Chapter 19
Verification and Validation
Assuring that a software system meets a user’s need

Objectives
are to
• introduce software verification and validation (V&V) and to discuss the distinction between them,
• describe the program inspection process and its role in V&V,
• explain static analysis as a verification technique,
• describe the Clean-Room software development process.

Topics covered
• Verification and validation planning
• Software inspections
• Automated static analysis
• Clean-room software development

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.

The V&V process
• is a whole life-cycle process, i.e., it must be applied at each and every stage in the software process.
• has two principal objectives, viz.,
  • discovery of defects in a system, and
  • assessment of usability in an operational situation.

Static and dynamic verification
• Software inspections: concerned with analysis of the static system representation to discover problems (static verification).
  May be supplemented by tool-based document and code analyses.
• Software testing: concerned with exercising and observing product behavior (dynamic verification).
  The system is executed with test data and its operational behaviour is observed.
Static and dynamic V&V

- Requirements specification
- High-level design
- Formal specification
- Detailed design
- Prototype
- Static verification
- Program testing
- Dynamic validation

Program testing

- Can reveal the presence of errors but CANNOT prove 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
  - It is designed to discover system defects.
  - A successful defect test is one which reveals at least one defect.
  - Covered in Chapter 20
- Statistical (or operational) testing
  - It is designed to reflect the frequency of user inputs. Used for reliability estimation.
  - Covered in Chapter 21

V&V goals

Verification and validation should establish confidence that the software is acceptable for the intended use.

This does NOT mean completely free of defect. 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 the 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

Testing and debugging

- 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, and then testing these hypotheses to find the system error
The debugging process

V & V planning

- 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

The structure of a software test plan

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

Software inspections

- Involve people examining the source representation with the aim of discovering anomalies and defects
- Do not require execution of a system so may be used before implementation
- May be applied to any representation of the system (requirements, design, test data, etc.)
- Very effective technique for discovering errors if done properly

Inspection success

- Many different defects may be discovered in a single inspection. In testing, not all statements will be involved in every test execution. Thus several executions are required.
- When we find an error in an inspection, we also know its nature and location. That is not so in testing.
Inspections and testing

- Inspections and testing are complementary and not opposing verification techniques
- Both should be used during the V & V process
- Inspections can check conformance with a specification but not conformance with the customer’s real requirements
- Inspections cannot check non-functional characteristics such as performance, usability, etc.

Inspection pre-conditions

- A precise specification must be available
- Team members must be familiar with the organization standards
- Syntactically correct code must be available
- An error checklist should be prepared
- Management must accept that inspection will increase costs early in the software process
- Management must not use inspections for staff appraisal

The inspection process

- System overview is presented to the inspection team
- Code and associated documents are distributed to inspection team in advance
- Inspection takes place and discovered errors are noted
- Modifications are made to repair discovered errors
- Re-inspection may or may not be required, depending on the density and severity of defect discovered

Inspection team

An inspection team should consist of at least 4 members, each plays one or more of the following roles:

- **Author** (or owner) who fixes defects discovered
- **Inspector** who finds errors, omissions, and inconsistencies
- **Secretary (Scribe)** who records the results of the inspection meeting
- **Reader** who paraphrases the code
- **Moderator** who chairs the meeting and reports the results

Inspection checklists

- Checklist of common errors should be used to drive the inspection
- Error checklist is programming language dependent
- The 'weaker' the type checking, the larger the checklist
- Examples: Initialisation, Constant naming, loop termination, array bounds, etc.
Inspection check for data faults

- Are all program variables initialized before use?
- Have all constants been named?
- Should the upper bound of arrays be equal to the size of the array or one less?
- If character strings are used, is a delimiter explicitly assigned?
- Is there any possibility of buffer overflow?

Inspection check for control faults

- Is the condition correct for each conditional statement?
- Is each loop certain to terminate?
- Are compound statements correctly bracketed?
- Are all possible cases accounted for in each case statement?
- If a break is required after each case in case statements, has it been included?

Inspection check for I/O faults

- Are all input variables used?
- Are all output variables assigned a value before they are output?
- Can unexpected input cause corruption?

Inspection check for interface faults

- Do all function and method calls have the correct number of parameters?
- Do formal and actual parameter types match?
- Are the parameters in the right order?
- If components access shared memory, do they have the same model of the shared memory structure?

