENVOLVIMENTO DO SOFTWARE

O ambiente para o jogo de xadrez foi desenvolvido seguindo as três etapas básicas de construção de um software: análise, implementação e testes.

A análise do projeto seguiu as diretrizes da orientação a objetos utilizando diagramas da *Unified Modeling Language* (UML). Dentre eles, o diagrama de classes para representar o software como um todo e diagramas de estados, sequência e casos de uso para representar processos mais complexos do sistema. Para desenvolver os diagramas, foi utilizada a ferramenta *Enterprise Architect* (EA) 7.5.

Para o desenvolvimento do *software* foi utilizada a *Integrated Development Environment* (IDE) livre *NetBeans*, que já disponibiliza o servidor *GlassFish*, a biblioteca *Java API for XML* (JAX) e o gerador de código para o *Java Web Start*.

O desenvolvimento foi todo implementado no Mac OS X 10.6.8, porém testado no Windows XP e no Windows 7 também.

A seguir estão listados os processos do desenvolvimento, a lógica do desenvolvimento da ferramenta e alguns casos de teste.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

Durante a fase de análise foram especificados os seguintes requisitos para o ambiente do jogo de xadrez:

a) permitir que um desafio seja feito de um jogador para outro (Requisito Funcional – RF);
b) possibilitar que os jogadores joguem, além de xadrez clássico, mata-mata e xadrez randômico (RF);
c) permitir que qualquer jogador entre em uma partida em andamento e veja os lances executados (RF);
d) dispor de um relógio para cada jogador para marcar o tempo da partida (RF);
e) permitir que o desafio seja lançado numa tabela, para que o primeiro que clicar sobre ele o aceite (RF);
f) armazenar as partidas de cada jogador num banco de dados para futuras consultas

(RF);

g) mostrar num quadro todas as pessoas que estão na sala de jogo no momento (RF);
h) permitir que apenas um usuário por *e-mail* seja cadastrado (RF);
i) atribuir ou decrementar pontos de um usuário após o término da partida, dependendo do seu resultado (RF);
j) fazer distinção dos pontos de acordo com o tipo de jogo (RF);
k) permitir que o usuário faça o download de sua partida após o término desta, em formato *Portable Game Notation* (PGN) (RF);
l) utilizar o *MySQL* para gerenciar o banco de dados (Requisito Não-Funcional - RNF);
m) utilizar o *framework Java Web Start* para desenvolver a interface gráfica (RNF);
n) utilizar o *NetBeans* na implementação do software (RNF);
o) utilizar a biblioteca JAX no desenvolvimento do *webservice* (RNF);
p) utilizar o *GlassFish* como servidor de aplicação (RNF);
q) utilizar *threads* para implementar um *game-loop*, para atualizar a aplicação cliente com os dados do servidor (RNF).

3.2 ESPECIFICAÇÃO

O *software* foi dividido em dois projetos: o cliente, denominado `XadrezCliente` e o servidor, chamado de `XadrezServidor`.

No `XadrezServidor` está o *webservice* `XadrezWebService`, que possui todos os métodos que serão acessados remotamente pelo cliente. Também do lado do servidor estão as classes que interagem com o banco de dados e o gerador do arquivo JNLP do *Java Web Start*.

No `XadrezCliente` estão todas as chamadas remotas e verificação se a jogada é válida ou não. Caso seja, o lance é enviado ao servidor e partida é atualizada.

3.2.1 Casos de uso

O diagrama de casos de uso da Figura 15 mostra as ações executadas pelo ator, que é o

jogador.

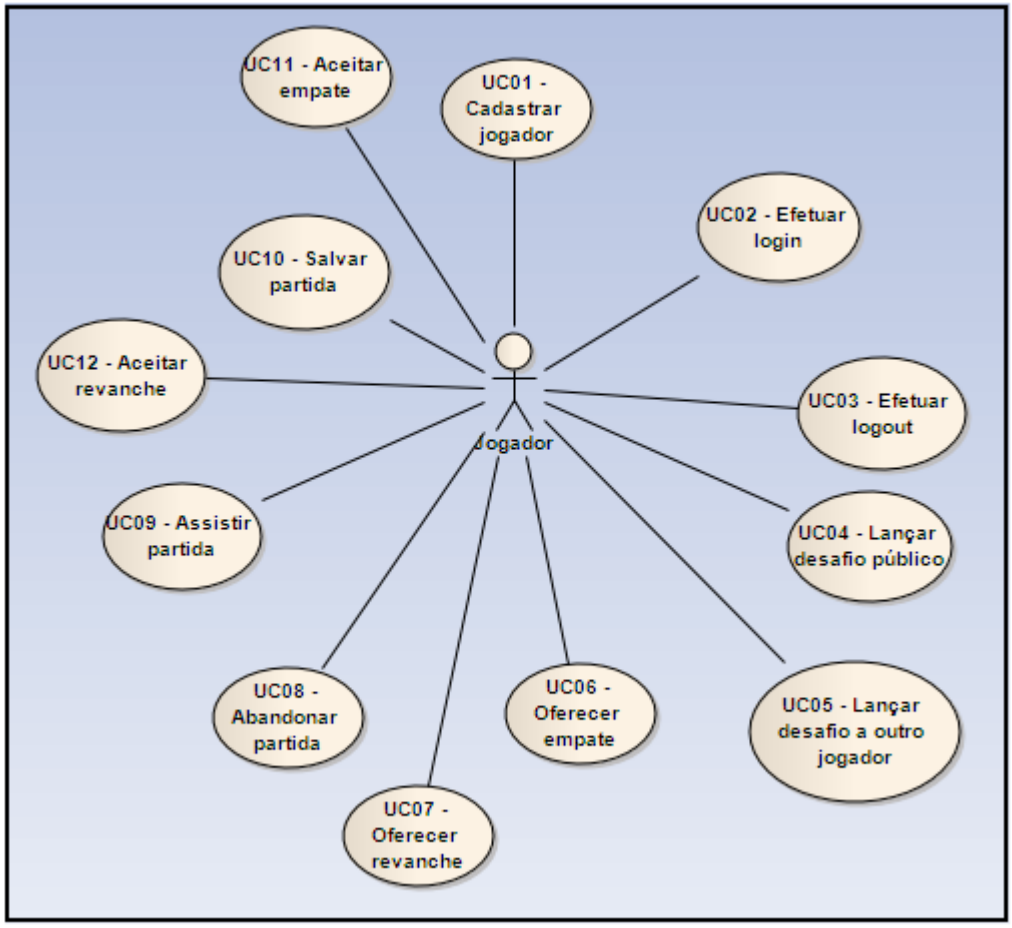


Figura 15 - Casos de uso

A descrição do funcionamento de cada método está no Apêndice A.

3.2.2 Projeto `XadrezServidor`

O projeto `XadrezServidor` é o responsável pelo gerenciamento das partidas e dos jogadores, bem como manter a conexão com o banco de dados.

Foi dividido em cinco pacotes: `beans`, `img`, `persistencia`, `controle` e `webservice`.

3.2.2.1 Pacote `beans`

O pacote `beans` contém as classes `Desafio`, `Partida` e `Jogador`. Estas três classes possuem apenas métodos *setters* e *getters*, atuando apenas na criação de objetos persistidos. O

relacionamento entre elas está demonstrado na Figura 16.

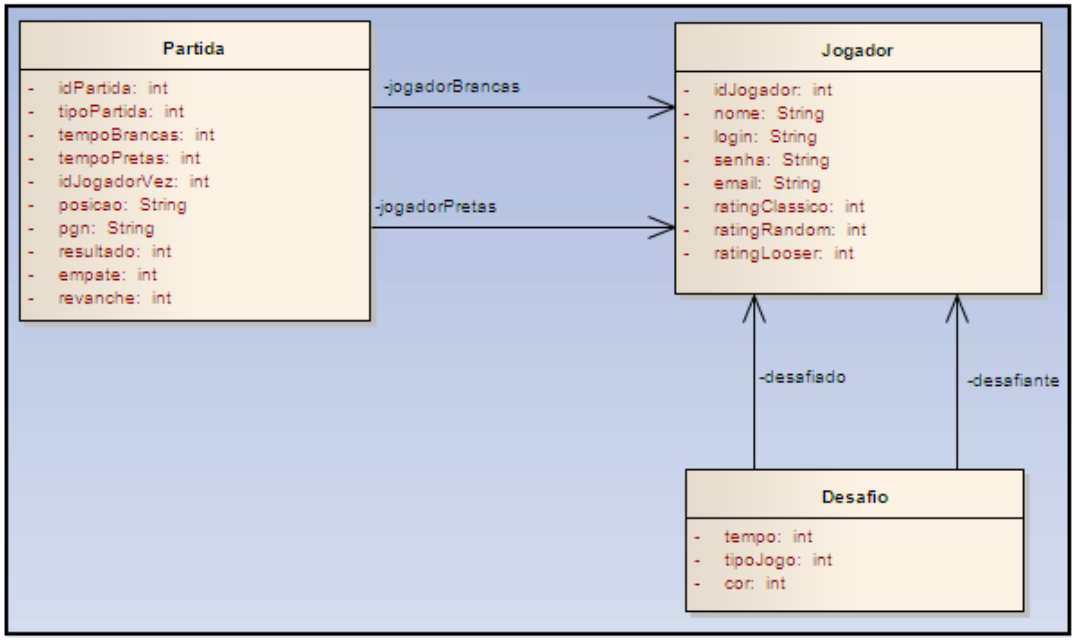


Figura 16 - Pacote `beans`

A classe `Jogador` possui os atributos `nome`, `login`, `senha` e `email` para cadastro e identificação do usuário. O atributo `email` foi criado para que um usuário não crie vários *logins* e fique passando pontos de um usuário para outro, além de servir para futuras implementações para que o usuário possa recuperar a senha por e-mail caso esqueça.

Os atributos `ratingClassico`, `ratingRandom` e `ratingLooser` serão as pontuações acumuladas de cada jogador para cada tipo de jogo no sistema. Esta pontuação será atualizada no banco de dados toda vez que uma partida terminar. O *rating* sempre será mostrado quando um desafio for lançado, para que o jogador que está recebendo o desafio saiba aproximadamente o nível do outro jogador.

A classe `Desafio` possui dois atributos da classe `Jogador`: o `desafiante` e o `desafiado`. O `desafiante` será aquele que lançará o desafio e que definirá o tempo de jogo no atributo `tempo` e o tipo de jogo no atributo `tipoJogo` – este inteiro definirá se o xadrez é clássico, randômico ou mata-mata. Caso o atributo `desafiado` seja `null`, significa que o desafio será aberto, lançado em uma tabela para que qualquer jogador conectado o aceite.

A classe `Partida` possui um atributo `tipoPartida` que irá definir qual tipo de jogo de xadrez é. Este atributo será utilizado pelo cliente para saber quais regras ele utilizará. Os atributos `tempoBrancas` e `tempoPretas` são do tipo `int`, e serão os cronômetros de cada jogador para terminar a partida. O atributo `idPartida` recebe o `idJogador` do `Jogador` que tem a vez. Quando um lance é executado, este atributo é mudado para que o outro jogador receba a mensagem de que é a vez dele.

O atributo `posicao` guardará a posição do tabuleiro em formato `String`, que será quebrado e passado para uma `JTable`. A regra para a `posicao` é a seguinte: vendo o tabuleiro como o jogador de brancas, a partir da primeira casa, é colocada a letra inicial da peça na `String`. Letras minúsculas são usadas para representar as peças pretas e maiúsculas as brancas. Quando a casa está vazia, é colocado um "x". O Quadro 5 mostra a posição inicial de um tabuleiro de xadrez no formato `String`, começando pela torre preta e terminando na torre branca, totalizando 64 caracteres para as 64 casas do tabuleiro.

```
posicao = tcbdrbctpppppppppxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxPPPPPPPPTCBDRBCT
```

Quadro 5 - Posição inicial no formato `String`

A `String pgn` serve para guardar os dados da partida em formato PGN, que poderá ser baixado após a partida. Este arquivo possui uma formatação específica, com os nomes dos jogadores e todos os lances executados.

