Various objectives for testing

- Different objectives demand different perspectives
- Is the objective to:
  - **Find as many errors as possible?** If so, develop a strategy aimed at revealing errors
  - **Increase our confidence in proper functioning of software?** If so, different strategy will result
- Techniques classified via **criteria for test adequacy**
  - **coverage**-based testing
  - **fault**-based testing
  - **error**-based testing
- Techniques classified by **source of information for test**
  - **black-box testing** (functional, specification-based)
  - **white-box testing** (structural, program-based)

But first, some definitions…

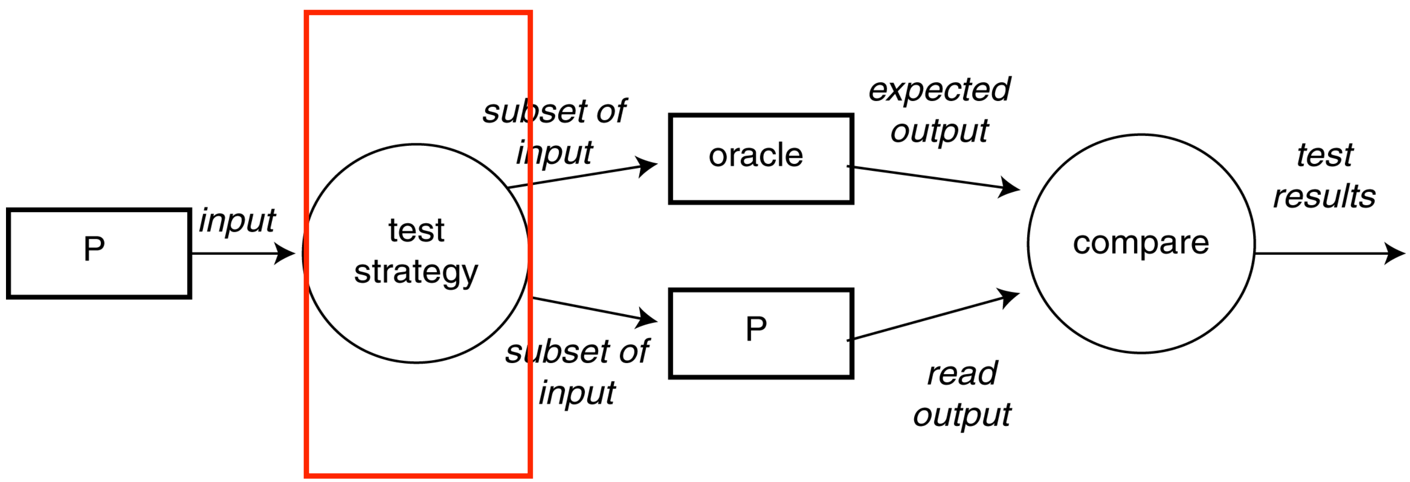
- Must distinguish amongst these terms:
  - **error**: a human action that produces an incorrect result
  - **fault**: a manifestation of an error (i.e., software contains fault)
  - **failure**: inability of system to produce correct result
- **Failure** may be caused by more than one fault.
- **Fault** may cause different failures.
- One possible aim of testing: **find faults**
  - Use specification as reference point
  - “Failure” occurs when software doesn’t meet specification
  - Yet what of users’ **expectations** of system behavior (i.e., "blame-the-silly-user" disease caught by some programmers)?

More terms

- We distinguish between two perspectives of system:
  - **Verification**: evaluating system / component against “success conditions” stated at start of development phase
  - **Validation**: evaluating system / component at end of development process to determine if it satisfies requirements
- Verification: “Have we built the **system right**?”
- Validation: “Have we built the **right system**?”

But how do we reveal faults?

- Faults may exist even if system does not fail!
- (That is, failure may only occur sometime in the future as environment changes.)
- Need to match against programmer's real intentions.
- Solution: use an **oracle** to generate tests



Test-adequacy criteria: definitions?

- "If we execute a program using test set S, and if 100% of all statements are executed, then S is adequate."
- Yet this is only one way of stating what is "adequate" testing
  - For example, we may change the percentage coverage
  - Or we may change the number of test sets
- **Test adequacy criterion** specifies requirements for testing
- May be used as:
  - A **stopping rule**: has enough testing taken place?
  - A **measurement**: how far has testing proceeded?
  - As a **test-case generator**: do we need to choose an additional test case?

