

# Introduction

Software Testing

# Software is Everywhere



# Outline

- Why is testing necessary?
- What is testing?
- Testing Myths
- Software Testing Terminologies
- Test principles
- Fundamental test process
- The psychology of testing
- Limitations of Software Testing

Why is testing necessary?

# Why Software Testing is Important?

- China Airlines Airbus A300



Crashed due to a **software bug** on April 26, 1994, killing **264** innocent lives

# Why Software Testing is Important?

- Canada's Therac-25 radiation therapy machine



Malfunctioned in 1985 due to **software bug** and delivered lethal radiation doses to patients, leaving **3 people dead** and critically **injuring 3 others**.

# Why Software Testing is Important?

- \$1.2 billion military satellite launch



**Failed** In April of 1999 because of a **software bug**. The costliest accident in history



# Why Software Testing is Important?

- US Bank Accounts



A **software bug** caused 823 customers to be **credited \$920** million



# Economic Impact

- The Cost of Poor Software Quality in the US 2020 Report by CISQ Consortium for Information & Software Quality:
  - The total Cost of Poor Software Quality (CPSQ) in the US is \$2.08 trillion
  - Operational failures 1.56 trillion
  - Legacy systems 520 billions
- We want software systems to be reliable
  - Testing is how, in most cases, we find out if they are

# A Case Study: Ariane 5 Launch Vehicle

- European expendable launch system
- Double the payload capacity of its predecessor Ariane 4
  - 113 successful launches, 3 failures
- Flight 501, maiden launch,  
*June 4, 1996, 12:34pm*



# Case Study – Ariane 5: What Happened?

- To - To + 36s : normal
- Within SRI 2:
  - BH (Bias Horizontal) >  $2^{15}$
  - `convert_double_to_int(BH)` fails!
  - exception SRI -> crash SRI 2 & 1
- OBC disoriented
  - angle >  $20^\circ$ , huge aerodynamics constraints
  - boosters separating...
- To + 39s: self destruction
  - cost: € 500M

# Case Study – Ariane 5: Why Did It Happen?

- Not a programming error
  - unprotected conversion = design decision (~1980)
- Not a design error
  - Justified against Ariane 4 trajectory & RT constraints
- Problem with integration testing
  - Theoretically detectable.
  - But huge test space vs. limited resources
  - Furthermore, SRI useless at this stage of the flight!
- Reuse of a component with a hidden constraint
  - Precondition :  $\text{abs}(\text{BH}) < 32768.0$
  - Valid for Ariane 4, but no longer for Ariane 5
    - *More powerful rocket*

# Case Study – Ariane 5: Lessons Learned in Software Eng.

- Test! Test! Test!
- Test! Even when the code is reused.
- When reuse, ensure the assumptions are still valid.
- When write reusable code, document the assumptions.
- Write fail-safe code.
- Do not propagate errors.

# Why Software Testing is Important?

- History is full of such examples. Look for more examples!
- Testing is **important** because software bugs could be **expensive** or even **dangerous**.
- Software bugs can potentially cause **monetary** and **human loss**,

# Why Programs fail

## Congratulations!

- Your code is complete. It compiles. It runs ...
- Your program fails. How can this be?
- There is a defect in the code. When the code is executed, the defect causes bad behavior, which later becomes visible as a failure.
- Before a program can be debugged, we must set it up such that it can be *tested* — that is, executed with the intent to make it fail.
- The first step in debugging is to *reproduce* the problem in question — that is, to create a test case that causes the program to fail in the specified way.
  - The first reason is to bring it under control, such that it can be observed.
  - The second reason is to verify the success of the fix.
- Now you will have the fun of testing and debugging!



# Why Do We Test?

- Testing is expensive.
  - So are failures!
- What do we gain from that cost?
  - Finding bugs
  - Leading to
    - Fixing bugs
    - Raising the quality of the program or system we are testing

# Trade-Offs of Cost and Failures

Total Cost of Quality (CoQ) =  
Cost of Conformance (CoC) +  
Cost of Non-Conformance (CoNC)

- Cost of Conformance
  - Prevention: quality planning, investment in tools, quality training
  - Appraisal: testing, inspection
- Cost of Non-Conformance
  - Internal failures: rework
  - External failure: liability, loss of properties, loss of lives

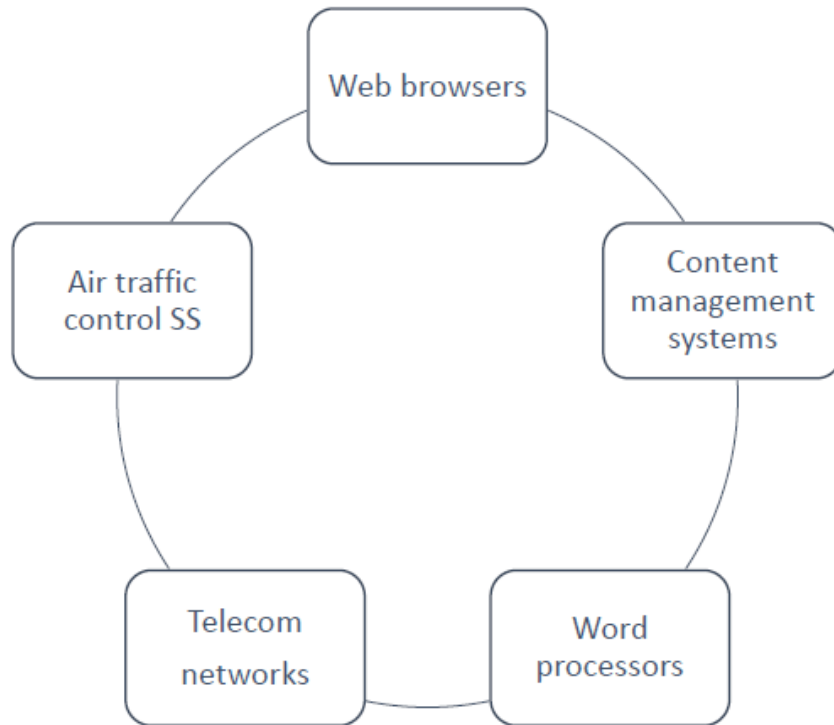
# Trade-Offs of Cost and Failures

- Testing adds to Cost of Conformance
- It must directly reduce Cost of Non-Conformance



