Software Testing Overview

Prof. Lionel Briand
Simula Research Laboratory
Oslo, Norway
briand@simula.no

Tentative Outline

- Class 1
  - – Software Testing Overview part I
  - – White-box Testing techniques
- Class 2
  - – Black-Box Testing techniques
  - – Software Testing Overview part II

Software Testing Overview: Part I

Software has become prevalent in all aspects of our lives

Qualities of Software Products

- Correctness
- Reliability
- Robustness
- Performance
- User Friendliness
- Verifiability
- Maintainability
- Repairability
- Evolvability
- Reusability
- Portability
- Understandability
- Interoperability

Pervasive Problems

- Software is commonly delivered late, way over budget, and of unsatisfactory quality
- Software validation and verification are rarely systematic and are usually not based on sound, well-defined techniques
- Software development processes are commonly unstable and uncontrolled
- Software quality is poorly measured, monitored, and controlled.
- Software failure examples: http://www.cs.bc.edu/~gtan/bug/softwarebug.html

Examples of Software Failures

- Communications: Loss or corruption of communication media, non delivery of data.
- Space Applications: Lost lives, launch delays, e.g., European Ariane 5 shuttle, 1996:
  - From the official disaster report: “Due to a malfunction in the control software, the rocket veered off its flight path 37 seconds after launch.”
- Defense and Warfare: Misidentification of friend or foe.
- Transportation: Deaths, delays, sudden acceleration, inability to brake.
- Electric Power: Death, injuries, power outages, long-term health hazards (radiation).

Examples of Software Failures (cont.)

- Money Management: Fraud, violation of privacy, shutdown of stock exchanges and banks, negative interest rates.
- Control of Elections: Wrong results (intentional or non-intentional).
- Control of Jails: Technology-aided escape attempts and successes, failures in software-controlled locks.
- Law Enforcement: False arrests and imprisonments.

Ariane 5 – ESA

On June 4, 1996, the flight of the Ariane 5 launcher ended in a failure.

Only about 40 seconds after initiation of the flight sequence, at an altitude of about 3,700 m, the launcher veered off its flight path, broke up and exploded.

Ariane 5 – Root Cause

- **Source: ARIANE 5 Flight 501 Failure, Report by the Inquiry Board**

  A program segment for converting a floating point number to a signed 16 bit integer was executed with an input data value outside the range representable by a signed 16-bit integer.

  This run time error (out of range, overflow), which arose in both the active and the backup computers at about the same time, was detected and both computers shut themselves down.

  This resulted in the total loss of attitude control. The Ariane 5 turned uncontrollably and aerodynamic forces broke the vehicle apart.

  This breakup was detected by an on-board monitor which ignited the explosive charges to destroy the vehicle in the air. Ironically, the result of this format conversion was no longer needed after lift off.

Ariane 5 – Lessons Learned

- Adequate exception handling and redundancy strategies (real function of a backup system, degraded modes?)
- Clear, complete, documented specifications (e.g., preconditions, post-conditions)
- But perhaps more importantly: *usage-based testing (based on operational profiles), in this case actual Ariane 5 trajectories*
- Note this was not a complex, computing problem, but a deficiency of the software engineering practices in place …

F-18 crash

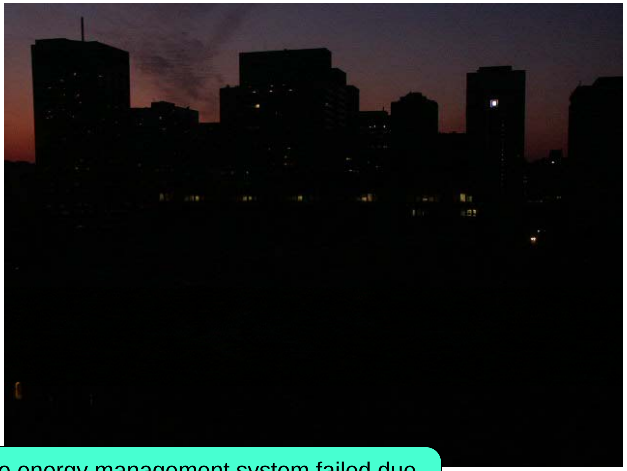
- An F-18 crashed because of a missing exception condition:

  An if … then … block without the else clause that was thought could not possibly arise.
- In simulation, an F-16 program bug caused the virtual plane to flip over whenever it crossed the equator, as a result of a missing minus sign to indicate south latitude.

