Based on Presentations by

- Nurit Gal-oz, Department of Computer Science Ben-Gurion university
- Mira Balaban Department of Computer Science Ben-Gurion university
The Surprising Relationship between Errors Remaining and Errors Found.

![Graph showing the relationship between the number of errors found and the probability of additional errors remaining.](image)
A self assessment Test

- Write a set of test cases (i.e., specific sets of data to test a simple program)
- Program description
  - The program reads three integer values from an input dialog
  - The three values represent the lengths of sides of a triangle
  - The program displays a message that states whether the triangle is
    - Scalene
    - Isosceles
    - equilateral
Scalene, Isosceles, Equilateral

- A scalene triangle is one where no two sides are equal
- Isosceles has two equal sides
- Equilateral has three sides equal length
- Angles opposite the equal sides in an isosceles also are equal
Evaluate test cases: 1

• Give yourself one point for each “yes” answer
  1. Do you have a test case that represents a valid scalene triangle? (note test cases such as 1,2,3, and 2,5,10 do not warrant a “yes”)
  2. Do you have a test case that represents a valid equilateral triangle?
  3. Do you have a test case that represents a valid isosceles triangle (2,2,4 would not count because it is not a valid triangle)
  4. Do you have at least three test cases that represent valid isosceles triangles such that you have tried all three permutations of two equal sides (such as 3,3,4; 3,4,3; 4,3,3)?
  5. Do you have a test case in which one side has a zero value?
  6. Do you have a test case in which one side has a negative value?
  7. Do you have a test case with three integers greater than zero such that the sum of two of the number is equal to the third? (e.g. 1,2,3)
Evaluate test cases: 2

8. Do you have at least three test cases in category 7 such that you have tried all three permutations where the length of one side is equal to the sum of the lengths of the other two sides (e.g. 1,2,3; 1,3,2; and 3,1,2)?

9. Do you have a test case with three integers greater than zero such that the sum of two of the numbers is less than the third (e.g. 1,2,4 or 12, 15, 30)?

10. Do you have at least three test cases in category 9 such that you have tried all three permutations (e.g. 1,2,4: 1,4,2: and 4,1,2)?

11. Do you have a test case in which all sides are zero (0,0,0)?

12. Do you have at least one test case specifying non-integer value (such as 2.5, 3.5, 5.5)?

13. Do you have at least one test case specifying the wrong number of values (two rather than three integers)

14. For each test case did you specify the expected output from the program in addition to the input values
Evaluation results

• Highly qualified professional programmers score (on average) 7.8 out of possible 14
• Congratulations if you have done > 7.8
• Now consider testing a 100,000 statement air traffic control system, a compiler, or even payroll system, C++ programs
  • Bottom line: complete testing of real-world program is very difficult test
Testing...
Software verification and validation

- **Verification and validation** is intended to show that a system conforms to its specification and meets the requirements of the system customer.
- Involves **checking** and **review processes** and **system testing**.
- **System testing** involves executing the system with test cases that are derived from the specification of the real data to be processed by the system.
Verification vs. Validation

• Verification:
  "Are we building the product right?"
  • The software should conform to its specification.

• Validation:
  "Are we building the right product?"
  • The software should do what the user really requires.
Verification & Validation Goals

• Why Test?
  • Establish confidence that the software is fit for purpose.
  • Prove that the software is correct ???
  • Demonstrate conformance to requirements
  • Find faults
  • Reduce costs
  • Show system meets user needs
  • Assess the software quality
Verification & Validation Goals

- Establish **confidence** that the software is fit for purpose.
- Two principal objectives
  - The **discovery of defects** in a system
  - The assessment of whether or not the system is **useful and useable** in an operational situation.
- This *does not mean*:
  - completely free of defects
  - correct
- Rather, it must be **good enough for its intended use** and the type of use will determine the degree of confidence that is needed.
V & V confidence

- Depends on system’s purpose, user expectations and marketing environment
  - Software function
    - The level of confidence depends on how critical the software is to an organization.
  - User expectations
    - Users may have low expectations of certain kinds of software.
  - Marketing environment
    - Getting a product to market early may be more important than finding defects in the program.
Static and dynamic verification

- **Static verification.** Concerned with Analysis of the static system representation to **discover problems**
  - **Software inspections** (walkthrough, peer rating)
    - May be supplement by tool-based document and code analysis
  - **Static analysis**
  - **Formal verification**

- **Dynamic Validation (and verification):** Concerned with exercising and observing product behavior, using **Software testing**.
  - The system is executed with **test data** and its operational behavior is observed
Static verification
Code Inspections

• Formalized approach to document reviews

• Intended explicitly for **defect detection** (**not correction**).