Fault detection vs. confidence building

- **Fault detection**: uses a constructive approach:
  - Input domain is partitioned into finite, small number of subdomains
  - Each of these subdomains corresponds to an **equivalence class**
  - That is, each member of a subdomain is as good as any other
  - Difficulty: we do not know where the needles are in the haystack
- **Confidence building**: daily operation of system is free of failure
  - must mimic that situation
  - large number of test cases representing typical usage scenarios

Summary so far

- Different approaches to testing can be applied in the same software-development phase depending upon our goal
- There are differences between errors, failures, and faults
  - There does not exist a one-to-one correspondence between them…
  - … and this makes testing hard
- Testing can also be more than just about finding faults
  - Depending upon what we are trying to achieve at any particular point, we may be happy to stop given a certain amount of "coverage"

Implementation (module) testing

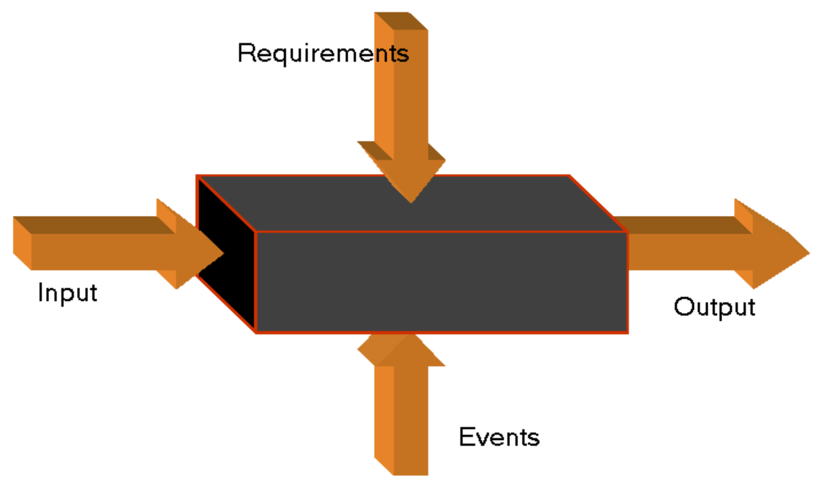
- Modules informally tested by programmer while coding ("desk checking")
- When programmer is satisfied that module functions correctly, methodical testing of module undertaken by separate test team.
  - Developer: understands system, but will test "gently"; driven by **delivery**
  - Independent tester: must learn about system, but will attempt to break it; driven by **quality**
- Two types of methodical testing:
  - **nonexecution** based
  - **execution** based

Execution-based testing

- Two ways of constructing test cases for a module: black-box testing and white-box testing
- **black-box:**
  - code is ignored
  - only specification document used to design test cases
- **white-box:**
  - code may be examined
  - specifications may or may not be used
- Most test plans are a combination of these two approaches

Black-box testing

- Functionality of each module tested against:
  - specifications (requirements)
  - context (events)
- Only correct input / output relationships are examined.
- Also called **specifications testing**, **behavioral**, **data-driven**, **functional**, **input/ output driven**



The Problem

- Every combination of input and output would require an immense number of test cases
- Result:
  - exhaustive black-box testing is often unreasonable
  - art of testing is in finding small, manageable sets of test cases (or their equivalence classes)
  - trying to maximize chances of detecting a fault
  - at the same time, minimize redundancy amongst tests

Equivalence Testing + Boundary Value Analysis

- Technique for selecting test case
  - new cases are chosen to detect previous undetected faults
  - an **equivalence class** is a set of test cases such that any one member of class is representative of any other member
- Example: **Database product**
  - must be able to handle any number of records from 1 through 16,383
  - if it can handle, say, 34 records and 14,870 records, then it probably works with 8534 records
  - equivalence class 1: less than one record
  - equivalence class 2: from 1 to 16,383 records
  - equivalence class 3: more then 16,383 records
- Testing database therefore requires one case per class

Equivalence Testing + Boundary Value Analysis

- Here a successful test case is one which detects a previously undetected fault
- Boundary-value analysis helps identify more possible test cases
  - testing one or just to one side of a boundary of an equivalence class
- This suggests the following test cases:

Test case 1	0 records	Member of equivalence class 1 and adjacent to boundary value
Test case 2	1 record	Boundary value
Test case 3	2 records	Adjacent to boundary value
Test case 4	723 records	Member of equivalence class 2
Test case 5	16382 records	Adjacent to boundary value
Test case 6	16383 records	Boundary value
Test case 7	16384 records	Member of equivalence class 3 and adjacent to boundary value

Equivalence Testing + Boundary Value Analysis

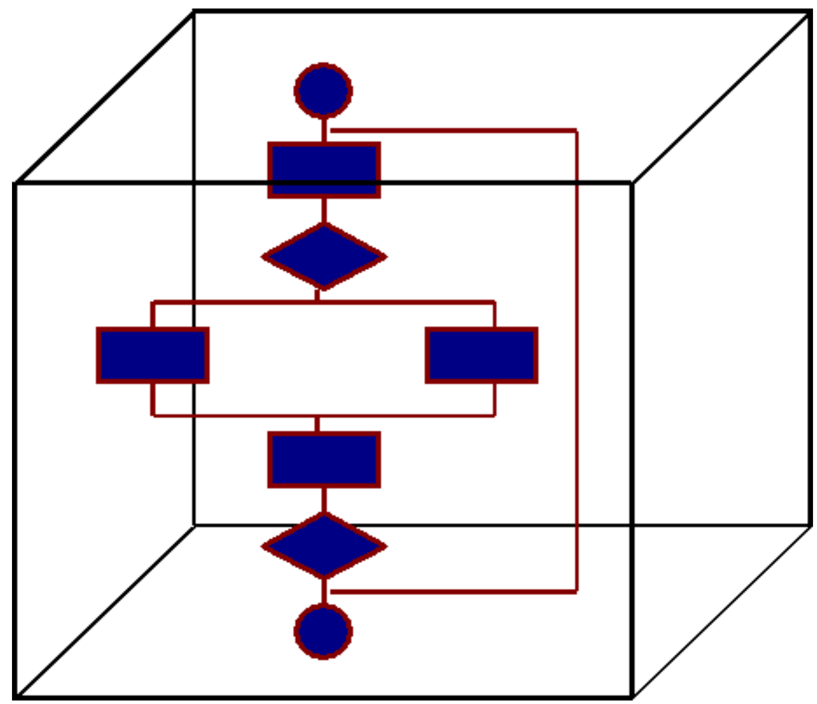
- The example was applied to input specifications
- Can also apply to output specifications
- Test space can be partitioned into “points”
  - ON point: on class boundaries
  - OFF point of a border: just inside the border
  - previous example had 7 points (2 ON, 5 OFF)
- Given subdomains D_i where $i = 1, \ldots, n$,
  - may construct a set of N test cases for ON points of each border B of each sub-domain D_i
  - and at least one test cases for an OFF point of each border
  - then the resulting test set is called **N * 1 domain adequate**

Functional testing

- Another form of black-box testing
- Test data is based on the **functionality** of module
- Each function implemented in the module is separately identified
  - Usually the function is expressed using some mathematical notation
- From this are devised test cases to be applied to each function separately
- Can be difficult to do:
  - functions may themselves consists of lower-level functions, each of which must be tested first
  - lower-level function may not be independent
  - functionality might not coincide with module boundaries (e.g. object invoking a method on another object)

White box testing

- Test cases are selected on basis of code examination rather than specification
- "paths" through code are examined and "exercised"
- Also known as **glass-box**, **structural**, **logic-driven**, **path-oriented**



Reality check

- After black-box testing:
  - requirements are fulfilled
  - interfaces are available and working
- So why bother with white-box testing?
  - If the probability of a path's execution is low…
  - Then logic errors and incorrect assumptions associated with path will not be detected durinb black-box.
  - Errors caused by types are random; it is probably that untested paths will contain some errors

Coverage-based testing

- Goodness is determined by the coverage of the product by the test set so far: e.g., % of statements or requirements tested
- Often based on control-flow graph of the program
- Three techniques:
  - control-flow coverage
  - data-flow coverage
  - coverage-based testing of requirements

