

- risk avoidance
- risk minimization
- contingency planning.

The **risk avoidance** strategy attempts to eliminate the likelihood/probability of the risk occurring. For example, if there is a risk of losing a key team member due to his/her unhappiness about flexible working hours, an offer to work from home on some days can be made to that person.

The **risk minimization** strategy attempts to diminish the impact of the risk when it occurs. With relation to the previous example, to minimize the impact of the staff loss, another team member may be trained to match the knowledge and skills of the team member who may quit the job.

The **contingency planning** strategy accepts that risk may become a reality and there is a need to know how to react when this happens. A contingency plan defines the sequence of actions to be taken when other strategies have failed and the risk has become a fact. For example, the key team member is leaving with two weeks' notice and there is no immediate replacement available. The contingency plan may then specify that a replacement employee/contractor should be hired, the budget and schedule should be adjusted accordingly, and the departing employee should spend the remaining two weeks documenting the status of his/her work.

## 5.3 Quality Management

Software **quality management** is an umbrella activity that intertwines with most other management undertakings and cuts across the entire development lifecycle and process. It intertwines and cuts across, but it is an **independent activity**. A quality management team should be separate from the development team and have its own reporting channels. Quality management should be independent from project management. The quality management plan should have its own budget and schedule, at least as far as the quality assurance is concerned (Section 5.3.3).

Quality management has been the subject of intense standardization efforts. The Capability Maturity Model (CMM), discussed at the beginning of this chapter, is a de facto standard for a quality process. The International Organization for Standardization adopted the ISO 9000 series of quality standards. The standards apply to any industry and all types of activity domains, including software engineering. An ancillary document, known as ISO 9000-3, elucidates ISO 9000 for software engineering.

More recently, the key standards within the ISO 9000 series of standards have been merged into a single standard, known as ISO 9001:2000 (ISO, 2003). The emphasis of the standard is on managing key processes to continually improve them (this is analogous to CMM Level 5). As in the case of CMM certification, ISO provides an opportunity for organizations to register their conformance to ISO-defined processes.

Quality management standards are predominantly concerned with defining **quality processes** which would ensure quality in the products. This is different to quality

mechanisms concerned with controlling quality in the products. Accordingly, *quality assurance* (Section 5.3.3) is distinguished from *quality control* (Section 5.3.2).

## Software Qualities

### 5.3.1

Different users (and different developers) may have different opinions about what constitutes **software quality**. There are many desired qualities and, depending on a software project, some of them may be more important than others. Software qualities need to be identified, defined, and classified.

The main classification divides software qualities into **product** and **process qualities**. Usually it is not possible to produce a quality product without having a quality process that defines production activities. The purpose of the process is a product. Consequently, although qualities can be attributed to the process per se, all process qualities can be directly linked to product quality. The reverse is also true.

From the viewpoint of quality management, product quality must be assured and controlled not only on the **end product** delivered to the customer, but also (first of all) on all intermediate **work products** (known collectively as **artifacts**). Managing quality in work products demands support of *configuration management tools* able to store and manipulate multiple artifacts, their versions and relationships between them.

Ghezzi *et al.* (2003) provide an excellent account of representative qualities of software products and processes. The representative qualities are:

- correctness
- reliability
- robustness
- performance
- usability
- understandability
- maintainability (repair ability)
- scalability (evolvability)
- reusability
- portability
- interoperability
- productivity
- timeliness
- visibility.

**Correctness** is a set of formal properties of software that institutes one-to-one correspondence between the software product and its functional specifications. There are two problems with the correctness quality. Firstly, there is a strong argument in favor of the well-known observation that it is not possible to prove a program correct because it is not possible to guarantee the absence of errors in such complex products as software. Secondly, functional specifications are rarely defined with enough precision and stability

to serve as an exhaustive reference point in deciding the correctness properties. It would be, therefore, possible to have correct software for incorrect requirements.

**Reliability** (also called **dependability**) is a property of software that makes the software trusted and believed in by the users because it behaves well and in the way users expect. Ideally, the notion of software reliability should include the notion of correctness. While this is true in traditional engineering disciplines, in software engineering a reliable software is allowed to contain 'known bugs'. Even if new errors are encountered, the software may be still considered reliable (enough).

**Robustness** implies that the software is unlikely to fail or break, or at least fail or break in an irrecoverable way. Robustness complements the notions of correctness and reliability. Jointly, these three qualities define whether the software performs its designated functions as expected. Moreover, all three properties can refer to product as well as process qualities.

