Maintenance guide:

The following document is a guide for developers who wish to maintain the system. We go through the main components of the system and dive into the code.

Our system was developed with Java 19 using the JavaFX framework for the GUI.

In addition to that, we also provide a detailed walk through of how to add some new functionality to the system in some specific use-cases.

Table of contents:

Facade:

- The facade module is implemented using the Facade DP - Reference (https://refactoring.guru/design-patterns/facade). Basically, this is the gateway to the backend of our system. All of its methods are organized and well documented in a java doc format here: Link (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/FinalProject/src/main/java/backend/finalproject/IAOSFacade.java).

- Each function returns a value wrapped by a Response (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/FinalProject/src/main/java/utils/Response.java) object. The Response object would hold the outcome of the function if no error has occurred, by using the following method:

```
response.getValue();
```

Otherwise, if an error has occured, the Response object would hold the error message, by using the following method:

```
response.getMessage();
```

- Best practice is to use the following pattern when calling a function from the facade:

```
// someFunction is a function that returns a Response with a generic type T wrapped inside it
Response<T> response = facade.someFunction();
if(response.hasErrorOccured()) {
    // handle error
    // ...
}
else{
    T value = response.getValue();
    // handle whatever you want to do with the response value
        }
```

Integration requests:

The integration requests module is implemented using the Visitor DP - Reference (https://refactoring.guru/design-patterns/visitor). The main classes taking part in this module are:

- **HttpRequest** (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/FinalProject/src/main/java/backend/finalproject/IntegrationRequests/HttpRequest.java) - An abstract class that contains the server info, some common headers, etc. It's recommended to extend this function for each of your new integration requests to avoid code duplication.
- **HttpRequestDTO** (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/FinalProject/src/main/java/DTO/HttpRequests/HttpRequestDTO.java) - The DTO interface representing the HTTP request. Acts as the visited class in the visitor DP. Any integration request must implement that class (more on that later). The interface contains one and only function:

```
String visit(IntegrationRequestsHandler handler);
```

The function is used to visit the visitor class (IntegrationRequestsHandler) and return the json response as a plain string. The implementation in each of the specific DTO classes should be identical and look like this:

```
@Override
    public String visit(IntegrationRequestsHandler handler) {
        return handler.handle(this);
    }
```

What it does it simply calls the visitor class' handle function and passes itself as an argument. Utilizing the double dispatch mechanism, the visitor class will then call the appropriate function to handle the request.

- **IntegrationRequestsHandler** (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/FinalProject/src/main/java/backend/finalproject/IntegrationRequests/IntegrationRequestsHandler.java) - This class acts as the visitor class in the visitor DP. It contains a handle function for each of the DTO classes. The main function that is used to handle **any** integration request is the following:

```
public String handle(HttpRequestDTO request) {
        return request.visit(this);
    }
```

In addition to that, the handler class also contains a function for each of the DTO classes. The function is used to handle the specific request and return the json response as a plain string. Usage example:

```
IntegrationRequestsHandler handler = new IntegrationRequestsHandler();
HttpRequestDTO request = new GetSimulatedStatesRequestDTO();
String response = request.visit(handler);
// do something with the response
// ...
```

- Currently, the supported integration requests in the system are:
    - GetExecutionOutcome (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/FinalProject/src/main/java/DTO/HttpRequests/GetExecutionOutcomeRequestDTO.java)
    - GetLogs (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/FinalProject/src/main/java/DTO/HttpRequests/GetLogsRequestDTO.java)
    - GetSimulatedStates (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/FinalProject/src/main/java/DTO/HttpRequests/GetSimulatedStatesRequestDTO.java)
    - GetSolverActions (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/FinalProject/src/main/java/DTO/HttpRequests/GetSolverActionsRequestDTO.java)

- InitProject (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/FinalProject/src/main/java/DTO/HttpRequests/InitProjectRequestDTO.java)
- ManualAction (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/FinalProject/src/main/java/DTO/HttpRequests/ManualActionPutRequestDTO.java)
- StopRobot (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/FinalProject/src/main/java/DTO/HttpRequests/StopRobotRequestDTO.java)
- UpdateLocalVariable (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/FinalProject/src/main/java/DTO/HttpRequests/UpdateLocalVariableRequestDTO.java)

Adding a new integration request:

The following section is a detailed walk through of how to add a new integration request to the system. We'd demonstrate that on an already existing http request.

