# EECS 4313
Software Engineering Testing

## Topic 01:
## Limits and objectives of software testing

Zhen Ming (Jack) Jiang

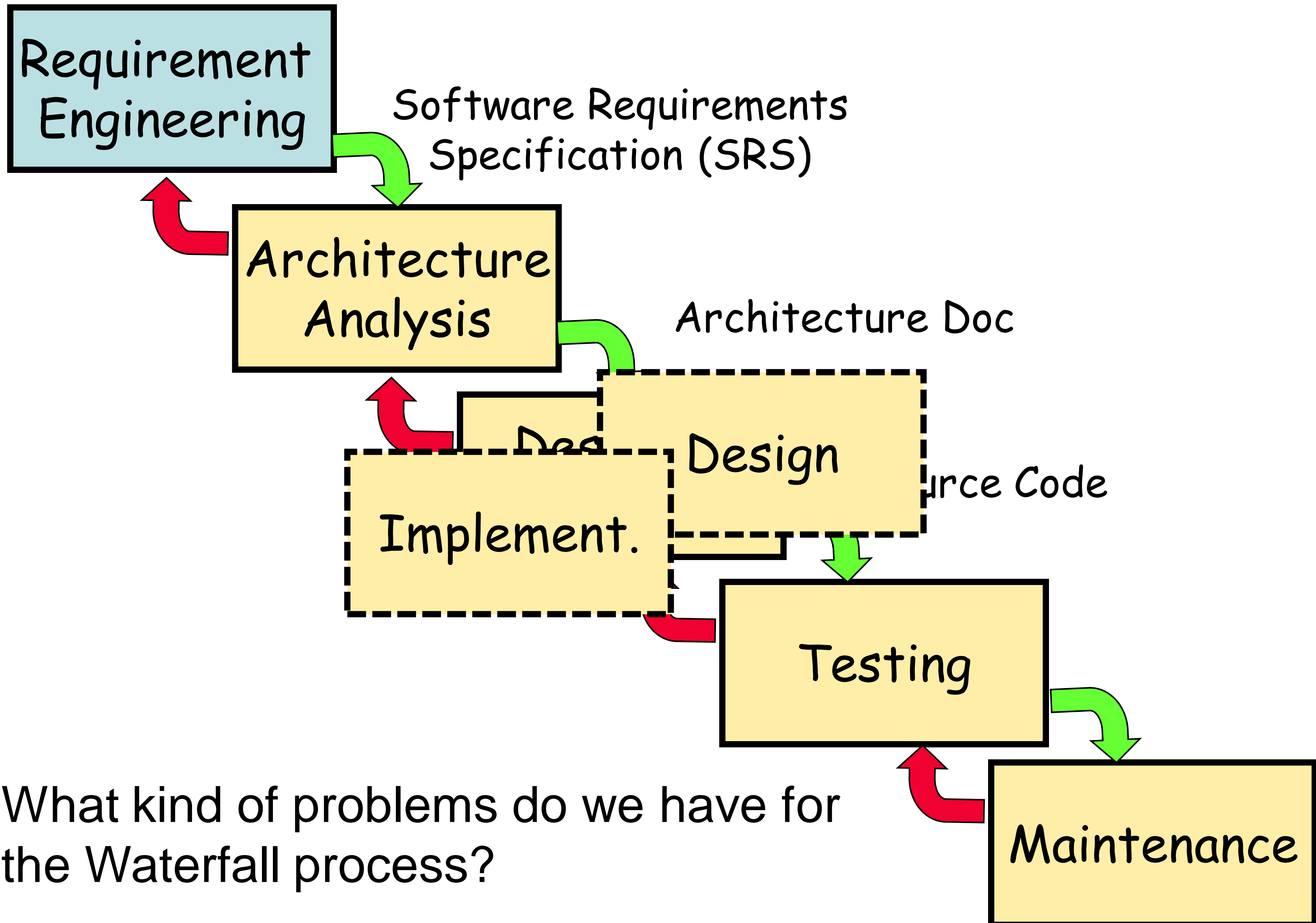
# Acknowledge

- Some of the contents are from Prof. Alex Orso, Bil Tzerpos and Gunnar Gotshalks

# Relevant Readings

- [Jorgensen] chapter 1 & 2 (for background), chapter 5, section 23.3

# Waterfall Development Process




What kind of problems do we have for the Waterfall process?

