# RELATÓRIO DE DESENVOLVIMENTO

## Grupo 2

### Projeto: The Binding Of Isaac

Desenvolvedor: Rafael Luis Sol Veit Vargas

Universidade Federal de Santa Catarina

Curso de Bacharelado em Ciências da Computação

Disciplina INE5404 – Programação Orientada a Objetos II – 2021.2

18/03/2022

Florianópolis/SC

# Identificação de Conceitos

## Abstração

Por todo o código é possível verificar a utilização de classes abstratas, pois elas são bastante úteis para construir um bom relacionamento entre diversas classes, sempre mantendo um padrão na comunicação entre os objetos, garantindo uma grande coesão e evitando muito código embaralhado. Como pode ver nessas duas classes exemplo, a AbstractItem e a AbstractView.

```python
class AbstractItem(ABC):
    @abstractmethod
    def modificar_status(self, status: Status) -> None:
        pass

    @abstractmethod
    def check_aplicado(self) -> bool:
        pass

    @property
    @abstractmethod
    def image(self) -> Surface:
        pass

    @property
    @abstractmethod
    def rect(self) -> Rect:
        pass
```

```python
class AbstractView:
    def __init__(self, state: States, position=None, size=None) -> None: ···

    @abstractmethod
    def desenhar(self, tela: TelaJogo) -> None:
        tela.janela.blit(self.image, self.rect)

    @abstractmethod
    def run(self, events: List[event.Event]) -> States:
        pass

    @property
    @abstractmethod
    def image(self) -> Surface:
        pass

    @property
    @abstractmethod
    def rect(self) -> Rect:
        pass
```

# Herança

Esse conceito de POO foi bastante utilizado para evitar replicação de código, por exemplo para fazermos a especialização das classes abstratas, como pode ser visto em nessas duas implementações.

```python
class Jogador(AbstractPersonagem):

    def __init__(self, posicao: tuple, mapa=None) -> None: ...

    @property
    def escudo(self) -> EscudoMadeira:
        return self.__escudo

    def processar_inputs(self) -> None: ...

    def atacar(self) -> bool: ...

    def tomar_dano(self, dano: int) -> int: ...

    def receber_ataque(self, ataque: Ataque) -> int: ...

    def tomar_dano_escudo(self, dano: int) -> int: ...
```

```python
class PocaoGenerica(AbstractItem):
    __PATH_TO_SPRITE = {}
    __SIZE_TO_SPRITE = {}
    __GENERIC_POTION_SOUND_PATH = 'Sounds/sounds/potions.wav'

    def __init__(self, path: str, size: tuple) -> None:
        if path not in PocaoGenerica.__PATH_TO_SPRITE or size no
            self.__image = self.__load_image(path, size)
            PocaoGenerica.__PATH_TO_SPRITE[path] = self.__image
            PocaoGenerica.__SIZE_TO_SPRITE[size] = self.__image
        else:
            self.__image = self.__PATH_TO_SPRITE[path]

    def _get_image(self) -> Surface:
        return self.__image

    @classmethod
    def __load_image(cls, path: str, size: tuple) -> Surface:
        return import_single_sprite(path, size)

    @property
    def sound_path(self) -> str:
        return PocaoGenerica.__GENERIC_POTION_SOUND_PATH
```

A classe Jogador por exemplo herda toda a lógica de controle da frente do personagem, controle do hitbox, como também do Status do personagem, possibilitando um código muito mais organizado, limpo e escalável.

A classe PocaoGenerica além de **herdar** de AbstractItem também está sendo especializada por diversas classes especializadas de Poções, como também lidando com uma característica comum à todas elas, que é o conhecimento do seu som ao ser consumida.

## Polimorfismo

O polimorfismo, utilizado por meio da herança, foi bastante utilizado nas classes especificas de Poções, nas quais por meio da assinatura modificar_status (definida na abstractItem) executam ações diversas.

```python
class PocaoForca(PocaoGenerica):
    __PATH = 'Assets/pocoes/pocao_forca.png'
    __SIZE = (30, 30)

    def modificar_status(self, status: Status) -> None:
        if not self.__aplicado:
            self.__status = status
            self.__status.ataque += self.__potencia
            self.__aplicado = True
```

```python
class PocaoMedia(PocaoGenerica):
    __PATH = 'Assets/pocoes/pocao_media.png'
    __SIZE = (30, 30)

    def modificar_status(self, status: Status) -> None:
        if not self.__pronto:
            status.vida += self.__potencia
            self.__pronto = True
```

Essa implementação também permite que poções executem várias ações em algum objeto da classe Status (objeto que possui os valores de defesa, ataque, vida e velocidade de personagens) de diversas formas diferentes e tomando quanto tempo for necessário, tudo isso por meio de dois métodos, modificar_status e verificar_aplicado.

