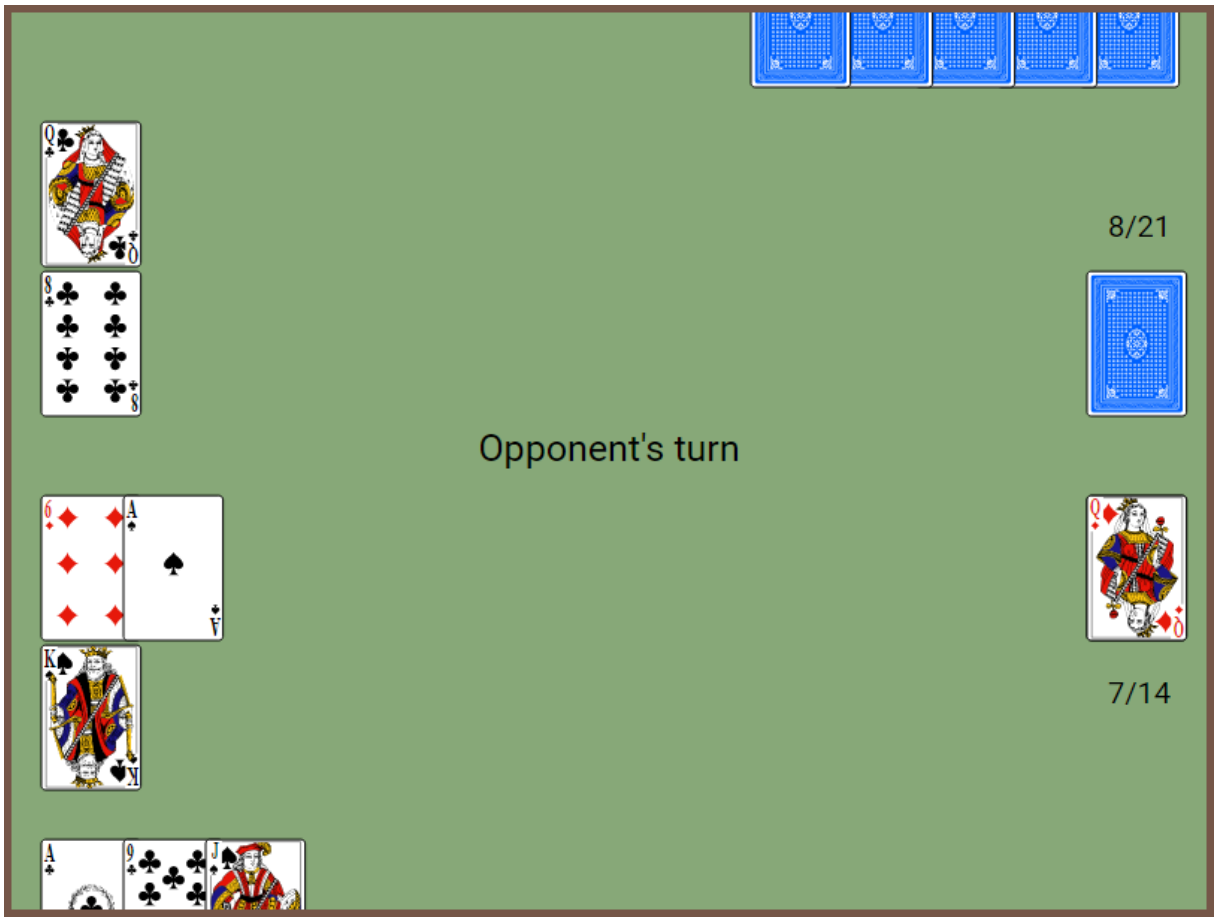
Documentação Trabalho Prático 1
Programação Modular

Renato Utsch e Victor Diniz

May 12, 2016

1 Introdução

Neste trabalho prático, implementamos o jogo chamado Cuttle. A implementação foi realizada para permitir o jogo entre dois jogadores em rede, através de interação cliente-servidor. O cliente implementado permite o controle do jogo por meio de uma interface gráfica web.



1.1 Cuttle

Cuttle é um jogo de baralho para duas pessoas com mecânicas de combate e estratégia. O jogo é jogado com um baralho completo sem curingas, e as cartas podem assumir diferentes comportamentos, divididos em duas categorias principais: cartas de pontos e cartas de efeito. As cartas do A ao 10 podem ser jogadas como cartas de pontos e ficam em campo até serem destruídas. Elas valem, nesse caso, seu valor numérico em pontos (o ás vale 1). As cartas de efeito se dividem entre cartas contínuas (8, J, Q e K) e cartas de uso único (A, 2, 3, 4, 5, 6, 7, 9), também chamadas *one-offs*. O objetivo do jogo é obter 21 pontos de vitória (ou mais).

O jogo é composto por um baralho, uma pilha de descartes e a mão e o campo de cada um dos dois jogadores. O primeiro jogador a jogar começa com cinco cartas em sua mão, enquanto o segundo inicia a jogar com seis. No seu turno, um jogador pode realizar uma dentre quatro ações:

- Sacar uma carta;
- Jogar uma carta como carta de pontos;
- Usar uma carta para fazer o *Scuttle*;
- Jogar uma carta como carta de efeito.

O *Scuttle* é uma jogada em que o jogador utiliza uma carta de pontos de sua mão para destruir uma carta de pontos no campo. É necessário que o valor da carta utilizada seja maior do que o da carta destruída. Caso o valor numérico das cartas seja igual, o critério de desempate é o naipe, para o qual a ordem decrescente de valor é espadas, copas, ouros e paus.

