SOFTWARE TESTING DOCUMENT

CSE3112: Software Engineering Lab

Title: Boikini

Submitted By:
Shahanaz Sharmin (07)
Anika Tabassum (61)

Contents

1 Unit Testing

1.1 Static Testing

1.1.1 Walk through

1.1.1.1 Account

Concern	Comment
Uninitialized Variables	We found no uninitialized variables in this section of our code since all the variables that were found to be uninitialized were omitted out.
Undocumented Empty Blocks	There was also no undocumented empty block in this class. All the blocks were appropriately commented and documented.
Code Guideline Violations	At first sight, the coding structure seemed good enough as it did not violate any coding norms like proper attribute names, class names, method names, spacing etc.
Code Anomalies	After the initial testing phase, no code anomalies were found.
Structural Anomalies	The code was constructed in a fairly modular way, so it was possible to minimize structural anomalies as much as possible.

1.1.1.2 bookstore

Concern	Comment
Uninitialized Variables	We found no uninitialized variables in this section of our code since all the variables that were found to be uninitialized were omitted out.
Undocumented Empty Blocks	There was also no undocumented empty block in this class. All the blocks were appropriately commented and documented.
Code Guideline Violations	At first sight, the coding structure seemed good enough as it did not violate any coding norms like proper attribute names, class names, method names, spacing etc.
Code Anomalies	After the initial testing phase, no code anomalies were found.
Structural Anomalies	The code was constructed in a fairly modular way, so it was possible to minimize structural anomalies as much as possible.

1.1.1.3 cart

Concern	Comment
Uninitialized Variables	We found no uninitialized variables in this section of our code since all the variables that were found to be uninitialized were omitted out.
Undocumented Empty Blocks	There was also no undocumented empty block in this class. All the blocks were appropriately commented and documented.
Code Guideline Violations	At first sight, the coding structure seemed good enough as it did not violate any coding norms like proper attribute names, class names, method names, spacing etc.
Code Anomalies	After the initial testing phase, no code anomalies were found.
Structural Anomalies	The code was constructed in a fairly modular way, so it was possible to minimize structural anomalies as much as possible.

1.1.1.4 catagory

Concern	Comment
Uninitialized Variables	We found no uninitialized variables in this section of our code since all the variables that were found to be uninitialized were omitted out.
Undocumented Empty Blocks	There was also no undocumented empty block in this class. All the blocks were appropriately commented and documented.
Code Guideline Violations	At first sight, the coding structure seemed good enough as it did not violate any coding norms like proper attribute names, class names, method names, spacing etc.
Code Anomalies	After the initial testing phase, no code anomalies were found.
Structural Anomalies	The code was constructed in a fairly modular way, so it was possible to minimize structural anomalies as much as possible.

1.1.1.5 checkout

Concern	Comment
Uninitialized Variables	We found no uninitialized variables in this section of our code since all the variables that were found to be uninitialized were omitted out.
Undocumented Empty Blocks	There was also no undocumented empty block in this class. All the blocks were appropriately commented and documented.
Code Guideline Violations	At first sight, the coding structure seemed good enough as it did not violate any coding norms like proper attribute names, class names, method names, spacing etc.
Code Anomalies	After the initial testing phase, no code anomalies were found.
Structural Anomalies	The code was constructed in a fairly modular way, so it was possible to minimize structural anomalies as much as possible.

1.1.2 Code Review

Code Review - Muztoba Sinha
App has a good interface. It's really intuitive and easy to use. All the core functionalities work properly.

The variables were named properly and the code was easy to go through as the structure was quite well made. The documentation was precise and easy to read. Though the app was not fully complete, the completed parts were fully functional and no notable issues were found

Code Review - Bholanath Das Niloy
I am in awe of the exceptional quality and structure exhibited in this code for this project. The level of meticulousness and attention to detail is truly commendable. It's evident that a significant amount of effort was devoted to crafting code that is not only functional but also easy to understand and maintain.

The thoughtfulness and skill showcased throughout the code are truly remarkable. The code's organization and clarity are a testament to your deep understanding of software development best practices. By prioritizing code quality, maintainability, and performance, they have created a solid foundation that will undoubtedly benefit the project in the long run.

1.2 Dynamic Testing

1.2.1 Black box Testing

1.2.1.1 Range Partitioning

1.Account Controller

test 1: Registering an Account

This test case is used to check if a user can register successfully with valid data. It checks if the registration form is submitted correctly, the user is created in the database, and if the page redirects to the login page after successful registration.

Code

```
def test_register_with_valid_data(self):
        # Test values within the range
        data = {
            'first_name': 'Shahanaz',
            'last_name': 'Bithi',
```

```
            'username': 'bithi123',
            'email': 'shahanazsharmin66@gmail,com',
            'password': 'password123',
            'confirm_password': 'password123',
            'phone': '1234567890',
        }
        response = self.client.post(self.register_url, data)
        self.assertEqual(response.status_code, 302)
        self.assertRedirects(response, reverse('login'))
        self.assertTrue(Account.objects.filter(username='bithi123').exists())
```

Result : Passed

test 2: Invalid Registration

The test verifies that the user is not created in the database by checking if an Account object with the specified username ('johndoe') exists.

Code

```
def test_register_with_invalid_data(self):
        # Test case for invalid data
        data = {
            'first_name': 'John123',  # First name contains a number
            'last_name': 'Doe456',  # Last name contains a number
            'username': 'johndoe',
            'email': 'invalid_email',  # Invalid email address
            'password': 'password123',
            'confirm_password': 'password456',  # Password and confirm password do
                not match
            'phone': '1234567890',
        }
        response = self.client.post(self.register_url, data)

        # Check if the response status code is 302 (redirect)
        self.assertEqual(response.status_code, 302)

        # Check if the user is not created in the database
        self.assertFalse(Account.objects.filter(username='johndoe').exists())

```

```
        # Check if the expected error messages are present in the messages framework
        messages = list(get_messages(response.wsgi_request))
        self.assertEqual(len(messages), 4)  # Expecting 4 error messages
        self.assertIn("Sorry, First Name can't contain number", [str(message) for
            message in messages])
        self.assertIn("Sorry, Last Name can't contain number", [str(message) for
            message in messages])
        self.assertIn("Enter a valid email address.", [str(message) for message in
            messages])
        self.assertIn("Password and Confirm Password do not match.", [str(message)
            for message in messages])
```

Result : Passed

2. BookController

test 1: Adding an book

This test case is used to check if a user can register successfully with valid data. It checks if the registration form is submitted correctly, the user is created in the database, and if the page redirects to the login page after successful registration.

Code

```
class BookModelTestCase(TestCase):
    def setUp(self):
        self.category = Category.objects.create(category_name='Test Category')
        self.book = Book.objects.create(
            title='Test Book',
            author='Test Author',
            price=10.0,
            description='Test Description',
            category=self.category,
            stocks=20
        )

    def test_book_creation(self):
        self.assertEqual(self.book.title, 'Test Book')
        self.assertEqual(self.book.author, 'Test Author')
        self.assertEqual(self.book.price, 10.0)
        self.assertEqual(self.book.description, 'Test Description')
```

```
        self.assertEqual(self.book.category, self.category)
        self.assertEqual(self.book.stocks, 20)
        self.assertEqual(self.book.stocks_available, True)
        self.assertIsNotNone(self.book.modified_on)
        self.assertIsNotNone(self.book.created_on)

    def test_book_str_method(self):
        self.assertEqual(str(self.book), 'Test Book')
```

Result : Passed

test 2: API call with Invalid Data

The test case expects the response status code to be 200, indicating a successful request. It also expects the books queryset in the response context to be empty. If both assertions pass, it means that the home view correctly handles the invalid data and doesn't render any books.

Code

```
def test_home_view_with_invalid_data(self, mock_get):

        mock_response = {
            # Invalid response data, missing required fields
        }
        mock_get.return_value.json.return_value = mock_response

        response = self.client.get(reverse('home'))

        self.assertEqual(response.status_code, 200)

        self.assertQuerysetEqual(
            response.context['books'],
            [],
            transform=lambda x: x
        )

    @patch('requests.get')
```

Result : Passed

3. CartController

test 1: Adding an book

This test case is used to add books in the cart, update cart or delete books from the cart.

Code

```
from django.test import Client, TestCase
from bookstore.models import Book
from cart.models import Cart, CartItems

class AddToCartTestCase(TestCase):

    def setUp(self):
        self.client = Client()
        self.book = Book.objects.create(title='Test Book', price=10, author='Test
            Author', slug='test-book')
        self.session = self.client.session
        self.session.save()
        self.cart = Cart.objects.create(cart_session=self.session.session_key)
        self.cart_item = CartItems.objects.create(cart=self.cart, book=self.book,
            quantity=1, is_active=True)
```

Result : Passed

4. CategoryController

test 1: Divide books into different categories

This section helps user to find the favorite genre section for books easily.

Code

```
from django.test import TestCase, RequestFactory
from django.urls import reverse
from bookstore.models import Book
from category.models import Category
from category.views import category
```

```
6
7  class CategoryViewTestCase(TestCase):
8
9      def setUp(self):
10         self.factory = RequestFactory()
11         self.cat1 = Category.objects.create(
12             category_name='Test Category 1',
13             slug='test-category-1',
14             category_image=None,
15             category_des='This is a test category'
16         )
17         self.book1 = Book.objects.create(
18             title='Test Book 1',
19             slug='test-book-1',
20             author='Test Author',
21             price=20.0,
22             stock=5,
23             category=self.cat1
24         )
```

Result : Passed

5. CheckoutController

test 1: Checkouts

This test case verifies that the checkout process works correctly with valid data. It covers the authentication, form submission, and creation of the order object.

