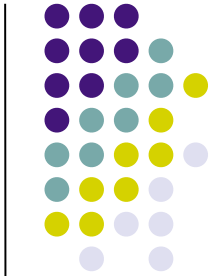


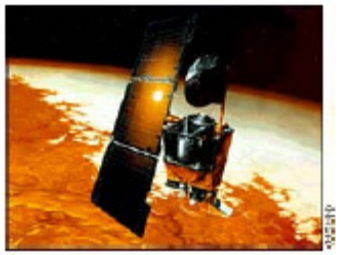
Verification and Validation

Ian Sommerville, SW Engineering, 7th/8th edition Ch 22

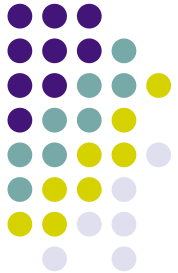
Why Test?

Mars Climate Orbiter

- Purpose: to relay signals from the Mars Polar Lander once it reached the surface of the planet
- Disaster: smashed into the planet instead of reaching a safe orbit
- Why: Software bug - failure to convert English measures to metric values
- $165M



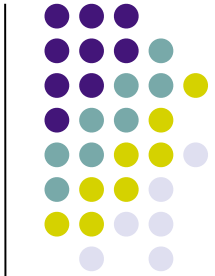
Why Test?



Shooting Down of Airbus 320

- 1988
- US Vicennes shot down Airbus 320
- Mistook airbus 320 for a F-14
- 290 people dead
- Why: Software bug - cryptic and misleading output displayed by the tracking software

Software is Buggy!



- On average, 1-5 errors per 1KLOC
- Windows 2000
  - 35M LOC
  - 63,000 known bugs at the time of release
  - 2 bugs per 1000 lines
- For mass market 100% correct software is infeasible, but
- We must verify software as much as possible

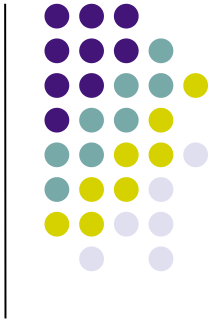
Verification vs validation

- **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

Verification and Validation

- **Verification**: Are we building the product right?
  - To which degree the implementation is consistent with its (formal or semi-formal) specification?
  - Testing, inspections, static analysis, …
- **Validation**: Are we building the right product?
  - To which degree the software fulfills its (informal) requirements?
  - Usability, feedback from users, …

