SOFE 3980U: Software Quality

Final Project Report:

Improving Quality of an Open-Source Software

Date: April 14th, 2022

Group 4

Members:

Foram Gandhi (100699245)

Kinjal Shah (100743551)

Rutvi Shah (100747171)

Krutarth Dave (100730797)

Introduction

The open source software we chose to investigate for improvements was called *MeetBotv9* by GitHub user Pratik Pathak. This software uses *Selenium* with *Python* in order to automate the process of joining Google Meets via a Telegram bot. Our goal was to improve the MeetBot by adding more features and observing its performance by running test cases.

This Google Meet bot is advantageous for students who wish to join scheduled virtual meetings, classes or events on time. It maximizes productivity by reducing the time that gets spent on repetitive tasks such as searching for a meeting link, logging into your Google account and enabling/disabling Google meet features such as a camera and microphone. This way, people do not have to worry about missing important events and can optimize their workflows.

Details of the Project

To activate the automation of the bot, the user is required to obtain their necessary credential from the Telegram Bot API. Then, they can use the Telegram desktop or mobile application to send commands to the bot. The three main functionalities of the bot include the ability of signing into the user's Google account, joining a Google Meet call with the provided link, and providing screenshots of the bot's current status. The available commands are described below:

- /login → Logs in to the Google account
- /meet <link> → Joins a Google Meet call
- /close → Leaves the meeting
- /status → Sends a screenshot of the bot's current status
- /restart → Restarts the Google Meet bot
- /reset → Resets the Chrome browser
- /owner → Displays information about the bot's owner
- /quit → Quits the running script
- /addws <day of the week><time in 24-hour format><link> → Adds a meeting slot into the user's weekly schedule
- /ssch → Starts the schedule by joining any scheduled meetings
- /help → Displays all available commands

Upon initial investigation of the codebase, as well as a walkthrough of the running software, it was revealed that there were some bugs in the code and major areas that needed to be tested. The /restart, /reset and /close commands were not functioning correctly. A commonality that was observed within each function for the commands was that a call to the os.execl() was made in order to restart the program. However, that was not working, so we replaced the function with os.exec() and re-initialized the Chrome webdriver. During the project timeline, there was a new

Chrome update which caused an impediment to the bot functionality and caused it to fail. It was unable to join the Google Meet automatically due to a warning dialogue that was displayed to the user, stating that the camera and microphone are blocked (Fig 1). To handle this issue, the script was modified using *Selenium* modules, which automated the process of clicking the 'Dismiss' button on the dialogue box (Fig 3). The group had also noticed that once the script was running, there was no functionality for quitting it, even if the 'CTRL + C' interrupt on the keyboard was used. Thus, the /quit command was implemented to properly quit the script.

