Verification and Validation

Verification and validation

• Verification and Validation (V&V) is a whole life-cycle process. V&V has two objectives:
  - Discovery of defects,
  - Assessment of whether or not the system is usable in an operational situation.
• Validation: Are we building the right product? i.e., checking that the program as implemented meets the expectations of the software procurer.
• Verification: Are we building the product right? i.e., does the program conform to its specification?
• Verifiability: is the ease of preparing acceptance procedures, especially test data, and procedures for detecting failures and tracing them to errors during the validation and operation phases.
Validation

- Validation techniques include:
  - Requirements reviews – specifications reviewed by:
    • Requirements team
    • Design team
    • Customer
    • Quality assurance team
  - Rapid prototyping:
    • Prototype components built for client demonstration.
    • Components need not be complete or reliable.
  - Formal specification:
    • Mathematical model of the system.

Verification

- Verification techniques include:
  - Code and design inspections – code reviewed by:
    • Design team
    • Programming team
    • Testing team
    • Quality assurance team
  - Testing:
    • Run software with inputs with known outputs and inspect the results.
  - Formal verification:
    • Mathematical proof of correctness to prove that the code satisfies the requirements.
    • “Beware of bugs in the following code. I have proved it correct, but I have not tested it”, Donald Knuth.
Verification and validation

- Static V&V techniques are concerned with analysis of the system representations such as requirements, design and program listing. They are applied at all stages of development through structured reviews.
- Static techniques (program inspections, analysis, formal verification) can only check the correspondence between a program and its specification - they can not demonstrate that software is operationally useful.
- A software product is correct only if is always behaves as specified (i.e., it does what the client wants).
- “For every 3 faults fixed, 1 new fault is introduced”.

Software reliability

- Informally, reliability of a software system is a measure of how well it provides the services expected of it by its users.
- Users do not consider all services to be of equal importance and a system might be viewed as unreliable if it ever failed to provide some critical service.
- Reliability is a dynamic system characteristic which is a function of a number of software systems.
  - A software failure is an execution event where the software behaves in an unexpected way. This is not the same as a software fault.
  - A software fault results in a software failure when the faulty code is executed with a particular set of inputs.
  - Unexpected behaviour can occur when the software conforms to its requirements, but the requirements are incomplete.
  - Incomplete software documentation can also lead to unexpected behaviour.
Cost of reliability

- For software to be very reliable, it must include extra, often redundant, code to perform the necessary checking. This reduces execution speed and increases storage space required. This can automatically increase development costs.

Reliability versus efficiency

- Increasing reliability should normally take precedence over efficiency because:
  - Computers are cheap and fast.
  - Unreliable software is likely to be avoided by users.
  - There are increasing numbers of systems (e.g., nuclear reactors) where human and economic costs of a catastrophic system failure are unacceptable.
  - Inefficient systems can be tuned (most execution time is spent in small program sections).
  - Inefficiency is predictable.
  - Unreliable systems often result in information being lost.
Error rate

- Studies indicate that after completion of coding we have 30-85 errors per 1,000 lines of code.
- Extensive testing leads to identification of repair of many errors. Some are simply just patched.
- On delivery we may have 0.5-3 errors per 1,000 lines of code.
- A serious program of 0.5MB will have 5-30 errors! Is this acceptable? Can you trust it?

Testing

- Dynamic V&V techniques (test) involve exercising an implementation.
- There are two kinds of testing:
  1. Statistical testing. Tests designed to reflect frequency of actual user inputs. Results used to estimate operational reliability of the system.
  2. Defect testing. Tests designed to reveal defects in the system. (A successful defect test is one which reveals the presence of a defect).
- Defect testing and debugging are NOT the same. Testing establishes the presence of defects, debugging is the location and correction of those defects.
Test stages

• Testing should proceed in stages in conjunction with system implementation.
  (1) Unit testing
  (2) Module testing
  (3) Sub-system testing (Integration testing)
  (4) System testing
  (5) Acceptance testing (alpha testing)
  (6) Beta testing
  (7) Regression testing – running old tests after a change.

