UNIVERSIDADE FEDERAL DE LAVRAS

DESORIENTADOS

ANDRÉ CHATEAUBRIAND

LETÍCIA DINIZ TSUCHIYA

MARCO TÚLIO COSTA

TRABALHO 2

JJSoccer

LAVRAS – MG

2016

1) Contextualização

O projeto é um jogo de futebol com intuito de realizar partidas, seguindo as regras do mesmo, de forma bem simplificada, o objetivo do jogo é realizar o maior número possível de gols.

O jogo é *multiplayer*, portanto dois usuários podem controlar um jogador de um time numa partida. Esse jogador controlado poderá ser facilmente alterado entre os jogadores do time que estão na partida. Existe a possibilidade de movimentar os jogadores, chutar a bola e trocar de jogador. Os jogadores de ambos os times que não estão sendo controlados pelo usuário serão controlados por uma IA (inteligência artificial).

Os jogadores controlados pela IA tem suas ações definidas de acordo com sua posição no time (goleiro ou jogador comum). O usuário terá a possibilidade de controlar apenas os jogadores comuns, que estão jogando em campo, não é possível assumir a posição de goleiro.

Durante o jogo será exibido o placar atual e a partida será limitada por um tempo previamente definido de dez minutos.

2) Modelagem

Conceitos Gerais do Motor do Jogo:

O Motor do Jogo basicamente funciona com um *loop*, conhecido como *game loop*, que executa as ações do jogo e em seguida o renderiza (imprime as imagens do jogo).

O jogo é feito de cenas, cada cena tem sua própria lógica. No protótipo temos apenas a *MatchScene*, que é a cena no jogo efetivamente, mas prevemos a cena de menu, MenuScene, que será responsável pelo controle do menu principal.

Uma cena é composta por vários atores (*Actor*). Um ator possui uma posição na cena, definida pelas coordenadas X e Y. Também possui uma ação/atuação dentro da cena, que cada ator implementa tal de uma forma diferente. Algumas funções também fazem parte do ator, como as funções de movimentar (*move*), verificar se o movimento é valido (*canMove*), a definição de sua caixa de colisão (explicado abaixo) e o seu *sprite*.

Além dos atores, a cena também possui um campo, que não atua na cena, mas é apenas renderizado no fundo da cena, e define os limites do campo.

Basicamente no fluxo do jogo, a cena recebe a chamada para executar suas ações no *gameloop*. Por sua vez a cena delega tal chamada aos seus atores, dizendo-os para atuar (*act*), como cada ator implementa sua ação de sua forma, os atores farão seus papéis de acordo com o definido.

Os conceitos descritos até então foram seguidos e baseados nos conceitos gerais de motores de jogos.

Dos Modelos

No jogo temos os modelos básicos, que são as entidades independentes do motor e a lógica não interfere na definição dessas classes, estando isoladas. São elas:

- O ***Jogador*** com seu nome e suas habilidades. As *Habilidades* compõe um jogador, somente existindo se esse existir, por isso composição. Essa classe possui apenas construtores para o jogador e *getters* de informações úteis sobre o jogador.
- O ***Time*** com seu nome, sua cor e uma lista de jogadores. No jogo os jogadores estão ligados à um *Time*, pois são criados dentro do construtor, e são intransferíveis. Portanto, sem o time não existem, sendo assim uma composição.
- A ***Partida*** é composta por dois times, Time da Casa e Time Visitante, e seus respectivos gols. A ligação é uma agregação, pois a parte, que são os times, existem independente das partidas.
- A ***Habilidades*** é composta pelas habilidades que os jogadores podem assumir, velocidade, drible, defesa e específica. É feito o uso de enumeradores para esse gerenciamento.

