Introduction to Software Engineering

5. Software Validation

Roadmap

> Reliability, Failures and Faults

> Fault Avoidance

> Fault Tolerance

> Verification and Validation

> The Testing process
  — Black box testing
  — White box testing
  — Statistical testing

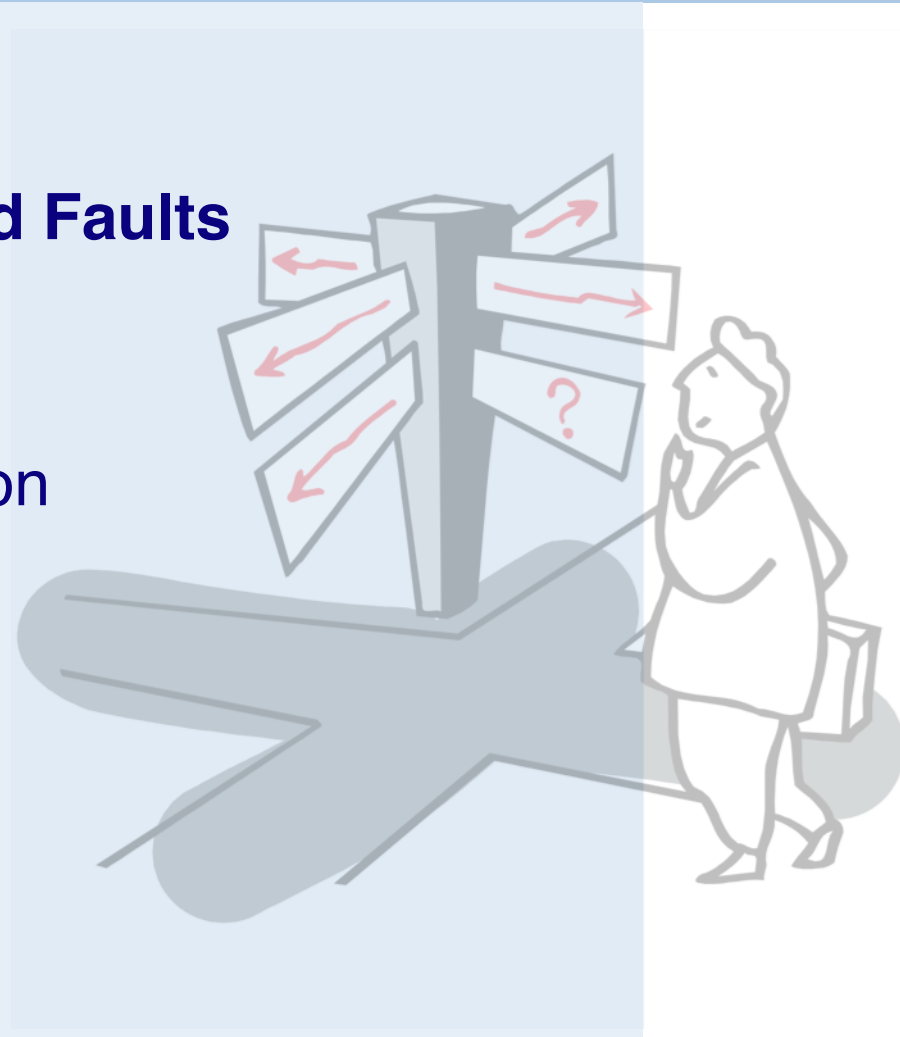# Source

> *Software Engineering*, I. Sommerville, 7th Edn., 2004.

Roadmap

> **Reliability, Failures and Faults**

> Fault Avoidance

> Fault Tolerance

> Verification and Validation

> The Testing process
  — Black box testing
  — White box testing
  — Statistical testing

Software Reliability, Failures and Faults

The <u>reliability</u> of a software system is a measure of how well it provides the services expected by its users, expressed in terms of software failures.

> A <u>software failure</u> is an *execution event* where the software behaves in an unexpected or undesirable way.

> A <u>software fault</u> is an *erroneous portion of a software system* which may cause failures to occur if it is run in a particular state, or with particular inputs.

