University “Dzemal Bijedic” Mostar

Faculty of information technologies

TESTING TECHNIQUES

Seminar paper on the subject of Formal methods

Amina Gačević

Mostar, January 2025

Contents

Introduction

This introduction covers the basic information about software testing as an essential segment throughout the software development cycle, as well as information about the web application that will be used to demonstrate different testing techniques and the objectives of this seminar paper.

Software testing represents a crucial phase in the development process, aimed at ensuring the quality and correctness of applications before their distribution to users. Given the complexity of modern software systems, testing has become an indispensable part of every development project, as it helps identify bugs, reduce risks, and ensure the reliability of applications. In the context of formal methods, testing is not only a pragmatic approach but also an analytical one, using various techniques and approaches to thoroughly examine the system's functionalities.

In the following sections, students will apply the knowledge they gained through the course on Formal Methods, which was conducted during the 2024/2025 academic year, to a specific example – the web application https://www.univerzalno.com/. The functionality that will be tested using different testing techniques in this seminar paper is the "order completion" process, which occurs when a user adds desired products to their shopping cart and proceeds to place an order via the web application. This functionality provides a rich environment for applying various testing techniques, as it allows for the exploration and evaluation of the system to ensure its correctness, security, and performance. It is important to note that this functionality pertains to users who do not have a created user profile!

Throughout the course, students became familiar with the basic techniques of testing software systems, including black-box testing, white-box testing, and experience-based testing techniques. Black-box testing focuses on testing the functionality of the software without insight into its internal structure, while white-box testing requires a detailed understanding of the internal logic and structure of the application. On the other hand, experience-based testing relies on the tester's previous knowledge and intuition, using their experience to predict potential issues and weaknesses in the application.

The goal of this paper is to demonstrate various testing techniques on a concrete example and to show that students have successfully mastered the material outlined in the course syllabus.

Note: Within the various testing techniques for input fields such as phone numbers and postal codes, we were unable to include them because the form itself lacks the appropriate validations needed to cover the testing techniques required in the task.

1. Equivalence Partitioning

Equivalence partitioning is one of the black-box testing techniques where the equivalent partition divides data into partitions in such a way that all members of a given partition are expected to be processed in the same manner. There are equivalence partitions for both valid and invalid values.

Parameter	Equivalence Partitioning	Valid/Invalid	Input Value
Email input field	Valid email format	Valid	test@domain.com
	Invalid email format (missing @ symbol)	Invalid	test
	Invalid email format (multiple @ symbols)	Invalid	test@test@
	Invalid email format (missing domain)	Invalid	test.com
	Invalid email format (special characters in domain)	Invalid	test@domain#com
	Invalid email format (spaces in address)	Invalid	test@ domain.com

Table 1

What was done?

1. **Identification of the parameter for testing:**
   As an example of using this technique, the "email" field was selected, where the user enters a value when filling out the order completion form.
2. **Division of input values into valid and invalid partitions**
3. **Definition of the input values for each partition:**
   - Valid email address: test@domain.com
   - Invalid addresses: test (no @), test@ (no domain), test@domain#com (special characters), etc.
4. **Creation of a parameter table:**
   The table was created in a way that clearly displays: the parameter being tested, equivalence partitions, input validity, and the specific input value for each case.

Why was task done this way?

1. Reduction of the number of tests, saving time and resources, while ensuring high testing efficiency.
2. Optimization of time and resources, enabling a high degree of coverage with fewer tests.
3. Coverage of key scenarios, where only the most important input cases are tested using this technique, with each representing a broader set of possible inputs (partitions).

The input data was divided into six partitions based on whether they are valid or invalid, and the equivalence partitioning technique is based on the idea that if we test just one element from each partition, it can represent the entire group of data.

The valid partition includes an example of a correctly formatted email address, while the invalid partitions include examples where the email address was not entered correctly.

