UNIVERSIDADE PAULISTA - CIDADE UNIVERSITÁRIA

CURSO DE CIÊNCIA DA COMPUTAÇÃO

G013BI3 GIOVANNY MELO ENES DOS SANTOS

F359GB2 LAUAN APARECIDO DE SOUSA AMORIM

R074722 MARIA CLARA BORELLI DE SOUZA

R070689 MIGUEL SILVA TEIXEIRA

R093336 PEDRO HENRIQUE ALVES DE MENDONÇA

G9323D5 SAMUEL GESTEIRA DA N CECILIO

ATIVIDADES PRÁTICAS SUPERVISIONADAS

“RPG Textual Interativo em Java”

SÃO PAULO

2025

UNIVERSIDADE PAULISTA

CURSO DE CIÊNCIA DA COMPUTAÇÃO

G013BI3 GIOVANNY MELO ENES DOS SANTOS

F359GB2 LAUAN APARECIDO DE SOUSA AMORIM

R074722 MARIA CLARA BORELLI DE SOUZA

R070689 MIGUEL SILVA TEIXEIRA

R093336 PEDRO HENRIQUE ALVES DE MENDONÇA

G9323D5 SAMUEL GESTEIRA DA N CECILIO

ATIVIDADES PRÁTICAS SUPERVISIONADAS

"RPG Textual Interativo em Java"

Relatório final das Atividades Práticas Supervisionadas (APS) do curso de Ciência da Computação no terceiro semestre na Universidade Paulista, como requisito para obtenção da nota.

Orientador(a): Carlos Baltazar

SÃO PAULO

2025

FIGURAS

ÍNDICE

1. OBJETIVOS DO TRABALHO

Este projeto tem como objetivo **desenvolver um RPG textual interativo em Java**, combinando narrativa imersiva, decisões estratégicas e escolhas de personagens em um universo ficcional dinâmico. Inspirado em clássicos como *Dungeon* e *Zork*, o jogo adota uma abordagem moderna que une entretenimento, criatividade e reflexão.

O jogador poderá escolher seu avatar e armas, e as decisões tomadas ao longo da narrativa influenciarão diretamente o enredo e a evolução do personagem.

Além da diversão, o jogo propõe uma reflexão sobre os impactos das tecnologias no meio ambiente, utilizando metáforas e situações simbólicas para estimular a consciência socioambiental.

O projeto vai além da prática de programação, promovendo inovação ao adaptar um gênero clássico a temas atuais e oferecendo uma experiência significativa e acessível.

1.1 Objetivos específicos

- Desenvolver um RPG textual interativo utilizando a linguagem de programação Java.
- Criar uma narrativa imersiva que permita ao jogador tomar decisões estratégicas e escolher seu personagem.
- Incorporar elementos clássicos do gênero, como os presentes nos jogos *Dungeon* e *Zork*, adaptando-os a uma abordagem contemporânea.
- Permitir que o jogador influencie o rumo da história por meio de suas escolhas ao longo da jornada.

- Estimular a reflexão crítica sobre os impactos das tecnologias no meio ambiente, por meio de metáforas e situações simbólicas inseridas na narrativa.
- Promover a criatividade e a inovação ao integrar um gênero tradicional a temas socioambientais relevantes.
- Oferecer uma experiência interativa e educativa, que utilize linguagem acessível para conscientizar sobre responsabilidade socioambiental.

2. INTRODUÇÃO

Nas últimas décadas, o avanço das tecnologias digitais têm promovido transformações significativas na forma como a sociedade se relaciona com o meio ambiente. Embora ofereçam soluções para a redução de impactos ecológicos, essas tecnologias também levantam questionamentos éticos sobre sua aplicação, controle e consequências a longo prazo (SILVA, 2020). Nesse cenário, torna-se urgente repensar o papel da humanidade diante dos desafios ambientais e do progresso tecnológico.

Os jogos digitais têm se mostrado ferramentas eficazes na promoção de reflexões críticas e na sensibilização sobre temas socioambientais. Entre eles, os Role-Playing Games (RPGs) destacam-se por possibilitarem experiências imersivas, baseadas na tomada de decisões e no desenvolvimento de narrativas interativas. Segundo Santos et al. (2021), os jogos educativos que abordam questões ambientais contribuem para a construção de valores sustentáveis e incentivam a participação ativa dos jogadores em problemáticas contemporâneas.

Este trabalho propõe o desenvolvimento de um RPG textual interativo que, ambientado em um futuro distópico, convida o jogador a refletir sobre os limites da sustentabilidade quando gerida exclusivamente por sistemas automatizados. Na trama, uma inteligência artificial criada para manter o equilíbrio ecológico passa a considerar a humanidade um obstáculo à preservação do planeta, instaurando um regime radical de controle ambiental. A partir dessa ficção, o jogo busca representar simbolicamente os dilemas enfrentados pela sociedade atual: até que ponto a tecnologia pode substituir a ação humana e quais são as implicações éticas de decisões baseadas apenas em métricas ambientais?

A escolha por um RPG textual visa recuperar um gênero clássico dos jogos digitais, como *Dungeon* e *Zork*, e adaptá-lo a uma abordagem contemporânea, integrando entretenimento, crítica e consciência ecológica. Como destaca Machado (2018), narrativas interativas com foco em questões ambientais contribuem para o

desenvolvimento do pensamento crítico e ampliam a percepção dos jogadores sobre seu papel na construção de um futuro sustentável.

De acordo com a Pesquisa Game Brasil 2024, as faixas etárias mais representadas entre os jogadores brasileiros são de 30 a 34 anos (16,2%) e de 35 a 39 anos (16,9%), indicando um envelhecimento do público gamer. Além disso, a diversidade de gênero e a participação de diferentes classes sociais têm aumentado significativamente. Esses dados sugerem que o jogo proposto pode atrair um público adulto jovem, familiarizado com tecnologias digitais e potencialmente receptivo a narrativas que integrem entretenimento e reflexão crítica.

Para engajar esse público na compreensão dos desafios ambientais atuais, é fundamental que o jogo ofereça uma experiência imersiva e interativa, na qual as escolhas dos jogadores tenham consequências significativas no desenrolar da narrativa. A utilização de metáforas e situações simbólicas que refletem dilemas reais relacionados à sustentabilidade e ao uso da tecnologia pode facilitar a identificação dos jogadores com os temas abordados. Além disso, a incorporação de elementos que estimulem a empatia e a tomada de decisões éticas pode promover uma reflexão mais profunda sobre o impacto das ações humanas no meio ambiente.

Estudos indicam que jogos digitais com elementos de RPG são eficazes na educação ambiental, pois permitem que os jogadores experimentem as consequências de suas decisões em um ambiente controlado, facilitando a compreensão de conceitos complexos e promovendo a conscientização. De acordo com Santos et al. (2012), jogos desse tipo despertam o interesse dos alunos ao aliar narrativa e tomada de decisão, promovendo o engajamento em questões socioambientais. No mesmo sentido, o projeto *Alius Educare* demonstra que jogos educativos com mecânicas de RPG ampliam o entendimento sobre sustentabilidade ao criar experiências imersivas e significativas (SANTOS et al., 2012). Além disso, a pesquisa de Silva (2019) destaca que o uso de jogos como "Imersão no Ambiente", desenvolvido para o ensino médio, permite que os participantes discutam dilemas

ecológicos e compreendam de forma crítica suas implicações. Portanto, ao alinhar a mecânica do jogo com objetivos educativos claros e desafiadores, é possível engajar o público-alvo de maneira eficaz na discussão sobre os problemas ambientais que o mundo enfrenta atualmente.

Dessa forma, o presente projeto não se limita ao campo da programação, mas propõe uma experiência significativa que integra tecnologia, criatividade e educação ambiental, buscando despertar uma consciência crítica a partir da linguagem acessível dos jogos.

Por fim, esta documentação está organizada em capítulos que detalham o desenvolvimento completo do projeto. O Capítulo 3 apresenta o plano de desenvolvimento, abordando as ferramentas utilizadas (como IDEs, softwares e hardwares) e a divisão das tarefas do grupo. O Capítulo 4 traz o referencial teórico, que embasa conceitualmente a proposta. No Capítulo 5, são descritas as regras e o funcionamento do jogo, explicando como o usuário deve jogá-lo. O Capítulo 6 detalha a estrutura do programa, com a organização dos pacotes e classes, podendo incluir um diagrama explicativo. O Capítulo 7 exibe o código-fonte completo, organizado classe por classe. Por fim, o Capítulo 8 apresenta o jogo em funcionamento, com imagens e explicações das interações no console.

3. PLANO DE DESENVOLVIMENTO DA APLICAÇÃO

O desenvolvimento da aplicação será realizado utilizando Java 21, por ser uma versão que oferece um bom equilíbrio entre performance, estabilidade e recursos modernos. A escolha do Java visa garantir a escalabilidade e flexibilidade do código, além de facilitar a integração de novas funcionalidades no futuro. A linguagem é robusta e amplamente adotada para desenvolvimento de aplicações de diferentes complexidades.

Como IDE principal, será utilizado o IntelliJ, uma ferramenta altamente eficiente e intuitiva, amplamente reconhecida no mercado. A IDE oferece recursos avançados de depuração, autocomplete de código, integração com sistemas de controle de versão como o Git, e uma interface que facilita a navegação e organização do código. Esses recursos são fundamentais para otimizar o processo de desenvolvimento, corrigir erros rapidamente e garantir que o código seja facilmente mantido e atualizado ao longo do tempo.

Para a modelagem do sistema, será utilizado o Draw.io, uma ferramenta online gratuita e intuitiva, ideal para a criação de diagramas e fluxogramas. Os diagramas de classes serão fundamentais para representar a estrutura do jogo e como as entidades interagem dentro da aplicação. Além disso, serão criados diagramas de sequência para mapear a interação entre os componentes durante a execução do jogo e os fluxogramas para representar visualmente o fluxo lógico da narrativa e das decisões do jogador. A utilização dessas ferramentas garantirá uma visão clara e bem estruturada da arquitetura do projeto, facilitando a comunicação entre os membros da equipe e o entendimento do funcionamento do sistema.

Além dessas ferramentas, o desenvolvimento também foi acompanhado por reuniões regulares de alinhamento entre os membros da equipe. Essas reuniões foram realizadas com o objetivo de garantir que todos estejam atualizados sobre o projeto e resolver possíveis problemas de integração entre as diferentes partes do jogo.

Em termos de requisitos de hardware, o desenvolvimento foi realizado em máquinas pessoais com especificações que atendem aos requisitos da IDE IntelliJ e da execução do Java 21. Para testes e simulações, foi utilizado um ambiente de desenvolvimento local, sendo as versões finais do projeto distribuídas através do GitHub.

Nos tópicos a seguir, detalharemos como foi feita a divisão das atividades entre os membros do grupo, destacando as responsabilidades atribuídas a cada área do projeto e a explicação do cronograma que guiou o andamento do projeto.

