A Project Report On

Software Testing and Quality Assurance
(Mini Project I)

SUBMITTED BY

Abhishek Sawalkar **Roll No:41402**

Sanket Bhatlawande **Roll No: 41415**

Samarth Bhadane **Roll No: 41414**

CLASS: BE-4

GUIDED BY

Prof K.C. Waghmare

DEPARTMENT OF COMPUTER ENGINEERING

PUNE INSTITUTE OF COMPUTER TECHNOLOGY

DHANKAWADI, PUNE-43

SAVITRIBAI PHULE PUNE UNIVERSITY

2020-21

Title:

Select the appropriate system environment/platform and programming languages to create a small application. Create a simple Test Plan that includes the features to be tested as well as a bug taxonomy. Create Test Cases that provide Test Procedures for the selected Test Scenarios. Using

appropriate testing tools, do selected black-box and white-box testing encompassing unit and integration tests. Prepare Test Reports based on Test Pass/Fail Criteria and assess the acceptability of the produced application.

Problem Definition:

Perform Desktop Application testing using unittest library in python.

Objective

We will learn how to create Test Cases that include Test Procedures for specified Test Scenarios.

Using appropriate testing tools, do selected black-box and white-box testing encompassing unit and integration tests.

Prepare test reports depending on the pass/fail criteria of the tests.

Theory

Test Plan for Application Testing

The Product Description, Software Requirement Specification SRS, or Use Case Documents are used to create the Test Plan document. The test is concerned with what to test, how to test, when to test, and who will test. The test plan document serves as a means of communication between the test team and the test management.

A common application testing test plan should include the following features:

- Define the scope of the testing.
- Define the testing aim.
- Methodology for assessing activity
- The testing schedule
- Tracking and reporting of bugs

UNIT TESTING : UNIT TESTING is a level of software testing where individual units/ components of a software are tested. The purpose is to validate that each unit of the software performs as designed. A unit is the smallest testable part of any software. It usually has one or a few inputs and usually a single output.

INTEGRATION TESTING : INTEGRATION TESTING is a level of software testing where individual units are combined and tested as a group. The purpose of this level of testing is to expose faults in the interaction between integrated units. Test drivers and test stubs are used to assist in Integration Testing

Extreme Programming & Unit Testing

In Extreme Programming, unit testing makes considerable use of testing frameworks. To generate automated unit tests, a unit test framework is utilised. Unit testing frameworks are not unique to extreme programming, although they are necessary. The following are some of the benefits that extreme programming provides to the field of unit testing:

• Tests are written prior to programming.

• Make extensive use of testing frameworks

• Every class in the apps has been tested.

• It is now possible to integrate quickly and easily.

Unittest library python

The unittest unit testing framework was inspired by JUnit and has a similar taste to other major unit testing frameworks. It allows for test automation, the sharing of test setup and shutdown code, the grouping of tests into collections, and the independence of the tests from the reporting system.

Application Tested :

All four key CRUD functions are supported by a library system application with a simple graphical user interface (GUI). Some test cases are used to validate these operations.

Test Plan:

The key goal is to see if we are getting the right result for each input and if the test is being run appropriately.

- We begin by providing a variety of inputs in order to conduct CRUD operations on the database.

- For our database, test cases involve authentication for various Author, Book name, ISBN code, and publishing year variables.

- For this, we test our file by opening a new tab and writing our test code for the various actions, assertEquals is utilised here.

- Finally, we receive test results that show the overall performance of our tests, i.e. test pass or test fail.

