You know ...

• ... Software requirements
• ... Software design
• ... Software coding

• ...

DILBERT by Scott Adams

YOUR USER REQUIREMENTS INCLUDE FOUR HUNDRED FEATURES.

DO YOU REALIZE THAT NO HUMAN WOULD BE ABLE TO USE A PRODUCT WITH THAT LEVEL OF COMPLEXITY?

GOOD POINT. I'D BETTER ADD "EASY TO USE" TO THE LIST.
Software Testing
Outline

• **Software Testing?**
• **Why Test?**
• **What Do We Do When We Test?**
  – Understand *basic techniques for software verification and validation*
  – Analyze *basics of software testing techniques*
The Term Bug

- Bug is used informally

- Defect
- Fault
- Problem
- Error
- Incident
- Anomaly
- Variance
- Failure
- Inconsistency
- Product Anomaly
- Product Incidence
- Feature
Some “Bug detection” Techniques

- **Formal methods**: proving software correct

- **Testing**: executing program in a controlled environment (input) and “validating” output (IEEE definition).

Why Test?
Northeast Blackout of 2003

- 508 generating units and 256 power plants shut down
- Affected 10 million people in Ontario, Canada
- Affected 40 million people in 8 US states
- Financial losses of $6 Billion USD

The alarm system in the energy management system failed due to a software error and operators were not informed of the power overload in the system
Costly Software Failures!

  - Inadequate software testing costs the US alone between $22 and $59 billion annually

- Huge losses due to web application failures
  - Financial services: $6.5 million per hour (just in USA!)
  - Credit card sales applications: $2.4 million per hour (in USA)
Discussion ...

• Have you heard of other software bugs?
  – In the media?
  – From personal experience?

• Does this embarrass you as a future software engineer?
Cost of **Not Testing**

Poor Program Managers might say: “Testing is too expensive.”

- Testing is the **most time consuming and expensive** part of software development.
- **Not** testing is even **more expensive**.
- If we do not have enough testing effort early, the cost of testing **increases**.
Cost of **Not** Testing

1. WE CAN ONLY AFFORD TO FIX THE HIGH-PRIORITY BUGS.
2. IF WE DON'T FIX 100% OF THE BUGS, THE SOFTWARE WILL BE 100% USELESS.
3. SO OUR PLAN IS TO FAIL?
   MORE SLOWLY.
Testing Goals

• The Major Objectives of Software Testing:
  - Detect errors (or bugs) as much as possible in a given timeline.
  - Demonstrate a given software product matching its requirement specifications.
  - Validate the quality of a software testing using the minimum cost and efforts.

• Testing can NOT prove product works 100% -
  - even though we use testing to demonstrate that parts of the software works
Testing Overview

• **Who tests**
  – *Programmers*
  – *Testers/Req. Analyst*
  – *Users*

• **What is tested**
  – *Unit Code* testing
  – *Functional Code* testing
  – Integration/*system* testing
  – *User interface* testing

• **How (test cases designed)**
  – Intuition
  – Specification based (*black box*)
  – Code based (*white-box*)
Software Testing
Exhaustive Testing is Hard

- Number of possible test cases (assuming 32 bit integers)
  - \(2^{32} \times 2^{32} = 2^{64}\)

- Do bigger test sets help?
  - Test set \{(x=3,y=2), (x=2,y=3)\} will detect the error
  - Test set \{(x=3,y=2),(x=4,y=3),(x=5,y=1)\} will not detect the error although it has more test cases

- It is not the number of test cases
- But, if \(T_1 \supseteq T_2\), then \(T_1\) will detect every fault detected by \(T_2\)

```c
int max(int x, int y)
{
    if (x > y)
        return x;
    else
        return x;
}
```

18446744073709551616 possibilities
Exhaustive Testing is Hard

• Assume that the input for the $\text{max}$ procedure was an integer array of size $n$
  – Number of test cases: $2^{32\times n}$

• Assume that the size of the input array is not bounded
  – Number of test cases: $\infty$
Generating Test Cases Randomly

```cpp
bool isEqual(int x, int y) {
    if (x == y) 
        z := false;
    else 
        z := false;
    return z;
}
```