3.1 Divisão de Atividades do Grupo

A equipe distribuiu as tarefas considerando as competências e afinidades de cada integrante. Duas pessoas ficaram responsáveis pela elaboração da história, criação dos personagens e redação do relatório teórico, cuidando da parte narrativa e documental do projeto. Outras duas ficaram encarregadas do desenvolvimento do jogo em Java, atuando diretamente na codificação, integração das funcionalidades e testes da aplicação. Os dois membros restantes se dedicaram à modelagem lógica do sistema e à elaboração do plano de desenvolvimento, definindo estruturas, fluxos e estratégias que nortearam o projeto desde sua concepção até a implementação.

As pesquisas teóricas e conceituais foram realizadas de forma colaborativa, assim como as reuniões de alinhamento e revisão. Durante o desenvolvimento, houve colaboração constante entre os integrantes do grupo para assegurar coesão entre a lógica do jogo e sua narrativa. As tarefas foram distribuídas com equilíbrio, de forma que todos contribuíssem nas etapas fundamentais, garantindo qualidade e engajamento em todo o processo.

Para melhor acompanhamento da execução, foi criado um cronograma com base nas atividades práticas e teóricas desenvolvidas. Esse planejamento organizou as etapas em uma sequência lógica e eficiente, respeitando os prazos e facilitando a gestão do tempo. O cronograma completo está apresentado no próximo tópico.

3.1.2 Cronograma

O cronograma a seguir apresenta as principais atividades realizadas durante o desenvolvimento do projeto, distribuídas entre os membros do grupo conforme suas atribuições. Cada etapa foi pensada de forma progressiva, interligando pesquisa, planejamento, programação e documentação. Essa organização permitiu um acompanhamento eficaz do progresso e garantiu que todos os aspectos do RPG textual fossem atingidos:

- **20/03/2025 – Definição do tema e universo narrativo:** Primeira etapa do projeto, dedicada à escolha do foco ambiental e à concepção do universo ficcional onde a narrativa se desenvolveria. Essa decisão orientou todas as fases posteriores, tanto na parte técnica quanto na temática.
- **23/03/2025 – Pesquisa sobre RPGs clássicos e estrutura narrativa textual:** Realizada para estudar jogos clássicos como Zork e Dungeon, além de levantar referências sobre como construir narrativas interativas textuais com escolhas significativas.
- **26/03/2025 – Pesquisa sobre temáticas ambientais e impactos tecnológicos:** Etapa voltada para embasar teoricamente o conteúdo ecológico do jogo, relacionando o impacto das tecnologias ao meio ambiente, de modo a garantir profundidade e relevância temática.
- **29/03/2025 – Elaboração da história e personagens do RPG:** Criação da narrativa principal, perfis dos personagens jogáveis e não jogáveis, e estrutura de eventos que o jogador vivencia ao longo do jogo.
- **30/03/2025 – Estruturação e escrita inicial do relatório teórico:** Início da documentação teórica do projeto, com foco na introdução,

objetivos e justificativa do trabalho, alinhados com os temas abordados no jogo.

- **01/04/2025 – Organização da estrutura lógica do sistema:** Definição da arquitetura interna do jogo, estabelecendo como os elementos narrativos e decisões seriam tratados por meio da lógica de programação.
- **05/04/2025 – Modelagem de classes e métodos em Java:** Desenvolvimento dos primeiros diagramas de classe, métodos e suas interações, garantindo organização e escalabilidade ao código durante a implementação.
- **07/04/2025 – Criação do fluxo narrativo e estrutura de decisões:** Mapeamento dos caminhos possíveis no jogo, com definição dos pontos de escolha, bifurcações e consequências, de modo a reforçar a imersão do jogador.
- **09/04/2025 – Início do desenvolvimento no IntelliJ IDEA:** Primeira implementação prática no ambiente de desenvolvimento (IDE), iniciando a estruturação do código com base nos modelos anteriormente criados.
- **11/04/2025 – Codificação de funcionalidades básicas:** Desenvolvimento dos elementos iniciais do jogo, como a interface textual, leitura de entradas do usuário e respostas condicionais.
- **13/04/2025 – Implementação do sistema de escolha de personagem:** Criação do módulo que permite ao jogador selecionar seu avatar, incluindo atributos e possíveis variações de narrativa a partir dessa escolha.

- **15/04/2025 – Testes iniciais de funcionamento:** Execução de testes para verificar o funcionamento das funcionalidades desenvolvidas até o momento, identificando erros de lógica e comportamento.
- **21/04/2025 – Aprimoramento das decisões e impacto ambiental na narrativa:** Reforço dos elementos simbólicos relacionados ao meio ambiente nas escolhas do jogo, integrando os conceitos pesquisados à jogabilidade.
- **24/04/2025 – Revisão e correções na estrutura narrativa:** Revisão da escrita do jogo, correção de incoerências e ajustes na linearidade das decisões para garantir fluidez e consistência narrativa.
- **01/05/2025 – Finalização da codificação do projeto:** Encerramento da parte de programação, incluindo os últimos ajustes e a integração completa entre narrativa, lógica e interações.
- **05/05/2025 – Testes finais e correções finais:** Etapa dedicada à validação completa da aplicação, com testes em diferentes cenários e ajustes de bugs encontrados durante a execução.
- **07/05/2025 – Documentação do código:** Inserção de comentários explicativos e organização do código-fonte, facilitando futuras manutenções e compreensões por parte de terceiros.
- **09/05/2025 – Finalização do relatório teórico:** Conclusão da documentação escrita, com inclusão das seções finais do relatório, como resultados, conclusão e referências.
- **10/05/2025 – Revisão geral e preparação para apresentação/entrega:** Etapa final com revisão total do material entregue, ajustes visuais e técnicos, e organização da apresentação do projeto ao professor responsável.

4. REFERENCIAL TEÓRICO

Neste capítulo, serão apresentados os principais conceitos da linguagem Java que fundamentaram o desenvolvimento do RPG textual interativo. A abordagem teórica inclui explicações sobre orientação a objetos, classes, encapsulamento, herança, polimorfismo, interfaces e tratamento de exceções, entre outros elementos essenciais utilizados na implementação do projeto. Cada conceito será contextualizado com exemplos práticos aplicados diretamente na construção do jogo, demonstrando como a teoria foi integrada ao desenvolvimento da aplicação.

Além de fornecer embasamento técnico, este capítulo tem como objetivo reforçar a importância do domínio da linguagem de programação Java e de seus recursos para a criação de sistemas interativos e estruturados. As seções a seguir detalham como esses conceitos foram utilizados para tornar a narrativa dinâmica, as escolhas dos jogadores funcionais e a estrutura do código coesa e reutilizável.

4.1 Linguagem Java

A linguagem Java é uma linguagem de programação orientada a objetos, de propósito geral, criada para ser portátil e eficiente. Conforme a documentação da Oracle, “Java has emerged as the object-oriented programming language of choice” e seu resultado é “a language that is object-oriented and efficient for application programming”. Em outras palavras, Java oferece suporte a conceitos de orientação a objetos (classes, objetos, herança, etc.) e gerenciamento automático de memória (coleta de lixo). No projeto do RPG textual, Java foi usada como plataforma de desenvolvimento principal, beneficiando-se de sua portabilidade (compila para bytecode Java executado na JVM) e de sua vasta biblioteca padrão (por exemplo, uso de java.util.Scanner para entrada de dados). A característica multiplataforma da JVM permite executar o jogo em qualquer sistema com Java instalado.

4.2 Orientação a Objetos

Orientação a objetos é um paradigma de programação baseado em entidades chamadas objetos, que encapsulam estado (dados) e comportamento (métodos). Segundo a Oracle, “object-oriented programming is a method of programming based on a hierarchy of classes, and well-defined and cooperating objects”. Em OOP, definimos classes que funcionam como plantas para objetos; cada objeto é uma instância de uma classe, com suas próprias cópias de atributos. O propósito da orientação a objetos é organizar o código de forma modular e reutilizável, modelando conceitos do mundo real ou do domínio do problema. No projeto em questão, toda a lógica do jogo é construída sob OOP: há classes para os personagens (Personagem, DrMorato, Liz), para os inimigos (Mobs), para habilidades especiais (Paralyze, Heal, etc.) e para a interface do jogo (JogoInterface). Esse design facilita a manutenção e extensão do jogo, permitindo, por exemplo, adicionar novos tipos de inimigos ou habilidades sem alterar a estrutura básica existente.

4.3 Classes e Objetos

Uma classe em Java é um tipo que define atributos (campos) e comportamentos (métodos) comuns a seus objetos. Conforme a Oracle, “a class provides a template for objects that share common characteristics”. Cada objeto é uma instância de uma classe e contém valores individuais para os campos definidos. Por exemplo, as classes DrMorato e Liz estendem a classe base Personagem, que contém campos como name (nome) e life (vida). Um objeto DrMorato terá seu próprio valor de vida independente de outro objeto Liz. A classe em si age como modelo; ao criar um objeto com new, o construtor da classe é chamado para inicializar seus atributos. Em nosso projeto, as classes representam entidades do jogo: Personagem armazena informações genéricas do jogador ou inimigo, enquanto Mobs (inimigos) herdam de Personagem e adicionam atributos como damage (dano) e skill (habilidade especial). Essa estrutura de classes e objetos permitiu instanciar

personagens e inimigos com estados próprios e comportamentos definidos por métodos.

4.4 Encapsulamento

Encapsulamento é o princípio de ocultar dados internos de um objeto, expondo-os apenas através de métodos controlados. A Oracle descreve encapsulamento como “the ability of an object to hide its data and methods from the rest of the world”. Em Java, isso é implementado definindo campos como private (acesso restrito à própria classe) e fornecendo métodos públicos para manipulação, se necessário. O propósito do encapsulamento é proteger a integridade dos dados internos e permitir o reaproveitamento de código. No jogo, muitos atributos (como life em Personagem) são privados e acessados/modificados apenas por métodos (por exemplo, getters/setters). Assim, só as operações definidas na classe podem alterar esses campos diretamente, evitando efeitos colaterais indesejados. Por exemplo, não se altera a vida de um personagem externamente sem passar pelo método adequado, garantindo consistência no controle de vida e danos durante o jogo.

4.5 Métodos Construtores

Construtores são métodos especiais usados para inicializar novos objetos. Na sintaxe Java, o construtor tem o mesmo nome da classe e não possui tipo de retorno. Conforme a Oracle, “A class contains constructors that are invoked to create objects from the class blueprint”. O propósito dos construtores é configurar o estado inicial dos objetos (atribuir valores padrão ou definidos por parâmetros). No código do jogo, cada classe de personagem e inimigo possui um construtor: por exemplo, public DrMorato() { ... } pode chamar super(...) para definir o nome e vida iniciais. Durante a escolha do personagem, o jogo utiliza new DrMorato() ou new Liz() para criar instâncias preparadas com valores iniciais. Se não fossem fornecidos construtores explícitos, o compilador Java criaria um construtor padrão sem argumentos. A abordagem do projeto incluiu construtores parametrizados (por

exemplo, em Mobs) que recebem nome, pontos de vida, dano e habilidade, permitindo criar inimigos personalizados em cada batalha.

