Software Testing & Quality Assurance

Introduction

- Basic terminology and definitions
- Motivation, importance and limitations
- First look at some systematic testing techniques and tools

Credits & Readings

- The material included in these slides are mostly adopted from the following books:
  - *Software Testing: A Craftsman's Approach*, by Paul Jorgensen, CRC PRESS, third edition, ISBN: 0-8493-7475-8
  - Cem Kaner, Jack Falk, Hung Q. Nguyen, *"Testing Computer Software"* Wiley (see also http://www.testingeducation.org/)
  - Paul Ammann and Jeff Offutt, *"Introduction to Software Testing"*, Cambridge University Press
  - Glen Myers, *"The Art of Software Testing"*

Why do we test software?

- What is your opinion?
- What would an IT professional say?
- Can you share your findings with the rest of the class?

IT professionals say they test software in order to:

- Check programs against specifications
- Determine user acceptability
- Gain confidence that it works
- Show that a program performs correctly
- Demonstrate that errors are not present
- Understand the limits of performance
- Learn what a system is not able to do
- Insure that a system is ready for use
- Evaluate the capabilities of a system
- Verify documentation
- Find important bugs, to get them fixed
- Check interoperability with other products
- Help managers make ship/no-ship decisions
- Block premature product releases
- Minimize technical support costs
- Assess conformance to specification
- Conform to regulations
- Minimize safety-related lawsuit risk
- Convince oneself that the job is finished
- Find safe scenarios for use of the product

NOTE: Different objectives require different testing strategies and will yield different tests, different test documentation and different test results.

Definition of software testing

"The process of exercising or evaluating a system by manual or automatic means to verify that it satisfies specified requirements or to identify differences between expected and actual results" [ANSI/IEEE std 729-1983]

Course learning objectives

- Understand the motivation, importance and limitations of systematic testing
- Learn and understand the pros and cons of particular testing techniques and the situations in which they apply
- Practice with modern industrial best-practices, testing tools and frameworks
- Learn how to craft and execute test cases for large software systems
- Learn how to produce quality problem reports

Basic definitions

- What is an *error?*
  - "It is a *mistake*, people make them" [Jorgensen]
  - Error of *omission*: something is missing
  - Error of *commission*: something is incorrect
- Other views
  - "A software error is present when the program doesn't do what it's user reasonably expects it to do" [Myers]
  - "A mismatch between the program and it's specifications, if and only if the specifications exist and are correct" [Kaner]
  - "There can never be an absolute definition for *bugs*, nor an absolute determination of their existence. The extend to which a program has bugs is measured by the extend to which it fails to be useful" [Beizer]

Basic definitions--continued

- What is a *fault?*
  - It is the result of an error, or the representation of an error (e.g. inaccurate requirements text, erroneous design, buggy source code etc.)
  - Synonyms used: *defect, bug*
- What is a *failure?*
  - The program's actual incorrect or missing behavior under the error-triggering conditions
  - A failure occurs when a fault executes
- A fault won't yield a failure without the *conditions* that trigger it
- What is an *incident?*
  - An incident is the symptom that alerts/indicates the occurrence of a failure (note: when a failure occurs, it may not be always apparent to the user/tester)

Example

- Here's a defective program
  - INPUT A
  - INPUT B
  - PRINT A / B
- What is the error? What is the fault?
- What is the critical condition?
- What will we see as the incident of the failure?

Example-answers

- **Error** (mistake): the mistake that the programmer made when he/she forgot to handle the special case where B=0 (division by 0 not allowed)
- **Fault** (representation of error): the resulted buggy/faulty source code
- **Critical Condition**: if B=0 then it triggers the error
- **Failure** (incorrect behavior): execution at halt or crash
- **Incident** (symptom): non-responsive screen (or error message displayed)

Basic definitions--continued

- What is a *test?*
  - The act of investigating/exercising software with an intent to
    - Find failures
    - Demonstrate correctness
    - Expose quality-related information
- What is a *test case?*
  - Set of inputs and outputs
  - Has an identity and is associated with a program behavior

When are errors introduced?

