ECE 322
Software Testing and Maintenance

Fall 2025

ECE 322
Software Testing and Maintenance

Witold Pedrycz, PhD, P.Eng.,

office: Donadeo Innovation Centre for Engineering (D-ICE),
11th floor, room 11-293

email: wpedrycz@ualberta.ca

CONTACT INFORMATION (email, zoom)

Suggested:
Monday, Tuesday, Wednesday, Thursday, Friday 11:00 AM – 12:00 noon
any time

Organizational issues (1)

All course material on canvas system
Grading scheme

Final examination: **50**%

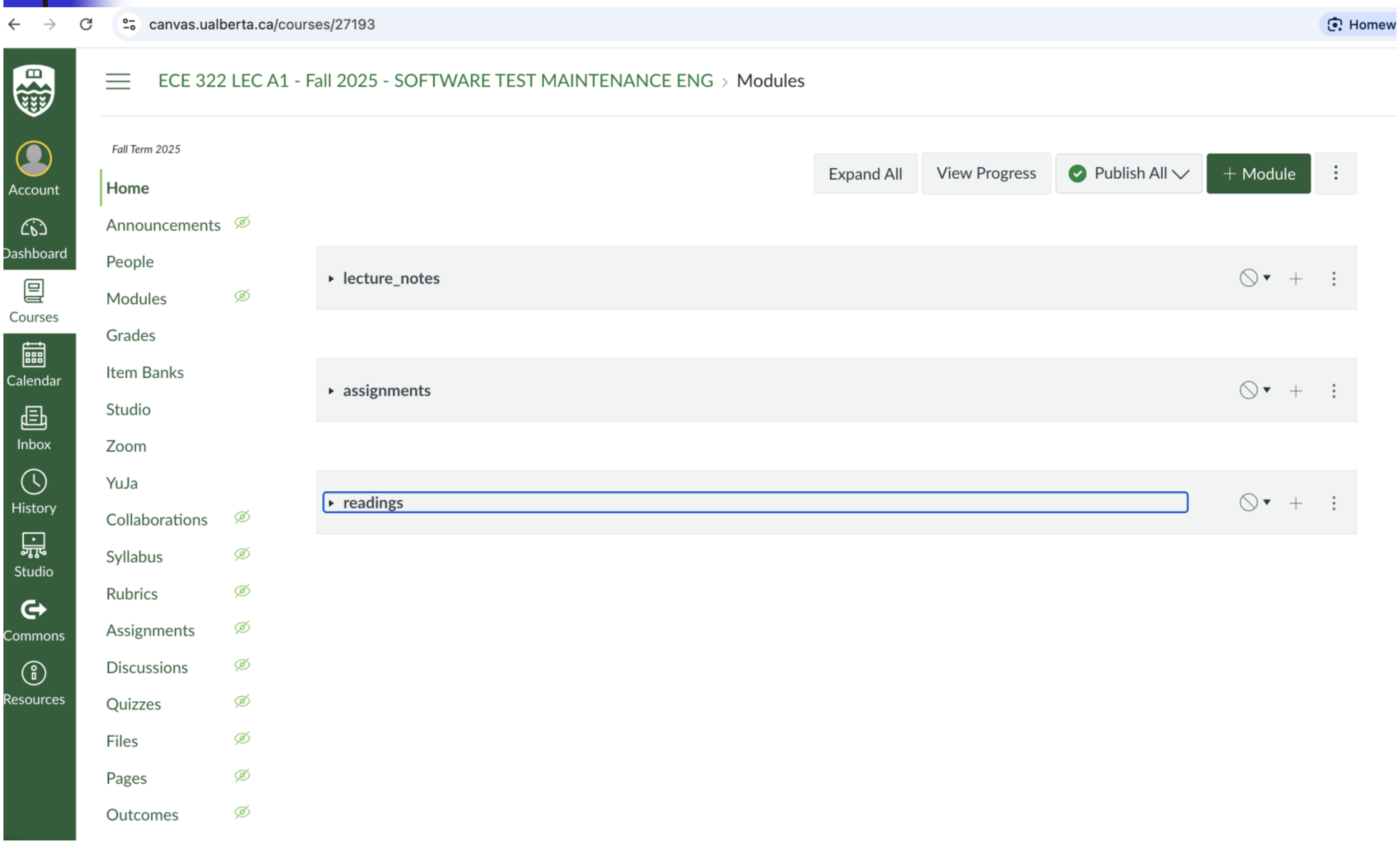
Assignments (6 assignments; e-submission): **6**%

Assignment #1	posted: September 8, 2025; due date: September 15, 2025
Assignment #2	posted: September 22, 2025; due date: September 29, 2025
Assignment #3	posted: October 6, 2025; due date: October 14,2025
Assignment #4	posted: October 20, 2025; due date: October 27,2025
Assignment #5	posted: November 3, 2025; due date: November 17, 2025
Assignment #6	posted: November 24, 2025; due date: December 1, 2025

Midterm examination: **10%** Friday, October 31, 2025
Project (LLM-oriented): **9%**
Laboratory experiments: **25**%

Active participation and feedback strongly encouraged!

Organizational issues (2)



Organizational issues (3)

Course material (lecture notes, assignments, lab notes…)
on the Web: e-class ECE 322

Lab

Yuan Feng(LI) yfeng3@ualberta.ca
Jessie Chen (Technician) xchen3@ualberta.ca

Lectures (Assignments)

Baizhen Li (TA) baizhen1@ualberta.ca

Course outline

Introduction: organizational issues, discipline of Software Engineering, software failures, software testing

Software testing – definitions, test plan, coverage, verification and validation, taxonomy of testing approaches. Software development vs. software testing, software maintenance

Software maintenance, testing principles, software quality assurance, errors-faults-failures

Software requirements engineering: a concise and focused review

Black box testing: concept, error guessing, checklists, partition-based testing, equivalence classes, testing strategies, examples, boundary value analysis, random testing, operational profiles, input domain testing, extreme point combination, weak *n* x1 testing strategy, decision tables, syntax-driven testing, Black box testing with finite state machines, coverage criteria, combinatorial testing, functional testing, test patterns

White box testing: control flow graphs, control flow coverage testing (statement coverage, branch coverage, path coverage), complexity measure, data flow testing

Fault injection and fault seeding, mutation testing

Testing oracles

Integration testing

Regression testing

Software maintenance: software evolution, types of systems (S, P, E), software maintainability software maintenance process, system re-engineering, maintenance metrics (process and product)

Artificial Intelligence/Machine Learning, Large Language Models (LLMs) and software testing

Software reliability: software vs. hardware, overview of reliability models, models of software reliability (time-dependent and time-independent)

Textbook

P. Ammann and J. Offutt,

Introduction to Software Testing

Cambridge University Press, 2018

Roadmap of Software Courses

ECE 322
Software Testing & Maintenance

Software Requirements Engineering

Introduction to Software Engineering

Discipline of Software Engineering (1)

60ties

Complexities, paradoxes and anomalies of software engineering
Unchartered territories: high risk, unpredictable outcomes, costs, schedule overruns
Assembly languages, Fortran, Cobol, Algol

70ties

Belief that software engineering problems are of *technical* nature and could be resolved through techniques of specification, design, and verification
structured design, analysis, programming
C, Pascal

Discipline of Software Engineering (2)

80ties

Knowledge-based software engineering
Problems of *managerial* and *organizational* nature
Fifth Generation Computing initiative
Prolog, Lisp, Ada

90ties

Reuse in software engineering; not studied in the past
Emergence of of object-oriented programming supporting
bottom-up design discipline supporting reuse
C, C++, Eiffel. Smalltalk

Ultra Large Scale systems

Software Engineering Institute,
Carnegie Mellon University

Decentralization
Conflicting, unknown and diverse requirements
Continuous evolution and deployment
Heterogeneous, inconsistent, changing elements
Deep erosion of people-system boundary
Failure is normal and frequent

Software Testing and Maintenance: An Introduction

Software Failures

Patriot-Scud
(rounding error, 1991)

Ariane 5
(data conversion of a too large number, 1996)

Mars Climate Orbiters
(Mixture of pounds and kilograms, 1999)

Software Failures- Ariane 5

Programming : incorrectly handled software exception resulted from a data conversion of a 64 bit floating point to a 16-bit signed integer value

Design : system specs account for hardware not software failures

Requirements: incorrect analysis of changing requirements (not needed; left operational in Ariane 5 without satisfying traceable requirements)