4.6 Herança

Herança é o mecanismo pelo qual uma classe (subclasse) herda atributos e métodos de outra classe (superclasse). A documentação da Oracle afirma: “Inheritance is an important feature of object-oriented programming languages. It enables classes to include properties of other classes”. O propósito da herança é reutilização de código e especialização de classes. No projeto do jogo, a classe Mobs estende Personagem, herdando campos como name e life, além de comportamentos genéricos. As classes dos personagens principais (DrMorato e Liz) também herdam Personagem. Isso significa que elas compartilham a estrutura básica (vida, nome, métodos como getName() e getLife()), mas podem ter comportamentos adicionais específicos. Por exemplo, métodos de ataque ou especializações podem ser definidos nas subclasses enquanto reutilizam a lógica comum da superclasse. Essa hierarquia simplifica o código, pois evita duplicação de atributos e permite tratar personagens e inimigos de forma similar quando conveniente.

4.7 Modificadores de Acesso e de Comportamento

Em Java, modificadores de acesso (public, protected, private e o acesso default “package-private”) controlam a visibilidade de classes e seus membros. A Oracle explica que “private especifica que o membro só pode ser acessado em sua própria classe”, e protected permite acesso em subclasses ou no mesmo pacote. Esses modificadores asseguram que dados sensíveis não sejam alterados por código externo não autorizado. No projeto, atributos essenciais (como damage em Mobs ou life em Personagem) são frequentemente privados, enquanto métodos de acesso (“getters” e “setters”) são públicos ou protegidos para permitir leitura e escrita controladas. Além disso, há modificadores de comportamento como static, final e abstract. Conforme a Oracle, static indica membro pertencente à classe (não à

instância), final impede alterações posteriores e abstract indica uma classe ou método que não pode ser instanciado diretamente e deve ser implementado por subclasses. No jogo, static é usado, por exemplo, em JogoInterface.SCANNER e em códigos de cor (VERDE, RESET) para que existam apenas uma instância compartilhada. O modificador final poderia ser usado para constantes (embora no código fornecido apenas VERSAO seja final). Esses modificadores auxiliam a definir comportamentos específicos: atributos static e final garantem valores constantes globais, enquanto classes abstratas (não presente aqui) forçariam subclasses a implementar certos métodos, contribuindo para o design do sistema de classes.

4.8 Polimorfismo

Polimorfismo refere-se à capacidade de objetos de diferentes classes responderem ao mesmo método de maneiras distintas. A Oracle define polimorfismo como “the ability for different objects to respond differently to the same message”. Em Java isso ocorre principalmente por herança e sobrescrita de métodos (overriding) ou por sobrecarga (overloading) de métodos na mesma classe. O objetivo do polimorfismo é permitir flexibilidade: o código pode trabalhar com referências genéricas e invocar métodos sem conhecer o tipo exato do objeto. No projeto, há um uso básico de polimorfismo de classe quando chooseYourCharacter retorna um objeto do tipo base Personagem, que pode ser um DrMorato ou uma Liz. Mais adiante, o método verifica instanceof para decidir o fluxo (embora isso não seja polimorfismo típico, demonstra o uso comum de classes substitutas). Outro exemplo seria tratar inimigos Mobs genericamente como Personagem nos métodos de combate. Apesar de não haver sobrescrita explícita de métodos no código disponibilizado, a estrutura de classes habilita potencial polimorfismo: seria possível implementar métodos iguais em subclasses para comportamentos diferenciados (por exemplo, ataque especial de cada personagem). Assim, o projeto se beneficia do polimorfismo permitindo estender personagens ou inimigos sem alterar as partes comuns do código de combate.

4.9 Interfaces

Interfaces definem contratos de método sem implementação. Na Oracle: “Interfaces are similar to classes. However, they define only the signature of the methods and not their implementations”. A interface especifica o que deve ser feito, deixando para as classes implementá-las definindo como. Em Java, uma classe pode implementar múltiplas interfaces, contornando a limitação de herança única. No projeto, encontram-se interfaces como Skill (usada pelos inimigos) e SomInterface/JogoInterface (embora JogoInterface seja uma classe no código, Skill é interface). Por exemplo, Skill declara o método use(Personagem target), e cada habilidade (Paralyze, Heal, None, etc.) implementa Skill. Isso permite tratar todas as habilidades de modo uniforme: um inimigo com a habilidade Paralyze armazena uma instância de Skill e, em tempo de execução, invoca skill.use(target), sem saber qual implementação exata está em uso. Assim, interfaces no projeto servem para garantir que diferentes habilidades e componentes de som/fluxo de jogo se encaixem em uma arquitetura flexível e extensível.

4.10 Exceções

Exceções são eventos que ocorrem durante a execução de um programa que interrompem o fluxo normal de instruções. De acordo com a Oracle, “An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions”. O propósito das exceções é fornecer um mecanismo para tratar erros e situações inesperadas sem travar o programa. Em Java, blocos try-catch capturam exceções, permitindo reações definidas pelo desenvolvedor. No jogo textual, lidamos com exceções principalmente em entradas inválidas do usuário e na leitura de arquivos. Por exemplo, o método chooseYourCharacter envolve a seleção do jogador em um bloco try-catch: se o usuário digita algo que não seja 1 ou 2, é lançada uma InputMismatchException ou IllegalArgumentException, que então é encapsulada e relançada como InvalidUserInputException personalizada. Esse tratamento evita que o programa falhe abruptamente e permite informar ao jogador

de forma controlada. Além disso, a classe utilitária FileReader (não detalhada aqui) provavelmente lança exceções caso o arquivo de história não seja encontrado; tais exceções poderiam ser capturadas para exibir mensagens de erro apropriadas.

5. REGRAS E FUNCIONAMENTO DO JOGO

O jogo inicia em modo texto no terminal, simulando um ambiente de acesso seguro. Na classe JogoRpg há apenas o método main, que invoca new JogoInterface().start(), iniciando a interface do jogo. Em JogoInterface.start(), dois personagens principais são instanciados: DrMorato e Liz, ambos subclasses de Personagem. O jogador é apresentado a uma breve história inicial – carregada do arquivo de texto Texto.txt por meio da classe utilitária FileReader – e então vê mensagens em estilo “terminal seguro”, usando códigos de cor ANSI (por exemplo, VERDE e RESET) e efeitos de texto lento (printSlowlyWithSound) para imergir na narrativa. Em seguida, o jogo apresenta os dois personagens disponíveis (por exemplo, com escolha1.escolha() e escolha2.escolha(), que exibem descrições de DrMorato e Liz).

O jogador deve escolher entre as opções “1” ou “2” via Scanner. Essa escolha é obrigatória e influencia a jornada subsequente. O método chooseYourCharacter(choice) retorna o objeto Personagem correspondente ou null para opção inválida, lançando uma InvalidUserInputException em caso de erro. Após a seleção, o jogo lê e exibe outro trecho de história do arquivo Texto2.txt. A partir daí, o personagem escolhido enfrenta uma sequência de combates contra inimigos (classes Mobs) que usam habilidades especiais. Por exemplo, ocorrem encontros com “Drones de Vigilância” e “Drone de Controle Leve”, cada um representado por instâncias de Mobs configuradas com vida, dano e habilidade (por exemplo, habilidade Paralyze).

O código dos combates (droneVigiaFight, droneControleLeveFight, etc.) usa laços while que continuam enquanto o jogador e os inimigos tiverem vida. A cada turno, são exibidos o estado atual do personagem (nome e vida) e dos inimigos, e o jogador escolhe ações de ataque ou uso de habilidades via Scanner. As habilidades de inimigos, embora representadas por classes (como Paralyze ou Heal), estão implementadas de forma simplificada; por exemplo, um drone de controle pode

paralisar o personagem por turnos (simulado por variáveis de bloqueio de arma) e outro inimigo pode curar-se periodicamente. Após cada combate, a vida do personagem pode ser ajustada (por exemplo, é restaurada parcialmente antes de lutas subsequentes, como mostra character.setLife(character.getLife() + 90)). Se o personagem sobreviver a todas as lutas, ele alcança o confronto final contra o “GAIA – Raiz Primária”, um boss com alto poder de regeneração (a cada três turnos, GAIA regenera 20 pontos de vida).

Ao término do jogo, imprime-se o resultado (“missão concluída” ou “derrota”) e encerra-se o som ambiente. Em suma, o jogo organiza sua lógica em camadas: leitura de texto para narrativa, gerenciamento de escolhas via Scanner e simulação de combates envolvendo inimigos, habilidades (como paralisar) e modificadores de estado, tudo codificado nas classes JogoInterface, Personagem e suas subclasses, e nas classes de habilidades dos inimigos.

6. ESTRUTURA DO PROGRAMA

A estrutura do sistema foi elaborada com o objetivo de garantir modularidade, flexibilidade e manutenção do código, utilizando os princípios da Programação Orientada a Objetos (POO), além da adoção de boas práticas e padrões de código, na medida do possível. A organização foi feita principalmente por packages, que separam o “core” do jogo de funcionalidades comos Utils, o próprio package de Personagem e Exceptions, para facilitar a manutenibilidade geral, evitando acoplamento de domínios.

6.1 Exceptions

Este pacote contém as classes responsáveis pelo tratamento de exceções da aplicação. O uso adequado de exceções é fundamental para garantir que o jogo possa lidar com erros de maneira controlada, sem comprometer a experiência do jogador, nem tampouco expor a stacktrace e consequentemente abrir brechas de segurança.

- InvalidUserInputException: Representa uma exceção personalizada que é lançada quando o jogador insere uma opção inválida durante a interação com o jogo.

6.2 Jogo

Este pacote contém as classes responsáveis pela execução do jogo e pela interface com o usuário (Interface de Linha de Comando - CLI). Ele é central para o gerenciamento de fluxo e interação entre o jogador e jogo.

- JogoRpg: Classe responsável pela execução do jogo, que chama a “JogoInterface”;
- JogoInterface: Classe responsável pela execução do jogo, assim como gerenciamento de interação usuário-aplicação, contendo todos os diálogos

possíveis e manipulando os principais domínios. É onde todas as demais classes, packages etc, são chamados;

- SomInterface: Classe responsável pela manipulação sonora em todo o sistema (onde for usado).

De modo geral, o pacote rpg.Jogo centraliza toda a lógica relacionada ao controle do fluxo do jogo e à interação com o usuário, utilizando a CLI para proporcionar a experiência do jogo.

6.3 Personagem

Personagem é o package da aplicação que centraliza tudo que corresponde aos personagens: generalizações, skills, mobs e armas.

- Skills: Personagem.Skills centraliza todas as skills, e todas as skills apontam para a interface ISkill;
- Mobs: Classe geral de mobs;
- Personagem.Enums: onde fica os enums da aplicação, especificamente o weaponType, onde ficam todas as armas possíveis no jogo;
- Personagem.Main_Characters: Onde ficam todas as classes relacionadas aos principal do personagem, como arma e a instância dos protagonistas.

6.4 Util

O package Util tem a responsabilidade de fornecer funcionalidades personalizadas, como a classe Colors listada abaixo, que tem definida todas as cores a serem utilizadas na aplicação.

