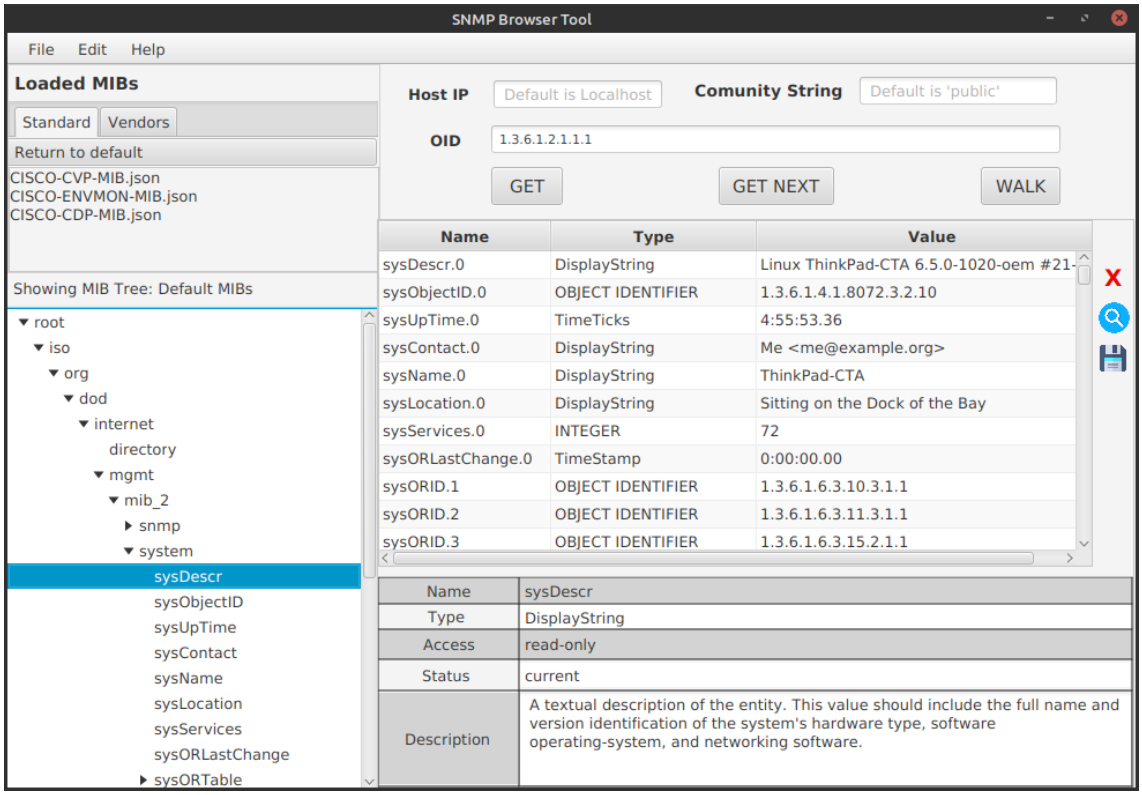
0.1 Overview

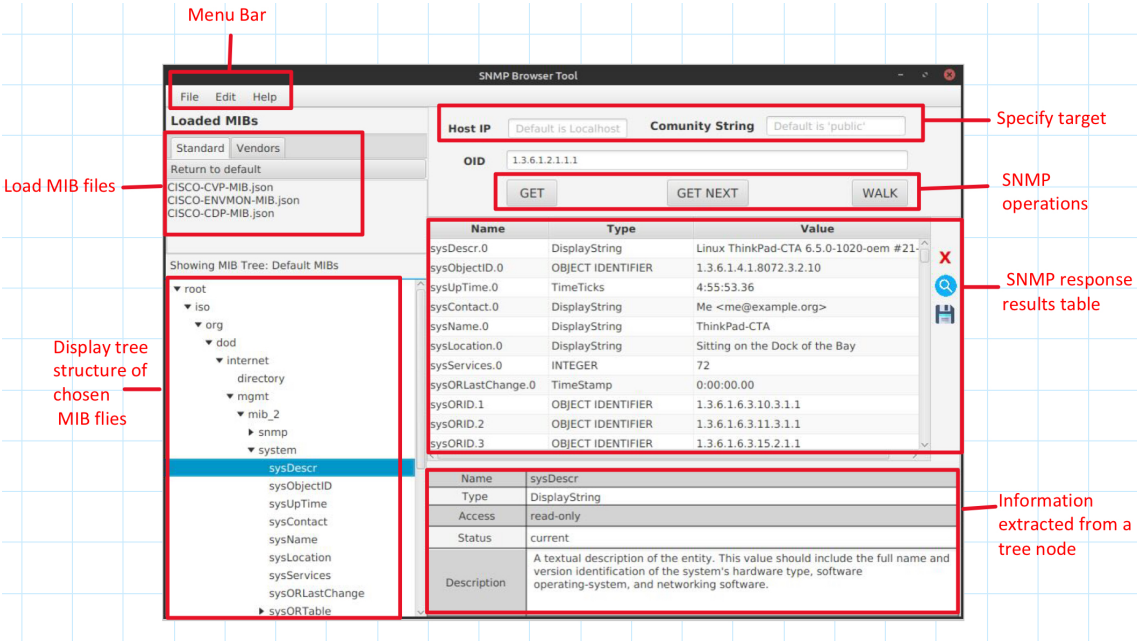
We have developed an SNMP browser application using Java and the JavaFX framework for the user interface. Our app currently supports SNMP Get, GetNext, and Walk operations using the snmp4j library, working on SNMPv2c. It reads MIB files in JSON format to display the MIB tree structure. Additionally, these files help extract relevant information from provided OIDs to format raw SNMP responses. The primary goal of this app is to demonstrate how SNMP can be used to collect data from server and network devices, effectively managing them. The app also supports Dark Mode that is made through a CSS file. Right now, this feature is still being worked on, and the CSS file will be tweaked further to make improve the app appearance.