Fatal Therac-25 Radiation

- In 1986, a man in Texas received between 16,500-25,000 radiations in less than 10 sec, over an area of about 1 cm.
- He lost his left arm, and died of complications 5 months later.

Power Shutdown in 2003



508 generating units and 256 power plants shut down

Affected 10 million people in Ontario, Canada

Affected 40 million people in 8 US states

Financial losses of $6 Billion USD

The alarm system in the energy management system failed due to a software error and operators were not informed of the power overload in the system

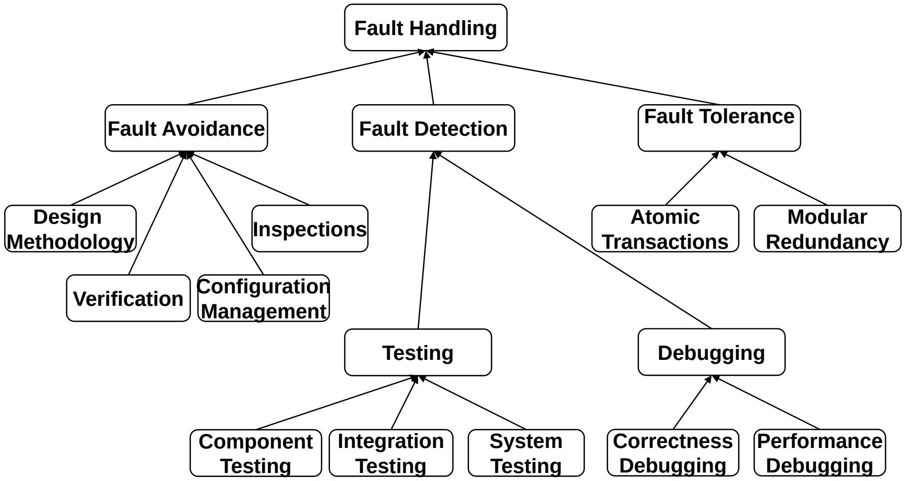
Consequences of Poor Quality

- Standish Group surveyed 350 companies, over 8000 projects, in 1994
- 31% cancelled before completed, 9-16% were delivered within cost and budget
- US study (1995): 81 billion US$ spend per year for failing software development projects
- NIST study (2002): bugs cost $ 59.5 billion a year. Earlier detection could save $22 billion.

Quality Assurance

- Uncover faults in the documents *where they are introduced*, in a systematic way, in order to avoid ripple effects. *Systematic*, *structured reviews* of software documents are referred to as *inspections*.
- Derive, in a *systematic* way, effective test cases to uncover faults
- Automate *testing* and *inspection* activities, to the maximum extent possible
- Monitor and control *quality*, e.g., reliability, maintainability, safety, across *all* project phases and activities
- All this implies the quality *measurement* of SW products and processes

Dealing with SW Faults



Fault Handling

Fault Avoidance

Fault Detection

Fault Tolerance

Design Methodology

Inspections

Verification

Configuration Management

Atomic Transactions

Modular Redundancy

Testing

Debugging

Component Testing

Integration Testing

System Testing

Correctness Debugging

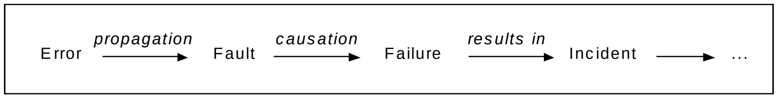
Performance Debugging

Testing Definition

- SW Testing: Techniques to execute programs with the intent of finding as many defects as possible and/or gaining sufficient confidence in the software system under test.
  - "Program testing can show the presence of bugs, never their absence" (Dijkstra)

