1. Software Verification and Validation (evaluation)

Software engineering goals

- ** Build a high quality system ** : Complete the project on time, on budget, and with all originally specified functions and features.
- ** Software crisis ** : Many software projects fail to achieve these goals.
- **16.3% project success ** : Completed on time and on budget with all initially specified functions and features.
  **52.7% of projects overspent, delayed, or reduced functionality ** : Projects were completed and put into operation, but were over budget (189%), delayed, and had fewer functions and features than originally specified (42%).
- **31% of projects cancelled ** : stopped at some stage in the development process and not put into operation.

Why is software hard to build

- ** Complexity ** : The extreme complexity of a software system leads to residual errors in the finished and running system.
- ** Multiple decision paths ** : A program with a few hundred lines of code may contain dozens of decisions, meaning thousands of alternative execution paths. It is not practical to test all possible paths.
- ** Reality is complex ** : The reality of the software response is almost infinite.
- ** Lack of knowledge ** : Our ability to ensure software reliability is much lower than it should be.
- ** Insufficient Mathematical analysis ** : Other engineers use mathematical analysis to predict system behavior and thus find defects before a product is put into operation. Traditional mathematical methods are suitable for describing physical systems, but not for the synthetic binary universe of software.
  ** * The field of software systems is dominated by discrete mathematics, a relatively new and immature discipline that was not explored in depth until the advent of computers.

Verification and validation

- ** Definition ** :
- Verification: According to IEEE standards, verification is the assessment of compliance of products, services, or systems with regulations, requirements, specifications, or imposed conditions. It's usually an internal process.
  validation: According to IEEE standards, validation is ensuring that a product, service, or system meets the needs of customers and other identified stakeholders. It usually involves acceptance and suitability by external customers.
- **Boehm's definition ** :
- Verification: Establish the authenticity of correspondence between software products and their specifications (derived from the Latin word veritas, meaning "truth").

validation: Establishing the suitability or value of a software product for its operational task (derived from the Latin word "valere", meaning "worth").

- ** Informal definition ** :
- Verification: Am I building products correctly?
- validation: Am I building the right product?

Defect Evaluation -Evaluating Functionality

- ** Goal ** : Look for defects at all stages of product development.
- ** Missing quest ** : The target quest is missing.
- ** Error Response/Effect ** : An error response or effect occurs.
- ** Causes failure ** : These defects eventually cause the system to fail.
- ** Errors, Failures, and Failures ** : Error Fault Failure
- ** error ** : Human error caused by human wrong reasoning, eventually causing failure.
- ** Fault ** : An error is written into the software product and there is a problem with the product.
- ** Failure ** : The software system does not behave as expected.

Static evaluation and dynamic evaluation

- ** Static Evaluation ** :
- ** Purpose ** : To detect faults.
- ** Applicable object ** : any development product.
- ** Dynamic Evaluation (Test) ** :
- ** Purpose ** : Generate test cases, detection failed.
- ** Applicable object ** : Code only.

Software quality control and assurance activities

- ** Preventive Measures ** : Evaluate the development product during the development process, rather than waiting until the code is complete.
- ** Code thoroughly tested ** : Make sure your code is thoroughly tested.

Software Quality

- ** Goal ** : Deliver high quality products.
- ** Quality Definition ** : An abstract concept that needs to be decomposed into more concrete properties.
- ** Quality Standard ** :

- ** Reliability ** : The system will not fail. Reliability
- ** Functionality ** : The system accomplishes the tasks expected by the user. Functionality
- Functional

- ** Efficiency ** : The system responds at a reasonable speed. Efficiency
- ** Usability ** : User comfort. Usability
- ** Maintainability ** : easy to modify, and the change cost is reasonable. Maintainability
- ** Portability ** : Run in different environments. Portability
- ** Security ** : Protects against intrusions and unauthorized use of resources. Security
- Non-functional
- ** Functionality ** : The system accomplishes the tasks expected by the user.
- ** Reliability ** : The system will not fail.

