Instituto de Ensino Superior

Relatório técnico da construção do jogo “Mercador”

Carlos Mariano
Gabriel Coelho
Paulo Vinícius

Dimmy Magaglhães - Estrutura de Dados
3° Período de Engenharia de Software

Sumário

RESUMO

Criação de um jogo eletrônico de estratégia desktop, desenvolvido em JAVA, cuja a trama concentra-se na viagem em que o personagem Maxwell, o viajante, deve fazer de sua cidade natal (Ubud) até a cidade de Nargumun, carregando consigo uma joia com poderes mágicos, um mapa e três moedas. Seu objetivo principal é chegar vivo na cidade de destino com no mínimo quatro moedas de ouro, completando ou não as missões que aparecerem em seu caminho.

INTRODUÇÃO

Este projeto teve início em meados do mês de julho do ano de 2023, durante o curso de Engenharia de Software da faculdade ICEV, quando o professor Dimmy Magalhães o propôs como método de avaliação da nota do segundo bimestre da disciplina de *ESTRUTURA DE DADOS* e também, como forma de avaliar os conceitos até então ministrados sobre estruturação, organização e otimização de dados em algoritmos.

A realização do mesmo propiciou também o desenvolvimento das habilidades:

- Trabalho em equipe;
- Comunicação interpessoal,
- Distribuição de tarefas e obediência aos prazos pré estabelecidos.

DESCRIÇÃO DO PROJETO

1. Classe: Entity
    a. Membros:
        i. positionX: Um inteiro que representa a coordenada X da posição da entidade.
        ii. positionY: Um inteiro que representa a coordenada Y da posição da entidade.
        iii. movementSpeed: Um inteiro que representa a velocidade de movimento da entidade.
        iv. getUp1, getUp2, getDown1, getDown2, getLeft1, getLeft2, getRight1, getRight2: Essas são variáveis do tipo BufferedImage que representam diferentes sprites ou imagens usadas para a entidade. Parece que cada direção (cima, baixo, esquerda e direita) tem dois sprites correspondentes (1 e 2).
        v. spriteCounter: Uma variável inteira usada para acompanhar o contador de sprites atual.
        vi. isSpriteOne: Uma variável booleana que indica se o sprite atual sendo usado é o primeiro (true) ou o segundo (false).
        vii. direction: Uma variável do tipo String que representa a direção para a qual a entidade está voltada.

A classe Entity, apenas define o estado (campos) da entidade e não especifica nenhum comportamento.

2. Classe: Player
    a. Membros:
        i. powerThreshold: Um inteiro que representa o limite de energia do jogador.
        ii. powerCurrent: Um inteiro que representa a energia atual do jogador.
        iii. activeQuest: Uma Quest estática que representa a missão ativa do jogador.
        iv. coins: Um inteiro que representa a quantidade de moedas que o jogador possui.
        v. itens: Uma lista de itens pertencentes ao jogador.
        vi. gp: Uma instância da classe GamePanel.
        vii. city: Uma instância da classe City.

    b. Construtor:
        i. Player(GamePanel gp, City city): Um construtor que recebe uma instância de GamePanel e City. Define os valores padrão, posição padrão e carrega a imagem do jogador.

c. Métodos:
   i. setDefaultValues(): Define os valores padrão do jogador.
   ii. setDefaultPosition(): Define a posição padrão do jogador com base na cidade.
   iii. updatePosition(City city): Atualiza a posição do jogador com base na cidade fornecida.
   iv. getPlayerImage(): Carrega as imagens do jogador a partir dos arquivos.
   v. update(): Método para atualização do jogador. Atualmente, está comentado e vazio.
   vi. draw(Graphics2D g2): Desenha o jogador na posição atual com base na direção.
   vii. isAlive(): Verifica se o jogador está vivo com base na energia atual e no limite de energia.
   viii. hasCoins(): Verifica se o jogador possui moedas.
   ix. updateRewards(int coinReward, int powerThresholdReward): Atualiza as recompensas do jogador adicionando moedas e ajustando o limite de energia.
   x. getCoins(): Retorna a quantidade de moedas do jogador.
   xi. getPowerThreshold(): Retorna o limite de energia do jogador.
   xii. acceptQuest(Quest quest): Aceita uma missão, adicionando as moedas da recompensa e definindo-a como missão ativa.
   xiii. completeQuest(): Completa a missão ativa, desde que a cidade atual seja o destino da missão. Atualiza as recompensas e define a missão ativa como nula.
   xiv. abandonQuest(): Abandona a missão ativa, definindo-a como nula.
   xv. updateStats(int updatePower, int updateCoins): Atualiza as estatísticas do jogador, adicionando ou subtraindo energia e moedas.
   xvi. exchangeCoins(int coins, int powerThreshold): Troca moedas por um ajuste no limite de energia.

