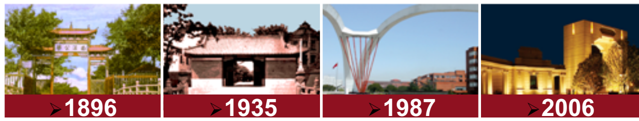
上海交通大學
SHANGHAI JIAO TONG UNIVERSITY



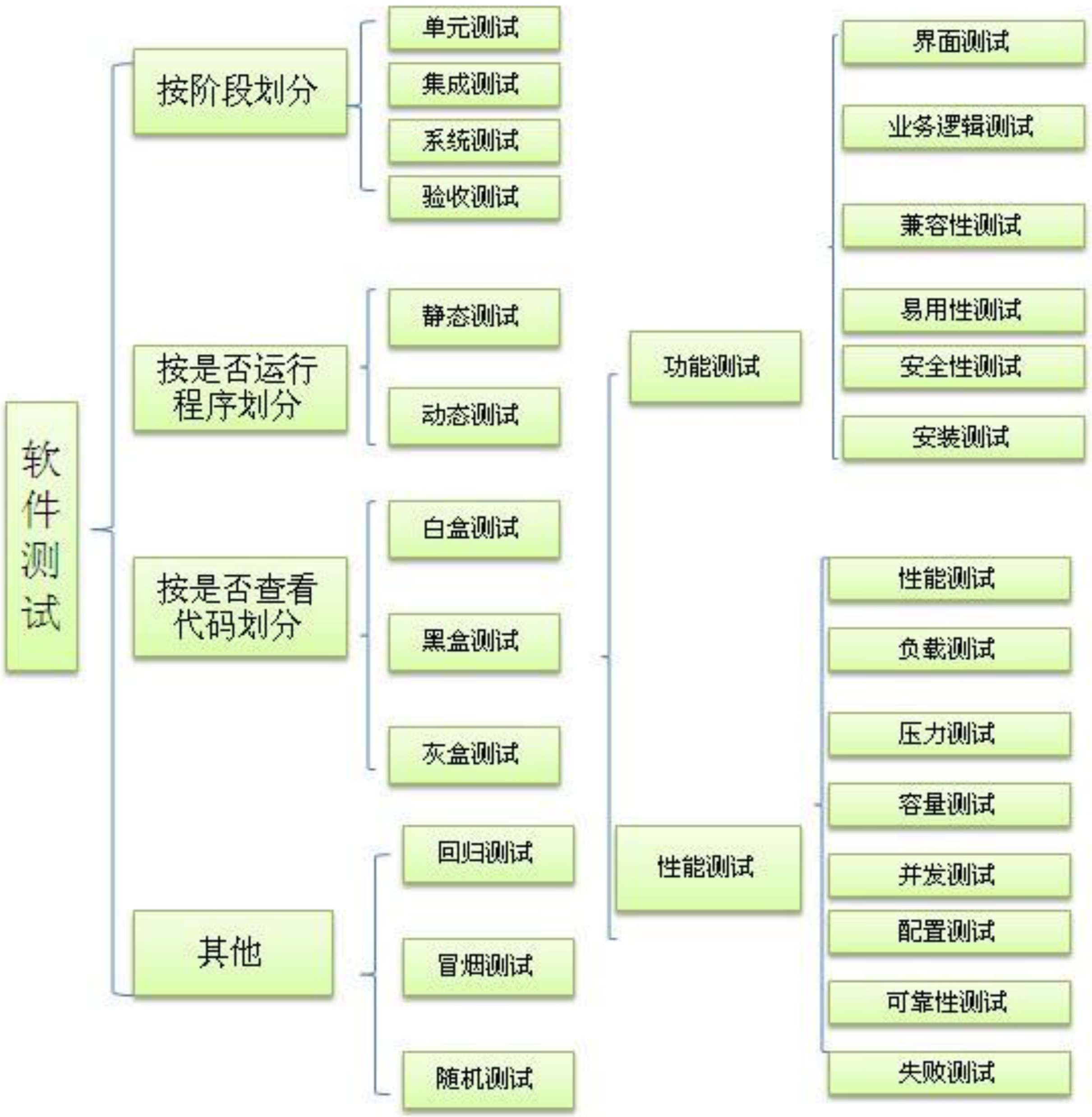
Chapter 3

Fundamental Theory and Methods

思考时间

结合自己的编程与软件开发经历，如何提高代码的质量？

测试方法概览


软件测试
按阶段划分
单元测试
集成测试
系统测试
验收测试
按是否运行程序划分
静态测试
动态测试
按是否查看代码划分
白盒测试
黑盒测试
灰盒测试
其他
回归测试
冒烟测试
随机测试
功能测试
界面测试
业务逻辑测试
兼容性测试
易用性测试
安全性测试
安装测试
性能测试
性能测试
负载测试
压力测试
容量测试
并发测试
配置测试
可靠性测试
失败测试

Outline

3.1 Non-execution based Verification

- Not execute the program, tester could use some tools to review and analyze the specification and program code.
    - Walk through（走查）
    - Inspection（审查）

