2019 级本科

软件工程

Software Engineering

张昕

zhangxin@cust.edu.cn

计算机科学技术学院软件工程系

Implementation

Introduction

- Usually, coding and testing are collectively referred to as implementation.
- The so-called coding is to translate the results of software design into a program written in a certain programming language.
- As a stage of the software engineering process, coding is the further specification of the design. Therefore, the quality of the program mainly depends on the quality of the software design.
    - The characteristics and coding style of the selected programming language will also have a profound impact on the reliability, readability, testability and maintainability of the program.
- The purpose of testing is to find as many errors as possible in the software before it is put into production operation. At present, software testing is still a key step to ensure software quality. It is the final review of software specifications, design and coding.

Introduction

- Software testing spans two stages in the software life cycle.
  - Usually after writing each module, it is necessary to test it (called unit testing). The writer and tester of the module are the same person, and coding and unit testing belong to the same stage of the software life cycle.
  - A variety of comprehensive tests should also be carried out on the software system. This is another independent stage in the software life cycle, and this work is usually undertaken by specialized testers.

Introduction

- A lot of statistics show that the workload of software testing often accounts for more than 40% of the total workload of software development. In extreme cases, the cost of testing software that is related to human life safety may be equivalent to the total amount of other development steps in software engineering. 3 times to 5 times the cost.
- We must attach great importance to software testing. Never think that software development is almost complete after the program is written. In fact, there is about the same amount of development work to complete.
- As far as testing is concerned, its goal is to find bugs in the software, but finding bugs is not the ultimate goal. The fundamental goal of software engineering is to develop high-quality software that fully meets the needs of users. Therefore, after errors are found through testing, they must be diagnosed and corrected. This is the purpose of debugging. Debugging is the most difficult task in the testing phase.
- When the test results were collected and evaluated, the reliability of the software began to become clear. The software reliability model uses failure rate data to estimate the future failure of the software and predict the reliability of the software.

Choose a programming language

- Programming language is the most basic tool for communication between humans and computers. Its characteristics will inevitably affect the way people think and solve problems, the way and quality of communication between humans and computers, and the difficulty for others to read and understand programs. Easy degree.
- Therefore, an important task before coding is to choose an appropriate programming language.
- Generally speaking, there is a sentence-to-many-sentence correspondence between high-level language source program statements and assembly code instructions.
    - Statistics show that the number of high-level language statements that a programmer can write in the same time is roughly the same as the number of assembly language instructions, so the productivity of writing programs in high-level languages can be several times higher than writing programs in assembly language.
- High-level languages generally allow users to assign clear names to program variables and subroutines, and it is easy to associate program objects with the entities they represent through names;
- In addition, the symbols and concepts used in high-level languages are more in line with people's habits. Therefore, programs written in high-level languages are easy to read, easy to test, easy to debug, and easy to maintain.

Choose a programming language

- High-level language is significantly better than assembly language
  - Therefore, except in very special application fields (for example, there are very strict restrictions on program execution time and space used; arbitrary or even illegal instruction sequences need to be generated; microprocessors with special architectures, so that in this type of High-level language compilers are usually not implemented on the machine), or a small part of the code that is critical to the execution time (or directly dependent on the hardware) in a large system needs to be written in assembly language,
- All other programs should be written in high-level languages.

Practical criteria

- In order to make the program easy to test and maintain to reduce the total cost of the software, the selected high-level language should have an ideal modular mechanism, as well as a readable control structure and data structure;
- In order to facilitate debugging and improve software reliability, language features should enable the compiler to find as many errors in the program as possible;
- In order to reduce the cost of software development and maintenance, the selected high-level language should have a good independent compilation mechanism.
- The above-mentioned requirements are ideal criteria for selecting a programming language. However, in the actual selection of a language, not only theoretical criteria must be used, but practical limitations must also be considered.

Practical criteria

- (1) Requirements of system users. If the developed system is maintained by the user, the user usually requires the program to be written in a language they are familiar with.
- (2) Compiler programs that can be used. The compilers that can be provided in the environment of the target system often limit the range of languages that can be selected.
- (3) Available software tools. If a certain language has software tools that support program development, it will be easier to implement and verify the target system.
- (4) Project scale. If the scale of the project is very large and the existing language is not fully applicable, then it may be a correct choice to design and implement a programming language dedicated to this project.
- (5) The programmer's knowledge. Although it is not difficult for experienced programmers to learn a new language, it takes practice to fully master a new language. If it is not inconsistent with other standards, you should choose a language that is already familiar to programmers.
- (6) Software portability requirements. If the target system will run on several different computers, or the expected service life is very long, it is important to choose a language with a high degree of standardization and good program portability.
- (7) Application areas of the software. The so-called general programming language is actually not equally applicable to all application fields. Therefore, the application range of the target system should be fully considered when choosing a language.

Coding style

- The logic of the source code is concise, clear, easy to read and understand is an important criterion for a good program, and the following rules should be followed.
- 1. Documents inside the program
- The so-called internal documents of the program include appropriate identifiers, appropriate annotations, and visual organization of the program, and so on.
- Choose a name with a clear meaning so that it can correctly prompt the entity represented by the program object, which is very important to help readers understand the program. If abbreviations are used, then the abbreviation rules should be consistent, and each name should be annotated.
- Annotation is an important means of communication between programmers and program readers. Correct annotations are very helpful to the understanding of the program.
    - There is usually a preface at the beginning of each module, which briefly describes the module's functions, main algorithms, interface features, important data, and a brief history of development. A note related to a piece of program code inserted in the middle of the program mainly explains the necessity of including this piece of code.
- For source programs written in high-level languages, there is no need to translate each sentence into natural language in the form of annotations, and annotations should be used to provide some additional information. A space or blank line should be used to clearly distinguish between comments and procedures.
    - The content of the annotation must be correct. Wrong annotations are not only useless to understand the program, but will hinder the understanding of the program.

Coding style

- The layout of the program list also has a great impact on the readability of the program, and an appropriate ladder form should be used to make the hierarchy of the program clear and obvious.
- 2. Data Description
- Although the organization and complexity of the data structure have been determined during the design, the style of data description is determined when the program is written.
  - In order to make the data easier to understand and maintain, there are some simple principles that should be followed.
- The order of data description should be standardized.
  - It is easy to check in order, so it can speed up the process of testing, debugging and maintenance.
- When multiple variable names are described in a statement, these variables should be arranged in alphabetical order.
- If a complex data structure is used in the design, annotations should be used to explain the method and characteristics of the data structure in a programming language.

Coding style

- 3. Statement structure
- The logical structure of the software is determined during the design, but the construction of individual statements is a major task in writing a program. The principle that should be followed when constructing statements is that each statement should be simple and straightforward, and the program cannot be overly complicated in order to improve efficiency. The following rules help make the statement simple and clear:
  - Do not write multiple statements on the same line in order to save space;
  - Try to avoid complicated condition testing;
  - Minimize the testing of "non" conditions;
  - Avoid extensive use of loop nesting and conditional nesting;
  - Use parentheses to make the order of operations of logical expressions or arithmetic expressions clear and intuitive.

Coding style

- 4. Input and output
- The following rules about input and output styles should be considered when designing and writing programs:
  - Check all input data;
  - Check the legality of important combinations of input items;
  - Keep the input format simple;
  - Use the end of data tag, do not require the user to specify the number of data;
  - Clearly prompt the request for interactive input, specifying the available options or boundary values;
  - When the programming language has strict requirements on the format, the input format should be kept consistent;
  - Well-designed output report;
  - Add flags to all output data.

Coding style

- 5. Efficiency
- Efficiency mainly refers to two aspects: processor time and memory capacity.
- To follow 3 principles:
  - First of all, efficiency is a performance requirement, so the efficiency requirement should be determined in the requirements analysis stage. Software should be as effective as required of it, not as effective as humans might be able to do.
  - Secondly, efficiency is improved by good design.
  - Third, the efficiency of the program is consistent with the simplicity of the program. Don't sacrifice the clarity and readability of the program to increase efficiency unnecessarily. The efficiency issue will be further discussed in the following three aspects.

Coding style

- 5. Efficiency
- Efficiency mainly refers to two aspects: processor time and memory capacity.
- (1) Program running time
  - The efficiency of the source program is directly determined by the efficiency of the algorithm determined in the detailed design stage, but the style of writing the program can also affect the execution speed and memory requirements of the program. When translating detailed design results into programs, the following rules can always be applied:
    - Simplify arithmetic and logical expressions before writing programs;
    - Carefully study the nested loops to determine if any statements can be moved from the inner layer to the outer layer;
    - Try to avoid using multi-dimensional arrays;
    - Try to avoid using pointers and complicated tables;
    - Use arithmetic operations with short execution time;
    - Do not mix different data types;
    - Try to use integer arithmetic and Boolean expressions.
    - In application areas where efficiency is a decisive factor, try to use compilers with good optimization characteristics to automatically generate efficient target codes.

Coding style

- 5. Efficiency
- Efficiency mainly refers to two aspects: processor time and memory capacity.
- (2) Memory efficiency
  - In large computers, the characteristics of operating system page scheduling must be considered. Generally speaking, using a structured control structure that can maintain functional domains is a good way to improve efficiency.
  - If the minimum storage unit is required in the microprocessor, a compiler with compact memory characteristics should be selected, and assembly language can be used when necessary.
  - Techniques that improve execution efficiency generally also improve memory efficiency. The key to improving memory efficiency is also "simplicity."

Coding style

- 5. Efficiency
- Efficiency mainly refers to two aspects: processor time and memory capacity.
- (3) Input and output efficiency
  - If the user's mental work is economical in order to provide input information to the computer or to understand the information output by the computer, then the efficiency of communication between humans and computers is high. Therefore, simplicity and clarity are also the key to improving the efficiency of human-machine communication.
  - The communication efficiency between hardware is a very complicated issue, but from the point of view of writing programs, there are some simple principles that can improve the efficiency of input and output. e.g:
    - All input and output should be buffered to reduce the overhead for communication;
    - The simplest access method should be selected for secondary storage (such as disk);
    - The input and output of the secondary storage should be carried out in units of information groups;
    - If the "super efficient" input and output are difficult to understand, this method should not be used.
    - These simple principles apply to both the design and coding phases of software engineering.

Fundamentals of Software Testing

- The purpose of software testing is the opposite of the purpose of all other stages of software engineering.
  - The other stages of software engineering are "constructive": software engineers try to start from abstract concepts and gradually design specific software systems until they write executable program codes in an appropriate programming language.
  - However, in the testing phase, the testers worked hard to design a series of test schemes, but the purpose was to "destroy" the already built software system-trying to prove that there were errors in the program that could not work correctly according to the predetermined requirements.

Target of Software Testing

- G.Myers gave some rules about testing, which can be regarded as the goal or definition of testing:
  - (1) Testing is the process of executing the program in order to find errors in the program;
  - (2) A good test plan is a test plan that is very likely to find errors that have not been discovered so far;
  - (3) A successful test is a test that finds errors that have not been discovered so far.

Target of Software Testing

- The correct definition of test is "the process of executing a program in order to find errors in the program".
  - This is quite the opposite of what some people usually imagine that "testing is to show that the program is correct", "a successful test is a test that finds no errors", and so on.
- It is very important to understand the goal of the test correctly. The goal of the test determines the design of the test plan.
  - If the test is performed to show that the program is correct, some test schemes that are not easy to expose errors will be designed;
  - On the contrary, if the test is to find errors in the program, it will strive to design a test plan that best exposes the errors.
- Since the goal of testing is to expose errors in the program, from a psychological point of view, it is not appropriate to test the program by the programmer himself. Therefore, in the comprehensive testing phase, a testing team is usually composed of other personnel to complete the testing work.
- In addition, it should be recognized that testing can never prove that the program is correct.
  - Even after the most rigorous testing, there may still be undiscovered errors lurking in the program.
- The test can only find errors in the program, but cannot prove that there are no errors in the program.

