COMP354

Software Engineering

Lecture 19–21

Validation and Verification, Testing

Outline

What is V&V?

What must be verified?

Static and dynamic verification

Testing = verifying correctness

- Aim of testing
- Phase-driven testing
- Top-down vs bottom-up
- Test plan, test data, test cases
- Designing tests: black-box, white-box
- When to stop testing

Reviews, walkthroughs, inspections

Debugging

Validation and Verification

validation: Does the product satisfy the users' needs?
"building the right product"

verification: Does a product, say the code, conform to another product, say the SRD?
"building the product right"

What must be verified?

EVERY quality of the product (and deliverables)
correctness, performance, reliability,
robustness, portability, maintainability,
user friendliness, ...

Results may be Yes/No

Results may be quantity : eg percentage of tests passed

Results may be subjective : eg how portable is a system

Static and dynamic verification

Static verification: analyzing the product to deduce its correct operation as a logical consequence of the definition (document)

eg reviews, walkthroughs, inspections, compiler and cross reference checking

Dynamic verification: experimenting with the behaviour of a product to see whether it performs as expected

eg testing

Note partial dynamic nature of symbolic execution, hand execution, walkthroughs

Testing

Testing is verifying the quality of correctness

Aim of testing: *To discover the presence of errors!!!*

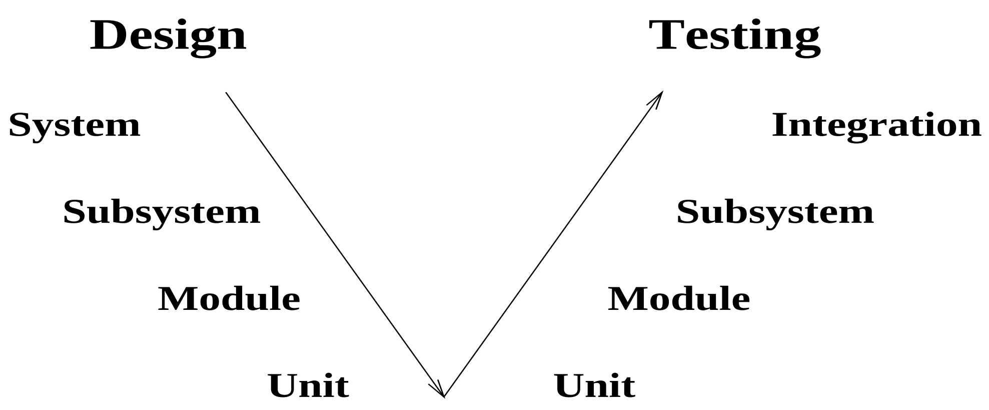
Main problem: **testing can never be complete!**

Varieties of Testing

1. Goal-driven testing.
   (a) Requirements-driven testing. Develop a *test-case matrix* (requirements *vs* tests) to insure that each requirement undergoes at least one test.
   (b) Structure-driven testing. Construct tests to cover as much of the logical structure of the program as possible. A *test coverage analyzer* is a tool that helps to ensure full coverage.
   (c) Statistics-driven testing. These tests are run to convince the client that the software is working by running typical applications. Results are often statistical.
   (d) Risk-driven testing. These tests check "worst case" scenarios and boundary conditions. They ensure robustness.

2. Phase-driven testing.

Phase-Driven Testing



Stages of Testing during Development

Unit Testing testing an individual unit or basic component of the system
eg testing a function SQRT

Module Testing testing a module consisting of several units, to check that their interaction and interfaces are ok
eg testing that pop and push in a stack module provide a last-in-first-out behaviour

Subsystem Testing testing a subsystem consisting of several modules, to check module interaction and module interfaces are ok
eg an output subsystem writing to a database, and keeping an audit trail, and providing rollback services

Integration Testing testing the entire system once all the subsystems have been integrated