Kinds of failures

Failure class	Description
Transient	Occurs only with *certain inputs*
Permanent	Occurs with *all inputs*
Recoverable	System can recover *without operator intervention*
Unrecoverable	Operator intervention is needed to recover from failure
Non-corrupting	Failure does not corrupt data
Corrupting	Failure corrupts system data

Programming for Reliability

Fault avoidance:

> development techniques to *reduce the number of faults in a system*

Fault tolerance:

> developing programs that will *operate despite the presence of faults*

Roadmap

> Reliability, Failures and Faults
> **Fault Avoidance**
> Fault Tolerance
> Verification and Validation
> The Testing process
— Black box testing
— White box testing
— Statistical testing

Fault Avoidance

Fault avoidance depends on:

1. A precise *system specification* (preferably formal)
2. Software design based on *information hiding and encapsulation*
3. Extensive *validation reviews* during the development process
4. An organizational *quality philosophy* to drive the software process
5. Planned *system testing* to expose faults and assess reliability

Common Sources of Software Faults

Several features of programming languages and systems are common sources of faults in software systems:

> **Goto statements** and other unstructured programming constructs make programs *hard to understand, reason about and modify*.
>> — Use structured programming constructs

> **Floating point numbers** are *inherently imprecise* and may lead to invalid comparisons.
>> — Fixed point numbers are safer for exact comparisons

> **Pointers** are dangerous because of *aliasing*, and the risk of *corrupting memory*
>> — Pointer usage should be confined to abstract data type implementations

Common Sources of Software Faults ...

> *Parallelism* is dangerous because *timing differences* can affect overall program behaviour in *hard-to-predict* ways.
> — Minimize inter-process dependencies

> *Recursion* can lead to *convoluted logic*, and may exhaust (stack) memory.
> — Use recursion in a disciplined way, within a controlled scope

> *Interrupts* force transfer of control *independent of the current context*, and may cause a critical operation to be terminated.
> — Minimize the use of interrupts; prefer disciplined exceptions

Roadmap

> Reliability, Failures and Faults
> Fault Avoidance
> **Fault Tolerance**
> Verification and Validation
> The Testing process
— Black box testing
— White box testing
— Statistical testing

Fault Tolerance

A fault-tolerant system must carry out four activities:

1. **Failure detection**: *detect* that the system has reached a particular state or will result in a system failure

2. **Damage assessment**: *detect which parts* of the system state have been affected by the failure

3. **Fault recovery**: *restore the state* to a known, "safe" state (either by correcting the damaged state, or backing up to a previous, safe state)

4. **Fault repair**: *modify the system* so the fault does not recur (!)

Approaches to Fault Tolerance

N-version Programming:

Multiple versions of the software system are implemented *independently by different teams*.

The final system:

> runs all the versions in *parallel*,

> *compares* their results using a voting system, and

> *rejects* inconsistent outputs.
> (At least three versions should be available!)

Approaches to Fault Tolerance ...

Recovery Blocks:

A finer-grained approach in which a program unit contains a *test* to check for failure, and *alternative code* to back up and try in case of failure.

> alternatives are executed in *sequence*, not in parallel
> the *failure test is independent* (not by voting)

Defensive Programming

Failure detection:

> Use the *type system* to ensure that variables do not get assigned invalid values.

> Use *assertions* to detect failures and raise exceptions. Explicitly state and check all invariants for abstract data types, and pre- and post-conditions of procedures as assertions. Use exception handlers to recover from failures.

> Use *damage assessment procedures*, where appropriate, to assess what parts of the state have been affected, before attempting to fix the damage.

Fault recovery:

> *Backward recovery:* backup to a previous, consistent state

> *Forward recovery:* make use of redundant information to reconstruct a consistent state from corrupted data

Examples

> Concurrency control
— Pessimistic (locking)
– *Java synchronization; rcs*
— Optimistic (check for conflict before commit)
– *Cvs, Subversion*
— Distributed
– *Git, Monticello*

> Fault recovery
— Change logs (rollback and replay)
– *Smalltalk image and changes*
— Transactional Memory (software and hardware)
– *ACID (Atomicy, Consistency, Isolation, Durability)*

Roadmap

> Reliability, Failures and Faults
> Fault Avoidance
> Fault Tolerance
> **Verification and Validation**
> The Testing process
>> — Black box testing
>> — White box testing
>> — Statistical testing

Verification and Validation

Verification:

