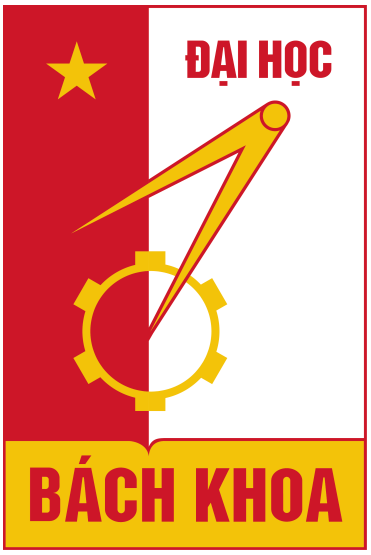
December 26, 2024



HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

Object-oriented Programming Mini-Project

Student :
Pham Trung Kien
Pham Khanh Chi
Nguyen Luong Uy
Doan Phuong Khang
Nguyen Minh Duc
Class ID : 152626

Instructor:
Tran The Hung

Contents

1 Project Description

1.1 Project Overview

Users can draw their UML diagram in draw.io, then they can download in drawio formal file, which is converted into XML format in the program. After several decoding stages, the program gets information and generates actual Java code (class files including attributes, methods, and class relationships).

1.2 Project Requirements

- **Libraries**
  - JavaFX library
  - json library
  - jdbc postgresql
  - FastAPI
  - Google Generative AI
- **Techstack**
  - PostgreSQL
  - CSS
  - Java

