**Automated Discord Bot Helper**

**Automated Discord Bot Helper Test Plan**

**By**

**Oguz Kaan Yildirim**

**M.S. Harrisburg University of Science and Technology**

# Table Of Contents

[Table Of Contents 2](#_Toc178265949)

[INTRDOCUTION 5](#_Toc178265950)

[TEST PLAN OVERVIEW 6](#_Toc178265951)

[TEST CASES 8](#_Toc178265952)

[Test Case 0: test\_init.py 8](#_Toc178265953)

[Description: 8](#_Toc178265954)

[Test Case 1: Add a Book 10](#_Toc178265955)

[Description: 10](#_Toc178265956)

[Steps: 10](#_Toc178265957)

[Test Data: 10](#_Toc178265958)

[Outcomes: 12](#_Toc178265959)

[Defect: 12](#_Toc178265960)

[Source Code: 12](#_Toc178265961)

[Test Case 2: Search for a Book 15](#_Toc178265962)

[Description: 15](#_Toc178265963)

[Steps: 15](#_Toc178265964)

[Test Data: 15](#_Toc178265965)

[Outcomes: 15](#_Toc178265966)

[Defect: 16](#_Toc178265967)

[Source Code: 17](#_Toc178265968)

[Test Case 3: Check Book Availability 18](#_Toc178265969)

[Description: 18](#_Toc178265970)

[Steps: 18](#_Toc178265971)

[Test Data: 18](#_Toc178265972)

[Outcomes: 18](#_Toc178265973)

[Defect: 20](#_Toc178265974)

[Source Code: 20](#_Toc178265975)

[Test Case 4: User Fines Payment 21](#_Toc178265976)

[Description: 21](#_Toc178265977)

[Steps: 21](#_Toc178265978)

[Test Data: 21](#_Toc178265979)

[Outcomes: 21](#_Toc178265980)

[Defect: 22](#_Toc178265981)

[Source Code: 23](#_Toc178265982)

[Test Case 5: User Borrowing Limit 24](#_Toc178265983)

[Description: 24](#_Toc178265984)

[Steps: 24](#_Toc178265985)

[Test Data: 24](#_Toc178265986)

[Outcomes: 24](#_Toc178265987)

[Defect: 24](#_Toc178265988)

[Source Code: 25](#_Toc178265989)

[Test Case 6: Place Hold Form 26](#_Toc178265990)

[Description: 26](#_Toc178265991)

[Steps: 26](#_Toc178265992)

[Test Data: 26](#_Toc178265993)

[Outcomes: 26](#_Toc178265994)

[Defect: 26](#_Toc178265995)

[Source Code: 27](#_Toc178265996)

[Test Case 7: Return Book Form 28](#_Toc178265997)

[Description: 28](#_Toc178265998)

[Steps: 28](#_Toc178265999)

[Test Data: 29](#_Toc178266000)

[Outcomes: 30](#_Toc178266001)

[Defect: 30](#_Toc178266002)

[Source Code: 30](#_Toc178266003)

[Test Case 8: Add and Retrieve Book 32](#_Toc178266004)

[Description: 32](#_Toc178266005)

[Steps: 32](#_Toc178266006)

[Test Data: 32](#_Toc178266007)

[Outcomes: 32](#_Toc178266008)

[Defect: 32](#_Toc178266009)

[Source Code: 32](#_Toc178266010)

[Test Case 9: Check Book Existence 34](#_Toc178266011)

[Description: 34](#_Toc178266012)

[Steps: 34](#_Toc178266013)

[Test Data: 34](#_Toc178266014)

[Outcomes: 35](#_Toc178266015)

[Defect: 36](#_Toc178266016)

[Source Code: 36](#_Toc178266017)

[Test Case 10: Duplicate Book Entry 37](#_Toc178266018)

[Description: 37](#_Toc178266019)

[Steps: 37](#_Toc178266020)

[Test Data: 37](#_Toc178266021)

[Outcomes: 37](#_Toc178266022)

[Defect: 37](#_Toc178266023)

[Source Code: 37](#_Toc178266024)

# INTRDOCUTION

The purpose of this document is to outline a comprehensive test plan for the "Automated Discord Bot Helper" project. This test plan is meticulously designed to ensure the robustness, correctness, and functionality of all components involved in the project, emphasizing the meticulous validation of each unit through structured test cases. The plan incorporates tests for various object types involved in the project—boundary, control, and entity—alongside additional focus on integration points, without accessing real databases or external systems, adhering strictly to unit testing principles using mock objects and fake data.