# Software systems context

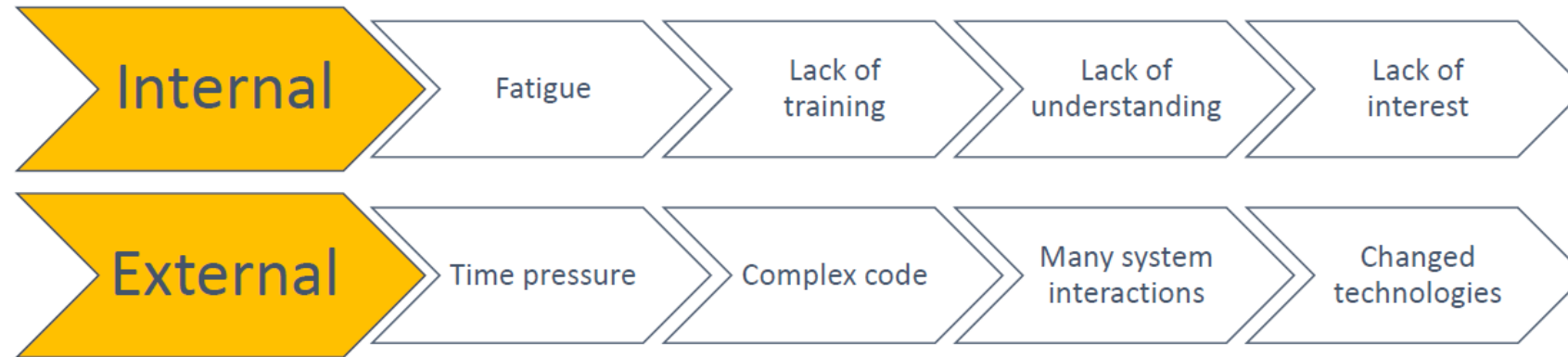
- Software systems are an important part of life:



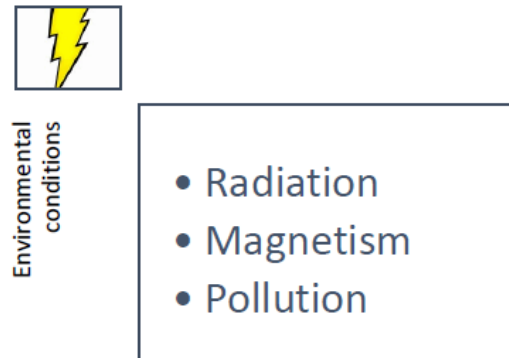
- Most people had experience with software not working as expected.
- If the SW system doesn't work correctly, it can lead to problems like:
  - Loss of money
  - Loss of business reputation
  - Injury or death

# Causes of software defects

- Human Error



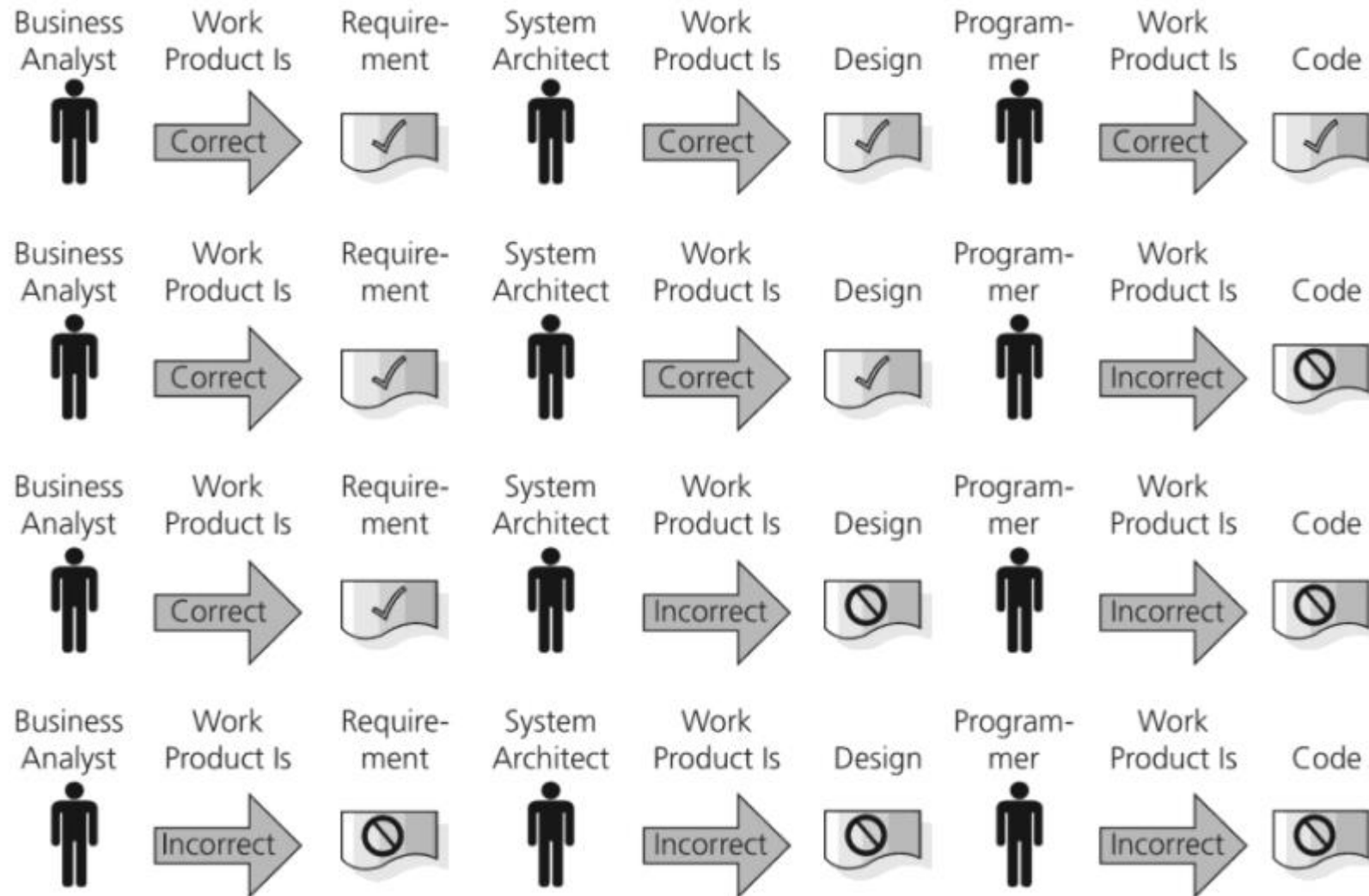
- Non-controllable events (i.e. environmental conditions)