Existem além dos modelos básicos, os demais modelos são utilizados e definidos de acordo com a lógica do jogo. São eles:

- O ***Campo*** é a definição dos limites do campo do jogo. A *matchScene* possui um campo, que a compõe - o campo só existe nessa cena e essa cena só existe com um campo, portanto existe um relacionamento de composição.
- O ***Actor*** é o centro de toda lógica do jogo. É uma classe abstrata, pois não existe uma atuação (*act*) genérico, necessitando de cada ator - filho deste - implementar sua atuação. Descrito mais detalhadamente a baixo.
- O ***JogadorActor*** estende de *Actor*. Foi definido que o *JogadorActor* estende de *Actor* e não de *Jogador* pois seus métodos utilizados são do *Actor*, e toda sua lógica se baseia nesse. Métodos como a sua movimentação, verificação de colisão, controle da direção são idênticos à qualquer ator, sendo assim mais lógico herdar as implementações de ator - ao invés de implementar o *Actor* que por sua vez ser uma interface com o método *act*. O *JogadorActor* tem um atributo que é um *Jogador*. Apesar de soar estranho *JogadorActor* possuir um *Jogador* e não ser um *Jogador*, na prática faz mais sentido, uma vez que só é utilizado o *Jogador* para referenciá-lo, e não para a lógica do jogo, muito menos para polimorfismo. Na *MatchScene*, é utilizado polimorfismo com *Actor*.
- A ***Bola*** estende de *Actor*, para utilização do polimorfismo na *MatchScene*. O seu *act* define seu movimento. Possui sua velocidade no eixo X e Y e seu movimento vai de acordo com essa velocidade..
- O ***GamePlayer*** estende de *JogadorActor*. O *GamePlayer* é basicamente um *JogadorActor*, porém controlado pelo usuário, onde seu *act* é sobrescrito para se comportar de acordo com os comandos do usuário, controlador pelo *InputManager*. O *GamePlayer* usa o *InputManager*.
- O ***Sprite*** e ***AnimatedSprite*** são a definição das imagens do *Actor* que será impressa na tela. Um *Actor* é composto por um *Sprite*. Um *Actor* só existe com um *sprite* e um *sprite* só existe por conta do *Actor*. Um *AnimatedSprite* é um caso especial de um Sprite, portanto o estende.
- ***Dimensao*** é uma classe que auxilia na coesão por encapsular uma dimensão, possuindo altura e largura.
- ***CollisionArea*** define um retângulo de colisão de um *Actor*. Possui os métodos para controlar as colisões. Verificando se dois atores estão colidindo, por análise de suas *CollisionAreas*.

O *JogadorActor* e o *Design Pattern Strategy*

O *JogadorActor* utiliza o *design patter Strategy* para definir seu comportamento. Como diferentes *JogadorActors* tem diferente comportamentos, eles possuem um atributo que se refere ao comportamento, um atributo do tipo *ComportamentoAtua*, uma interface. Esse comportamento pode

ser entre um jogador Controlado, que sua ação é definida pelo *input*, podendo ser tanto *Player1* quanto *Player2*, pode ser uma IA, que sua ação é programada, sendo tanto um *JogadorIA* quanto um *GoleiroIA*. Quando o usuário requer a troca de personagem que está controlando, o programa simplesmente altera o comportamento dos jogadores em campo, colocando o comportamento *JogadorIA* no antigo jogador controlado, e o comportamento controlado em um outro jogador que esteja como IA.

O Ator

Como dito, o *Actor* é coração do motor do jogo. Ele define tudo aquilo que compõe uma cena, exceto pelo chão, chamado de campo no jogo.

Todo ator possui sua posição X e Y na Cena, e sua movimentação é definida alterando sua posição X e Y. Todo ator pode se mover na cena, mesmo que esses não movam na prática. A movimentação de um ator só é permitida se esse não colidir com outro ator. Portanto um ator é capaz de movimentar e de verificar se esse movimento é valido pelos métodos *move* e *canMove,* respectivamente. A colisão de um ator é controlada pela sua *CollisionArea*. Todo *Actor* tem uma *CollisionArea*.

Um ator possui vários métodos implementados que os seus filhos utilizam, por este motivo não pode ser uma interface. Porém o método atuar não tem uma forma genérica, precisando que seus filhos definam a forma que ele desejam atuar.