How to test

- ** Reliability ** : Assessed by looking for defects that cause the system to fail.
- ** Functionality ** : evaluated by comparing the tasks accomplished by the system with user expectations (requirements specifications).
- ** Is the task complete ** : Has the system completed all tasks?
- ** Is the task correct ** : Does the system give the correct or agreed response or effect?

When to evaluate

- ** Evaluation timing ** : Evaluate possible defects as early as possible in the development process.
- ** Task Missing ** : The system failed to complete all tasks.
- ** Task Error ** : Task execution error (wrong response or effect).
- ** System failed ** : The system failed.
- ** Early defects ** : Many code defects form at an early stage (requirements, design, etc.).

The cost of fixing the software

The relative cost of | in | phase |
| -------- | -------- |
| requirements | 0.1 to 0.2 |
| Design | 0.5 |
| codes for | 1 |
| Unit test | 2 |

| take the test | 5 |
| maintains | 20 |

Summary

** The importance of software evaluation ** : Developing products should be evaluated during the development process, not after the code is completed.

- ** Code thoroughly tested ** : Make sure your code is thoroughly tested.
- ** Quality Objectives ** : The goal of the software engineer is to deliver high quality products.

Static evaluation

Overview

- ** Software quality Control and Assurance activities ** :
- ** Preventive Measures ** : Software evaluation (static evaluation).
- ** Corrective Action ** : Software evaluation (dynamic evaluation).
- ** Good software construction practices ** : including methodology, inspection of not performing software (static aspects), inspection of any software product, inspection of performing software results (dynamic aspects), code-only inspection.

The benefits of static assessment

1. ** Reduce costs ** :

- ** Early detection ** : Early detection of defects can reduce costs in subsequent stages. Requirements defects: 52% of software projects deliver only 42% of expected functionality on average, with incomplete requirements being the biggest source of problems during the requirements phase (13.1%).
- ** Design and Code ** : Static assessment can detect 30% to 70% of design and code defects.
- ** Cost savings ** : Making corrections during static assessments is easier and less costly.

2. ** Quality Estimate ** :

- ** Early detection ** : Early detection of one defect means there may be more defects.
- ** Quality assessment ** : The number of defects detected in the product can reflect the quality of the development work.
- ** Corrective action ** : If the measurement results show that the quality of the work is insufficient, corrective action can be taken.

Defect types

- ** General Type ** :
- ** Correctness ** : Whether the procedure or requirement is correct.

- ** Completeness ** : Whether all necessary information is included.
- ** Redundancy ** : Whether there is duplicate information.
- ** Consistency ** : Whether there is conflicting information.
- ** Ambiguity ** : Whether it is easy to generate ambiguity.
- ** Traceability ** : Whether it can be traced back to requirements or designs.
- ** Specific type ** :
- ** Demand defects ** :
- ** Correctness ** : Incorrect requirements do not specify what is required.
- ** Complete ** : The requirement contains all necessary information and does not contain unnecessary information.
- ** Redundancy ** : There are no multiple requirements for the same problem.
- ** Consistency ** : There are no conflicting requirements.
- ** Ambiguity ** : Requirements should not be ambiguous.
- ** Design defects ** :
- ** Correctness ** : Whether the design meets the corresponding requirements.
- ** Integrity ** : The design contains all necessary information and does not contain unnecessary information.
- ** Redundancy ** : There are no parts of the design that handle the same requirement (unless specifically required).
  ** Consistency ** : There are no contradictory parts in the design, matching between different models (static, dynamic, etc.) and consistent with the requirements.
- ** Ambiguity ** : Usually related to completeness.
- ** Code defect ** :
- ** Correctness ** : No sign errors, behavior as designed (and therefore as required).
- ** Integrity ** : Provides all the functions required by the system, without providing additional functions.
- ** Redundancy ** : No two pieces of code implement the same function.
- ** Consistency ** : Consistent with design and requirements.
- ** Traceability ** : Consistent with requirements and design.
- ** Clarity ** : Code should be clear and easy to understand.

Classification of defects

- ** Basic product defects (correctness) ** : Defects that are not dependent on earlier products.

- ** Internal Product Defects ** : Defects detected by relationship to phase input products.
- ** External product defects (validation) ** : Defects that can only be detected by comparison with requirements (usually requirements).