• Defects may be: logical errors, anomalies in the code that might indicate an erroneous condition (e.g. an uninitialized variable) or non-compliance with standards.

• Inspections **can** check **conformance with a specification** but **not** conformance with the customer’s real requirements.

• **Non-functional characteristics:**
Static verification
Code Inspections

- Formalized approach to document reviews
- Intended explicitly for **defect detection** (*not correction*).
- Defects may be: logical errors, anomalies in the code that might indicate an erroneous condition (e.g. an uninitialized variable) or non-compliance with standards.
- Inspections **can** check **conformance with a specification** but **not** conformance with the customer’s real requirements.
- Non-functional characteristics: Inspections **cannot** check non-functional characteristics such as performance, usability, etc.
- Usually done by a group of programmers
  - The comments are to the program not the programmer!!
Software Inspections
**Static verification**

**Code Inspections**

- **An Error Checklist for Inspections**
  - The checklist is largely *language independent*
  - Sometimes (unfortunately) concentrate more on issues of style than on errors
    - for example, “Are comments accurate and meaningful?” and “Are if-else, code blocks, and do-while groups aligned?"), and the error checks are too nebulous to be useful (such as “Does the code meet the design requirements?”).

- **Beneficial side effects**
  - The programmer usually receives *feedback* concerning programming style, choice of algorithms, and programming techniques.
  - The other participants gain in a similar way by *being exposed* to another programmer’s errors and programming style.
## Static verification

### Code Inspections

#### Table 3.1: Inspection Error Checklist Summary, Part I

<table>
<thead>
<tr>
<th>Data Reference</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Unset variable used?</td>
<td>1. Computations on nonarithmetic variables?</td>
</tr>
<tr>
<td>3. Non integer subscript?</td>
<td>3. Computations on variables of different lengths?</td>
</tr>
<tr>
<td>4. Dangling references?</td>
<td>4. Target size less than size of assigned value?</td>
</tr>
<tr>
<td>5. Correct attributes when aliasing?</td>
<td>5. Intermediate result overflow or underflow?</td>
</tr>
<tr>
<td>6. Record and structure attributes match?</td>
<td>6. Division by zero?</td>
</tr>
<tr>
<td>8. Passed bit-string arguments?</td>
<td>8. Variable’s value outside of meaningful range?</td>
</tr>
<tr>
<td>11. Are inheritance requirements met?</td>
<td></td>
</tr>
</tbody>
</table>

#### Table 3.2: Inspection Error Checklist Summary, Part II

<table>
<thead>
<tr>
<th>Control Flow</th>
<th>Input/Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Multway branches exceeded?</td>
<td>1. File attributes correct?</td>
</tr>
<tr>
<td>2. Will each loop terminate?</td>
<td>2. OPEN statements correct?</td>
</tr>
<tr>
<td>4. Any loop bypasses because of entry conditions?</td>
<td>4. Buffer size matches record size?</td>
</tr>
<tr>
<td>5. Are possible loop fall-throughs correct?</td>
<td>5. Files opened before use?</td>
</tr>
<tr>
<td>6. Off-by-one iteration errors?</td>
<td>6. Files closed after use?</td>
</tr>
<tr>
<td>7. DO-END statements match?</td>
<td>7. End-of-file conditions handled?</td>
</tr>
<tr>
<td>8. Any nonexhaustive decisions?</td>
<td>8. I/O errors handled?</td>
</tr>
<tr>
<td>9. Any textual or grammatical errors in output information?</td>
<td></td>
</tr>
<tr>
<td>Interfaces</td>
<td>Other Checks</td>
</tr>
<tr>
<td>1. Number of input parameters equal to number of arguments?</td>
<td>1. Any unreferenced variables in cross-reference listing?</td>
</tr>
<tr>
<td>2. Parameter and argument attributes match?</td>
<td>2. Attribute list what was expected?</td>
</tr>
<tr>
<td>3. Parameter and argument units system match?</td>
<td>3. Any warning or informational messages?</td>
</tr>
<tr>
<td>4. Number of arguments transmitted to called modules equal to number of parameters?</td>
<td>4. Input checked for validity?</td>
</tr>
<tr>
<td>5. Attributes of arguments transmitted to called modules equal to attributes of parameters?</td>
<td>5. Missing function?</td>
</tr>
<tr>
<td>6. Units system of arguments transmitted to called modules equal to units system of parameters?</td>
<td></td>
</tr>
<tr>
<td>7. Number, attributes, and order of arguments to built-in functions correct?</td>
<td></td>
</tr>
<tr>
<td>8. Any references to parameters not associated with current point of entry?</td>
<td></td>
</tr>
<tr>
<td>9. Input-only arguments altered?</td>
<td></td>
</tr>
<tr>
<td>10. Global variable definitions consistent across modules?</td>
<td></td>
</tr>
<tr>
<td>11. Constants passed as arguments?</td>
<td></td>
</tr>
</tbody>
</table>
Static verification