Essa classe representa um jogador com atributos como posição, velocidade de movimento, energia, moedas e missões ativas. Ele também possui métodos para atualizar e manipular esses atributos.

3. Classe: Item
   a. Membros:
      i. name: Uma string que representa o nome do item.
      ii. powerInfluence: Um inteiro que representa a influência de poder do item.

   b. Construtor:
      i. Item(String name, int powerInfluence): Um construtor que recebe um nome e uma influência de poder para criar um novo item.

c. Métodos:
    i. getName(): Retorna o nome do item.
    ii. getPowerInfluence(): Retorna a influência de poder do item.

Essa classe representa um item com um nome e uma influência de poder. Os itens podem ser criados com um nome e uma influência específicos, e os métodos de acesso permitem obter o nome e a influência de poder do item.

4. Classe: Quest
    a. Membros:
        i. name: Uma string que representa o nome da missão.
        ii. destination: Uma instância da classe City que representa o destino da missão.
        iii. coinsAid: Um inteiro que representa a quantidade de moedas de ajuda da missão.
        iv. reward: Uma instância da classe Reward que representa a recompensa da missão.
        v. isActive: Um booleano que indica se a missão está ativa.
        vi. isComplete: Um booleano que indica se a missão está completa.

    b. Construtor:
        i. Quest(String name, City destination, int coinsAid, Reward reward): Um construtor que recebe o nome da missão, o destino, a quantidade de moedas de ajuda e a recompensa para criar uma nova missão.

    c. Métodos:
        i. acceptQuest(): Aceita a missão, definindo-a como ativa e retornando a própria instância da missão.
        ii. completeQuest(): Completa a missão, marcando-a como completa e retornando a recompensa da missão.
        iii. isNotComplete(): Verifica se a missão não está completa.
        iv. abandonQuest(): Abandona a missão, definindo-a como inativa.

Essa classe representa uma missão com um nome, destino, quantidade de moedas de ajuda e uma recompensa. As missões podem ser aceitas, completadas, verificadas se estão completas e abandonadas usando os métodos fornecidos.

5. Classe: Reward
    a. Membros:
        i. itemReward: Uma instância da classe Item que representa a recompensa do item.

ii. coinsReward: Um inteiro que representa a quantidade de moedas como recompensa.

b. Construtor:
   i. Reward(Item itemReward, int coinsReward): Um construtor que recebe o item de recompensa e a quantidade de moedas de recompensa para criar uma nova recompensa.

c. Métodos:
   i. getItemReward(): Retorna o item de recompensa.
   ii. getCoinsReward(): Retorna a quantidade de moedas como recompensa.

Essa classe representa uma recompensa que pode ser obtida ao completar uma missão. Ela consiste em um item de recompensa e uma quantidade de moedas como recompensa. Os métodos de acesso permitem obter o item de recompensa e a quantidade de moedas.

6. Classe: City
   a. Membros:
      i. code: Um inteiro que representa o código da cidade.
      ii. name: Uma string que representa o nome da cidade.
      iii. frontiers: Uma lista de objetos Frontier que representa as fronteiras da cidade.
      iv. quest: Uma instância da classe Quest que representa a missão disponível na cidade.
      v. powerInfluence: Um inteiro que representa a influência de poder da cidade.
      vi. xPosition: Um inteiro que representa a posição X da cidade.
      vii. yPosition: Um inteiro que representa a posição Y da cidade.
      viii. getCity: Uma instância da classe BufferedImage que representa a imagem da cidade.
      ix. arrival: Um booleano que indica se o jogador chegou à cidade.

   b. Construtor:
      i. City(int code, String name, int powerInfluence, int xPosition, int yPosition): Um construtor que recebe o código, nome, influência de poder, posição X e posição Y para criar uma nova cidade.

   c. Métodos:
      i. getxPosition(): Retorna a posição X da cidade.
      ii. getyPosition(): Retorna a posição Y da cidade.
      iii. addFrontier(City destination, int cost): Adiciona uma fronteira à cidade, especificando o destino e o custo.
      iv. addQuest(Quest quest): Adiciona uma missão à cidade.

v. getQuest(): Retorna a missão disponível na cidade.
vi. getFrontier(): Retorna a lista de fronteiras da cidade.
vii. getCityImage(): Obtém a imagem da cidade.
viii. getPowerInfluence(): Retorna a influência de poder da cidade.
ix. getShortestPath(City destination): Retorna o caminho mais curto até uma cidade de destino.