Guidelines of Software Testing

- In order to design an effective test plan, software engineers must thoroughly understand and correctly use the basic principles that guide software testing.
  - (1) All tests should be traceable to user needs.
    - The goal of software testing is to find errors. From the user's point of view, the most serious errors are those that cause the program to fail to meet the user's needs.
  - (2) The test plan should be made well before the start of the test.
    - In fact, once the requirements model is completed, the test plan can be developed, and the detailed test plan can be designed immediately after the design model is established. Therefore, all test work can be planned and designed before coding.
  - (3) Apply the Pareto principle to software testing.
    - Pareto principle shows that 80% of the errors found in the test are likely to be caused by 20% of the modules in the program. Of course, the question is how to find these suspicious modules and test them thoroughly.

Guidelines of Software Testing

- In order to design an effective test plan, software engineers must thoroughly understand and correctly use the basic principles that guide software testing.
  - (4) It should start with "small-scale" testing and gradually carry out "large-scale" testing.
    - Usually, first focus on testing a single program module, then turn the focus of testing to find errors in the integrated module cluster, and finally find errors in the entire system.
  - (5) Exhaustive testing is impossible.
    - The so-called exhaustive test is a test that checks all possible execution paths of the program. Even a medium-scale program has a huge number of execution paths. Due to time, manpower and resource constraints, it is impossible to execute every possible path during the test. Therefore, the test can only prove that there are errors in the program, not that there are no errors in the program. However, by carefully designing the test plan, it is possible to fully cover the program logic and make the program achieve the required reliability.
  - (6) In order to achieve the best test results, an independent third party should perform the test work.
    - The so-called "best effect" refers to the test that has the greatest probability of finding errors. For the reasons mentioned above, software engineers who develop software are not the best candidates to complete all the testing work (usually they are mainly responsible for module testing).

Testing methods

- There are two ways to test any product:
  - If you already know the functions that the product should have, you can test whether each function can be used normally;
  - If you know the internal working process of the product, you can test whether the internal actions of the product are proceeding normally according to the specifications.
- The former method is called black box testing, and the latter method is called white box testing.
  - For software testing, the black box testing method treats the program as a black box, and does not consider the internal structure and processing of the program at all. In other words, the black box test is a test performed on the program interface. It only checks whether the program functions can be used normally according to the specifications, whether the program can properly receive input data and generate correct output information. Whether to maintain the integrity of external information. Black box testing is also called functional testing.
  - The white box testing method is opposite to the black box testing method. Its premise is that the program can be regarded as being installed in a transparent white box, and the tester fully knows the structure and processing algorithm of the program. This method follows the logic test program inside the program to check whether the main execution path in the program can work correctly according to the predetermined requirements. White box testing is also called structural testing.

Steps of Testing

- Unless you are testing a small program, it is unrealistic to test the entire system as a single entity from the beginning.
- A large-scale software system is usually composed of several subsystems, and each subsystem is composed of many modules. Therefore, the testing process of a large-scale software system basically consists of the following steps.

Steps of Testing

- 1. Module testing
  - In a well-designed software system, each module completes a clearly defined sub-function, and there is no interdependence between this sub-function and the functions of other modules at the same level.
  - Therefore, it is possible to test each module as a separate entity, and it is usually easier to design a test plan to verify the correctness of the module.
  - The purpose of module testing is to ensure that each module can run correctly as a unit, so module testing is usually called unit testing.
  - What is found in this test step is often coding and detailed design errors.

Steps of Testing

- 2. Subsystem testing
  - Subsystem testing is to put together unit-tested modules to form a subsystem for testing.
  - The coordination and communication between the modules are the main problems in this testing process. Therefore, this step focuses on testing the interface of the modules.

Steps of Testing

- 3. System Test
  - System testing is to assemble the tested subsystems into a complete system for testing.
  - In this process, not only should the design and coding errors be found, but it should also be verified that the system can indeed provide the functions specified in the requirements specification, and that the dynamic characteristics of the system also meet the predetermined requirements.
  - What is found in this test step is often an error in the software design, and an error in the requirement specification may also be found.
  - Whether it is subsystem testing or system testing, both have the dual meanings of testing and assembly, and are usually called integration testing.

Steps of Testing

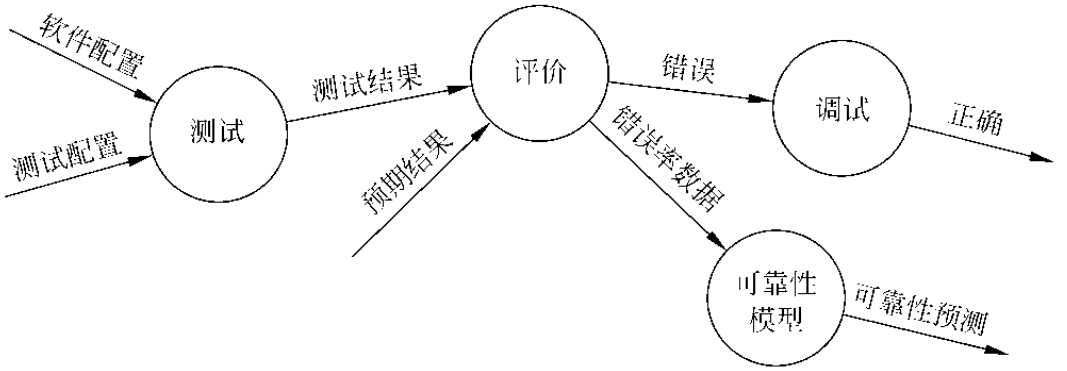
- 4. Acceptance test
  - Acceptance testing tests the software system as a single entity. The test content is basically similar to system testing, but it is carried out with the active participation of users, and may mainly use actual data (information that the system will process in the future) for testing.
  - The purpose of the acceptance test is to verify that the system can indeed meet the needs of users. What is found in this test step is often the error in the system requirements specification.
  - Acceptance testing is also called confirmation testing.

Steps of Testing

- 5. Parallel operation
  - Important software products are often not put into productive operation immediately after acceptance, but have to pass the test of a period of parallel operation time.
  - The so-called parallel operation is to run the newly developed system and the old system that will be replaced by it at the same time, in order to compare the processing results of the new and old systems.
  - The specific purposes are as follows:
    - (1) The new system can be run in a quasi-production environment without taking risks;
    - (2) Users can have a period of time to be familiar with the new system;
    - (3) Documents such as user guides and manuals can be verified;
    - (4) The new system can be tested at full load in a quasi-production mode, and the test results can be used to verify performance indicators.

Information flow during the testing phase

- The following figure depicts the information flow of the testing phase. There are two types of input information at this stage:
  - (1) Software configuration, including requirements specification, design specification and source program list, etc.;
  - (2) Test configuration, including test plan and test plan.



- The so-called test plan is not only the input data used in the test (called test cases), but also the functions that each set of input data is scheduled to be tested, and the correct output expected for each set of input data.
- In fact, the test configuration is a subset of the software configuration, and the final software configuration should include the above-mentioned test configuration, the actual results of the test, and the record of debugging.

Information flow during the testing phase

- When the test results were collected and evaluated, the qualitative indicators of software reliability began to become clear.
  - If serious errors that require modification of the design frequently occur, then the quality and reliability of the software are questionable and should be further tested carefully.
  - Conversely, if it seems that the software functions are completed normally and the errors encountered are easily corrected, there are still two possibilities that should be considered:
    - (1) The reliability of the software is acceptable;
    - (2) The tests performed are not enough to find serious errors.
  - If an error has not been found after testing, it is probably because of insufficient thinking about the test configuration, so that the hidden errors in the software cannot be exposed.
  - These errors will eventually be discovered by the user, and they need to be corrected in the maintenance phase (but the cost of correcting the same error is many times higher than in the development phase).
  - The results accumulated during the testing phase can also be evaluated in a more formal way. The software reliability model uses the error rate data to estimate the future occurrence of errors, and then predicts the software reliability.

Unit Testing

- Unit testing focuses on detecting the smallest unit of software design-module.
  - Usually, unit testing and coding belong to the same stage of the software process.
- After writing the source code and passing the grammar check of the compiler, you can use the detailed design description as a guide to test important execution paths in order to find errors in the module.
- Two different types of testing methods, such as manual testing and computer testing, can be used to complete unit testing.
  - The two test methods have their own strengths and complement each other.
  - Usually, unit testing mainly uses white box testing technology, and testing of multiple modules can be performed in parallel.

Unit Testing-key issues

- 1. Module interface
  - The data flow through the module interface should be tested first. If the data cannot be entered and exited correctly, all other tests are impractical.
  - When testing the module interface, the following aspects are mainly checked:
    - Whether the number, order, attribute or unit system of the parameters are consistent with the arguments;
    - Whether the argument that is only used for input is modified;
    - Whether the definition and usage of global variables are consistent in each module.
- 2. Local data structure
  - For modules, local data structures are a common source of errors. The test plan should be carefully designed to find errors in local data description, initialization, default values, etc.
- 3. Important execution pathways
  - Since it is usually impossible to perform exhaustive testing, it is critical to select the most representative and most likely execution path for testing during unit testing. A test plan should be designed to find errors caused by incorrect calculations, incorrect comparisons, or inappropriate control flow.

Unit Testing-key issues

- 4. Error handling path
  - A good design should be able to foresee error conditions, and set up an appropriate path for handling errors, so that when an error occurs, the corresponding error handling path is executed or the processing ends cleanly. Not only should error-handling paths be included in the program, but such paths should be carefully tested. When evaluating error handling paths, you should focus on testing the following possible errors:
    - (1) The description of the error is difficult to understand;
    - (2) The recorded error is different from the actual error encountered;
    - (3) Before handling the error, the error condition has caused system intervention;
    - (4) Incorrect handling of errors;
    - (5) The information describing the error is not enough to help determine the location of the error.
- 5. Boundary conditions
  - Boundary testing is the last and possibly the most important task in unit testing. Software often fails on its boundaries
    - For example, when processing the n-th element of an n-element array, or when doing the i-th repetition in the i-th loop, errors often occur. Using test schemes that use data structures, control quantities, and data values that are just less than, just equal to, and just greater than the maximum or minimum value, it is very likely to find errors in the software.

Unit Testing-code examination

- Manual testing of the source program can be carried out informally by the author himself, or formally by the review team. The latter is called code review, and it is a very effective program verification technique
  - For a typical program, 30% to 70% of logic design errors and coding errors can be detected.
- The review team should preferably consist of the following 4 persons:
  - (1) The group leader, who should be a very capable programmer, and has not directly participated in the project;
  - (2) The designer of the program;
  - (3) The writer of the program;
  - (4) The tester of the program.

Unit Testing-code examination

- If a person is both the designer and the writer of the program, or both the writer and the tester, then another programmer should be added to the review team.
- Before the review, the team members should first study the design specification and strive to understand the design. To help understand, the designer can first briefly introduce his design.
- At the review meeting, the programmer of the program explained how he implemented the design with program code, usually describing the logic of the program sentence by sentence. Other members of the team listened carefully to his explanation and tried to find the error.
- Another task carried out at the review meeting was to analyze and review the program against the list of common program design errors similar to the one introduced in the previous section.
- When errors are found, the team leader will record them and the review will continue (the task of the review team is to find errors rather than correct them).

Unit Testing-code examination

- There is another common method for review meetings, called walkthrough: one person acts as the "tester" and the other person acts as the "computer".
  - Before the meeting, the tester prepares the test plan, and at the meeting, the members who play the computer simulate the computer to execute the tested program. Of course, because people are extremely slow to execute programs, the test data must be simple, and the number of test schemes cannot be too much. However, the test plan itself is not very critical, it only plays a role in promoting thinking and discussion.
  - In most cases, by asking the programmer about the logic of his program and the assumptions he made when writing the program, more errors can be found than those directly found by the test plan.

Unit Testing-code examination

- The advantage of code review over computer testing is
  - Many errors can be found in a review meeting; after a computer test is used to find an error, it is usually necessary to correct the error before continuing the test, so the errors are found and corrected one by one.
    - That is, the use of code review methods can reduce the total workload of system verification.
- Practice has shown that for finding certain types of errors, manual testing is more effective than computer testing; for other types of errors, the opposite is true.
- Therefore, manual testing and computer testing are complementary to each other, and the lack of any of these methods will reduce the efficiency of finding errors.

