Verification and Validation

Objectives

- To introduce software verification and validation and to discuss the distinction between them
- To describe the program inspection process and its role in V & V
- To explain static analysis as a verification technique
- To describe the Cleanroom software development process

The testing process

- Component (or unit) 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.
- System 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.

Other forms of testing: Performance testing

- Part of release testing may involve testing the emergent properties of a system, such as performance and reliability.
- Performance tests usually involve planning a series of tests where the load is steadily increased until the system performance becomes unacceptable.

Stress testing

- Exercises the system beyond its maximum design load. Stressing the system often causes defects to come to light.
- Stressing the system test failure behaviour. Systems should not fail catastrophically. Stress testing checks for unacceptable loss of service or data.
- Stress testing is particularly relevant to distributed systems that can exhibit severe degradation as a network becomes overloaded.

Interface testing

- Objectives are to detect faults due to interface errors or invalid assumptions about interfaces.
- Particularly important for object-oriented development as objects are defined by their interfaces.

Interface errors

- Interface misuse
  - A calling component calls another component and makes an error in its use of its interface e.g. parameters in the wrong order.
- Interface misunderstanding
  - A calling component embeds assumptions about the behaviour of the called component which are incorrect.
- Timing errors
  - The called and the calling component operate at different speeds and out-of-date information is accessed.

Testing process goals

- Validation testing
  - To demonstrate to the developer and the system customer that the software meets its requirements;
  - A successful test shows that the system operates as intended.
- Defect testing
  - To discover faults or defects in the software where its behavior is incorrect or not in conformance with its specification;
  - A successful test is a test that makes the system perform incorrectly and so exposes a defect in the system.
  - Tests show the presence not the absence of defects

Verification vs validation

- Verification:
  "Are we building the product right".
  - The software should conform to its specification.
  - Usually a priori process
- Validation:
  "Are we building the right product".
  - The software should do what the user really requires.
  - Usually testing after being built

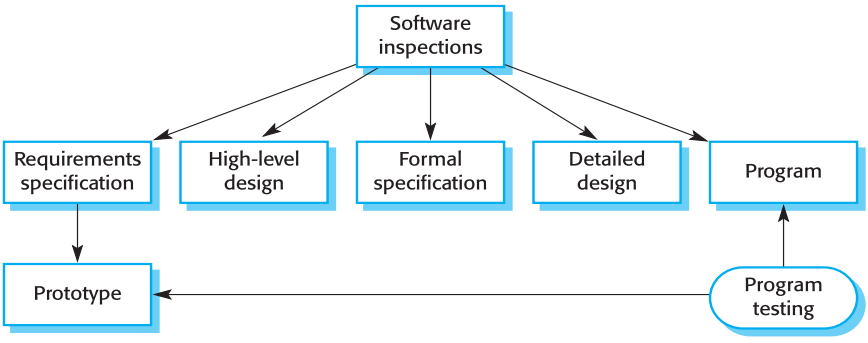
V& V goals

- Verification and validation should establish confidence that the software is fit for purpose.
  - Generally this does not mean to be 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.
    - Unfortunately, users accept this

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 behavior
  - The system is executed with test data and its operational behavior is observed

Static and dynamic V&V



Program testing

- Can reveal the presence of errors NOT their absence - Dijkstra.
- 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.

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 behavior then testing these hypotheses to find the system error.

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 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

The systematic study of a product artifact (e.g., specification document, design document, source program) for the purpose of discovering flaws in it.

- These involve people examining the source representation with the aim of discovering anomalies and defects.
- Inspections do not require execution of a system so 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.

Inspections

Many studies have shown that:

- Finding errors early in a project costs less to fix than finding errors later in a project.
- The factors of 1 time unit for a specification error, 10 time units for a design error, 100 time units for a coding error, to 1000 time units for an integration testing error.
- These numbers have not been formally validated, but the basic principle that later errors are harder to fix seems to be sound.

All of the following techniques are based on the principle that thinking about an artifact results in better results than simply random testing of the artifact.

Inspection pre-conditions

- A precise specification must be available.
- Team members must be familiar with the organization 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 e.g., finding out who makes mistakes.

Reading technologies

All of the following techniques are an improvement over simply testing a program; however, some are more effective than others. Several related concepts:

Walkthroughs: The developer of the artifact describes its structure at a meeting. Attendees look for flaws in the structure. Weakness – reviewers do not understand deep structure so error finding is weak.

Code reviews: An individual who is not the developer of the artifact reads the text of the artifact looking for errors and defects in structure. Quite effective since reader does not have the same preconceived notion of what the artifact does.

Review: A meeting to discuss an artifact – less formal than an inspection. A traditional checkpoint in software development

(Fagan) Inspections

Developed by Michael Fagan at IBM in 1972.