Here's a first look at what the app looks like:



Hình 0.1: App Demo

0.2 App Features

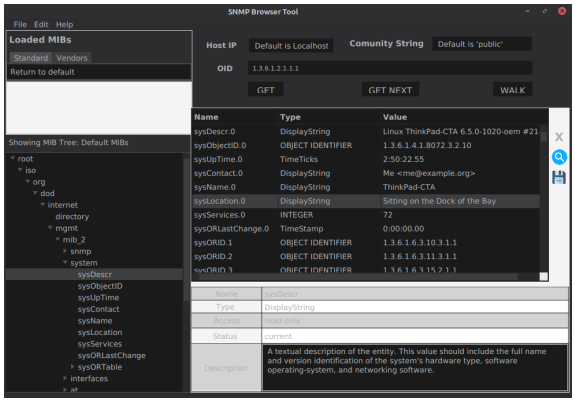


Hình 0.2: App UI breaks into parts

Below, we highlight some key features of the application:

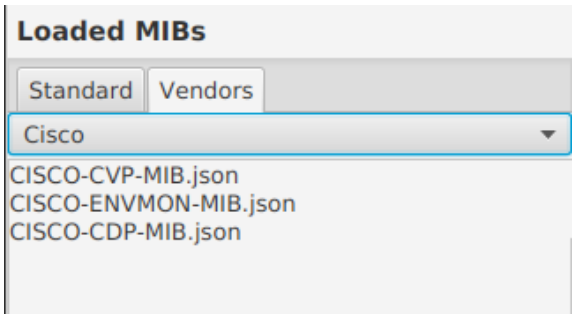
- **Menu bar:** there are three items in the menu bar:
  - File: it contains three items:
  - ∗ Open MIB: let the user select a MIB file from their system to load into the program, which will appear in the MIB Load section.
    - ∗ Import MIB: Similar to Open MIB, but it also saves a copy of the MIB file to the application's MIB Databases directory.
    - ∗ Unload all MIBs: Clears the MIB Load section below.