# Causes of software defects

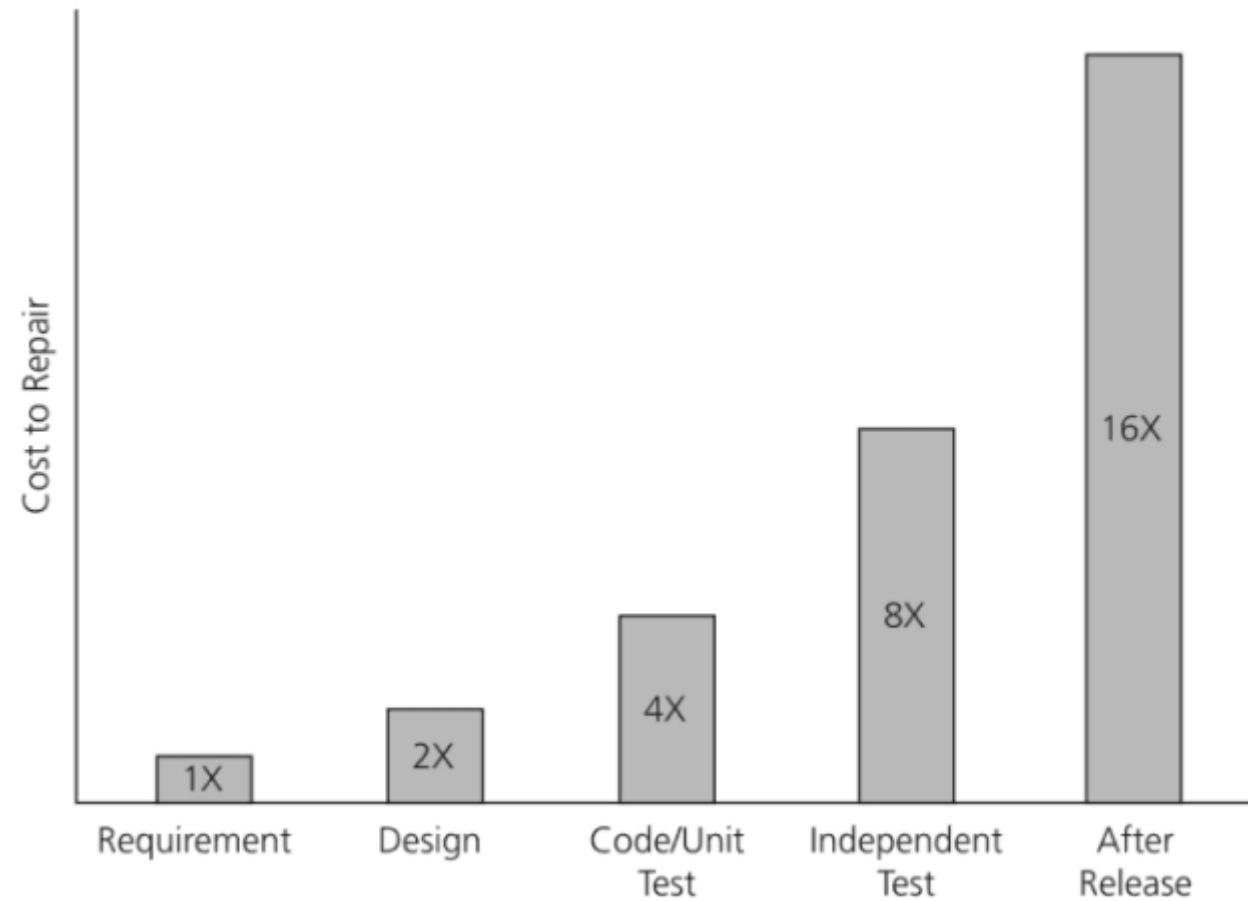
- Both causes of errors produce defects( = faults, bugs) in the code.
- Defects, if executed, may result in failures of the SW system (the system will fail to do what it should).
- Failures can affect seriously the users of the SW system, i.e.:
  - Break pedal not working
  - Miscalculations in financial SW systems

# Four typical scenarios





# Cost to repair



# Role of testing

- Testing has an important role in all stages of a SW product's life cycle:
  - Planning
  - Development
  - Maintenance
  - Operations

# Role of testing

- To reduce the risk of problems occurring during operation
- To check if the SW system meets:
  - legal requirements
  - Industry specific standards
- To learn more about the SW system

# Testing ...

Measures the quality the SW in terms of defects found

- Functional aspects
- Non functional aspects (Reliability, Usability, Portability)

Creates confidence in the quality of the SW

- If it's properly tested and a minimum of defects are found

Teaches us lessons to apply in future projects

- By understanding the root causes of defects, processes can be improved. This can prevent defects from reoccurring.

# What is testing?

# What is Software Testing?

- It is a **systematic process** used to identify the **correctness, completeness, and quality** of developed software. It includes a set of activities conducted with the intent of **finding errors** in a software so it can be corrected before the product is released to the end users
- **In simple words:** Software testing is an activity that ensures the software system is **defect free**.
- It can be either done **manually** or using **automated tools**.

# Definition of testing

- The **process of testing** all SW life-cycle activities:
  - *both static and dynamic,*
- concerned with:
  - *planning, preparation and evaluation*
- of :
  - software products and related work products
- to:
  - determine that they satisfy specified requirements
  - demonstrate that they are fit for purpose
  - detect defects.



# Definition of testing

- Depending on the objectives of the test process, testing can be focused on
  - Confirming that the SW system meets the requirements
  - Causing as many failures as possible
  - Checking that no defects have been introduced during changes
  - Assessing the quality of the SW (with no intention of finding
  - Finding defects
    - reduces the probability of undiscovered defects
  - Creating confidence in the level of quality
  - Providing information for decision-making
  - Preventing defects

# Definition of testing

- Finding defects
  - reduces the probability of undiscovered defects
- Creating confidence in the level of quality
- Providing information for decision-making
- Preventing defects



# Software Testing

# Myth #1 in Software Testing

Q: What is the objective of software testing?

A: Testing is to show that there are no errors/bugs/defects in the software.



**BUSTED**

▶ Fact:

- ▶ No!! The main objective of testing is to *discover* defects.
- ▶ Testing is a *destructive* activity.

# Myth #2 in Software Testing

Q: What is the objective of software testing?

