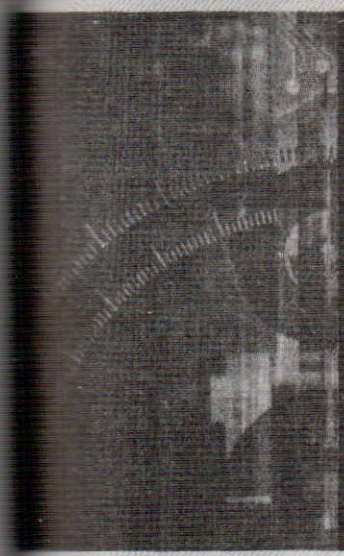


PART FIVE

Verification and Validation

19 Verification and validation

Objectives

The objective of this chapter is to introduce software verification and validation with a particular focus on static verification techniques. When you have read this chapter, you will:

- understand the distinctions between software verification and software validation;
- have been introduced to program inspections as a method of discovering defects in programs;
- understand why static analysis of programs is an important verification technique;
- understand the Cleanroom method of program development and why it can be effective.

Contents

Verification and validation (V & V) is the name given to the checking and analysis processes that ensure that software conforms to its specification and meets the needs of the customers who are paying for that software. Verification and validation is a whole life-cycle process. It starts with requirements reviews and continues through design reviews and code inspections to product testing. There should be V & V activities at each stage of the software process. These activities check that the results of process activities are as specified.

Verification and validation are not the same thing although they are easily confused. The difference between them is succinctly expressed by Boehm (1979):

- 'Validation: Are we building the right product?'
- 'Verification: Are we building the product right?'

These definitions tell us that the role of verification involves checking that the software conforms to its specification. You should check that the system meets its specified functional and non-functional requirements. Validation, however, is a more general process. You should ensure that the software meets the expectations of the customer. It goes beyond checking conformance of the system to its specification to showing that the software does what the customer expects as distinct from what has been specified.

As discussed in Chapter 6, early validation of the system requirements is very important. It is easy to make errors and omissions in the system's requirements and, in such cases, the final software will probably not meet its customer's expectations. However, in reality, requirements validation is unlikely to discover all requirements problems. Some flaws and deficiencies in the requirements can sometimes only be discovered when the system implementation is complete.

Within the V & V process, two techniques of system checking and analysis may be used:

1. *Software inspections* analyse and check system representations such as the requirements document, design diagrams and the program source code. They may be applied at all stages of the process. Inspections may be supplemented by some automatic analysis of the source text of a system or associated documents. Software inspections and automated analyses are static V & V techniques as they do not require the system to be executed.
2. *Software testing* involves executing an implementation of the software with test data and examining the outputs of the software and its operational behaviour to check that it is performing as required. Testing is a dynamic technique of verification and validation because it works with an executable representation of the system.

Figure 19.1 shows the place of software inspections and testing in the software process. The arrows indicate the stages in the process where the techniques may be used. Therefore, software inspections can be used at all stages of the software process. Testing, however, can only be used when a prototype or an executable program is available.

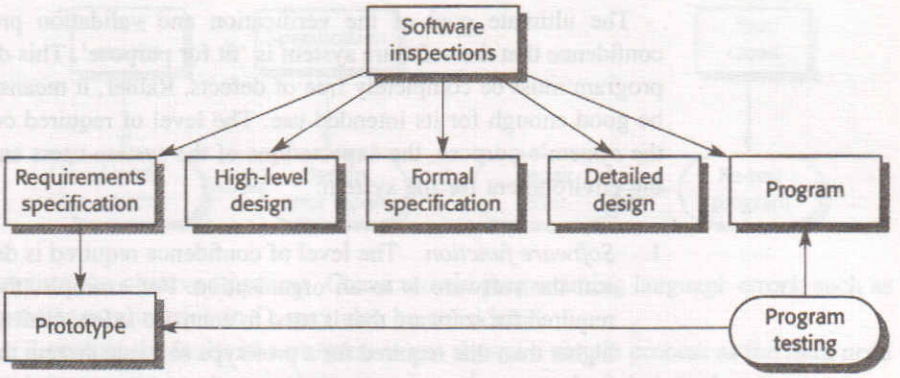


Figure 19.1
Static and dynamic verification and validation

Inspection techniques include program inspections, automated source code analysis and formal verification. However, static techniques can only check the correspondence between a program and its specification (verification); they cannot demonstrate that the software is operationally useful. Nor can they check non-functional characteristics of the software such as its performance and reliability. Therefore, some testing is always required to validate a software system.

Although software inspections are now widely used, program testing is still the predominant verification and validation technique. Testing involves exercising the program using data like the real data processed by the program. The existence of program defects or inadequacies is inferred by examining the outputs of the program and looking for anomalies. Testing may be carried out during the implementation phase to verify that the software behaves as intended by its designer and after the implementation is complete.