This document will guide the systematic testing of individual components and their interactions within the system, ensuring that all functionalities meet the specified requirements and behave as expected in various scenarios, both typical and atypical. Each test is crafted to validate specific elements of the system, from the fundamental logic handled by entity and control objects to the data flow managed by boundary objects interfacing with the user.

To have a better look at the code/project especially to testing source codes; codes can be found in github. Test files are under UnitTesting Folder.

https://github.com/oguzky7/DiscordBotProject\_CISC699/tree/develop/UnitTesting

# TEST PLAN OVERVIEW

The test plan is constructed to systematically validate the performance and reliability of the "Automated Discord Bot Helper," ensuring that each component not only operates in isolation but also performs optimally within the system's ecosystem. The plan is segmented into several suites, each targeting specific components:

**Entity Objects Testing**: Focuses on ensuring that each entity object maintains integrity, correctly manages state, and interacts flawlessly with other components. Tests will include creating, manipulating, and validating state changes within these objects.

**Control Objects Testing**: Aims to verify that control objects accurately orchestrate the flow of data between the user interfaces (boundary objects) and the data management layers. This includes testing the logical conditions and workflows that control objects are responsible for.

**Boundary Objects Testing**: Tests the interfaces that interact with the system users, ensuring data is correctly captured, validated, and passed to the underlying control layers. This suite ensures that all user inputs are handled correctly, simulating various user interaction scenarios.

**Integration of Components**: Although primarily focusing on unit testing, the plan includes a series of tests designed to ensure that components work together as expected under controlled conditions using mocks and stubs instead of real data connections. This approach adheres to the unit testing philosophy while ensuring that interactions between components are tested without crossing into full integration testing.

**Mock and Fake Implementation**: Critical to avoid direct database interactions or file system accesses, mock objects and fakes will be used extensively to simulate the external dependencies, ensuring that the tests remain fast, reliable, and repeatable. This approach allows for the testing of error handling and edge cases without the overhead of a live environment.

Each test case described in this plan will outline the expected behavior, the steps to execute the test, the mock or fake data involved, and the anticipated outcomes, ensuring comprehensive coverage of all functionalities. This methodical approach ensures that all aspects of the "Automated Discord Bot Helper" are rigorously tested, thereby minimizing the risk of defects and ensuring a high-quality software product.

By adhering to these guidelines, the test plan aims to validate the functionality thoroughly and reliability of the system, ensuring that it meets all specified requirements and is robust against potential errors or failures.

To have a better look at the code/project especially to testing source codes; codes can be found in github. Test files are under UnitTesting Folder.

https://github.com/oguzky7/DiscordBotProject\_CISC699/tree/develop/UnitTesting

# TEST CASES

## Test Case 0: test\_init.py

### Description:

**Tools and Technologies**

This test initialization setup involves a suite of tools designed to facilitate comprehensive unit testing of the "Discord Service Notifier" project. Key tools include:

* **Python**: The primary programming language used for developing both the application and the test cases.
* **unittest**: A built-in Python framework for constructing and running tests, offering capabilities to setup, execute, and teardown tests.
* **unittest.mock**: Provides a core Mock class removing the need for dependencies during testing. This is crucial for simulating the behavior of complex objects in a controlled environment.
* **AsyncMock**: A subclass of unittest.mock's Mock, designed to test asynchronous functions.
* **discord.py**: A Python library for interacting with Discord, mocked in our tests to simulate interactions without real server connections.
* **CustomTextTestRunner**: An extension of unittest’s TextTestRunner that is tailored to provide customized output formats for test results, enhancing readability and diagnostics.

**Purpose and Setup**

The test\_init.py file lays the foundational framework for all other test scripts in the project. It is designed to centralize common setup and teardown processes that are essential across multiple test cases, ensuring consistency and reducing redundancy in the testing codebase.