3.1.1 Walk through

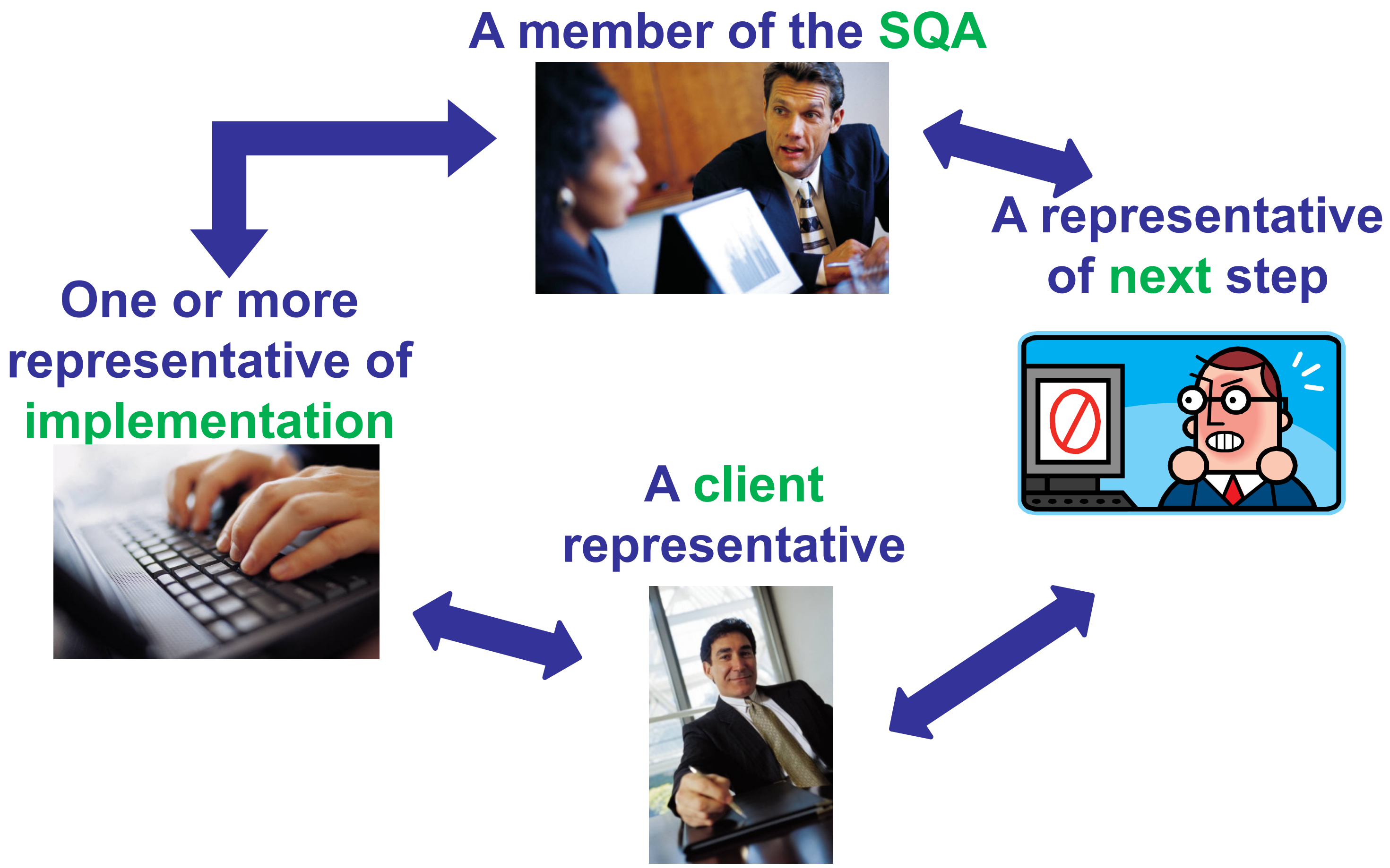
— Consist a walk through test group

— Check program code logic

— Generate test case, include input data and expect result.

— Put the data into program code, if the calculate result unequal the expect result, find an error.

— Check the code line by line.