A: Testing is to ensure that the software does what it is supposed to do.



► Fact:

- Only partly true.
- Testing is also to ensure the software *does not* do what it is *not supposed* to do.

# Myth #3 in Software Testing

Q: How challenging is software testing?

A: Testing is easier than design and implementation.



**BUSTED**

▶ Fact:

- ▶ Must consider all possible scenarios.
- ▶ Implied and unstated requirements and threats.
- ▶ Must be imaginative and creative.

# Myth #4 in Software Testing

Q: How challenging is software testing?

A: Testing is an extremely creative and intellectually challenging task.



**CONFIRMED**



# A Self Assessment Test

# Test the Following Program

- The program reads in 3 integer values that represent the lengths of the sides of a triangle.
- The program prints a message that states whether the triangle is
  - Equilateral (all 3 sides are equal)
  - Isosceles (exactly 2 of the 3 sides are equal)
  - Scalene (all 3 sides are of a different length)

Write a set of test cases that you feel would adequately test this program.

Test Case	Sample Test Data		Expected Results
Enter no values	nil		Error. 3 integers must be given.
Enter 1 value	12		Error. 3 integers must be given.
Enter 2 values	12, 13		Error. 3 integers must be given.
Happy path - Equilateral	5, 5, 5		Equilateral
Happy path - Isosceles	3, 3, 4		Isosceles
Happy path - Scalene	2, 3, 4		Scalene
Use incorrect format	2.000, 3.000, 4.000		Error. 3 integers must be given.
Use non-integers	2.5, 7.5, 8.7		Error. 3 integers must be given.
Use spaces			Error. 3 integers must be given.
Use other characters	a, %, 2	&, \$, =	Error. 3 integers must be given.
<b>Equilateral Stuff</b> ... all 3 entered values are equal			
Valid equilateral	12, 12, 12		Equilateral
At least one is 0 (boundary case)	0, 0, 0		Error. All values must be > 0.
At least one is max (boundary case)	25, 25, 25 *		Equilateral
At least one is < 0	-1, -1, -1		Error. All values must be > 0.
At least one is > max	27, 27, 27		Error. All values must be <= max
<b>Isosceles Stuff</b> ... 2 entered values are equal			
Valid isosceles	2, 2, 1	2, 2, 3	Isosceles
Invalid isosceles	2, 2, 5	2, 2, 4	Error. Not a triangle.
At least one is 0	3, 3, 0	0, 0, 5	Error. All values must be > 0.
At least one is max, and valid triangle	25, 25, 24	24, 24, 25	Isosceles
At least one is max, and invalid triangle	10, 10, 25		Error. Not a triangle.
At least one is < 0	20, 20, -1	-1, -1, -20	Error. All values must be > 0.
At least one is > max	27, 27, 20	16, 16, 28	Error. All values must be <= max
<b>Scalene Stuff</b> ... none of the 3 entered values are equal			
Valid scalene	2, 3, 4		Scalene
Invalid scalene	2, 3, 25	2, 3, 5	Error. Not a triangle.
At least one is 0	0, 7, 9		Error. All values must be > 0.
At least one is max, and valid triangle	15, 20, 25		Scalene
At least one is max, and invalid triangle	10, 15, 25	2, 3, 25	Error. Not a triangle.
At least one is < 0	-2, -1, 5		Error. All values must be > 0.
At least one is > max	15, 20, 29		Error. All values must be <= max

\* For the max value, I just arbitrarily set it to 25 just to better illustrate the cases involving the max value.

# How Will You Do It?

Write a set of test cases that you feel would adequately test this program.

- How many cases are needed?
- How do you plan to test the program?
  - Run it once and manually enter the values?
  - Run it many times with different inputs?
  - Run it with a file containing a set of lines with test values?

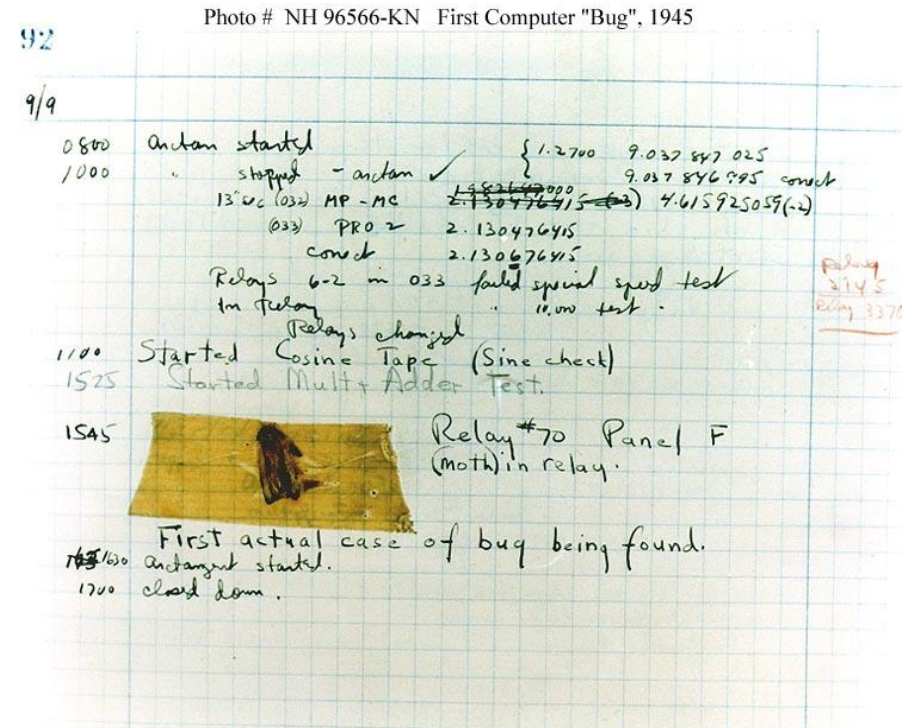
# How will you do it?

- “Smoke test.”, aka “Hello World!”
- Check handling of inputs
  - Illegal inputs – text instead of integers
  - Impossible inputs – floating vs. integer
  - Outrageous values – infinities, max and min values
  - Not in domain – negative numbers,  
values outside specifications
  - Input errors – wrong input, e.g. mis-spellings
- Stress test
  - multiple inputs without restart
  - run program for long periods of time

# Software Testing Terminologies

# The Very First Software Bug


- A moth found trapped between points at Relay # 70, Panel F
  - Mark II Aiken Relay Calculator
  - Harvard University
  - September 9, 1945.