1.3 Usecase Diagram

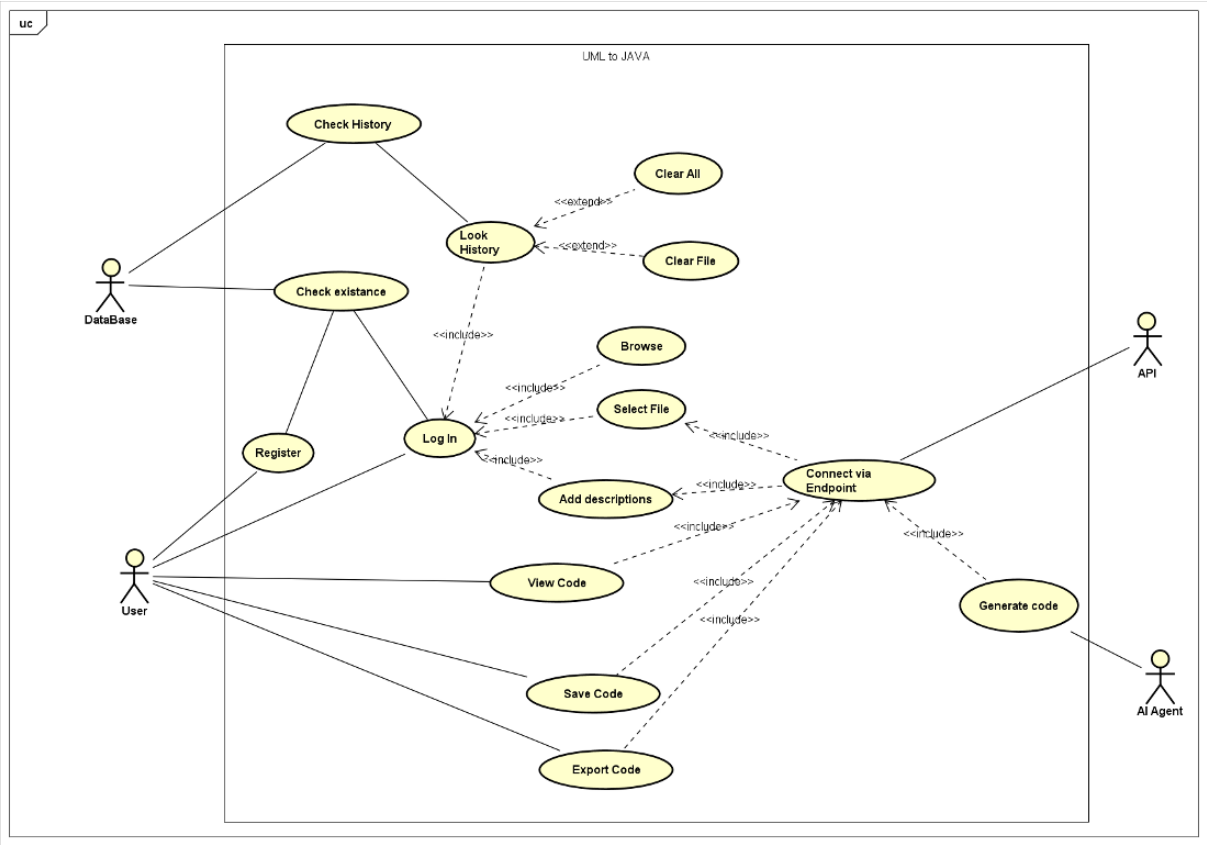


Figure 1: Usecase Diagram

2 Design and Details

2.1 Class Diagram

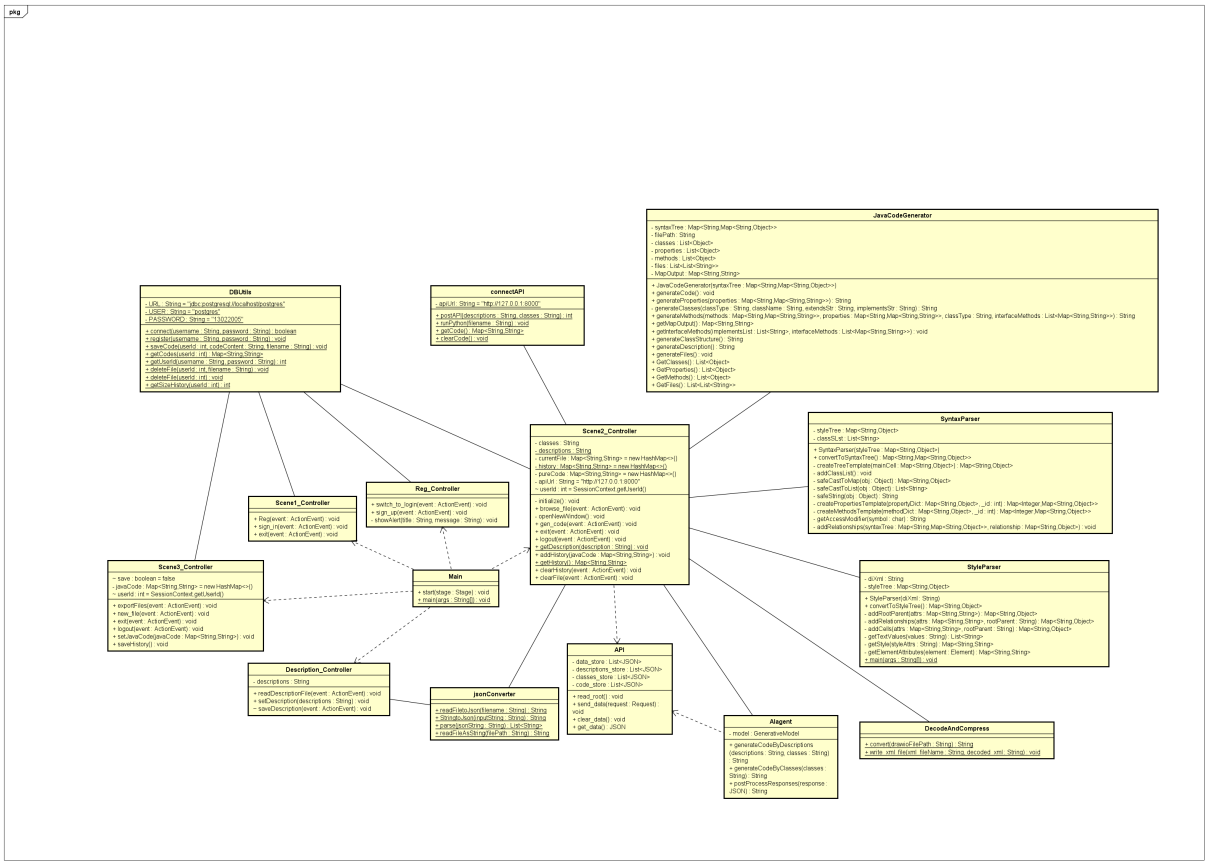


Figure 2: Class Diagram

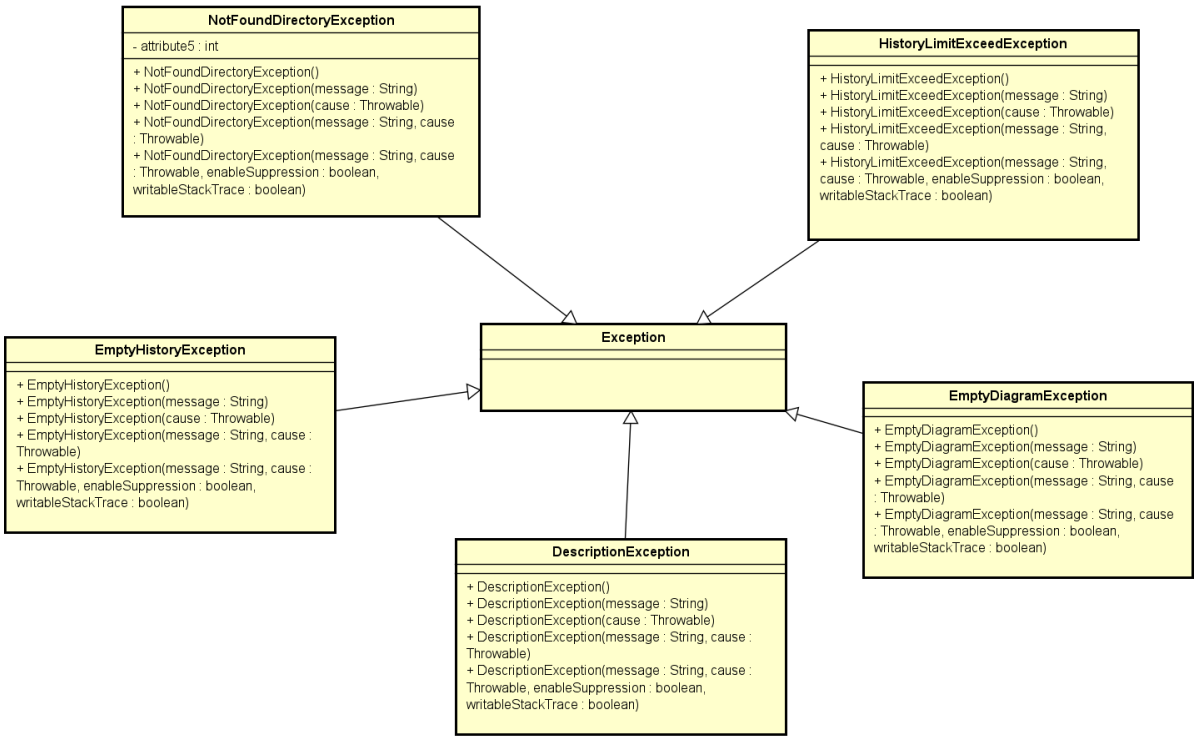


Figure 3: Exception

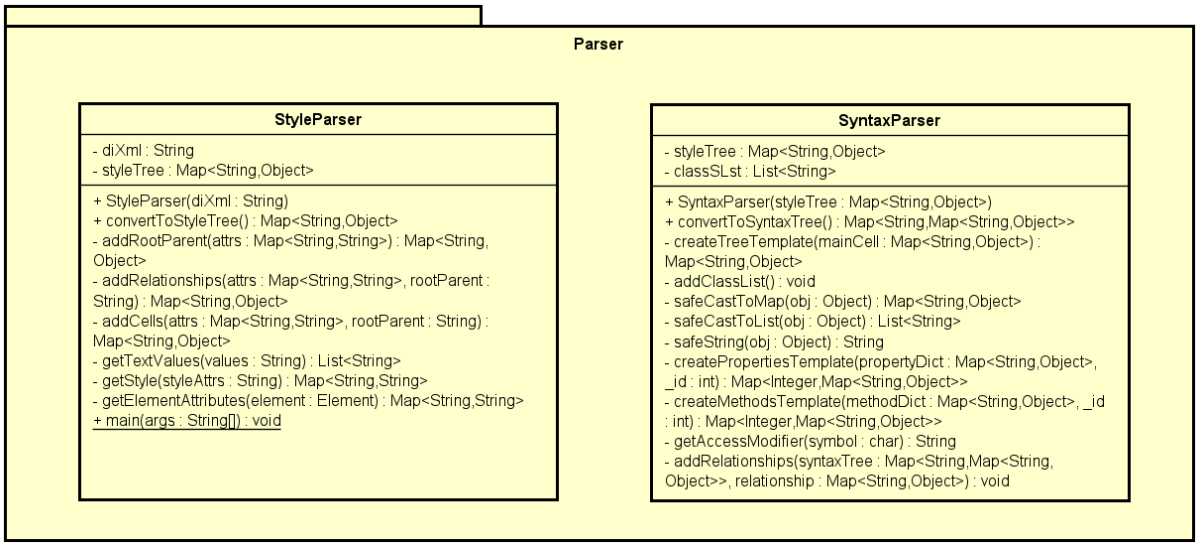


Figure 4: ParserPackage

2.2 Packages

2.2.1 Decode

Here we defines a class 'DecodeAndCompress' that provides two main functionalities:

- Decoding and decompressing the content of a .drawio file to extract the raw XML using method convert

- Saving the decompressed XML content to a .xml file using method write_xml_file
- **Input:** drawioFilePath (String): Path to the input drawio file that needs to be decoded and decompressed.
- **Output:** Returns the decompressed and decoded XML string (UTF-8 format). If an error occurs during any step (file reading, Base64 decoding, decompression), it returns null.
- **Attributes:** This class does not have instance variables (attributes) as it relies purely on methods that operate on passed arguments.
- **Method 1: convert(String drawioFilePath)**
  - **Input:**
    - drawioFilePath: The path to the drawio file containing the <diagram> tag to be decoded.
  - **Output:**
    - Returns the decoded and decompressed XML string (in UTF-8 format) if successful.
    - Returns null if an error occurs during the process (file parsing, Base64 decoding, decompression, etc.).
  - **Attributes/Local Variables:**
    - content: Holds the content of the file after reading.
    - document: A Document object parsed from the HTML content using Jsoup.