SAMPLE UNIT TESTS:

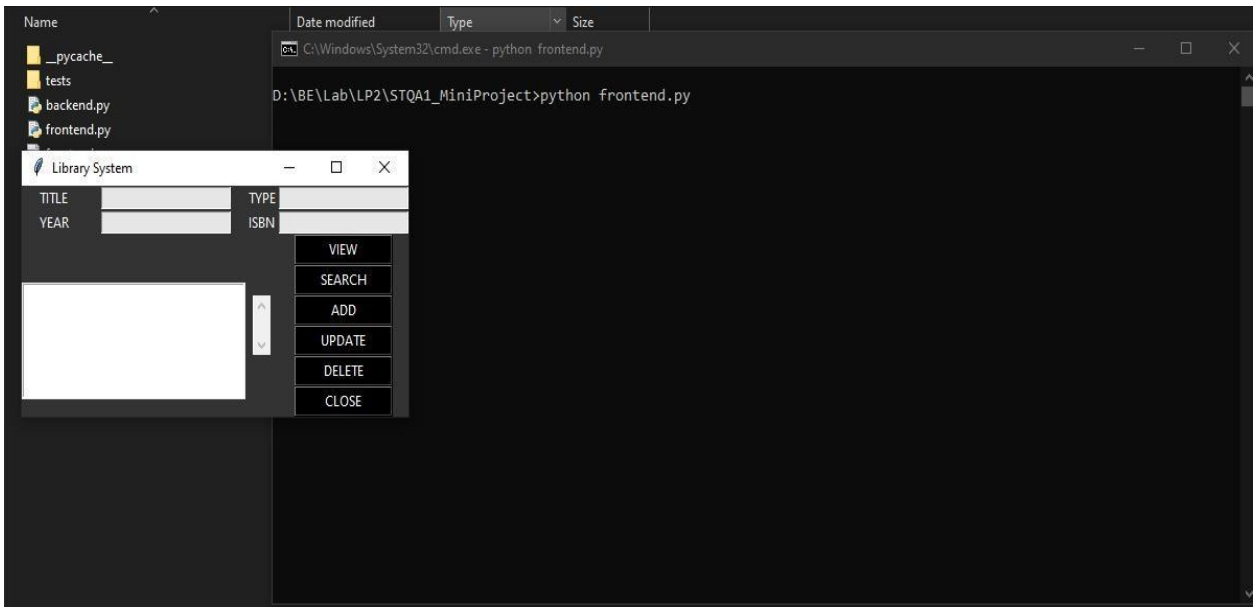
```
  def test_shortTitle_INSERT(self):         flag =
database.insert("t","Author","1976","IBN000cd")
      self.assertFalse(flag,"TITLE name must have more than 2 letters.")

   def test_shortAuthorName_INSERT(self):         flag =
database.insert("testbook","Au","1976","IBN000cd")
self.assertFalse(flag,"AUTHOR name must have more than 3 letters")

   def test_nonNumericYear_INSERT(self):
      flag = database.insert("testbook", "Author", "vfrz", "IBN000cd")
self.assertFalse(flag, "YEAR must be a Numeric Value.")

   def test_floatYear_INSERT(self):
      flag = database.insert("testbook", "Author", "1976.95", "IBN000cd")
self.assertFalse(flag, "Year can't be float value.")

   def test_emptyFields_INSERT(self):
flag = database.insert("", "", 1976, "")
      self.assertFalse(flag, "Blank arguments in insert function are  invalid.")
```

SAMPLE INTEGRATION TESTS:

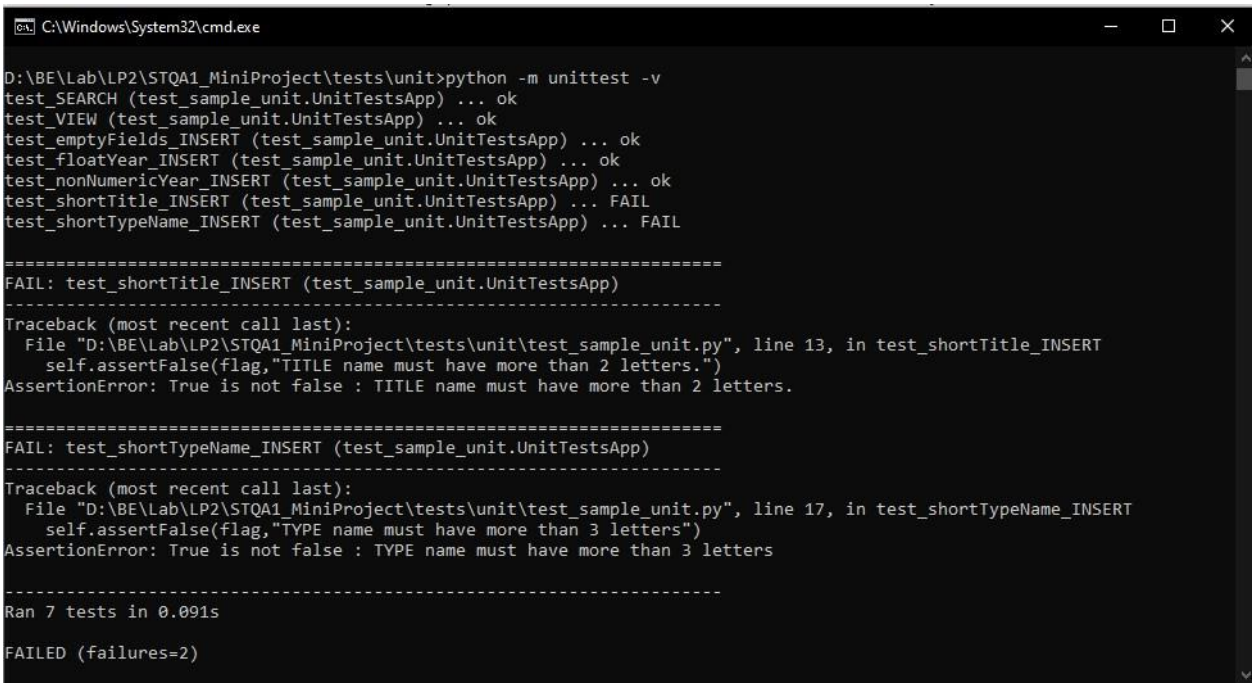
```
    def test_INSERT(self):
        flag = self.wrapper.insert(self.title,self.author,self.year,self.isbn)
self.assertTrue(flag,"INSERT OPERATION FAILED.")        print("INSERT
INTEGRATION PASS.")



    def test_DELETE(self):
        flag = self.wrapper.delete(self.title,self.author,self.year,self.isbn)
self.assertTrue(flag,"DELETE OPERATION FAILED.")
print("DELETE INTEGRATION PASS.")
```