Basic Testing Definition

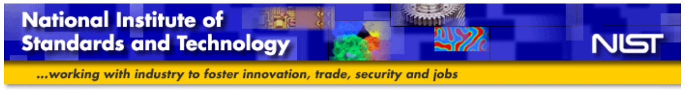
- Errors: People commit errors
- Fault: A fault is the result of an error in the software documentation, code, etc.
- Failure: A failure occurs when a fault executes
- Many people use the above three terms inter-changeably. It should be avoided
- Incident: Consequences of failures – Failure occurrence may or may not be apparent to the user
- The fundamental chain of SW dependability threats:



Error *propagation* → Fault *causation* → Failure *results in* → Incident → ...

Why is SW testing important?

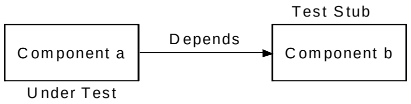
- According to some estimates: ~50% of development costs
- A study by (the American) NIST in 2002:
  - The annual national cost of inadequate testing is as much as $59 Billion US!
  - The report is titled: "The Economic Impacts of Inadequate Infrastructure for Software Testing"


National Institute of Standards and Technology
NIST
...working with industry to foster innovation, trade, security and jobs

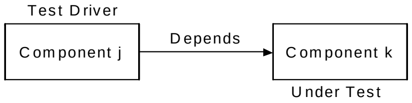
Testing
Definitions & Objectives

Test Stubs and Drivers

- Test Stub: Partial implementation of a component on which a unit under test depends.

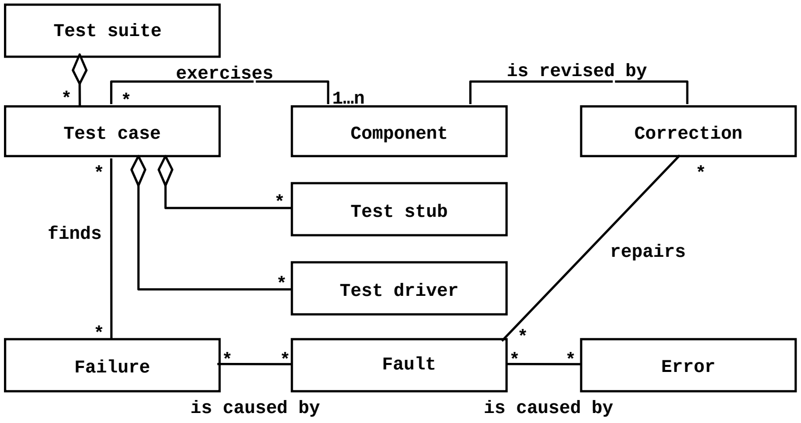

Test Stub
Component a
Depends
Component b
Under Test

- Test Driver: Partial implementation of a component that depends on a unit under test.


Test Driver
Component j
Depends
Component k
Under Test

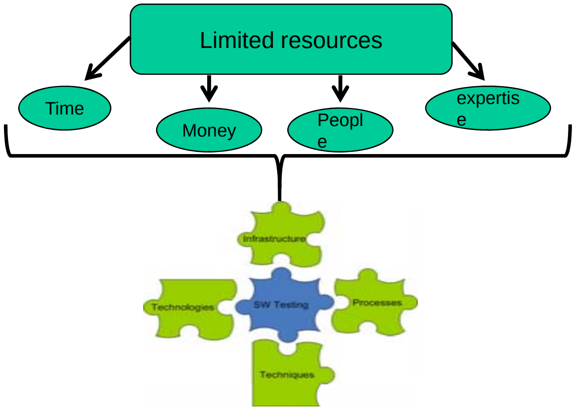
- Test stubs and drivers enable components to be isolated from the rest of the system for testing.