Two approaches toward inspections:

- Part of development process – used to identify problems
- Part of quality assurance process – used to find unresolved issues in a finished product

"Fagan inspections" are the former type. Goal is to find defects – A defect is an instance in which a requirement is not satisfied.

Fagan Inspections

- Development process consists of series of stages (e.g., system design, design, coding, unit testing, ...)
- Develop exit criteria for any artifact passing from one stage to the next
- Validate that every artifact correctly passes the exit criteria before starting on the next phase via an inspection meeting
- Everyone at the meeting must observe that all the exit criteria have been met

Inspection process

Planning – Author gives moderator an artifact to be inspected. Materials, attendees, and schedule for inspection meeting must be set. High level documentation given to attendees in addition to artifact.

Overview – Moderator assigns inspection roles to participants.

Preparation – Artifact given to participants before meeting to allow for study. Participants spend significant time studying artifact.

Inspection – A defect logging meeting for finding defects. (Don't fix them at meeting; Don't assign blame; Don't have managers present; …)

Rework – The author reworks all defects.

Follow-up – Verification by inspection moderator that all fixes are effective.

Observations about inspections

Costly –Participants need many hours to prepare

Intensive – Limit inspections to no more than 2 hours

No personnel evaluation – limits honesty in finding defects

Cannot inspect too much – perhaps 250 non-commented source lines/hour. More than that causes discovery rate to drop and the need for further inspections to increase.

- Up to 80% of errors found during testing could have been found during an inspection.
- Lucent study (Porter—Votta) Inspection meetings often throw away false positives but find few new errors, so eliminate meeting

Inspection participants

Author – one who developed artifact – does not lead or defend artifact, but controls repair

Reader – one who tries to develop a design of the artifact (Several at meeting.)

Tester – one who tries to test the artifact

Moderator – Conducts inspection; Comes from a neutral organization

Scribe – records actions at meeting

If anyone doesn't prepare adequately, the meeting is often cancelled.

Inspections versus Reviews

CONCEPT	REVIEW	INSPECTION
Defects	Opinions of attendees	Violation of specification
Author	Defends position	Observer of proceedings
Error fixing	Discussions at meeting	Later process
Results	List of errors to author	Formal defect list for QA and author
Measurement	None required	Data collected that can be used to develop baseline

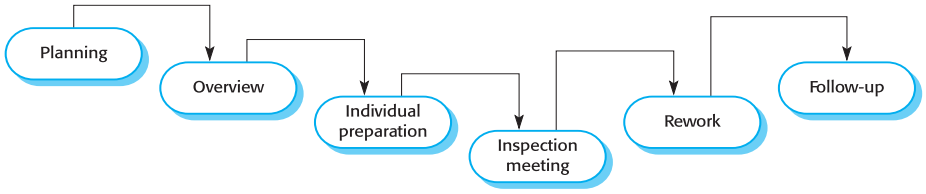
Inspection success

- 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 so 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.

The inspection process



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 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: Initialization, Constant naming, loop termination, array bounds, etc.

Inspection checks 1

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

Inspection checks 2

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?
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.
- Limited to a 2 hour block, maximum

- Inspecting 500 lines costs about 40 hours of effort.

Perspective-based reading

- Developed by Basili.
- Increased error detection if each participant takes on a different role in looking for defects, e.g.,
  - Tester
  - User
  - Designer
- In addition, readers are permitted to address any other defects that they may discover.

PBR: Tester role

Develop tests as if you are in the role of the individual who will test the given artifact:

- Do I have all information necessary to identify the item being tested and identify my test criteria? Can I make up reasonable test cases for each item based upon the criteria?
- Can I be sure that the tests yield the correct answer?
- Do the requirements make sense from what I know about the application or what is specified by the general description?
- Are there other interpretations of this requirement that the implementation might make based upon how the requirement is specified?
- Is there another requirement that would generate a similar test, but yield a different result?

PBR: User role -1

Develop tests as if you are in the role of the individual who will use the given artifact.

- Use scenarios – scripts of sequences of actions the interact with the given artifact being tested.
- Is there any information that prevents you from writing this scenario?
- Are all the necessary functions necessary to write this scenario specified in the requirements, e.g., all the capabilities listed in the general description?
- Are the effects of this scenario specified under all possible circumstances?

PBR: User role-2

- Are the initial conditions for starting up this scenario clear and correct?
- Might some portion of the scenario give different answers depending on how a functional requirement is interpreted?
- Are the interfaces well defined and compatible?
- Do the requirements make sense from what I know about the application or what is specified by the general description?
- Can you get into an improper state, e.g., security or safety problems?