Code

```
1  from django.test import TestCase
2  from django.contrib.auth import get_user_model
3  from checkout.models import order
4  from accounts.models import Account
5
6  class CheckoutTestCase(TestCase):
7      def setUp(self):
8          client = Account.objects.create(username="test_client", email="test2@example.
               com", password="testpassword")
9          order.objects.create(order_id=1, client=client, order_status="PENDING")
```

```
10
11      def test_checkout_with_invalid_data(self):
12          self.client.login(username='testuser', password='testpassword')
13          data = {
14              'transaction_id': '',  # Invalid: Empty transaction ID
15              'order_note': 'Test order',
16              'first_name': 'John',
17              'last_name': 'Doe',
18              'address': '',  # Invalid: Empty address
19              'city': 'City',
20              'division': 'Division',
21              'zip': '12345',
22              'country': 'Country',
23          }
24          response = self.client.post('/checkout_req/process', data)
25          self.assertEqual(response.status_code, 200)  # Check that the form submission
                is not successful
26
27          # Check that the order object is not created
28          self.assertFalse(order.objects.exists())
```

Result : Passed

1.2.1.2 Set Partitioning

1. CartController

test 1: Divide books into different categories

This section helps user to find the favorite genre section for books easily.

Code

```
1  from django.test import TestCase
2  from django.urls import reverse
3  from django.contrib.auth.models import User
4  from bookstore.models import Book
5  from cart.models import Cart, CartItems
6
7  class CartTestCase(TestCase):
8      def setUp(self):
```

```
        self.user = User.objects.create_user(username='testuser', password='
            testpassword')
        self.book = Book.objects.create(title='Test Book', author='Test Author',
            price=9.99, slug='test-book')
        self.client.login(username='testuser', password='testpassword')
```

Result : Passed

2. CategoryController

test 1: Making an Order

This test case verifies that the checkout process works correctly with valid data. It covers the authentication, form submission, and creation of the order object.

Code

```

from django.test import TestCase, Client
from django.urls import reverse
from bookstore.models import Book
from category.models import Category

class CategoryViewTestCase(TestCase):
    def setUp(self):
        self.client = Client()
        self.category = Category.objects.create(
            category_name="Test Category",
            slug="test-category",
            category_des="This is a test category",
        )
        self.book1 = Book.objects.create(
            title="Test Book 1",
            author="Test Author",
            price="19.99",
            category=self.category,
            description="This is a test book",
        )
        self.book2 = Book.objects.create(
            title="Test Book 2",
            author="Test Author",
            price="24.99",
```

```
            category=self.category,
            description="This is another test book",
        )
```

Result : Passed

3. CheckoutController

test 1: Checkouts

This test case verifies that the checkout process works correctly with valid data. It covers the authentication, form submission, and creation of the order object.

Code

```
from django.test import TestCase
from django.contrib.auth import get_user_model
from checkout.models import order
from accounts.models import Account

class CheckoutTestCase(TestCase):
    def setUp(self):
        client = Account.objects.create(username="test_client", email="test2@example.
            com", password="testpassword")
        order.objects.create(order_id=1, client=client, order_status="PENDING")

    def test_checkout_with_valid_data(self):
        self.client.login(username='testuser', password='testpassword')
        data = {
            'transaction_id': '123456',
            'order_note': 'Test order',
            'first_name': 'John',
            'last_name': 'Doe',
            'address': '123 Street',
            'city': 'City',
            'division': 'Division',
            'zip': '12345',
            'country': 'Country',
        }
        response = self.client.post('/checkout_req/process', data)
        self.assertEqual(response.status_code, 302)  # Check for successful redirect

```

```
27          # Check if the order object is created
28          self.assertTrue(order.objects.exists())
```

Result : Passed

1.2.2 White box Testing

1.2.2.1 Statement/Code Coverage

Statement 1: Account Create

Flow chart

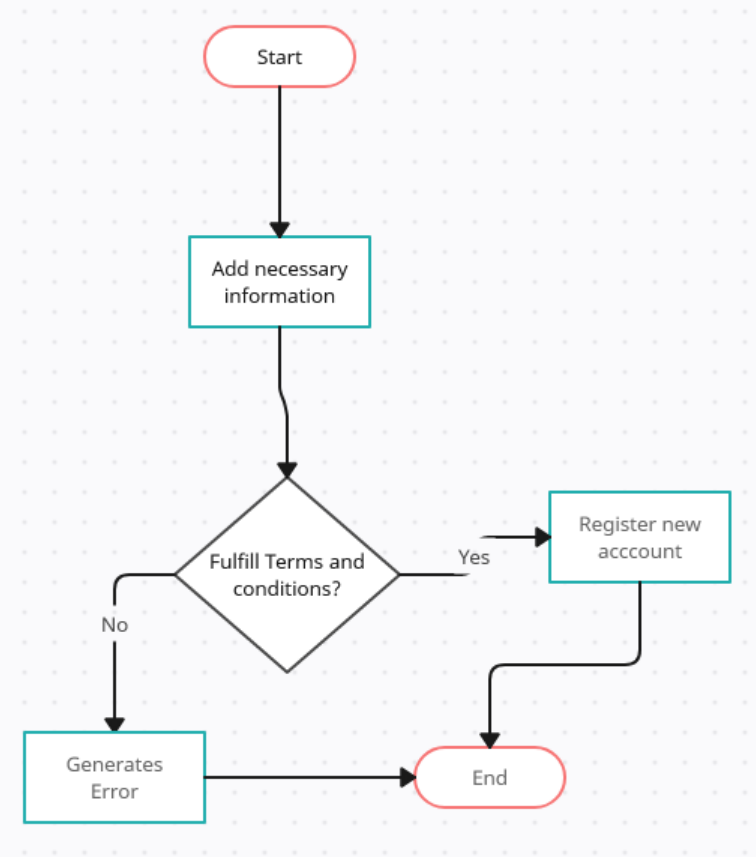

Figure 1: Statement 1

Test Case 1: register(request):= True

Output: Add new Acoounts

Statement coverage: $5/6 * 100 = 83.3\%$

Test Case 2: register(request):= False

Output: Generates Error

Statement coverage: $5/6 * 100 = 83.3\%$

These two test cases cover the entire statement.

Statement 2: Search

Flow chart

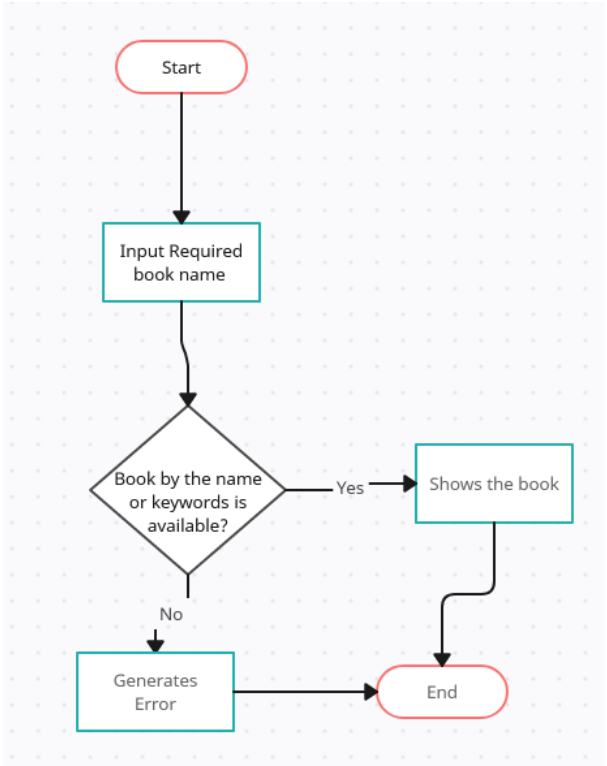

Figure 2: Statement 2

Test Case 1: search_result(request):= True

Output: Shows the available books

Statement coverage: $5/6 * 100 = 83.3\%$

Test Case 2: search_result(request):= False

Output: Shows the unavailability of the books

Statement coverage: $5/6 * 100 = 83.3\%$

These two test cases cover the entire statement.

Statement 3: Login

Flow chart

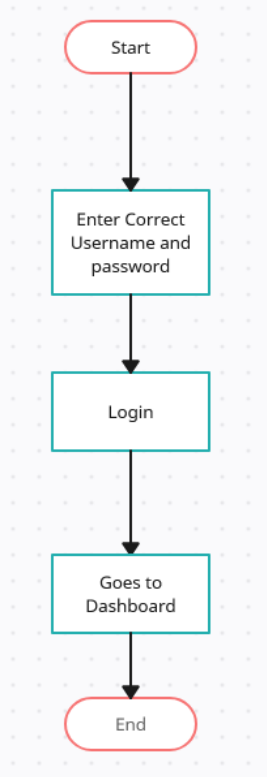

Figure 3: Statement 3

Test Case 1: login(request)

Output: Logins to user account

Statement coverage: $5/5 * 100 = 100\%$

This test case covers the entire statement.

Statement 4: Edit Profile

Flow chart

Figure 4: Statement 4

Test Case 1: Edit_account

Output: Edits editable informations

Statement coverage: $4/4 * 100 = 100\%$

This test case covers the entire statement.

Statement 5: Orders

Flow chart

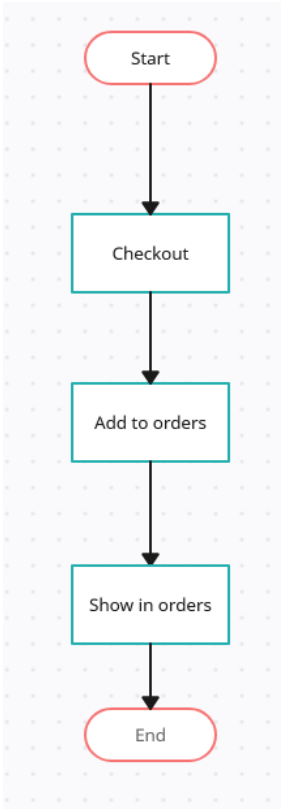

Figure 5: Statement 5

Test Case 1: view_order(request, order_id):

Output: Shows the order lists

Statement coverage: $5/5 * 100 = 100\%$

This test case covers the entire statement.