- System Life Cycle (SLC)
  - Requirements *(here)*
  - Design *(here)*
  - Implementation *(here)*
  - Maintenance *(here)*
- Snowball effect (errors introduced early can propagate into later stages and become deeper and harder to find)

- Testing Life Cycle (TLC)
  - Fault detection
  - Fault classification
  - Fault isolation
  - Fault resolution *(errors are possible here too!)*
  - *See figure 1.1 in Jorgensen*

Other related terminology*

Caution: related terminology found in the literature may vary depending on the context used

- *Software Problem* = a discrepancy between a delivered artifact of a SD phase and its:
  - documentation
  - the product of an earlier phase
  - user requirements
- *Problem Status* = a problem can be
  - *open* (i.e. the problem has been reported)
  - *closed-available* (i.e. a tested fix is available)
  - *closed* (i.e. a tested fix has been installed)
- *Error* = A problem found during the review of the phase where it was introduced
- *Defect* = A problem found later than the review of the phase where it was introduced
- *Fault* = Errors and defects are considered faults
- *Failure* = Inability of software to perform its required function
  - It can be caused by a defect encountered during software execution (i.e. testing and operation)
  - When a failure is observed, problem reports are created and analyzed in order to identify the defects that are causing the failure

**Motorola SQA and Measurements Program (Daskalantonakis, 1996)*

A broad taxonomy of software errors*

- User Interface errors
  - customers complain about serious human-factor errors as much they complain about crashes
- Error handling
  - Failure to detect and handle an error in a reasonable way
- Boundary-related errors
  - Typically numeric values and memory size
- Calculation errors
  - Incorrect arithmetic due to truncation, misinterpreted formulas, incorrect algorithms
- Errors in handling or interpreting data
  - Passing corrupted data from one module to another
- Race conditions
  - Handling of asynchronous events (e.g. event B has to happen after event A for some reason)
- Load conditions
  - Program misbehaves when overloaded
- Hardware
  - Programs send bad data to devices, and try to use devices that are busy or don't exist
- Source and version control
  - Old problems reappear when we link an older version with the latest one
- Documentation
  - Poor documentation leads to mistrust of the software's capabilities
- Testing errors
  - Testers make mistakes too!

*Cem Kaner et al, *"Testing Computer Software"* (http://www.testingeducation.org/)

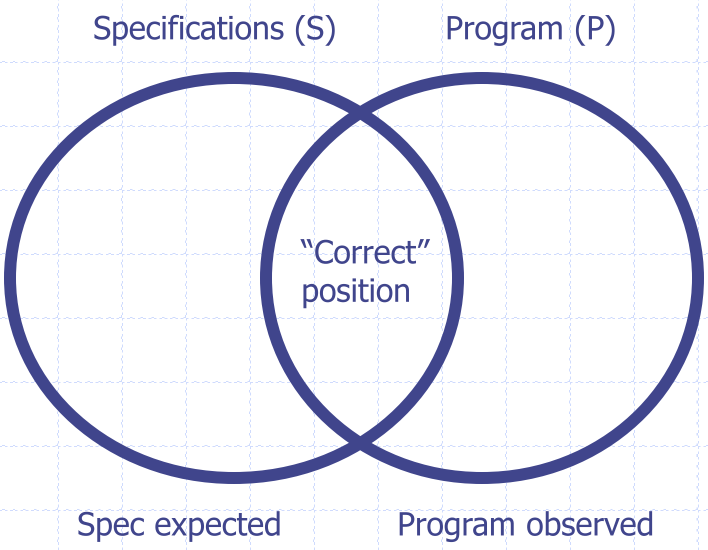
Anatomy of a test case

- Test case ID
- Purpose and objectives
- Inputs
  - Preconditions
  - Actual data inputs that were identified by some testing method
- Expected outputs
  - Post conditions
  - Actual outputs

Views of testing

- Behavioral view
  - It considers “what the code does i.e. how it behaves”
- Structural view
  - It focuses on “what it is”
  - Base documents are written by and for developers – the emphasis is on structural info rather than behavioral info