Os efeitos das cartas estão a seguir. Primeiramente, os efeitos das cartas de uso único são:

A Destrói todas as cartas de pontos do tabuleiro;

2 Destrói uma carta alvo de efeito contínuo
OU
Permite que, quando o oponente jogar uma carta de efeito de alvo único, o jogador cancele o efeito dessa carta. Esta jogada também permite reação por parte do oponente;

3 Recupera da pilha de descartes uma carta alvo;

4 Força o oponente a descartar duas cartas de sua escolha;

5 Permite que o jogador saque duas cartas;

6 Destrói todas as cartas de efeito contínuo do tabuleiro;

7 Permite que o jogador saque uma carta, que deve ser utilizada imediatamente. Caso não seja possível, ela deve ser descartada;

9 Retorna uma carta alvo de efeito contínuo para o topo do baralho.

Os efeitos das cartas de efeito contínuo são:

8 Enquanto estiver em jogo, o oponente deve manter sua mão revelada, assim como toda carta que ele venha a sacar;

J Enquanto estiver em jogo, troca uma carta alvo de pontos para o campo do jogador oposto. Se sair de jogo, a carta troca novamente de campo. Se a carta alvo for destruída, o valete também é destruído;

Q Enquanto estiver em jogo, o oponente não pode escolher como carta alvo para cartas de efeito cartas do possuidor da dama;

K Enquanto estiver em jogo, o número de pontos de vitória necessário para que o possuidor da dama é reduzido para os seguintes valores:

  - Um rei: 14 pontos;
  - Dois reis: 10 pontos;
  - Três reis: 7 pontos;
  - Todos os quatro reis: 5 pontos.

Vale notar que o 10 não tem efeito algum. Caso o baralho não contenha mais cartas, a jogada de sacar se torna equivalente a passar o turno. Se ambos jogadores passarem o turno por três turnos seguidos, o jogo termina em um empate.

2 Implementação do jogo

A implementação desse trabalho foi feita em duas partes distintas. O programa CuttleServer, em Java, tem dois pacotes principais: cuttle.game e cuttle.server. O pacote cuttle.game implementa o funcionamento do jogo em si, e permite a comunicação com um servidor por meio de uma interface. cuttle.server, por sua vez, implementa um servidor que se comunica com seus clientes por meio de WebSockets.

2.1 Aspectos principais do jogo

O jogo de Cuttle interage constantemente com os jogadores através da interface com o servidor. A comunicação é feita através de um protocolo simples, baseado em dois tipos de interação: prompts e updates. **Updates** são mensagens unidirecionais, através das quais o jogo informa a ambos jogadores sobre uma ocorrência. **Prompts** envolvem tomada de decisão por parte de um dos jogadores. Quando o jogo realiza um prompt para um jogador, ele envia uma lista de jogadas possíveis e o jogador escolhe qual realizar. O jogo, então, recebe o feedback e realiza a jogada.

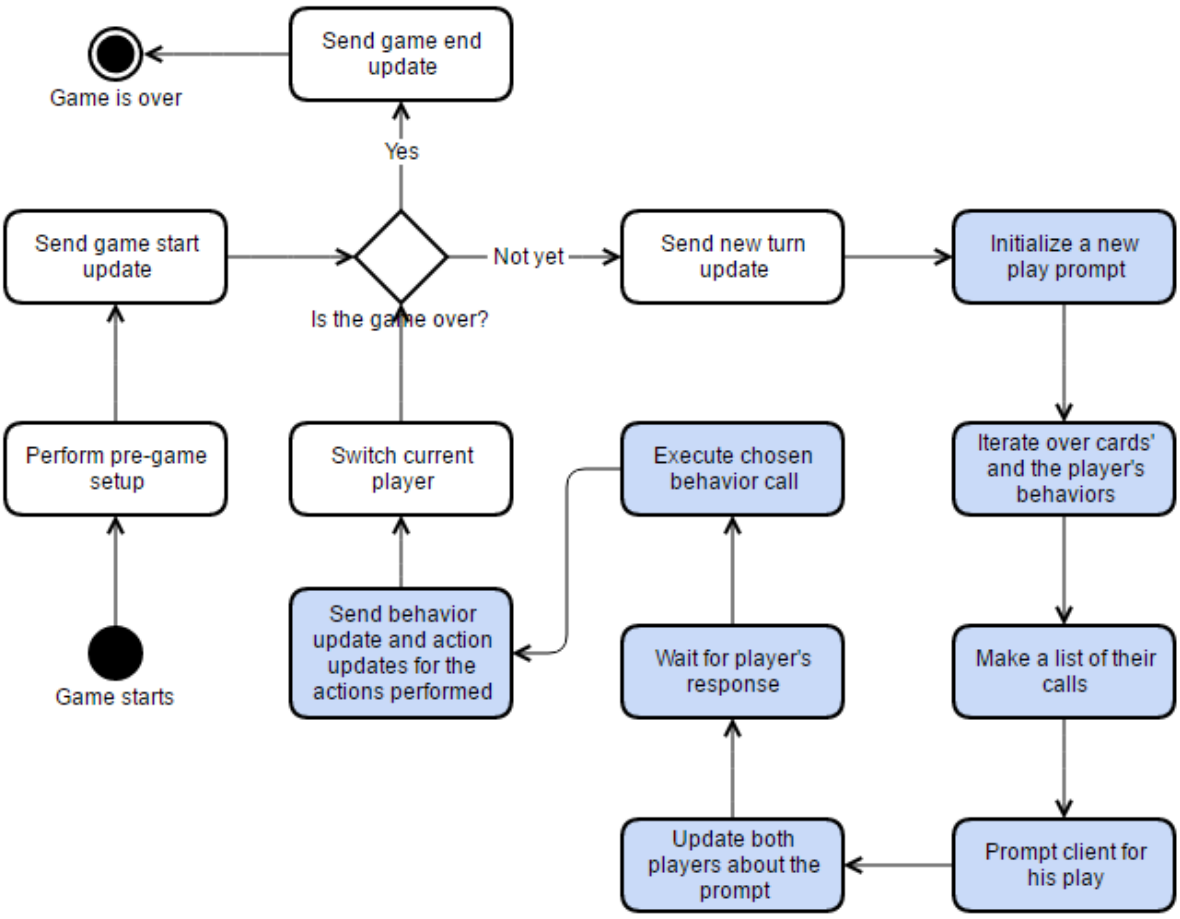
O ciclo do jogo de Cuttle é relativamente simples. Enquanto o jogo está em execução, checa-se se algum dos dois jogadores tem pontos suficientes para ganhar. Caso contrário, inicia-se um novo turno e é enviado um prompt para o jogador do turno, através do qual o jogador deve decidir que jogada realizar. A jogada é realizada e o turno é encerrado, voltando ao início do ciclo.