There are two distinct types of testing that may be used at different stages in the software process:

1. *Defect testing* is intended to find inconsistencies between a program and its specification. These inconsistencies are usually due to program faults or defects. The tests are designed to reveal the presence of defects in the system rather than to simulate its operational use. I cover this type of testing in Chapter 20.
2. *Statistical testing* is used to test the program's performance and reliability and to check how it works under operational conditions. Tests are designed to reflect the actual user inputs and their frequency. After running the tests, an estimate of the operational reliability of the system can be made by counting the number of observed system failures. Program performance may be judged by measuring the execution time and the response time of the system as it processes the statistical test data. I discuss statistical testing and reliability estimation in Chapter 21.

Of course, there is not a hard and fast boundary between these approaches to testing. During defect testing, testers will get an intuitive feel for the software's reliability; during statistical testing, it is likely that some defects will be discovered.

The ultimate goal of the verification and validation process is to establish confidence that the software system is 'fit for purpose'. This does not mean that the program must be completely free of defects. Rather, it means that the system must be good enough for its intended use. The level of required confidence depends on the system's purpose, the expectations of the system users and the current marketing environment for the system:

1. *Software function* The level of confidence required is dependent on how critical the software is to an organisation. For example, the level of confidence required for software that is used to control a safety-critical system is very much higher than that required for a prototype software system that has been developed to demonstrate some new ideas.
2. *User expectations* It is a sad reflection on the software industry that many users have low expectations of their software and are not surprised when it fails during use. They are willing to accept these system failures when the benefits of use outweigh the disadvantages. However, user tolerance of system failures has been decreasing since the 1990s. It is now less acceptable to deliver unreliable systems, so software companies must devote more effort to verification and validation.
3. *Marketing environment* When a system is marketed, the sellers of the system must take into account competing programs, the price that customers are willing to pay for a system and the required schedule for delivering that system. Where a company has few competitors, it may decide to release a program before it has been fully tested and debugged because they want to be the first into the market. Where customers are not willing to pay high prices for software, they may be willing to tolerate more software faults. All of these factors must be considered when deciding how much effort should be spent on the V & V process.

During software verification and validation, defects in the program are normally discovered and the program must then be modified to correct these defects. This *debugging process* is often integrated with other verification and validation activities. However, testing (or, more generally, verification and validation) and debugging are different processes that do not have to be integrated:

1. Verification and validation is a process that establishes the existence of defects in a software system.
2. Debugging is a process (Figure 19.2) that locates and corrects these defects.

There is no simple method for program debugging. Skilled debuggers look for patterns in the test output where the defect is exhibited and use knowledge of the type of defect, the output pattern, the programming language and the programming process to locate the defect. Process knowledge is important. Debuggers know of common programmer errors (such as failing to increment a counter) and match these

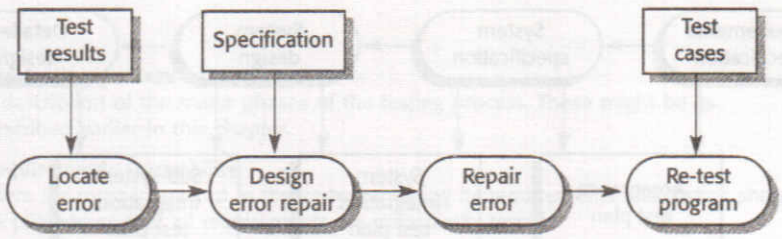


Figure 19.2 The debugging process

against the observed patterns. Characteristic programming language errors, such as pointer misdirection in C, may also be considered.

Locating the faults in a program is not always a simple process as the fault need not necessarily be close to the point where the program failed. To locate a program fault, the programmer responsible for debugging may have to design additional program tests that repeat the original fault and that help discover the source of the fault in the program. Manual tracing of the program, simulating execution, may be necessary. In some cases, debugging tools that collect information about the program's execution may be helpful.

Interactive debugging tools are generally part of a suite of language support tools that are integrated with a compilation system. They provide a specialised run-time environment for the program that allows access to the compiler symbol table and, from there, to the values of program variables. Users can often control execution by 'stepping' their way through the program statement by statement. After each statement has been executed, the values of variables can be examined and potential errors discovered.

After a defect in the program has been discovered, it must be corrected and the system should then be revalidated. This may involve reinspecting the program or repeating previous test runs (regression testing). Regression testing is used to check that the changes made to a program have not introduced new faults into the system. Experience has shown that a relatively high proportion of 'fault repairs' are either incomplete repairs or introduce new faults into the program.

In principle during regression testing, all tests should be repeated after every defect repair; in practice, this is too expensive. As part of the test plan, dependencies between parts of the system and the tests associated with each part should be identified. That is, there should be traceability from the test cases to the program features that are tested. If this traceability is documented, you may then run a subset of the entire test data set to check the modified component and its dependants.