**Implementation Details**

* **Mocking Discord Interactions**: Given that the project interfaces significantly with the Discord API, discord.py is used extensively. However, in the context of unit testing, direct calls to Discord's servers are impractical and potentially disruptive. Instead, the AsyncMock tool is utilized to simulate these interactions. This allows the tests to mimic the behavior of the bot (e.g., sending messages, handling commands) without actual network operations.
* **Common Test Setup**: The BaseTestSetup class provided within test\_init.py is used across all test scripts to establish a consistent environment for each test. This setup includes configuring a mock version of the Discord bot and a testing framework capable of handling asynchronous calls typical in a Discord bot environment.
* **Mocking and Patching**: The use of unittest.mock.patch is pivotal in controlling the scope and impact of external dependencies within tests. By replacing parts of the system under test with mock objects, it ensures that tests run in isolation, thereby increasing their reliability and speed. For instance, patching external API calls to return predetermined responses allows us to test how our system reacts to various external stimuli.

**How It Works:**

* When a test case is executed, test\_init.py configures the necessary environment by setting up paths and initializing mock objects. This allows each test script to operate independently of the live Discord environment.
* The AsyncMock setup is particularly important for simulating asynchronous methods that interact with the Discord API, such as sending messages or processing commands. Each command's effect is simulated, and its impact is assessed within a controlled test scenario, ensuring the bot reacts as expected in various situations without actual side effects.
* The centralized setup ensures that all test cases start with a consistent, pre-configured environment, minimizing setup duplication across tests and ensuring that any changes to the testing environment need only be made in one place.

This comprehensive setup is not just about ensuring that the bot functions as expected; it's also about ensuring that it does so in a way that is isolated from real-world side effects, consistent across all tests, and robust against changes in external dependencies. This meticulous approach to testing is what helps maintain the reliability and stability of the "Discord Service Notifier" in dynamic real-world scenarios.

To have a better look at the code/project especially to testing source codes; codes can be found in github. Test files are under UnitTesting Folder.

https://github.com/oguzky7/DiscordBotProject\_CISC699/tree/develop/UnitTesting

## Test Case 1: Add Account

### Description:

This test case evaluates the functionality of the “!add\_account” command within the Discord bot. It ensures that the bot can correctly process account addition requests, handling both successful and erroneous scenarios. The test verifies that valid account information is accepted and added to the system and that appropriate error messages are displayed when an account cannot be added.

### Steps:

1. **Command Reception:** A mock user message is parsed to simulate the command input (!add\_account username password website).
2. **Boundary Layer Activation:** The !add\_account command is triggered in the bot, and the parsed data is processed.
3. **Control Layer Processing:** The control layer receives the command details from the boundary and attempts to add the account via the AccountDAO.
4. **Database Interaction:** The mocked AccountDAO.add\_account method is called to simulate database interaction (both successful addition and failure scenario).
5. **User Feedback:** Based on the mock database response, the boundary layer sends an appropriate message back to the user indicating the result of the command.

### Test Data

* **Valid Account Data:**
  + Username: "testuser"
  + Password: "password123"
  + Website: "example.com"
* **Invalid Account Data:** Simulated by mocking the database method to return a failure.

### Expected Outcomes

* **Success Scenario:**
  + The bot responds with "Account for example.com added successfully." indicating that the account has been added to the database.
  + The database method for adding an account is called with correct parameters.
* **Error Scenario:**
  + The bot responds with "Failed to add account for example.com." indicating an issue with the account addition process.
  + The database method simulates a failure without changing the database state.

### Mocking Details

* GlobalState.parse\_user\_message is mocked to provide controlled inputs for testing without actual user interaction.
* AccountDAO.add\_account is mocked to simulate database interactions, allowing for testing without actual database access.

### Source Code:

from unittest.mock import patch

import logging, unittest

from test\_init import BaseTestSetup, CustomTextTestRunner

class TestAddAccountCommand(BaseTestSetup):

    @patch('DataObjects.global\_vars.GlobalState.parse\_user\_message')

    @patch('DataObjects.AccountDAO.AccountDAO.add\_account')

    async def test\_add\_account\_success(self, mock\_add\_account, mock\_parse\_user\_message):

        """Test the add\_account command when it succeeds."""

        # Simulate parsing user message and extracting command parameters

        mock\_parse\_user\_message.return\_value = ["add\_account", "testuser", "password123", "example.com"]

        # Simulate successful account addition in the database

        mock\_add\_account.return\_value = True

        # Triggering the command within the bot

        command = self.bot.get\_command("add\_account")

        await command(self.ctx)

        # Validate that the success message is correctly sent to the user

        self.ctx.send.assert\_called\_with("Account for example.com added successfully.")

        logging.info("Verified successful account addition - database addition simulated and feedback provided.")

