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 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

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 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
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

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

Inspection success

- Many different defects may be discovered in a single inspection
  - In testing, one defect may mask another so several executions are required
- The reuse domain and programming knowledge
  - reviewers are likely to have seen the types of error that commonly arise

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 characteristics such as performance, usability, etc.
Program 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

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

1. Planning
2. Overview
3. Individual preparation
4. Inspection meeting
5. Rework
6. Follow-up

Inspection procedure

- System overview presented to 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
**Inspection teams**

- Made up of at least 4 members
- Author of the code being inspected
- Inspector who finds errors, omissions and inconsistencies
- Reader who reads the code to the team
- Moderator who chairs the meeting and notes discovered errors

**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: Initialization, loop termination, array bounds, etc.

**Inspection checks**

<table>
<thead>
<tr>
<th>Fault class</th>
<th>Inspection check</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data faults</td>
<td>Are all program variables initialized before their values are used?</td>
</tr>
<tr>
<td></td>
<td>Have all constants been named?</td>
</tr>
<tr>
<td></td>
<td>Should the lower bound of arrays be 0, 1, or something else?</td>
</tr>
<tr>
<td></td>
<td>Should the upper bound of arrays be equal to the size of the array or Size - 1?</td>
</tr>
<tr>
<td></td>
<td>If character strings are used, is a delimiter explicitly assigned?</td>
</tr>
<tr>
<td>Control faults</td>
<td>For each conditional statement, is the condition correct?</td>
</tr>
<tr>
<td></td>
<td>Is each loop certain to terminate?</td>
</tr>
<tr>
<td></td>
<td>Are compound statements correctly bracketed?</td>
</tr>
<tr>
<td></td>
<td>In case statements, are all possible cases accounted for?</td>
</tr>
<tr>
<td>Input/output faults</td>
<td>Are all input variables used?</td>
</tr>
<tr>
<td></td>
<td>Are all output variables assigned a value before they are output?</td>
</tr>
</tbody>
</table>

**Inspection checks**

| Interface faults       | Do all function and procedure calls have the correct number of parameters?       |
|                       | Do formal and actual parameter types match?                                     |
|                       | Are the parameters in the right code?                                           |
|                       | If components access shared memory, do they have the same model of the shared memory structure? |

| Storage management faults | If a linked structure is modified, have all links been correctly reassigned? |
|                          | If dynamic storage is used, has space been allocated correctly?                |
|                          | Is space explicitly de-allocated after it is no longer required?              |

| Exception management faults | Have all possible error conditions been taken into account?                |
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 = $$

Automated static analysis

- Static analysers are software tools for source text processing
- They parse 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

Static analysis checks

<table>
<thead>
<tr>
<th>Fault class</th>
<th>Static analysis check</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data faults</td>
<td>Variables used before initialisation</td>
</tr>
<tr>
<td></td>
<td>Variables declared but never used</td>
</tr>
<tr>
<td></td>
<td>Variables assigned twice but never used between assignments</td>
</tr>
<tr>
<td></td>
<td>Possible array bound violations</td>
</tr>
<tr>
<td></td>
<td>Undeclared variables</td>
</tr>
<tr>
<td>Control faults</td>
<td>Unreachable code</td>
</tr>
<tr>
<td></td>
<td>Unconditional branches into loops</td>
</tr>
<tr>
<td>Input/output faults</td>
<td>Variables output twice with no intervening assignment</td>
</tr>
<tr>
<td>Interface faults</td>
<td>Parameter type mismatches</td>
</tr>
<tr>
<td></td>
<td>Parameter number mismatches</td>
</tr>
<tr>
<td></td>
<td>Non-use of the results of functions</td>
</tr>
<tr>
<td></td>
<td>Uncalled functions and procedures</td>
</tr>
<tr>
<td>Storage management</td>
<td>Unassigned pointers</td>
</tr>
<tr>
<td></td>
<td>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.