Discord Bot Automation Assistant

Discord Bot Automation Assistant Chapter 3

Oguz Kaan Yildirim

307637

TABLE OF CONTENTS

TABLE OF CONTENTS 2
CHAPTER THREE: PROJECT ORGANIZATION/STRUCTURE 6
1. Project Requirements (CISC695_Assignment2 and CISC695_Assignment3) 8
1.1 Actors 8
1.2 Use Cases 8
1.2.1 Stop Bot (!stop_bot) 8
1.2.2 Project Help (!project_help) 9
1.2.3 Navigate to Website (!navigate_to_website) 9
1.2.4 Close Browser (!close_browser) 10
1.2.5 Login to a Website (!login) 10
1.2.6 Receive Email (!receive_email) 11
1.2.7 Get Price (!get_price) 11
1.2.8 Start Monitoring Price (!start_monitoring_price) 12
1.2.9 Stop Monitoring Price (!stop_monitoring_price) 12
1.2.10 Check Availability (!check_availability) 13
1.2.11 Start Monitoring Availability (!start_monitoring_availability) 14
1.2.12 Stop Monitoring Availability (!stop_monitoring_availability) 14
1.3 UML use case diagram 15
2. Architecture (CISC695_Assignment5 and CISC695_Assignment10) 16
2.1 Design goal for the project 16
2.2 Component Descriptions 16
2.2.1 Authentication Subsystem 16
2.2.2 Communication Subsystem 16
2.2.2.1 Messaging Subsystem 16
2.2.2.2 Notification Subsystem 17
2.2.3 Monitoring Subsystem 17
2.2.3.1 Price Monitoring Subsystem 17
2.2.3.2 Availability Monitoring Subsystem 17
2.2.4 Data Handling Subsystem 17
2.2.4.1 Data Storage and Retrieval 17

2.2.4.2 Data Export Subsystem 17
2.2.5 Browser Operation Subsystem 17
2.3 Component Diagram 18
2.4 Three-Layer Architecture for Your Project 19
2.4.1 Presentation Layer 19
2.4.2 Business Logic Layer 19
2.4.3 Data Access Layer 19
2.5 Three-Layer Architecture Diagram 21
3. Design (CISC695_Assignment4 and CISC695_Assignment8) 22
3.1 Boundary Objects 22
3.1.1 project_help 22
3.1.2 stop_bot 22
3.1.3 receive_email 22
3.1.4 close_browser 23
3.1.5 login 23
3.1.6 navigate_to_website 23
3.1.7 check_availability 23
3.1.8 start_monitoring_availability 23
3.1.9 stop_monitoring_availability 23
3.1.10 get_price 24
3.1.11 start_monitoring_price 24
3.1.12 stop_monitoring_price 24
3.2 Control Objects 24
3.2.1 project_help 24
3.2.2 stop_bot 24
3.2.3 receive_email 24
3.2.4 navigate_to_website 25
3.2.5 login 25
3.2.6 close_browser 25
3.2.7 check_availability 25
3.2.8 start_monitoring_availability 25
3.2.9 stop_monitoring_availability 25

3.2.10 get_price 26
3.2.11 start_monitoring_price 26
3.2.12 stop_monitoring_price: 26
3.3 Entity Objects 26
3.3.1 AvailabilityEntity 26
3.3.2 BrowserEntity 27
3.3.3 DataExportEntity 27
3.3.4 EmailEntity 28
3.3.5 PriceEntity 28
3.4 Associations Among Objects 28
3.5 Aggregates Among Objects 29
3.6 Attributes for Each Object 29
3.7 UML class diagram 31
3.8 Subsystems 31
3.8.1 Authentication Subsystem 32
3.8.2 Communication Subsystem 32
3.8.3 Browser Operation Subsystem 33
3.8.4 Monitoring Subsystem 34
3.8.5 Data Handling Subsystem 35
3.9 Mapping Associations 35
3.10 Mapping Contracts to Exceptions 38
4. Data Management Strategy (CISC695_Assignment9) 40
5. Technology Stack and Framework 43
5.1 Programming Languages and Frameworks 43
5.1.1 Python 43
5.1.2 Selenium 43
5.1.3 Discord.py 43
5.2 Tools and Platforms 43
5.2.1 Visual Studio Code 43
5.2.2 Git 44
5.2.3 GitHub 44
5.3 Data Management and Storage 44

5.3.1 Configuration Files 44
5.3.2 JSON Files 44
5.3.3 Excel and HTML 44
5.4 Testing Strategy 45
6. Conclusion 46

CHAPTER THREE: PROJECT ORGANIZATION/STRUCTURE

This chapter delves into the detailed design and architecture of the Discord bot project, a sophisticated system engineered to facilitate robust interactions within the Discord environment through a series of automated tasks and responses. The design and development of this project are structured around a series of carefully planned assignments, each contributing essential components to the bot's functionality and operational efficiency.

The objective of this chapter is to provide a thorough exposition of the project's requirements, its architectural blueprint, and the intricate design decisions that collectively underpin the bot's functionality. It serves to bridge the theoretical frameworks discussed in previous chapters with the practical implementations that follow, illustrating the transition from conceptual models to executable solutions.

Project Requirements: The chapter begins by revisiting the project's requirements as outlined in earlier coursework, specifically focusing on the use cases developed for the Discord bot. This section will present a UML use case diagram accompanied by detailed textual descriptions, highlighting how these use cases address the specific needs of the system.

Architecture: After the requirements, the architecture section introduces the UML component and deployment diagrams that illustrate the high-level structure and distribution of the system across various platforms and services. This includes detailing how different components interact within the system and how they are deployed to support scalability and reliability.

Design: The design portion of this chapter will explore the UML class diagrams from the project’s development phase. It will include comprehensive descriptions of each class, their associations, and the dynamic interactions within the system. This section aims to showcase the logical structure of the object-oriented approach used in developing the Discord bot.

Object Detailing: Following the class diagrams, a detailed discussion on the attributes, methods, and contracts of each object within the system will be provided. This will include how these objects map to the underlying data store, emphasizing data handling and object persistence.

Technology Stack/Framework: The chapter will also provide an overview of the technology stack and frameworks utilized in the project. It will include diagrams and explanations of how these technologies are implemented within the framework of the project to meet the set objectives efficiently.

Additional Considerations: This section will touch upon the supplementary design considerations such as programming patterns, principles of clean design, and strategies for achieving a zero-defect implementation. It aims to reflect on the theoretical aspects of software design in light of practical, real-world application.

Conclusion: Finally, the chapter will conclude by summarizing the design and architectural choices made during the project's development phase, reflecting on how these decisions align with the project's initial goals and requirements.

1. Project Requirements (CISC695_Assignment2 and CISC695_Assignment3)

In this section, we will cover the project requirements, including the use case diagram and detailed descriptions of the use cases. We will also integrate relevant parts from assignments to provide a comprehensive understanding.

1.1 Actors

- **User**: The individual or entity utilizing the bot to manage tasks like website navigation, price monitoring, and system control.
- **Bot**: Handles commands, processes data, interacts with websites/APIs, and returns results. It operates within the Discord environment using various controls like browser control, price control, etc.
- **External Systems (Websites/APIs)**: Websites and APIs from which the bot fetches or interacts with data, such as retrieving product prices or checking availability.

1.2 Use Cases

1.2.1 Stop Bot (!stop_bot)

- **Actor**: User
- **Description**: Allows the user to send a command to terminate the bot's operations immediately.
- **Preconditions**: Bot must be operational.
- **Trigger**: User sends the "!stop_bot" command.
- **Main Flow**:
    1. User sends "!stop_bot" command.
    2. Bot recognizes the command and proceeds to shut down.
    3. Bot confirms shutdown process and ceases all operations.

- **Postconditions**: Bot stops running, ceasing all active tasks and interactions.

1.2.2 Project Help (!project_help)

- **Actor**: User
- **Description**: Provides the user with a list of available commands and descriptions on how to use them.
- **Preconditions**: Bot must be operational and accessible to the user.
- **Trigger**: User sends the "!project_help" command.
- **Main Flow**:
    1. User requests help by sending "!project_help".
    2. Bot receives the command and fetches a list of all usable commands along with descriptions.
    3. Bot displays the command list to the user.
- **Postconditions**: User receives the information needed to utilize the bot effectively.