Specified and implemented program behaviors



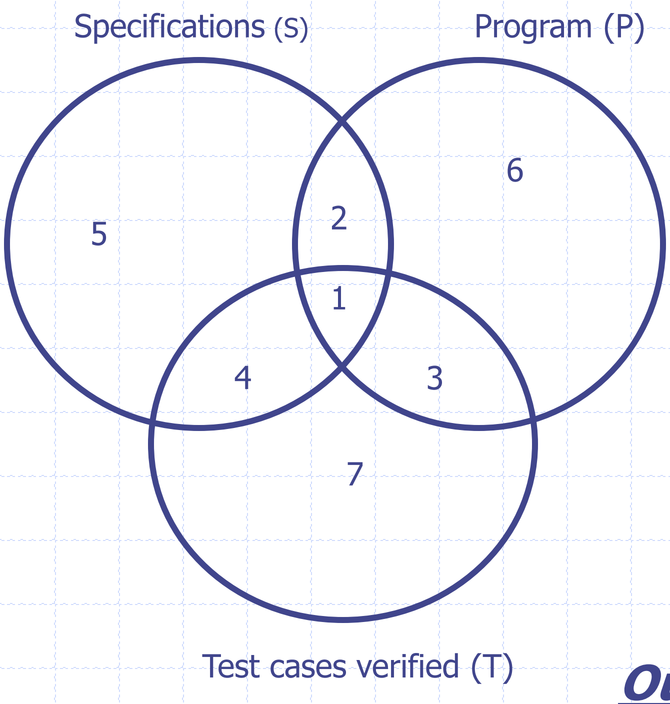
The Venn diagram shows

1) certain specified behaviors are not programmed
   - Faults of omission
2) certain programmed behaviors were never specified
   - Faults of commission

The *football* shape is the correct position

- Behaviors are both specified and implemented

Specified, implemented and tested program behaviors



Regions 2 and 5 are specs that are never tested

Regions 1 and 4 are specified and tested

Regions 3 and 7 are test cases that are not specified

Regions 2 and 6 are program behaviors that are not tested

Regions 1 and 3 are program behaviors that are tested

Regions 4 and 7 are test cases that correspond to non programmed behaviors

Our goal: How can we make region 1 larger?

A simple example

Specs for an ADDER:

- Adds two numbers that the user enters
- Each number should be one or two digits
- The program echoes the entries, then prints the sum.
- Press <ENTER> after each number

test run output

```
?4
4
?2
2
6
?
```

First reactions? Observations?

- Can you list some potential areas for improvement on this application?

First observations

- No on-screen instructions
- Nothing shows what this program is or does
- You don’t even know if you run the right program
- How do you stop the program?
- The 6 should probably line up with the 4 and 2

A first set of test cases

Can you suggest some test cases you would like to try in order to test this application?

A first set of test cases

99 + 99

99 + 56

99 + -14

38 + -99

-99 + -43

9 + 9

0 + 23

-99 + -99

56 + 99

-14 + 99

-99 + 38

-43 + -99

0 + 0

-23 + 0

Choosing test cases

- Not all test cases are significant
- Impossible to test everything (this simple program has tens of thousands of possible different test cases)
- If you expect the same result from two tests, they belong to the same class. Use only one of them
- When you choose representatives of a class for testing, pick the ones most likely to fail

Further test cases

100 + 100

<Enter> + <Enter>

123456 + 0

1.2 + 5

A + b

<CTRL-C> + <CTRL-D>

<F1> + <Esc>

Other things to consider

- Test cases with extra whitespace
- Test cases involving <Backspace>
- The order of the test cases might matter
  - E.g. <Enter> + <Enter>

What is functional testing?

- Based on a view that any program can be considered to be a function that maps values from its input domain to values onto its output range
- *Black Box* testing
  - Systems are considered to be black boxes
  - Content or implementation is unknown
  - The function of the black box is completely understood in terms of its inputs and outputs

What are the advantages?