    In the MIB Database directory, approximately 70 MIBs are currently available in JSON format. These have been sourced from the MIB databases of the iReasoning MIB browser software and subsequently converted to JSON format.
  - Edit: Allows toggling between light mode and dark mode. Note that the dark mode is currently under development as we are still fine-tuning the CSS file.



– Help: Provides information about the app, including contributors and acknowledgments.

- **Load MIBs Panel**: There are two tabs
  - Standard: This option includes a ‘Reset to Default’ button that restores the MIB tree to its original state, just like when the app is first launched. The default MIB tree is composed of the following files: **SNMPv2-SMI.json**, **RFC1213-MIB.json**, **HOST-RESOURCES-MIB.json**, **SNMPv2-MIB.json**, and **IF-MIB.json**.
  - Vendor: This allows the user to select a vendor from a drop-down menu. Each vendor is associated with a predefined list of available MIBs.

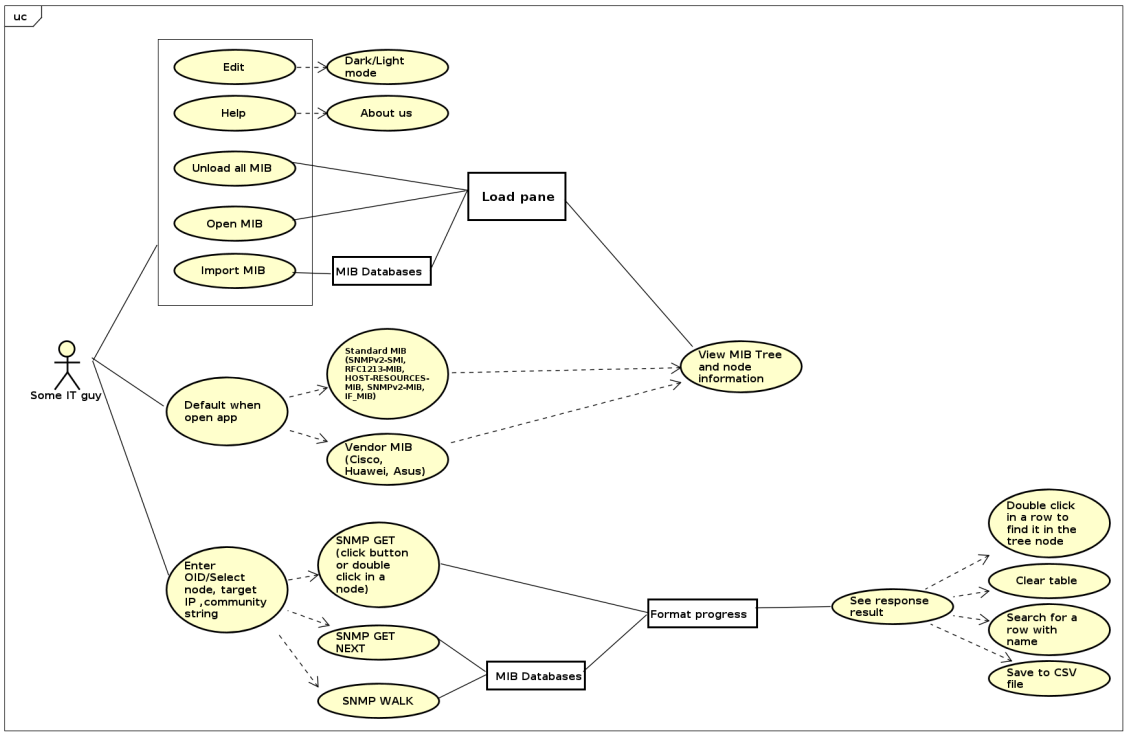


All MIBs that are opened, imported, or selected from a vendor will be loaded into the application and displayed in their file name in the below the flow pane section.