1.2.3 Navigate to Website (!navigate_to_website)

- **Actor**: User
- **Description**: Enables the user to command the bot to open a web browser and navigate to a specified URL.
- **Preconditions**: Bot must be operational.
- **Trigger**: User sends the "!navigate_to_website [URL]" command.
- **Main Flow**:
    1. User inputs the command with a URL.
    2. Bot recognizes the command and extracts the URL.
    3. Bot launches the web browser and navigates to the specified URL.

4. Bot confirms navigation success to the user.

- **Postconditions**: The browser is opened at the desired web page.

1.2.4 Close Browser (!close_browser)

- **Actor**: User
- **Description**: Allows the user to send a command to the bot to close the currently opened web browser.
- **Preconditions**: A web browser must be opened by the bot.
- **Trigger**: User sends the "!close_browser" command.
- **Main Flow**:
    1. User sends the command to close the browser.
    2. Bot receives the command and proceeds to close any open browsers.
    3. Bot confirms the closure of the browser.
- **Postconditions**: Any browser opened by the bot is closed.

1.2.5 Login to a Website (!login)

- **Actor**: User
- **Description**: Enables the user to command the bot to log into a web application using provided credentials.
- **Preconditions**: The target website's login page is accessible.
- **Trigger**: User sends the "!login [website] [username] [password]" command.
- **Main Flow**:
    1. User inputs the command with website URL, username, and password.
    2. Bot recognizes the command, extracts the details, and navigates to the login page of the website.

3. Bot inputs the credentials and attempts to log in.
4. Bot confirms to the user whether the login was successful or if there were any errors.

- **Postconditions**: User is logged into the website if credentials are correct and the website is reachable.

1.2.6 Receive Email (!receive_email)

- **Actor**: User
- **Description**: Commands the bot to send an email with an attached file specified by the user.
- **Preconditions**: Bot must be operational, and the specified file must be present in the system.
- **Trigger**: User sends the "!receive_email [file_name]" command with a valid file name.
- **Main Flow**:
    1. User inputs the command with the name of the file to be emailed (e.g., "!receive_email fileToEmail.html").
    2. Bot recognizes the command and verifies the presence of the file in the system.
    3. Bot attaches the file to an email and sends it to a predetermined recipient.
    4. Bot confirms to the user that the email has been sent successfully or informs them of any issues encountered (e.g., file not found or email delivery failure).
- **Postconditions**: The email is sent with the specified attachment if all conditions are met.

1.2.7 Get Price (!get_price)

- **Actor**: User
- **Description**: Retrieves the current price of a product from a specified URL and logs this information to an Excel or HTML file.
- **Preconditions**: Bot must be operational, and the URL must be accessible.
- **Trigger**: User sends the "!get_price [URL]" command.

- **Main Flow**:
    1. User sends a command with the URL of the product.
    2. Bot recognizes the command, retrieves the current price from the specified URL using web scraping.
    3. Bot logs the price retrieval event to an Excel and HTML file.
    4. Bot displays the price to the user.
- **Postconditions**: The price is displayed to the user and data is logged.

1.2.8 Start Monitoring Price (!start_monitoring_price)

- **Actor**: User
- **Description**: Initiates an ongoing process to monitor price changes at a specified URL, alerting the user via email if there are price changes.
- **Preconditions**: Bot must be operational, and the URL must be accessible.
- **Trigger**: User sends the "!start_monitoring_price [URL] [frequency]" command.
- **Main Flow**:
    1. User specifies the URL and frequency of checks.
    2. Bot begins monitoring the price at the given URL at the specified frequency.
    3. For each check, the bot calls the "!get_price" command to log the current price and check for changes.
    4. The bot sends the saved document as an email.
    5. Bot continues to monitor until the "!stop_monitoring_price" command is issued.
- **Postconditions**: Price monitoring is active, logs are being created at each interval, and emails are sent on price changes.

1.2.9 Stop Monitoring Price (!stop_monitoring_price)

- **Actor**: User
- **Description**: Terminates an ongoing price monitoring process and provides a summary of the results.
- **Preconditions**: Price monitoring process must be active.
- **Trigger**: User sends the "!stop_monitoring_price" command.
- **Main Flow**:
    1. User sends the command to stop monitoring.
    2. Bot receives the command and terminates the ongoing price monitoring.
    3. Bot provides a final summary of monitoring results to the user using the array of results collected during monitoring.
- **Postconditions**: Price monitoring is ceased, and final results are reported to the user.

1.2.10 Check Availability (!check_availability)

- **Actor**: User
- **Description**: Checks the availability of a reservation or booking at a specified URL and logs this information to an Excel or HTML file.
- **Preconditions**: Bot must be operational, and the URL must be accessible.
- **Trigger**: User sends the "!check_availability [URL]" command.
- **Main Flow**:
    1. User sends a command with the URL where the availability needs to be checked.
    2. Bot recognizes the command, retrieves availability data from the specified URL using web scraping.
    3. Bot logs the availability check event to an Excel and HTML file.
    4. Bot displays the availability status to the user.

- **Postconditions**: The availability status is displayed to the user and data is logged.

1.2.11 Start Monitoring Availability (!start_monitoring_availability)

- **Actor**: User
- **Description**: Initiates an ongoing process to monitor changes in availability at a specified URL, alerting the user via email if there are changes in availability.
- **Preconditions**: Bot must be operational, and the URL must be accessible.
- **Trigger**: User sends the "!start_monitoring_availability [URL] [frequency]" command.
- **Main Flow**:
    1. User specifies the URL and frequency of checks.
    2. Bot begins monitoring the availability at the given URL at the specified frequency.
    3. For each check, the bot calls the "!check_availability" command to log the current availability and check for changes.
    4. If an availability change is detected, the bot sends an email with the updated availability information.
    5. Bot continues to monitor until the "!stop_monitoring_availability" command is issued.
- **Postconditions**: Availability monitoring is active, logs are being created at each interval, and emails are sent on availability changes.

1.2.12 Stop Monitoring Availability (!stop_monitoring_availability)

- **Actor**: User
- **Description**: Terminates an ongoing availability monitoring process and provides a summary of the results.
- **Preconditions**: Availability monitoring process must be active.
- **Trigger**: User sends the "!stop_monitoring_availability" command.

- **Main Flow**:
    1. User sends the command to stop monitoring.
    2. Bot receives the command and terminates the ongoing availability monitoring.
    3. Bot provides a final summary of monitoring results to the user using the array of results collected during monitoring.
- **Postconditions**: Availability monitoring is ceased, and results are reported to the user.