**Performance** means to operate satisfactorily, according to predefined targets. The usual target is response time of the software system, which specifies how long the user is prepared to wait for the system to respond to the user's request. Unlike many other qualities, performance is a black-and-white quality: either the system meets the performance indicator or it does not.

**Usability** relates to the user friendliness of the software; how easy it is for the users to make the software work for them. This is a very subjective property. What is friendly and easy for one person may be unfriendly and difficult for another. The usability quality is predominantly a feature of the design of the user interface for the system. As in traditional engineering disciplines, the more standard and typical the user interface, the more usable it is. Innovative designs of user interfaces may look nice but they are not terribly usable, especially for novice but computer-literate users.

**Understandability** refers to the ease with which the meaning of the internal structure and behavior of the software can be analyzed and comprehended by a software maintainer. Understandability is a condition for maintainability.

**Maintainability** of software is unlike maintainability of other engineering products. Software parts do not break due to prolonged use, power surges, or similar causes. In software engineering, maintainability is an all-encompassing concept meaning any of the three tasks: (1) correcting software errors and deficiencies, (2) adapting the software to new requirements, and (3) perfecting the software to give it new qualities. In the narrow meaning of the word, accepted here, maintainability is restricted to the first point. This is called **repairability** by Ghezzi *et al.* (2003).

**Scalability**, or **evolvability**, is the ease with which the software can be scaled up or evolved in response to the growth in demand for its functionality. Scalability can only be achieved in systems with clear, understandable architectural design. Once the architecture of the system deteriorates due to maintenance (repairs), the scalability diminishes. Understandability, maintainability, and scalability are known under the combined name of **supportability**.

**Reusability** of software defines the level to which the software components can be used for construction of other products. Software can be reused in two ways. It can be used as a **component** of another product, or it can be used as a generic **framework** that needs to be customized to create another product. Reusability applies also to software processes.

**Portability** means that the software can run on various hardware/software platforms without any modifications or after undergoing minor customization or parameterization. Portability has an economic significance. This quality is essential in system software, less so in application software.

**Interoperability** defines the ability of software to coexist or work together with other software, possibly even with the future software that does not yet exist. Like portability, interoperability is crucial in system software. The interoperability quality underpins software known as open systems.

**Productivity** is a process quality factor. Productivity defines the rate at which the process allows software to be produced, given a certain amount of resources. Productivity is the measure of the process efficiency and performance.

**Timeliness** is another process quality factor, which defines the ability of the process to produce software on time. On time, usually means according to the baseline schedule (a revised schedule, according to which the product has been delivered, is unlikely to be timely). In case of commercial products, timeliness may mean on-time-to-market.

Finally, **visibility** is also a process quality factor. A visible process is a transparent process – the process with clearly defined and documented stages and activities. A visible process is a condition for good project and risk management. A visible process is also a condition for an organization to obtain a CMM certification or ISO registration.

## Quality Control

### 5.3.2

**Quality control** is mostly about testing the quality of a product, as opposed to **quality assurance** that is about building quality into the product. Quality control has a reactive flavor. Quality assurance is very much a proactive undertaking. Quality control is mainly an operational and tactical effort. Quality assurance has a strong strategic aspect. Sometimes, quality control is considered a part of quality assurance.

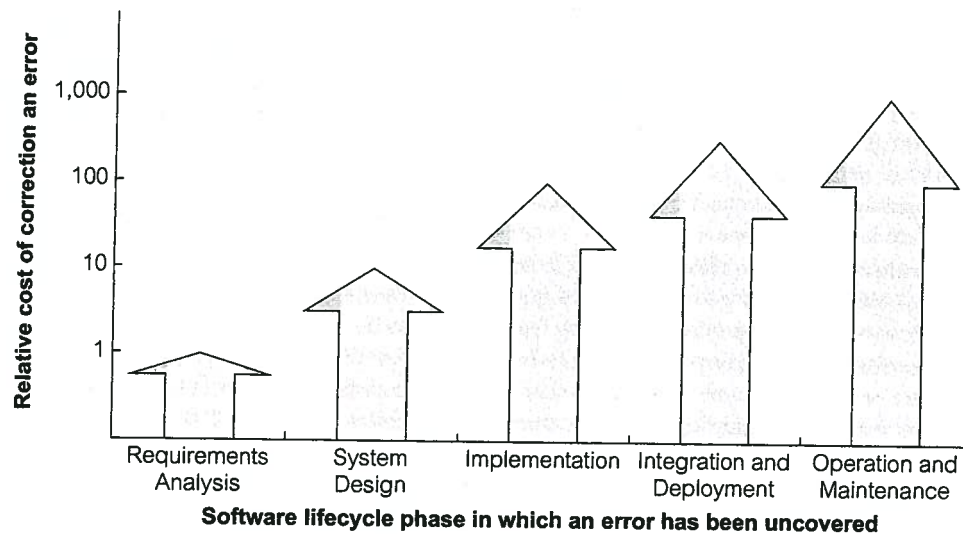
Pressman (2001) equates quality control with **variation control**. Variation control is the concept used in manufacturing where multiple copies of the same product are produced and they should be identical. A copy, which shows variations compared with a model (template) copy, fails the quality test.