Summary of Definitions

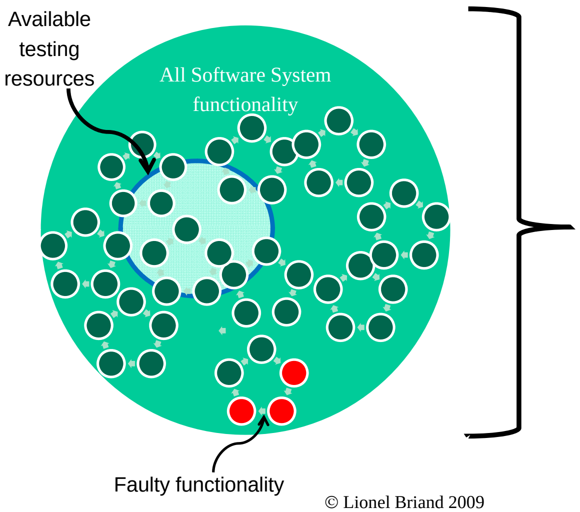

Test suite
exercises
is revised by
*
*
1…n
Test case
Component
Correction
*
*
Test stub
finds
repairs
*
Test driver
*
*
Failure
*
*
Fault
*
*
Error
is caused by
is caused by

Motivations

- No matter how rigorous we are, software is going to be faulty
- Testing represent a substantial percentage of software development costs and time to market
- Impossible to test under all operating conditions – based on incomplete testing, we must gain confidence that the system has the desired behavior
- Testing large systems is complex – it requires strategy and technology- and is often done inefficiently in practice

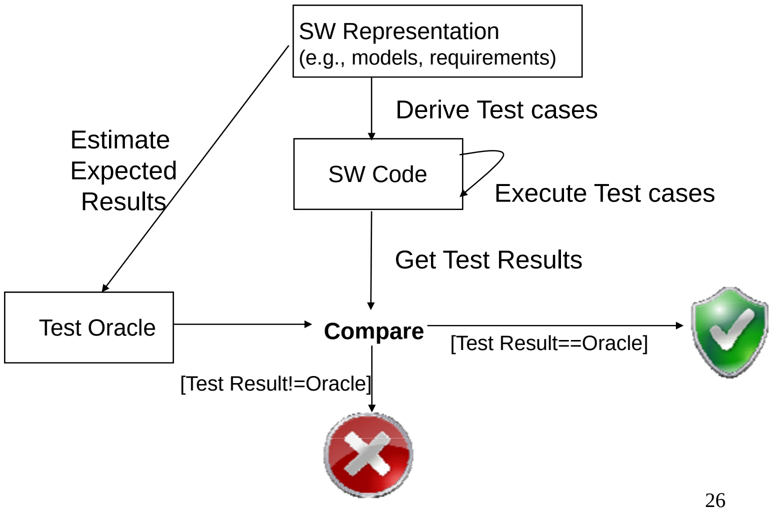

Limited resources
Time
Money
Peopl e
expertis e
Infrastructure
Technologies
SW Testing
Processes
Techniques

The Testing Dilemma


Available testing resources
All Software System functionality
Faulty functionality

Potentially thousands of items to test

Testing Process Overview



SW Representation (e.g., models, requirements)

Derive Test cases

Estimate Expected Results

SW Code

Execute Test cases

Get Test Results

Test Oracle

Compare

[Test Result==Oracle]

[Test Result!=Oracle]

Qualities of Testing

- *Effective* at uncovering faults
- Help *locate* faults for debugging
- *Repeatable* so that a precise understanding of the fault can be gained
- *Automated* so as to lower the cost and timescale
- *Systematic* so as to be predictable in terms of its effect on dependability