V & V confidence

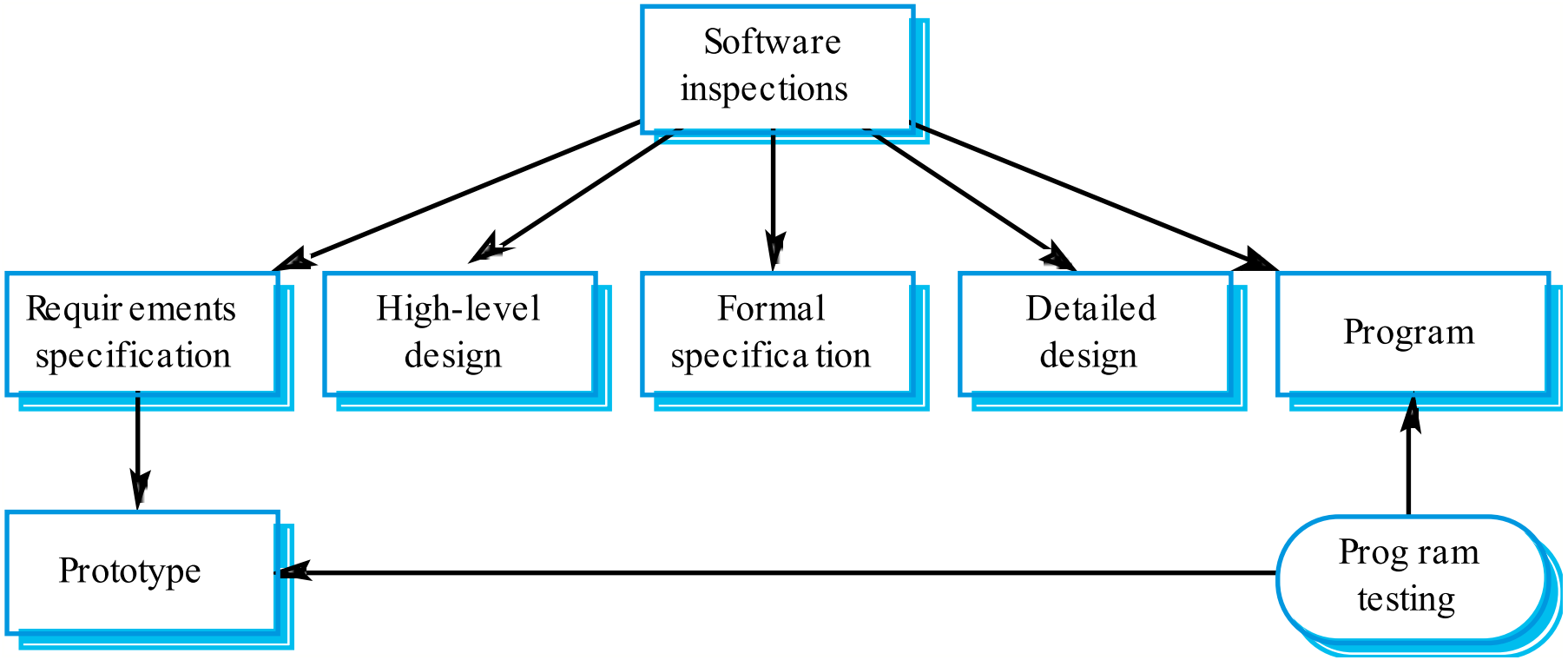


- Depends on system's purpose, user expectations and marketing environment
  - Software function
    - The level of confidence depends on how critical the software is to an organization
  - User expectations
    - Users may have low expectations of certain kinds of software
  - Marketing environment
    - Getting a product to market early may be more important than finding defects in the program

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 behavior (dynamic verification)
  - The system is executed with test data and its operational behavior is observed

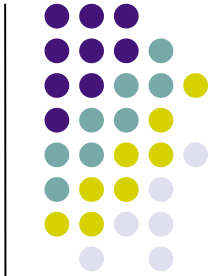
Static and dynamic V&V



Approaches to Verification

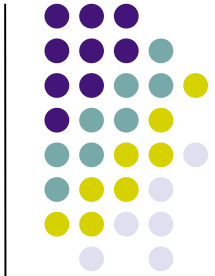
- **Testing**: run software to try and generate failures
- **Static verification**: identify (specific) problems by looking at source code, that is, considering all execution paths statically
- **Inspection/review/walkthrough**: systematic group review of program text to detect faults
- **Formal proof**: proving that the program text implements the program specification

Comparison



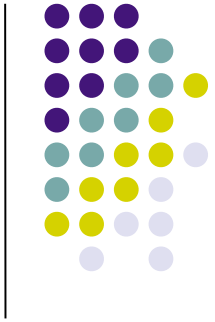
- **Testing**
  - Purpose: reveal failures
  - Limits: small subset of the domain (=> risk of inadequate test set)
- **Static verification**
  - Purpose: consider all program behaviors (and more)
  - Limits: false positives, may not terminate
- **Review**
  - Purpose: systematic in detecting defects
  - Limits: informal
- **Proof**
  - Purpose: prove correctness
  - Limits: complexity/cost (requires a formal spec)

Program testing



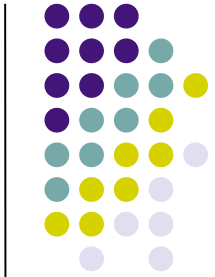
- Can reveal the presence of errors NOT their absence
- The only validation technique for non-functional requirements as the software has to be executed to see how it behaves
- Should be used in conjunction with static verification to provide full V&V coverage