Code Inspections - example

Data Flow Errors
• Will the program, module, or subroutine eventually terminate?

    for (i=x ; i<=z; i++) {
    ...
} 
while (NOTFOUND) {
    ...
}

• what happens if NOTFOUND never becomes false?

Comparison errors
• Does the way in which the compiler evaluates Boolean expressions affect the program?

    If ((x!=0) & (y/x)>z)

• This statement may be acceptable for compilers that end the test as soon as one side of an and is false, but may cause a division-by-zero error with other compilers.
Static verification
Automated Static Analysis

- Static analyzers are software tools for source text processing.
- A static Analyzer:
  - A collection of algorithms and techniques used to analyze source code in order to automatically find bugs
  - Parses the program text and try to discover potentially erroneous conditions and bring these to the attention of the V & V team.
- Very effective as an aid to inspections – they are a supplement to but not a replacement for inspections.
Static verification
Automated Static Analysis

Method invocation 'fromCurrency.equals("USD")' may produce 'java.lang.NullPointerException'.

```java
double answer = 0;
if (fromCurrency.equals("USD") && toCurrency.equals("CDN")) {
    Assert 'fromCurrency != null'
```

```java
attribute = parseAttribute(isempty, asp, php);
if (attribute == null) {
    ...
    return;
}
value = parseValue(attribute, false, isempty, delim);
if (attribute != null) {
    ...
    Condition 'attribute != null' is always 'true'.
}
else {
    av = new AttrVal( null, null, null, null,
                      0, attribute, value );
    Report.attrError(this, this.token, value,
                     Report.BAD_ATTRIBUTE_VALUE);
}
```
Static verification
Example of Static Analyzer output

Example from the PVS-Studio's Static Code Analyzer for C/C++/

Diagnostic message:
- V523 The 'then' statement is equivalent to the 'else' statement.
- V501 There are identical sub-expressions 'p_transactionId' to the left and to the right of the '!' operator.
- V532 Consider inspecting the statement of '*pointer++' pattern. Probably meant: '(*pointer)++'.

• http://www.viva64.com/en/a/0069/#ID0EY6BI

```c
if (Condition) {
    Block StatementA
} else {
    Block StatementA
}
```

```c
p_transactionId!=p_transactionId
```

```c
static IppTableInitAlloc(Ipp32s *tbl, ...) {
    for (i = 0; i < num_tbl; i++) {
        *tbl++;
    }
}
```
Formal Verification Methods

- Formal methods can be used when a mathematical specification of the system is available.
- Form the ultimate static verification technique.
- Involve detailed mathematical analysis of the specification.
- May develop formal arguments that a program conforms to its mathematical specification.
  - Model checking.
  - Predicate abstraction.
  - Termination analysis.
Static verification
Arguments for/against formal methods

For

- Producing a mathematical specification requires a detailed analysis of the requirements and this is likely to uncover errors.
- Can detect implementation errors before testing when the program is analyzed alongside the specification.

Against

- Require specialized notations that cannot be understood by domain experts.
- Very expensive to develop a specification and even more expensive to show that a program meets that specification.
- It may be possible to reach the same level of confidence in a program more cheaply using other V & V techniques.
Dynamic V & V
Software Testing

- According to Boehm (1975),
- Programmers in large software projects typically spend their time as follows:
  - 45-50% program checkout
  - 33% program design
  - 20% coding
Developers reported spending a little less than half of their time (49% ± 39%) fixing bugs, 36% (± 37%) writing new features, and the rest (15% ± 21%) making code more maintainable.
Software testing

• Yet, in spite of this checkout expense, delivered “verified” and “validated” code is still unreliable.

Software Testing

• The only validation technique for non-functional requirements as the software has to be executed to see how it behaves.
• Should be used in conjunction with static verification to provide full V&V coverage.
Two types of Testing

- **Defect testing**
  - Tests designed to discover **system defects**.
  - A *successful* defect test is one which **reveals** the presence of defects in a system.