Continuity Property

- Problem: Test a bridge ability to sustain a certain weight
- Continuity Property: If a bridge can sustain a weight equal to W1, then it will sustain any weight W2 <= W1
- Essentially, continuity property= small differences in operating conditions should not result in dramatically different behavior

- BUT, the same testing property cannot be applied when testing software, why?
- In software, small differences in operating conditions <u>can</u> result in dramatically different behavior (e.g., value boundaries)
- Thus, the continuity property is not applicable to software

Subtleties of Software Dependability

- *Dependability:* Correctness, reliability, safety, robustness
- A program is correct if it obeys its specification.
- Reliability is a way of statistically approximating correctness.
- Safety implies that the software must always display a safe behavior, under any condition.
- A system is robust if it acts reasonably in severe, unusual or illegal conditions.

Subtleties of Software Dependability II

- *Correct but not safe or robust:* the specification is inadequate
- *Reliable but not correct:* failures rarely happen
- *Safe but not correct:* annoying failures may happen
- *Reliable and robust but not safe:* catastrophic failures are possible

Software Dependability Ex: Traffic Light Controller

• *Correctness, Reliability:*
The system should let traffic pass according to the correct pattern and central scheduling on a continuous basis.
• *Robustness:*
The system should provide degraded functionality in the presence of abnormalities.
• *Safety:*
It should never signal conflicting greens.

An example degraded function: the line to central controlling is cut-off and a default pattern is then used by local controller.

Dependability Needs Vary

- Safety-critical applications
  - flight control systems have strict safety requirements
  - telecommunication systems have strict robustness requirements
- Mass-market products
  - dependability is less important than time to market
- Can vary within the same class of products:
  - reliability and robustness are key issues for multi-user operating systems (e.g., UNIX) less important for single users operating systems (e.g., Windows or MacOS)

Fundamental Principles

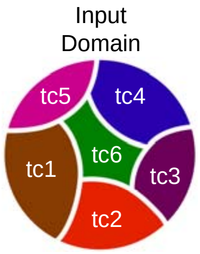
Exhaustive Testing

- Exhaustive testing, i.e., testing a software system using all the possible inputs, is most of the time impossible.
- Examples:
  - A program that computes the factorial function (n!=n.(n-1).(n-2)…1)
    - Exhaustive testing = running the program with 0, 1, 2, …, 100, … as an input!
  - A compiler (e.g., javac)
    - Exhaustive testing = running the (Java) compiler with any possible (Java) program (i.e., source code)

Input Equivalence Classes

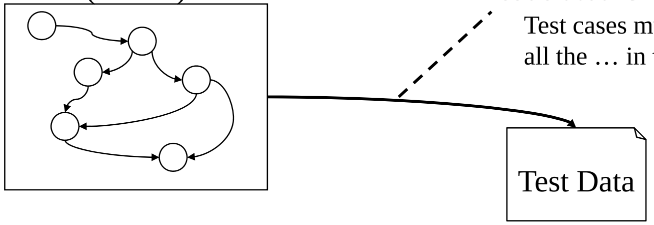
➢ General principle to reduce the number of inputs

- Testing criteria group input elements into (equivalence) classes
- One input in selected in each class (notion of test coverage)



Test Coverage

Software Representation (Model)



Associated Criteria

Test cases must cover all the … in the model

Representation of

- the <u>specification</u> ⇒ Black-Box Testing
- the <u>implementation</u> ⇒ White-Box Testing

Complete Coverage: White-Box

```
if x > y then
   Max := x;
else
   Max :=x ;  // fault!
end if;
```