Estruturalmente, o jogo é composto de um objeto principal correspondente ao jogo em si, que contém jogadores, o deck de cartas e a pilha de descartes. Um jogador contém três coleções de cartas referentes à mão, ao campo de cartas de pontos e ao campo de cartas contínuas, além de uma lista de Behaviors, que serão explicados posteriormente, e um identificados. Finalmente, uma carta do jogo de Cuttle, além de um naipe e um valor, como toda carta, tem uma lista de Behaviors, uma lista de Events (também a ser detalhados) e um identificador numérico.

2.2 Prompts e Behaviors

Behaviors são, como o nome diz, comportamentos. Uma carta de Cuttle pode ter vários comportamentos e é importante que o jogo permita que o jogador escolha qual comportamento será utilizado. Por exemplo, ao jogar um 7, é essencial determinar se o jogador quer acionar seu efeito, jogá-lo como carta de pontos ou fazer Scuttle com ele em outra carta. Os comportamentos, ao contrário dos Events, descritos posteriormente, são inatos à carta. Uma carta de um determinado valor tem sempre os mesmos comportamentos possíveis. Além disso, há, também, comportamentos associados ao próprio jogador, como, por exemplo, sacar uma carta ou realizar um descarte (quando sob efeito do 4).

Prompts não são apenas uma forma de comunicação com os jogadores, mas também uma ocorrência no jogo de Cuttle, com um tipo definido, associado ao enum PromptType. Há quatro Prompts diferentes, de três tipos:



- PlayPrompt: O Prompt enviado no começo do turno, que permite que o jogador realize uma jogada comum;
- ReactionPrompt: O Prompt enviado quando o oponente aciona uma carta de efeito de uso único, permitindo reação com um 2. O valor booleano que determina se o oponente reagiu ou não é armazenado como um atributo do Prompt;
- DiscardPrompt: O Prompt enviado quando o oponente aciona o efeito do 4, que força o descarte de duas cartas (no caso, o Prompt é acionado duas vezes);
- ImmediatePlayPrompt: Subclasse do PlayPrompt do mesmo tipo que ele, enviado quando é jogado um 7. Permite apenas jogadas com a carta que foi sacada (ou o descarte da mesma, caso não seja possível jogá-la).

Todo Behavior é associado a um tipo de Prompt. Um Prompt, ao ser feito a um jogador, realiza uma busca pelos Behaviors de seu tipo presentes no jogador e nas suas cartas, chamando o método listValidCalls em cada Behavior para obter uma lista de possíveis chamadas de behavior - BehaviorCalls. O jogador, então, escolhe, através de um ID numérico atribuído às BehaviorCalls, qual chamada realizar. O jogo recebe o ID e realiza a chamada, que, pór sua vez, chama a função call do Behavior, passando como parâmetro a própria chamada. Essa função não pode modificar diretamente o estado do jogo. Ao invés disso, ela chama a função perform do CuttleGame, que recebe como parâmetro uma ação a ser realizada.

Esse parâmetro é importante porque a BehaviorCall pode conter informações essenciais, no caso de um comportamento que envolve a escolha de um alvo. Nesse caso, a classe TargetedBehaviorCall, que recebe

como parâmetro um alvo em sua construção, é gerada pelo listValidCalls do Behavior. A chamada call do comportamento acessa a propriedade correspondente ao alvo da chamada para realizar sua operação.

Todos os Behaviors herdam da classe Behavior e são associados a um tipo de Prompt (PromptType) e a uma chamada de Behavior (BehaviorCall). A chamada pode ser alvejada (TargetedBehaviorCall) ou não (BehaviorCall). As subclasses diretas dele são PlayerBehavior e CardBehavior, que são comportamentos inatos a jogadores e cartas, respectivamente. As subclasses de PlayerBehavior são todas concretas, sendo elas:

- DrawBehavior: o jogador saca uma carta (PlayPrompt, sem alvo);
- DiscardBehavior: o jogador escolhe uma carta para descartar (DiscardPrompt, com alvo);
- ImmediateDiscardBehavior: o jogador descarta a carta que sacou após jogar o 7 (ImmediatePlayPrompt, com alvo);
- PassBehavior: o jogador ignora a oportunidade de reação (ReactionPrompt, sem alvo).