Static technology and reading technology

Static technology

- ** Three types of technology ** :
- ** Judging ** :
- ** Technical Review ** : Technical experts evaluate the product.
- ** Walkthrough ** : Developers show code to peers and discuss.
- ** Inspection ** : A structured review process, usually conducted by a dedicated inspector.
- ** Audit ** : A formal, independent evaluation process.
- ** Desktop Review ** : Personal initial inspection of the product.
- ** Formal proof ** : Using mathematical methods to prove the correctness of a program.
- ** Tracking ** : Ensure product alignment with requirements and design.

Reading technology

A reading technique is a guide to help detect defects in a software product, consisting of a series of steps or procedures that allow the reader to gain an in-depth understanding of the product being read.

- ** Objective ** : Defect detection.
- ** Type ** :
- ** Free Reading ** : No instructions or guidance are provided, and the reader reviews the product systematically on his or her own.
- ** Checklist-based reading ** : Guide the reader through a list of questions, but the questions are often too general and not specific enough.
- ** Step-up abstraction ** : Detects defects by comparing program specifications with actual program behavior.
- ** Other ** : Other specific reading techniques.

Summary

- ** Importance of Static evaluation ** : Static evaluation reduces costs in subsequent stages by detecting defects early and provides an estimate of product quality.
- ** Defect Type ** : includes correctness, integrity, redundancy, consistency, ambiguity, and traceability.
  Static technology: includes reviews, proof-of-form and tracking to help ensure that products meet requirements and design.

- ** Reading technology ** : Helps detect defects in software products, including casual reading, checklist-based reading, and step-by-step abstraction.

---

2. Introduction to software testing

The role of software testing

- ** Quality Control and Assurance activities ** : Software testing is an important part of ensuring software quality, including preventing and correcting software defects.
  Static Analysis and Dynamic Analysis ** :
- ** Static analysis ** : The inspection of software without executing the software, applicable to any software product, mainly examining static aspects.
- ** Dynamic analysis (testing) ** : Checking results by executing software, focusing primarily on dynamic aspects, usually for code.

Maturity level

- **Level 0** : Testing is equivalent to debugging.
- **Level 1** : Proves that the software works.
- **Level 2** : Indicates that the software does not work.
- **Level 3** : Reduces the risk of software not working.
- **Level 4** : Testing is an intellectual discipline designed to reduce software risk.

Test definition

- ** Process ** : Run a program or system to find defects.
- ** Activities ** : Evaluate the attributes or capabilities of a process or system to determine whether the expected results are being achieved.
- ** Information Collection ** : Collect information to evaluate work efficiently.

Test principle

** * Testing is to find software defects ** :

- Good test cases ** : The ability to find undetected defects.
- ** Expected results ** : Need to be clearly defined.
- ** Detailed inspection ** : Every test case requires a detailed inspection.
- ** Valid input ** : The test case should cover valid and invalid, expected and unexpected input conditions.
  Functional testing: Make sure the software does what it's supposed to do and doesn't do what it shouldn't.
- ** Independent testing ** : Programmers should not test their own programs and teams should not test their own programs.

- ** Keep test cases ** : Do not discard test cases.
- ** Plan testing efforts ** : It should not be assumed that defects will not be found.
- ** Defect concentration ** : The more defects are found in a part of the software, the more defects are likely to exist in that part.
- ** Full test impossible ** : Full test is impossible.
- ** Prevention of defects ** : One purpose of testing is to prevent defects from occurring.
- ** Risk-based ** : Testing is risk-based.
- ** Intellectual activity ** : Testing is an extremely challenging, creative and intellectual activity.

Testing process

1. ** Generate and specify test cases ** :

- ** Test Case ID** : Unique identification of the test case.
- ** Test objective ** : A high-level description of the test case, such as "Do not create a file when the test does not provide a file name".
- ** Context ** : Background information about the test case.
- ** Input ** : Data actually entered into the program, such as "File name not provided".
- ** Expected output ** : Program output as described by the specification, such as "Program displays error message: Input file not provided".
- ** Observation Output ** : Actual output when the program is running, such as "Program shows error message: file not provided".

