# EECS 4313
# Software Engineering: Testing

Vassilios Tzerpos
bil@cse.yorku.ca

## What is testing?

**A technical investigation done to expose quality-related information about the product under test**

## Defining Testing

- **A technical**
  - Logic, mathematics, models, tools
- **investigation**
  - An organized and thorough search for information.
  - We ask hard questions (aka run hard test cases) and look carefully at the results.
- **done to expose quality-related information**
  - see the next slide
- **about the product under test.**

## Information Objectives

- 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
- Find safe scenarios for use of the product

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

## Our goal

- Learn testing techniques and the situations in which they apply
- Practice with real testing tools and frameworks
- Learn how to produce quality problem reports
- Study special issues for object-oriented systems
- Understand the importance of systematic testing

## Tools - Eclipse

- IDE for Java development
- Works seamlessly with Junit for unit testing
- Open source – Download from www.eclipse.org
- In the lab, do: `eclipse`
- Try it with your own Java code

## Tools - Junit

- A framework for automated unit testing of Java code
- Written by Erich Gamma (of Design Patterns fame) and Kent Beck (creator of XP methodology)
- Uses Java features such as annotations and static imports
- Download from www.junit.org
- Integrated with Eclipse

## A first example

- Test 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
- Screen for a test run

```
? 2
? 3
5

?
```

## Immediate issues

- Nothing shows what this program is. You don't even know you run the right program.
- No on-screen instructions.
- How do you stop the program?
- The 5 should probably line up with the 2 and 3.

## A first set of test cases

| | |
|---|---|
| 99 + 99 | -99 + -99 |
| 99 + 56 | 56 + 99 |
| 99 + -14 | -14 + 99 |
| 38 + -99 | -99 + 38 |
| -99 + -43 | -43 + -99 |
| 9 + 9 | 0 + 0 |
| 0 + 23 | -23 + 0 |

## Choosing test cases

- Not all test cases are significant.
- Impossible to test everything (this simple program has 39,601 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

- Storage for the two inputs or the sum
  - 127 or 128 can be an important boundary case
- Test cases with extra whitespace
- Test cases involving <Backspace>
- The order of the test cases might matter
  - E.g. <Enter> + <Enter>

## An object-oriented example

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

## Test case classes

- Valid scalene, isosceles, equilateral triangle
- All permutations of two equal sides
- Zero or negative lengths
- All permutations of a + b < c
- All permutations of a + b = c
- All permutations of a = b and a + b = c
- MAXINT values
- Non-integer inputs

## Example implementation

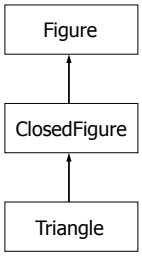
```
class Triangle{
  public Triangle(LineSegment a, LineSegment b,
                                 LineSegment c)
  public boolean is_isosceles()
  public boolean is_scalene()
  public boolean is_equilateral()
  public void draw()
  public void erase()
}
class LineSegment {
  public LineSegment(int 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 or form an interior triangle

## 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



## 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": Tested every line/path?
  - Testers not finding new bugs?
  - Test plan complete?
- 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.

## Measuring and achieving high coverage

- Coverage measurement is a good tool to show **how far** you are from complete testing.
- But it’ s a lousy tool for investigating how close you are to completion.

## Testers live and breathe tradeoffs

- The time needed for test-related tasks is infinitely larger than the time available.
- Example: Time you spend on
  - Analyzing, troubleshooting, and effectively describing a failure
- Is time no longer available for
  - Designing tests
  - Executing tests
  - Reviews, inspections
  - Documenting tests
  - Automating tests
  - 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 (an example)

- MASPAR is a parallel computer used for mission-critical and life-critical applications.
  - To test the 32-bit integer square root function, all 4,294,967,296 values were checked. This took 6 minutes.
  - There were 2 (two) errors, neither of them near any boundary.
    - The underlying error was that a bit was sometimes mis-set, but in most error cases, there was no effect on the final calculated result.
  - Without an exhaustive test, these errors probably wouldn't have shown up.
  - What about the 64-bit integer square root? How could we find the time to run all of these?

## Testing valid inputs

- There were 39,601 possible valid inputs in ADDER
- 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

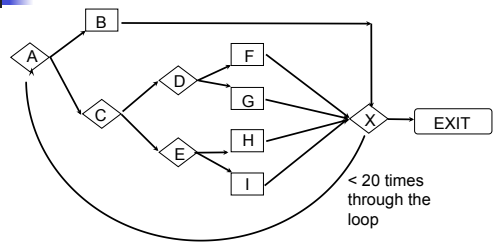
## Combination testing

- Example 1: a program crashed when attempting to print preview a high resolution (back then, 600x600 dpi) output on a high resolution screen. The option selections for printer resolution and screen resolution were interacting.
- Example 2: American Airlines couldn't print tickets if a string concatenating the fares associated with all segments was too long.
- Example 3: Memory leak in WordStar if text was marked Bold/Italic (rather than Italic/Bold)

## What if you don't test all possible inputs?

- Based on the test cases chosen, an implementation that passes all tests but fails on a missed test case can be created.
- If it can be done on purpose, it can be done accidentally too.
  - A word processor had trouble with large files that were fragmented on the disk (would suddenly lose whole paragraphs)

## 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. This is 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

- Complete testing is impossible
  - *There is no simple answer for this.*
  - *There is no simple, easily automated, comprehensive oracle to deal with it.*
  - *Therefore testers live and breathe tradeoffs.*