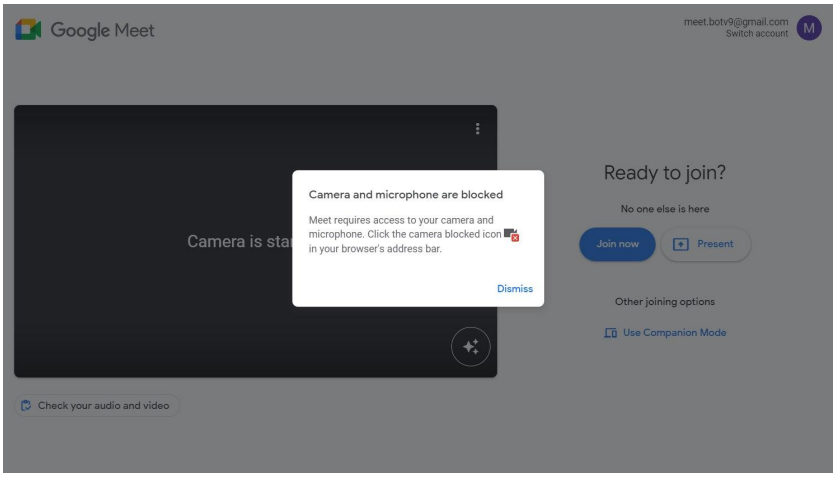

Fig 1. Warning dialogue generated due to Chrome update

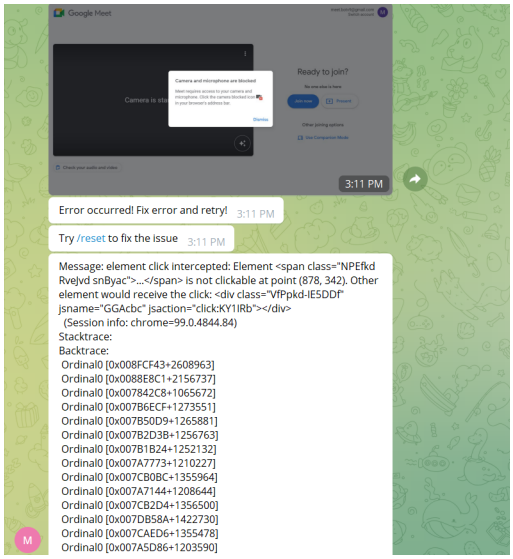

Fig 2. Error messages sent by bot

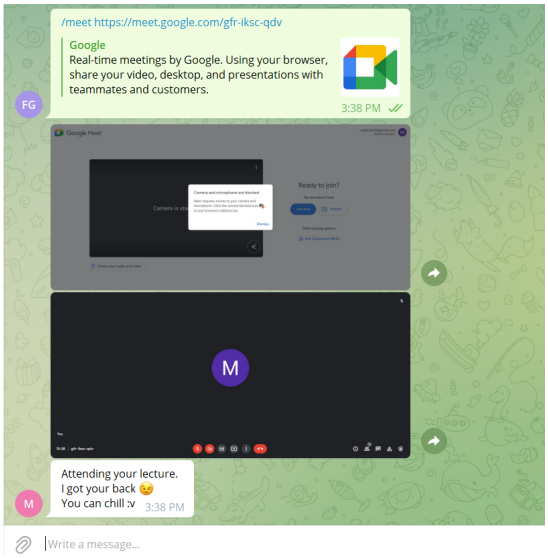

Fig 3. Fixed errors with *Selenium*

There were two new features that were added to the project, in addition to the fixed bugs. The first feature was the quit command, as mentioned previously, and the second was the scheduler. The scheduler consisted of two functionalities, which were the ability to add weekly scheduled Meets (/addws) and start the scheduler session (/ssch). When the first command is invoked, the

user's weekly meeting information gets stored in a Python dictionary, which is then saved to a file for future use. Input validation was also conducted to prevent errors and ensure that the user was using the correct format for the day, time and meeting link entered. The bot responded with the respective error messages and instructions on how to correct them. A few examples can be seen in the image below:

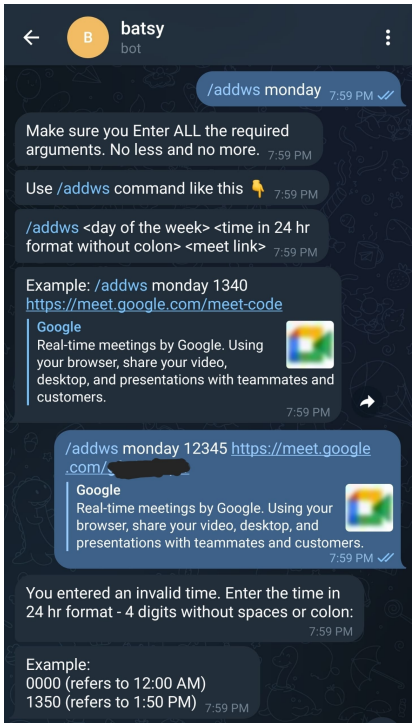

Fig 4. Error messages for invalid inputs

Fig 5. Successful addition of meeting to personal schedule

The second command was implemented using the Python *schedule* library. When called, the bot automatically joins the user's scheduled meeting at the requested time and proceeds with the usual process of providing status screenshots and success/fail messages back to the user.