In software engineering, multiple copies of software product are not a concern. The majority of application software products exist in a single copy. When multiple copies are produced, such as with commercial system software, the duplication is a simple matter of producing compact disks or other storage media. Frequently, commercial software distribution does not even produce multiple copies – the software can be downloaded from the website of the software vendor.

## Software testing

However, there is a parallel between software quality control and variation control. Quality control is about finding variations in a work or end product compared with a given specification. The specification can demand a fulfillment of a functional requirement and/or a satisfaction of the qualities listed in Section 5.3.1. The aim of software quality

**Figure 5.6**  
Relative cost of  
correcting an error



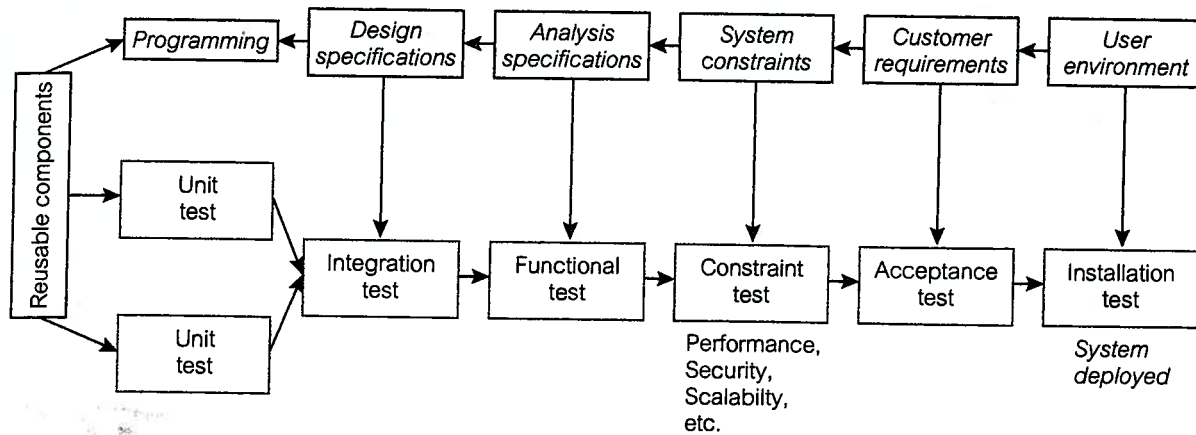
control is to subject the software to frequent tests in order to eliminate problems as early as possible. In many ways, software quality control is synonymous with software testing.

As explained in Section 1.2 and elsewhere in the book, testing is not a separate phase of the development lifecycle. Like quality management, testing is an umbrella activity that applies to all lifecycle phases. Naturally, testing is most intense when the code is available. Paradoxically, however, testing of code is not necessarily the most valuable kind of testing in terms of the project schedule and budget.

As reported in old studies by IBM, the relative cost of correcting an error grows almost exponentially depending on the lifecycle phase in which the error has been detected. The IBM research has established a belief that the cost of correcting an error may grow by a factor as high as 10s between lifecycle phases (Figure 5.6). Hence, assuming that correcting an error found during requirements analysis costs \$1 to correct, correcting the same error would cost as much as \$10 to correct if it was uncovered during system design, \$100 if uncovered during implementation, and as much as \$1,000 if detected after the system has been deployed to the users.