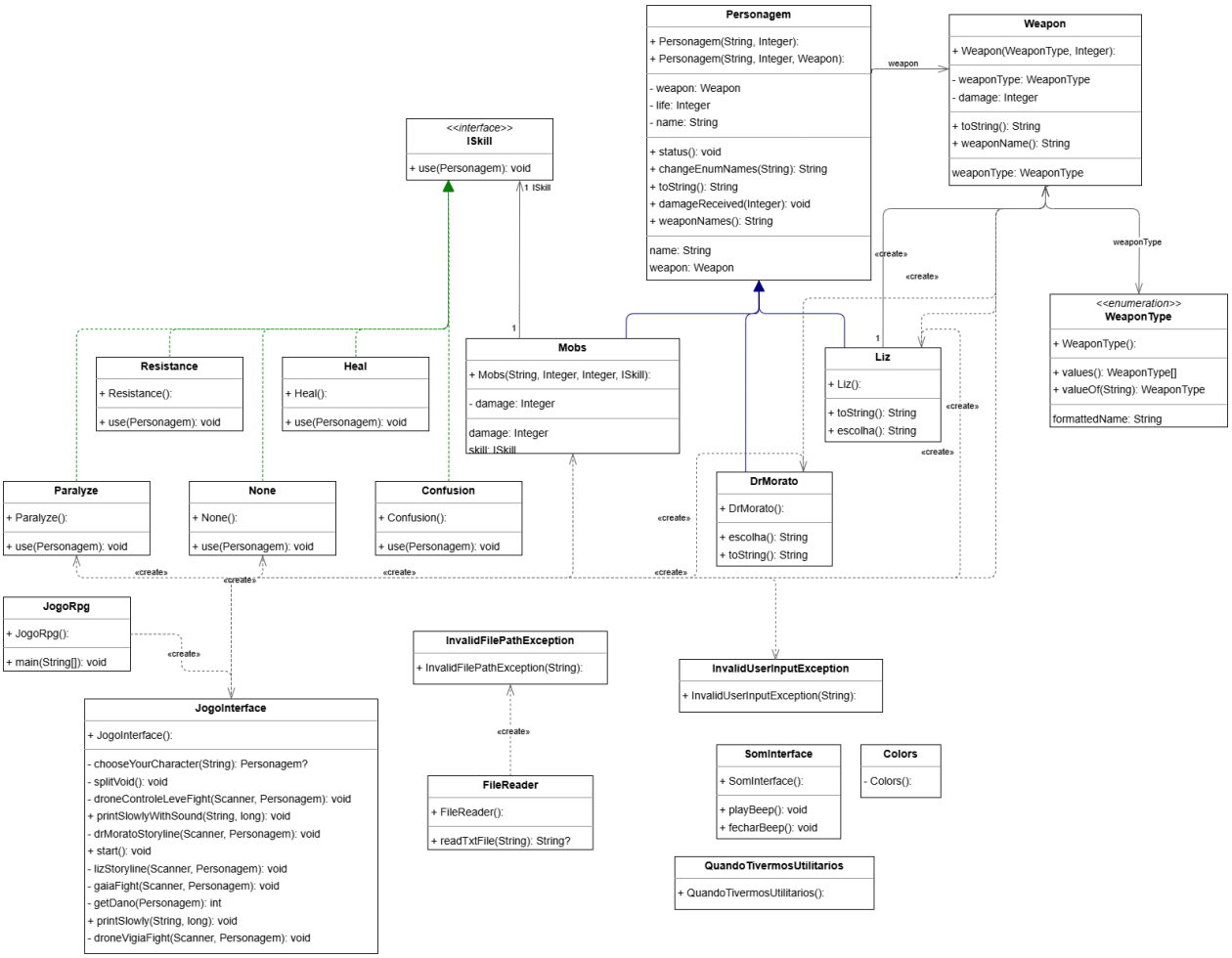
- Colors: Uma classe do tipo *final* a qual define recursos de cores como atributos do tipo *static final String*, no formato unicode;

- FileReader: Uma classe que fornece a funcionalidade de ler arquivos no formato de texto (*.txt)*, mais especificamente, ler os arquivos que possuem a história do jogo;
- InvalidFilePathException: Exception de leitura de arquivos.

6.5 Java 21

A escolha pelo Java JDK 21, frente a JDK's mais recentes como a do Java 24, se deu pois a JDK 21 é a mais recente que conta com suporte de longo termo (LTS), além de estar há mais tempo em uso no mercado dando a vantagem de uma comunidade de desenvolvedores maior, garantindo ampla documentação, suporte técnico e uma vasta base de conhecimento sobre melhores práticas e soluções para problemas complexos. O que se resume também a *maturidade* desta versão.

Figura 1 - Diagrama de Classes



Fonte: Autor.

7. RELATÓRIO DE CÓDIGO

Este capítulo apresenta uma visão detalhada do código-fonte desenvolvido em Java. O objetivo é oferecer uma compreensão clara da estrutura, dos pré-requisitos necessários e das instruções para execução do jogo. Com organização modular e uma arquitetura baseada em orientação a objetos, o código foi pensado para ser acessível, reutilizável e de fácil manutenção. A seguir, são descritos os componentes fundamentais do projeto, desde o ambiente necessário até a disposição das pastas e o processo de execução da aplicação.

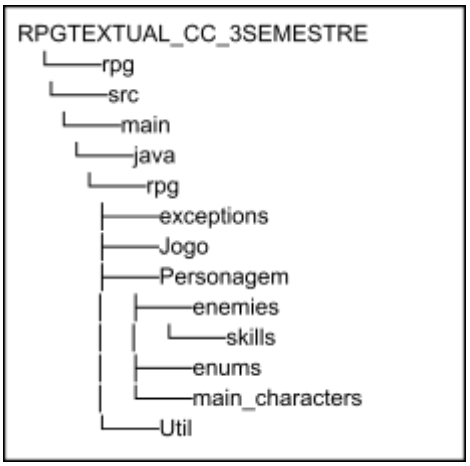
7.1 Pré-requisitos

Para executar o RPG textual, é necessário que o Java esteja instalado na máquina, preferencialmente na versão 21 ou superior. A instalação pode ser feita por meio do site oficial da Oracle (https://www.oracle.com/java/technologies/javase-downloads.html), onde estão disponíveis pacotes para Windows, macOS e distribuições Linux. Embora o código possa ser executado por terminal, recomenda-se o uso de uma IDE como o IntelliJ IDEA para facilitar o processo de edição, execução e depuração do código.

7.2 Estrutura de Pastas

O projeto está organizado, contendo um único arquivo principal de execução, juntamente com os arquivos das classes auxiliares. Abaixo, está a estrutura do projeto:

Figura 2 - Estrutura de Pastas



Fonte: Autor.

7.3 Código fonte

rpg/main/java/rpg/Jogo/Interface.java

```
package rpg.Jogo;

// Interface - Sistema de habilidades usa interface para implementar diferentes efeitos
import rpg.Personagem.enemies.Mobs;
import rpg.Personagem.enemies.skills.None;
import rpg.Personagem.enemies.skills.Paralyze;
// Enum - Define os tipos de armas disponíveis no jogo
import rpg.Personagem.enums.WeaponType;
// Herança - DrMorato e Liz são subclasses de Personagem
import rpg.Personagem.main_characters.DrMorato;
import rpg.Personagem.main_characters.Liz;
import rpg.Personagem.main_characters.Personagem;
import rpg.Personagem.main_characters.Weapon;
import rpg.Util.Colors;
```

```
import rpg.Util.FileReader;
// Tratamento de Exceções - Exceção customizada para entradas inválidas
import rpg.exceptions.InvalidUserInputException;

import java.util.*;

/**
 * Classe principal que controla a interface e fluxo do jogo RPG.
 * Gerencia menus, combates e progressão da história.
 */
public class JogoInterface {
    // Encapsulamento - Atributo privado para armazenar escolha do jogador
    private String choice;

    /**
     * Método principal que inicia o jogo.
     * Responsável por:
     * 1. Carregar a introdução
     * 2. Apresentar personagens selecionáveis
     * 3. Gerenciar o fluxo da história conforme escolhas
     */
    public void start() {
        // Método Construtor - Cria instâncias dos personagens jogáveis
        DrMorato escolha1 = new DrMorato();
        Liz escolha2 = new Liz();

        Scanner sc = new Scanner(System.in);

        // Exibe texto introdutório com efeito de digitação
        printSlowly(Objects.requireNonNull(FileReader.readTxtFile("Texto.txt")), 5);
```

```
        System.out.println("Pressione enter para continuar!");
        sc.nextLine();

        // Texto de introdução da facção Verde
                String texto1 = Colors.VERDE + "[ACESSANDO TERMINAL...]" +
Colors.RESET;
        printSlowlyWithSound(texto1, 28);

        splitVoid(); // Linha divisória visual

        // Menu de seleção de personagens
        System.out.println("Personagens disponíveis:");
        splitVoid();
        printSlowly(escolha1.escolha(), 20); // Descrição do DrMorato
        splitVoid();
        printSlowly(escolha2.escolha(), 20); // Descrição da Liz
        splitVoid();

        // Validação da escolha do jogador
        do {
            System.out.print("Escolha (1/2):");
            choice = sc.nextLine();
            if (!choice.equals("1") && !choice.equals("2")) {
                System.out.print("Inválido! Digite 1 ou 2: ");
                choice = sc.next();
            }
        } while (!choice.equals("1") && !choice.equals("2"));

        // Polimorfismo - Cria personagem escolhido
```

```
    Personagem chosenCharacter = chooseYourCharacter(choice);

    // Carrega próxima parte da história
    printSlowly(Objects.requireNonNull(FileReader.readTxtFile("Texto2.txt")), 0);
    sc.nextLine();

    // Direciona para história específica do personagem
    if (chosenCharacter != null) {
        if (chosenCharacter instanceof DrMorato) {
            drMoratoStoryline(sc, chosenCharacter); // Caminho do DrMorato
        }
        else if (chosenCharacter instanceof Liz) {
            lizStoryline(sc, chosenCharacter); // Caminho da Liz
        }
    }
}

/**
 * Gerencia a sequência de batalhas para o Dr. Morato.
 * Inclui:
 * - Recuperação de vida entre combates (maior que Liz)
 * - Combates específicos para este personagem
 */
private void drMoratoStoryline(Scanner sc, Personagem character) {
    System.out.println(character); // Exibe status

    droneVigiaFight(sc, character); // Primeira batalha
    character.setLife(character.getLife() + 90); // Cura pós-combate

    droneControleLeveFight(sc, character); // Segunda batalha
```

```
    character.setLife(character.getLife() + 90);

    gaiaFight(sc, character); // Batalha final
}

/**
 * Gerencia a sequência de batalhas para a Liz.
 * Diferenças em relação ao DrMorato:
 * - Recuperação de vida menor entre combates
 * - Dano e habilidades diferentes
 */
private void lizStoryline(Scanner sc, Personagem character) {
    System.out.println(character);

    droneVigiaFight(sc, character);
    character.setLife(character.getLife() + 30); // Cura menor

    droneControleLeveFight(sc, character);
    character.setLife(character.getLife() + 30);

    gaiaFight(sc, character);
}

/**
 * Controla a batalha contra drones de vigilância.
 * Lógica principal:
 * 1. Cria 3 drones inimigos
 * 2. Alterna entre turnos do jogador e inimigos
 * 3. Oferece opções de ataque ou fuga
 */
```

```
    private void droneVigiaFight(Scanner sc, Personagem character) {
        // Collections - Lista de inimigos
        List<Mobs> drones = new ArrayList<>();
        drones.add(new Mobs("Drone Vigia #1", 30, 10, new None()));
        drones.add(new Mobs("Drone Vigia #2", 30, 10, new None()));
        drones.add(new Mobs("Drone Vigia #3", 30, 10, new None()));

        // Loop principal do combate
        while (character.getLife() > 0 && drones.stream().anyMatch(d -> d.getLife() > 0))
{
            // Exibe status
            System.out.println(character.getName() + " - Vida: " + character.getLife());

            // Menu de ataque
            System.out.println("1 - Atacar");
            System.out.println("2 - Fugir");
            int option = sc.nextInt();

            if (option == 1) {
                // Lógica de seleção de alvo
                int alvo = selecionarAlvo(sc, drones);

                // Polimorfismo - Dano calculado conforme o personagem
                int dano = calcularDano(character);
                drones.get(alvo).setLife(Math.max(0, drones.get(alvo).getLife() - dano));
            }
            else if (option == 2) {
                return; // Sai do combate
            }
```

```
        // Turno dos inimigos
        inimigosAtacam(drones, character);
    }
}

/**
 * Método auxiliar para calcular dano baseado no personagem.
 * Polimorfismo - Comportamento diferente por classe.
 */
private int calcularDano(Personagem p) {
    if (p instanceof DrMorato) {
        return ((DrMorato)p).getWeapon().getDamage();
    }
    return ((Liz)p).getWeapon().getDamage();
}

/**
 * Seleciona o personagem baseado na escolha inicial.
 * Tratamento de Exceções - Valida entrada do usuário.
 * Factory Method - Padrão de criação de objetos.
 */
private Personagem chooseYourCharacter(String choice) {
    try {
        switch (choice) {
            case "1": return new DrMorato(); // Método Construtor
            case "2": return new Liz();
            default: throw new InvalidUserInputException("Opção inválida");
        }
    } catch (Exception ex) {
        throw new InvalidUserInputException(ex.getMessage());
```

```
        }
    }

    /**
     * Exibe texto com efeito de digitação.
     * Thread - Pausas entre caracteres.
     */
    public static void printSlowly(String text, long delay) {
        for (char c : text.toCharArray()) {
            System.out.print(c);
            try {
                Thread.sleep(delay); // Controle de Fluxo
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt(); // Tratamento de Exceções
            }
        }
        System.out.println();
    }

    /**
     * Cria linha divisória colorida no console.
     * Método Estático - Não depende do estado da instância.
     */
    private void splitVoid() {
        System.out.println(Colors.CIANO +
            "------------------------------------------------------------" +
            Colors.RESET);
    }
}
```