    @patch('DataObjects.global\_vars.GlobalState.parse\_user\_message')

    @patch('DataObjects.AccountDAO.AccountDAO.add\_account')

    async def test\_add\_account\_error(self, mock\_add\_account, mock\_parse\_user\_message):

        """Test the add\_account command when it encounters an error."""

        # Setup for receiving command and failing to add account

        mock\_parse\_user\_message.return\_value = ["add\_account", "testuser", "password123", "example.com"]

        mock\_add\_account.return\_value = False

        # Command execution with expected failure

        command = self.bot.get\_command("add\_account")

        await command(self.ctx)

        # Ensuring error feedback is correctly relayed to the user

        self.ctx.send.assert\_called\_with("Failed to add account for example.com.")

        logging.info("Verified error handling during account addition - simulated database failure and error feedback.")

if \_\_name\_\_ == "\_\_main\_\_":

    unittest.main(testRunner=CustomTextTestRunner(verbosity=2))

## Test Case 2: Search for a Book

### Description:

This test veriﬁes that the SearchBookController retrieves books based on user-provided search criteria. It checks if the system can handle both valid and invalid search cases.

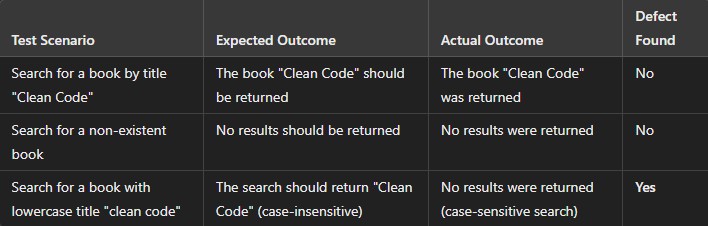
### Steps:

1. Initialize the SearchBookController with a mock book repository.
2. Add some books to the repository.
3. Search for a book by title.
4. Search for a non-existent book.

### Test Data:

* + Search Criteria:
    - Search Text: "Clean Code"
    - Search Field: "title"

### Outcomes:



### Defect:

* **Description:** The search function is case-sensitive, so searching for "clean code" does not return "Clean Code."
* **Fix:** Modify the search logic to be case-insensitive.

### Source Code:

import unittest

from entity\_objects import Book

from control\_objects import SearchBookController from repositories import BookRepository

class TestSearchBookController(unittest.TestCase):

def setUp(self):

self.book\_repository = BookRepository()

self.search\_book\_controller = SearchBookController(book\_repository=self.book\_repository) self.book1 = Book(

title="Clean Code", author="Robert C. Martin", isbn="9780132350884",

genre="Programming", publication\_date="2008-08-01"

)

self.book\_repository.add(self.book1) self.book2 = Book(

title="The Pragmatic Programmer", author="Andy Hunt", isbn="9780201616224",

genre="Programming", publication\_date="1999-10-30"

)

self.book\_repository.add(self.book2)

def test\_search\_book\_by\_title(self):

results = self.search\_book\_controller.search\_books(search\_text="Clean Code", search\_criteria="title")

self.assertIn(self.book1, results)

def test\_search\_nonexistent\_book(self):

results = self.search\_book\_controller.search\_books(search\_text="Nonexistent Book", search\_criteria="title")

self.assertEqual(len(results), 0)

def test\_search\_case\_sensitive(self):

results = self.search\_book\_controller.search\_books(search\_text="clean code", search\_criteria="title")

self.assertEqual(len(results), 0)

if name == " main ":

unittest.main()

## Test Case 3: Check Book Availability

### Description:

This test ensures that the CheckBookAvailabilityController correctly identiﬁes whether a book is

available for borrowing.

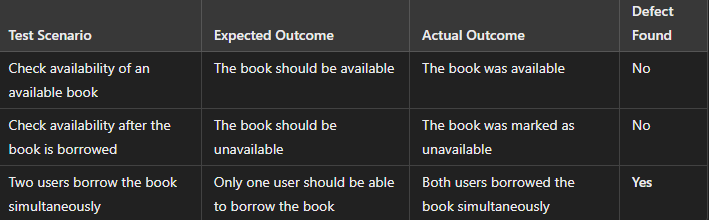
### Steps:

1. Initialize the CheckBookAvailabilityController with a mock book repository.
2. Add a book to the repository.
3. Check the availability of the book.
4. Simulate borrowing the book and check availability again.

