Ano Letivo:
2015/2016

Instituto Politécnico de Setúbal

Escola Superior de Tecnologia de Setúbal

Entrega Final

Versão Atualizada

PROFESSORA RESPONSÁVEL: PATRÍCIA MACEDO

CADEIRA: PROGRAMAÇÃO AVANÇADA

TRABALHO REALIZADO POR:

ANDRÉ BASTOS - 140221017

LUÍS MESTRE - 140221002

Introdução

O projeto tem como objetivo nós aplicarmos os conhecimentos adquiridos no âmbito da unidade curricular de Programação Avançada.

O trabalho consiste em realizar uma aplicação que disponibilize o jogo Três em Linha. O jogo Três em Linha será feito num tabuleiro que vai sendo preenchido por peças que irão ser disponibilizadas de modo aleatório. O objetivo deste jogo é juntar três peças da mesma cor para forma uma linha horizontal, de seguida fazendo com que essas 3 peças sejam apagadas da linha.

Nesta última fase do projeto iremos mostrar e explicar o uso do padrão MVC, pedido para o bom funcionamento do jogo em JavaFX. Iremos também mostrar o uso de novos padrões e alterações que foram necessárias no código da parte Model. Para esclarecer de forma mais coerente as alterações iremos deixar os diagramas do Milestone2 e acrescentamos as imagens desta última fase.

Diagrama de classes do modelo de desenho

Milestone2

Completo

Link: http://i.imgur.com/JAfjdmy.jpg

Jogo

Link: https://i.imgur.com/bgwly5O.png

Fase Final[1]

Completo

Link: http://i.imgur.com/Z3jRJjb.png

Jogo (Model)

Link: http://i.imgur.com/1BKQsIk.jpg

(1) Só mostramos os diagramas da parte Model do jogo porque iremos falar mais à frente do padrão MVC mostrando os respetivos diagramas de classe com as classes Controller e View

Descrição:

Criámos a classe Logger que é um singleton, esta classe vai guardar as informações todas que acontecem na aplicação num ficheiro.

Temos a classe GestorJogo que tem uma lista de jogadores os comparadores, os 4 rankings, o jogo, o jogador que vai jogar e o careTaker do jogo. Esta classe tem o objetivo de gerir os jogadores e os seus jogos, introduzindo-os nos rankings os resultados. Nesta fase tornámos a classe num singleton para que o seu acesso fosse mais facilitado.

A classe Jogador tem o username, pontuação, pontuação máxima, o tempo total de jogo, o número de jogos que foram jogados e o seu histórico. Representa um utilizador, tendo o objetivo de cada utilizador que jogue, crie o seu Jogador.

Temos o HistoricoJogador que tem um TADHistorico, esta classe guarda tudo o que um utilizador faz.

ElementoHistorico, esta classe é usada no histórico, sendo um elemento do histórico algo com várias características, decidimos criar esta classe. Tem o tipo de jogo, a estratégia de pontuação, a duração do jogo e o jogador.

Temos a classe Jogo que irá conter tudo o que está relacionado com um jogo como por exemplo o jogador, um conjunto aleatório e tabuleiro. Temos também um tipo diferente de jogo, logo é uma classe filha do Jogo (JogoRapido).

O jogo tem um tabuleiro que por sua vez tem várias LinhaDeJogo, onde o número destas depende do tamanho desejado que o tabuleiro tenha.

Cada LinhaDeJogo vai conter o número de elementos em cada um dos lados e a TADLinhaTres que por sua vez tem as peças da linha.

Existe também uma peça especial, logo a classe Peca tem uma classe filha chamada PecaEspecial que tem um comportamento diferente.

Fase Final:

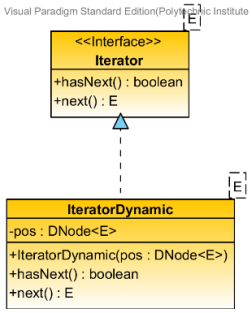
Temos o RankingJogador que tem um TADRanking, esta classe guarda as pontuações dos jogadores mas só guarda uma cópia do jogador (para não haver alterações na pontuação guardada no ranking). As variáveis como o número de jogos e a pontuação máxima são atualizadas sempre que um jogador acaba um jogo.

Criámos também a classe ListaJogador que irá guardar todas as contas de jogador criadas na aplicação.

Nota: as imagens sendo demasiado grandes para por diretamente no documento, por perder qualidade e legibilidade, decidimos fazer upload das mesmas online. Caso o link esteja o corrompido, as mesmas imagens vêm em anexo com o projeto.

Padrões Implementados

Iterator



O padrão iterator, no nosso projeto, consiste em ter um acesso sequencial aos elementos que estão guardados numa classe do tipo TAD. Este acesso é feito de forma a que não seja necessário revelar a representação do elemento guardado.