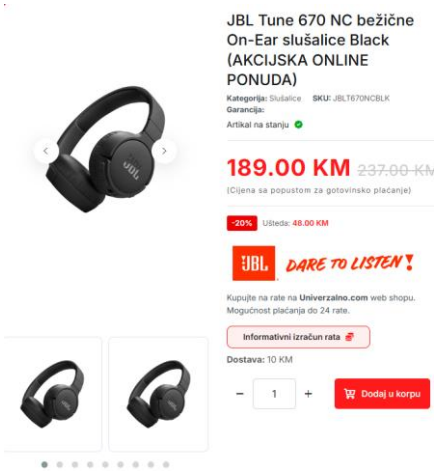
2. Boundary Value Analysis

Boundary Value Analysis is a testing technique that focuses on testing values that lie at the **boundaries** of input ranges or domains.

We will analyze the functionality related to adding products to the shopping cart, where the user modifies the product quantity.

Note: Depending on the type of product the customer selects, the maximum quantity varies.

We will specifically consider the case when the user selects the following headphones:



Picture 1

Parameter	**Equivalence Partition**	**Valid/Invalid**	**Boundary Values**
Product Quantity	Valid quantity (1 to 3)	Valid	1 (v), 3 (v)
	Less than the minimum quantity (< 1)	Invalid	0 (i)
	Greater than the maximum quantity (> 3)	Invalid	4 (i)
	Invalid input (non-numeric, special characters, empty)	Invalid	"." (i)
			*(v) - valid
			*(i) - invalid

Table 2

What was done?

1. **Identification of the parameter for testing:**
   The "količina" field was selected to test this technique.
2. **Division into equivalence partitions and boundary values analysis:**
   - The input data was divided into partitions (groups) based on their validity (valid and invalid).
   - For each partition, test cases were identified to cover key boundaries.

3. **Creation of table:**
   - Once the parameter, equivalence partitions, and boundary values were defined, a table was created to ensure the test cases were clear and comprehensive.

Why was the task done this way?

1. **Testing efficiency:**
   Using boundary value analysis and equivalence partitioning allows for the identification of key errors with a minimal number of test cases. These techniques focus on the most critical input values, where errors are most likely to occur.
2. **Test coverage:**
   Testing values at the boundaries (e.g., 1 and 3) ensures that the system correctly handles minimum and maximum values, while testing values outside the boundaries (e.g., 0 and 4) checks whether the system properly recognizes invalid inputs.

3. Decision Table Testing

Decision tables are a good way to document complex business rules that the system must implement.

		Case 1	Case 2	Case 3	Case 4
Case	The user filled out the form with the delivery information	TRUE	TRUE	FALSE	FALSE
Case	User accepted Terms of Use and Conditions	TRUE	FALSE	TRUE	FALSE
Action	Complete the purchase	TRUE	FALSE	FALSE	FALSE

Table 3

What was done?

A simple decision table was created with the conditions that need to be met for the user to complete the order. Given that the data entry form is complex for testing this technique, it was considered best to focus on two key conditions the user must fulfill:

- The user must enter delivery details in the form.
- The user must accept the Terms and Conditions.