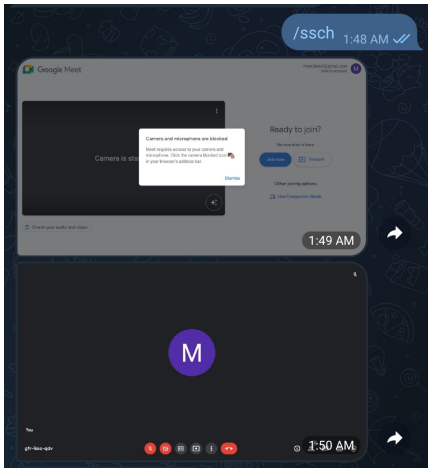

Fig 6. Scheduler session is started and status screenshots are sent to user

Design Choices

Testing was an important aspect of this project, as it allowed for the assessment of the software's quality. Actual results were compared to the expected results and bugs/errors were dealt with accordingly. Python's unittest framework, specifically the mock object library, was utilized for this phase. It allowed for the emulation of a real object within the testing environment with the help of assertions. This was the only way for us to test the response functionality of the commands sent to the bot from the user-end. In order to apply assertions, return statements would be required in each function that is being tested, either returning 0 or 1. Initially, the functions were not returning anything (void), therefore we added a return 1 statement to indicate a passing test and return 0 statement to indicate failure.

```
def status(update, context):
    user = update.message.from_user
    if user.id == int(USER_ID):
        chromium_Scripts.browser.save_screenshot("snapshot.png")
        f = open("snapshot.png", "rb")
        context.bot.send_chat_action(
            chat_id=USER_ID, action=ChatAction.UPLOAD_PHOTO)
        context.bot.send_photo(
            chat_id=USER_ID, photo=f, timeout=100
        )
        try:
            os.remove("snapshot.png")
        except:
            f.close()
            os.remove("snapshot.png")
        return 1
    else:
        telegram_bot_sendtext("You are not authorized to use this bot.\nUse /owner to know about me")
        return 0
```

Fig 7. Code for status command with added return statements

```
def test_status(self):
    mocked_update.message.from_user.id = int(user_id)
    mocked_update.message.text = "/status"

    self.assertEqual(status(mocked_update, mocked_context), 1)

    mocked_update.message.from_user.id = 0
    self.assertEqual(status(mocked_update, mocked_context), 0)
```

Fig 8. Test for status command with mock library

The stated functions were tested in a similar way: test_setup(), test_sendText(), test_start(), test_help(), test_owner(), test_retart() and test_reset().

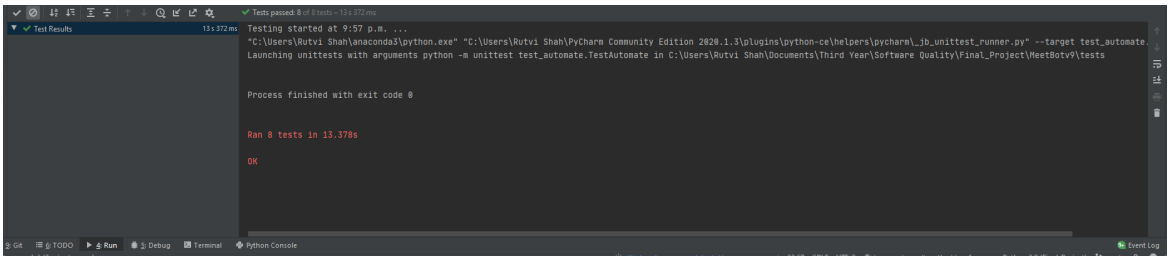

Fig 9. Passing test cases for all functions

The test_login() function was used to test a valid login, according to the user-specified Google account credentials. After a successful or failed login, the new login attempt was tested. Finally, the user authorization was tested to ensure that a valid user ID was entered. While logging in, the bot would click the dismiss button on the dialogue box, which was verified by outputting success messages to the console.