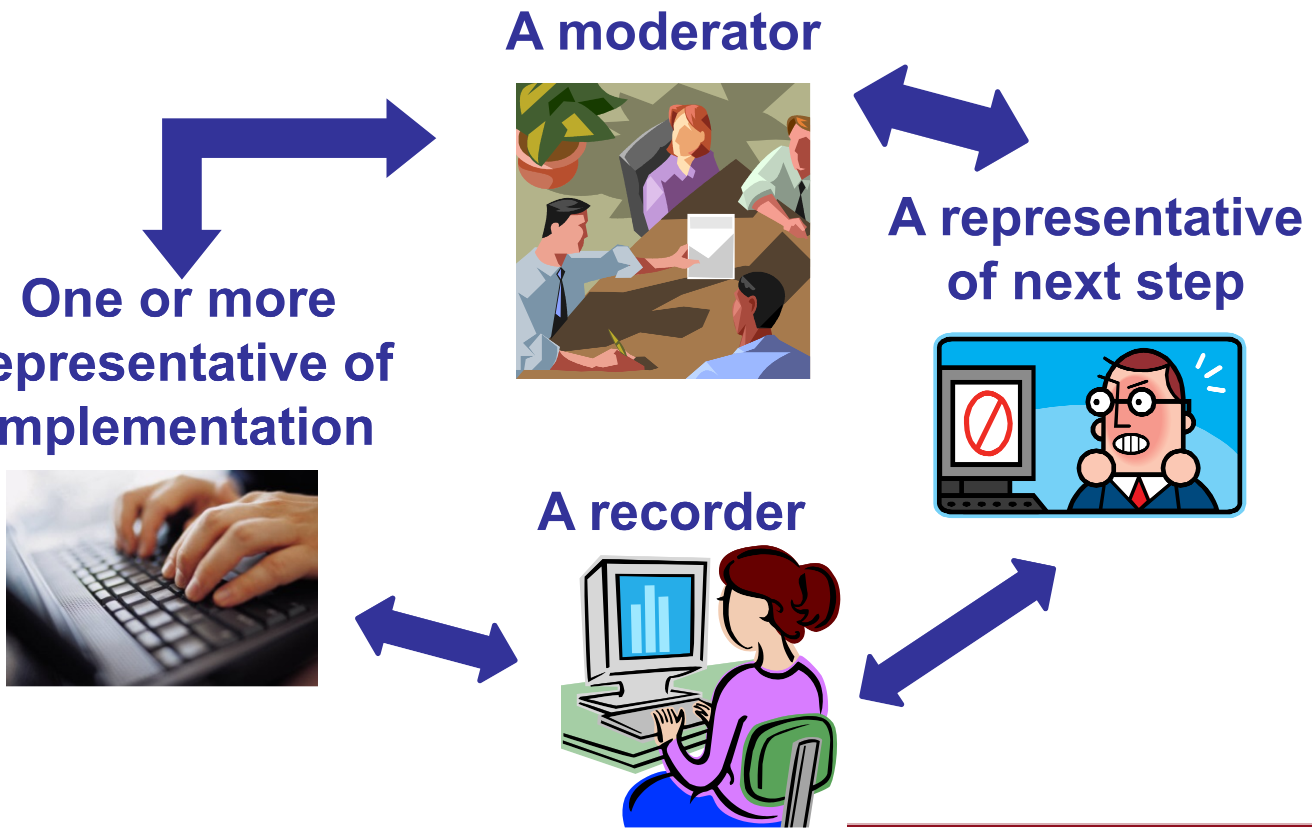

A member of the SQA
A representative of next step
One or more representative of implementation
A client representative

The walkthrough team should consist of four to six individuals

- The goal of the walkthrough team is to detect faults, not to correct them.
- The person leading the walkthrough guides the other members of the team through the code.
- The walkthrough can be driven by the lists of issues compiled by team members or by the code itself, with team members raising their concerns at the appropriate time.
- In both cases, each issue will be discussed as it comes up and resolved into either a fault that needs to be addressed or a point of confusion that will be cleared up in the discussion

3.1.2 Inspection

- An **inspection** is a far more **formal** activity than a code walkthrough.
- It is conducted by a team of **four to six** people.


A moderator
A representative of next step
One or more representative of implementation
A recorder

The review process consists of five formal steps:

- In the overview step, the author of the module gives a presentation to the team.
- In the preparation step, the participants try to understand the code in detail and compile lists of issues, ranked in order of severity(严重性). They are aided in this process by a checklist of potential faults for which to be on the lookout.
- In the inspection step, a thorough walkthrough of the code is performed, aiming for fault detection through complete coverage of the code. Within a day of the inspection, the moderator produces a meticulous written report.
- In the rework step, the individual responsible for the code resolves all faults and issues noted in the written report. （**Individual Reviewer Issues Spreadsheet**）
- In the follow-up step, the moderator must make sure that each issue has been resolved by either fixing the code or clarifying confusing points. （**Review Report Spreadsheet**）

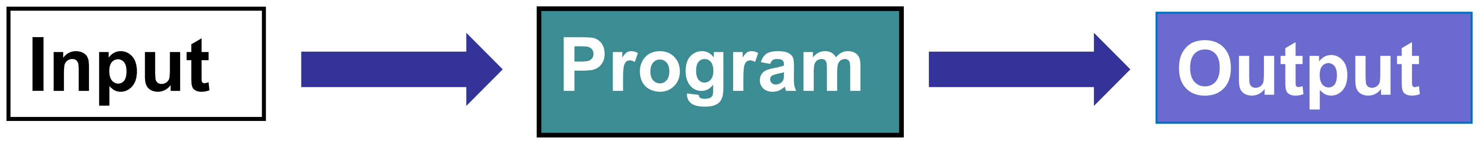
- An important product of an inspection is the number and kinds of faults found rated by severity.
- If a module comes through an inspection exhibiting a significantly larger number of faults than other modules in the system, it is a good candidate for rewriting.
- If the inspection of two or three modules reveals a large number of errors of specific types, this may warrant (re)checking other modules for similar errors.
- If more than 5 percent of the material inspected must be reworked, the team must reconvene for a full re-inspection.

3.2 Execution-based Verification

- Use test cases to execute the program, get all results from the execution.

- There are two basic approaches to testing modules, each with its own weaknesses.
  - Testing to Specifications, also known as black-box test.
  - Testing to Code, also called glass-box, white-box test.

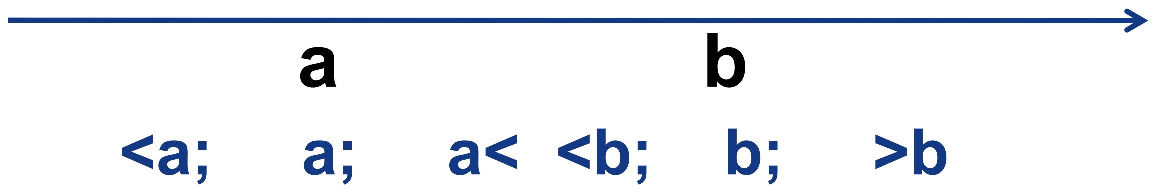
3.2.1 Black-box test



- Seeks to verify that the module conforms to the specified input and output while ignoring the actual code
- It is rarely possible to test all modules for all possible input cases, which may be a huge number of modules