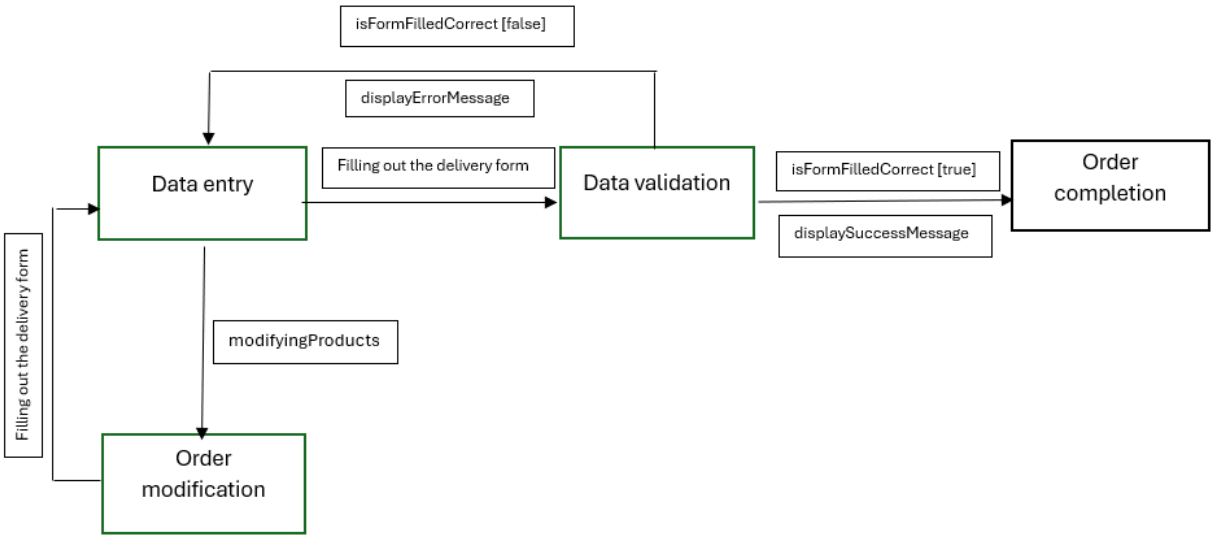
Based on these two inputs, the action (output) "Završi kupovinu" was defined. The table was created in such a way that based on the number of conditions, we created the number of cases as $2^{\text{number of conditions}}$. This means that our decision table has four cases. We then proceeded to fill out the table by placing true values in half of the cases in the first row and false values in the other half. The second row was filled by alternating true-false values

because, in the first row, there are two true and false values, and in the second row, we reduce them by half.

Why was the task done this way?

This technique is suitable for cases where there are not many conditions (input data). This way, we have all combinations of input values, and each test case represents a unique combination of input values. For the action, we calculate what the output should be. If the input values do not match the expected values, a bug is reported.

4. State Transition Testing



	Data entry	Data validation	Order modification	Order completion
Filling out the delivery form	Data validation	-	Data entry	-
isFormFilledCorrect [true]	-	Order completion	-	-
isFormFilledCorrect [false]	-	Data entry	-	-
modifyingProducts	Order modification	-	-	-
displaySuccessMessage	-	Order completion	-	-
displayErrorMessage	-	Data entry	-	-

Table 4

What was done?

First, a state transition diagram was drawn for the functionality related to completing the order, where the user can modify the products they added to the cart and fill out the delivery

form.
We identified the states and transitions that lead from one state to another. Then, guards, events, and actions were defined as integral parts of the diagram.

The initial state represents "Data entry." This is the state in which the user enters delivery details and can simultaneously modify the order (Order modification) by changing the product quantity or removing the product from the cart. Once the user enters all the data and clicks the "Završi kupovinu" button, validation ("Data validation") is performed on the mandatory fields. If the user has not entered any of the required values, an appropriate message will appear on the screen, and the user will return to the "Data entry" state to complete the delivery form.

In our example, the guards are "isFormFilledCorrect [true]" and "isFormFilledCorrect [false]." In the second part of the task, test cases are listed in a state transition table, where the number of columns represents the number of states, and each row represents the events and actions covered by this example. The table shows the relationship between states and inputs.

Why was the task done this way?

This example represents a simple process that contains all the basic elements of a state diagram (states, transitions, guards, events, and actions) and can easily be visualized and linked to the functionality on the page.