Os atributos `resultado`, `empate` e `revanche` são inicializados com -1. Quando a partida terminar, o `resultado` receberá o valor do `idJogador` do `Jogador` vencedor. O `empate` receberá o valor do `idJogador` do `Jogador` que estiver oferecendo o empate. Caso a oferta seja recusada, o valor volta a ser -1. A `revanche` receberá o `idJogador` do `Jogador` que oferecer revanche. Quando a partida terminar, caso alguém clique no botão "revanche", este valor será mudado e o outro `Jogador` buscará a atualização no servidor, verificando que a revanche foi oferecida pelo outro `Jogador` já que o valor não é mais -1.

3.2.2.2 Pacote `img`

O pacote `img` (Figura 17) contém todas as imagens, em formato *Portable Network Graphics* (PNG), das peças que serão utilizadas no tabuleiro.

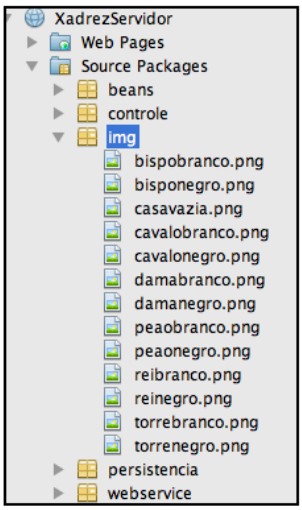


Figura 17 - Pacote `img`

3.2.2.3 Pacote `persistencia`

O pacote `persistencia` possui apenas a classe `ConexaoMySQL`, que faz a conexão com o banco de dados *MySQL* (Figura 18).

Figura 18 - Classe `ConexaoMySQl`

O atributo `DBAdress` define o *Internet Protocol* (IP) da máquina onde está o banco de dados. Como o servidor *GlassFish* está executando na mesma máquina do banco, o `DBAdress` foi deixado com valor *localhost.* A porta de acesso do *MySQL* é a 3306, definida no atributo `DBPort`.

Os atributos `DBName`, `DBUser` e `DBPassword` definem, respectivamente, nome do banco, usuário e senha. O nome do banco usado no trabalho foi `xadrezbanco`, com usuário `root` sem senha.

O atributo `DBURL` faz uma concatenação dos outros atributos para criar o endereço

completo do banco.

O atributo `DBDriver` guarda uma referência para o *driver* `mysql-connector-java-5.1.3.jar`, adicionado na biblioteca do projeto (Figura 19).

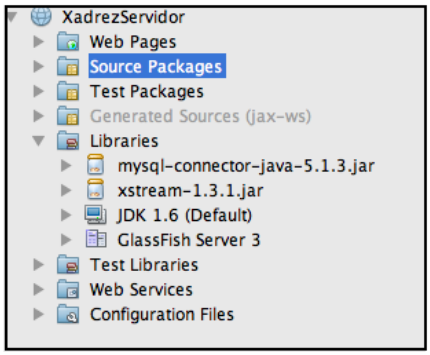


Figura 19 - *Driver* do *MySQL*

O método `cadastrarJogador` insere um jogador no banco de dados.

Os métodos `jahExisteLogin` e `jahExisteEmail` verificam se o *login* e e-mail já existem quando um usuário tenta efetuar o seu cadastro no sistema.

O método `loginEhValido` verifica se usuário e senha estão corretos na execução do *login*. Caso estejam, o método `efetuarLogin` retorna o objeto `Jogador` pronto.

O método `alterarRating` altera, baseado no `idJogador` passado por parâmetro e do tipo do jogo, a pontuação do `Jogador` no banco de dados. Também baseado no `idJogador` do `Jogador`, o método `recuperarPartidas` procura por todas as partidas cadastradas do usuário no banco.

3.2.2.4 Pacote `controle`

O pacote `controle` contém apenas a classe `ControleServidor`, que possui métodos de controle interno e também de acesso aos métodos da classe `ConexaoMySQL`. Foi criado para separar o acesso do *webservice* ao banco de dados por uma camada, numa espécie de *Model View Control* (MVC).

A classe `ControleServidor` possui quatro atributos (Figura 20). O atributo `conexao` guarda a conexão com o banco de dados *MySQL* que é inicializada no seu construtor.

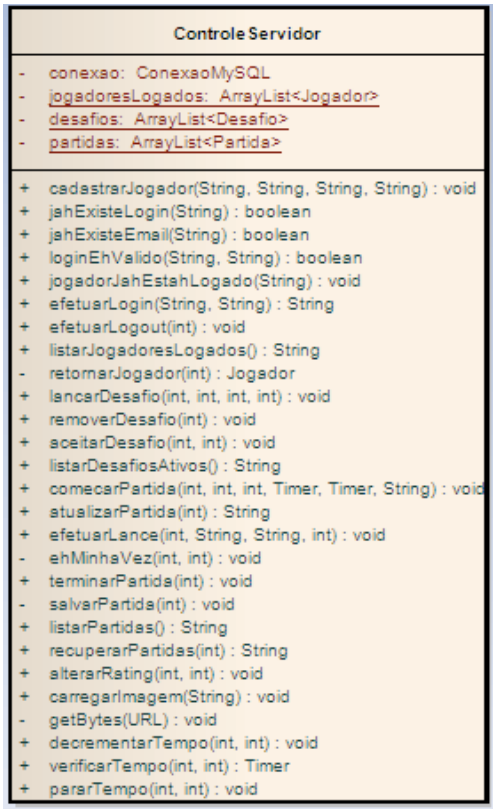


Figura 20 - Classe `ControleServidor`

O atributo `jogadoresLogados` armazena todos os jogadores que estão conectados na sala de jogo. O atributo `desafios` guarda todos os desafios que estão lançados e o atributo `partidas` guarda as partidas que estão em andamento no momento. Estes três atributos são estáticos, já que as listas de jogadores, partidas e desafios possuem apenas uma instância.

Os métodos `cadastrarJogador`, `jahExisteLogin`, `jahExisteEmail` e `alterarRating` chamam os métodos homônimos da classe `ConexaoMySQL`.

Na execução do *login*, o primeiro método executado é o `loginEhValido`, que chama o método da classe de conexão. Depois é executado o `jogadorJahEstahLogado`, que pesquisa na lista `jogadoresLogados` se este usuário já está conectado ao sistema.

O método `efetuarLogin` adiciona o usuário, já validado, na lista de usuários conectados. O método `efetuarLogout` remove o usuário desta mesma lista.

Os métodos `listarJogadoresLogados`, `listarDesafiosAtivos` e `listarPartidas` possuem uma lógica semelhante, todas retornando uma `String`. Nesta `String` está a estrutura XML dos atributos, que será passado via *webservice*. Como o *webservice* não aceita tipos abstratos de dados, é necessário passar os objetos para XML.

O método `lancarDesafio` cria um novo desafio com as condições passadas por parâmetro e adiciona na lista `desafios`. O método `atualizarPartida` será utilizado pelo *thread* para atualizar a situação no tabuleiro e nos relógios de cada jogador.

Quando o jogador fizer a primeira conexão, o método `carregarImagens` irá passar as imagens das peças para a memória local do cliente, tornando a renderização do tabuleiro mais rápida. A imagem é passada como um *array* de *bytes* que será remontado pelo cliente, produzindo a mesma imagem.

3.2.2.5 Pacote `webservice`

O pacote *webservice* contém a classe `XadrezWebService`, que possui todos métodos do `ControleServidor`. Seu único atributo é um objeto do tipo `ControleServidor` para a chamada dos métodos.

Ao fazer o *deploy* da aplicação, o arquivo `XadrezWebServiceService.wsdl` é gerado com todos os métodos implementados estruturados em formato WSDL (Quadro 6).

```
<message name="cadastrarJogador">
<part name="parameters" element="tns:cadastrarJogador" />
</message>
<message name="loginJahExiste">
<part name="parameters" element="tns:loginJahExiste" />
</message>
<message name="loginJahExisteResponse">
<part name="parameters" element="tns:loginJahExisteResponse" />
</message>
<message name="jahExisteEmail">
<part name="parameters" element="tns:jahExisteEmail" />
</message>
<message name="jahExisteEmailResponse">
<part name="parameters" element="tns:jahExisteEmailResponse" />
</message>
<message name="loginEhValido">
<part name="parameters" element="tns:loginEhValido" />
</message>
<message name="loginEhValidoResponse">
<part name="parameters" element="tns:loginEhValidoResponse" />
</message>
<message name="jogadorJahEstahLogado">
<part name="parameters" element="tns:jogadorJahEstahLogado" />
</message>
<message name="jogadorJahEstahLogadoResponse">
<part name="parameters" element="tns:jogadorJahEstahLogadoResponse" />
</message>
<message name="efetuarLogin">
<part name="parameters" element="tns:efetuarLogin" />
</message>
<message name="efetuarLoginResponse">
<part name="parameters" element="tns:efetuarLoginResponse" />
</message>
<message name="efetuarLogout">
<part name="parameters" element="tns:efetuarLogout" />
</message>
<message name="listarJogadoresLogados">
<part name="parameters" element="tns:listarJogadoresLogados" />
</message>
```

Quadro 6- Trecho do arquivo XadrezWebServiceService.wsdl

3.2.3 Projeto `XadrezCliente`

O projeto `XadrezCliente` possui todas as telas e interações com o usuário, fazendo uso dos métodos do projeto `XadrezServidor`.

O único controle feito é em relação às regras do xadrez e renderização das imagens com as informações buscadas no servidor.

O projeto foi dividido em oito pacotes: `beans`, `excecoes`, `telas`, `pecas`, `modelosTabelas`, `threads`, `controle` e `webservice`.

O pacote `beans` tem o mesmo objetivo do projeto `XadrezServidor`, ou seja, guardar as classes que possuem apenas métodos *set* e *get* para receber o objeto XML e desmontá-lo.

O pacote `excecoes` guarda as classes personalizadas de exceções, como a `LoginException`, que será chamada quando as validações de *login* do usuário retornarem com erro do servidor.

O pacote `telas` guarda as classes de *interface* gráfica. As classes `LoginGUI`, `LoginDialog`, `PrincipalGUI`, `DesafioDialog` e `PartidaGUI` serão usadas para a interação com o usuário.

3.2.3.1 Pacote `pecas`

O pacote `pecas` guarda todas as classes das peças do tabuleiro (Figura 21).

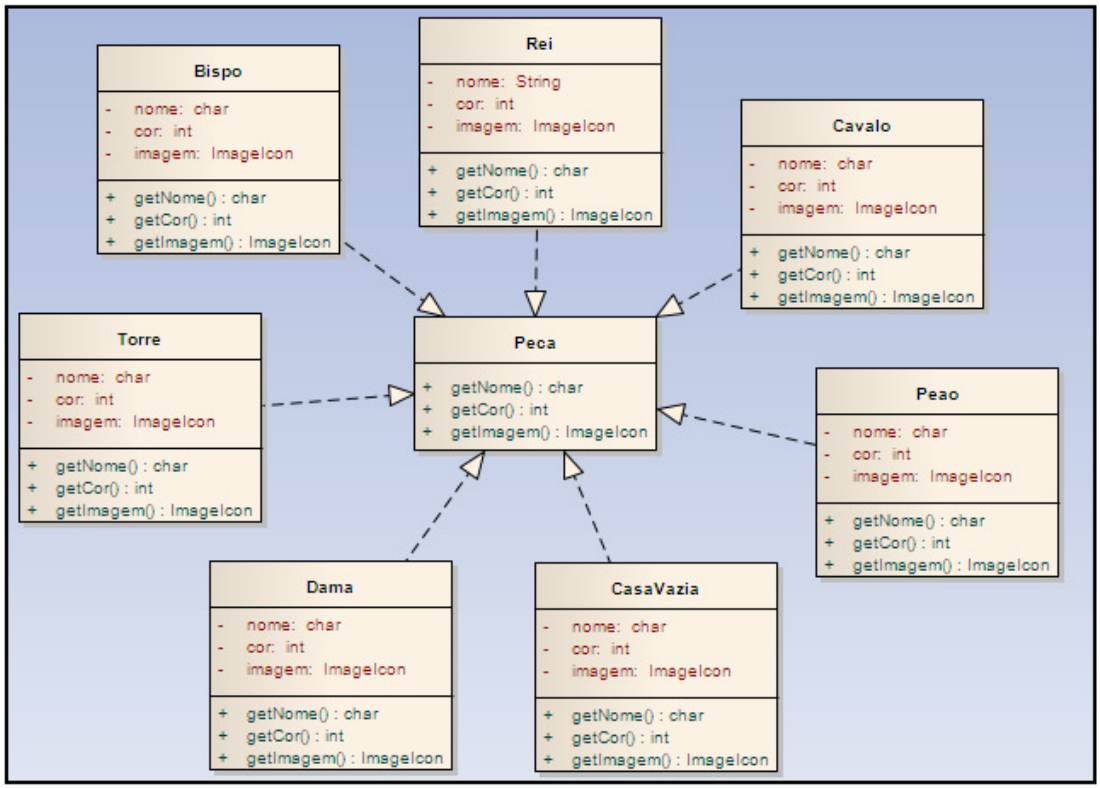


Figura 21 - Pacote `pecas`

Cada classe possui um atributo `nome` de tipo *char*, que já identificará a cor da peça. Por exemplo: na classe `Bispo`, se o nome for “B”, o bispo é branco e se for “b”, preto. A `CasaVazia` é exceção pois não possui cor e serve apenas para os algoritmos de verificação de regras. O atributo `imagem` guardará localmente as imagens trazidas do servidor após a primeira execução do *software*.