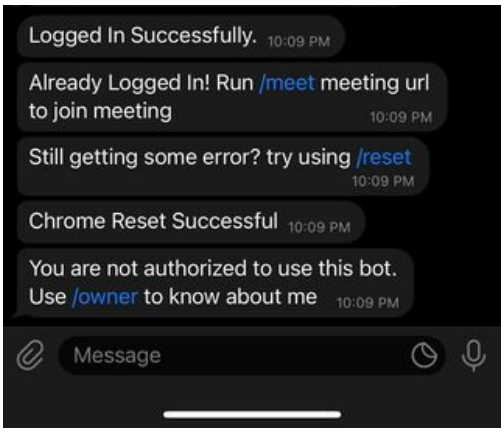

Fig 10. Messages from the bot for each test, respectively

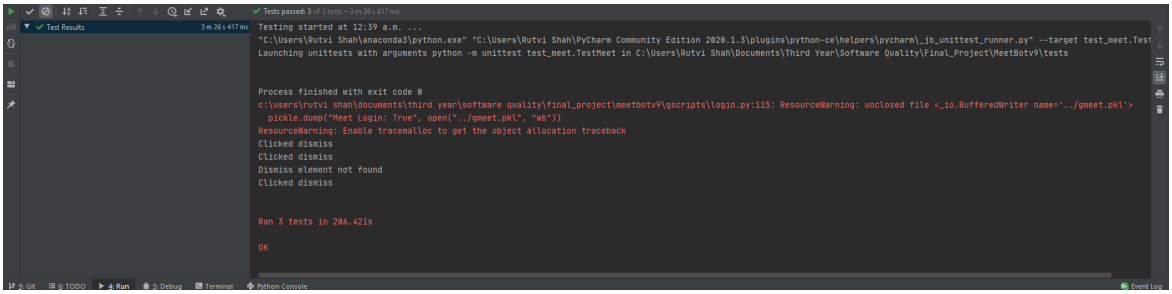

Fig 11. Passing meet test cases with dialogue box messages

The test_meet(), test_meet_url() and test_close() functions were implemented to verify the validity of the user ID and meet link, as well as the close meeting functionality. The success/fail messages for each can be seen in the images below:

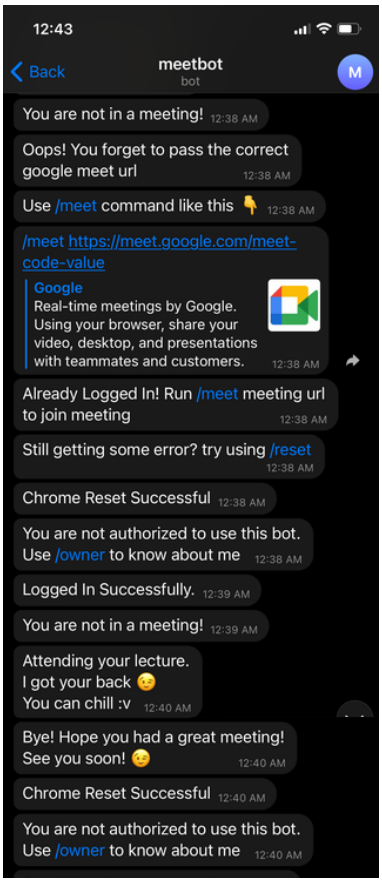

Fig 12. Output for TestMeet class

Testing the scheduler functionality included the methods test_addws() and test_ssch(). Refer to Fig 4. for the message outputs from the bot. The image below shows the passing test cases for these two functions and the new dictionary every time the user adds a new meeting to their schedule.

