
Licenciatura em Engenharia Informática

Regime Pós-Laboral

# Programação Avançada

# Quatro em linha

# Entrega FINAL

## 2020/2021

Paul Bob 2016013407

# Índice

# Introdução

Este trabalho prático, realizado no âmbito da disciplina de Programação Avançada, tem como objectivo utilizar os conceitos e a matéria leccionada ao longo do semestre a fim de desenvolver uma versão em java do jogo de tabuleiro quatro em linha.

O tabuleiro está na posição vertical possui uma largura de 7 células e uma altura de 6 células. Cada jogador, alternadamente, deixa cair uma peça numa coluna à sua escolha, fazendo com que as peças se acumulem até um máximo de 6 peças por coluna (altura do tabuleiro). Ganha o jogador que conseguir colocar 4 peças seguidas em linha, na vertical, horizontal ou diagonal.

Para tornar o jogo mais entusiasmante existem várias modificações tais como turnos de mini jogos para ganhar peças especiais, “undos” e muito mais!

Como este trabalho foi desenvolvido ao longo da aprendizagem de novos conceitos, algumas decisões tomadas faziam sentido quando foram tomadas mas atualmente seriam repensadas e simplificadas.

NOTA: Visto que este projeto teve duas metas de entrega mantive o relatório da primeira meta adicionando apenas o essencial feito na implementação da segunda meta ficando assim o relatório do projeto completo.

# Decisões tomadas

A primeira decisão tomada foi a validação do nome, caso o segundo jogador insira um nome igual ao primeiro estes ficam com o mesmo nome diferenciando o segundo jogador por uma concatenação de "_2" no nome. Ainda durante a preparação, depois de serem registados ambos os jogadores é feito um sorteio de 50% para ver qual o jogador que começa.

Decidi que o tabuleiro seria uma matriz bidimensional de inteiros sem pensar muito no assunto e mais tarde arrependi-me visto que poderia ter usado várias das funções de pesquisa que a linguagem java disponibiliza.

Devido a falta de tempo e da implementação do memento e de toda a parte das gravações na noite anterior à entrega omiti alguns estados que poderiam ter deixado esta parte mais legível e elegante. Infelizmente tive que improvisar e a função que trata da interface para a preparação ficou um bocado sobrecarregada.

Decidi usar memento para os "undos" apesar de não ter sido estritamente necessário. Esta decisão foi tomada mais para praticar / estudar o conceito de memento lecionado durante algumas aulas.

Tanto o CareTaker como o Originator estão implementados em classes já utilizadas para controle, evitando assim uma lista desnecessária de métodos "repetidos".