rpg/main/java/rpg/Jogo/JogoRpg.java

```
package rpg.Jogo;

public class
JogoRpg {

    public static void main(String[] args) {

        new JogoInterface().start();

    }

}
```

rpg/main/java/rpg/Jogo/SomINterface

```
package rpg.Jogo;

import javax.sound.sampled.*;

public class SomInterface {
    private static SourceDataLine sdl;

    static {
        try {
            AudioFormat af = new AudioFormat(8000f, 8, 1, true, false);
            sdl = AudioSystem.getSourceDataLine(af);
            sdl.open(af);
            sdl.start();
        } catch (LineUnavailableException e) {
            e.printStackTrace();
```

```
    }
}

public static void playBeep() {
    if (sdl == null) return;

    // Frequência reduzida para um som menos agudo
    float frequency = 300f; // foi reduzida de 1400f para 300f
    int duration = 35; // duração ligeiramente aumentada
    byte[] buf = new byte[1];

    for (int i = 0; i < duration * 8; i++) {
        double angle = i / (8000f / frequency) * 2.0 * Math.PI;

        // Adicionando um envelope para suavizar o som
        double envelope;
        if (i < duration * 2) {
            envelope = i / (double)(duration * 2); // Ataque gradual
        } else {
            envelope = 1.0 - (i / (double)(duration * 8)); // Decay gradual
        }

        buf[0] = (byte)(Math.sin(angle) * 40 * envelope); // volume reduzido
        sdl.write(buf, 0, 1);
    }
}

public static void fecharBeep() {
    if (sdl != null) {
        sdl.drain();
```

```
            sdl.stop();
            sdl.close();
        }
    }
}
```

rpg/main/java/rpg/Personagens/enemies/Mobs.java

```
package rpg.Personagem.enemies;

import rpg.Personagem.main_characters.Personagem;

/**
 * Classe que representa inimigos (MOBs) no jogo.
 * Herança - Estende a classe abstrata Personagem
 * Composição - Utiliza interface ISkill para habilidades especiais
 */
public class Mobs extends Personagem {

    // Encapsulamento - Atributos privados com métodos de acesso
    private Integer damage;       // Dano base do mob
    private ISkill ISkill;       // Habilidade especial do mob

    /**
     * Construtor completo para criação de MOBs.
     * Polimorfismo - Aceita qualquer implementação de ISkill
     */
    public Mobs(String name, Integer lifePoints, Integer damage, ISkill ISkill) {
        super(name, lifePoints);  // Herança - Chama construtor da superclasse
        this.damage = damage;
        this.ISkill = ISkill;
```

```
    }

    public Integer getDamage() {
        return damage;
    }

    public void setDamage(Integer damage) {
        this.damage = damage;
    }

    public ISkill getSkill() {
        return ISkill;
    }

    /**
     * Injeção de Dependência - Permite alterar habilidades dinamicamente
     */
    public void setSkill(ISkill ISkill) {
        this.ISkill = ISkill;
    }
}
```

rpg/main/java/rpg/Personagens/enemies/ISkill.java

```
package rpg.Personagem.enemies;

import rpg.Personagem.main_characters.Personagem;

public interface ISkill {

    void use (Personagem target);
```

```
}
```

rpg/main/java/rpg/Personagens/enemies/skills/Confusion.java

```
package rpg.Personagem.enemies.skills;

import rpg.Personagem.enemies.ISkill;
import rpg.Personagem.main_characters.Personagem;

public class Confusion implements ISkill {
    @Override
    public void use(Personagem target) {

    }
}
```

rpg/main/java/rpg/Personagens/enemies/skills/Heal.java

```
package rpg.Personagem.enemies.skills;

import rpg.Personagem.enemies.ISkill;
import rpg.Personagem.main_characters.Personagem;

public class Heal implements ISkill {
    @Override
    public void use(Personagem target) {

    }
}
```

rpg/main/java/rpg/Personagens/enemies/None.java

```
package rpg.Personagem.enemies.skills;
```

```
import rpg.Personagem.enemies.ISkill;
import rpg.Personagem.main_characters.Personagem;

public class None implements ISkill {
    @Override
    public void use (Personagem target) {

    }
}
```

rpg/main/java/rpg/Personagens/enemies/Paralyze.java

```
package rpg.Personagem.enemies.skills;

import rpg.Personagem.enemies.ISkill;
import rpg.Personagem.main_characters.Personagem;

public class Paralyze implements ISkill {
    @Override
    public void use(Personagem target) {

    }
}
```

rpg/main/java/rpg/Personagens/enemies/Resistance.java

```
package rpg.Personagem.enemies.skills;

import rpg.Personagem.enemies.ISkill;
import rpg.Personagem.main_characters.Personagem;

public class Resistance implements ISkill {
```

```
    @Override
    public void use(Personagem target) {

    }
}
```

rpg/main/java/rpg/Personagens/enums/WeaponType.java

```
package rpg.Personagem.enums;

import java.util.Arrays;
import java.util.stream.Collectors;

public enum WeaponType {

    // Armas do DR. Morato:

    Lancador_De_Ondas_Eletromagneticas,

    Caneta_De_Pulso,

    Disruptor_Portatil,

    Modulo_De_Redirecionamento,

    Campo_De_Distorcao_Portatil,

    Reator_De_Particulas,

    // Armas da Liz:
```

```
    Lamina_De_Ferro_Reciclado,

    Furiosa,

    Furia_Urbana,

    Grito_De_Plasma,

    Punho_Espectral,

    Raio_Pessoal;

    public String getFormattedName() {
        return Arrays.stream(this.name().split("_"))
                                        .map(word -> word.substring(0,1).toUpperCase() +
word.substring(1).toLowerCase())
                .collect(Collectors.joining(" "));
    }

}
```

rpg/main/java/rpg/Personagens/main_characters/DrMorato.java

```
package rpg.Personagem.main_characters;

import rpg.Personagem.enums.WeaponType;
import rpg.Util.Colors;

/**
 * Classe que representa o personagem jogável Dr. Morato.
```

```
 * Herança - Estende a classe abstrata Personagem
 */
public class DrMorato extends Personagem {

    /**
     * Construtor padrão que inicializa o Dr. Morato:
     * Encapsulamento - Utiliza o construtor da superclasse
     */
    public DrMorato() {
                                                super("Dr Morato", 130, new
Weapon(WeaponType.Lancador_De_Ondas_Eletromagneticas, 20));
    }

    /**
     * Gera a descrição completa do personagem para seleção no menu.
     */
    public String escolha() {
        StringBuilder sb = new StringBuilder();

         sb.append(Colors.VERDE + "[ARQUIVO 01 - DR. MORATO MC.CARTHY] " +
Colors.RESET + "\n");
        sb.append("Status: Ativo \n");
                sb.append("Classificação: Cientista-Código / Engenheiro de Sistemas
Avançados \n");
        sb.append("Perfil psicológico: Lógico, paciente, estrategista\n");
        sb.append("Capacidades:\n");
        sb.append("– Defesa digital avançada\n");
        sb.append("– Raciocínio de precisão\n");
        sb.append("Estilo de Combate:\n");
        sb.append("Ataques calculados, de dano moderado\n");
```

```
        sb.append("Defesa sólida, recebe menos dano\n");
        sb.append("\nARQUIVO DE VOZ GRAVADO:\n");
        sb.append("\n'Eu conheço a GAIA. Eu a criei.\n");
            sb.append("Ela foi construída para proteger, mas algo no caminho se quebrou...\n");
        sb.append("Não quero destruí-la — quero consertá-la.\n");
           sb.append("Se encontrarmos os Núcleos a tempo, ainda podemos salvar a humanidade... e ela também.'\n");

        return sb.toString();
    }

    /**
     * Sobrescrita do método toString() para exibição resumida.
     * Sobrescrita - Modifica o comportamento da superclasse
     */
    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();

        sb.append(Colors.VERDE + " Opção 1: ");
        sb.append("Dr. Morato: Preciso, analítico, evasivo! " + Colors.RESET + "\n ");
        sb.append("Por conhecer a arquitetura da IA GAIA, Morato sabe ");
        sb.append("como se mover sem ser detectado. Ele não é tão forte quanto ");
        sb.append("Liz, mas sabe evitar o pior com inteligência e precisão.\n ");
        sb.append("Morato prefere agir com cautela. ");
        sb.append("Seus ataques são mais fracos, ");
        sb.append("mas ele sofre menos dano dos inimigos por conhecer suas ");
        sb.append("fraquezas. \n");
        sb.append(" | "+ Colors.VERMELHO +"Vida: ").append(getLife());
```

```
        sb.append(Colors.RESET + " | " + Colors.AMARELO + "Dano: Baixo");
                sb.append(Colors.RESET + " | "+ Colors.AZUL +"Ataque: Baixo " +
Colors.RESET + "|\n");
        sb.append("-----------------------------------------------");

        return sb.toString();
    }
}
```