19.1 Verification and validation planning

Verification and validation is an expensive process. For some large systems, such as real-time systems with complex non-functional constraints, half the system

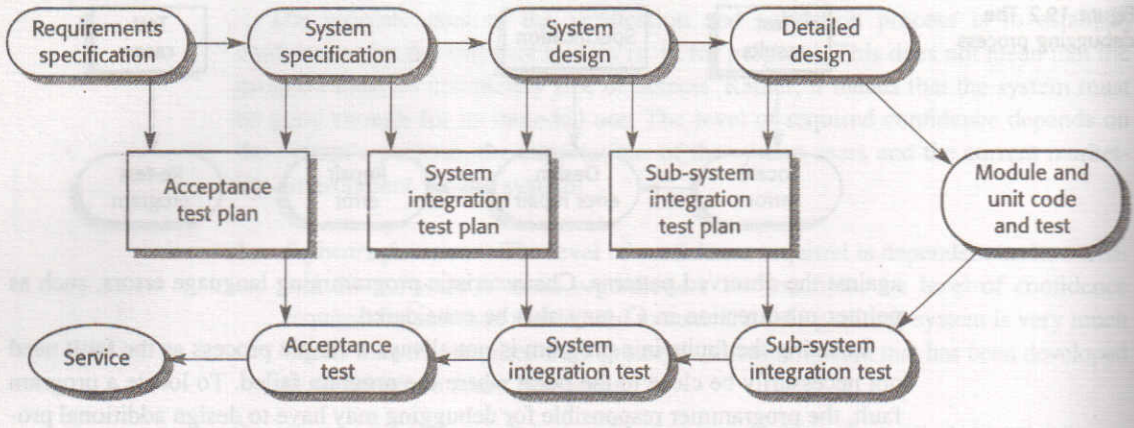


Figure 19.3 Test plans as a link between development and testing

development budget may be spent on V & V. Careful planning is needed to get the most out of inspections and testing and to control the costs of the verification and validation process.

The planning of validation and verification of a software system should start early in the development process. The model of software development shown in Figure 19.3 shows how test plans should be derived from the system specification and design. This is sometimes called the V-model (turn Figure 19.3 on end to see the V). This diagram also shows how the verification and validation activity is broken down into a number of stages with each phase driven by tests that have been defined to check the conformance of the program with its design and specification.

The V & V planning process should decide on the balance between static and dynamic approaches to verification and validation, draw up standards and procedures for software inspections and testing, establish checklists to drive program inspections (see section 19.2) and define the software test plan. The relative effort devoted to inspections and testing depends on the type of system being developed and the organisational expertise. The more critical a system, the more effort should be devoted to static verification techniques.

Test planning is concerned with setting out standards for the testing process rather than describing product tests. Test plans are not just management documents. They are also intended for software engineers involved in designing and carrying out system tests. They allow technical staff to get an overall picture of the system tests and to place their own work in this context. Test plans also provide information to staff who are responsible for ensuring that appropriate hardware and software resources are available to the testing team.

The major components of a test plan are shown in Figure 19.4. This plan should include significant amounts of contingency so that slippages in design and implementation can be accommodated and staff allocated to testing can be deployed in other activities. A good description of test plans and their relation to more general quality plans is given in Frewin and Hatton (1986).

Figure 19.4
The structure of a software test plan

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 which 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.

Like other plans, the test plan is not a static document. It should be revised regularly as testing is an activity that is dependent on implementation being complete. If part of a system is incomplete, it cannot be delivered for integration testing as discussed in the following chapter. The plan must therefore be revised to redeploy the testers to some other activity.

19.2 Software inspections

Systematic program testing requires a large number of tests to be developed, executed and examined. This means that the process is time-consuming and expensive. Each test run tends to discover a single program fault or, at best, only a few faults. The reason for this is that failures resulting from system faults often cause system data corruption. Consequently, it can be difficult to tell if output anomalies are a result of a new fault or are a side-effect of an existing fault.

Software inspections do not require the program to be executed so may be used as a verification technique before programs are implemented. During an inspection, you examine the source representation of a system. This could be a system model,

a specification or high-level language code. You use knowledge of the system to be developed and the semantics of the source representation to discover errors. Each error can be considered in isolation without being concerned about how it will affect the behaviour of the system. Error interactions are not significant and an entire component can be verified in a single session.

Inspections have proved to be an effective technique of error detection. Errors can be found more cheaply through inspection than by extensive program testing. This was demonstrated in an experiment by Basili and Selby (1987) who empirically compared the effectiveness of inspections and testing. They found that static code reviewing was more effective and less expensive than defect testing in discovering program faults. Gilb and Graham (1993) have also found this to be true.