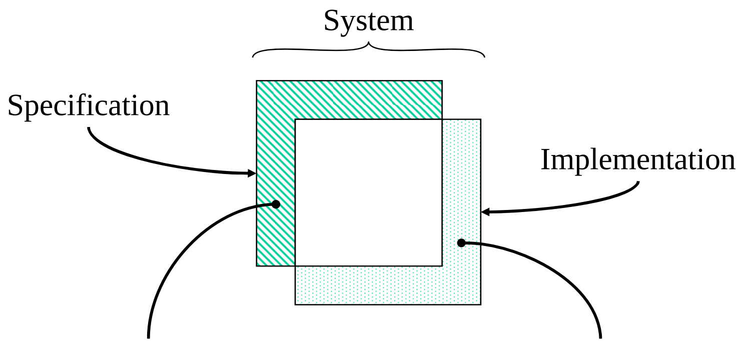
{x=3, y=2; x=2, y=3} can detect the error, more “coverage”
{x=3, y=2; x=4, y=3; x=5, y=1} is larger but cannot detect it

- **Testing criteria group input domain elements into (equivalence) classes (control flow paths here)**
- **Complete coverage attempts to run test cases from each class**

Complete Coverage: Black-Box

- **Specification of Compute Factorial Number:** If the input value n is < 0, then an appropriate error message must be printed. If 0 <= n < 20, then the exact value of n! must be printed. If 20 <= n < 200, then an approximate value of n! must be printed in floating point format, e.g., using some approximate method of numerical calculus. The admissible error is 0.1% of the exact value. Finally, if n>=200, the input can be rejected by printing an appropriate error message.
- Because of expected variations in behavior, it is quite natural to divide the input domain into the classes {n<0}, {0<= n <20}, {20 <= n < 200}, {n >= 200}. We can use one or more test cases from each class in each test set. Correct results from one such test set support the assertion that the program will behave correctly for any other class value, but there is no guarantee!

Black vs. White Box Testing



Missing functionality:
Cannot be revealed by white-box techniques

Unexpected functionality:
Cannot be revealed by black-box techniques

White-box vs. Black-box Testing

- Black box
  - + Check conformance with specifications
  - + It scales up (different techniques at different granularity levels)
  - – It depends on the specification notation and degree of detail
  - – Do not know how much of the system is being tested
  - – What if the software performed some unspecified, undesirable task?

- White box
  - + It allows you to be confident about code coverage of testing
  - + It is based on control or data flow code analysis
  - – It does not scale up (mostly applicable at unit and integration testing levels)
  - – Unlike black-box techniques, it cannot reveal missing functionalities (part of the specification that is not implemented)

Software Testing Overview: Part II

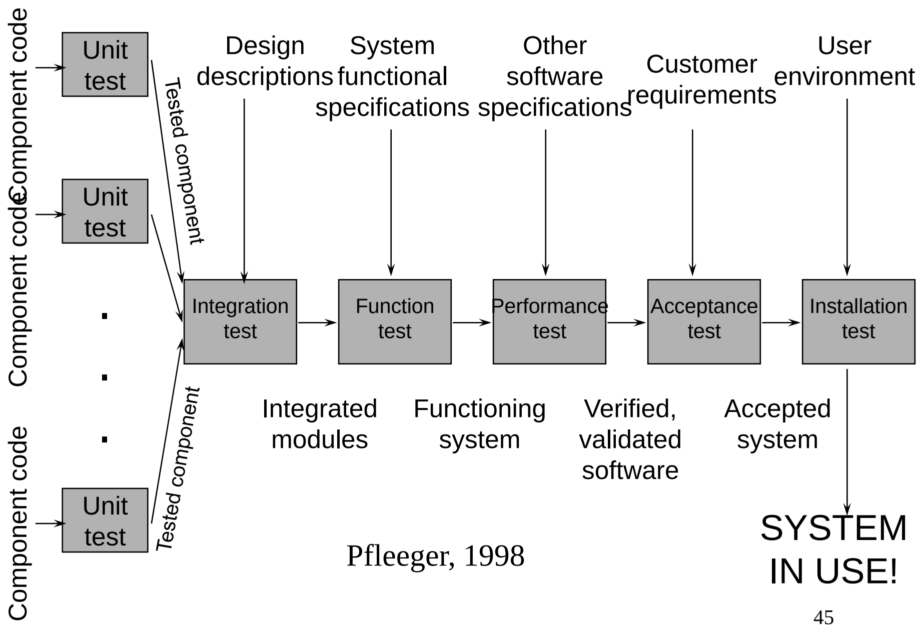
Practical Aspects

Many Causes of Failures

- The specification may be wrong or have a missing requirement
- The specification may contain a requirement that is impossible to implement given the prescribed software and hardware
- The system design may contain a fault
- The program code may be wrong