Software bugs are costing the US Economy an estimated $60 billion each year. Improvements in verification and validation could reduce this cost by about a third (i.e., $20 billion).

*[NIST Estimated Planning Report 2002 – 10]*

# Software is Buggy!

- On average, 1-5 errors per 1 KLOC
- Windows 2000
  - 35 MLOC
  - 63,000 known bugs at the time of release
  - On average, 2 bugs for every 1,000 lines
- For mass market software 100% correct is infeasible, but we must verify the software as much as possible

# US prisoners released early by software bug

23 December 2015 | Technology

The bug meant that, in one case, a prisoner was released almost two years early

**More than 3,200 US prisoners have been released early because of a software glitch.**

The bug miscalculated the sentence reductions prisoners in Washington state had received for good behaviour.

# Failure, fault, error and incident

- Failure
  - Observable incorrect **behavior** of a program. Conceptually relate to the behavior of the program, rather than its code
    - A failure occurs when a fault executes
- Fault (bug)
  - Related to the **code**. Necessary (not sufficient) condition for the occurrence of a failure
    - A fault is the result of an error
    - A fault won’t yield a failure without the **conditions** that trigger it.
    - Example: if the program yields 2+2=5 on the 10th time you use it, you won’t see the error before or after the 10th use.
- Error
  - **Cause** of a fault. Usually a human error (conceptual, typo, etc.)
    - “People make errors”
- Incident
  - the symptom associated with a failure that alerts the user to the occurrence of a failure

# An Example of the failure, fault, error

```
1. int double (int param) {
2.     int result;
3.     result = param * param;
4.     return result;
5. }
```

A call to double(3) returns 9

- Result 9 represents a **failure**
- Such failure is due to the **fault** at line 3
- The **error** is a typo (hopefully)

# Approach to Verification

- Testing
  - Exercising software to try and generate failures
- Static verification
  - Identify (specific) problems statically, that is considering all possible executions
- Inspection/review/walkthrough
  - Systematic group review of program text to detect faults
- Formal proof
  - Proving that the program text implements the program specification