More Terminology of Testing

Acceptance Testing testing with real data for the customer to satisfy the customer that the system meets the requirements

in customer environment

Stress Testing (Overload Testing) check the capability of the system to perform under “overloaded” conditions

eg full tables, memory, filesystem more simultaneous users than expected

done after integration testing

Regression Testing testing old features again when new features are added or changes are made

to make sure you fully understood the changes to be made and did not impact parts of the system supposedly isolated from the changes

regression is degradation in level of correctness (of old features)

Testing for Robustness testing the system with unexpected situations like wrong user commands

Top-down vs Bottom-up Testing

Top-down testing: test subsystems, then modules, then units

need a STUB (prototype) for each non-completed module or unit
a stub simulates the behaviour of the module/unit
may do nothing; may always exhibit the same default behaviour; may only handle those cases needed for tests

Advantages:
catches high-level design errors early: saves cost

Disadvantages:
not so easy to devise tests at this level; additional cost of implementing stubs
(high-level design errors can be caught in design walk-throughs)

Bottom-up testing: test units, then modules, then subsystems

need a DRIVER to supply necessary input, output, data structures to test the unit, module, subsystem

Advantages:
every component is tested before integration into a larger component; makes location of errors easier, since we first assume it is in the integration/interaction of subcomponents rather than within the subcomponents

Disadvantages:
delays detection of high-level errors: more costly to fix; additional cost of writing drivers

Test plan, test data, test cases

test data: a set of inputs devised to exercise a test

test case consists of

1. the purpose of the test in terms of the system requirements it exercises
2. an input specification (ie test data)
3. a specification of the expected output

test plan: the major components are

- a description of the major phases of testing (eg unit testing, module testing, ...)
- objectives (acceptance criteria) for the testing process
- an overall testing schedule and resource allocation (when, who, time and machine resources)
- a description of the relationship between the test plan and other project plans (eg implementation schedule)
- a description of how traceability of test results to system requirements is to be demonstrated
- a description of how tests results are recorded (It must be possible to audit the testing process to guarantee that tests have been carried out on latest versions of the software.)
- a description of how the test cases were designed, and how the test data was generated
- a description of all the test cases, including all test data

The test schedule should allow for slippage

Designing Tests

Ideal: completely test for each requirement in the SRD

"cover" the SRD

Suppose the SRD contains the following requirement:
When the user enters X and Y, the program displays $X + Y$.

X and Y are 32-bit numbers $\Rightarrow 2^{64}$ tests!

In practice, assume some form of **continuity**:

If the program adds a few numbers correctly,
and it works at the boundaries,
we **assume** that it adds all numbers correctly.

There are two possibilities:

Black box testing: choose tests without knowledge of how the program works,
i.e. based on requirements only.

White box testing: choose tests based on our knowledge of how the program works.

Black-Box Testing

also called *Functional Testing*

based on our view of the program as a *function* from inputs to outputs

use "equivalence partitioning" of input space:

> partition the domain of inputs into disjoint sets such that inputs in the same set exhibit
> *similar/identical/equivalent*
> properties with respect to the test being performed

```
Example 1: function which takes a 5-digit number as input
   set 1 = { integers < 10000 }
   set 2 = { integers  in the range 10000 .. 99999 }
   set 3 = { integers > 99999 }

Example 2: function which takes a linked list as input
   set 1 = { empty list }
   set 2 = { lists of length 1 }      first node = last node
   set 3 = { lists of moderate length }
   set 4 = { very long lists }        might be space problems
```

Guidelines:

- Test for success and failure.
- Test boundary conditions.
- Test as many *combinations* as feasible.