A *interface* `Peca` é realizada por todas as classes do pacote. A criação da *interface* é para facilitar o manuseio do *array* de peças do tabuleiro, tornando possível a generalização de conteúdo do mesmo.

3.2.3.2 Pacote `modelosTabelas`

O pacote `modelosTabelas` contém classes para fazer o manuseio das tabelas `JTable` utilizadas nas telas (Figura 22).

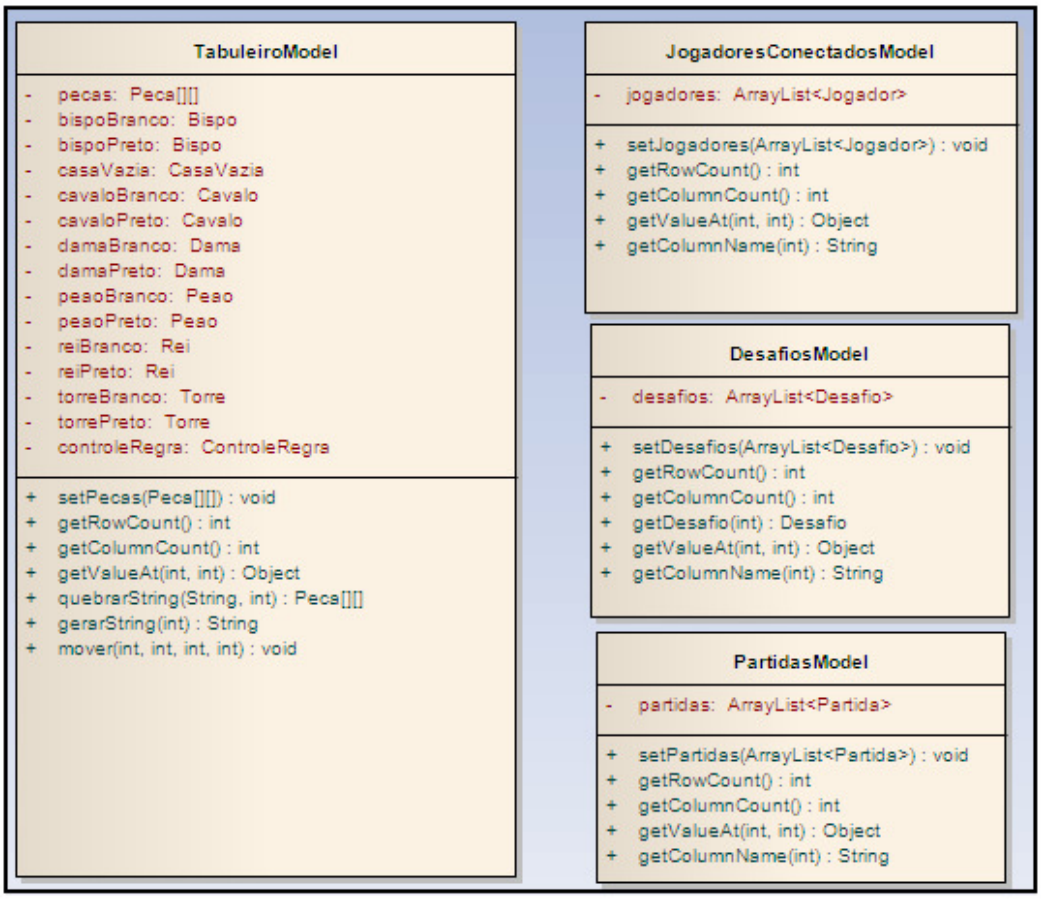


Figura 22 - Pacote `modelosTabelas`

Todas elas estendem da classe `AbstractTableModel` e, por isso, precisam implementar os métodos abstratos `getRowCount`, `getColumnCount` e `getValueAt`. O método `getColumnName`, apesar de não obrigatório, também pertence à classe `AbstractTableModel`, e determina o nome que aparecerá na coluna da `JTable`.

A classe `JogadoresConectadosModel` possui um *array* de jogadores que estiverem conectados no momento. Este *array* será constantemente atualizado pelos *threads* através do método `setJogadores`. Uma lógica semelhante foi utilizada nas classes `DesafiosModel` e `PartidasModel`. Estas três classes buscam os *arrays* da classe `ControleServidor`, através do *webservice*.

A classe `TabuleiroModel` possui um *array* bidimensional de `Peca` como atributo, que será o *array* que o usuário verá na tela. Baseado neste *array* que as regras serão testadas.

O método `quebrarString` receberá como parâmetros a posição do tabuleiro e a cor do jogador que verá aquela posição. O controle de cor foi feito, pois a `JTable` contém sempre os mesmos índices de linhas e colunas. No entanto, quando o jogador estiver jogando de brancas, é necessário que as peças brancas estejam embaixo e quando estiver de pretas, que as peças pretas estejam embaixo. Baseado na cor, o método `quebrarString` gera *arrays* diferentes, mas com a mesma posição. Por exemplo, a Figura 23 mostra a mesma posição vista pelos dois

lados. Enquanto que na posição esquerda o índice (0,0) da `JTable` é uma torre preta, o mesmo índice na posição à direita é uma torre branca.

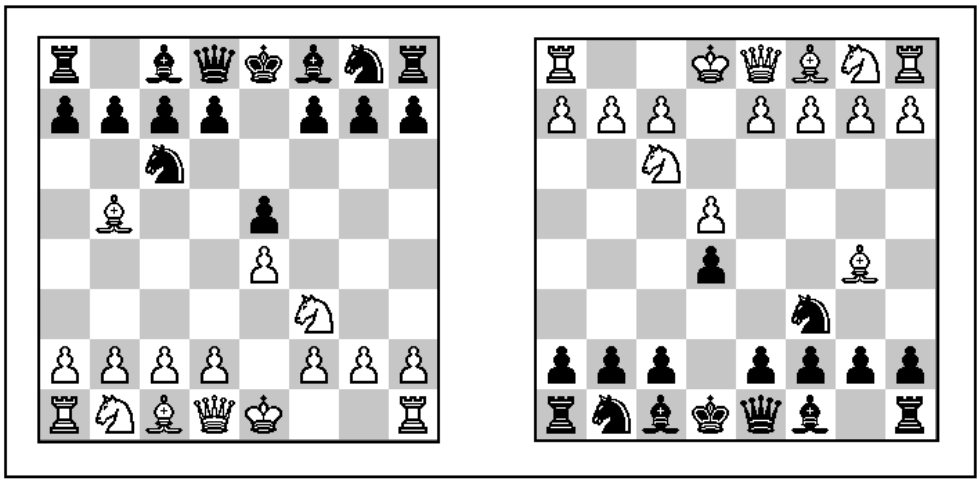


Figura 23 - Posições iguais

O método `gerarString` faz a operação inversa do método `quebrarString`. Baseado na cor do jogador, ele gera a *String* com a posição e envia ao servidor.

O método `mover (linhaInicial, colunaInicial, linhaFinal, colunaFinal)` transfere a peça localizada no par `(linhaInicial, colunaInicial)` para a `(linhaFinal, colunaFinal)`, se o lance for legal. A verificação se o lance é legal está na classe `ControleRegra`.

3.2.3.3 Pacote `threads`

O pacote `threads` contém as classes `DesafiosThread`, `PartidasThread`, `TabuleiroThread` e `JogadoresThread`, todas estendendo da classe `Thread` e implementando o método `run()` (Figura 24).

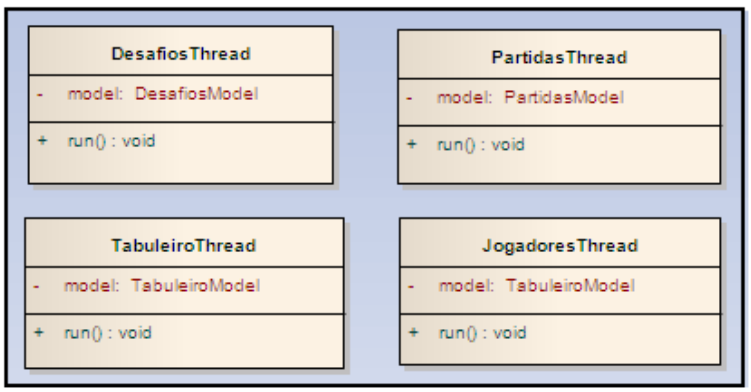


Figura 24 - Pacote `threads`

Cada uma dessas classes possui um respectivo *model* do pacote `modelosTabelas`, no

qual cada *thread* irá trabalhar e renderizar.

3.2.3.4 Pacote `controle`

O pacote controle possui três classes: `ControleCliente`, `LimitadorField` e `ControleRegra` (Figura 25).

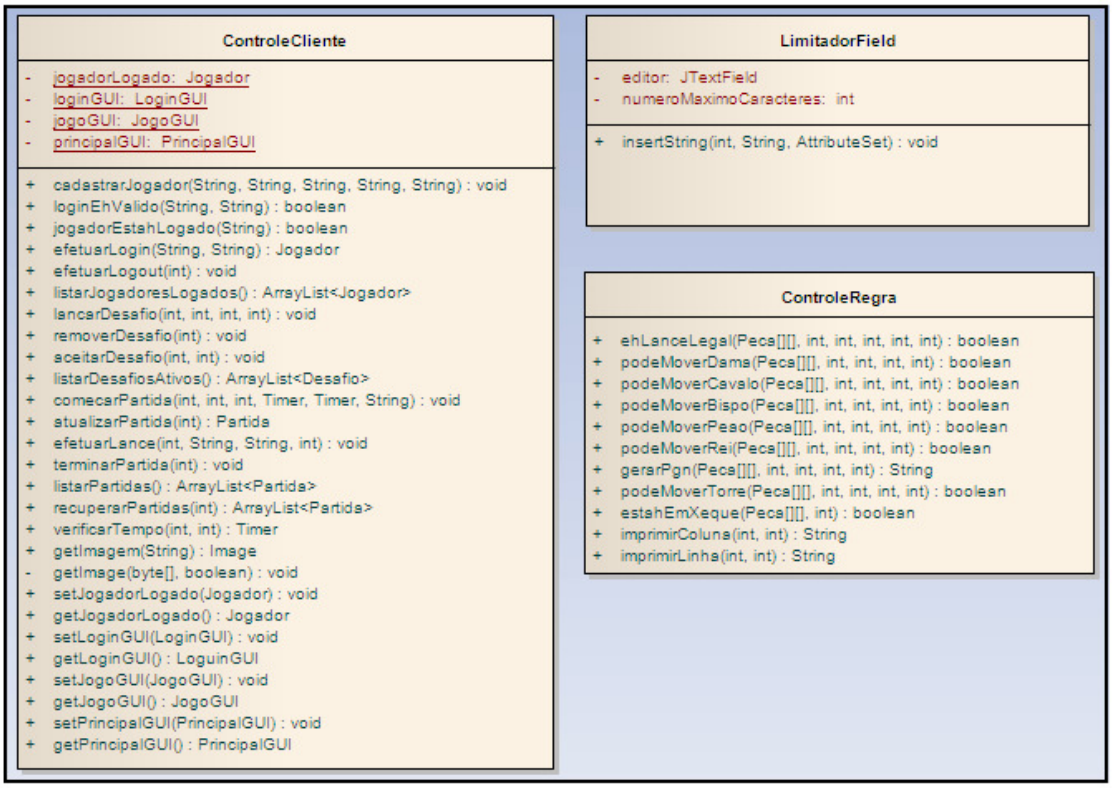


Figura 25 - Pacote `controle`

A classe `LimitadorField` estende da classe `PlainDocument`, e é utilizada para limitar o tamanho dos `JTextField` que serão preenchidos pelo usuário.