Testing: inadequate testing

Project managing: ineffective development and project management

Software Failures

NASA Mariner 1 , Venus probe
(period instead of comma in FORTRAN DO-Loop, 1962)

Purpose: to relay signals from the Mars Polar Lander once it reached the surface of the planet

Disaster: smashed into the planet instead of reaching a safe orbit

$165M

Shooting down Airbus 320 (1988)

US Vicennes shot down Airbus 320

Mistook Airbus 320 for a F-14, 290 people dead

Why: Software error - cryptic and misleading output displayed by the tracking software

Therac-25 Radiation Therapy

THERAC-25, a computer controlled radiation-therapy machine

1986: two cancer patients at the East Texas Cancer Center in Tyler received fatal radiation overdoses

Software failure - mishandled race condition (i.e., miscoordination between concurrent tasks)

London Ambulance Service (1992)

London Ambulance Service Computer Aided Dispatch (LASCAD)

Purpose: automate many of the human-intensive processes of manual dispatch systems associated with ambulance services in the UK

Functions: Call taking

Failure of the London Ambulance Service on 26 and 27 November 1992

London Ambulance Service (1992)

Load increased
• Emergencies accumulated

• System made incorrect allocations
– more than one ambulance being sent to the same incident
– the closest vehicle was not chosen for the emergency

• At 23:00 on November 28 the LAS eventually instigated a backup procedure, after the death of at least 20 patients

Other examples

– British destroyer H.M.S. Sheffield; sunk in the Falkland Islands war; ship's radar warning system software allowed missile to reach its target

– An Air New Zealand airliner crashed into an Antarctic mountain

– North American Aerospace Defense Command reported that the U.S. was under missile attack; traced to faulty computer software - generated incorrect signals

– Manned space capsule Gemini V missed its landing point by 100 miles; software ignored the motion of the earth around the sun

Software Failures

AT&T long distance service fails for nine hours
(Wrong BREAK statement in C-Code, 1990)

Phobos 1, Russian Mars Probe
(Wrong command leads to rotation, 1988)

Vancouver Stock Exchange Index
(Rounding Error, 1983)

Software Failures

Automated baggage sorting system of a major airport in February 2008 prevented thousands of passengers from checking baggage for their flights. It was reported that the breakdown occurred during a software upgrade, despite pre-testing of the software.

Tens of thousands of medical devices were recalled in March of 2007
The software would not reliably indicate when available power to the device was too low.

In 2005 a new government welfare management system in Canada costing several hundred million dollars was unable to handle a simple benefits rate increase after being put into operation. Reportedly the original contract allowed for only 6 weeks of acceptance testing and the system was never tested for its ability to handle a rate increase.

Top 5 Software Failures in 2015 (1)

Software Glitch Causes F-35 to Detect Targets Incorrectly

A serious software glitch in the F-35 Joint Strike Fighter air crafts gathered wide public attention in the month of March this year. The planes when flying in formation were unable to detect potential targets from different angles. In fact the engineers identified that the software bug caused the aircraft to detect targets incorrectly. The sensors on the plane were unable to distinguish between isolated and multiple threats. As reported by Fox News, Air Force Lt. Gen. Christopher Bogdan, Program Executive Officer, F-35 said: "We want to fix this so it is inherent in the airplane. We have always said that fusion was going to be tough. We are going to work through this." (Source: Fox News)

Nissan's Airbag Software Malfunction

Nissan Motors has been under investigation by US safety regulators for recalling over one million vehicles in the past two years. The vehicles were recalled due to a software failure in the airbag sensory system. The automakers have reasoned that a software glitch in the system rendered it incapable of detecting an adult sitting in the passenger seat. The issue surfaced when two accidents took place and the airbags did not inflate. Several complaints were registered even after the issue was supposedly resolved. Gorge Zack in his article in bidnessetc.com even mentioned that 104,871 vehicles had been recalled by Honda Motors as well due to faulty airbag systems made by Takata. (Source: BIDNESSETC)

Software Failures in 2017

Suncorp Bank – Vanishing cash

In February of this year, a malfunction during a routine upgrade caused the disappearance of money from customers' bank accounts. Additional customer complaints included overdrawn and locked out accounts.

Dodge Ram – 1.25 million recalls

A major software glitch that could cause the airbags and seatbelts in Ram trucks to fail during rollover collisions caused Dodge to recall more that 1.25 million trucks. To prevent the problem, the FCA must now reprogram the onboard sensor of every impacted vehicle.

Software failures in 2018

Airline industry
Crew scheduling and tracking system
AC, air traffic control Brussels

Automakers
Recalls

Communications
Outages

Cyber crime
Software vulnerabilities

Financial institutions

Software failures in 2020

Airline industry

Heathrow airport disruption

critical flaw in the Zoom meetings client software recalls

enabled a remote hacker to control any PC running Windows 7 or earlier.
“Zoom security issues were a notable software failure in 2020,”

Software Failures in 2021

T-Mobile data breach affects 50 million customers

TikTok glitch resets followers to zero

Colonial Pipeline’s costly ransomware attack

NHS 4-hour outage leaves passengers stranded (application/website; covid vaccination status)

Tesla recalls almost 12,000 vehicles (forward collision threat)

Software Failures in 2024

Digital Crash July 2024

update released by **CrowdStrike**, cybersecurity company.
The update - intended to fix a minor vulnerability in the Falcon security software

fatal error in the Windows operating system (kernel level)

update set to auto-install on all devices that had the Falcon software, without giving the users any option to decline or postpone it. Content-product update

Windows allows third-party software to operate at a kernel level- direct access to the core functions of the system.

Software Failures in 2025

Feb, 2025 - catastrophic failure of ChatGPT, memory implosion, mass erasure

July 2025- UK air traffic control, flights delays and cancellations, failure of radar system, similar in August 2023

Untested Edge cases
The radar system crashed when it encountered rare but realistic input scenarios that weren't part of the test coverage. These edge conditions should have been identified and simulated during QA.

Poor Regression Control
Known bugs that were previously fixed resurfaced because regression tests weren't comprehensive. Patches were deployed without validating against historical failures, allowing issues to slip back in.

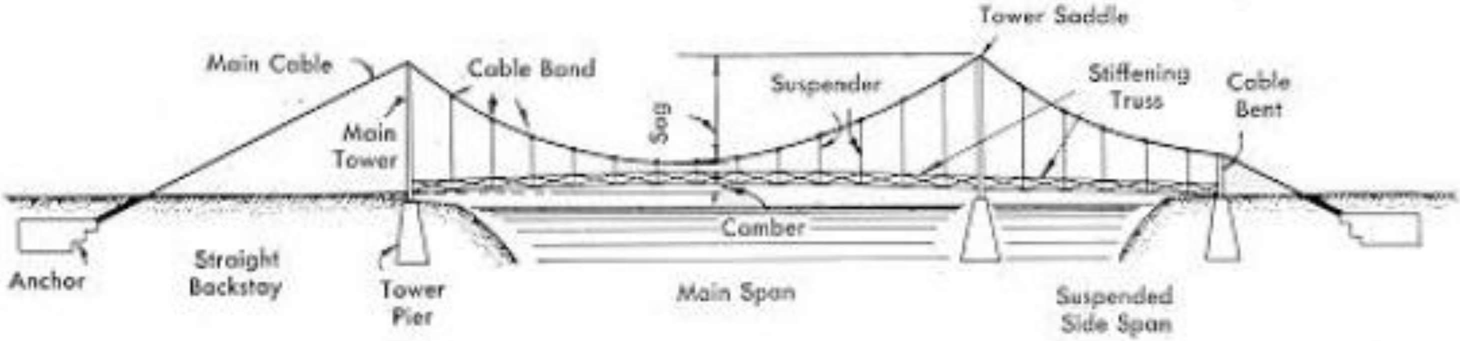
No Chaos or Stress Testing
Backup systems existed but weren't tested under real-world load. When the primary system failed, the backups couldn't handle production-level traffic and failed to take over effectively.

