PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL

ESCOLA POLITÉCNICA

CURSO DE ENGENHARIA DE SOFTWARE

LUCAS DA PAZ OLIVEIRA

ACME AIR DRONES

PORTO ALEGRE

2024

LUCAS DA PAZ OLIVEIRA

ACME AIR DRONES

Trabalho apresentado no curso de Engenharia de Software da Pontifícia Universidade Católica do Rio Grande do Sul, referente à cadeira de Programação Orientada a Objetos.

Professor: Marcelo Hideki Yamaguti

PORTO ALEGRE

2024

SUMÁRIO

I. INTRODUÇÃO

O trabalho final da disciplina consiste na implementação de um sistema com interface gráfica e persistência de dados em arquivos para uma empresa fictícia de transporte por drones. A linguagem de programação empregada foi *Java*, a ferramenta construtora de *layouts* para a interface do usuário utilizada foi a *NetBeans IDE GUI Builder* e, para o *build* e resolução de dependências, valeu-se da ferramenta *Apache Maven*. Foram criados testes unitários com *JUnit* e o sistema foi testado em dispositivos com sistema operacional *Windows 11*, *Linux Mint* e *macOS Sequoia*.

O presente relatório consiste, basicamente, de três seções principais; a primeira aborda o diagrama de classes do projeto, a segunda apresenta as coleções de dados utilizadas e suas descrições e a terceira discorre sobre as formas de armazenamento de dados em arquivos de texto que foram utilizadas. Discute-se, também, sobre o *design* e a execução da aplicação. Por fim, conclui-se o relatório e o projeto.

II. DESENVOLVIMENTO

Nesta seção discorre-se sobre o desenvolvimento do trabalho, tratando das seções abordadas na introdução deste documento.

2.1 Diagrama de Classes

O diagrama de classes do projeto está disponível em três formatos: arquivo *.asta*, SVG e PNG, todos localizados dentro da pasta *docs* partindo do diretório raiz. Dependências consideradas importantes (e.g.: a utilização de uma enumeração em um método de uma classe), mesmo quando sua sinalização era opcional, foram explicitadas no diagrama.