Types of testing



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

Testing and debugging

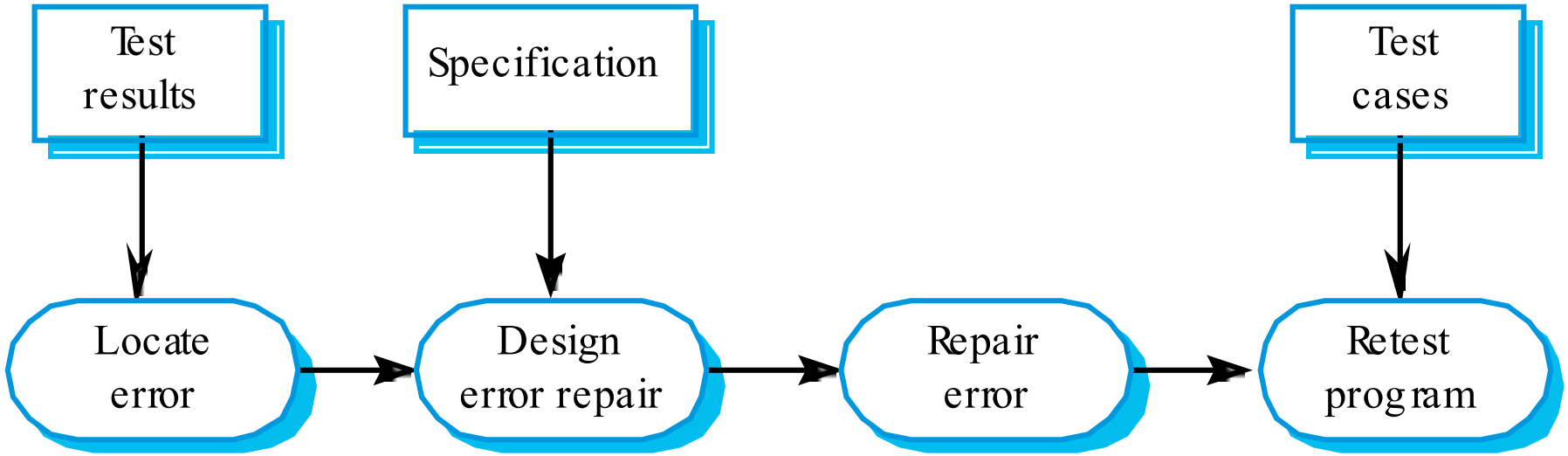


- What is the difference between these two?