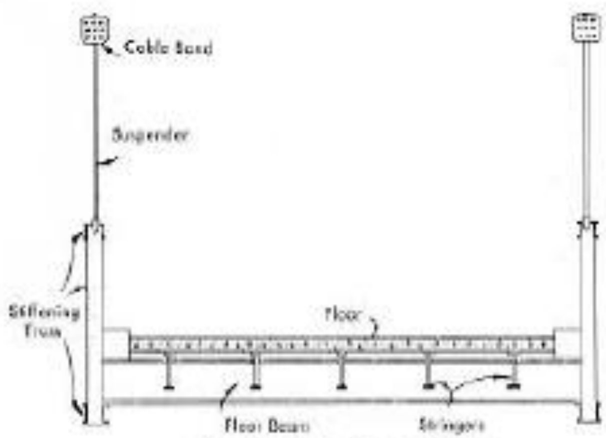
17 fatalities, 736 crashes: The shocking toll of Tesla's Autopilot

Tesla's driver-assistance system, known as Autopilot, has been involved in far more crashes than previously reported

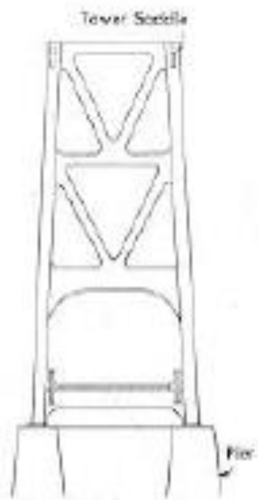
By Faiz Siddiqui and Jeremy B. Merrill
June 10, 2023 at 7:00 a.m. EDT

Software and Physical Systems





Cross-Section in Suspended Span



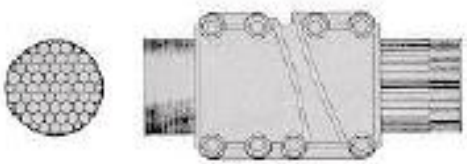
Cross-Section at Tower

Galvanized Bridge Wire for Parallel Wire Bridge Cables. Recommended diameter .196 inch.

Galvanized Bridge Strand--consists of several bridge wires, of various diameters twisted together.

Galvanized Bridge Rope--consists of six strands twisted around a strand core.

Parallel Wire Cable



Detail of Main Cable and Cable Band. The wrapping wire is omitted at the right for clarity. Note the closed construction and aluminum fillers.

Approxima

Known values (same for all tri

Unstressed length (U.L.)

(Σ U.L. =)

w_c (cable weight per foot)

AE

K

B

$\frac{B}{K} = \text{Tan}\ \alpha$

Sec α

Try H_T

$C = \frac{H_T}{w_c}$

$2C$

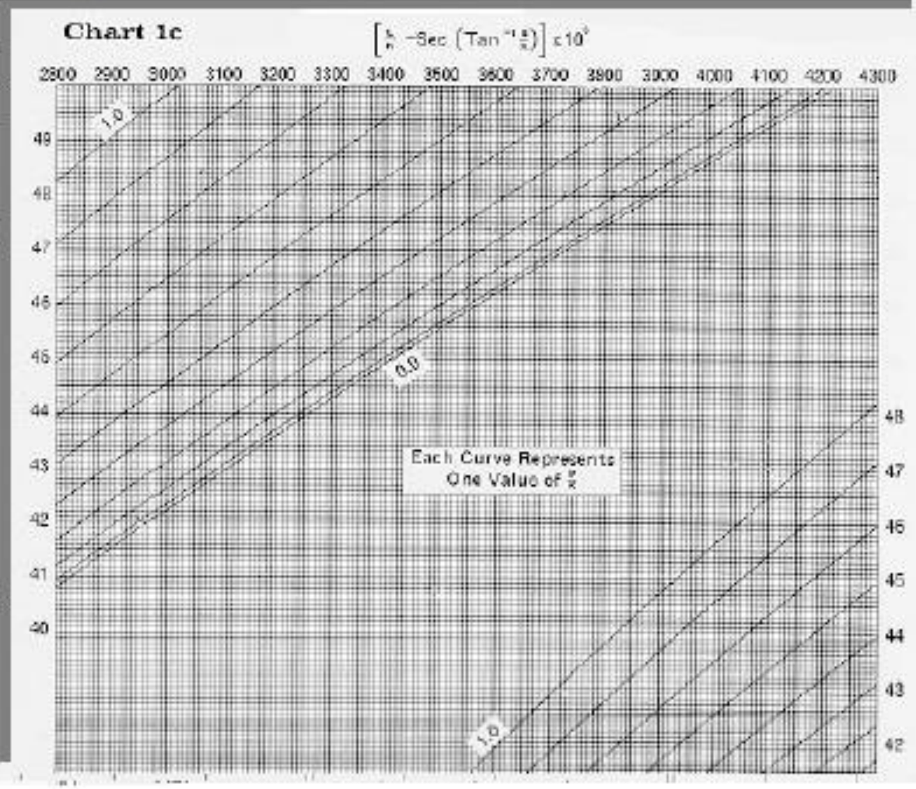
$\frac{K}{2C}$

$\frac{L}{K}$ (from charts 1a, 1b and 1

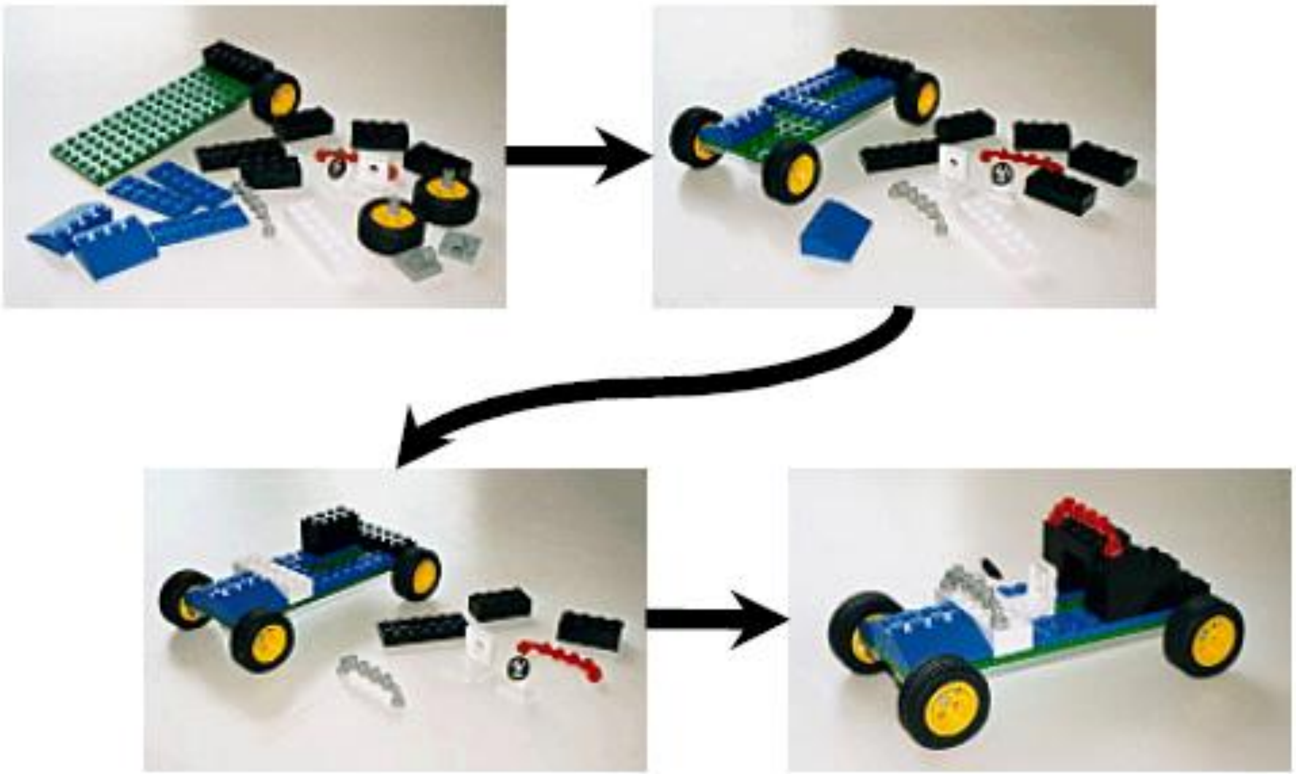
$\frac{H_T}{AE}$

$\frac{L'}{K}$

$L = K + k\left(\frac{L}{K} - 1\right)$



How Cars Are REALLY Engineered (A Detailed View)



How cars are developed

User requirements
– Engine power, all-wheel, seating, comfort, MP3 player
• Detailed design
– Blueprints, design documents
• Verify design
– Simulation, prototyping
• Develop parts (components)
– Test each component
– Components may be reused
– Mass produced
• Assemble the car
– Test the car (Front/side crash tests, Stability tests)
– Usability testing (Feedback from drivers/passengers)

Software and cars

"If the automobile industry had developed like the software industry, we would all be driving $25 cars that get 1,000 miles to the gallon."