Step 1 - Create a new DTO class:

Create a new DTO class for your new integration request. The class should implement the HttpRequestDTO interface. Add any fields you'd like to your class - body fields, params of the request, etc. In our case, this is a simple GET request without body/params. The visit function should look like this:

```
public class GetSimulatedStatesRequestDTO implements HttpRequestDTO{
    @Override
    public String visit(IntegrationRequestsHandler handler) {
        return handler.handle(this);
    }
}
```

Step 2 - Create a new class extending the HttpRequest class (optional):

This should be straight forward to implement. see the following example - GetSimulatedStatesRequest (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/FinalProject/src/main/java/backend/finalproject/IntegrationRequests/GetExecutionOutcomeRequest.java).

Step 3 - Add a new function to the IntegrationRequestsHandler class:

Add a new function to the IntegrationRequestsHandler class. The function should take your new DTO class as an argument.

```
public String handle(GetSimulatedStatesRequestDTO request) {
        // send the request
        // ...
        return response;
    }
```

Or for a more specific implementation:

```
    public String handle(GetExecutionOutcomeRequestDTO request){
        GetExecutionOutcomeRequest getSolverReq = new GetExecutionOutcomeRequest(request.getBeliefSize());
        return send(getSolverReq,getSolverReq.getBody(),GetSolverActionsRequest.REQUEST_TYPE);
    }

public String send(HttpRequest httpRequest, String body, String REQUEST_TYPE) {
        try {
        OkHttpClient client = new OkHttpClient().newBuilder()
        .readTimeout(0, java.util.concurrent.TimeUnit.SECONDS)
        .build();
        MediaType mediaType = MediaType.parse("application/json");
        RequestBody reqBody = body != null ? RequestBody.create(mediaType, body) : null;
        Request.Builder builder = new Request.Builder()
        .url(httpRequest.endpoint)
        .method(REQUEST_TYPE, reqBody);
        for (String key : httpRequest.headers.keySet())
        builder.addHeader(key, httpRequest.headers.get(key));
        Request request = builder.build();
        Response response = client.newCall(request).execute();

        String jsonResponse = new GsonBuilder().create().toJson("");
        return response.body() != null ? response.body().string() : "";
        } catch (IOException e) {
        throw new RuntimeException(e);
        }
        }
```

Basically, the send function is general and can be used for any request. You can utilize it and implement your method similarly.

Step 4: Call the new request:

Say you have an instance of the AOSFacade named facade somewhere in your code. Then you can call the new request like this:

```
// ...
// Create your request DTO class from user input, etc
HttpRequestDTO request = new GetSimulatedStatesRequestDTO();
Response<String> response = facade.sendRequest(request);
if(!response.hasErrorOccured()){
    // if no errors occured, do something with the response
        }
else{
    // handle the error
        }
// ...
```

Application screens

The GUI of the application is designed using the MVC design pattern (Ref: Link (https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller)). Basically, each screen/window is divided into the 3 following parts:

- Model - The data that is displayed/manipulated in the screen. E.g, the env model when creating a new project or the SD/AM of each skill (Ref: EnvModel (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/FinalProject/src/main/java/frontend/finalproject/Model/Env/EnvModel.java))

- View - The visual representation of the screen. I.e. the .fxml files in the resource folder. The layout as well as the styling is defined in those files (Ref: create-env-view (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/FinalProject/src/main/resources/frontend/finalproject/Controllers/create-env-view.fxml))
- Controller - Used to manipulate the model, bind it to the relevant nodes in the view and/or render it whenever it's necessary. Basically, the logic of the screen (Ref: CreateEnvController (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/FinalProject/src/main/java/frontend/finalproject/Controllers/CreateEnvController.java))

In the following section we'll describe some of the main screens of the application.