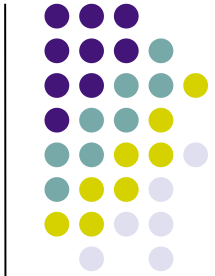
Testing and debugging

- 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

The debugging process

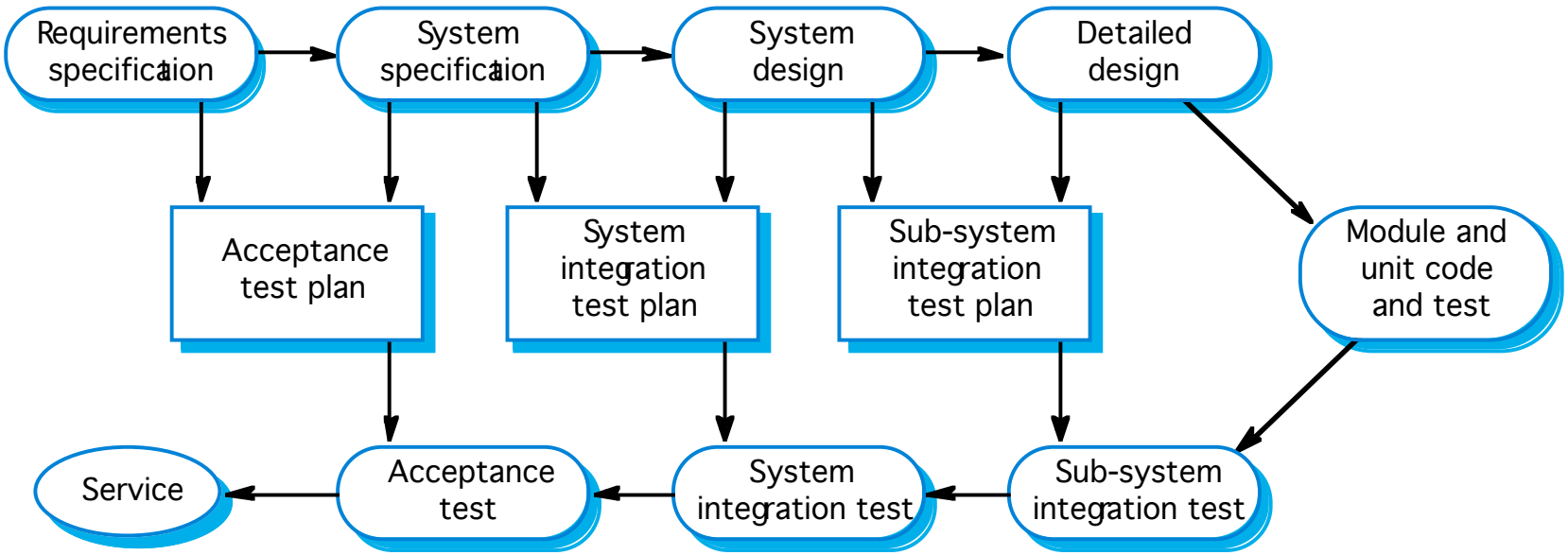


V & V planning



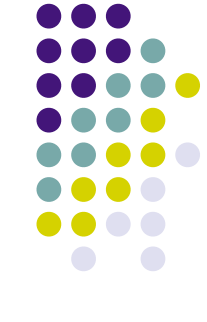
- Careful planning is required to get the most out of testing and inspection processes
- Planning should start early in the development process
- The plan should identify the balance between static verification and testing
- Test planning is about defining standards for the testing process rather than 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



The testing process
A description of the major phases of the testing process. These might be as described earlier in this chapter.

Requirements traceability
Users are most interested in the system meeting its requirements and testing should be planned so that all requirements are individually tested.

Tested items
The products of the software process that are to be tested should be specified.

Testing schedule
An overall testing schedule and resource allocation for this schedule. This, obviously, is linked to the more general project development schedule.

Test recording procedures
It is not enough simply to run tests. The results of the tests must be systematically recorded. It must be possible to audit the testing process to check that it been carried out correctly.

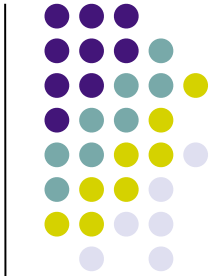
Hardware and software requirements
This section should set out software tools required and estimated hardware utilisation.

Constraints
Constraints affecting the testing process such as staff shortages should be anticipated in this section.

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 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

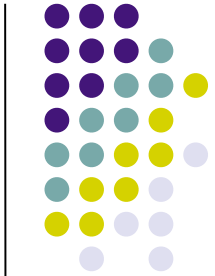
Inspection success



- Many different defects may be discovered in a single inspection. In testing, one defect, may mask another so several executions are required

- They reuse domain and programming knowledge so reviewers are likely to have seen the types of errors 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

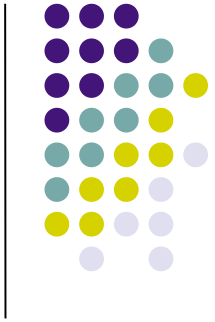
Program inspections

- 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 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

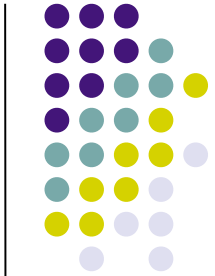


- System overview presented to inspection team
- Code and associated documents are distributed to inspection team in advance
- Inspection takes place and discovered errors are noted
- Modifications are made to repair discovered errors
- Re-inspection may or may not be required

Inspection roles

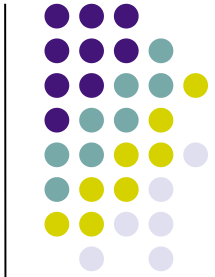
Author or owner	The programmer or designer responsible for producing the program or document. Responsible for fixing defects discovered during the inspection process.
Inspector	Finds errors, omissions and inconsistencies in programs and documents. May also identify broader issues that are outside the scope of the inspection team.
Reader	Presents the code or document at an inspection meeting.
Scribe	Records the results of the inspection meeting.
Chairman or moderator	Manages the process and facilitates the inspection. Reports process results to the Chief moderator.
Chief moderator	Responsible for inspection process improvements, checklist updating, standards development etc.