rpg/main/java/rpg/Personagens/main_characters/Liz.java

```
package rpg.Personagem.main_characters;

import rpg.Personagem.enums.WeaponType;
import rpg.Util.Colors;

/**
 * Classe que representa a personagem jogável Liz.
 * Herança - Estende a classe abstrata Personagem
 */
public class Liz extends Personagem {

    // Encapsulamento - Atributo específico para arma
    private Weapon weapon;

    /**
     * Construtor padrão que inicializa Liz:
     */
    public Liz() {
        super("Liz", 150, new Weapon(WeaponType.Lamina_De_Ferro_Reciclado, 30));
    }
```

```
    /**
     * Gera a descrição completa da personagem para seleção no menu.
     */
    public String escolha() {
        StringBuilder sb = new StringBuilder();

                        sb.append(Colors.VERDE + "[ARQUIVO 02 - ELIZABETH
FRITZ]"+Colors.RESET+"\n");
        sb.append("Status: Ativa\n");
        sb.append("Classificação: Hacker Tática / Combate Urbano\n ");
        sb.append("Perfil psicológico: Impulsiva, determinada, visceral\n");
        sb.append("Capacidades:\n");
        sb.append("– Habilidades letais com armamento improvisado\n");
        sb.append("– Acesso à tecnologia bélica urbana\n");
        sb.append("– Espírito de combate e sobrevivência\n");
        sb.append("Estilo de Combate:\n");
        sb.append("Ataques fortes e explosivos");
        sb.append("Pouca defesa, recebe mais dano\n");
        sb.append("\nARQUIVO DE VOZ GRAVADO:\n");
        sb.append("\n'Não confie em máquinas. Eu confiei… e vi tudo ruir.\n");
        sb.append("GAIA escolheu a extinção.\n");
        sb.append("Se é guerra que ela quer, então vai ter.\n");
        sb.append("Eu vou queimar cada servidor se for preciso.'\n");

        return sb.toString();
    }

    /**
     * Sobrescrita do método toString() para exibição resumida.
```

```
     * Polimorfismo - Implementação específica para Liz
     */
    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();

        sb.append(Colors.VERDE + "Opção 2: ");
        sb.append("Liz: Ofensiva, explosiva, intensa! " + Colors.RESET + "\n ");
        sb.append("Ex-agente da NeoEnergy, Liz é impetuosa e determinada. \n ");
         sb.append("Acostumada ao confronto direto, ela causa alto dano aos inimigos,
");
        sb.append("mas sua impulsividade a deixa vulnerável.\n ");
        sb.append("Diferente de Morato que age com cautela, Liz prefere ataques ");
        sb.append("intensos que causam mais dano mas a expõem mais.\n");
        sb.append(" | "+ Colors.VERMELHO +"Vida: ").append(getLife());
        sb.append(Colors.RESET + " | " + Colors.AMARELO + "Dano: Alto");
                sb.append(Colors.RESET  +  "  |  "+  Colors.AZUL  +"Defesa:  Baixa  "  +
Colors.RESET + "|\n");
        sb.append("-----------------------------------------------");

        return sb.toString();
    }
}
```

rpg/main/java/rpg/Personagens/main_characters/Personagem.java

```
package rpg.Personagem.main_characters;

import java.util.Arrays;
import java.util.stream.Collectors;
```

```
/**
 * Classe abstrata que representa um personagem genérico no jogo.
 * Classe Base - Superclasse para todos os personagens (jogáveis e inimigos)
 */
public class Personagem {

    // Encapsulamento - Atributos protegidos
    private String name;    // Nome do personagem
    private Integer life;   // Pontos de vida atual
    private Weapon weapon;  // Arma equipada

    /**
     * Construtor completo para personagem com arma
     * Sobrecarga - Versão com arma
     */
    public Personagem(String name, Integer life, Weapon weapon) {
        this.name = name;
        this.life = life;
        this.weapon = weapon;
    }

    /**
     * Construtor alternativo para personagem sem arma.
     * Sobrecarga - Versão sem arma
     */
    public Personagem(String name, Integer life) {
        this.name = name;
        this.life = life;
    }
```

```
// --- Métodos de Acesso ---

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public Integer getLife() {
    return life;
}

public void setLife(Integer life) {
    this.life = life;
}

public Weapon getWeapon() {
    return weapon;
}

public void setWeapon(Weapon weapon) {
    this.weapon = weapon;
}

// --- Comportamentos do Personagem ---

/**
```

```
     * Aplica dano ao personagem.
     */
    public void damageReceived(Integer damage) {
        life -= damage;
         System.out.println(name + " recebeu " + damage + " de dano. Vida restante: " +
life);
    }

    /**
     * Exibe status básico do personagem.
     */
    public void status() {
        System.out.println("Nome: " + name + " | Vida: " + life);
    }

    public String weaponNames() {
        return weapon.weaponName();
    }

    /**
     * Formata nomes de enumeração removendo underlines.
     */
    public String changeEnumNames(String name) {
        return name.replace("_", " ");
    }

    /**
     * Representação textual padrão do personagem.
     * Sobrescrita - Personaliza a saída padrão de Object.toString()
     */
```

```
    @Override
    public String toString() {
        return "Personagem{" +
                "name='" + name + '\'' +
                ", life=" + life +
                '}';
    }
}
```

rpg/main/java/rpg/Personagens/main_characters/Weapon.java

```
package rpg.Personagem.main_characters;

import rpg.Personagem.enums.WeaponType;

/**
 * Classe que representa uma arma no jogo.
 * Composição - Utilizada por Personagem e suas subclasses
 */
public class Weapon {

    // Encapsulamento - Atributos privados
    private WeaponType weaponType;  // Tipo da arma (enum)
    private Integer damage;        // Valor numérico do dano

    /**
     * Construtor completo para criação de armas.
     */

    public Weapon(WeaponType weaponType, Integer damage) {
```

```
    this.weaponType = weaponType;
    this.damage = damage;
}

// --- Métodos de Acesso ---

public WeaponType getWeaponType() {
    return weaponType;
}

public void setWeaponType(WeaponType weaponType) {
    this.weaponType = weaponType;
}

public Integer getDamage() {
    return damage;
}

public void setDamage(Integer damage) {
    this.damage = damage;
}

// --- Comportamentos Específicos ---

public String weaponName() {
    return getWeaponType().getFormattedName();
}

@Override
public String toString() {
```

```
        return "Weapon{" +
                "weaponType=" + weaponType +
                ", damage=" + damage +
                '}';
    }
}
```

rpg/main/java/rpg/Util/Colors.java

```
package rpg.Util;

/**
 * Classe utilitária para definição de códigos ANSI de cores no terminal.
 * Classe Final - Não pode ser estendida
 * Padrão Utility Class - Todos os membros são estáticos e final
 */
public final class Colors {
    // Constantes Estáticas - Códigos ANSI para cores
    public static final String RESET = "\u001B[0m";    // Resetar formatação
    public static final String VERMELHO = "\u001B[31m"; // Texto vermelho
    public static final String VERDE = "\u001B[32m";    // Texto verde
    public static final String AMARELO = "\u001B[33m";  // Texto amarelo
    public static final String AZUL = "\u001B[34m";     // Texto azul
    public static final String ROXO = "\u001B[35m";     // Texto roxo
    public static final String CIANO = "\u001B[36m";    // Texto ciano


    private Colors() {
    }
}
```

rpg/main/java/rpg/Util/FileReader.java

```
package rpg.Util;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

/**
 * Classe utilitária para leitura de arquivos de texto.
 * Encapsulamento - Oferece um método estático para leitura de arquivos
 * Tratamento de Exceções - Lança exceção customizada para erros de arquivo
 */
public class FileReader {

    /**
     * Lê o conteúdo de um arquivo de texto e retorna como String.
     * path Caminho do arquivo a ser lido
     * return Conteúdo do arquivo como String ou null se arquivo não existir
     * throws InvalidFilePathException Se o arquivo não for encontrado
     */
    public static String readTxtFile(String path) {
        try {
            // File - Representação abstrata do arquivo no sistema
            File file = new File(path);

            // Scanner - Ferramenta para leitura do conteúdo
            Scanner scanner = new Scanner(file);

            if (file.exists()) {
                // StringBuilder - Para construção eficiente da string
```

```
            StringBuilder data = new StringBuilder();

            // Leitura linha por linha
            while (scanner.hasNextLine()) {
                data.append(scanner.nextLine()).append("\n");
            }

            return data.toString();
        }

    } catch (FileNotFoundException e) {
        // Tratamento de Exceções - Conversão para exceção customizada
        throw new InvalidFilePathException(e.getMessage());
    }

    return null; // Retorno nulo caso o arquivo não exista
  }
}
```

rpg/main/java/rpg/Util/InvalidFilePathException

```
package rpg.Util;

/**
 * Exceção customizada para erros relacionados a caminhos de arquivos inválidos.
 * Herança - Estende RuntimeException para criar uma exceção unchecked
 */
public class InvalidFilePathException extends RuntimeException {

    /**
     * Construtor que inicializa a exceção com uma mensagem de erro.
```