Statement 6: Add to Carts

Flow chart

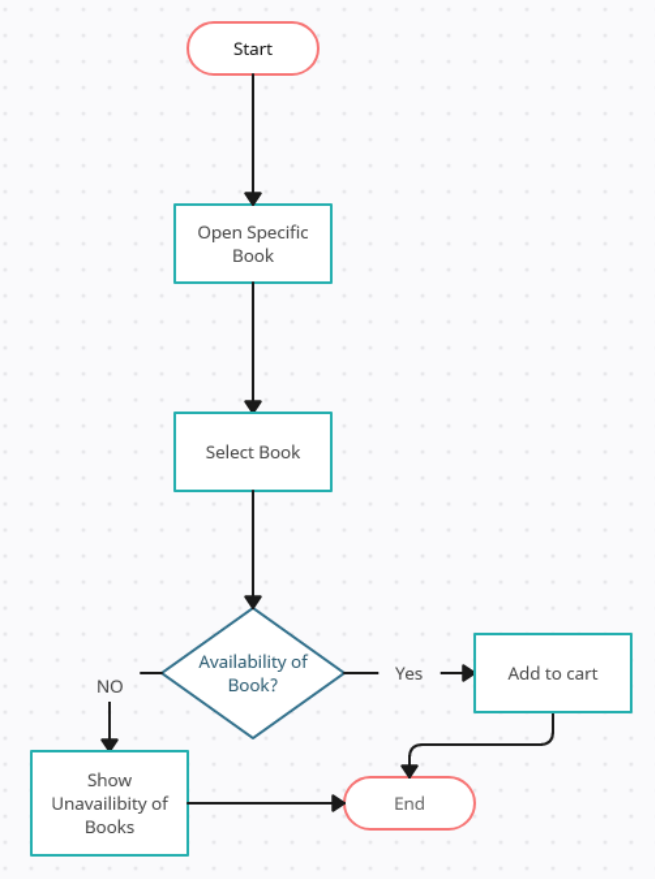

Figure 6: Statement 6

Test Case 1: add_to_cart:= True

Output: Adds Books To the Cart

Statement coverage: $6/7 * 100 = 85.7\%$

Test Case 2: add_to_cart:= False

Output: Shows the unavailability of the books

Statement coverage: $6/7 * 100 = 85.7\%$

These two test cases cover the entire statement.

Statement 7: Update Carts

Flow chart

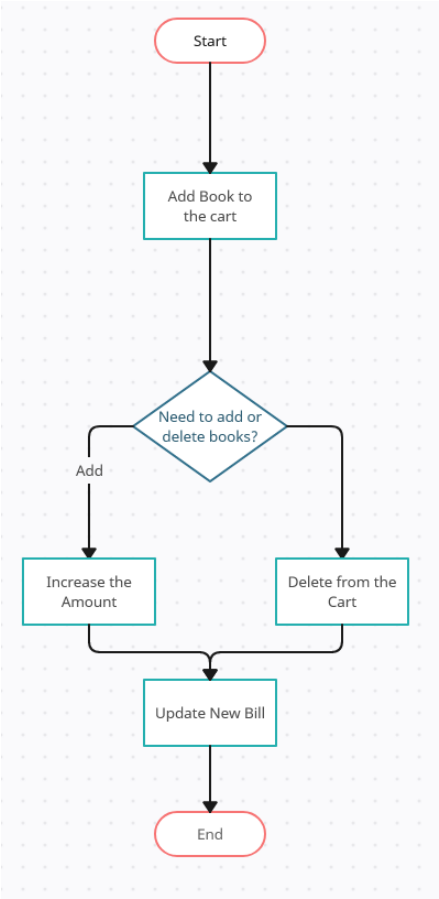

Figure 7: Statement 7

Test Case 1: update_cart_item:= True

Output: Updates Books To the Cart nad updates the bill

Statement coverage: $6/7 * 100 = 85.7\%$

Test Case 2:delete_cart_item:= True

Output: Deletes the books from the cart and updates the bill

Statement coverage: $6/7 * 100 = 85.7\%$

These two test cases cover the entire statement.

Statement 8: Checkout

Flow chart

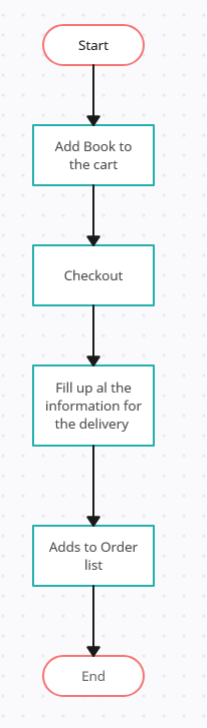

Figure 8: Statement 8

Test Case 1: checkout_req

Output: Checkouts and updates the billing and delivery informations

Statement coverage: $6/6 * 100 = 100\%$

This test case covers the entire statement.

Statement 9: Details of books

Flow chart

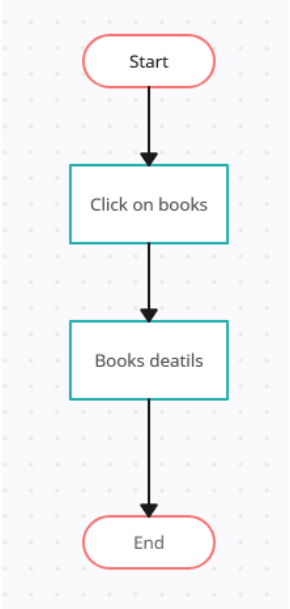

Figure 9: Statement 9

Test Case 1: book_data

Output: Loads Book Details

Statement coverage: $4/4 * 100 = 100\%$

This test case covers the entire statement.

Statement 10: Category

Flow chart

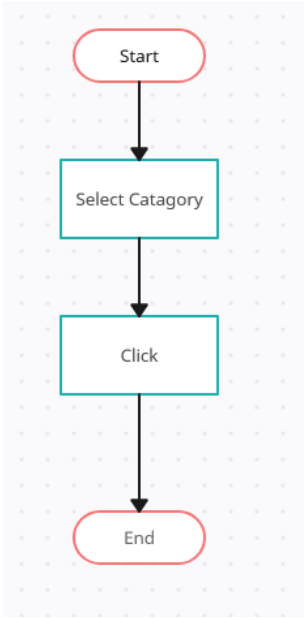

Figure 10: Statement 10

Test Case 1: category

Output: Loads Book categories

Statement coverage: $4/4 * 100 = 100\%$

This test case covers the entire statement.

1.2.2.2 Branch and Conditional Coverage

Statement 1: Account Create

Flow chart

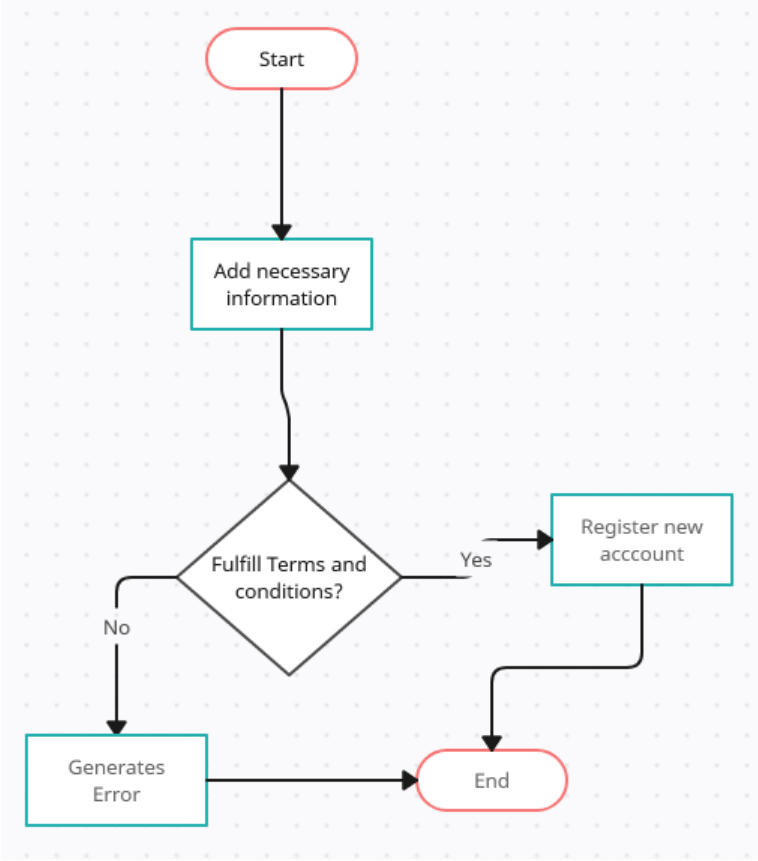

Figure 11: Statement 1

Test Case 1: register(request)= True. This is for the case when new user wants to create new account.

Test Case 2: register(request)= False. This is for the case when the user fails to follow the protocol.

These branches cover the entire statement.