# Encapsulamento

Esse conceito, bastante importante para desenvolvermos um código **menos frágil**, foi utilizado em praticamente todas as classes do Jogo. Foi utilizado por exemplo na classe ControladorFases, o qual cria fases e controla a ordem nas quais elas devem ser executadas.

```
class ControladorFases():
    def __init__(self, jogador: Jogador) -> None:
        self.__fases = []
        self.__fases.append(DungeonFase(jogador))
        self.__current_fase: AbstractFase = None
```

```
    def mover(self, hit_jogador: Hitbox) -> None: ···

    def __seguir_jogador(self, hit_jogador: Hitbox) -> None: ···

    def __set_caminho(self, caminho) -> None: ···

    def __procurar_jogador(self, hit_jogador: Hitbox) -> None: ···

    def __mover_caminho(self): ···

    def __chegou_no_ponto(self, ponto: tuple) -> bool: ···

    def __dumb_movement(self, hit_jogador: Hitbox) -> None: ···

    def __mover_para_o_centro(self, hit_jogador: Hitbox) -> None: ···

    def __mover_para_ponto(self, ponto: tuple) -> None: ···
```

Também pode ser verificado um grande **encapsulamento** de métodos por todas as classes do Jogo, característica que, ao mesmo tempo que cria uma quantidade maior de métodos, mas com nomes claros e específicos (**Clean Code**) torna o código bem mais legível e mais organizado quanto a responsabilidades de cada método (**Single Responsability**), tudo isso sendo utilizado pelo método mover, o único método público referente a essa responsabilidade (**Encapsulamento**).

# Princípios SOLID

A classe abstrata Button é uma das classes mais especializadas que foram feitas no jogo, pois por meio dela, outras 12 classes de botões são especializadas, que ao mesmo tempo que fazem as mesmas coisas que a classe Button (**Liskov Substitution**) também resolvem problemas, mas específicos conforme eles vão aparecendo, permitindo um padrão de criação de novas classes para expandir as já desenvolvidas (**Open-Closed**).

```python
class Button(ABC):
    def __init__(self, position, size, next_state: States) -> None: ...

    def run(self, events: List[Event]) -> None: ...

    @abstractmethod
    def desenhar(self, tela: TelaJogo) -> None: ...
```

Também é possível verificar que sempre foi mantido um padrão no qual classes de alto nível dependem de abstrações, nesse primeiro caso, uma classe View depende exclusivamente da classe Button, que mesmo não estando especificado em sua assinatura é uma classe abstrata, dessa forma a classe View pode possuir diversos tipos diferentes de botões que fazem diversas funções diferentes, mas todos eles estarãos seguindo uma Abstração, tornando o código bem mais resistente **(Dependency Inversion**).

O mesmo pode ser verificado na classe AbstractMapa, que lida com diferentes objetos de AbstractInimigo, AbstractObjeto e AbstractItem, sempre mantendo as dependências referenciadas quanto a Abstrações, nunca a implementações concretas.

```python
self.__buttons: List[Button] = [
    MenuButton('PLAY', self.__BTN_POS[0], States.PLAY),
    MenuButton('OPTIONS', self.__BTN_POS[1], States.OPTIONS),
    MenuButton('GUIDE', self.__BTN_POS[2], States.GUIDE),
    MenuButton('QUIT', self.__BTN_POS[3], States.QUIT)]
```

```python
self.__inimigos: List[AbstractInimigo] = [] if enemies is None else enemies
self.__personagens_morrendo: List[AbstractInimigo] = []
self.__objetos: List[AbstractObjeto] = []
self.__itens: List[AbstractItem] = []
```

Essa implementação abaixo é um exemplo de como ferir o princípio **Interface Segragation**, pois mesmo que possuamos uma implementação mais específica da classe AbstractObjeto, um objeto invisível, como seu próprio nome diz, não é desenhado na tela, logo esse está herdando uma função dumb, que não é executada. Uma forma de contornar esse problema seria invertendo as posições, possuindo um AbstractObjeto que não executa algum tipo de desenho, e a partir dele poderíamos criar um AbstractObjetoDesenhavel, ou seja, estaríamos removendo um método que não se aplica a todas as classes filhas, e organizando a cadeia familiar de forma que todas as classes que herdam esse método realmente o executem.

```
class ObjetoInvisivel(AbstractObjeto):
    def __init__(self, posicao: tuple, tamanho: tuple, t
        super().__init__(posicao, tamanho, transpassavel

    def desenhar(self, tela: TelaJogo) -> None:
        return
```