• The testing process is iterative.

What to test for?

• Correctness of a program is not absolute, but relative.
• If this code correct?
  ```
  from
  
  s := 0
  i := a.lower
  until
  i = a.upper + 1
  loop
  s := s + a.item(i)
  end
  ```
• We will test a class by testing each of its features.
• To test a feature, we need to know what it is supposed to do.
• Yet another reason to document the code fully!
• The primary objective of testing is to make the system fail! A successful test plan is one that finds bugs!
• “Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.” – E.W.Dijkstra.
Testing

- The primary objective of testing is to make the system fail! A successful test plan is one that finds bugs!
- “Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.” – E.W. Dijkstra.
- Exhaustive testing is impractical:
  - Imagine you want to test a 64-bit floating point division function. There are $2^{128}$ combinations! At 1 test every $\mu$s second, it will take $10^{25}$ years.
- The key is to look for equivalence classes. A representative member of some range of possible values.
- Don’t forget to check boundary conditions.
- The challenge is to find inputs that will make the system fail - and then to trace those failure back to the fault in the code that cause it.

Boundary conditions and equivalence classes

- Boundary conditions are often overlooked (especially by students – makes it too easy for us to identify bugs in the code handed in ;-) )
- What are the equivalence for a routine that searches a sorted list for a specific element:
  - Sorted and target present
  - Sorted and target not present
  - Unsorted
- What are the boundary conditions for a routine that searches a sorted list for a specific element:
  - No elements
  - Just one element
  - Target is first or last
- Note the “0, 1, many” principle.
Planning

- Test planning
- System planning is expensive. In large complex systems, testing may consume about half of overall development costs.

Responsibility

- Unit and module testing may be the responsibility of the programmers developing the component. Programmers develop their own test data and incrementally test the code as it is developed.
  - Psychologically, programmers do not usually want to "destroy" their work, therefore, tests may not be selected which will not highlight defects.
  - Should develop a test harness - a small program designed to exercise a unit or subsystem.
  - A monitoring procedure (i.e., retesting by independent tester) helps to ensure that components have been properly tested i.e. need to illustrate that the programmers testing was adequate.
- Later stages of testing involve integrating the work of a number of programmers and must be planned in advance. They should be undertaken by independent testers.
Defect testing

- Testing has two purposes:
  - Show that the program meets its specification
  - Detect defects by exercising the system.
- Component, module and subsystem testing should be orientated toward defect detection. System and acceptance testing should be orientated toward validation.
- In principle, testing for defects should be exhaustive – every possible path through the program should be executed at least once. Cost of this is astronomical.

Testing

- A subset of all possible test cases must be chosen. The test cases must be carefully chosen, making use of knowledge of the application domain, and guidelines such as:
  - Testing a system's capabilities is more important than testing its component. Users want to get a job done and test cases should be chosen to identify aspects of the system that will stop them doing their job.
  - Testing old capabilities is more important than testing new capabilities. Users expect existing functions to keep working and are less concerned by failure of new capabilities which they may not use.
  - Testing typical situations is more important than testing boundary value cases. This does not mean boundary conditions are unimportant, but it is more important that the system works under normal conditions.
Testing

• There are two approaches to testing:
  – Functional or black-box testing where the tests are derived from the program specification.
  – Structural or white-box testing where the tests are derived using knowledge of the program's implementation.
• NOTE: For professional programmers, static code reviews find more faults than either testing approach.

Black-box testing

• The component being tested is treated as a black-box whose behaviour is studied by considering its inputs and related outputs.

$$\text{Input set } I_e \rightarrow \text{Component} \rightarrow \text{Output set } O_e$$
Black-box testing

- Equivalence Partitioning
- Determine which input data have common properties.
  Equivalence partitions are identified from the program
  specification, user documentation and by experience on the
  tests behalf.
- For example, if a program expects input in the range 10,000 to
  99,999, then 3 input equivalence classes are:
  1. numbers < 10000
  2. numbers in the range 10000 <= n <= 99999
  3. numbers > 99999
  - The system should be tested with examples from each
  equivalence class.