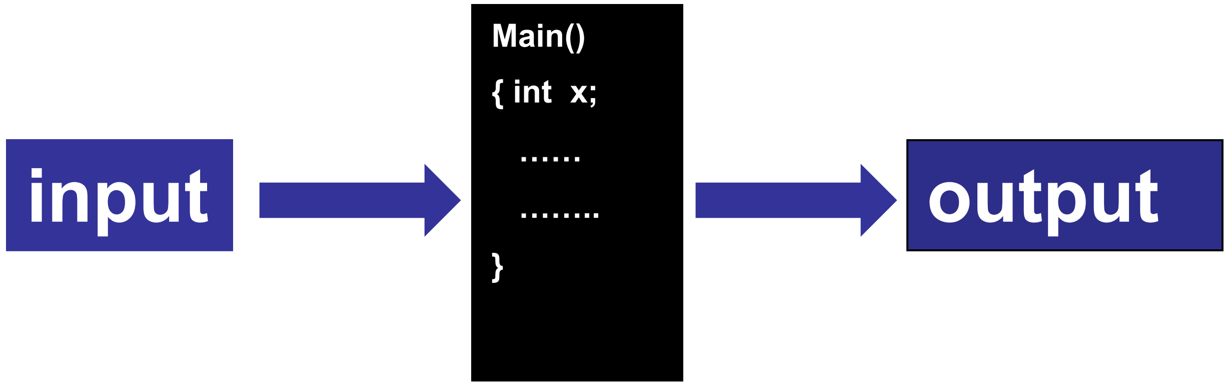
- Equivalence testing is a technique based on the idea that the input specifications give ranges of values for which the software product should work the same way.
- Boundary value analysis seeks to test the product with input values that lie on and just to the side of boundaries between equivalence classes.

For example the input is (a,b), there are at least 5 equivalence classes:



Functional testing, the tester identifies each item of functionality or each function implemented in the module and uses data to test each function.

3.2.2 White-box test



- Also called logic-driven, or path-oriented testing.
- Testing each path through the code is generally not feasible, even for simple flowcharts.
- Moreover, it is possible to test every path without finding existing faults, because, for example, the fault lies in the decision criterion for selecting between paths.

- Statement coverage **and amounts to running tests in which every statement in the code is executed at least once.**
- Branch coverage**, makes sure that each branching point is tested at least once.**
- Path coverage, **make sure that every different path is tested at least once.**

Example : black-box testing

- Discussion: MAC/ATM machine
- Specs
  - Cannot withdraw more than $300
  - Cannot withdraw more than your account balance

Test case?