Test Organization

- May different potential causes of failure, Large systems -> testing involves several stages
- Module, component, or unit testing
- Integration testing
- Function test
- Performance test
- Acceptance test
- Installation test



Component code
Unit test

Component code
Unit test

. . .

Component code
Unit test

Tested component

Tested component

Design descriptions

System functional specifications

Other software specifications

Customer requirements

User environment

Integration test → Function test → Performance test → Acceptance test → Installation test

Integrated modules

Functioning system

Verified, validated software

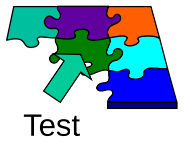
Accepted system

SYSTEM IN USE!

Pfleeger, 1998

Unit Testing

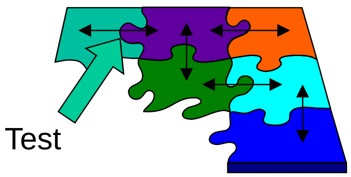
- (Usually) performed by each developer.
- Scope: Ensure that each module (i.e., class, subprogram) has been implemented correctly.
- Often based on White-box testing.



- A unit is the smallest testable part of an application.
- In procedural programming, a unit may be an individual subprogram, function, procedure, etc.
- In object-oriented programming, the smallest unit is a method; which may belong to a base/super class, abstract class or derived/child class.

Integration/Interface Testing

- Performed by a small team.
- *Scope*: Ensure that the interfaces between components (which individual developers could not test) have been implemented correctly, e.g., consistency of parameters, file format



- Test cases have to be planned, documented, and reviewed.
- Performed in a relatively small time-frame

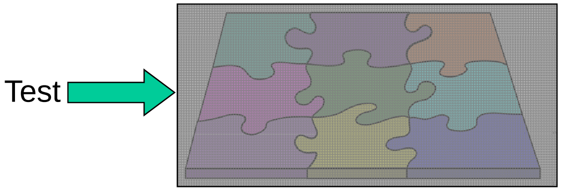
Integration Testing Failures

Integration of well tested components may lead to failure due to:

- Bad use of the interfaces (bad interface specifications / implementation)
- Wrong hypothesis on the behavior/state of related modules (bad functional specification / implementation), e.g., wrong assumption about return value
- Use of poor drivers/stubs: a module may behave correctly with (simple) drivers/stubs, but result in failures when integrated with actual (complex) modules.

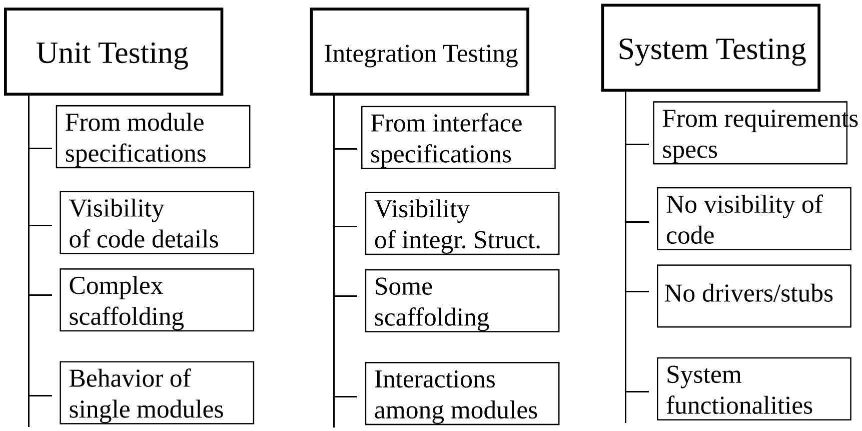
System Testing

- Performed by a separate group within the organization (Most of the times).
- *Scope*: Pretend *we* are the end-users of the product.
- Focus is on functionality, but may also perform many other types of non-functional tests (e.g., recovery, performance).



