Relatório de Jogo Grupo 3

Anthon Gretter, Mauricio Konrath, Nicolas Elias e Rian Serena

Universidade Federal de Santa Catarina

Curso de Bacharelado em Ciências da Computação

Disciplina INE5404 - Programação Orientada a Objetos II - 2021.1

Prof. Jônata Tyska Carvalho
Prof. Mateus Grellert

Nome do jogo: Quebra-ossos

Identificação, no código produzido, da utilização dos principais conceitos e técnicas estudadas ao longo do semestre.

Os conceitos utilizados para a produção do projeto foram os seguintes: Abstração, Herança, Encapsulamento, Polimorfismo, Interface Gráfica com o usuário, Tratamento de Exceções, utilização de Design Patterns: criacional (Singleton), estrutural (Data Access Object (DAO)) e comportamental (State). Todos eles se encontram comentados no código.

Abstração:

Classes abstratas foram muito exploradas no projeto, grande parte dos objetos mais concretos são derivados de classes abstratas, como por exemplo a classe Personagem, Animação, Poder, State.

```python
# CLASSE ABSTRATA
class Personagem(ABC):

    def __init__(self, vida, vida_maxima, velocidade, posicao):
        self.__cor = (255, 0, 0)
        self.__mostrar = True
        self.__vida = vida
        self.__vida_maxima = vida_maxima
        self.__invulneravel = False
        self.__tempo_inicial_inv = 0
        self.__tempo_inv = 0
        self.__velocidade = velocidade
        self.__posicao = posicao
```

Herança:

A herança foi explorada ao máximo para evitar replicação de código, identificamos muitas funcionalidades em comum de certas classes e generalizamos em uma classes abstrata, e estendendo tais com a utilização da herança.

```python
# HERANÇA de personagem com especializaçao
class Jogador(Personagem):

    def __init__(self):
        super().__init__(vida=3, vida_maxima=3, velocidade=400, posicao=[10,420])
        self.constantes = Constantes()

        self.__animacoes = [Animacao((26, 29), 'versao_final/src/cavaleiro/movimento/'),
                            Animacao((26, 29), 'versao_final/src/cavaleiro/pulo/')]

        self.__animacao_atual = pygame.sprite.Group(self.__animacoes[0])

        self.__tamanho_pulo = 55
        self.__velocidade_y = 10

        self.__pulando = False
        self.__atacando = False
        self.__ataque_rect = None

        self.__rect = self.__animacao.sprites()[0].rect
```

Encapsulamento:

Protegemos todo nosso código com atributos privados, deixando todos nossas classes bem encapsuladas e tentamos seguir o princípio Single Responsability tornando nossas classes mais coesas e sem acoplamento bem encapsuladas.

```python
def __init__(self):
    super().__init__(velocidade=400, posicao=[10,420])
    self.constantes = Constantes()

    self.__animacoes = [Animacao((26, 29), 'versao_final/src/cavaleiro/movimento/'),
                        Animacao((26, 29), 'versao_final/src/cavaleiro/pulo/'),
                        Animacao((26, 29), 'versao_final/src/cavaleiro/ataque/', rodar_uma_vez=True)]

    self.__animacao_atual = pygame.sprite.Group(self.__animacoes[0])
```

Polimorfismo:

Utilizamos alguns métodos polimórficos, que por meio da herança implementam a mesma assinatura de formas diferentes, um bom exemplo se encontra nas classes State, onde o método executar, tem a mesma assinatura de todos, porém ele é executado de formas diferentes dependendo da classe que a chama (mudando a view). Obs. Imagem é ilustrativa classe state completa no código.

```python
# Classe Abstrata
class State(ABC):

  @abstractmethod
  def executar(self):
      pass
```

```python
# Executa uma tela com todas as opçoes do jogo antes de inicialo (menu)
def executar(self):
    self.sistema.desenhar_fundo()
    mx, my = pygame.mouse.get_pos()
    proximo_estado = None

    # mapeia o click na tela para mudar a tela de acordo com o botao clicado
    if self.sistema.click:
        for bttn in [self.button_jogar, self.button_ranking, self.button_sair]:
            if bttn.collidepoint((mx, my)):
                sword = pygame.mixer.Sound('versao_final/src/efeitos_sonoros/espada2.mp3')
                sword.play()
                if bttn == self.button_jogar:
                    proximo_estado = Jogando(self.sistema)
                elif bttn == self.button_ranking:
                    proximo_estado = Ranking(self.sistema)
                elif bttn == self.button_sair:
                    self.sistema.sair()

    # passa para o proximo estado
    if proximo_estado != None:
        self.sistema.proximo_estado(proximo_estado)
```

```python
  # Executa toda a parte da logica do jogo até perder ou fechar a janela
  def executar(self):
      self.sistema.jogo.atualizar(self.sistema.dt)

      # quando o jogador fica sem vida, é passado para o proximo estado
      if self.sistema.jogo.final:
          self.sistema.proximo_estado(Final(self.sistema))
      super().__init__(sistema, fundo, musica)
      self.sistema.tocar_musica(loop=True)
```

Interface Gráfica com o usuário com o Padrão de projeto State:

Exploramos uma interface gráfica amigável e bonita, com a inserção de músicas e efeitos sonoros alegres que deixam o usuário no clima para jogar. Nossa view consiste em estados, onde eles decidem o que deve aparecer na tela, como por exemplo ao iniciar o jogo o estado é “setado” para menu, ao clicar em jogar, o estado a ser exibido na tela é o jogando, usando funcionalidades da engine do pygame de colisões para detectar quando há interação

do usuário com a GUI.

```
# HERANÇA de State executada até receber um evento de click em algum dos botoes
class Menu(State):
```

```
# HERANÇA de State parte principal do jogo
class Jogando(State):
```

```
# HERANÇA de State, mostra as miores pontuaçoes dos players
class Ranking(State):
```

```
# HERANÇA de State, tela de lose
class Final(State):
```

Tratamento de Exceções:

Foi realizado alguns tratamentos de exceções em todo o código onde o grupo julgou necessário, evitando ocorrer erros durante a jogatina. Principalmente utilizado na classe State, especificamente na classe Final, onde há uma interação maior com o usuário que ele coloca um nome ou número, não pode ser vazio e não pode ser maior que 15 caracteres, levantamos uma exceção com classes que criamos herdando Exception e tratamos informando ao usuario como proceder.

```python
class NomeVazio(Exception):
    def __init__(self):
        super().__init__("O nome não pode ser um valor vazio!")
```

```python
class Maior15_Caracteres(Exception):
    def __init__(self):
        super().__init__("Insira um nome menor que 15 caracteres!")
```

```python
# botao de salvar o nick do jogador, reiniciar o jogo, e passa para o proximo estado
if self.button_salvar.collidepoint((mx, my)):
        if self.sistema.click:
            #tratamento de excesões de nome vazio e nome > que 15 caracteres
            try:
                if self.nome != '':
                    if len(self.nome) <= 15:
                        self.sistema.salvar(self.nome)
                        sword = pygame.mixer.Sound('versao_final/src/efeitos_sonoros/espada2.mp3')
                        sword.play()

                        self.sistema.reiniciar_jogo()
                        self.sistema.proximo_estado(Menu(self.sistema))

                    else:
                        raise Maior15_Caracteres
                else:
                    raise NomeVazio
            except NomeVazio as e:
                self.erro_msg = str(e)
            except Maior15_Caracteres as e:
                self.erro_msg = str(e)
```

```python
# texto de erro e desenho
txt_exception = self.font.render(self.erro_msg, True, pygame.Color("red"))
tela.screen.blit(txt_exception, (190, 300))
```

Design Pattern criacional (Singleton):