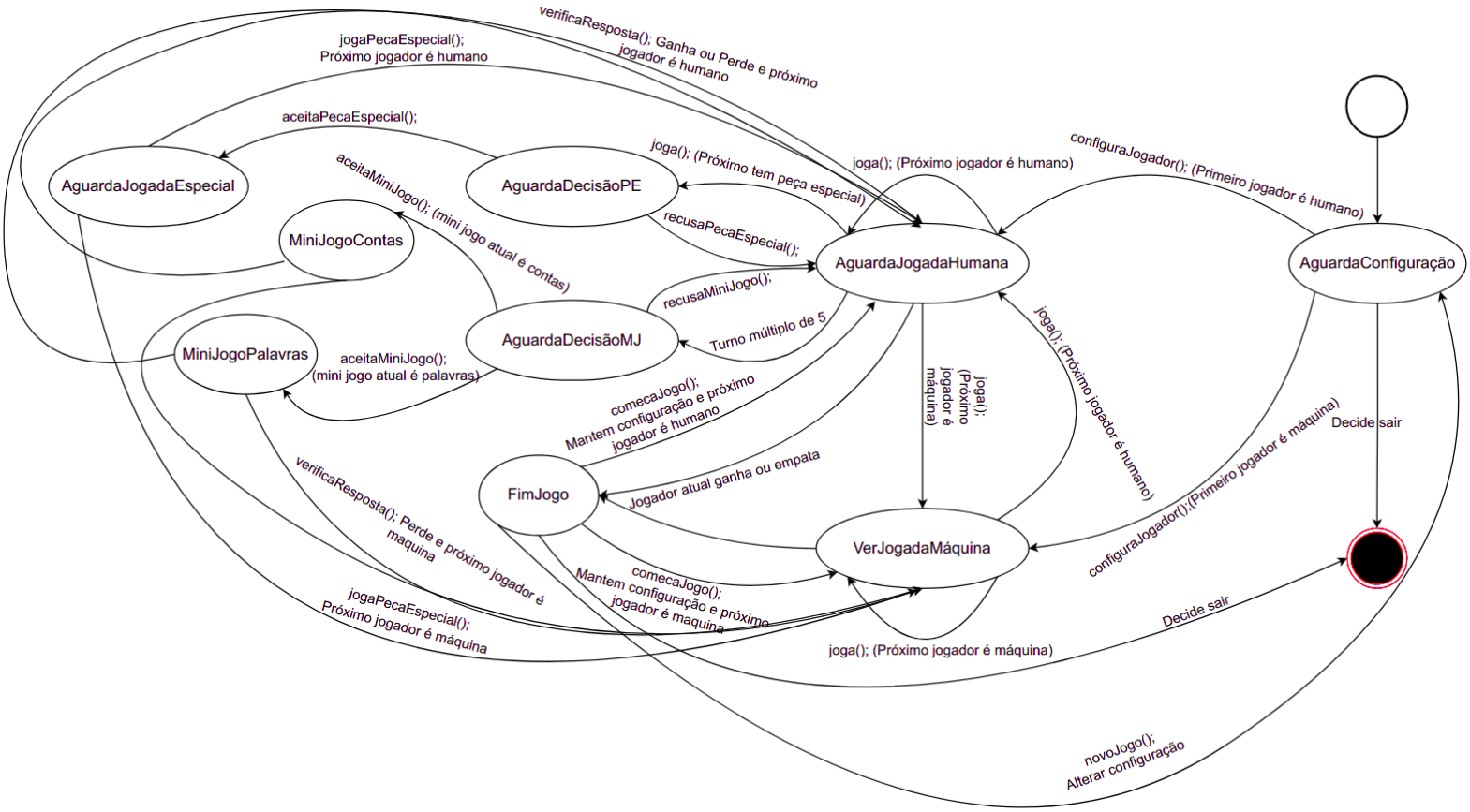
A parte do "undo" omite alguns detalhes no enunciado pelo que tomei a liberdade de manter a vez do jogador que faz o "undo" independentemente das vezes que o faz. Para manter os jogadores sempre atualizados quebrei um bocado o encapsulamento na função setMemento(); implementada na classe "ContextoMaqEstados".

Para os replays decidi adicionar uma string num "ArrayList <String>" e ir imprimindo esse array ao longo dos pedidos do utilizador.

Até aqui todas as decisões foram escolhidas para a primeira meta com interface de texto, agora irei descrever as decisões tomadas aquando da implementação gráfica.

Os replays foram substituídos para melhor na interface gráfica. Agora para ser feito o replay carrego o objeto serializado e inverto um array deque de mementos que vai "mostrando" o próximo memento sempre que o utilizador carrega na tecla enter. Para isso ser possível tive que adicionar um novo estado "Replay" no qual a máquina de estados fica "presa" do início ao final do replay sem possibilidade de sair do estado, apenas no fim do replay. Visto que é um estado isolado e controlado externamente não irei refazer o diagrama de estados. A máquina mantém-se a mesma , sendo forçada a ignorar certos estados na interface gráfica, tais como o AguardaDecisãoPE. Isto deve-se ao facto de esse estado não ser possível visto que a peça especial vai ser jogada a partir dum clique num botão.

# Diagrama de estados




Tentei fazer o diagrama de estados o mais explícito possível apesar de estar ciente das suas falhas e fraquezas. Tentei reduzir ao máximo o número de estados, resultando isso em muitas verificações pouco elegantes antes da mudança da maioria dos estados.

