Would You ... 

- trust a completely-automated nuclear power plant? 
- trust a completely-automated pilot whose software was written by yourself? A colleague? 
- dare to write an expert system to diagnose cancer? What if you are personally held liable in a case where a patient dies because of a malfunction of the software
Verification and Validation

- Assuring that a software system meets a user's needs
Verification vs. Validation

“Verification is the act of reviewing, inspecting, testing, checking, auditing, or otherwise establishing and documenting whether or not items, processes, services or documents conform to specified requirements” (ANSI A3-1978).

Validation is, according to its ANSI/IEEE definition, “the evaluation of software at the end of the software development process to ensure compliance with the user requirements”. Validation is, therefore, “end-to-end verification.”
The V&V Process

- Is a whole life-cycle process - V&V must be applied at each stage in the software process

- Has two principal objectives
  - The discovery of defects in a system
  - The assessment of whether or not the system is usable in an operational situation
Software inspections Concerned with analysis of the static system representation to discover problems (static verification)
- May be supplement by tool-based document and code analysis

Software testing Concerned with exercising and observing product behaviour (dynamic verification)
- The system is executed with test data and its operational behaviour is observed
Static and Dynamic V&V

Static verification

- Requirements specification
- High-level design
- Formal specification
- Detailed design
- Program

Prototype

Dynamic validation
Program Testing

- Can reveal the presence of errors not their absence
- A successful test is a test which discovers one or more errors
- The only validation technique for non-functional requirements
- Should be used in conjunction with static verification to provide full V&V coverage
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.

- **Statistical testing**
  - Tests designed to reflect the frequency of user inputs.
  - Used for reliability estimation.
Cost of Testing

<table>
<thead>
<tr>
<th>Normal SDLC</th>
<th>Accumulated Test Cost</th>
<th>Accumulated Errors/ 1000 LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>REAQUIREMENTS</td>
<td>0</td>
<td>20</td>
</tr>
</tbody>
</table>

**SDLC TESTING**

<table>
<thead>
<tr>
<th>Accumulated Errors/ 1000 LOC</th>
<th>Accumulated Test Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>REQUIREMENTS 20 ERRORS</td>
<td>10</td>
</tr>
<tr>
<td>DESIGN 20 ERRORS</td>
<td>15</td>
</tr>
<tr>
<td>CODE 20 ERRORS</td>
<td>18</td>
</tr>
<tr>
<td>TEST 80% ERROR REDUCTION</td>
<td>4</td>
</tr>
<tr>
<td>PRODUCTION 0 ERRORS</td>
<td>0</td>
</tr>
</tbody>
</table>

Cost Detect = 1

Verification and Validation of Modern Software Intensive Systems - Schumeyer, G. Gordon
V&V Goals

- Verification and validation should establish confidence that the software is fit for purpose.
- This does not mean completely free of defects.
- 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 organisation
  - 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
Defect testing and debugging are distinct processes.

Verification and validation is concerned with establishing the existence of defects in a program.

Debugging is concerned with locating and repairing these errors.

Debugging involves formulating a hypothesis about program behaviour then testing these hypotheses to find the system error.
The Debugging Process

Locate error -> Design error repair -> Repair error -> Re-test program

Test results -> Specification -> Test cases
Careful planning is required to get the most out of testing and inspection processes
Planning should start early in the development process
The plan should identify the balance between static verification and testing
Test planning is about defining standards for the testing process rather than describing product tests
The V-model of Development

- Requirements specification
- System specification
- System design
- Detailed design
- Acceptance test plan
- System integration test plan
- Sub-system integration test plan
- Service
- Acceptance test
- System integration test
- Sub-system integration test
- Module and unit code and test
The Structure of a Software Test Plan

- The testing process
- Requirements traceability
- Tested items
- Testing schedule
- Test recording procedures
- Hardware and software requirements
- Constraints
Defect Testing

- Testing programs to establish the presence of system defects
### The Testing Process

- **Component testing**
  - Testing of individual program components
  - Usually the responsibility of the component developer (except sometimes for critical systems)
  - Tests are derived from the developer’s experience

- **Integration testing**
  - Testing of groups of components integrated to create a system or sub-system
  - The responsibility of an independent testing team
  - Tests are based on a system specification
Defect Testing

- The goal of defect testing is to discover defects in programs.
- A *successful* defect test is a test which causes a program to behave in an anomalous way.
- Tests show the presence not the absence of defects.
Only exhaustive testing can show a program is free from defects.

```ada
for i in 1..100 loop
  if a(i)=true then
    Ada.Integer_Text_IO.put (1);
  else
    Ada.Integer_Text_IO.put (0);
  end if;
end loop;
```

Has $2^{100}$ different outcomes
Test Data and Test Cases

- **Test data** Inputs which have been devised to test the system

- **Test cases** Inputs to test the system and the predicted outputs from these inputs if the system operates according to its specification
Testing should be:

