Humans, Errors, and Testing

- Human makes errors
  - Incorrect algorithm
  - Incorrect use of arithmetic/logical operators
  - Incorrect data conversion
  - Incorrect use of pointer or object reference
  - ...

- Testing
  - Determine if the thoughts, actions, and products conform to the requirements
  - A process of identifying semantic errors (i.e., faults)

Errors, Faults (Bugs, Defects), and Failures

Test Automation

- Automated tools are must-have.
- General-purpose test automation
  - GUI testing
    - Eggplant, Marathon, Pounder, ...
  - Performance, load testing
    - eLoadExpert, DBMonster, JMeter, Dieseltest, LoadRunner, Grinder, ...
  - Regression testing
    - Echelon, TestTube, WinRunner, Xtest, ...
- Automating test input generation
  - AETG, DART, ...
- Often, must be customized for special-purpose
- Testing embedded devices
Software Quality

- **Quality Attributes**
  - Static
    - Code quality (structure, maintainability, testability, readability, ...)
    - Documentation quality (use manual, design document, ...)
  - Dynamic
    - Application behavior related attributes
    - Reliability
    - Correctness
    - Completeness
    - Consistency
    - Usability
    - Performance
    - ...

Dynamic Quality Attributes

- **Reliability**
  - Probability of failure-free operation over a given time interval and under given conditions
- **Correctness**
  - Correct operation with reference to some artifacts
- **Completeness**
  - Availability of all features in the requirements
- **Consistency**
  - Adherence to a set of conventions
- **Usability**
- **Performance**
- ...

Requirements, Behavior, and Correctness

- Testers are often faced with incomplete and/or ambiguous requirements
- **Example**

  Example 1.3: Two requirements are given below, each of which leads to a different program.

  Requirement 1: It is required to write a program that inputs two integers and outputs the maximum of these.
  Requirement 2: It is required to write a program that inputs a sequence of integers and outputs the sorted version of this sequence.

- **Input domain**
  - The set of all possible inputs to a program $P$
- **A program is correct** if it behaves as desired on each element of its input domain
  - Called **exhaustive testing** (infeasible in many cases)
- **What about invalid input?**
  - Identifying invalid inputs and testing program against these inputs are important in testing.
  - Need to augment the input domain to account for the invalid inputs
Requirements, Behavior, and Correctness

- Example input domain modified to account for the invalid inputs

The modified input domain consists of pairs of values. The first value in the pair is any ASCII character that can be typed by a user as a request character. The second element of the pair is a sequence of integers, interspersed with invalid characters, terminated by a period. Thus, for example, the following are sample elements from the modified input domain:

```plaintext
< A 7 19 . >
< D 7 9F 19 . >
```

Correctness and Reliability

- Correctness is not the objective of testing
  - Handled via mathematical proofs of programs
- Testing attempts to show the existence of errors and cannot show the correctness of a program.
  - But, it usually helps increasing the reliability of a program.
  - But not guaranteed.
  - Counter example

```plaintext
integer x, y
input x, y
if (x < y) { This condition should be x ≤ y.
    print f(x, y)
} else
    print g(x, y)
```

Correctness vs. Reliability

- Reliability
  - Probability of failure-free operation over a given time interval and under given conditions, on a randomly selected element from its input domain
- A simple example

Example 1.9: Consider a program P which takes a pair of integers as input. The input domain of this program is the set of all pairs of integers. Suppose now that in actual use there are only three pairs that will be input to P. These are as follows:

```plaintext
(0, 0) (-1, 1) (1, -1)
```

- Suppose each pair represents the input domain segment equally likely in practice.
- Suppose P fails on one of the three pairs.

Reliability and Operational Profile

- Operational profiles depending on user types

Example 1.10: Consider a `sort` program which, on any given execution, allows any one of two types of input sequences. One sequence consists of numbers only and the other consists of alphanumeric strings. One operational profile for `sort` is specified as follows:

<table>
<thead>
<tr>
<th>Operational profile 1</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>0.9</td>
</tr>
<tr>
<td>Alphanumeric strings</td>
<td>0.1</td>
</tr>
</tbody>
</table>

Another operational profile for `sort` is specified as follows:

<table>
<thead>
<tr>
<th>Operational profile 2</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>0.09</td>
</tr>
<tr>
<td>Numbers only</td>
<td>0.1</td>
</tr>
<tr>
<td>Alphanumeric strings</td>
<td>0.09</td>
</tr>
</tbody>
</table>
Test and Debug Cycle

13

Constructing Test Data

- A test case is:
  - a pair of test data to be input to the program and the expected output
- A test suite is:
  - A collection of test cases

Questions

- Which test set is the best?
- Is a given test set adequate?