Fagan (1986) reported that more than 60 per cent of the errors in a program can be detected using informal program inspections. Mills *et al.* (1987) suggest that a more formal approach, using mathematical verification, can detect more than 90 per cent of the errors in a program. This technique is used in the Cleanroom process which I describe in section 19.3. The inspection process can also consider other quality attributes such as compliance with standards, portability and maintainability. These quality attributes are discussed in Chapter 24.

There are two reasons why reviews and inspections are usually more effective than testing for discovering defects in components and sub-systems:

1. Many different defects may be detected in a single inspection session. The problem with testing is that it can often detect only one error per test because defects can cause the program to crash or interfere with the symptoms of other program defects.
2. They reuse domain and programming language knowledge. In essence, the reviewers are likely to have seen the types of error that commonly occur in particular programming languages and in particular types of application. They can therefore focus on these error types during the analysis.

This does not mean that inspections should completely replace system testing. Rather, they should be used as an initial verification process to find most program defects. Inspections of the software can check conformance with a specification but they cannot validate dynamic behaviour. Furthermore, it is often impractical to inspect a complete system that is integrated from a number of different sub-systems. Testing is the only possible V & V technique at the system level. Testing is also necessary for reliability assessment, performance analysis, user interface validation and to check that the software requirements are what the user really wants.

Reviews and testing are not competing V & V techniques. They have their own advantages and disadvantages and should be used together in the verification and validation process. Indeed, Gilb and Graham (1993) suggest that one of the most effective uses of reviews is to review the test cases for a system. Reviews can discover problems with these tests and can help design more effective ways to test the system.

It is sometimes difficult to introduce inspections into traditional software development organisations. Software engineers with experience of program testing may be reluctant to accept that these techniques can be more effective than testing for defect detection. Managers may be suspicious of these techniques because they require additional costs during design and development and they may not wish to take the risk that there will not be corresponding savings during program testing. Inspections inevitably 'front-load' software V & V costs and only result in cost savings after the development teams become experienced in their use.

I discuss software inspections in this chapter by describing program inspections where the program source code is inspected for defects. However, software inspections may be applied to any documents produced during the software process. Comparable inspection techniques can be applied to requirements specifications, detailed design definitions, data structure designs, test plans and user documentation.

19.2.1 Program inspections

Program inspections are reviews whose objective is program defect detection. The notion of a formalised inspection process was first developed at IBM in the 1970s and is described by Fagan (1976, 1986). It is now a widely used method of program verification. From Fagan's original method, a number of alternative approaches to inspection have been developed (Gilb and Graham, 1993). However, these are all based on Fagan's original notion that a team with members from different backgrounds should make a careful, line-by-line review of the program source code.

The key difference between program inspections and other types of quality review is that the principal goal of inspections is defect detection rather than to consider broader design issues. Defects may be either logical errors, anomalies in the code that might indicate an erroneous condition or non-compliance with organisational or project standards. By contrast, other types of review may be more concerned with schedule, costs, progress against defined milestones or assessing whether or not the software is likely to meet organisational goals.

The process of inspection is a formal one carried out by a small team of at least four people. Team members systematically analyse the code and point out possible defects. In Fagan's original proposals, he suggested roles such as author, reader, tester and moderator. The reader reads the code aloud to the inspection team, the tester inspects the code from a testing perspective and the moderator organises the process.

As organisations have gained experience with inspection, other proposals for team roles have emerged. In a discussion of how inspection was successfully introduced in Hewlett-Packard's development process, Grady and Van Slack (1994) suggest six roles as shown in Figure 19.5. Different roles may be adopted by the same person, so the team size may vary from one inspection to another.

Grady and Van Slack report that there is not always a need for a reader role. In this respect, they have modified the process from that originally proposed by Fagan,

Figure 19.5 Roles in the inspection process

Role	Description
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 which are outside the scope of the inspection team.
Reader	Paraphrases 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.

where an integral part of the process involved reading the program aloud. Gilb and Graham (1993) also do not think that a reader is needed. They suggest that inspectors should be selected to reflect different viewpoints such as testing, end-user, quality management, etc.

Before a program inspection begins, it is essential that:

1. There is a precise specification of the code to be inspected. It is impossible to inspect a component at the level of detail required to detect defects without a complete specification.
2. The members of the inspection team are familiar with the organisational standards.
3. There is an up-to-date, syntactically correct version of the code available. There is no point in inspecting code which is 'almost complete' even if a delay causes schedule disruption.

A very general inspection process is shown in Figure 19.6. This is adapted as required by organisations using program inspections.

The moderator is responsible for inspection planning. This involves selecting an inspection team, organising a meeting room and ensuring that the material to be inspected and its specifications are complete. The program to be inspected is presented to the inspection team during the overview stage where the author of the code describes what the program is intended to do. This is followed by a period of individual preparation. Each inspection team member studies the specification and the program and looks for defects in the code.

