Trabalho Final

RPG de turnos

Eduardo Brioso Luceiro
14607621

Heloísa Pazeti
14577991

Luís Filipe Forti
14592348

16 de dezembro de 2024

Sumário

1 Descrição

O objetivo do trabalho é produzir um jogo de RPG no qual o jogador controla uma *party* de quatro membros (bárbaro, bardo, mago e paladino) em batalhas contra seis monstros (centauro, dragão, fada, fantasma, grifo e sereia). Os adversários são sorteados aleatoriamente a cada execução e o jogador avança ao derrotá-los sem que a *party* seja perdida. Caso isso aconteça, o jogador poderá sair do jogo ou recomeçar.

2 Planejamento e estrutura

No desenvolvimento do jogo, optamos por criar quatro classes principais: Controller, Item, Personagem e PrintFile.

2.1 Classe Controller

Esta classe é responsável por controlar e integrar as demais classes do jogo, além de compor a interface das batalhas e controlar o fluxo do jogo.

A classe possui os atributos *_party*, *_enemies* e *_items*, todos vetores, cujos elementos são da classe Personagem nos dois primeiros casos e da classe Item no último. Ademais, possui atributos e métodos para o controle do fluxo de jogabilidade (como variáveis para indicar se o jogo foi ganho ou não, qual é o membro ativo da *party* e métodos para mostrar informações ao jogador).

2.2 Classe Item

A classe contém o que é necessário para criar e caracterizar os itens a serem usados no jogo.

Seus atributos dizem respeito às propriedades dos personagens que serão alteradas ao usar o item (como ganho de vida, aumento da efetividade da arma, armadura ou ferramenta) e seu único método é o construtor da classe Item. Nesse caso, fez-se um construtor único com a opção de adicionar quaisquer propriedades a fim de facilitar a construção de itens com atributos combinados.

2.3 Classe Personagem

A classe personagem possui tudo que é usado para caracterizar e modificar os personagens do jogo, incluindo os membros da *party* e os adversários.

Seus atributos são as próprias propriedades dos personagens (como vida, armadura, sorte), seus modificadores temporários (de esquiva, defesa e de quantidade de ataques), os *buffs* de determinadas propriedades (ou seja, quantidades a serem somadas no valor base das propriedades) e variáveis que determinam se o personagem está apto a utilizar habilidades ou se está sob algum efeito temporário (paralisado, encantado, provocado ou amedrontado).

Os métodos da classe são utilizados para modificar as propriedades dos personagens. Dentre eles, é válido destacar os de provocar ou receber dano (alteram a vida do próprio personagem

ou de um outro; existem três diferentes tipos de dano e, por isso, existem três funções de receber dano), o de usar um consumível (aplica as propriedades de um item ao personagem), de verificar o status do personagem (se está atordoado, paralisado ou sob outro efeito, modifica temporariamente alguns atributos e tenta estabilizar o personagem com base em sua sorte), de aplicar um status ao personagem (os mesmos efeitos temporários mencionados anteriormente) e de seguir um comando (ataque, esquiva, usar efeito auxiliar ou consumível). Há também métodos para que outras classes possam obter os dados da classe personagem.

Cada personagem pertence a uma classe herdeira da classe Personagem. Logo, são herdeiras de Personagem as classes Barbaro, Bardo, Mago, Paladino, Centauro, Dragao, Fada, Fantasma, Grifo e Sereia. O intuito é que os personagens tenham as mesmas propriedades básicas, mas diferentes atributos.

Cada classe de personagem privilegia atributos diferentes em sua construção, e o intuito do jogo é dar ao jogador a possibilidade de usar essas diferenças a seu favor.

2.4 Class PrintFile

Essa classe é usada ao longo do código para imprimir dados de um arquivo *txt*. Seu único atributo é o nome do arquivo.