- Functional test cases have two distinct advantages:
  1) They are independent of how the software is implemented
     - If implementation changes the test cases are still useful
  2) Test case development can occur in parallel with the implementation, thereby reducing overall project development interval

What are the disadvantages?

- Functional test cases have the following disadvantages:
  1) Significant redundancies may exist among test cases
  2) There is a possibility of gaps of untested software

What is structural testing?

- *White box* or *clear box* testing
- Implementation is known and used to identify test cases
- Testers identify test cases based on how the function is actually implemented

Which approach is better?

- Neither approach alone is sufficient; both are needed
- The two approaches are complementary

Software Quality Assurance (SQA) vs. Software Testing

- What is the difference in your opinion?

Software Quality Assurance (SQA) vs. Software Testing

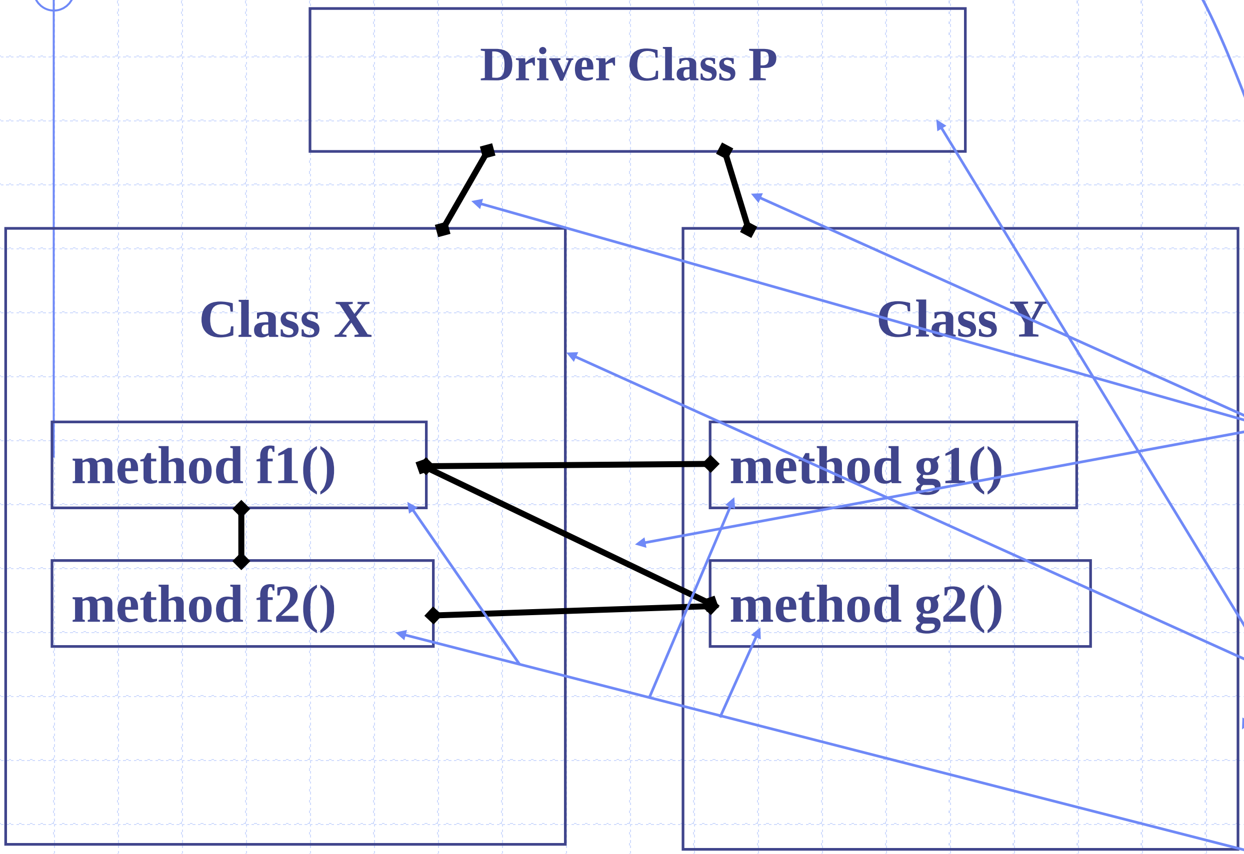
- What is a *process?*
  - how we do something
- What is a *product?*
  - The end result of a process
- <u>Testing</u> is clearly product-oriented
  - It finds *faults* in a product
- <u>SQA</u> tries to improve the product by first improving the process
  - It wants to fix *errors* in the development process