# Comparison

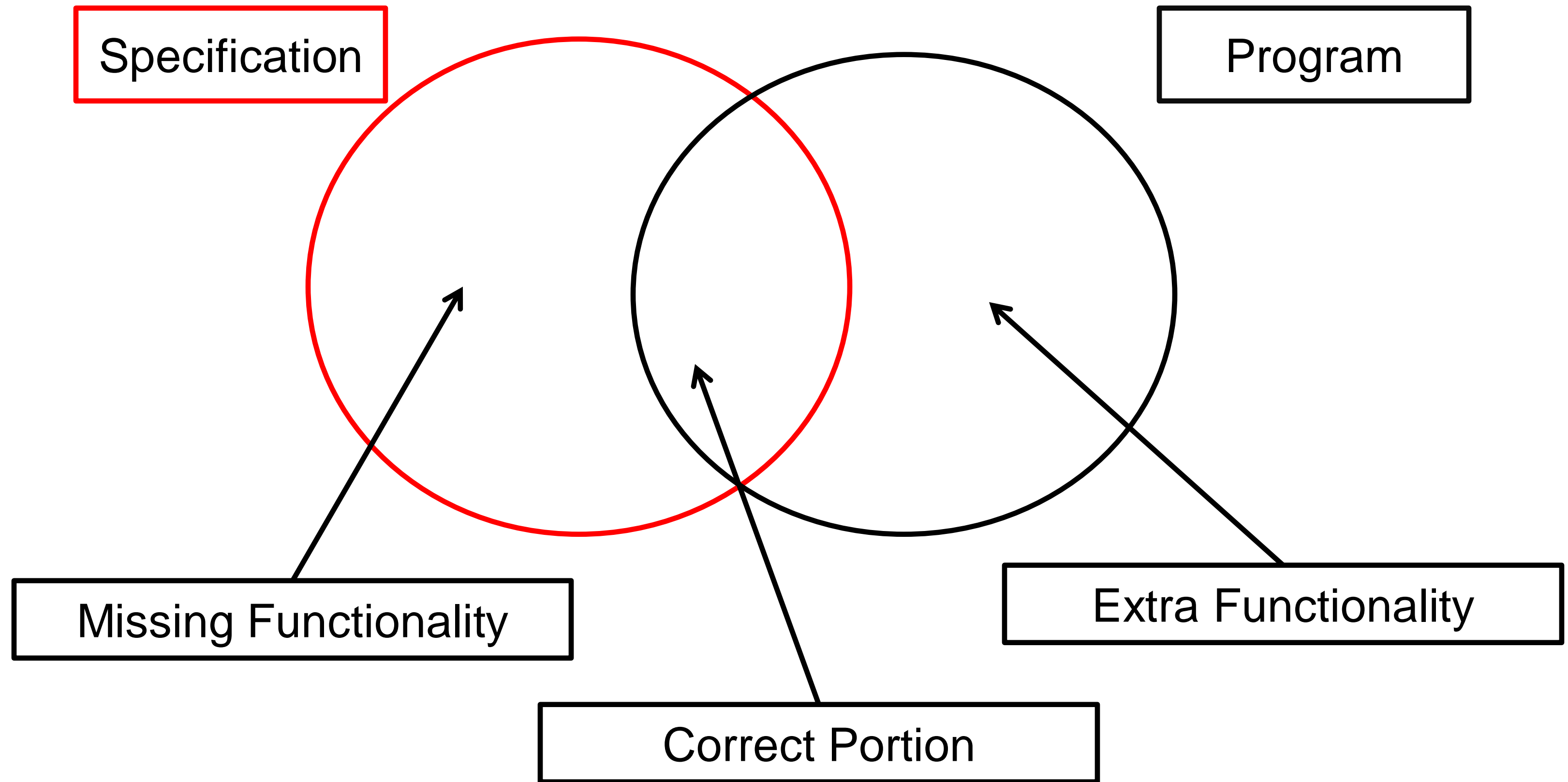
- Testing
  - Pros: no false positives
  - Cons: incomplete
- Static verification
  - Pros: complete (consider all program behavior)
  - Cons: false positive (main issue), expensive
- Inspection
  - Pro: systematic, thorough
  - Cons: informal, subjective
- Formal proof (of correctness)
  - Pro: strong guarantees
  - Cons: complex, expensive (requires a specification)

# Today, QA is mostly testing

"50% of my company employees are testers and the rest spends 50% of their time testing"