A classe `ControleRegra` contém o método `ehLanceLegal`, que sempre será chamado quando o jogador tentar efetuar um lance. Este método, ao verificar a peça que está sendo mexida, chama o método privado mais adequado. Por exemplo: caso a peça que esteja sendo mexida seja um bispo, o método `podeMoverBispo` será executado.

Um lance só é válido quando o método `estahEmXeque` retorna *false* e quando o método `podeMover` referente à peça retorna *true*. Este método será chamado dentro de cada método privado, com exceção do mata-mata, pois neste jogo o rei pode ser capturado.

O método `gerarPgn` gera um lance que será concatenado na `String pgn` do servidor. Para gerar a `String`, ele utiliza os métodos `imprimirColuna` e `imprimirLinha`, passando a

cor como parâmetro. A cor é passada, pois caso o jogador esteja de brancas, a primeira linha deve ser a número 1; caso esteja de pretas, deve ser a 8, pois o tabuleiro possui apenas uma nomenclatura, independentemente do ponto de vista do jogador.

A classe `ControleCliente` possui os métodos de chamada aos métodos descritos no *webservice*. A diferença está nos métodos que geravam XML: no `ControleServidor`, era passada uma `String` com o objeto serializado e aqui, o objeto é construído novamente a partir da `String`.

Os atributos `loginGUI`, `jogoGUI`, `principalGUI` e `jogadorLogado` são estáticos e referenciam as telas e o jogador que fizer o *login* em memória. Tendo o usuário, é mais fácil para fazer as requisições ao servidor passando o `idJogador`, e as telas não precisarão de instâncias a cada nova execução.

3.2.3.5 Pacote `webservice`

O pacote `webservice` é gerado pela biblioteca JAX (Figura 26) ao adicionar um cliente de *webservice* no projeto, apontando para o *webservice* do servidor.

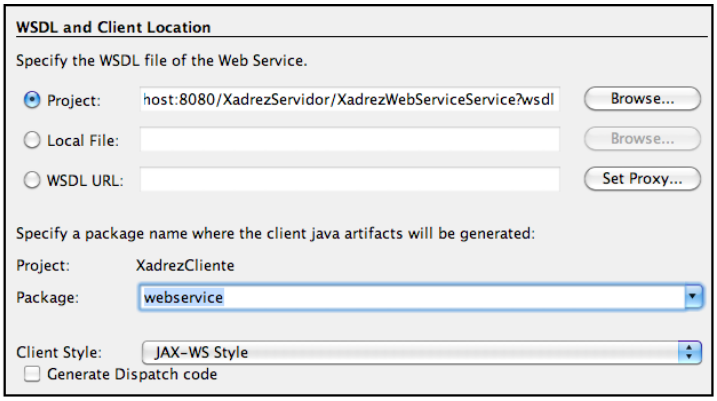


Figura 26 – Interface gráfica do JAX

Neste pacote, todos os métodos do *webservice* viram classes. Por exemplo: o método `cadastrarJogador` possui uma classe `CadastrarJogador` correspondente neste pacote. Caso o método possua algum retorno, é gerada uma classe `Response` correspondente. Por exemplo: o método `listarDesafios` retorna uma `String`, então, são criadas as classes `ListarDesafios` (Quadro 7) e `ListarDesafiosResponse` (Quadro 8).

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "listarDesafiosAtivos")
public class ListarDesafiosAtivos {


}
```

Quadro 7 - Classe `ListarDesafios`

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "listarDesafiosAtivosResponse", propOrder = {
    "_return"
})
public class ListarDesafiosAtivosResponse {

    @XmlElement(name = "return")
    protected String _return;

    /**
     * Gets the value of the return property.
     *
     * @return
     *     possible object is
     *     {@link String }
     *
     */
    public String getReturn() {
        return _return;
    }

    /**
     * Sets the value of the return property.
     *
     * @param value
     *     allowed object is
     *     {@link String }
     *
     */
    public void setReturn(String value) {
        this._return = value;
    }

}
```

Quadro 8 - Classe `ListarDesafiosResponse`

As classes `Response` são responsáveis por mapear os tipos de retorno das classes correspondentes através do elemento `@XmlElement`.

Dentro do pacote `webservice` são geradas as classes `XadrezWebServiceService` e `XadrezWebService`. A classe `XadrezWebServiceService` possui um mapeamento do arquivo WSDL e o método `getXadrezWebServicePort`, que retorna um `XadrezWebService`. Através deste método é chamado o objeto `XadrezWebService` que possui os mapeamentos

dos métodos do *webservice*, relacionando-os com as classes geradas.

3.3 IMPLEMENTAÇÃO

Para o usuário, o uso da aplicação é bastante simples, pois ele precisa apenas acessar a URL onde está localizado o arquivo JNLP através de qualquer navegador. Caso a JVM não esteja instalada corretamente, o *Java Web Start* iniciará o *download* e as instalações pendentes para que a aplicação possa ser utilizada.

3.3.1 Preparação do ambiente

Antes de acessar a aplicação, é necessário preparar o servidor corretamente para receber as conexões.

Primeiramente, é necessário habilitar o *Java Web Start* no projeto `XadrezCliente`. Para isto, nas propriedades do projeto, aba "Web Start", deve-se habilitar o botão "Enable Web Start" e selecionar a opção de comunicação via HTTP no campo "Codebase" (Figura 27). Em "Codebase Preview", deve-se informar o endereço completo de onde será colocado o arquivo JNLP no projeto servidor.

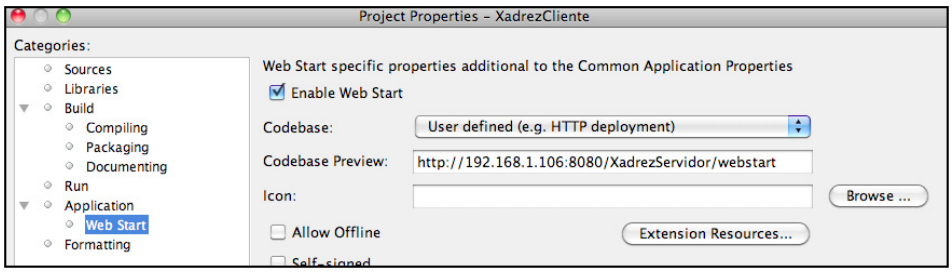


Figura 27 - Habilitando *Java Web Start*

Feito isso, o projeto pode receber o comando "*clean and built*", que vai gerar a pasta "dist" no diretório da aplicação cliente, contendo o arquivo JNLP. Nesta pasta estarão todos os arquivos necessários para a execução do JNLP (Figura 28).

Name	Date Modified	Size
launch.html	Today, 2:01 PM	4 KB
launch.jnlp	Today, 2:01 PM	4 KB
lib	Today, 2:01 PM	--
README.TXT	Today, 2:01 PM	4 KB
XadrezCliente.jar	Today, 2:01 PM	188 KB

Figura 28 - Arquivos gerados pelo *Java Web Start*

O arquivo `launch.html` é uma página *web* simples, gerada pelo *NetBeans* para acessar o arquivo JNLP. O arquivo `XadrezCliente.jar` é o responsável pela aplicação que será chamado pelo JNLP, uma vez que este funciona como um XML. A pasta lib contém as bibliotecas utilizadas pelo projeto e que precisam estar no mesmo diretório em que estavam durante o desenvolvimento.

Estes cinco arquivos devem ser copiados para uma pasta "webstart" dentro da pasta principal do projeto servidor que, no caso do *NetBeans*, é a pasta "web" (Figura 29).

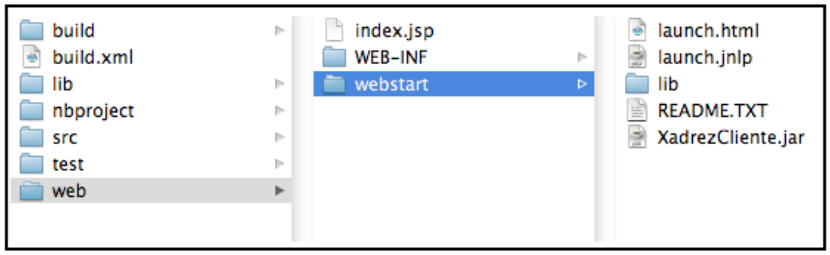


Figura 29 - Estrutura de pastas do servidor

Feito isso, o servidor pode ser inicializado. O *GlassFish* irá verificar se as portas estão abertas e mostrará que o `XadrezWebService` está pronto para receber requisições de clientes. Os serviços do *MySQL* também podem ser inicializados.

3.3.2 Iniciando a execução

A primeira tela criada é do tipo `LoginGUI`, que apresenta campos de texto para usuário e senha. Caso o jogador ainda não possua cadastro, existe um link para a `JDialog LoginDialog`, que efetua o cadastro do jogador.

Para cadastrar o jogador, o `LoginDialog` chama uma instância do objeto `ControleCliente`. Este chama o método de cadastro do *webservice*, estabelecendo a primeira comunicação com o servidor.

No método `cadastrarUsuario` (Quadro 9), depois de feitas algumas consistências, é criado um objeto do tipo `XadrezWebServiceService` que parametriza um objeto `XadrezWebService`. Este último faz a chamada dos métodos.

```
public void cadastrarUsuario(String nome, String login, String senha,
String reSenha, String email)
throws LoginException {
        if (login.equals("")) {
            throw new LoginException("Campo login não pode ser vazio!");
        }

        if (senha.equals("")) {
            throw new LoginException("Campo senha não pode ser vazio!");
        }

        if (email.equals("")) {
            throw new LoginException("Campo email não pode ser vazio!");
        }

        if (nome.equals("")) {
            throw new LoginException("Campo nome não pode ser vazio!");
        }

        if (!senha.equals(reSenha)) {
            throw new LoginException("Senhas estão diferentes!");
        }

        XadrezWebServiceService webservice = new XadrezWebServiceService();
        XadrezWebService remoto = webservice.getXadrezWebServicePort();

        if (remoto.loginJahExiste(login)) {
            throw new LoginException("Usuário já existe!");
        }

        if (remoto.jahExisteEmail(email)) {
            throw new LoginException("Este email já está cadastrado!");
        }

        remoto.cadastrarJogador(nome, login, senha, email);
    }
```

Quadro 9 - Método `cadastrarJogador` do `ControleCliente`

O método `efetuarLogin` faz uma consistência parecida com o `cadastrarUsuario`, verificando o objeto `Jogador` é trazido via XML do servidor. No caso de haver alguma exceção, significa que a `String` do XML veio incompleta, e o método retorna `null`. Dessa forma, é mandada uma mensagem ao usuário avisando que ou usuário ou senha estão incorretos.

No caso do *login* ser feito com sucesso, o atributo estático `jogadorLogado`, da classe `ControleCliente`, recebe o `Jogador` trazido no método `efetuarLogin`. Com este objeto, toda interação com o servidor poderá ser feita através do `idJogador`. Em seguida o atributo estático `PrincipalGUI`, que guarda a tela principal, é mostrado e a tela `LoginGUI`, escondida.

3.3.3 Sala principal

No método construtor da classe `PrincipalGUI`, são instanciados objetos dos tipos `DesafiosModel`, `PartidasModel` e `JogadoresConectadosModel`. Esses modelos serão

atribuídos a objetos `JTable`, mostrando os desafios disponíveis, as partidas em andamento e os jogadores conectados na sala de jogo.