2. ** Run test cases ** : Execute test cases.
3. ** Compare ACTUAL Results with Expected Results ** : Compare actual results with expected results.
4. ** Identify system faults ** : Find faults in the system.
5. ** Identify and correct the cause of the failure ** : Find and repair the cause of the failure (debugging steps).

Test case design techniques

- ** Test Case Design techniques ** : Introduces various test case design methods to help generate effective test cases.

Chapter 1: Self-assessment test

Overview

** Changes in Software Testing ** : Since the book was first published, software testing has become both easier and more difficult. The diversity of programming languages, operating systems, and

hardware platforms increases the complexity of testing, but the maturity of modern software and operating systems also provides some off-the-shelf, tested functional modules that reduce the programmer's workload.

The importance of software testing

Modern software systems affect more people and businesses, so the quality and reliability of software becomes more important.
** * Reuse of functional modules ** : Modern development languages provide pre-programmed objects that have been debugged and tested, reducing the need for testing custom code.

Chapter 2: The Psychology and Economics of Procedural Testing

Testing Principles

1. ** Define Expected output ** : The test case must contain a definition of the expected output or result.
2. ** Avoid self-testing ** : Programmers should avoid testing their own programs.
3. ** Avoid team self-testing ** : Programming organizations should not test their own programs.
4. ** Check test results thoroughly ** : The results of each test should be carefully checked.
5. ** Test for invalid and unexpected inputs ** : Test cases should include invalid and unexpected input conditions, not just valid and expected input conditions.
6. Check for undesired behavior: Check not only if the program does what it should, but also if it does what it shouldn't.
7. ** Keep test cases ** : Test cases should not be discarded unless the program itself is temporary.
8. ** Avoid assuming no errors ** : Testing should not be planned under the assumption that no errors will be found.
9. Clustering of defects: The more bugs found in one part of the program, the more likely there are more bugs in that part.
10. ** Creativity of testing ** : Testing is an extremely creative and intellectually challenging task that may require a higher level of creativity than designing programs.

Black box test

Black box testing (also known as data-driven or input/output driven testing) treats a program as a black box, where the tester is not concerned with the internal structure and behavior of the program, but instead focuses on finding cases where the program does not conform to the specification.

- ** Test data ** : Test data is generated only according to the specification and does not utilize knowledge of the internal structure of the program.
- ** Exhaustive input testing ** : In theory, all possible input combinations need to be tested in order to ensure that the program handles all cases correctly. In practice, however, this is usually not feasible because the number of input combinations can be very large.

white box test

White box testing (also known as logic-driven testing) allows the tester to examine the internal structure of the program. The test data is derived from an examination of the program logic.

- ** Exhaustive path testing ** : The theoretical goal is to execute all possible paths in the program through the test case. In practice, however, this is usually not feasible because the number of paths can be extremely large.
- ** Limitations of Path testing ** :
- ** Huge number of paths ** : Even simple programs can have astronomical paths.
- ** Path testing does not guarantee correctness ** : Even if all paths have been tested, the program may still have errors due to not including the necessary paths or logical errors.

Test psychological questions

- ** Goal Oriented ** : The goal of testing should be to find errors, not to prove that the program is bug free. Psychological research has shown that when people try to prove something, they tend to ignore evidence against that conclusion.
  ** The importance of independent testing ** : It is difficult for programmers to objectively test their own programs because they may not be willing to find errors, and misunderstandings about the program may be introduced into testing.
  Testing is an inherently destructive process, similar to a doctor diagnosing a disease with the goal of finding and fixing the problem.

Summary

- ** The importance of testing ** : Software testing is a key part of ensuring software quality, improving the reliability and performance of software by finding and fixing errors.
- ** Principles of testing ** : Following a series of testing principles, such as defining expected outputs, avoiding self-testing, thoroughly checking test results, testing invalid and unexpected inputs, retaining test cases, etc., can improve the effectiveness and efficiency of testing.
- ** Test method ** : Black box test and white box test each have advantages and disadvantages, combined use can be more comprehensive test software.

Introduction to white box testing

Test objectives

- ** Find defects ** : Improve perceived quality.
- ** Evaluate software quality ** : Evaluate the overall quality of the software based on the number of defects found.