Computer-aided Test

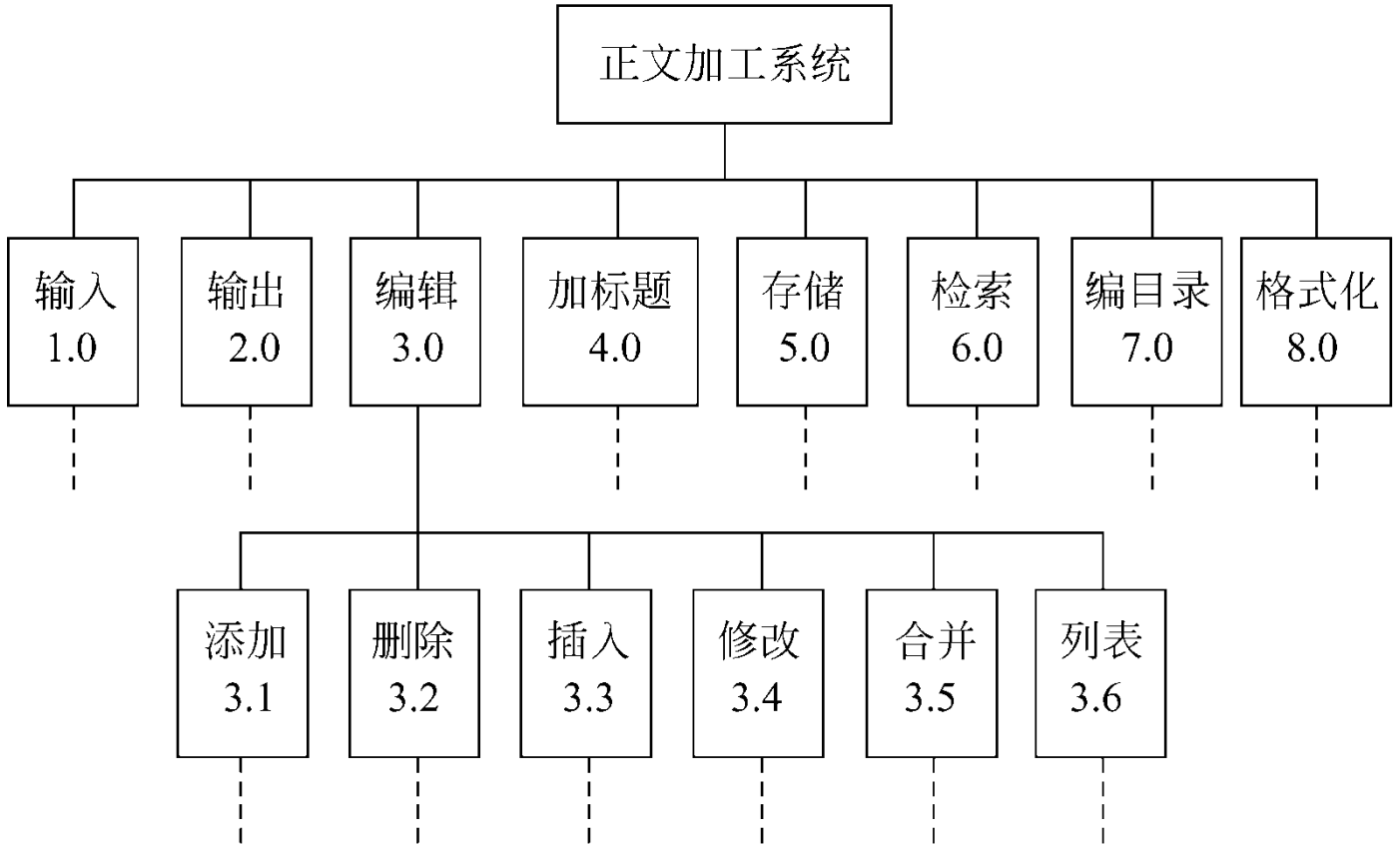
- The module is not an independent program, so the driver software and/or stub software must be developed for each unit test.
  - Usually the driver is a "main program", which receives test data, transmits these data to the tested module, and prints out the relevant results.
  - The stub program replaces the module called by the module under test. Therefore, stub programs can also be called "virtual subroutines".
    - It uses the interface of the module that it replaces, may do the least amount of data operations, print out the inspection or operation results of the entry, and return control to the module that called it.

Computer-aided Test

- For example, given a partial hierarchical diagram of a text processing system, suppose to test the key module numbered 3.0—the text editing module.
  - Because the text editing module is not an independent program, a test driver is needed to call it.
  - This driver explains the necessary variables, receives test data-character strings, and sets the editing function of the text editing module.
  - Because in the original software structure, the text editing module completes specific editing functions by calling its lower-level modules, so a stub program is needed to simplify the simulation of these lower-level modules.
  - For the sake of simplicity, the only editing functions that can be set during testing are CHANGE and APPEND. The required editing functions are marked with the control variable CFUNCT, and only one stub program is used to simulate all lower-level modules of the text editing module.

Computer-aided Test

- For example, given a partial hierarchical diagram of a text processing system, suppose to test the key module numbered 3.0—the
  - Because the t
  - This driver ex
    function of th
  - Because in the
    calling its lowe
    modules.
  - For the sake
    APPEND. The
    program is use



d to call it.
d sets the editing
c editing functions by
of these lower-level
e CHANGE and
T, and only one stub

Computer-aided Test

- The following uses pseudocode written stub programs and drivers.

```
I . TEST STUB (*Stub program for test text editing module*)
      Initialization
      Output message "entered the text editing program";
      Output "input control information is" CFUNCT;
      String in the output buffer;
      IF CFUNCT=CHANGE
            THEN
            Change the second word in the buffer to ***
            ELSE
            Add ??? at the end of the buffer
      END IF;
      The new string in the output buffer;
END TEST STUB
```

Computer-aided Test

- The following uses pseudocode written stub programs and drivers.

Ⅱ. TEST DRIVER (*Driver program for test text editing module*)

Description of a buffer with a length of 2500 characters;

Set CFUNCT to the state you want to test;

Input string;

Call the text editing module;

Stop or start again;

END TEST DRIVER

Computer-aided Test

- Drivers and stub programs represent overhead, that is, test software must be written in order to perform unit testing, but they are usually not given to users as part of the software product.
  - Many modules cannot be fully tested with simple test software. In order to reduce the cost, the incremental test method that will be introduced in the next section can be used to complete the detailed test of the module at the same time during the integration test.
- The high degree of module cohesion can simplify the unit testing process.
  - If each module only completes one function, the number of required test schemes will be significantly reduced, and the errors in the module will be easier to predict and find.

Integrated Testing

- Integration testing is a systematic technique for testing and assembling software
  - For example, subsystem testing is to test while assembling the modules according to the design requirements. The main goal is to find problems related to the interface (system testing is similar to this).
  - For example, data may be lost when passing through the interface; one module may have harmful effects on another module due to negligence; combining sub-functions may not produce the expected main function; individual errors that seem acceptable may accumulate to be unacceptable The extent of the data structure; there may be problems with the whole data structure and so on.
  - Unfortunately, there are too many interface problems that can occur.
- There are two ways to assemble a program from modules.
  - One method is to test each module separately, and then put all the modules together according to the design requirements to form the desired program. This method is called a non-incremental test method;
  - Another method is to combine the next module to be tested with those modules that have already been tested for testing, and then combine the next module to be tested for testing after the test is completed.
    - This method of adding one module at a time is called incremental testing. This method actually completes unit testing and integration testing at the same time.

Integrated Testing

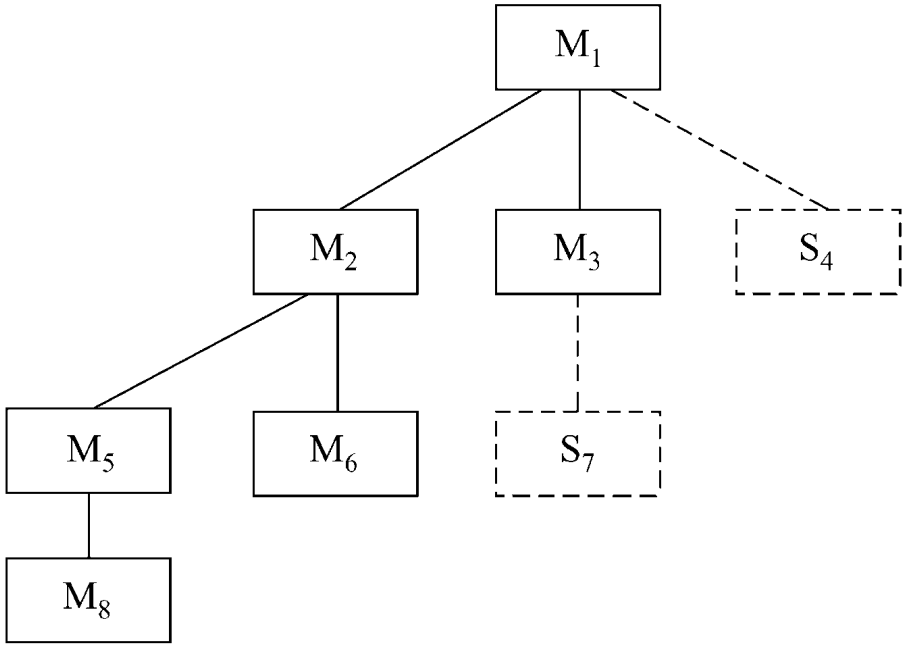
- The main advantages and disadvantages of non-incremental testing:
  - Non-incremental testing puts all the modules together at once, and tests the huge program as a whole. The situation faced by the tester is very complicated. Many errors will be encountered during testing, and it is extremely difficult to correct them, because it is very difficult to diagnose and locate an error in a huge program. And once an error is corrected, a new error will be encountered immediately, and the process will continue, and it seems that there will never be an end to it.
  - Incremental testing is the opposite of "one step in place" non-incremental testing. It divides the program into small sections for construction and testing. In this process, it is easier to locate and correct errors; the interface can be tested more thoroughly; it can be used Systematic testing method. Therefore, the incremental testing method is generally used in integration testing.

Integrated Testing

- When using the incremental method to integrate the module into the program, there are two integration strategies
  - Top down
  - Bottom up

Top down integration

- The top-down integration approach is an increasingly widely adopted approach to testing and assembling software.
  - Starting from the main control module, moving down the control level of the program, gradually combining the various modules.
  - When assembling the modules attached to (and ultimately attached to) the main control module into the program structure, either the depth-first strategy or the breadth-first strategy is used.
- The depth-first combination method first assembles all modules on a main control path of the software structure.
  - The choice of a main control path depends on the characteristics of the application and is very arbitrary.
- The width-first combination method is to move horizontc ll modules on the same control level.



Top down integration

- The specific process of integrating the module into the software structure is completed by the following 4 steps:
  - The first step is to test the main control module, and replace all modules directly attached to the main control module with a stub program during the test;
  - In the second step, according to the selected combination strategy (depth first or width first), a stub program is replaced with one actual module each time (newly combined modules often require new stub programs);
  - The third step is to test while integrating into a module;
  - In the fourth step, in order to ensure that no new errors are introduced by adding the module, regression testing (that is, all or part of the previous tests should be repeated).

- From the second step, continue to repeat the above process until the complete software structure is constructed.
- The top-down combination strategy can test the main controls or key choices early in the test. In a well-decomposed software structure, the key decisions are located in the upper layer of the hierarchical system, so they are encountered first. If there are indeed problems with the main control, it is very beneficial to recognize such problems early and you can find ways to solve them early. If you choose the depth-first combination method, you can realize a complete function of the software and verify this function at an early stage. Early confirmation of a complete function of the software can enhance the confidence of both developers and users.