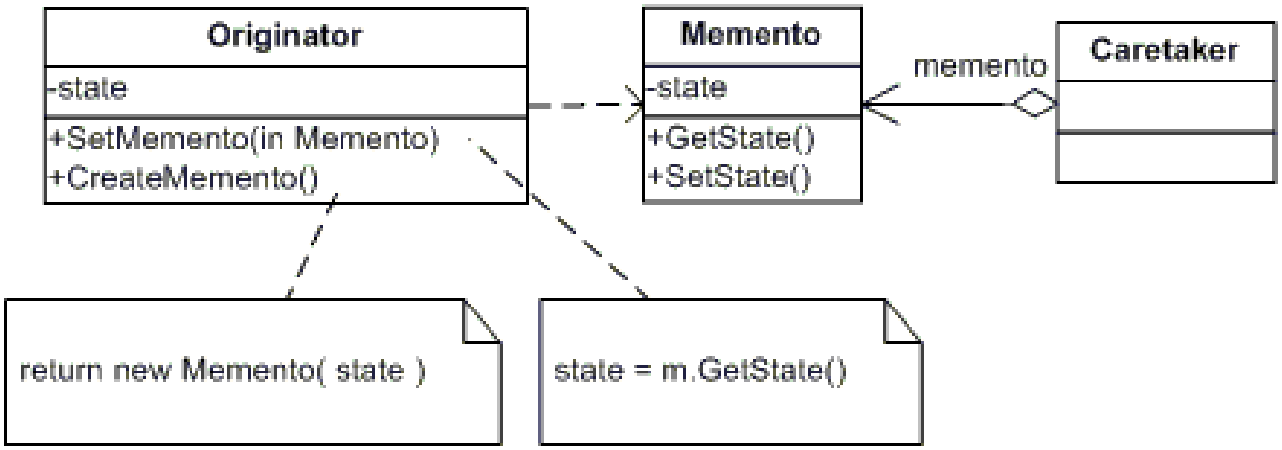
O diagrama foi sofrendo alterações ao longo que a implementação do trabalho foi decorrendo, pelo que é possível que alguns dos pormenores me tenham escapado.

Com a implementação da GUI existem estados que deixaram de fazer muito sentido mas para não fazer grandes alterações na máquina nesta altura decidi forçar a mudança do estado AguardaDecisaoPE para recusar sempre a peça especial e quando o jogador clica na interface gráfica no botão para jogar a peça especial forço a máquina a mudar para AguardaJogadaEspecial.

Aprendi com isto alguns pormenores da máquina de estado e certamente a próxima que irei implementar ( em outros projetos ) será bem mais consistente e sem estas falhas.

