**Assignment 1**

To start developing software for clients, one would intuitively start with carrying out these following tasks:

1) Find out about requirements: during this phase, we try to gather as much information from the clients to be able to find out what their requirements and needs are.

2) Outlay the design: here we try to plan the programming language that will be used for the software.

3) Implementation of the code: during this phase, we start to code the software.

4) Testing of the code: and lastly, we test the software to assure it is in accordance with the user’s requirements and needs.

5) Maintenance.

This model is called the waterfall model and constitutes the software development life cycle. It has a very intuitive flow to it, where the tasks are executed in a sequential fashion: we start at the top of the waterfall and flow down to the design phase, which flows into the build phase and finally into the testing phase.

This model is very old and has some flaws to it. In fact, since it is easy to make errors during the earlier phases, the following phases will also be affected by it, which will result in a completely different finished product. Those errors are also more prevalent in the earlier phases, such as during the requirements or design phases. Such errors can be quite costly to fix, as you will have to start from the beginning if something like that were to happen. Moreover, the cost to fix them will be a lot higher as time goes on in the development life cycle.

To address the concerns of this model, a newer model has been developed called the “V-model”. In this model, every phase in the development life cycle is attributed to a corresponding testing phase. These testing phases are called the “software testing life cycle” and are as follows:

1) User requirements -> preparation acceptance test -> acceptance test execution.

2) System requirements -> preparation system test -> system test execution.

3) Global design -> preparation integration test -> integration test execution.

4) Detailed design -> component test execution.

5) Implementation.

Moreover, with time, several testing principles have been suggested over the years and offer a general guideline for testers to follow. These principles are as follows:

Principle 1 – Testing shows the presence of defects: testing shows the presence of defects but cannot prove that there are no defects.

Principle 2 – Exhaustive testing is impossible: testing everything is not possible.

Principle 3 – Early testing: the testing activities should start as early as possible in the software development life cycle.

Principle 4 – Defect clustering: most of the defects found during testing are related to a small number of modules.

Principle 5 – Pesticide paradox: if the same test cases are executed over and over, then these tests will no longer be able to identify new defects in the system.

Principle 6 – Testing is context dependent: testing is done differently in different contexts.

Principle 7 – Absence-of-errors fallacy: testing needs to be done on the right requirements which are on par with the user’s needs.

Now that we have looked at the V-model and the ISTQB’s seven principles, we will now show how the “A Ticket Machine” system can be tested at the testing levels in the V-model and how they relate to the seven principles:

1) Unit testing:

In this testing phase, individual units or components of a software are tested independently. Here we can test the following:

- Looking up the ticket prices.

- Buying a 7-day ticket.

- Check the balance of the card.

Such individual tests cannot cover every possible issue that can occur in this system, because we cannot test everything (principle 1). That is why we need to move to more vast testing methods. It is also the earliest testing phase (principle 3), and this is usually where most of the defects are found in a small number of modules (principle 4).

2) Integration testing:

In this phase, we ensure that the system functions across all components integrations. As such, we test the logical link between the units. We can test the following:

- Recharge the travel card with 100 NOK. Check the balance of the card to see the change.

- Register a trip with a rechargeable card and observe that the balance of the card has been reduced with the cost of the trip.

Again, with the integration testing, we see a return of the principles from the unit testing phase. Exhaustive testing is still not possible (principle 1), integration testing is done during the early testing phase so that the bugs are caught earlier than late (principle 3) and integration with some 3rd party software may also introduce new defects to the code (principle 4).

3) System testing:

In the system testing phase, the behavior of the whole system is defined as a scope and is then tested. We get the following examples:

- Scenario 1:

\* Check the ticket prices.

\* Buy a 7-day ticket.

\* Check the ticket expiration date.

- Scenario 2:

\* Check the payment conditions.

\* Buy a rechargeable travel card.

\* Recharge the card with enough money for a trip.

\* Register a trip.

\* Check the balance of the card.

\* Check whether the card is valid or not.

One very important aspect of this testing phase is to have the systems requirements specification on point. Testing needs to be done according to the user’s needs and requirements (principle 7) to ensure that the finished bug-free software will still be usable by the users, and thus address the business needs of the system.

4) Acceptance testing:

In this last level of software testing, the system is tested for acceptability. The purpose is to evaluate if the system is acceptable for delivery:

For this system, we might use alpha or beta testing as a testing example. Alpha testing is performed on the project site itself, whereas beta testing is done by the end users themselves.

When it comes to the principles, the testing here is very context dependent. It must be tested according to the context of the developed software (principle 6). It also must be tested according to the user’s needs and requirements (principle 7), which is usually done through user acceptance testing (also known as beta testing).