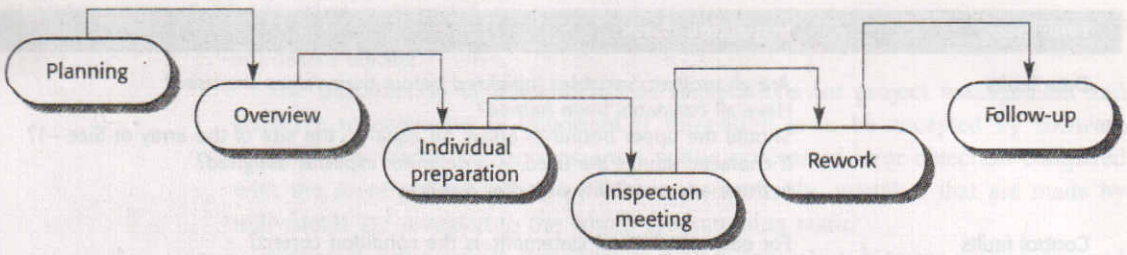


Figure 19.6 The inspection process

The inspection itself should be relatively short (no more than two hours) and should be exclusively concerned with identifying defects, anomalies and non-compliance with standards. The inspection team should not suggest how these defects should be corrected nor recommend changes to other components.

Following inspection, the program is modified by its author to correct the identified problems. In the follow-up stage, the moderator must decide whether a reinspection of the code is required. Alternatively, he or she may decide that a complete reinspection is not required and that the defects have been successfully fixed. The document is then approved by the moderator for release.

The inspection process should always be driven by a checklist of common programmer errors. This should be established by discussion with experienced staff and regularly updated as more experience is gained of the inspection process. Different checklists should be prepared for different programming languages.

This checklist varies according to programming language because of the different levels of checking provided by the language compiler. For example, an Ada compiler checks that functions have the correct number of parameters, a C compiler does not. Possible checks which might be made during the inspection process are shown in Figure 19.7. Gilb and Graham (1993) emphasise that each organisation should develop its own inspection checklist. This should be based on local standards and practices and should be updated as new types of defects are found.

As an organisation gains experience of the inspection process, it can use the results of that process as a means of process improvement. An analysis of defects found during the inspection process can be made. The inspection team and the authors of the code which was inspected can suggest reasons why these defects occurred. Wherever possible, the process should then be modified to eliminate the reasons for defects so they will not recur in future systems.

Gilb and Graham report that a number of organisations have abandoned unit testing in favour of inspections. They have found that program inspections are so effective at finding errors that the costs of unit testing are not justifiable. As I discuss later in the chapter, software inspections have replaced unit testing in the Cleanroom software development process.

The amount of code which can be inspected in a given time depends on the experience of the inspection team, the programming language and the application domain. When the inspection process was measured in IBM, Fagan made the following observations:

Fault class	Inspection check
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?
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?

Figure 19.7 Inspection checks

1. About 500 source code statements per hour can be considered during the overview stage.
2. During individual preparation, about 125 source code statements per hour can be examined.
3. From 90 to 125 statements per hour can be inspected during the meeting.

These figures are confirmed by data collected by AT & T (Barnard and Price, 1994) where measurements of the inspection process showed comparable values.

Fagan suggests that the maximum time spent on an inspection should be about two hours as the efficiency of the defect detection process falls off after that time. Inspection should therefore be a frequent process, carried out on relatively small software components, during program development.

With four people involved in an inspection team, the cost of inspecting 100 lines of code is roughly equivalent to one person-day of effort. This assumes that the inspection itself takes about an hour and that each team member spends 1–2 hours in preparing for the inspection. Testing costs are very variable and depend on the number of faults in the program. However, the effort required for the program

inspection is probably less than half the effort that would be required for equivalent defect testing.

The introduction of assessment has implications for project management and sensitive management is important if inspection is to be accepted by software development teams. Inspections are a public process of error detection compared with the more private testing process and, inevitably, mistakes that are made by individuals are revealed to the whole programming team.

Managers must ensure that there is a clear separation between inspections and personnel appraisals. Inspection results that reveal errors should never be used in career assessments of engineers. Inspection team leaders must be carefully trained to manage the process and to develop a culture where support is provided when errors are discovered and there is no notion of blame associated with these errors.

19.3 Automated static analysis

Static program analysers are software tools which scan the source text of a program and detect possible faults and anomalies. They do not require the program to be executed. Rather, they parse the program text and thus recognise the different types of statement in the program. They can then detect whether or not statements are well formed, make inferences about the control flow in the program and, in many cases, compute the set of all possible values for program data. They complement the error detection facilities provided by the language compiler.

The intention of automatic static analysis is to draw the verifier's attention to anomalies in the program such as variables that are used without initialisation, variables which are unused, data whose value could go out of range, etc. Some of the checks which can be detected by static analysis are shown in Figure 19.8. While these are not necessarily erroneous conditions, it is often the case that many of these anomalies are a result of programming errors or omissions. Automated static analysis is best used with software inspections. It provides additional information for the inspection team.

