
Chap. 22

Verification and Validation

Software Engineering, Addison-Wesley

Topics covered

- Verification and validation planning
- Software inspections
- Automated static analysis

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.

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.

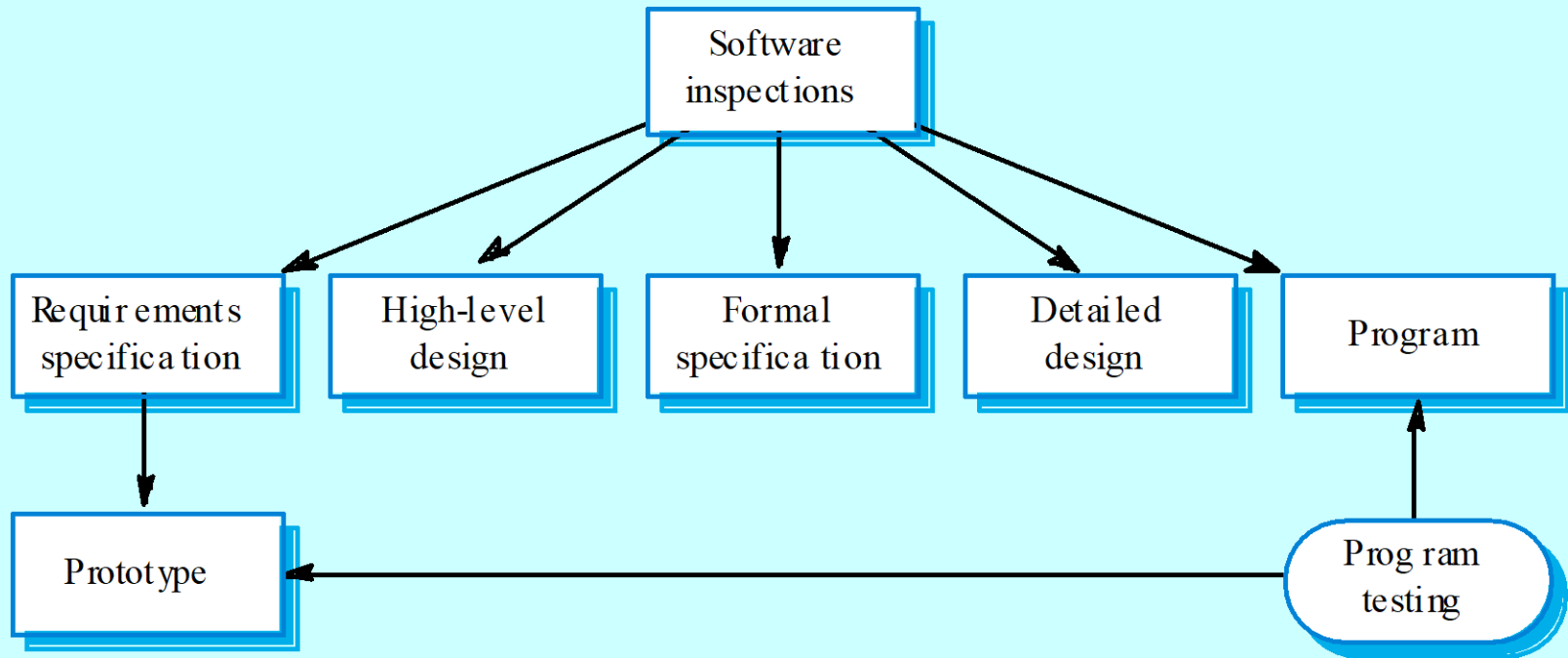
V & V confidence

- 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.

Static and dynamic verification

- **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



Program testing

- Can reveal the presence of errors NOT their absence.
- The only validation technique for non-functional requirements as the software has to be executed to see how it behaves.
- Should be used in conjunction with static verification to provide full V&V coverage.