Top down integration

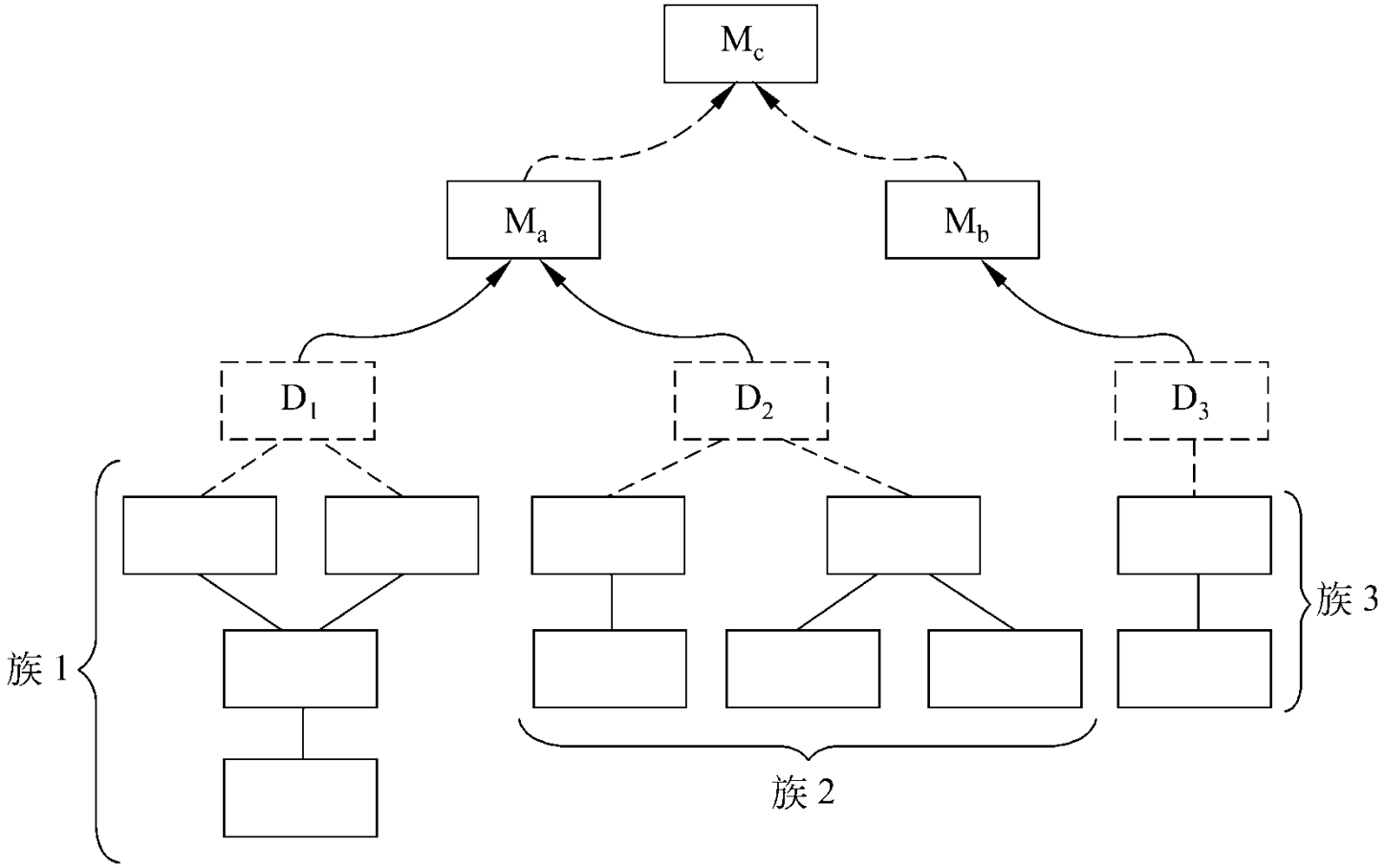
- The top-down method may encounter logical problems in actual use
  - In order to fully test the higher level of the software system, processing at the lower level is required. However, in the early stages of top-down testing, stub programs replaced low-level modules, so there is no important data in the software structure that flows from bottom to top.
  - In order to solve this problem, testers have two options:
    - First, postpone many tests until after replacing the stub programs with real modules;
      - The first method loses the precise correspondence between the specific test and the assembly of the specific module, which may lead to difficulties in determining the location and cause of the error.
    - Second, the software is assembled from the bottom of the hierarchical system upwards.

Bottom up integration

- Bottom-up testing starts with the "atomic" module (that is, the module at the lowest level of the software structure) assembling and testing.
  - Because the modules are combined from the bottom up, the required lower-level module processing functions can always be obtained, so there is no need for stub programs.
- The bottom-up combination strategy can be realized with the following steps:
  - The first step is to combine low-level modules into a family that realizes a specific software sub-function;
  - The second step is to write a driver (control program for testing) to coordinate the input and output of test data;
  - The third step is to test the sub-function family composed of modules;
  - The fourth step is to remove the driver and move from bottom to top along the software structure to combine the sub-function families to form a larger sub-function family.
- As the combination moves upward, the need for test drivers is also reduced. In fact, if the top two layers of the software structure are assembled in a top-down method, the number of drivers can be significantly reduced, and the combination of families will be greatly simplified.

Bottom up integration

- Bottom-up testing starts with the "atomic" module (that is, the module at the lowest level of the software structure) assembl
  - Because the e processing functions ca
- The bottom-up co
  - The first ste tware sub-function;
  - The second s out and output of test data;
  - The third st
  - The fourth s re structure to combine the
- As the combinatio top two layers of the software stru significantly reduced, and the



Comparison of different integration testing strategies

- Generally speaking, the advantages of one method correspond to the disadvantages of another method.
  - The main advantage of the top-down test method is that it does not require a test driver, can realize and verify the main functions of the system in the early test phase, and can find the interface errors of the upper module in the early stage.
  - The main disadvantage of the top-down test method is the need for stub programs, which may encounter test difficulties associated with this, the detection of errors in low-level key modules is late, and this method cannot fully deploy manpower in the early stages.
  - The advantages and disadvantages of the bottom-up test method are just the opposite of the advantages and disadvantages of the above-mentioned top-down test method.

Comparison of different integration testing strategies

- When testing the actual software system, an appropriate testing strategy should be selected according to the characteristics of the software and the schedule of the project. Generally speaking, a purely top-down or purely bottom-up strategy may not be practical. In practice, a mixed strategy can be used:

(1) Improved top-down testing method.

- Basically use the top-down test method, but use the bottom-up method to test a few key modules in the software in the early days. The advantages of the general top-down method are also available in this method, and errors in key modules can be found early in the test;
    - It also has one more shortcoming than the top-down approach, that is, a driver is required when testing key modules.

(2) Mixed method.

- The top-down method used for the upper layer of the software structure is combined with the bottom-up method used for the lower layer of the software structure.
    - This method has the advantages and disadvantages of both methods. When there are more key modules in the software being tested, this hybrid method may be the best compromise method.

Regression Testing

- In the integration test process, whenever a new module is incorporated, the program changes: a new data flow path is established, a new I/O operation may appear, and a new control logic is activated.
- These changes may cause problems with functions that were originally working properly.
- In the category of integration testing, the so-called regression testing refers to re-executing a certain subset of the tests that have been done to ensure that the above-mentioned changes do not bring unintended side effects.
- More broadly, any successful test will find errors, and the errors must be corrected.
    - Whenever a software error is corrected, certain components of the software configuration (programs, documents, or data) are also modified.
- Regression testing is a testing activity used to ensure that changes due to debugging or other reasons will not lead to unexpected software behavior or additional errors.
- Regression testing can be performed manually by re-executing a subset of all test cases, or it can be performed automatically using automated capture and playback tools.
    - Using the capture and playback tool, software engineers can capture test cases and actual running results, and then replay (ie re-execute the test cases), and compare the running results before and after the software changes.

Regression Testing

- The regression test set (a subset of the executed test cases) includes the following three different test cases:
  - (1) Representative test cases for testing all functions of the software;
  - (2) Additional testing specifically for software functions that may be affected by the modification;
  - (3) Testing for software components that have been modified.
- In the process of integration testing, the number of regression test cases may become very large.
  - Therefore, the regression test set should be designed to include only those test cases that can detect one or more types of errors in each main function of the program.
  - Once the software is modified, it is inefficient and impractical to re-execute all test cases for each function of the test program.

Confirmation test

- Confirmation testing is also called acceptance testing, and its goal is to verify the effectiveness of the software.
- Generally, verification refers to a series of activities to ensure that the software correctly implements a specific requirement, while verification refers to a series of activities to ensure that the software does meet the needs of users.

Scope of confirmation test

- Confirmation testing must be actively participated by users, or user-oriented.
  - Users should participate in the design of test plans, use the user interface to input test data and analyze and evaluate the output results of the test.
- In order to enable users to actively participate in the confirmation test, especially to enable users to use the system effectively, the development unit usually trains users before acceptance.
- Confirmation testing usually uses black box testing.
  - The test plan and test process should be carefully designed. The test plan includes the types of tests to be performed and the schedule. The test process specifies the test plan used to check whether the software is consistent with the requirements.
  - Through testing and debugging, it is necessary to ensure that the software can meet all functional requirements and meet each performance requirement. The documentation is accurate and complete. In addition, it should also be ensured that the software can meet other predetermined requirements (for example, security, portability, Compatibility and maintainability, etc.).
- The confirmation test has the following two possible results:
  - (1) The functions and performance are consistent with user requirements, and the software is acceptable;
  - (2) There is a gap between functions and performance and user requirements.
- In order to formulate a strategy for solving software defects or errors found in the confirmation test process, it is usually necessary to fully negotiate with the user.

Software configuration review

- An important part of the confirmation test is to review the software configuration.
  - The purpose of the review is to ensure that all the components of the software configuration are complete, the quality meets the requirements, the documents and procedures are completely consistent, there are details necessary to complete the software maintenance, and the catalog has been compiled.
- In addition to the manual review of the software configuration in accordance with the content and requirements of the contract, the user guide and other operating procedures should be strictly followed in the confirmation test process to verify the completeness and correctness of these user manuals. The omissions or errors found must be carefully recorded and supplemented and corrected appropriately.

Alpha and Beta testing

- If the software is specifically developed for a certain customer, a series of acceptance tests can be carried out so that the user can confirm that all requirements have been met.
- Acceptance testing is performed by the end user, not the developer of the system.
  - Acceptance testing can last for several weeks or even months, so it is possible to find cumulative errors that may degrade the quality of the system over time.
- If a piece of software is developed for many customers (for example, a boxed software product sold to the public), it is unrealistic to have each customer undergo a formal acceptance test. In this case, the vast majority of software developers use processes called Alpha testing and Beta testing to find errors that seem to be discovered only by the end user.

Alpha and Beta testing

- Alpha testing is carried out by the user at the developer's premises, and the test is carried out under the "guidance" of the developer to the user. The developer is responsible for recording errors found and problems encountered during use. Alpha testing is conducted in a controlled environment.
- Beta testing is conducted by end users of the software at one or more customer sites. Unlike the Alpha test, the developer is usually not on the beta test site.
    - Therefore, Beta testing is a "real" application of software in an environment beyond the control of the developer. Users record all the problems (real or imaginary) encountered during the Beta testing process, and report these problems to the developer on a regular basis.
    - After receiving the problems reported during the Beta test, the developer makes necessary modifications to the software product and prepares to release the final software product to all customers.

White box testing

- Designing a test plan is a key technical issue in the testing phase.
  - The so-called test plan includes the specific test purpose (for example, the specific function scheduled to be tested), the test data that should be input and the expected result.
  - Usually the test data and expected output results are called test cases. One of the most difficult problems is designing input data for testing.
- The ability of different test data to find program errors varies greatly. In order to improve test efficiency and reduce test costs, efficient test data should be selected.
  - Because it is impossible to perform exhaustive tests, it is even more important to choose a small amount of "the most effective" test data to achieve as complete a test as possible.
- The basic goal of designing a test plan is to determine a set of test data that is most likely to find a certain error or a certain type of error.
  - Many techniques for designing test data have been developed. These techniques have their own advantages and disadvantages. No one is the best, and no one can replace all the other technologies; the same technology may have very different effects in different applications. Therefore, it is usually necessary to jointly use multiple techniques for designing test data.