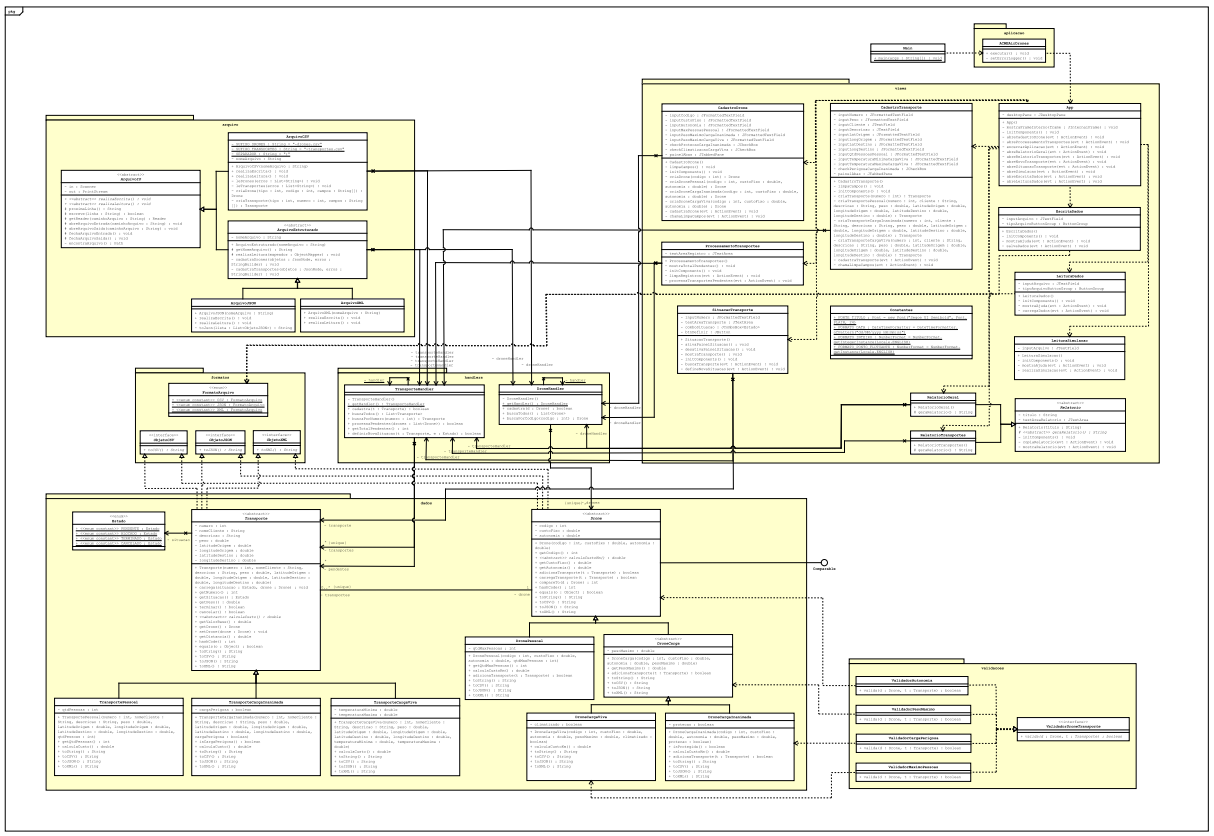

Figura 1: Visão geral do diagrama de classes do projeto.

O desenvolvimento contemplou um total de 40 classes, separadas em 7 pacotes. Excetuando-se a classe *Main*, localizada no pacote padrão, as demais classes foram agrupadas por coerência:

- Pacote "*aplicacao*": Contém a classe que configura o sistema e chama a interface gráfica do usuário;
- Pacote "*arquivo*": Classes utilizadas para trabalhar com leitura e escrita de arquivos;
- Pacote "*dados*": Modelos utilizados na aplicação (e.g. drones, transportes etc.);

- Pacote "*formatos*": Conserva interfaces e enumerações relacionadas aos formatos de arquivo com os quais o sistema trabalha;
- Pacote "*handlers*": Armazena as classes catálogo utilizadas pela aplicação;
- Pacote "*validacoes*": Classes e interfaces que encapsulam validações dos modelos;
- Pacote "*views*": Telas e demais classes relacionadas à interface gráfica.

Todas as classes e grande maioria dos métodos estão documentados segundo o padrão *Javadoc*, explicando seu funcionamento e detalhando sua utilização.

2.2 Coleções de Dados

De modo geral, o armazenamento de dados em coleções ficou a cargo de duas classes catálogo, denominadas "*handlers*": *DroneHandler* e *TransporteHandler*. Também houve uso de coleções para representar a associação entre uma instância de drone e instâncias de transportes relacionadas.

Considerando como identificadores únicos o código do drone e o número do transporte, foram sobrescritos os métodos *equals* e *hashcode* nas respectivas classes; além disso, a classe *Drone* também implementou a *interface Comparable*, favorecendo a ordenação entre drones utilizando seu código.

Com relação às classes catálogo e considerando os identificadores únicos definidos, foram utilizados conjuntos (*sets*) do pacote *java.util*. De modo mais específico: considerando a restrição imposta de que drones devem ser ordenados por código, a classe *DroneHandler* utiliza um *TreeSet* de drones para realizar o armazenamento; já a classe *TransporteHandler* utiliza um *HashSet* de transportes para armazenar todos os transportes cadastrados, além de uma fila (*interface Queue* do pacote *java.util*) implementada com *LinkedList* para armazenar os transportes com situação "PENDENTE".

Na classe *Drone* utilizou-se um *HashSet* para armazenar os transportes relacionados a cada instância de drone. Por fim, foram utilizadas listas (classes que implementam a *interface List* do pacote *java.util*) em diversos métodos, e.g. listas imutáveis para retornar o conteúdo armazenado pelas classes *handlers*, para realizar a validação de transportes com relação aos drones, para definir os valores de um *JComboBox* etc.

2.3 Persistência de Dados

Foram implementadas classes específicas para trabalhar com o armazenamento de dados em arquivos. No pacote "*formatos*" estão dispostas uma enumeração chamada

FormatoArquivo, que contém constantes para cada formato aceito, e 3 *interfaces* chamadas *ObjetoCSV*, *ObjetoJSON* e *ObjetoXML* que servem para, respectivamente, retornar uma representação do objeto em formato CSV, JSON ou XML.

No pacote "*Arquivo*", a classe abstrata *ArquivoIO* serve para buscar, abrir e fechar arquivos, tanto para entrada quanto para saída. Também permite realizar operações de leitura e de escrita, agindo como um *wrapper*. É importante ressaltar que a busca de arquivos foi implementada de modo *case insensitive*, i.e., o sistema não reconhece a diferença entre letras maiúsculas e minúsculas na hora de buscar arquivos.

Outra classe com comportamento semelhante é a classe *ArquivoEstruturado*; ela serve para encapsular a lógica de leitura de arquivos JSON e XML, dado que foi utilizada a biblioteca *jackson-databind* (juntamente com *jackson-dataformat-xml* e *woodstox-core*) para a leitura destes formatos de arquivo e seu código é muito similar.