CardBehavior tem três subclasses concretas, duas das quais relativas às cartas de pontos e uma ao comportamento do 2 como carta de reação:

- PointPlayBehavior: o jogador joga a carta como carta de pontos (PlayPrompt, sem alvo);
- ScuttleBehavior: o jogador realiza Scuttle com a carta em outra carta (PlayPrompt, com alvo);
- TwoReactionBehavior: o jogador levanta um prompt de reação novo e, caso o oponente não reaja, reage ao prompt de reação (ReactionPrompt, sem alvo).

As subclasses abstratas de CardBehavior, por sua vez, são ContinuousBehavior e OneOffBehavior, relativas aos efeitos contínuos e de uso único, respectivamente. Ambas subclasses já especificam o tipo de Prompt como PlayPrompt, pois não há jogadas de efeito fora desse cenário.

Os Behaviors que extendem OneOffBehavior realizam sempre um ReactionPrompt para verificar se o efeito será cancelado antes de ser executado, e são os seguintes:

- AceOneOffBehavior: destrói todas as cartas de pontos do tabuleiro (sem alvo);
- TwoOneOffBehavior: destrói uma carta de efeito contínuo (com alvo);
- ThreeOneOffBehavior: recupera uma carta alvo do cemitério (com alvo);
- FourOneOffBehavior: gera dois DiscardPrompts para o oponente (sem alvo);
- FiveOneOffBehavior: o jogador saca duas cartas (sem alvo);
- SixOneOffBehavior: destrói todas as cartas de efeito contínuo do tabuleiro (sem alvo);
- SevenOneOffBehavior: saca uma carta e levanta um ImmediatePlayPrompt cujo alvo é a carta sacada (sem alvo);
- NineOneOffBehavior: retorna uma carta alvo para o topo do baralho (com alvo).

Por sua vez, os Behaviors que herdam de ContinuousBehavior são:

- EightContinuousBehavior: revela a mão do oponente e vincula EightExitEvent ao 8 jogado (sem alvo);
- JackContinuousBehavior: troca a carta de pontos alvo de campo, vincula ao valete jogado JackExitEvent e vincula também ao alvo JackTargetExitEvent (com alvo);
- QueenContinuousBehavior: aciona a defesa contra cartas de efeito com alvo e vincula à dama QueenExitEvent (sem alvo);
- KingContinuousBehavior: reduz os pontos necessários para vitória do jogador e vincula ao rei KingExitEvent (sem alvo).

2.3 Events e Triggers

Enquanto os Behaviors são persistentes e inatos às cartas, existe outro tipo de ocorrência mais perene: os Events. Events são associados à carta durante o jogo por efeito de um Behavior, e podem ser removidos posteriormente, se necessário. Events são disparados na ocorrência de um gatilho específico, acionado pelo método trigger da carta, que recebe um tipo de Trigger. Todo Event é associado a um tipo de trigger específico. Equipotentemente aos Behaviors, quando disparados, Events realizam uma série de ações para modificar o estado do jogo de alguma maneira.

No jogo de Cuttle, a maioria dos Events são relativos ao efeito de cartas contínuas, cujo efeito persiste até que sejam destruídas. Uma dama, ao adentrar o campo, ativa seu efeito. É necessário que, quando ela seja destruída, o efeito seja desativado. Para isso, é vinculado à rainha um evento disparado pela saída dela do jogo que remove a proteção do jogador. Quando ações que removem a carta do tabuleiro são realizadas, elas disparam eventos do tipo OnBoardExit. Assim, quando a rainha é removida do tabuleiro por qualquer razão ela aciona o seu efeito de saída, sem necessidade de programar isso em várias ações diferentes.

Para fins de extensibilidade, o jogo define a enumeração Trigger, mas há apenas um tipo de trigger necessário: OnBoardExit. Os eventos definidos todos são ativados por essa Trigger, e são os seguintes:

- EightExitEvent: esconde a mão do oponente do possuidor do 8 e a embaralha;
- JackExitEvent: troca a carta de pontos alvo de campo outra vez;
- JackTargetExitEvent: destrói o valete associado à carta de pontos;
- QueenExitEvent: desativa a defesa contra cartas de efeito com alvo;
- KingExitEvent: aumenta os pontos necessários para vitória do possuidor do rei.

2.4 Actions

Finalmente, Actions são as únicas ocorrências com o poder de modificar permanentemente o estado do jogo. Elas são chamadas através do método CuttleGame.perform por Behaviors e Events. Toda ação é comunicada em detalhes a ambos jogadores, visto que o desfecho das mesmas é imprescindível para acompanhar o estado do jogo de Cuttle. Há, também, subclasses abstratas da classe Action. PlayerActions são ações vinculadas a um jogador, TargetedActions são ações que possuem um alvo e TargetedPlayerActions unem as duas características.

Estruturalmente, elas foram divididas entre dois pacotes: cuttle.game.actions.playeractions e cuttle.game.actions.gameactions. Essa divisão foi feita com base na semântica por trás dessas ações: a ação Draw é uma PlayerAction porque o jogador saca a carta, enquanto a ação Destroy faz mais sentido como sendo relativa ao jogo em si, pois a destruição de uma carta nunca é causada diretamente por um jogador, mas sim por consequência de um efeito ou Scuttle.

As ações associadas ao jogo são:

- Destroy: destrói uma carta presente no campo;
- Return: retorna uma carta do campo para o topo do baralho;
- Switch: troca uma carta de pontos para o outro lado do campo.