> Are we *building the product right*?
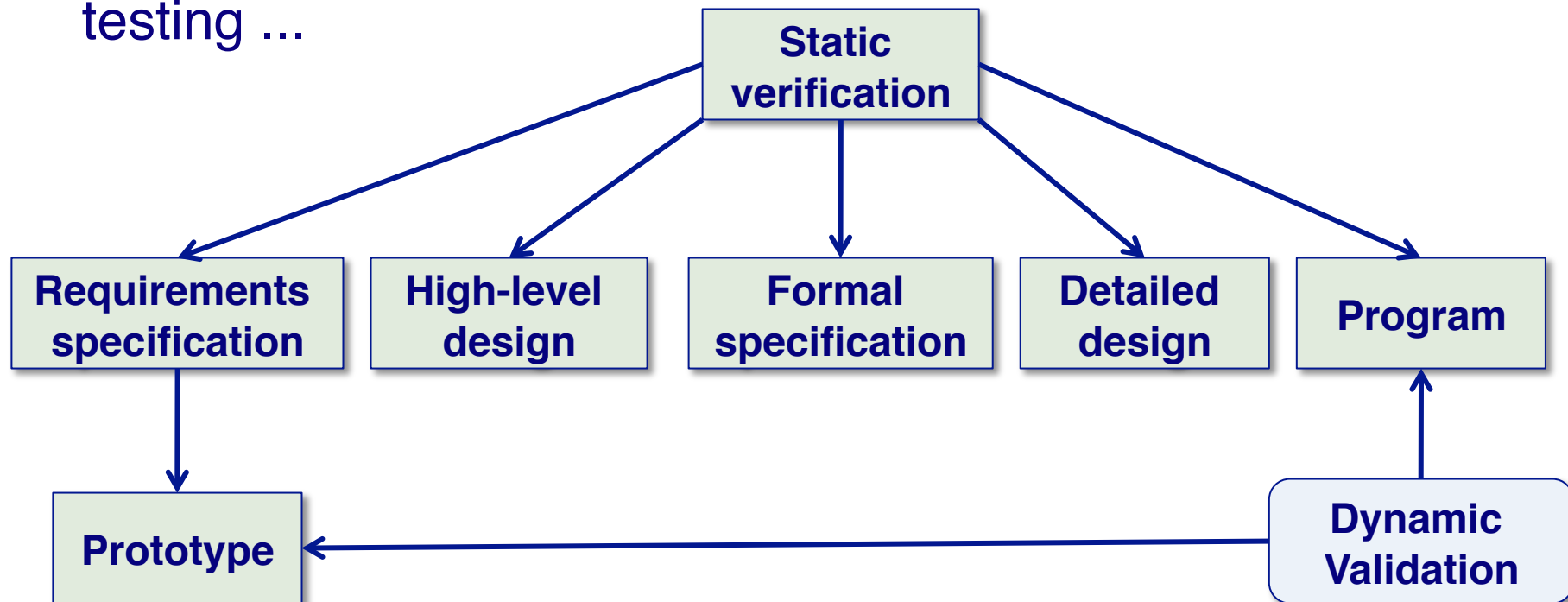>> — i.e., does it conform to specs?

Validation:

> Are we building the *right product*?
>> — i.e., does it meet expectations?

Verification and Validation ...

Static techniques include program inspection, analysis and formal verification.

Dynamic techniques include statistical testing and defect testing ...

Static Verification

Program Inspections:

> Small team systematically checks program code

> Inspection checklist often drives this activity

— e.g., "Are all invariants, pre- and post-conditions checked?" ...

Static Program Analysers:

> Complements compiler to check for common errors

— e.g., variable use before initialization

Mathematically-based Verification:

> Use mathematical reasoning to demonstrate that program meets specification

— e.g., that invariants are not violated, that loops terminate, etc.

— e.g., model-checking tools

Roadmap

> Reliability, Failures and Faults

> Fault Avoidance

> Fault Tolerance

> Verification and Validation

> **The Testing process**

 — Black box testing

 — White box testing

 — Statistical testing

The Testing Process

1. Unit testing:

    — Individual (stand-alone) *components* are tested to ensure that they operate correctly.

2. Module testing:

    — A collection of *related components* (a module) is tested as a group.

3. Sub-system testing:

    — The phase tests a *set of modules* integrated as a sub-system. Since the most common problems in large systems arise from sub-system interface mismatches, this phase focuses on testing these interfaces.

The Testing Process ...

4.   System testing:

—   This phase concentrates on (i) detecting errors resulting from unexpected interactions between sub-systems, and (ii) validating that the complete systems fulfils functional and non-functional requirements.

5.   Acceptance testing (alpha/beta testing):

—   The system is tested with *real* rather than simulated data.

Testing is iterative! <u>Regression testing</u> is performed when defects are repaired.

Regression testing

<u>Regression testing</u> means testing that everything that used to work *still works* after changes are made to the system!

> tests must be *deterministic* and *repeatable*

> should test "all" functionality
  — every interface
  — all boundary situations
  — every feature
  — every line of code
  — everything that can conceivably go wrong!