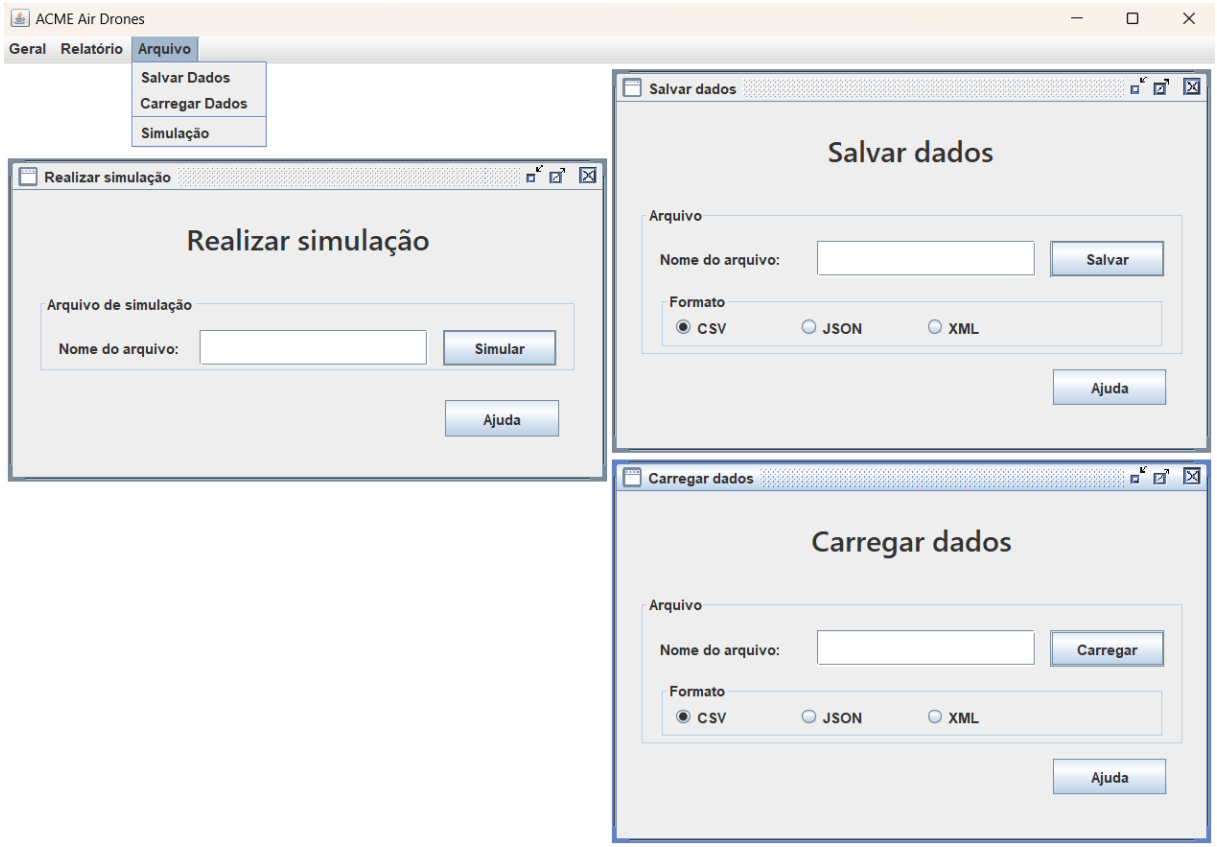

Figura 2: Aplicação mostrando o menu "Arquivo" e as telas utilizadas para salvar, carregar e simular dados.

Para escrita de dados em arquivos, o usuário informa o nome do arquivo desejado (sem extensão), seleciona o formato e clica em "Salvar"; caso o arquivo especificado já exista, seu conteúdo será sobrescrito. No caso dos formatos JSON e XML, todos os dados da aplicação

são salvos em apenas um arquivo, com o nome especificado e o formato escolhido; no caso do formato CSV, são gerados dois arquivos, um para os drones e outro para os transportes, utilizando os sufixos "-drones.csv" e "-transportes.csv", respectivamente.

Para ler dados de arquivos o procedimento é similar: o usuário informa o nome do arquivo que contém os dados (sem extensão), seleciona o formato e clica em "Carregar". O sistema buscará pelo arquivo com o nome especificado e o formato escolhido; caso não exista, uma mensagem de erro é exibida, caso exista, o arquivo será lido e os dados cadastrados no sistema. Para os formatos JSON e XML é esperado apenas um arquivo; no caso do formato CSV são esperados dois arquivos, um para os drones e outro para os transportes, utilizando os sufixos "-drones.csv" e "-transportes.csv", respectivamente.