Screenshots of application

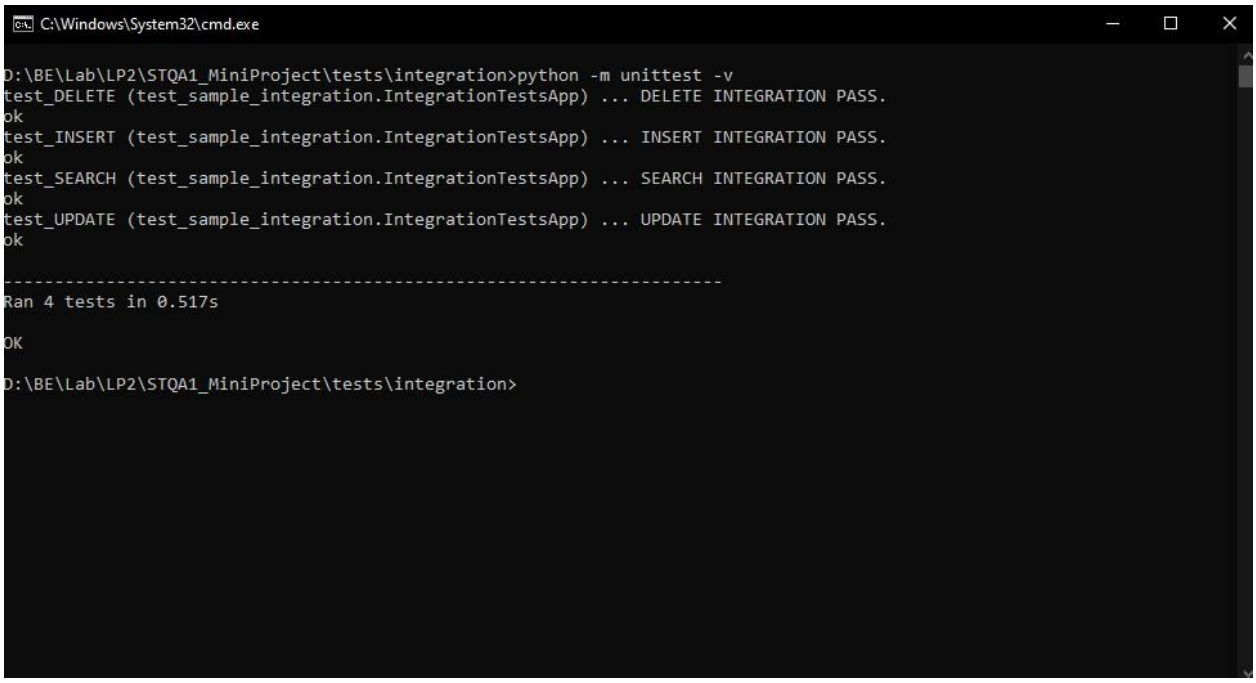
1 . GUI of the application



2 . Execution of unit tests



3. Execution of integration tests



Conclusion:

Hence, we have successfully performed Unit and Integration testing on a Bookstore application performing CRUD operations.

unit_test.py

```
import unittest  import sys  sys.path.insert(1,
'D:/BE/Lab/LP2/STQA1_MiniProject')  from backend
import Database
database = Database("../../library.db")

class UnitTestsApp(unittest.TestCase):



    def test_shortTitle_INSERT(self):
        flag = database.insert("t","testType","1976","IBN000cd")
self.assertFalse(flag,"TITLE name must have more than 2 letters.")

    def test_shortTypeName_INSERT(self):
        flag = database.insert("testbook","ty","1976","IBN000cd")
self.assertFalse(flag,"TYPE name must have more than 3 letters")

    def test_nonNumericYear_INSERT(self):
        flag = database.insert("testbook", "testType", "vfrz", "IBN000cd")
self.assertFalse(flag, "YEAR must be a Numeric Value.")

    def test_floatYear_INSERT(self):
        flag = database.insert("testbook", "testType", "1976.95", "IBN000cd")
self.assertFalse(flag, "Year can't be float value.")

    def test_emptyFields_INSERT(self):          flag = database.insert("", "",
1976, "")          self.assertFalse(flag, "Blank arguments in insert function are
invalid.")

    def test_VIEW(self):
        result = database.view()
self.assertTrue(len(result)>0,"Empty Database !!")

    def test_SEARCH(self):
        result = database.search("","","","")          self.assertFalse(result,"Atleast one search
parameter must be present for successful search.")

if __name__ == '__main__':
    unittest.main()
```