*Reference(s): 1.3 TESTING PRINCIPLES, 2.1.1 V-model.*

**Assignment 2**

Here are the following test conditions, ranked from less to most severe:

These test conditions are not very specific, and thus will be potentially less time-consuming to test than the more difficult conditions. That is why I have decided to group them into the “less severe” part of this list:

10) Try to buy a travel card.

9) Attempt to get payment information.

8) Attempt to get the expiration time.

7) Attempt to get the balance of the card.

The following test conditions go more into detail, and as such will contribute more to defect prevention:

6) Confirm purchase on the confirmation window.

5) Cancel the transaction on the confirmation window.

4) Cancel the payment window when you must insert your credit card.

Lastly, these conditions are the most detailed of them all and will require very specific actions. Advantages of such test conditions would be that they enable more coverage for testing to be applied:

3) Try to buy a trip with insufficient funds.

2) Try to recharge a non-travel card.

1) Enter a non-valid sum to deposit (with illegal characters for example).

**Assignment 3**

Black box testing is a type of testing that is entirely based on software requirements and specifications. It involves the testing of the system’s functionality without looking at the internal code structure of the software.

A black box testing may be functional. This means that it focuses on what the system can do, and it is usually done by software testers. It may also be non-functional, which means it will be based on testing a system’s non-functional requirements such as performance for example. And lastly, we have regression testing, which is done after any change to a system to check if the new code has not introduced new defects to the existing code.

With all this in mind, we will explore four different black box testing techniques for this assignment to develop test cases. These are the following:

- Equivalence partitioning:

This black box testing technique is the most intuitive of them all. It is very easy to use and can be applied to any level of testing (unit, integration, system…). The way it works is by dividing a set of input conditions into groups that can be considered equivalent, and the goal is to minimize the number of possible test cases to an optimum level while still maintaining good test coverage. An equivalence partitioning for this system could look like this:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| invalid | valid | invalid | valid | invalid | valid | invalid | valid | invalid | valid | invalid | valid | invalid | valid | invalid |
| 0-18 NOK | 19 NOK | 20-36 NOK | 37 NOK | 38-154 NOK | 155 NOK | 156-309 NOK | 310 NOK | 311-369 NOK | 370 NOK | 371-384 NOK | 385 NOK | 386-461 NOK | 462 NOK | > 462 NOK |

However, since this is equivalence partitioning, we only pick one value from each partition for testing. That is because we suppose that if one condition in a partition passes or fails then all others will also pass or fail, respectively.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| invalid | valid | invalid | valid | invalid | valid | invalid | valid | invalid | valid | invalid | valid | invalid | valid | invalid |
| 9 NOK | 19 NOK | 28 NOK | 37 NOK | 96 NOK | 155 NOK | 233 NOK | 310 NOK | 340 NOK | 370 NOK | 378 NOK | 385 NOK | 424 NOK | 462 NOK | 463 NOK |

In this test case, we test the following condition for the ticket machine module: “the ticket machine should be able to inform the passenger of the ticket price”. This enables us to test valid and invalid price entries.

- Boundary value analysis:

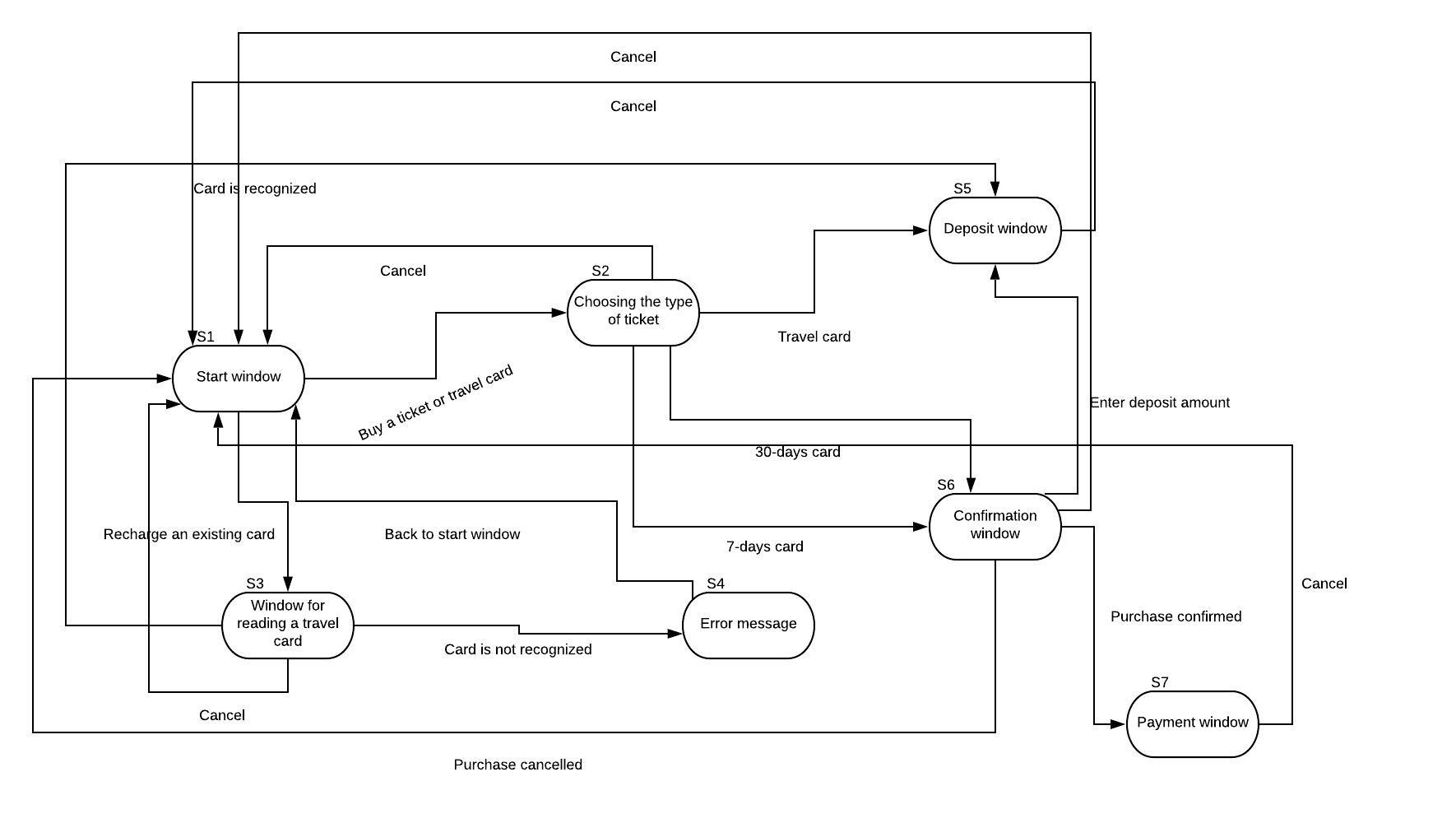
This testing technique focuses on the values at boundaries. It involves checking if test conditions are valid for a given range. This range can either contain extreme ends or boundaries between partitions of the input values. The common idea for a boundary value analysis is to test input values at their minimum, just above minimum, median, just below maximum, and maximum values. It is a useful technique to reduce the number of test cases and is very good for systems where the inputs are given in a range of values. A boundary value analysis for this system could look like this:

|  |  |  |
| --- | --- | --- |
| valid | valid | invalid |
| 0 mins | 0-60 mins | 61 mins |

This test case involves the testing of the condition “a traveler with a rechargeable card should be able to get a ticket valid for one hour after registration”. In boundary value analysis, it is good practice to check the boundary values because they usually contain the most errors. That is why I have decided to check the boundary values of 0 and 61.

- State transition testing:

This black box testing technique is used when a system can be defined in a finite number of states. The transition from one state to another is defined by the rules of the system. A finite state system implies that you get a different output from the same input and is often shown as a state diagram. In this diagram, the tester provides both positive and negative input test values and records the system state behavior. An example of a state transition testing of this system would look like this:

This state transition diagram is of the ticket machine module and shows the interaction between the user and the system.

- Decision table testing:

The final testing technique we will talk about is called decision table testing. This is the method we use if the system contains different combinations of inputs that lead to different actions being taken. We usually use a table format to show the different outputs. It is also called a “cause effect” table, because the cause and effects are represented to achieve a better test coverage. Here is an example of a decision table testing table for this system:

Before we make a decision testing table, we must define conditions. We also need to define the rules of the conditions, and the number of possible combinations. We also assume that the day is on a weekend, if not stated otherwise. The conditions will be:

- The ticket buyer is an adult.

- The ticket buyer is a child.

- The ticket buyer is a student.

- It is a weekday.

