Entrega Final - Jogo de War

Gabriel Ripper, Gustavo Lauria, Henrique Martine, Marcelo Valentino

O Documento abaixo serve para ilustrar algumas das escolhas e implementações que foram feitas para chegarmos até essa entrega. Alguns pontos de destaque:

1. **Padrões Gof (Gang of Four)**
2. **Framework Utilizado - Pygame**
3. **Padrões GRASP**
4. **Princípios SOLID**
5. **LPS**
6. **Detalhes adicionais - Implementação**

Padrões Gof (Gang of Four)

Padrão State

O padrão **State** permite que um objeto altere seu comportamento quando seu estado interno muda. Esse padrão é útil para encapsular o comportamento associado a um determinado estado de um objeto e permite que o objeto mude seu comportamento quando seu estado muda.

```python
class UIState:
    def handleState(self, game_ui):
        pass
```

Aqui, temos uma interface *'UIState'* que define o método *'handleState'* que deve ser implementado por todos os estados concretos. Essa interface permite que diferentes estados implementem seu próprio comportamento.

```python
class InactiveState(UIState):
    def handleState(self, game_ui):
        print("hiding UI")
        game_ui.selectableTroops.remove_items(list(ma
        game_ui.selectableTroops.disable()
        if game_ui.blitzButton.is_enabled:
            game_ui.blitzButton.disable()
        game_ui.phase = 'Inactive'
```

O estado *'InactiveState'* implementa o método *'handleState'* para lidar com o comportamento específico do estado inativo. Outros estados concretos *(AttackState, MoveState, DeployState)* seguem o mesmo padrão, cada um implementando seu comportamento específico.

```python
def setPhase(self, phase: str):
    if phase == 'Inactive':
        self.state = InactiveState()
    elif phase == 'Attack':
        self.state = AttackState()
    elif phase == 'Move':
        self.state = MoveState()
    elif phase == 'Deploy':
        self.state = DeployState()

    self.state.handleState(self)
```

'GameUI' é o contexto que mantém uma referência para um objeto *'UIState'* que representa o estado atual. O método *'setPhase'* permite alterar o estado atual e delegar o comportamento para o novo estado.

Benefícios do Padrão State

1. **Encapsulamento de Comportamento:** Cada estado encapsula o comportamento específico de um estado particular, evitando longos condicionais no código e melhorando a manutenção e legibilidade.
2. **Facilidade de Adição de Novos Estados:** Novos estados podem ser adicionados sem modificar as classes existentes. Basta criar uma nova classe que implementa *'UIState'* e adicionar a lógica de transição no método *'setPhase'*.
3. **Flexibilidade:** Permite que o objeto *'GameUI'* altere seu comportamento dinamicamente ao mudar de estado.
4. **Responsabilidade Única:** Cada estado concreto é responsável por um comportamento específico, seguindo o princípio de responsabilidade única.

Padrão Observer

O padrão **Observer** define uma dependência de um-para-muitos entre objetos, de modo que quando um objeto muda de estado, todos os seus dependentes são notificados e atualizados automaticamente. Esse padrão é útil para implementar sistemas de eventos ou sistemas onde mudanças em um objeto precisam ser refletidas em outros objetos.

```
class OptionsLabel(Observer):

    def update(self, value):
        self.text_render = self.font.render(self.label + ":"
```

A classe *'OptionsLabel'* herda de *'Observer'* e implementa o método update. Isso significa que *'OptionsLabe'* é um observador que pode ser registrado para receber notificações de mudanças do sujeito que está observando. Quando o método *'update'* é chamado com um novo valor, *'OptionsLabel'* atualiza a renderização do texto para refletir o novo valor.