Logical coverage

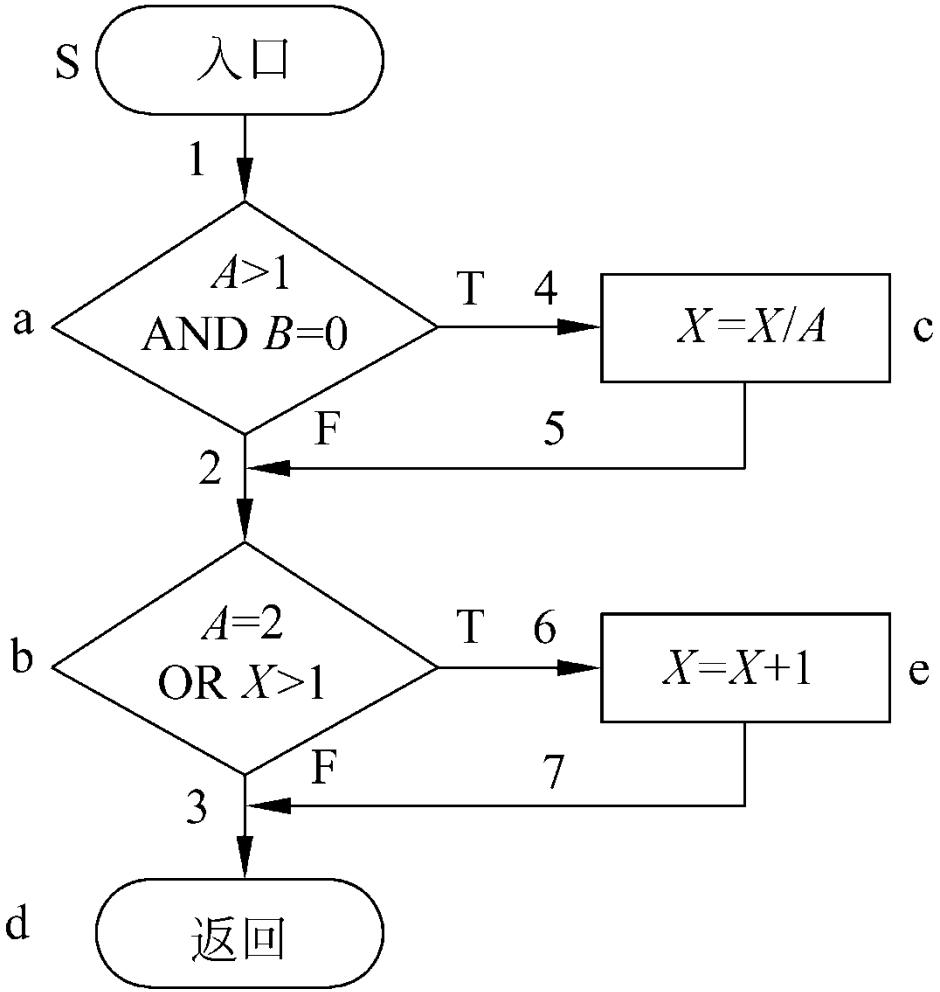
- The so-called logical coverage is the general term for a series of test processes, which gradually carry out more and more complete path tests.
- Covering standards
  - Statement coverage
  - Decision coverage
  - Conditional coverage
  - Judgment/condition coverage
  - Conditional combination coverage
  - Point coverage
  - Edge cover
  - Path coverage

Logical coverage

- 1. Statement coverage
  - In order to expose errors in the program, at least each statement should be executed once. The meaning of statement coverage is to select enough test data so that each statement in the program under test is executed at least once. For example, the program flow chart shown in Figure 7.5 depicts the processing algorithm of a module under test.
  - In order to execute each statement once, the execution path of the program should be sacbed. For this, you only need to enter the following test data (in fact, X can be any real number): A=2, B=0, X=4
  - Statement coverage rarely covers the logic of the program. In addition, statement coverage only cares about the value of the decision expression, and does not separately test the situation when each condition in the decision expression takes a different value.
  - In summary, it can be seen that statement coverage is a very weak logic coverage standard. In order to test the program more fully, the following logic coverage standards can be used.

Logical coverage

- 1. Statement coverage
  - In order to expose errors
  - In order to execute each
  - Statement coverage rare
  - In summary, it can be see



Logical coverage

- 2. Determining coverage
  - Decision coverage is also called branch coverage. It means that not only must each statement be executed at least once, but every possible result of each decision should be executed at least once, that is, every branch of each decision must be executed at least once .
  - Judgment coverage is stronger than statement coverage, but the coverage of program logic is still not high. For example, the above test data only covers half of the entire path of the program.

Logical coverage

- 3. Conditional coverage
  - The meaning of conditional coverage is that not only each statement is executed at least once, but also that each condition in the judgment expression can obtain various possible results.
  - Condition coverage is usually stronger than decision coverage, because it makes each condition in the decision expression get two different results, but decision coverage only cares about the value of the entire decision expression.

Logical coverage

- 4. Judgment/condition coverage
  - Since decision coverage does not necessarily include conditional coverage, and conditional coverage does not necessarily include decision coverage, it is natural to propose a logical coverage that can meet these two coverage standards at the same time, which is decision/condition coverage.
  - Its meaning is to select enough test data so that each condition in the judgment expression can take various possible values, and each judgment expression can also take various possible results.
  - Sometimes decision/condition coverage is not stronger than condition coverage.

Logical coverage

- 5. Condition combination coverage
  - Conditional combination coverage is a stronger logical coverage standard. It requires selecting enough test data to make each possible combination of conditions in each decision expression appear at least once.
  - Obviously, the test data that meets the condition combination coverage standard must also meet the judgment coverage, condition coverage, and judgment/condition coverage standards. Therefore, the conditional combination coverage is the strongest among the aforementioned coverage standards. However, the test data that meets the conditional combination coverage standard may not necessarily enable every path in the program to be executed.

Logical coverage

- Based on the detailed level of the source program statement detection based on the test data, several logic coverage standards are briefly discussed. In the above analysis process, the program path of the test data execution is often talked about. Obviously, the number of program paths that the test data can detect also reflects the degree of detail of the program test. From the analysis of the degree of coverage of the program path, the following main logic coverage standards can be put forward.

Logical coverage

- 6. Point coverage
  - The concept of point coverage in graph theory is defined as follows: If the subgraph G' of the connected graph G is connected and contains all the nodes of G, then G' is called the point coverage of G.
  - Under normal circumstances, the flow graph is a connected directed graph. To meet the requirements of the point coverage standard, select enough test data so that the program execution path passes through each node of the flow graph at least once. Since each node of the flow graph corresponds to one or more sentences, obviously, the point coverage standard and The statement coverage criteria are the same.

Logical coverage

- 7. Edge Covering
  - The definition of edge cover in graph theory is: if the subgraph G" of the connected graph G is connected and contains all the edges of G, then G" is said to be the edge cover of G. In order to meet the test criteria for edge coverage, it is required to select enough test data so that the program execution path passes through each edge in the flow graph at least once. Usually edge coverage and decision coverage are the same.
- 8. Path coverage
  - The meaning of path coverage is to select enough test data so that each possible path of the program is executed at least once (if there are loops in the program diagram, each loop is required to pass at least once).

Control structure test

- 1. Basic path test
  - Basic path testing is a white box testing technique proposed by Tom McCabe. When using this technology to design test cases, first calculate the circular complexity of the program, and use this complexity as a guide to define the basic set of execution paths. The test cases derived from this basic set can ensure that each statement in the program is executed at least once. And each condition will take two values of true and false when it is executed.
  - The steps for designing test cases using basic path testing techniques are as follows:
    - The first step is to draw the corresponding flow diagram based on the process design results.
    - The second step is to calculate the loop complexity of the flow graph.
      - The ring complexity is a quantitative measure of the logical complexity of a program.
      - With the flow graph depicting the control flow of the program, the loop complexity can be calculated. The loop complexity of the flow graph in the example graph is calculated to be 6.
    - The third step is to determine the basic set of linear independent paths.
    - The fourth step is to design test cases that can enforce each path in the basic set.

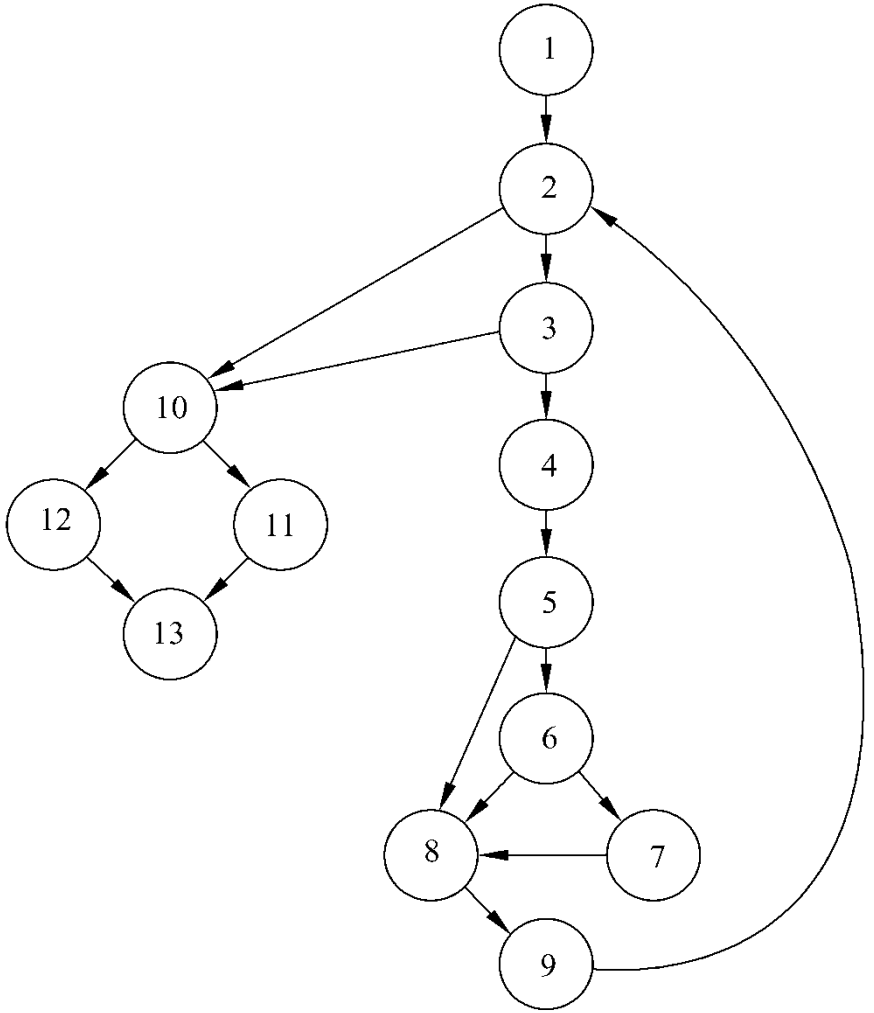
Control structure test

- For example, in order to use the basic path test technique to test the following averaging process described in PDL, first draw the flow diagram shown in Figure 7.6. Note that in order to draw the flow graph correctly, we serialized the PDL statements that are mapped to the flow graph nodes.

```
PROCEDURE average;
/* This process calculates the average value of no more than 100 significant digits within the specified value
range; at the same time calculates the sum and number of significant digits. */
INTERFACE RETURNS average, total.input, total.valid;
    INTERFACE ACCEPTS value, minimum, maximum;
    TYPE value   [ 1…100 ]   IS SCALAR ARRAY;
    TYPE average, total.input, total.valid;
minimum,maximum, sum IS SCALAR;
    TYPE i IS INTEGER;
```

Control structure test

```
1:    i=1;
          total.input=total.valid=0;
          sum=0;
2:    DO WHILE value  [ i ]  <> -999
3:          AND total.input<100
4:          increment total.input by1;
5:          IF value  [ i ]  >=minimum
6:          AND value  [ i ]  <=maximum
7 :               THEN increment total.valid by 1;
                  sum=sum+value  [ i ]  ;
8:          ENDIF
          increment i by 1;
9:    ENDDO
10:   IF total.valid>0
11:         THEN average=sum/total.valid;
12:   ELSE average=-999;
13:   ENDIF
END average
```



In this example, nodes 2, 3, 5, 6, and 10 are decision nodes.

