Testing Techniques

# Static Testing techniques

What is static testing?

### Why reviews?

Reviews are a way of testing software work products (including code) and can be performed well before dynamic test execution.

Defects detected during reviews early in the life cycle (e.g., defects found in requirements) are often much cheaper to remove than those detected by running tests on the executing code.

So, we are applying the 'Early testing' principle of the 7 Testing Principles!

Compared to dynamic testing, static techniques find causes of failures (defects) rather than the failures themselves.

The types of defects typically found by reviews are:

* Deviations from standards
* Requirements defects eg. Missing, ambiguous elements, Omissions in requirements
* Design defects eg. Design does not match the requirements
* Insufficient maintainability – the code is too complex to maintain

# Difference between Static and Dynamic testing

**Dynamic Testing:**

* Software is executed using a set of input values and its output is then examined and compared to what is expected.
* Dynamic testing can be started by identifying test conditions and test cases as early as possible in life cycle. But dynamic test execution happens only after the code is completed.

**Static Testing:**

* In Static Testing, software work products are examined manually or with a set of tools but are not executed.
* Source code is examined before execution.
* Evaluate and Analyse the source code before it goes to next level.

# Types of reviews

There are 4 types - in increasing order of formality - Informal, Walkthrough, Technical Review and Inspection.

**Informal**

**Walkthrough**

**Technical Review**

**Inspection**

***Formality of reviews***

**High**

**Level of formality**

**Low**

## Informal review

* The least formal or no formal process
* The review is not documented
* The main purpose is to find defects and this is an inexpensive way to achieve some limited benefit
* May not involve a review meeting.
* Use of checklist of optional.
* The usefulness of the review depends on the reviewer
* The review may be implemented by pair programming or by a technical lead reviewing designs and code

## Walkthrough:

* The meeting is led by the author of the document
* Attended by members of the author’s peer group.
* Individual preparation before the meeting is optional.
* Use of scribe is mandatory.
* Use of Checklist is optional.
* Potential Defect logs and review reports may be produced.
* Review sessions are open ended and may vary from informal to very formal.
* The main purposes are to enable learning, understanding and to find defects

## Technical reviews:

* Are well documented
* Scribe is mandatory.
* Reviewers prepare for the review meeting, optionally using check-lists, and prepare a review report with a list of findings, the verdict whether the software product meets its requirements and, where appropriate, recommendations related to findings
* Technical reviews may vary in practice from the quite informal to very formal
* Use of checklist is optional.
* Reviewers must be technical peers or technical experts in the particular domain.

## Inspections

* The most formal
* The inspection process is formal, based on rules and checklists and uses entry and exit criteria.
* Pre-meeting preparation is essential
* After the meeting a formal follow-up process is used to ensure that corrective action is completed and timely.
* Inspection report including list of findings
* The main purpose is to find defects, and process improvement may be a secondary purpose.
* Specified Entry and Exit Criteria are used.
* Scribe is mandatory.
* Matrics are collected and used for improving the process.

# Review techniques

* **Ad-hoc review** - Read the work product and find possible defects.
* **Checklist based review** - Very effective because the review is done against a checklist
* **Scenario based review** - A review technique where the review is guided by determining the ability of the work product to address specific scenarios.
* **Role based review** - Roles based includes specific end user types such as experienced versus inexperienced, age-related or accessibility related.
* **Perspective based review** - Reviewer reviews from different perspective. Eg from testers perspective, stakeholder perspective would be marketing, end users operations etc.

# Blackbox Testing techniques

* Equivalence Partitioning
* Boundary value analysis
* Decision table testing
* State transition testing
* Use case testing.

## EQuivalence class partitioning

Let us consider a program that accepts from number between -5 to 5

-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5

* Negative Integers: Values between -5 and 0
* Zero (neither positive nor negative)
* Positive Integers: Values between 1 and 5
* Values > 5 (Invalid)
* Values < -5 [Invalid]

Example 1:

Let us consider a bank account program that offers

0.5 percent interest for the first $1000 credit;

1percent for the next $1000 and

1.5 percent for the rest.