Em seguida, ainda no construtor, objetos das classes `DesafiosThread`, `JogadoresThread` e `PartidasThread` são instanciados e os *threads* são inicializados. Através desta inicialização, os threads iniciam o *game-loop*, requisitando todas as atualizações ao servidor, através da classe `ControleCliente`. No Quadro 10, é mostrado o método `run()` da classe `JogadoresThread`, indicando que a cada segundo ele fará a requisição da lista de jogadores logados através do método `listarJogadoresLogados`, atribuirá o resultado ao `model` e fará aparecer na tela através do método `fireTableDataChanged`.

```
@Override
   public void run () {
       while (true) {
           try {
               model.setJogadores(ControleCliente.listarJogadoresLogados());
               model.fireTableDataChanged();
               sleep(1000);
           } catch (InterruptedException ex) {
               JOptionPane.showMessageDialog(null, "ERRO");
               Logger.getLogger(JogadoresThread.class.getName()).log(Level.SEVERE, null, ex);
           }
       }
   }
```

Quadro 10 - Método run da classe `JogadoresThread`

O método `listarJogadoresLogados` traz uma `String` do `ControleServidor` para que possa passar pelo *webservice*. No `ControleCliente`, ele é desmontado e um *array* de `Jogador` é criado (Quadro 11). Através do método `alias`, é informada qual a classe que será utilizada para interpretar o XML. O método deve ser `synchronized`, pois será utilizado por *threads* concorrentemente com outros métodos de listas e de verificação.

```
public static synchronized ArrayList<Jogador> listarJogadoresLogados() {
        XadrezWebServiceService webservice = new XadrezWebServiceService();
        XadrezWebService remoto = webservice.getXadrezWebServicePort();
        XStream arquivo = new XStream(new DomDriver());
        arquivo.alias("jogador", Jogador.class);
        ArrayList<Jogador> lista = (ArrayList<Jogador>)
arquivo.fromXML(remoto.listarJogadoresLogados());
        return lista;
    }
```

Quadro 11 - Método `listarJogadoresLogados` do `ControleCliente`

Na tela, a `JTable` que contém o `JogadoresConectadosModel` será atualizado (Figura 30).

Figura 30 - Jogadores "aaa" e "bbb" conectados

3.3.4 Desafios

Na tela `PrincipalGUI`, o *thread* `DesafiosThread` iniciará uma verificação atualizando a lista de desafios e verificando se houve algum desafio direcionado ao jogador. O comportamento deste thread está exemplificado na Figura 31.

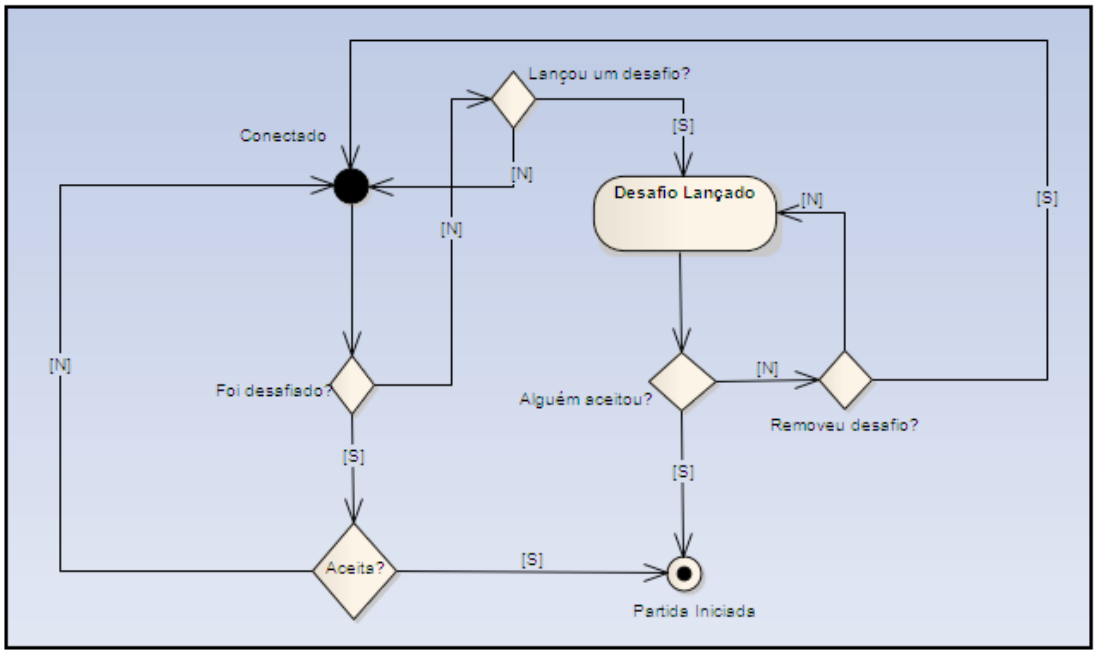


Figura 31 - Comportamento do *thread* `DesafiosThread`

Caso o jogador não tenha lançado nenhum desafio, o *thread* fica perguntando se alguém lançou um desafio endereçado a ele. Caso sim, ele mostra uma tela para que o usuário possa aceitar ou recusar. Se ele recusar, volta ao estado "Conectado".

Se o jogador lançar um desafio, entra em estado "Desafio Lançado" até que alguém o

aceite ou que ele o remova. Não é possível fazer mais de um desafio e caso o jogador feche a janela principal, o método `removerDesafio` é executado, para que não fique nenhum desafio preso.

Quando é aceito, o desafio é removido do *array* `desafios` e é cadastrada uma partida no *array* `partidas`, com os dados obtidos do desafio como tempo, quem jogará de brancas e pretas e qual o tipo do jogo.

3.3.5 Classe `ControleRegra`

A classe `ControleRegra` faz a verificação se o lance é legal ou não.

Cada peça possui um método privado apropriado, que fará a verificação se a casa correspondente ao segundo clique válido pode ser alcançada pela peça selecionada no primeiro clique. É considerado um clique válido qualquer seleção de peça da cor do jogador. Por exemplo: se o jogador que está conduzindo as peças brancas clicar numa peça preta ou numa casa vazia, este clique não será validado.

Para ser validado, um lance nunca pode deixar o próprio rei em xeque. Para isso, foi desenvolvido o método `estahEmXeque`, passando como parâmetro a posição atual e a cor que será verificada (Quadro 12).

```
public boolean estahEmXeque(Peca[][] pecas, int cor) {
        int linhaDoRei = 0;
        int colunaDoRei = 0;
        int corAdversaria = -1;

        if (cor == 1) {
            corAdversaria = 2;
        }

        if (cor == 2) {
            corAdversaria = 1;
        }

        for (int i = 0; i < 8; i++) {
            for (int j = 0; j < 8; j++) {
                Peca p = pecas[i][j];
                if ((p.getNome() == 'r' || p.getNome() == 'R') && p.getCor() == cor) {
                    linhaDoRei = i;
                    colunaDoRei = j;
                }
            }
        }

        for (int i = 0; i < 8; i++) {
            for (int j = 0; j < 8; j++) {
                if (pecas[i][j] != null) {
                    Peca p = pecas[i][j];
                    if (p.getCor() == corAdversaria) {
                        if ((p.getNome() == 'C' || p.getNome() == 'c') && podeMoverCavalo(pecas, i, j, linhaDoRei, colunaDoRei) ||
                            (p.getNome() == 'B' || p.getNome() == 'b') && podeMoverBispo(pecas, i, j, linhaDoRei, colunaDoRei)  ||
                            (p.getNome() == 'D' || p.getNome() == 'd') && podeMoverDama(pecas, i, j, linhaDoRei, colunaDoRei)   ||
                            (p.getNome() == 'T' || p.getNome() == 't') && podeMoverTorre(pecas, i, j, linhaDoRei, colunaDoRei)  ||
                            (p.getNome() == 'P' || p.getNome() == 'p') && podeMoverPeao(pecas, i, j, linhaDoRei, colunaDoRei)) {
                            return true;
                        }
                    }
                }
            }
        }
        return false;
    }
```

Quadro 12 - Método `estahEmXeque`

O primeiro objetivo do método é localizar a posição onde o rei está. As variáveis

`linhaDoRei` e `colunaDoRei` guardarão a posição do rei da cor passada por parâmetro, fazendo uma busca completa no *array* bidimensional do tabuleiro.

Quando o rei é localizado, é feita outra busca completa no *array*, verificando se a peça é de cor diferente da do rei encontrado. Caso seja, verifica qual peça é e chama o método `podeMover` referente à ela, passando como casa de destino as coordenadas do rei. Se o método retornar `true`, significa que uma peça adversária pode ir para a casa do rei, o que caracteriza uma posição de xeque.

Para a movimentação do bispo, foi utilizada uma lógica de soma de valores iguais para linhas e colunas, já que ele só anda em diagonal (Quadro 13).

```
int x;
int y;
if (linhaFinal < linhaInicial && colunaFinal < colunaInicial) {
    x = linhaInicial - 1;
    y = colunaInicial - 1;
    while (x != linhaFinal || y != colunaFinal) {
        if (pecas[x][y].getCor() != 0) {
            return false;
        }
        x--;
        y--;
    }
}

if (linhaFinal < linhaInicial && colunaFinal > colunaInicial) {
    x = linhaInicial - 1;
    y = colunaInicial + 1;
    while (x != linhaFinal || y != colunaFinal) {
        if (pecas[x][y].getCor() != 0) {
            return false;
        }
        x--;
        y++;
    }
}

if (linhaFinal > linhaInicial && colunaFinal < colunaInicial) {
    x = linhaInicial + 1;
    y = colunaInicial - 1;
    while (x != linhaFinal || y != colunaFinal) {
        if (pecas[x][y].getCor() != 0) {
            return false;
        }
        x++;
        y--;
    }
}
```

Quadro 13 - Trecho do método `podeMoverBispo`

Para a movimentação da torre, é verificado se a `colunaInicial` é igual à `colunaFinal` ou a `linhaFinal` é igual à `linhaInicial`, pois esta peça só anda em linha reta. Também é verificado se todas as casas no caminho estão vazias (Quadro 14).

```
int x;
int y;
if (linhaFinal < linhaInicial && colunaFinal < colunaInicial) {
    x = linhaInicial - 1;
    y = colunaInicial - 1;
    while (x != linhaFinal || y != colunaFinal) {
        if (pecas[x][y].getCor() != 0) {
            return false;
        }
        x--;
        y--;
    }
}

if (linhaFinal < linhaInicial && colunaFinal > colunaInicial) {
    x = linhaInicial - 1;
    y = colunaInicial + 1;
    while (x != linhaFinal || y != colunaFinal) {
        if (pecas[x][y].getCor() != 0) {
            return false;
        }
        x--;
        y++;
    }
}

if (linhaFinal > linhaInicial && colunaFinal < colunaInicial) {
    x = linhaInicial + 1;
    y = colunaInicial - 1;
    while (x != linhaFinal || y != colunaFinal) {
        if (pecas[x][y].getCor() != 0) {
            return false;
        }
        x++;
        y--;
    }
}
```

Quadro 14 - Trecho do método `podeMoverTorre`

Como a dama possui os mesmos movimentos da torre ou do bispo, o método `podeMoverDama` chama os dois métodos e verifica se um deles retorna `true` para validar o lance.

O controle do movimento em "L" do cavalo é ilustrado no Quadro 15. As somas de 1 pra uma direção e 2 pra outra são para formar o "L".

```
private boolean podeMoverCavalo(Peca[][] pecas, int linhaInicial,
        int colunaInicial, int linhaFinal, int colunaFinal) {
    if (pecas[linhaInicial][colunaInicial].getCor() == pecas[linhaFinal][colunaFinal].getCor()) {
        return false;
    }

    if ((linhaFinal == linhaInicial + 1 || linhaFinal == linhaInicial - 1)
     && (colunaFinal == colunaInicial + 2 || colunaFinal == colunaInicial - 2)) {
        return true;
    }

    if ((linhaFinal == linhaInicial + 2 || linhaFinal == linhaInicial - 2)
    && (colunaFinal == colunaInicial + 1 || colunaFinal == colunaInicial - 1)) {
        return true;
    }
    return false;
}
```

Quadro 15 - Método `podeMoverCavalo`

O método `podeMoverRei` controla que a `linhaFinal` e a `colunaFinal` estejam a apenas uma casa da casa inicial (Quadro 16).

```
private boolean podeMoverRei(Peca[][] pecas, int linhaInicial,
         int colunaInicial, int linhaFinal, int colunaFinal) {
     if (pecas[linhaInicial][colunaInicial].getCor() == pecas[linhaFinal][colunaFinal].getCor()) {
         return false;
     }


     if ((linhaFinal == linhaInicial && colunaFinal == colunaInicial - 1)
             || (linhaFinal == linhaInicial - 1 && colunaFinal == colunaInicial - 1)
             || (linhaFinal == linhaInicial - 1 && colunaFinal == colunaInicial)
             || (linhaFinal == linhaInicial - 1 && colunaFinal == colunaInicial + 1)
             || (linhaFinal == linhaInicial && colunaFinal == colunaInicial + 1)
             || (linhaFinal == linhaInicial + 1 && colunaFinal == colunaInicial + 1)
             || (linhaFinal == linhaInicial + 1 && colunaFinal == colunaInicial)
             || (linhaFinal == linhaInicial + 1 && colunaFinal == colunaInicial - 1)) {
         return true;
     }
     return false;
 }
```

Quadro 16 - Método `podeMoverRei`

Depois que a jogada foi validade, é executado o método `gerarPgn` (Quadro 3.12), que irá gerar uma `String` a ser concatenada na `String` existente no servidor, na partida correspondente. No final da partida, a `String pgn` poderá ser baixada pelos dois jogadores e ser colocada em programas com *engines* de análise.