This universally accepted truth about the relative cost of correcting errors gives additional motivation and importance to quality control and testing activities. Testing takes different forms, depending on the development phase. During requirements analysis, testing attempts to establish the completeness of requirements, to eliminate inconsistencies and contradictions, to validate any software prototypes with the users, etc. During system design, testing cross-checks various design models, traces design artifacts to specifications, ensures that all requirements are addressed, etc. During implementation and later, testing of the code takes a leading position.

There is an interesting dependency between software development artifacts and testing of code. The dependency is that later stages of testing concentrate on validating earlier development artifacts. This observation is pictured in Figure 5.7 (Pfleeger, 1998). Hence,



**Figure 5.7** Dependencies between testing types and development phases

for example, integration tests are conducted against architectural and detailed design specifications. However, acceptance tests emphasize current customer requirements, which hopefully have not changed since they were documented for the analysis and design purpose. If they did change, acceptance tests may bring negative customer feedback and the issues raised need to be resolved between customers and project managers.

### Testing techniques

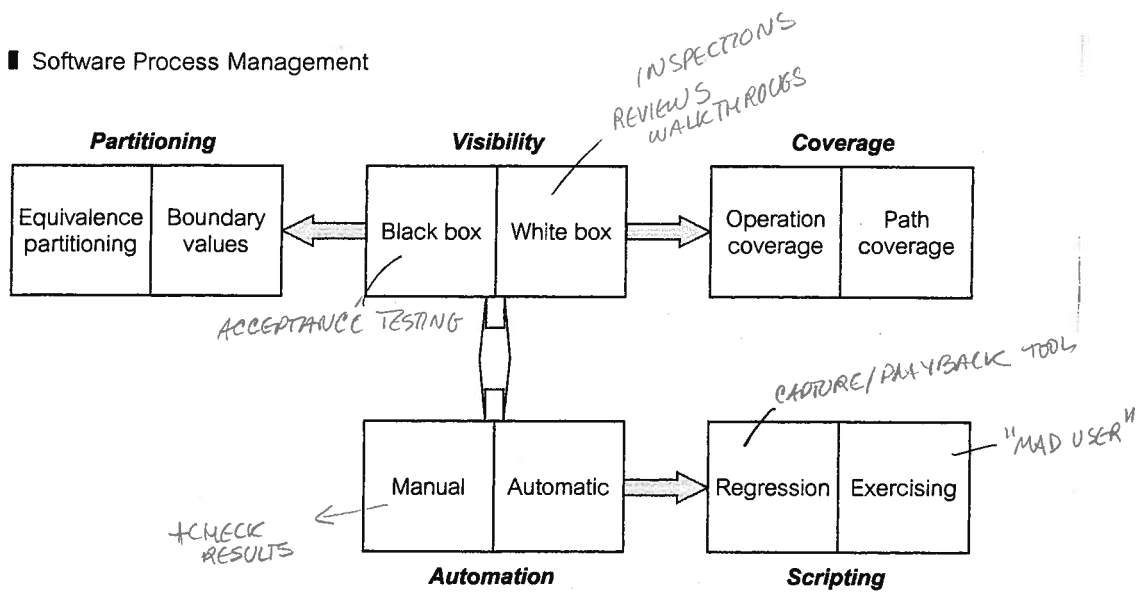
There are many possible **testing techniques**, which can be intermixed to provide the best possible testing coverage and outcomes. No matter how extensive the testing is, it cannot be exhaustive and guarantee program/system correctness. It is never possible to test for all possible data inputs or code execution paths.

Testing techniques (strategies) can be classified according to various criteria (Unhelkar, 2003). The following discussion is based on five criteria: (1) **visibility**, (2) **automation**, (3) **partitioning**, (4) **coverage**, and (5) **scripting** (Figure 5.8). The criteria are not independent. For example, the two partitioning techniques apply mostly to black box testing, but not really to white box. The scripting techniques (regression tests and exercising tests) are automatic tests.

**Black box testing** (or **testing to specs**) assumes that the tester does not know or chooses to ignore the internal workings of the program unit under test. The test unit is considered a black box, which takes some input and produces some output. The testing is done by feeding the test unit with data inputs and verifying that the expected output is produced. Because no implementation knowledge is required, black box testing can be conducted by the users. Accordingly, this is the main technique of **acceptance testing**.

The black box technique tests system functionality, or the system functional outputs, to be precise. It is also applicable for constraint testing, such as system performance or security. A special target of black box testing is **missing functionality**, i.e. when an expected functional requirement has not been implemented in the software. In such a case,

**Figure 5.8**  
Testing techniques



the test unit gets data to trigger such functionality and fails because the code to do the job is not there.

