Department of Computer Engineering

Academic Term: First Term 2023-24

Class: T.E /Computer Sem – V / Software Engineering

Practical No:	8
Title:	**Design test cases for performing black box testing**
Date of Performance:	
Roll No:	9567, 9552, 9623
Team Members:	Shruti Patil, Mrunal Kotambkar, Dhruv Mayekar

Rubrics for Evaluation:

Sr. No	**Performance Indicator**	**Excellent**	**Good**	**Below Average**	**Total Score**
1	On time Completion & Submission (01)	01 (On Time)	NA	00 (Not on Time)	
2	Theory Understanding(02)	02(Correct)	NA	01 (Tried)	
3	Content Quality (03)	03(All used)	02 (Partial)	01(rarely followed)	
4	Post Lab Questions (04)	04(done well)	3 (Partially Correct)	2(submitted)	

Signature of the Teacher:

Department of Computer Engineering

Academic Term: First Term 2022-23

Class: T.E /Computer Sem – V / Software Engineering Signature of the Teacher:

Black Box testing techniques are:

Code:

```
class Item:
    def __init__(self, item_name, item_description, selling_price):
        self.item_name = item_name
        self.item_description = item_description
        self.selling_price = selling_price

    def display_details(self):
        print("Item Name:", self.item_name)
        print("Item Description:", self.item_description)
        print("Selling Price:", self.selling_price)


def get_item_details():
    item_name = input("Enter item name: ")
    item_description = input("Enter item description: ")
    selling_price = float(input("Enter selling price: "))
    return item_name, item_description, selling_price
def main():
    num_items = int(input("Enter the number of items to sell: "))

    items = []
    for i in range(num_items):
        print(f"\nEnter details for item {i + 1}:")
        item_name, item_description, selling_price = get_item_details(
        item = Item(item_name, item_description, selling_price)
        items.append(item)

    print("\nDetails of the items:")
    for i, item in enumerate(items, start=1):
        print(f"\nDetails for item {i}:")
        item.display_details()


if __name__ == "__main__":
    main()
```

```
import re

def get_payment_details():
    card_number = input("Enter your card number: ")
    card_expiry = input("Enter your card expiry (MM/YY): ")
    card_cvv = input("Enter your card CVV: ")

    return card_number, card_expiry, card_cvv

def validate_card_number(card_number):
    # Basic check: Card number should be 16 digits
    return re.match(r'^\d{16}$', card_number)

def validate_card_expiry(card_expiry):
    # Basic check: Expiry should be in MM/YY format
    return re.match(r'^\d{2}/\d{2}$', card_expiry)

def validate_card_cvv(card_cvv):
    # Basic check: CVV should be 3 digits
    return re.match(r'^\d{3}$', card_cvv)

def process_payment():
    card_number, card_expiry, card_cvv = get_payment_details()

    if validate_card_number(card_number) and \
      validate_card_expiry(card_expiry) and \
      validate_card_cvv(card_cvv):
        print("Payment successful! Proceeding to the payment page.")
    else:
        print("Invalid payment details. Please check your payment information.")

if __name__ == "__main__":
    process_payment()
```

1. Equivalence Partitioning:

It is the black-box technique that divides the input domain into classes of data from which test cases can be derived. Equivalence partitioning defines test cases that uncover classes of errors thereby reducing the no. of test cases that must be developed. If the input condition specifies a range, one valid and two invalid equivalence classes are defined. If an input condition requires a specific value, one valid and two invalid equivalence classes are defined. If an input condition specifies a member of a set, one valid and one invalid equivalence class is defined. If an input condition is boolean, one valid and one invalid equivalence class is defined. By applying these guidelines, test cases for each input domain can be developed.

E.g. A program that selects card number length as 16 digit so it will have 3 partitions as

i. Card number length less than 16 digit

ii. Card number length equal to 16 digit

iii. Card number length more than 16 digit

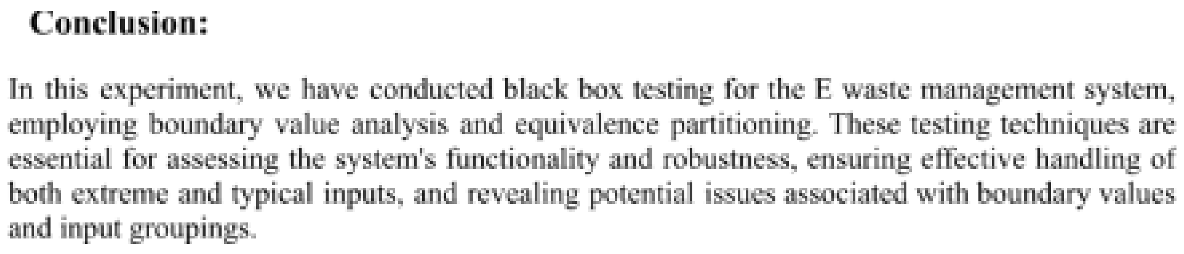
Similarly the CVV number should be 3 digits only here also there will be three partitions

- CVV less than 3 digits
- CVV equal to 3 digits
- CVV greater than 3 digits

2. Boundary Value Analysis:

Boundary Value Analysis (BVA) in black box testing is a technique that tests values on the edges of valid input ranges. It aims to identify potential errors at boundaries as they are more likely to trigger defects.