O método atuar (*act*) recebe por parâmetro uma lista de atores que estão colidindo com esse, para que o ator possa tratar as colisões, sem precisar conhecer a cena para obter essas colisões. O método também tem como parâmetro uma *Action*. Essa *Action* tem o objetivo similar à lista de colisões, que é passar informações ao ator sem que esse tenha que conhecer a cena. Essas informações podem variar de ator para ator, a única informação que todo ator deve conhecer é o limite do campo, para o controle do *canMove*.

Por ter um método abstrato e vários atributos e métodos concretos, um Ator é uma classe abstrata. A única ligação da cena com os *models* é o *Actor*, a cena controla apenas um *Actor*, que por sua vez pode ser os modelos filhos de *Actor*, sem que a cena saiba exatamente, assim permitindo com que mudanças na lógica dos modelos não alterem a lógica da cena, aproveitando o máximo do polimorfismo.

O *Actor* também implementa a interface *Renderable*, que o exige ter métodos para informar sua imagem (*getImage*) e posição (*getX* e *getY*) para que o *Renderer* (explicado mais a frente) possa o imprimir na tela. Vale lembrar que apesar de todo *Actor* ter esses métodos, nem todo *Renderable* é um *Actor*, pois o campo é um *Renderable*, porém não é um *Actor*.

Dos Controladores

Enquanto a lógica do jogo fica no *Actor*, quem delega essa lógica são os controladores. Os controladores são, em sua maioria, o motor do jogo. São classes controladoras:

- O ***Game***, que é onde o jogo é inicializado, define as configurações iniciais do jogo e inicializa o gameloop. Delega ações ao *act* e *render* a cena. O *Game* possui um *Frame*, que é onde o jogo é impresso, e uma *GameScene* que é a cena atual do jogo. Ambos são composição, uma vez que o jogo (*Game*) só faz sentido com eles, e eles só fazem sentido num jogo (*Game*). O game utiliza o design pattern Observer com suas scenas, em que essas o avisam quando trocar a cena e para qual cena. O Game é um SceneListener, isso é, ele ouve a cena, assim implementando o método changeScene, o qual altera a cena vigente. Isso garante que as cenas possam solicitar mudança de cena (como o menu requerer o inicio da partida) sem que conheçam o Game.
- O ***GameScene*** é uma classe abstrata, utilizada apenas para o polimorfismo no *Game*. Ela define a cena do jogo, seja ela um menu, uma partida, um cena de créditos, uma animação, ou quaisquer outras cenas. O jogo não tem de se importar com o que seja a cena, desde que essa seja capaz de se controlar e se imprimir.
- O ***MenuScene*** é uma cena de menu, a qual pode inicializar uma nova partida, ou sair do jogo.
- O ***MatchScene*** estende *GameScene*, portanto possui os métodos *act* e *render*. A *MatchScene* é uma cena da partida, responsável por controlar a "jogabilidade" desta. Possui uma lista de atores que compõe a partida e um campo que define o limite do mapa. Existe um relacionamento de composição, pois os atores e o campo só existem numa cena. Vale lembrar que um jogador não é um ator, portanto esse existe fora da cena, mas um *JogadorActor* é um ator, e esse não existe fora da cena. A *act* da *MatchScene* apenas delega o *act* de cada um de seus atores.
- ***InputManager*** é um *singleton* (*Design Pattern*) que gerencia as teclas de entrada do usuário. Possui métodos para verificar se uma tecla específica foi pressionada, liberada, ou *justPressed* (pressionada no último loop de jogo), a classe foi fortemente baseada em uma classe disponível na internet para esse mesma finalidade, não sendo totalmente desenvolvida pelos membros do projeto.
- ***ImageManager*** é um *singleton* que controla as imagens do jogo. Possui um *HashMap* que salva as imagens. Quando é requerido uma imagem para o *ImageManage* esse verifica se essa existe no *HashMap*, caso exista retorna uma referência à essa. Caso não existe, carrega essa imagem do HD para o *HashMap*, e

retorna uma referência dessa, garantindo que não há replicações de instâncias de uma mesma imagem e que não carregue imagens não utilizadas.