**White box testing** (or testing to code) is the opposite of black box testing. The test unit is considered a white box or, better, a transparent **glass box** that reveals its content. In this approach, the tester studies the code and decides on data inputs that are able to 'exercise' selected execution paths in the code. Black box testing must normally be followed by white box testing to pinpoint the errors discovered when testing to specs.

Unlike black box testing, white box testing is not restricted to code testing. It is equally suitable for other development artifacts, such as design models and specification documents. Testing of this kind uses review techniques, such as walkthroughs and inspections (Section 5.3.3). An interesting discovery possible with white box testing, but not with black box, is the existence of 'dead code'. The **dead code** are statements in the program that cannot be exercised, i.e. they will never be used no matter what input is given to the code.

The equivalence partitioning and boundary value techniques apply mostly to black box testing, although white box testing can also benefit from them. They are specific approaches to the black box technique to counteract the impossibility of conducting exhaustive tests.

**Equivalence partitioning** groups data inputs (and, implicitly, data outputs) into partitions constituting homogeneous test targets. The assumption is that testing with any one member of the partition is as good as testing with other members. Therefore, testing with the other members can be forfeited. Choosing the homogeneous partitions is the main difficulty. Partitions must be chosen with a good knowledge of data and the application demands for data. In this sense, partitioning per se is not really black box (it is not even a testing technique as such). Black box is the testing technique that equivalence partitioning supports.

**Boundary value** is merely an additional data analysis technique to assist in equivalence partitioning and, consequently, to assist in black box testing. Boundary values are extreme cases within equivalence partitions. For example, if the partition is a set of integer values from 1 to 100, the boundary value analysis will recommend tests to be done on the values

on the edges of the partition, i.e. for -1, 0, +1 as well as for 99, 100, and 101. Naturally, the expected outcome for testing on -1 and 101 would be an error condition.

Coverage techniques determine how much code is going to be exercised by a white box test. **Operation coverage**, called also **method coverage**, ensures that each operation in the code is exercised at least once by the white box test. Operation coverage is a modern object-oriented substitute for statement and branch coverage, which apply in procedural programming languages.

**Path coverage** aims at numbering possible execution paths in the program and exercising them one by one. Clearly, the number of such paths is indefinite in large programs. The tester is only able to define the most critical and most frequently used paths, and only these paths will be exercised.

Testing can be manual or automatic (Unhelkar, 2003). A human tester who interacts with the application under test conducts manual testing. The tester launches the application, systematically executes its functions and observes the results. The execution steps are conducted according to a predefined test script. The test script defines step-by-step testing actions and expected outcomes. Use cases and other requirements specification documents are used to write test scripts.

The main problem with manual tests is that, in most practical situations, they are performed on 'live data'. There is no fixed baseline data predefined to guarantee the same output for repeating execution of manual tests. For this reason, expected output is not always defined precisely in the test scripts. Frequently, the output is not even presented to the screen, but it is manifested in database changes. This forces the tester to write and execute SQL scripts or other *test harnesses* to check the results of test actions.

Manual tests are expensive to prepare, execute, and manage. They are unable to satisfy the 'volume' demands of testing. Automated testing employs software testing tools to execute large volumes of tests without human participation. The tools are also able to produce necessary post-test reports to facilitate management of test outcomes. Naturally the preparatory tasks, such as deciding what to test, programming some test scripts, and setting up the testing environment, must still be done by human experts.

As indicated in Figure 5.8, automated testing can be divided into regression testing and exercising testing. **Regression testing** is a popular term to mean repetitive execution of the same test scripts on the same baseline data to verify that previously accepted functionality of the system has not been broken by successive changes to the code, i.e. changes seemingly unrelated to the tested functionality.

Regression testing is performed by automatic execution of pre-recorded test scripts at scheduled test times. Original test scripts are recorded by a **capture/playback tool** in the process of capturing the human tester's actions on the application under test. Regression testing is a playback activity of playing back the scripts. In many cases, the tool-captured scripts are modified (improved) by a test programmer to allow regression testing of features, which the tool could not record automatically.

**Exercising**, for the lack of a better term, is really an automated coverage testing. A tool that implements exercising testing generates automatically and randomly various possible actions that otherwise could be performed by the user on the application under test. This could be compared to a mad user hitting any possible key on the keyboard, selecting any possible menu item, pressing any available command button, etc.