It costs extra work to define tests up front, but they pay off in debugging & maintenance!

Test Planning

The preparation of the test plan should begin *when the system requirements are formulated*, and the plan should be developed in detail *as the software is designed.*
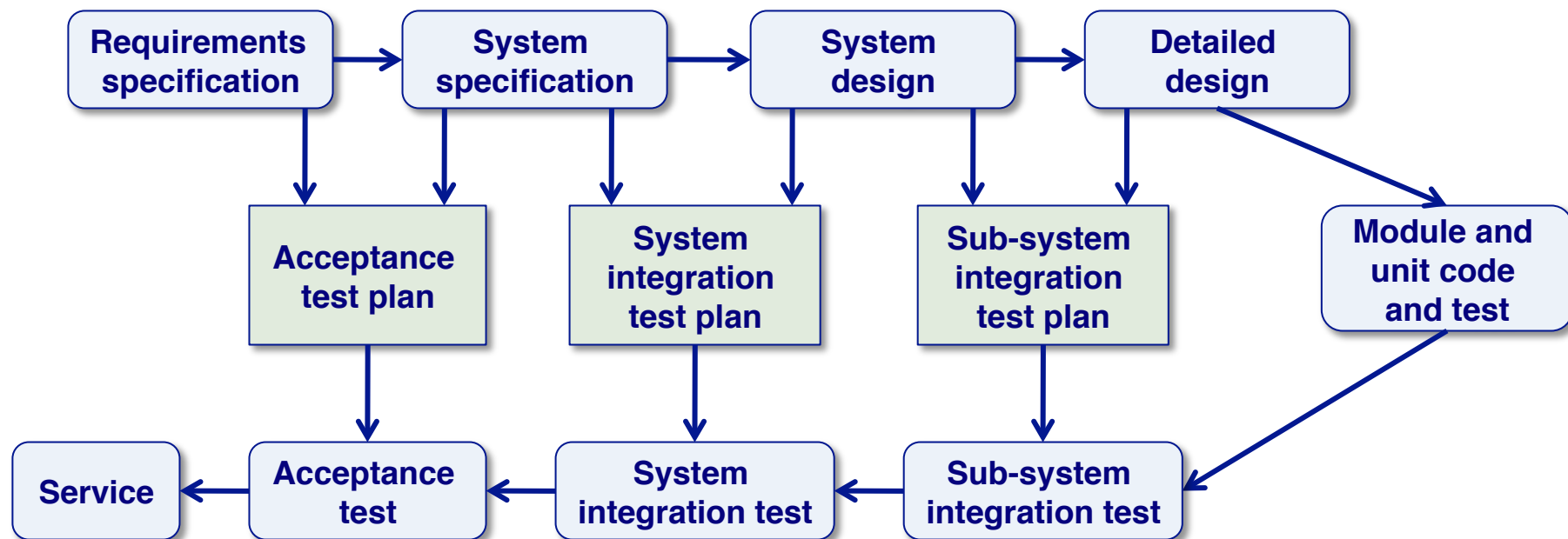
The plan should be *revised regularly*, and tests should be *repeated and extended* where the software process iterates.

Top-down Testing

> *Start with sub-systems*, where modules are represented by "stubs"
> Similarly test modules, representing functions as stubs
> *Coding and testing* are carried out as a *single activity*
> Design errors can be detected early on, avoiding expensive redesign
> Always have a running (if limited) system!

BUT: may be impractical for stubs to simulate complex components

Bottom-up Testing

> *Start by testing units* and modules

> *Test drivers* must be written to exercise lower-level components

> Works well for *reusable components* to be shared with other projects

BUT: pure bottom-up testing will not uncover *architectural faults* till late in the software process

Typically a combination of top-down and bottom-up testing is best.

Testing vs Correctness

> "Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence."

— *Edsger Dijkstra, The Humble Programmer, ACM Turing lecture, 1972*

Defect Testing

Tests are designed to *reveal the presence of defects* in the system.

Testing should, in principle, be exhaustive, but in practice can only be representative.

Test data are inputs devised to test the system.

Test cases are input/output specifications for a particular function being tested.

Defect Testing ...

Petschenik (1985) proposes:

1. "Testing a system's *capabilities* is more important than testing its components."
   — Choose test cases that will identify situations that may prevent users from doing their job.

2. "Testing *old capabilities* is more important than testing new capabilities."
   — Always perform regression tests when the system is modified.