*Bill Gates, Microsoft*

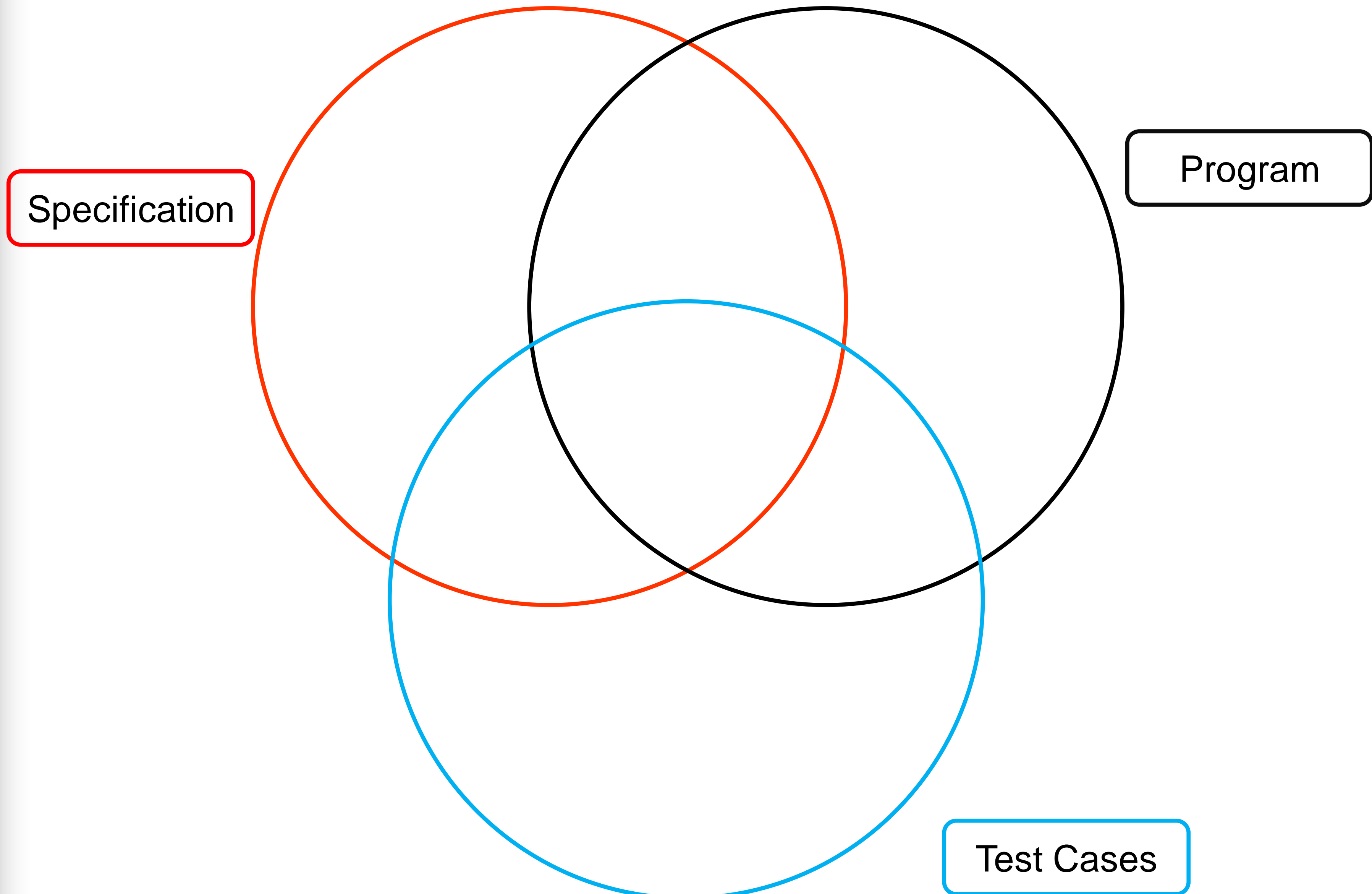
# Program Behaviour



# Correctness

- Impossible to demonstrate
- Better viewpoint
  - Program P is correct with respect to specification S
- Do the specification and the program meet the customer/user's expectations?
- Test can never reveal the absence of a fault

# Testing Program Behaviour



# What is testing?

- Testing
  - Execute a program with a sample of the input data
- Dynamic technique
  - Program must be executed
- Test cases
  - Test method is a repeatable way to generate test cases
- Optimistic approximation
  - The program under test is exercised with a (very small) subset of all the possible input data
  - We assume that the behavior with any other input is consistent with the behavior shown for the selected subset of input data

# How Volkswagen Is Grappling With Its Diesel Scandal

By GUILBERT GATES, JACK EWING, KARL RUSSELL and DEREK WATKINS UPDATED Dec. 20, 2016

Volkswagen has admitted that 11 million of its vehicles were equipped with software that was used to cheat on emissions tests. The company is now contending with the fallout. RELATED ARTICLE

**Writing software with the WRONG specification!**

## How Did the System Work?

The software sensed when the car was being tested and then activated equipment that reduced emissions, United States officials said. But the software turned the equipment down during regular driving, increasing emissions far above legal limits, most likely to save fuel or to improve the car's torque and acceleration.