Examples

Example 1.7.1: The following test cases are generated for the sort program using the test plan in Figure 1.8.

Test case 1:
- Test data: "3" 12 20 32
- Expected output:
- Output "20 12 32"

Test case 2:
- Test data: "12" 20 32
- Expected output:
- Output "20 32 12"

Test case 3:
- Test data: "A" 12 20 32
- Expected output:
- Output "A" 12 20 32"

Test case 4:
- Test data: "A" 12 20 32
- Expected output:
- Output "A" 12 20 32"

Test plan

- Test input patterns
- Test adequacy
- Test method

Example

Test plan for sort:
The sort program is to be tested to meet the requirements given in Example 1.5. Specifically, the following needs to be done:

1. Execute the program on at least two input sequences, one with "A" and the other with "D" as request characters.
2. Execute the program on an empty input sequence.
3. Test the program for robustness against erroneous inputs such as "B" typed in as the request character.
4. All failures of the test program should be recorded in a suitable file using the Company Failure Report Form.

Executing the program

- Build test harness to aid program execution
- Test setup
- Execute program with test data
- Check results
Specifying Program Behavior

- Can specify in a natural language (ambiguous)
- A formal specification can be used to specify program behavior
  - Example: state transition sequence using state vector (snapshot of variable values)

Program P1.1
1 integer X, Y, Z;
2 input (X, Y);
3 if (X < Y)
4 (Z=X;)
5 else
6 (Z=X;)
7 endif
8 output (Z);
9 end

Specifying Program Behavior

- For Program P1.1, only final states are of interest
  - Tester may be of interest for other applications
    - e.g., GUI state change

Assessing Correctness of Program Behavior

- Test oracle
  - Observe program behavior
  - Analyze the correctness of the observed behavior
  - Is it really simple?

- Types
  - Human oracle
  - Programs

Construction of Oracles

- Can be automatically generated if models are available (e.g., finite-state machines)
- In general, a daunting task
Test Metrics

- **A standard of measurement**
  - Used for progress monitoring and decision making

- Classification based on the level

General classification
- Schedule-related metrics: estimated and completion time
- Quality-related metrics: quality of artifacts or processes
- Resource-related metrics: test cost
- Size-related metrics: size of various artifacts

Organizational metrics
- Useful in overall project planning and management
- Examples
  - Number of reported defects over interval or artifacts (KLOC)
  - Testing cost per KLOC

Project metrics
- Related to specific project
- Examples
  - Actual-to-planned system test effort (man-months)
  - Ratio of the number of successful tests to the total number of tests in the system test phase

Process metrics
- Assess the goodness of a process
- Examples
  - Number of defects found in each V-model test phase and the cost to fix the defects

Product metrics (Generic)
- Related to specific product
- Many complexity-related metrics

Complexity-related Product Metric
- Cyclomatic complexity metric (1976)
  - Based on the program control flow
  - Given the CFG G of program P containing N nodes, E edges, and p connected procedures, the cyclomatic complexity \( V(G) \) is:
    \[
    V(G) = E - N + 2p
    \]
  - \( V(G) \) of the values 5 or less are recommended

- Halstead complexity estimator (1977)
  - Given program size (S) and effort (E), the estimated number of errors (B) found during the effort is:
    \[
    B = 7.6 E^{0.667} S^{0.333}
    \]
  - Empirical results
### Product Metrics for OO Software