X

1. >300
2. 0~300
3. <0

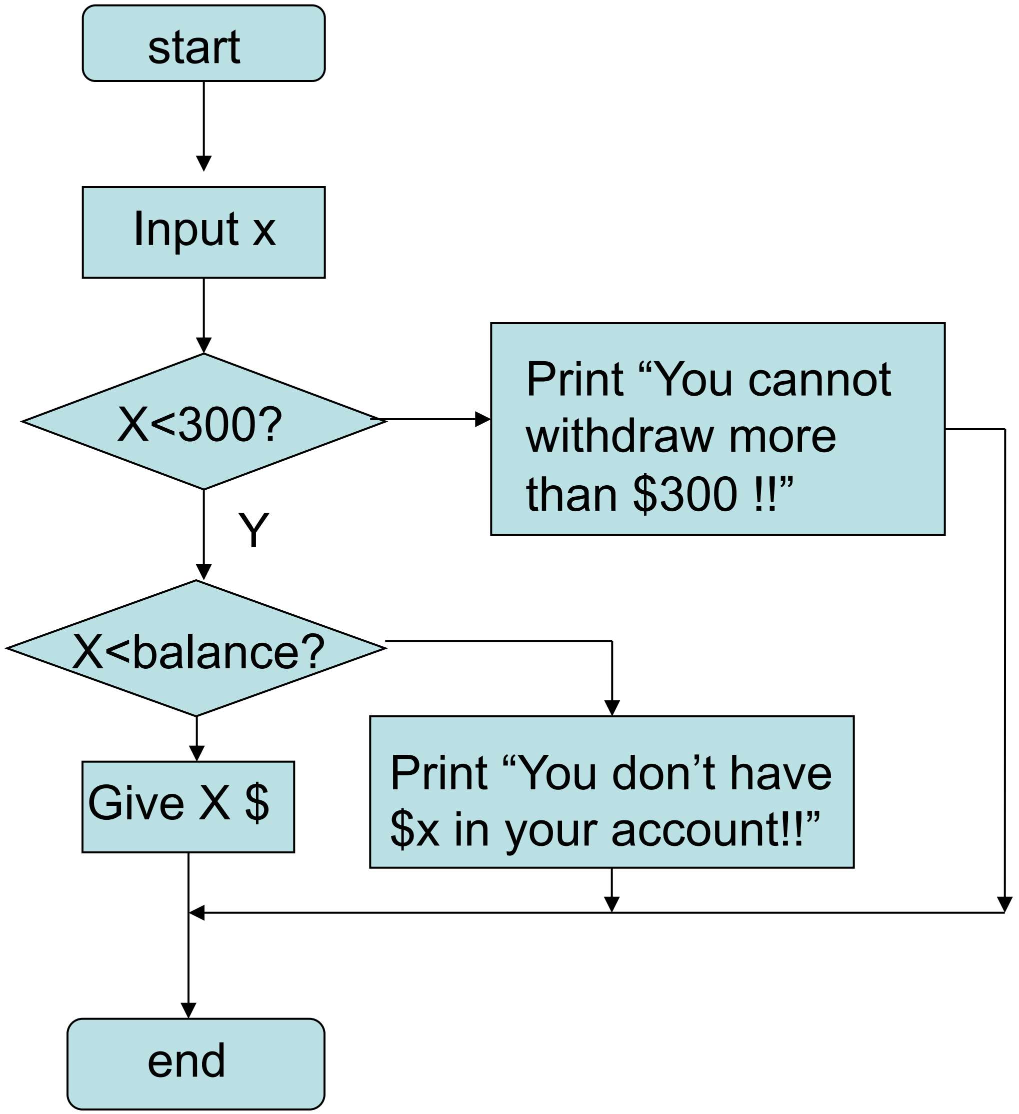
Balance：

1. = >300,
2. 0~300

White-box Testing

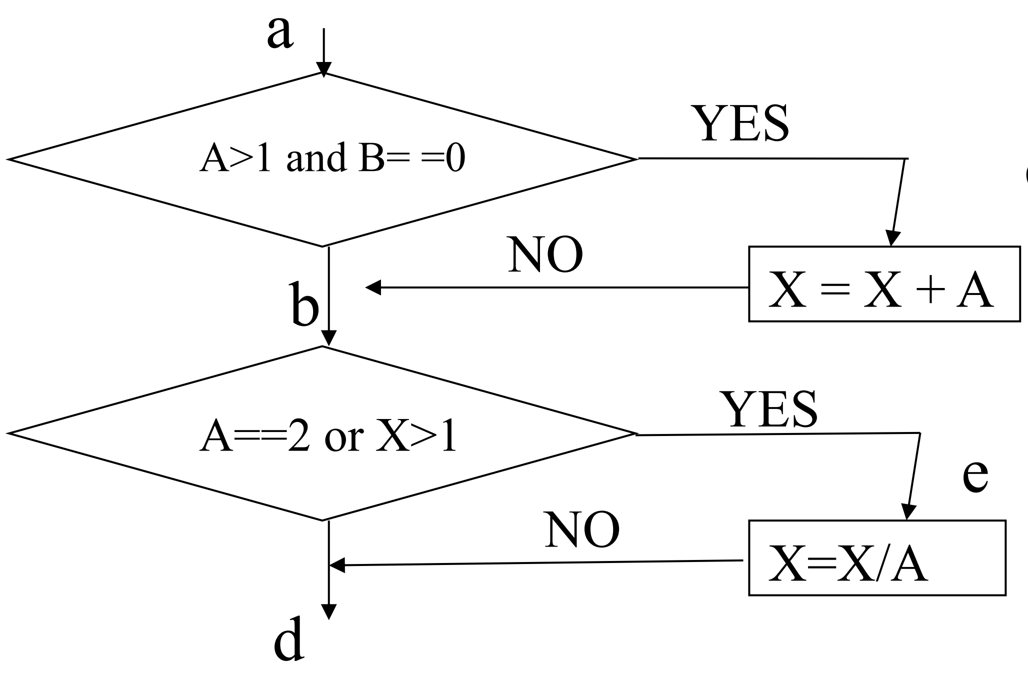
Example

```
// x: 1..1000;
1   INPUT-FROM-USER(x);
      If (x <= 300) {
2         INPUT-FROM-FILE(BALANCE);
            If (x <= BALANCE)
3                 GiveMoney x;
4           else Print "You don't have $x in your account!!"}
       else
5      Print "You cannot withdraw more than $300";
6    Eject Card;
```


start
Input x
X<300?
Print “You cannot withdraw more than $300 !!”
Y
X<balance?
Give X $
Print “You don’t have $x in your account!!”
end

Test case?

How about example 2 ?



- For example 2, branch coverage is: point A, should have $A>1$, $A\leq 1$, $B=0$, $B\neq 0$, point B, should have $A=2$, $A\neq 2$, $X>1$, $X\leq 1$.

  so {2, 0|2, 5} and {0, 1|0, 0} is enough.

- How about path coverage?

  4

- Predicates Test case are: $2\times2\times2\times2=16$

3.3 Formal verification

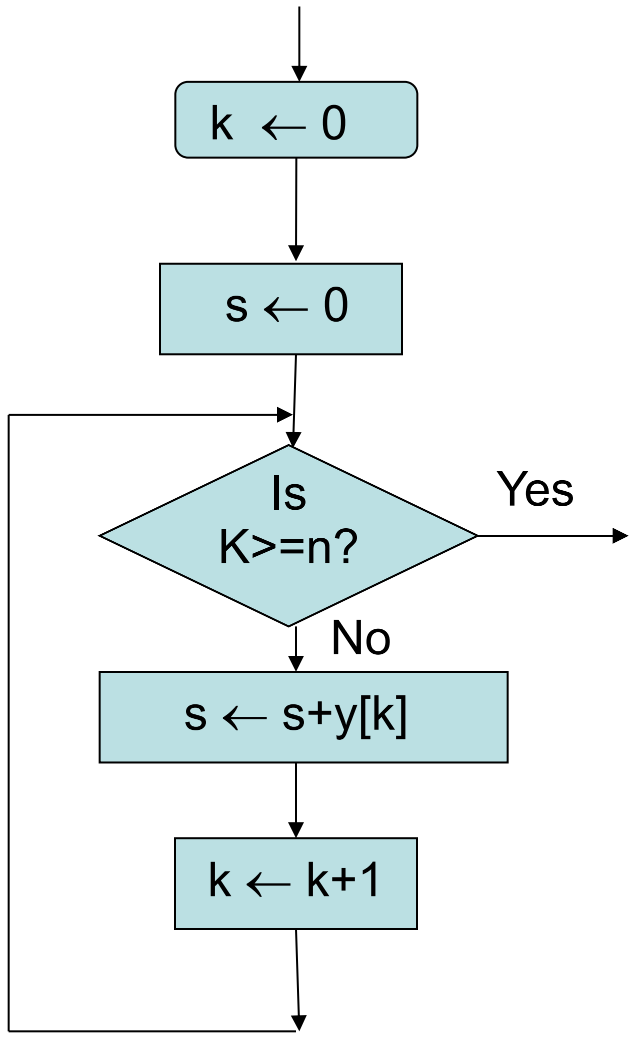
- Correctness proofs

  It is a mathematical technique for show that a product is correct, in other word, that it satisfies its specifications.