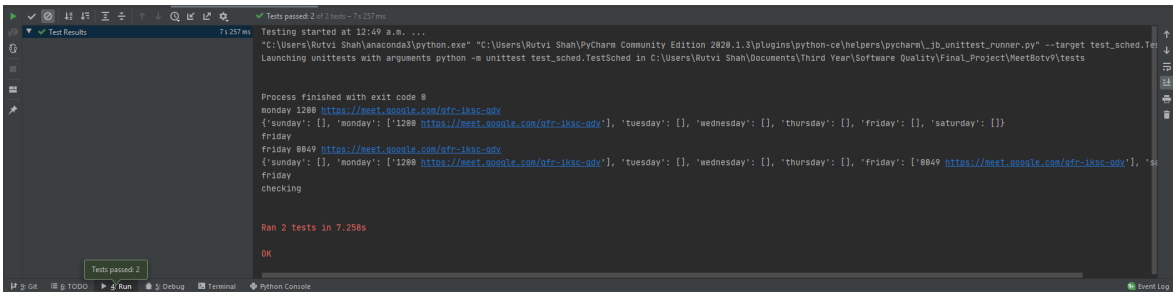

Fig 13. Passing scheduler test cases with updated dictionaries

Implementation Challenges

When working with unfamiliar code, there can be a few challenges that can emerge during the development and testing phases. One of the implementation challenges we encountered was during testing, as the Telethon library was not working properly. This library helps test applications that use Telegram. The main issue with implementing it, however, was that there was very limited documentation online and other issues that surfaced during the integration process (Fig 14.).

```
                r = functions.InvokeWithoutUpdatesRequest(r)

        request_index = 0
        last_error = None
        self._last_request = time.time()

        for attempt in retry_range(self._request_retries):
            try:
                future = sender.send(request, ordered=ordered)
                if isinstance(future, list):
                    results = []
                    exceptions = []
                    for f in future:
                        try:
                            result = await f
                        except RPCError as e:
                            exceptions.append(e)
                            results.append(None)
                            continue
                        self.session.process_entities(result)
                        self._entity_cache.add(result)
                        exceptions.append(None)
                        results.append(result)
                        request_index += 1
                    if any(x is not None for x in exceptions):
                        raise MultiError(exceptions, results, requests)
                    else:
                        return results
                else:
>                   result = await future
E                   telethon.errors.rpcerrorlist.AuthKeyUnregisteredError: The key is not registered in the system (caused by GetDialogsRequest)

..\..\..\..\..\..\anaconda3\lib\site-packages\telethon\client\users.py:84: AuthKeyUnregisteredError

Assertion failed

Assertion failed

Assertion failed
```

Fig 14. Shows the telethon test failing

To overcome this challenge, we opted to use Python unittest's *Mock* library instead. *Mock* is a unit testing framework that reinstates segments of the code with Mock objects so that the behavior of the bot can be mimicked. This framework was used to automate tests for multiple functions. Another challenge we faced was the Google update during the timeline of our project, which caused some errors in the prior functionality. One of the errors was that the auto-join meet function failed due to a Chrome warning dialog (Fig 15.).

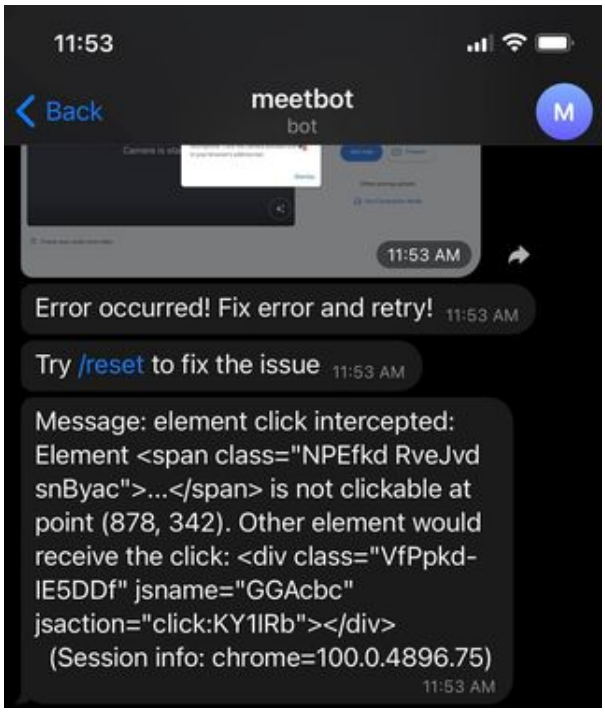

Fig 15. Join button cannot be found due to Dismiss button prompt, causing an error