3. "Testing *typical situations* is more important than testing boundary value cases."
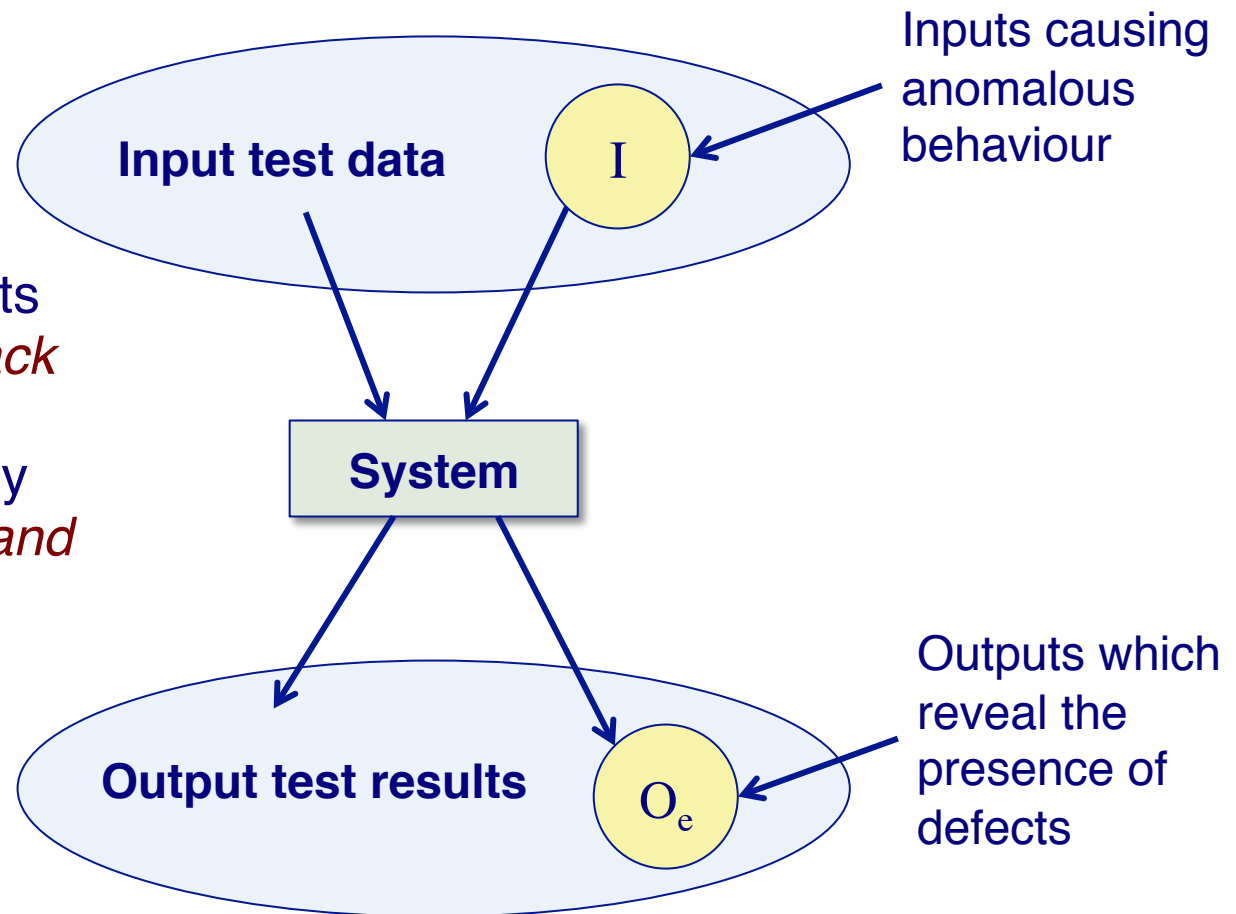   — If resources are limited, focus on typical usage patterns.

Roadmap

> Reliability, Failures and Faults

> Fault Avoidance

> Fault Tolerance

> Verification and Validation

> The Testing process
  — **Black box testing**
  — White box testing
  — Statistical testing

Functional (black box) testing

Functional testing treats a component as a *"black box"* whose behaviour can be determined only by studying its *inputs and outputs*.

Input test data

I

Inputs causing anomalous behaviour

System

Output test results

O_e

Outputs which reveal the presence of defects

Coverage Criteria

Test cases are derived from the *external specification* of the component and should cover:

> all exceptions

> all data ranges (incl. invalid) generating different classes of output

> all boundary values

Test cases can be derived from a component's *interface*, by assuming that the component will behave similarly for all members of an *equivalence partition* ...