Inspection checklists



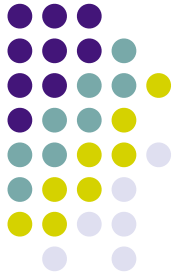
- Checklist of common errors should be used to drive the inspection
- Error checklists are programming language dependent and reflect the characteristic errors that are likely to arise in the language
- In general, the 'weaker' the type checking, the larger the checklist
- Examples: Initialization, Constant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

Inspection checks 1



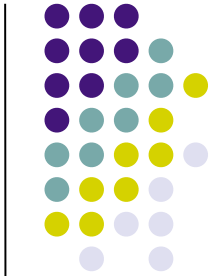
Data faults	Are all program variables initialised before their values are used? Have all constants been named? Should the upper bound of arrays be equal to the size of the array or Size -1? If character strings are used, is a de limiter explicitly assigned? Is there any possibility of buffer overflow?
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
- Inspecting 500 lines costs about 40 man/hours effort – about £2800 at UK rates

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

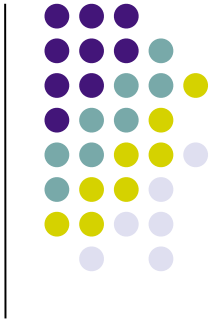
Stages of static analysis

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

LINT static analysis



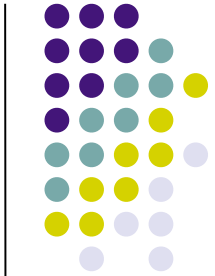
```
138% more lint_ex.c
#include <stdio.h>
printarray (Anarray)
 int Anarray;
{   printf("%d",Anarray);  }

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



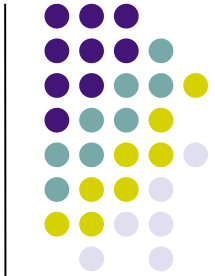
- Particularly valuable when a language such as C is used which has weak typing and hence many errors are undetected by the compiler

- 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
- They involve detailed mathematical analysis of the specification and may develop formal arguments that a program conforms to its mathematical specification

Arguments for formal methods



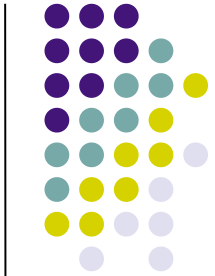
- Producing a mathematical specification requires a detailed analysis of the requirements and this is likely to uncover errors

- They can detect implementation errors before testing when the program is analyzed alongside the specification

Arguments against formal methods

- Require specialized 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

Other implications for formal methods

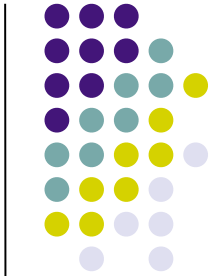


- The specs may not reflect real requirements
  - In this case formal methods can not detect problems; furthermore the users can not understand formal notation
- The proof may contain errors