Black-box testing

- Output equivalence classes can also be identified. As far as
  possible, input should be selected so that erroneous values
  result if that input was processed as correct input. Recall, we
  are trying to identify defects.
- Sometimes equivalence classes are obvious, sometimes the
  testers experience must be used, e.g., if an input array must be
  ordered, then experience indicates three equivalence classes:
  1. Input array with a single value
  2. Input array with an even number of values
  3. Input array with an odd number of values
- In addition, boundary conditions should be tested, e.g., binary
  search algorithm where:
  1. Key is in the first location
  2. Key is in the last location
  3. Key is elsewhere
White-box testing

- Tester uses knowledge of the implementation to devise test data. Equivalence classes can be identified using this knowledge.
- For example, with a binary search algorithm which divides the search space into three parts, test cases would be where the key lies at the boundary of these partitions.

Top-down testing

- Top-level classes are integrated and tested first.
- Lower-level classes represented by stubs - limited functionality.
- Good - design faults are found early.
- Bad - testing of basic classes is deferred.
Bottom-up testing

- Bottom-level classes are integrated and tested first.
- Upper level classes are replaced by harnesses (programs to exercise the class under test with test data).
- GOOD – basic classes are thoroughly tested.
- BAD – design faults are not discovered until later.

Hybrid

- Bottom-up and top-down testing can be combined.
- Use top-down testing for:
  - Classes with application-specific logic
  - Classes which occur near the top of the dependance hierarchy
- Use bottom-up testing for:
  - Reusable classes with generic functionality
  - Classes near the bottom of the dependency hierarchy
- Such a combination is sometimes called sandwich testing.
Path testing

- Derive a program flow graph which makes all paths through a program explicit. Only selection and repetition statements are important in deriving the flow graph. Sequential statements, such as assignment and procedure calls, are uninteresting.
- An independent program path is one which traverses at least one new edge in the flow graph, i.e., exercising one or more conditions.
- The number of tests needed to test all conditions is equivalent to the number of conditions (in the case of programs without goto’s). Compound expressions with N simple predicates counts as N conditions.
- Knowing the number of tests required does not make it any easier to derive test cases. You should also not be seduced into thinking that such testing is adequate.
- Path testing is based on the control complexity of the program, not the data complexity.
- It is generally true that the number of paths through a program is proportional to its size. Thus, as modules are integrated into systems, it becomes infeasible to use structured testing methods. These techniques are most appropriate at unit and module testing stages.

Static verification

- Program inspections are a form of static verification. They are targeted at defect detection. Inspections can be applied to code, data structure design, detailed design definitions, requirements specifications, user documentation, test plans, etc.
- Defects can be either logical errors, anomalies in the code which might indicate an erroneous condition or non-compliance with project or organizational standards.
- Effective program inspections require that the following conditions be met:
  - Precise specification of the code be available.
  - Inspection team members are familiar with organizational standards.
  - Up-to-date syntactically correct version of the code is available.
  - Checklist of likely errors is available.
  - Management must be aware that static verification will "front load" project costs - there should be a reduction in testing costs.
  - Project management must consider inspections as part of the verification process, not as personnel appraisals.
Static verification

- Inspection team members:
  - author
  - reader
  - tester
  - chairman/moderator.
- There are six stages in the inspection process:
  - planning
  - overview
  - individual preparation
  - program inspection
  - re-work
  - re-inspection
- The inspection team is only concerned with defect detection. It should not suggest how these defects should be corrected, nor recommend changes to other components.