The set rules will be true or false. With this in mind, we can calculate that the total number of conditions is 2^4 = 16 different combinations.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Condition** | **Rule 1** | **Rule 2** | **Rule 3** | **Rule 4** | **Rule 5** | **Rule 6** | **Rule 7** | **Rule 8** | **Rule 9** | **Rule 10** | **Rule 11** | **Rule 12** | **Rule 13** | **Rule 14** | **Rule 15** | **Rule 16** |
| Adult | false | false | false | false | false | false | false | false | true | true | true | true | true | true | true | true |
| Child | false | false | false | false | true | true | true | true | false | false | false | false | true | true | true | true |
| Student | false | false | true | true | false | false | true | true | false | false | true | true | false | false | true | true |
| Weekday | false | true | false | true | false | true | false | true | false | true | false | true | false | true | false | true |
| **Output** |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Discount (%) | X | X | 50 | 0 | 100 | 0 | 100 | 0 | 30 | 0 | 50 | 0 | X | X | X | X |

At this point, we can simplify this table by combining the columns with the same output. We can observe that rule 1, 2, 13, 14, 15, 16 have the same output, and as such can combined into one column with the same output X. The output X here means that it is impossible for a discount to be given, either due to there not being any inputs or if the inputs contradict each other (for example, an adult can’t be a child at the same time).

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Condition** | **Rule 1** | **Rule 3** | **Rule 4** | **Rule 5** | **Rule 6** | **Rule 7** | **Rule 8** | **Rule 9** | **Rule 10** | **Rule 11** | **Rule 12** |
| Adult | false | false | false | false | false | false | false | true | true | true | true |
| Child | false | false | false | true | true | true | true | false | false | false | false |
| Student | false | true | true | false | false | true | true | false | false | true | true |
| Weekday | false | false | true | false | true | false | true | false | true | false | true |
| **Output** |  |  |  |  |  |  |  |  |  |  |  |
| Discount (%) | X | 50 | 0 | 100 | 0 | 100 | 0 | 30 | 0 | 50 | 0 |

Moreover, we can observe that a weekday will always result in there being no discount. As such, we can combine all the columns with weekday as true into one column.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Condition** | **Rule 1** | **Rule 3** | **Rule 4** | **Rule 5** | **Rule 7** | **Rule 9** | **Rule 11** |
| Adult | false | false | - | false | false | true | true |
| Child | false | false | - | true | true | false | false |
| Student | false | true | - | false | true | false | true |
| Weekday | false | false | true | false | false | false | false |
| **Output** |  |  |  |  |  |  |  |
| Discount (%) | X | 50 | 0 | 100 | 100 | 30 | 50 |

And lastly, we can see that being a student will overlap being an adult and being a child will overlap being a student. We can then merge those together which brings us to the final version of the simplified decision table:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Condition** | **Rule 1** | **Rule 3** | **Rule 4** | **Rule 5** | **Rule 9** |
| Adult | false | - | - | - | true |
| Child | false | - | - | true | false |
| Student | false | true | - | - | false |
| Weekday | false | - | true | - | false |
| **Output** |  |  |  |  |  |
| Discount (%) | X | 50 | 0 | 100 | 30 |

*Reference(s): 4.3 SPECIFICATION-BASED OR BLACK-BOX TECHNIQUES.*

**Assignment 4**

Test coverage is a way to measure in a specific way the amount of testing performed by several tests. The usual formula for doing that is as follows:

Coverage = (number of coverage items exercised / total number of coverage items) \* 100%, where coverage items means whatever we can count and see whether a test has used this item.

In brief, it is a technique to ensure the tests are actually testing your code or how much of your code is being used by running the tests.

We differentiate between different types of ways to measure the test coverage. The following techniques are:

- Statement coverage:

Statement coverage is a test coverage technique that ensures all the statements in the system have been tested at least once, even the failed ones. This is a very good technique as it validates what the code is expected to do and not do. However, one disadvantage is that it guarantees very basic coverage and as such is not a good way to get 100% coverage.

- Decision coverage:

Decision coverage may involve many different methods. It reports the true or false of Boolean expressions, and as such is a way to check that every possible branch in the code is covered (that is why it is also called “branch” coverage). Unlike the statement coverage, it covers both the true and false conditions, which makes it better than its counterpart in this regard.

For this assignment, I will use the state transition testing technique from the previous assignment to calculate the statement coverage. By obtaining the shortest way across all states in the diagram, that will enable us to say we have 100% statement coverage. Moreover, by covering 100% transition coverage, we can say that we also covered 100% statement coverage, because it is possible to go through all states in the diagram without using all the branches, but not vice versa. Here is an example of a sequence that covers 100% statement coverage:

S1 – S2 – S1 – S3 – S4 – S1 – S2 – S5 – S1 – S2 – S6 – S7.

And thus, we can calculate the statement coverage using the following formula:

Statement coverage = (number of executed statements / total number of statements) \* 100% = (12 / 7) \* 100 = 171.42% state coverage.

*Reference(s): 4.4.1 Using structure-based techniques to measure coverage and design tests.*