**Discord Bot Automation Assistant**

**Discord Bot Automation Assistant Chapter 3**

**Oguz Kaan Yildirim**

**307637**

# TABLE OF CONTENTS

[TABLE OF CONTENTS 2](#_Toc179120250)

[CHAPTER THREE: PROJECT ORGANIZATION/STRUCTURE 2](#_Toc179120251)

[1. Project Requirements (CISC695\_Assignment2) 3](#_Toc179120252)

[1.1 Actors: 3](#_Toc179120253)

[1.2 Use Cases: 3](#_Toc179120254)

[1.2.1 Stop Bot (!stop\_bot) 3](#_Toc179120255)

[1.2.2 Project Help (!project\_help) 4](#_Toc179120256)

[1.2.3 Navigate to Website (!navigate\_to\_website) 4](#_Toc179120257)

[1.2.4 Close Browser (!close\_browser) 5](#_Toc179120258)

[1.2.5 Login to a Website (!login) 5](#_Toc179120259)

[1.2.6 Receive Email (!receive\_email) 6](#_Toc179120260)

# CHAPTER THREE: PROJECT ORGANIZATION/STRUCTURE

This chapter covers the system design and implementation details of the Discord Bot Automation Assistant. We will start with the project requirements, including the use case diagram and descriptions of the use cases. Then, we will present the architecture, including UML component, deployment, and activity diagrams. The design section will include UML package and class diagrams. Finally, we will discuss the technology stack and frameworks used, followed by a short conclusion.

## 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.

### 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.

### Use Cases:

#### 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.

#### 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.

#### 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.

#### 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.

#### 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.

#### 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.

#### 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.

#### 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.

#### 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.

#### 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.

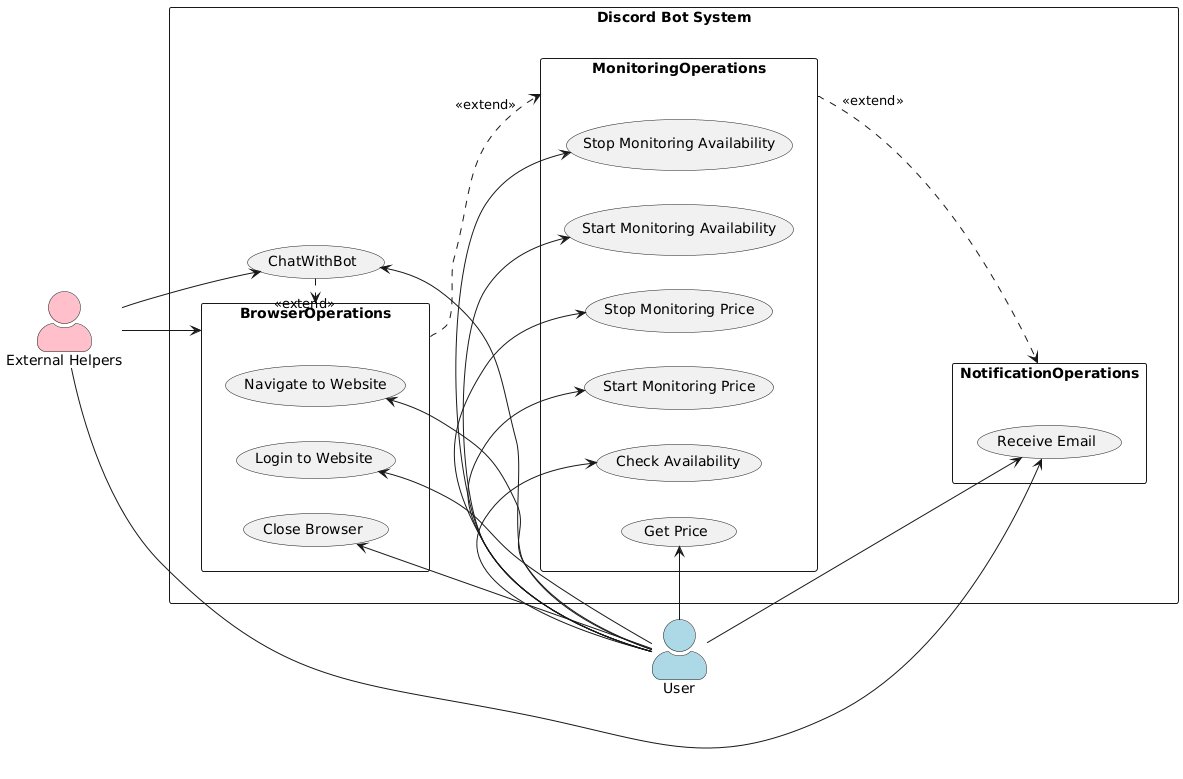
#### Start Monitoring Availability (!start\_monitoring\_availability)

* **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.

#### 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 final results are reported to the user.

### UML use case diagram



## Architecture (CISC695\_Assignment5 and CISC695\_Assignment10)

**Data Flow and Interaction**

When a user types a command in Discord, the message is captured by the appropriate Boundary object. The command is parsed and sent to the relevant Control object. The Control then processes the command, interacting with the Entity objects if data retrieval or manipulation is needed. After processing, the results are sent back through the control to the boundary, which then communicates the outcome to the user on Discord

### BCE objects

**Boundary Objects**

Boundary objects in your project serve as the interface between the user (through Discord) and the application's internal workings. They are responsible for capturing user inputs, parsing these inputs, and delegating tasks to the Control layer for further processing. Here are the primary Boundary objects:

#### BotBoundary

* + Captures system-related commands such as stopping the bot, fetching help information, or sending an email.
  + Utilizes the GlobalState class to parse user messages, extracting commands and potential parameters which are then packaged and sent to the BotControl.
  + Acts as the first line of interaction for commands, managing error handling directly from user inputs and responding with appropriate feedback or results.