3.3.6 Partidas

Quando o jogador começa a partida e efetua um lance, o método `efetuarLance` da classe `ControleCliente` irá chamar o *webservice* e executar o método homônimo na classe `ControleServidor`. Neste método é passada a posição pronta no formato `String` que será atribuída à partida referenciada no servidor. O diagrama de sequência da Figura 32 mostra o passo a passo da execução de um lance por parte do jogador até o `ControleCliente`.

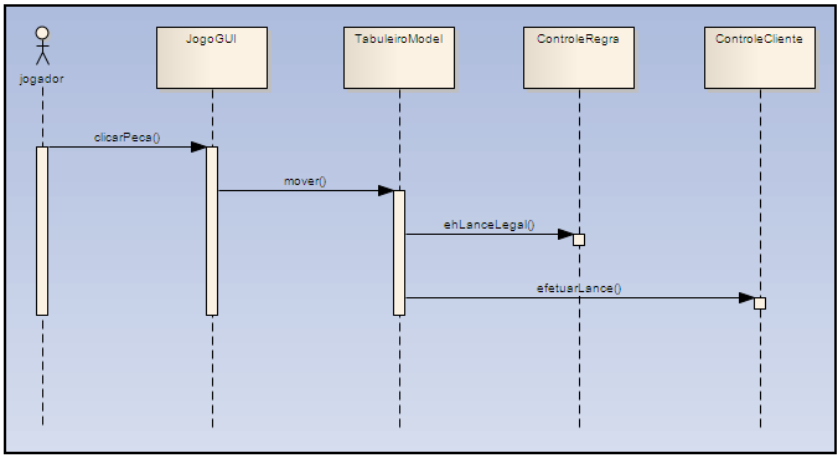


Figura 32 - Processo `efetuarLance`

O *thread* `TabuleiroThread` executa o método `atualizarPartida` que busca um objeto `Partida` serializado do servidor e verifica de quem é a vez, além de atualizar os tempos dos jogadores.

3.3.7 Banco de dados

O banco de dados possui três tabelas: a tabela `Jogador`, a tabela `Rating` e a tabela `Partida` (Figura 33).

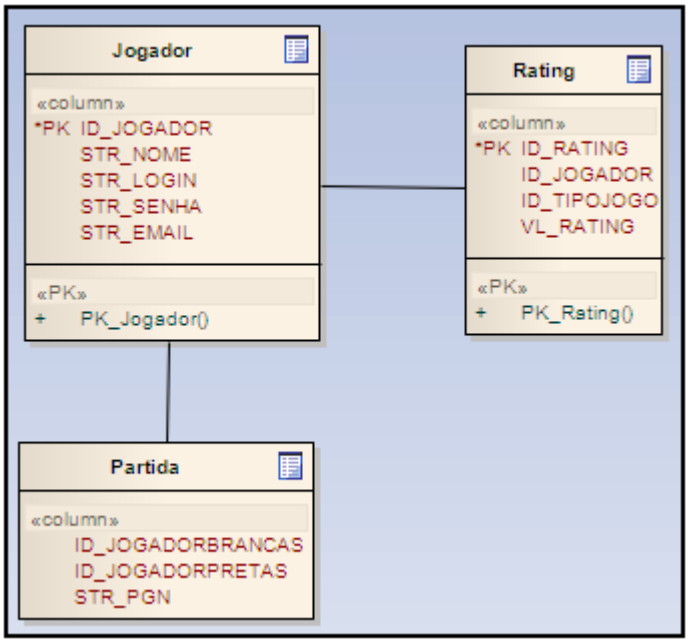


Figura 33 - Tabelas do banco de dados

A tabela `Jogador` guarda o nome, *login*, senha e e-mail do usuário, todos em tipo `VARCHAR`.

A tabela `Rating` guarda qual a pontuação do jogador em cada tipo de jogo. Cada usuário possui quatro *ratings*, compatíveis com os quatro jogos que o ambiente disponibiliza.

A tabela `Partida` guarda as partidas em formato PGN, associada aos `ID_JOGADOR` dos dois jogadores.

3.3.8 Operacionalidade da implementação

O ambiente possui cinco telas: a de cadastro, *login*, principal, de lançamento de desafios e a tela de jogo.

Na tela de cadastro, é pedido o nome, o *login*, senha duas vezes e e-mail (Figura 34).

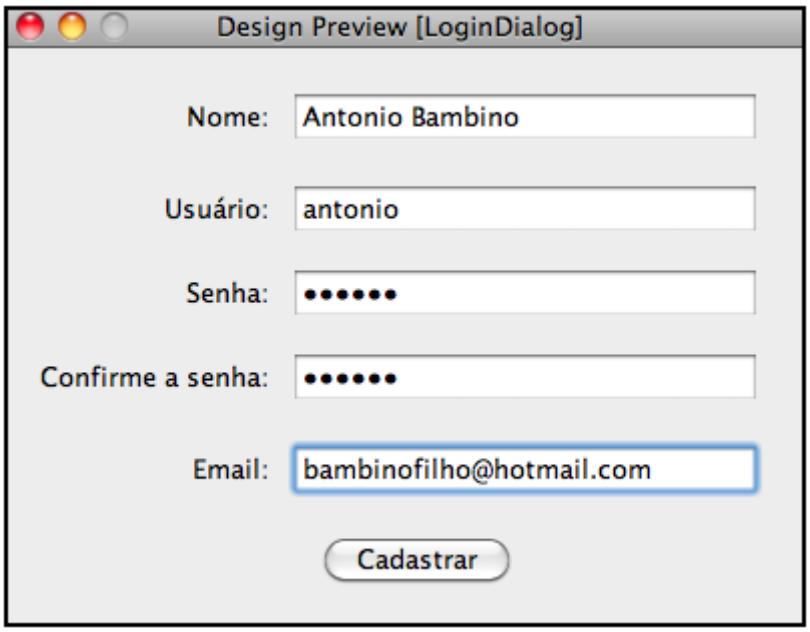


Figura 34 - Tela de cadastro

Nenhum dos campos pode ser vazio, e os dois campos de senha devem estar iguais. Caso o *login* ou o e-mail já existam, aparecerá uma mensagem pedindo que o usuário cadastre outro. O cadastro é finalizado quando o botão "Cadastrar" é apertado ou quando a tecla "Enter" é apertada em qualquer um dos campos.

Terminado o cadastro, o jogador pode efetuar o *login* com o usuário criado (Figura 35).

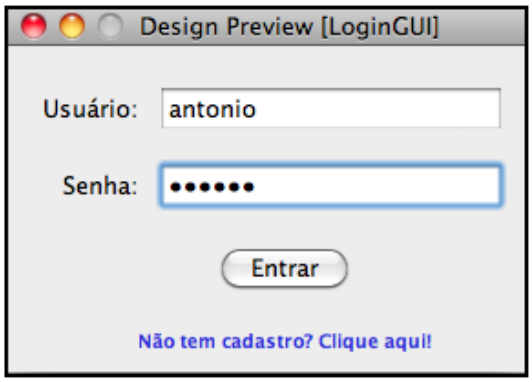


Figura 35 - Tela de *login*

Caso o *login* seja validado, a tela se fechará e será aberta a tela principal do programa, que mostrará o gráfico de desafios, os jogadores conectados e as partidas em andamento (Figura 36).

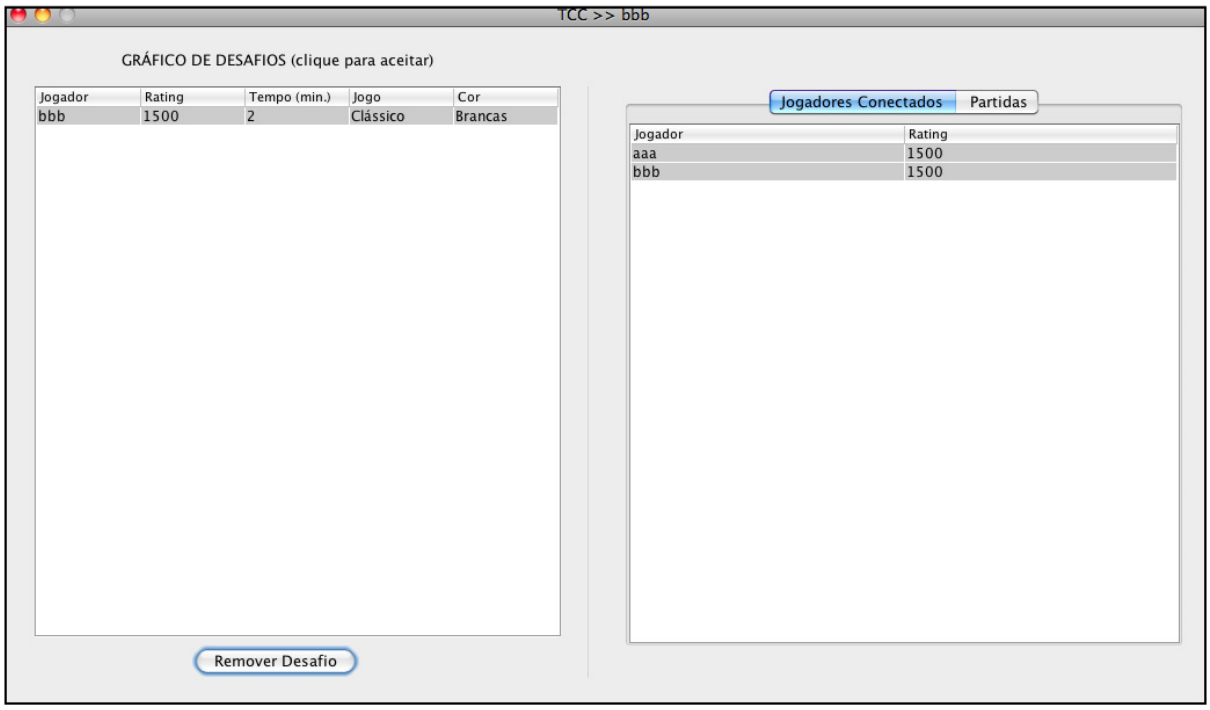


Figura 36 - Tela principal

Para acompanhar uma partida em andamento é necessário apenas clicar uma vez sobre a partida. Feito isso, a tela do jogo será aberta e o usuário poderá ver o PGN do jogo e os jogadores conectados à partida além dele.

Para lançar um desafio, o usuário deve clicar no botão "Lançar Desafio" para abrir a tela de parametrização (Figura 37).