    - diagramTag: An Element object representing the <diagram> tag in the drawio file.
    - decodedBase64: A byte array that stores the Base64-decoded content of the <diagram> tag.
    - inflater: An Inflater object used to decompress the Base64-decoded content.
    - buffer: A byte array used to store data temporarily during decompression.
    - outputStream: A ByteArrayOutputStream used to collect the decompressed data.
- Method 2: write_xml_file(String xml_fileName, String decoded_xml)
  - **Input:**
    - xml_fileName: The name of the file (without extension) where the XML will be written.
    - decoded_xml: The XML string that was decoded and decompressed in the convert() method.
  - **Output:**
    - No return value. The method writes the decoded XML to a file.

    - **Attributes/Local Variables:**
        - None
    - **Description:**
        - This method takes the decoded XML string and writes it to a file with the specified name, appending .xml to the file name.
        - It uses BufferedWriter and FileWriter to handle file writing.

Application of Object-oriented Programming

- **Association:**

  The class has objects from external libraries (e.g., Inflater, Base64, BufferedWriter) as part of its

- **Composition:**

  BufferedWriter is composed of a FileWriter

- **Runtime Polymorphism:**

  Method overriding occurs when the methods of BufferedWriter are called — even though BufferedWriter extends Writer, it uses the FileWriter to perform the actual file writing.

2.2.2 Parser

1. **StyleParser** class is designed to parse DrawIO XML content and convert it into a structured **style tree**.