Control structure test

- The so-called independent path refers to a path that introduces at least a new set of processing sentences or a new condition into the program. It is described in flow graph terms. The independent path contains at least one edge that has not been used before the path is defined.
- When using the basic path testing method to design test cases, the loop complexity of the program determines the number of independent paths in the program, and this number is the upper bound of the number of tests required to ensure that all statements in the program are executed at least once.
- Usually when designing test cases, it is necessary to identify the judgment nodes.
- The test data should be selected so that the conditions of each judgment node are appropriately set when testing each path.
- During the testing process, each test case is executed and the actual output result is compared with the expected result. Once all test cases are executed, it can be ensured that all statements in the program have been executed at least once, and that each condition has taken the true value and the false value respectively.
- It should be noted that certain independent paths cannot be tested in an independent manner, that is, the normal flow of the program cannot form the data combination required to execute the path independently. In this case, these paths must be tested as part of another path.

Control structure test

- 2. Condition test
  - Although the basic path testing technology is simple and efficient, this technology alone is not enough, and other control structure testing technologies are needed to further improve the quality of white box testing.
  - Test cases designed with conditional testing technology can check the logical conditions contained in program modules. A simple condition is a Boolean variable or a relational expression, and there may be a NOT operator before the Boolean variable or relational expression. The form of the relational expression is as follows:
    - E1<relational operator>E2
      - Among them, E1 and E2 are arithmetic expressions, and <relational operator> is one of the following operators: "<", "≤", "=", "≠", ">" or "≥". Compound conditions consist of two or more simple conditions, Boolean operators, and parentheses. The Boolean operators are OR ("|"), AND ("&") and NOT. Conditions that do not contain relational expressions are called Boolean expressions.

Control structure test

- 2. Condition test
  - The types of conditional components include Boolean operators, Boolean variables, Boolean brackets (enclose simple conditions or compound conditions), relational operators, and arithmetic expressions.
  - If the condition is incorrect, at least one component of the condition is incorrect. Therefore, the types of conditional errors are as follows:
    - Boolean operator error (the Boolean operator is incorrect, the Boolean operator is missing, or there are redundant Boolean operators)
    - Boolean variable wrong
    - Boolean wrong parenthesis
    - Wrong relational operator
    - Arithmetic expression is wrong

Control structure test

- 2. Condition test
  - The condition testing method focuses on testing each condition in the program. Conditional testing strategies have two advantages:
    - ①It is easy to measure the test coverage of conditions;
    - ②The test coverage of the conditions in the program can guide the design of additional tests.
  - The purpose of condition testing is not only to detect errors in program conditions, but also to detect other errors in the program.
    - If the test set of program P can effectively detect errors in conditions in P, it is likely that it can also effectively detect other errors in P.
    - In addition, if a test strategy is effective for detecting conditional errors, it is likely that the strategy is also effective for detecting other errors in the program.

Control structure test

- 2. Condition test
  - Branch testing may be the simplest condition testing strategy: for compound condition C, the true and false branches of C and each simple condition in C should be executed at least once.
  - Domain testing requires 3 or 4 tests to be performed on a relational expression. For a relational expression of the form E1<relational operator>E2, three tests are required to make the value of E1 greater than, equal to, or less than the value of E2. If the <relational operator> is wrong and E1 and E2 are correct, then these three tests can find the error of the relational operator. In order to find errors in E1 and E2, the test data that makes the E1 value greater or less than the E2 value should make the difference between these two values as small as possible.

Control structure test

- 2. Condition test
  - A Boolean expression containing n variables requires 2n tests (each variable takes the number of combinations of two possible values, true or false). This strategy can find errors in Boolean operators, variables and parentheses, but this strategy is only practical when n is very small.
  - Based on the above-mentioned conditional testing techniques, K.C.Tai proposed a conditional testing strategy called BRO (branch and relational operator) testing. If all Boolean variables and relational operators appear only once in the condition and there is no common variable, the BRO test guarantees that the branch error and relational operator error in the condition can be found.
    - BRO testing uses the conditional constraints of condition C to design test cases. The condition constraint of condition C containing n simple conditions is defined as (D1, D2,..., Dn), where Di($0<i\leq n$) represents the output constraint of the i-th simple condition in condition C. If during one execution of condition C, the output of each simple condition in C satisfies the corresponding constraint in D, it is said that this execution of C covers C's constraint D.
    - For the Boolean variable B, the output constraint of B states that B must be true (t) or false (f). Similarly, for relational expressions, use the symbols >, = and <to specify the output constraints of the expression.

Control structure test

- 3. Cycle test
  - Loops are the basis of most software algorithms, but the loop structure is often not adequately tested when testing software.
  - Loop testing is a white box testing technique that focuses on testing the validity of the loop structure. There are usually only three types of loops in structured programs, namely, simple loops, serial loops, and nested loop test methods.
  - (1) Simple cycle. The following test set should be used to test simple loops, where n is the maximum number of cycles allowed to pass.
    - Skip the loop.
    - Only go through the loop once.
    - Go through the loop twice.
    - Go through the loop m times, where m<n-1.
    - Through the loop n-1, n, n+1 times.

Control structure test

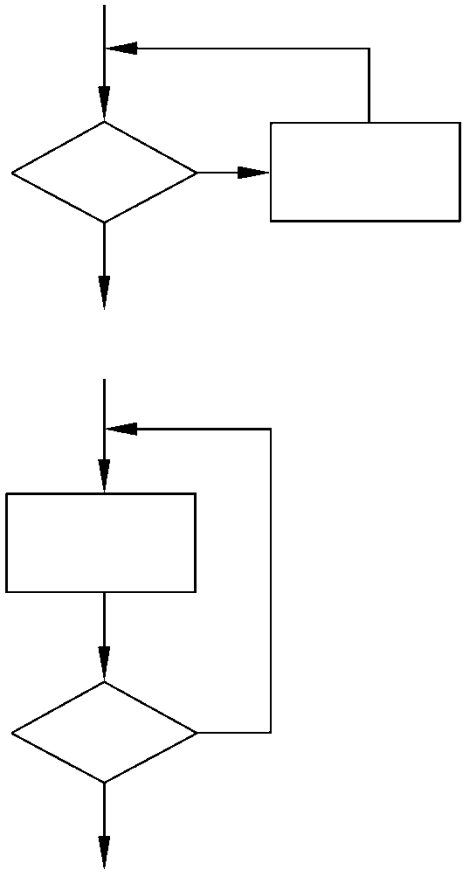
- 3. Cycle test
  - Loops are th
    when testing
  - Loop testing
    There are us
    nested loop
    - (1) Simp
      number
      - Sk
      - On
      - Go
      - Go
      - Th

adequately tested

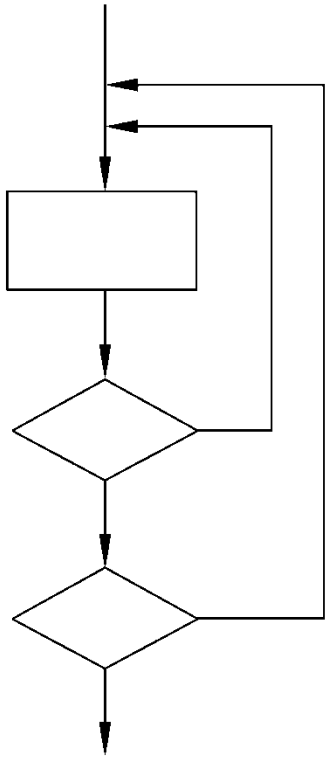
the loop structure.

ops, serial loops, and

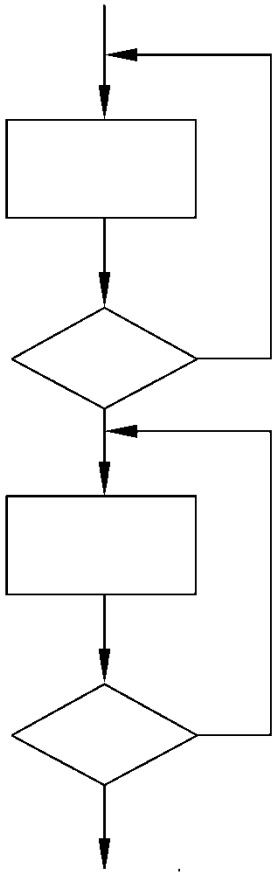
re n is the maximum



简单循环



嵌套循环



串接循环

Control structure test

- 3. Cycle test
  - (2) Nested loops. If the simple loop test method is directly applied to the nested loop, the number of possible tests will increase geometrically as the number of nesting levels increases, which will result in an unrealistic number of tests. B.Beizer proposed a method to reduce the number of tests:
    - Start the test from the innermost loop and set all other loops to the minimum.
    - Use a simple loop test method for the innermost loop, and make the iteration parameters of the outer loop (for example, loop counter) take the minimum value, and add some additional tests for out-of-bounds or illegal values.
    - From the inside to the outside, test the next loop, but keep all other outer loops to a minimum, and other nested loops to "typical" values.
    - Continue to continue until all cycles have been tested.

Control structure test

- 3. Cycle test
  - (3) Series connection loop. If each cycle of the cascade cycle is independent of each other, the aforementioned simple cycle test method can be used to test the cascade cycle. However, if two loops are connected in series, and the loop counter value of the first loop is the initial value of the second loop, the two loops are not independent. When the loops are not independent, it is recommended to use the method of testing nested loops to test series loops.

Black box testing

- Black box testing focuses on testing software functions.
  - Black box testing cannot replace white box testing. It is a complementary testing method to white box testing. It is likely to find other types of errors that are not easy to find in white box testing.
- Black box testing attempts to find the following types of errors:
  - ① Incorrect function or missing function;
  - ②Interface error;
  - ③Data structure error or external database access error;
  - ④Performance error;
  - ⑤ Initialization and termination errors.

Black box testing

- White box testing is carried out in the early stages of the testing process, while black box testing is mainly used in the later stages of the testing process. When designing a black box test plan, the following issues should be considered:
  - (1) How to test the effectiveness of the function?
  - (2) What types of input can constitute a good test case?
  - (3) Is the system particularly sensitive to specific input values?
  - (4) How to delimit the boundary of the data category?
  - (5) What data rate and data volume can the system bear?
  - (6) What impact will the specific combination of data have on the operation of the system?

Black box testing

- Using black box testing technology, it is possible to design a set of test cases that meet the following standards:
  - (1) The designed test cases can reduce the total number of test cases that need to be designed to achieve reasonable testing;
  - (2) The designed test case can tell us whether there are certain types of errors, rather than just indicate whether there are errors related to a specific test.

Black box testing

- Equivalence division
  - Equivalence division is a black-box testing technique. This technique divides the input field of the program into several data types and derives test cases accordingly. An ideal test case can find a class of errors by itself.
    - Exhaustive black box testing (that is, testing the program with all valid and invalid input data) is usually unrealistic. Therefore, only a small amount of the most representative input data can be selected as the test data in order to expose more program errors at a relatively small cost.
    - The equivalence division method attempts to design test cases that can find several types of program errors, thereby reducing the number of test cases that must be designed.

Black box testing

- Equivalence division
  - If all possible input data (valid and invalid) are divided into several equivalence classes, the following assumptions can be made reasonably: the role of a typical value in each class in the test is the same as that of all in this class. The other values have the same effect.
    - Therefore, only one set of data can be taken from each equivalence class as test data. The test data selected in this way is the most representative and most likely to find errors in the program.
  - Using the equivalence division method to design a test plan first needs to divide the equivalence classes of the input data. For this reason, it is necessary to study the function description of the program to determine the valid equivalence classes and invalid equivalence classes of the input data. When determining the equivalence class of the input data, it is often necessary to analyze the equivalence class of the output data, so as to derive the corresponding input data equivalence class according to the equivalence class of the output data.

Black box testing

- Equivalence division
  - Dividing equivalence classes requires experience. The following heuristic rules may help the division of equivalence classes:
    - (1) If the input value range is specified, one valid equivalence class (input value is within this range) and two invalid equivalence classes (input value is less than the minimum value or greater than the maximum value) can be divided;
    - (2) If the number of input data is specified, one valid equivalence class and two invalid equivalence classes can also be divided similarly;
    - (3) If a set of values of input data is specified, and the program treats different input values differently, each allowed input value is a valid equivalence class, in addition to an invalid equivalence class (any one Input value not allowed);
    - (4) If the rules that the input data must follow are specified, one valid equivalence class (conforming to the rules) and several invalid equivalence classes (violating the rules from various angles) can be divided;
    - (5) If the input data is specified as an integer, it can be divided into three valid categories: positive integer, zero and negative integer;
    - (6) If the processing object of the program is a table, an empty table and a table containing one or more items should be used.