Note: There are some screens that are built dynamically as they depend on the json outcome of the server. All of those screens must implement the IJsonVisualizer (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/FinalProject/src/main/java/frontend/finalproject/ServerResponseDisplayers/IJsonVisualizer.java) class. This class basically receives a plain json string in its constructor and generates a JavaFX Node component which will be placed in the center of the screen. Each screen that implements this interface can be displayed using the loadResponseStage (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/FinalProject/src/main/java/frontend/finalproject/Utils/UtilsFXML.java#L172) function.

Execution screen

This screen is split into 3 tabs:

Tab #1 : Simulated State

This tab displays the simulated state of the robot. It is implemented by the SimulatedStateVisualizer (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/FinalProject/src/main/java/frontend/finalproject/ServerResponseDisplayers/SimulatedStateVisualizer.java) which implements the IJsonVisualizer interface. The constructor of the class receives the json response of the GetSimulatedStates request. The displayJson() function returns a node that contains the following:

- A tree view of the simulated state (we can also navigate to next/previous states).
  - The variables of the state that have changed as a result of the action that took place, are highlighted.
  - References: TreeVisualizer (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/FinalProject/src/main/java/frontend/finalproject/ServerResponseDisplayers/JsonTreeViewVisualizer.java), Highlighting the tree (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/FinalProject/src/main/java/frontend/finalproject/ServerResponseDisplayers/SimulatedStateVisualizer.java#L214)
- A display button, that displays the current state in a graphical manner (more on that later).

Tab #2 : Execution Outcome

This tab displays the execution outcome of the robot. It is implemented by the ExecutionOutcomeVisualizer.DisplayContainer (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/FinalProject/src/main/java/frontend/finalproject/ServerResponseDisplayers/ExecutionOutcomeVisualizer.java#L383) class. The constructor of the class receives the json response of the GetExecutionOutcome request.

`executionOutcome` is an array. In each element of the array, there is a list of the possible belief states of the robot.

The component that is generated contains the following:

- A raw view of the json response which is displayed in a tree view. We use a VBox named `rawDataContainer` to store the tree view of the json response. An example of how this content is updated:

```
rawDataContainer = new VBox();
rawDataContainer.getChildren().add(new JsonTreeViewVisualizer(executionOutcome.get(0).toString()).displayJSON());
rawDataContainer.setVisible(false);
```

- For each element in the `executionOutcome` array, a histogram (JavaFX's BarChart) is built.
  - There is a histogram for every variable in the state structure. Moreover, histograms can be filtered by the user by variable's name.
  - Reference: buildHistograms (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/FinalProject/src/main/java/frontend/finalproject/ServerResponseDisplayers/ExecutionOutcomeVisualizer.java#L121)

Tab #3 (optional) : Manual Control

This tab is only displayed if the user initialized the last project with Manual Control flag to be true. We keep track of that flag here (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/FinalProject/src/main/java/frontend/finalproject/Utils/UtilsFXML.java#L60)

This tab is implemented by the manual-action-request-view.fxml (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/FinalProject/src/main/resources/frontend/finalproject/Controllers/manual-action-request-view.fxml) view and the ManualActionRequestController (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/FinalProject/src/main/java/frontend/finalproject/Controllers/ManualActionRequestController.java) controller.