1.3 UML use case diagram

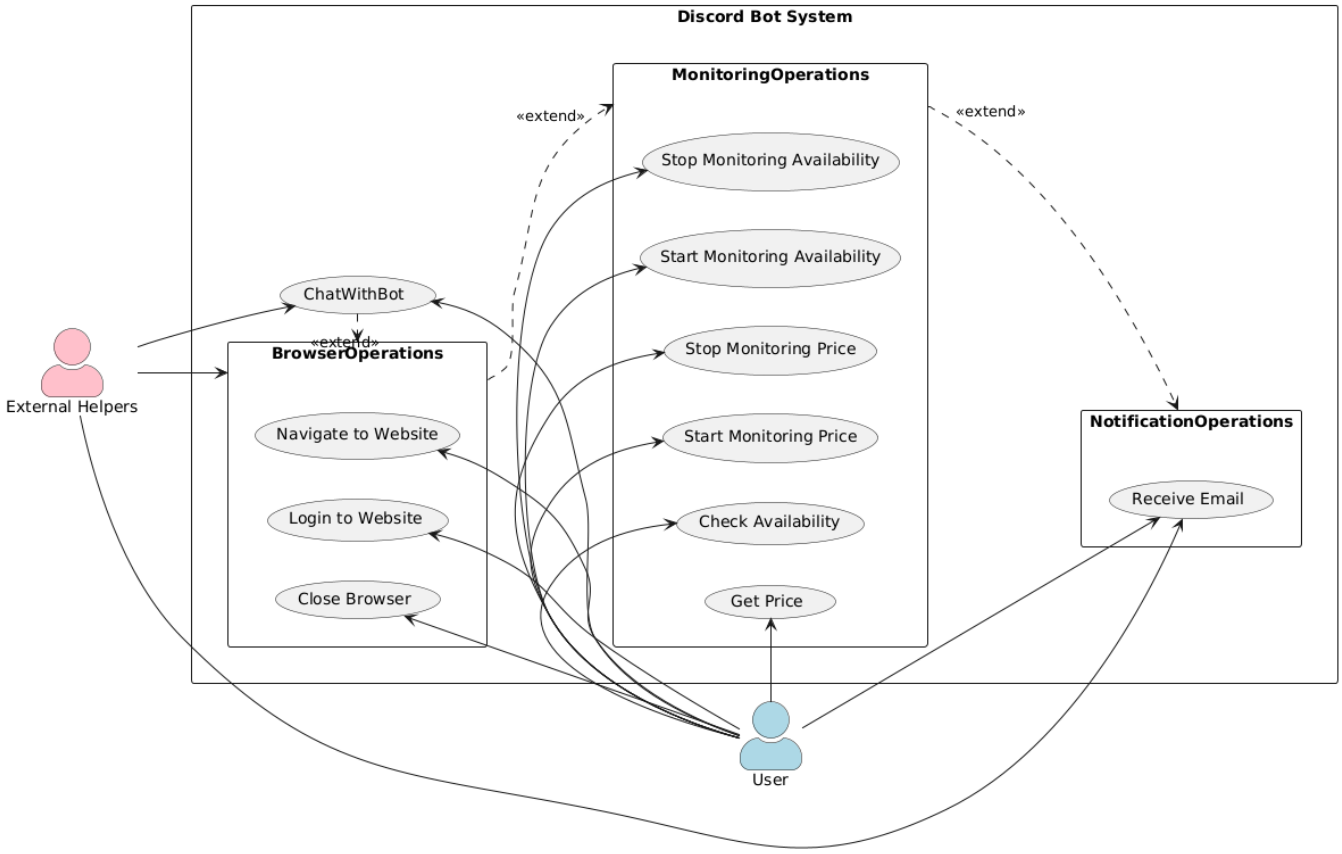

Figure 1: UML use case diagram

2. Architecture (CISC695_Assignment5 and CISC695_Assignment10)

The architecture of the Discord bot project forms the backbone of its functionality, providing a robust framework for managing interactions within the Discord environment. This section outlines the system's architectural design, explaining how it supports the operational requirements and enhances the bot's capabilities. By detailing the architectural components and their deployment, this section demonstrates the scalability, reliability, and efficiency of the system.

2.1 Design goal for the project

Usability: The system should be designed to ensure ease of use, where users can effectively interact with the bot to perform tasks such as monitoring prices, checking availability, and receiving notifications with minimal effort and learning curve. It should provide clear, understandable feedback and guidance to the user, enabling them to accomplish their desired tasks efficiently. The user interface should be simple to navigate, with commands that are easy to remember and execute, and errors should be handled gracefully, providing users with informative messages to guide their next steps.

2.2 Component Descriptions

2.2.1 Authentication Subsystem

This subsystem would handle user authentication processes, including verifying credentials, managing session tokens, and integrating with email for user verification or password resets.

2.2.2 Communication Subsystem

2.2.2.1 Messaging Subsystem

Manages all interactions within the bot environment, including command parsing and response handling.

2.2.2.2 Notification Subsystem

Handles outgoing notifications, such as email alerts for price changes or availability updates, which could be a distinct or integrated part of the Messaging Subsystem

2.2.3 Monitoring Subsystem

2.2.3.1 Price Monitoring Subsystem

Monitors price changes of specified items and logs these changes. It may trigger notifications based on defined criteria.

2.2.3.2 Availability Monitoring Subsystem

Like price monitoring but focuses on the availability status of items or services, providing updates and logs as defined.

2.2.4 Data Handling Subsystem

2.2.4.1 Data Storage and Retrieval

Manages data interactions, including saving logs to databases or files and retrieving them for user requests.

2.2.4.2 Data Export Subsystem

Handles the formatting and exporting of data to various formats like Excel or HTML, serving both internal logging needs and user-requested exports.

2.2.5 Browser Operation Subsystem

Manages all interactions requiring a web browser, such as navigating to websites, logging in, and other browser-based actions.

2.3 Component Diagram

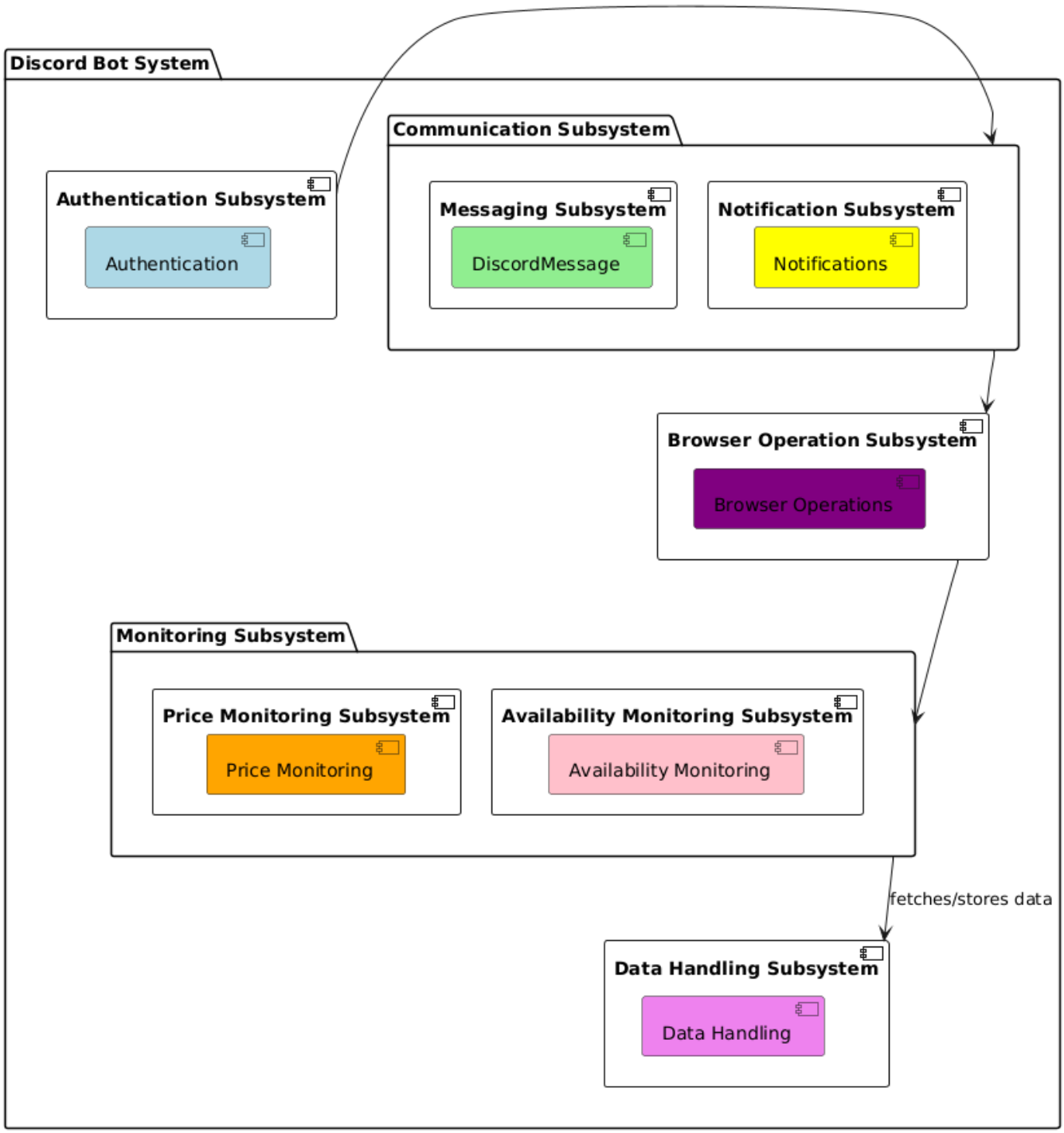

Figure 2: Component Diagram

2.4 Three-Layer Architecture for Your Project

The Discord bot is architected around a three-layer model, designed to separate concerns, enhance maintainability, and optimize the scalability of the application. This architectural structure allows for clear delineation of responsibilities, making the system easier to manage, extend, and scale. Each layer is tailored to handle specific aspects of the system's functionality, from user interaction to core processing logic and data management.