    - **Input:** A valid DrawIO XML string that has been decoded and decompressed.
    - **Output:** A hierarchical **style tree** represented as a Map.
    - **Main method:**
        - Parse XML:The input XML is parsed using Jsoup to extract the root and its child elements.
        - Process Elements:
            - Grandparent: Identifies the top-most node without a parent attribute.
            - Root Parent: Creates a dictionary entry in the style tree for the primary parent node (addRootParent).
            - Cells: Adds child elements with a parent attribute to the cells map (addCells).
            - Relationships: Defers elements with source and target attributes for later processing.
            - Process Relationships: Resolves the final source and target for relationships by mapping them to the root parent and adds them to the relationships map (addRelationships).
        - Output: Returns a nested Map representing the hierarchical structure of the XML content, with root, cells, and relationships.
    - **Attributes:**

  - diXml (String): Stores the input DrawIO XML (decoded and decompressed).
  - styleTree (Map<String, Object>): Stores the hierarchical representation (style tree) of the XML after parsing and processing, initialized as null in the constructor

- **Methods and Their Roles:** The methods are organized to provide modular and reusable functionality:
  - **Public Methods**
    * StyleParser(String diXml) (Constructor):
      Initializes the StyleParser object with diXml input and sets styleTree to null.
    * Map<String, Object> convertToStyleTree():
      Main method that converts the XML into a style tree representation. It encapsulates the workflow and directly interacts with the class attributes.
  - **Private Helper Methods**
    * addRootParent(Map<String, String> attrs): Creates a formatted dictionary for the root parent, adding default fields like cells and relationships.
    * addRelationships(Map<String, String> attrs, String rootParent): Processes relationship elements, ensuring source and target are mapped correctly to the root parent.
    * addCells(Map<String, String> attrs, String rootParent): Converts a cell element into a structured dictionary, handling attributes like id, parent_id, style, and values.
    * getTextValues(String values): Extracts and formats individual values from concatenated raw strings.
    * getStyle(String styleAttrs): Parses the style attribute of elements into a dictionary for easy manipulation.
    * getElementAttributes(Element element): Extracts all attributes of an XML element into a Map<String, String>.

```
"id": "WIyWlLk6GJQsqaUBKTNV-1",
"parent_id": "WIyWlLk6GJQsqaUBKTNV-0",
"cells": {
    "zkfFHV4jXpPFQw0GAbJ--0": {
        "id": "zkfFHV4jXpPFQw0GAbJ--0",
        "parent_id": "WIyWlLk6GJQsqaUBKTNV-1",
        "style": {
            "type": "swimlane",
            "fontStyle": "0",
            "align": "center",
            "verticalAlign": "top",
            "childLayout": "stackLayout",
            "horizontal": "1",
            "startSize": "26",
            "horizontalStack": "0",
            "resizeParent": "1",
            "resizeLast": "0",
            "collapsible": "1",
            "marginBottom": "0",
            "rounded": "0",
            "shadow": "0",
            "strokeWidth": "1"
        },
        "values": [
            "Person"
        ]
    },
```


Figure 5: Example of style tree

- **OOP application:** The StyleParser class **aggregates** helper objects like Map, List, and Element to build its functionality. These external objects are used to store and process data, but they are not tightly bound to the StyleParser class.

2. **Parse Syntax**: Converts the style tree into a syntax tree to understand the UML structure.

- **Input**

  - Style Tree: The primary input is a Map<String, Object> representing a styleTree that contains:
    * root: A map with cells and relationships.
    * cells: Objects representing nodes or elements in the style tree.
    * relationships: Objects representing connections between elements in the tree.

- **Output**
  - Syntax Tree: The result is a Map<String, Map<String, Object» that represents the parsed syntax tree. Each key is an element ID, and its value is a map containing:
    * Type: Indicates if the element is a class, abstract class, or interface.
    * Name: The name of the element.
    * Properties: A map of property details.
    * Methods: A map of method details.
    * Relationships: A map categorizing relationships (e.g., implements, extends, association, aggregation, composition).
- **Attributes**
  - styleTree: Input data structure for the style tree.
  - classSLst: A list of known class-like elements (e.g., "int", "String").
  - Local Variables in Methods:
    * _id: Tracks property/method counts.
    * propertiesDone: Boolean to separate properties and methods.
- **Methods**
  - convertToSyntaxTree:
    * Parses the styleTree to generate the syntax tree.
    * Handles relationships and properties of the elements.
  - createTreeTemplate: Builds a template for each cell in the syntax tree.
  - addClassList: Dynamically updates the list of classes (classSLst) from the styleTree.
  - createPropertiesTemplate: Maps property details from the style tree to the syntax tree.
  - createMethodsTemplate: Maps method details, including parsing parameters and their types.
  - getAccessModifier: Converts access modifiers (+, -, #) into readable formats (public, private, protected).
  - addRelationships: Maps relationships between elements based on connection styles.

```
"zkfFHV4jXpPFQw0GAbJ--6": {
    "type": "class",
    "name": "Student",
    "properties": {
        "1": {
            "access": "public",
            "name": "studentNumber",
            "type": "int"
        },
        "2": {
            "access": "public",
            "name": "avgMark",
            "type": "float"
        }
    },
    "methods": {
        "1": {
            "access": "public",
            "name": "isEigible(int studentNumber, double score)",
            "return_type": "boolean"
        },
        "2": {
            "access": "public",
            "name": "getName",
            "return_type": "Constructor"
        }
    },
    "relationships": {
        "implements": [],
        "extends": [
            "zkfFHV4jXpPFQw0GAbJ--0"
        ],
        "association": [],
        "aggregation": [],
        "composition": []
    }
},
```


Figure 6: Example of syntax tree

Application of Object-oriented Programming

- **Encapsulation:**

  All attributes and utility methods are private, ensuring modular functionality.

Access to styleTree is limited to the class itself.

- **Abstraction:**

  Methods abstract away the complexity of parsing and building syntax trees, focusing on specific tasks such as handling relationships or creating templates.

- **Composition:**

  Utilizes nested maps and lists to structure data and relationships effectively.