Whether the goal can be achieved

- ** Impossibility of exhaustive testing ** : It is not feasible to exhaustively test all possible inputs or paths because the number of inputs and paths can be extremely large.

- ** Select Input set ** : The current method is to make an estimate by selecting some inputs from the input field, but the selected inputs affect the quality of the estimate and may lead to bias.

Test case design techniques

Technical classification

- ** Classification Standard ** :
- ** Adequacy criteria ** : Selection criteria for test cases.
- ** Method ** : Realization knowledge based method (white box), fault based method (variation), error based method (function random).

Control flow technology

- ** Definition ** : Treat the program as a white box and generate test cases according to the control structure of the program.
- ** Coverage concept ** :
- ** Statement coverage ** : indicates whether each statement is executed.
- ** Branch coverage ** : Whether each branch of each control structure is executed.
- ** Conditional coverage ** : Whether each Boolean subexpression is evaluated as true and false.
- ** Decision/Conditional coverage ** : Whether every Boolean subexpression and every branch is covered.
- ** Multiple conditional coverage ** : Whether all combinations of Boolean subexpressions are evaluated.

Basic path test

- ** Path Definition ** : A series of statements from program input to output.
- ** Strategy ** :
- ** Analyze the control flow diagram ** : Find a set of linearly independent execution paths.
- ** Generate test cases ** : Generate test cases for each path.
- ** Based on McCabe cycle complexity ** :
- ** Calculate cycle complexity ** : Determine the number of linear independent paths.
- ** Guaranteed ** : Full branch coverage, but not necessarily covering all possible paths.

Data streaming technology

- ** Definition ** : Treat the program as a white box and generate test cases based on the assignment of variable values and the effect of the assignment on execution.
- ** Variable occurs ** :

- ** Definition (def) ** : Associated with a node, indicating that the node contains globally defined variables.
- ** Computing Use (c-use) ** : Associated with a node, indicating that the node contains variables used in global computing.
- ** Predicate use (p-use) ** : Associated with an edge, indicating that the edge contains the variable used by the predicate.
- ** Definition - Use on ** :
- **dcu(x, i)** : All nodes j on the defined free path from node i to node j, where x is the computationally used variable in node j.
- **dpu(x, i)** : All sides (j, k) on the defined free path from node i to node j, where x is the predicate use variable in the edge (j, k).

mutation technology

- ** Definition ** : Treat the program as a white box, generating variants based on possible failures in the program.
- ** Mutation operators ** : Generate variants using mutation operators specific to the programming language.
- ** Generate test cases ** : Generate test cases from variants, the number of generated test cases should be enough to kill all variants.
- ** Technical variants ** :
- ** Strong mutation ** : Using all mutation operators, the goal is to achieve 100% coverage.
- ** Weak variation ** : Relax coverage conditions.
- ** Select Mutation ** : Only partial mutation operators are used.

Test case generation

equivalence class partition

- ** Input condition ** : Each input condition is divided into groups, defining valid and invalid equivalence classes.
- ** Generate test cases ** :
- ** Cover as many valid equivalence classes as possible ** : Generate test cases until all valid equivalence classes are covered.
- ** Override invalid equivalence classes ** : Generate test cases until all invalid equivalence classes are overridden.

boundary value analysis

- ** Value Range ** :

- ** Design test cases ** : two boundary values of coverage and cases just beyond these two boundary values.
- ** Number of values ** :
- ** Design test cases ** : cover the minimum and maximum values, as well as just above the maximum and less than the minimum.
- ** Ordered set ** :
- ** Focus on the first and last element of the set **.
- ** Output data ** : Apply the above rules to the output data.

Test case description

- ** Test Case ID** : Unique identification of the test case.
- ** Test objective ** : A high-level description of the test case, such as "Do not create a file when the test does not provide a file name".
- ** Context ** : Background information about the test case.
- ** Input ** : Data actually entered into the program, such as "File name not provided".
- ** Expected output ** : Program output as described by the specification, such as "Program displays error message: Input file not provided".
- ** Observation Output ** : Actual output when the program is running, such as "Program shows error message: file not provided".