2.4.1 Presentation Layer

This layer is the front-end of the bot, interfacing directly with users. It manages all interactions through command interfaces on Discord, ensuring that user commands are interpreted and responded to efficiently. Key components of the Presentation Layer include:

- **User Interface**: Handles various types of user commands, such as navigation, price checks, monitoring, and managing email notifications, providing a smooth and intuitive user experience.

2.4.2 Business Logic Layer

The Business Logic Layer is the core of the bot, where the functional logic resides. This layer processes all user commands and manages the bot's operations, ensuring that all tasks are executed correctly. It includes:

- **Control Objects**: These are crucial for managing browser operations, monitoring prices, checking availability, and handling user authentication. Each control object is responsible for a specific set of tasks and works together to ensure the bot operates seamlessly.

2.4.3 Data Access Layer

At the foundation of the architecture is the Data Access Layer, which handles all data storage and retrieval operations. Unlike traditional applications that might use a relational database, this bot

utilizes a file-based approach to store and manage data. Components of this layer focus on:

- **Entity Management and Data Exports**: Managing data entities and facilitating the export of data to various formats such as Excel and HTML. This allows for effective data management and easy access to analytics and reports needed for monitoring and decision-making purposes.

This three-layered approach not only supports a clean separation of concerns but also enhances the system's ability to evolve and adapt to changing requirements without disrupting other components of the architecture.

2.5 Three-Layer Architecture Diagram

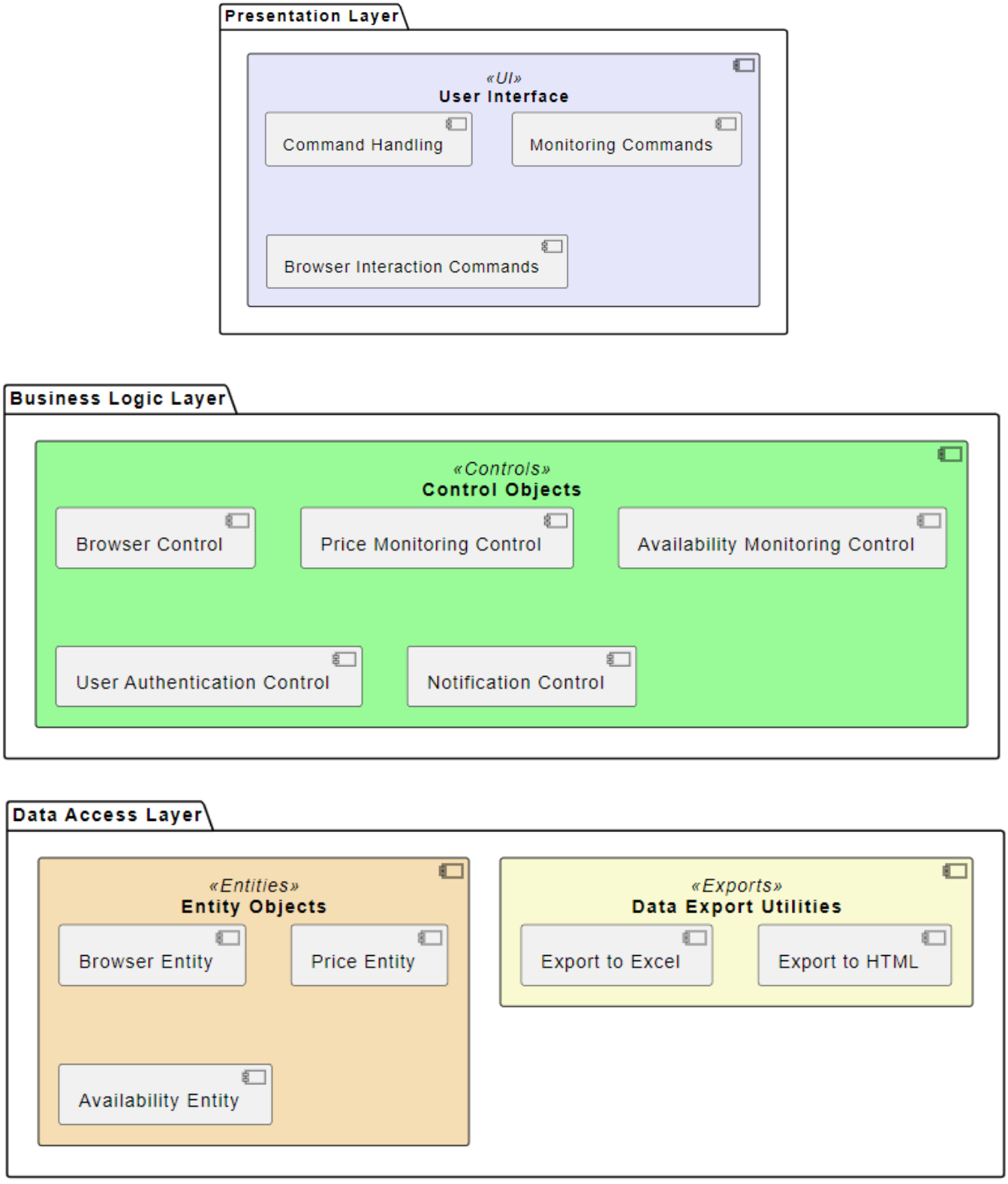

Figure 3: Three-Layer Architecture Diagram

3. Design (CISC695_Assignment4 and CISC695_Assignment8)

This section delves into the design framework of the Discord bot, illustrating how complex interactions are managed within a structured environment. Through UML class diagrams and detailed descriptions, we explore the bot's operational logic and its component interactions.

The design section outlines the configuration of boundary objects, control objects, and entity objects, each integral to the bot's functionality. Boundary objects serve as the interface between user commands and the system’s logic, ensuring inputs are accurately processed. Control objects, the decision-making core, manage the operational flow, orchestrating responses and actions efficiently. Entity objects, responsible for data manipulation, ensuring data integrity and availability, crucial for the bot's operations.

3.1 Boundary Objects

Each boundary object is specifically designed to parse user commands received via Discord, extracting necessary data before interacting with the appropriate control objects to fulfill the user's requests.

Bot Boundary Objects

3.1.1 project_help

Interprets the user's request for help, parses the command, and communicates with the bot control to retrieve and display a list of available commands along with their descriptions.

3.1.2 stop_bot

Processes the user's command to terminate all bot operations, parses the message, and interacts with the bot control to initiate the shutdown.

3.1.3 receive_email

Handles the command to send an email with an attached file, parses the user's message to

determine the file to be attached, and coordinates with the control object to manage the email sending process.

Browser Boundary Objects

3.1.4 close_browser

Processes the command to close the web browser, parses the message, and instructs the browser control to end the browser session.

3.1.5 login

Manages the user's command to log into a website, parsing details like the website URL, username, and password before passing them to the browser control for the login operation.

3.1.6 navigate_to_website

Captures and parses the user's command to navigate to a specific URL, then communicates with the browser control to perform the navigation.

Availability Boundary Objects

3.1.7 check_availability

Receives and parses the user's message to extract necessary data such as the URL and date, then contacts the corresponding control object to check availability at the provided URL.

3.1.8 start_monitoring_availability

Takes the user's input to begin monitoring availability at a specified URL with certain frequency parameters, parses the message, and forwards the data to the control layer to initiate monitoring.

3.1.9 stop_monitoring_availability

Captures the command to cease monitoring availability, parses the user's instructions, and passes the command to the control object to stop the monitoring process.

Price Boundary Objects

3.1.10 get_price

Receives the command to retrieve a price from a specified URL, parses the command to extract the URL, and contacts the price control to obtain and return the price.

3.1.11 start_monitoring_price

Receives the command to start monitoring the price at a specified URL and interval, parses the message for necessary details, and forwards these to the price control to begin the monitoring process.

3.1.12 stop_monitoring_price

Processes the command to stop price monitoring, parses the user's instructions, and notifies the price control to end the monitoring and summarize the findings.

3.2 Control Objects

Each control object acts as a decision-making hub that processes input from its corresponding boundary object, directs operations by interacting with entity objects or utilities (like logging or sending emails), and ultimately returns the outcome to the boundary object for user communication.