Black box testing

- Equivalence division
  - The heuristic rules listed above are only a small part of the situations that may be encountered during testing. The actual situation is so varied that it is impossible to list them all.
    - In order to correctly classify equivalence classes, one must pay attention to accumulating experience, and the other is to correctly analyze the function of the program under test.
    - In addition, the error detection function of the compiler must be considered when dividing invalid equivalence classes. Generally speaking, there is no need to design test data to expose errors that the compiler can definitely find.
    - Although the heuristic rules listed above are all for the input data, most of them also apply to the output data.

Black box testing

- Equivalence division
  - After the equivalence class is divided, the following two steps are mainly used when designing the test plan according to the equivalence class:
    - (1) Design a new test plan to cover as many effective equivalence classes as possible, and repeat this step until all effective equivalence classes are covered;
    - (2) Design a new test plan to cover one and only one invalid equivalence class that has not been covered. Repeat this step until all invalid equivalence classes are covered.
  - Note that usually the program will not check whether there are other errors after it finds a type of error. Therefore, each test plan should only cover an invalid equivalence class.

Black box testing

- Equivalence division - example
  - The following uses the equivalent division method to design a test plan for a simple program.
    - Suppose there is a function that converts a string of numbers into integers. The word length of the computer running the program is 16 bits, and the integer is represented by two's complement. This function is written in Pascal language, and its description is as follows:
      - function strtoint (dstr:shortstr):integer;
    - The parameter type of the function is shortstr, and its description is:
      - type shortstr=array [1..6] of char;
    - The processed number string is right-justified, that is, if the number string is shorter than 6 characters, a space is added to the left of it. If the number string is negative, the minus sign and the highest digit are immediately adjacent (the minus sign is one to the left of the highest digit).

Black box testing

- Equivalence division - example
  - The following uses the equivalent division method to design a test plan for a simple program.
    - Analyzing the specifications of this program, the following equivalence classes can be divided:
    - The equivalent classes of valid inputs are
      - (1) A digit string composed of 1 to 6 digit characters (the highest digit is not zero);
      - (2) A string of digits with the highest digit being zero;
      - (3) A string of digits with a minus sign to the left of the most significant digit;
    - Equivalence classes for invalid input are
      - (4) Empty string (all spaces);
      - (5) The characters filled on the left are neither zeros nor spaces;
      - (6) The right side of the most significant number is composed of a mixture of numbers and spaces;
      - (7) The right side of the highest digit is composed of numbers and other characters;
      - (8) There is a space between the minus sign and the highest digit;
    - The equivalent classes of legal output are
      - (9) The negative integer between the smallest negative integer that the computer can represent and zero;
      - (10) Zero;
      - (11) A positive integer between zero and the largest positive integer that the computer can represent;

Black box testing

- Equivalence division - example
  - The following uses the equivalent division method to design a test plan for a simple program.
    - Analyzing the specifications of this program, the following equivalence classes can be divided:
    - The equivalent classes of illegal output are
      - (12) Negative integers smaller than the smallest negative integer that the computer can represent;
      - (13) A positive integer larger than the largest positive integer that the computer can represent.
    - Because the computer used is 16 bits long and uses twos complement to represent integers, the smallest negative integer that can be represented is -32 768, and the largest positive integer that can be represented is 32 767.

Black box testing

- Boundary value analysis
  - Experience has shown that the program is most prone to errors when dealing with boundary conditions.
    - For example, many program errors occur near the boundaries of subscripts, scalars, data structures, loops, and so on. Therefore, designing a test plan that allows the program to run near the boundary is more likely to expose program errors.
  - Using boundary value analysis method to design a test plan should first determine the boundary conditions, which requires experience and creativity. Usually the boundary of the input equivalence class and the output equivalence class is the program boundary conditions that should be tested.
    - The selected test data should be just equal to, just less than, and just greater than the boundary value.
    - According to the boundary value analysis method, data just equal to, slightly less than and slightly greater than the boundary value of the equivalence class should be selected as the test data, instead of selecting the typical value or any value in each equivalence class as the test data.
  - Usually, when designing a test plan, the two techniques of equivalence division and boundary value analysis are always combined.

Black box testing

- Error Guessing
  - The use of boundary value analysis and equivalence division techniques helps to design a representative test program that easily exposes program errors.
    - However, different types of programs with different characteristics usually have some special error-prone situations.
    - In addition, sometimes the program can work normally when each set of test data is used separately, but the combination of these input data may detect program errors.
  - Generally speaking, even for a relatively small program, the number of possible input combinations is often very large. Therefore, it is necessary to rely on the experience and intuition of the tester to select some programs that are most likely to cause program errors from various possible test programs. . Speculation about what kind of errors may exist in the program is an important factor when choosing a test plan.

Black box testing

- Error Guessing
  - The method of wrong speculation relies to a large extent on intuition and experience.
    - Its basic idea is to enumerate the possible errors in the program and the special situations that are prone to errors, and select the test plan based on them.
  - There are also some experience summed up for the error-prone situations in the program,
    - For example, zero input data or zero output data is often prone to errors; if the number of inputs or outputs allows changes (for example, the number of entries in the retrieved or generated table), the number of inputs or outputs is 0 and 1. Situations (for example, the table is empty or have only one item) are error-prone situations. Should also carefully analyze the program specification, pay attention to find out the missing or omitted parts, in order to design the corresponding test plan, to detect whether the programmer's handling of these parts is correct.

Black box testing

- Error Guessing
  - In addition, experience has shown that the number of errors that have been found in a program is often proportional to the number of errors that have not yet been discovered.
    - For example, in the IBM OS/370 operating system, 47% of all errors found by users are only related to 4% of the system's modules. Therefore, in further testing, we should focus on testing those program segments that have found many errors.
  - Both the equivalence division method and the boundary value analysis method only consider the test efficacy of each input data in isolation, and do not consider the combined effect of multiple input data, which may miss the error-prone combination of input data.
    - An effective way to select input combinations is to use the decision table or decision tree as a tool to list the correspondence between the various combinations of input data and the actions that the program should do (and the corresponding output results), and then for each column of the decision table Design at least one test case.

Black box testing

- Error Guessing
  - Another effective way to select input combinations is to combine computer testing with manual code inspection.
    - For example, through code inspection, it is found that two modules in the program use and modify some shared variables. If one module modifies these variables incorrectly, it will cause an error in the other module. Therefore, this is a possible cause of an error in the program. .
  - A test plan should be designed to detect these two modules at the same time in one run of the program, especially to detect whether the other module can use these variables as expected after one module modifies the shared variables. Conversely, if the two modules are independent of each other, there is no need to test their input combinations. The interdependence of modules can also be found through code inspection.

Debug

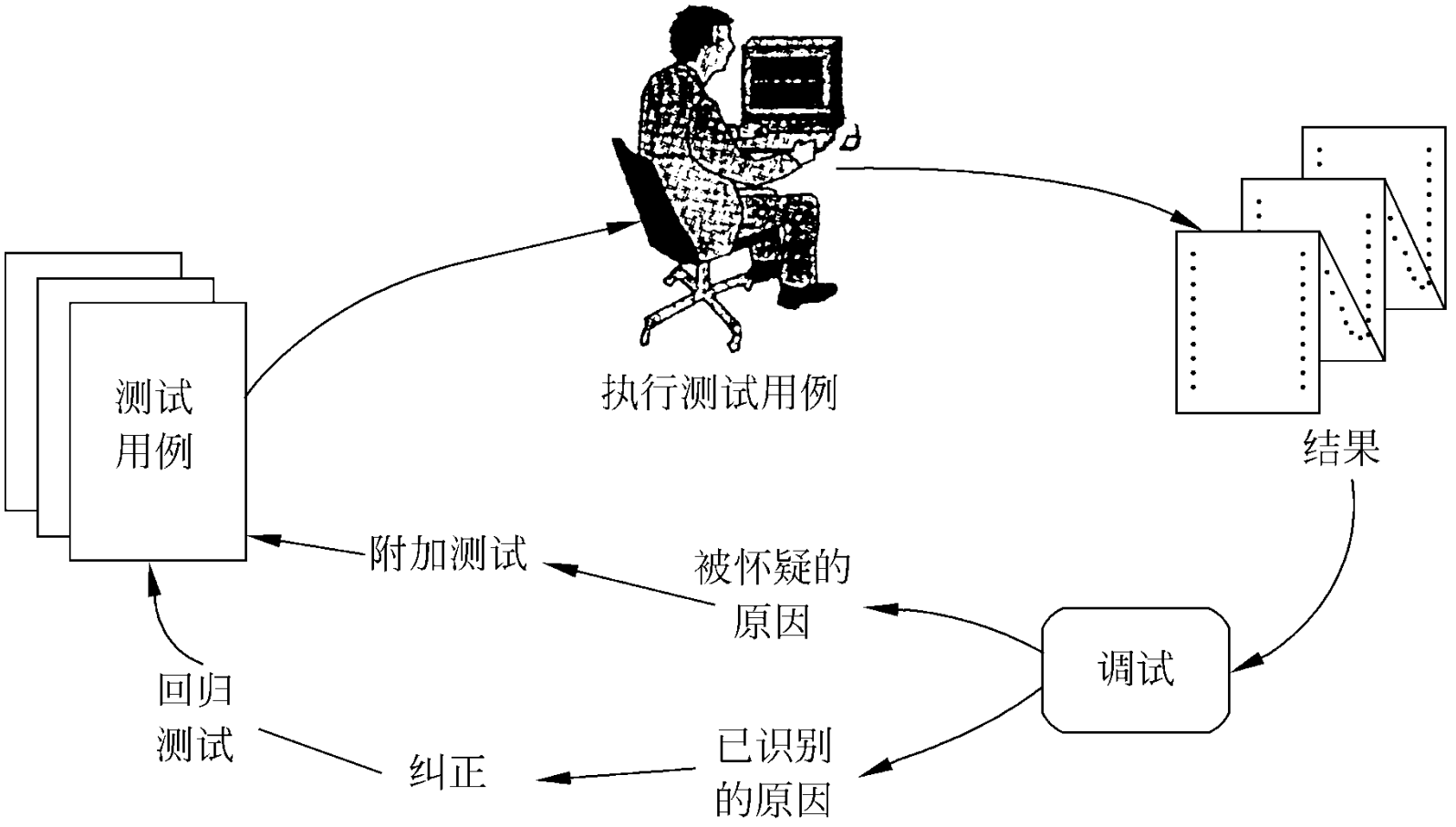
- Debugging (also known as error correction) occurs as a consequence of successful testing, that is, debugging is the process of eliminating errors after the errors are found in the test. Although debugging should and can be an orderly process, it is still largely a skill for now.
- Software engineers often only face the symptoms of software errors when evaluating test results, that is, there may be no obvious connection between the external manifestations of software errors and its internal causes.
- Debugging is an intellectual process that connects symptoms and causes that have not yet been deeply understood.

Debug process

- Debugging is not a test, but it always happens after the test.
- The debugging process starts with the execution of a test case and evaluates the test results. If the actual results are found to be inconsistent with the expected results, this inconsistency is a symptom, which indicates that there are hidden problems in the software.
- The debugging process attempts to find out the cause of the symptoms in order to correct the error.
- The debugging process will always have one of the following two results:
  - ① Find the cause of the problem and correct and eliminate the problem;
  - ② Did not find out the cause of the problem. In the latter case, the debugger can guess a reason, and design test cases to verify this hypothesis, repeat this process until the reason is found and the error is corrected.