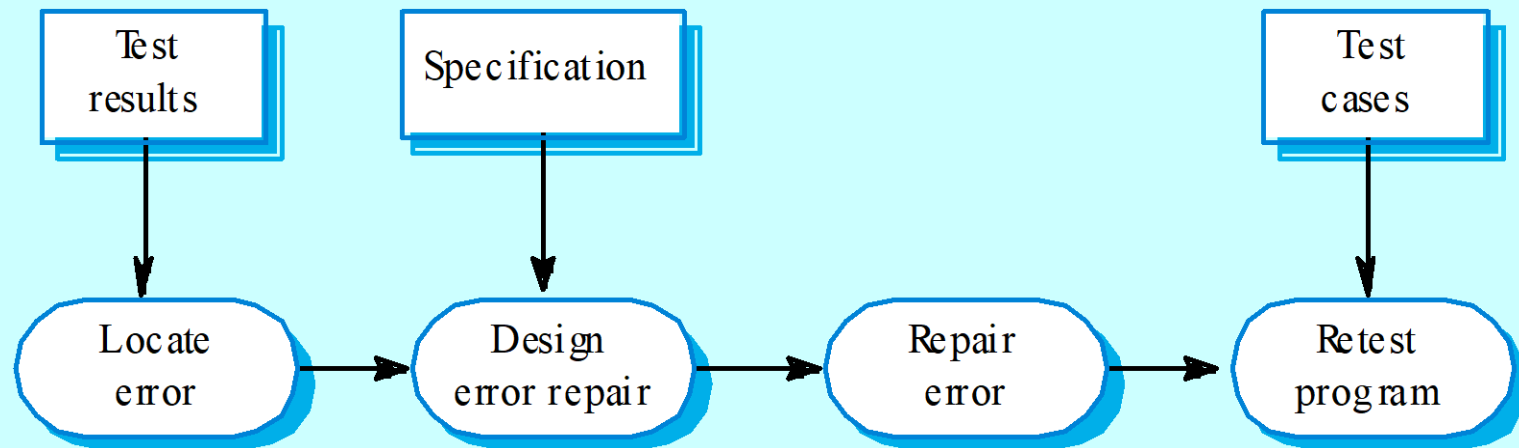
Types of testing

- **Validation testing**
 - Intended to show that the software meets its requirements.
 - A successful test is one that shows that a requirements has been properly implemented.
- **Defect testing**
 - Tests designed to discover system defects.
 - A successful defect test is one which reveals the presence of defects in a system.

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.
- After a defect in the program has been corrected, re-inspection and regression testing of the program are required.
 - Regression testing is used to check that the changes made to a program have not introduced new faults.