Generated actions are recorded (captured) in a script, sometimes called the **best script**, which is deemed to be able to exercise the application. Any errors or application failures are recorded as well and a separate script is generated for actions that led to the problem. This is called a **defect script**. The defect script can be played back at any time to reproduce the error and try to determine the reason behind it.

Because both regression and exercising tests use capture/playback tools, which result in programs in some scripting language, they can feed off each other, thus creating a powerful automated testing environment. In practice, the difficulty with automated testing is not in conducting it, but in running consecutive tests on a stable testing environment, with an identical testbed database, with the same state of open and active applications on the test workstation, with predictable network speed, with network broadcasting to the workstation disabled, with the same desktop appearance (including video resolution, desktop colors, fonts), etc. Finally, all automatic tests should clean up after themselves and re-create the database and application testbed as it was before the tests started.

### Test planning

A **test plan** is a part of the quality management plan. The test plan defines the testing schedule, budget, tasks (test cases) and resources. The plan must not be restricted to code testing. It should include testing of other project artifacts. It should also link to change management tasks responsible for handling defects.

Part of test planning is the identification of the **test environment**, as separate from the development environment. The plan should specify which test tasks should be conducted in a test environment and which in a development or production environment. A test environment requires allocation of human and material resources. Test workstations must be specified, the test database created, and the test software tools installed.

Figure 5.9 is a class model of testing concepts relevant to test planning. The test plan is developed around **test cases** (or tasks in a traditional planning sense). Requirements constitute test input to test cases. A requirement can be a use case requirement or test requirement. Ideally, test requirements are derived from and correspond to use case requirements. Therefore, test cases should contain statements about test requirements.

Each test case defines the hardware/software configurations for conducting the test as well as iterations recommending when the tests should be executed. A configured test case can be instantiated for execution as an automated test.

Test cases are realized in **test scripts**. Each test script consists of detailed listing of test steps and **verifications points** (which are the questions checking any significant outcomes of test steps). Test scripts can be combined in **test suites**. Test suites can contain other test suites. Automated testing is capable of performing test suites as single entities.

A test script can be destined for an automated or manual test. A human tester performs a manual test. A virtual tester, which is a workstation that can launch the application under test and execute a test program on that application, performs an automated test. Automated tests can be run on a local computer (within the **development environment**) or on an agent computer (within the **test environment**). Running the test on an agent computer does not preclude viewing and recording the test results on the local computer.