Associadas a jogadores, por outro lado, são as seguintes ações:

- ContinuousPlay: joga uma carta de efeito contínua;
- Discard: descarta uma carta;
- Draw: saca uma carta;

- HideHand: esconde as cartas da mão;
- LowerProtection: reduz a defesa contra cartas de alvo;
- LowerVictoryReq: reduz os pontos necessários para vencer;
- LowerVisibility: reduz a visibilidade da mão;
- PointPlay: joga uma carta de pontos;
- RaiseProtection: aumenta a defesa contra cartas de alvo;
- RaiseVictoryReq: aumenta os pontos necessários para vencer;
- RaiseVisibility: aumenta a visibilidade da mão;
- Recover: recupera uma carta da pilha de descartes para a mão;
- ShowHand: revela as cartas da mão;
- ShuffleHand: embaralha as cartas da mão.

2.5 Cartas e pilhas

As cartas de Cuttle são implementadas através da classe CuttleCard, que extende uma classe que contém as características básicas de uma carta de baralho, PlayingCard. Toda carta tem uma lista de Behaviors e uma lista de Events. Coleções de cartas no jogo de Cuttle são representadas pela classe Pile, que é utilizada para o deck e a pilha de descartes, e, para cada jogador, a mão, o campo de cartas de pontos e o campo de cartas contínuas. A classe responsável pelo jogo também mantém um mapeamento entre cada carta e a pilha que a contém. Esse mapeamento é inicializado no começo do jogo e a responsabilidade de atualizá-lo é das ações que modificam a localidade de cada carta.

Cada carta concreta de Cuttle extende CuttleCard e, no seu construtor, vincula à carta os Behaviors apropriados. Para juntá-las todas no mesmo baralho, foi implementado o builder DeckBuilder, que gera um deck comum e o embaralha, antes de retorná-lo ao jogo.

2.6 Protocolo de comunicação

Para se comunicar com os jogadores, foi implementado um protocolo de comunicação baseado no sistema previamente mencionado de Prompts e Updates. Esse protocolo permite a interação com o servidor de Cuttle, que implementa a interface ServerInterface. Para intermediar a comunicação entre o jogo, que tem seus próprios objetos, e o servidor, que se comunica sempre por meio se JSONs, há um adapter da classe ServerAdapter. Tanto a ServerInterface quanto o ServerAdapter definem métodos chamados prompt e update, mas com assinaturas diferentes. Os métodos do ServerAdapter acessam objetos do jogo e obtém JSONs utilizados na chamada dos métodos da ServerInterface.

Existem, além dos tipos de Prompt já citados, vários tipos de Update. Todo Update implementa a interface UpdateInterface, que obriga a implementação de apenas um método, buildUpdate, que retorna um UpdateContainer. UpdateContainer é uma classe que define dois métodos essenciais: playerUpdate e opponentUpdate. Quando um Update é emitido, ele envia mensagens para o jogador atual e seu oponente, possivelmente diferentes. Esses dois métodos retornam objetos JSON que são comunicados ao jogador e ao oponente, respectivamente. Esses containers são gerados para armazenar a mensagem a ser enviada, e repassados para o ServerAdapter. Há, também, um SymmetricUpdateContainer, que extende UpdateContainer e serve como um container de conveniência para updates que fornecem a mesma informação a ambos jogadores.

Há várias classes concretas que implementam a interface de update:

- ActionUpdate: comunica os jogadores sobre a execução de uma Action;

- GameEndUpdate: notifica os jogadores sobre o fim de jogo e informações relacionadas, como quem venceu o jogo ou se o jogo terminou em um empate;
- GameStartUpdate: informa aos jogadores sobre o início de jogo e fornece informações essenciais para que o cliente possa agir, como o ID dos dois jogadores, a mão inicial do jogador e o mapeamento entre os IDs numéricos de cada carta e seus naipes e valores;
- NewTurnUpdate: avisa os jogadores quanto ao início de um novo turno e seu jogador;
- PromptUpdate: informa os jogadores de que um Prompt foi realizado a um jogador específico;
- BehaviorCall: comunica os jogadores sobre uma chamada de Behavior realizada e quaisquer informações importantes sobre ela.

Os JSONs utilizados para comunicação são gerados pela própria classe de update ou por métodos das ocorrências sendo notificadas (Action, Prompt e Behavior tem métodos que retornam JSONs sobre o que ocorreu). A serialização desses objetos é possível devido à correspondência dos tipos de Behavior, Prompt e Action e Strings identificadoras, assim como os identificadores das cartas e das pilhas de cartas. É, ainda, possível identificar a localização de uma carta em uma pilha por meio do índice posicional da carta na pilha, que se mantém sempre consistente (exceto quando a mão de um jogador é embaralhada). Esses JSONs são gerados e mantidos através da biblioteca auxiliar org.json[1].

A implementação genérica do protocolo de comunicação permite que qualquer servidor que implemente a ServerInterface definida seja capaz de servir jogos de Cuttle.

3 Implementação do servidor

A classe Main da pasta CuttleServer, fornecida com o trabalho, é capaz de executar dois servidores diferentes: Server e DebugServer. O primeiro é o servidor real, que se comunica pela rede por meio de WebSockets, enquanto o segundo é um servidor de testes, que atua localmente pela linha de comando e permite o comando de ambos jogadores pela entrada/saída padrão. A existência dos dois servidores funcionando de maneiras fundamentalmente diferentes mas ainda assim compatíveis com o jogo demonstra a robustez da interface genérica de servidor implementada.

O DebugServer funciona de extremamente simples e mínima: ele executa o jogo e simplesmente imprime no terminal todos os updates e prompts recebidos, funcionando também como cliente para ambos jogadores. Quando ocorre um prompt, o jogador responde com o ID da chamada escolhida pela entrada padrão. É o absoluto mínimo suficiente para jogar, mas obviamente não convém para um jogador real, visto que é necessário memorizar as cartas de cada pilha e interpretar em um formato muito pouco agradável e legível.