1. **BotControl**

3.2.1 project_help

Generates and returns a list of all available commands and their descriptions, assisting the user in navigating the bot's functionalities.

3.2.2 stop_bot

Coordinates the shutdown process of the bot, ensuring all operations are cleanly terminated.

3.2.3 receive_email

Manages the attachment and sending of an email with specified files, liaising with EmailEntity to perform the email operations.

2. **BrowserControl**

3.2.4 navigate_to_website

Checks if the URL is valid or provided. If everything is valid, then contacts the BrowserEntity to perform the actual action.

3.2.5 login

Checks if the URL, username, and password are valid or provided. If everything is valid, then contacts the BrowserEntity to perform the actual login action.

3.2.6 close_browser

Checks if there is an open session, then contacts the BrowserEntity to close the browser.

3. **AvailabilityControl**

3.2.7 check_availability

Checks if the URL is provided or not. If not, takes the default URL from the CSS selectors file. Contacts the AvailabilityEntity to verify availability at a specified URL and date, retrieves the availability status, calls the entity's data export method to save data.

3.2.8 start_monitoring_availability

Initiates a monitoring process at defined intervals by repeatedly calling the check_availability method, handling the scheduling and continuation of this process, and calls the receive_email method/control object after obtaining data.

3.2.9 stop_monitoring_availability

Ends the monitoring process, summarizes the collected data, and returns the final status to the boundary object for user notification.

4. **PriceControl**

3.2.10 get_price

Checks if the URL is provided or not. If not, takes the default URL from the CSS selectors file. Contacts the PriceEntity to fetch the price at a specified URL and calls the entity's data export method to save data.

3.2.11 start_monitoring_price

Initiates a monitoring process at defined intervals by repeatedly calling the get_price method, handling the scheduling and continuation of this process, and calls the receive_email method/control object after obtaining data.

3.2.12 stop_monitoring_price:

Terminates the price monitoring process, summarizes the collected data, and communicates the results back to the boundary for user notification.

3.3 Entity Objects

These entities act as the data manipulation layer of your architecture, directly interacting with the data sources and external systems to fetch, process, and store the required information. They provide a clean separation of concerns by encapsulating the logic needed to interact with data sources from the rest of the application, ensuring that the control objects can remain focused on application logic without needing to deal directly with data handling specifics.

3.3.1 AvailabilityEntity

- **Purpose**: Handles all data operations related to checking and monitoring availability. It directly interacts with external systems or databases to retrieve availability information.
- **Key Methods**:
  - **check_availability**: Connects to external services to check availability at the given

URL on a specified date. It manages direct interactions with web APIs or databases to fetch availability data.

- **export_data**: Saves or logs availability data to local storage or a database. It might format the data for export to files such as Excel or HTML formats, which are then used for reporting or email notifications.

3.3.2 BrowserEntity

- **Purpose**: Manages all operations that require direct interaction with a web browser, such as opening, navigating, or closing a browser. It encapsulates all functionalities that involve web automation tools like Selenium.
- **Key Methods**:
  - **launch_browser**: Opens a web browser session with predefined configurations.
  - **navigate_to_website**: Navigates to a specified URL within an open browser session.
  - **close_browser**: Closes the currently open web browser session to free up resources.

3.3.3 DataExportEntity

- **Purpose**: Responsible for exporting data into various formats for storage or transmission. This entity ensures data from operations like price checks or availability monitoring is logged appropriately.
- **Key Methods**:
  - **export_to_excel**: Formats and writes data to an Excel file, organizing data into sheets and cells according to specified schemas.
  - **export_to_html**: Converts data into HTML format for easy web publication or

email attachments.

3.3.4 EmailEntity

- **Purpose**: Handles the configuration and process of sending emails. This entity works with email servers to facilitate the sending of notifications, alerts, or reports generated by the system.
- **Key Methods**:
  - **send_email_with_attachments**: Prepares and sends an email with specified attachments. It manages attachments, formats the email content, and interacts with email servers to deliver the message.

3.3.5 PriceEntity

- **Purpose**: Specializes in fetching and monitoring price data from various online sources. It uses web scraping techniques to extract pricing information from web pages.
- **Key Methods**:
  - **get_price**: Retrieves the current price of a product from a specified URL. It scrapes the web page to find pricing information and returns it to the control layer.
  - **export_data**: Similar to the AvailabilityEntity, it exports price data to various file formats for reporting or further analysis.

3.4 Associations Among Objects

- **Boundary to Control Associations**
  - AvailabilityBoundary communicates with AvailabilityControl.
  - BotBoundary communicates with BotControl.
  - BrowserBoundary communicates with BrowserControl.

- PriceBoundary communicates with PriceControl.

- **Control to Entity Associations**
  - AvailabilityControl interacts with AvailabilityEntity, DataExportEntity, and EmailEntity.
  - BotControl interacts with EmailEntity.
  - BrowserControl interacts with BrowserEntity.
  - PriceControl interacts with PriceEntity and DataExportEntity.

3.5 Aggregates Among Objects

- **Availability Aggregate**:
  - Root: AvailabilityEntity
  - Includes: AvailabilityControl (manages AvailabilityEntity and potentially accesses DataExportEntity and EmailEntity for output operations).
- **Price Aggregate**:
  - Root: PriceEntity
  - Includes: PriceControl (manages PriceEntity and handles data through DataExportEntity).
- **Email Aggregate**:
  - Root: EmailEntity
  - Includes: Both BotControl and AvailabilityControl may use this for sending emails, positioning it as a shared resource.

3.6 Attributes for Each Object

- **Boundary Objects Attributes**:
  - BotBoundary: commands !stop_bot, !project_help, !receive_email
  - BrowserBoundary: commands !navigate_to_website, !login, !close_browser,

- AvailabilityBoundary: Commands: !check_availability, !start_monitoring_availability, !stop_monitoring_availability
- PriceBoundary: commands !get_price, !start_monitoring_price, !stop_monitoring_price

- **Control Objects Attributes**:
  - AvailabilityControl: monitoring_active (boolean), scheduled_tasks (list of tasks).
  - BotControl: active_sessions (number of active bot sessions).
  - BrowserControl: browser_instance (current instance of the browser).
  - PriceControl: price_history (historical prices), monitoring_active (boolean).
- **Entity Objects Attributes**:
  - AvailabilityEntity: availability_data, last_checked.
  - BrowserEntity: cookies, session_data.
  - DataExportEntity: file_paths (locations of saved data).
  - EmailEntity: email_queue (emails waiting to be sent).
  - PriceEntity: price_data, last_updated.