Essa classe representa uma cidade no mundo do jogo. Ela contém informações como o código, nome, fronteiras, missão disponível, influência de poder e posição da cidade. Os métodos fornecidos permitem adicionar fronteiras, adicionar uma missão, obter informações sobre a cidade e encontrar o caminho mais curto até uma cidade de destino.

7. Classe: CityManager
    a. Membros:
        i. cities: Um HashMap que mapeia o nome da cidade para uma instância da classe City.
        ii. gp: Uma instância da classe GamePanel.
        iii. cityCurrent: Uma instância da classe City que representa a cidade atual.
        iv. cityLast: Uma instância da classe City que representa a cidade anterior.
        v. cityDestination: Uma instância da classe City que representa a cidade de destino.
        vi. scanner: Uma instância da classe Scanner para entrada de dados.

    b. Construtor:
        i. CityManager(GamePanel gp): Um construtor que recebe uma instância de GamePanel e inicializa os membros da classe.

    c. Métodos:
        i. setDefaultValues(): Define os valores padrão para as cidades atual, anterior e de destino.
        ii. activateEndGame(): Verifica se a cidade atual é igual à cidade de destino e ativa o estado final do jogo.
        iii. getCityCurrent(): Retorna a cidade atual.
        iv. setCityCurrent(City city): Define a cidade atual.
        v. getCities(): Retorna o HashMap de cidades.
        vi. loadCityInfo(): Carrega as informações das cidades a partir do arquivo "cityInfo.txt".
        vii. loadCityFrontiers(): Carrega as fronteiras das cidades a partir do arquivo "cityFrontiers.txt".
        viii. loadCityQuests(): Carrega as missões das cidades a partir do arquivo "cityQuests.txt".
        ix. draw(Graphics2D g2): Desenha as cidades na tela.

x. navigate(City city, int coinsCost): Navega para uma cidade específica, atualizando as estatísticas do jogador e completando a missão ativa.
xi. navigateShortestPath(): Retorna o caminho mais curto até a cidade de destino.
xii. pathListToString(List<City> path): Converte a lista de cidades do caminho em uma string formatada.

Essa classe gerencia as cidades no jogo. Ela carrega as informações das cidades, suas fronteiras e missões a partir de arquivos de texto. Ela também permite a navegação entre as cidades, o cálculo do caminho mais curto e o desenho das cidades na tela.

8. Classe: Frontier
    a. Membros:
        i. destination: Uma instância da classe City que representa a cidade de destino.
        ii. coinsCost: Um valor inteiro que representa o custo em moedas para alcançar a cidade de destino.

    b. Construtor:
        i. Frontier(City destination, int coinsCost): Um construtor que recebe uma instância da classe City e o custo em moedas, e inicializa os membros da classe.

    c. Métodos:
        i. getCityDestination(): Retorna a cidade de destino.
        ii. getCoinsCost(): Retorna o custo em moedas para alcançar a cidade de destino.

Essa classe representa uma fronteira entre duas cidades no jogo. Ela armazena a cidade de destino e o custo em moedas para alcançar essa cidade a partir da cidade atual.

9. Classe: Tile
    a. Membros:
        i. image: Uma instância da classe BufferedImage que representa a imagem associada ao tile.
        ii. collision: Uma variável booleana que indica se o tile possui uma colisão (true) ou não (false). Por padrão, é definido como false.

Essa classe representa um tile no jogo, com uma imagem associada a ele e uma indicação opcional de colisão. Ela pode ser usada para construir um mapa do mundo do jogo, onde cada tile possui sua própria imagem e propriedades. A propriedade de colisão pode ser usada para determinar se os jogadores ou objetos podem atravessar ou interagir com determinados tiles.