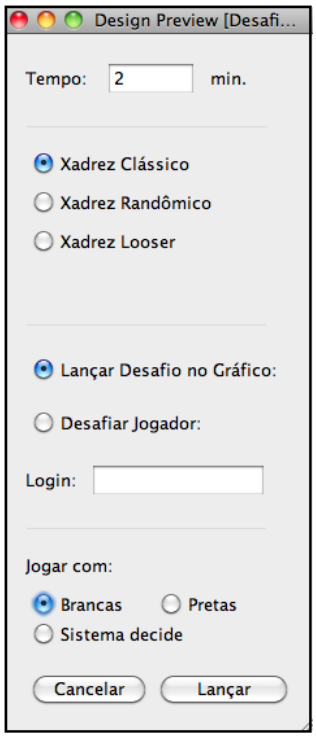


Figura 37 - Tela de parametrização do desafio

O usuário deve informar o tempo, a modalidade de xadrez que deseja jogar e a cor. Feito isso, o desafio será lançado no "Gráfico de Desafios" e estará sujeito a ser aceito por

qualquer jogador. Para desafiar um jogador específico, o usuário deve clicar com o botão direito no jogador na lista de jogadores conectados e depois em desafiar. A mesma tela de parametrização será aberta. Para aceitar um desafio, será aberta uma janela estilo *pop-up* na tela do jogador, com as opções "Aceitar" e "Recusar".

Ao iniciar uma partida, o jogador irá para a tela de jogo (Figura 38).

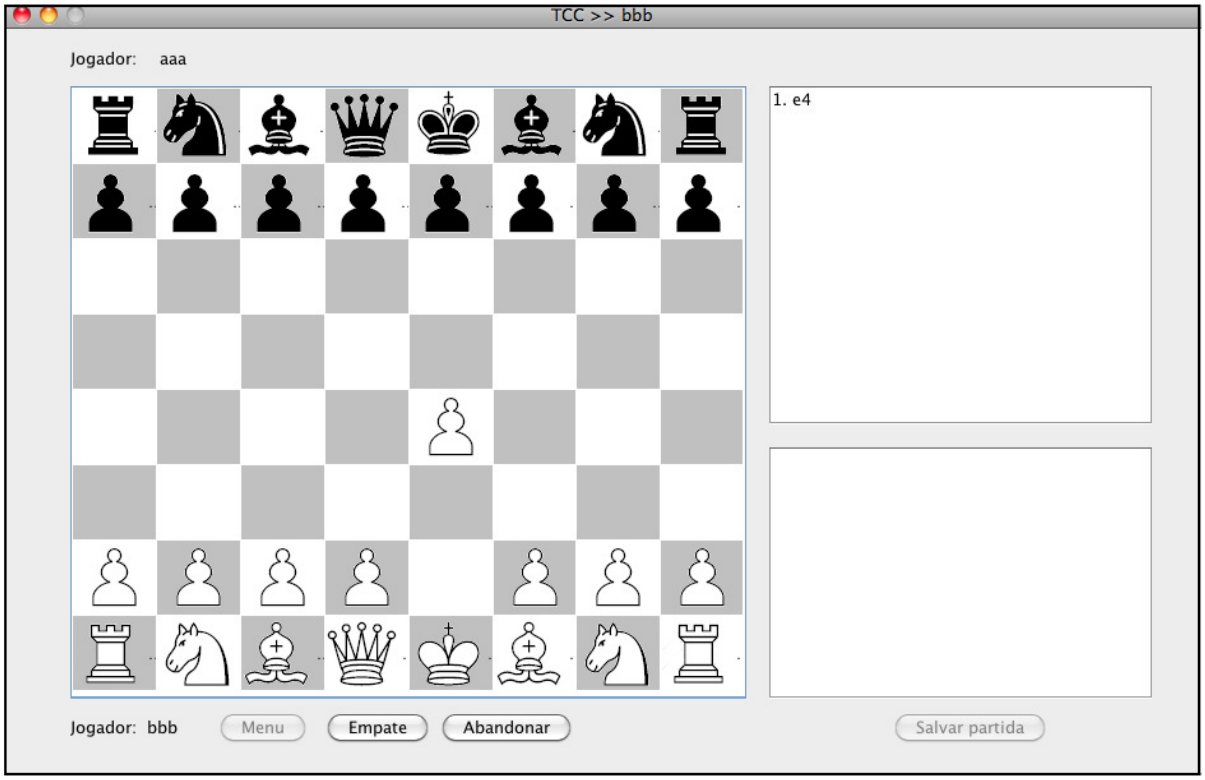


Figura 38 - Tela de jogo

Os botões "Abandonar" e "Empate" estarão habilitados no início do jogo. Quando o usuário clicar em abandonar, o jogo será terminado. Quando clicar em "Empate", uma janela estilo pop-up aparecerá para o outro jogador, com as opções de "Aceitar" ou "Recusar".

Para efetuar um lance, o jogador deve clicar primeiro na peça que deseja movimentar depois na casa de destino. Caso clique na peça e depois queira jogar outra, pode clicar novamente na peça ou em qualquer casa que caracterize um lance ilegal.

Os botões "Revanche" e "Menu" serão habilitados no fim da partida. Caso um dos jogadores clique em "Menu", os dois são redirecionados para a tela principal novamente.

A área "Partida" mostrará o PGN da partida até o momento, que poderá ser baixado no término da partida. A área de "Jogadores Conectados" mostra, além dos dois jogadores da partida, qualquer outro jogador que estiver assistindo a partida.

3.4 RESULTADOS E DISCUSSÃO

Os testes realizados foram satisfatórios, apesar do alto consumo de processamento dos threads.

As regras do xadrez foram testadas de várias formas e o sistema foi executado em vários sistemas operacionais, respondendo a todos com sucesso.

O sistema torna-se lento quando colocado em um servidor com IP fixo, por isso, os testes foram todos executados por clientes em uma mesma rede.

Na parte gráfica não houve nenhuma perda em relação aos principais sistemas, pois as peças do servidor são passadas por XML e não há perda de nitidez.

As tecnologias foram integradas através do *NetBeans*, uma vez que o JAX gerou as os arquivos WSDL e o arquivo JNLP, que, colocado na pasta *webstart*, funcionou com o *GlassFish*. O MySQL funcionou com o servidor, uma vez que os acessos eram feitos por esse apenas.

Para o *webservice*, a serialização dos objetos funcionou, para que tipos abstratos de dados fossem passados através de *Strings*.

O principal diferencial do sistema perante os trabalhos correlatos é o fato de poder fazer o download do PGN da partida após o término.

4 CONCLUSÕES

Este trabalho apresentou um ambiente para o jogo de xadrez com a utilização de *webservices, Java Web Start* e *threads*, sendo apoiados por um servidor *GlassFish* e um banco de dados MySQL.

O *webservice* foi desenvolvido com o auxílio da API JAX, que fez com que o autor não precisasse se preocupar com a geração dos arquivos WSDL nem com o protocolo SOAP, pois o JAX gera isso automaticamente.

O *Java Web Start* permitiu que o projeto fizesse uso da linguagem Java sem nenhum tipo de problema ou perda de desempenho. Isto é uma vantagem perante outros projetos, pois a linguagem Java é mais específica para modelagem de objetos do que linguagens mais limitadas para internet.

Com o uso do *NetBeans*, tanto o gerenciamento do *Java Web Start* quanto do JAX foram facilitados, uma vez que a ferramenta dispõe de auxílios visuais no estilo *drag and drop*. Também o servidor *GlassFish* está contido na ferramenta, evitando que o desenvolvedor precise acessar linhas de comando para executar qualquer operação.

Os jogos estão implementados em sua totalidade e com suas regras oficiais. Para o xadrez randômico, foram testados vários sorteios de peças iniciais para verificar a regra dos bispos em casas diferentes, todos com sucesso. O xadrez mata-mata tinha a diferença de obrigar o jogador a fazer uma captura se ela fosse possível, além de controlar o número de peças no tabuleiro caso não haja nenhum lance legal a ser feito.

A maior limitação do projeto é precisar da máquina virtual do Java instalada para poder rodá-lo. Quem tiver uma máquina com pouca memória ou espaço no disco, ou mesmo não tiver muita experiência na instalação pode ter alguma dificuldade.

Outra limitação foi o consumo excessivo de processamento dos threads. Como o projeto utiliza um *game-loop*, uma máquina com processador um pouco mais antigo pode ter problemas.

O ideal para a implementação do *webservice* seria a utilização de um padrão de projetos *Observer* em vez do uso dos *threads*. No entanto, é necessário achar uma tecnologia de *webservice* que implemente este padrão.

4.1 EXTENSÕES

Um trabalho possível é implementar uma *engine* para que os usuários consigam jogar contra computadores. A *engine* pode consumir o *webservice* da mesma forma que o projeto cliente.

Outra proposta é implementar o padrão de projeto *Observer*, fazendo com que os threads não precisem ficar requisitando mudanças no servidor, mas sim que o servidor avise aos clientes registrados na lista de *Observers* a hora que houver alguma mudança.

REFERÊNCIAS BIBLIOGRÁFICAS

ADOBE. **Benefícios das RIAs (aplicações ricas para internet)**. [S.l.], 2009. Disponível em: <http://www.adobe.com/br/resources/business/rich_internet_apps/benefits/>. Acesso em: 15 ago. 2011.

ALBUQUERQUE, Fernando. **Programação distribuída usando java**. Brasília, 2006. Disponível em: <http://www.cic.unb.br/~fernando/matdidatico/textosintro/texto04.pdf>. Acesso em: 04 abr. 2011.

ALCÂNTARA, Andréia. **O que são threads?** [S.l.], 1996. Disponível em: <http://www.di.ufpe.br/~java/verao/aula8/definicao.html>. Acesso em: 13 ago. 2011.

ALECRIM, Emerson. **Conhecendo o servidor Apache (HTTP Server Project)**. [S.l.], 2006. Disponível em: <http://www.infowester.com/servapach.php>. Acesso em: 11 set. 2010.

AURÉLIO, Marco. **Como funciona um servidor web**. [S.l.], 2005. Disponível em <http://www.malima.com.br/article_read.asp?id=156>. Acesso em: 15 ago. 2011.

BASHA, Jeelani et al. **Profissional java web services**. Tradução: Marcus Rolo. Rio de Janeiro: Alta Books, 2002.

BATISTA, Gerson. **Como criar um arquivo PGN**. [S.l.], 2001. Disponível em: <http://www.clubedexadrezonline.com.br/artigo.asp?doc=872>. Acesso em: 15 ago. 2011.

BECKER, Idel. **Manual de xadrez**. São Paulo: Nobel, 2002.

BRAIN, Marshall. **Como funcionam os servidores web**. [S.l.], [1999?]. Disponível em: <http://informatica.hsw.uol.com.br/servidores-da-web1.htm>. Acesso em: 15 ago. 2011.

BUHO21. [S.l.], [2001?]. Disponível em: <www.buho21.com>. Acesso em: 28 ago. 2011.

CARNEIRO, **Luís. Programação para redes de computadores**. [S.l.], 2007. Disponível em: <http://www.slideshare.net/liusfc/sistemas-distribudos-rmi-corba-e-soa>. Acesso em: 23 out. 2011.

CHESSFORALLAGES. **Portable game notation**. [S.l.], [2001?]. Acesso em: <http://www.mark-weeks.com/aboutcom/aa02j12.htm>. Acesso em: 13 nov. 2011.

CHESSVARIANTS. **Losing chess**. [S.l.], 2005. Disponível em: <http://www.chessvariants.com/diffobjective.dir/giveaway.html>. Acesso em: 24 set. 2011.

CRANE, Dave; JAMES, Darren; PASCARELLO, Eric. **Ajax em ação**. Birmingham: Manning, 2007.

DANTAS, Daniel. **Simple object access protocol**. Rio de Janeiro, 2007. Disponível em: <http://www.gta.ufrj.br/grad/07_2/daniel/index.html>. Acesso em: 01 out. 2011.

D'AGOSTINI, Orfeu. **Xadrez básico**. Rio de Janeiro: Ediouro, 1998.

DEITEL, Harvey; DEITEL, Paul. **Ajax, rich internet applications e desenvolvimento web para programadores**. São Paulo: Prentice Hall, 2008.

GUEDES, Luiz A. **Sistemas distribuídos**. [S.l.], 2004. Disponível em: <www.dca.ufrn.br/~affonso/DCA2401/2004_1/aulas/threads.ppt>. Acesso em: 3 abr. 2011.

ICC. **The internet chess club**. [S.l.], 1996. Disponível em: <www.chessclub.com>. Acesso em: 29 ago. 2011.

IXC. **Internet xadrez clube**. Porto Alegre, [1999?]. Disponível em: <www.ixc.com.br>. Acesso em: 20 out. 2011.

LONGO, João S. **Glassfish**: o servidor de aplicações para todos os seus aplicativos web. [S.l.], 2009. Disponível em: <http://www.slideshare.net/joaosavio/glassfish-2311747>. Acesso em: 24 set. 2011.