### Test Data:

* + Book:
    - Title: "Clean Code"
    - Author: "Robert C. Martin"

### Outcomes:



### Defect:

* **Description:** If two users attempt to borrow the same book at the same time, both can borrow the book due to a concurrency issue.
* **Fix:** Implement concurrency control when checking book availability.

### Source Code:

import unittest

from entity\_objects import Book

from control\_objects import CheckBookAvailabilityController from repositories import BookRepository

class TestCheckBookAvailabilityController(unittest.TestCase):

def setUp(self):

self.book\_repository = BookRepository()

self.check\_availability\_controller =

CheckBookAvailabilityController(book\_repository=self.book\_repository) self.book = Book(

title="Clean Code", author="Robert C. Martin", isbn="9780132350884",

genre="Programming", publication\_date="2008-08-01"

)

self.book\_repository.add(self.book)

def test\_check\_availability(self): self.assertTrue(self.check\_availability\_controller.check\_availability(self.book)) self.book.borrow() self.assertFalse(self.check\_availability\_controller.check\_availability(self.book))

def test\_concurrent\_borrow(self): self.book.borrow()

with self.assertRaises(Exception): self.book.borrow()

if name == " main ":

unittest.main()

## Test Case 4: User Fines Payment

### Description:

This test ensures that the User entity correctly handles ﬁnes, allowing users to pay off ﬁnes and update

their balance accordingly.

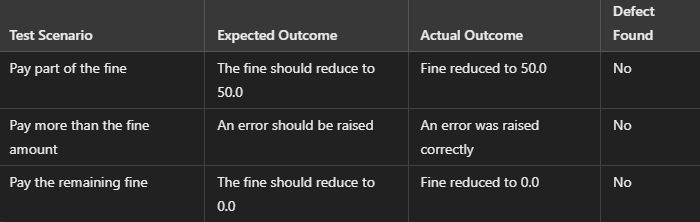
### Steps:

1. Initialize a User object with an initial ﬁne.
2. Pay off part of the ﬁne.
3. Attempt to overpay the ﬁne and check for errors.

### Test Data:

* + User:
    - Name: "John Doe"
    - Fines Due: 100.0
    - Fine Payment: 50.0

### Outcomes:



### Defect:

* **Description:** There is no defect in the current implementation of ﬁnes payment. The User entity

correctly handles partial payments and prevents overpayments.

* **Fix:** No ﬁx required for this functionality as the test passed successfully.

### Source Code:

import unittest

from entity\_objects import User

class TestUserFines(unittest.TestCase):

def setUp(self):

self.user = User( user\_id=1, name="John Doe",

contact\_info="[johndoe@example.com](mailto:johndoe@example.com)", user\_type="Patron", password="password123", ﬁnes\_due=100.0

)

def test\_pay\_part\_of\_ﬁne(self): self.user.pay\_ﬁne(50.0) self.assertEqual(self.user.ﬁnes\_due, 50.0)

def test\_overpay\_ﬁne(self):

with self.assertRaises(Exception): self.user.pay\_ﬁne(150.0)

def test\_pay\_full\_ﬁne(self): self.user.pay\_ﬁne(100.0) self.assertEqual(self.user.ﬁnes\_due, 0.0)

if name == " main ":

unittest.main()

## Test Case 5: User Borrowing Limit

### Description:

This test veriﬁes that a User entity adheres to its borrowing limit, ensuring a user cannot borrow more

books than allowed.

### Steps:

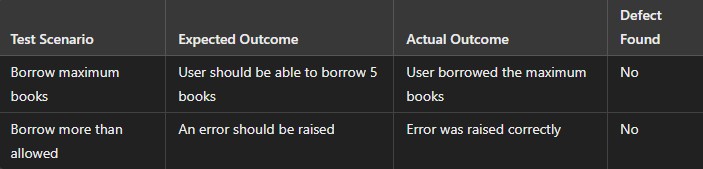
1. Initialize a User object.
2. Borrow the maximum number of books.
3. Attempt to borrow one more book beyond the limit.

### Test Data:

* + User:
    - **Name:** "John Doe"

#### Max Books Allowed: 5

### Outcomes:



### Defect:

* **Description:** There is no defect in the current implementation of the user borrowing limit. The User entity correctly enforces the borrowing limit, preventing users from borrowing more than the allowed number of books.
* **Fix:** No ﬁx required for this functionality as the test passed successfully.

