CS446
Software Engineering

Richard Bartels

University of Waterloo
Faculty of Mathematics
Department of Computer Science

June 12, 1996

CS446
Software Engineering
Testing Overview

Richard Bartels

University of Waterloo
Faculty of Mathematics
Department of Computer Science

June 12, 1996

Outline

- Key Words
- Process Overview
- Granularity
- Modes
- Sampling of Tools
- Strategies
- Test Plan Elements
- Tracking and Completion
- Integration
- Components and Selected Details

Key Words

Granularity

Testing will be done at different levels of detail on different scales of components

- **unit**, **module**, **subsystem**, **system**

Planning

Testing may go from smaller scales to larger or the reverse; each will require some utility infrastructure

- **top-down**, **bottom-up**
- **stubs**, **scaffolding**

Aspects

Testing may be directed at the code itself or at executions of it

- **static testing**, **dynamic texting**

Static Modes

The code may be assessed mechanically or humanly at different levels of rigor

- **syntax checking**, **strong typing**, **formal systems**
- **inspections**, **clean room development**

Key Words

Dynamic Modes

Dynamic testing may concentrate on the interface or on internals; it may cover normal behavior or response to extremes

- **black box**, **white box**
- **standard tests**, **stress tests**

Environments

Code execution may proceed under special running conditions that monitor behavior

- **assertions**, **leak detection**, **profiling**, **...**

Evaluation

Specific test results will generate specific corrective actions; aggregate test results will provide tracking and management

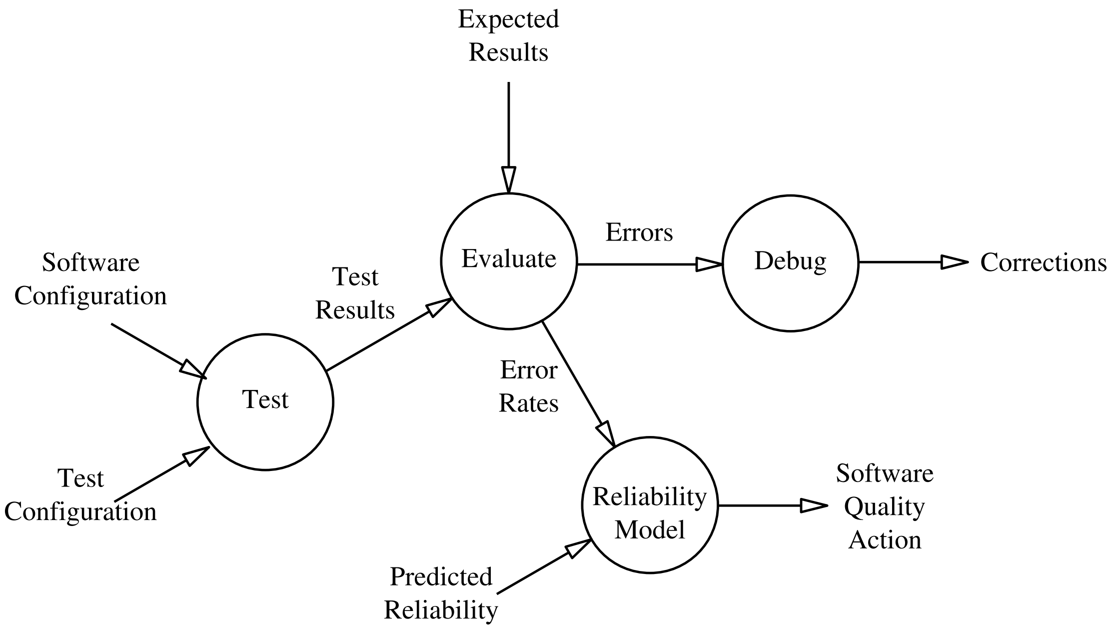
- **debugging**
- **software quality assurance**, **reliability assessment**

Overview

Testing Objectives

- Testing is a process of executing a program unit (i.e. module, procedure, code section, ...) with the intent of finding errors
- A good test is one that has a high probability of finding an error; a successful test is one that uncovers an as yet undiscovered error; a test that finds no error is a test that provides no information
- Complete testing constitutes exercising all representative states of the program unit (i.e. independent paths, boundary conditions, average cases, ...)