$1 -1000 0.5%

$1001-2000 1%

>2001 1.5%

|  |  |  |  |
| --- | --- | --- | --- |
| Invalid | Valid - 0.5% | Valid - 1% | Valid 1.5% |
| -0.01 $, alphabets etc | 0.00 - 1000.00 | 1000.01 - 2000.00 | >2000.01 |

Example 2:

A mail-order company selling flower seeds charges

$3.95 for postage and packing on all orders up to $20 value and

$4.95 for orders above $20 value and up to $40 value.

For orders above $40 value there is no charge for postage and packing.

What are the valid and non-valid partitions?

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Valid - Packaging 3.95 | Valid - Packaging 4.95 | Valid- Packaging Free | Invalid | Invalid |
| $0.01 to $20.00 | $20.01 to $40.00 | $40.01 | 0 | alphabets |

## Boundary value analysis

Boundary Value Analysis is based on testing at the boundaries between partitions

For example, if a program should accept a sequence of numbers between 1 and 10, the most likely fault will be that values just outside this range are incorrectly accepted or that values just inside this range are incorrectly rejected.

Eg. The boiling point of water is at 100 degrees Celsius, so the boundary values will be at 99, 100 and 101 degrees

n

n+1

n-1

Ex 2:

If Age >= 23and Age < 44 then

Ex: 3

In a system designed to work out the tax to be paid:

An employee has $4000 of salary tax free.

The next $1500 is taxed at 10%

The next $28,000 is taxed at 22%.

Any further amount is taxed at 40%.

What are the valid equivalence partitions and boundary values for each partition?

Ans:

Valid Equivalence Partitions: 1 to 4000, 4001 to 5500 , 5501 to 33500, >= 33501

|  |  |  |  |
| --- | --- | --- | --- |
| Valid - Tax Free | Valid - 10% Tax | Valid - 22% Tax | Valid - 40% Tax |
| 1 to 4000 | 4001 to 5500 | 5501 to 33,500 | 33,501 to |

Full value BVA for each valid partition:

* 0,1 ,2
* 3900,4000, 4001
* 5499,5500, 5501
* 33500, 33501, 33502

Example 3:

A component takes a score for an exam and returns “Pass”, “Fail”, “Distinction" or “Fault”.

The score must be between 0 and 40.

Between 0 and 24 return “Fail”,

25 and 38 return “Pass”,

over 38 return “Distinction”.

Any other value return “Fault”.

Ans------------------

Equivalence Partition:

|  |  |  |  |
| --- | --- | --- | --- |
| Valid(Fail) | Valid(Pass) | Valid(Distinction) | Invalid |
| 0 to 24 | 25 to 38 | 39 to 40 | Numbers > 40, less than zero, alphabets etc |

Boundary for each partition: -1,0, 1; 23,24,25

24,25,26; 37,38,39

39,40,41

Ex:An input field takes the year of birth between 1900 and 2004. The boundary values for testing this field are

a. 0,1900,2004,2005

b. 1900, 2004

c. 1899,1900,2004,2005

d. 1899, 1900, 1901,2003,2004,2005

Ex: Find Boundary values and equivalence for below:

In a competition ribbons are awarded as follows:

No ribbon for less than 12 mts

Yellow ribbon for up to 25mts

Red ribbon up to 35 mts

And blue ribbon for further.

## Decision Tables

Decision Table is focused on business logic or business rules. DT is a good way to deal with combinations of inputs.

Example : **If Age over 23 and Clean driving record, supply rental car, else reject.**

Conditions: Age and Clean driving record

Actions: supply rental car or do not supply rental car

2 conditions = 4 rules.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Conditions/Input | Rule 1 | Rule 2 | Rule3 | Rule4 |
| Age > 23 | T | T | F | F |
| Clean driving record | T | F | T | F |
|  |  |  |  |  |
| Action/Output |  |  |  |  |
| Supply rental car | Y | N | N | N |
|  |  |  |  |  |

Example 2: If you are a new customer opening a credit card, you will get a 15% discount on all your purchases today. If you are existing customer and you hold a loyalty card, you get a 10% discount. If you have a coupon, you can get 20% off today (but it can't be used with the new customer discount). Discount amounts are added, if applicable.