Statement 2: Search

Flow chart

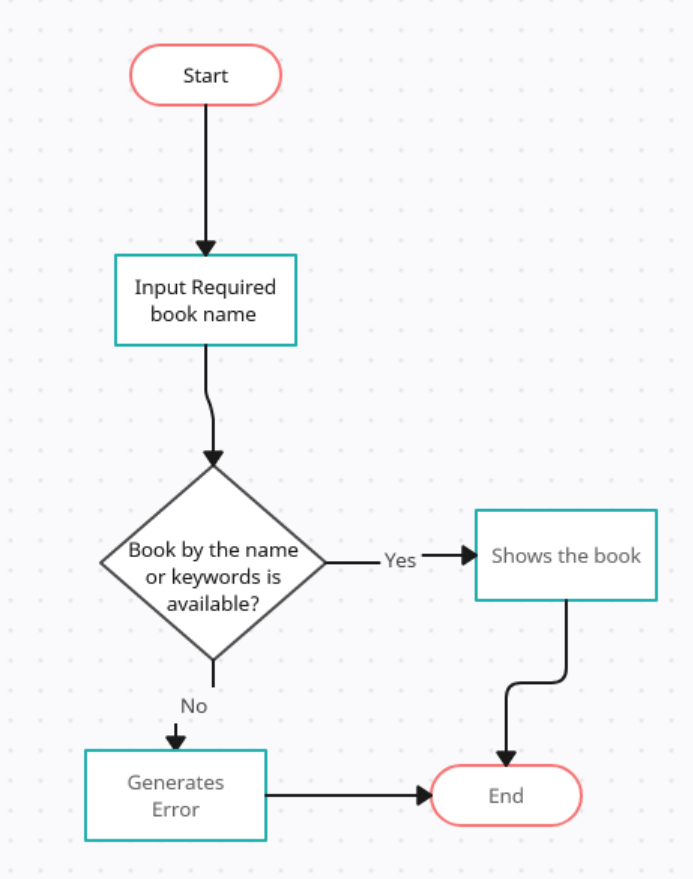

Figure 12: Statement 2

Test Case 1: search_result(request)= True. This is for the case when new user searches for new books.

Test Case 2: search_result(request)= False. This is the case when the searched book is not available.

These branches cover the entire statement.

Statement 3: Login

Flow chart

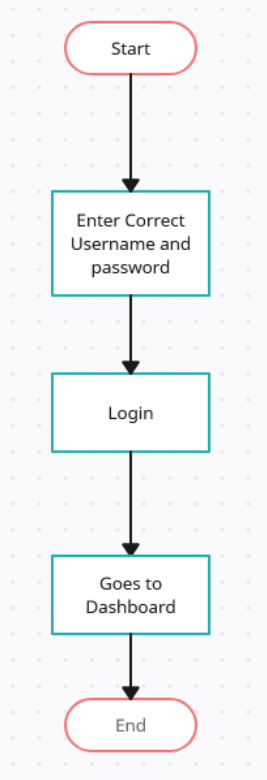

Figure 13: Statement 3

Test Case 1: Logins to the user account.

This test case covers the entire statement.

Statement 4: Edit Profile

Flow chart

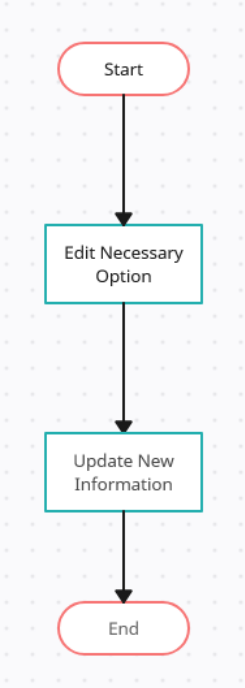

Figure 14: Statement 4

Test Case 1: Edit if the user needs to update any information.

This branch covers the entire statement.

Statement 5: Orders

Flow chart

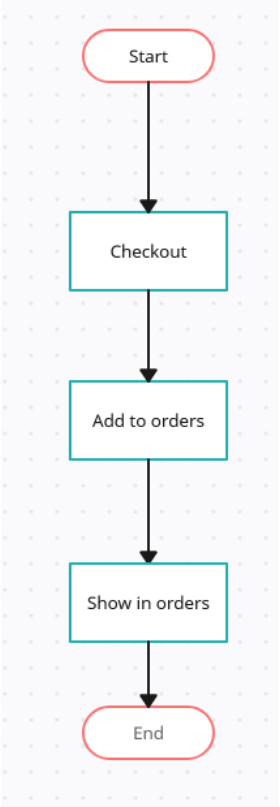

Figure 15: Statement 5

Test Case 1: Shows the order lists

This test case covers the entire statement.

Statement 6: Add to Carts

Flow chart

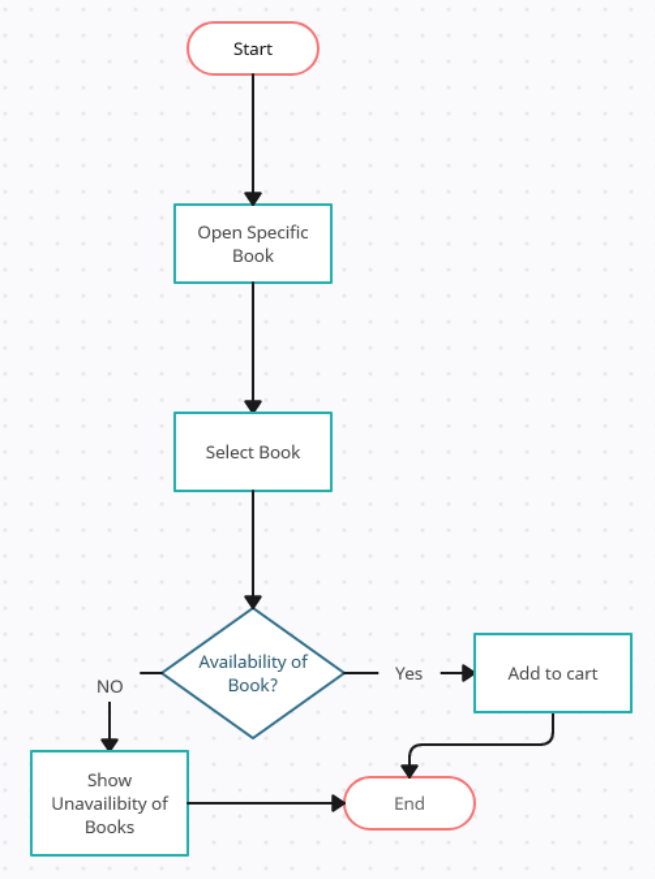

Figure 16: Statement 6

Test Case 1: add_to_cart:= True. This is for the case when user wants to add books to the cart.

Test Case 2: add_to_cart:= False. This is for the case when the book in not available or the user wants to add more than 20 books.

These two test cases cover the entire statement.

Statement 7: Update Carts

Flow chart

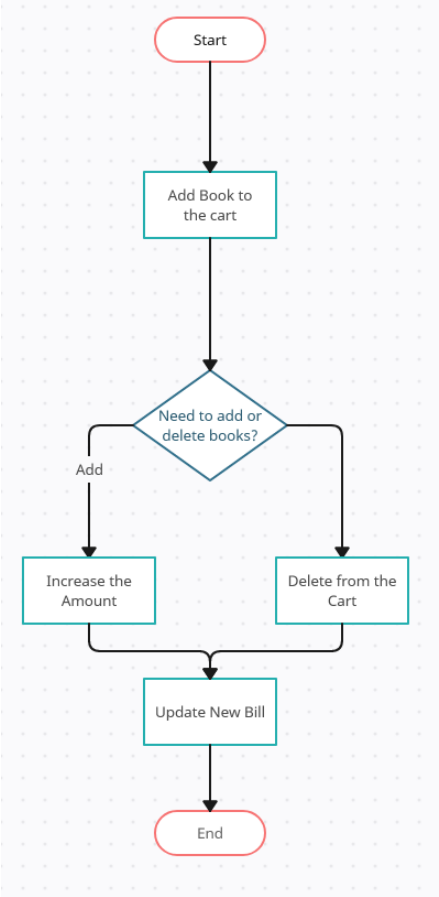

Figure 17: Statement 7

Test Case 1: update_cart_item:= True. This is for the case when the user wants to add new books or increases the book quantity.

Test Case 2:delete_cart_item:= True. This is the case when the user wants to remove books from the order list.

These two test cases cover the entire statement.

Statement 8: Checkout

Flow chart

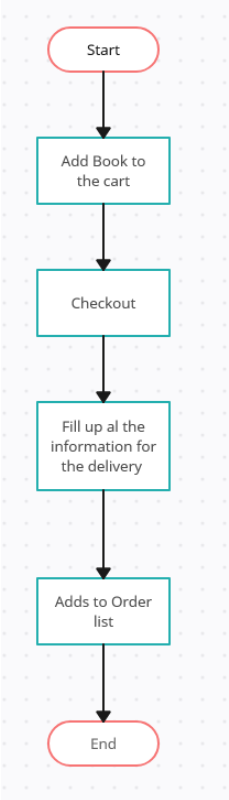

Figure 18: Statement 8

Test Case 1: checkout_req = true. This is for the case when the user wants to order news books and clears payments.

This test case covers the entire statement.

Statement 9: Details of books

Flow chart

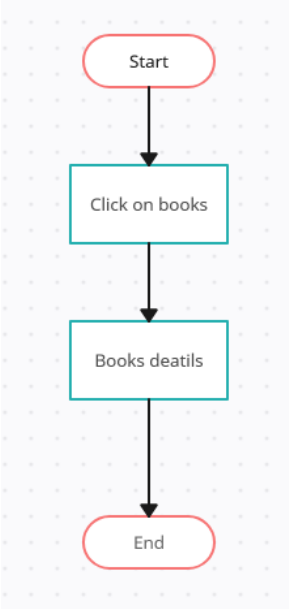

Figure 19: Statement 9

Test Case 1: book_data . This is for the case when the user wants to get the details information about any books.

This test case covers the entire statement.

Statement 10: Category

Flow chart

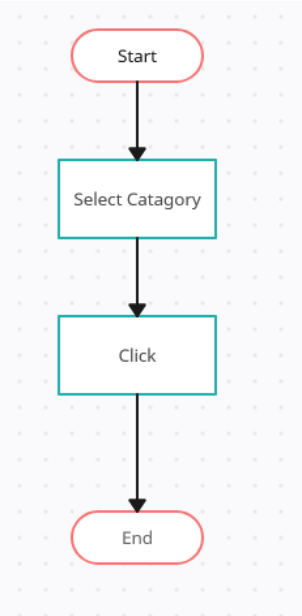

Figure 20: Statement 10

Test Case 1: category= This is for the case when user wants to getse any specific type of genre for books.