If this tab is displayed, a polling thread runs in the background that does the following:

- Polls the server in intervals of 1 second each.
- It checks whether there are any new states that the robot has reached.
    - If there are, the robot updates the view in all of the tabs accordingly.
    - Otherwise, it goes to sleep and wakes up in the next iteration (after ~1 second).
- Reference:
    - polling thread (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/FinalProject/src/main/java/frontend/finalproject/ServerResponseDisplayers/ExecutionOutcomeVisualizer.java#L232) - running the polling thread.
    - onActionSentCallback (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/FinalProject/src/main/java/frontend/finalproject/ServerResponseDisplayers/ExecutionOutcomeVisualizer.java#L245) - the function that is run on each iteration.

This tab contains the following:

- A list of all the possible actions that the robot can perform (double-clicking on an actions triggers a send-action request)
    - When sending an action for the robot to execute
- A list of all the actions the robot has performed.

Create/Edit project screen

This screen is implemented by the create-env-view.fxml (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/FinalProject/src/main/resources/frontend/finalproject/Controllers/create-env-view.fxml) view and the CreateEnvController (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/FinalProject/src/main/java/frontend/finalproject/Controllers/CreateEnvController.java) controller. The model manipulated by the screen is the EnvModel (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/FinalProject/src/main/java/frontend/finalproject/Model/Env/EnvModel.java) class.

The screen contains the following:

- A tree view for each of the variables:
    - Once the user inserts a new variable, it is added to the tree view display.
    - The user can navigate between all the variables and add/edit/delete them.
        - When the user clicks on a variable to edit, a sub-window pops up with the relevant fields to edit.
        - This is done by the loadEditStage (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/FinalProject/src/main/java/frontend/finalproject/Utils/UtilsFXML.java#L138) function. The function receives the name of the fxml file to load (the sub-window), the model of which the sub-window is manipulating, the selected item in the tree view (this will be updated as well once the edit is done) and a callback to be executed once the user completes the edit process (mainly updating the model).
        - Each sub-window that is used as an edit window, its controller implements the EditSubController (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/FinalProject/src/main/java/frontend/finalproject/Controllers/SubControllers/EditSubController.java) interface
    - Similarly, there's also a AddSubController (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/FinalProject/src/main/java/frontend/finalproject/Controllers/SubControllers/AddSubController.java) interface for the sub-windows that are used to add new variables.
- A button to create the project and a button to display the current json of the env model.

Create/Edit skill screen

This screen is implemented by the create-skill-view.fxml (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/FinalProject/src/main/resources/frontend/finalproject/Controllers/create-skill-view.fxml) view, by the CreateSkillController (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/FinalProject/src/main/java/frontend/finalproject/Controllers/CreateSkillController.java) contoller and by the AMModel (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/FinalProject/src/main/java/frontend/finalproject/Model/AM/AMModel.java), SDModel (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/FinalProject/src/main/java/frontend/finalproject/Model/SD/SDModel.java) models representing the AM and SD models respectively.

The screen contains a tab for each of the models.

- Similarly to the add/edit project screen, the user can add/edit/delete variables in the am/sd models.
- There's also a create/save button for each of the models.
- Finally, there's a button to display the current json of the am/sd models.

Logs screen

This screen is implemented by the LogsDisplayer (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/FinalProject/src/main/java/frontend/finalproject/ServerResponseDisplayers/LogsDisplayer.java) class that implemetns the `IJsonVisualizer` interface. The displayer receives the json of the logs and displays it in a table.

This screen contains the following:

- A table containing all the logs that were sent by the server.
- A button to clear the logs table.
- A toggle button to enable/disable notifications for new incoming logs.
    - When enabled, a notification will pop up whenever a new log is received. When disabled, no notifications will pop up.
    - We utilize a polling thread in the background LogsNotificationsFetcher (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/FinalProject/src/main/java/frontend/finalproject/Utils/LogsNotificationsFetcher.java) (running its run() method).
        - In intervals of 1 second, the thread polls the server for new logs.
        - Once the user disabled the logs fetcher, we decrement the semaphore that the thread is waiting on, and the thread is blocked.
        - Alternatively, once the user enables the logs fetcher, we increment the semaphore and the thread is unblocked and continues its polling process.
        - We initialize the thread at the very beginning of the program - here (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/FinalProject/src/main/java/frontend/finalproject/Controllers/Main.java#L20)

Init project screen

This screen contains the form that the user needs to fill in order to initialize a project.