- **Validation testing**
  - Intended to show that the software meets its requirements.
  - A *successful* test is one that **shows** that a requirements has been properly implemented.
Psychology of Testing (1)
Psychology of Testing (2)

- A program is its programmer’s baby!
  - Trying to find errors in one’s own program is like trying to find defects in one’s own baby.
  - It is best to have someone other than the programmer doing the testing.
- Tester must be highly skilled, experienced professional.
Good Software Tester

- Technical Skills
- Analytical Skills
- Passion
- Attitude
- Verbal and Written Communication
- Productivity

Good Software Tester
A traditional testing approach

- Show that the system:
  - does what it should
  - doesn't do what it shouldn't

Goal: show working
Success: system works

Fastest achievement: easy test cases

Result: faults left in

Source: AiTi Education Software Testing Session 01 b
http://www.slideshare.net/thecarpenter/session-01-b-38371060
A better Testing Approach

- Show that the system:
  - does what it shouldn't
  - doesn't do what it should

Goal: find faults
Success: system fails

Fastest achievement: difficult test cases

Result: fewer faults left in

Source: AiTi Education Software Testing Session 01 b
http://www.slideshare.net/thecarpenter/session-01-b-38371060
Psychology of Testing (3)

• Testing achievements depend a lot on what are the goals.
• Myers says (79):
  • If your goal is to show absence of errors, you will not discover many.
  • If you are trying to show the program correct, your subconscious will manufacture safe test cases.
  • If your goal is to show presence of errors, you will discover large percentage of them.

Testing is the process of executing a program with the intention of finding errors (G. Myers)
Testing

When?

- Testing along software development process.

What?

- What to test?

How?

- How to conduct the test?
- Discussion on Integration testing
Testing along software development
The Testing Process

- Unit testing
- Module testing
- Sub-system testing
- System testing
- Acceptance testing

Component testing
Integration testing
User testing
Test Planning

- Test planning is concerned with scheduling and resourcing all of the activities in the testing process.
- It involves defining the testing process, taking into account the people and the time available.
- Usually, a test plan will be created, which defines what is to be tested, the predicted testing schedule, and how tests will be recorded.
- For critical systems, the test plan may also include details of the tests to be run on the software.
Black-box and White-box Testing

**Black Box Testing Approach**

**White Box Testing Approach**
White-box Testing

- White-box Testing
  - Testing the degree to which test cases cover the logic
    - Also named: open-box, clear box, glass box, logic-driven
  - Examine the internal structure of the program.
  - Test all possible paths of control flow (theoretically)
  - Capture:
    - errors of omission – neglected specification
    - errors of commission – not defined by the specification
Black-box Testing

- **Black-box testing**
  - Also named: closed box, data-driven, input/output driven, behavioral testing
  - Provide both valid and invalid inputs.
  - The output is matched with expected result (from specifications).
  - The tester has no knowledge of internal structure of the program.
  - It is impossible to find all errors using this approach.
  - Test all possible types of inputs (including invalid ones like characters, float, negative integers, etc.).
# Testing Stages

- **Unit testing**
  - Individual components are tested.

- **Module testing**
  - Related collections of dependent components are tested.

- **Sub-system testing/System Testing**
  - Modules are integrated into sub-systems and tested. The focus here should be on **interface testing**.

- **Acceptance testing**
  - **Black Box**
    - Testing with customer data to check that it is acceptable.
  - **System testing**
    - Testing of the system as a whole. Testing of emergent properties.
Component testing – unit and module (1)

- Testing of individual program components, *in isolation from specifications*;
- Usually the **responsibility** of the component developer (except sometimes for critical systems);
- Tests are derived from the developer’s experience.
- Based on program structure – **structural tests**.
- **white-box** testing
- It is a **defect** testing process.
Component testing – unit and module (2)

- Components may be:
  - Individual functions or methods within an object;
  - Object classes with several attributes and methods;
  - Composite components with defined interfaces used to access their functionality.

- Ideal object class testing:
  - Complete test coverage of a class involves
    - Testing all operations associated with an object;
    - Setting and interrogating all object attributes;
    - Exercising the object in all possible states.
  - Inheritance makes it more difficult to design object class tests as the information to be tested is not localized.
    - You can’t simply test an operation in the class where it is defined and assume that it will work as expected in the subclasses that inherit the operation.
An Example of Object Class Testing

- Need to define test cases for `reportWeather`, `calibrate`, `test`, `startup`, and `shutdown`.
- Using a state model, identify sequences of state transitions to be tested and the event sequences to cause these transitions.
- Example: `waiting`→`calibrating`→`testing`→`transmitting`→`waiting`.
Integration testing

- Testing of groups of components integrated to create a system or sub-system;
  - The test team has access to the system source code.
  - The system is tested as components are integrated.

