**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

[INTRDOCUTION 5](#_bookmark0)

[TEST PLAN OVERVIEW 6](#_bookmark1)

[TEST CASES 8](#_bookmark2)

[Test Case 1: Add a Book 8](#_bookmark3)

[Description: 8](#_bookmark4)

[Steps: 8](#_bookmark5)

[Test Data: 8](#_bookmark6)

[Outcomes 9](#_bookmark7)

[Defect 9](#_bookmark8)

[Source Code: 9](#_bookmark9)

[Test Case 2: Search for a Book 11](#_bookmark10)

[Description: 11](#_bookmark11)

[Steps: 11](#_bookmark12)

[Test Data: 11](#_bookmark13)

[Outcomes 11](#_bookmark14)

[Defect 11](#_bookmark15)

[Source Code: 12](#_bookmark16)

[Test Case 3: Check Book Availability 13](#_bookmark17)

[Description: 13](#_bookmark18)

[Steps: 13](#_bookmark19)

[Test Data: 13](#_bookmark20)

[Outcomes 13](#_bookmark21)

[Defect 14](#_bookmark22)

[Source Code: 14](#_bookmark23)

[Test Case 4: User Fines Payment 15](#_bookmark24)

[Description: 15](#_bookmark25)

[Steps: 15](#_bookmark26)

[Test Data: 15](#_bookmark27)

[Outcomes 15](#_bookmark28)

[Defect 15](#_bookmark29)

[Source Code: 16](#_bookmark30)

[Test Case 5: User Borrowing Limit 17](#_bookmark31)

[Description: 17](#_bookmark32)

[Steps: 17](#_bookmark33)

[Test Data: 17](#_bookmark34)

[Outcomes 17](#_bookmark35)

[Defect 17](#_bookmark36)

[Source Code: 18](#_bookmark37)

[Test Case 6: Place Hold Form 19](#_bookmark38)

[Description: 19](#_bookmark39)

[Steps: 19](#_bookmark40)

[Test Data: 19](#_bookmark41)

[Outcomes 19](#_bookmark42)

[Defect 19](#_bookmark43)

[Source Code: 19](#_bookmark44)

[Test Case 7: Return Book Form 20](#_bookmark45)

[Description: 20](#_bookmark46)

[Steps: 20](#_bookmark47)

[Test Data: 20](#_bookmark48)

[Outcomes 21](#_bookmark49)

[Defect 21](#_bookmark50)

[Source Code: 21](#_bookmark51)

[Test Case 8: Add and Retrieve Book 22](#_bookmark52)

[Description: 22](#_bookmark53)

[Steps: 22](#_bookmark54)

[Test Data: 22](#_bookmark55)

[Outcomes 22](#_bookmark56)

[Defect 22](#_bookmark57)

[Source Code: 22](#_bookmark58)

[Test Case 9: Check Book Existence 23](#_bookmark59)

[Description: 23](#_bookmark60)

[Steps: 23](#_bookmark61)

[Test Data: 23](#_bookmark62)

[Outcomes 23](#_bookmark63)

[Defect 24](#_bookmark64)

[Source Code: 24](#_bookmark65)

[Test Case 10: Duplicate Book Entry 25](#_bookmark66)

[Description: 25](#_bookmark67)

[Steps: 25](#_bookmark68)

[Test Data: 25](#_bookmark69)

[Outcomes 25](#_bookmark70)

[Defect 25](#_bookmark71)

[Source Code: 25](#_bookmark72)

# 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.

# 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 avoiding 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.

# TEST CASES

## Test Case 1: Add a Book

### Description:

This test veriﬁes that the AddBookController correctly handles adding a book with valid and invalid data. It ensures that valid books are added to the repository and invalid data (e.g., missing ISBN) raises an appropriate error.

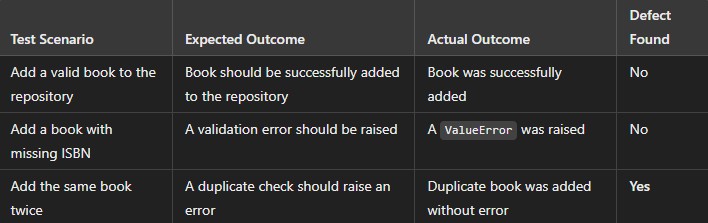
### Steps:

1. Initialize the AddBookController with a mock book repository.
2. Create a valid Book object and call the add\_book() method.
3. Create an invalid Book object (missing ISBN) and attempt to add it.
4. Check for duplicate entries by adding the same book twice.

