**Discord Bot Automation Assistant**

**Discord Bot Automation Assistant Defects**

**Oguz Kaan Yildirim**

**307637**

# Table Of Contents

[Table Of Contents 2](#_Toc178785359)

[INTRDOCUTION 3](#_Toc178785360)

[DEFECTS 4](#_Toc178785361)

[**Defect 1** 4](#_Toc178785362)

[Description 4](#_Toc178785363)

[Possible Causes 4](#_Toc178785364)

[Repair Method 4](#_Toc178785365)

[Screenshot of Defect 4](#_Toc178785366)

[**Defect 2** 5](#_Toc178785367)

[Description 5](#_Toc178785368)

[Possible Causes 5](#_Toc178785369)

[Repair Method 6](#_Toc178785370)

[Screenshot of Defect 6](#_Toc178785371)

[**Defect 3** 7](#_Toc178785372)

[Description 7](#_Toc178785373)

[Possible Causes 7](#_Toc178785374)

[Repair Method 7](#_Toc178785375)

[Screenshot of Defect 8](#_Toc178785376)

# INTRDOCUTION

This documentation presents an overview of the defects encountered during the development of the Discord Bot Automation Assistant. Over the past 4-5 weeks, the focus has been on building and refining unit tests to validate various components of the project. The process has not been without challenges, and several defects were identified along the way. Although exact dates for when these defects were discovered and resolved are unclear, they were primarily addressed during the month of September 2024 as part of the ongoing testing and debugging efforts.

The purpose of this documentation is to provide a detailed account of the defects encountered, their possible causes, the repair methods used to resolve them, and any relevant screenshots that illustrate the issues. Each defect is assigned a unique ID, and the description includes a thorough explanation of the problem, the root cause, and how the issue was fixed.

Each defect is categorized and described in detail, including the problems encountered, their underlying causes, and the steps taken to repair them. This documentation aims to provide a comprehensive view of the defect resolution process and the challenges involved in developing robust, maintainable code for the Discord Bot Automation Assistant.

The following sections provide in-depth descriptions of the defects, including:

1. **Defect IDs** – A unique identifier for each defect.
2. **Defect Names** – A descriptive name summarizing the issue.
3. **Date Repaired/Documented** – The approximate date when the defect was addressed.
4. **Description** – An explanation of the problem and how it manifested in the project.
5. **Possible Causes** – An analysis of the potential causes leading to the defect.
6. **Repair Methods** – The steps taken to resolve the defect, including code changes and modifications to the testing environment.
7. **Screenshots** – Visual representations of the errors or warnings encountered during the testing process.

By documenting these defects, the aim is to provide insight into the complexities of software development and testing, ensuring that future development efforts are better informed and more efficient.

# DEFECTS

## **Defect 1**

**Defect ID:** DEF01

**Defect Name:** ImportError

**Date Repaired/Documented:** September, 2024

### Description

The unit test for the AccountEntity class fails due to an ImportError. The test file is unable to locate and import the AccountEntity class because the folder structure causes incorrect module paths. Without the proper path configuration, the module cannot be recognized by the test script. This happened in all test cases, and it is not only specific to AccountEntity class.

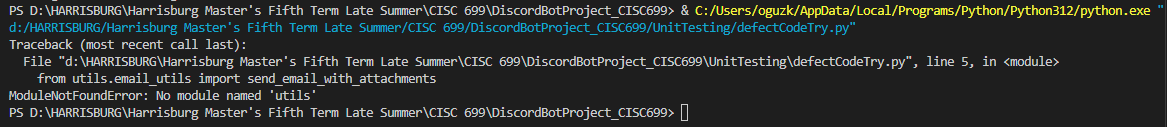
### Possible Causes

* Incorrect folder structure leading to broken module imports.
* Missing or misconfigured sys.path.append() to adjust Python’s path.

### Repair Method

The issue was resolved by adding the following line to the test script:  
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(\_\_file\_\_))))  
This line adjusts the Python path so that any module can be correctly imported into the test file.