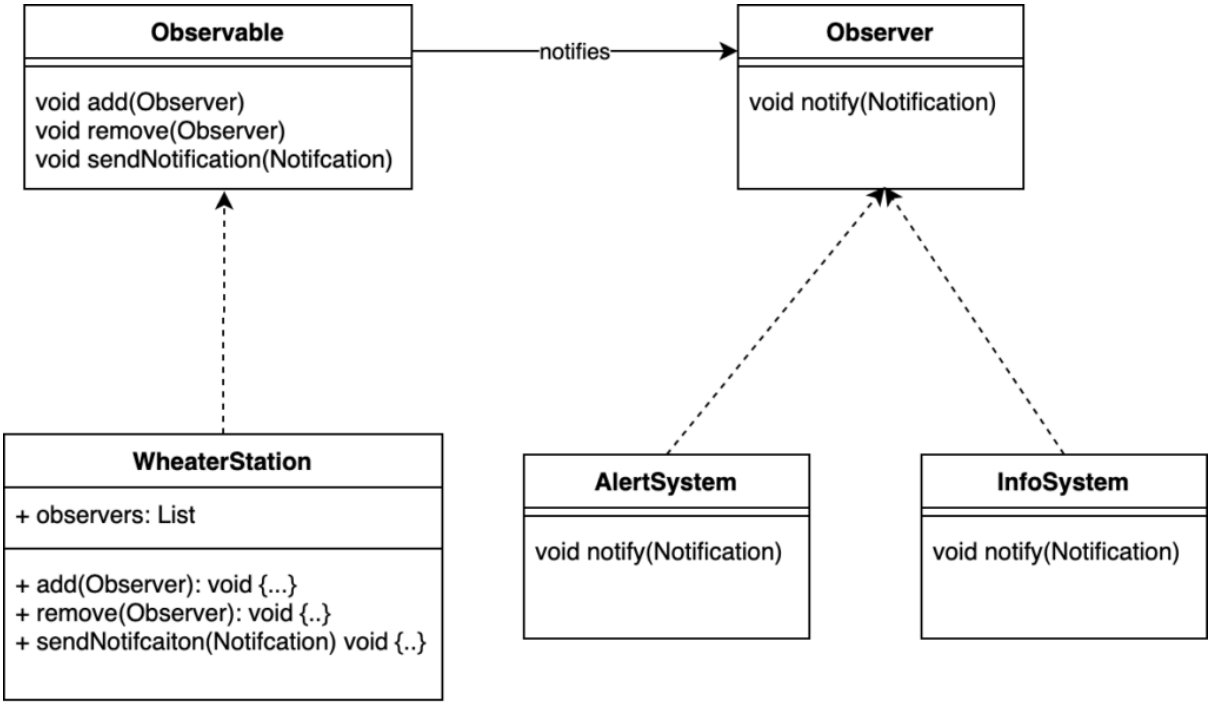
# Diagrama de outros padrões

Como referido anteriormente tomei a decisão de usar o padrão memento para a implementação da funcionalidade de “undo”. Não há muito por onde inventar, pelo que deixo o diagrama pelo qual me guiei na implementação.




Para evitar camadas de métodos “repetidos” o Originator e o Caretaker são interfaces implementadas na classe “ContextoMaqEstados” e “GestaoJogo” respetivamente.

Implementei ainda para a GUI o paradigma de observer observable que tem o seguinte aspecto:




Este padrão serve para “desencadear” uma série de ações em classes observer distintas que se “inscreveram” numa classe chamada de observable. No meu projeto implementei isso de forma ligeira e um pouco desorganizada visto que é o meu primeiro contacto com este padrão mas serviu para aprender devidamente o seu funcionamento.

# Descrição das classes

## iu

### texto

IUTexto -> Interface de texto, trata do Output e do Input do utilizador para o manter sempre informado. Tentei proteger ao máximo contra inputs inválido mas certamente tem alguns pontos menos protegidos.

### gui

#### resources

*css*

ficheiros .css utilizados para afinar os estilos utilizados na GUI

*fonts*

fontes externas utilzadas para embeleazr a GUI

*images*

imagens utilizadas em algumas partes da GUI

*sounds*

sons utilizados na GUI, neste momento existe apenas um.

CSSManager -> Classe encarregue de carregar e adicionar os estilos dos ficheiros .css

FontsManager -> Classe que trata de carregar fontes externas do ficheiro fonts/…

ImageLoader -> Classe que carrega imagens externas utilizadas na GUI, tem uma cache utilizando um HashMap para não sobrecarregar o programa com pedidos para importar imagens externas.

MusicPlayer -> Classe encarregue de adicionar e manipular ficheiros de áudio durante a execução do programa.

Resources -> Classe utilizada por todos os managers acima indicados para obter um determinado recurso .

#### stageMiniJogos

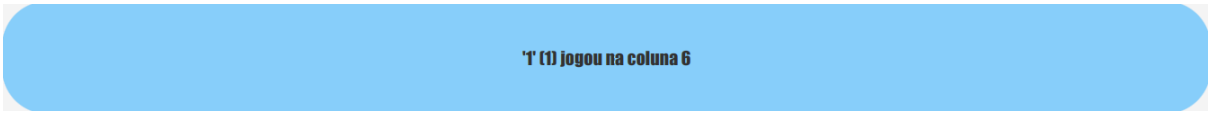
StageContas -> Classe que é chamada quando um jogador aceita o mini jogo das contas, é um pequeno stage que mostra uma string e obtém o input do utilizador.