Equivalence partitioning

```
public static void search(int key, int [] elemArray, Result r)
    { … }
```

Check input partitions:

> Do the inputs fulfil the *pre-conditions*?
— is the array sorted, non-empty ...
> Is the key in the array?
— leads to (at least) 2x2 equivalence classes

Check boundary conditions:

> Is the array of length 1?
> Is the key at the start or end of the array?
— leads to further subdivisions (not all combinations make sense)

Test Cases and Test Data

Generate test data that cover all *meaningful* equivalence partitions.

| Test Cases	Test Data
Array length 0	key = 17, elements = { }
Array not sorted	key = 17, elements = { 33, 20, 17, 18 }
Array size 1, key in array	key = 17, elements = { 17 }
Array size 1, key not in array	key = 0, elements = { 17 }
Array size > 1, key is first element	key = 17, elements = { 17, 18, 20, 33 }
Array size > 1, key is last element	key = 33, elements = { 17, 18, 20, 33 }
Array size > 1, key is in middle	key = 20, elements = { 17, 18, 20, 33 }
Array size > 1, key not in array	key = 50, elements = { 17, 18, 20, 33 }
...	

Roadmap

> Reliability, Failures and Faults
> Fault Avoidance
> Fault Tolerance
> Verification and Validation
> The Testing process
> — Black box testing
> — **White box testing**
> — Statistical testing

Structural (white box) Testing

<u>Structural testing</u> treats a component as a *"white box"* or "glass box" whose *structure can be examined to generate test cases.*

Derive test cases to *maximize coverage* of that structure, yet *minimize the number of test cases*.

Tests

Derives

Test data

Component code

Test outputs
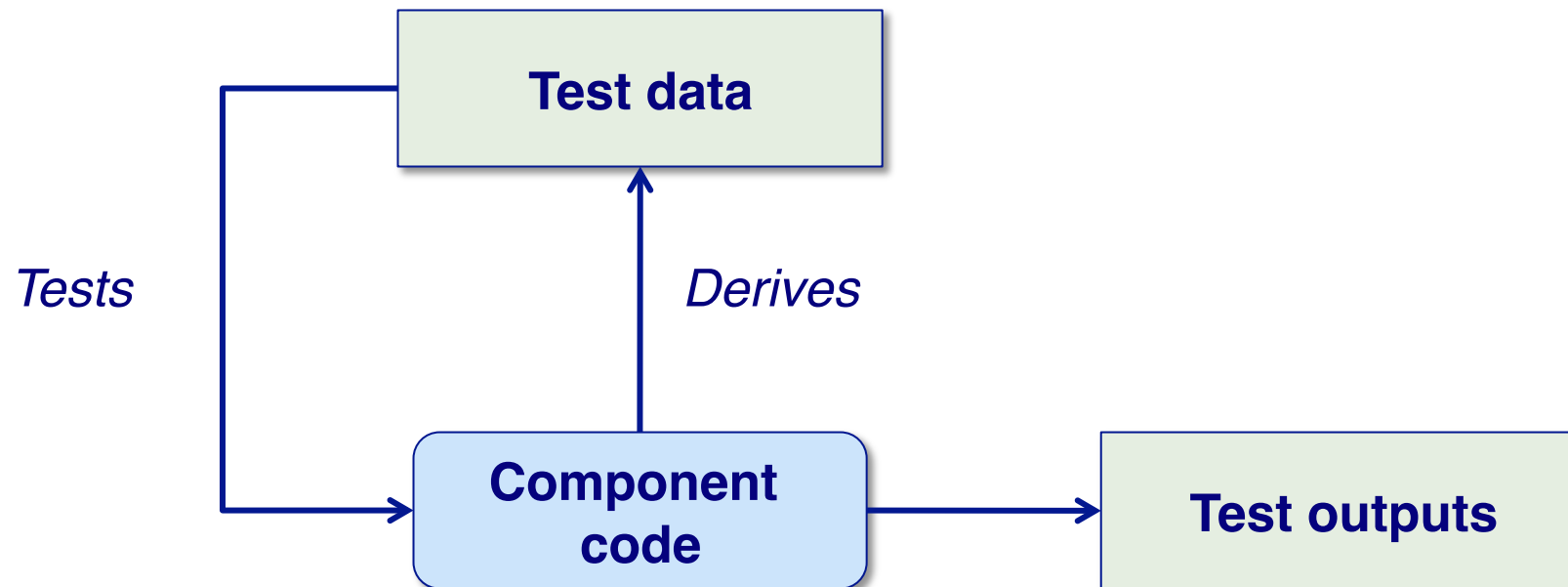
Coverage criteria

> *every statement* at least once
> *all portions of control flow* at least once
> *all possible values of compound conditions* at least once
> *all portions of data flow* at least once
> for *all loops* L, with n allowable passes:
>> I.   skip the loop;
>> II. 1 pass through the loop
>> III. 2 passes
>> IV. m passes where 2 < m < n
>> V. n-1, n, n+1 passes