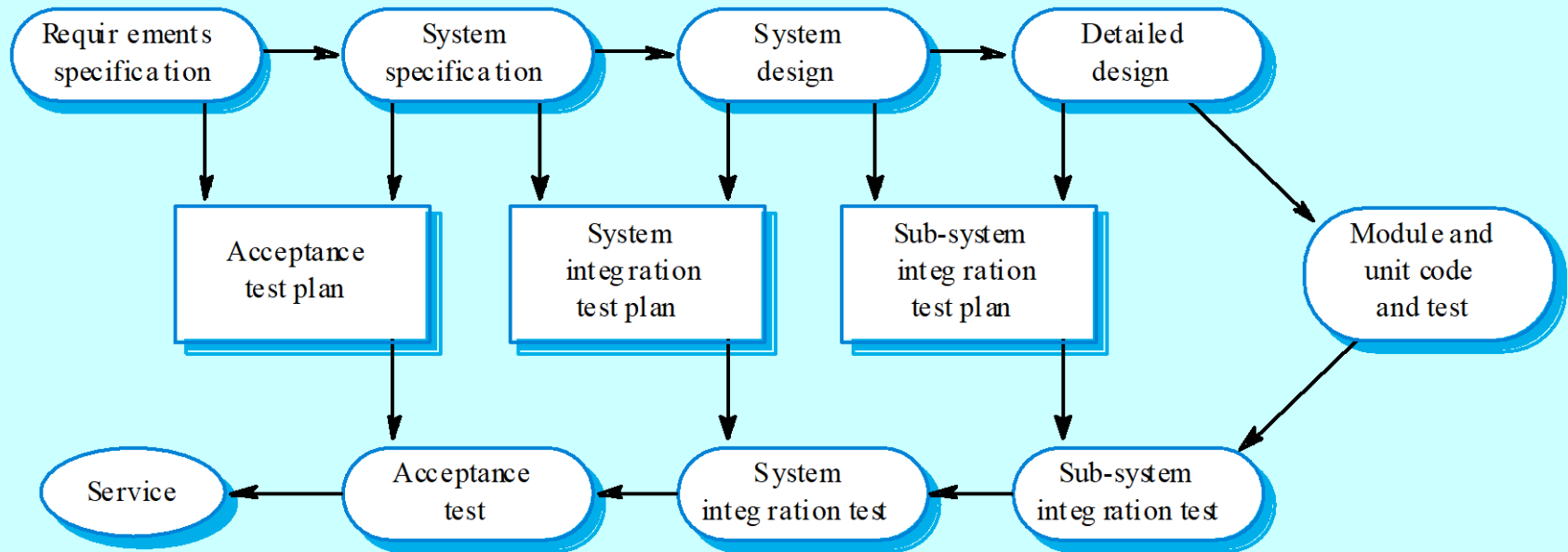
The debugging process



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.
 - The more critical a system, the more effort should be devoted to static verification techniques.

The V-model of development



Software inspections

- These involve people examining the source representation with the aim of discovering anomalies and defects.
- Inspections do not require execution of a system so they may be used before implementation.
- They may be applied to any representation of the system (requirements, design, configuration data, test data, etc.).
- They have been shown to be an effective technique for discovering program errors.

Inspection success

- Many different defects may be discovered in a single inspection. In testing, one defect may hide another so several executions are required.
- With 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 non-functional characteristics such as performance, usability, etc.

Program inspections

- A formal 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 uninitialised variable) or non-compliance with standards.

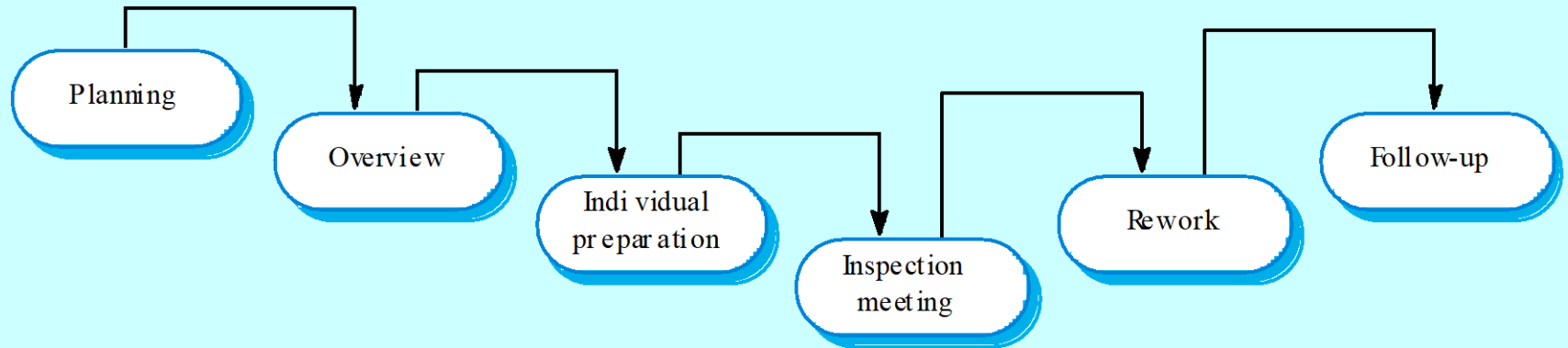
Inspection pre-conditions

- A precise specification must be available.
- Team members must be familiar with the organisation standards.
- Syntactically correct code or other system representations must be available.
- An error checklist should be prepared.
- Management must accept that inspection will increase costs early in the software process.
- Management should not use inspections for staff appraisal, i.e. finding out who makes mistakes.