- **Repeatable**
  - If you find an error, you’ll want to repeat the test to show others
  - If you correct an error, you’ll want to repeat the test to check you did fix it

- **Systematic**
  - Random testing is not enough
  - Select test sets that
    - cover the range of behaviors of the program
    - are representative of real use

- **Documented**
  - Keep track of what tests were performed, and what the results were
Random Testing is not Enough

Structurally …

--Effects: returns True if A=B, False otherwise
if A=B then
   Ada.Text_IO.Put (Item => “True”);
else
   Ada.Text_IO.Put (Item => “False”);
end if;

Test strategy: pick random value for A and B and test “equals” on them

Functionally …

--Requires: LIST is a list of integers
--Effects: returns the maximum element in the list
Maximum (LIST)

Try these test cases:

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
<th>Correct?</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 16 4 32 9</td>
<td>32</td>
<td>Yes</td>
</tr>
<tr>
<td>9 32 4 16 3</td>
<td>32</td>
<td>Yes</td>
</tr>
<tr>
<td>22 32 59 17 88 1</td>
<td>88</td>
<td>Yes</td>
</tr>
<tr>
<td>1 88 17 59 32 22</td>
<td>88</td>
<td>Yes</td>
</tr>
<tr>
<td>6 1 7 4 5 2 2 5 5 8</td>
<td>8</td>
<td>Yes</td>
</tr>
<tr>
<td>8 5 5 2 2 5 4 7 1 6</td>
<td>8</td>
<td>Yes</td>
</tr>
<tr>
<td>553 511 1024 332</td>
<td>1024</td>
<td>Yes</td>
</tr>
<tr>
<td>332 1024 511 553</td>
<td>1024</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Test Techniques

- Classified according to the criterion used to measure the adequacy of a set of test cases:
  - Coverage-based testing
    - Testing requirements are specified in terms of the coverage of the product to be tested
  - Fault-based testing
    - Fault detecting ability of the test set determines the adequacy
  - Error-based testing
    - Focus on error-prone points, based on knowledge of the typical errors that people make
Error
  - Error is a human action that produces an incorrect result
Fault
  - Consequence of an error is software containing a fault. A fault thus is the manifestation of an error.
Failure
  - If encountered, a fault may result in a failure
What we observe during testing are failures.
Or, classify test techniques based on the source of information used to derive test cases:

- **White (glass) box testing**
  - Also called structural or program-based testing
- **Black box testing**
  - Also called functional or specification-based testing
Black-box Testing

- An approach to testing where the program is considered as a ‘black-box’
- The program test cases are based on the system specification
- Test planning can begin early in the software process
Black-box Testing

- Input test data (I)
  - Inputs causing anomalous behavior

- System

- Output test results (O)
  - Outputs which reveal the presence of defects
Equivalence Partitioning

- Input data and output results often fall into different classes where all members of a class are related.
- Each of these classes is an equivalence partition where the program behaves in an equivalent way for each class member.
- Test cases should be chosen from each partition.
Equivalence Partitioning

Invalid inputs

Valid inputs

System

Outputs
Partition system inputs and outputs into ‘equivalence sets’
- If input is a 5-digit integer between 10,000 and 99,999, equivalence partitions are <10,000, 10,000-99, 999 and >99,999
- Choose test cases at the boundary of these sets
  - 00000, 09999, 10000, 99999, 100000
Equivalence Partitions

Number of input values

Input values
procedure Search (Key : Elem;
T     : Elem_Array;
Found : in out Boolean;
L     : in out Elem_Index)

Pre-Condition
-- the array has at least one element
T’First <= T’Last

Post-Condition
-- the element is found and is referenced by L
( Found and T(L) = Key)
or
-- the element is not in the array
( not Found and
not (Exists I, T’First >= I <= T’Last, T (I) = Key ))
Search Routine - Input Partitions

- Inputs which conform to the pre-conditions
- Inputs where a pre-condition does not hold
- Inputs where the key element is a member of the array
- Inputs where the key element is not a member of the array
Testing Guidelines (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
## Search Routine - Input Partitions

<table>
<thead>
<tr>
<th>Array</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><strong>Input sequence</strong> (T)</th>
<th><strong>Key</strong> (Key)</th>
<th><strong>Output</strong> (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>
Structural Testing

- Also called white-box testing
- Derivation of test cases according to program structure. Knowledge of the program is used to identify additional test cases
- Objective is to exercise all program statements

Flowchart:
- Tests
  - Derives
    - Component code
      - Test data
        - Test outputs
White box testing

- Exercise all independent paths within a module at least once
- Exercise all logical decisions on their true and false sides
- Exercise all loops at their boundaries and within their operational bounds
- Exercise all internal data structures to assure their validity
Why White Box Testing

- Why not simply check that
  - Requirements are fulfilled?
  - Interfaces are available and working?