Notifications

In the UtilsFXML (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/FinalProject/src/main/java/frontend/finalproject/Utils/UtilsFXML.java#LL93C1-L121C6) class, we have 2 types of notifications:

- Regular (info) notifications - these are notifications that are displayed for a short period of time and then disappear. Those are used mainly to provide information to the user. Implemented by the following function:

```
private static void showNotification(String title, String text) {
    Notifications notificationBuilder = Notifications.create()
    .title(title)
    .text(text)
    .graphic(null)
    .hideAfter(Duration.seconds(5))
    .position(Pos.TOP_RIGHT);

    notificationBuilder.showConfirm();
    }
```

- Error notifications - These notifications are used mainly to notify the user of an error. Implemented by the following function:

```
private static void showErrorNotification(String title, String text) {
    Notifications notificationBuilder = Notifications.create()
    .title(title)
    .text(text)
    .graphic(null)
    .hideAfter(Duration.INDEFINITE)
    .position(Pos.TOP_RIGHT);

    notificationBuilder.showError();
    }
```

- In order to display a notification to the user simply call the relevant function with the title and text of the notification.
    - There are some predefined texts for the notifications in the NotificationUtils (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/FinalProject/src/main/java/frontend/finalproject/Utils/NotificationUtils.java) class

Display state script

One main requirement of the system is to be able to display the current state of the robot in a graphical manner. We've implemented this requirement with a little help from the user.

The main points of the solution we have provided are as follows:

- The user will specify a path to a python script.
- The purpose of the script is to present the current state of the robot in a graphical manner.
- The script would be executed and its outcome would be displayed to the user.

Script conventions

The script should be written in python and should follow the following conventions:

- The script should contain a function called `display(state,filename = None,is_merging_images = False)` that receives a state (python dictionary), and two optional parameters for internal use. The script displays the state. The script shouldn't return anything.
    - If the user chooses to print the representation to stdout, we'd collect that and display it to the user in a new window.
    - Another, more advanced option, is to use a graphical python library such as matplotlib (https://matplotlib.org/) to display the state in a graphical manner.
        - Another recommended library is PIL (https://pillow.readthedocs.io/en/stable/) (Python Imaging Library) that can be used to display images.
        - **Note: if the user chooses graphical python library, he should save the displayed state to PNG image using the library methods. for example, in matplotlib, we can use savefig method.**
- References:
    - Tic Tac Toe Displayer (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/StateDisplayers/tic_tac_toe_display.py)

    - Toys Displayer (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/StateDisplayers/toys_displayer.py)
    - General Displayer (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/StateDisplayers/general_displayer.py) - This is a general displayer that can be used for any state. It simply prints the state to stdout.

Filenames, paths and other conventions

In this section we'll describe the conventions we've used in the project.

- Repository folder path: `~/Autonomous-Operating-System`
- Project folder path: `~/Autonomous-Operating-System/FinalProject/Projects`
    - All the projects in the system are stored in this folder. The system WILL NOT load any other projects not located inisde of this folder.
    - Conventions:
        - Each folder represents a project. The name of the folder is the name of the project.
        - Inside the folder file naming conventions:
            - `environment.json` - The environment file of the project.
            - `<skill_name>.json` - The SD file of the skill.
            - `<skill_name> glue.json` - The AM file of the skill.
- image_merger.py script path: `~/Autonomous-Operating-System/FinalProject/image_merger.py`
- More constants that we've used can be seen in the Constants (https://github.com/vladiodes/Autonomous-Operating-System/blob/main/FinalProject/src/main/java/backend/finalproject/Constants.java) class.