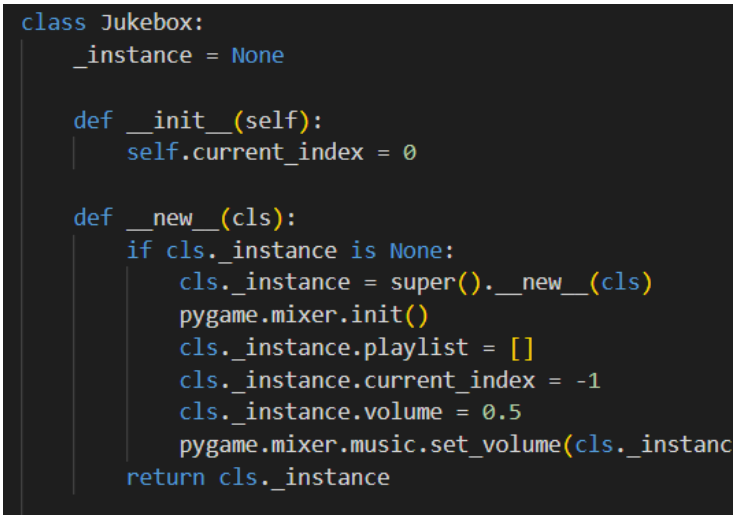
Benefícios do Padrão Observer

1. **Desacoplamento:** O padrão Observer desacopla o sujeito dos seus observadores. Isso significa que o sujeito não precisa saber quem são seus observadores ou o que eles fazem com as notificações.
2. **Flexibilidade:** Novos observadores podem ser adicionados a qualquer momento sem modificar o sujeito. Isso permite uma flexibilidade significativa na expansão do sistema.

3. **Reatividade:** Observadores são notificados e atualizados automaticamente quando o sujeito muda de estado, permitindo um comportamento reativo e em tempo real.

Padrão Singleton

O padrão **Singleton** garante que uma classe tenha apenas uma única instância e fornece um ponto global de acesso a essa instância. Esse padrão é útil quando precisamos controlar o acesso a alguns recursos compartilhados, como conexões de banco de dados, configurações de aplicativos, ou um tocador de música.



```
class Jukebox:
    _instance = None

    def __init__(self):
        self.current_index = 0

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
            pygame.mixer.init()
            cls._instance.playlist = []
            cls._instance.current_index = -1
            cls._instance.volume = 0.5
            pygame.mixer.music.set_volume(cls._instanc
        return cls._instance
```

A classe Jukebox implementa o padrão **Singleton**:

- Atributo de classe *'_instance'* é usado para armazenar a instância única da classe 'Jukebox'.
- O método *'__new__* ' é redefinido para controlar a criação da nova instância da classe. Se *'_instance'* for *'None'*, uma nova instância é criada e inicializada com a configuração padrão, caso contrário, a instância existente é retornada.

Embora o método *'__init__'* esteja presente, ele não controla a criação da instância. Ele é chamado toda vez que uma instância é criada, o método *'__new__'* garante que apenas uma instância é criada.

Benefícios do Padrão Singleton

1. **Controle de acesso único:** Garante que apenas uma instância da classe Jukebox seja criada, controlando o acesso ao recurso de música.
2. **Ponto de acesso global:** A única instância é acessível globalmente, permitindo que qualquer parte do código interaja com o tocador de música de maneira consistente.

3. **Economia de recursos:** Previne a criação de múltiplas instâncias que podem consumir recursos desnecessários, como múltiplas inicializações do mixer de áudio.

Pygame

Pygame é um framework que fornece um conjunto de módulos e funções que ajudam no desenvolvimento de jogos e aplicações multimídia.

1. **Estrutura e Organização:** Pygame oferece uma estrutura organizada e modular para o desenvolvimento de jogos, incluindo módulos para gráficos, som, eventos, entrada de usuário e muito mais. Isso permite seguir uma abordagem consistente e padronizada para criar jogos.
2. **Funcionalidades Predefinidas:** Pygame fornece muitas funcionalidades predefinidas, como renderização de imagens, manipulação de áudio, gerenciamento de eventos, detecção de colisão, e manipulação de entrada do usuário (teclado, mouse, etc.). Esses componentes ajudam os desenvolvedores a se concentrar mais na lógica do jogo em si, em vez de se preocupar com detalhes de baixo nível.
3. **Reutilização de Código:** Como um framework, Pygame promove a reutilização de código através de suas funções e módulos prontos para uso. Isso reduz o tempo de desenvolvimento e facilita a manutenção do código.
4. **Comunidade e Recursos:** Pygame tem uma grande comunidade de desenvolvedores que contribuem com tutoriais, exemplos de código, e bibliotecas adicionais. Isso cria um ecossistema robusto que apoia os desenvolvedores, similar a outros frameworks populares.
5. **Flexibilidade e Extensibilidade:** Embora Pygame forneça muitas funcionalidades prontas, ele também permite que os desenvolvedores estendam e personalizem a biblioteca conforme necessário para atender às necessidades específicas de seus jogos ou aplicativos.