Summary

The importance of White Box Testing ** : Generate test cases by examining the internal structure and logic of the program to ensure that every part of the program is fully tested.

- ** Test Case design techniques ** : including control flow techniques, data flow techniques, and variation techniques, which help generate effective test cases.
- ** Test case generation ** : Ensure that test cases cover all possible input and output cases through equivalence class partitioning and boundary value analysis.

Chapter 2: Graph Overlay

The basis of the image overlay

- ** Goal of graph coverage ** : Generate test cases through the abstraction of the graph to ensure that the test cases can cover all parts of the graph.
  Figure - * * * * : the definition of figure \ \ (G) set by the node \ [N], the initial node set \ (N_0), the final node set \ [N_f) and edge set \ [E].

Common graph coverage guidelines

- ** Node overlay ** : Ensure that every node is accessed.

- ** Edge overlay ** : Ensure that every edge is executed.
- ** Condition overlay ** : Ensures that every Boolean subexpression is evaluated as true and false.
- ** Decision/Condition Overlay ** : Ensure that every Boolean subexpression and every branch is covered.
- ** Multiple condition coverage ** : Ensures that all combinations of Boolean subexpressions are evaluated.

diagram application

- ** Finite State Machine (FSM) ** : Early research focused on using FSM to generate test cases, especially in telecommunications systems.
- ** Control Flow Diagram ** : The most common form of diagram abstraction that maps code to a control flow diagram.
- ** Graph generation Method ** : including generating spanning tree, path combination and other methods.

Chapter 3: Logical overrides

Formalization of logical expressions

- ** Predicate ** : An expression evaluated as a Boolean, containing comparison operators for Boolean variables, non-Boolean variables, and function calls.
- ** Logical operator ** :
- ** Negate operator (¬) **
- ** and the operator (∧) **
- ** or the operator (∨) **
- ** contains the operator (→) **
- ** XOR operator (⊕) **
- ** Equivalence operator (↔) **

Common logical coverage criteria

- ** Predicate overlay ** : Ensures that all possible values of each predicate are evaluated.
- ** Condition overlay ** : Ensures that every Boolean subexpression is evaluated as true and false.
- ** Decision overlay ** : Ensure that every Boolean subexpression and every branch is covered.
- ** Multiple condition coverage ** : Ensures that all combinations of Boolean subexpressions are evaluated.

Limitations of logical coverage

- ** Path combination explosion ** : Complex logical expressions can cause the number of path combinations to increase dramatically, making test case generation impractical.
- ** Code simplification ** : Sometimes the number of path combinations can be reduced by simplifying the structure of the code, such as merging duplicate blocks of code.

Chapter 4: Input space partitioning

Enter the combined policy for space partitioning

- ** Full Combination (ACoC) ** : A combination that overrides all input values.
- ** Pairs of Combinations (PWC) ** : Pairs of combinations covering all input values.
- ** Multi-base Selection (MBCC) ** : A combination that covers multiple base selections.

properties of the partition

- ** Integrity ** : Each input value belongs to a partition.
- ** Mutually exclusive ** : Each input value belongs to only one partition.

Selection of partitions

- ** Valid Values ** : Contains at least one set of valid values.
- ** Subpartition ** : Divides the valid value range into smaller subpartitions.
- ** Boundary values ** : Inputs close to boundary values often cause problems.
- ** Normal use ** : If the operating configuration is mainly focused on "normal use", the failure rate depends on the value of non-boundary conditions.
- ** Invalid Value ** : Contains at least one set of invalid values.

Summary

- ** Graph coverage ** : Generate test cases through the abstraction of the graph to ensure that the test cases can cover all parts of the graph. Common graph coverage criteria include node coverage, edge coverage, condition coverage, decision/condition coverage, and multiple condition coverage.
- ** Logical coverage ** : Generate test cases through the evaluation of logical expressions to ensure that the test cases can cover all parts of the logical expression. Common logical coverage criteria include predicate coverage, conditional coverage, decision coverage and multiple conditional coverage.
- ** Input space partition ** : Generate test cases by partitioning input values, ensuring that test cases cover all possible combinations of input values. Common input space partitioning strategies include full combination, pair combination and multi-base selection.