integration_test.py

```
import unittest  import sys  sys.path.insert(1,
'D:/BE/Lab/LP2/STQA1_MiniProject')  from backend
import Database
database = Database("../../library.db")
```

```
class Wrapper:

    def search(self,title="",author="",year="",isbn=""):
        database.insert(title,author,year,isbn)
rows = database.search(title)
database.delete(rows[0][0])          for row
in rows:              if(row[1]==title):
return True
        return False

    def insert(self,title,author,year,isbn):
database.insert(title,author,year,isbn)          rows =
database.search(title,author,year,isbn)
database.delete(rows[0][0])
if(len(rows)==1):              return True
        return False

    def delete(self,title,author,year,isbn):
database.insert(title,author,year,isbn)          rows =
database.search(title,author,year,isbn)           for
row in rows:
            database.delete(row[0])          rowsAfter =
database.search(title,author,year,isbn)
if(len(rowsAfter)==0):
            return True
        return False

    def update(self,updISBN,title,author,year,isbn):
        database.insert(title,author,year,isbn)          rowsBefore =
database.search(title)
database.update(rowsBefore[0][0],title,author,year,updISBN)
        rowsAfter = database.search(title)
database.delete(rowsBefore[0][0])          for row in rowsAfter:
if(row[0]==rowsBefore[0][0] and row[4]==updISBN):
return True
        return False



class IntegrationTestsApp(unittest.TestCase):

    wrapper = Wrapper()
title = "testbook"       type
= "testType"       year =
"1976"
    isbn = "ISBNtest"


    def test_SEARCH(self):
```

```
        flag = self.wrapper.search(self.title,self.type,self.year,self.isbn)
self.assertTrue(flag,"SEARCH OPERATION FAILED.")          print("SEARCH
INTEGRATION PASS.")

    def test_INSERT(self):
        flag = self.wrapper.insert(self.title,self.type,self.year,self.isbn)
self.assertTrue(flag,"INSERT OPERATION FAILED.")
print("INSERT INTEGRATION PASS.")



    def test_DELETE(self):
        flag = self.wrapper.delete(self.title,self.type,self.year,self.isbn)
self.assertTrue(flag,"DELETE OPERATION FAILED.")
print("DELETE INTEGRATION PASS.")

    def test_UPDATE(self):
        flag = self.wrapper.update("updatedISBN",self.title,self.type,self.year,self.isbn)
        self.assertTrue(flag,"UPDATE OPERATION FAILED.")
print("UPDATE INTEGRATION PASS.")




if __name__ == '__main__':
    unittest.main()
```

frontend.py

```
from tkinter import *  from
backend import  Database

database = Database("library.db")

class Window(object):
def __init__(self,window):
self.window = window
        self.window.wm_title("Library System" )

        l1 = Label(window, text="TITLE" , bg='#333333', fg='white')
l1.grid(row=0, column=0)

        l2 = Label(window, text="TYPE" , bg='#333333', fg='white')
l2.grid(row=0, column=2)

        l3 = Label(window, text="YEAR" , bg='#333333', fg='white')
l3.grid(row=1, column=0)
```

```
        l4 = Label(window, text="ISBN" , bg='#333333', fg='white')
l4.grid(row=1, column=2)

        self.title_text = StringVar()          self.e1 = Entry(window,
textvariable=self.title_text , bg='#e6e6e6')          self.e1.grid(row=0,
column=1)

        self.author_text = StringVar()
        self.e2 = Entry(window, textvariable=self.author_text , bg='#e6e6e6')          self.e2.grid(row=0, column=3)

        self.year_text = StringVar()          self.e3 = Entry(window,
textvariable=self.year_text , bg='#e6e6e6')          self.e3.grid(row=1,
column=1)

        self.ISBN_text = StringVar()          self.e4= Entry(window,
textvariable=self.ISBN_text , bg='#e6e6e6')          self.e4.grid(row=1,
column=3)

        self.list1 = Listbox(window, height=10, width=45)
        self.list1.grid(row=3, column=0, rowspan=6, columnspan=2)

        self.list1.bind('<<ListboxSelect>>', self.get_selected_row)

        # now we need to attach a scrollbar to the listbox, and the other direction,too
sb1 = Scrollbar(window)          sb1.grid(row=2, column=2, rowspan=6)
self.list1.config(yscrollcommand=sb1.set)
sb1.config(command=self.list1.yview)

        b1 = Button(window, text="VIEW", width=12, command=self.view_command , bg='black', fg='white')
b1.grid(row=2, column=3 )

        b2 = Button(window, text="SEARCH", width=12, command=self.search_command , bg='black', fg='white')
b2.grid(row=3, column=3)

        b3 = Button(window, text="ADD", width=12, command=self.add_command , bg='black', fg='white')
b3.grid(row=4, column=3)

        b4 = Button(window, text="UPDATE", width=12, command=self.update_command , bg='black', fg='white')
b4.grid(row=5, column=3)

        b5 = Button(window, text="DELETE", width=12, command=self.delete_command , bg='black', fg='white')
b5.grid(row=6, column=3)

        b6 = Button(window, text="CLOSE", width=12, command=window.destroy , bg='black', fg='white')
b6.grid(row=7, column=3)


    def get_selected_row(self,event):   #the "event" parameter is needed b/c we've binded this function to the listbox
try:
            index = self.list1.curselection()[0]
self.selected_tuple = self.list1.get(index)
```

```
self.e1.delete(0,END)
self.e1.insert(END,self.selected_tuple[1])
self.e2.delete(0, END)
self.e2.insert(END,self.selected_tuple[2])
self.e3.delete(0, END)
self.e3.insert(END,self.selected_tuple[3])
self.e4.delete(0, END)
self.e4.insert(END,self.selected_tuple[4])          except
IndexError:
          pass               #in the case where the listbox is empty, the code will not execute

   def view_command(self):
      self.list1.delete(0, END)  # make sure we've cleared all entries in the listbox every time we press the View all b
utton          for row in database.view():               self.list1.insert(END, row)

   def search_command(self):          self.list1.delete(0, END)          for row in database.search(self.title_text.get(),
self.author_text.get(), self.year_text.get(), self.ISBN_text.get()):
          self.list1.insert(END, row)

   def add_command(self):          database.insert(self.title_text.get(), self.author_text.get(), self.year_text.get(),
self.ISBN_text.get())          self.list1.delete(0, END)          self.list1.insert(END, (self.title_text.get(),
self.author_text.get(), self.year_text.get(), self.ISBN_text.get()))

   def delete_command(self):
database.delete(self.selected_tuple[0])
      self.view_command()

   def update_command(self):
      #be careful for the next line ---> we are updating using the texts in the entries, not the selected tuple
       database.update(self.selected_tuple[0],self.title_text.get(), self.author_text.get(), self.year_text.get(), self.ISBN_
text.get())          self.view_command()

#code for the GUI (front end)
window = Tk()
Window(window)

window.configure(bg='#333333')
window.geometry("500x300")  window.mainloop()
```