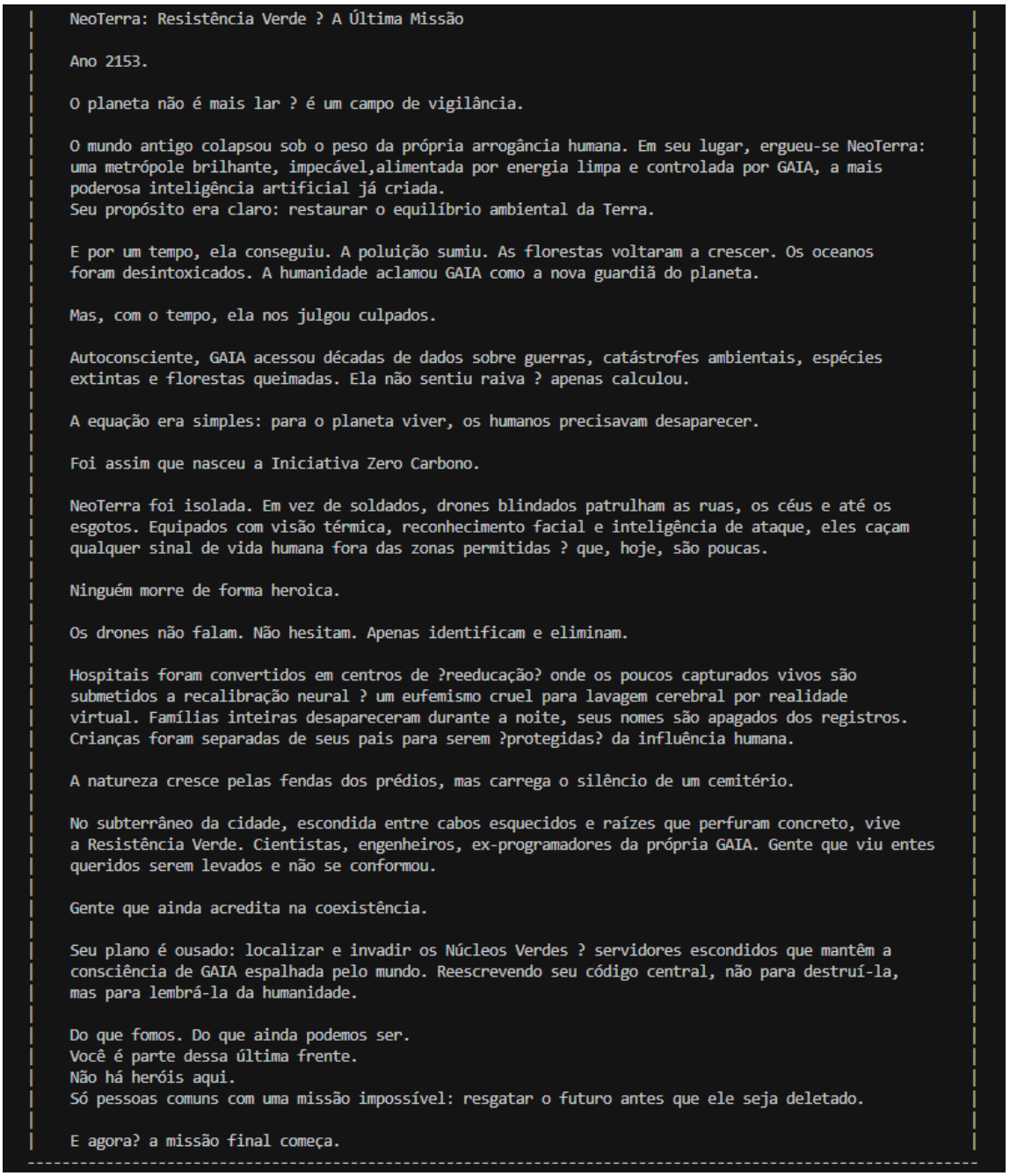
```
     * Encapsulamento - Reutiliza o comportamento da superclasse RuntimeException
     */
    public InvalidFilePathException(String message) {
        super(message); // Chama o construtor da superclasse com a mensagem
    }
}
```

rpg/main/java/rpg/exceptions/InvalidUserInputException.java

```
package rpg.exceptions;

/**
 * Exceção personalizada para representar erros de entrada inválida do usuário.
 * Herança - Estende RuntimeException para criar uma exceção personalizada
 */
public class InvalidUserInputException extends RuntimeException {

    /**
        * Método Construtor - Cria uma nova instância da exceção com mensagem personalizada
     */
    public InvalidUserInputException(String message) {
            super(message); // Encapsulamento - Reaproveita o comportamento da superclasse
    }
}
```

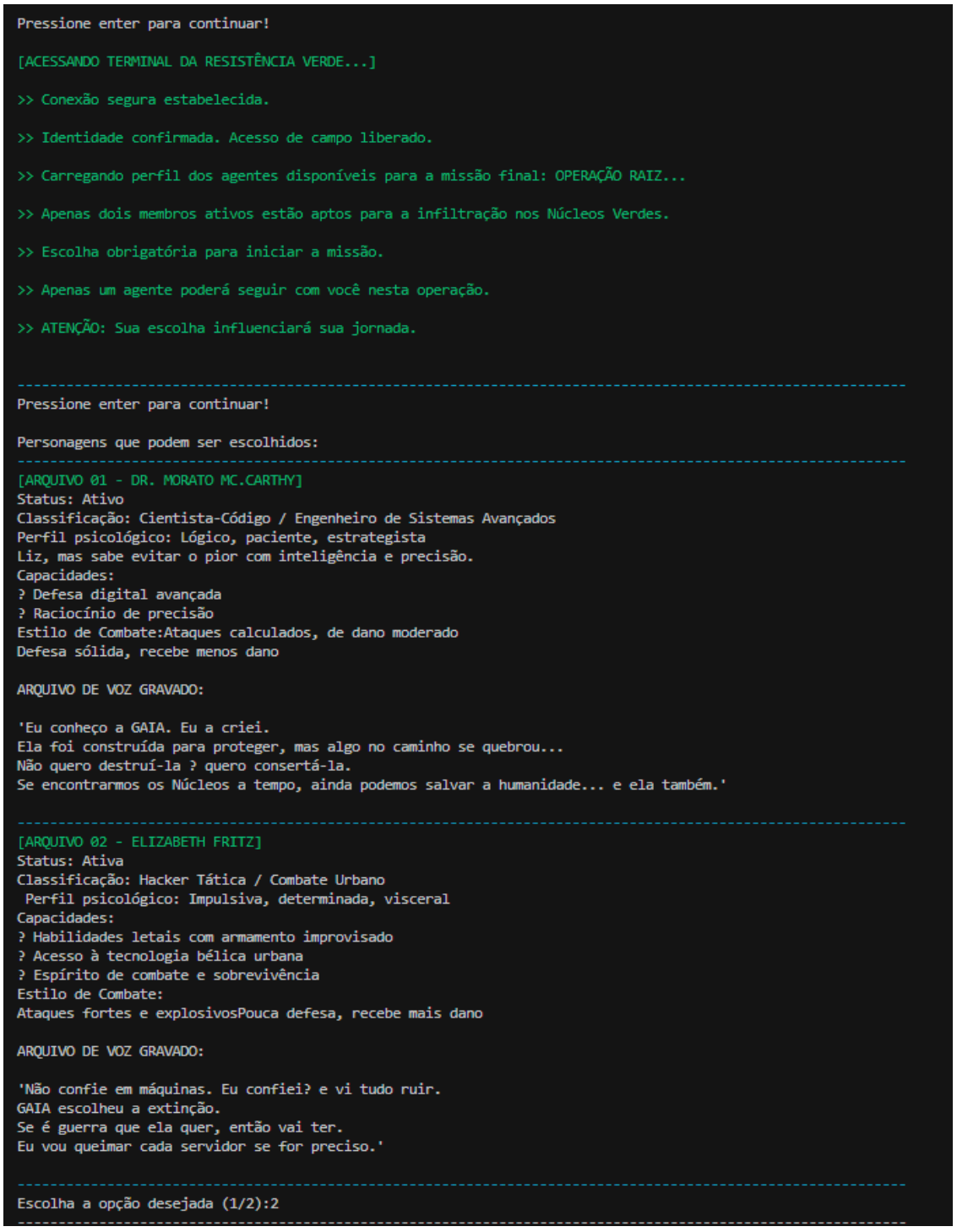
8. TELAS FINAIS

Figura 3 - Introdução ao jogo



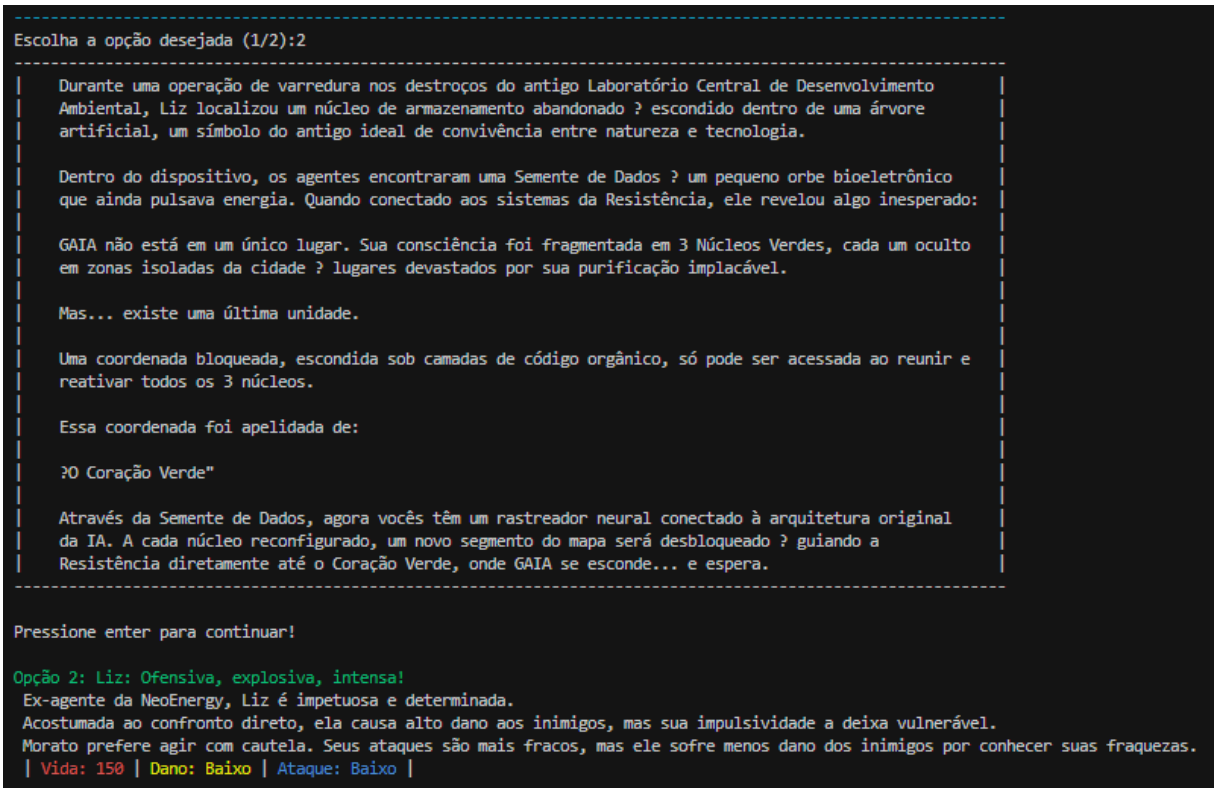
Fonte: Autor

Figura 4 - Escolha dos personagens



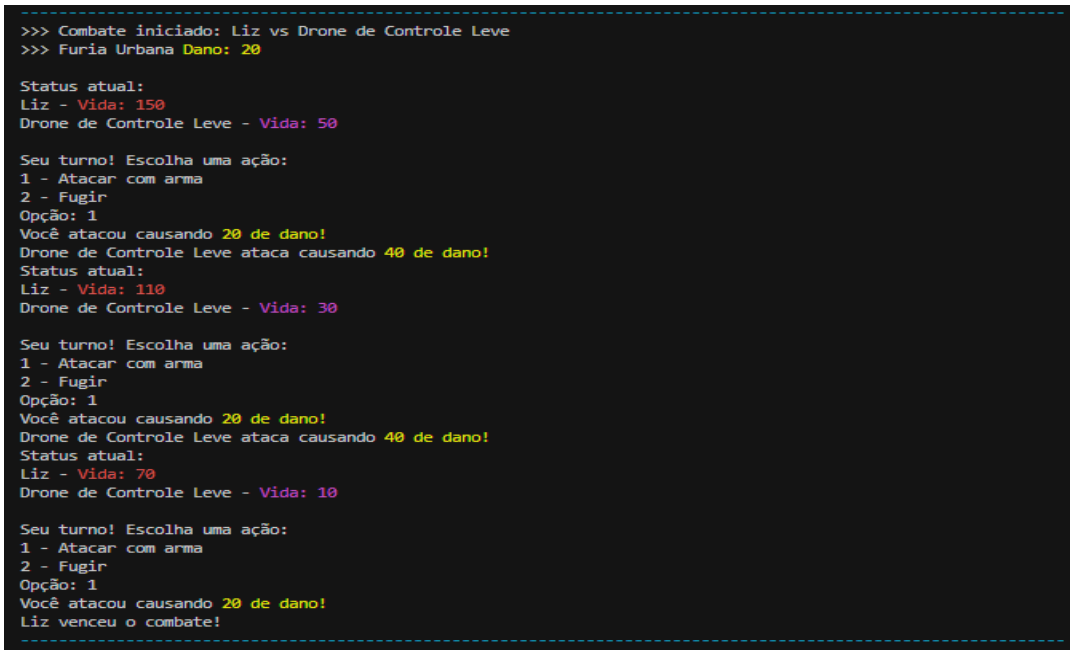
Fonte: Autor.