On display in Smithsonian

# Failures

- Failures are
  - deviation of the observed behavior of a system from its specification, i.e., its expected behavior.
- Failures can only be determined with respect to the specifications.
- Failures are concerned with the observed behavior and outcome of the system.




```
++CDatabase::_stats.mem_used_u
_params.max_unrelevance = (int
if (_params.max_unrelevance <
_params.max_unrelevance =
_params.min_num_clause_lits fo
if (_params.min_num_clause_lit
_params.min_num_clause_lit
_params.max_num_clause_le
if (_params.max_num_clause_le
_params.max_num_clause_le
CHECK(
cout << "Forced to reduce unre
cout << "MaxUnrel: " << _params
<< " MinLenDel: " << _pa
<< " MaxLenCL : " << _pa
);
```



# Defects


- Defects are
  - flaws in a system that can cause the system to fail to perform its required function
    - e.g. an incorrect condition or statement.
- Defects are concerned with specific parts or components of the system.
- Defects are synonymous with *faults*, *bugs*.



```
++CDatabase::_stats.mem_used_u
_params.max_unrelevance = (int
if (_params.max_unrelevance <
_params.max_unrelevance =
_params.min_num_clause_lits fo
if (_params.min_num_clause_lit
_params.min_num_clause_lit
_params.max_num_clause_le
if (_params.min_conflict_claus
_params.min_conflict_claus
CHECK(
cout << "Forced to reduce unre
cout << "MaxUnrel: " << _params
<< " MinLenDel: " << _pa
<< " MaxLenCL : " << _pa
);
```

# Errors

- Errors are
  - human actions that result in a fault or defect in the system.
- Errors are concerned with the underlying causes of the defects.
- Errors are synonymous with *mistakes*.



```
++CDatabase::_stats.mem_used_u
_params.max_unrelevance = (int
if (_params.max_unrelevance <
_params.max_unrelevance =
_params.min_num_clause_lits fo
if (_params.min_num_clause_lit
_params.min_num_clause_lit
_params.max_num_clause_le
if (_params.max_conflict_claus
_params.max_conflict_claus
CHECK(
cout << "Forced to reduce unre
cout << "MaxUnrel: " << _params
<< " MinLenDel: " << _pa
<< " MaxLenCL : " << _pa
);
```

# The Relations among Failures, Defects, and Errors

- A human being makes an error (*mistake*)
  - can occur in design, coding, requirements, even testing.
- An *error* can lead to a defect (*fault*)
  - can occur in requirements, design, or program code.
- If a *defect* in code is executed, a failure may occur.
  - Failures only occur when a *defect* in the code is executed.
  - Not all defects cause failures all the time.
- Defects occur because human beings are fallible
- Failures can be caused by environmental conditions as well.

# Failures, Defects, and Errors

- **An example:** For any integer  $n$ ,  $\text{square}(n) = n * n$ .

```
int square (int x)
{
  return x*2;
}
```

Error: should be  $x^2$

A defect

$\text{square}(3) = 6$

A failure

# Failures, Defects, and Errors

- **An example:** For any integer  $n$ ,  $\text{square}(n) = n * n$ .

```
int square (int x)
{
    return x*2;
}
```

A defect

$\text{square}(2) = 4$

Correct result  
Not a failure

# Test Cases Terminology

- Test Case Verdicts

- Pass

- The test case execution was completed
    - The function being tested performed as expected

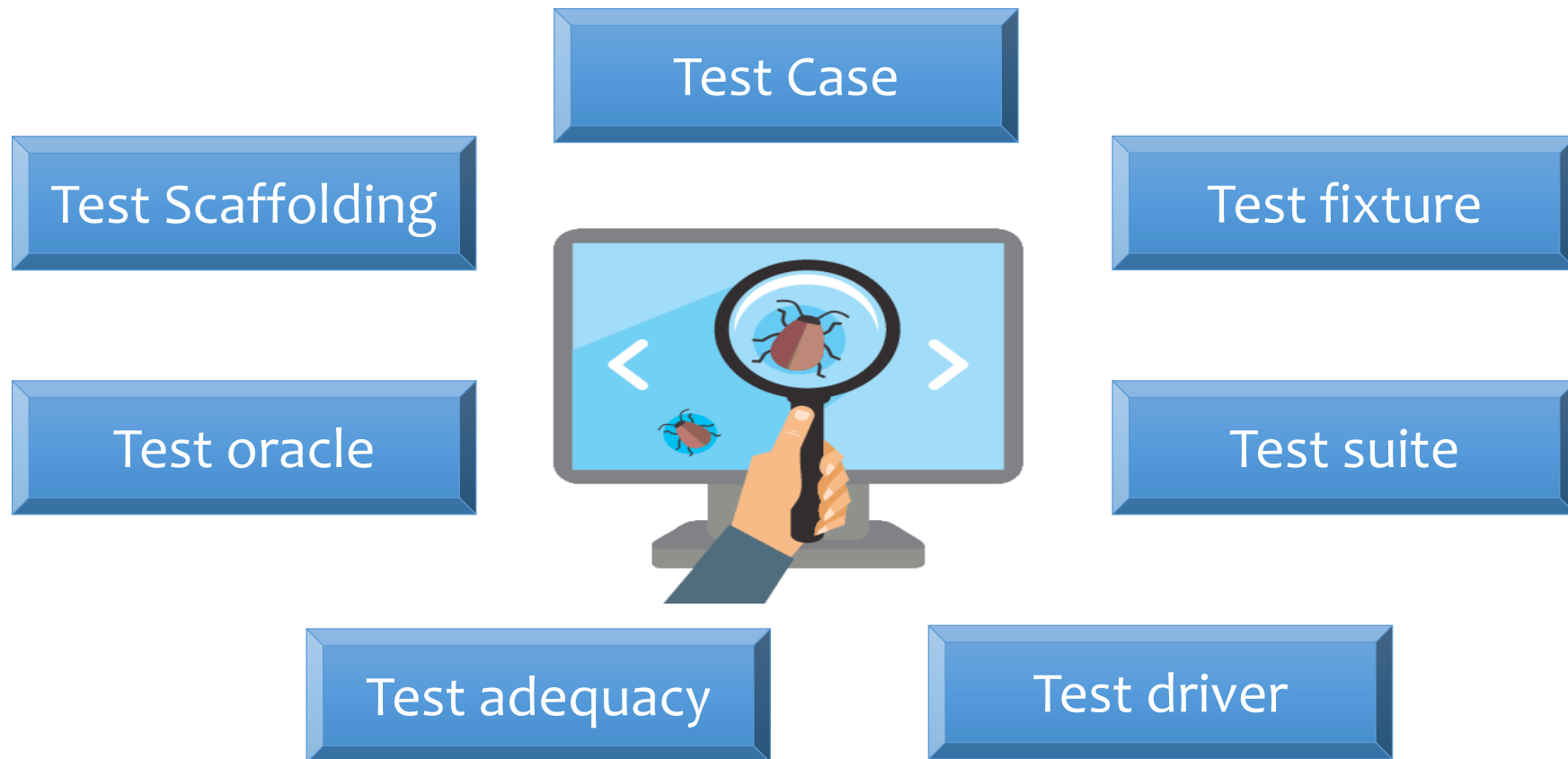
- Fail

- The test case execution was completed
    - The function being tested did not perform as expected