PBR: Designer role

Develop tests as if you are in the role of the individual who will design the given artifact:

- Are all the necessary objects (e.g., data, types, functions) well defined?
- Are all the interfaces defined and consistent?
- Are all the data types compatible?
- Is all the necessary information available to do the design? Are all the conditions involving all objects specified?
- Are there any points that are not clear about what you should do, either because the requirement is not clear or not consistent?
- Is there anything in the requirements that you cannot implement in the design?
- Do the requirements make sense from what I know about the application or what is specified by the general description?

Early Studies of PBR

A Study of PBR ["The empirical investigation of perspective based reading" by Basili et al. In *Empirical Software Engineering* vol. 1 (1996) 133-164] shows that:

- 3 readers find more errors than an unstructured reading of a document.
- Two classes of documents read – a "typical" document and a "precise" document. Results more significant in the "formal" document. In the "typical" document, perspectives found fewer unique errors – Why?

Problem with a study like this – Would like to review 2 documents A and B switching technologies. But not possible to do PBR on A and not do PBR on B second. So PBR always second document to read and learning effects cannot be discounted.

Preparing for a PBR inspection

Try to find unique errors – Take on unique role – view artifact from role of designer, tester, user

- Write down questions you need to ask author
- Work on artifact at your convenience to find these defects
- Estimate type and severity level of defects found
- Try to describe each defect in a few key words.
- DO NOT TRY AND FIX DEFECT. Fixing on the fly leads to: sloppy fixes and missing many other defects.

Data collected at inspection meeting:

- Preparation time
- Size of artifact inspected
- Number of defects found at each severity level
- Number of unresolved questions to ask author
- Author offline fixes defect, fixes specification, decides problem isn't a defect

Automated static analysis

- Static analyzers 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
```

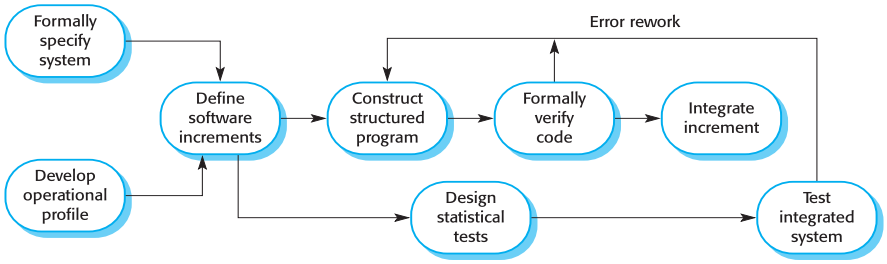
Use of static analysis

- 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.

Cleanroom software development

- The name is derived from the 'Cleanroom' process in semiconductor fabrication. The philosophy is defect avoidance rather than defect removal.
- Developed by Harlan Mills of IBM
- This software development process is based on:
  - Incremental development;
  - Formal specification;
  - Static verification using correctness arguments;
  - Statistical testing to determine program reliability.

The Cleanroom process



Cleanroom process characteristics

- No unit testing!
- Formal specification using a state transition model.
- Incremental development where the customer prioritizes increments.
- Structured programming - limited control and abstraction constructs are used in the program.
- Static verification using rigorous inspections before code is ever executed.
- Statistical testing of the system (covered in Ch. 24).

Formal specification and inspections

- The state based model is a system specification and the inspection process checks the program against this model.
- The programming approach is defined so that the correspondence between the model and the system is clear.
- Mathematical arguments (not proofs) are used to increase confidence in the inspection process.

Cleanroom 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.

Cleanroom process evaluation

- The results of using the Cleanroom process have been very impressive with few discovered faults in delivered systems.
- Independent assessment shows that the process is no more expensive than other approaches.
- There were fewer errors than in a 'traditional' development process.
- "However, the process is not widely used. It is not clear how this approach can be transferred to an environment with less skilled or less motivated software engineers." - Sommerville
  - Not true! – NASA/GSFC Software Engineering Laboratory experience
  - So why is it not used more?

When project is completed:

Postmortem analysis

- Design and promulgate a project survey to collect relevant data.
- Collect objective project information.
- Conduct a debriefing meeting.
- Conduct a project history day.
- Publish the results by focusing on lessons learned.

When post-implementation evaluation is done

Time period	*Percentage of respondents (of 92 organizations)*
Just before delivery	27.8%
At delivery	4.20%
One month after delivery	22.20%
Two months after delivery	6.90%
Three months after delivery	18.10%
Four months after delivery	1.40%
Five months after delivery	1.40%
Six months after delivery	13.90%
Twelve months after delivery	4.20%

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

- Program inspections are very effective in discovering errors.
- Program code in inspections is systematically checked by a small team to locate software faults.
- Static analysis tools can discover program anomalies which may be an indication of faults in the code.
- The Cleanroom development process depends on incremental development, static verification and statistical testing.