  - Program proofs are large and complex, thus more prone to “bugs”
- The proof may assume a usage pattern which is incorrect
  - If the system is not used as anticipated, the proofs may be invalid

Cleanroom software development

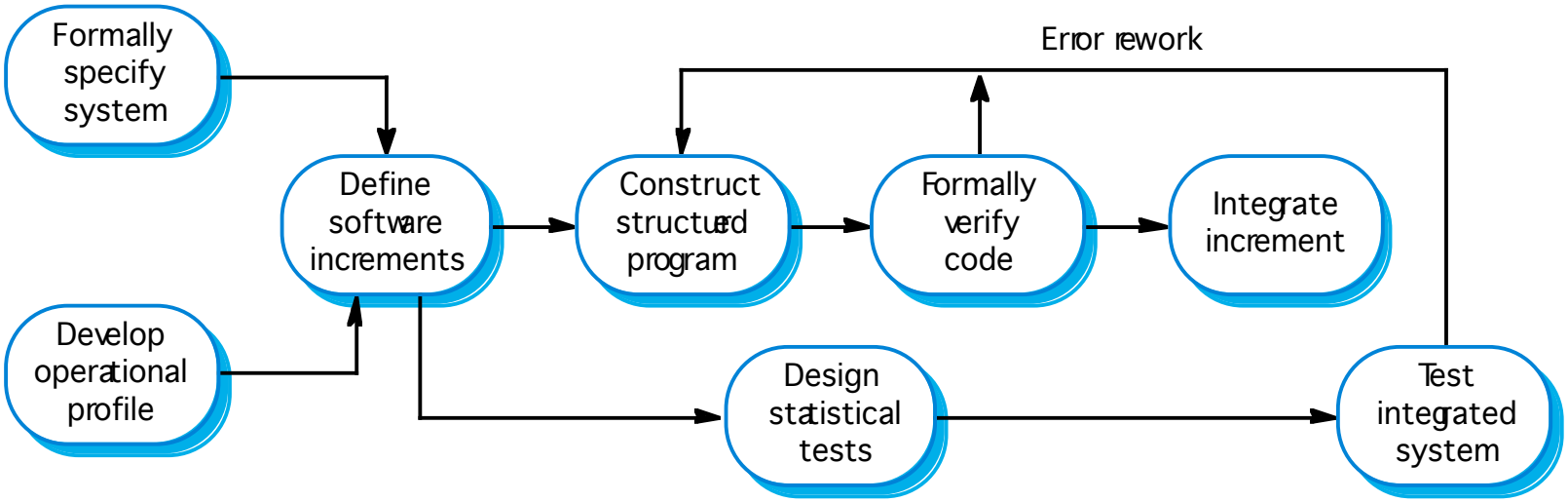
- The name is derived from the 'Cleanroom' process in semiconductor fabrication. The philosophy is defect avoidance rather than defect removal
- This software development process is based on:
  - Incremental development
  - Formal specification
  - Static verification using correctness arguments
  -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

Cleanroom process characteristics

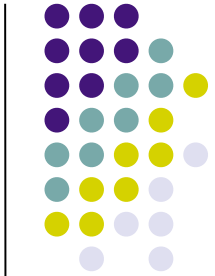


- Formal specification using a state transition model
- Incremental development where the customer prioritizes increments
- Structured programming – limited control and abstraction constructs are used in the program
- Static verification using rigorous inspections
- Statistical testing of the system (covered in Ch. 24)

The Cleanroom process



Formal specification and inspections

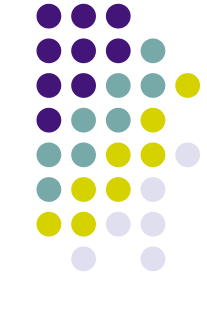


- The state based model is a system specification and the inspection process checks the program against this model
- The programming approach is defined so that the correspondence between the model and the system is clear
- Mathematical arguments (not proofs) are used to increase confidence in the inspection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

Cleanroom process teams

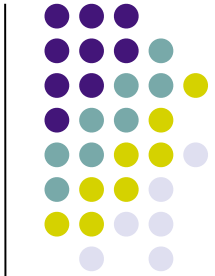
- Specification team. Responsible for developing and maintaining the system specification
- Development team. Responsible for developing and verifying the software. The software is NOT executed or even compiled during this process
- Certification team. Responsible for developing a set of statistical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

Cleanroom process evaluation



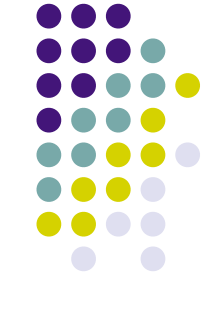
- The results of using the Cleanroom process have been very impressive with few discovered faults in delivered systems
- Independent assessment shows that the process is no more expensive than other approaches
- There were fewer errors than in a 'traditional' development process
- However, the process 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

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