White-Box Testing

also called *Structural Testing* or *Glass-Box Testing*

based on the structure of the code:

- the control flow graph, and
- the conditions controlling if-statements and loops

general idea is to ensure "coverage" of every component

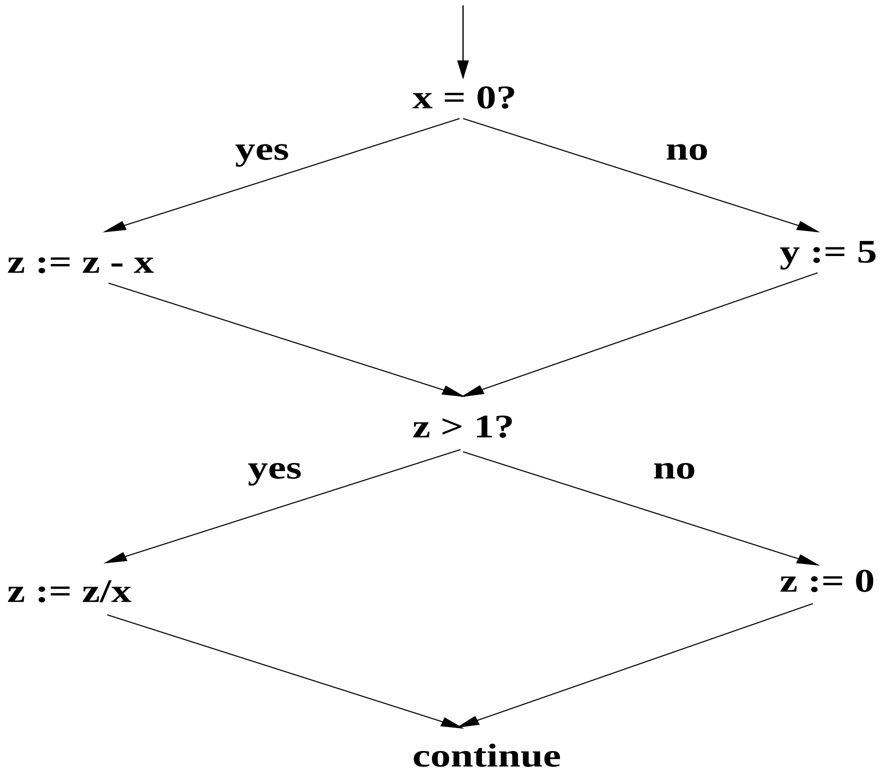
all statements Every statement in the program must be exercised during testing.

all edges All edges of the control graph must be exercised.

all branches Each possibility at a branch point (`if` or `case` statement) should be exercised.

all paths Exercise all paths: usually intractable.

White-Box Testing Example



Consider the following tests:

Test	X	Z
1	0	1
2	1	3
3	0	3
4	1	1

use either {1,2} or {3,4} to cover all edges

{3,4} reveals the “divide by zero” error, whereas {1,2} does not

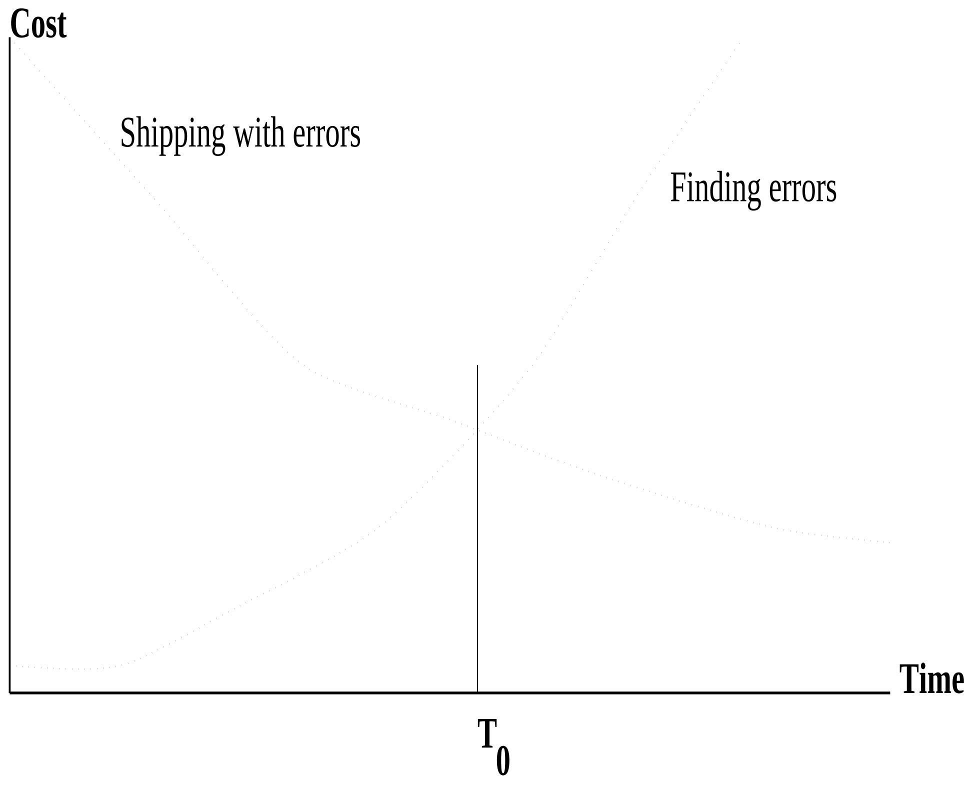
use {1,2,3,4} to cover all paths

When to Stop Testing

Ideal: stop when all tests succeed.

Practice: stop when the cost of testing exceeds the cost of shipping with errors.

Microsoft: let the customer find the errors :).



As the number of errors gets smaller, the cost of finding errors increases.

Shipping with errors has a cost, because customer support is needed, but this cost falls with the number of remaining errors.

Problem: hard to know when T_0 has been reached!

Reviews, Walkthroughs, Inspections

Aim: discovery of errors — NOT fixing errors

Basic idea: team examines a software document during a meeting

Why: a group of people are more effective

Common features include:

- a small group of people;
- the person responsible for the document (analyst, designer, programmer, etc) should attend;
- one person records the discussion;
- managers must **not** be present, because they inhibit discussion;
- errors are recorded, but are **not** corrected.

Some general rules:

- Teams prepare in advance, e.g. read the document
- Meetings are not long — at most 3 hours — so concentration can be maintained.
- A *moderator* is advisable to prevent discussions from rambling.
- The author may be required to keep silent except to respond to questions. If the author explains what s/he thought s/he was doing, others may be distracted from what is actually written.
- All members must avoid possessiveness and egotism, cooperating on finding errors, not defending their own contributions.

During a **review**:

- the author of the document presents the main themes;
- others criticize, discuss, look for omissions, inconsistencies, redundancies, etc.
- faults and potential faults are recorded.

During a **walkthrough**:

- Each statement or sentence is read by the author;
- others ask for explanation or justification if necessary.

Example:

> "Since $n > 0$, we can divide . . . ."
> "How do you know that $n > 0$?"

During an **inspection**:

- code is carefully examined, with everyone looking for common errors.

Note that various authors and companies use the terms review, walkthrough, inspection *differently.*

Sample Checklist of Common Errors

- use of uninitialized variables
- have all constants been named
- confusing = with == in C boolean expression
- for each conditional statement, is the condition correct
- jumps into loops
- incompatible types in an assignment
- nonterminating loops
- for each array,
  should the lower bound be 0, 1, or something else
  should the upper bound be Size or Size-1
- array indexes out of bounds
- are delimiters `\0` explicitly assigned for character strings
- improper storage allocation/deallocation
- actual-formal parameter mismatches in procedure calls: correct number of parameters; matching types of each formal-actual parameter pair
- comparison for equality of floating point values
- are compound statements correctly bracketted
- if links/pointers are used, are link assignments correct when updating data structures using links
- have all possible error conditions been taken into account

Debugging

Steps:

1. locate error

   verification has discovered presence of an error

   Locating errors is difficult!

   Reviews etc often give location of error.

   Debugger tools help immensely (xxgdb)

2. ascertain how to correct error

   Is it a mistake in
   - coding?
   - design?
   - requirements?

3. fix it, or report a change request (CR)

   fix it if it is a coding error

   fix should explain **all** symptoms of the error

4. re-run **all** tests (regression testing)

   once all reported errors have been located, explained fully, and coding errors fixed