Test Information Flow



Granularity

Unit Testing

- Testing at the level of individual components (procedures, functions, classes, abstract data types ...) whose encapsulation permits independent testing

Module Testing

- Testing of fundamental cooperating components (class clusters, modules, ...) that can, within reason, be addressed independently of other system components

Sub-system Testing

- Testing the integration with and interactions between various modules that form separatable foundations in the system's design

System Testing

- Integration and interaction testing during the assembly of the whole system from its subsystems; starting with simulated use tests and ending with alpha testing

Acceptance Testing

- Testing in actual use: beta testing and/or evaluative delivery to the customer

Testing Modes

Black Box Testing

- Tests at the interface of a program unit
- Tests determining whether each desired functionality of a unit is fully operational
- Tests that do not focus on the details of implementation
  - Note: interface/interaction must be capable of testing through scripts to ensure control and reproducability
  - Hit boundary cases as well as representative cases
  - Look for equivalence classes of tests

White Box Testing

- Tests of the implementation logic
- Tests of all internal components and all control paths
- Generally impossible, so importance testing is used
  - Hit all independent paths
  - Hit all decisions on true and false side
  - Hit all loops at boundaries and average cases
  - Hit all data structures at boundaries and for average cases

Automated Testing Tools

A Sampling:

- Static analyzer
  Check code conformance to structure and format
- Code auditor
  Check code conformance to style and standards
- Assertion processors
  Check pre/postcondition assertions
- Test file generators
  Generate input files according to high-level specification
- Test data generators
  Path tracers that assist in choosing test data
- Test verifiers
  Measure test coverage and compare with expectation
- Test harnesses
  Scaffold environments that assist in unit testing
- Output comparators
  Compare results with predetermined or historical ones
- Symbolic execution
  Exercise code with symbolic input and interpreter
- Environment simulators
  Model real-time operating conditions
- Data flow analyzers
  Track data for find abnormal uses and references

Testing Strategies

Psychological/sociological issues

- Developers test vs. Independent Test Group
  - Constructive vs. destructive attitudes
  - Test to show things work vs. tests to reveal errors
  - Egos and blind spots

Technical issues

- Unit testing: each unit alone, white box focus
- Integration testing: unit-unit interaction, black box focus
- Validation testing: integrated clusters, specification checks
- System testing: full integration, all specifications
- Beta testing: customer trials
- Stress testing: push beyond limits, check graceful death
- Regression testing: verify past testing after new testing begun

Test Plan Elements

Process outline

- A description of the major testing phases and their sequencing

Requirements traceability

- A clear documentation of when and how satisfaction of all requirements will be verified

Tested items

- A list of the software products to be developed and tested (code, documentation, scaffolding, customer training, ...)

Testing schedule

- An overall schedule with attention to the testing itself, the timing, and the resources allocated

Test recording procedures

- Procedures for keeping records systematically and preserving test cases and test automation tools

Hardware, human, and software requirements

- Setting out and corporate or team utilities and skills that may have to be scheduled or acquired

Constraints, risks, fallbacks

- An anticipation of critical issues and an outline plan of contingencies

Quality assurance management

- A layout of the checkpointing, monitoring, review, and auditing processes to be used

Completion of Testing

When to quit

- Some probabilistic model of "error survival"
  - What is "error half life" under a testing strategy?
- Example: logarithmic Poisson

  $f(t) = \left(\frac{1}{p}\right) \ln(i_0 pt + 1)$

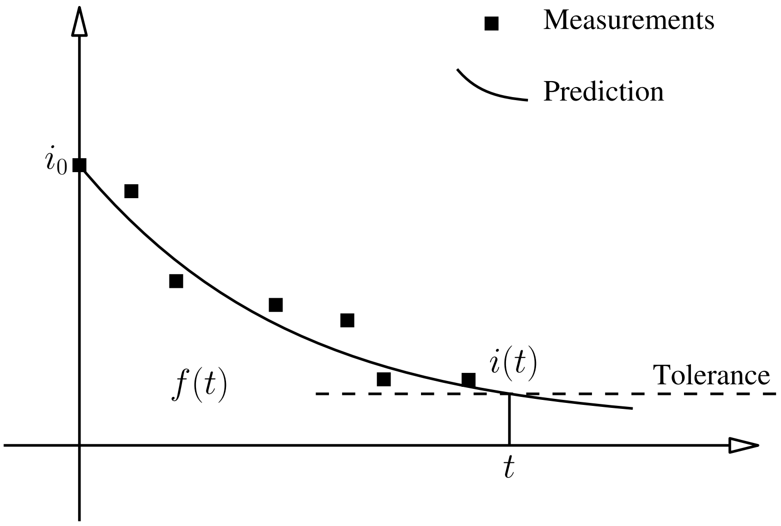
  $i(t) = \frac{i_0}{i_0 pt+1}$

  $i(t)$: instantaneous failure intensity (failures per unit time)