5. Statement Testing and Coverage

```
function validateDeliveryData(Ime, Email, Telefon,
PostanskiBroj, Grad, Adresa){
    if Ime is empty:
    return "Please fill out this field."

    if Email is empty:
    return "Please fill out this field."

    if Telefon is empty:
    return "Please fill out this field."

    if PostanskiBroj is empty:
    return "Please fill out this field."

    if Grad is empty:
    return "Please fill out this field."

    if Adresa is empty:
    return "Please fill out this field."

    return "Success: Data is valid"
}
```

Test case	Input values	Expected result
TC1	Ime="", Email="user@example.com", Telefon="061123456", PostanskiBroj="12345", Grad="Sarajevo", Adresa="ABC"	"Please fill out this field."
TC2	Ime="Ime", Email="", Telefon="061123456", PostanskiBroj="12345", Grad="Sarajevo", Adresa="ABC"	"Please fill out this field."
TC3	Ime="Ime", Email="user@example.com", Telefon="", PostanskiBroj="12345", Grad="Sarajevo", Adresa="ABC"	"Please fill out this field."
TC4	Ime="Ime", Email="user@example.com", Telefon="061123456", PostanskiBroj="", Grad="Sarajevo", Adresa="ABC"	"Please fill out this field."
TC5	Ime="Ime", Email="user@example.com", Telefon="061123456", PostanskiBroj="12345", Grad="", Adresa="ABC"	"Please fill out this field."
TC6	Ime="Ime", Email="user@example.com", Telefon="061123456", PostanskiBroj="12345", Grad="Sarajevo", Adresa=""	"Please fill out this field."
TC7	Ime="Ime", Email="user@example.com", Telefon="061123456", PostanskiBroj="12345", Grad="Sarajevo", Adresa="ABC"	"Success: Data is valid"

Table 5

What was done?

In this example, we aim to cover statements using the fewest possible test cases. Within the pseudocode, each individual field that the user must fill out to successfully complete the purchase was tested. The focus of this technique is on statements that represent a single line of code and individual statements of the test object. If the user does not enter a value in any of these fields, an appropriate message will be displayed. The point of this technique is to identify test cases that execute a predefined percentage or all of the program's executable statements.

The table displays test cases based on pseudocode with input parameters and expected results.

Why was the task done this way?

In the pseudocode, each check is separated into individual conditions, which makes it easier to identify and select errors. This example allows for expansion by adding new validations without disrupting the current structure. Regarding the test cases, TC1-TC6 ensure that no individual field can be left empty, and TC7 checks the successful entry of all data.

6. Decisions Testing and Coverage

```
function validateOrder
(formaPopunjena,nacinDostave,drugaAdresaZaDostavu=None,
usloviKoristenja, proizvodDostupan, ):

    if not formaPopunjena:
        return "Please fill out this field."

    if not usloviKoristenja:
        return "Please check this box if you want to proceed"

    if nacinDostave == "Alternate Address":
        if drugaAdresaZaDostavu is None or
drugaAdresaZaDostavu.strip() == "":
            return "Order rejected: Alternate address cannot be empty"

    if not proizvodDostupan:
        return "Order rejected: Item not in stock"

    return "Order accepted"
```

Test case	Input values	Expected result
TC1	Form not filled properly	Please fill out this field.
TC2	Terms and conditions not accepted	Terms and conditions must be accepted
TC3	Alternate address =" "	Alternate address cannot be empty
TC4	No items in stock	Item not in stock
TC5	Valid data	Order accepted

Table 6

What was done?

In this technique, we examined whether the user fulfilled the conditions that come after filling out the form for valid order completion. The *validateOrder* function accepts parameters: *formaPopunjena*, *nacinDostave*, *drugaAdresaZaDostavu*, *usloviKoristenja*, and *proizvodDostupan*. If the user hasn't filled in all the mandatory fields, they will receive an

appropriate warning message. Then, if the user hasn't accepted the Terms and Conditions, they won't be able to complete the order. The next condition involves checking the delivery method for cases where the user selects a different delivery address. If the second address is not provided, the user should not be able to complete the order. The strip() function in Python removes leading and trailing spaces from a string. Additionally, it should not be possible for the user to order a product that is out of stock. Only when all previous conditions are met should the order be approved.