Example of control-flow coverage

```
procedure bubble (var a: array [1..n] of integer; n: integer);
    var i, j: temp: integer;
    begin
        for i:= 2 to n do
            if a[i] >= a[i-1] then goto next endif;
            j:= i;
        loop: if j <= 1 then goto next endif;
            if a[j] >= a[j-1] then goto next endif;
            temp:= a[j]; a[j]:= a[j-1]; a[j-1]:= temp; j:= j-1; goto loop;
        next: skip;
        enddo
    end bubble;
```

input: n=2, a[1] = 5, a[2] = 3

Control-flow coverage

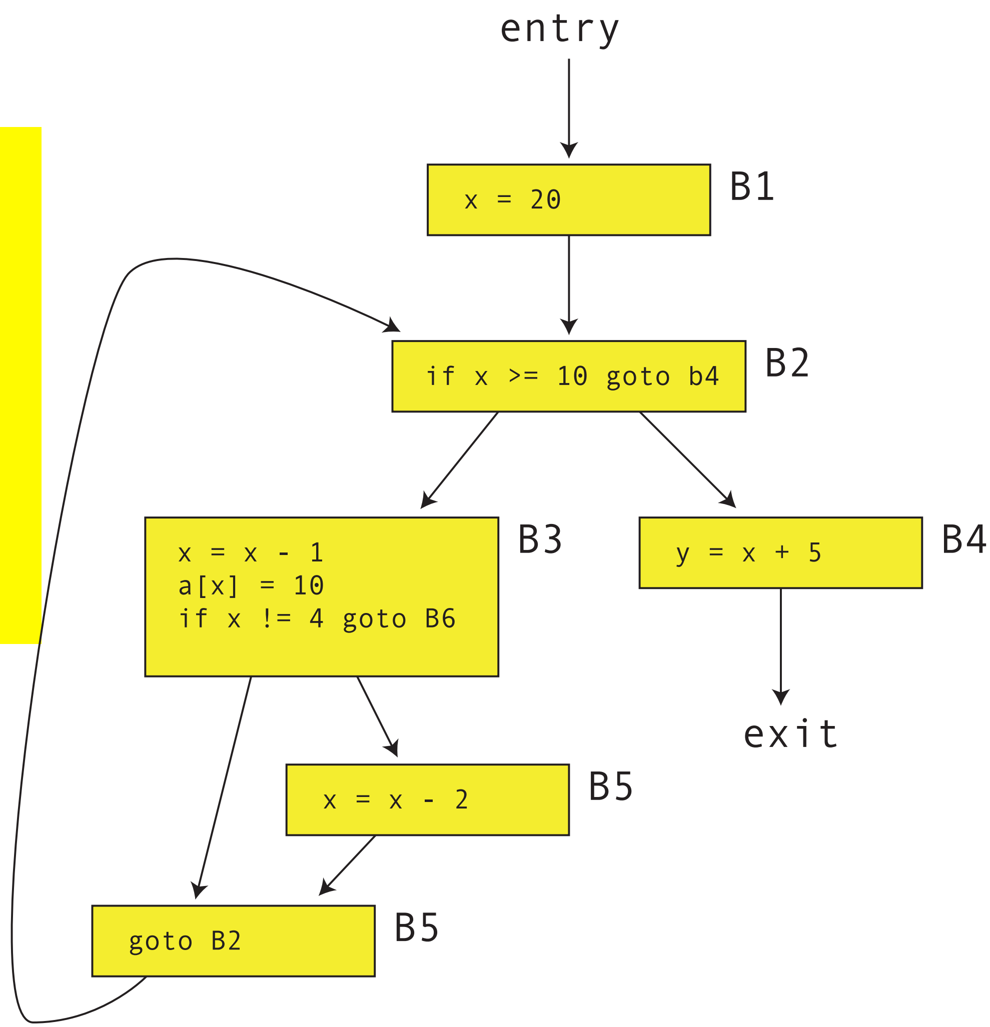
- Previous example is about **All-Nodes coverage**, or **statement coverage**
- A stronger criterion: **All-Edges coverage,** or **branch coverage**
- Variations exercise all combinations of elementary predicates in a branch condition
- Strongest: **All-Paths coverage** (≡ exhaustive testing)

Another code example with flow graph

```
x = 20

while x < 10:
    x = x - 1
    a[x] = 10
    if x == 4:
        x = x - 2

y = x + 5
```



Data-flow coverage

- Looks how variables are treated along paths through the control graph.
- Variables are **defined** when they get a new value.
- A definition in statement X is **alive** in statement Y if there is a path from X to Y in which this variable is not defined anew. Such a path is called **definition-clear.**
- We may now test all definition-clear paths between each definition and each use of that definition and each successor of that node: **All-Uses coverage**.