Software anomalies*

- Fault Severity
  - Mild
  - Moderate
  - Annoying
  - Disturbing
  - Serious
  - Very serious
  - Extreme
  - Intolerable
  - Catastrophic
  - Infectious
- Fault types
  - Input/output
  - Logic
  - Computation
  - Interface
  - Data

*ANSI/IEEE std 1044-1993 "*Classification of Software Anomalies*"

Levels of Testing*



- **Acceptance testing: Is the software acceptable to the user?**
- **System testing: Test the overall functionality of the system**
- **Integration testing: Test how modules interact with each other**
- **Module testing: Test each class, file, module or component**
- **Unit testing: Test each unit (method) individually**

*Paul Ammann and Jeff Offutt, *"Introduction to Software Testing"*, Cambridge University Press

Granularities of testing

- Testing-in-the-small
  - It refers to the testing activities of a single module/unit
- Testing-in-the-large
  - It refers to the testing activities of groups of modules/units up-to and including the entire system (integration testing techniques)

Other dimensions of software testing

Integration testing
(performance, stress, functional, structural)
Acceptance testing (*alpha:* first testing phase, internal;
beta: second testing phase, usually public)

Testing-in-the-large

Static Testing

Testing

Dynamic Testing

Desk checking
Hand execution
Inspections
Formal proof
Program slicing

Program execution is necessary

Testing-in-the-small

Unit testing (Black box, White box)

Typical steps involved in testing techniques

1. Select *what* is to be measured by the test
2. Decide *how* is to be tested (walkthrough, inspection, proof, BB/WB, etc.)
3. Develop a *test-bed* (a set of test cases)
4. Create the *test oracle* (predicted results for a set of test cases)
5. Execute the test cases
   - Prepare the *test harness* software
   - Compare the results with the test oracle
   - Identify any discrepancies between the predicted results and the actual results

What is white box testing?

It is a test-bed design method, which uses the control-flow structure of the program to derive the necessary test cases.

How does WB testing work?

- *Coverage* is a measure of how much of a module or system has been exercised (executed) by a test or series of tests
- To obtain coverage, we make sure every statement in the source code being tested, is executed at least once
  - Segments (sequential)
  - Decisions (if-then-else is executed twice)
  - Loops (provide test cases for skipping the execution of a loop, execute the body exactly once, or more than once)

Tools

Eclipse

- IDE for Java development
- Works seamlessly with Junit for unit testing
- Open source – Download from www.eclipse.org
- Try it with your own Java code

Junit

- A framework for automated unit testing of Java code
- Written by Erich Gamma and Kent Beck
- Download from www.junit.org
- Related technical manuals available on web

What is black box testing?

It is a test-bed design method, which focuses on the behavioral requirements of the program to derive the necessary test cases.

How does BB testing work?

- *Equivalence partitioning*
  - The goal is to reduce the number of necessary test cases to a manageable number
- Every input condition is divided into a number of *equivalence classes*.
  - Each class consists of a set of data items all of which are similar to each other on some relevant dimension
  - Example 1: If the input data is supposed to be valid across a range of values (e.g. an age, a price), there will be a minimum of three equivalence classes:
    - *Below*, *within* and *above* the range
  - Example 2: If the input data is valid when it is a value from a set of discrete or nominal values, (e.g. letter grades) there will be two equivalence classes:
    - One with valid discrete values
    - One with any other input values

Glen Myers triangle problem

"The program reads three integer values from a card. The three values are interpreted as representing the lengths of the sides of a triangle. The program prints a message that states whether the triangle is scalene, isosceles, or equilateral."