- **MIBTreeDisplay pane:** When user click on a file name in the MIB Load pane above, the tree structure of that file will be displayed. This tree is created using the TreeView object in JavaFX. A single click on any node in the tree will reveal that node's details in the bottom right pane of the app. Although each node contains numerous fields, we only show the most important ones in the UI: **Name**, **Type**, **Access**, **Status**, **Description**, and **OID**. Double-clicking on a node will execute an SNMP Get request for that selected node.
- **Host IP, Community String**: Enter the IP address of the target device and the community string to authenticate the communication. The community string is entered into a Password Field in JavaFX. If left blank, the default IP address is set to localhost (127.0.0.1) and the community string defaults to "public". note that currently our application only support SNMP Get, Get Next and Walk operation, so the port used is fixed to UDP 161.
- **Get, Get Next, Walk button**:The functions of these buttons are indicated by their names. They use the current IP address, community string, and OID displayed in the UI to perform their actions.
  - Get: retrieves the value of the passed in OID. Double-click on a tree node can perform similar action.
  - Get Next: retrieve the value of the OID that comes after the one we pass in. If users select a node and then click Get Next several times, it will sequentially retrieve the values of subsequent OIDs (incremental from the first OID). The OID currently being used will only be reset if you click on a different node in the TreeView or if you press the Clear Table button.
  - Walk: retrieve values by exploring the subtree starting from the entered OID. Using Walk on a large and complex MIB tree can be demanding on resources. In our tests, starting from node 1.3.6.1 with around 70 MIB files in the database, the Walk operation took approximately 1.5 seconds to complete. Currently, Walk searches for OIDs in MIBs by comparing the entered OID with the root index of a MIB file, and moves to the next file if there's no match. We plan to implement more efficient techniques in the future to speed up execution.
- **Result Table:** Display the SNMP response, which includes the **name**, **data type**, and **formatted values**. If the information required to format an OID is unavailable, append "(raw response)" after the value. Double-clicking on a row will highlight it and navigate to the corresponding node in the MibTreeDisplay panel. On the right side, there is a toolbox panel that allows users to perform

several actions: clear the results table, prompt for a name to locate a specific row, and export all results in the table to a CSV file. Hovering over any tool will display a tooltip describing its usage.

Here is the summary of all important features in the User case diagram:



Hình 0.3: User Case Diagram

0.3 Code Structure of Application

We designed our code following the Maven standard structure and the Model-View-Controller (MVC) pattern. Here is the main structure and their short purpose:

```
Project_I_Collect_SNMP_Data/
  └── src/
    ├── main/
    │   ├── java/
    │   │   ├── Control/
    │   │   │   ├── MainController.java
    │   │   │   └── ARowInQueryTable.java  # Used to define a row structure in the query table
    │   │   └── Model/
    │   │       ├── MIBTreeStructure/
    │   │       │   ├── Node.java  # Used to define a node in the MIB tree
    │   │       │   ├── BuildMIBTree.java  # Used to build the MIB tree from the JSON file
    │   │       │   ├── MibRootOidFinder.java  # Used to find the root OID of a MIB file
    │   │       │   └── MibLoader.java  # Used to load MIB files for GET NEXT and Walk
    │   │       └── SNMPRequest/
    │   │           ├── SNMPGet.java
    │   │           ├── SNMPGetNext.java
    │   │           ├── SNMPWalk.java
    │   │           └── SnmpResponseFormat.java  # Used to format the raw data to a more readable format
    │   └── resources/
    │       ├── Images/
    │       ├── styles.css # Use to render the Dark Mode
    │       └── View/  # Contains the FXML files for the GUI
    │
    │
    └── test/java/TestFiles  # Contains test files for individual functions of the project
```