No milestone1

Na primeira fase do projeto, apesar de não nos ter sido pedido, utilizámos o iterator na classe TADRanking de modo a que pudéssemos ver as pontuações a serem adicionados e ordenadas de acordo com o que nos foi dito pela professora responsável.

No milestone2

Nesta segunda fase do projeto sentimos e necessidade de implementar este mesmo padrão para pesquisas noutras classes, de modo a que tenhamos uma persistência maior no código e afim de pudermos averiguar se o projeto está ou não a funcionar bem.

Adapter

O padrão adapter permite o uso de uma classe para adaptar o comportamento de outra, para que não seja necessário a criação de 2 classes de raiz com comportamentos ligeiramente semelhantes. Usando este padrão conseguimos com que não haja código repetido e fazemos com que o cliente no final não chegue a saber que é um adapter apesar da mesma ter o comportamento que deveria de ter.

No milestone1

Diagrama TADRanking: http://i.imgur.com/238fMuj.png

Na primeira fase do projeto, apesar de não nos ter sido pedido, utilizámos o padrão adapter para simplificar o código no TADRanking. Como já tínhamos feito em laboratório a TADListLinked, decidimos usá-la como adaptee do ranking, a fim de não repetir código e de simplificar o mesmo.

No milestone2

Diagrama Linha de Jogo: http://i.imgur.com/yhzKiYP.png
Diagrama Historico Jogador: http://i.imgur.com/YTBqoMz.png

Na segunda fase do projeto usamos este padrão para a classe LinhaJogo, fazendo de adaptee a classe TADLinhaTres e também para a classe HistoricoJogador, fazendo de adaptee a classe TADHistorico.

Fase Final

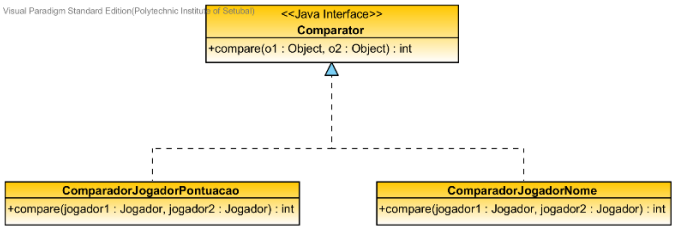
Diagrama RankingJogador: http://i.imgur.com/fhX2wBv.png

Na última fase sentimos a necessidade de criar a classe RankingJogador e para tal usámos a mesma lógica do HistoricoJogador, neste caso fazendo de adaptee a classe TADRanking.

Strategy

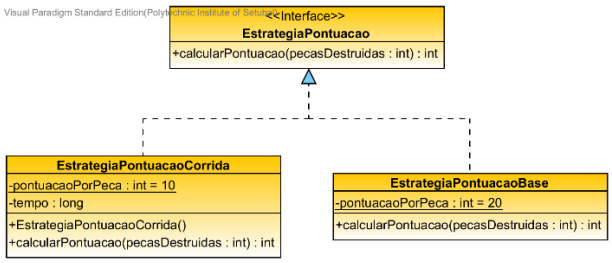
O padrão strategy é um padrão onde se fazem classes que simplesmente diferem no algoritmo, e também é feito para que a sua estrutura não esteja em exposição.

No milestone1



Na primeira fase do projeto sentimos a necessidade de usar este padrão para que o ranking pudesse ordenar as pontuações que recebia.

No milestone2



Na segunda fase do projeto usamos este padrão devido à variante de pontuação que iria existir no jogo. Onde o que variava era a existência de penalidade e o valor por peça destruída.

Abstract Factory

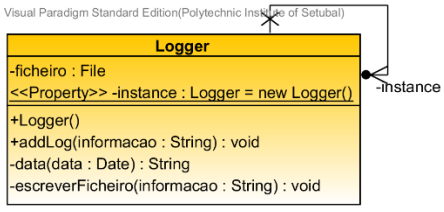
Diagrama Factory: http://i.imgur.com/ve9xsFc.png

O padrão Abstract Factory é usado para criar famílias de objetos relacionados ou interdependentes sem especificar as suas classes concretas.

Na fase final do projeto criámos as classes AbstractFactory, GestorFactory e as classes factory referentes a cada tema. O padrão foi usado de forma a que a classe AbstractFactory fosse a fábrica Abstrata, as classes referentes a cada tema são as fábricas concretas da classe AbstractFactory e a classe GestorFactory seria uma classe que iria ajudar no acesso a uma fábrica concreta específica a fim de a mesma ser usada no cliente.

Nota: no diagrama que mostramos vemos as ligações entre a AbstractFactory e os seus clientes, essa ligação é complementada com o uso do GestorFactory

Singleton



Foi sugerido no enunciado que tivéssemos um logger, com o objetivo de escrever num ficheiro tudo o que vai acontecendo desde o início da aplicação até que esta feixe.