"Yeah, and if cars were like software, they would crash twice a day for no reason, and when you called for service, they'd tell you to reinstall the engine."

Software and systems governed by laws of physics

Lack of "continuity"

Bridge designed for 1,000 tons could withstand 1,001 tons

Software with 105 lines of code could not function if one line is altered or removed

Levels of abstraction

Essential difficulties of Software Engineering

Complexity

Conformity - no unifying principles; software conforms to existing reality, systems, interfaces.

Changeability – constantly subject to pressure for change. Successful software gets changed (users-new functionality, physical platform)

Invisibility - software is invisible and unvisualizable (no geometric abstractions)

F. P. Brooks, No silver bullet, *IEEE Computer*, April 1987

"If in physics there's something you don't understand, you can always hide behind the uncharted depths of nature. You can always blame God. You didn't make it so complex yourself.

But if your program doesn't work, there is no one to hide behind. You cannot hide behind an obstinate nature. If it doesn't work, you've messed up."

E.W. Dijkstra

Some basic arithmetic

Can't we expect software to execute correctly?

- Carefully developed programs
  - 5 faults/1,000 LOC
  - 1M LOC will have 5,000 faults

- Windows XP has 45M LOC
  - How many faults?
  - 45 x 5,000 = 225,000

- Why not remove the faults?

Assurance of software quality and software testing

SOFTWARE TESTING

is one of the essential means to accomplish a high level of SOFTWARE QUALITY

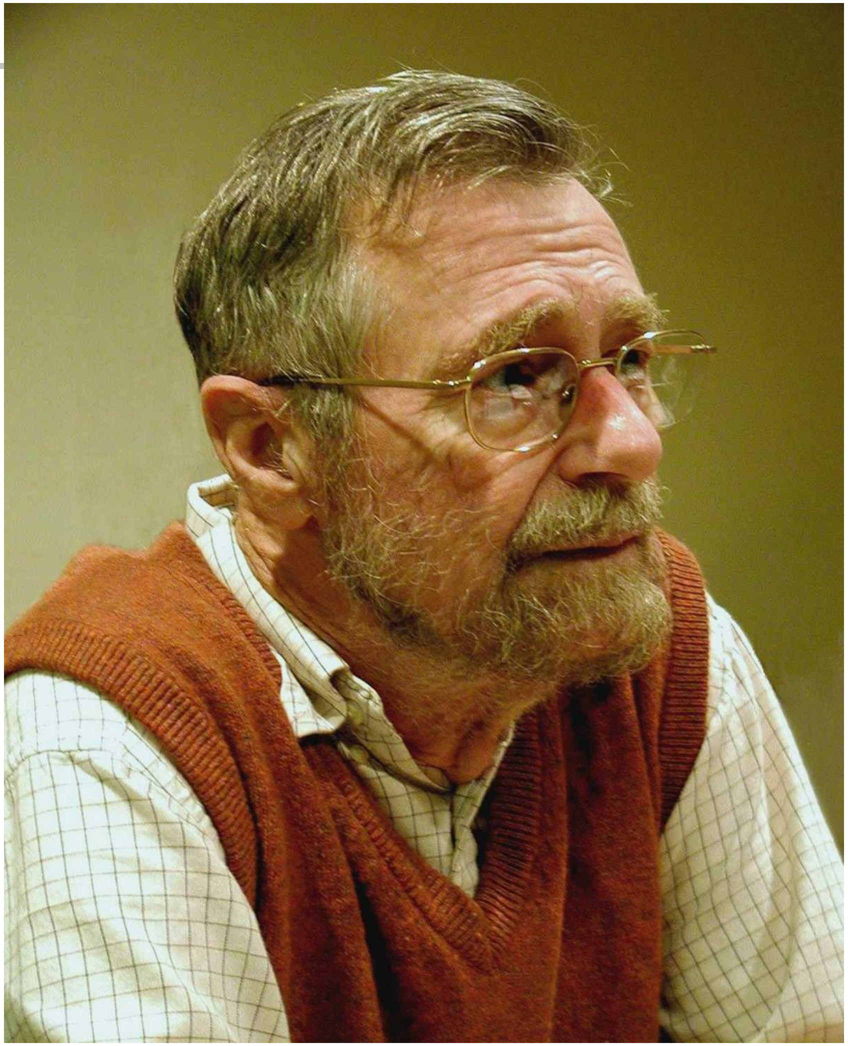
Testing:
defining the process

Testing is the process of executing a program with the intent of finding errors

The Peculiarity of Testing

- Testing is the process of executing a program with the intent of **finding an error**
- A **good** test case has a high probability of finding an as-yet **undiscovered error**
- A **successful test** is one that uncovers an as-yet **undiscovered error**

Software Testing
E W Dijkstra



Software Testing

Program testing can be used to show the presence of bugs, but never to show their absence!

Source: Notes On Structured Programming, 1970,

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

Source: The Humble Programmer

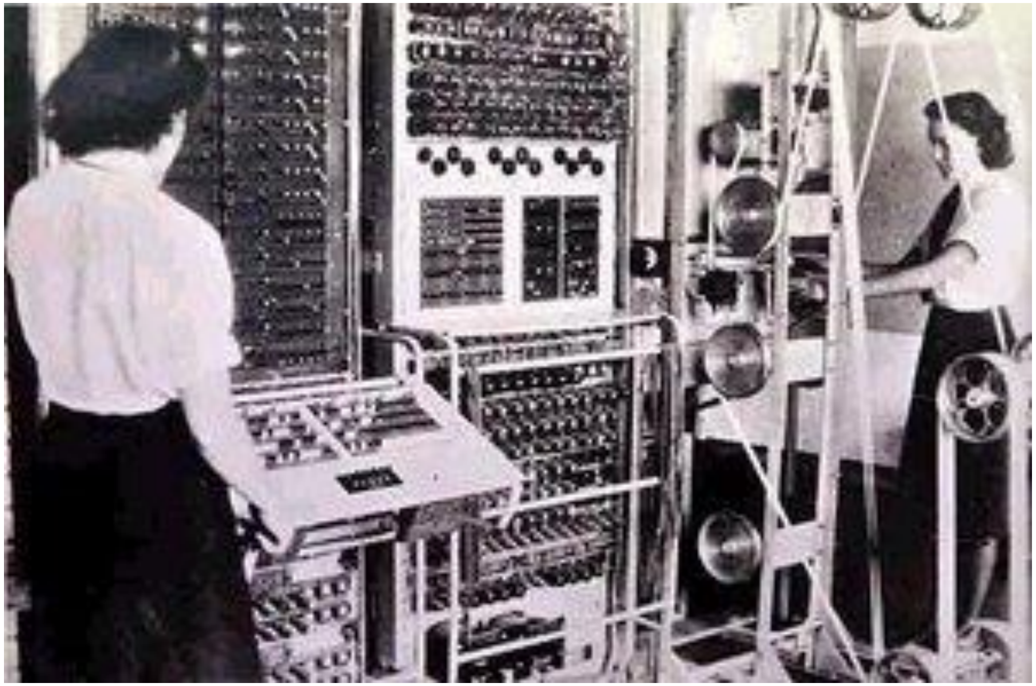
Bugs, debugging…

http://en.wikipedia.org/wiki/Debugging

- There is some controversy over who first used the term "bug" (see the Computer bug article for that discussion). Some claim that the term "debugging" was first defined by Glenford J Myers in his 1976 book Software Reliability: Principles and Practices as "diagnosing the precise nature of a known error and then correcting the error".

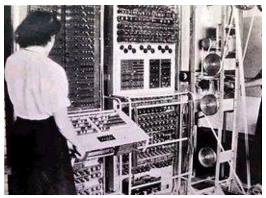
- The story goes that when one of the early computers malfunctioned Admiral Grace Hopper discovered that the problem was that a moth had got into the circuitry and caused a short circuit. This was the origin of the term bug in reference to problems with computer programs running correctly. The process of removing errors from computer programs has therefore become known as debugging.