# Interface Gráfica

Em relação a interface gráfica foi utilizado um **padrão de projeto State**, permitindo que por meio de uma máquina de estados controlássemos de forma escalável e organizada qual o estado da nossa aplicação como um todo, como também qual View deve ser apresentada ao usuário.

```
class StateMachine:
    def __init__(self) -> None:
        self.__states: Dict[States, AbstractState] = {
            States.MENU: MenuState(),
            States.OPTIONS: OptionsState(),
            States.QUIT: QuitState(),
            States.PLAY: PlayGameState(),
            States.NEW: NewGameState(),
            States.LOAD: LoadGameState(),
            States.PLAYING: PlayingState(),
            States.CREATE_NEW: CreateNewGameState(),
            States.LOAD_GAME: LoadingGameState(),
            States.WINNER: WinnerState(),
            States.LOSER: LoserState(),
            States.RESET: ResetState(),
            States.GUIDE: GuideState()
        }
```

# Tratamento de Exceções

Foi utilizado principalmente na classe **DAO**, responsável por fazer o save do estado atual do jogo, e por lidar com arquivos possui alto risco de exceções serem levantadas.

```
class DAO(Singleton):
    def __init__(self, path='DAO/saves/saves') -> None:
        if not super().created:
            self.__connected = False
            self.__path = path
            self.__cache = {}

            try:
                self.__load()
                print(f'Arquivo {path}.pkl aberto com sucesso')
            except:
                self.__dump()
                print('Erro na hora de abrir o arquivo, tente outro nome')
```

# Design Patterns

Os designs patterns que foram utilizados são State, DAO e Singleton, o segundo foi especializado pela classe JogoDAO, onde ocorre a ligação entre a classe genérica DAO e objetos da classe Jogo, os quais são desmontados por uma classe JogoDaoAdapter, armazenados de forma binária pela biblioteca Pickle e depois montados pelo Adapter.

Singleton foi utilizado em classes que necessitavam guardar valores que eram globais em todo o sistema, e não únicos a um State, como é o caso do Controlador do Jogo, guardando a instância do Jogo e das Opções, guardando as opções escolhidas pelo usuário.

```python
class JogoDAO(DAO):
    def __init__(self) -> None:
        super().__init__('DAO/saves/saves')

    def add(self, jogo: Jogo) -> None:
        if isinstance(jogo, Jogo):
            jogo_dao = JogoDaoAdapter.add(jogo)
            save_name = jogo.save_name
            super().add(save_name, jogo_dao)

    def get(self, key: str) -> Jogo:
        if isinstance(key, str):
            jogo_dao = super().get(key)
            jogo = JogoDaoAdapter.create(jogo_dao)
            return jogo

    def remove(self, key: str) -> None:
        if isinstance(key, str):
            return super().remove(key)
```

```python
class ControllerJogo(Singleton):
    def __init__(self) -> None: ...

    def save(self) -> None: ...

    def create_new_game(self) -> None: ...

    def load_game(self) -> None: ...

    def current_game(self) -> Jogo: ...
```

```python
class Opcoes(Singleton):

    def __init__(self) -> None: ...

    @property
    def tocar_musica(self) -> bool: ...

    @tocar_musica.setter
    def tocar_musica(self, value) -> None: ...

    @property
    def dificuldade(self) -> Dificuldade: ...
```

## Principais Dificuldades encontradas

A implementação de um jogo nesse nível é bastante desafiadora, manter todas as 6000 linhas de código utilizadas para a execução desse jogo, de forma que no final estivesse seguindo todos os padrões ensinados pelos professores, fosse totalmente escalável, permitindo novas implementações seguindo o princípio Open-Closed tomou diversas semanas de reflexão e refatoração.

Além disso, o desafio de criar inimigos que não fossem burros, tentando atravessar paredes e buracos e o desejo por construir mapas complexos, levou a necessidade de implementar um algoritmo pathfinder A* que fosse executável dentro das limitações da biblioteca Pygame, além disso, sendo a eficiencia do pathfinder, à princípio, inaceitável para a execução fluida do jogo, foi necessário mudanças em algumas lógicas de criação e representação de mapas e também a criação de uma HeapTree para aumentar a eficiência de armazenamento de pontos a serem verificados dentro do algoritmo pathfinder.

O que também causou uma grande dificuldade no desenvolvimento foi o fato de que durante 99% do desenvolvimento do Jogo não existia um UML explícito a ser seguido. Essa característica ocorreu, pois, durante boa parte do projeto não era muito claro ao desenvolvedor onde era necessário chegar e nem quais as melhores formas de executar isso. A alta volatilidade do projeto – principalmente devido a más implementações que feriam alguns princípios de POO e Solid – e o curto tempo de execução de projeto levou o UML a ser esquecido.