10. Classe: TileManager
    a. Membros:
        i. gp: Uma instância da classe GamePanel que representa o painel de jogo.
        ii. tile: Um array de objetos Tile que armazena os diferentes tipos de tiles do jogo.
        iii. mapTileNum: Uma matriz de inteiros que representa os números dos tiles no mapa.

    b. Construtor:
        i. O construtor da classe recebe uma instância de GamePanel como argumento e inicializa os membros correspondentes. Ele também chama os métodos getTileImage() e loadMap() para carregar as imagens dos tiles e o mapa do arquivo, respectivamente.

    c. Métodos:
        i. getTileImage(): Esse método carrega as imagens dos diferentes tipos de tiles do jogo. Ele utiliza a classe ImageIO para ler as imagens dos arquivos correspondentes e as armazena nos objetos Tile do array tile. Cada imagem é associada a um número de tile específico.
        ii. loadMap(): Esse método carrega o mapa do jogo a partir de um arquivo de texto. Ele lê cada linha do arquivo, que contém números separados por espaços, e armazena esses números na matriz mapTileNum. Cada número representa o tipo de tile a ser exibido na posição correspondente do mapa.
        iii. draw(Graphics2D g2): Esse método desenha os tiles do mapa na tela. Ele percorre a matriz mapTileNum e, para cada posição, obtém o número do tile correspondente. Em seguida, desenha a imagem desse tile na posição adequada no painel de jogo.

Essa classe é responsável por gerenciar os tiles do mapa do jogo. Ela carrega as imagens dos tiles e o mapa do arquivo, além de desenhar os tiles na tela durante a renderização do jogo.

11. Classe: Shopkeeper
    a. Membros:
        i. coinCount: O número de moedas do comerciante.
        ii. jewelPower: O poder da joia do comerciante.
        iii. tradeCoins: Um sinalizador booleano indicando se o comerciante está disposto a trocar moedas.
        iv. dialogues: Uma matriz de strings que armazena os diálogos do comerciante.
        v. dialogueLevel: O nível atual de diálogo.
        vi. ui: Uma instância da classe UI para exibir a interface do usuário.

vii. cityManager: Uma instância da classe CityManager para gerenciar as cidades do jogo.
viii. player: Uma instância da classe Player para representar o jogador.
ix. gp: Uma instância da classe GamePanel que representa o painel de jogo.
x. dialogCounter: Um contador de diálogos.

b. Construtor:
   i. O construtor da classe recebe instâncias de UI, CityManager, Player e GamePanel como argumentos e inicializa os membros correspondentes. Ele também define o contador de diálogos como 0.

c. Métodos:
   i. welcome(): Esse método é responsável por iniciar a conversa com o comerciante. Ele verifica se há diálogos disponíveis e decide se deve falar ou não.
   ii. talkStatus(): Esse método lida com a resposta do jogador sobre o estado atual do jogo. Ele verifica se há diálogos disponíveis e decide se deve falar ou não.
   iii. askCityDestination(): Esse método lida com a resposta do jogador sobre a cidade de destino desejada. Ele verifica se há diálogos disponíveis e decide se deve falar ou não.
   iv. setDialogue(): Esse método preenche a matriz de diálogos com as falas iniciais do comerciante.
   v. setDialogueStatus(): Esse método preenche a matriz de diálogos com informações sobre o estado atual do jogador.
   vi. setDialogueQuest(): Esse método preenche a matriz de diálogos com informações sobre a missão disponível.
   vii. setDialogueCity(): Esse método preenche a matriz de diálogos com a pergunta sobre a cidade de destino desejada.
   viii. setDialogueExchange(): Esse método preenche a matriz de diálogos com a pergunta sobre a troca de moedas.
   ix. checkQuest(): Esse método verifica se há uma missão disponível na cidade atual e se o jogador não possui uma missão ativa. Retorna true se uma missão estiver disponível e false caso contrário.
   x. speak(): Esse método lida com a fala do comerciante. Ele verifica o nível de diálogo atual e exibe o diálogo correspondente na interface do usuário. Ele também verifica se há uma missão disponível e exibe o diálogo da missão, se aplicável.
   xi. updateDialogue(): Esse método atualiza o índice de diálogo para avançar para a próxima fala.

xii. endConversation(Frontier destination): Esse método encerra a conversa com o comerciante e navega até a cidade de destino fornecida como argumento.
xiii. exchange(): Esse método realiza a troca de moedas do jogador. Dependendo da quantidade de moedas que o jogador possui, diferentes quantidades de moedas são trocadas por poder da joia.
xiv. getCoinCount(): Esse método retorna o número de moedas do comerciante.
xv. getJewelPower(): Esse método retorna o poder da joia do comerciante.

Essa classe representa o comerciante do jogo. Ele lida com as interações do jogador com o comerciante, exibe diálogos, oferece missões e realiza trocas de moedas. Além disso, ele mantém o controle do estado do jogo relacionado ao comerciante, como o número de moedas e o poder da joia.