- Involves building a system from its components and testing it for problems that arise from **component interactions**.

- Top-down integration
  - Develop the skeleton of the system and populate it with components.

- Bottom-up integration
  - Integrate infrastructure components then add functional components.
System/Release testing (1)

- Testing a release of a system that will be distributed to customers.

- **Primary goal**: Increase the supplier’s confidence that the system meets its requirements.

- Release testing is usually *black-box or functional* testing:
  - Based on the system specification only;
  - Testers do not have knowledge of the system implementation.

- Part of release testing may involve testing the emergent properties of a system, such as performance and reliability (System testing).
System Testing (1)

- **Usability Testing**
  - Has each user interface been tailored to the intelligence, educational background, and environmental pressures of the end user?
  - Are the outputs of the program meaningful, nonabusive, and devoid of computer gibberish?
Usability Testing
System Testing (1)

- **Usability Testing**
  - Has each user interface been tailored to the intelligence, educational background, and environmental pressures of the end user?
  - Are the outputs of the program meaningful, nonabusive, and devoid of computer gibberish?

- **Security Testing**
  - Attempting to devise test cases that subvert the program’s security checks.
  - For example, get around an operating system’s memory protection mechanism.
  - You can try to subvert a database management system’s data security mechanisms.

- **Volume Testing**
  - Heavy volumes of data
  - The purpose of volume testing is to show that the program cannot handle the volume of data specified in its objectives
  - For instance, a compiler could be fed an absurdly large source program to compile.
System Testing (2)

- **Stress Testing**
  - Subjects the program to heavy loads, or stresses.
  - The tester attempts to **stress or load an aspect of the system to the point of failure**
  - The tester identifies peak load conditions at which the program will fail to handle required processing loads within required time spans.
  - Stress testing has **different meaning for different industries** where it is used.
  - Stressing the system often causes defects to come to light.
  - Stressing the system test **failure behavior**.
  - Systems should not fail catastrophically.
  - It is applicable, however, to programs that operate under varying loads, or interactive, real-time, and process control programs.
System Testing (3)

- **Performance Testing**
  - Determine if the system meets the **stated performance criteria:**
    - response times and throughput rates under certain workload and configuration conditions.
  - E.g. A Login request shall be responded to in 1 second or less under a typical daily load of 1000 requests per minute.

- **Recovery Testing**
  - Programs such as operating systems, database management systems, have recovery objectives that state how the system is to recover from programming errors, hardware failures, and data errors.
  - **Programming errors can be purposely injected into a system** to determine whether it can recover from them.
    - Hardware failures such as memory parity errors or I/O device errors can be simulated.
    - Data errors such as noise on a communications line or an invalid pointer in a database can be created purposely or simulated to analyze the system’s reaction.
Finding What to Test (1)

- Content of a test case:
  - describe *what is tested*
  - give *input data*
  - give *output data (assumed to be expected)*
  - Further *constraints, if any.*

http://www.wintestgear.com/products/TCMLite/TCMLite.html
Finding What to Test (2)

- Only **exhaustive testing** can show a program is free from defects.

- Exhaustive testing is impossible – leads to combinatorial explosion.

- We need to group cases into **equivalence classes**.
  - Two test cases are in the same equivalence class if they find the **same error**.
  - Try to find a set of test cases with exactly **one element** of each equivalence class.
Finding What to Test (3)

- Each new test case added to suite should expose at least one (and as many as possible) previously undetectable error.
  - Each tested thing must appear in at least, and preferably at most one, test case.
  - A test case may and should test more than one thing.
- It is hoped to get enough test cases to find all errors — high test coverage.
- A certain amount of skill and deviousness is needed to generate test cases.
Test case design

- Involves designing the test cases (inputs and outputs) used to test the system.

- The goal of test case design is to create a set of tests that are effective in validation and defect testing.