3.7 UML class diagram

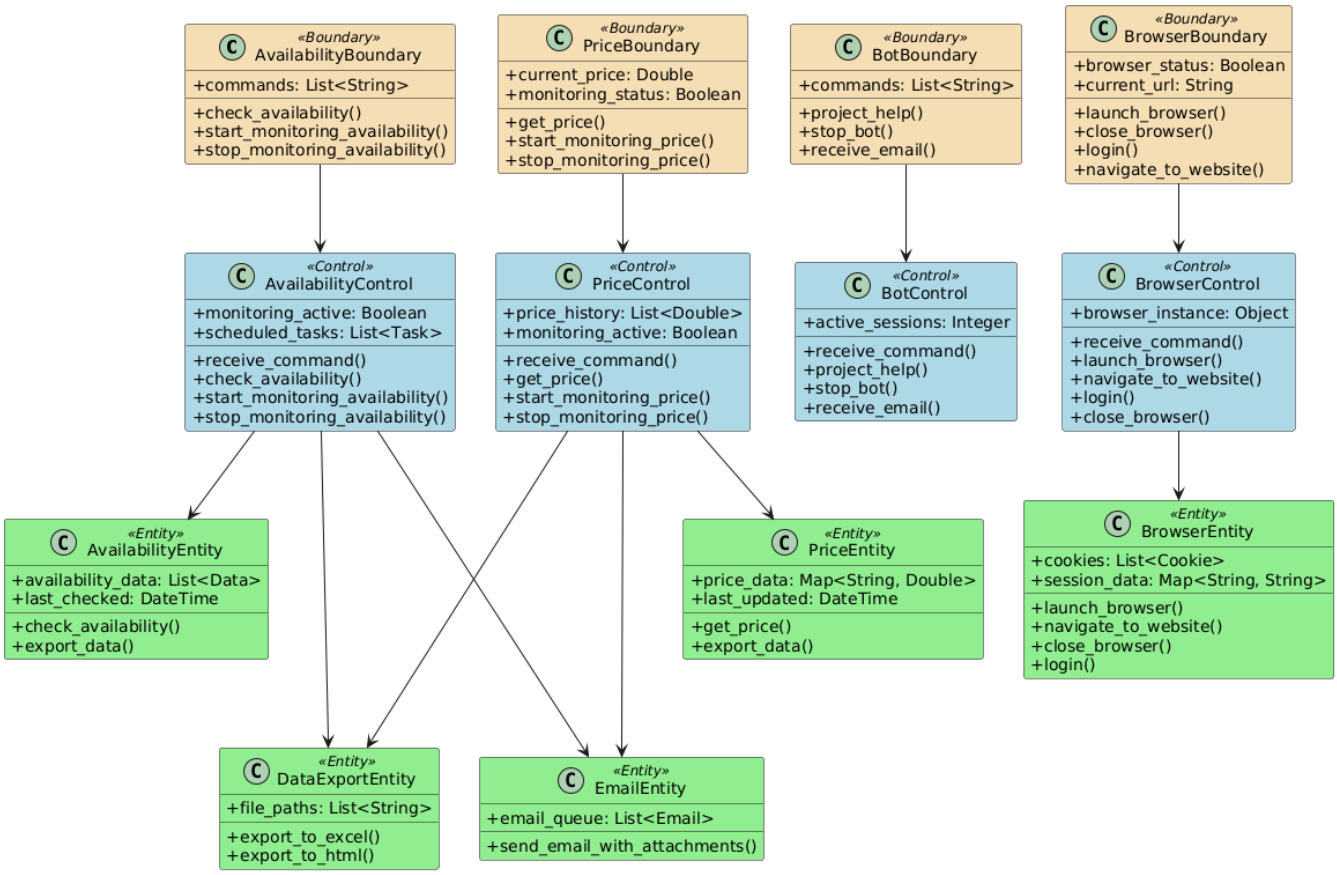

Figure 4: UML Class Diagram

3.8 Subsystems

This part of the chapter delves into the intricate design elements of the Discord bot, focusing on the structured organization of its subsystems. Each subsystem is designed to execute specific functions that collectively ensure the bot operates effectively within its environment. We explore Authentication, Monitoring, Browser Operation, Data Handling, and Communication subsystems, detailing their classes, attributes, operations, and interconnections. This section underscores how these components interact to manage data flow, process commands, and maintain system integrity, enhancing the bot's responsiveness and reliability.

3.8.1 Authentication Subsystem

Class: Auth

- **Attributes**:
  - user_id: integer, private - Unique identifier for the user.
  - session_token: string, private - Token that maintains the session state across interactions.
  - email_host: string, private - SMTP server host, derived from config (EMAIL_HOST).
  - email_port: integer, private - SMTP server port, derived from config (EMAIL_PORT).
  - email_user: string, private - SMTP server username, derived from config (EMAIL_USER).
  - email_password: string, private - SMTP server password, derived from config (EMAIL_PASSWORD).
- **Operations**:
  - verify_credentials(username, password): boolean, public - Validates user credentials against stored data.
  - send_verification_email(email_address): void, public - Sends an email to verify the user's email address using SMTP settings.

3.8.2 Communication Subsystem

This subsystem facilitates all user interactions and communications, both inbound and outbound.

Class: MessagingSubsystem

- **Attributes**:
  - command_parser: CommandParser, private - An instance of the CommandParser class to parse user commands.

- response_templates: dict, private - A dictionary mapping commands to response templates for generating user messages.

- **Operations**:
  - receive_message(user_input): string, public - Receives input from the user, parses it to determine the command, and retrieves the appropriate response.
  - send_message(channel_id, message): void, public - Sends a message to a specific Discord channel, using the channel ID.
  - format_response(command, data): string, public - Formats the response based on the command executed and the data retrieved from the system.

Class: NotificationSubsystem

- **Attributes**:
  - email_config: EmailConfig, private - Configuration settings for email, containing host, port, user, and password.

- **Operations**:
  - send_email(subject, body, recipient): void, public - Sends an email to the specified recipient with a subject and body.
  - notify_user(channel_id, message): void, public - Sends a notification message to a user via a specific Discord channel.

generate_email_body(template_id, data): string, public - Generates the body of the email based on a template and data, useful for price changes, availability updates, etc.

3.8.3 Browser Operation Subsystem

Class: BrowserController

- **Attributes**:

- browser_instance: Object, private - Instance of the browser controlled by the bot.

- **Operations**:
  - open_browser(url): void, public - Opens a browser window and navigates to the specified URL.
  - close_browser(): void, public - Closes the currently open browser window.

login(url, username, password): boolean, public - Logs into a website with provided credentials.

3.8.4 Monitoring Subsystem

Class: PriceMonitor

- **Attributes**:
  - url: string, private - URL to monitor for price changes.
  - current_price: float, private - Most recent fetched price.
- **Operations**:
  - check_price(): float, public - Fetches and returns the current price from the URL.
  - start_monitoring(frequency): void, public - Begins monitoring the price at specified intervals.
  - stop_monitoring(): void, public - Stops monitoring the price.

Class: AvailabilityMonitor

- **Attributes**:
  - url: string, private - URL to monitor for availability changes.
  - availability_status: string, private - Latest availability status fetched.
- **Operations**:
  - check_availability(): string, public - Checks and returns the current availability from the URL.

- start_monitoring_availability(frequency): void, public - Starts monitoring availability at specified intervals.
- stop_monitoring_availability(): void, public - Stops monitoring availability.

3.8.5 Data Handling Subsystem

Class: DataHandler

- **Attributes**:
  - database_connection: Object, private - Connection object for the database.
- **Operations**:
  - save_data(data): void, public - Saves specified data to the database.
  - retrieve_data(query): Object, public - Retrieves data from the database based on the query.
  - export_data_to_excel(data): void, public - Exports data to an Excel file.
  - export_data_to_html(data): void, public - Converts and saves data in HTML format.

3.9 Mapping Associations

1. **Authentication Subsystem**
   - **Auth** associates with **NotificationSubsystem** for sending verification emails.
     - **Association Type**: Composition
     - **Multiplicity**: One Auth may utilize one NotificationSubsystem to handle email functionalities.
2. **Monitoring Subsystem**
   - **PriceMonitor** and **AvailabilityMonitor** should be able to log data using **DataHandler**.
     - **Association Type**: Aggregation
     - **Multiplicity**: One Monitoring class may use one DataHandler to log multiple

datasets.

- Both monitors may need to send notifications through **NotificationSubsystem** when specific thresholds or conditions are met.
  - **Association Type**: Aggregation
  - **Multiplicity**: One Monitoring class can use one NotificationSubsystem for various alert purposes.

3. **Browser Operation Subsystem**
   - **BrowserController** might need to authenticate via the **Auth** class for operations requiring secure access.
     - **Association Type**: Aggregation
     - **Multiplicity**: One BrowserController uses one Auth for managing user sessions and logins.
4. **Data Handling Subsystem**
   - **DataHandler** might be used by almost every other subsystem for data logging and retrieval purposes, establishing a broad association with many classes.
     - **Association Type**: Aggregation
     - **Multiplicity**: Multiple classes (like PriceMonitor, AvailabilityMonitor, etc.) can use one DataHandler.
5. **Communication Subsystem**
   - **MessagingSubsystem** uses **DataHandler** to retrieve data for response formatting.
     - **Association Type**: Aggregation
     - **Multiplicity**: One MessagingSubsystem retrieves and processes multiple data points from DataHandler.

- **NotificationSubsystem** should interact with **DataHandler** to log notification activities or email statistics.
    - **Association Type**: Aggregation
    - **Multiplicity**: One NotificationSubsystem records multiple logs via DataHandler.