Debug process

- Debugging is not a test, but it always happens after the test.
- The debuggi . If the actual results are t , which indicates th
- The debuggi e error.
- The debuggi
  - ① Find
  - ② Did r reason, and design nd the error is correct



Debug process

- Debugging is the most difficult mental work in the software development process. Debugging is so difficult. There may be more psychological reasons than technical reasons. However, the following characteristics of software errors are also very important reasons:
  - (1) The symptom and the cause of the symptom may be far apart in the program
  - Symptoms may appear in one part of the program, and the actual cause may be in another part far away from it. The tightly coupled program structure exacerbates this situation.
  - (2) When another error is corrected, the symptom may temporarily disappear.
  - (3) The symptoms may not actually be caused by errors (for example, rounding errors).
  - (4) Symptoms may be caused by human error that is not easy to track.
  - (5) Symptoms may be caused by timing problems rather than processing problems.
  - (6) It may be difficult to reproduce exactly the same input conditions (for example, a real-time application system with uncertain input sequence).
  - (7) Symptoms may be absent from time to time, which is especially common in embedded systems where hardware and software are tightly coupled.
  - (8) Symptoms may be caused by causes that are distributed in many tasks, and these tasks are running on different processors.
- In the debugging process, you will encounter various errors ranging from small annoying errors (for example, incorrect output format) to catastrophic errors (for example, system failures leading to serious economic losses). The more serious the consequences of an error, the greater the pressure to find the cause of the error. Usually, this pressure causes software developers to introduce two or more errors while correcting one error.

Debugging approach

- No matter what method is used, the goal of debugging is to find the cause of the software error and correct the error. It usually requires a combination of systematic analysis, intuition, and luck to achieve the above goals.
- Generally speaking, there are the following 3 kinds of debugging methods can be used
  - Rashly acting
  - Backtracking
  - Cause elimination

Debugging approach

- 1. Rashly acting
  - It may be the least efficient way to find the cause of the software error.
  - This method should only be used if all other methods have failed. According to the strategy of "Let the computer find errors by itself", this method prints out the contents of the memory, activates the tracking of the running process, and writes WRITE (output) statements everywhere in the program, hoping to be in a certain part of the ocean of information generated in this way. A clue to the cause of the error was found in a place.
  - Although the large amount of information generated may eventually lead to successful debugging, in more cases doing so will only waste time and effort.
  - Before using any debugging method, you must first think carefully, you must have a clear purpose, and the amount of irrelevant information should be minimized.

Debugging approach

- 2. Backtracking
  - Backtracking is a fairly common debugging method, which is effective when debugging small programs.
  - The specific method is to start from the place where the symptoms are found, and manually trace and analyze the source code back and forth along the control flow of the program until the cause of the error is found.
  - However, as the scale of the program expands, the number of paths that should be backtracked has become larger and larger, so that a complete backtracking becomes completely impossible.

Debugging approach

- 3. Cause elimination method
  - The bisection search method, induction method and deduction method all belong to the cause elimination method.
  - The basic idea of the binary search method is that if you already know the correct value of each variable at several key points in the program, you can use an assignment statement or an input statement to "inject" the correct value of these variables near the midpoint of the program, and then run Program and check the output obtained.
    - If the output result is correct, the cause of the error is in the first half of the program;
    - On the contrary, the cause of the error is in the latter half of the program. Repeat this method for the part where the error is located, until the scope of the error is reduced to the extent that it is easy to diagnose.
  - Induction is a way of thinking that infers general conclusions from individual phenomena.
    - When using this method to debug a program, first organize and analyze the data related to the error in order to find the possible cause of the error. Then derive one or more assumptions about the cause of the error, and use existing data to prove or eliminate these assumptions. Of course, if the existing data is not enough to prove or rule out these assumptions, you need to design and execute some new test cases to obtain more data.
  - The deductive method starts from general principles or premises, and derives conclusions through the process of elimination and refinement.
    - When using this method to debug a program, first imagine all possible causes of error, and then try to use tests to rule out each hypothetical cause. If the test shows that a hypothetical cause may be the true cause, then refine the data to accurately locate the error.

Debugging approach

- If you have used various debugging methods and tools but still cannot find the cause of the error, you should ask your colleagues for help. Presenting the problems encountered to colleagues and analyzing and discussing together can often broaden your thinking and find out the cause of the error faster.
  - Once an error is found, it must be corrected. However, correcting one error may introduce more other errors, even "the gain is not worth the loss." Therefore, software engineers should carefully consider the following three questions before correcting errors by hand:
    - (1) Does the same error exist elsewhere in the program?
      - In many cases, a program error is caused by a wrong logical thinking mode, and this logical thinking mode may also be used in other places. A careful analysis of this logic model may reveal other errors.
    - (2) What is the "next error" that the upcoming modification may introduce?
      - Before correcting errors, you should carefully study the source program (and preferably study the design document) to evaluate the degree of coupling between the logic and the data structure. If the modification to be made is in the highly coupled section of the program, special care must be taken when modifying it.
    - (3) What should be done to prevent similar errors in the future?
      - If you not only modify the software product but also improve the software process of developing the software product, you will not only eliminate errors in the existing program, but also avoid errors that may occur in the program in the future.

Software reliability

- Definition of software reliability
  - There are many different definitions of software reliability, one of which most people recognize is: software reliability is the probability of a program running successfully in accordance with the specifications in a given time interval.
  - As the running time increases, the probability of program failure during operation will also increase, that is, the reliability will decrease with the increase of the given time interval.
  - According to IEEE regulations, the term "error" means a software error (bug) caused by a developer, and the term "fault" means an incorrect behavior of the software caused by an error.

Software reliability

- Software availability
  - Usually users are also very concerned about the extent to which the software system can be used. Generally speaking, for any system whose failure can be repaired, reliability and availability should be used to measure its pros and cons at the same time.
  - One definition of software usability is: Software usability is the probability of a program running successfully at a given point in time, in accordance with the specifications of the specification.

Software reliability

- The main difference between reliability and availability is
  - Reliability means that the system did not fail during the time interval from 0 to t, while availability only means that the system is operating normally at time t.
    - Therefore, if the system is available at time t, there are the following possibilities: the system has not failed (reliable) during the period from 0 to t; it has failed once during this period, but is repaired; In this period of time, it failed twice and repaired twice;...
- If within a period of time, the downtime of software system failure is td1, td2,..., and the normal running time is tu1, tu2,..., then the steady-state availability of the system is:
  - Ass=Tup/(Tup+Tdown) (7.1)
    - Where Tup=∑tui, Tdown=∑tdi
- If the concepts of system mean time between failures MTTF and mean time to repair MTTR are introduced, equation (7.1) can be changed to
  - Ass=MTTF/(MTTF+MTTR) (7.2)
- Mean time to repair MTTR is the average time it takes to repair a fault. It depends on the technical level of the maintenance personnel and the familiarity of the system. It also has an important relationship with the maintainability of the system.
- The mean time between failures MTTF is the average time for the system to run successfully according to the specifications. It mainly depends on the number of latent errors in the system, so it has a very close relationship with the test.

Method of estimating mean time between failures

- The MTTF of the software is an important quality index, which is often put forward by the user as a requirement for the software. In order to estimate MTTF, first introduce some relevant quantities.
  - 1. Symbols
    - In the process of estimating MTTF, the following symbols are used to indicate the relevant quantities:
      - ET——the total number of errors in the program before the test;
      - IT——program length (total number of machine instructions);
      - τ——Test (including debugging) time;
      - Ed(τ)——The number of errors found between 0 and τ;
      - Ec(τ)——The number of errors corrected from 0 to τ.

Method of estimating mean time between failures

- The MTTF of the software is an important quality index, which is often put forward by the user as a requirement for the software. In order to estimate MTTF, first introduce some relevant quantities.
  - 2. Basic assumptions
    - Based on empirical data, the following assumptions can be made.
      - (1) In a similar program, the number of errors per unit length ET/IT is approximately constant.
        - Some statistics in the United States indicate that usually 0.5×10-2≤ET/IT≤2×10-2 that is to say, there are about 5-20 errors in every 1000 instructions before the test.
      - (2) The failure rate is proportional to the number of remaining (latent) errors in the software, and the mean time between failures MTTF is inversely proportional to the number of remaining errors.
      - (3) In addition, in order to simplify the discussion, it is assumed that every error found is corrected immediately and correctly (that is, no new errors are introduced during the debugging process). therefore
        - Ec(τ)=Ed(τ)
        - The number of errors remaining is
        - Er(τ)=ET-Ec(τ) (7.3)
        - The number of errors remaining in the unit length program is
        - εr(τ)=ET/Ir-Ec(τ)/IT (7.4)

Method of estimating mean time between failures

- The MTTF of the software is an important quality index, which is often put forward by the user as a requirement for the software. In order to estimate MTTF, first introduce some relevant quantities.
  - 3. Estimate the mean time between failures
    - Experience has shown that the mean time between failures is inversely proportional to the number of errors remaining in the program per unit length, namely
      - MTTF=1/[K(ET/IT-Ec(τ)/IT)] (7.5)
        - Among them, K is a constant, and its value should be selected based on experience. Some statistics in the United States indicate that the typical value of K is 200.
    - The formula for estimating the mean time between failures can evaluate the progress of software testing. In addition, from (7.5), we can get
      - Ec=ET-IT/(K×MTTF) (7.6)
    - Therefore, it is also possible to estimate how many errors need to be corrected before the test work can be completed according to the requirements for the mean time between failures of the software.

Method of estimating mean time between failures

- The MTTF of the software is an important quality index, which is often put forward by the user as a requirement for the software. In order to estimate MTTF, first introduce some relevant quantities.
  - 4. Method of estimating the total number of errors
    - The number of hidden errors in the program is a very important quantity, which not only directly marks the reliability of the software, but also an important parameter for calculating the software's mean time between failures. Obviously, the total number of errors in the program ET is closely related to the program scale, type, development environment, development methodology, technical level and management level of the developers. Two methods for estimating ET are described below.
    - (1) Implant error method
      - Using this estimation method, some errors are randomly implanted in the program by a dedicated person before the test. After the test, the original error in the program is estimated based on the ratio of the original and the implanted errors found by the test team. Total ET.
      - Assuming that the number of artificially implanted errors is Ns, after a period of testing, ns implanted errors are found, in addition to n original errors. If it can be considered that the test program has the same ability to find implant errors and original errors, then the total number of original errors in the program can be estimated as
        - $\hat{N}=n/ns \times Ns$ (7.7)
        - Where $\hat{N}$ is the estimated value of the total number of errors ET.

Method of estimating mean time between failures

- The MTTF of the software is an important quality index, which is often put forward by the user as a requirement for the software. In order to estimate MTTF, first introduce some relevant quantities.
  - 4. Method of estimating the total number of errors
    - (2) Separate test method
      - The basic assumption of the implant error method is that the test scheme used has the same probability of discovering the implant error and the original error. However, the nature of artificially implanted errors and the original errors in the program may be very different, and the difficulty of finding them is naturally different. Therefore, the above basic assumptions may sometimes not be completely consistent with the facts.
      - If there is a way to randomly mark some of the original errors in the program, and then estimate the total number of errors in the program based on the ratio of the marked errors to the unmarked errors found during the test, then the result obtained in this way is better than the use of planting. The results obtained by entering the error method are more credible.
      - In order to mark a part of the errors randomly, the separate test method uses two testers (or test groups) to test two copies of the same program independently of each other, and consider the errors found by one of the testers as marked errors. The specific method is that in the early stage of the testing process, tester A and tester B test two copies of the same program separately, and another analyst analyzes their test results. Use τ to represent the test time, assuming
        - When τ=0, the total number of errors is B0;
        - When τ=τ1, the number of errors found by Tester A is B1;
        - When τ=τ1, the number of errors found by tester B is B2;
        - When τ=τ1, the same number of errors found by the two testers is bc.

Method of estimating mean time between failures

- If it is considered that the errors found by tester A are marked, that is, the total number of marked errors in the program is B1, then bc out of B2 errors found by tester B are marked. Assuming that tester B has the same probability of finding marked errors and unmarked errors, the total number of errors in the program before the test can be estimated as
  - B0^ =B2/bcB1 (7.8)
- Using the separate test method, in the early stage of the test, the analyst analyzes the test results of two testers at regular intervals, and calculates B0^ using equation (7.8). If the results of several estimates are similar, the average value of B0^ can be used as the estimated value of ET. After that, a tester can change to other tasks, and the remaining tester can continue to complete the test work, because he can inherit the test results of another tester, so the test cost of the separate test method is not too much.