O Server, por outro lado, foi escrito para se comunicar com o cliente web implementado nesse trabalho. A comunicação realizada por ele é feita por meio de WebSockets, através da biblioteca Java-WebSocket[2]. Quando algum cliente se conecta, o servidor espera por outro cliente. Na conexão do segundo cliente, uma nova GameThread é inicializada com o jogo. O servidor gera IDs únicos para cada cliente, que são repassados para o CuttleGame pelo construtor.

Os métodos prompt e update são implementados de maneira relativamente simples. Update consiste apenas de enviar uma mensagem via WebSocket para o cliente com o conteúdo do JSON recebido. Prompt, por outro lado, requer execução síncrona, então o servidor entra em espera, entrando em `sleep` até que uma mensagem tenha sido recebida, sendo então acordado. A classe adicional SocketData foi implementada para guardar informações recebidas e enviadas juntamente com identificação quanto ao socket associado às mensagens. Essa classe também é responsável por garantir que as mensagens cheguem na ordem certa, através de uma fila e métodos com a palavra-chave synchronized.

4 Implementação do cliente

O cliente foi implementado utilizando duas tecnologias principais: o *Polymer* [3], um framework para a criação de web components (objetos reutilizáveis em HTML) para cuidar de alguns elementos da tela e o *d3.js* [4], uma biblioteca de visualização de dados para Javascript, usado para desenhar a interface gráfica do jogo. Além disso, foram utilizadas a linguagens *JADE* [5] – ao invés de HTML – e *LESS* [6] – ao invés de CSS – ao lado de JavaScript. Essas linguagens são linguagens que simplificam o HTML e o CSS para ser mais rápido e menos verboso escrever código, e compilam diretamente para HTML e CSS para serem executadas pelo navegador.

Com o uso de JADE e LESS, é preciso compilar o código antes de ser executado pelo navegador. Aproveitando já haver a necessidade disso, executamos um *minifier* nos códigos HTML, CSS e JavaScript, que reduzem o tamanho dos arquivos e os concatena para apenas um arquivo HTML e um JavaScript, para reduzir o tamanho e a quantidade de requisições que precisam ser enviadas do navegador ao servidor. O código fonte do cliente está na pasta `CuttleClient/src`, enquanto a versão compilada está em `CuttleClient/debug` e a minificada em `CuttleClient/release`.

A tela de jogo é um simples SVG (*Scalable Vector Graphics*, um formato de imagem). Porém, com o o *d3js* e os últimos padrões de HTML 5, é possível modificar SVGs dinamicamente e adicionar eventos a cada elemento da imagem. Essa é uma propriedade de SVGs que é extremamente explorada em bibliotecas de visualizações de dados. Usando disso, o cliente modifica a imagem SVG toda vez que alguma ação é recebida do servidor, fazendo com que reflita o estado do jogo no servidor. Como é possível ligar eventos a objetos no SVG, uma callback é disparada toda vez que o usuário clica em algum elemento da tela, fornecendo as informações de o que foi clicado e em qual posição da tela foi clicado. A partir disso é possível alterar o SVG em resposta ao usuário e dar a impressão de que na verdade a tela é um container renderizado em tempo real, e não apenas uma imagem estática.

O cliente funciona de uma maneira simples: ele possui classes para guardar informações do campo e dos jogadores e, ao se conectar ao servidor, apenas passa a esperar por mensagens. Assim que uma mensagem do servidor chegar, uma callback é invocada (`CuttleClient._onMessage`), e, então, a mensagem passa por uma série de switches que apenas diferenciam cada tipo de mensagem (é um prompt que chegou? Ou é uma action? Se é uma action, de qual tipo? Etc). Para cada tipo de mensagem, há uma função correspondente na classe `CuttleGame`. Com isso, para cada mensagem que chega o cliente apenas faz a ação que a mensagem representa ou modifica o SVG para esperar por uma resposta do jogador. Devido à natureza das mensagens enviadas pelo servidor, não é necessário manter muito estado nem fazer tipo algum de processamento do lado do cliente. Isso é bom por dois motivos: primeiro, deixa o cliente simples, como apenas uma *view* do jogo (a mesma view do padrão *MVC*), segundo, impede que o jogo seja explorado por hackers, visto que o servidor decide quais são as ações possíveis de serem realizadas pelo jogador. O cliente não toma nenhuma decisão, apenas informa o servidor.

Como explicado acima, é extreammente difícil explorar o jogo para algum tipo de vantagem ilícita, já que o servidor apenas responde para mensagens válidas. Por exemplo, ao início de uma jogada, o servidor envia todas as jogadas possíveis na rodada para o cliente, cada uma com um identificador único. Desse modo, a única coisa que o cliente pode fazer é responder com o identificador único da decisão tomada pelo jogador, e não pode adulterar nenhum dado. Jogadas que não são possíveis no momento simplesmente não têm um identificador e não são realizadas pelo servidor.

O código fonte do a parte do cliente que manipula o jogo pode ser encontrado na pasta `CuttleClient`, no diretório `src/elements/page-game/cuttle`.

5 Jogando o jogo no cliente

Para jogar o jogo no cliente, abra o navegador no endereço `http://localhost:4544` (veja a seção 6), coloque o endereço `localhost:42001` e pressione jogar em duas abas do navegador (ou em computadores diferentes, não tem problema). O jogo irá iniciar. Em cada rodada, o jogador pode escolher uma ação (comprar uma carta ou jogar alguma das cartas da mão), o que pode ser feito clicando nas cartas correspondentes. As cartas

da mão que tiverem jogadas possíveis abrirão uma janela com as opções para jogar. Caso uma carta for jogada como efeito único (one-off), o oponente terá que confirmar se vai deixar isso ser executado (pass) ou se vai tentar parar a carta de efeito único (usando um dois).