Testing and the software engineer

- Software engineers have test plans.
- These test plans are thought about before the code is written.
- Test plans are written down (and adhered to).
- Software engineering record the results of their testing.
- Software engineers record the changes made to classes during testing.
  - “Maybe the reason that things aren’t going according to plan is that there never was a plan”.

--
Correctness

- Two basic techniques for attempting to produce programs without bugs:
  - Testing: run the program on various sets of data and see if it behaves correctly in these cases.
  - Proving correctness: show mathematically that the program always does what it is supposed to do.
- Both techniques have their particular problems:
  - Testing is only as good as the test cases selected.
  - A proof of correctness may contain errors.
- A detailed formal proof is typically a lot of work. However, even an informal proof is helpful in clarifying your understanding of how a program works and in convincing yourself that it is probably correct.
- Informal proofs are little more than a way of describing your understanding of how the program works - such proofs can easily be produced while writing the program in the first place - Excellent program documentation!

- Before looking at program proving in detail, there is something else that must be pointed out:
  - A program can only be judged correct in relation to a set of specifications for what it is supposed to do.
- All programs do something correctly; the question is: does it do what it is supposed to do?
- A really formal proof amounts to showing that a (mathematical) description of what the program does is the same as a (mathematical) description of what it should do.
- Aspects of a program's correctness include:
  - (1) Partial correctness: whenever the program terminates, it performs correctly.
  - (2) Termination: the program always terminates.
  - (1) + (2) ? Program is totally correct.
Program Correctness Proofs

- Consider the handout "Proof of Program Correctness" and the function "exponentiate" on the first page.

\[
\text{function exponentiate (x: in integer) return integer is}
\]

\[
\text{-- Evaluates } 2^{**x}, \text{ for } x \geq 0 \quad \text{(1)}
\]

\[
i, \text{ sum: integer;}
\]

\[
\text{begin}
\]

\[
\text{sum := 1;}
\]

\[
\text{-- sum = } 2^0 \quad \text{(2)}
\]

\[
\text{for } i \text{ in } 1 .. x \text{ loop}
\]

\[
\text{sum := sum + sum}
\]

\[
\text{-- sum = } 2^i, \quad i > 0 \quad \text{(3)}
\]

\[
\text{end loop;}
\]

\[
\text{-- sum = } 2^{**x}, \quad x \geq 0 \quad \text{(4)}
\]

\[
\text{return sum;}
\]

\[
\text{end exponentiate;}
\]

\{1\} lists the goals of the function
\{2\} asserts the initial value of "sum"
We can prove \{3\} by induction.

The first time \{3\} is reached we have
\[
i = 1
\]

\[
\text{sum} = 1 + 1
\]

\[
= 2
\]

\[
= 2^0 = 2^1
\]

Assume that the nth time \{3\} is reached
\[
\text{sum} = 2^n
\]

then the (n+1)th time sets
\[
\text{sum'} = \text{sum} + \text{sum}
\]

\[
= 2^n + 2^n
\]

\[
= 2^{n+1}
\]

dependence \{3\} always holds.
If \{4\} is ever reached, there are two possibilities:

a) The loop was never executed, in which case \(x=0\), and sum remains unchanged from \{2\}, i.e., sum = 1 = 20.

b) The loop was executed, in which case \{3\} was reached \(x\) times. Hence at \{4\}, sum = 2^x.

See handout for further examples involving induction.

• For large programs, a major obstacle of program correctness proofs is an inability of the human to visualize the entire operation.

• The remedy is modularization. As we can not write a large program without the aid of modularization and top-down design, we can not understand an algorithm and prove correctness unless it is modularized.