Conditions : New customer , Existing customer, Discount coupon

Expected results: 15%, 10%, 20%,

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Conditions | Rule1 | Rule2 | Rule3 | Rule4 | Rule5 | Rule6 | Rule7 | Rule8 |
| New Customer(15%) | T | T | T | T | F | F | F | F |
| Loyalty Card(10%) | T | T | F | F | T | T | F | F |
| Coupon (20%) | T | F | T | F | T | F | T | F |
|  |  |  |  |  |  |  |  |  |
| Actions |  |  |  |  |  |  |  |  |
| Discount(%) | X (?) | X(?) | 20 | 15 | 30 | 10 | 20 | 0 |
|  |  |  |  |  |  |  |  |  |

2 to the power of no. of conditions(3 )= 2 x 2 x2 = 8 rules

Rule 1 & 2 - should not occur.

## Usecase testing

* A use case is a description of a particular use of the system by an actor
* Use cases are a sequence of steps that describe interactions between the actor and the system.

|  |  |  |
| --- | --- | --- |
| **Main Success Scenario**  **A:Actor**  **S:System** | **Step** | **Description** |
| 1 | A: Insert Card |
| 2 | S:Validates Card and Asks for PIN |
| 3 | A:Enter PIN |
| 4 | S:Validates PIN |
| 5 | S:Allow access to the Account |
| **Extensions** | 2a | Card Not Valid  S: Display message and reject Card |
| 4a | PIN Not Valid  S: Display Message and ask for re-try(twice) |
| 4b | PIN invalid 3 times  S:Eat card and Exit |

## State transition testing

The idea behind state transition testing is that system might behave differently for the same user input when the state changes.

Any system where you get a different output for the same input, depending on what has happened before, is a finite state system

**State Transition Testing Examples**

If you request to withdraw £100 from a bank ATM, you may be given cash. Later you may make exactly the same request but be refused the money (because your balance is insufficient).

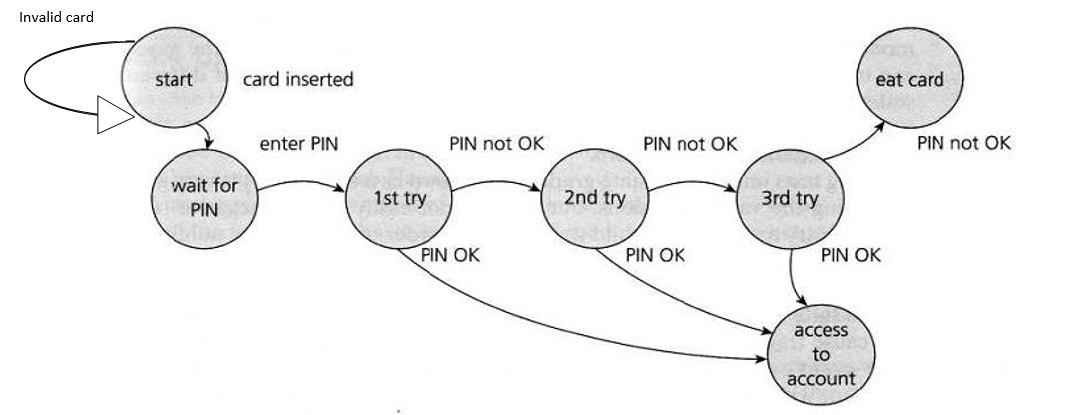
This later refusal is because the state of your bank account had changed from having sufficient funds to cover the withdrawal to having insufficient funds. The transaction that caused your account to change its state was probably the earlier withdrawal.

Another example is a word processor. If a document is open, you are able to Close it. If no document is open, then “Close” is not available. After you choose “Close” once, you cannot choose it again for the same document unless you open that document. A document thus has two states: open and closed.

A state transition model has four basic parts:

* The states that the software may occupy (open/closed or funded/insufficient funds);
* The transitions from one state to another (not all transitions are allowed);
* The events/conditions that cause a transition (withdrawing money, closing a file); --> Inputs
* The actions that result from a transition (an error message, or being given your cash). --> Output

Example of state transition in ATM card withdrawal



# Whitebox Testing techniques

There are two types in Whitebox testing techniques:

* + 1. Statement Testing
    2. Branch Testing

Coverage talks about the number of tests required to cover all the lines of code.