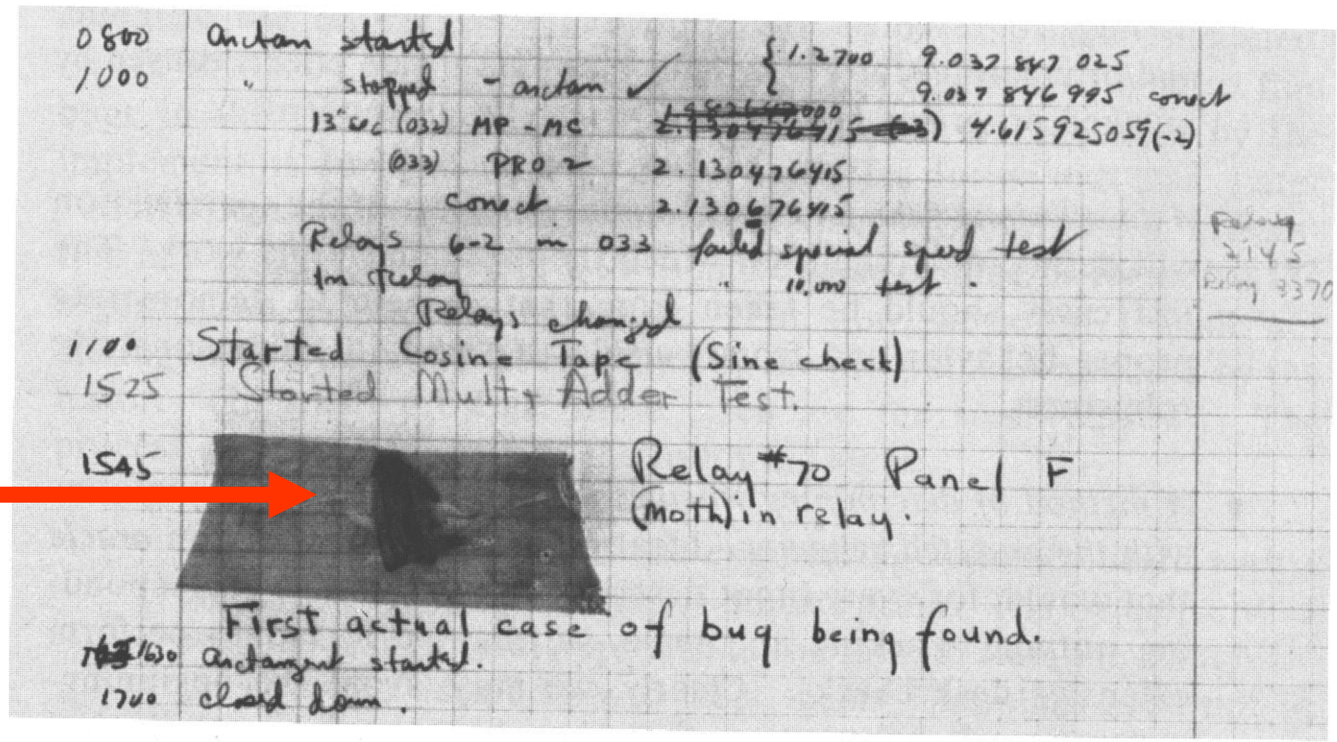
Bugs, debugging…



Mark II computer

Bugs, debugging…





We are debugging the computer…

Testing: the psychology of the process

We are goal-oriented and testing is a *destructive* process

Semantics of

Successful – unsuccessful

in software testing

Destructive process of trying a find the errors

The Psychology of Testing: false(?) definitions

Testing is the process of demonstrating that errors are not present

The purpose of testing is to show that a program performs its intended functions correctly

Testing is the process of establishing confidence that a program does what it supposed to do

Software Testing: A Multifaceted Activity

Knowledge, imagination, and ingenuity

Technical tasks

Economics

Human psychology

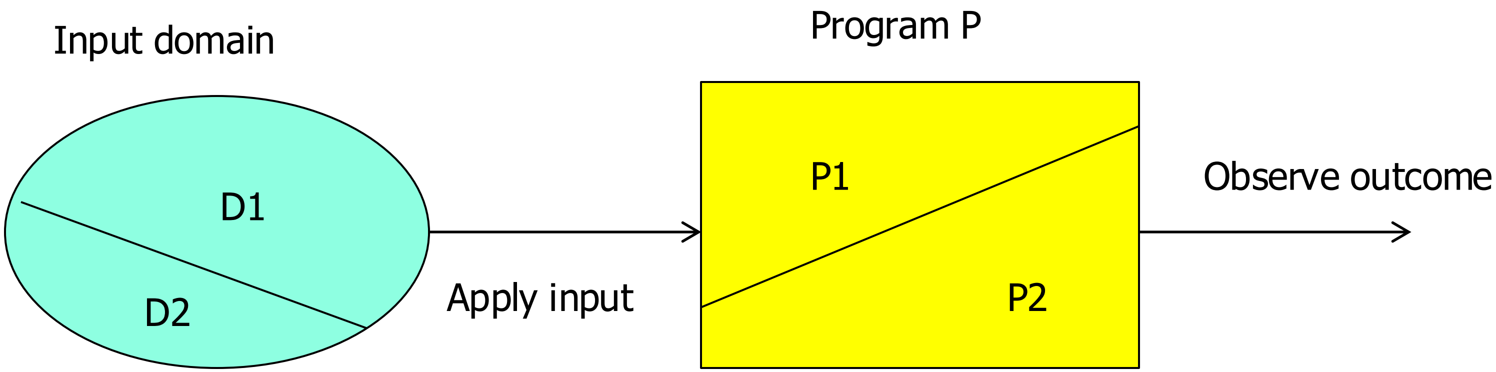
Testing activities (1)

Identify an objective to be tested

the objective to be tested; the objective defines an intention of designing test cases

Select inputs

select test inputs; based on requirements specification, source code, expectations
keep test objectives in mind



Testing activities (2)

Determine the expected outcome

determine(compute) expected outputs for selected inputs.
Based on high-level understanding of the test objective and specifications

Set up the execution environment

satisfy all assumptions external to the program must be satisfied (local system external to the program, initialize remote, external system

Testing activities (3)

Execute the program

execute the program with selected inputs, observe the actual output

Analyze the test result

select test inputs; based on requirements specification, source code, expectations
keep test objectives in mind

Determine the expected outcome

compare the actual outcome with the expected execution.
Three major test verdicts:
pass (produced expected outcome)
fail
inconclusive (not possible to assign pass / fail; e.g., timeout on distributed application)

Software testability

Testability

a system facilitates the formation of test criteria and the performance of tests to determine whether these criteria have been met

Software observability

How easy is to observe the behavior of a program in terms of output, effects on the environment

Software controllability

How easy is to provide a program with the needed inputs (values, operations, behavior)

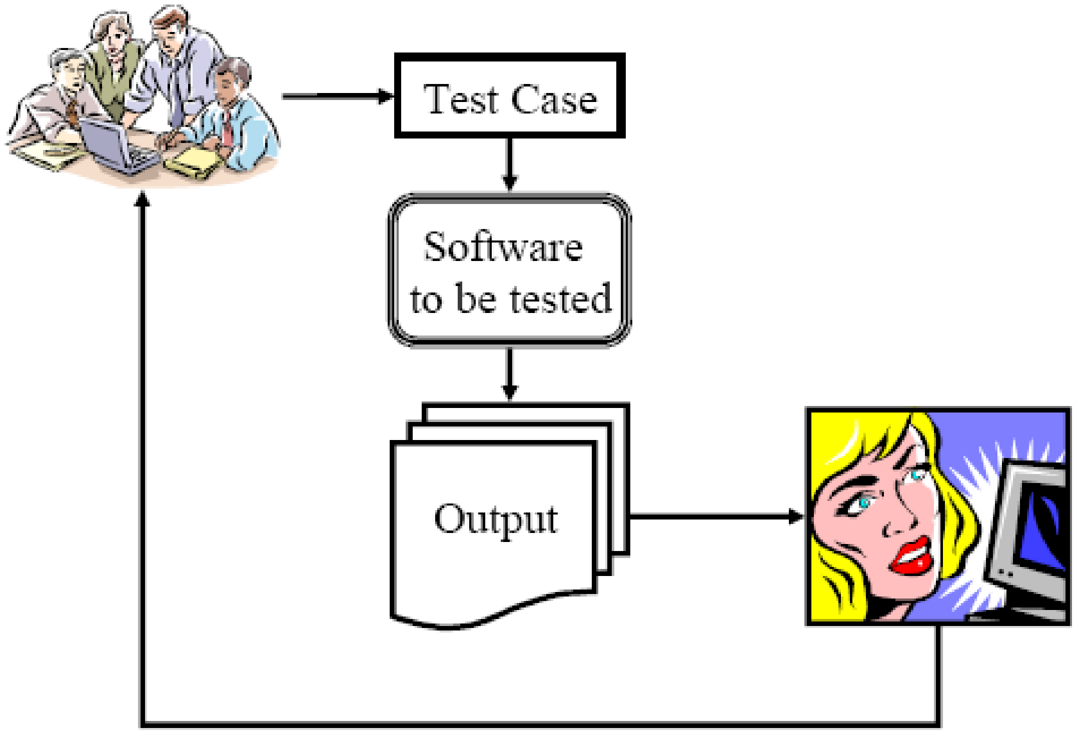
Planning and executing tests

- Run tests as early and as many times as possible
- Guidelines in choosing set of regression tests
  - Exercising all existing software functions
  - Focusing on changed components
  - Targeting functions likely to be affected by changes