- To see how correctness is proven, consider the fragment (next slide)
- We now show that the code fragment is correct- after execute, the variable s will contain the sum of the n elements of array y.

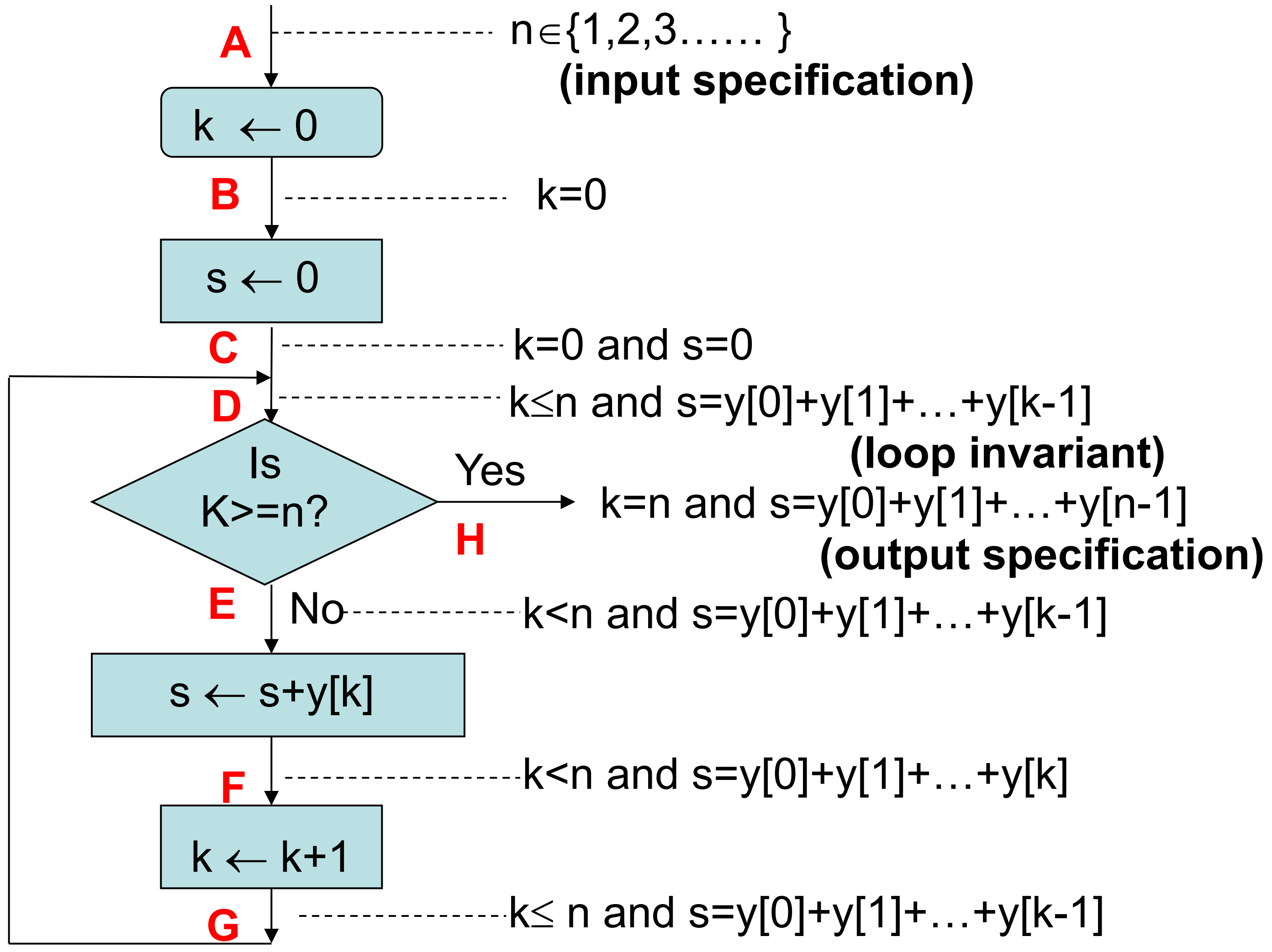
```
int k,s;
int y[n];
……………….
k=0;
s=0;
while (k<n)
{
    s=s+y[k];
    k=k+1;
  }
```

Code fragment



Flowchart

An assertion is placed before and after each statement, at the place labeled with the letters A through H;



A
n∈{1,2,3…… }
(input specification)
k ← 0
B
k=0
s ← 0
C
k=0 and s=0
D
k≤n and s=y[0]+y[1]+…+y[k-1]
(loop invariant)
Is K>=n?
Yes
H
k=n and s=y[0]+y[1]+…+y[n-1]
(output specification)
E
No
k<n and s=y[0]+y[1]+…+y[k-1]
s ← s+y[k]
F
k<n and s=y[0]+y[1]+…+y[k]
k ← k+1
G
k≤ n and s=y[0]+y[1]+…+y[k-1]

Input specification, n is a positive integer,

A: $n \in \{1,2,3\ldots\ldots\}$ (1)

Output specification is that if control reaches point H, the value of ***s*** contains the sum of ***n*** values stored in array ***y***,

H: $s= y[0]+y[1]+\ldots+y[n-1]$ (2)

A stronger output specification is,

H: $k=n$ and $s= y[0]+y[1]+\ldots+y[n-1]$ (3)

How proven from A to H? key is proven the invariant of the loop, that is,

D: $k \leq n$ and $s=y[0]+y[1]+\ldots+y[k-1]$ (4)

First B: $k=0$ (5)

This is easy to prove, because of assignment statement

$k \leftarrow 0$ is executed.