Used mainly at the component level, where structure-based testing is most commonly used, the test case design techniques involve generating test cases from code, so we need to be able to read and analyze the code.

**Code Coverage analysis is the Process of:**

* Finding areas of program not exercised by a set of Test Cases.
* Creating additional test cases to increase Coverage

## STATEMENT COVERAGE:

**Statement Coverage = Number of executable Statements executed**

**------------------------------------------------------------------- X 100**

**Total number of executable statements**

**Statement testing – test cases for the following programs:**

**Program 1:**

**IF Age > 16**

**Process License application**

**END IF**

**Mention the test case for 100%coverage.**

**Program 2:**

**IF Age > 16**

**Process License application**

**ELSE**

**Decline the License application Process**

**END IF**

**Program 3:**

**IF Age > 16**

**IF Age < 50**

**Print Age is between 17 and 50**

**Else IF Age > 50**

**Print Age is grater then 50**

**End IF**

**ELSE**

**Print Age is less than 16**

## Decision Coverage and Testing

* Decision testing aims to ensure that the decisions in a program are adequately exercised.
* Decisions, are part of selection and iteration structures; we see them in IF THEN ELSE constructs and in DO WHILE or REPEAT UNTIL loops.
* To test a decision we need to exercise it when the associated condition is true and when the condition is false; this guarantees that both exits from the decision are exercised.

Decision coverage is measured by counting the number of decision outcomes exercised (each exit from a decision is known as a decision outcome) divided by the total number of decision outcomes in a given program. It is usually expressed as a percentage.

**Number of decision outcomes exercised**

**Decision Coverage = ---------------------------------------------------------------------- x 100**

**Total number of decision outcomes**

**Program to calculate Interest:**

**Interest is based on the Balance in the Account. There is a Base Interest rate which can change periodically. Base Interest rate is 3.5%**

**If Balance > 1000 and < 10,000, the interest rate is base interest rate + half a percent. (0.5 /100 = 0.005)**

**If Balance > 10,000, the interest rate is base interest rate + one percent. (1/100 = 0.01)**

**Program INTEREST**

**Interest, BaseRate, Balance: Real**

**BEGIN**

**Interest = BaseRate**

**Read (Balance) = 999, 2000, 20000**

**IF Balance > 1000 -- T**

**THEN**

**IF Balance <10000**

**THEN**

**Interest= 4**

**ELSE**

**Interest=5**

**END IF**

**END IF**

**Balance = Balance + (Balance \* Interest)**

**END**

**100% Statement Coverage:**

**Number of test cases =2, between 1000 and 10000 and > 10000**

**Decision Coverage:**

**Number of test cases = 3, <1000, between 1000 and 10000 and > 10000**

**So 100% statement coverage does not mean 100% decision coverage.**

**But 100% decision coverage always means 100% statement coverage**

# Cyclomatic Complexity

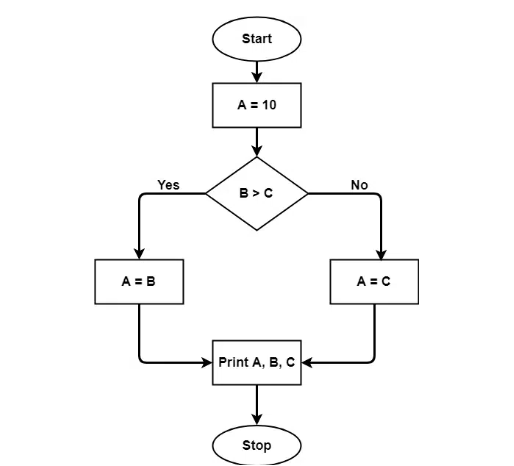
Cyclomatic complexity, developed by Thomas McCabe, is a metric that measures the complexity of a program. The higher the number, the more complex the program. Lower complexity means simpler, more manageable code, reducing the chances of errors and making it easier to maintain and modify.

***M = E – N + 2P***

*where*

*E = the number of edges in the control flow graph   
N = the number of nodes in the control flow graph   
P = Number of disconnected parts of the flow graph (e.g. a calling program and a subroutine)*

*Example:*



Complexity = 7 – 7 + 2