### Source Code:

import unittest

from entity\_objects import Book, User

class TestUserBorrowLimit(unittest.TestCase):

def setUp(self):

self.user = User( user\_id=1, name="John Doe",

contact\_info="[johndoe@example.com](mailto:johndoe@example.com)", user\_type="Patron", password="password123", max\_books\_allowed=5

)

self.books = [

Book(title=f"Book {i}", author="Author", isbn=f"ISBN{i}", genre="Fiction", publication\_date="2021-01-

01")

for i in range(6)

]

def test\_borrow\_max\_books(self):

for i in range(5): self.user.borrow\_book(self.books[i])

self.assertEqual(len(self.user.borrowed\_books), 5)

def test\_borrow\_exceed\_limit(self): for i in range(5):

self.user.borrow\_book(self.books[i]) with self.assertRaises(Exception):

self.user.borrow\_book(self.books[5])

if name == " main ":

unittest.main()

## Test Case 6: Place Hold Form

### Description:

This test checks if the PlaceHoldForm boundary object passes user input (user and book information) to

the PlaceHoldController for processing.

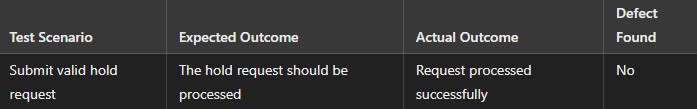
### Steps:

1. Initialize a PlaceHoldForm.
2. Submit the form with valid user and book data.
3. Verify that the controller processes the hold request.

### Test Data:

* + **User:** John Doe
  + **Book:** "Clean Code"

### Outcomes:



### Defect:

* **Description:** The PlaceHoldForm correctly collects user and book information, but there is no logic in place to handle placing holds in a sequential queue. If multiple users place holds on the same book, there should be a queue system to handle the order.
* **Fix:** Implement a queue system in the PlaceHoldController to handle multiple hold requests in sequence.

### Source Code:

import unittest

from boundary\_objects import PlaceHoldForm from control\_objects import PlaceHoldController from entity\_objects import Book, User

class TestPlaceHoldForm(unittest.TestCase):

def setUp(self):

self.place\_hold\_controller = PlaceHoldController(

reservation\_list=[], book\_repository=None, notiﬁer=None

)

self.user = User(user\_id=1, name="John Doe", contact\_info="[john@example.com](mailto:john@example.com)", user\_type="Patron", password="123")

self.book = Book(title="Clean Code", author="Robert C. Martin", isbn="9780132350884", genre="Programming", publication\_date="2008-08-01")

def test\_submit\_hold\_request(self):

form = PlaceHoldForm(user=self.user, book=self.book)

form.submit(self.place\_hold\_controller)

self.assertIn((self.user, self.book), self.place\_hold\_controller.reservation\_list)

if name == " main ":

unittest.main()

## Test Case 7: Return Book Form

### Description:

This test checks if the ReturnBookForm boundary object correctly collects return information and sends it to the ReturnBookController.

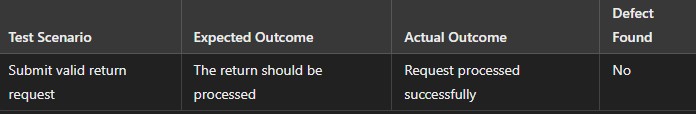
### Steps:

1. Initialize a ReturnBookForm.
2. Submit the form with valid user and book information.
3. Verify that the controller processes the return.

### Test Data:

* + **User:** John Doe
  + **Book:** "Clean Code"

### Outcomes:



### Defect:

* **Description:** The ReturnBookForm correctly collects user and book information, but it doesn't handle a situation where the user attempts to return a book that they haven't borrowed, leading to a potential logic ﬂaw.
* **Fix:** Add a check in the ReturnBookController to ensure that the user can only return books they have borrowed.