Todos os atributos dos drones são exportados, excetuando-se a lista de transportes associados. Para os transportes, também, todos os atributos são exportados, respeitando-se que caso o drone relacionado não seja nulo, o campo "*codDrone*" conterá o código deste drone; caso contrário, o valor deste campo será 0. Para garantir o funcionamento deste formato, é garantido na leitura que todos os drones serão cadastrados antes dos transportes e, para cada transporte com código de drone válido, a situação informada e o drone localizado serão associados ao transporte, bem como o transporte será adicionado ao conjunto de transportes do drone em questão.

A simulação de dados permite apenas o uso de arquivos CSV. No contexto da simulação, os transportes declarados no arquivo de transportes não devem conter o campo "*situacao*" e nem o campo "*codDrone*", tendo em vista que cada transporte cadastrado para simulação estará com a situação "PENDENTE".

Cada uma das telas de arquivos possui um botão "Ajuda" que exibe para o usuário uma mensagem explicativa sobre a funcionalidade em questão; além disso, em caso de dificuldades com a leitura ou escrita de arquivos (e até mesmo com o sistema como um todo), é possível consultar o *log* de erros, armazenado na pasta *logs*. O *log* é sobrescrito a cada execução da aplicação; sendo assim, se for preciso preservar um log específico é necessário copiá-lo para um novo arquivo (ou renomear o arquivo existente) antes de executar novamente a aplicação.

2.4 Sobre o *Design*

Foi utilizada uma tela (*JFrame*) principal: a classe *App*; ela contém um menu com opções para abrir todas as outras telas, implementadas como *frames* internos (*JInternalFrame*) e que podem ser adicionados ao *JDesktopPane* da tela principal.

Para a organização dos *layouts* das telas foi utilizada a ferramenta *NetBeans IDE GUI Builder* e, na maioria nos painéis, *GroupLayout*, conforme recomendado em *The Java™ Tutorials* (disponível em <https://docs.oracle.com/javase/tutorial/uiswing/layout/using.html>). Cada grupo de componentes foi agrupado em painéis específicos, dispostos organizadamente dentro do *frame* trabalhado, e foram definidas lacunas (*gaps*) de tamanho variável, garantindo o alinhamento e o tamanho dos componentes conforme definidos no código.

Devido ao uso da ferramenta, para que seja possível visualizar os *layouts* na *IDE NetBeans*, alguns trechos de código gerados pela ferramenta não devem ser modificados (e.g.: declaração dos componentes, assinatura dos métodos ligados aos eventos, método *initComponents*); estes códigos estão delimitados entre comentários nas classes do pacote *views* (e.g. *GEN-FIRST*/*GEN-LAST* e *GEN-BEGIN*/*GEN-END*). Outra peculiaridade da ferramenta que pode ser ressaltada é a declaração dos componentes utilizados no final do arquivo.

Foi criada uma classe de constantes que contém a fonte utilizada para os títulos e os formatos utilizados para datas, números inteiros e com ponto flutuante. Se mais estilizações fossem realizadas (e.g. outras fontes, cores etc.), seriam declaradas neste arquivo.

2.5 Execução

O projeto pode ser executado a partir de qualquer *IDE* com suporte à linguagem *Java*. Também é possível compilar, empacotar e executar a aplicação utilizando *Maven* pela linha de comando. E.g.:

- *mvn clean*: limpa o diretório *target*, onde estão localizados os arquivos gerados;
- *mvn compile*: compila a aplicação dentro do diretório *target*;
- *mvn test*: executa os testes automatizados;
- *mvn package*: gera um pacote *.jar* da aplicação dentro do diretório *target*;
- *mvn exec:java*: executa a aplicação (deve ser usado após o *goal* "*mvn compile*").

É possível utilizar mais de um *goal*, e.g. "*mvn clean compile exec:java*" irá limpar o diretório target, compilar a aplicação e executá-la, nesta ordem.

III. CONCLUSÃO

Neste projeto foram utilizados diversos conceitos trabalhados durante o semestre, incluindo UML, herança, polimorfismo, tratamento de exceções, interface gráfica, padrões de projeto, coleções, leitura/escrita de arquivos etc. Também foram utilizados outros recursos como *Maven* e testes unitários com *JUnit*.

Acredita-se que este relatório, em conjunto com a documentação e o código do sistema, cumpre em elucidar as escolhas, implementações e peculiaridades do projeto e que o produto final entregue cumpre com os requisitos definidos no enunciado.