- **Design approaches:**
  - **Requirements-based testing (black-box)**
    - Mainly in release testing;
  - **Partition testing**
    - Based on structure of input or output domain;
    - In *component* and *release testing*.
  - **Structural testing (based on code, white-box).**
    - In component testing.
Requirements based testing

- A general principle of requirements engineering: **requirements should be testable.**
- A validation testing technique where you consider each requirement and derive a set of tests for that requirement.
- **Use cases can be a basis for deriving the tests for a system.**
  - They help **identify operations between an actor and the system, or** internal system operations.
  - For each operation, there is a **contract, that specifies input of the** operation, preconditions and postconditions for the operation.
  - The contracts **specify test cases**
A Matrix for Testing Specific Scenarios

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Scenario / Condition</th>
<th>Description</th>
<th>Data Value 1 / Condition 1</th>
<th>Data Value 2 / Condition 2</th>
<th>...</th>
<th>Expected Result</th>
<th>Actual Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Scenario 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Scenario 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Scenario 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Partition based testing (2)

Example: Partition of input in a search routine:

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single value</td>
<td>In sequence</td>
</tr>
<tr>
<td>Single value</td>
<td>Not in sequence</td>
</tr>
<tr>
<td>More than 1 value</td>
<td>First element in sequence</td>
</tr>
<tr>
<td>More than 1 value</td>
<td>Last element in sequence</td>
</tr>
<tr>
<td>More than 1 value</td>
<td>Middle element in sequence</td>
</tr>
<tr>
<td>More than 1 value</td>
<td>Not in sequence</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input sequence (T)</th>
<th>Key (Key)</th>
<th>Output (Found, L)</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>17</td>
<td>true, 1</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td>false, ??</td>
</tr>
<tr>
<td>17, 29, 21, 23</td>
<td>17</td>
<td>true, 1</td>
</tr>
<tr>
<td>41, 18, 9, 31, 30, 16, 45</td>
<td>45</td>
<td>true, 7</td>
</tr>
<tr>
<td>17, 18, 21, 23, 29, 41, 38</td>
<td>23</td>
<td>true, 4</td>
</tr>
<tr>
<td>21, 23, 29, 33, 38</td>
<td>25</td>
<td>false, ??</td>
</tr>
</tbody>
</table>
Partition based testing (3)

Example: *Testing guidelines for sequences:*

- Test software with sequences which have only a single value.
- Use sequences of different sizes in different tests.
- Derive tests so that the first, middle and last elements of the sequence are accessed.
- Test with sequences of zero length.
- Test with sequences of maximal length or nesting – if exists.
Structural testing – code-based (1)

- Sometime called \textit{white-box testing}.
  - \textit{To distinguish from functional (requirements-based, “black-box” testing)}
- Used mainly in \textit{component testing}
- Testing product functionality against its \textit{specification}.
- \textbf{Derivation} of test cases according to \textit{program structure}.
- Knowledge of the program is used to identify additional test cases.
- A large variety of coverage metrics exists
  - Statement Coverage, Condition Coverage, Path Coverage, etc.
Structural testing – code-based (4)

- **Statement Coverage**
- This metric checks executable statements
- Declarative statements that generate executable code are considered executable statements.
- Objective is to exercise all program statements (not all path combinations).
You could execute every statement by writing a single test case that traverses path *ace*

\[ A = 2, \quad B = 0, \quad X = 3 \]
Statement Coverage

```java
public int returnInput(int input, boolean condition1, boolean condition2, boolean condition3) {
    int x = input;
    int y = 0;
    if (condition1)
        x++;
    if (condition2)
        x--;
    if (condition3)
        y=x;
    return y;
}
```

shouldReturnInput(x, true, true, true) - 100% statement covered
Structural testing – code-based (2)

- **Condition Coverage**
  - Test each conditional branch in each direction
  - Test each path at least once if reasonable:
    - For loops, test repetition of:
      - 0
      - 1
      - representative number of times;
      - max number (if exists) of times.

- For loops – termination verification:
  - Try to find invariants that guarantee termination:
    - e.g., show for some loop variable
      - Its value decreases with every round,
      - the value domain for this variable is well founded – no infinite descending chains.
      - \( x' = x - i \), where \( x' \) is value in next loop round, \( i > 0 \),
      - \( x \) range over natural numbers.
Structural testing – code-based (3)

- Test sensitivities to particular values, e.g.,
  - division by 0,
  - overflow,
  - underflow,
  - subscript boundaries,
  - null pointers.
- when you have a large range, use boundary, typical, and sensitive values.
Structural testing – code-based (5)

- **Path Coverage**
  - The objective of path testing is to ensure that the set of test cases is such that each path through the program is executed at least once.
  - The starting point for path testing is a program flow graph that shows nodes representing program decisions and arcs representing the flow of control.
  - Statements with conditions are therefore nodes in the flow graph.