The software was modified to adjust components such as catalytic converters or valves used to recycle some of the exhaust gasses. The components are meant to reduce emissions of nitrogen oxide, a pollutant that can cause emphysema, bronchitis and other respiratory diseases.

# Testing techniques

- There are a number of techniques. Each has different processes, artifacts, or approaches
- There are no perfect techniques
  - Testing is a best effort activity
- There is no best technique
  - Different contexts
  - Complementary strengths and weakness
  - Trade-offs

# Basic Approaches

- Black Box (Functional Testing)
  - based on a description of the software (specification)
  - covers as much specified behavior as possible
  - cannot reveal errors due to implementation details
- White Box (Structural Testing)
  - based on the code
  - covers as much coded behavior as possible
  - cannot reveal errors due to missing paths
- Grey Box

# Content of a Test Case

- “Boilerplate”: author, date, propose (summary), test case ID, reference to specification, version
- Pre-conditions (including environment)
- Inputs
- Expected Outputs
- Observed Outputs
- Pass/Fail

# Testing Levels

- Unit Testing
  - assess software with respect to implementation
- Module Testing
  - assess software with respect to detail design
- Integration Testing
  - assess software with respect to subsystem design
    - “big bang”
- System Testing
  - assess software with respect to architecture design
    - *ility: Reliability, maintainability, usability
- Acceptance Testing
  - assess software with respect to requirements
    - Against customer requirements
- Regression testing
  - ensures the new changes not breaking the functionalities of the existing code

# Two unit tests, zero integration tests

Purpose of Testing

# Beizer’s testing levels on test process maturity

- There are four levels of maturity!
  - Fundamental differences!
    - In viewpoint!
    - Effect on the individual!
    - Effect on the organization!
    - Effect on developed systems

# Beizer's testing levels on test process maturity

- Level 4
  - Testing is a mental discipline that helps all IT professionals develop higher quality software!
- Level 3
  - Purpose of testing is not to prove anything specific but to reduce the risk of using the software
- Level 2
  - Purpose of testing is to show that software doesn't work
- Level 1
  - Purpose of testing is to show that software works
- Level 0
  - No difference between testing and debugging

# Level 0

- No difference between testing and debugging!
  - Adopted by undergraduate CS students!
    - Get their programs to compile!
    - Debug with few arbitrarily chosen inputs or those provided by the instructor!
- Does not distinguish between incorrect program behavior and programming mistakes!
- Does little to help develop programs that are reliable or safe

# Level 1

- Purpose of testing is to show that software works
  - Significant step up
  - But correctness is virtually impossible to either achieve or demonstrate
    - Run test suite with no failures
    - Is program correct?
    - Do we have bad tests?
- Test engineers have no strict goal, real stopping rule or formal test technique
- Test managers are powerless because they have no way to quantitatively express or evaluate their work

# Level 2

- Purpose of testing is to show that the software doesn’t work
  - Valid but negative goal!
  - Testers may like it but developers do not
    - Level 1 thinking is natural for developers
  - Have adversarial relationship
    - Bad for team morale
    - Conflict of interest if the same person
  - What to do if no failures are found?
    - Is software good?
    - Is testing bad?
  - Having confidence when testing is complete is an important goal

# Level 3

- Purpose of testing is not to prove anything specific but to reduce the risk of using the software
  - Realize that testing can show the presence of failures but not their absence. (*Edsger W. Dijkstra*)
  - Accept fact that using software incurs some risk
    - May be small with unimportant consequences
    - May be big with important consequences, or even catastrophic
  - Entire team wants the same thing
    - Reduce the risk
    - Developers and testers work together