- If we pick test cases randomly it is unlikely that we will pick a case where x and y have the same value
- If x and y can take $2^{32}$ different values, there are $2^{64}$ possible test cases. In $2^{32}$ of them x and y are equal
  - probability of picking a case where x is equal to y is $2^{-32}$
- It is not a good idea to pick the test cases randomly (with uniform distribution) in this case
- So, naive random testing is pretty hopeless too
Types of Testing

• Functional (Black box) vs. Structural (White box) testing
  – **Functional testing**: Generating test cases based on the functionality of the software
  – **Structural testing**: Generating test cases based on the structure of the program

• Black box testing and white box testing are synonyms for functional and structural testing, respectively.
  – **In black box testing** the internal structure of the program is hidden from the testing process
  – **In white box testing** internal structure of the program is taken into account
Criteria Based on Structures

**Structures**: Four ways to model software

1. Graphs

2. Logical Expressions
   
   \[(\neg X \text{ or } \neg Y) \text{ and } A \text{ and } B\]

3. Input Domain Characterization
   
   A: \{0, 1, >1\}
   B: \{600, 700, 800\}
   C: \{swe, cs, isa, infs\}

4. Syntactic Structures

   if \((x > y)\)
   
   
   \[z = x - y;\]

   else

   
   \[z = 2 * x;\]
1. Graph Coverage – Structural

This graph may represent:
- statements & branches
- methods & calls
- components & signals
- states and transitions

Cover every path:
- 12567
- 1257
- 13567
- 1357
- 1343567
- 134357 …
1. Graph Coverage – Structural

• Coverage metrics
  – *Statement coverage*: all statements in the programs should be executed at least once
  – *Branch coverage*: all branches in the program should be executed at least once
  – *Path coverage*: all execution paths in the program should be executed at least once
areTheyPositive(int x, int y) {
    if (x >= 0)
        print("x is positive");
    else
        print("x is negative");
    if (y >= 0)
        print("y is positive");
    else
        print("y is negative");
}

Following test set will give us statement coverage:
\( T_1 = \{(x=12,y=5), (x= -1,y=35), (x=115,y= -13),(x= -91,y= -2)\} \)

There are smaller test cases which will give us statement coverage too:
\( T_2 = \{(x=12,y= -5), (x= -1,y=35)\} \)
Statement vs. Branch Coverage

Consider this program segment, the test set \( T = \{x=1\} \) will give statement coverage, however not branch coverage.

Test set \( \{x=-1\} \) does not execute this edge, hence, it does not give branch coverage.
Path Coverage

areTheyPositive(int x, int y)
{
    if (x >= 0)
        print("x is positive");
    else
        print("x is negative");
    if (y >= 0)
        print("y is positive");
    else
        print("y is negative");
}

Test set:
T₂ = {(x=12,y=–5), (x=–1,y=35)}
gives both branch and statement coverage but it does not give path coverage

Set of all execution paths: {(B0,B1,B3,B4,B6), (B0,B1,B3,B5,B6), (B0,B2,B3,B4,B6), (B0,B2,B3,B5,B6)}
Test set T₂ executes only paths: (B0,B1,B3,B5,B6) and (B0,B2,B3,B4,B6)
areTheyPositive(int x, int y)
{
    if (x >= 0)
        print("x is positive");
    else
        print("x is negative");
    if (y >= 0)
        print("y is positive");
    else
        print("y is negative");
}

Test set: $T_1 = \{(x=12,y=5), (x=-1,y=35), (x=115,y=-13),(x=-91,y=-2)\}$
gives both branch, statement and path coverage
something(int x)
{
    if (x < 0 || y < x)
    {
        y := -y;
        x := -x;
    }
    z := x;
}

T = {(x=1, y=1), (x=1, y=1)} will achieve statement, branch and path coverage, however T will not achieve condition coverage because the boolean term (y < x) never evaluates to true. This test set satisfies part (1) but does not satisfy part (2).