2.5 Diagrama de classes

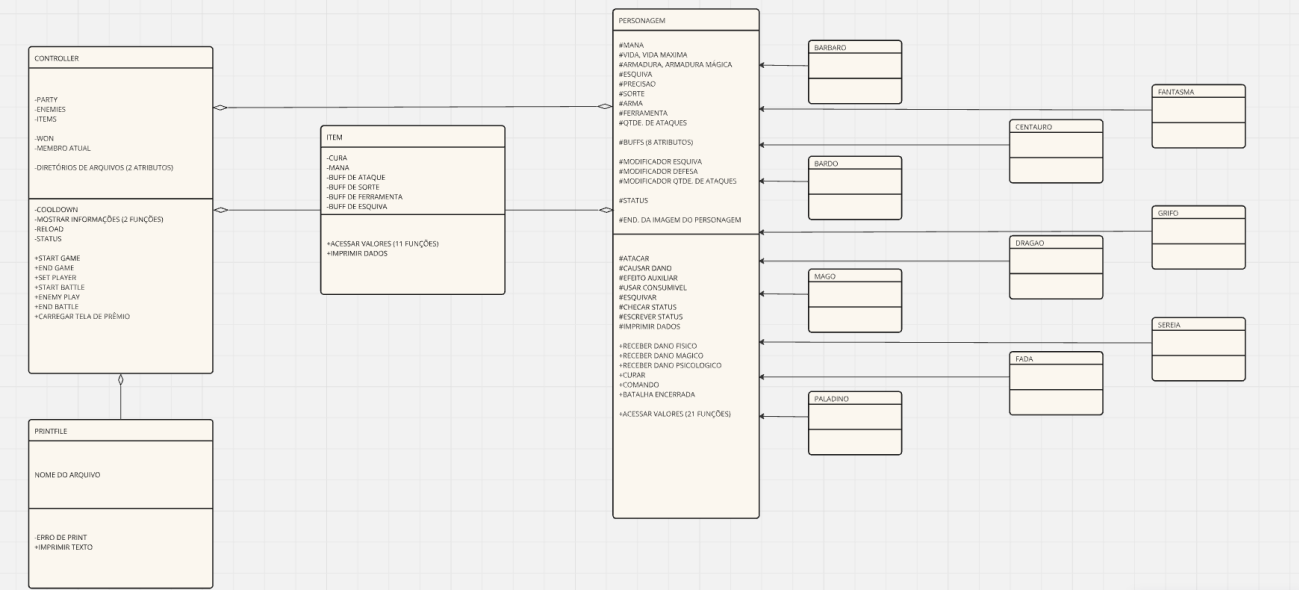

Figura 1: Diagrama de classes. As setas indicam classes herdeiras, os losangos indicam que as classes se comunicam, mas não dependem umas das outras para existir.

3 Métodos

Os métodos de cada classe estão descritos a seguir

3.1 Classe Controller

- *void Cooldown(int seconds)* (privado): aplica o tempo necessário para carregar textos;
- *void Print(std::string file, bool erase)* (privado): escreve um arquivo *txt* na tela;
- *void PrintEnemyLife()* (privado): escreve a vida do inimigo na tela;
- *void ReloadScreen* (privado): recarrega a tela com novas informações;
- *void StartGame()* (público): inicializa o jogo;
- *void EndGame()* (público): termina o jogo;
- *void SetPlayer()* (público): inicializa os membros da equipe, os inimigos e os itens, colocando-os em vetores (biblioteca *vector*);
- *void StartBattle()* (público): começa a batalha;
- *void EnemyPlay(int op)* (público): executa a jogada do inimigo;
- *void EndBattle()* (público): encerra a batalha e dá um item caso o jogador tenha ganhado ou mostra a tela de fim de jogo caso ele tenha perdido;
- *void LoadPrizeScreen(Item prize)* (público): mostra a tela de recebimento de prêmio, na qual o jogador deve escolher o membro da *party* que receberá o item.
- *void Won()* (público): mostra a tela de vitória do jogo;
- *void Lose()* (público): mostra a tela de fim de jogo;

3.2 Classe Item

Todos os métodos são públicos.

- *int GetCura()*
  bool GetMana()
  int GetBuffAtaque()
  int GetBuffSorte()
  int GetBuffArmadura()
  int GetBuffEsquiva()
  int GetBuffFerramenta()
  std::string GetNome()

std::string GetDesc(): retornam as propriedades indicadas em seus títulos;

- *friend std::ostream& operator«(std::ostream out, const Item& p)*: imprime os dados do item.

3.3 Classe Personagem

A classe Personagem possui, conforme mencionado, 10 classes herdeiras. Essas classes diferem apenas pelas funções virtuais.

- *virtual void Atacar(std::vector<Personagem*> alvos)* (protegido): método virtual, pois cada classe utiliza valores e alvos diferentes. Ataca um alvo;
- *virtual void CausarDano(Personagem *alvo)* (protegido): é virtual pelo mesmo motivo do anterior. Provoca dano a um personagem alvo;
- *virtual void EfeitoAuxiliar(std::vector<Personagem*> alvos)* (protegido): virtual pelo motivo explicitado anteriormente. Aplica um efeito auxiliar a um personagem;
- *void UsarConsumivel()* (protegido): aplica um item consumível ao personagem.
- *void Esquivar()* (protegido): dobra a esquiva até a próxima ação;
- *virtual void ImprimirDados(std::ostream out) const* (protegido): virtual pelos motivos supracitados. Mostra os dados do personagem;
- *bool CheckStatus(std::vector<Personagem*> alvos* (público): retorna se o usuário pode realizar ações;
- *void ReceberDanoFisico(int dano)* (público): aplica dano físico ao jogador, reduzindo-o conforme a armadura que ele possui;
- *void ReceberDanoMagico(int dano)* (público): aplica dano mágico ao jogador, reduzindo-o conforme a armadura mágica que ele possui;
- *void ReceberDanoPsicologico(int dano)* (público): aplica dano psicológico ao jogador;
- *void Curar(int cura)* (público): restaura vida do jogador;
- *void AplicarStatus(Estados status)* (público): aplica um status ao jogador;
- *virtual void Comando(int instr, std::vector<Personagem*> alvos)* (público): virtual pois o bardo apresenta uma variação. Recebe jma instrução e os possíveis alvos;
- *void BatalhaEncerrada()* (público): reinicia os valores temporários dos personagens, é chamada ao final de cada batalha;
- *friend std::ostream& operator«(std::ostream& out, const Personagem& p)* (público): imprime os dados do personagem;
- *Personagem& operator++(int)*: aumenta o nível do personagem;