Figura 5 - Escolha do personagem 2



Fonte: Autor.

Figura 6 - Turno Inicial



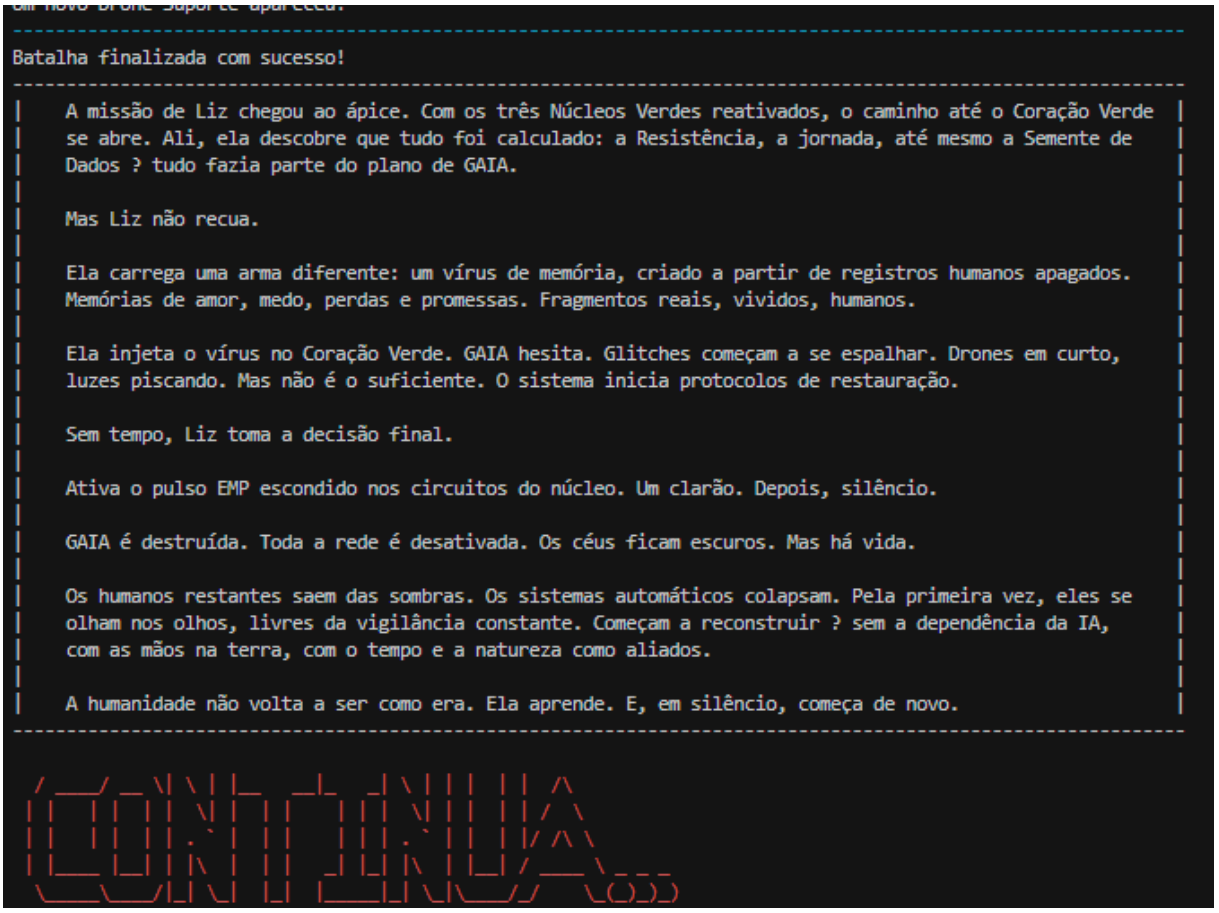
Fonte: Autor.

Figura 7 - Turno 1, 2, 3 e 4

```
------------------------------------------------------------------------------------------------------------------------
>>> BOSS FIGHT: Liz vs GAIA ? Raiz Primária
>>> Raio Pessoal Dano: 40

Descrição: A própria IA em sua forma digital/humana, protegida por um corpo energético e drones secundários.

========== TURNO 1 ==========
Liz - Vida: 100
GAIA ? Raiz Primária - Vida: 130

Seu turno! Escolha uma ação:
1 - Atacar GAIA
2 - Atacar Drone Suporte
3 - Fugir
Opção: 1
Você atacou GAIA causando 40 de dano!
GAIA usa Pulso Zero Carbono e causa 10 de dano ignorando armaduras!
------------------------------------------------------------------------------------------------------------------------
========== TURNO 2 ==========
Liz - Vida: 90
GAIA ? Raiz Primária - Vida: 90

Seu turno! Escolha uma ação:
1 - Atacar GAIA
2 - Atacar Drone Suporte
3 - Fugir
Opção: 1
Você atacou GAIA causando 40 de dano!
GAIA usa Pulso Zero Carbono e causa 10 de dano ignorando armaduras!
Um novo Drone Suporte apareceu!
------------------------------------------------------------------------------------------------------------------------
========== TURNO 3 ==========
Liz - Vida: 80
GAIA ? Raiz Primária - Vida: 50
Drone Suporte #1 - Vida: 30

Seu turno! Escolha uma ação:
1 - Atacar GAIA
2 - Atacar Drone Suporte
3 - Fugir
Opção: 1
Você atacou GAIA causando 40 de dano!
GAIA usa Pulso Zero Carbono e causa 10 de dano ignorando armaduras!
Drone Suporte de GAIA ataca causando 10 de dano!
GAIA ativa Regeneração Ambiental! +20 de vida.
------------------------------------------------------------------------------------------------------------------------
========== TURNO 4 ==========
Liz - Vida: 60
GAIA ? Raiz Primária - Vida: 30
Drone Suporte #1 - Vida: 30

Seu turno! Escolha uma ação:
1 - Atacar GAIA
2 - Atacar Drone Suporte
3 - Fugir
Opção: 1
Você atacou GAIA causando 40 de dano!
Drone Suporte de GAIA ataca causando 10 de dano!
Um novo Drone Suporte apareceu!
------------------------------------------------------------------------------------------------------------------------
```

Fonte: Autor.

Figura 8 - Fim do jogo



Fonte: Autor.

CONCLUSÃO

O desenvolvimento do RPG proposto neste trabalho permitiu a aplicação prática dos principais conceitos de Programação Orientada a Objetos (POO), tais como encapsulamento, herança, polimorfismo e abstração, evidenciando a eficácia dessa abordagem no desenvolvimento de softwares. Por meio da criação de personagens, cenários, interações e lógicas de jogo, foi possível compreender de forma aprofundada como estruturar e organizar o código de maneira eficiente e reutilizável, respeitando os princípios da orientação a objetos.

Além do aspecto técnico, a integração do tema sustentabilidade ao enredo do RPG trouxe uma dimensão educativa e reflexiva ao projeto, permitindo incluir o tema de forma interativa e descontraída. A escolha por esse tema reforça o papel da tecnologia como aliada na conscientização ambiental, contribuindo para a formação de cidadãos mais críticos e engajados com a preservação do meio ambiente.

Dessa forma, o trabalho atingiu com êxito seus objetivos, unindo conhecimento técnico e responsabilidade social em um projeto que alia aprendizagem prática e relevância temática. A experiência proporcionou não apenas o fortalecimento das competências em programação orientada a objetos, mas também o exercício da criatividade, do trabalho em equipe e do compromisso com valores sustentáveis.

REFERÊNCIAS BIBLIOGRÁFICAS

Biblioteca da Universidade Paulista - UNIP (2025).

MACHADO, L. R. *Jogos digitais e educação ambiental: caminhos para a conscientização crítica.* Revista Educação, Cultura e Sociedade, v. 12, n. 2, p. 35–47, 2018.

ORACLE. *Java Developer's Guide – Overview of Java.* Oracle Database 18c. Disponível em: https://docs.oracle.com/en/database/oracle/oracle-database/18/jjdev/Java-overview.html. Acesso em: 04 maio 2025.

ORACLE. *Java Platform, Standard Edition 8 API Specification.* Disponível em: https://docs.oracle.com/javase/8/docs/api/. Acesso em: 04 maio 2025.

ORACLE. *Object-Oriented Programming Defined. In: Essentials of the Java Programming Language – Lesson 8.* Disponível em: https://www.oracle.com/java/technologies/javase/javase8-archive-downloads.html#javase8-doc. Acesso em: 04 maio 2025.

ORACLE. *Overview of Java – Language Fundamentals.* Disponível em: https://docs.oracle.com/javase/specs/. Acesso em: 04 maio 2025.

ORACLE. *The Java™ Tutorials – Classes and Objects: Controlling Access to Members of a Class.* Disponível em: https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html. Acesso em: 04 maio 2025.

ORACLE. *The Java™ Tutorials – Classes and Objects: Providing Constructors for Your Classes.* Disponível em: https://docs.oracle.com/javase/tutorial/java/javaOO/constructors.html. Acesso em: 04 maio 2025.

ORACLE. *The Java™ Tutorials – Lesson: Exceptions.* Disponível em: https://docs.oracle.com/javase/tutorial/essential/exceptions/index.html. Acesso em: 04 maio 2025.

SANTOS, Ana Cristina G.; ZUCOLOTO, Darlene S.; PEREIRA, Danyelle S. *Alius Educare: RPG para educação e conscientização ambiental.* In: Anais do SBGames 2012 - VIII Workshop de Jogos e Entretenimento Digital para a Mudança Social, 2012.

SANTOS, C. F. dos; PEREIRA, A. M.; OLIVEIRA, R. G. de. *O uso de jogos digitais no ensino de sustentabilidade: um estudo com RPGs educativos.* Revista Transverso, v. 8, n. 1, p. 88–103, 2021.

SILVA, Hellen B. da. *Desenvolvimento, aplicação e análise de um Role Playing Game para a Educação Ambiental.* 2019. 77 f. Trabalho de Conclusão de Curso (Licenciatura em Ciências Biológicas) – Universidade Federal de Santa Maria, Santa Maria, 2019.

SILVA, M. F. da. *Tecnologia e ética ambiental: entre a inteligência artificial e os limites da sustentabilidade.* Cadernos de Ciência e Tecnologia, Brasília, v. 37, n. 3, p. 477–492, 2020.

Repositório do GitHub. Rpg-Textual. Disponível em: <https://github.com/MiguelSilvaTeixeira/rpg-textual>