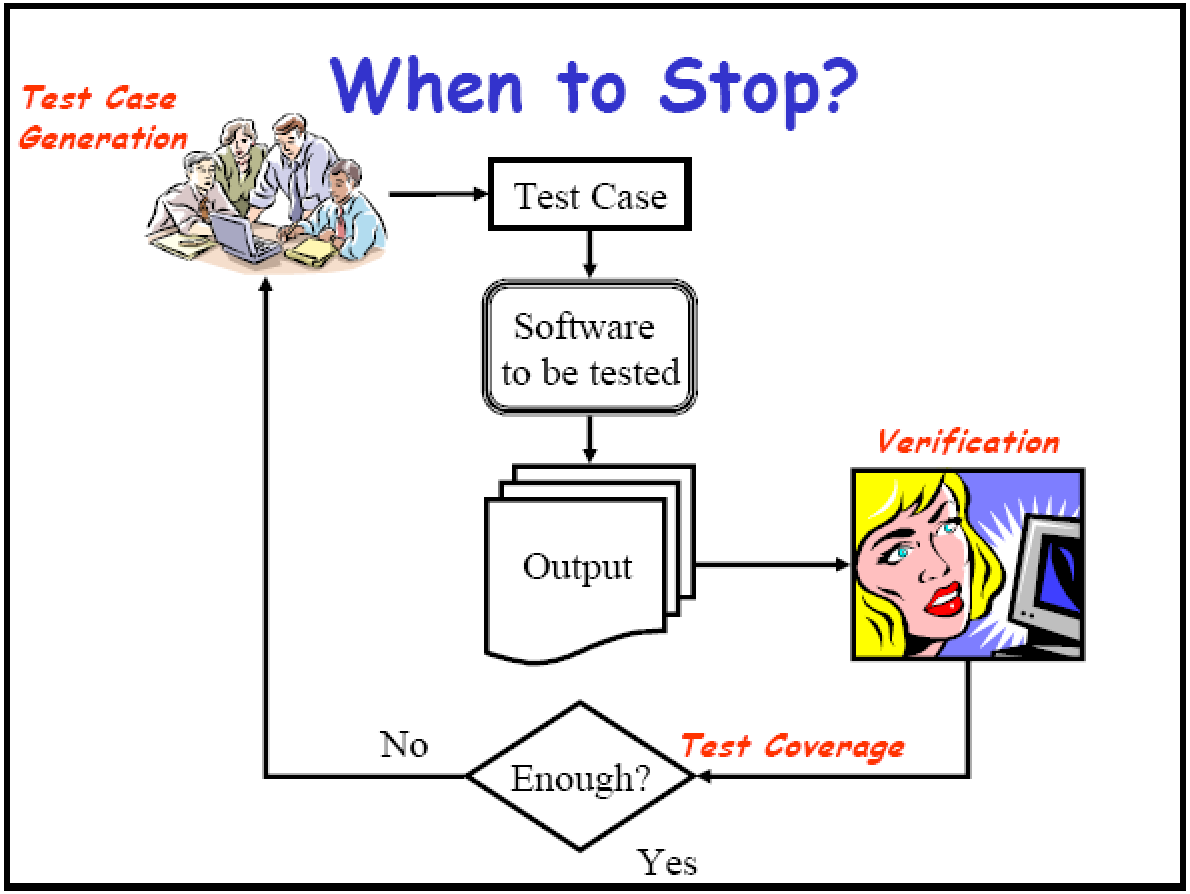
In general,

- *Testing is iterative and essential during development*

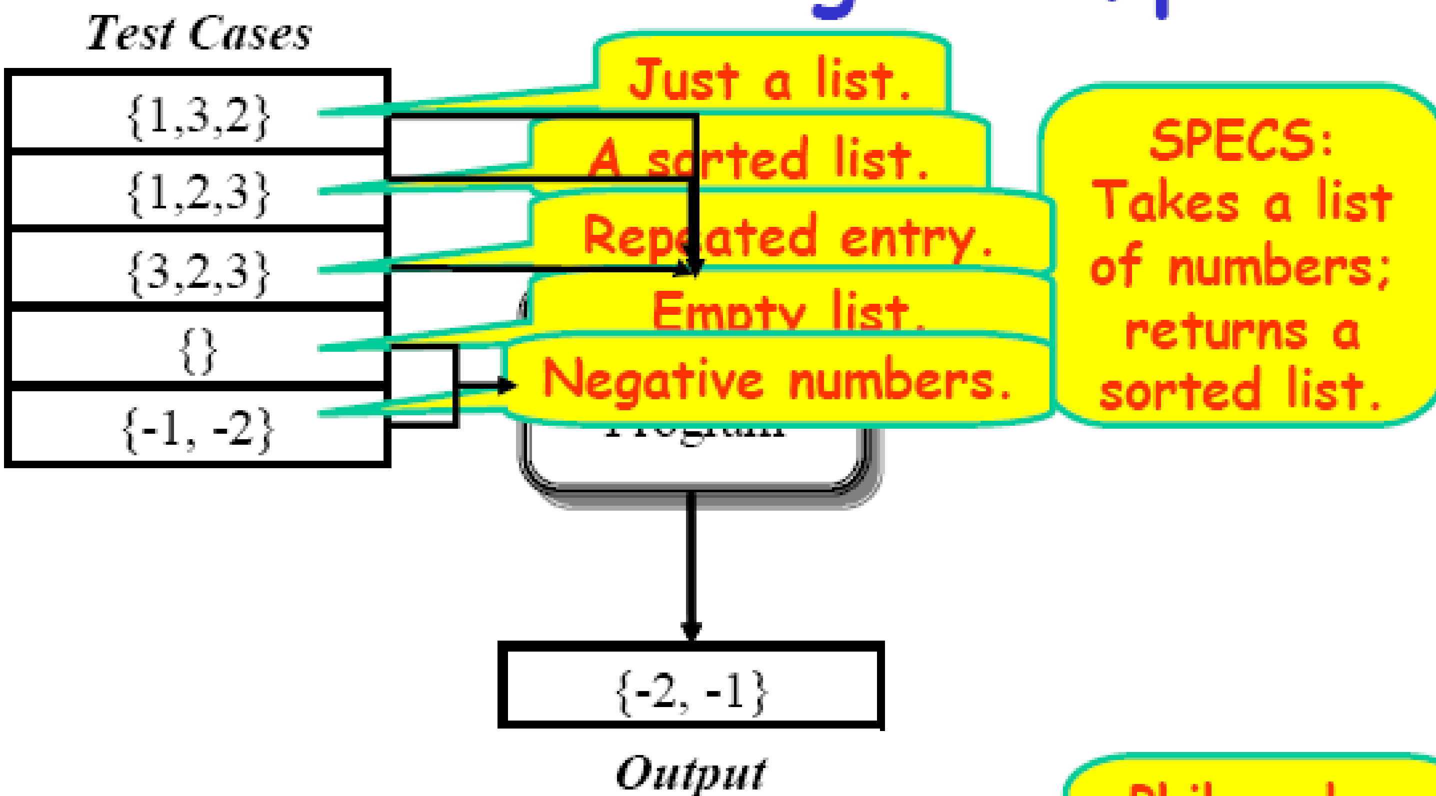
Testing loop



Testing loop: stopping criterion

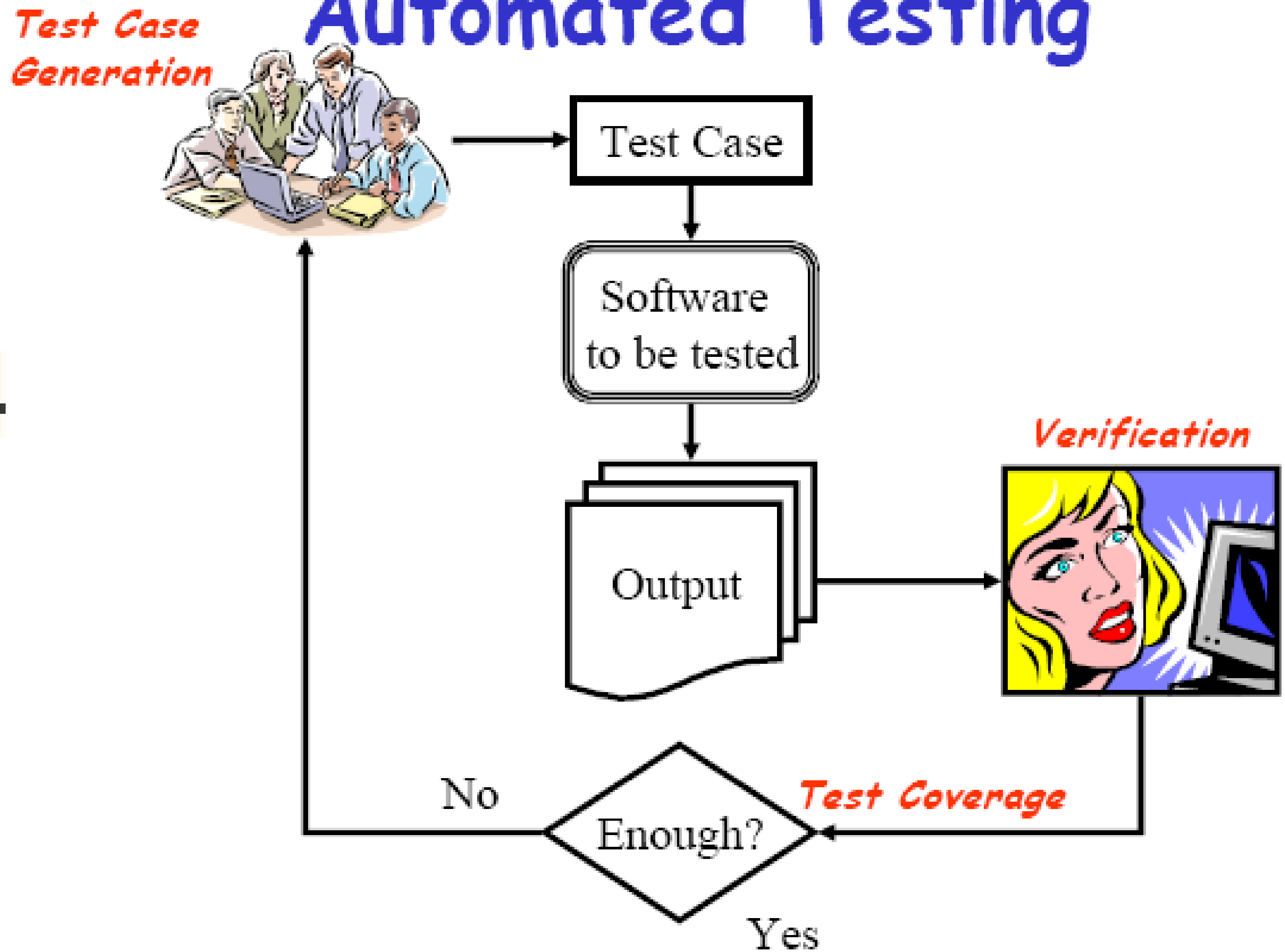


A Real Testing Example



Philosophy:
What are we
trying to do?

Automated Testing



JUnit

Embedding test cases in executable scripts

Junit used for unit and integration testing

Python

`Unittest` package
Realizing various assertions

`self.assertEqual`(4, self.calc.add(2,3))

```
assertEqual(x,y,msg=None)
assertGreater(x,y,msg=None)
…
assertIsInstance(obj,class,msg=None)
```

Who tests the software better??

developer

independent tester

Understands the system
but, will test "gently"
and, is driven by "delivery"

Must learn about the system,
but, will attempt to break it
and, is driven by quality

Stakeholders of Testing

- Developer → plans out and conducts a multi-phased testing effort to validate and verify the software product
- Manager → allocates resources to ensure a thorough testing effort can be conducted on the product
- Customer → provides additional info to the testers as specific test cases are written

Economics of software testing

- Cost of software failures surpasses the cost of testing
- Both physical costs and conceptual costs to software failures
  - Expenses for debugging and recall of product
  - Loss of customers’ faith in company
- Effective test building
  - Uncovering as many defects as possible with minimum amount of time and effort