- *int GetVida()*
  int GetVidaMaxima()
  int GetArmadura()
  int GetArmaduraMagica()
  int GetEsquiva()
  int GetPrecisao()
  int GetSorte()
  int GetArma()
  int GetFerramenta()
  int GetQuantidadeAtaques()
  int GetBuffVida()
  int GetBuffArmadura()
  int GetBuffArmaduraMagica()
  int GetBuffEsquiva()
  int GetBuffPrecisao()
  int GetBuffSorte()
  int GetBuffArma()
  int GetBuffFerramenta()
  int GetModificadorEsquiva()
  int GetModificadorDefesa()
  int GetModificadorQuantidadeAtaques()
  Estados GetStatus()
  Item GetItem() (públicos): retornam os atributos indicados no título da função;
- *std::string Status() const* (público): para escrever o estado do personagem;
- *void SetItem(Item consumivel)* (público): atribui o item ao personagem;
- *bool HasItem()* (público): indica se opersonagem possui algum item;
- *std::string GetFileId()*: retorna o ID do arquivo.

3.4 Class PrintFile

- *void PrintError(int typeError = -1)* (privado): se há um erro no arquivo ou em sua abertura, indica qual é;
- *int PrintText(bool erase = false))* (público): imprime o conteúdo do arquivo. Deve ser alterada para cada sistema operacional.

4 Implementação e funcionalidades

Na implementação final, utilizamos três pilares: o controlador (instrui e organiza os personagens), os personagens (cada um com a habilidade de interagir com outros personagens, consumir itens e responder a comandos e situações dependendo do seu estado atual) e os itens, que servem para complementar a experiência do jogador ao oferecer a possibilidade de modificar os personagens. Essa divisão foi possibilitada pelo uso da programação orientada a objetos, que tornou a implementação mais organizada e coerente, além de torná-la mais flexível para a construção de itens e personagens diferentes.

Optamos por criar uma classe geral de personagens e classes herdeiras para cada personagem específico a fim de que os personagens compartilhassem das mesmas propriedades, mas que, ainda assim, pudessem expressar uma das características do gênero RPG: a caracterização de classes. Desse modo, todos os personagens possuem, por exemplo, uma vida máxima, mas essa difere conforme a classe (o bárbaro tem uma vida máxima maior que a do mago pois isso é idiossincrático da classe). A diferenciação também pode ser vista em outras propriedades.

5 Guia de uso

Para usar o programa, é necessário utilizar um ambiente Linux. Então, entre no diretório do arquivo e, em um terminal, execute o comando *make* e, em seguida, o comando *make run*. Depois disso, deve ser mostrada a tela inicial do jogo.

Figura 2: Comando *make*

Figura 3: Comando *make run*

Figura 4: Tela de início

A interação com o jogo é feita a partir do teclado. A cada interação, o jogador deve escolher o número correspondente à ação que deseja realizar e confirmar com *enter*.

Figura 5: Escolha do primeiro ataque

6

DESAFIOS E SOLUÇÕES

Ao longo do desenvolvimento do código, encontramos alguns desafios. Os principais estão listados abaixo:

- Unir códigos escritos por membros diferentes do grupo: optamos por dividir as funcionalidades entre os membros do grupo. Embora a tática tenha se mostrado eficiente, precisamos em alguns momentos juntar as partes para compor o todo e ter uma visão geral do projeto, além de precisar de funcionalidades que estavam sendo desenvolvidas por outro membro. Para solucionar o problema, nos utilizamos da plataforma *GitHub*, que, com o uso de *branches*, permitiu que cada um trabalhasse em sua parte e unisse ao final e mantivemos a comunicação entre os membros, a fim de evitar conflitos e saber no que os demais estavam trabalhando.
- Uso de métodos e atributos protegidos: inicialmente, abordou-se as classes derivadas de Personagem como classes amigas. No entanto, por se tratar de uma má prática de programação, optou-se por deixá-las como herdeiras e usar as funções *Get* e outros métodos públicos para realizar a comunicação entre classes.