Inversão de Controle e Fluxo de Execução

Eventos do Pygame:

- A função **'onEvent'** reage a eventos específicos, como *'pygame.QUIT'* para sair do jogo ou *'pygame.MOUSEBUTTONDOWN'* para detectar cliques do mouse. Isso permite que o Pygame controle quando o código do jogo deve reagir aos eventos, invertendo o controle típico onde o código do jogo chamaria funções para verificar o estado dos dispositivos de entrada.

Renderização e Atualização de UI:

- A função *pygame.display.flip()* é chamada para atualizar a tela do jogo com as alterações feitas durante a renderização. Isso coloca o controle da atualização da tela nas mãos do Pygame.

Controle do Loop Principal:

- O loop principal do jogo *'(while(self.running))'* é uma estrutura típica de jogos, onde o Pygame controla o ritmo do jogo através de *'self.clock.tick(60)'*, limitando o loop para 60 frames por segundo. Isso delega o controle do tempo de execução ao Pygame.

Atualização de Sprites:

- O grupo de sprites *'self.pieces_group'* usa a funcionalidade de grupo de sprites do Pygame '(pygame.sprite.Group())' para atualizar e desenhar automaticamente todos os sprites que fazem parte do grupo. Isso inverte o controle, pois o Pygame lida com a atualização e renderização de todos os sprites do grupo.

Áudio com Jukebox:

- A classe **Jukebox** controla eventos de áudio. A chamada *'jukebox.check_event()'* mostra que o controle do áudio usa o Pygame para reproduzir e gerenciar sons.

Padrões GRASP

Expert:

- A classe **Button** é responsável por gerenciar os detalhes de renderização e interação dos botões, o que a torna uma escolha adequada para encapsular esse conhecimento específico.

- A classe **Jukebox** é especialista em gerenciar a reprodução de músicas utilizando o módulo ‘pygame.mixer.music’, encapsulando o conhecimento necessário para manipular músicas em um jogo ou aplicativo que utilize Pygame.

Controller:

- A classe **MainMenu** atua como um controlador principal que gerencia a interação entre os botões do menu, eventos do Pygame e transições de estado entre o menu principal e o menu de opções ‘(OptionsMenu)’.

- A classe **GameUI** atua como um controlador central para gerenciar a interação entre elementos da interface de usuário e estados do jogo. Ela coordena a lógica de mudança de estados ‘(setPhase)’ e interações de UI ‘(verifyMouseCollision)’.

- A classe **OptionsMenu** atua como um controlador que gerencia a interação entre os elementos da interface gráfica (como Slider, OptionsLabel) e os eventos do Pygame. Ela coordena a lógica de desenho e interação do menu de opções.

Princípios SOLID

Responsabilidade Única

- A classe **Button** cuida da representação e interação com botões na interface gráfica, enquanto a subclasse **MenuButton** especializa um tipo específico de botão usado em menus.

- A classe **Window** é responsável por gerenciar a janela principal do jogo (__init__) e renderizar a superfície do mapa (showMap).

Aberto/Fechado:

- Referente a classe **GameUI,** o código está aberto para extensão ao adicionar novos estados (novas subclasses de **UIState**) sem alterar o comportamento existente nas classes **GameUI** e **UIState**.

- Na classe **Jukebox,** a estrutura permite a adição de novas músicas (add_song) e a reprodução contínua do playlist, seguindo o princípio de extensibilidade de comportamento.

- A estrutura da classe **MainMenu** permite a adição de novos botões ou funcionalidades de forma relativamente simples, de maneira que, seria fácil adicionar novos botões ao menu principal sem modificar drasticamente a lógica existente.

- A estrutura da classe **OptionsLabel** permite facilmente a adição de novos rótulos de opções com diferentes textos ou formatos de exibição sem modificar a lógica central da classe.