At point C, the second assignment statement $s \leftarrow 0$ is executed, the follow assertion is true:

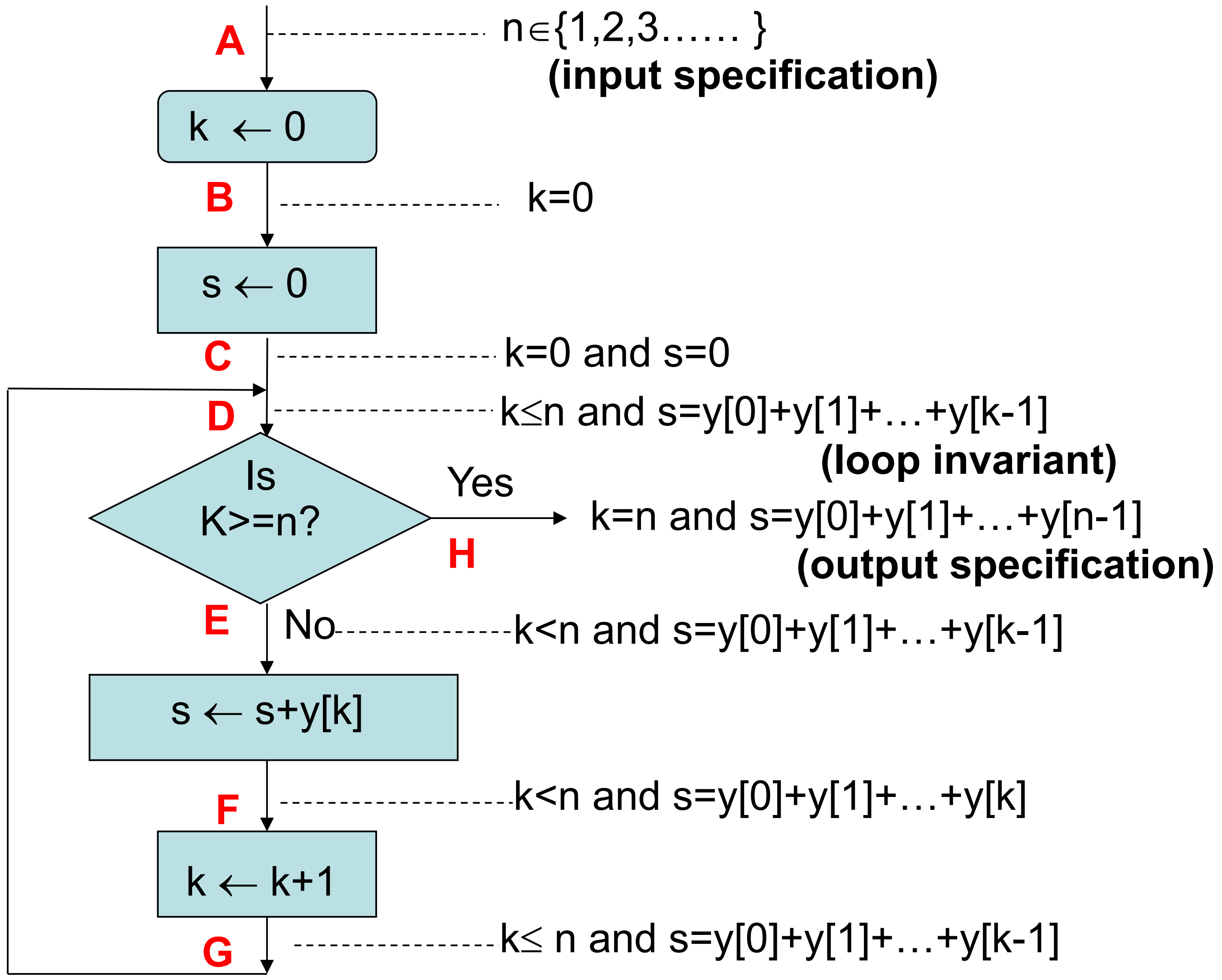
$$k=0 \text{ and } s=0 \qquad (6)$$

Now the loop is entered. Before the loop execute, assertion(6) holds $k=0$ and $s=0$. By assertion(1), $n \geq 1$. so it follows that $k \leq n$ is required. Because $k=0$, it follows $k-1=-1$, so the sum in (4) is empty $s=0$. there for loop invariant (4) is true just before the first loop is entered.

Now, assume that, at some stage during the execution of the code fragment, the loop invariant holds. That is k to some value k_0, $0 \leq k_0 \leq n$,

$$D: \quad k_0 \leq n \text{ and } s=y[0]+y[1]+\ldots+y[k_0-1] \qquad (7)$$

We should to prove execute the code fragment, the loop invariant is still true or get the output specification.


A
n∈{1,2,3…… }
(input specification)
k ← 0
B
k=0
s ← 0
C
k=0 and s=0
D
k≤n and s=y[0]+y[1]+…+y[k-1]
(loop invariant)
Is K>=n?
Yes
H
k=n and s=y[0]+y[1]+…+y[n-1]
(output specification)
E
No
k<n and s=y[0]+y[1]+…+y[k-1]
s ← s+y[k]
F
k<n and s=y[0]+y[1]+…+y[k]
k ← k+1
G
k≤ n and s=y[0]+y[1]+…+y[k-1]

Pass to the test box. If $k_0 \geq n$, then because $k_0 \leq n$ by hypothesis, it follows that $k_0=n$, this implies that

$$H: K_0=n \text{ and } s=y[0]+y[1]+\ldots+y[n-1] \qquad (8)$$

Just the output specification(3).