2.2.3 JavaGenerator

Creates Java source code files based on the parsed syntax.

- **Input** The class expects a syntaxTree map representing UML class diagrams.
- **Output** It generates Java code, including classes, methods, properties, and their relationships, based on the syntaxTree.
- **Attributes**
  - syntaxTree: Represents the UML diagram structure.
  - filePath: Stores the output file path. classes, properties, methods: Lists to track generated code elements.
  - files: Tracks file content per class. MapOutput: Stores the final class structure as a string.
- **Methods**
  - generateCode: Processes the syntax tree and creates Java class definitions.
  - generateProperties: Converts property definitions into Java attributes.
  - generateClasses: Constructs the Java class header (e.g., public class, abstract class, interface).
  - generateMethods: Generates methods, including stubs for interface methods.
  - getInterfaceMethods: Recursively gathers methods from implemented interfaces.
  - generateClassStructure: Outputs a JSON-like description of the classes.

Application of Object-oriented Programming

- **Ecapsulation**

  The JavaCodeGenerator class encapsulates the logic for generating Java code. The internal state (like syntaxTree, classes, properties, methods, files, and MapOutput) is managed and accessed through the class's methods. The use of private methods like generateClasses helps in hiding the internal implementation details. The generateProperties method also encapsulates the logic for creating property strings.

- **Abstraction**

  The JavaCodeGenerator class provides a high-level abstraction for generating Java code. The user interacts with the class by providing a syntax tree and calling the generateCode() method, without needing to know the complex details of how the code is generated. The syntaxTree itself is an abstraction of the UML diagram or other input source. The methods like generateMethods abstract the process of creating method signatures.

2.2.4 API package

The package facilitates the process of connection between Java code and Python code. Using POST and GET methods, as well as some utilities to process JSON-typed information gained, the package helps the different programming languages communicate well with each other via local hosted API.

- **Class overview:**
  - **API class:** Located in API package, it has the mission to create the API by using installed module FastAPI and handle incoming requests using the basic package Requests from Python.
    - **Attributes:**
      - data_store: belongs to the data structure of List in Python, used to store all the information gained during the code generation process.
      - description_store: belongs to the data structure of List in Python, used to store all descriptions taken from the user to pass to an AI agent later.
      - class_store: belongs to the data structure of List in Python, used to store all classes taken from the UML diagram as well as their attributes and methods, which would be passed to an AI agent later.
      - code_store: belongs to the data structure of List in Python, used to store all the codes generated by the AI agent, which would be brought to the user interface later.
    - **Methods:**
      - read_root: initialize the root for the API.
      - send_data: this method is used to send data such as descriptions, class structure, or generated code between Java files and Python files, via POST method, with different endpoints. The data gained is in JSON-format, and will be stored in the corresponding list.
      - get_data: this method is used to get all data stored in the list with the corresponding endpoints, via GET method.
      - clear_data: this method is used to empty all the data lists, using the POST method, so that the memory is refreshed after Java code has been generated.
- **connectAPI class:** located in API.utils package, it provides functions which would be utilized by other classes in order to perform connection between information from Java files and Python files using API.

  - **Attributes:**
    * apiUrl: a String represents basic URL of local hosted API
  - **Methods:**
    * postAPI: this method requires two Strings, consisting of descriptions and class structure, but represented in JSON-format in order to POST it to an endpoint, which would be obtained by a Python file.
    * runPython: this method is used to execute a Python file, provided its path.
    * getCode: Utilizing GET method to get the generated code from the endpoint "/get_code", then postprocess to convert that data from String to a Map<String, String> type.
    * clearCode: set a request of POST to the end point '/clear_data'.

- **jsonConverter class:** this class is used to bring about some utilities for processing normal String into JSON-format String, and vice versa.
  - **Methods:**
    * StringtoJson: take an input of normal String, which is the descriptions from user, and convert it to JSON-format String, which can be used for POST methods.
    * parse: take an input of JSON-format String, extract the needed information from it to a new List of Strings.
    * readFileAsString: concatenate all lines from a file.
    * readFiletoJson: return the text from a file in JSON format.

Application of Object-oriented Programming

- **Encapsulation:**

  The data in API class is encapsulated, and can only be accessed through API requests.

- **Abstraction:**

  The abstraction principle hides the complex implementation details while providing a simple interface to the user. The routes @app.post(), @app.get() abstract the complexity of request handling, allowing the user to simply send requests without worrying about the underlying logic.The request.json() method abstracts the process of extracting JSON data from the HTTP request, making it easier for developers to work with the incoming data.

- **Polymorphism:**

  Inexplicitly use method overloading in methods sending POST and GET requests. Although the signatures are similar to each other, the endpoints are completely different, leading to different results.

2.2.5 genAI package

The package contains only a class AI Agent, which helps to generate the executable code according to the descriptions from users.

- **Class overview:**
  - **AIgenerator:** located in the genAI package, it is used to generate Java code using the information gained from API.
    - **Attributes:**
      - model: the LLM used to generate code based on descriptions and class structure. Here, we will make requests to Gemini API, and choose the latest model "gemini-2.0-flash-exp".
    - **Methods:**
      - generateCodeByDescriptions: Taking two Python dictionary, which represent class structures and the corresponding descriptions, to make a prompt and call the Gemini API to get the response Java code.
      - generateCodeByClasses: Only need the class structures, without descriptions, and make a prompt for Gemini model to get the desired response.
      - postProcessResponse: convert the text response from the model to different Java files and their corresponding code.