#### BrowserBoundary

* + Handles all user commands that involve web browser interactions such as launching a browser, navigating to specific URLs, or executing login operations.
  + Similar to other boundaries, it parses the user's command using GlobalState to determine the exact browser action required.
  + Transfers these commands to BrowserControl, which then executes the detailed operations using BrowserEntity.

#### AvailabilityBoundary

* + Specifically manages availability checking and monitoring commands.
  + Parses user commands for details like URL, date, and monitoring frequency, then forwards this information to AvailabilityControl.
  + The boundary ensures that user inputs are converted into actionable data that the control layer can use to interact with external websites for availability data.

#### PriceBoundary

* + Dedicated to handling price-related inquiries and monitoring from users.
  + Extracts important parameters such as product URL and frequency of price checks from the command, then sends these to PriceControl.
  + Provides immediate feedback to users about price fetching status or monitoring initiation based on control responses.

**Control Objects**

Control objects take commands from Boundary objects, decide on the business logic to be applied, and interact with Entity objects to manipulate application data or state. They play a pivotal role in ensuring the correct execution of user commands.

#### BotControl

* + Processes and responds to systemic commands like stopping the bot or displaying help.
  + Interacts with Entity objects as needed for operations that require data retrieval or system manipulation, such as fetching account details or managing user sessions.
  + Directly manages the lifecycle of the bot based on commands like stop, ensuring that all activities are gracefully terminated.

#### BrowserControl

* + Manages all commands related to web browsers.
  + Decides how to execute user commands to navigate, login, or close web browsers.
  + Uses the BrowserEntity to interact with the web directly, maintaining state like open browser sessions, navigating pages, or entering login details.

#### AvailabilityControl

* + Processes commands related to availability checking and monitoring.
  + Determines whether to start or stop monitoring based on user input and system state, handling frequency settings, and monitoring intervals.
  + Utilizes AvailabilityEntity to perform the actual checks on external websites and manage continuous monitoring tasks.

#### PriceControl

* + Handles commands related to price fetching and monitoring.
  + Decides on the frequency of price updates, initiates monitoring tasks, and responds to stop commands.
  + Communicates with PriceEntity to fetch current prices from specified URLs and to manage ongoing price monitoring tasks.

**Entity Objects**

Entity objects are the core data handlers in your system, interacting with external systems and performing the heavy lifting of data retrieval and manipulation.

#### AvailabilityEntity

* + Directly interacts with web services to check availability dates and times for services or products.
  + Manages detailed web scraping logic to retrieve availability data, which is then formatted and returned to AvailabilityControl.
  + Handles the persistence of monitoring results and notifications regarding availability changes.

#### BrowserEntity

* + Manages the actual web browser interface, executing commands like open, close, navigate, and login.
  + Maintains the state of the web browser session, handles errors during navigation or interaction, and ensures data is correctly entered on web forms.

#### PriceEntity

* + Responsible for fetching current price data from web pages.
  + Implements logic to scrape web pages for pricing information, manage errors during data retrieval, and log results for monitoring purposes.

### Identifying subsystems in the project

#### User Interaction Subsystem:

* + **Purpose**: Manages direct interactions with users via Discord, processing commands and delivering responses.
  + **Components**:
    - **UserInterface**: Parses user inputs and commands received through Discord.
    - **CommandDispatcher**: Directs commands to the appropriate subsystems based on the command type and user input.

#### Authentication and Configuration Subsystem:

* + **Purpose**: Manages access and configuration settings to ensure secure and customized use of the bot.
  + **Components**:
    - **ConfigManager**: Retrieves and manages configuration settings from the system such as Discord tokens, channel IDs, and email settings.

#### Price and Availability Monitoring Subsystem:

* + **Purpose**: Handles monitoring and retrieval of data regarding prices and availability.
  + **Components**:
    - **PriceControl**: Checks and monitors prices.
    - **AvailabilityControl**: Checks and monitors availability.

#### Notification and Messaging Subsystem:

* + **Purpose**: Manages notifications and email communications with users, informing them about updates or changes based on their subscriptions or commands.
  + **Components**:
    - **EmailControl**: Handles the preparation and sending of emails with attachments or notifications.
    - **NotificationManager**: Generates and sends other types of notifications to users through the Discord interface.

#### Data Management and Reporting Subsystem:

* + **Purpose**: Manages logging, storage, and reporting of operational data and user activities.
  + **Components**:
    - **ReportGenerator**: Provides functionalities to export data into user-friendly formats such as Excel or HTML for reports.

### Identification of a Design Goal as a Nonfunctional Requirement

In designing and developing the Discord bot, our primary nonfunctional requirement is **usability**. By prioritizing usability, our system ensures that the bot not only functions effectively but also provide a pleasant and straightforward experience for its users. This focus is critical as the bot is intended to assist users with diverse tasks such as navigating websites, monitoring prices, and checking availability, all through a Discord interface.

### UML component diagram