# Level 4

- Testing is a mental discipline that helps all IT professionals develop higher quality software
  - Testing is a mental discipline that increases quality
  - Testers become technical leaders of projects
  - Primary responsibility is measuring and improving software quality
  - Improve the ability of developers to produce quality software
    - Testers train developers

Essence of Testing

# Information Objectives of Software Testing

- 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
# - Learning objectives

- Learn testing techniques and the situations in which they apply
- Apply real-world testing tools and frameworks
- Learn how to file bug reports
- Understand and apply different manual and automated software testing techniques
- 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 within (-100,100) 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
  - 198 or -198 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 - The triangle problem

- 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

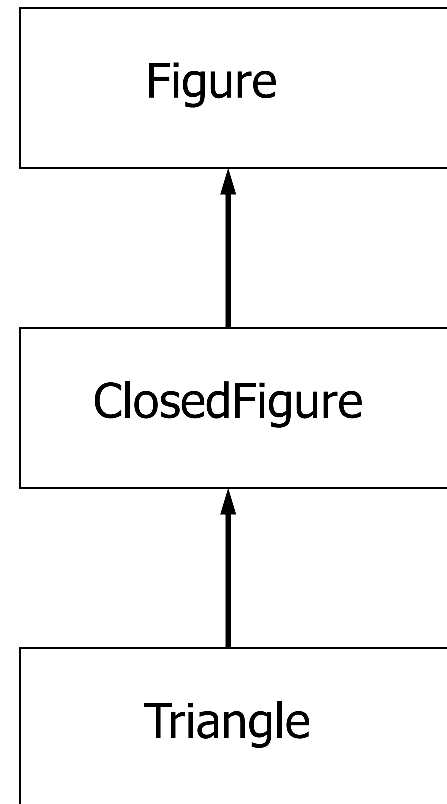
```
class Triangle{
  public Triangle(LineSegment a, LineSegment b,
                                LineSegment c)
  public boolean is_isosceles()
  public boolean is_scalene()
  public boolean is_equilateral()