2.2.6 GUI

This package provides users with the GUI where they can interact with the system. This package contains fxml files for the graphic and java files for the controller, with css files in addition for beautifying the interface.

- **Class Overview:**
  - **Main:** class in the main package is the main class of the application. It extends the Application class provided by JavaFX and contains the main entry point and lifecycle methods of the application.
    - **Attributes:**
      - stage: window displaying the application.
      - scene: container holding all graphical elements.
    - **Methods:**
      - start: The overridden start method from the Application class. It is called when the application is launched. This method sets up the main scene, loads initial data, configures the stage, and displays the UI.
  - **Reg_Controller:**
    - **Attributes:**
      - stage: window displaying the application.
      - scene: container holding all graphical elements.
    - **Methods:**

- switch_to_login: cancel sign up, return to sign in screen
- sign_up: submit information to complete signing up
- success: provides feedback to the user upon successful completion of the sign-up process.

– **Scene1_Controller:**

* **Attributes:**
  - stage: window displaying the application.
  - scene: container holding all graphical elements.
  - scenePane: identity of the Anchor Pane in fxml file.
* **Methods:**
  - Reg: move to register screen
  - sign_in: method for sign in button, then move to next screen
  - exit: exit the program

– **Scene2_Controller:**

* **Attributes:**
  - browse: identity of button to browse file in fxml file.
  - scenePane: identity of the Anchor Pane in fxml file.
  - btnGenCode: identity of button to generate code in fxml file.
  - codeText: identity of text area for code preview in fxml file.
  - descriptionBrowse: identity of button to open the description window in fxml file.
  - btnSelectFile: identity of dropdown menu to view generated files in fxml file.
  - historyOption: identity of dropdown menu to view saved generated files in fxml file.
  - menuClear: identity of menu to clear all history in fxml file.
  - btnClear: identity of button to clear the chosen saved java file in fxml file.
  - stage: window displaying the application.
  - scene: container holding all graphical elements.
  - classes: a JSON-format String represent all classes with their attributes and methods.
  - Descriptions: a JSON-format String represent the description for each method.
  - currentFile: a map containing information about the files whose code is shown on the text field.
  - History: a map containing all generated code filenames and contents,
  - pureCode: a map representing all generated code from class diagram (just attributes and empty methods)
  - apiUrl: a String showing the root of API
* **Methods:**

- browse_file: a method to choose the path to the class diagram (.drawio files), then generate code, get descriptions and class structures.
- openNewWindow: open a new window to get descriptions from users
- gen_code: passing descriptions and classes to API.connectAPI.postAPI utility, then get the response from the AI Agent, propagating to the next scene if successful.
- exit: choose whether to exit the current app or not
- logout: choose whether to logout from the current account or not.
- getDescription: a getter function to get descriptions
- getHistory: a getter function to get history
- clearHistory: clear all items from the history
- clearFile: clear the chosen file from history

– **Scene3_Controller:**

* **Attributes:**
  - stage: window displaying the application.
  - scene: container holding all graphical elements.
  - scenePane: identity of the Anchor Pane in fxml file.
  - fileSelector: identity of the dropdown menu to select generated files in fxml file.
  - codeField: identity of the Text Field in fxml file.
  - btnExport: identity of the the button to export files in fxml file.
  - btnSaveHistory: identity of the button to save all generated code to history in fxml file.
  - save: boolean value to choose whether the generated code be saved or not.
  - javaCode: a map containing information about filenames and their corresponding code.
* Methods
  - exportFiles: this method chooses the desired directory and save all files to local machine.
  - newFile: move back to the screen 2.
  - exit: choose whether to exit the current app or not
  - logout: choose whether to logout from the current account or not.
  - setJavaCode: a setter method to set attribute javaCode
  - saveHistory: save generated code to the history if the number of files saved in the history do not exceed the limitation.

Application of Object-oriented Programming

- **Abstraction:**

  The classes and methods are designed to hide implementation details and only expose relevant functionalities, such as Reg, sign_up, exit, exportFiles, ...

- **Encapsulation:**

  - Private fields and public methods Many private fields (e.g., stage, scene, javaCode) are used alongside public methods to control access and maintain the integrity of the objects.
  - Use of @FXML annotations for encapsulated UI components (e.g., scenePane, fileSelector) limits direct access to them.
  - Static fields: some methods such as descriptions and history are static, encapsulating shared information across instances while restricting direct access.

- **Inheritance:**

  The Main class inherits from the Application class (a JavaFX base class) to leverage its pre-defined behavior for JavaFX applications.

- **Polymorphism:**

  Method overriding: The start method in the Main class overrides the method from the Application class.

- **Association:**

  The class heavily relies on association by using external classes:

  - API utilities: connectAPI, jsonConverter.
  - Decode: DecodeAndCompress
  - Parsers: StyleParser, SyntaxParser.
  - Generators: JavaCodeGenerator.

2.3 Exceptions handling

- **DescriptionException:**

  - In cases when the description text from the user is different from the class structure in terms of methods, it is really hard for the AI Agent to understand the desire of the user, so it may generate the code not similar to the expectation of the user. Therefore, to guarantee that the code has a high quality, we define these cases as exceptions, and notify the user so that they can reformat their input.
  - Sometimes, the user may not have any further description for the diagram, this is also a hard situation for the AI Agent to take into account their wish, so we also label these circumstances as exceptions.