- **AudioManager** é um *singleton* que controla os audios do jogo. Funciona como o ImageManager, porém com audios WAV.

Da Visão

A Visão é o conjunto de classes responsáveis por imprimir de fato a cena de jogo. As classes da visão são:

- O ***Frame*** estende de *JFrame*, sendo assim uma Janela, que será onde as imagens da cena serão impressas.
- O ***Renderable*** é uma interface responsável por definir a imagem e a posição dessa na cena. A Visão não tem conhecimento de nenhuma classe dos modelos e do controle, conhecendo apenas *Renderables*, pois para a visão só importa, qual imagem deve ser desenhada e aonde esse imagem deve ser desenhada.
- ***Renderer*** é um *singleton* onde está a lógica do desenho dos *Renderables*. Seu método *renderer* recebe por parâmetro uma lista de *Renderables* e um *Graphics* (classe da *Swing* que controla o desenho num container), e utiliza o *graphics* para desenhar os *Renderables* em seus respectivos locais. O *Frame*, ao chamar o render do *Renderer*, passa por parâmetro seu Graphics, e uma lista de Renderables recebida do Game, para que possa ser desenhado as imagens da cena em si, pelo *Graphics*.

Toda a representação UML do projeto está presente no arquivo enviado juntamente com esse relatório.

3) Manual de usuário

Ao executar o jogo "*JJSoccer*" será exibido um menu, onde será possível realizar novas partidas ou sair. Ao inicializar uma nova partida, será exibido o campo do jogo, onde cada time terá no total seis jogadores, um goleiro e cinco em campo. O time casa possui uniforme amarelo e seu lado do campo é esquerdo, portanto os gols devem ser feitos no gol a direita da tela. O time visitante possui uniforme azul e seus gol devem ser feitos no gol a esquerda da tela, assim como um jogo de futebol normal.

O jogador que está sendo controlado possui uma indicação com um círculo abaixo do jogador, vermelho para o time casa e azul para o time visitante. As teclas utilizadas para o controle dos jogadores são:

Player 1:

"W"	movimenta para cima
"S"	movimenta para baixo
"A"	movimenta para a esquerda
"D"	movimenta para a direita
"Q"	troca de jogador
"SPACE"	chuta

Player 2:

"↑"	movimenta para cima
"↓"	movimenta para baixo
"←"	movimenta para a esquerda
"→"	movimenta para a direita
"SHIFT"	troca de jogador
"CTRL"	chuta

Os jogadores que podem ser controlados são apenas aqueles que estão em campo. Ao chutar a bola, esta segue a direção para a qual o jogador está caminhando.

O placar é exibido durante toda a partida e contabiliza os gols marcados. Ao pressionar a tecla “M” o áudio executado durante a partida é colocado no mudo, para ativar o áudio, basta pressionar novamente a tecla “M”. A duração de cada partida é de no máximo dez minutos, vencendo o time que obtiver a maior pontuação.

Ao pressionar a tecla “ESC” o jogo é redimensionado para a tela de menu, onde poderá iniciar uma nova partida, ou sair do jogo.

4) Observações técnicas

Para a implementação do jogo, foi feita a utilização das seguintes bibliotecas Java disponíveis:

- java.awt.Color: Utilizada para a coloração dos objetos impressos na tela, a classe Color é usada para encapsular cores no espaço de cor RGB padrão, portanto facilita a renderização de imagens.
- java.awt.event.KeyEvent: Utilizada para indicar o pressionamento de uma tecla.
- java.awt.event.KeyListener: Interface ouvinte para receber eventos de teclado (teclas digitadas).
- java.awt.Graphics: Classe abstrata base para todos os contextos gráficos que permite que um aplicatico desenhe os componentes em diversos dispositivos.
- java.awt.image.BufferedImage: Classe utilizada para descrever uma Image com um buffer dos dados da imagem.
- java.awt.image.BufferStrategy: Classe que representa o mecanismo necessário para organizar a memória complexa sobre uma determinada janela.
- java.awt.Image: Pacote que fornece classes para criar e modificar imagens.
- java.awt.Toolkit:
- java.io.IOException; Classe que gera exceção produzidas por operações de E / S com falha ou interrompidos.
- java.net.URL: A classe URL representa um Uniform Resource Locator, um ponteiro para um "recurso" na World Wide Web. Um recurso pode ser algo simples como um arquivo ou uma pasta, ou pode ser uma referência a um objeto mais complicado, como uma consulta para um banco de dados ou para um motor de busca.
- java.util.Collections: Pacote dos conjuntos que representam um grupo de objetos, conhecido como seus elementos. Foram utilizados no projeto ArrayList, HashMap, LinkedList e List para realizar as manipulações dos objetos.
- java.util.Random: Utilizada para a geração de números aleatórios.
- javax.imageio.ImageIO;
- javax.swing.JFrame: Utilizada para a construção de janela, que será utilizado para exibir o jogo.
- javax.swing.JOptionPane: Utilizada para a criação de janelas de diálogo com o usuário, pedindo informações ou usada como forma para informar algo.

Foram criadas bibliotecas para a organização do projeto, conforme descrito na modelagem e representado na UML.

5) Resultados alcançados

Imagem 1: Logo do jogo - JJSoccer

Durante o processo de criação do programa muita coisa foi alterada em relação ao planejado inicial. O resultado final alcançado foi o descrito nas seções anteriores desse documento. O resultado foi satisfatório, em vista que muitas das alterações em relação a ideia inicial foram melhorias, otimizações e aplicações de padrões e boas praticas de projeto. Tento algumas *features* retiradas por questões de tempo, muitas das *features* foram refinadas sua implementação.

As principais mudanças em relação à ideia inicial foram:

- **JogadorActor herando de Actor e sendo composto por Jogador:**

O que inicialmente parecia óbvio e natural um JogadorActor herdar de Jogador, por conta de ser um caso específico de jogador, na prática percebemos que o JogadorActor era um Actor, e a utilização do polimorfismo nessa herança foi muito mais vantajoso - uma vez que o MatchScene possui lista de atores, e não precisa se importar com que tipo de ator é, desde que todos tenham o método act.

- **Strategy em JogadorActor e seus comportamentos**

Inicialmente um jogadorActor teria uma arvore de heranças, em que cada casa do JogadorActor seria um filho do mesmo, como o Goleiro, o JogadorIA, e o GamePlayer (jogador controlado pelo usuário). Uma solução natural para o caso. Porém, ao implementar a troca de jogadores, em que, em tempo de execução, o jogador mudaria seu comportamento - um GamePlayer se tornaria JogadorIA, e um JogadorIA se tornaria GamePlayer - percebemos a complicação da solução com herança, e decidimos utilizar o design pattern Strategy, solucionando o problema.

- **Alteração da variação de Jogadores para apenas Goleiro e Jogador**

Como a IA do jogo é bastante simples, a ideia inicial de possuir atacante e defensor não foi implementada, tendo apenas diferença entre um jogador comum e um goleiro.

- **Alteração do Placar**

  A ideia inicial de ter o placar com times salvos e seus históricos foi abandonada por questões de tempo, mas ainda será implementada para versões futuras, tendo a atual modelagem prevendo tal cenário, em que não teria de refatorar o projeto para adicionar tal funcionalidade.

  Como resultado final, obtivemos o jogo descrito nas seções 1, 2 e 3 deste relatório. De forma geral, o jogo é inicializado com uma tela de menu onde é possível realizar partidas ou sair. Ao iniciar uma nova partida é possível jogar com dois *players* e a partida tem duração de dez minutos. No final aparece o placar com o resultado da partida.