DESCRIÇÃO DA EQUIPE

Nesta seção, apresentamos a equipe responsável pelo desenvolvimento do projeto da disciplina Estrutura de Dados. Cada membro desempenhou um papel fundamental na criação e implementação do jogo, contribuindo com suas habilidades e conhecimentos específicos. A seguir, descrevemos as atividades realizadas por cada desenvolvedor:

Desenvolvedor 1: Carlos Mariano

1. Criação do layout de elementos visuais:
    a. Projetar ícones, fundos e personagens do jogo.
    b. Utilizar ferramentas de design gráfico para criar imagens e gráficos.
    c. Estudar referências visuais e selecionar cores adequadas para o ambiente virtual.
    d. Criar um estilo visual coeso e atraente para o jogo.
2. Programação da navegação do personagem pelas cidades do jogo:
    a. Desenvolver algoritmos para permitir a movimentação do personagem em diferentes direções.
    b. Implementar a interação do personagem com o ambiente virtual.
    c. Aplicar programação orientada a objetos para garantir a modularidade e reutilização de código.
3. Desenvolvimento do player do jogo:
    a. Criar o código responsável pelo controle do personagem pelo jogador.
    b. Implementar os controles de movimento, pulo e interação com o ambiente.
    c. Programar as ações especiais ou habilidades do personagem, se aplicável.
    d. Testar e otimizar o desempenho e a jogabilidade do player.
4. Produção da documentação do relatório:
    a. Elaborar a documentação técnica do projeto, descrevendo as etapas e os processos envolvidos.
    b. Detalhar as escolhas de design gráfico e explicar as técnicas utilizadas.
    c. Documentar a lógica de programação implementada para a navegação do personagem e o funcionamento do player.
    d. Incluir diagramas, fluxogramas ou outros recursos visuais para facilitar a compreensão do projeto.
    e. Revisar e editar o relatório para garantir clareza e coerência.

Desenvolvedor 2: Gabriel Coelho

1. Desenvolvimento da arquitetura da interface gráfica:
    a. Criou a estrutura e organização dos elementos visuais do jogo.
    b. Definiu a hierarquia e relação entre as diferentes telas e componentes gráficos.
    c. Planejou e implementou a interação entre a interface gráfica e a lógica do jogo.
2. Configurações básicas de exibição da janela (main):
    a. Definiu as configurações iniciais da janela do jogo, como tamanho, título e ícone.
    b. Gerenciou as ações de inicialização e encerramento do jogo.
    c. Estabeleceu as opções de renderização e atualização da tela.
3. Controle do loop do jogo (GamePanel):
    a. Implementou o loop principal do jogo, controlando a renderização e atualização contínuas da tela.
    b. Gerenciou a lógica de tempo, incluindo o controle de FPS (quadros por segundo).
    c. Tratou eventos do teclado e mouse para interações do jogador.
4. Gerenciamento de interface do usuário e estados do jogo (KeyHandler):
    a. Criou o gerenciador de eventos do teclado para lidar com as ações do jogador.
    b. Implementou a lógica para controlar os diferentes estados do jogo, como pausa, menus e diálogos.
    c. Manipulou a troca de telas, exibição de diálogos e interações com os elementos da interface.
5. Implementação das classes de lógica do jogo:
    a. Criou classes específicas para controlar a lógica do jogo, como movimento, colisões, pontuação, etc.
    b. Desenvolveu os respectivos métodos "draw" para renderizar os elementos visuais correspondentes a cada classe.
    c. Integrou essas classes à interface gráfica, garantindo que a lógica e a renderização funcionassem em conjunto.

Desenvolvedor 3: Paulo Vinicius

1. Criação da lógica do MERCADOR:

   a. Desenvolvimento da lógica envolvendo a manipulação dos dados da quantidade de moedas do jogador.
   b. Implementação do controle do limiar da joia, gerenciando as interações do jogador com o mercador.
   c. Direcionamento do personagem para novas cidades por meio da lógica do MERCADOR.
2. Desenvolvimento do player do jogo:
   a. Criação do código responsável pelo controle do personagem pelo jogador.
   b. Implementação dos controles de movimento, pulo e interação com o ambiente.
   c. Programação das ações especiais ou habilidades do personagem, se aplicável.
   d.
3. Desenvolvimento do relatório do projeto do jogo:
   a. Elaboração da documentação técnica do projeto, descrevendo as etapas e os processos envolvidos.
   b. Detalhamento das atividades desenvolvidas, incluindo a lógica do mercador, o player do jogo e outras funcionalidades implementadas.
   c. Inclusão de diagramas, fluxogramas ou outros recursos visuais para facilitar a compreensão do projeto.
4. Criação do handler:
   a. Implementação do handler para lidar com eventos e interações do jogo.
   b. Desenvolvimento das funções responsáveis por tratar as ações do jogador, como pausar o jogo, acessar menus ou diálogos.
5. Desenvolvimento da metodologia de carregamento do grafo por arquivos externos (função loadMap) e loadQuest.
   a. Implementação da funcionalidade para carregar o mapa do jogo a partir de arquivos externos.
   b. Desenvolvimento da lógica para carregar as quests (missões) do jogo por meio de arquivos externos.