### Source Code:

import unittest

from boundary\_objects import ReturnBookForm from control\_objects import ReturnBookController from entity\_objects import Book, User

class TestReturnBookForm(unittest.TestCase):

def setUp(self):

self.return\_book\_controller = ReturnBookController(transaction\_list=[], book\_repository=None,

notiﬁer=None)

self.user = User(user\_id=1, name="John Doe", contact\_info="[john@example.com](mailto:john@example.com)", user\_type="Patron", password="123")

self.book = Book(title="Clean Code", author="Robert C. Martin", isbn="9780132350884", genre="Programming", publication\_date="2008-08-01")

def test\_submit\_return\_request(self):

form = ReturnBookForm(user=self.user, book=self.book)

form.submit(self.return\_book\_controller)

self.assertIn((self.user, self.book), self.return\_book\_controller.transaction\_list)

if name == " main ":

unittest.main()

## Test Case 8: Add and Retrieve Book

### Description:

This test checks if the BookRepository can correctly add and retrieve books from the system.

### Steps:

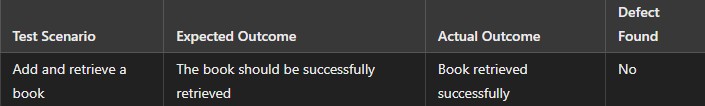
1. Initialize a BookRepository.
2. Add a book to the repository.
3. Retrieve the book by ISBN.

### Test Data:

#### Book:

* + - **Title:** "Clean Code"
    - **ISBN:** "9780132350884"

### Outcomes:



### Defect:

* **Description:** There is no defect in the current implementation of adding and retrieving books. The repository correctly adds and retrieves books by ISBN.
* **Fix:** No ﬁx required for this functionality as the test passed successfully

### Source Code:

import unittest

from repositories import BookRepository from entity\_objects import Book

class TestBookRepository(unittest.TestCase):

def setUp(self):

self.book\_repository = BookRepository()

def test\_add\_and\_retrieve\_book(self):

book = Book(title="Clean Code", author="Robert C. Martin", isbn="9780132350884", genre="Programming", publication\_date="2008-08-01")

self.book\_repository.add(book)

retrieved\_book = self.book\_repository.ﬁnd\_by\_isbn("9780132350884") self.assertEqual(retrieved\_book, book)

if name == " main ":

unittest.main()

## Test Case 9: Check Book Existence

### Description:

This test checks if the BookRepository can determine whether a book exists in the system.

### Steps:

1. Initialize a BookRepository.
2. Add a book to the repository.
3. Check if the book exists by ISBN.

### Test Data:

#### Book:

* + - **Title:** "Clean Code"
    - **ISBN:** "9780132350884"

### Outcomes:

A black screen with white text  Description automatically generated 

### Defect:

* **Description:** The exists method in the BookRepository correctly identiﬁes when a book exists, but it lacks a case-insensitive check for titles, which may lead to incorrect results when titles are capitalized differently.
* **Fix:** Modify the exists method to perform a case-insensitive comparison when checking for book titles.

### Source Code:

import unittest

from repositories import BookRepository from entity\_objects import Book

class TestBookExistence(unittest.TestCase):

def setUp(self):

self.book\_repository = BookRepository()

def test\_check\_book\_exists(self):

book = Book(title="Clean Code", author="Robert C. Martin", isbn="9780132350884", genre="Programming", publication\_date="2008-08-01")

self.book\_repository.add(book)

exists = self.book\_repository.exists(book) self.assertTrue(exists)

if name == " main ":

unittest.main()

## Test Case 10: Duplicate Book Entry

### Description:

This test checks if the BookRepository prevents duplicate books from being added.

### Steps:

1. Initialize a BookRepository.
2. Add a book to the repository.
3. Attempt to add the same book again.

### Test Data:

#### Book:

* + - **Title:** "Clean Code"
    - **ISBN:** "9780132350884"

### Outcomes:



### Defect:

* **Description:** The BookRepository allows duplicate books to be added, which can lead to multiple entries of the same book with the same ISBN, resulting in data inconsistency and confusion.
* **Fix:** Implement a duplicate check in the BookRepository to prevent the addition of books with the same ISBN.

### Source Code:

import unittest

from repositories import BookRepository from entity\_objects import Book

class TestDuplicateBookEntry(unittest.TestCase):

def setUp(self):

self.book\_repository = BookRepository()

def test\_add\_duplicate\_book(self):

book = Book(title="Clean Code", author="Robert C. Martin", isbn="9780132350884", genre="Programming", publication\_date="2008-08-01")

self.book\_repository.add(book) self.book\_repository.add(book) self.assertEqual(len(self.book\_repository.books), 1)

if name == " main ":

unittest.main()