This test case covers the entire statement.

1.2.2.3 Path Coverage

Statement 1: Account Create

Flow chart

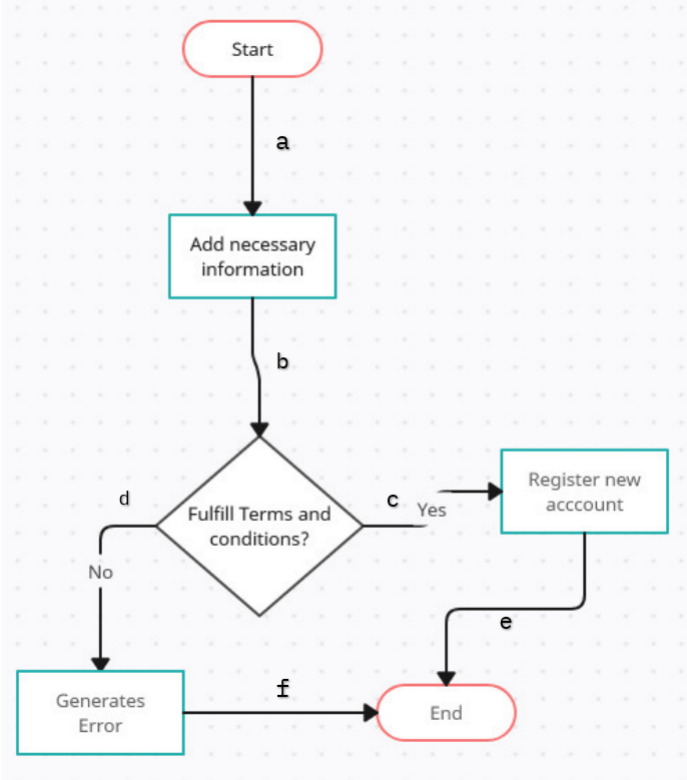

Figure 21: Statement 1

Test Case 1: register(request)= True.
Path: a-b-c-e

Test Case 2: register(request)= False.
Path: a-b-d-f

These branches cover the entire statement.

Statement 2: Search

Flow chart

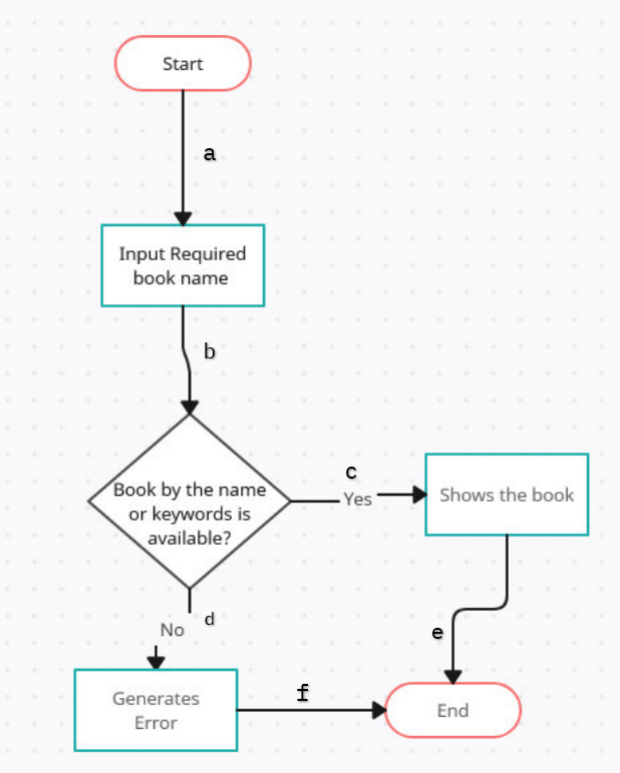

Figure 22: Statement 2

Test Case 1: search_result(request)= True.
Path: a-b-c-e

Test Case 2: search_result(request)= False.
Path: a-b-d-f

These branches cover the entire statement.

Statement 3: Login

Flow chart

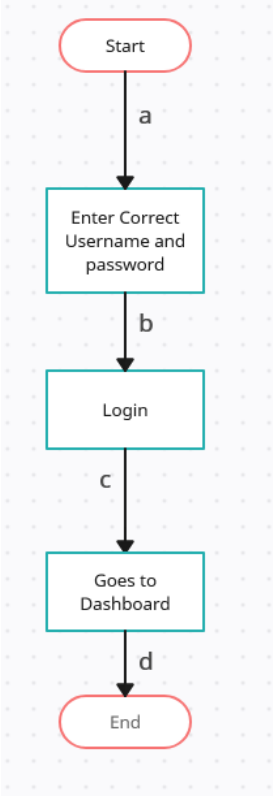

Figure 23: Statement 3

Test Case 1: Logins to the user account.
Path: a-b-c-d

This test case covers the entire statement.

Statement 4: Edit Profile

Flow chart

Figure 24: Statement 4

Test Case 1: Edit if the user needs to update any information.
Path: a-b-c

This branch covers the entire statement.

Statement 5: Orders

Flow chart

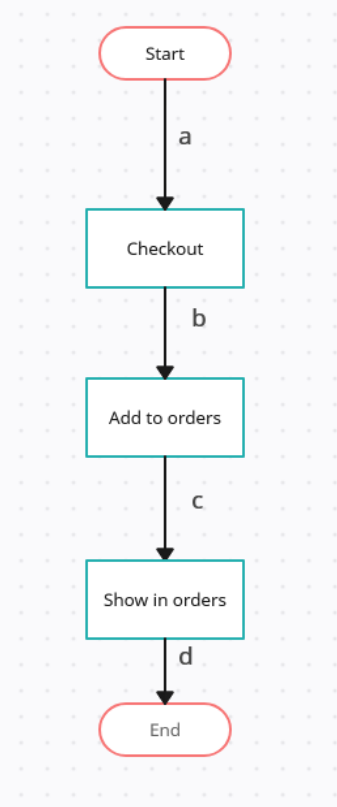

Figure 25: Statement 5

Test Case 1: Shows the order lists
Path: a-b-c-d

This test case covers the entire statement.

Statement 6: Add to Carts

Flow chart

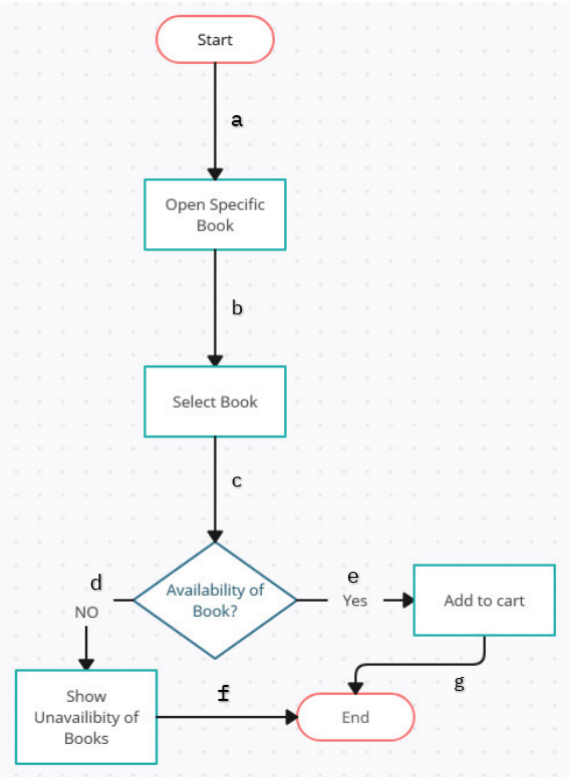

Figure 26: Statement 6

Test Case 1: add_to_cart:= True. This is for the case when user wants to add books to the cart.
Path: a-b-c-e-g

Test Case 2: add_to_cart:= False. This is for the case when the book in not available or the user wants to add more than 20 books.
Path: a-b-c-d-f

These two test cases cover the entire statement.

Statement 7: Update Carts

Flow chart

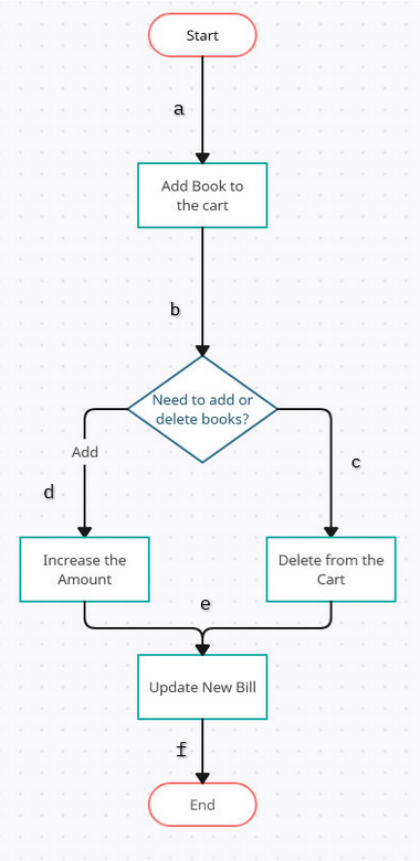

Figure 27: Statement 7

Test Case 1: update_cart_item:= True. This is for the case when the user wants to add new books or increases the book quantity.
Path: a-b-d-e-f

Test Case 2:delete_cart_item:= True. This is the case when the user wants to remove books from the order list.
Path: a-b-c-e-f

These two test cases cover the entire statement.