On the other hand, $k_0<n$, and (7), becomes

$$E: \quad k_0<n \text{ and } s=y[0]+y[1]+\ldots+y[k_0-1] \qquad (9)$$

Just the assertion of E, then the statement $s \leftarrow s+y[k_0]$ is executed, the point F is

$$F: \quad k_0<n \text{ and } s= y[0]+y[1]+\ldots+y[k_0-1]+y[k_0]$$
$$= y[0]+y[1]+\ldots+y[k_0] \qquad (10)$$

The next statement to be executed is $k_0 \leftarrow k_0+1$. then because before increasing $k_0<n$, now $k_0 \leq n$. and (10) becomes

$$G: \quad k_0 \leq n \text{ and } s= y[0]+y[1]+\ldots+y[k_0-1] \qquad (11)$$

Assertion (11) is equal to assertion (7). That by assumption, holds at point D.

Now we have proved that 0≤k≤ n, the invariant (4) is true .

Last prove is the terminates of the loop. Since the initial k is 0, and each iteration the loop increases the k by 1, surely k will reaches n, k=n. Then by assertion (8), got the output specification (3).

From the example, we could know, that only a 5 lines simple codes, the correctness proofs need 5 pages!

So for a more large program code, it is impossible to use correctness proofs for all codes. Even it is the most reliable method.

3.4 Other methods

- Def-use test
- Mutation test
- Regression Test
- Fault Statistics and Reliability Analysis
- Clean room

3.4.1 Def-Use test

Data-flow based adequacy criteria

- **All definitions criterion**
  - **Each definition to some reachable use**
- **All uses criterion**
  - **Definition to each reachable use**
- **All def-use criterion**
  - **Each definition to each reachable use**

3.4.2 Mutation test

- **Error seeding**
  - Introducing artificial faults to estimate the actual number of faults
- **Program mutation testing**
  - Distinguishing between original and mutants
- **Competent programmer assumption**
  - Mutants are close to the program
- **Coupling effect assumption**
  - Simple and complex errors are coupled

3.4.3 Regression Testing

- **Developed first version of software**
- **Adequately tested the first version**
- **Modified the software; Version 2 now needs to be tested**
- **How to test version 2?**
- **Approaches**
  - Retest entire software from scratch
  - Only test the changed parts, ignoring unchanged parts since they have already been tested
  - Could modifications have adversely affected unchanged parts of the software?

- Software maintenance task performed on a modified program to instill confidence that changes are correct and have not adversely affected unchanged portions of the program.”

3.4.4 Fault Statistics

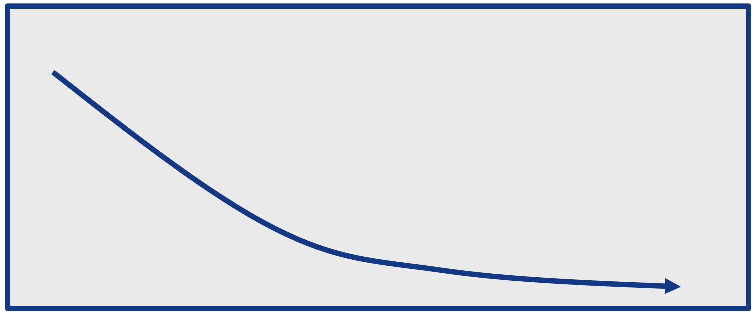
- Fault statistics provide a useful metric for determining whether to continue testing a module of product or whether to recode it.
- The number of faults detected via execution- and non-execution-based techniques must be recorded.
- Data on different types of faults found via code inspections (faults such as misunderstanding the design, initializing improperly, and using variables inconsistently) can be incorporated into checklists for use during later reviews of the same product and future products.

Fishing Creel Counts and Fault Insertion

- For example, that the Rogue River is stocked with 1000 marked trout(鳟鱼)；and that during the fishing season, the creel report totals are 500 fish caught, of which 300 are marked trout.
- The stream management team would conclude that 60% of the trout in the Rogue River are hatchery (planted) fish. The total populations could also be estimated;
- In this case, the 500 fish caught represent 30% of the total population. Total fish population is 1667.
- The same idea can be applied to estimate the success of test cases and the number of remaining faults not caught by an existing set of test cases.

3.4.5 Reliability Analysis

- **Reliability analysis** uses statistical-based techniques to provide estimates of how many faults are remaining and how much longer it is desirable to keep testing.
- It can also be applied in both implementation and integration phases.



3.4.6 Clean-room

- The *Clean-room technique* is a combination of several different software development techniques.
- Under this technique, a module isn’t compiled until it has passed an inspection, or another kind of non-execution-based review, such as a code walkthrough or inspection.
- The relevant metric is testing fault rate, which is the total number of faults detected per KLOC (thousand lines of code).

- The Cleanroom technique has had considerable success in finding and weeding out faults before execution-based testing.
- In one case, all faults were found via correctness-proving techniques, using largely informal proofs, but with a few formal proofs as well. The resulting code was found to be free of errors both at compilation and at execution time.
- In a study of 17 other Cleanroom software products, the products did not perform quite as faultlessly, but they still achieved remarkably low fault rates, an average of 2.3 faults per KLOC.

Summary

- Non-execution based Verification
  - Walkthrough
  - Inspection
- Execution based Verification
  - Black-box
  - White-box
- Formal verification
- Other methods
  - Def-use test
  - Mutation test
  - Regression Test
  - Fault Statistics and Reliability Analysis
  - Clean room