StagePalavras -> Classe chamada quando um jogador aceita o mini jogo das palavras, é um pequeno stage que mostra uma string e obtém o input do utilizador.


stagePrincipal

GuiaPane -> Um BorderPane que fica no topo do PaneOrganizer, servindo para guiar os jogadores.

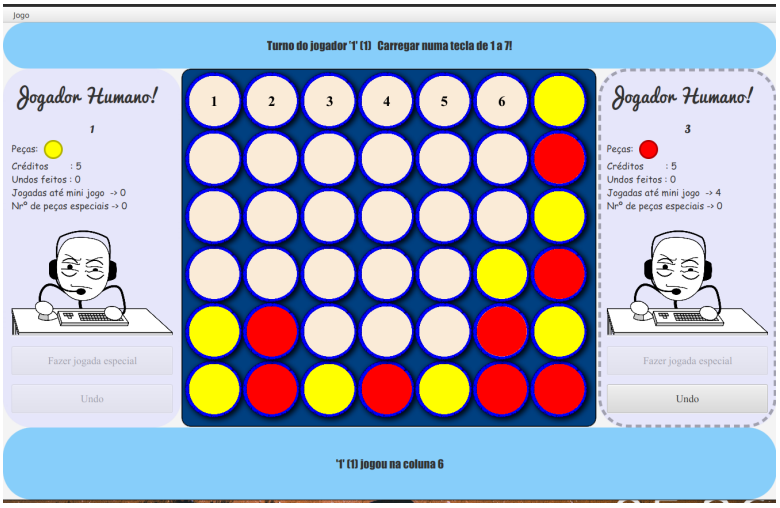

InfoPane -> BorderPane que fica em baixo do PaneOrganizer, tendo como objetivo informar os jogadores do resultado das suas ações.




menuBar-> Bara de menu com um menu que serve para iniciar um novo jogo / carregar um jogo de memória/ salvar um jogo atual ou ver replay de um dos últimos 5 jogos concluidos.

PaneJogador-> Uma VBox que serve para mostrar todos os dados de um jogador registado. Existem 2 no PaneOrganizer, uma a esquerda e outra a direita.

PaneOrganizer-> BorderPane que engloba toda a GUI, tendo no topo uma VBox com o menuBar e com o GuiaPane, no centro o tabuleiro, dos lados os jogadores e em baixo o InfoPane.



PaneRegisto-> Uma VBox que serve para receber input de um jogador que se quer registar. Existem 2 no PaneOrganizer, uma a esquerda e outra a direita apenas sendo apresentados na fase de preparação.

Tabuleiro-> GridPane que representa o tabuleiro do jogo 4 em linha, atualizando sempre que existe alguma interação com os jogadores, mostrando peças amarelas para o jogador da esquerda e peças vermelhas para o jogador da direita.

## logica

### dados

Jogador -> Classe que guarda e manipula alguns dos dados dos jogadores ao longo do progresso do jogo.

Jogo -> Classe que trata de toda a lógica do jogo, por vezes excessivamente. Tem algumas responsabilidades que poderiam ser melhor redistribuídas.

MiniJogoContas -> Classe que permite o funcionamento da lógica do mini jogo das contas.

MiniJogoPalavras -> Classe que permite o funcionamento da lógica do mini jogo das palavras.

### estados

AguardaConfiguracao -> Estado onde o utilizador escolhe se deseja ver replays, se deseja começar jogo guardado ou se decide começar um novo jogo e insere os dados dos jogadores.

AguardaDecisaoMJ -> Estado onde o utilizador escolhe se aceita ou rejeita o mini jogo atual.

AguardaDecisaoPE -> Estado onde o utilizador escolhe se joga ou não uma peça especial.

AguardaJogadaEspecial -> Estado onde o utilizador escolhe a coluna para jogar a peça especial.

AguardaJogadaHumana -> Estado onde o utilizador é humano e escolhe a coluna onde jogar ou sair ou fazer undo.