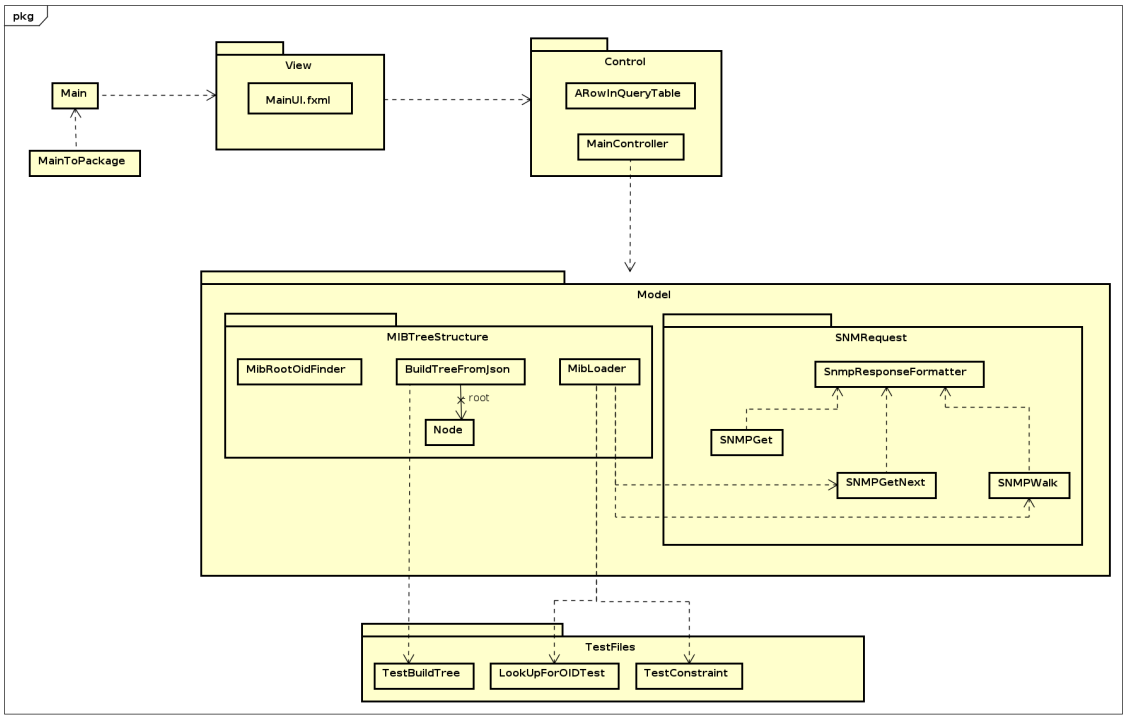
Hình 0.4: Code structure

The project also incorporates the following components:

•

- **MIB Databases** directory: contains all the MIB files that the application utilizes to gather information. There are two instances of this directory in the project: the first is located within **Project**$_{I}collect_{S}NMP_{D}ata/out/artifacts/SNMP_{B}rowser/MIBDatabasesandisus$