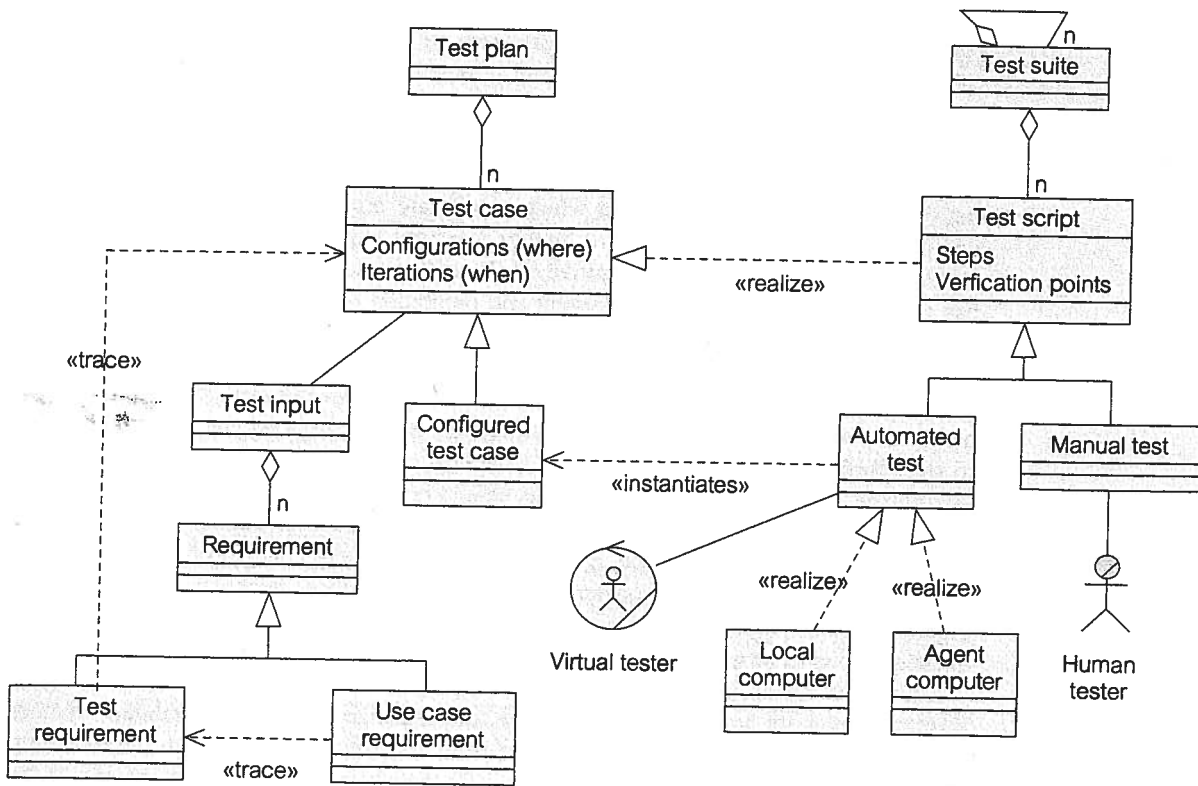


Figure 5.9 Testing concepts

## Quality Assurance

### 5.3.3

Quality control and quality assurance are two sides of the same coin – quality management. Both include a feedback loop to the development process. Apart from the already mentioned differences between quality control and quality assurance, perhaps the most distinguishing difference is that **quality assurance** is a management-level function (even though some quality assurance activities are conducted by the developers alone).

The management-designated **software quality assurance (SQA)** group conducts quality assurance. The group is independent from developers and reports to functional management (this could be at operational, tactical or even strategic level, depending on the significance of the project). Being a management function, quality assurance monitors not just development product and processes, but also project plans – schedule, budget, allocation and utilization of resources.

Quality assurance verifies the compliance of software products and processes with the adopted standards. As the name suggests, quality assurance is a reassuring activity – to reassure all project stakeholders (owners, customers and developers) that the system quality is high.

From a range of quality assurance undertakings, three deserve closer look. These are: (1) checklists, (2) reviews, and (3) audits.

### Checklists

In everyday life, a checklist is a list of things to do. In quality assurance, a **checklist** is a list of issues and questions against which a software product or process is verified for quality properties. It is a baseline of quality measures expected in software. Although a checklist is probably the most primitive of all quality assurance techniques, it is easy to use and can provide an early assessment and prediction of the quality of the product or process.

A checklist may be a simple list of relatively independent checkpoints or it can be a logical sequence of steps such that each step feeds off the previous step. In the former case, a checklist resembles a questionnaire. In the latter case, a checklist can contain branches to steer further checks to specific paths. Checklist items can be prioritized according to importance or to the time when the checks should be conducted.

Checklists can be used independently or as part of another quality assurance task, such as a review or audit. Indeed, a checklist guides the very conduct of a formal review (walkthrough or inspection). Standards organizations produce checklists, which software organizations use to check if their processes conform to standard requirements. Checklist questions need to be addressed to obtain a CMM certification or ISO registration.

Checklists are an important element of shaping the quality culture within an organization. They are a great educational device, in particular when used to validate whether software models, including programs, have been constructed according to the modeling principles and patterns. Making a mistake, and then being told about the nature of the mistake, is arguably the most effective learning technique. Checklists are about locating mistakes and informing about their nature.

The last point makes it clear that the quality of a checklist itself is a prerequisite for the successful use of the technique. The best people in the organization and people who have experience in software development must create checklists. Bad checklists will have a negative and long-lasting impact on the quality culture.

### Reviews

A **review** in quality assurance refers to a formal meeting of developers, and possibly managers, to review a work product or process. The notion encompasses a range of pre-organized technical meetings, of which the best known are **walkthroughs** and **inspections**.

Reviews are document driven. A document is prepared in advance and made available to review participants prior to the meeting. Participants are expected to study the available material in preparation for the meeting. The review is restricted to a small number of people: a *review leader* (and meeting moderator), the developer whose work is reviewed (a *producer*), and a couple of fellow developers.