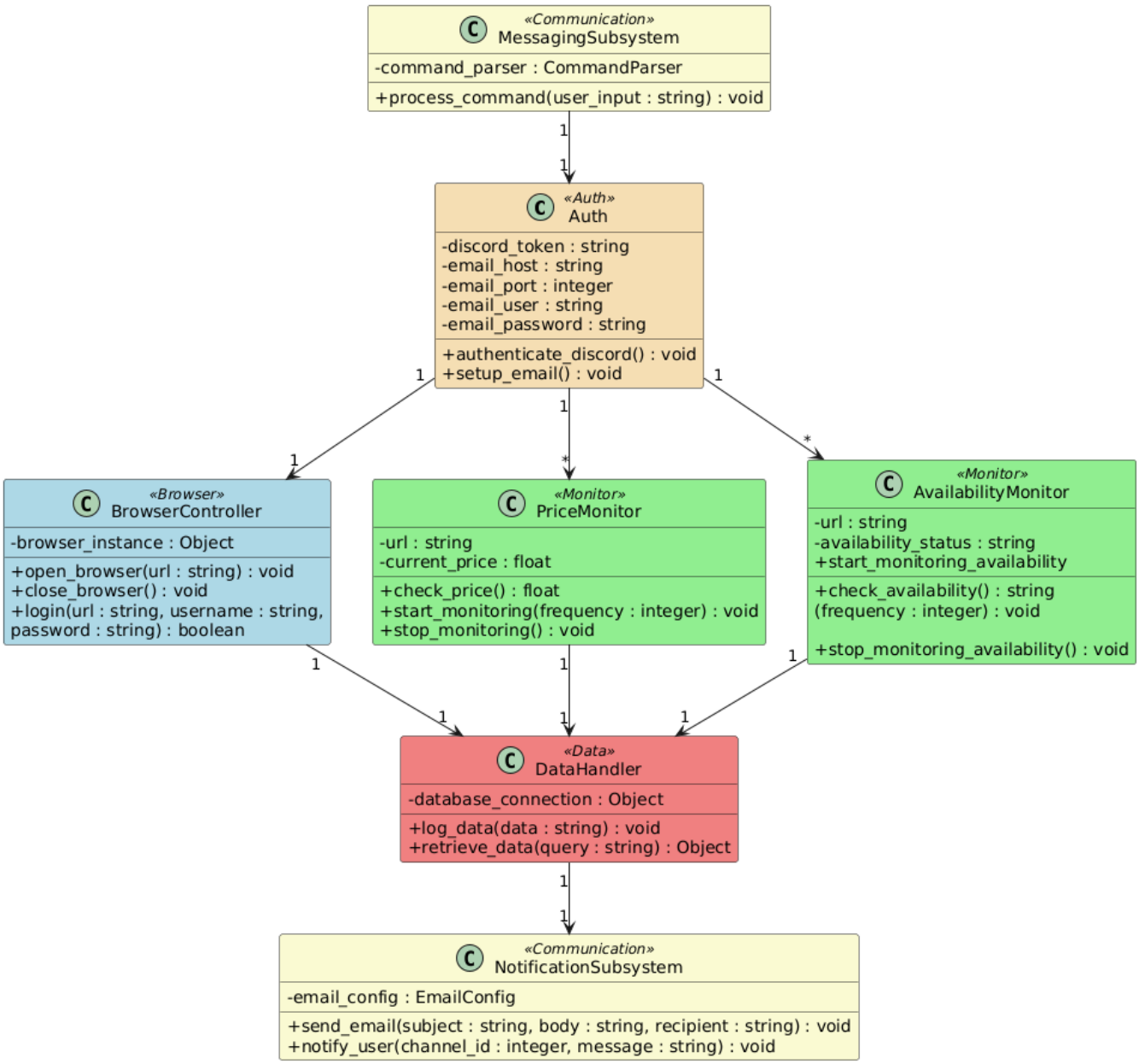

Figure 5: Class Interaction Diagram

3.10 Mapping Contracts to Exceptions

Auth Class:

- **verify_credentials(username, password)**:
    - **Contract**: Username and password must not be null or empty.
    - **Exceptions**:
        - InvalidCredentialsException: Thrown if the credentials do not match.
        - NullValueException: Thrown if any parameter is null.
- **send_verification_email(email_address)**:
    - **Contract**: Email address must be valid and not null.
    - **Exceptions**:
        - InvalidEmailException: Thrown if the email format is invalid.
        - EmailSendFailureException: Thrown if the email fails to send.

Monitoring Classes (PriceMonitor & AvailabilityMonitor):

- **check_price() / check_availability()**:
    - **Contract**: URL must be accessible and properly formatted.
    - **Exceptions**:
        - URLNotReachableException: Thrown if the URL cannot be accessed.
        - InvalidURLException: Thrown if the URL format is incorrect.

BrowserController Class:

- **open_browser(url)**:
    - **Contract**: URL must be valid and the browser instance must not already be open.
    - **Exceptions**:
        - BrowserAlreadyOpenException: Thrown if attempting to open a new browser

session while one is already active.

- InvalidURLException: Thrown if the URL format is incorrect.

DataHandler Class:

- **save_data(data)**:
  - **Contract**: Data must not be null and database connection must be active.
  - **Exceptions**:
    - DatabaseConnectionException: Thrown if the connection is inactive.
    - NullDataException: Thrown if null data is passed.

MessagingSubsystem Class:

- **send_message(channel_id, message)**:
  - **Contract**: Channel ID must exist and message must not be null.
  - **Exceptions**:
    - InvalidChannelException: Thrown if the channel does not exist.
    - NullMessageException: Thrown if the message is null.

4. Data Management Strategy (CISC695_Assignment9)

In the latest iteration of our Discord bot, a strategic decision was made to deviate from conventional relational database systems, favoring a more agile and less resource-intensive file-based data storage mechanism. This pivot was driven by a comprehensive analysis of the project's unique requirements, namely, the need for speed, flexibility, and handling predominantly non-transactional data.

System Requirements and Flexibility

The dynamic nature of interactions within the Discord environment necessitates a data management system that can swiftly adapt to changes without the latency often associated with relational databases. The bot's operations, primarily non-transactional and ephemeral data exchanges (such as session data or temporary preferences), benefit significantly from the agility offered by a file-based approach.

Reduced Complexity and Overhead

Managing a traditional database involves significant setup, maintenance, and overhead costs. By employing file-based storage, the system sidesteps complexities related to database schema design, data normalization, and transaction management. This reduction in overhead not only simplifies deployment but also enhances the system's overall performance.

Scalability and Security

The chosen strategy simplifies scaling operations horizontally by distributing file storage across multiple nodes or services, without the need for complex database replication strategies. Security is enhanced as sensitive information, such as authentication tokens and SMTP settings, is managed through .env files. These configurations are loaded into the environment at runtime, isolating sensitive details from the core application logic and minimizing exposure to security vulnerabilities.

Environmental Variables and Security

Tokens and credentials are stored securely within environment files, parsed and loaded at runtime using files like .py or .json. This ensures that sensitive data is not hardcoded into the application's source code, providing an added layer of security by segregating configuration from deployment.

```
import os
from dotenv import load_dotenv

load_dotenv()  # Load all the environment variables from a .env file
DISCORD_TOKEN = os.getenv('DISCORD_TOKEN')
EMAIL_HOST = os.getenv('EMAIL_HOST')
EMAIL_PORT = int(os.getenv('EMAIL_PORT'))
EMAIL_USER = os.getenv('EMAIL_USER')
EMAIL_PASSWORD = os.getenv('EMAIL_PASSWORD')
```

Transient and Persistent Data Handling

JSON for Transient Data

Transient data such as user preferences or session states are stored in JSON files. This format is particularly advantageous for its human-readable format and ease of integration with Python, allowing for quick reads and writes. It effectively addresses the need for storing non-sensitive, session-specific data which does not require long-term persistence.

HTML and Excel for Persistent Data Logging

For long-term data storage, such as logging price monitoring histories or user interaction data, the bot utilizes Excel and HTML formats. This method not only ensures data is easily accessible and reviewable by end-users but also supports automated reporting functionalities through email, enhancing user engagement and satisfaction.

Data Flow and Processing

The data flow in our project is structured to minimize latency and maximize responsiveness.

User commands are parsed and executed in real-time, with data processed immediately and output generated without delay typically associated with database transactions:

- *Command Processing:* Commands from users, such as checking prices or setting alerts, are parsed by the bot and processed immediately. The results of these commands dictate the subsequent actions, whether they're fetching data from a web API or logging information to a file.
- *Immediate Feedback and Output:* Upon processing commands, feedback is immediately generated and provided to the user either via Discord messages or through generated reports in Excel or HTML format. This instant feedback loop is crucial for the interactive nature of a Discord bot.