Note que o cliente não possui todas as animações para ficar totalmente claro para ambos os jogadores o que ocorreu na jogada. Devido a isso, também há um log na página (ou à direita ou embaixo da tela do jogo, dependendo da resolução do monitor) que possui os JSONS de ação e update que foram enviados do servidor. Com isso, é possível vasculhar o que aconteceu na última rodada.

É possível redimensionar o jogo com os simples Ctrl+ ou Ctrl- do navegador.

6 Compilação e execução

Para compilar e executar automaticamente o cliente e o servidor, digite na pasta raiz do jogo (a pasta com o README) o comando `make run`. Isso compilará e executará ambos o cliente e o servidor. O servidor irá iniciar no endereço `localhost:42001`, enquanto o cliente irá iniciar no endereço `http://localhost:4544`. Após isso, basta abrir o navegador (no Chrome, o cliente não foi testado em outros navegadores) e acessar o link. O resto dessa seção explica como compilar separadamente o cliente e o servidor e também como recompilar o código do website caso seja de interesse.

O servidor foi implementado exclusivamente em Java, e pode ser executado através da Makefile presente na pasta CuttleServer por meio do comando `make run`. A porta padrão é 42001, e é possível alterar a porta de acesso do servidor com a sintaxe `make run -- --port <porta>`. Caso haja interesse em executar o jogo em modo de debug no terminal (instanciando o DebugServer), é possível, também, executar o comando `make run -- --debug`.

Caso acessar o cliente remotamente seja suficiente, ele está disponível em `http://altaria.cf/cuttle/`. Nesse caso, para acessar o servidor hospedado remotamente, basta definir o endereço de conexão na tela inicial como `altaria.cf:42001`. É possível, também, acessar o servidor em execução local através do endereço padrão `localhost:42001`.

A execução local do cliente pode ser feita através de um mini-servidor HTTP em Java que utiliza a biblioteca NanoHTTPD[7], implementado em CuttleClient/server/. Esse modo de execução simplesmente executa uma versão previamente compilada do cliente, fornecida na pasta CuttleClient/release/. Para isso, basta executar, a partir da pasta CuttleClient, o comando `make run` e acessar o servidor no endereço `http://localhost:4544/`. Caso seja preciso modificar a porta de acesso, é possível, também, definir isso como parâmetro do comando, no formato `make run <porta>`.

Para executar o cliente a partir do seu código-fonte original, é necessário ter instalado o *npm*[8], utilizado para instalar as dependências do cliente. Para instalar essas dependências, basta, então, executar a partir da pasta CuttleClient:

```
npm install -g bower gulp
npm install
bower install
```

Talvez seja necessário reiniciar o terminal entre os `npm install`s e o `bower install`.

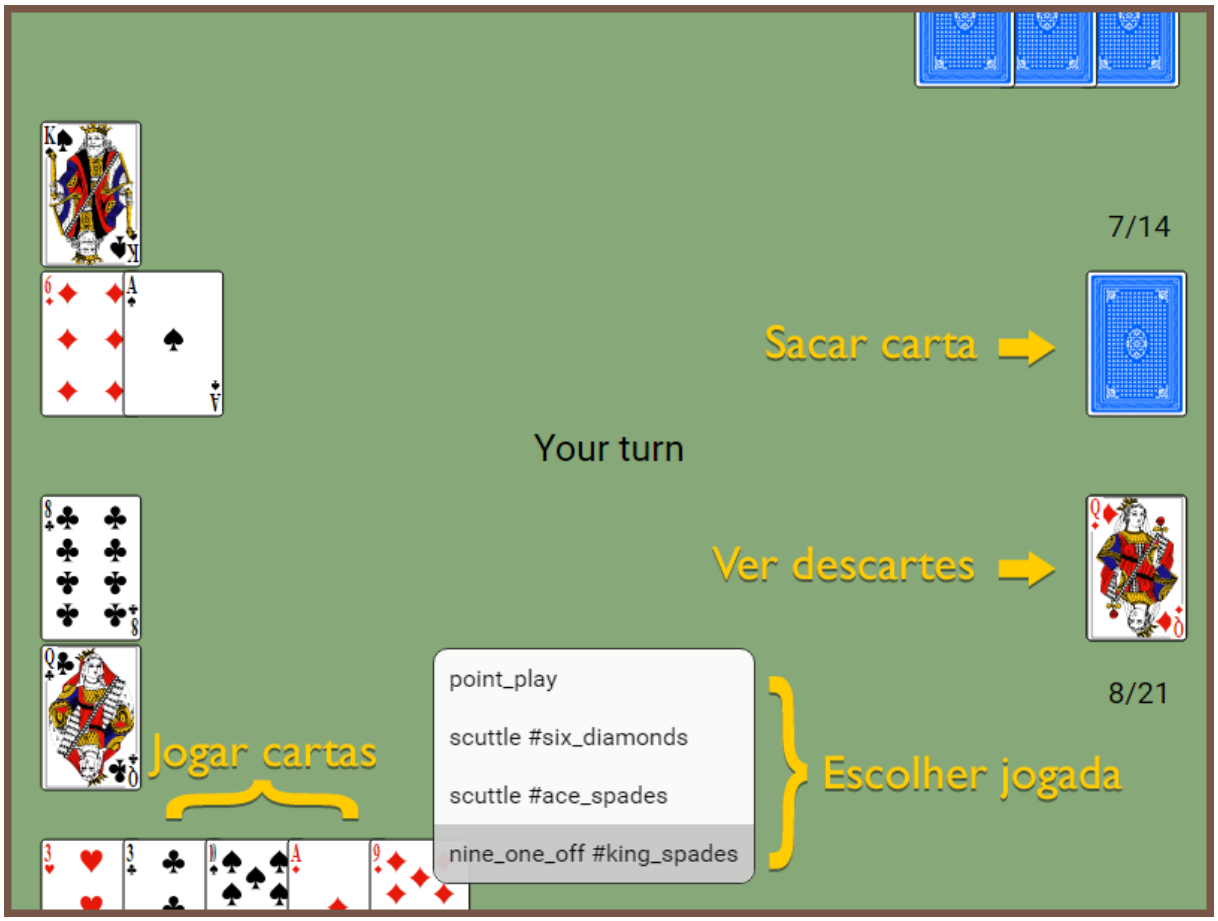
Após isso, para executar o cliente, o comando a ser utilizado é `gulp serve`. O cliente estará acessível pela url `http://localhost:5000`. O cliente foi apenas testado no navegador *Google Chrome* atualizado, e, portanto, pode não funcionar em outros navegadores. Sugerimos a utilização do mesmo.