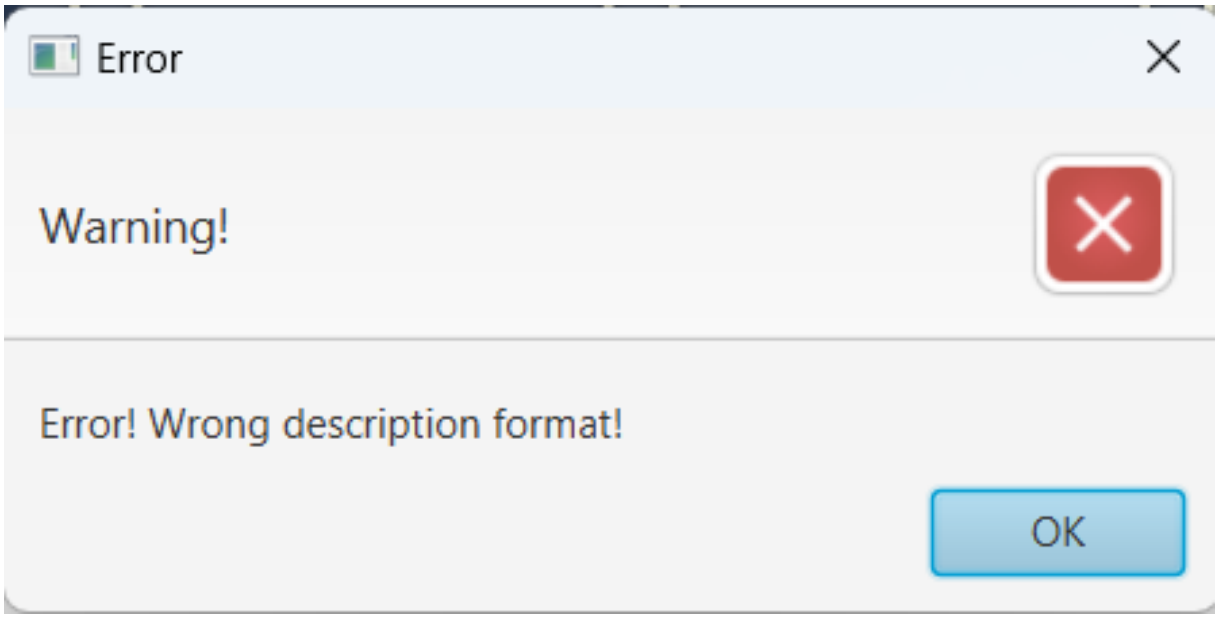


Figure 7: Example of syntax tree

- **EmptyDiagramException:**
  - When the user browse their files, some situations may occur:
    - The user may not choose any files, so no diagram is found.
    - The user may choose a wrong .drawio file, which has no classes and relationships.
  - Therefore, we handle these circumstances by making a notification of error pop up in a new window.

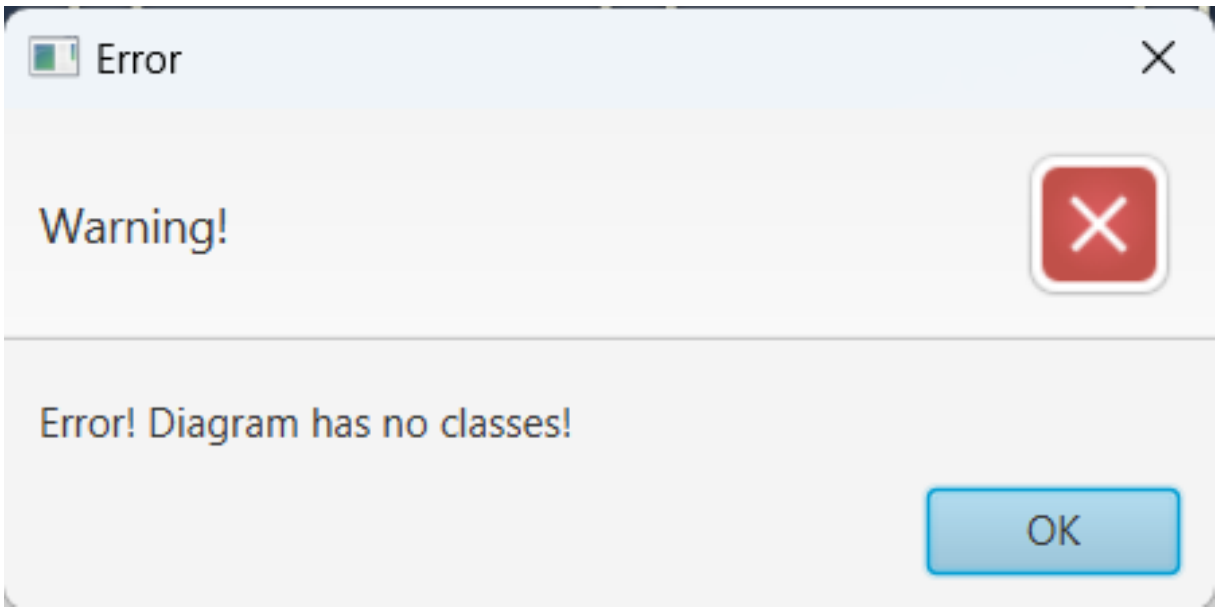


Figure 8: Example of syntax tree

- **NotFoundDirectoryException:** when no description file is chosen, a warning will appear on the screen.

Figure 9: Example of syntax tree

- **EmptyHistoryException:** occurs when the user tries to clear the history whereas nothing has been stored in the history or it has just been cleared.