Path testing is a white-box strategy which exercises *every independent execution path* through a component.

```java
class BinSearch {
// This is an encapsulation of a binary search function that takes an array of
// ordered objects and a key and returns an object with 2 attributes namely
// index - the value of the array index
// found - a boolean indicating whether or not the key is in the array
// An object is returned because it is not possible in Java to pass basic types by
// reference to a function and so return two values
// the key is -1 if the element is not found
   public static void search (int key, int [] elemArray, Result r)
   {
       int bottom = 0;
       int top = elemArray.length - 1;
       int mid;
       r.found = false; r.index = -1;                               (1)
       while ( bottom <= top)                                       (2)
       {
           mid = (top + bottom) / 2;
           if (elemArray [mid] == key)                              (3)
           {
               r.index = mid;                                       (8)
               r.found = true;
               return ;                                          -> (9)
           } // if part
           else
           {
               if (elemArray [mid] < key)                           (4)
                   bottom = mid + 1;                                (5)
               else
                   top = mid -i;                                    (6)
           }                                                        (7)
       } //while loop
   } //search                                                       (9)
} //BinSearch
```

Program flow graphs

> Each branch is shown as a separate path and loops are shown by arrows looping back to the loop condition node

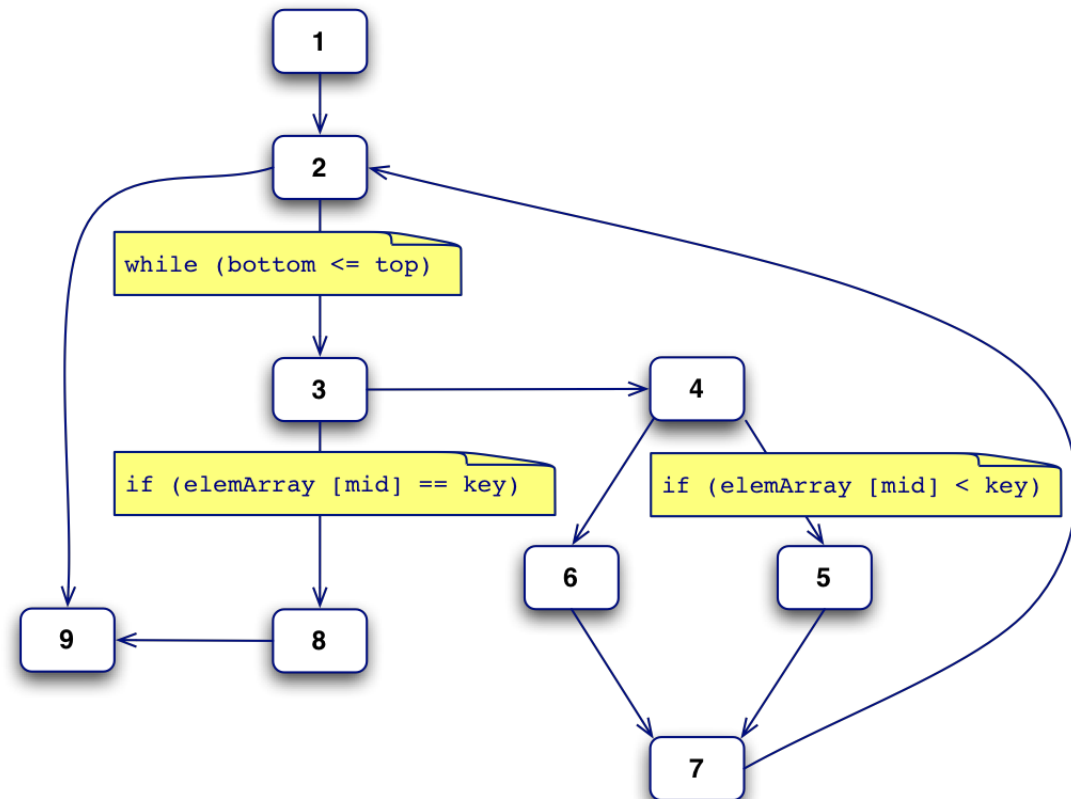> The number of tests to test all control statements equals the *cyclomatic complexity*