- Reasons for white-box testing:
  - logic errors and incorrect assumptions are inversely proportional to a path’s execution probability
  - we often believe that a path is not likely to be executed; in fact, reality is often counter intuitive
  - typographical errors are random; it’s likely that untested paths will contain some
Exhaustive Testing

There are $5^{20}=10^{14}$ possible paths.

If we execute one test per millisecond, it would take 3,170 years to test this program.
Selective Testing

- Basis path testing
- Condition testing
- Loop testing
- Dataflow testing

\[ \text{loop } \leq 20x \]
Basis Set

- **Basis set** of execution paths = set of paths that will execute all statements and all conditions in a program at least once
- **Cyclomatic complexity** defines the number of independent paths in the basis set
- Basis set is not unique
- Goal: Define test cases for basis set
Flow Graph Notation

Graph Cyclomatic Number \( V(G) = e - n + p \)
Cyclomatic Complexity \( CV(G) = V(G) + 1 \)
Basis Path Testing

- Derive a logical complexity measure
  - Cyclomatic complexity $CV(G)$
    - Number of simple decisions $+p$
      (compound decisions have to be split)
    - Number of enclosed areas $+1$ (uses flow-graph notation)
    - In this case, $CV(G) = 4$

- Use $CV(G)$ to define a basis set of execution paths
  - $CV(G)$ provides an lower bound of tests that must be executed to guarantee coverage of all programs
A number of industry studies have indicated that the higher CV(G), the higher the probability of errors.
Basis Path Testing

CV(G) = 4

There are four paths

Path 1: 1,2,3,6,7,8
Path 2: 1,2,3,5,7,8
Path 3: 1,2,4,7,8
Path 4: 1,2,4,7,2…7,8

We derive test cases to exercise these paths.
Selective Testing

- Basis path testing
- Condition testing
- Loop testing
- Dataflow testing
Condition Testing

- Exercises each logical condition in a program module
- Possible conditions:
  - Simple condition:
    - Boolean variable (T or F)
    - Relational expression (a<\text{b})
  - Compound condition:
    - Composed of several simple conditions
      ((a=b) \text{ and } (c>d))
Condition Testing Methods

- **Branch testing:**
  - Each branch of each condition needs to be exercised at least once

- **Domain testing:**
  - Relational expression $a < b$:
    - 3 tests: $a < b$, $a = b$, $a > b$
  - Boolean expression with $n$ variables
    - $2n$ tests required
Selective Testing

- Basis path testing
- Condition testing
- Loop testing
- Dataflow testing
Loop Testing

- Loops are the cornerstone of every program
- Loops can lead to non-terminating programs
- Loop testing focuses exclusively on the validity of loop constructs

```plaintext
while x < 20 loop
  do something
end loop;
```
Loop Testing

simple loop

nested loops

concatenated loops

unstructured loops
Testing Simple Loops

- **Minimum conditions - simple loops**
  - skip the loop entirely
  - only one pass through the loop
  - two passes through the loop
  - \( m \) passes through the loop \( m < n \)
  - \((n-1), n, \) and \((n+1)\) passes through the loop
  - \( n = \) maximum number of allowable passes
Testing Nested Loops

- Just extending simple loop testing: number of tests grows geometrically
- Reduce the number of tests:
  - start at the innermost loop; set all other loops to minimum values
  - conduct simple loop test; add out-of-range or excluded values
  - work outwards while keeping inner nested loops to typical values
  - continue until all loops have been tested
Testing Concatenated Loops

- Loops are independent of each other:
  - Use simple-loop approach

- Loops depend on each other:
  - Use nested-loop approach
Testing Unstructured Loops

Bad Programming!
Selective Testing

- Basis path testing
- Condition testing
- Loop testing
- Dataflow testing
Dataflow Testing

- Partition the program into pieces of code with a single entry/exit point.
- For each piece find which variables are set/used.
- Various covering criteria:
  - For all set-use pairs
  - For all set to some use
Key Points

- Verification and validation are not the same thing. Verification shows conformance with specification; validation shows that the program meets the customer’s needs.
- Test plans should be drawn up to guide the testing process.
- Static verification techniques involve examination and analysis of the program for error detection.
Key Points

- Test parts of a system which are commonly used rather than those which are rarely executed.
- Equivalence partitions are sets of test cases where the program should behave in an equivalent way.
- Black-box testing is based on the system specification.
- Structural testing identifies test cases which cause all paths through the program to be executed.
- Test coverage measures ensure that all statements have been executed at least once.
Examination schedule
- Wednesday, Dec 18. 1:30 pm – 4:30 pm.

web.mit.edu/www/16.35
- Questions and Answers section on project page

Software development plan and problem set 1.

SRS document due on Monday 9/30
- X-38 Software Requirements Specification
- Software Requirements Specification Template