- Black-box form of testing, but code coverage can be monitored.
- Test case specification driven by system's use-cases.

Differences among Testing Activities


Unit Testing
From module specifications
Visibility of code details
Complex scaffolding
Behavior of single modules
Integration Testing
From interface specifications
Visibility of integr. Struct.
Some scaffolding
Interactions among modules
System Testing
From requirements specs
No visibility of code
No drivers/stubs
System functionalities

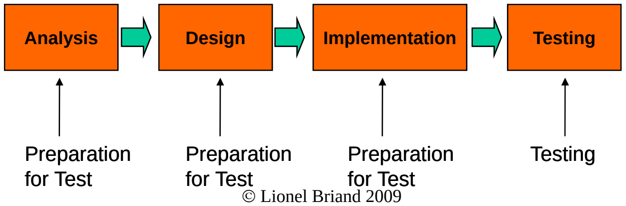
Pezze and Young, 1998

System vs. Acceptance Testing

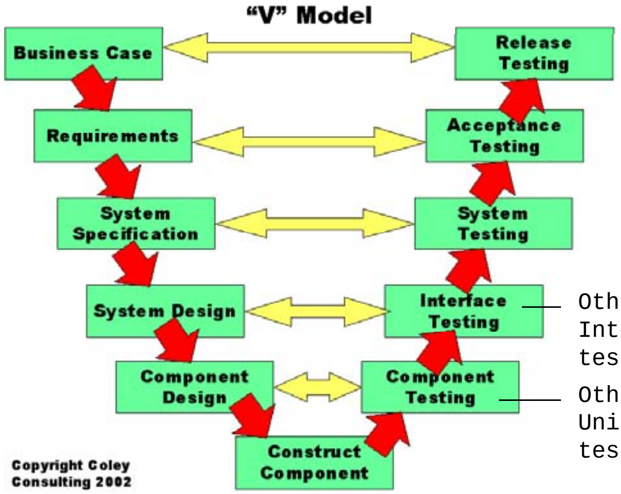
- System testing
  - The software is compared with the requirements specifications (verification)
  - Usually performed by the developers, who know the system
- Acceptance testing
  - The software is compared with the end-user requirements (validation)
  - Usually performed by the customer (buyer), who knows the environment where the system is to be used
  - Sometime distinguished between α - β-testing for general purpose products

Testing through the Lifecycle

- Much of the life-cycle development artifacts provides a rich source of test data
- Identifying test requirements and test cases early helps shorten the development time
- They may help reveal faults
- It may also help identify early low testability specifications or design



Life Cycle Mapping: V Model



"V" Model

Business Case

Release Testing

Requirements

Acceptance Testing

System Specification

System Testing

System Design

Interface Testing

Component Design

Component Testing

Construct Component

Copyright Coley Consulting 2002

Other name: Integration testing

Other name: Unit testing

Testing Activities BEFORE Coding

- Testing is a time consuming activity
- Devising a test strategy and identify the test requirements represent a substantial part of it
- Planning is essential
- Testing activities undergo huge pressure as it is is run towards the end of the project
- In order to shorten time-to-market and ensure a certain level of quality, a lot of QA-related activities (including testing) must take place early in the development life cycle

Testing takes creativity

- Testing often viewed as dirty work (though less and less).
- To develop an effective test, one must have:
  - Detailed understanding of the system
  - Knowledge of the testing techniques
  - Skill to apply these techniques in an effective and efficient manner
- Testing is done best by independent testers
- Programmer often stick to the data set that makes the program work
- A program often does not work when tried by somebody else.