- Error

- The test case execution was not completed, due to an unexpected event, exceptions, or improper set up of the test case, etc.

# Testing: Concepts



# Testing: Concepts

- Test case (or, simply test)
  - An execution of the software with a given test input, including:
    - Input values
    - Sometimes include execution steps
    - Expected outputs

```
int actual_output=sum(1,2)  
assertTrue(actual_output==3);
```

Example JUnit test case for testing “sum(int a, intb)”



# Testing: Concepts

- Test oracle
  - The expected outputs of software for given input
  - A part of test cases
  - Hardest problem in auto-testing: test oracle generation

```
int actual_output=sum(1,2)  
assertTrue(actual_output==3);
```

Example JUnit test case for testing “sum(int a, intb)”

# Testing: Concepts

- Test Scaffolding
  - To do Unit tests, we have to provide replacements for parts of the program that we will omit from the test.
  - Additional code needed to execute a unit or subsystems in isolation
  - Not useful in production code
    - Needs to be removed

# Testing: Concepts

- Test fixture: a fixed state of the software under test used as a baseline for running tests; also known as the test context, e.g.,
  - Loading a database with a specific, known set of data
  - Preparation of input data and set-up/creation of fake or mock objects

# Testing: Concepts

- Test suite
  - A collection of test cases
  - Usually these test cases share similar pre-requisites and configuration
  - Usually can be run together in sequence
  - Different test suites for different purposes
  - Certain platforms, Certain feature, performance, ...
- Test Script
  - A script to run a sequence of test cases or a test suite automatically

# Testing: Concepts

- Test driver
  - A software framework that can load a collection of test cases or a test suite
  - It can also handle the configuration and comparison between expected outputs and actual outputs

# Testing: Concepts

- Test adequacy
  - We can't always use all test inputs, so which do we use and when do we stop?
  - We need a strategy to determine when we have done enough
- Adequacy criterion: A rule that lets us judge the sufficiency of a set of test data for a piece of software

# Testing: Concepts

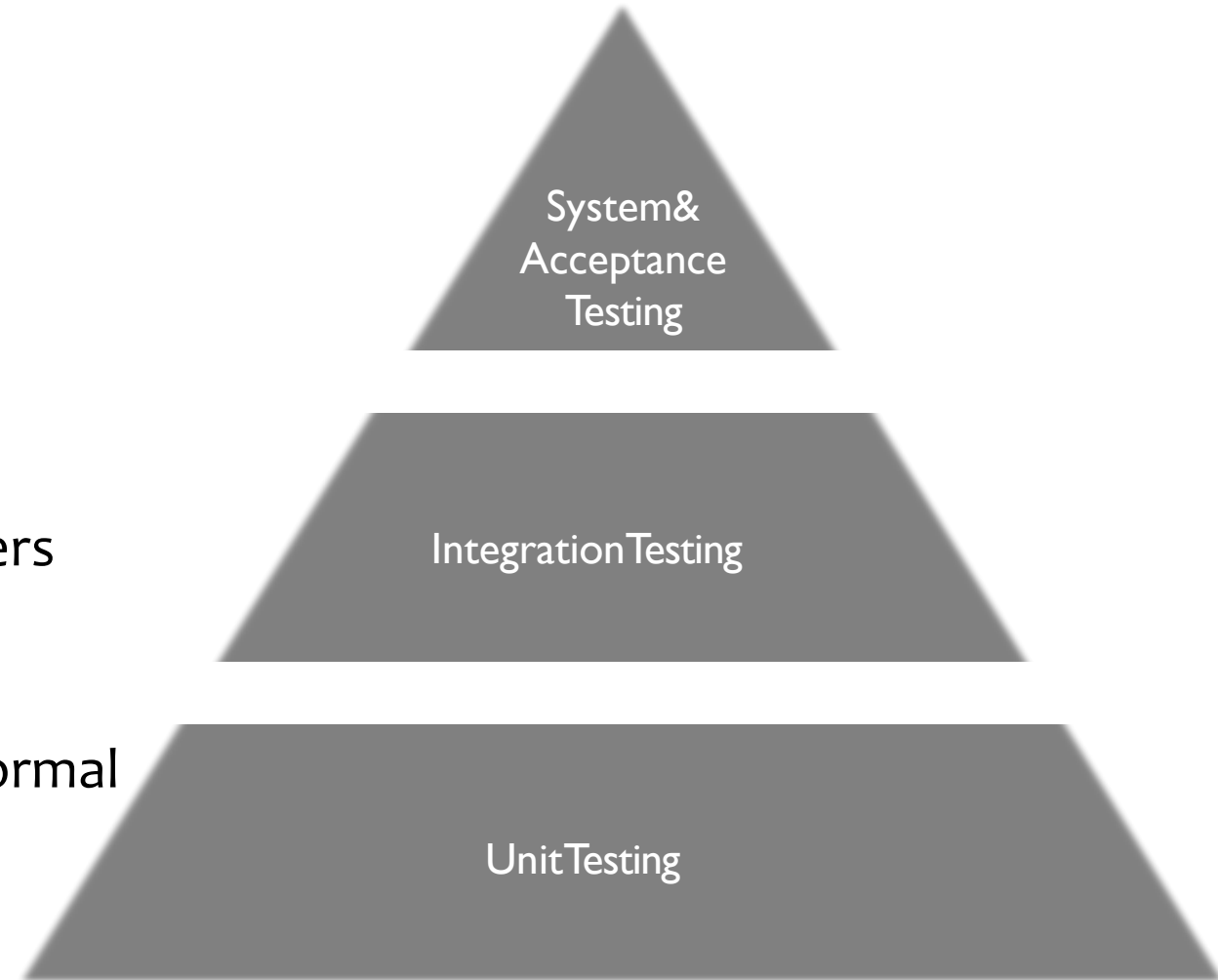
- Test adequacy example: test coverage
  - A measurement to evaluate the percentage of tested code
- Statement coverage
- Branch coverage, ...