Cada membro da equipe desempenhou um papel essencial na criação e desenvolvimento do jogo, trabalhando em colaboração para alcançar os objetivos do projeto. Suas habilidades individuais combinadas resultaram em um produto final coeso e funcional.

METODOLOGIA

FERRAMENTAS UTILIZADAS

Para a realização do seguinte projeto, foram utilizadas as seguintes ferramentas:

- Eclipse IDE for Enterprise Java and Web Developers e IntelliJ IDEA:
    - Ambientes de desenvolvimento integrados para formulação e edição de código em JAVA, utilização de terminal de comando, execução de script, debug e compilação.
    - O uso dessas ferramentas possibilitou uma melhor construção do jogo do projeto de forma íntegra, ágil e dinâmica.
- Photoshop CS6:
    - Ferramenta de criação e edição de imagens e designers gráficos.
    - O uso dessa ferramenta permitiu a criação dos personagens do jogo e alguns elementos visuais e layout do Relatório Técnico do projeto.
- Microsoft Word 2019:
    - Editor de texto:
    - Ferramenta utilizada para realização de resumos acerca do projeto e da confecção do Relatório Técnico do mesmo.

ESTRUTURA DO CÓDIGO

O código apresentado demonstra a utilização de vários conceitos fundamentais da linguagem de programação Java. Esses conceitos são essenciais para o desenvolvimento de programas orientados a objetos eficientes e bem estruturados. Listamos a seguir os conceitos utilizados:

1. **Atributos e Variáveis:**

   São elementos que armazenam dados durante a execução de um programa. Ambos são utilizados para representar informações e valores que são manipulados e processados pelo programa.

   Na classe “Entity”, os atributos "positionX" e "positionY" armazenam as coordenadas da posição da entidade no jogo, enquanto o atributo "movementSpeed" registra a velocidade de movimento da entidade. Esses atributos permitem controlar e atualizar a posição da entidade, bem como definir a sua velocidade de deslocamento. Além disso, o código faz uso de atributos do tipo BufferedImage, como "getUp1", "getUp2", "getDown1", "getDown2", "getLeft1", "getLeft2", "getRight1" e "getRight2". Esses atributos armazenam imagens relacionadas à entidade, permitindo a exibição de diferentes sprites de acordo com a direção ou a ação realizada pela entidade no jogo. A utilização de objetos do tipo BufferedImage demonstra a capacidade do Java de lidar com manipulação de imagens.

2. **Atributos Finais (Final):**

   Outro conceito aplicado neste código é o de atributos finais (final). Os atributos finais garantem que os valores desses atributos sejam imutáveis, o que é útil quando se deseja evitar alterações indesejadas ou inesperadas nos objetos após a sua criação.

3. **Classes e Objetos**

   Em Java, as classes são estruturas fundamentais que servem como modelos para a criação de objetos. Já os objetos são instâncias concretas de uma classe. Eles representam entidades individuais que podem ser manipuladas e interagir com outros objetos no programa. A classe "Entity", por exemplo, é definida para representar uma entidade no jogo. Os objetos dessa classe são criados para representar entidades específicas no jogo.

A classe "Player" é definida como uma subclasse da classe "Entity" e representa um jogador no jogo. Através dessa classe, é possível criar objetos que encapsulam as características e comportamentos específicos de cada jogador. Isso permite a reutilização do código e uma organização lógica do programa.

4. **Coleções**

Estruturas de dados que permitem armazenar e manipular grupos de elementos de maneira organizada e eficiente. As coleções oferecem métodos e funcionalidades que facilitam a manipulação, busca, ordenação e iteração dos elementos contidos nelas. A classe "Player" utiliza a interface List<Item> para armazenar uma lista de itens. Essa lista é utilizada para representar os itens que o jogador possui no jogo.

5. **Construtor**

Método especial responsável por inicializar os atributos do objeto quando uma instância da classe é criada. Através do uso do construtor, é possível criar objetos com valores específicos para seus atributos, garantindo a individualidade de cada instância da classe.