  public void draw()
  public void erase()
}
class LineSegment {
  public LineSegment(int 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, do they have the same results?
- 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

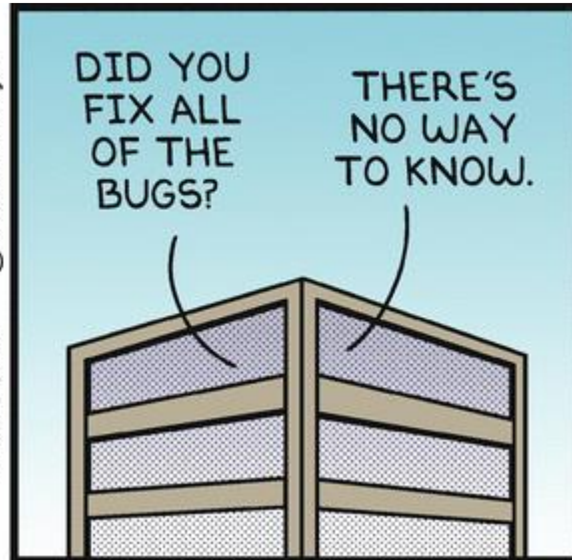


# Testing limits

- 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.”

# Software is never finished

Dilbert.com @ScottAdamsSays




10-02-17 © 2017 Scott Adams, Inc./Dist. by Andrews McMeel

# 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 code 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: The time you spend on
  - Analyzing, troubleshooting, and effectively describing a failure
- Is time no longer available for
  - Designing tests
  - Documenting tests
  - Executing 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 (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 input

- 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

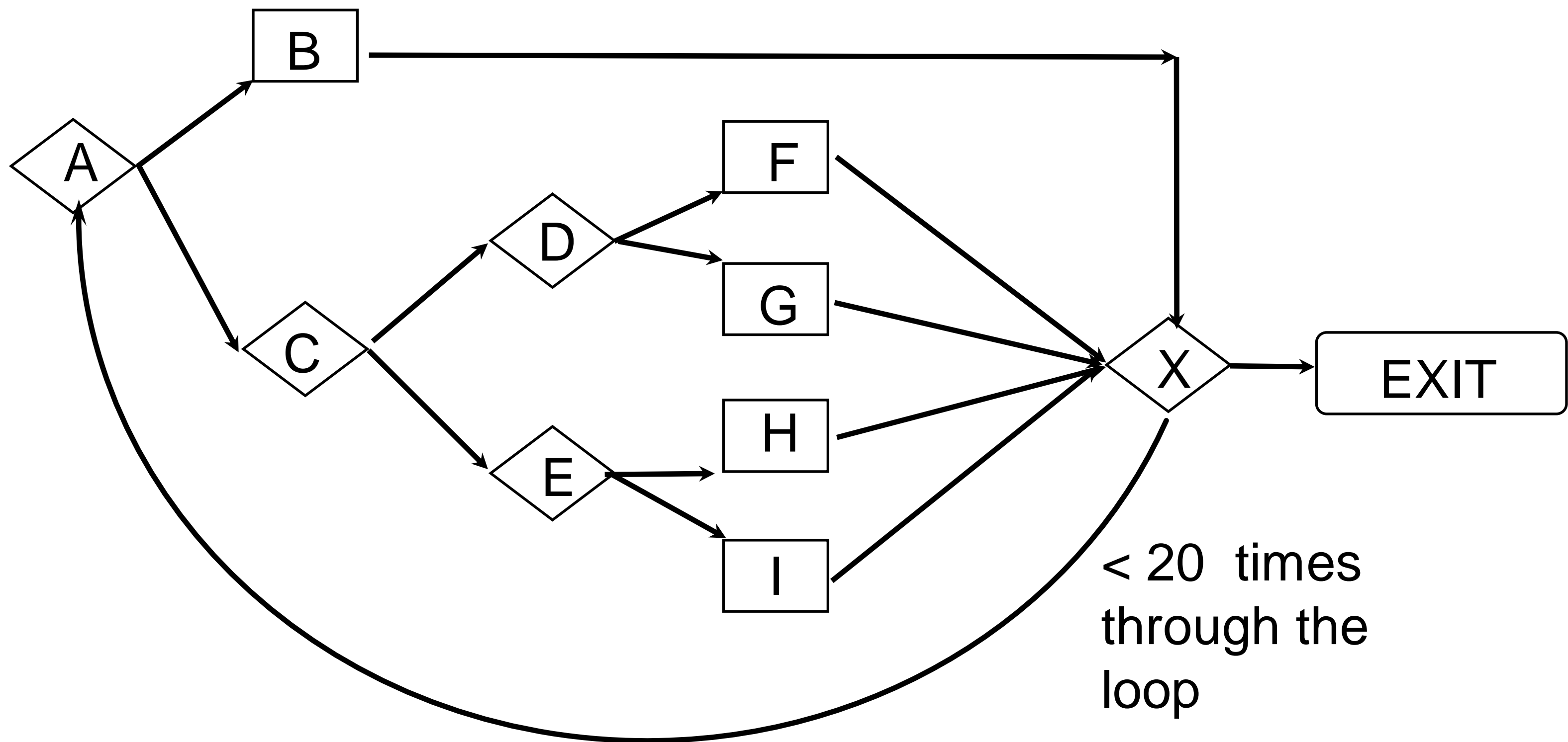
# Combination testing

- Example 1: Apache webserver has 172 user configuration parameters (158 binary options). This system has $1.8 \times 10^{55}$ possible configurations to test!
- Example 2: American Airlines could not 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.*