Advantages and Considerations

This non-database approach, while unconventional, offers several advantages, including simplicity in deployment and lower overhead in terms of database management and maintenance. However, it also poses challenges, particularly in handling large volumes of data or ensuring data integrity during concurrent accesses. These challenges are mitigated through careful system design and the use of file locks and temporary storage conventions.

5. Technology Stack and Framework

This section delves into the technology stack and frameworks that power the Discord bot, focusing on the tools and technologies that facilitate rapid development, seamless user interaction, and efficient data management.

5.1 Programming Languages and Frameworks

5.1.1 Python

- **Role**: Primary programming language for developing the bot.
- **Features**: Chosen for its readability, robust standard library, and extensive support through third-party libraries, Python underpins all major functionalities of the bot, from data scraping to process automation and interaction handling.

5.1.2 Selenium

- **Role**: Automates web browsers to extract real-time product prices and availability.
- **Capabilities**: Simulates human interactions with web pages, allowing the bot to perform complex navigations and data extraction tasks, critical for accurate price monitoring.

5.1.3 Discord.py

- **Role**: Handles communications with the Discord API.
- **Functionality**: Manages user interactions, receives commands, sends notifications, and embeds the bot seamlessly within Discord communities.

5.2 Tools and Platforms

5.2.1 Visual Studio Code

- **Role**: Preferred IDE for writing, testing, and debugging the bot's code.
- **Advantages**: Offers extensive plugin support, built-in Git control, and integrated terminal, which streamline the coding and version control processes.

5.2.2 Git

- **Role**: Manages source code versions and collaborative features.
- **Benefits**: Essential for tracking code changes, managing branches, and integrating changes from multiple contributors, ensuring consistency and continuity in the development process.

5.2.3 GitHub

- **Role**: Hosts the source code repository and facilitates collaborative features like issue tracking and code reviews.
- **Integration**: Centralizes source control and acts as a platform for continuous integration and deployment strategies.

5.3 Data Management and Storage

Tjis project utilizes a combination of configuration files, JSON, and direct file output mechanisms for managing both transient and persistent data:

5.3.1 Configuration Files

- **Role**: Manage operational parameters and sensitive credentials, such as Tokens, SMTP settings.
- **Implementation**: Stored in .py files, these parameters are loaded dynamically into the application environment, enhancing security by segregating configuration from the code.

5.3.2 JSON Files

- **Role**: Handle transient data like session states and user preferences.
- **Advantages**: Offers flexibility and speed in accessing and updating data, ideal for non-sensitive, temporary information.

5.3.3 Excel and HTML

- **Role**: Serve as formats for logging long-term data and generating reports.
- **Functionality**: Facilitates easy distribution and accessibility of data, allowing comprehensive reporting and analysis through automated emails.

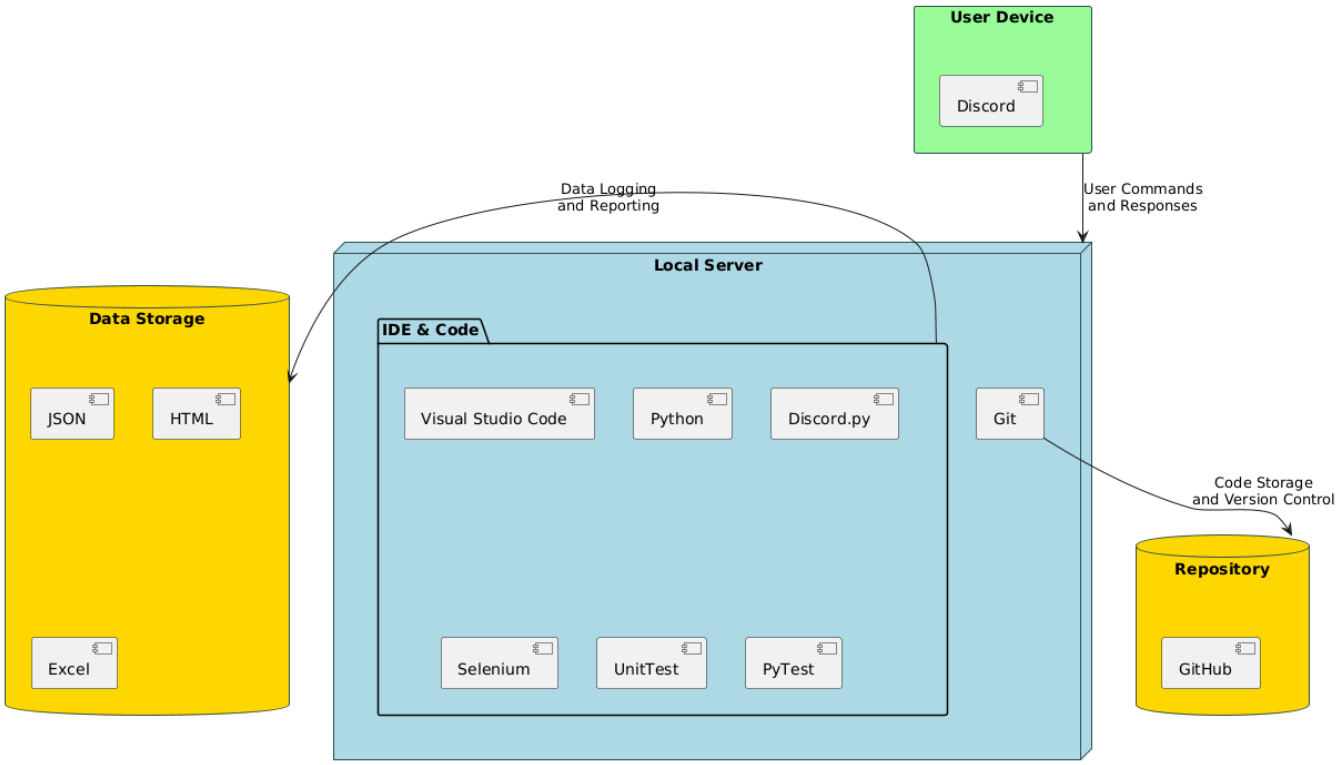

Figure 6: System Architecture Diagram

5.4 Testing Strategy

Our project employs a robust testing framework using Python's unittest library and unittest.mock for mocking external dependencies. This strategy ensures that each component of the bot functions as expected under various scenarios. A detailed exploration of our testing approach and methodologies will be presented in the subsequent chapter.

6. Conclusion

This chapter has thoroughly examined the structured organization and design of the Discord Bot project, a sophisticated system engineered to facilitate robust interactions within the Discord environment. From detailed use case diagrams to intricate class and component diagrams, we have explored the deep interconnectivity and logical architecture that empower the bot's functionality and operational efficiency.

Project Requirements and Use Cases: Initially, we revisited the bot's foundational requirements, emphasizing practical scenarios it needs to handle, which were illustrated through comprehensive UML diagrams and descriptions. These use cases not only demonstrated the bot's responsiveness but also its capability to handle diverse tasks efficiently.

Architectural Design: The architecture section delineated the multi-layered setup of the system, ensuring scalability, reliability, and manageability. By separating concerns across distinct layers—from presentation and business logic to data access—the system's architecture promises enhanced maintainability and easier future expansions.

Detailed Design: In the design segment, we delved into the specifics of each system component. The UML class diagrams provided a clear visualization of the system's structure, showcasing the relationships and responsibilities across various objects within the bot's framework.

Data Management Strategy: Transitioning from a traditional database to a file-based storage system, the project adopts an innovative approach to handle data, which aligns with the dynamic requirements of the Discord environment. This strategy ensures flexibility, rapid data access, and simplifies the system's scalability.

Technology Stack: The chapter also outlined the technology stack that underpins the bot's functionality, highlighting the synergy between various tools and platforms that streamline

development and enhance the bot's performance.

As we conclude this chapter, it is evident that the design and architectural decisions made throughout the project are in perfect alignment with the initial goals—creating a responsive, efficient, and scalable bot. Chapter 4 will continue this narrative by focusing on the implementation details, testing strategies using unittest and mock, and how these elements contribute to the robustness of the bot.