



UNIDADE CURRICULAR: Programação por Objetos

CÓDIGO: 21093

DOCENTES: Jorge Morais e Leonel Morgado (professores) e José Félix Póvoa e Rúdi Gualter (tutores)

A preencher pelo estudante

NOME: Luís Carlos Crispim Pereira

N.º DE ESTUDANTE: 2300163

CURSO: LEI – Licenciatura em Engenharia Informática

DATA DE ENTREGA: 06/02/25

Intrdução:

O meu projeto para a unidade curricular de Programação por Objetos consiste num jogo de cartas Blackjack (21) desenvolvido em Python, utilizando o IDE PyCharm. O projeto foi dividido em três fases, conforme especificado pelo docente da unidade.

Na primeira fase (efolioA), foi criado apenas o esqueleto do jogo, com a definição das classes e da estrutura básica.

Na segunda fase (efolioB), o desenvolvimento do jogo foi avançado, tornando-o mais completo e funcional. A organização do código foi significativamente aprimorada, com a separação das classes em ficheiros distintos (.py), o que favoreceu a modularidade e a clareza. Foi implementado um menu interativo, permitindo ao utilizador escolher entre as opções de jogar, visualizar as classificações, configurar o número de jogadores e o nível de dificuldade, além de sair do jogo. Todas as regras foram implementadas de modo a tornar o jogo completamente funcional.

Na terceira fase (efolioG, fase atual), o ambiente gráfico foi desenvolvido utilizando a biblioteca Tkinter, proporcionando uma interface interativa para o utilizador. A implementação do Tkinter foi escolhida de modo a alterar o mínimo possível do código e da lógica desenvolvidos nas fases anteriores, garantindo uma transição suave para a interface gráfica. O jogo agora suporta interações com o rato, facilitando a navegação entre menus e a tomada de decisões durante a partida. Mantendo a coerência com as versões anteriores, as cartas continuam a ser representadas em ASCII, embora pudessem ter sido implementadas com imagens e outra biblioteca.

Foram adicionadas animações de áudio básicas, mas, devido ao modo multiplayer e à organização escolhida, não foi possível incluir efeitos sonoros para todas as funções de vitória e derrota. No entanto, a minha previsão inicial sobre a dificuldade de manter o modo multiplayer nesta implementação revelou-se infundada, pois consegui implementá-lo sem complicações significativas. A maior dificuldade nesta implementação deveu-se à organização original do código, que foi estruturado para ser executado de forma sequencial, tornando desafiador adaptá-lo para ocorrer numa única janela Tkinter, como optei por manter o código "original" inalterado, todas as interações de texto continuam a ocorrer em janelas novas, garantindo que a lógica do jogo se mantivesse fiel às versões anteriores.

O layout foi cuidadosamente organizado para garantir uma experiência intuitiva e envolvente. O sistema de classificações também foi aprimorado, permitindo visualizar agora além do anteriormente implementado a criação de um gráfico. Esta fase conclui o desenvolvimento do projeto, garantindo um jogo funcional, modular e bem estruturado.

Nota: Foram criados dois Easter Eggs no jogo no efolioB e mantidos aqui em efolioG: um na dificuldade impossível, onde o Dealer atinge sempre 21, e outro ao utilizar o nome de jogador "Kaxeszer", garantindo que este obtenha sempre 21. A escolha deste pseudónimo deve-se ao facto de ser o meu próprio alias, garantindo assim que as classificações me mantenham sempre em 1º lugar. Já a dificuldade impossível foi adicionada simplesmente por diversão.

Implementação Técnica:

1. Alterações Gerais

- Integração com Tkinter: Adicionado um novo sistema de interface gráfica sem alterar a lógica principal do jogo.
- Novo módulo sound.py: Introduzido para suportar sons básicos (embora não tenha sido possível incluir áudio para todas as interações).
- Adaptação para GUI: As interações que antes ocorriam na consola agora foram convertidas para janelas separadas.