MARCELINO, Leonardo. **Java web start**. [S.l.], [2002?]. Disponível em: <http://www.guj.com.br/content/articles/jws/jws.pdf>. Acesso em: 22 out. 2011.

MARLBORO, Alberto. Buho21. [S.l.], 2009. Disponível em: <http://www.caissacafe.com.br/artigos/33-buho21.html>. Acesso em: 20 out. 2011.

MORIMOTO, Carlos. **Threads**. [S.l.], 2005. Disponível em: <http://www.hardware.com.br/termos/thread>. Acesso em: 23 out. 2011.

MOUSSINE, Alexis; PELEGRI, Eduardo; YOSHIDA, Yutaka. **The GlassFish community delivering a Java EE application server**. [S.l.], 2007. Disponível em: <https://glassfish.dev.java.net/faq/v2/GlassFishOverview.pdf>. Acesso em: 24 set. 2010>.

MYCHESS. **Laws of chess**: chess960. [S.l.], 2003. Disponível em: <http://www.mychess.de/Chess960.htm>. Acesso em: 02 set. 2011.

NISHA, Hirat; SNESHA, Sami. **Implementing thread program**. [S.l.], 2009. Disponível em: <http://software.intel.com/en-us/articles/implementing-thread-program>. Acesso em: 04 out. 2011.

NUNES, Pedro. **WSDL e UDDI**. Lisboa, 2006. Disponível em: <http://www.gsd.inesc-id.pt/~ler/docencia/tm0607/slides/WSDL-UDDI-PedroNunes.pdf>. Acesso em: 04 set. 2011.

OLIVEIRA, Eric. **Tecnologia java web start**. São Paulo, 2005. Disponível em: <http://www.linhadecodigo.com.br/Artigo.aspx?id=627>. Acesso em: 31 ago. 2011.

ORACLE. **JNLP file syntax**. [S.l.], 2004. Disponível em: <http://download.oracle.com/javase/6/docs/technotes/guides/javaws/developersguide/syntax.html> Acesso em: 31 ago. 2011.

_____. **O que é o java web start e como ele é iniciado?** [S.l.], 2006. Disponível em: <http://www.java.com/pt_BR/download/faq/java_webstart.xml>. Acesso em: 07 mar. 2011.

POUCHKINE, Alexis. **GlassFish**: delivering a better application server one step at a time. [S.l.], 2007. Disponível em: <http://www4.java.no/presentations/javazone/2007/slides/5575.pdf>. Acesso em: 03 nov. 2010.

SCIMIA, Edward. **How to play chess960**. [S.l.], 2002. Disponível em: <http://chess.about.com/od/chessvariants/a/Chess960.htm>. Acesso em: 01 set. 2011.

SILVA, Vitor A. da. **O game loop**. [S.l.], 2007. Disponível em: <http://www.vsoftgames.com/site/articles/view/18>. Acesso em: 10 ago. 2011.

SOBRAL, João. **Servidores de aplicação web**. Florianópolis, [2001?]. Disponível em: <www.inf.ufsc.br/~bosco/old_page/downloads/Servidores.ppt>. Acesso em: 29 ago. 2011.

SOL, Paloma. **Aplicações desktop via web**: java web start na prática. [S.l.], 2001. Disponível em: <http://www.devmedia.com.br/articles/viewcomp.asp?comp=10265>. Acesso em: 03 abr. 2011.

SPÍNOLA, Eduardo O. **Desenvolvendo web services utilizando JAX-WS**. [S.l.], 2010. Disponível em: <http://www.devmedia.com.br/post-2374-Desenvolvendo-Web-Services-utilizando-JAX-WS.html>. Acesso em: 21 set. 2010.

TOPLEY, Kim. **Java web services in a nutshell**. Sebastopol: O'Reilly, 2003.

W3C. **Web services description language**. [S.l.], 2001. Disponível em: <http://www.w3.org/TR/wsdl>. Acesso em: 01 out. 2011.

APÊNDICE A – Explicação dos métodos referentes aos casos de uso do usuário

Caso de uso – Cadastrar Jogador

Ator: Jogador

Objetivo: Efetuar o cadastro de um usuário no sistema

Pré-condições: Possuir máquina virtual Java instalada

Pós-condições: Usuário cadastrado no sistema

Cenário Principal:

1. Usuário informa o nome, login, senha e e-mail
2. Sistema faz o cadastro no banco de dados

Cenário Alternativo:

1. Caso o usuário não informe todos os dados no item 1, o sistema emitirá uma mensagem de alerta de que o cadastro não foi validado

Exceção:

1. Caso o login informado não esteja cadastrado, o sistema lançará uma exceção e solicitará um novo login ao usuário
2. Caso o e-mail informado não esteja cadastrado, o sistema lançará uma exceção e solicitará um novo e-mail ao usuário
3. Caso a senha não seja digitada duas vezes no item 1, o sistema lançará uma exceção e exibirá uma mensagem de erro, pedindo que o usuário digite as senhas novamente

Quadro 17 – Descrição do caso de uso "Cadastrar jogador"

Caso de uso – Efetuar Login

Ator: Jogador

Objetivo: Efetuar o login no sistema

Pré-condições: Possuir cadastro

Pós-condições: Acesso às principais funcionalidades do sistema

Cenário Principal:

1. Usuário informa login e senha
2. Sistema verifica a validade dos dados no banco de dados
3. Sistema habilita a janela principal

Cenário Alternativo:

1. Caso o usuário não informe todos os dados no item 1, o sistema emitirá uma mensagem de alerta de que os campos devem ser preenchidos

Exceção:

1. Caso login e senha não coincidam com os que constam no banco de dados, o sistema emitirá uma mensagem de alerta informando que ou login ou senha estão errados

Quadro 18 – Descrição do caso de uso "Efetuar login"

Caso de uso - Efetuar Logout

Ator: Jogador

Objetivo: Efetuar o logout do usuário do sistema

Pré-condições: Estar logado no sistema

Pós-condições: Fechamento do sistema

Cenário Principal:

1. Usuário fecha qualquer janela do sistema
2. Sistema faz a remoção do usuário da lista de usuários logados

Quadro 19 - Descrição do caso de uso "Efetuar logout"

Caso de uso - Lançar desafio público

Ator: Jogador

Objetivo: Lançar um desafio que possa ser aceito por qualquer outro jogador

Pré-condições: Estar logado, não estar jogando uma partida e não ter nenhum outro desafio lançado

Pós-condições: Desafio adicionado na lista de desafios públicos

Cenário Principal:

1. Usuário informa o tempo, tipo de jogo e a cor que deseja jogar
2. Sistema insere um novo desafio na lista de desafios públicos
3. Sistema renderiza a tela com o novo desafio

Cenário Alternativo:

1. Caso o usuário informe um tempo maior do que 15 ou menor do que 0, uma mensagem de alerta informará ao usuário de que o tempo deve ser maior do que zero e menor ou igual a 15.

Exceção:

1. Caso o valor informado no tempo não seja numérico, uma exceção será lançada para informar ao usuário que o tempo deve ser um valor numérico

Quadro 20 – Descrição do caso de uso "Lançar desafio público"

Caso de uso - Lançar desafio a outro jogador

Ator: Jogador

Objetivo: Lançar um desafio para um jogador especificamente

Pré-condições: Estar logado, não estar jogando uma partida, não ter nenhum outro desafio lançado e o outro jogador estar logado

Pós-condições: Desafio adicionado na lista de desafios nominais

Cenário Principal:

1. Usuário informa o tempo, tipo de jogo e a cor que deseja jogar
2. Usuário informa o login do jogador que deseja desafiar
3. Sistema insere um novo desafio na lista de desafios nominais
4. Sistema informa ao outro jogador que um desafio foi feito a ele

Cenário Alternativo:

1. Caso o usuário informe um tempo maior do que 15 ou menor do que 0, uma mensagem de alerta informará ao usuário de que o tempo deve ser maior do que zero e menor ou igual a 15

Exceção:

1. Caso o valor informado no tempo não seja numérico, uma exceção será lançada para informar ao usuário que o tempo deve ser um valor numérico
2. Caso o login do jogador desafiado não esteja na lista de jogadores logados, o sistema envia um alerta ao usuário de que o outro jogador não está logado no sistema

Quadro 21 – Descrição do caso de uso "Lançar desafio a outro jogador"

Caso de uso – Oferecer empate **Ator:** Jogador **Objetivo:** Oferecer empate na partida corrente **Pré-condições:** Estar com uma partida em andamento **Pós-condições:** Empate ofertado ao adversário **Cenário Principal:** 1. Usuário clica no botão "Empate", oferecendo empate ao outro jogador 2. Sistema informa ao outro jogador que uma oferta de empate foi feita **Cenário Alternativo:** 1. Caso o adversário aceite o empate, a partida termina e é removida da lista de partidas 2. Caso o adversário recuse o empate, a partida continua

Quadro 22 - Descrição do caso de uso "Oferecer empate"

Caso de uso – Oferecer revanche **Ator:** Jogador **Objetivo:** Oferecer revanche na partida corrente **Pré-condições:** Ter acabado de jogar a partida **Pós-condições:** Revanche ofertada ao adversário **Cenário Principal:** 1. Usuário clica no botão "Revanche", oferecendo revanche ao outro jogador 2. Sistema informa ao outro jogador que uma oferta de revanche foi feita **Cenário Alternativo:** 1. Caso o adversário aceite a revanche, uma nova partida é iniciada, com cores invertidas 2. Caso o adversário recuse a revanche, a situação permanece a mesma

Quadro 23 - Descrição do caso de uso "Oferecer revanche"

Caso de uso – Abandonar partida **Ator:** Jogador **Objetivo:** Desistir da partida corrente **Pré-condições:** Estar jogando uma partida **Pós-condições:** Partida finalizada **Cenário Principal:** 1. Usuário clica no botão "Abandonar" 2. Sistema finaliza a partida

Quadro 24 - Descrição do caso de uso "Abandonar partida"

```
Caso de uso - Assistir partida

Ator: Jogador

Objetivo: Assistir a uma partida corrente

Pré-condições: Estar logado

Pós-condições: Partida selecionada aberta em uma nova tela

Cenário Principal:

1. Usuário seleciona a partida que deseja assistir na lista de
partidas
2. Sistema busca a partida no servidor
3. Sistema habilita uma nova tela com a partida em andamento
```

Quadro 25 - Descrição do caso de uso “Assistir partida”

```
Caso de uso - Salvar partida

Ator: Jogador

Objetivo: Salvar a partida que acabou de ser jogada

Pré-condições: Ter acabado de terminar uma partida

Pós-condições: Arquivo em formato PGN com a partida

Cenário Principal:

1. Usuário clica no botão "Salvar"
2. Usuário seleciona o local onde deseja salvar o arquivo
3. Sistema gera um arquivo PGN com a partida jogada
```

Quadro 26 - Descrição do caso de uso “Salvar partida”

```
Caso de uso - Aceitar empate

Ator: Jogador

Objetivo: Aceitar uma oferta de empate

Pré-condições: Estar jogando uma partida e o outro jogador ter
oferecido empate

Pós-condições: Empate aceito ou recusado

Cenário Principal:

1. Sistema informa tela com opções "Aceito" ou "Recuso" ao usuário
2. Usuário seleciona a opção desejada

Cenário Alternativo:

1. No item 1, se o jogador aceitar o empate, a partida termina
2. No item 1, se o jogador recusar o empate, a partida continua
```

Quadro 27 - Descrição do caso de uso “Aceitar empate”

Caso de uso – Aceitar revanche

Ator: Jogador

Objetivo: Aceitar uma oferta de revanche

Pré-condições: Ter acabado de terminar uma partida

Pós-condições: Revanche aceita ou recusada

Cenário Principal:

1. Sistema informa tela com opções "Aceito" ou "Recuso" ao usuário
2. Usuário seleciona a opção desejada

Cenário Alternativo:

1. No item 1, se o jogador aceitar a revanche, uma nova partida é iniciada
2. No item 1, se o jogador recusar a revanche, a situação permanece a mesma de antes da oferta

Quadro 28 - Descrição do caso de uso "Aceitar revanche"