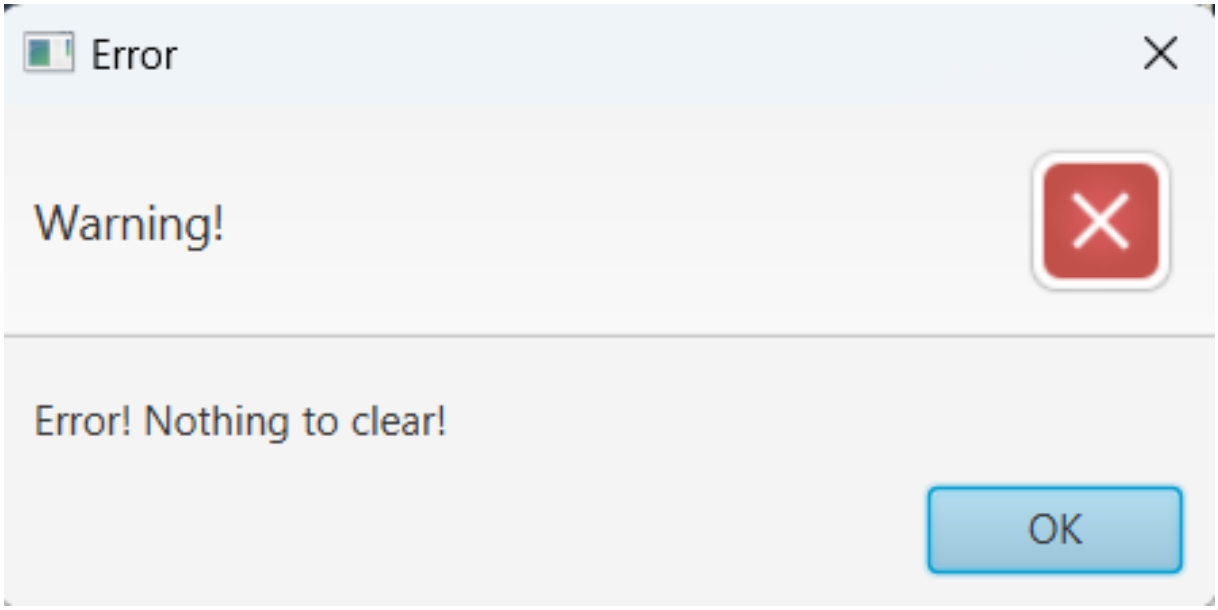


Figure 10: EmptyHistoryException

- **HistoryLimitExceedException:** the database for users has a limitation, which can be increased in case they upgrade their account to Pro. So when the number of saved Java code files exceeds this limit, we will notify the user, and recommend them to upgrade their account.

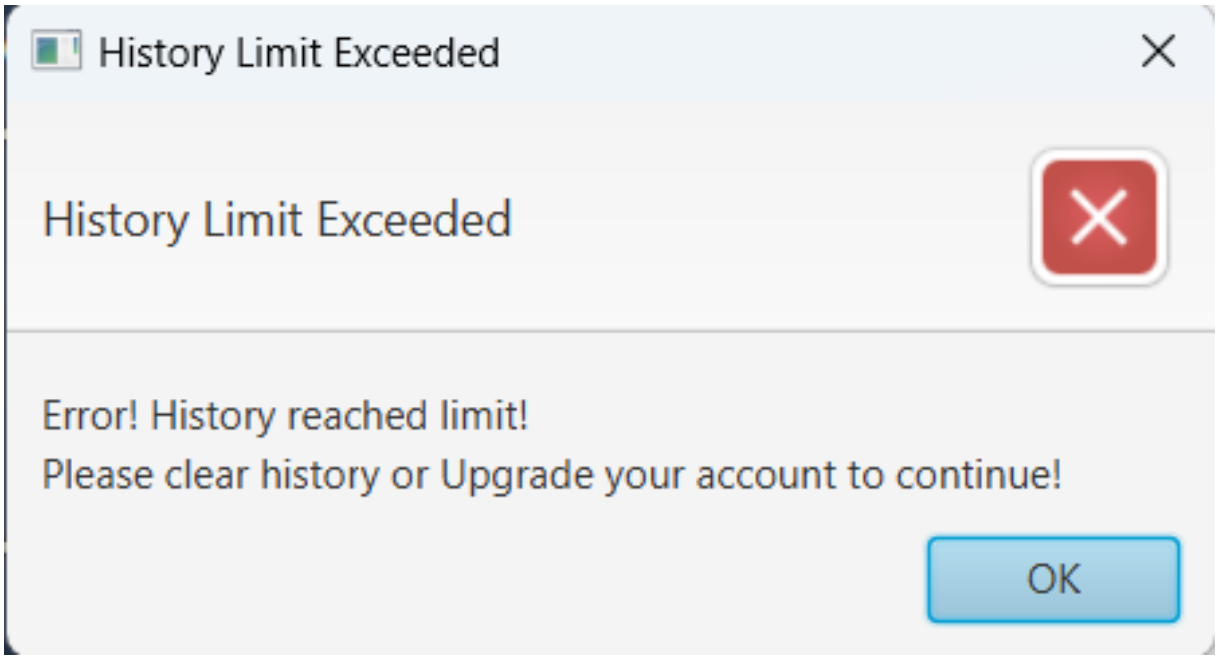


Figure 11: HistoryLimitExceedException

- Besides, we also handle other exceptions using default **Java Exceptions**.
  - During the process in parsers, when no class is found in the diagram, or user design a wrongly formatted class diagram. So there will be a NullPointerException, which would be thrown to the GUI controller and caught, after that a notification will let the user know their diagram is invalid.
  - StringIndexOutOfBoundException is also used besides the our own defined EmptyDiagramException to handle cases where the diagram is blank.