backend.py

```
import sqlite3  class
Database:      def
__init__(self,db):
      self.conn = sqlite3.connect(db)
self.cur = self.conn.cursor()
      self.cur.execute("CREATE TABLE IF NOT EXISTS book (id INTEGER PRIMARY KEY, title TEXT, "
```

```
                "author TEXT, year INTEGER, isbn INTEGER)")
self.conn.commit()

    def insert(self,title, author, year, isbn):
        #the NULL parameter is for the auto-incremented id
if(len(title)==0 or len(author)==0 or len(year)==0 or len(isbn)==0):
            return False
if(not year.isdigit()):
            return False
        self.cur.execute("INSERT INTO book VALUES(NULL,?,?,?,?)", (title,author,year,isbn))
        self.conn.commit()
        return True



    def view(self):
        self.cur.execute("SELECT * FROM book")
rows = self.cur.fetchall()          return rows

    def search(self,title="", author="", year="", isbn=""):          if(len(title)==0
and len(author)==0 and len(year)==0 and len(isbn)==0):               return False
        self.cur.execute("SELECT * FROM book WHERE title = ? OR author = ? OR year = ? "
                "OR isbn = ?", (title, author, year, isbn))
rows = self.cur.fetchall()
        #conn.close()
        return rows

    def delete(self,id):
        self.cur.execute("DELETE FROM book WHERE id = ?", (id,))
self.conn.commit()
        return True
        #conn.close()

    def update(self,id, title, author, year, isbn):          if(len(title)==0 or
len(author)==0 or len(year)==0 or len(isbn)==0):
            return False
if(not year.isdigit()):
            return False          self.cur.execute("UPDATE book SET title = ?, author = ?, year = ?, isbn = ? WHERE id =
?", (title, author, yea r, isbn, id))          self.conn.commit()
        return True

    #destructor-->now we close the connection to our database here
def __del__(self):          self.conn.close()
```