  $f(t)$: cumulative number of failures

  i_0: initial failure intensity (measured)

  p: reduction rate (decay rate) in failure intensity



Calibrating the Poisson Model

Take measurements:

time unit	**bugs found**
$t_0 = 0$	
$t_0 \to t_1 = t_0 + \Delta t$	i_0
$t_1 \to t_2 = t_1 + \Delta t$	i_1
$t_2 \to t_3 = t_2 + \Delta t$	i_2
$\vdots$	$\vdots$

Formula:

$$\frac{i_0}{i_0 pt + 1} = i(t) \to [i(t) i_0 t]p - [i_0 - i(t)] = 0$$

Residuals:

$$\begin{gathered}(i_1 i_0 t_1)p - (i_0 - i_1)\\(i_2 i_0 t_2)p - (i_0 - i_2)\\(i_3 i_0 t_3)p - (i_0 - i_3)\\\vdots\end{gathered}$$

Linear regression:

$$\min \sum_{j=1}^{k} \left[(i_j i_0 t_j)p - (i_0 - i_j)\right]^2$$

Unit Tests

Unit: The smallest encapsulated entity that can reasonably be isolated and executed

Unit testing is a disciplined version of the debugging and development we are all used to carrying out in our work

The components of unit testing are: functional (interface) testing, limit (boundary) testing, path (structure) testing, internal (state) monitoring, error handling tests, stress testing. Such testing usually requires that drivers and stubs be designed.

Components

- Interface check: average case in/out/actions correct
- Boundary checks: data, loops, conditions
- Structure checks: all basis (independent) paths
- State monitoring: selected or all transitions reported
- Error handling: abnormal paths/cases/data intercepted
- Stress testing: beyond limits (unexpected/unreasonable cases)
- Driver/stub: design and construction necessary

Black Box Testing

Boundary value analysis: input side

- For each parameter or input category there are clear extremes ($0 \leq x \leq M$). One test should be included at each extreme as well as neighboring each extreme on both sides ($x = -1$, $x = 0$, $x = 1$, $x = M$, $x = M$, $x = M + 1$)

Boundary value analysis: output side

- For each output parameter or category there may also be clear extremes (maximum UI deduction). Input data should be prepared whose output will be at and (to the extent possible) neighboring these extremes

Equivalence testing

- The normal ranges of input ($0 \ll x \ll M$, $0 \ll$ UI deduction $\ll$ UI max may have predictable, average cases; if not, mid-range cases chosen at random (possible according to a predicted distribution) will have to serve. Such cases serve as experimental probes on the quality of the software.

State Monitoring

- For each normal invocation of the test object there may be an expected change of state (adjustment in the member data that is retained past the invocation). Equivalence classes of input must be designed to exercise all categories of state change, and some mechanism must be acquired or provided to ensure that this (normally hidden) state information is reported.

Testing Procedures: Integration

Top down

- Main module is driver; direct subsidiaries are stubs
- Stub replaced by unit; its subsidiaries are stubs
- Regression testing (re-exicute prior tests)

Bottom up

- Low-level modules combined into clusters ("builds")
- Create a driver and any necessary stubs
- Test the cluster
- Combine clusters with others, forming new builds
- Regression testing

Progressive Development

Stub progression

- Stages of functionality
  - Display a trace ("X called")
  - Display arguments ("X called with y")
  - Generate a value (return z)
  - Return a value (look up y in a table, return z)
  - Replace with actual routine

Driver progression

- Stages of functionality
  - Call trace stubs along a fixed path
  - Call value-generator stubs to test branches/computations
  - Call look-up stubs giving predetermined arguments
  - Fill in real code

Scaffolding

- Build a driver/stub system and fill in pieces
  - Integration mix: top-down/bottom-up
    depends on completeness of stub population

A Bad Thing

No integration; units thrown together for final system assembly