Chapter 4: Test Case Design

Overview

The importance of Test Case design ** : The most critical part of program testing is designing and creating effective test cases.
Limitations of Testing: No matter how creative and comprehensive testing is, there is no guarantee that all errors will be found.
** The need for testing ** : Since complete testing is not possible, testing must be as comprehensive as possible.

Test case design method

- ** Black Box Test Method ** :
- ** Equivalence partition ** : The input field is divided into several equivalence classes, and representative values in each equivalence class are selected for testing.
- ** Boundary Value analysis ** : Tests the boundary values of the input field and the cases just outside the boundary values.
- ** Causation diagram ** : The relationship between input conditions and output results is mapped to generate test cases.
- ** Wrong Guess ** : Based on experience and intuition, guess possible errors and design test cases.
- ** White Box Test Method ** :
- ** Statement overlay ** : Ensure that each statement is executed at least once.
- ** Branch overlay ** : Ensure that each branch is executed at least once.
- ** Condition overlay ** : Ensures that every Boolean subexpression is evaluated as at least true and false.
- ** Decision/Condition Overlay ** : Ensure that every Boolean subexpression and every branch is overwritten at least once.
- ** Multiple condition coverage ** : Ensures that all combinations of Boolean subexpressions are evaluated at least once.

A comprehensive approach to test case design

- ** Integrated Approach ** : It is recommended to use a combination of black box and white box test methods to design more comprehensive test cases.
- ** Black Box testing ** : Mainly used for functional testing to ensure that the program meets the required specifications.
- ** White Box testing ** : Mainly used for logic testing to ensure that the internal structure and logic of the program are correct.

The specific method of test case design

1. ** Equivalence division ** :

- ** Define equivalence classes ** : Divide the input field into several equivalence classes, and the values in each equivalence class are treated as equivalent in the test.
- ** Select Test Cases ** : Select one or more representative values from each equivalence class as test cases.

2. ** Boundary Value Analysis ** :

- ** Select Boundary values ** : Tests the boundary values of the input field and the cases just outside the boundary values.
- ** Number of values ** : Tests the minimum and maximum values, as well as cases just above the maximum and less than the minimum.
- ** Ordered set ** : Focus on the first and last elements of the set.

3. ** Causal Graph ** :

- ** Define causality ** : Plot the relationship between input conditions and output results.
- ** Generate test cases ** : Generate test cases from the causation diagram to ensure that every causation is tested.

4. ** Wrong guess ** :

- ** Based on experience and intuition ** : Guess at possible errors and design test cases to verify those errors.

5. ** statement overlay ** :

- ** Ensure each statement ** : Ensure that each statement is executed at least once.

6. ** Branch coverage ** :

- ** Ensure each branch ** : Ensure that each branch is executed at least once.

7. ** Condition coverage ** :

- ** Ensure every Boolean subexpression ** : Ensure that every Boolean subexpression is evaluated as at least true and false.

8. ** Decision/Condition Overlay ** :

- ** Ensure every Boolean subexpression and branch ** : Ensure that every Boolean subexpression and every branch is overwritten at least once.

9. ** Multiple condition coverage ** :

- ** Ensure all combinations of Boolean subexpressions ** : Ensure that all combinations of Boolean subexpressions are evaluated at least once.

Considerations for test case design

- ** Random input testing ** : Randomly selecting input values for testing is the least effective method, as it may not catch most errors.

- ** Selection of test cases ** : When selecting test cases, consider the validity and invalidity of input values, as well as boundary values and special cases.
- ** Retention of test cases ** : Test cases should not be discarded unless the program itself is temporary.

Summary

The importance of Test Case design ** : Designing and creating effective test cases is key to ensuring software quality.

- ** Test Case Design Method ** : Using a combination of black box and white box test methods, software can be tested more comprehensively.
- ** Specific methods ** : Equivalence partitioning, boundary value analysis, causal graphs, false guesses, statement coverage, branch coverage, condition coverage, decision/condition coverage, and multiple condition coverage can help generate effective test cases.