### Screenshot of Defect



## **Defect 2**

**Defect ID:** DEF02

**Defect Name:** unittest Async Method Handling Issue

**Date Repaired/Documented:** September, 2024

### Description

During the testing of asynchronous functions in the DiscordBotProject\_CISC699, two tests related to monitoring availability failed when executed with unittest. The primary issue arose because unittest is not designed to handle async def functions natively. This resulted in runtime warnings and deprecation warnings, with the async coroutines being marked as "never awaited" during the execution of the tests.

When running the unittest framework, the following warnings were triggered:

* **RuntimeWarning**: coroutine 'TestAvailabilityControl.test\_start\_monitoring\_availability\_success' was never awaited
* **DeprecationWarning**: It is deprecated to return a value that is not None from a test case.

Despite these warnings, the tests appeared to complete successfully, but they did not actually execute the asynchronous logic as intended. This led to false positives, as the underlying issues in the async methods went undetected.

### Possible Causes

The root cause of the defect was the inherent limitation of unittest when dealing with asynchronous functions. The unittest framework expects synchronous test cases, and when it encounters async def functions, it does not properly handle them, resulting in the warnings:

* **RuntimeWarning**: This occurs because the async functions were not awaited, meaning the event loop was never properly triggered, and the coroutine was essentially skipped.
* **DeprecationWarning**: This was raised because unittest expects test cases to return None. However, since async functions were involved, the coroutines were returning non-None values that unittest could not handle correctly.

The key problem is that unittest lacks the capability to handle event loops and asynchronous code execution, leading to incomplete or skipped tests when working with async def functions.

### Repair Method

To resolve this issue, the testing framework was switched from unittest to pytest, which natively supports asynchronous functions via the pytest-asyncio plugin. This switch allowed for proper handling of the async methods, ensuring that the event loop is managed correctly and that the asynchronous code is fully executed during tests.

### Screenshot of Defect

A black screen with many colorful lines

Description automatically generated with medium confidence

## **Defect 3**

**Defect ID:** DEF03

**Defect Name:** Missing pytest Fixture Decorator

**Date Repaired/Documented:** September, 2024

### Description

This defect occurred due to the omission of the @pytest.fixture decorator from the base\_test\_case fixture in the test\_init.py file. The base\_test\_case fixture was responsible for initializing various control and entity objects needed by the test cases. However, without the @pytest.fixture decorator, the fixture could not be detected and injected into the test functions, leading to errors during test execution.

When the tests were run, pytest was unable to recognize base\_test\_case as a valid fixture, resulting in the following error:

“fixture 'base\_test\_case' not found”

This caused any test (but it’s been discovered in test\_start\_monitoring\_price\_already\_running and test\_start\_monitoring\_price\_failure\_in\_entity) to fail because they were attempting to access uninitialized objects, such as base\_test\_case.price\_control.

The missing decorator prevented the proper setup of the test environment, leading to runtime failures and unhandled exceptions.

### Possible Causes

The root cause of the defect was the omission of the @pytest.fixture decorator in the fixture definition. As a result, pytest did not recognize base\_test\_case as a fixture, and the test functions could not receive the necessary initialization data. Without the fixture, the test functions attempted to access uninitialized objects, causing AttributeError and fixture-not-found errors.

### Repair Method

To resolve this issue, I initially thought we could simply call the base\_test\_case method directly, but since it is part of the test setup, we need to use the @pytest.fixture decorator. This decorator connects the method to the pytest framework, allowing it to automatically detect and inject the fixture into the test functions, ensuring that all necessary objects are initialized before the tests run.

Added the @pytest.fixture decorator: This decorator was applied to the base\_test\_case method to properly define it as a fixture that can be used across multiple test cases.

Once the @pytest.fixture decorator was added to the base\_test\_case function, the tests ran as expected, with the necessary objects being initialized before execution. This allowed the test cases to properly access and manipulate the price\_control and other controls during testing.

### Screenshot of Defect

A screen shot of a computer

Description automatically generated