[Taken From Glen Myers, "The Art of Software Testing"]

Classes of test cases for the triangle problem

- Can you find some candidate categories of test cases for this problem?

Hint: Refined Specs

- *Input:* Three integers, a, b, c, the lengths of the side of a triangle
- *Output:* Scalene, isosceles, equilateral, or invalid triangle

Reminder: triangle properties

- No side may have a length of zero
- Sum of two sides must be greater than the other side of the triangle
- <u>Equilateral</u> triangle: all three sides are equal
- <u>Isosceles</u> triangle: any two sides are equal
- <u>Scalene</u> triangle: all sides are unequal

Classes of test cases for the triangle problem

- Valid triangle:
  - Test case for a valid scalene triangle
  - Test case for a valid equilateral triangle
  - Three test cases for valid isosceles triangles try (a=b, c), (a, b=c), (a=c, b)
- Invalid triangle:
  - All permutations of a + b = c (e.g. a=1, b=2, c=3) try 3 permutations a+b=c, a+c=b, b+c=a
  - All permutations of a + b < c (e.g. a=1, b=2, c=4) try 3 permutations
  - All permutations of a = b and a + b = c (e.g. a=3, b=3, c=6)
- Invalid side values:
  - One, two or three sides has zero value (5 cases)
  - One side has a negative
  - MAXINT values
  - Non-integer
  - Wrong number of values (too many, too few)

Caveat

- The triangle problem typifies some of the *incomplete definitions and assumptions* that impair communication among customers, developers, and testers
- Glen Myers specification assumes the developers know some details about triangles
  - ***Triangle properties:***
    - sum of two sides must be greater than the other side of the triangle
    - No side may have a length of zero

Object-oriented implementation

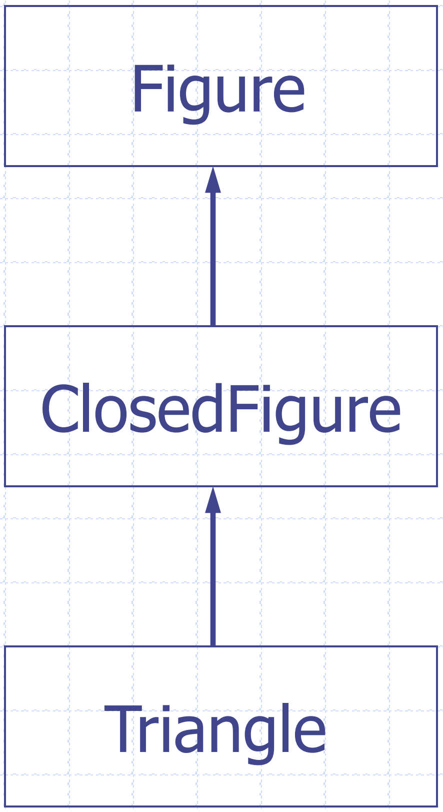
```
class Triangle{
  public Triangle(LineSegment a, LineSegment b,
                              LineSegment c)
  public boolean is_isosceles()
  public boolean is_scalene()
  public boolean is_equilateral()
  public void draw()
  public void erase()
}
class LineSegment {
  public LineSegment(int x1, int y1,
                     int x2, int y2)
}
```

Extra Tests

- Is the constructor correct?
- Is only one of the **`is_*`** methods true in every case?
- Do results repeat, e.g. when running **`is_scalene`** twice or more?
- Results change after **`draw`** or **`erase`**?
- Segments that do not intersect

Inheritance tests

- Tests that apply to all *Figure* objects must still work for *Triangle* objects
- Tests that apply to all *ClosedFigure* objects must still work for *Triangle* objects



What is data-structure based testing?