The stages involved in static analysis include:

1. *Control flow analysis* This stage identifies and highlights loops with multiple exit or entry points and unreachable code. Unreachable code is code that is surrounded by unconditional goto statements or which is in a branch of a conditional statement where the guarding condition can never be true.
2. *Data use analysis* This stage highlights how variables in the program are used. It detects variables that are used without previous initialisation, variables that are written twice without an intervening assignment and variables that are declared but never used. Data use analysis also discovers ineffective tests where the test

Figure 19.8 Automated 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

condition is redundant. Redundant conditions are conditions whose value never changes – they are either always true or always false.

3. *Interface analysis* This analysis checks the consistency of routine and procedure declarations and their use. It is unnecessary if a strongly typed language like Java is used for implementation as the compiler carries out these checks. Interface analysis can detect type errors in weakly typed languages like FORTRAN and C. Interface analysis can also detect functions and procedures which are declared and never called or function results that are never used.

4. *Information flow analysis* This phase of the analysis identifies the dependencies between input variables and output variables. While it does not detect anomalies, the derivation of the values used in the program are explicitly listed. Erroneous derivations should therefore be easier to detect during a code inspection or review. Information flow analysis can also show the conditions that affect a variable's value.

5. *Path analysis* This phase of semantic analysis identifies all possible paths through the program and sets out the statements executed in that path. It essentially unravels the program's control and allows each possible predicate to be analysed individually.

Information flow analysis and path analysis generate an immense amount of information. This information does not highlight anomalous conditions but simply presents the program from a different viewpoint. Because of the large amount of information generated, these phases of static analysis are sometimes left out of the process. Only the early phases, which detect anomalous conditions directly, are used.