7 Jogando o jogo através do cliente web

Após a conexão de ambos jogadores, tem início o jogo nos clientes. A interface tem os elementos do jogo de Cuttle: a mão dos dois jogadores, o deck, a pilha de descartes e o tabuleiro, dividido em áreas do jogador e do oponente, cada uma com um campo de cartas de pontos e um campo de cartas contínuas. Acima do

baralho e abaixo dos descartes, há a contagem de pontos de cada jogador, assim como o número de pontos necessários para vitória.

No turno do jogador, é possível clicar no baralho para sacar uma carta ou clicar em uma carta que pode ser utilizada na mão, para revelar jogadas viáveis que podem ser realizadas com a carta em questão.



Durante o turno do oponente, caso ele jogue uma carta de pontos ou jogue um 4, pode ser necessário que o jogador reaja de alguma forma. Para isso, pop-ups similares ao de escolha de comportamento da carta surgirão e o jogador deverá escolher uma jogada para dar seguimento ao jogo.

No fim de jogo, o jogo mostra uma mensagem afirmando se houve vitória, derrota ou empate. Caso haja alguma falha de conexão, há, também, feedback para o jogador. Finalmente, à direita, do tabuleiro, há um registro de ocorrências do jogo.

8 Conclusão

Nesse trabalho, implementamos um jogo de Cuttle. A implementação foi feita em rede, com servidor em Java e um cliente em HTML e JavaScript com interface gráfica. Documentamos a modelagem do jogo, do servidor e do cliente em diagramas de classes UML e, também, o fluxo de execução do jogo por meio de um diagrama de atividades UML. Além disso, detalhamos a implementação e as decisões tomadas nesta documentação. Aplicamos neste trabalho muitos dos conceitos vistos em sala e tivemos a oportunidade de fazê-lo de maneira didática e lúdica.

Appendices

A Diagramas UML

Anexa a esta documentação, estão os diagramas UML realizados para demonstrar a estrutura do trabalho implementado, na pasta `doc/uml/`. O diagrama do jogo e do servidor foi dividido em três diagramas. O diagrama principal (`cuttle.png`) contém as classes do jogo e do servidor, exceto pelas ações e pelos behaviors, que estão em dois diagramas distintos (`behaviors.png`, `actions.png`). Além disso, há o diagrama de atividades (`activity.png`) e o diagrama de classes do cliente (`client.png`).

Alguns diagramas contém várias classes em um só bloco, para economizar espaço. Não foram representados métodos que simplesmente retornam uma propriedade. Caso um construtor tenha sido omitido, ele é idêntico (exceto em nome) ao construtor da sua superclasse. Caso os parâmetros ou o tipo de retorno de uma função tenha sido omitida, considera-se que é a mesma assinatura da função de mesmo nome na superclasse.

B Trilha sonora

O jogo possui uma trilha sonora original feita através do Famitracker, que pode ser escutada através do cliente web. É possível interromper a reprodução da mesma ou alterar o volume pelos controles acima do tabuleiro.

References

[1] S. Leary, "Json-java: A reference implementation of a json package in java." `https://github.com/stleary/JSON-java`, 2016. [Online; acessado em 11 de maio de 2016].

[2] D. Rohmer and N. Rajlich, "Java-websocket: A barebones websocket client and server implementation written in 100% java." `http://java-websocket.org/`, 2015. [Online; acessado em 11 de maio de 2016].

[3] "polymer." `https://polymer-project.org/`, 2013 - 2016. [Online; acessado em 11 de maio de 2016].

[4] M. Bostock, "d3js - data driven documents." `https://d3js.org/`, 2010 - 2016. [Online; acessado em 11 de maio de 2016].

[5] "jade - node template engine." `https://jade-lang.com/`, 2010 - 2016. [Online; acessado em 11 de maio de 2016].

[6] "less - css preprocessor." `https://lesscss.org/`, 2010 - 2016. [Online; acessado em 11 de maio de 2016].

[7] P. Hawke, J. Elonen, and R. van Nieuwenhoven, "Nanohttpd – a tiny web server in java." `http://www.nanohttpd.org/`, 2016. [Online; acessado em 11 de maio de 2016].

[8] I. Z. Schlueter, "npm - node.js package manager." `https://www.npmjs.com/`, 2009 - 2016. [Online; acessado em 11 de maio de 2016].

[9] "Java platform, standard edition 7 api specification." `https://docs.oracle.com/javase/7/docs/api/`. [Online; acessado em 11 de abril de 2016].