Imagem 2: Tela de menu

Imagem 3: Jogo em execução

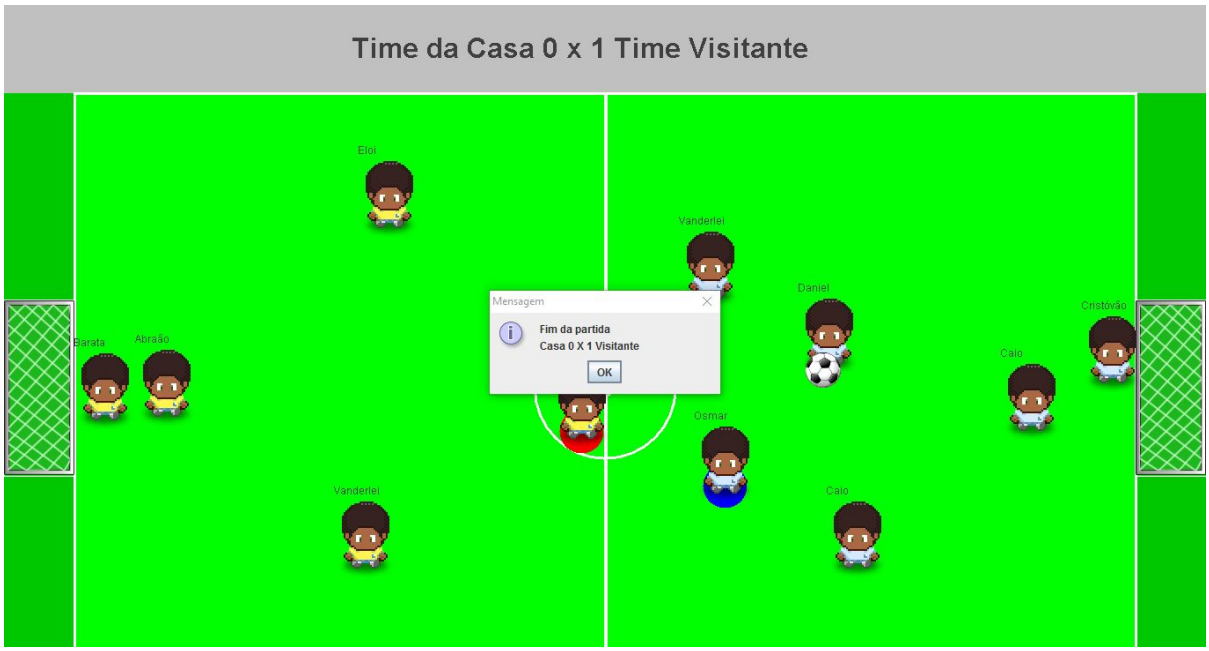

Imagem 4: Fim da partida

6) Conclusões

Como descrito na seção anterior, durante a confecção do projeto, os conceitos aprendidos na disciplina de OO foram cruciais para otimização do jogo, e até mesmo para possibilidade do mesmo ser concluído. A utilização do polimorfismo foi o principal pilar do jogo, com a boa coesão , e consequentemente baixo acoplamento, adicionar novas funcionalidades foi um trabalho simples, em que o código pronto pouco, se não nada, era alterado. Os *Design Patterns* aprendidos na disciplina foram fundamentais para resolver os principais problemas encontrados. A preocupação da equipe em manter a alta coesão e o sistema modularizado, teve grande papel na garantia de um sistema legível. E esse esforço não foi em vão. A equipe não se encontrou perdida no código em nenhum momento, sendo facilmente identificado o código feito pelo colega de equipe. Como o programa foi feito utilizando a ferramenta de versionamento de software *Git*, acessávamos o código na versão oficial mais recente, e alterações feitas simultaneamente por diferentes membros da equipe não conflitaram entre si, sendo cada *feature* adicionada não causando consequências no código pronto. O que demonstra o baixo acoplamento. As poucas vezes que houve conflito de código foram quando os membros trabalhavam na mesma classe, mas foram conflitos simples, vide que, mesmo numa mesma classe, cada membro trabalhou numa parte diferente dessa. Os conceitos aprendidos em aula permitiram o trabalho em equipe, que talvez fosse inviável de serem feitos sem tais conceitos.