Here is the general class diagram

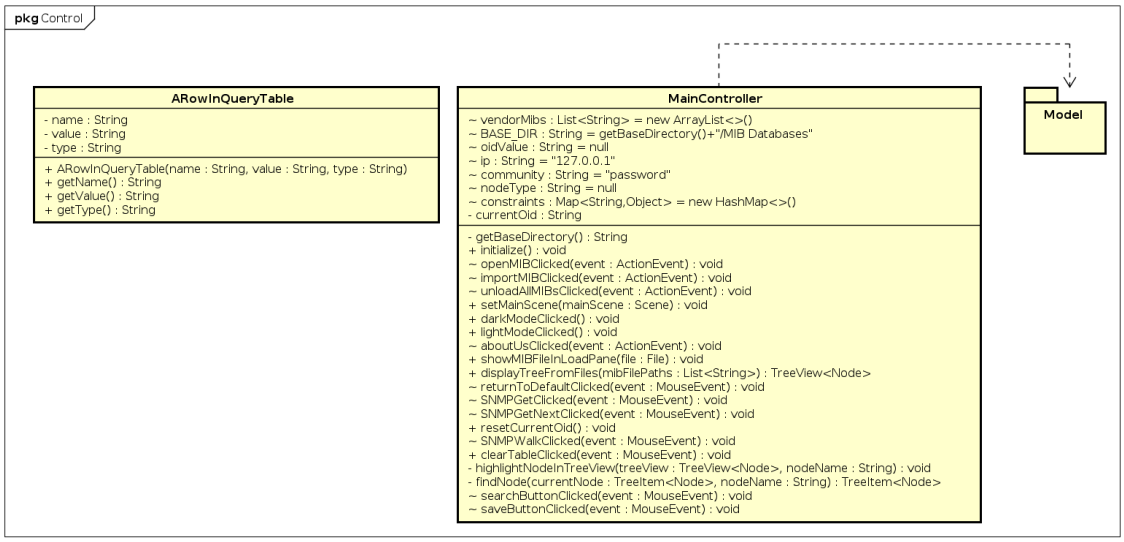


Hình 0.5: General Class Diagram

The `Main` class is responsible for launching the main window (JavaFX Stage) and must therefore extend the `Application` class. However, when packaging the application, it is often recommended often the main entry point class of the artifact does not have any external dependencies (like extending another class). Therefore, we use `MainToPackage` class, which just simply call the `main` method from `Main` class, as the main class when making the Artifact.

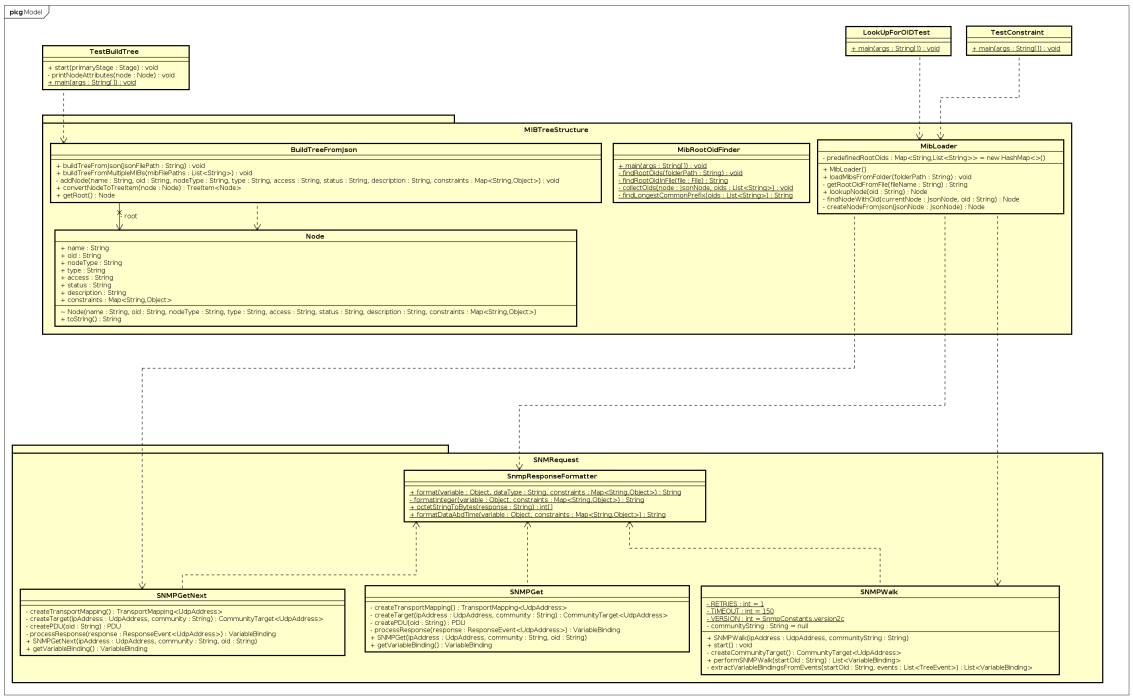
Two of the most important packages in the project are:

- **Control:** contains two classes: **ARowInQueryTable**, which defines the structure of a row in the display table, and **MainController**, which is responsible for managing all events from **MainUI.fxml**. The MainController class monitors user actions and interacts with the Model. Below is a detailed class diagram of the Control package:



Hình 0.6: Cotrol Class Diagram

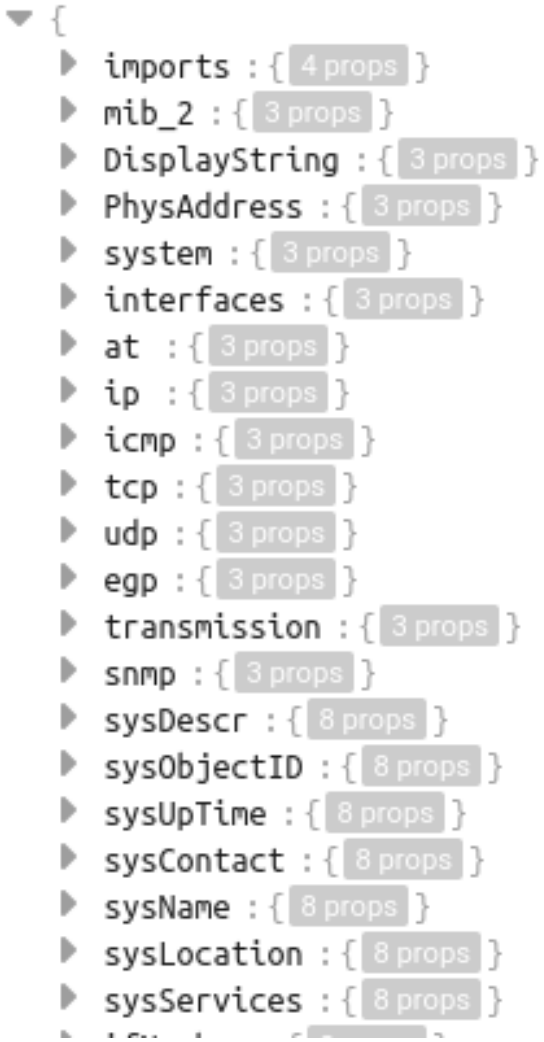
- **Model:** contains two sub packages **MIBTreeStructure** and **SNMPRequest**:



Hình 0.7: Model Class Diagram

– The **MIBTreeStructure** class responsible for all action related to MIB files.

  * Its primary function is to construct the tree structure from the provided MIBs. Initially, the JSON files, converted by pysmi, are flat, as shown below:



Hình 0.8: The sysDescr, sysObjectID are on the same level with the system instead of being sub objects inside it

To establish the tree structure, we utilize the OID path of each node. By parsing the JSON file, we create the tree with an entity stored as a Node, as defined in the Node class. Each Node possesses the following attributes:

```
public String name;
public String oid;
public String nodeType;
public String type;
public String access;
public String status;
public String description;
public Map<String, Object> constraints;
//Constraints, include the name of the constraints
and the actual constraints, since we do not know in
```

advance the type of the constraints, we use the Object type to store the constraints object.

```
Map<String, Node> children = new HashMap<>();
```

Leveraging the OID path for construction allows this class to merge multiple JSON files into a single, comprehensive MIB tree. To verify the successful build and display of the tree, use the **TestBuildTree** class within the `test` package.

* The second responsibility is to locate information for a given OID across all MIB files. Upon invocation, it loads MIB files into memory and searches for the node with the matching index, returning it if found. Currently, we compare whether the given OID starts with the same sequence as the root OID of a MIB file, continuing the search if it does, or skipping the file if it does not. Identifying the root OID of each file is solving the Longest Common Sequence problem (we find the longest OID path between all OIDs in a JSON file), which is accomplished using the **MibRootOidFinder** class. This class is called once to retrieve the root OID information and does not run each time the application is executed. All these responsibility is handle by `MibLoader` class. To check with a given OID, this class can correctly find the node with matching OID, use the **LookUpForOIDTest** in the `test` package.

– The **SNMPRequest**: component handles the retrieval and formatting of SNMP data. It consists of the following key elements:

* SNMPResponseFormatter: Take in the raw response from SNMP operations, together with data type and constraint to convert to human-readable format. Most of SNMP response result have been already in human-readable format already, we just handle some type such as: DateAndTime, Enumeration of Integer, ... To check with a given response with enumeration constraint, this class can correctly handle, use **TestConstraint** in `test` package.