During the meeting, the producer explains ('walks through') the piece of work under review and allows the reviewers to raise issues. The meeting pinpoints and documents the problems, but does not attempt to solve them. Acknowledged problems are recorded in a

**review issues list** (Pressman, 2001), which is handed to the producer after the meeting. The list guides the producer in making corrections to the work product or process. Once the corrections are made, the review leader is informed about it and evaluates the work and corrections made. If necessary, a follow-up meeting is called.

Special features of a formal technical review are as follows:

- It can be used for any work product and process that is contained in any kind of document (this can be a UML model, a test plan, user manual, a piece of code, a process definition, etc.).
- Complete materials for the meeting are distributed and studied in advance (the preparation should not demand more than a couple of hours of work by each reviewer).
- The membership of the meeting is clearly defined and normally restricted to up to five people.
- The duration of the meeting is short (two hours or less).
- The tasks allocated to the participants are clearly specified:
  - a review leader evaluates the material for readiness for the meeting, distributes documents to participants, organizes and moderates the meeting
  - a producer 'walks through' the document and answers questions
  - all reviewers, including the review leader, raise issues based on their advance studies (one of the reviewers takes notes and produces a review issues list; this person is called a recorder or secretary).
- The meeting pinpoints and documents the problems in a review issues list, but does not attempt to solve them (the review issues list is then filed as part of the formal project documentation).
- All care must be taken to ensure that the meeting reviews the work product or process, not the producer (and no personal appraisal consequences result from the meeting).
- The number of meetings is not restricted but any meeting is set at the discretion of a review leader or project leader.

Walkthroughs and inspections are similar. The main difference is that inspections are carried out under close management supervision, are less frequent, and address larger and less technical problems. There is lots of evidence that formal reviews work well provided they are done well. They contribute to overall quality culture, increased productivity, meeting deadlines, project awareness, development of professionalism, etc.

## Audits

Audits have a much more comprehensive charter than checklists or reviews. An **audit** is an IS/IT quality assurance process similar in scope, required resources, and level of assumed expertise to traditional accounting audits. It is the most formal of all quality assurance techniques. An audit starts with extensive preparations and studies of the audited system and continues with interviews and inspections. Obtained information is validated as much as possible, and conclusions are reached. The conclusions about the status of the process or product are documented in a formal report, which also includes a risk assessment for the continuation of the project.

Unhelkar (2003) points out the following differences between an audit and other quality assurance techniques:

- The producer of the audited product or process is usually a team, not a person.
- An audit can be performed without the presence of the producer.
- Audits make extensive use of checklists and interview and less of reviews.
- Audits can be external, i.e. conducted by auditors external to the organization.
- An audit can last from a day to a week or more (but is always restricted to strictly defined scope and objectives).

The scope of an audit is normally much wider than that of other quality assurance undertakings. Auditing recognizes the strategic importance of IT to the business and focuses on IT governance and business reengineering issues. The audited project is analyzed from the viewpoint of overall IT investments and its alignment with the business mission.

Auditing involves the whole range of organizational plans and finances, not just the project plan. It considers legal implications and project management issues. Typical objectives are the detection and prevention of frauds and security breaches as well as planning for contingencies and disaster recovery. A particular emphasis is on evaluating the efficiency and effectiveness of the use of all project resources.

There is a close dependency between the audited product and process and wider business policies and procedures. An audit of the project provides inputs to ensure consistency with business policies and procedures. It also recommends complementary policies and procedures applicable to the project.

## 5.4 Change and Configuration Management

The topic of change and configuration management has been introduced in Section 3.4 in the context of capabilities of tools supporting these activities. In many ways, the sophistication of change and configuration management is evidence of the maturity of the whole development process. As noted at the beginning of this chapter, an improvement in change and configuration management is a key factor in reaching the optimal level of maturity on the CMM scale (Figure 5.1).

**Change and configuration management** is the process of managing product and process artifacts and managing the teamwork activities in an evolving software system. The process relates to the entire lifecycle of software, from its inception to retirement. Organizations must plan for change. Change and configuration management heralds and strengthens all other management processes, including quality management.

Figure 5.10 shows the main concepts and dependencies in change and configuration management. Change requests can result in changes to use cases and use case requirements. Some changes become enhancements (future features) and are not considered in the current development. There is a mapping between use case requirements and test requirements. Test requirements can be thought of as verification points in test scripts.

System modeling draws on use case requirements to create model artifacts, including the code. Artifacts are versioned, i.e. there may be multiple versions of the same artifact.