This challenge was overcome by modifying the scripts using Selenium, a browser automation support tool and library. Selenium modules were used to handle the use cases (Fig 16.).

```
# Due to recent Google Chrome update, a "pop up dialogue" shows up if the user does not have cam/mic permissions
# This was causing the script to crash because the button to join the meet was being rendered unclickable
# if the dismiss prompt is found, clicks on the dismiss button, otherwise clicks the join button
try:
    print("Clicked dismiss")
    chromium_Scripts.browser.find_element_by_xpath("//*[@id='yDmH0d']/div[3]/div[2]/div/div[2]/button").click()
    time.sleep(10)
except:
    print("Dismiss element not found")
    time.sleep(10)
```

Fig 16. Modified scripts using Selenium

Another challenge we experienced was that when experimenting with exec() functions to fix bugs, other commands stopped working. The main problem with the exec() function was that it

did not terminate the old processes correctly before new ones were created during the next run of the script, so more than one process ended up using a single bot token, which is not allowed. This prompted an error stating, “Your bot token is being used in more than one instance” (Fig 17.). The commands that stopped functioning properly and became buggy were “/reset”, “/restart”, and “/close”, since they all made calls to exec(). As a result, we had to keep generating new bot tokens to test whether the functionality was working.

Fig 17. Error messages regarding more than one instance of bot running

This challenge was overcome by creating a new bot and reinitializing the Chrome webdriver (Fig. 18).

```
# To do, send a message to the user when the bot is restarted : Finished
def restart(update, context):
    user = update.message.from_user
    if user.id == int(USER_ID):
        telegram_bot_sendtext("Restarting, Please wait!")
        r = open("restart.pkl", "wb")
        pickle.dump("restart msg check", r)
        r.close()
        chromium_Scripts.browser.quit()

        if os.path.exists("restart.pkl"):
            try:
                os.remove("restart.pkl")
                telegram_bot_sendtext("Bot Restarted")
            except:
                pass
        chromium_Scripts.browser = webdriver.Chrome(options=options)  # restart the chrome window
        return 1
    else:
        telegram_bot_sendtext("You are not authorized to use this bot.\nUse /owner to know about me")
        return 0
```

Fig 18. Re-initialization of the Chrome webdriver

The final challenge we encountered during the implementation was attempting to add a 'Record Call' feature. After writing the code for the feature, we noticed that it was not working as intended. Upon analysis of the situation, we found that the problem was not with our code, instead it was due to the Google policy which required users to have a Premium account to record calls.

Lessons Learned

Over the duration of the project, we received the opportunity to utilize and implement many different tools and softwares we had never encountered before. Some tools include: the *Telegram* library, the *Telethon* library and lastly the unittest *Mock* library. The *Telegram* library was incorporated in the project to allow for the sending and receiving of messages from the bot in the Telegram application. Furthermore, we attempted to integrate the *Telethon* library into the project as it would automate the bot's actions by automatically sending and validating messages from the Telegram application. However, we ran into many issues in the implementation of it so we decided to use the *Mock* and *MagicMock* object libraries in the *unittest* framework. These libraries were used for testing in Python, and allowed us to create *Mock* objects which imitated the bot's responses to requests made within the testing environment, with the help of assertions. Overall, we learned how to become familiar with new software written by other developers and make meaningful contributions to it to improve its quality.

Conclusion

In conclusion, we achieved our goal to improve the MeetBotv9 software by adding features such as the *quit* command, which allows the user to terminate the program with ease, and the scheduler (*addws*, *ssch*), which provided an autonomous method of joining meetings based on a personalized schedule. Furthermore, we improved the quality of the software by adding quality assurance measures such as unit testing to gauge the accuracy of the program. During the testing process, we started off by testing the pre-existing functions that were already made by the author of the repository and uncovered some test failures and bugs in the program, such as the /restart, /reset and /close functions. They were not functioning as intended because of an exec() function call within each method. Additionally, there were errors in the *meet* command as it failed to auto-join the meet due to a recent Chrome update. After the observation of all these errors, we were able to fix them, which evidently improved the quality of the software. We also conducted testing on the added commands mentioned previously, such as *quit*, *addws* and *ssch*. This testing process ensured that the newly-implemented features did not have unintended consequences on the pre-existing functionalities/commands. Therefore, testing ensured the quality of the software as it uncovered inadvertent behaviors in the program and verified that all software requirements were met. We hope that our contributions to this software can elevate the experience of users and help the open-source community.