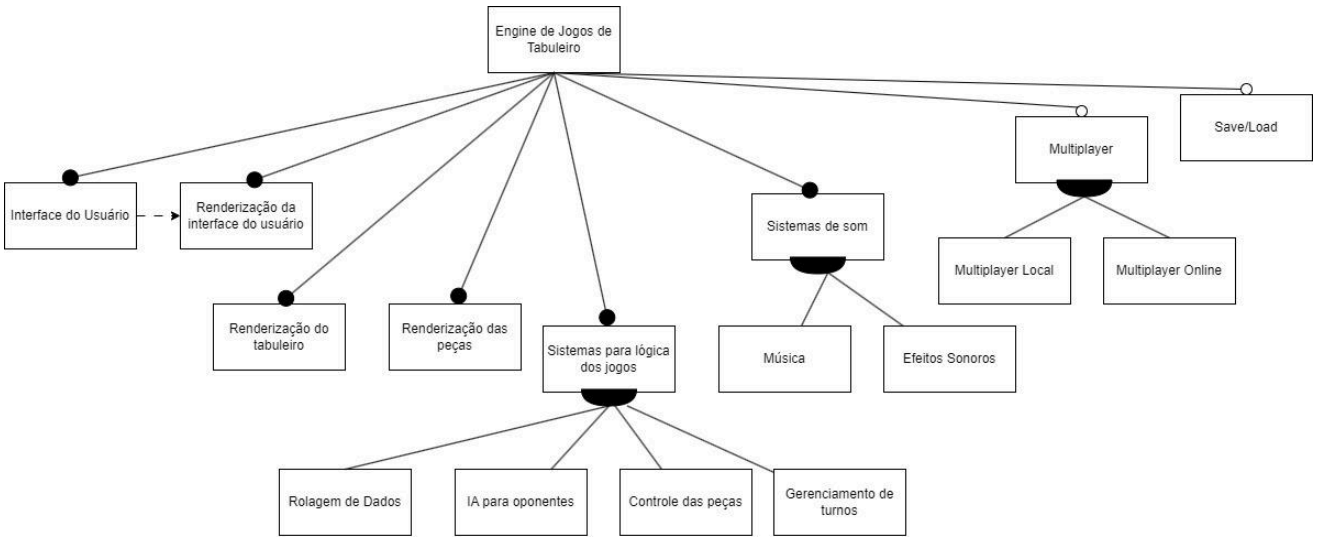
- A classe **OptionsMenu** permite a extensão do comportamento ao adicionar novos elementos de interface (como botões ou sliders) sem modificar diretamente a lógica existente.

- A estrutura da classe **Window**,permite adicionar novos métodos para manipular diferentes aspectos da janela ou renderização do jogo sem alterar a lógica central.

Substituição de Liskov:

- Referente a classe **GameUI,** as subclasses de **UIState** podem ser substituídas pela classe base **UIState** onde um **UIState** é esperado, mantendo a consistência e o comportamento esperado na interface do usuário.

LPS



Detalhes adicionais - Implementação

Além dos tópicos abordados acima, foi implementado também um sistema de save/load, utilizando docker e Postgres, que faz a integração do jogo com Banco de Dados, fazendo possível manter a persistência do sistema ao longo de múltiplos acessos.

Alguns exemplos de tabelas:

sessaojogador

id	idjogador	idsessao	vez	napartida	ehia	cor
1	0	1	[v]	[v]	[ ]	bran
2	1	1	[ ]	[v]	[v]	verm
3	2	1	[ ]	[v]	[v]	verd
4	3	1	[ ]	[v]	[v]	azul
5	4	1	[ ]	[v]	[v]	pret
6	5	1	[ ]	[v]	[v]	amar

territoriosessaojogador

	id	idsessaojogador	idterritorio	contagemtropas
1	1	1	1	4
2	2	1	15	4
3	3	1	22	4
4	4	1	27	4
5	5	1	28	4
6	6	1	37	1
7	7	1	42	4
8	8	2	19	1
9	9	2	20	6
10	10	2	24	2
11	11	2	25	1
12	12	2	26	4
13	13	2	40	3
14	14	3	16	4
15	15	3	23	6
16	16	3	38	1
17	17	3	39	1
18	18	4	7	1
19	19	4	18	3
20	20	4	21	7