BVA complements Equivalence Partitioning and helps ensure thorough testing by focusing on critical boundary values for input parameters or conditions. BVA assesses values just inside and outside the valid range to uncover potential software issues, making it an efficient and effective testing method for black box testing.



Conclusion:

In this experiment, we have conducted black box testing for the E waste management system, employing boundary value analysis and equivalence partitioning. These testing techniques are essential for assessing the system's functionality and robustness, ensuring effective handling of both extreme and typical inputs, and revealing potential issues associated with boundary values and input groupings.

POSTLAB:

Evaluate the effectiveness of black box testing in uncovering defects and validating the software's functionality, comparing it with other testing techniques.

Black box testing is a software testing technique that focuses on evaluating the functionality of a software application without examining its internal code or structure. It is often compared with other testing techniques, primarily white box testing, to assess its effectiveness in uncovering defects and validating software functionality. Here's an evaluation of the effectiveness of black box testing in comparison to other testing techniques:

1. Effectiveness in Uncovering Defects:

a. Black Box Testing:

- Pros:

- Emulates the user's perspective, making it effective in finding user-related defects.
- Can uncover issues related to functionality, usability, and integration. • Testers do not need access to the source code, making it suitable for testing third-party software.

• Cons:
- Limited ability to uncover low-level or structural defects.
- May miss complex integration issues or security vulnerabilities.

b. White Box Testing:

• Pros:
- Offers deep code-level insights, making it effective for identifying logic errors, security vulnerabilities, and performance issues.
- Can be used to create test cases that target specific code paths.

• Cons:
- Requires access to the source code.
- May not adequately test user-level functionality and usability.

2. Effectiveness in Validating Software Functionality:

a. Black Box Testing:

• Pros:
- Validates software functionality from a user's perspective.
- Helps ensure that the software meets user requirements and behaves as expected.

• Cons:
- Limited in its ability to verify the internal correctness of the software.

b. White Box Testing:

• Pros:
- Validates internal code logic and verifies that the software operates correctly at the code level.
- Effective in verifying the integrity of algorithms and data structures. •

Cons:
- Does not directly address user-level functionality and may not guarantee that user requirements are met.

3. Integration Testing:

• Integration testing is another technique that focuses on testing the interactions between different components or modules of a software system. It ensures that these components work together as expected. It can be used in conjunction with both black box and white box testing to verify the correct functioning of integrated parts.

Assess the challenges and limitations of black box testing in ensuring complete test coverage and discuss strategies to overcome them.

Black box testing, while effective in many scenarios, has its share of challenges and limitations when it comes to ensuring complete test coverage. Test coverage refers to the extent to which a software application has been tested. Here are some challenges and strategies to overcome them:

Challenges:

1. Limited Visibility: Testers lack visibility into the internal code, which makes it challenging to design test cases for specific code paths and complex logic.
2. Incomplete Test Coverage: Since testers are not aware of the internal code structure, they may miss some code paths or edge cases during testing.
3. Ambiguity in Requirements: Ambiguous or incomplete requirements can lead to incomplete test cases and coverage.
4. Time and Resource Constraints: Limited time and resources can result in prioritizing certain test cases over others, leaving some untested.
5. Complex Systems: For complex systems with numerous interactions and dependencies, achieving comprehensive test coverage can be extremely challenging.

Strategies to Overcome These Challenges:

1. Requirement Analysis: Ensure a thorough understanding of the requirements. Collaborate with developers and stakeholders to clarify ambiguities and ensure that all requirements are well-documented.
2. Equivalence Partitioning: Use equivalence partitioning to group input data into classes with similar behavior. This helps reduce the number of test cases needed to cover various scenarios.

3. Boundary Value Analysis: Pay special attention to boundary values, as these are often where defects are found. Test values at the lower and upper boundaries of input ranges. 4. Exploratory Testing: Conduct exploratory testing sessions where testers use their creativity and domain knowledge to uncover defects. This can help discover issues that scripted test cases might miss.

5. Use of Test Design Techniques: Employ structured test design techniques like decision tables, state transition diagrams, and use case testing to ensure test cases cover various scenarios comprehensively.

6. Risk-Based Testing: Prioritize test cases based on risk assessment. Focus testing efforts on areas of the application that are most critical and likely to contain defects. 7. Code Reviews and Static Analysis: While not part of black box testing, code reviews and static analysis by developers can help identify issues before they propagate into the testing phase.

8. Pair Testing: Collaborate closely with developers in a pair testing approach, where testers and developers work together to design test cases and validate the functionality. 9. Regression Testing: Continuously perform regression testing to ensure that changes made in one part of the software do not introduce defects in other areas. 10. Test Automation: Implement test automation to run repetitive and labor-intensive test cases consistently. Automation can help achieve higher coverage and detect regression issues quickly.

11. Traceability Matrix: Create a traceability matrix to map test cases to requirements, ensuring that all requirements are covered by corresponding test cases. 12. Crowdsourced Testing: Consider leveraging crowdsourced testing platforms to involve a diverse set of testers from different backgrounds and perspectives. They can uncover issues that might be overlooked by in-house teams.