2. Alterações nas Classes:

- a. Classe Card (Sem grandes alterações)
 - Mantém os atributos e métodos originais.
 - Continua a ser representado em ASCII, apesar da interface gráfica.
- b. Classe Deck (Pequena alteração para adaptação gráfica)
 - O método deal_card() agora precisa atualizar a interface gráfica sempre que uma nova carta é distribuída.
- c. Classe Player (Sem grandes alterações, mas ajustes para GUI)
 - Métodos continuam semelhantes:
 - Apesar da previsão inicial de dificuldades, a implementação do multiplayer manteve-se estável.
- d. Classe Game (Alterações significativas)
 - start_game(): Agora precisa controlar a interface gráfica além da lógica do jogo.
 - player_turn(): Modificado para interagir com a GUI em vez da consola.
 - dealer_turn(): Agora exhibe visualmente a ação do dealer.
 - exibir_resultados(): Atualiza a interface gráfica em vez de imprimir no terminal.
- e. Classe Classificacoes (Aprimoramentos na exibição)
 - Mantém o armazenamento em JSON.
 - Nova funcionalidade: Implementação opcional para mostrar estatísticas em gráfico.

3. Módulos

- a. sound.py (Novo)
 - Controla sons básicos do jogo.
- b. screen.py (Modificado)
 - Controla a renderização de janelas Tkinter.
 - Gere elementos visuais do jogo.
- c. menu.py (Modificado)
 - Agora exhibe opções através da interface gráfica (Tkinter) em vez do terminal.
- d. game.py (Modificado)
 - Adaptação do fluxo do jogo para suportar a interface gráfica.
 - start_game(), player_turn() e dealer_turn() foram ajustados para atualizar a interface Tkinter.
 - Adição de verificações para atualizar visualmente as cartas e ações.

- e. `deck.py` (Modificado)
 - Mantém a estrutura do baralho e a lógica de embaralhamento.
 - O método `deal_card()` agora notifica a interface gráfica.
- f. `card.py` (Sem alterações)
 - Mantém a representação das cartas em ASCII.
- g. `player.py` (Modificado)
 - Mantém a estrutura original da classe `Player`.
 - `show_hand(reveal_dealer=False)`: Modificado para atualizar a interface gráfica.
 - Mantém compatibilidade com o multiplayer sem alterar a lógica.
- h. `utils.py` (Modificado)
 - Agora inclui algumas funções auxiliares para suportar a interface gráfica.
 - Mantém a modularidade sem alterar a lógica central.
- i. `classificacoes.py` (Modificado)
 - Melhor organização e estrutura para suportar exibição gráfica.
 - Permite visualizar estatísticas detalhadas dos jogadores.
 - Possibilidade de gerar gráficos comparativos (caso implementado).

4. Diagrama de dependências:



Apenas acrescentado o novo modulo `sound.py`

Todos os módulos restantes mantiveram a mesma estrutura e funcionamento.

TRABALHO / RESOLUÇÃO:

1.a)

No meu projeto, o conceito de polimorfismo foi aplicado através da herança entre as classes Player, JogadorHumano e Dealer.

Classes envolvidas:

- Player (classe base)
- JogadorHumano (subclasse de Player)
- Dealer (subclasse de Player)

Métodos polimórficos:

- take_action():
 - Definido na classe Player, mas sobrescrito em JogadorHumano e Dealer para comportamentos distintos.
 - Em JogadorHumano, permite entrada do utilizador.
 - Em Dealer, segue lógica automática baseada nas regras do jogo.
- show_hand(reveal_dealer=False)
 - Comportamento diferenciado na classe Dealer, ocultando a primeira carta do baralho enquanto revela todas nas outras instâncias de Player.

No código, os comentários # Resposta 1a foram adicionados nos métodos envolvidos.