Cobertura tool

# Granularity of Testing

- Unit Testing
  - Test of each single module
- Integration Testing
  - Test the interaction between modules
- System Testing
  - Test the system as a whole, by developers
- Acceptance Testing
  - Validate the system against user requirements, by customers, without formal test cases





# Testing Purpose

- The purpose of testing
  - to find defects.
  - to discover every conceivable weakness in a software product.

1. Software testing ≠ Debugging.
2. Software testing ≠ Quality assurance

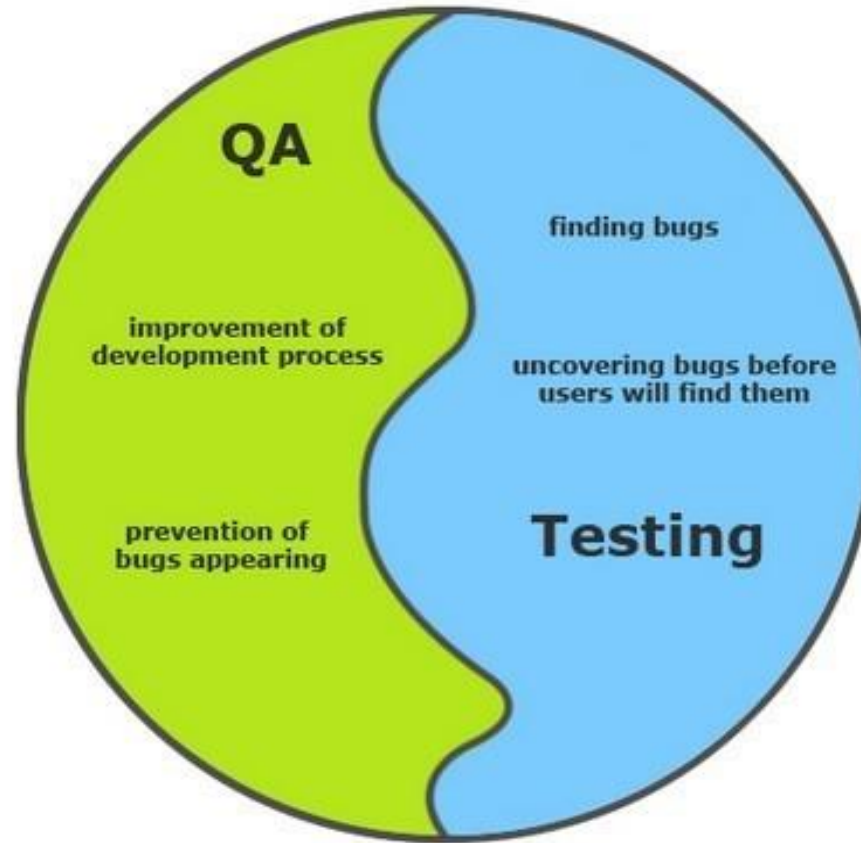
# Software Testing vs. Quality Assurance (QA)

- Software testing is a planned process that is used to identify the correctness, completeness, security and quality of software.
- Quality Assurance (QA) is planned and systematic way to evaluate quality of process used to produce a quality product.
- The goal of a QA is to provide assurance that a product is meeting customer's quality expectations.

# Software Testing vs. Quality Assurance (QA)

- Testing is necessary, but not sufficient for quality assurance
  - Testing contributes to improve quality by identifying problems.
- Quality assurance sets the standards for the team/organization to build better software.

# Software Testing vs. Quality Assurance (QA)



# The seven test principles

# The test principles

- **P1: Testing shows presence of defects**

- Testing can show that defects are present, but cannot prove there are no defects.
- Testing reduces the probability of undiscovered defects remaining in the software; but even if no defects are found, this is not a proof of correctness .

- **P2: Exhaustive testing is impossible**

- Testing everything is not feasible. We use risks and priorities to focus test effort.

- **P3: Early testing**

- Testing should start as soon as possible in the development life-cycle and should be focused on defined objectives.

- **P4: Defect clustering**

- A small number of modules contain most of the defects discovered during pre-release testing.

- **P5: Pesticide paradox**

- If the same set of tests will be repeated over and over, it will no longer find new bugs.

- **P6: Testing is context dependent**

- I.e., safety-critical SW is tested differently from an e-commerce site.

- **P7: Absence-of-errors fallacy**

- Finding and fixing defects does not help if the SW system is un-usable or does not meet user's expectations.

# P1: Testing shows the presence of defects, not their absence

- Testing can show that defects are present, but cannot prove that there are no defects.
- Testing reduces the probability of undiscovered defects remaining in the software. However, even if no defects are found, this is not a proof of correctness.

# P2: Exhaustive testing is impossible

- Testing everything(all combinations of input and preconditions) is not feasible except for specific cases.
- We use risks and priorities to focus the testing.



## P3: Early testing saves time and money

- Testing activities should start as early as possible in the software or system development life cycle and should be focused on defined objectives.

# P4: Defect clustering defects cluster together

- A small number of modules contains most of the defects discovered during pre release testing.

# P5: Pesticide paradox

- If the same tests are repeated over and over again, the same set of test cases will no longer find any new bugs.
- To overcome this 'pesticide paradox', the test cases need to be regularly reviewed and revised, and new and different tests need to be written to investigate different parts of the software.

# P6: Testing is context dependent

- Testing is done differently in different contexts.
- For example, testing of safety-critical software is different from e-commerce site testing.

# P7: Absence of error fallacy

- Finding and fixing defects does not help if the software system does not fulfill users' needs and expectations .

# Fundamental Test Processes

# Test Cases and Test Suites

- ***Test case***

- A test case consists of inputs, steps/actions, and expected results, i.e., pass-fail criterion

- ***Test suite***

- A group of test cases (usually organized around some principles, e.g. smoke test suite)

# Test Plan and Testing Process

- ***Test plan***

- A document that specifies how a system will be tested, including criteria for success, i.e., the exit criteria.