- **Independent Paths**
  - 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 14
  - 1, 2, 3, 4, 5, 14
  - 1, 2, 3, 4, 5, 6, 7, 11, 12, 5, …
  - 1, 2, 3, 4, 6, 7, 2, 11, 13, 5, …
  - Test cases should be derived so that all of these paths are executed
  - A dynamic program analyser may be used to check that paths have been executed
Path coverage

```java
public int returnInput(int input, boolean condition1, boolean condition2, boolean condition3) {
    int x = input;
    int y = 0;
    if (condition1)
        x++;
    if (condition2)
        x--;
    if (condition3)
        y=x;
    return y;
}
```
Error testing

- Test all specified error conditions.
- Test all algorithm implied errors.
- Try to crash the program.
Regression Testing

- When you are doing an enhancement:
  - you should run all old tests
  - even ones that are invalid for original purpose;
  - just provide new expected results.

- An expensive process. Requires automation.
  - Can apply risk analysis to reduce the amounts of tests.

Regression: "when you fix one bug, you introduce several newer bugs."
Regression Testing

- Adding new or changing module impacts the system
  - New data flow paths established
  - New I/O may occur
  - New control logic invoked
- Regression testing is re-execution of subset of tests that have already been conducted
  - Ensures changes have not propagated unintended side effects
Regression Test (Cont.)

• Approaches
  • Manual testing
  • Capture/Playback tools: capture test cases and results for subsequent playback and comparison

• Test suite contains following classes of test cases:
  • Representative sample of tests that exercises all software functions
  • Focus on functions likely affected by change
  • Focus on components that have been changed
Testing guidelines – (Whittaker 02)

- Testing guidelines - hints for the testing team to help them choose tests that will reveal defects in the system
  
  - Choose inputs that force the system to generate all error messages;
  - Design inputs that cause buffers to overflow;
  - Repeat the same input or input series several times;
  - Force invalid outputs to be generated;
  - Force computation results to be too large or too small.
Testing policies – for choosing tests

• Testing policies may define the approach to be used in selecting system tests

• For example:
  • All functions accessed through menus should be tested;
  • Combinations of functions accessed through the same menu should be tested;
  • Where user input is required, all functions must be tested with correct and incorrect input.

  …
Test automation

- Testing is an expensive, difficult activity.
- Testing workbenches provide a range of tools to reduce the time required and total testing costs.
- Systems such as Junit support the automatic execution of tests:
  - comparing output with expected output.
  - reporting when there is no match.
  - Combining tests into test suits.
- There are tools for generating test cases from code,
  - each branch in each direction
  - each path once
- There are tools for running a program against test cases.
- Most testing workbenches are open systems because testing needs are organisation-specific.
- They are sometimes difficult to integrate with closed design and analysis workbenches.
Integration Test

Integration test: A group of dependent components are tested together to ensure their the quality of their integration unit.

The focus is to uncover errors in:
- Design and construction of software architecture
- Integrated functions or operations at sub-system level
- Interfaces and interactions between them
- Resource integration and/or environment integration

Integration testers: either developers and/or test engineers.
Integration Test
Integration testing (1)

- Integration testing is testing how a system consisting of more than one module works together, i.e., **testing the interfaces**.

- Assume a system structure having:
  - a *root component (no client components)*, or components,
  - and *leaf components (no dependencies on other components)*.

- Two Major Kinds of Testing
  - **big-bang!!!!!!**
  - **Incremental** - **several orders possible**:
    - top-down, bottom-up, critical piece first
    - first integrating functional subsystems and then integrating the subsystems in separate phases using any of the basic strategies.
Integration testing (2)

- For a given module $m$, a **driver for $m$ is a module that repeatedly calls $m$ with** test data & maybe even checks results for each test case.

- A **stub for replacing $m$ is a skeletal module with:**
  - interface identical to that of $m$;
  - A body that either
    - does nothing, or
    - prints out name of $m$ maybe with parameter values.
  - A stub may return pre-cooked results for precooked inputs for planned test cases, maybe even testing that actual input is same as planned or some such.
Driver and Stub

- A test driver is a routine that calls a particular component and passes test cases to it. (Also it should report the results of the test cases).