Figure 19.9 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
```

Static analysers are particularly valuable when a programming language like C is used. C does not have strict type rules and the checking which the C compiler can do is limited. Therefore, there is a great deal of scope for programmer errors which can be automatically discovered by the analysis tool. This is particularly important when C (and to a lesser extent C++) is used for critical systems development. In this case, static analysis can significantly reduce testing costs.

Unix and Linux systems include a static analyser called LINT for C programs. LINT provides static checking which is equivalent to that provided by the compiler in a strongly typed language such as Java. An example of the output produced by LINT is shown in Figure 19.9. In this transcript of a Unix terminal session, commands are shown in italics. The first command lists the (nonsensical) program. It defines a function with one parameter called printarray, then calls this function with three parameters. Variables i and c are declared but are never assigned values. The value returned by the function is never used.

The line numbered 139 shows the C compilation of this program with no errors reported by the C compiler. This is followed by a call of the LINT static analyser which detects and reports program errors.

The static analyser shows that the scalar variables c and i have been used but not initialised and that printarray has been called with a different number of arguments than are declared. It also identifies the inconsistent use of the first argument in printarray and the fact that the function value is never used.

Tool-based analysis cannot replace inspections as there are some types of error that static analysers cannot detect. For example, they can detect uninitialised variables

but they cannot detect initialisations that are incorrect. In weakly typed languages like C, static analysers can detect functions that have the wrong numbers and types of arguments but they cannot detect situations where an incorrect argument of the correct type has been passed to a function.

There is no doubt that, for languages like C, static analysis is an effective technique for discovering program errors. However, with modern programming languages like Java, the language designers have removed some error-prone language features. All variables must be initialised, there are no goto statements so unreachable code is less likely to be created accidentally and storage management is automatic. This approach of error avoidance rather than error detection is more effective in improving program reliability. Therefore, automatic static analysis may not be cost-effective for Java programs.

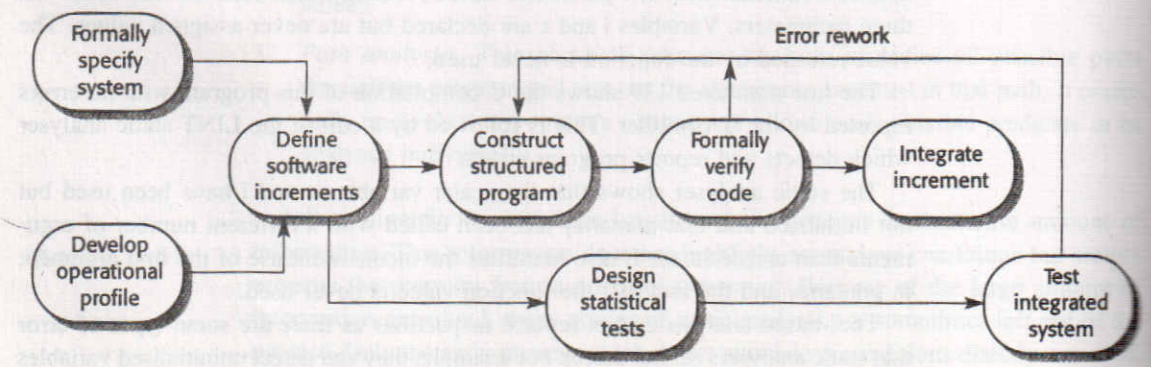
19.4 Cleanroom software development

Cleanroom software development (Mills *et al.*, 1987; Cobb and Mills, 1990; Linger, 1994; Prowell *et al.*, 1999) is a software development philosophy that is based on avoiding software defects by using a rigorous inspection process. The objective of this approach to software development is zero-defect software. The name 'Cleanroom' was derived by analogy with semiconductor fabrication units. In these units (cleanrooms) defects are avoided by manufacturing in an ultra-clean atmosphere. I discuss it in this chapter because it has replaced the unit testing of system components by inspections to check the consistency of these components with their specifications.

A model of the Cleanroom process, adapted from the description given by Linger (1994), is shown in Figure 19.10.

Figure 19.10 The Cleanroom development process

The Cleanroom approach to software development is based on five key characteristics:



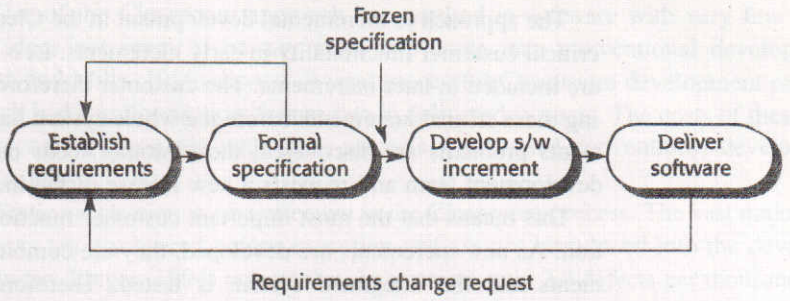


Figure 19.11
An incremental development process

1. *Formal specification* The software to be developed is formally specified. A state-transition model which shows system responses to stimuli is used to express the specification.
2. *Incremental development* The software is partitioned into increments which are developed and validated separately using the Cleanroom process. These increments are specified, with customer input, at an early stage in the process.
3. *Structured programming* Only a limited number of control and data abstraction constructs are used. The program development process is a process of stepwise refinement of the specification. A limited number of constructs are used and the aim is to apply correctness-preserving transformations (discussed in Chapter 3) to the specification to create the program code.
4. *Static verification* The developed software is statically verified using rigorous software inspections. There is no unit or module testing process for code components.
5. *Statistical testing of the system* The integrated software increment is tested statistically, as discussed in Chapter 21, to determine its reliability. These statistical tests are based on an operational profile which is developed in parallel with the system specification as shown in Figure 19.10.

Incremental development, shown in Figure 19.11 and discussed earlier in Chapter 3, involves producing and delivering the software in discrete increments. An increment can be executed by user commands and is a useful, limited, system in its own right. Users feed back reports of the system and propose changes that are required. Incremental development is important because it minimises the disruption caused to the development process by customer-requested requirements changes.

When a specification is specified as a single unit, customer requirements changes (which are inevitable) disrupt the specification and development process. The specification and design must be continually reworked. With incremental development, the specification for the increment is frozen although change requests for the rest of the system are accepted. The software increment is delivered on completion.

The approach to incremental development in the Cleanroom process is to deliver critical customer functionality in early increments. Less important system functions are included in later increments. The customer therefore has the opportunity of trying these critical increments before the whole system has been delivered. If requirements problems are discovered, the customer feeds back this information to the development team and requests a new release of the increment.

This means that the most important customer functions receive the most validation. As new increments are developed, they are combined with the existing increments and the integrated system is tested. Therefore, existing increments are re-tested with new test cases as new system increments are added.

The Cleanroom process is designed to support rigorous program inspection. A state-based system model is produced to serve as a system specification and this is refined through a series of different system models to an executable program. The approach used for development is based on well-defined transformations that attempt to preserve the correctness at each transformation to a more detailed representation. This general approach to development is discussed in Chapter 3. Program inspections are supplemented by rigorous mathematical arguments which demonstrate that the output of the transformation is consistent with its input.