Statement 8: Checkout

Flow chart

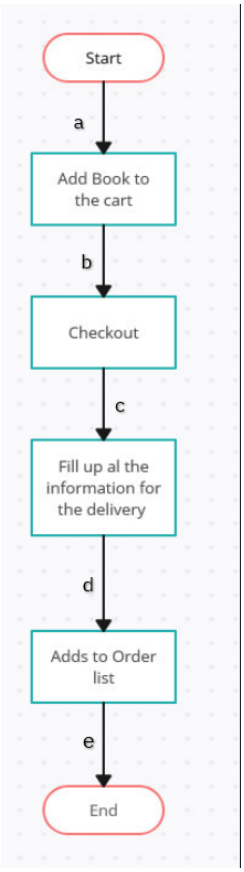

Figure 28: Statement 8

Test Case 1: checkout_req = true. This is for the case when the user wants to order news books and clears payments.
Path: a-b-c-d-e

This test case covers the entire statement.

Statement 9: Details of books

Flow chart

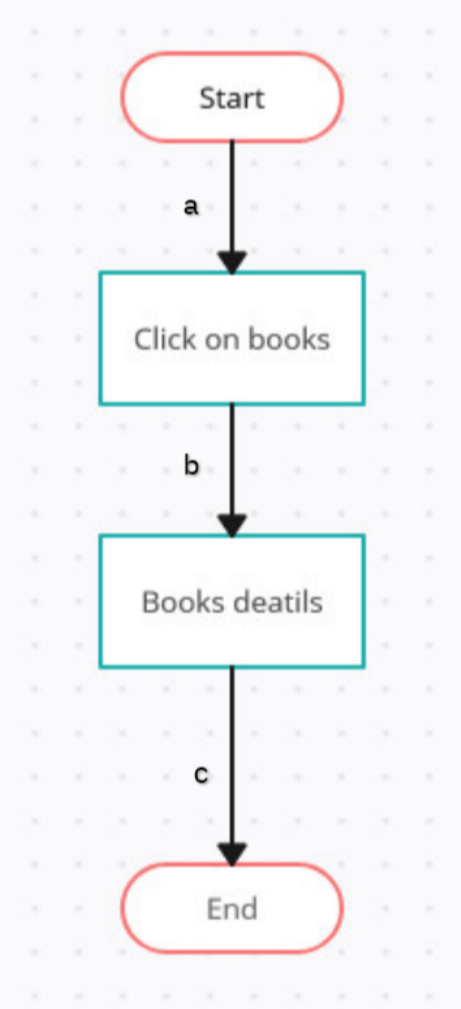

Figure 29: Statement 9

Test Case 1: book_data . This is for the case when the user wants to get the details information about any books.
Path: a-b-c

This test case covers the entire statement.

Statement 10: Category

Flow chart

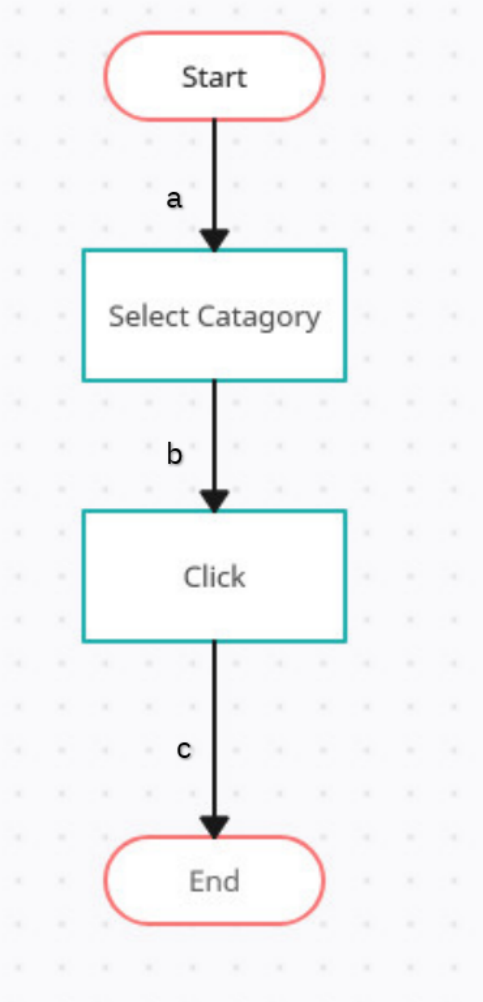

Figure 30: Statement 10

Test Case 1: category= This is for the case when user wants to gets any specific type of genre for books.
Path: a-b-c

This test case covers the entire statement.

2 Integration Testing

For our application, we have decided to use a Bottom-up approach for integration testing. The reason for this is that our application follows an object-oriented approach and this approach would be quite suitable.

2.1 Bottom-up Testing

Not applicable.

2.2 Top down testing

Test 1: Create account and all the processes

Test case objective
User will create a new account and go through the whole website.

Test outcome
The walk through of the entire website will occur.

Stub code

```
def register(request):

  if request.POST:
      post_username = request.POST['username']
      post_password = request.POST['password']
      post_conf_password =request.POST['confirm_password']
      post_email = request.POST['email']
      post_phone =  request.POST['phone']
      post_first_name = request.POST['first_name']
      post_last_name = request.POST['last_name']
      check_username = Account.objects.all().filter(username=post_username)
      check_email = Account.objects.all().filter(email=post_email)
      check_phone = Account.objects.filter(phone=post_phone).exists()

      # Checking for number

      if num_checker(post_first_name)==True:
```

```
18              messages.error(request, "Sorry, First Name can't contain number")
19              return redirect("register")
20
21          if num_checker(post_last_name) == True:
22              messages.error(request, "Sorry, Last Name can't contain number")
23              return redirect("register")
24
25          # Checking for special character
26
27          if special_char_checker(post_first_name):
28              messages.error(request, "Sorry, First Name can't contain a special
                    character.")
29              return redirect("register")
30
31
32          if special_char_checker(post_last_name):
33              messages.error(request,"Sorry, Last Name can't contain a special
                    character.")
34              return redirect("register")
35
36          if special_char_checker(post_username):
37              messages.error(request, "Sorry, Username can't contain a special
                    character.")
38              return redirect("register")
39
40          if email_special_char_checker(post_email):
41              messages.error(request, "Sorry, Email can't contain a special character."
                    )
42              return redirect("register")
43
44
45
46          if post_password != post_conf_password:
47
48              messages.error(request, 'Password and Confirm Password Does not match')
49              return redirect("register")
50          if check_phone ==True:
51              messages.error(request,"An user with the phone number already exits.")
52              return redirect("register")
53
54          if not check_username or not check_email:
55              user = Account.objects.create(
56                  first_name=post_first_name,
```

```
                last_name=post_last_name,
                username=post_username,
                email=post_email,
                phone = post_phone,
            )
            user.set_password(post_password)
            user.save()
            messages.success(request, 'Your account has been registered. Please Login
                now')
            return redirect("login")
        else:
            messages.error(request, "Sorry, an user with the same credentials already
                exits. Please login to your account")
            return redirect("login")

    else:
        if request.user.is_authenticated:
            return redirect('dashboard.html')
        else:
            return render(request, 'register.html')
```

Output

User went through the entire app

Status

Passed

Test 2: Login

Test case objective

User can login to the account and purchase books.

Test outcome

Log in to the account.

Stub code

```
def login(request):
    if request.user.is_authenticated:
        return redirect("dashboard")
    if request.POST:
```

```
        session_old = request.session.session_key
        post_email = request.POST['email']
        post_password = request.POST['password']
        user = auth.authenticate(email=post_email,password=post_password)
        if user is not None:

            auth.login(request, user)
            session_new = request.session.session_key
            try:
              act =Account.objects.get(email=post_email)
              act.last_active = datetime.now()
              act.save()
              cart = Cart.objects.all().filter(cart_session=session_old)
              cart.update(cart_session = session_new)
            except:
                pass
            messages.success(request, "You have been logged in.")
            return redirect('dashboard')
        else:
            messages.error(request, "Sorry your Email/Password don't match")
            return redirect('login')

    return render(request,"login.html")
```

Output

Logged in to the account

Status

Passed

Test 3: Home Page

Test case objective

This will lead the user to the home page after logging in.

Test outcome

Home page will occur.

Stub code

```
def account_home(request):
    user = Account.objects.get(email=request.user.email)
    orders = order.objects.all().filter(client=user).order_by('date_created')[:4]
    total_oders = len(order.objects.all().filter(client=user).order_by('date_created'
        ))
    dilevered_orders = len(order.objects.all().filter(client=user,order_status="
        COMPLETED"))
    print(total_oders)
    print(dilevered_orders)
    registered_on = user.registered_on
    registered_on = datetime.fromisoformat(str(registered_on)).strftime("%d/%m/%Y")
    last_login = user.last_active
    last_login = datetime.fromisoformat(str(last_login)).strftime("%d/%m/%Y")
    if request.user.is_authenticated:
        context={
            'first_name': request.user.first_name,
            'last_name': request.user.last_name,
            'order_id_list' : orders,
            'total_orders':total_oders,
            'registered_on':registered_on,
            'dilevered_orders':dilevered_orders,
            'last_login':last_login,

        }
        return render(request, "dashboard.html",context=context)
    else:
         messages.error(request,"Sorry, You are not logged in. Please Login and try
             again")
         return redirect("login")
```

Output

Home page occurs

Status

Passed

Test 4: Search for books

Test case objective

User will search for new books and can choose the required books.

Test outcome

Searched Book will occur in the screen.