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.

The inspection process



Inspection checklists

- Checklist of common errors should be used to drive the inspection.
- Error checklists are programming language dependent and reflect the characteristic errors that are likely to arise in the language.
- In general, the 'weaker' the type checking, the larger the checklist.
- Examples: Initialisation, Constant naming, loop termination, array bounds, etc.

Inspection checks 1

Data faults	<p>Are all program variables initialised before their values are used?</p> <p>Have all constants been named?</p> <p>Should the upper bound of arrays be equal to the size of the array or Size -1?</p> <p>If character strings are used, is a delimiter explicitly assigned?</p> <p>Is there any possibility of buffer overflow?</p>
Control faults	<p>For each conditional statement, is the condition correct?</p> <p>Is each loop certain to terminate?</p> <p>Are compound statements correctly bracketed?</p> <p>In case statements, are all possible cases accounted for?</p> <p>If a break is required after each case in case statements, has it been included?</p>
Input/output faults	<p>Are all input variables used?</p> <p>Are all output variables assigned a value before they are output?</p> <p>Can unexpected inputs cause corruption?</p>

Inspection checks 2

Interface faults	<p>Do all function and method calls have the correct number of parameters?</p> <p>Do formal and actual parameter types match?</p> <p>Are the parameters in the right order?</p> <p>If components access shared memory, do they have the same model of the shared memory structure?</p>
Storage management faults	<p>If a linked structure is modified, have all links been correctly reassigned?</p> <p>If dynamic storage is used, has space been allocated correctly?</p> <p>Is space explicitly de-allocated after it is no longer required?</p>
Exception management faults	<p>Have all possible error conditions been taken into account?</p>

Inspection rate

- During overview, 500 statements/hour
- During individual preparation, 125 source statement/hour
- During inspection meeting, 90-125 statements/hour
- Inspection is therefore an expensive process.

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.
- They are very effective as an aid to inspections - they are a supplement to but not a replacement for inspections.

Static analysis checks

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

Stages of static analysis

- **Control flow analysis.** Checks for loops with multiple exit or entry points, finds unreachable code, etc.
- **Data use analysis.** Detects uninitialised 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. They must be used with care.

LINT static analysis

```
138% more lint_ex.c
#include <stdio.h>
printarray (Anarray)
int Anarray;
{ printf("%d",Anarray); }
```

```
main ()
{
int Anarray[5]; int i; char c;
printarray (Anarray, i, c);
printarray (Anarray) ;
}
```

```
139% cc lint_ex.c
140% lint lint_ex.c
```

```
lint_ex.c(10): warning: c may be used before set
lint_ex.c(10): warning: i may be used before set
printarray: variable # of args. lint_ex.c(4) :: lint_ex.c(10)
printarray, arg. 1 used inconsistently lint_ex.c(4) :: lint_ex.c(10)
printarray, arg. 1 used inconsistently lint_ex.c(4) :: lint_ex.c(11)
printf returns value which is always ignored
```

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.

Verification and formal methods

- Formal methods can be used when a mathematical specification of the system is produced.
- They are the ultimate static verification technique.
- The usage is mostly confined to safety and security-critical software development processes.

Arguments for formal methods

- Producing a mathematical specification requires a detailed analysis of the requirements.
- They can detect implementation errors before testing when the program is analysed alongside the specification.

Arguments against formal methods

- Require specialised notations that cannot be understood by domain experts.
- Very expensive to develop a specification and even more expensive to show that a program meets that specification.
- It may be possible to reach the same level of confidence in a program more cheaply using other V & V techniques.