T = {(x=1, y=1), (x=1, y=0)} will not achieve condition coverage either. This test set satisfies part (2) but does not satisfy part (1). It does not achieve branch coverage since both test cases take the true branch, and, hence, it does not achieve condition coverage by definition.

T = {(x=1, y=2), (x=1, y=1)} achieves condition coverage.
Testing Overview

• **Who tests**
  – *Programmers*
  – *Testers/Req. Analyst*
  – *Users*

• **What is tested**
  – *Unit Code* testing
  – *Functional Code* testing
  – Integration/*system* testing
  – *User interface* testing

• **How (test cases designed)**
  – Intuition
  – Specification based (*black box*)
  – Code based (*white-box*)
## Unit testing

| Objectives | To test the function of a program or unit of code such as a program or module  
|            | To test internal logic  
|            | To verify internal design  
|            | To test path & conditions coverage  
|            | To test exception conditions & error handling |
| When       | After modules are coded |
| Input      | Internal Application Design  
|            | Master Test Plan  
<p>|            | Unit Test Plan |
| Output     | Unit Test Report |</p>
<table>
<thead>
<tr>
<th>Who</th>
<th>Developer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Methods</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>White Box testing techniques</td>
</tr>
<tr>
<td></td>
<td>Test Coverage techniques</td>
</tr>
<tr>
<td><strong>Tools</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Debug</td>
</tr>
<tr>
<td></td>
<td>Re-structure</td>
</tr>
<tr>
<td></td>
<td>Code Analyzers</td>
</tr>
<tr>
<td></td>
<td>Path/statement coverage tools</td>
</tr>
<tr>
<td><strong>Education</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Testing Methodology</td>
</tr>
<tr>
<td></td>
<td>Effective use of tools</td>
</tr>
</tbody>
</table>
**Integration testing**

<table>
<thead>
<tr>
<th>Objectives</th>
<th>• To technically verify proper interfacing between modules, and within sub-systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>When</td>
<td>• After modules are unit tested</td>
</tr>
<tr>
<td>Input</td>
<td>• Internal &amp; External Application Design</td>
</tr>
<tr>
<td></td>
<td>• Master Test Plan</td>
</tr>
<tr>
<td></td>
<td>• Integration Test Plan</td>
</tr>
<tr>
<td>Output</td>
<td>• Integration Test report</td>
</tr>
<tr>
<td>Who</td>
<td>Developers</td>
</tr>
<tr>
<td>---------</td>
<td>------------</td>
</tr>
</tbody>
</table>
| Methods | • White and Black Box techniques  
          | • Problem / Configuration Management |
| Tools   | • Debug  
          | • Re-structure  
          | • Code Analyzers |
| Education | • Testing Methodology  
             | • Effective use of tools |
# System Testing

## Objectives
- To verify that the system components perform control functions
- To perform inter-system test
- To demonstrate that the system performs both functionally and operationally as specified
- To perform appropriate types of tests relating to Transaction Flow, Installation, Reliability, Regression etc.

## When
- After Integration Testing

## Input
- Detailed Requirements & External Application Design
- Master Test Plan
- System Test Plan

## Output
- System Test Report
<table>
<thead>
<tr>
<th>Who</th>
<th>Development Team and Users</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods</td>
<td>Problem / Configuration Management</td>
</tr>
<tr>
<td>Tools</td>
<td>Recommended set of tools</td>
</tr>
<tr>
<td>Education</td>
<td>Testing Methodology</td>
</tr>
<tr>
<td></td>
<td>Effective use of tools</td>
</tr>
</tbody>
</table>
Four Fundamental Challenges to Competent Testing

• Complete testing is impossible
• Testers misallocate resources because they fall for the company’s process myths
• Test groups operate under multiple missions, often conflicting, rarely articulated
• Test groups often lack skilled programmers, and a vision of appropriate projects that would keep programming testers challenged
Testing Pitfalls

Our goal is to write bug-free software. I'll pay a ten-dollar bonus for every bug you find and fix.

Yahoo! We're rich! Yes!!! Yes!!! Yes!!!

I hope this drives the right behavior. I'm gonna write me a new minivan this afternoon!
Software Testing