- It involves looking at the data structures used by the module with an eye toward errors that might be related to the structure being used
- For instance, an array (or linked-list) is being passed to the module being tested. Then, four test cases should be designed as follows:
    1. Zero elements in the array/list
    2. Exactly one element in the array/list
    3. One less than the maximum number of elements
    4. Maximum number of elements in the array/list

Integration testing activities

- Structure Testing
  - Exercise all I/O parameters for each module
  - Exercise all module calls including utility routines
- Functional Testing
  - Make sure that all functions are operational
- Performance Testing
  - Determine the amount of execution time needed to carry out different routine functions (is it within reasonable amount of time?)
- Stress Testing
  - Push integrated units to their limits (amount of load it can take)

Testing limits

- Dijkstra: "Program Testing can be used to show the presence of defects, but never their absence"
- It is impossible to fully test a software system in a reasonable amount of time or money
- When is testing complete?
  - When you run out of time or money

Complete testing?

- What do we mean by "complete testing"?
  - Complete *coverage* i.e. tested every line/path?
  - Testers not finding new bugs?
  - Test plan completed?
- Complete testing must mean that, at the end of testing, you know there are no remaining unknown bugs
- After all, if there are more bugs, you can find them if you do more testing. So testing couldn't yet be "complete"

Complete coverage?

- What is *coverage?*
  - Extent of testing of certain attributes or pieces of the program, such as *statement* coverage or *branch* coverage or *condition* coverage
  - Extent of testing completed, compared to a population of possible tests
- Why is complete coverage impossible?
  - Domain of possible inputs is too large
  - Too many possible paths through the program
- Coverage measurement is a good tool to show *how far* you are from complete testing and not to show *how close* you are to completion

Time-consuming test-related tasks

- Analyzing, troubleshooting, and effectively describing a failure
- Also
  - Designing tests
  - Executing tests
  - Documenting tests
  - Automating tests
  - Reviews, inspections
  - Training other staff

The infinite set of tests

- There are enormous numbers of possible tests. To test everything, you would have to:
  - Test every possible input to every variable
  - Test every possible combination of inputs to every combination of variables
  - Test every possible sequence through the program
  - Test every hardware / software configuration, including configurations of servers not under your control
  - Test every way in which any user might try to use the program

Testing valid inputs

- There are 39,601 possible valid inputs for the ADDER program
- In the Triangle example, assuming only integers from 1 to 10, there are 10^4 possibilities for a segment, and 10^{12} for a triangle. Testing 1000 cases per second, you would need 317 years!

Testing invalid inputs

- The error handling aspect of the system must also be triggered with invalid inputs
- Anything you can enter with a keyboard must be tried. Letters, control characters, combinations of these, question marks, too long strings etc…