- ***Testing process***

- The testing process involves developing test plans, designing test cases, running the test cases, and evaluating the results



# Fundamental test processes

- Test planning
- Test monitoring and control
- Test analysis
- Test design
- Test implementation
- Test execution
- Test completion

# Test planning , monitoring and control

- **Who, what , why, when and where**
- A plan encompasses: what, how, when, by whom?
  - Scope, objectives and risk analyses
  - Test levels and types that will be applied
  - Documentation that will be produced
  - Assign resources for the different test activities
  - Schedule test implementation, execution, evaluation
- Control and adjust the planning to reflect new information, new challenges of the project.

# Analysis and design

- Review test basis:
  - Requirements
  - Product architecture
  - Product design
  - Interfaces
  - Risk analysis report
- Analysis: general test objectives are transformed into:
  - Test conditions
- Design:
  - Test cases
  - Test environments
  - Test data
  - Create traceability

# Implementation and execution

- *Implement:*
  - Group tests into scripts
  - Prioritize the scripts
  - Prepare test oracles
  - Write automated test scenarios
- *Execute:*
  - Run the tests and compare results with oracles
  - Report incidents
  - Repeat test activities for each corrected discrepancy
  - Log the outcome of the test execution

# Test completion

- *Evaluate:*
  - Assess test execution against the defined objectives
  - Check if:
    - More tests are needed
    - Exit criteria should be changed
- *Report:*
  - Write or extract a test summary report for the stakeholders.
- *Test closure activities*
  - The activities that make the test assets available for later use.

# The psychology of testing

# A good tester needs:

- Curiosity
  - Professional pessimism
  - Attention to details
  - Good communication skills
  - Experience at error guessing
- 
- To communicate defects and failures in a constructive way:
    - fact-focused reports and review of findings

# Independence in testing

- A certain degree of independence is often more effective at finding defects and failures.
  - However, the developer can very efficiently find bugs in their own code.
- The level of independence in the testing depends on
  - the objective of testing.



# Independence test levels


- Independence levels:




- Tests designed by the same person who wrote the code



- Tests designed by another person from the same team, but same organization



- Tests designed by a person from a separate testing team, but in the same organization



- Tests designed by a person from an outside organization / company (outsourcing the testing)

# Tips and tricks

- Be clear and objective
- Confirm that:
  - You have understood the requirements
  - The person that has to fix the bug has understood the problem

# Limitations of Software Testing

# Testing vs. Verification

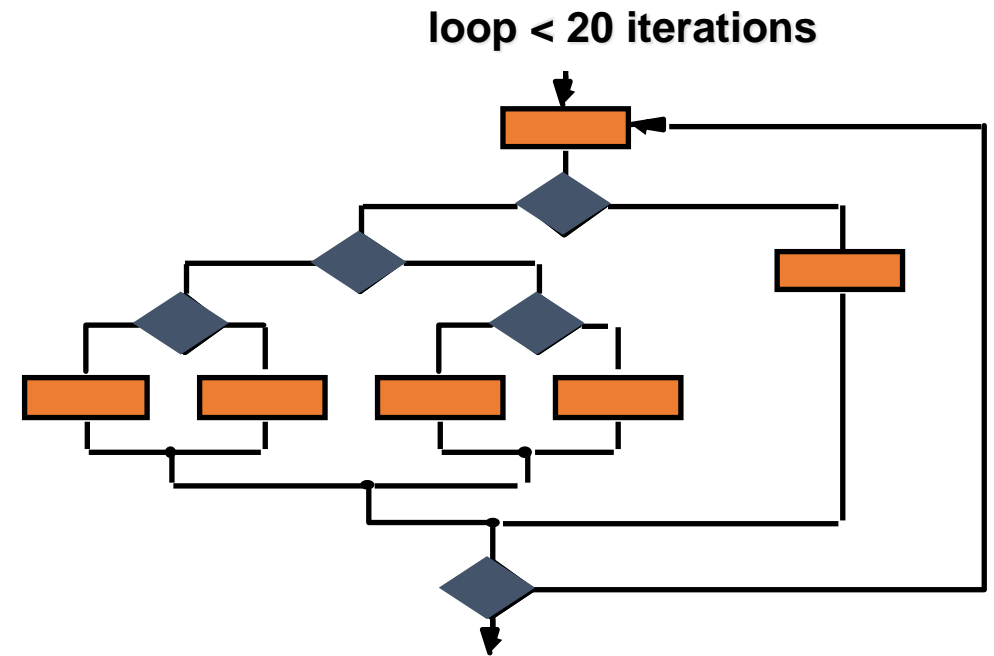
- Both, testing and verification attempts to exhibit **new** failures
- Debugging is a systematic process that finds and eliminates the defect that led to an observed failure
- Programs without **known** failures may still contain defects:
  - If they have not been verified
  - If they have been verified, but the failure is not covered by the specification

# Fundamental Principles Myth

- **Myth:** “Principles are just for reference. I will not use them in practice .”
- **Fact:**
  - This is so very untrue. Test Principles will help you create an effective Test Strategy and draft error catching test cases.
  - Experienced testers have internalized these principles to a level that they apply them without even thinking.
  - Hence the myth that the principles are not used in practice is simply not true.

# Exhaustive Testing

- Can we exhaustively test a program?
- Let's consider this simple program
- There are  $10^{14}$  possible paths!
  - If we execute one test per millisecond,
  - it would take 3,170 years to test this program!!
- Exhaustive testing is impossible!



# Absence of Defects

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

*(Dijkstra, 1969)*

Testing reduces the probability of undiscovered defects remaining in the software but, even if no defects are found, it is not a proof of absence of defects.

# How much testing is enough?

It depends on:

Level of risk:

- Technical risks
- Business risks

Project constraints:

- Time
- Budget

- It's impossible to test everything!
- Testing should provide sufficient information for the stakeholders to make informed decisions about:
  - Release of the software
  - Next development steps, etc.



# When to Stop Testing?

- One of the most difficult problems in testing is not knowing when to stop.
- You can use metrics to make a guess. Keep track of the defect rate. When it goes towards zero you can use it as an indicator. Or, ...
- Even if you do find the last bug, you'll never know it.

# How Much Testing is Enough?

- Take into account
  - the level of risk, including technical, safety, and business risks, and
  - project constraints such as time and budget.
- Testing should provide feedback to stakeholders to make informed decisions
  - about the release of the software,
  - for the next development step, or
  - handover to customers.
- Later, we will discuss some techniques for making this decision.