Verification & Validation (V&V)

- A “verification & validation” practice
- **Verification**
  - Specific tests to check for specific functions
  - “How” – the **process** of building
  - “Are we building the product <u>right</u>?”
- **Validation**
  - Ensuring that the product meets the customer’s requirements
  - “What” – the **product** itself
  - “Are we building the <u>right</u> product?”

Taxonomy of testing

	Black Box (Functional)	Glass Box (Structural)
Dynamic	Random testing Domain testing Cause–effect graphing	Computation testing Domain testing Path-based testing Data generation Mutation analysis
Static	Specification proving	Code walkthroughs Inspections Program proving Symbolic execution Anomaly analysis

Testing Level	Tests Based Upon	Kind of Testing
Unit	Low-Level Design Actual Code Structure	White Box
Integration	Low-Level Design High-Level Design Smooth components integration	White Box Black Box
Functional and System	High-Level Design Requirements Analysis Functional: testing specific functionality System: testing in different environment	Black Box
Acceptance	Requirements Performed by customers	Black Box
Regression	Change Documentation High-Level Design Spot check throughout all (or most) testing cycles	White Box Black Box

Types of software tests

Functional tests
Exercise code with nominal inputs; aspects of functionality

Performance test
Test performance of software (execution time, response time, device utilization)

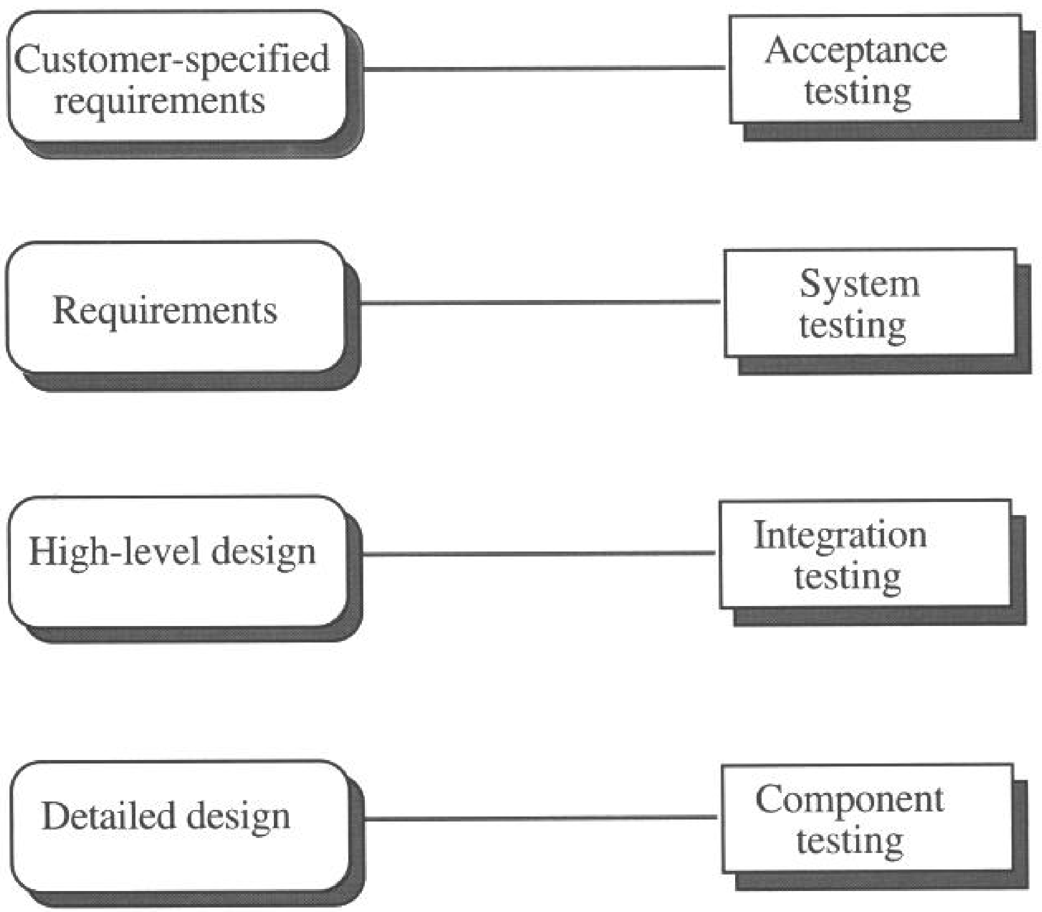
Stress test
Intentional break of the system