Inspection check for storage management faults

- Have all links been correctly reassigned (if a link structure is modified)?
- Has space been allocated correctly (if dynamic storage is used)
- Is space explicitly deallocated after it is no longer required?

Inspection check for exception handling faults

- Have all possible error conditions been taken into accounts?
Inspection rate

- 500 statements/hour during overview
- 125 source statement/hour during individual preparation
- 90-125 statements/hour can be inspected
- Inspection is therefore an expensive process
- Inspecting 500 lines costs about 40 man/hours effort \( \approx \) $2,000.

Automated static analysis

- Static analysers are software tools for source text analysis.
- They scan 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. A supplement to but not a replacement for inspections.

Checks that can be made through automatic static analysis

<table>
<thead>
<tr>
<th>fault types</th>
<th>check list</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data faults</td>
<td>Variables used before initialization, declared but never used, assigned and assigned again before any use. Possible array bound violations. Undeclared variables.</td>
</tr>
<tr>
<td>Control faults</td>
<td>Unreachable code, unconditional branch into loops</td>
</tr>
<tr>
<td>I/O faults</td>
<td>Variables output twice without intervening assignment</td>
</tr>
<tr>
<td>Interface faults</td>
<td>Parameter type or number mismatches, unused results of functions, uncalled functions/procedures</td>
</tr>
<tr>
<td>Storage faults</td>
<td>Unassigned pointers, pointer arithmetic</td>
</tr>
</tbody>
</table>

Stages of static analysis

- **Control flow analysis.** Checks for loops with multiple exit or entry points, finds unreachable code, etc.
- **Data use analysis.** Detects uninitialized variables, variables written twice without an intervening assignment, variables which are declared but never used, etc.
- **Interface analysis.** Checks the consistency of routine and procedure declarations and their use

**Stages of static analysis**

- **Information flow analysis.** Identifies the dependencies of output variables. Does not detect anomalies itself but highlights information for code inspection or review
- **Path analysis.** Identifies paths through the program and sets out the statements executed in that path. Again, potentially useful in the review process
- Both these stages generate vast amounts of information. Must be used with care.
Use of static analysis

- Particularly valuable when a language such as C is used, which has weak typing and hence many errors are undetected by the compiler.
- Less cost-effective for languages like Java that have strong type checking, and can therefore detect many errors during compilation.

Clean-room software development

- The name is derived from the 'clean-room' process in semiconductor fabrication. The philosophy is defect avoidance rather than defect removal.
- Software development process based on:
  - Incremental development.
  - Static verification (i.e., without test-execution).
  - Formal specification.
  - Statistical testing.

Incremental development

- Instead of doing software design, implementation, and testing in sequence, the software is produced by building a number of executable increments.

Formal methods for spec. and design

- It uses "structured specifications" to divide the product functionality into deeply nested set that can be developed incrementally.
- It uses both structured specifications as well as state machine models.

Development without program execution

- Functionally based programming developed by H. Mills is used to build the software right the first time.
- Developers are not allowed to test and debug the program.
- Developers use the techniques of code reading by stepwise abstraction, code inspection, group walkthrough, and formal verification to assert the correctness of their implementation.

Statistical testing

- Testing is done by a separate team.
- The test team develop an operational profile of the program, and perform a random testing.
- The independent testers assess and record the reliability of the product.
- The independent testers also use a limited number of test cases to ensure correct system operation for situations in which a software failure would be catastrophic.
**Incremental development**

- Establish requirements
- Formal specification
- Develop s/w increment
- Deliver software
- Requirements change request
- Frozen specification

**Clean-room process teams**

- **Specification team.** Responsible for developing and maintaining the system specification.
- **Development team.** Responsible for developing and verifying the software. The software is NOT executed or even compiled during this process.
- **Certification team.** Responsible for developing a set of statistical tests to exercise the software after development. Reliability growth models used to determine when reliability is acceptable.

**Clean-room process evaluation**

- Results in IBM have been impressive with few discovered faults in delivered systems.
- Independent assessment shows that the process is no more expensive than other approaches.
- Fewer errors than in a 'traditional' development process.
- Not clear how this approach can be transferred to an environment with less skilled or less highly motivated engineers.

**Key points**

- Verification shows conformance with the specification whereas validation shows that the program meets the customer’s needs.
- Test plans should be drawn up to guide the testing process.
- Program inspections are effective in discovering errors.
- Static verification involves examination and analysis of the source code for error detection, and can be used to discover anomalies in the source code.