FimJogo -> Estado que aguarda o utilizador decidir jogar outro jogo e se sim manter ou não a mesma configuração de jogadores.

JogoEstado -> Interface de estados.

JogoEstadoAdapter -> Classe abstrata que implementa a interface JogoEstado.

MiniJogoContas -> Estado durante o qual o jogador vai interagindo com o mini jogo das contas.

MiniJogoPalavras -> Estado durante o qual o jogador vai interagindo com o mini jogo das palavras.

Replay -> Estado no qual a máquina fica durante todo o replay, no final muda para o estado FimJogo aguardando decisões do utilizador.

VerJogadaMaquina -> Estado para fazer paragem antes da máquina jogar para fornecer uma melhor experiência ao utilizador.

### memento

ICareTaker -> Interface que contém os métodos necessários para salvar e fazer undo de um determinado estado.

IMementoOriginator -> Interface que contém os métodos necessários para que a classe que implementa o CareTaker funcione.

Memento -> Classe onde o estado é serializado.

## logica

ContextoMaqEstados -> Classe que tem todos os métodos a ser redirecionados para a máquina de estados propriamente dita. Além disso tem alguns gets essenciais ao bom funcionamento do programa.

GestaoGravacoes -> Classe com métodos estatísticos para permitir fazer gravações e leitura de ficheiros em memória.

GestaoJogo -> Classe conhecida e intermediária entre a interface e a máquina de estados.

JogoObs -> Classe observable que tem como principal função encaminhar os comandos para o GestaoJogo e lançar ProprietyChanges.

JogoSituacao -> Classe de enumerações que permite a interface apresentar o output pretendido de acordo com cada estado atual.

# jogo

Main -> É aqui que tudo começa.
MainJFX -> Para a GUI temos que começar por aqui.

# Descrição do relacionamento entre as classes

Visto que o foco principal do trabalho não era propriamente repetir matéria de POO desleixei-me um bocado no que toca a orientação a objetos. Existem algumas classes sobrecarregadas de responsabilidades que poderiam ser evitadas pela implementação de alguma herança.

A minha classe Main cria uma Classe GestaoJogo e esta é reencaminhada para a interface, ficando a interface a agregar o gestor.

A classe GestaoJogo compõe uma classe ContextoMaqEstados intitulada por originator.

O originator compõe dois objetos, um da classe JogoEstado outro da classe Jogo.

O JogoEstado é uma interface com os métodos dos vários estados, tal como apresentada nas aulas, não tenho muito a acrescentar. A classe JogoEstadoAdapter implementa os métodos da interface derivando depois para as várias classes que representam cada uma um estado diferente.

A classe Jogo compõe 5 objetos, 3 Jogadores e 2 MiniJogos, ficando sobrecarregada de alguma lógica desses objetos.

Para a implementação com a interface gráfica o paradigma é semelhante com as seguintes diferenças:

A MainJFX cria um GestaoJogo, um JogoObs , um PaneOrganizer e uma Scene.

A classe JogoObs agrega o GestaoJogo e por sua vez o PaneOrganizer agrega o JogoObs.

Inicialmente o PaneOrganizer é composto por 2 PaneRegisto um Tabuleiro, um InfoPane, um menuBar e um GuiaPane, alterando os PaneRegisto por PaneJogadores depois da preparação estar concluída.

# Cumprido / Implementado

Cumpri e implementei tudo o que era pedido no enunciado implementando algumas coisas extra também. Algumas decisões não explícitas foram tomadas por mim e serão discutidas na defesa deste projeto.

# Conclusão

Partes da implementação foram propositalmente omitidas para não tornar o relatório desnecessariamente longo e maçador.

Com a realização deste projeto tive a oportunidade de aplicar e conhecer melhor o conceito de máquina de estados, memento, entre outros lecionados ao longo do semestre.

Certamente contribuiu bastante para alargar os meus horizontes no mundo da programação.

Foi um dos meus primeiros projetos de interface gráfica pelo que adorei fazê-lo e ver o resultado.