6. **Desenhos e gráficos**

São utilizadas classes como Graphics e Graphics2D para realizar o desenho dos componentes gráficos na tela. Métodos como paintComponent() são sobrescritos para personalizar o desenho dos objetos.

7. **Encapsulamento**

Conceito fundamental na programação orientada a objetos que visa proteger os dados de uma classe e controlar o acesso a eles. Ele envolve o agrupamento de atributos e métodos em uma classe, e a definição de visibilidade para esses elementos. Como exemplo, temos os atributos da classe "Player", que são definidos como privados (exceto alguns que são públicos) para encapsular o estado interno da classe. Métodos públicos são utilizados para acessar e modificar esses atributos de forma controlada.

8. **Herança**

Conceito fundamental da programação orientada a objetos que permite que uma classe herde características e comportamentos de outra classe. A herança permite criar hierarquias de classes, onde classes mais especializadas podem estender e herdar os atributos e métodos de classes mais genéricas ou abstratas.

9. **Interfaces**

As interfaces em Java são estruturas que definem um contrato para as classes implementarem. Elas especificam um conjunto de métodos que as classes devem implementar e fornecem um mecanismo para alcançar a abstração e o polimorfismo. As interfaces são usadas para definir contratos e permitir que várias classes diferentes tenham comportamentos semelhantes. A classe GamePanel implementa a interface Runnable, que permite que ela seja executada em uma thread separada. Isso é útil para controlar o fluxo do jogo e atualizar a interface gráfica.

No código do jogo, a classe "Player" herda da classe "Entity", o que significa que ela possui todos os atributos e métodos da classe pai e pode adicionar funcionalidades específicas.

10. **Manipulação de Imagens:**

A manipulação de imagens em Java envolve o uso de classes e métodos fornecidos pela biblioteca Java AWT (Abstract Window Toolkit) e Java Swing. Essas classes permitem que os desenvolvedores leiam, processem, modifiquem e exibam imagens em seus aplicativos Java. A classe ImageIO é utilizada para carregar imagens do jogador a partir de arquivos de recursos.

11. **Manipulação de Eventos:**

A manipulação de eventos em Java envolve o gerenciamento e resposta a ações e interações do usuário em um programa. Os eventos podem ser gerados por diferentes fontes, como cliques de mouse, pressionamentos de teclas, movimentos do mouse, alterações em componentes gráficos, entre outros. O objetivo da manipulação de eventos é capturar e processar esses eventos para executar ações correspondentes. A classe "Player" recebe um objeto KeyHandler no construtor, que é usado para manipular eventos de teclado e controlar o movimento do jogador.

12. **Métodos de Acesso (Getters):**

Para permitir o acesso aos atributos privados, o código utiliza métodos de acesso (getters). Esses métodos fornecem uma interface controlada para obter informações específicas sobre o item, mantendo a proteção dos atributos e permitindo que o código cliente utilize os dados de forma segura. Na classe “Item”, por exemplo, temos os

métodos públicos "getName()" e "getPowerInfluence()", que são definidos para permitir o acesso controlado aos atributos privados. Esses métodos permitem obter os valores dos atributos correspondentes.

13. Modificadores de Acesso

São palavras-chave utilizadas para definir o nível de visibilidade e acesso de classes, atributos, métodos e construtores em um programa. Eles controlam a forma como esses elementos podem ser acessados e modificados por outras partes do código. Na classe "Entity", por exemplo, os atributos da classe "Entity" são definidos como públicos. Isso significa que esses atributos podem ser acessados diretamente de outros objetos e classes, facilitando a manipulação e atualização de suas informações. No entanto, é importante ressaltar que a utilização adequada dos modificadores de acesso é fundamental para garantir a encapsulação e a segurança dos dados.

14. Pacotes

O uso de pacotes é uma prática comum em Java, permitindo organizar e agrupar classes relacionadas em um mesmo contexto. Isso torna o código mais legível, evita conflitos de nomes e facilita a sua utilização em outros projetos. O código está organizado em pacotes (entity, main, quest, item e world), permitindo uma estrutura modular e organizada para as classes relacionadas.

15. Polimorfismo

Permite que objetos de diferentes classes sejam tratados de forma uniforme por meio de uma referência da classe pai. Em outras palavras, o polimorfismo permite que um objeto seja visto e manipulado de várias maneiras, dependendo do contexto em que é utilizado. O polimorfismo é aplicado quando um objeto da classe "Player" é tratado como um objeto da classe "Entity", permitindo o uso de polimorfismo de tipo.