Após a crítica dos professores, precisávamos de uma forma de organizar todas nossas constantes utilizadas no código, e que conseguíssemos utilizar em todo código. em aula nos foi apresentado o Singleton, e esse Design Pattern “nos caiu como uma luva”, onde necessitamos utilizar uma constante, fazemos uma instância dessa classe Constantes que herda do Singleton, e utilizamos livremente as constantes globais, basta uma alteração nos seus atributos para mudar alguma funcionalidade do jogo relacionada a constantes, como velocidade de pulo, colisão com o chão. Tornou o código mais legível e mais fácil de entendê-lo e alterar o funcionamento. Colocamos a tela como um singleton pois deve haver

apenas uma tela.

```python
class Singleton(object):
    __instance = None


    def __new__(cls, *args):
        if cls.__instance is None:
            cls.__instance = object.__new__(cls, *args)
        return cls.__instance
```

```python
# constantes globais
class Constantes(Singleton):
    def __init__(self):
        super().__init__()
        self.limite_esquerda = 0
        self.limite_direita = 908
        self.limite_chao = 420
        self.velocidade_queda = 0.06
        self.velocidade = 10
        self.atualizar_sprite = 0.030
```

***Design Pattern* estrutural (Data Access Object (DAO)):**

Nossa lógica de persistência de dados, feita para os rankings, onde os jogadores obtêm pontos ao jogar, utilizamos o Design Pattern DAO para persistir os dados em um arquivo .json, onde é salvo a pontuação obtida pelo jogador com seu nome (fornecido pela view final) e pontuação. Esse ranking é exibido na view ranking.

```python
# DAO classe Abstrata
class DAO(ABC):

    def __init__(self, fonte=''):
        self.fonte = fonte
        self.cache = {}

        # tratamento de exceçoes
        try:
            self.__load()
        except TypeError:
            self.__multiple_load()
        except FileNotFoundError:
            self.__dump()
```

```python
# DAO do ranking HERENÇAO do DAO
class PontuacoesDAO(DAO):

    def __init__(self):
        super().__init__('versao_final/src/pontuacoes.json')

    def add(self, key=None, value=None, ranks=None):
        if isinstance(key, str) and isinstance(value, float) and ranks is None:
            super().add(key, value)
```

Como foram divididas as responsabilidades na construção do jogo?

No início do projeto havia sido definido para cada integrante uma classe a ser feita: Personagem (Maurício), Obstáculo (Rian), Jogo (Anthon), Jogador (Nicolas) e o Sistema que foi feito em conjunto. Após isso, começamos a utilizar o LiveShare disponível no Visual Studio Code e o Discord para fazermos chamadas, dessa forma todos participavam e podiam editar ao mesmo tempo, achamos essa metodologia muito útil, pois permite que ajudasse-mos uns aos outros casos surgisse alguma dúvida. Posteriormente foram feitas outras divisões, como podemos ver abaixo, para que conseguíssemos terminar entregar o trabalho a tempo.

Anthon: Poder, Sistema, Jogo, Cenário, Lógica de gravidade, View, Dao, Ranking, Persistência da pontuação, Nova lógica de animação, Ajustes em geral,

Mauricio: Personagem, Sistemas, View, Jogador, GUI, Lógica de gravidade, Sprites, Soundtracks, Efeitos sonoros, Botões, Ajustes em geral

Nicolas: Jogador, Sistema, Jogo, Tela, GUI, Lógica de gravidade, Dao, View, Sprites, Botões, Lógica de estados, Ajustes em geral

Rian: Obstáculos, Colisões, Sistema, Jogo, Lógica de gravidade, Efeitos sonoros, Ranking, Ajustes em geral

Principais dificuldades encontradas na construção do projeto, dentro e fora da disciplina.

Para algumas implementações no jogo, precisamos fazer algumas pesquisas em sites na internet por exemplo nas funcionalidades da engine do pygame, que são um tanto específicas e é necessário obter um conhecimento específico para utilizá-las. Mas no geral ficou bastante claro o conteúdo passado em aula.