Structure test
Testing internal logic of a system

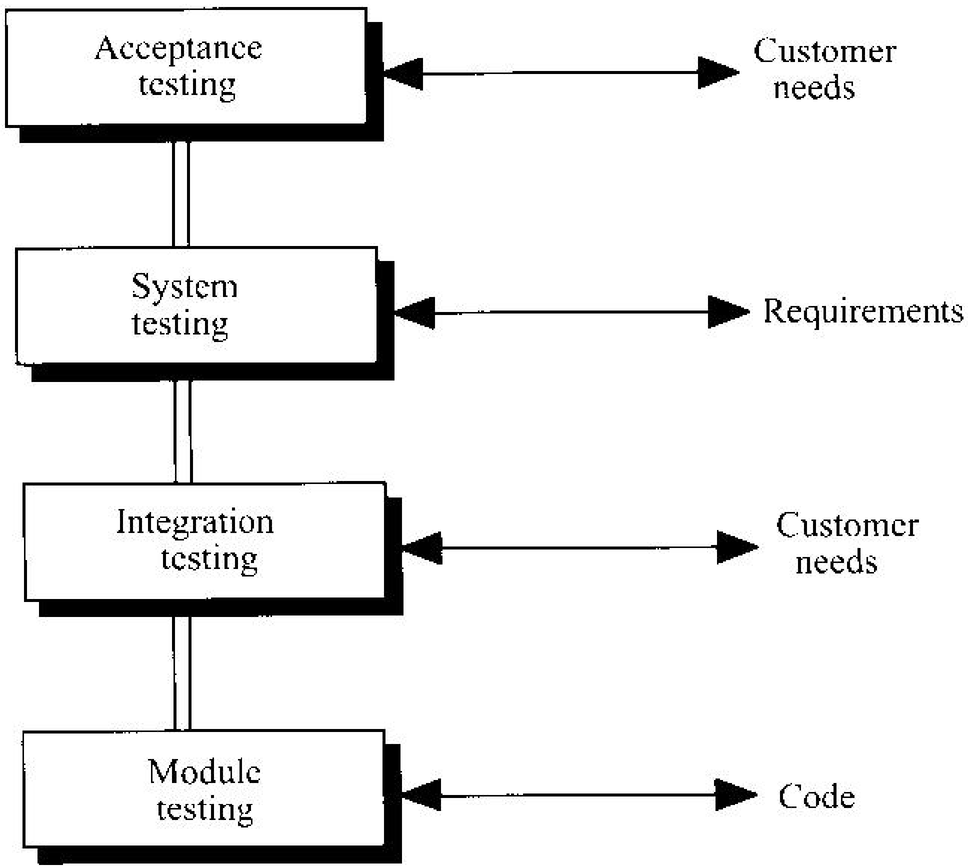
Testing in the small- testing in the large
Part of the system under testing versus whole system under testing

Black box – white box testing
Depends whether the internal logic becomes available for testing purposes

Software development and testing



Levels of software testing



Testing versus debugging

- testing is about making the system to fail
- debugging takes results of testing, localizes faults and errors and fixes them
  - debugging $\neq$ testing since debugging is performed after testing

- testing is performed using requirements as the "golden standard"
  - if requirements are wrong then testing may not discover failures

Software testing and reliability

Reliability of software and “physical” reliability

Similar formalisms,
concepts (Time to First Failure, TFF;
Mean Time Between Failures, MTBF)
and modeling paradigm (reliability theory)

Origin of faults
  systems -- physics of components (wear out…)
  software – design and implementation problems

Aging of software

Software Maintenance

Types of maintenance

Corrective maintenance – maintenance to correct errors (20%)

Adaptive maintenance – results from external changes to which the software system must respond (25%)

Perfective maintenance – all other changes (user enhancements, documentation changes, efficiency improvement) (55%)

Software engineering: corrective -20% adaptive+perfective – 80%
Engineering: corrective > 99% adaptive+perfective < 1%

Software Maintenance and laws of software evolution

Continuing change

Software undergoes continual change or becomes progressively less useful. The change or decay process continues until it is judged more cost effective to replace the system

Increasing complexity

As software changes continuously, its *complexity*, reflecting structure increases unless work is done to maintain or reduce it.

Software Testing Principles
(Myers, 2004)

1.
A necessary part of a test case is a definition of the expected output or result

Detailed examination of the output.
The test case must consist of two components

- **A description of the input data to the program**
- **A precise description of the correct output of the program for that set of input data**

Avoid a subconscious desire to see the correct result
(in spite of the destructive nature of testing

Software Testing Principles
(Myers, 2004)

2.

A programmer should avoid attempting to test his/her own program

Software Testing Principles
(Myers, 2004)

3.

A programming organization should not test its own programs

argument similar to the previous one. Living organization with psychological problems and constraints. Objectives to produce software on schedule and on budget. Extremely difficult to quantify the reliability of the software. This does not say it is *impossible*; there are more economical ways of doing that

Software Testing Principles
(Myers, 2004)

4.
Thoroughly inspect the results of each test

error that are found on later tests are often missed in the results from earlier tests

Software Testing Principles (Myers, 2004)

5.

Test cases must be written for input conditions that are invalid and unexpected, as well as for those that are valid and expected

Software Testing Principles
(Myers, 2004)

6.

Examining a program to see if it does not do what it supposed to do is only half of the battle; the other half is seeing whether the program does what it is not supposed to do

Payroll program: produces cheques for nonexistent employees or overwrites personal records, …

Software Testing Principles
(Myers, 2004)

7.

Avoid throwaway test cases unless the program is truly a throwaway program

Bad practice: invent test cases on the fly,
Tests are a valuable investment, regression testing

Software Testing Principles
(Myers, 2004)

8.

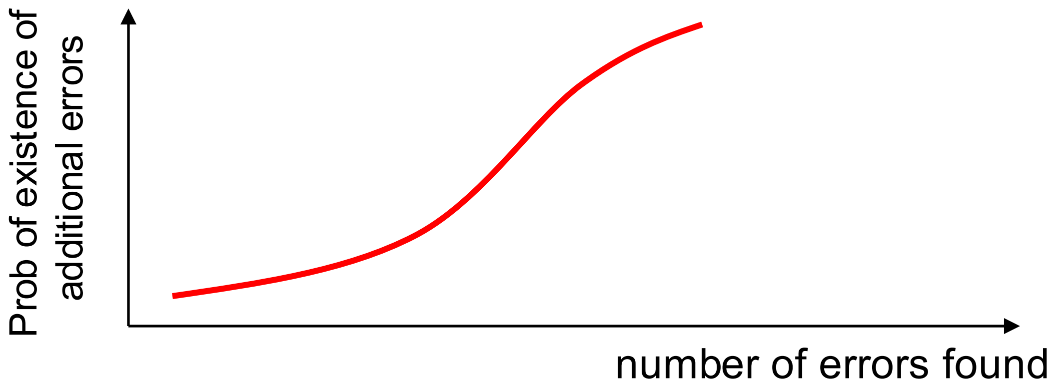
Do not plan a testing effort under the tacit assumption that no errors will be found

Incorrect definition of testing– showing that the program functions correctly

Software Testing Principles
(Myers, 2004)

9.

The probability of the existence of more errors in a section of a program is proportional to the number of errors already found in this section



Errors tend to come in clusters; some sections seem to be much more prone to errors than others; no good explanation for this phenomenon.
Additional testing effort need to be focused on error-prone sections

Software Testing Principles
(Myers, 2004)

10.

Testing is an extremely creative and intellectually challenging task

Developer – software tester (glamorous job?)
Distribution of resources within organization
Economical impact