* SNMP Get: Upon user selection of a Node, which provides the OID, name, data type, and constraints, an SNMP Get operation is performed using that OID. The raw response, along with the data type and constraints, is then processed by SNMPResponseFormatter to produce the final result for display in the table. For objects with a node type of ‘scalar’,

".0" is appended to the OID; for other types, ".1", ".2", etc., are tried until a 'noSuchObject' response is encountered. This is a temporary implementation and may require refinement later.

* SNMP Get Next: When a Node is selected by the user, the OID immediately following the selected one is retrieved. As the information about this new OID is unknown, MibLoader is used to look up its details, which are then passed to SNMPResponseFormatter. Note that with the OID we must remove the post fix (".someNumber ") at the end of OID so that it exists in the MIB files.
* SNMP Walk: retrieves a list of OIDs and their corresponding values, starting from the selected OID and continuing all the way down the subtree. Each retrieved OID is stripped of its post-fix, then searched using MibLoader before being processed by SNMPResponseFormatter.

0.4 Installation and Setup

The application has been packaged as a ZIP file containing all necessary JAR file and dependencies. This means it can run on any operating system with a Java Virtual Machine (JVM) installed. **Java version 21 or newer is required; older versions will not be able to launch the application.**

To get started:

1. Visit the Releases page on the project's GitHub repository:

   https://github.com/chutrunganh/Project-I-Collect-SNMP-Data/releases/

2. Download the latest release, a ZIP file named `SNMP_Browser`.
3. Unzip the file. You'll find a JAR file and a "MIB Databases" directory inside.

```
SNMP_Browser/
    SNMP Brower.jar
    MIB Databases
```

4. Double-click the JAR file to launch the application.

0.5 Future Improvement

The SNMP application we developed successfully demonstrates the use of SNMP for collecting data from servers and network devices, enabling effective management. However, there is still a lot of work to improvement in terms of user experience, performance, and functionality.

Current Issues and Solution:

- **Deployment:** Although the application is implemented using Java, with the slogan "Write Once Run Anywhere," we provided a JAR file that users can simply double-click to run the app. However, it requires a Java Virtual Machine (JVM) with a compatible Java version to be installed first. Therefore, we plan to deploy the application to platform-specific installers (e.g., `.exe`, `.dmg`, `.deb`) for better optimization and system integration, or provide a Docker image to simplify installation.
- **Apply machine learning algorithms and statistical techniques** to automatically retrieve device data, analyze it, and predict potential failures or misbehavior. .
- **UI/UX:** Some UI elements need refinement, and the Dark Mode implementation can be enhanced. Solution:
  - Address visual inconsistencies and improve the overall look and feel of the application
  - Ensure proper color contrast and readability in Dark Mode.
- **MIB Database:** The database lacks comprehensive MIB coverage, and the search process could be more efficient. Solution:
  - Add more MIB modules to the database for better device support
  - Explore the possibility of integrating with online MIB repositories or allowing community contributions to expand the database.
  - Implement more efficient search algorithms (e.g., indexing, filtering) to improve the speed and accuracy of MIB lookup.
- **Data Handling:** Additional data types and constraints need to be supported for broader compatibility.
- **Limited Functionality:** The application only implements basic SNMP Get, GetNext, and Walk operations. Solution:
  - Implement SNMP GetBulk operation for retrieving large amounts of data efficiently
  - Set for modifying device configurations
  - Trap handling to enable the application to receive and process SNMP traps
  - Inform to provide a more reliable trap notifications with acknowledgements
- **Security:** SNMPv3 support is missing, leaving the communication vulnerable. Also implement user roles and permissions to restrict access to sensitive SNMP

operations.

- **Error Handling:** Improve error handling and logging to provide more informative feedback to users and administrators.
- **Performance Optimization:** Profile and optimize code to reduce resource usage and improve response times.
- **Testing:** Develop a comprehensive testing strategy (unit tests, integration tests, etc.) to ensure the stability and reliability of the enhanced application.
- **Automated Quality Assurance:** Utilize Jenkins for continuous integration and deployment (CI/CD) and Integrate SonarQube for continuous code quality analysis.