Cyclomatic complexity = Number of edges - Number of nodes +2

Path Testing

Test cases should be chosen to cover all *independent paths* through a routine:

— 1, 2, 9
— 1, 2, 3, 8, 9
— 1, 2, 3, 4, 5, 7, 2, 9
— 1, 2, 3, 4, 6, 7, 2, 9

(Each path traverses at least one new edge)


```
1
2
while (bottom <= top)
3          4
if (elemArray [mid] == key)    if (elemArray [mid] < key)
6          5
9    8
7
```

Roadmap

> Reliability, Failures and Faults

> Fault Avoidance

> Fault Tolerance

> Verification and Validation

> The Testing process
   — Black box testing
   — White box testing
   — **Statistical testing**

Statistical Testing

The objective of <u>statistical testing</u> is to determine the *reliability* of the software, rather than to discover faults.

<u>Reliability</u> may be expressed as:

> *probability* of failure on demand
>> — i.e., for safety-critical systems

> *rate* of failure occurrence
>> — i.e., #failures/time unit

> *mean time* to failure
>> — i.e., for a stable system

> *availability*
>> — i.e., fraction of time, for e.g. telecom systems

Statistical Testing ...

Tests are designed to reflect the *frequency of actual user inputs* and, after running the tests, an estimate of the operational reliability of the system can be made:

1. *Determine usage patterns* of the system (classes of input and probabilities)
2. *Select or generate test data* corresponding to these patterns
3. *Apply the test cases*, recording execution time to failure
4. Based on a statistically significant number of test runs, *compute reliability*

When to Stop?

When are we done testing? When do we have enough tests?

Cynical Answers (sad but true)

> You're *never done*: each run of the system is a new test
>> — Each bug-fix should be accompanied by a new regression test

> You're done when you are out of time/money
>> — Include testing in the project plan and *do not give in to pressure*
>> — ... in the long run, tests save time

When to Stop? ...

Statistical Testing

> Test until you've reduced the failure rate to fall below the risk threshold

— Testing is like an insurance company calculating risks

Errors per test hour

Execution Time
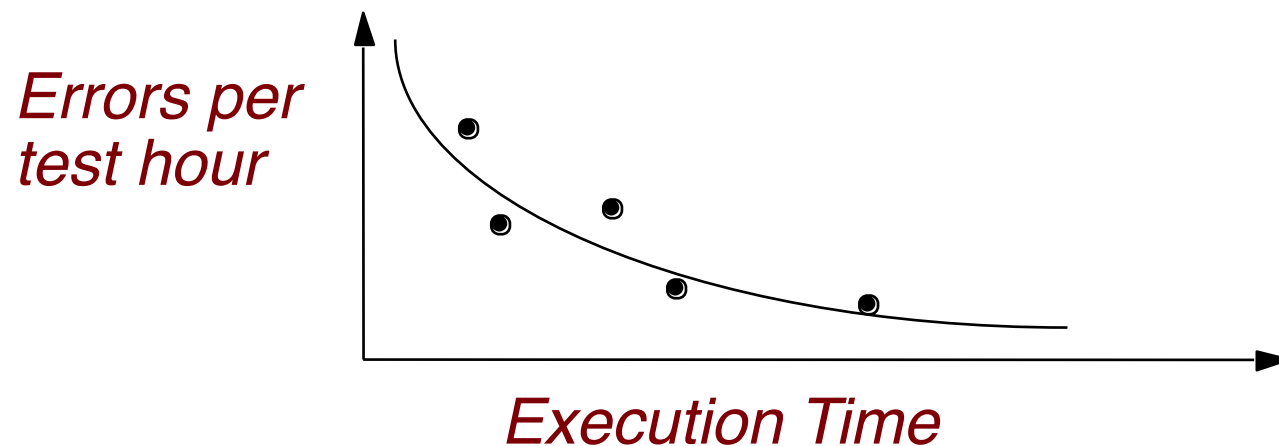
What you should know!

> What is the difference between a failure and a fault?

> What kinds of failure classes are important?

> How can a software system be made fault-tolerant?

> How do assertions help to make software more reliable?

> What are the goals of software validation and verification?

> What is the difference between test cases and test data?

> How can you develop test cases for your programs?

> What is the goal of path testing?

Can you answer the following questions?

> When would you combine top-down testing with bottom-up testing?

> When would you combine black-box testing with white-box testing?

> Is it acceptable to deliver a system that is not 100% reliable?

License