<table>
<thead>
<tr>
<th>Metric</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reliability</td>
<td>Probability of failure of a software product with respect to a given operational profile in a given environment</td>
</tr>
<tr>
<td>Defect density</td>
<td>Number of defects per KLOC</td>
</tr>
<tr>
<td>Defect severity</td>
<td>Distribution of defects by their level of severity</td>
</tr>
<tr>
<td>Test coverage</td>
<td>Fraction of testable items, e.g., basic blocks, covered. Also a metric for test adequacy or goodness of tests</td>
</tr>
<tr>
<td>Cyclomatic complexity</td>
<td>Measures complexity of a program based on its CFG</td>
</tr>
<tr>
<td>Weighted methods per class</td>
<td>$\sum_{i=1}^{n} \text{c}_i$ is the complexity of method $i$ in the class under consideration</td>
</tr>
<tr>
<td>Class coupling</td>
<td>Measures the number of classes to which a given class is coupled</td>
</tr>
<tr>
<td>Response set</td>
<td>Set of all methods that can be invoked, directly and indirectly, when a message is sent to object $O$</td>
</tr>
<tr>
<td>Number of children</td>
<td>Number of immediate descendants of a class in the class hierarchy</td>
</tr>
</tbody>
</table>

### Testability

- **IEEE definition**
  - *Degree to which a system or component facilitates the establishment of test criteria and the performance of test to determine whether those criteria have been met.*

- **Static testability metric example**
  - Software complexity

- **Dynamic testability metrics example**
  - Difficulty of generating test cases that satisfy test coverage requirements

### Software and Hardware Testing

- **Built-In Self Test (BIST)** can be applied to hardware but rarely to software

- **Differences**
  - *Fault model* (e.g., stuck-at model) works for hardware, but not for software (e.g., memory leak)
  - *Test domain* could be an infinite set of tuples for software

### Testing and Verification

- **Program verification**
  - Aims at proving the correctness of programs by showing that it contains no errors.

- **Correctness**
  - Not a perfect process
    - Human can introduce errors in specification and assumptions
    - Cannot be applied to complex, large-scale software

- **Testing**
  - Aims at uncovering errors in a program.

- **Reliability**
  - Not a perfect process
    - *But can show a product is usable and provide value to users*
Defect Management

- Defect classification
  - Severity, types, frequency, location, development phase, ...

- Typically defect handling process
  - Defect opening → assign developers → identifying and implementing fixes → testing the fixes → defect closing

- Defect prediction
  - Static source code analysis or use of defect repository
  - Imprecise in nature

- Tools
  - Bugzilla, FogBugz, ...

Typically defect handling process

- Defect opening
- Assign developers
- Identifying and implementing fixes
- Testing the fixes
- Defect closing

Defect prediction

- Static source code analysis
- Use of defect repository
- Imprecise in nature

Tools

- Bugzilla
- FogBugz

Execution History

- Known as execution trace
  - Organized collection of information about various elements of a program during a given program execution

- Execution slice
  - An executable subsequence of execution history
  - Examples: statement/function execution sequence, object/method invocation sequence, ...

Test-Generation Strategies

- Model-based
  - Model a subset of requirements using a formal notation

- Code-based
  - Generate or modify test cases to meet coverage
  - Also used for regression testing

Static Testing

- Testing without executing software under test (SUT)
  - Carried out by individual(s) who did not write the code

- Use:
  - Requirements document
  - Design document
  - Code
  - User manual
  - Static testing tools

- Walkthroughs
  - Informal process to review documents
  - Conduct under clear objectives and plan agreed by all team members
### Static Testing

- **Inspections**
  - More formal than walkthrough
  - Carried out by a team
  - Form based -- Inspection plan includes
    - Purpose statement
    - Subject artifact (code + documents)
    - Team formation (roles, tasks, ...)
    - Speed
    - Data collection
  - Roles: moderator, reader, recorder, author (helper)
  - Must be friendly and non-confrontational

### Static Testing

- **Static code analysis tools**
  - Provide control-flow and data-flow information valuable to the inspection team
  - Examples
    - IBM Rational Purify
    - Klockwork
    - Parasoft Test / C++test
    - Open source: LAPSE
  - Questions
    - Is the use of x in block 5 okay? (left figure)
    - Is the use of y in block 3 okay? (right figure)

### Model-based Testing and Model Checking

- **Model-based testing**
  - Act of software modeling and test generation from a formal model of application behavior

- **Model checking**
  - Techniques that allow the validation of one or more properties from a given software model
  - Model checker attempts to verify whether the properties are satisfied by the model