Stub code

```
def search_result(request):
    if 'query' in request.GET:
        query = request.GET['query']
        url = f"https://www.googleapis.com/books/v1/volumes?q={query}"
        response = requests.get(url)
        data = response.json()
        books = []

        for item in data.get('items', []):
            volume_info = item.get('volumeInfo', {})
            price_info = volume_info.get('saleInfo', {}).get('listPrice', {})
            book = Book(
            title=volume_info.get('title', ''),
            author=volume_info.get('authors', [''])[0],
            description=volume_info.get('description', ''),
            cover_image_url=volume_info.get('imageLinks', {}).get('thumbnail', ''),
            price=price_info.get('amount') if price_info else 300,
           # currency=price_info.get('currencyCode') if price_info else 300,
            category=Category.objects.get_or_create(volume_info.get('categories', ['']
                )[0])[0],
            slug=item.get('title', ''),  # Set the slug to the book's ID

            )
            books.append(book)
            book.save()

        context = {
            'books': books,
        }
        return render(request, 'search_res.html', context)
```

Output
Searched book occurs

Status
Passed

Test 5 : Category

Test case objective
User can see the category for the books.

Test outcome
Shows the list of the book categories.

Stub code

```
def category(request, cat_slug=None):
    cat_name = ""
    if cat_slug is None:
        all_books = Paginator(Book.objects.all().order_by('-modified_on'),20)
    else:
        print(cat_slug)
        cat = Category.objects.get(slug=cat_slug)
        all_books = Paginator(Book.objects.all().filter(category=cat).order_by('-
            modified_on'),20)
        cat_name= cat.category_name

    page = request.GET.get('page')

    try:
        books = all_books.page(page)
    except PageNotAnInteger:
        books = all_books.page(1)
    except EmptyPage:
        books = all_books.page(1)

    context = {
        'books': books,
         'category_name':cat_name,
```

```
    }
    return render(request,'books-cat.html', context)
```

Output

Book categories shown

Status

Passed

Test 6 : Details of books

Test case objective

User can see the details of any books.

Test outcome

Shows the details about the book.

Stub code

```
def single_book(request, single_book_slug):
    if single_book_slug is not None:
        book = get_object_or_404(Book,slug=single_book_slug)

        #releated_categories = get_object_or_404(Category,slug=single_book_slug)
        #releated_books = Book.objects.all().filter(category=book.category)[0:5]
        context = {

            'book': book,
           # 'related_books': releated_books,

        }

    return render(request, 'book-single-page.html',context)
```

Output

Books details shown

Status

Passed

Test 7 : Add to carts

Test case objective

User can add books in the cart.

Test outcome

New cart will be created.

Stub code

```
def add_to_cart(request, user_book):
    session  = request.session.session_key
    print(session)
    if not session:
        session = request.session.create()
        session = request.session.session_key
        cart_for_save = Cart.objects.create(
          cart_session=session,
         )
        cart_for_save.save()
    try:
      session_aa = Cart.objects.get(cart_session=session)
    except:
        session = request.session.session_key
        cart_for_save = Cart.objects.create(
            cart_session=session,
        )
        cart_for_save.save()

    op_book = Book.objects.get(slug=user_book)

    try:
        check_if_already_exits = CartItems.objects.get(cart=session_aa, book=op_book)
        print(check_if_already_exits)
        if check_if_already_exits:
            op_book = Book.objects.get(slug=user_book)
            quantity_update = CartItems.objects.get(cart=session_aa, book=op_book)
            quantity_update = quantity_update.quantity + 1

            cartitem = CartItems.objects.get(
                cart=Cart.objects.get(cart_session=session),
```

```
                book=Book.objects.get(slug=user_book),
            )
            cartitem.quantity = cartitem.quantity + 1
            cartitem.save()
    except:
        cartitem_save = CartItems.objects.create(
            cart=Cart.objects.get(cart_session=session),
            book=Book.objects.get(slug=user_book),
            quantity=1,
            is_active=True,
        )
        cartitem_save.save()
    return redirect('cart')
```

Output

New cart creates

Status

Passed

Test 8 : Update carts

Test case objective

User can update carts or delete books from the cart.

Test outcome

Cart will be updated.

Stub code

```
def update_cart_item(request, book_slug):
    if request.POST:
        session = request.session.session_key
        user_session = Cart.objects.get(cart_session=session)
        quantity_update = int(request.POST['quantity'])

        print(quantity_update)
        cartitem = CartItems.objects.get(
            cart=Cart.objects.get(cart_session=session),
```

```
            book=Book.objects.get(slug=book_slug),
        )
        if quantity_update!=cartitem.quantity:
          cartitem.quantity = quantity_update
          cartitem.save()
    else:
        return redirect('cart')
    return redirect('cart')


def delete_cart_item(request, book_slug):
    session = request.session.session_key
    my_cart = Cart.objects.get(cart_session=session)
    book_item = Book.objects.get(slug=book_slug)
    cart_items = CartItems.objects.all().filter(cart=my_cart,book=book_item)
    cart_items.delete()
    return redirect('cart')


def cart(request):
    session = request.session.session_key
    cart_num = Cart.objects.get(cart_session=session)
    cart_items = CartItems.objects.all().filter(cart=cart_num)
    total=0
    for cart_item in cart_items:
        total += cart_item.book.price*cart_item.quantity
    context = {
        'cart_items': cart_items,
        'total':total,
    }
    return render(request, "cart.html", context)
```

Output

New cart creates

Status

Passed

Test 9 : Checkout

Test case objective

User can checkout the bill and purchase books.

Test outcome

Purchases can be done.

Stub code

```
def checkout_req(request):

    special_char_list = r"!\"#$%&'()*+,-./:;<=>?@[\]^_`{|}~"
    email_special_char_list = r"!\"#$%&'()*+,/:;<=>?@[\]^`{|}~"

    def num_checker(string):
        return any(i.isdigit() for i in string)

    def special_char_checker(string):
        for i in string:
            if i in special_char_list:
                return True
        return False

    def email_special_char_checker(string):
        if "@" in string:
            email = string.split("@", "")
            for i in email[0]:
                if i in email_special_char_list:
                    return True
            return False
        else:
            return True

    if request.POST:
        req_user = request.user

        if req_user.is_authenticated:

            #checking if transaction ID alreay exits in db

```

```
            transaction_id = request.POST['transaction_id']
            invoice_exits = invoice.objects.filter(transaction_id=transaction_id).
                exists()

            if invoice_exits == True:
                messages.error(request,"Sorry, transaction Id alreay exits.")
                return redirect("checkout_page")


            # working on order model

            client = request.user
            print(client)
            order_note = request.POST['order_note']
            # Unsafe to grab total from get or post req
            # So, I think it's bettter for me to comment out this line.
            # But I could have used it because it's a university project
            # and not many people is going to use it in their production environment.
                 Feel free to use if if you like

            # total = request.POST['total']


            order_save = order.objects.create(
                client=client,
               # order_note_user=order_note,

            )
            order_save.save()

            # working on order_list


            session = request.session.session_key

            cart = Cart.objects.get(cart_session=session)
            print(cart)
            cart_items_list = CartItems.objects.all().filter(cart=cart)
            print(type(cart_items_list[0]))
            total = 0
            print(order_save)

            for item in cart_items_list:
```

```
            order_item= Book.objects.get(title=item.book, id=item.book_id)
            price = order_item.price
            quantity = item.quantity
            total += price*quantity

            order_list_save = order_list.objects.create(
                order_id=order_save,
                order_item=order_item,
                order_price=price,
                order_quantity=quantity
            )
            order_list_save.save()

        # working on invoice


        total_price = total
        first_name = request.POST['first_name']
        last_name = request.POST['last_name']
        address = request.POST['address']
        city = request.POST['city']
        division = request.POST['division']
        zip = request.POST['zip']
        country = request.POST['country']
        order_note = request.POST['order_note']

        # Chcking for any invalid/ suspecious inputs

        # Checking for number

        if num_checker(first_name) == True:
            messages.error(request, "Sorry, First Name can't contain number")
            return redirect("checkout_page")

        if num_checker(last_name) == True:
            messages.error(request, "Sorry, Last Name can't contain number")
            return redirect("checkout_page")

        if num_checker(division) == True:
            messages.error(request, "Sorry, Division can't contain a number")
            return redirect("checkout_page")

```

```
116             if num_checker(city) == True:
117                 messages.error(request, "Sorry, City can't contain number")
118                 return redirect("checkout_page")
119
120             if num_checker(country) == True:
121                 messages.error(request, "Sorry, Country Name can't contain number")
122                 return redirect("checkout_page")
123
124             # Checking for special character
125
126             if special_char_checker(first_name):
127                 messages.error(request, "Sorry, First Name can't contain a special
                        character.")
128                 return redirect("checkout_page")
129
130             if special_char_checker(last_name):
131                 messages.error(request, "Sorry, Last Name can't contain a special
                        character.")
132                 return redirect("checkout_page")
133
134             if special_char_checker(division):
135                 messages.error(request, "Sorry, Division can't contain a special
                        character.")
136                 return redirect("checkout_page")
137
138
139
140             save_invoice = invoice.objects.create(
141                 order_id=order_save,
142                 total_price=total_price,
143                 first_name=first_name,
144                 last_name=last_name,
145                 address=address,
146                 division=division,
147                 city=city,
148                 zip=zip,
149                 country=country,
150                 transaction_id=transaction_id,
151                 order_note=order_note,
152                 transaction_method = 'bkash',
153                 invoice_status="PENDING_CHECK",
154
155
```

```
            )
            # updating order

            order_status_update = order.objects.filter(order_id=order_save.order_id).
                update(order_status="PROCESSING")
            # removing cart
            cart.delete()
            # decreasing stock
            stocks_now = Book.objects.get(title=item.book, id=item.book_id)
            stocks_now.stocks = stocks_now.stocks-1
            stocks_now.save()
            messages.success(request,"Your order has been successfully received.")
            return redirect("orders")


        else:
            return redirect("login")
```

Output

Creates Checkout and add the purchase to the dashboard.

Status

Passed

Test 10 : Orders

Test case objective

Adds to the order lists.

Test outcome

If any new order is created, that will be added to this list.