1.b)

O polimorfismo foi fundamental no projeto, permitindo que a lógica do jogo se mantivesse flexível e extensível sem a necessidade de verificações constantes para diferenciar um jogador humano do dealer. Através da sobrescrita dos métodos take_action() e show_hand(), foi possível garantir que cada tipo de jogador executasse as suas ações de forma independente, evitando condicionais desnecessárias (if/else). O método take_action() foi redefinido na subclasse JogadorHumano para permitir interações diretas do utilizador, enquanto na subclasse Dealer seguiu uma lógica automatizada baseada nas regras do jogo. Da mesma forma, show_hand() garantiu que a primeira carta do dealer ficasse oculta sem necessidade de criar um método específico para isso. Esta abordagem não só manteve o código mais modular e organizado, como também facilitou futuras expansões, permitindo a criação de novos tipos de jogadores sem necessidade de alterar a estrutura base do jogo. Caso fosse necessário adicionar um bot com estratégia própria, bastaria criar uma nova subclasse de Player e sobrescrever take_action() novamente, demonstrando como o polimorfismo simplifica a reutilização de código e melhora a escalabilidade do projeto. Sem esta abordagem, o código dependeria excessivamente de estruturas condicionais, tornando-se menos eficiente e mais difícil de manter.

1.c)

```
class BotPlayer(Player): # Classe que representa um jogador controlado por IA - # Resposta 1c
```

```
    def take_action(self): # Resposta 1c
        return "P" if self.calculate_hand_value() < 17 else "F"
```

Classe implementada, mas a não ser utilizada. Mecânica parecida com a do Dealer já implementado.

```
    def exibir_nome(self): # Resposta 1c
        return f"Dealer ({self.name})"
```

ou

```
    def exibir_nome(self): # Resposta 1c
        return self.name
```

Implementado, e a funcionar em código, chamando o nome dos jogadores após a inserção.

1.d)

A utilidade deste exemplo de polimorfismo reside na automatização das decisões do BotPlayer, reduzindo o esforço de programação e garantindo que o jogo possa ser expandido sem necessidade de modificações na lógica principal. Ao sobrescrever o método `take_action()`, permitimos que um jogador controlado pela IA tome decisões sem intervenção do utilizador, seguindo uma estratégia simples baseada no valor da sua mão. Esta implementação evita a necessidade de condicionalismos manuais (if para verificar se o jogador é um bot ou humano) dentro da lógica do jogo, tornando o código mais modular e escalável. Se o projeto evoluir para suportar diferentes níveis de dificuldade para bots, bastará criar subclasses de BotPlayer com variações no método `take_action()`, sem tocar no código existente. Isto facilita a adição de bots mais inteligentes no futuro sem comprometer a estrutura do jogo. Além disso, melhora a organização ao separar claramente a lógica dos jogadores humanos e dos bots, evitando código redundante.

2.a)

No meu projeto, escolhi as bibliotecas `pygame` e `matplotlib` para o tópico 7, sendo utilizadas para melhorar a experiência do utilizador através de som e visualização de classificações.

A biblioteca `pygame` foi utilizada para adicionar efeitos sonoros no jogo, melhorando a imersão do jogador. Foi implementada no ficheiro `sound.py`, onde gere a reprodução de sons em eventos como vitórias e derrotas.

A biblioteca `matplotlib` foi utilizada para gerar gráficos das classificações dos jogadores, permitindo uma representação visual do desempenho ao longo do tempo. Esta funcionalidade foi integrada no ficheiro `classificacoes.py`, onde os dados armazenados são processados e exibidos de forma mais intuitiva.

No código, os comentários # Resposta 2a foram adicionados nas seções onde estas bibliotecas são importadas.

2.b)

A integração da biblioteca pygame exigiu a criação do módulo sound.py, onde toda a gestão do áudio foi centralizada. Esta adaptação evitou que múltiplas partes do código tivessem chamadas diretas ao pygame.mixer, garantindo maior organização e facilitando futuras modificações no sistema de som.

Já a biblioteca matplotlib foi integrada no ficheiro classificacoes.py, onde foi adicionada uma nova funcionalidade para transformar os dados das classificações em gráficos. Antes, as classificações eram apresentadas apenas em formato de texto, e esta adaptação exigiu uma reestruturação para recolher, processar e representar visualmente os dados.

Se no futuro o uso destas bibliotecas for expandido, um desafio poderá ser a otimização do desempenho, principalmente na reprodução simultânea de sons ou na geração dinâmica de gráficos com muitos jogadores. Para mitigar isso, poderia ser implementado caching dos gráficos gerados ou um sistema de threads para gerir os sons sem bloquear a execução do jogo.