• As a module is designed, an informal proof of correctness can be produced to show that the module matches the specification which describes its inputs and outputs.
• A proof of correctness for a module relying on "lower level" modules is only interested in what they do and not how they do it. The lower level modules are assumed to meet the specifications which state what they do.
• The specification of a module consists of two parts:
  - specification of the range of inputs of the module.
  - desired effect of the module.
• In addition to pre and post-conditions, a complex algorithm should contain assertions at key points. The more complex the algorithm, the more assertions that are necessary to bridge the gap between pre- and post-conditions.
• The assertions should be placed so that it is fairly easy to understand the flow of control from one assertion to the next. In practice, this usually means placing at least one assertion in each loop.
• Consider...

```pascal
procedure binary is
  -- binary search algorithm
  N: constant ...;        -- some number ≥1
  x: array (1..N) of float;
  key: float; L, R, K: integer; found: boolean;
begin  key := ...;
  -- (x[I] ≤ x[J] iff 1 ≤ I ≤ J ≤ N) and (X[I] ≤ key ≤ x[N])
  L := 1; R := N; found := false;
  -- 1 ≤ L ≤ R and x(L) ≤ key ≤ x(R)
  while (L ≤ R) and (not found) loop
    K := (L+R) div 2;
    -- 1 ≤ L ≤ K ≤ R ≤ N and (π ≤ x(L) ≤ key ≤ x(R))
    if not found then
      found := (x(K) = key); -- x(K) ≠ key
      if key < x(K) then
        R := K-1; -- π → key ≤ x(R)
      else L := K+1; -- π → x(L) ≤ key
    end if;
  -- π → x(L) ≤ key ≤ x(R)
  end if;
end loop;
-- found= π and (π → x(K)=key)
end binary;
```

[π is "key is present in the array x"]
• {0} is a precondition describing what this module expects of its input.
• {1} is a precondition describing the initial conditions before entering the loop.
• {2} is an assertion true at that point for each iteration of the loop.
• {3} is an assertion true whenever the if condition evaluates to true.
• {4} holds if the then clause is executed.
• {5} holds if the else clause is executed.
• {6} holds after the if statement. It is true irrespective of whether the then or else clause was executed.
• {7} is the postcondition of the module.

Termination

• A proof of partial correctness gives a reasonable degree of confidence in the results produced by an algorithm. Provided a result is output, we can be reasonable confident that it will be correct. However, a proof of partial completeness does not guarantee that a result is produced.
• In order to provide such a guarantee, one must produce a proof of total correctness, i.e., it is also necessary to prove termination.
• In order to prove termination it is necessary to show that conditions on loops are eventually satisfied, that recursive calls eventually stop, etc.
Termination

- A proof of partial correctness gives a reasonable degree of confidence in the results produced by an algorithm. Provided a result is output, we can be relatively confident that it will be correct. However, a proof of partial completeness does not guarantee that a result is produced.
- In order to provide such a guarantee, one must produce a proof of total correctness, i.e., it is also necessary to prove termination.
- In order to prove termination it is necessary to show that conditions on loops are eventually satisfied, that recursive calls eventually stop, etc.
- Consider the following function:

```cpp
function Ackermann(x, y: in integer) return integer
is
  -- x and y must be nonnegative integers
begin -- Ackermann
  if x = 0 then return (y+1);
  elsif y = 0 then return Ackermann((x-1), 1);
  else return Ackermann((x-1), Ackermann(x, (y-1)));
end if;
end Ackermann;
```

- It is not an easy task to follow the algorithm. Try tracing Ackermann(2, 2) or Ackermann(3,1).
- To consider termination, we need only understand enough about the algorithm to see that it terminates for any nonnegative x and y.
- There is no explicit loop ∴ don't need to consider its termination.
- However, there is recursion. Our aim is to find something which is steadily decreasing, because when x=0, no recursive call is made. Note that on two of the recursive calls, x is decreased by 1, so progress is being made. On the remaining recursive call, x is unchanged, but y is decreased by 1. This represents progress also, since when y=0, the recursive call Ackermann((x-1), 1) finally causes x to be decreased by 1.
- All three recursive calls either immediately decrease x, or eventually cause x to be decreased. In any case, the algorithm steadily grinds toward the termination condition, x=0.