Stub code

```
def orders(request):
        if request.user.is_authenticated:
            user = Account.objects.get(email=request.user.email)
            order_id = order.objects.all().filter(client=user).order_by('date_created
                ')
```

```
5
6             all_orders = Paginator(order.objects.all().filter(client=user).order_by('
                  -date_created'), 10)
7             page = request.GET.get('page')
8
9             try:
10                orders = all_orders.page(page)
11            except PageNotAnInteger:
12                orders = all_orders.page(1)
13            except EmptyPage:
14                orders=  all_orders.page(all_orders.num_pages)
15
16            context={
17
18                'order_id_list' : orders,
19            }
20            return render(request,"list-orders.html",context)
21        else:
22            messages.error("Sorry, you need to be logged in to view your orders")
23            return redirect("login")
```

Output

Crestes new orders

Status

Passed

Test 11 : View Orders

Test case objective

Shows the order lists.

Test outcome

If any new order is created, that will be added to this list and shows the order list.

Stub code

```
1 def view_order(request, order_id):
2       if request.user.is_authenticated:
```

```
            print(order_id)
            order_items_list = order_list.objects.all().filter(order_id=order_id)
            invoice_details = invoice.objects.all().filter(order_id=order_id)

            context={
                "order_id":order_id,

                "order_items_list":order_items_list,
                "invoice_list": invoice_details
            }
            return render(request,"view_order.html",context=context)
        else:
            return redirect('login')


def view_invoice(request, invoice_id):
     if request.user.is_authenticated:
         invoice_dat = invoice.objects.get(invoice_id=invoice_id)

         context = {
             'invoice':invoice_dat
         }
         return render(request,"view_invoice.html",context=context)
     else:
         return _RedirectStream("login")
```

Output

Showed the order list

Status

Passed

Test 12: Edit Profile

Test case objective

User can edit their profile if they want also they can change their password.

Test outcome

New information will occur and new password will be updated.

Stub code

```
def profile_edit(request):
    if request.user.is_authenticated:
        if request.POST:
            first_name = request.POST['first_name']
            last_name = request.POST['last_name']
            email = request.POST['email']
            phone = request.POST['phone']

            # Checking for numbers

            if num_checker(first_name) == True:
                messages.error(request, "Sorry, First Name can't contain number.")
                return redirect("profile_edit")

            if num_checker(last_name) == True:
                messages.error(request, "Sorry, Last Name can't contain number.")
                return redirect("profile_edit")

            # Checking for special character

            if special_char_checker(first_name):
                messages.error(request, "Sorry, First Name can't contain a special
                    character.")
                return redirect("profile_edit")

            if special_char_checker(last_name):
                messages.error(request, "Sorry, Last Name can't contain a special
                    character.")
                return redirect("profile_edit")

            if email_special_char_checker(email):
                messages.error(request, "Sorry, Email can't contain a special
                    character.")
                return redirect("profile_edit")


            user = Account.objects.all().filter(username=request.user.username)
```

```
            user.update(first_name=first_name,
                        last_name=last_name,
                        email=email,
                        phone=phone)
            messages.success(request, "Your Profile has been updated")

        return render(request, "edit_profile.html")

    else:
        messages.error(request,"Sorry, You need to be logged in to do this action.")
        return redirect('login')

#if we want to change password
def change_pwd(request):
    if request.POST:
        password = request.POST['password']
        confirm_password = request.POST['verify_password']
        old_password = request.POST['old_password']
        if password == confirm_password:
            user = Account.objects.get(email=request.user.email)
            if user.check_password(old_password):
              user.set_password(password)
              user.save()
              messages.success(request,"Your Password has been successfully chanaged.
                  ")
              return redirect("login")
            else:
              messages.error(request, "Sorry, your old password doesn't match our
                  record.")
              return redirect("change_pwd")
        else:
           messages.error(request, "Sorry your password and verify password doesn't
               match.")
           return redirect("change_pwd")
    else:
      return render(request,"change_password.html")
```

Output

New information and Password Updates

Status

Passed

2.3 Sandwich testing

Not applicable.

3 Acceptance Testing

3.1 Functional Testing

1. User registration and login functionality

Test 1: Select the Register tab and create a new account

Result: A new account was created

Status: Passed

Details: A new account was created

Test 2: Enter the username and password

Result: Logged in to the account

Status: Passed

Details: This is done for the user to login to the account.

2. Search functionality for books

Test 1: Writes the name of the required book in the search bar

Result: Required Book appears

Status: Passed

Details: This is to done for the user to easily search the book they want to purchase.

3. Display of book information

Test 1: Selects the books

Result: Shows the book details, price, author, description and last modifications.

Status: Passed

Details: This is to done for the user to know more about the book they want to purchase.

4. Ability to modify the cart

Test 1: Selects the books and adds to the cart

Result: Adds the books to the cart

Status: Passed

Details: This is to done for the user to add the books to the cart they want to buy.

Test 2: Increases the book quantity

Result: Increases the book quantity and updates the price.

Status: Passed

Details: This is to done for the user when to buy the same book.

Test 3: Deletes the book from the cart

Result: Deletes the book from the cart and updates the price.

Status: Passed

Details: This is done when the user don't want to buy the book.

5. Checkout functionality

Test 1: Purchases the books

Result: Completes the purchase

Status: Passed

Details: This test case verifies that a user can successfully purchase one or more books from the website.

6. Order functionality

Test 1: Checking order status

Result: Order status is displayed correctly

Status: Passed

Details: This test case verifies that a user can check the status of their order.

7. Display of user profile information and Edit information

Test 1: Edit user profile information

Result: User information is updated correctly

Status: Passed

Details: This test case verifies that a user can successfully update their profile information, such as name, address, and contact details. The updated information should be saved and displayed correctly in the user's profile page.

Test 2: Password reset functionality

Result: User password is reset successfully

Status: Passed

Details: This test case verifies that a user can reset their password in case they forget it. They are able to change it. The new password should be saved and allow the user to access their account.

Test 3: Display user information

Result: User information is displayed correctly

Status: Passed

Details: This test verifies to display the user information correctly.

3.2 Performance Testing

3.2.1 Security Testing

Threat Analysis

As an online book shop, Boikini will be storing sensitive information such as customer names, email addresses, and shipping addresses. Additionally, the website will also handle payment information such as credit card numbers, which can be targeted by hackers. Therefore, the potential threat of a data breach is a major concern for Boikini.

Vulnerability Analysis

The website will be hosted on a secure server with up-to-date security protocols to prevent unauthorized access and data breaches. Boikini will also use secure payment gateways, such as PayPal, to handle all financial transactions and store payment information securely. Additionally, the website will undergo regular security audits to identify and address any potential vulnerabilities.

Security Threat from Website Permissions

Boikini will only require access to basic user information, such as name and email address, for the purpose of processing orders and providing customer support. No additional permissions will be requested from the user, ensuring the website does not have access to any unnecessary or sensitive information.

3.2.2 Timing Testing

The test was done on a desktop with a moderate configuration, which includes a ACPI x64 based pc with an Intel Core i7 processor and 16GB Ram. This is a good test device as it represents an average user's device, and not a high-end, high-performance device.

Loading the website

Time taken: 3 seconds

Logging in as a user

Time taken: 5 seconds

Browsing book categories

Time taken: 2 seconds per category

Placing an order

Time taken: 5 seconds

3.2.3 Volume Testing

The volume testing for Boikini was conducted on a moderate configuration device, which includes a ACPI x64 based pc with an Intel Core i7 processor and 16GB Ram. This is a good test device as it represents an average user's device, and not a high-end, high-performance device.

Initially, we did not insert a large amount of data into the database as it can be time-consuming. However, as we added more data, the website did not encounter any issues, and we can assume that our website will work well with more data.

But, as this is a web-based website, there may be limitations on the server's storage capacity, and if there is too much data, it may cause the app to slow down or even crash. It's important to monitor the app's performance and ensure that it can handle a large volume of data without any issues.

Additionally, we must consider the possibility of data loss due to damage to the server's memory or storage. To prevent this, we must ensure that we have regular backups and disaster recovery plans in place to recover lost data

in the event of any unexpected failures.

3.3 Acceptance Testing

3.3.1 Alpha Testing

The feedback gotten from mock clients (our friends) in the development environment made us discover quite a few bugs and inconsistencies in our system and we had to make some modifications and corrections based on those reviews. Those feedbacks and modifications of the alpha testing phase are described below:

No	Feedback	Modification
1	Some product descriptions were incomplete	All product descriptions were updated to include all necessary information
2	The website navigation was confusing for some users	The navigation menu was reorganized to make it more intuitive and user-friendly
3	The checkout process was slow and buggy	The checkout code was optimized to improve speed and fix any bugs
4	Some images were not loading properly on certain devices	All images were resized and compressed to ensure they load properly on all devices

These modifications were implemented to improve the functionality and user experience of the Boikini website. By addressing the feedback received, the website became more user-friendly, efficient, and accessible to all users.

3.3.2 Beta Testing

The feedback gotten from mock clients (our friends) after finishing our system made us discover quite a few bugs and inconsistencies in our system and we had to make some modifications and corrections based on those reviews. Those feedbacks and modifications of the beta testing phase are described below:

No	Feedback	Modification
1	The website was not loading properly on certain browsers	This was due to a compatibility issue with older browser versions. We updated the code to ensure compatibility with older versions.
2	The book categories were not displaying correctly on mobile devices (UI related problem)	Necessary changes were made to the code to ensure proper display of book categories on mobile devices
3	Customers were unable to pre-order certain books due to an error in the code	The issue was resolved by fixing the code responsible for handling pre-orders
4	The search functionality was not returning accurate results	This was due to a problem with the search algorithm. We updated the code to improve search accuracy.
5	Users were unable to make payments using certain payment methods	The issue was due to a problem with the payment gateway integration. We fixed the integration to enable payments using the affected payment methods.

These modifications were implemented to improve the functionality and user experience of the Boikini website. By addressing the feedback received, the website became more accessible and efficient for users.