16. Sobrescrita de métodos (method overriding):

Quando uma classe filha herda um método da classe pai, ela tem a opção de modificar ou estender o comportamento desse método através da sobrescrita. Para isso, a classe filha declara um método com a mesma assinatura (nome, tipo de retorno e parâmetros) que o método da classe pai. Na classe "KeyHandler", os métodos keyTyped, keyPressed e keyReleased estão sendo sobrescritos na classe KeyHandler. Esses métodos

substituem as implementações padrão da interface KeyListener para lidar com eventos de teclado específicos.

17. Tratamento de Exceções:

Permite lidar com situações excepcionais ou erros que podem ocorrer durante a execução de um programa. O tratamento de exceções é importante para garantir que o programa possa se recuperar de erros e continuar sua execução normalmente.

O código da classe “Player” utiliza o bloco try-catch para lidar com possíveis exceções durante o carregamento de imagens usando a classe ImageIO.

18. Uso de Bibliotecas e Importação:

A importação de bibliotecas é necessária para utilizar os recursos e funcionalidades dessa classe pré-moldada na ferramenta, demonstrando a capacidade do Java de estender suas funcionalidades através do uso de bibliotecas externas. A classe “Player” por exemplo, faz uso de classes e métodos fornecidos por bibliotecas padrão do Java, como ImageIO e BufferedImage, para realizar operações específicas, como carregar imagens e manipular gráficos.

CONCLUSÃO

O projeto de criação de um jogo eletrônico de estratégia desktop em Java, apresentado neste relatório, foi um empreendimento desafiador e bem-sucedido. A equipe de desenvolvimento, composta por Carlos Mariano, Gabriel Coelho e Paulo Vinicius, demonstrou habilidades técnicas e conhecimentos específicos para criar um jogo coeso e funcional.

Durante o desenvolvimento, foram aplicados diversos conceitos fundamentais da programação orientada a objetos, como atributos e variáveis, classes e objetos, encapsulamento, herança e interfaces. Além disso, foram utilizadas ferramentas como Eclipse IDE for Enterprise Java and Web Developers, IntelliJ IDEA e Photoshop CS6 para a construção do jogo e a criação de elementos visuais.

A estrutura do código foi projetada de forma organizada, garantindo a modularidade e reutilização de código, e permitindo a implementação de funcionalidades como a movimentação do personagem, interações com o ambiente, controle de estados do jogo, manipulação de dados do jogador e carregamento do mapa e missões a partir de arquivos externos.

A equipe demonstrou habilidades de trabalho em equipe, comunicação interpessoal e distribuição de tarefas, cumprindo prazos pré-estabelecidos e colaborando para alcançar os objetivos do projeto.

Em conclusão, o jogo eletrônico de estratégia desenvolvido apresenta um enredo envolvente e desafiador, com a missão de levar o personagem Maxwell de Ubud a Nargumun, enfrentando diversas situações e completando missões ao longo do caminho. O projeto reflete o aprendizado e a aplicação dos conceitos de estrutura de dados e algoritmos estudados durante o curso de Engenharia de Software, além de destacar as habilidades individuais e coletivas da equipe de desenvolvimento.

O resultado final do projeto é um jogo eletrônico de qualidade, que combina elementos visuais atrativos, jogabilidade envolvente e desafios estratégicos. Através desse trabalho, os membros da equipe puderam aprimorar suas habilidades técnicas, adquirir experiência na criação de jogos e consolidar os conhecimentos teóricos da disciplina de Estrutura de Dados.

Por fim, o jogo eletrônico de estratégia desktop desenvolvido em Java representa não apenas um produto final de qualidade, mas também uma conquista pessoal e acadêmica para a equipe

envolvida. O trabalho em equipe, a aplicação de conhecimentos técnicos e a superação de desafios são aspectos que fortaleceram a capacidade dos desenvolvedores de enfrentar futuros projetos e contribuir para o avanço da área de desenvolvimento de jogos eletrônicos.

REFERÊNCIAS

Lima, Guilherme. **Saiba tudo sobre o IDE - Integrated Development Environment.** 2022. Disponível em: < https://www.alura.com.br/artigos/o-que-e-uma ide#:~:text=O%20ambiente%20de%20desenvolvimento%20integrado,compilar%20usando%20um%20%C3%BAnico%20ambiente.>. Acesso em 20 de junho de 2023.

RyiSnow. **How to Make a 2D Game in Java.** 2022. Disponível em: < https://www.youtube.com/playlist?list=PL_QPQmz5C6WUF-pOQDsbsKbaBZqXj4qSq>. Acessado em 18 de junho de 2023.