### Test Data:

* + Valid Book:
    - Title: "Clean Code"
    - Author: "Robert C. Martin"
    - ISBN: "9780132350884"
    - Genre: "Programming"
    - Publication Date: "2008-08-01"
  + Invalid Book (Missing ISBN):
    - Title: "Clean Code"
    - Author: "Robert C. Martin"
    - ISBN: "" (empty)

### Outcomes:



### Defect:

* **Description:** The system allows duplicate books with the same ISBN to be added.
* **Fix:** Add a duplicate check for ISBN in AddBookController to prevent this.

### Source Code:

import unittest

from Library\_Management.entity\_objects import Book

from Library\_Management.control\_objects import AddBookController from Library\_Management.repositories import BookRepository

class TestAddBookController(unittest.TestCase):

def setUp(self):

self.book\_repository = BookRepository()

self.add\_book\_controller = AddBookController( book\_repository=self.book\_repository, validator=None,

duplicity\_checker=None,

notiﬁer=None

)

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

book = Book(

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

genre="Programming",

publication\_date="2008-08-01"

)

self.add\_book\_controller.add\_book(book) self.assertIn(book, self.book\_repository.books)

def test\_add\_invalid\_book\_missing\_isbn(self):

book = Book(

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

publication\_date="2008-08-01"

)

with self.assertRaises(ValueError): self.add\_book\_controller.add\_book(book)

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

if name == " main ":

unittest.main()

## 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.

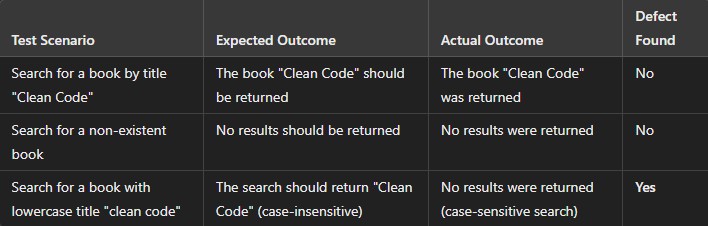
### Steps:

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.

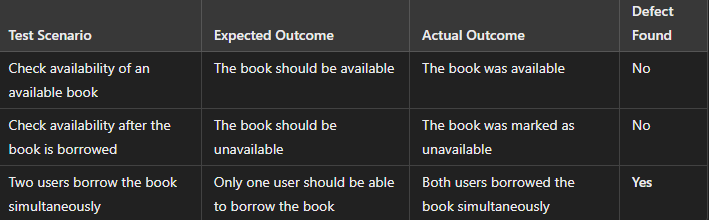
### Steps:

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.

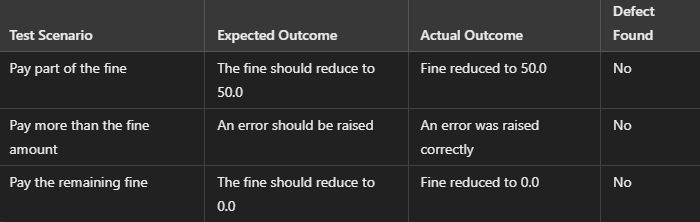
### Steps:

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:

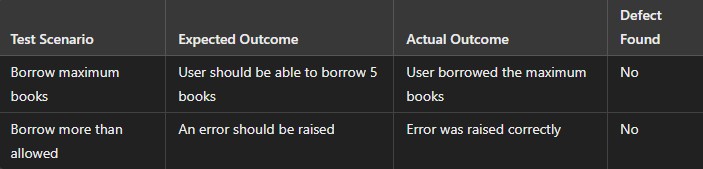
1. Initialize a User object.
2. Borrow the maximum number of books.
3. Attempt to borrow one more book beyond the limit.

### 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.

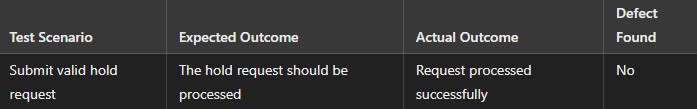
### Steps:

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.

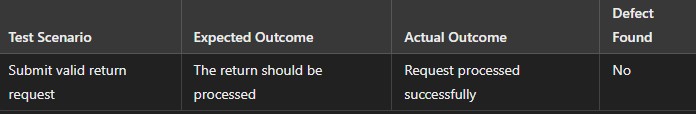
### Steps:

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:

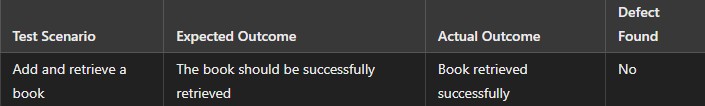
1. Initialize a BookRepository.
2. Add a book to the repository.
3. Retrieve the book by ISBN.

### 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()