Decidimos então usar o padrão singleton para a classe Logger, pois, esta vai estar “acima” de tudo o resto, querendo assim que a aplicação é iniciada, um logger é automaticamente criado e impedindo que não seja criada outra instância da classe Logger.

Fase Final

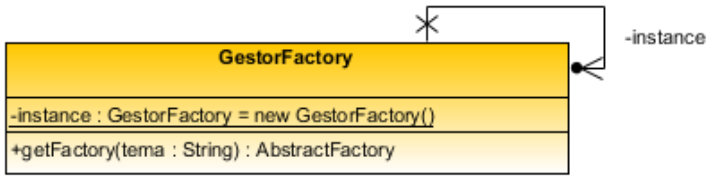


Diagrama GestorJogo como singleton: http://i.imgur.com/bmeHhB4.png

Para termos uma maior coerência de dados e uma maior facilidade de acesso tornámos a classe GestorJogo num Singleton e a classe GestorFactory também num singleton.

Observer

Diagrama Observer: http://i.imgur.com/XGILBRE.png

A classe Logger precisa ser notificada de que informação guardar no ficheiro, para este efeito escolhemos usar o padrão observer.

Decidimos o nosso jogo como Observable e criámos os nossos observers FimJogo e InicioJogo, estes vão ser notificados pelo observable em situações diferentes, no fim de um jogo e no início.

Dependendo da situação em que estes são notificados, um deles faz o seu update e diz ao Logger para escrever no ficheiro as informações que fazem sentido guardar relativamente ao cenário em que estão inseridas.

Fase Final

Usamos o padrão Observer para atualizar os Rankings quando um jogo acaba, para atualizar a Lista de Jogadores quando é criado um jogador novo, e para quando é feito uma jogada, quer esta seja uma jogada normal ou uma jogada undo (de modo a que seja atualizado a parte gráfica do jogo).

Memento

Diagrama Memento: http://i.imgur.com/b44mncW.png

Para obtermos o efeito de undo no projeto, decidimos utilizar o padrão memento como sugerido no enunciado. Utilizámos este padrão porque achámos simples e eficiente, e também porque neste caso seria muito difícil a implementação do comando.

O objetivo é dar ao utilizador a opção de retroceder múltiplas ações, podendo voltar mesmo ao início do jogo. Para tal guardamos cópias dos vários tipos de linhas, peças, as peças a jogar e as próximas peças, assim tendo cópias exatas dos vários momentos de jogo para onde retroceder.

Fase Final - atualização do diagrama Memento: http://i.imgur.com/lK80IOK.png

MVC - Implementação

Este padrão serve para separar as camadas de lógica, gráfica e de controle. Oferece também boa extensibilidade e reutilização de código, através da separação das classes de cada camada.

O estilo de padrão MVC que usamos na estrutura geral do projeto é usada de forma “JavaFXML-enabled application” mas com uma ligeira diferença. Em termos da parte de criação de jogador, por exemplo, temos o padrão Observer para atualizar e gravar a nova Lista de Jogadores, neste caso temos a classe ListaJogadores como classe Model, o Controlador FXMLLoginController e a sua View FXMLLogin feita através da aplicação JavaSceneBuilder.

Diagrama MVC1: http://i.imgur.com/RR07Han.png

Depois temos a parte do jogo, onde o utilizador interage mais na aplicação. Aqui usamos o MVC de duas formas particulares. Neste caso nós temos o controlador FXMLGameWindowController, este tem a sua View FXMLGameWindow. Neste caso, visto que queríamos que a organização das peças fosse feita de forma mais dinâmica decidimos criar essa parte do jogo sem o auxílio do JavaSceneBuilder, por isso criámos as classes JogoView, TabuleiroView e LinhaJogoView, que, tal como os nomes sugerem, são as classes que representam o jogo, o tabuleiro e as linhas de forma gráfica. Sendo classe JogoView a que contem a classe TabuleiroView, e esta as respetivas

LinhaJogoView's, tivemos que adicionar a JogoView à view FXMLGameWindow. Sendo estas três views classes independentes e visto que foram adicionadas a uma view com um controlador existente, decidimos que não seria necessário controladores para cada um.

Quando se chega à parte para jogar, é iniciado uma nova instância JogoView no controlador FXMLGameWindowController, de seguida o controlador será adicionados como Observador do jogo (classe Jogo do Model). Sempre que o utilizador faz uma jogada (método onAction que cada LinhaJogoView tem, ou voltar atrás na jogada fazendo Undo) é atualizado na classe Model e este notifica o seu observador da alteração feita, sendo este o controlador que de seguida vai ao JogoView e diz-lhe para atualizar o estado. Quando o jogo acaba, o controlador é notificado de que acabou, ele para o relógio do jogo desativa e ativa opções do menu e avisa o JogoView de que o jogo acabou.

Diagrama MVC2: http://i.imgur.com/9K414XZ.png