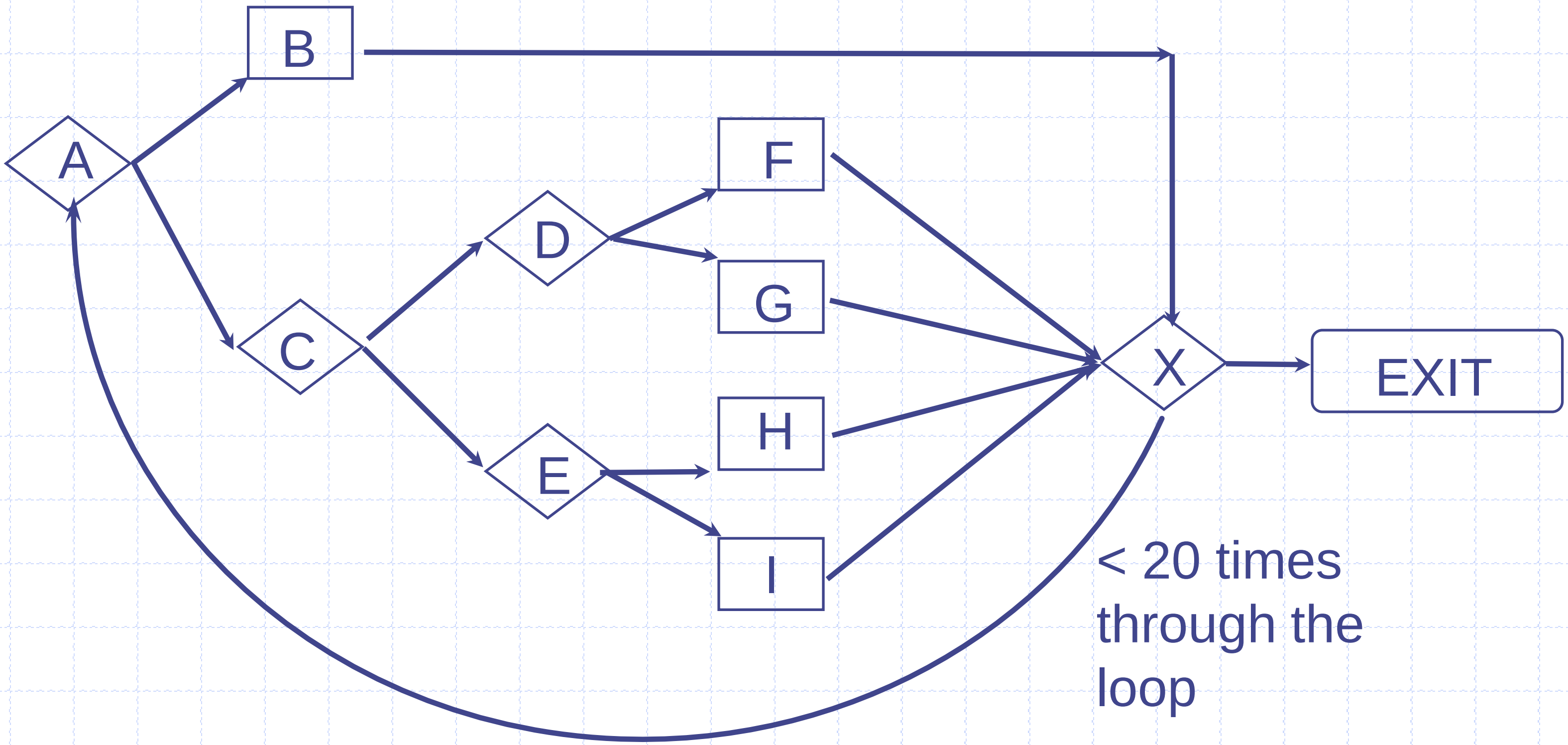
Testing edited inputs

- Need to test that editing works (if allowed by the spec)
- Test that any character can be changed into any other
- Test repeated editing
  - Long strings of key presses followed by <Backspace> have been known to crash buffered input systems

Testing input timing variations

- Try entering the data very quickly, or very slowly
- Do not wait for the prompt to appear
- Enter data before, after, and during the processing of some other event, or just as the time-out interval for this data item is about to expire
- Race conditions between events often leads to bugs that are hard to reproduce. For instance, handling of asynchronous events (e.g. event B has to happen after event A for some reason)

Testing all paths in the system



Here's an example that shows that there are too many paths to test in even a fairly simple program. (Taken from Myers, *The Art of Software Testing.)*

Number of paths

- One path is ABX-Exit. There are 5 ways to get to X and then to the EXIT in one pass.
- Another path is ABXACDFX-Exit. There are 5 ways to get to X the first time, 5 more to get back to X the second time, so there are 5 x 5 = 25 cases like this.
- There are $5^1 + 5^2 + ... + 5^{19} + 5^{20} = 10^{14}$ = 100 trillion paths through the program.
- It would take only a billion years to test every path (if one could write, execute and verify a test case every five minutes)

Further difficulties for testers

- Testing cannot verify requirements. Incorrect or incomplete requirements may lead to spurious tests
- Bugs in test design or test drivers are equally hard to find
- Expected output for certain test cases might be hard to determine

Conclusion

- It is impossible to completely test any non-trivial software module or system; therefore testers live and breathe tradeoffs
- Testing should be performed with the intention of finding errors
- Testing takes creativity and hard work
- Testing is best done by several independent testers