For the test cases, based on the pseudocode, we defined five possible scenarios. The first assumes that the user enters values in all fields required by the delivery form. The second scenario occurs when the user doesn't accept the Terms and Conditions, in which case an appropriate message will be displayed because the user cannot complete the order without accepting them. The next case involves when the user does not enter the delivery address details if they have chosen delivery to another address. The fourth case occurs when the user selects a product that is out of stock, and therefore cannot complete the order. Finally, the last test case involves the user fulfilling all the previous conditions, and the order is approved.

Why was the task done this way?

This testing technique represents an advanced criterion for white-box testing, where the decisions within the original code are the focus of the testing process. The task was done this way to validate all key inputs and cover every possible scenario with test cases. The effect of each decision is assessed, and as a result, it is determined which statement(s) should be executed next.

7. Error guessing

No	Potential error	Test case
1	System allows user to choose negative quantity of product.	Choose negative quantity of product.
2	System doesn't update price of order according to quantity.	Modify quantity of product.
3	System doesn't remove product from order form.	Remove product from order.
4	System allows user to add letters in quantity of product.	Add some letter in quantity of product.
5	System allows user to modify price of order.	Modify price of order.
6	System allows user to use special characters in name field.	Use special character in name field.
7	System doesn't processes special characters which can lead to unespected behaviour.	Use special characters in email field.
8	System process white space in email field which can cause unespected problems.	Use white space in email field.
9	System processes non-number values in phone field.	Use letters to add phone number.
10	System processes non-number values in postal code field.	Use letters to add postal coder.
11	System processes special characters in city field where they are not desirable.	Use special characters in city field.
12	System fails to process exceptionally long entries which can cause problems like overflow.	Enter too long note in note field.
13	System allows user to finish order without entering data in additional address field in case user chose delivering on second address.	Finish order without entering data in additional address field.
14	System allows user to finish order without entering data in mandatory field.	Clik on "Zavrsi kupovinu" without entering data in one of mandatory field.
15	System allows user to finish order without accepting terms of use.	Finish order without checking terms of use checkbox.

Table 7

What was done?

Error guessing is a software testing technique where tests are created based on experience and knowledge of common errors, targeting possible defects in the system. In this example, we aimed to cover as many potential errors as possible for the functionality related to completing an order. We considered cases that could arise from incorrect data entry, errors caused by not adhering to boundary values, the use of special characters where not allowed, and improper behavior of the application under specific circumstances.

We also designed tests to target possible "bugs," where each test case was clearly specified.

Why was the task done this way?

Within this technique, based on the tester’s experience, we created a list of bugs and failures that could occur, derived from experience and general reasons why software might fail. Ultimately, this technique allows for deeper analysis and the identification of new potential bugs.

8. Exploratory testing

Finally, let's talk about exploratory testing. Exploratory testing is a technique where testing is performed without predefined test cases, relying on the tester's experience and intuition.

How would we organize exploratory testing?

First, we would define the goals of the testing, ensuring that the focus is on validating different inputs and assessing system stability in the case of unexpected or extreme inputs. Then, we would review the application documentation to better understand the expected system behavior and analyze previously found issues to identify potential vulnerabilities within the system. We would allocate time for testing and prepare tools for documenting the testing process. After that, we would use various testing techniques to execute pre-prepared tests. We would test input fields, the system's behavior when modifying product quantities, and the functionality of the "Complete Purchase" button.

Which type of exploratory testing would we use and why?

In terms of the type of exploratory testing, based on all the previously mentioned points, we believe that the best approach would be *Chartered exploratory testing*. This type of testing allows for the definition of testing goals and timeframes and is used to focus on specific functionality while tracking the progress of testing through documentation. Considering the importance of documentation throughout the testing process, we believe that this type of testing is ideal when we want to document and analyze the testing process itself.

We believe that this type of testing offers high efficiency and flexibility while ensuring that the system is thoroughly tested, revealing potential issues.