The mathematical arguments used in the Cleanroom process are weaker than formal mathematical proofs. Formal mathematical proofs that a program is correct with respect to its specification are very expensive to develop. They depend on using knowledge of the formal semantics of the programming language to construct theories that relate the program and its formal specification. These theories must then be proven mathematically, often with the assistance of large and complex theorem-prover programs. Because of their high cost and the specialist skills that are needed, proofs are usually only developed for the most safety-critical or security-critical applications.

There are three teams involved when the Cleanroom process is used for large system development:

1. *The specification team* This group is responsible for developing and maintaining the system specification. Customer-oriented specifications (requirements definition) and mathematical specifications for verification are produced by this team. In some cases, when the specification is complete, the specification team will also take responsibility for development.

2. *The development team* This team has the responsibility of developing and verifying the software. The software is not executed during the development process. A structured, formal approach to verification based on inspection of code supplemented with correctness arguments is used.

3. *The certification team* This team is responsible for developing a set of statistical tests to exercise the software after it has been developed. These tests are based on the formal specification. Test case development is carried out in parallel with software development. The test cases are used to certify the software reliability. Reliability growth models, discussed in Chapter 21, may be used to decide when to stop testing.

Use of the Cleanroom approach has resulted in software with very few errors and does not seem to be any more expensive than conventional development. Cobb and Mills (1990) discuss several successful Cleanroom development projects which had a uniformly low failure rate in delivered systems. The costs of these projects were comparable with other projects which used conventional development techniques.

Static verification is cost-effective in the Cleanroom process. The vast majority of defects are discovered before execution and are not introduced into the developed software. Linger (1994) reports that, on average, only 2.3 defects per thousand lines of source code were discovered during testing for Cleanroom projects. Overall development costs are not increased because less effort is required to test and repair the developed software.

Selby *et al.* (1987), using students as developers, compared Cleanroom development with conventional techniques. They found that most teams could successfully use the Cleanroom method. The programs produced were of higher quality than those developed using traditional techniques – the source code had more comments and a simpler structure. More of the Cleanroom teams met the development schedule.

Cleanroom development works when practised by skilled and committed engineers. However, use of the method has been confined to technologically advanced organisations and its adoption has been relatively slow. Reports of the success of the Cleanroom approach in industry have mostly, though not exclusively, come from its developers. We don't know if this process can be transferred effectively to other types of software development organisation. These organisations usually have fewer, less committed and less skilled engineers. Transferring the Cleanroom approach to these organisations still remains a challenge.

KEY POINTS

- Verification and validation are not the same thing. Verification is intended to show that a program meets its specification. Validation is intended to show that the program does what the user requires.
- Test plans should include a description of the items to be tested, the testing schedule, the procedures for managing the testing process, the hardware and software requirements and any testing problems which are likely to arise.
- Static verification techniques involve examination and analysis of the program source code to detect errors. They should be used with program testing as part of the V & V process.

- Program inspections are effective in finding program errors. The aim of an inspection is to locate faults. The inspection process should be driven by a fault checklist.
- Program code is systematically checked by a small team. Team members include a team leader or moderator, the author of the code, a reader who presents the code during the inspection and a tester who considers the code from a testing perspective.
- Static analysers are software tools which process program source code and draw attention to anomalies such as unused code sections and uninitialised variables. These anomalies may be the result of faults in the code.
- Cleanroom software development is an approach to software development that relies on static techniques for program verification and statistical testing for system reliability certification. It has been successful in producing systems which have a high level of reliability.

FURTHER READING

Cleanroom Software Engineering: Technology and Process. A fairly new book on the Cleanroom approach that has sections on the basics of the technique, the process and a practical case study. (S. J. Powell, C. J. Trammell, R. C. Linger, J. H. Poore, 1999, Addison Wesley Longman.)

Software Inspection. A very thorough book which covers program inspections as a defect detection technique in detail. It tends to be rather wordy at times but the case studies are particularly useful for illustrating the practice of inspection. (T. Gilb and D. Graham, 1993, Addison-Wesley.)

'Using Inspections to Investigate Program Correctness'. This article discusses how a mathematically based but not completely formal inspection process is effective in discovering program errors. (R. N. Britcher, *IEEE Computer*, 21(11), November 1988.)

EXERCISES

19.1 Discuss the differences between verification and validation and explain why validation is a particularly difficult process.

19.2 Explain why it is not necessary for a program to be completely free of defects before it is delivered to its customers. To what extent can testing be used to validate that the program is fit for its purpose?

19.3 Explain why program inspections are an effective technique for discovering errors in a program. What types of error are unlikely to be discovered through inspections?

19.4 Explain why an organisation with a competitive, elitist culture would probably find it difficult to introduce program inspections as a V & V technique.

19.5 Using your knowledge of Java, C++, C or some other programming language, derive a checklist of common errors (not syntax errors) which could not be detected by a compiler but which might be detected in a program inspection.

19.6 Produce a list of conditions which could be detected by a static analyser for Java, Ada or C++. Comment on this list compared to the list given in Figure 19.7.

19.7 Read the published papers on Cleanroom development and write a management report highlighting the advantages, costs and risks of adopting this approach to software development.

19.8 A manager decides to use the reports of program inspections as an input to the staff appraisal process. These reports show who made and who discovered program errors. Is this ethical managerial behaviour? Would it be ethical if the staff were informed in advance that this would happen? What difference might it make to the inspection process?

19.9 One approach which is commonly adopted to system testing is to test the system until the testing budget is exhausted and then deliver the system to customers. Discuss the ethics of this approach.