- A test stub is a special-purpose program used to answer the calling sequence and passes back output data that lets the testing process continue.

```java
public class RandInt {
    ...  
    public int generateRandInt() {
        return (int)(Math.random()*100 + 1);  
    }  
    ...
}

public class RandIntStub {
    ...
    public int generateRandInt() {
        return 1;
    }
    ...
}

public static void main(String[] args) {
    RandInt myRand = new RandInt();
    System.out.println("My first rand int is: "+myRand.generateRandInt());
}
```

Driver  Module  Stub
Integration testing (3)

- **Big-Bang Testing**
- Each module is tested in isolation under a driver and with a stub for each module it invokes.
- And then one day … cross your fingers … do a big-bang test of the whole bloody system!
Integration testing (4)

- **Big-Bang Testing**

- **Advantage:**
  - Every module thoroughly tested in isolation.

- **Disadvantages:**
  - Driver and stub must be written for each module (except no driver for root and no stubs for leaves).
  - No error isolation — if error occurs, then in which module or interface is it?
  - **Interface testing** must wait until all modules are programmed
    - Critical interface & major design problems are found very late into project, after all code has been committed!
Incremental integration testing

Test sequence 1

Test sequence 2

Test sequence 3
Integration testing (5)

- **Incremental integration:**
  - There are several specific orders of adding modules one by one to an ever growing system until the whole system is obtained.
  - A module is tested as it is added in the context of whatever of its *callers* and *callees* are already in the System, and *drivers* and *stubs* for *callers* and *callees* that are not yet in the system, respectively.
Integration testing (6) – incremental

- **Orders:**
  - bottom-up
  - top-down
  - Sandwich
  - modified sandwich

- **General Advantages:**
  - Error isolation — assuming that all previously added modules were thoroughly tested, any errors that crop up should be in the module being added or in its interface.
  - Avoid making both driver and stub for each module – if no cyclic dependencies, then at most one is needed for each module.

- **General Disadvantage:**
  - Some modules (which ones depends on the order of adding modules) are not tested thoroughly in isolation, because do not have driver and stub for every module.
Integration testing (7) – incremental

- **Bottom-Up**
  - Test leaf modules in isolation with drivers.
  - Test any non-leaf module *m* in the context of all of its callees (previously tested!) with a driver for *m*.

- **Advantages:**
  - Thorough testing of each leaf module.
  - Abstract types tend to be tested early.

- **Disadvantages:**
  - Must make drivers for every module except root.
  - Major design flaws and serious interface problems involving high modules are not caught until late;
    - could cause a major rewrite of everything that has been tested before.
Bottom Up Integration

Sample 12-module program

Intermediate state in the bottom-up test
Integration testing (8) – incremental

- **Top-down**
  - Only the root is tested in isolation with stubs for its callees.
  - Test any non-root module *m* by having *it replace its stub in its* callers, and with stubs for its callees.

- **Advantages:**
  - Can test during top-down programming.
  - Major flaws and major interface problems are found early.
  - No drivers needed, as modules are tested in context of actual callers.

- **Disadvantages:**
  - Stubs can be difficult to write for planned test cases.
  - Lower modules are not tested thoroughly; tested only in ways used by upper modules and not as fully as possible against specs.
Top Down Integration

Second step in the top-down test

Intermediate state in the top-down test

Sample 12-module program
Bottom-up testing
Top-down testing

Level 1

Level 2
stubs

Level 2

Level 2

Level 2

Level 2

Level 3
stubs

Testing sequence

Level 1
Integration testing (9) – incremental

- **Sandwich**
  - Root and leaf modules are tested in isolation as in T-D and B-U methods.
  - Work from root to middle T-D, and work from leaves to middle BU.
- **Advantages:**
  - Root and leaf modules tested thoroughly.
  - Major design flaws and major interface problems found early.
  - Most abstract data types tested early.
  - No stubs and no drivers needed for middle modules.
- **Disadvantages:**
  - Stubs needed for upper modules and drivers needed for lower modules.
  - Upper-middle modules may not be thoroughly tested.
Integration testing (10) – incremental

- **Modified Sandwich**
  - Do sandwich.
  - Also test upper-middle modules in isolation using drivers and actual callees.

- **Advantages:**
  - Those of sandwich +
  - Removal of its second disadvantage.

- **Disadvantages:**
  - First of sandwich +
  - Extra drivers to write.
Decay of corrected programs

- Myers (79) observes that as the number of detected errors increases in a piece of software, so does the probability of the existence of more, undetected errors. Thus-

  
  testing and debugging, is no substitute for programming it right in the first place.

- Belady and Lehman (76, 85) have shown that assuming a non-zero probability that an attempted correction introduces new bugs, any system which is continually fixed eventually decays beyond usability.