too frequently, serious defects are found in a product as the delivery deadline approaches. The software organization must now choose between two unsatisfactory options. Either the product can be released on time but full of faults, leaving the client to struggle with faulty software, or the developers can fix the software but deliver it late. No matter what, the client probably will lose confidence in the software organization. The decision to deliver faulty software on time should not be made by the manager responsible for development, nor should the SQA manager be able to make the decision to perform further testing and deliver the product late. Instead, both managers should report to a more senior manager who can decide which choice would be in the best interests of both the software development organization and the client.

At first sight, having a separate SQA group would appear to add considerably to the cost of software development, but this is not so. The additional cost is relatively small compared to the resulting benefit—higher-quality software. Without an SQA group, every member of the software development organization would have to be involved to some extent with quality assurance activities. Suppose an organization has 100 software professionals and each devotes about 30 percent of his or her time to quality assurance activities. Instead, the 100 individuals should be divided into two groups, with 70 individuals performing software development and the other 30 people responsible for SQA. The same amount of time is devoted to SQA, the only additional expense being a manager to lead the SQA group. Quality assurance now can be performed by an independent group of specialists, leading to products of higher quality than when SQA activities are performed throughout the organization.

In the case of a very small software company (four employees or fewer), it may simply not be economically viable to have a separate SQA group. The best that can be done under such circumstances is to ensure that the analysis artifacts are checked by someone other than the person responsible for producing those artifacts and similarly for the design artifacts, code artifacts, and so on. The reason for this is explained in Section 6.2.

## 6.2 Non-Execution-Based Testing

Testing software without running test cases is termed **non-execution-based testing**. Examples of non-execution-based testing methods include reviewing software (carefully reading through it) and analyzing software mathematically (Section 6.5).

It is not a good idea for the person responsible for drawing up a document to be the only one responsible for reviewing it. Almost everyone has blind spots that allow faults to creep into the document, and those same blind spots prevent the faults from being detected on review. Therefore, the review task must be assigned to someone other than the original author of the document. In addition, having only one reviewer may not be adequate; we all have had the experience of reading through a document many times while failing to detect a blatant spelling mistake that a second reader picks up almost immediately. This is one principle underlying review techniques like walkthroughs or inspections. In both types of review, a document (such as a specification document or design document) is painstakingly checked by a team of software professionals with a broad range of skills. The strength of a review by a team of experts is that the different skills of the participants increase the chances of finding a fault. In addition, a team of skilled individuals working together often generates a synergistic effect.

Walkthroughs and inspections are two types of reviews. The fundamental difference between them is that walkthroughs have fewer steps and are less formal than inspections.

### 6.2.1 Walkthroughs

A walkthrough team should consist of four to six individuals. An analysis walkthrough team should include at least one representative from the team responsible for drawing up the specifications, the manager responsible for the analysis workflow, a client representative, a representative of the team that will perform the next workflow of the development (in this instance the design team), and a representative of the software quality assurance group. For reasons that will be explained in Section 6.2.2, the SQA group member should chair the walkthrough.

The members of the walkthrough team should, as far as possible, be experienced senior technical staff members because they tend to find the important faults. That is, they detect the faults that would have a major negative impact on the project [R. New, personal communication, 1992].

The material for the walkthrough must be distributed to the participants well in advance to allow for thorough preparation. Each reviewer should study the material and develop two lists: a list of items the reviewer does not understand and a list of items the reviewer believes are incorrect.

### 6.2.2 Managing Walkthroughs

The walkthrough should be chaired by the SQA representative because the SQA representative has the most to lose if the walkthrough is performed poorly and faults slip through. In contrast, the representative responsible for the analysis workflow may be eager to have the specification document approved as quickly as possible to start some other task. The client representative may decide that any faults not detected at the review probably will show up during acceptance testing and be fixed at that time at no cost to the client organization. But the SQA representative has the most at stake: The quality of the product is a direct reflection of the professional competence of the SQA group.

The person leading the walkthrough guides the other members of the walkthrough team through the document to uncover any faults. It is not the task of the team to correct faults, but merely to record them for later correction. There are four reasons for this:

1. A correction produced by a committee (that is, the walkthrough team) within the time constraints of the walkthrough is likely to be lower in quality than a correction produced by an individual trained in the necessary techniques.
2. A correction produced by a walkthrough team of five individuals takes at least as much time as a correction produced by one person and, therefore, costs five times as much when the salaries of the five participants are considered.
3. Not all items flagged as faults actually are incorrect. In accordance with the dictum, “If it ain’t broke, don’t fix it,” it is better for faults to be analyzed methodically and corrected only if there really is a problem, rather than have a team attempt to “fix” something that is completely correct.
4. There simply is not enough time in a walkthrough to both detect and correct faults. No walkthrough should last longer than 2 hours. The time should be spent detecting and recording faults, not correcting them.

There are two ways of conducting a walkthrough. The first is participant driven. Participants present their lists of unclear items and items they think are incorrect. The representative of the analysis team must respond to each query, clarifying what is unclear to the reviewer and either agreeing that indeed there is a fault or explaining why the reviewer is mistaken.

The second way of conducting a review is document driven. A person responsible for the document, either individually or as part of a team, walks the participants through that document, with the reviewers interrupting either with their prepared comments or comments triggered by the presentation. This second approach is likely to be more thorough. In addition, it generally leads to the detection of more faults because the majority of faults at a document-driven walkthrough are spontaneously detected by the presenter. Time after time, the presenter will pause in the middle of a sentence, his or her face will light up, and a fault, one that has lain dormant through many readings of the document, suddenly becomes obvious. A fruitful field for research by a psychologist would be to determine why verbalization so often leads to fault detection during walkthroughs of all kinds, including requirements walkthroughs, analysis walkthroughs, design walkthroughs, plan walkthroughs, and code walkthroughs. Not surprisingly, the more thorough document-driven review is the technique prescribed in the IEEE Standard for Software Reviews [IEEE 1028, 1997].

The primary role of the walkthrough leader is to elicit questions and facilitate discussion. A walkthrough is an interactive process; it is not supposed to be one-sided instruction by the presenter. It also is essential that the walkthrough not be used as a means of evaluating the participants. If that happens, the walkthrough degenerates into a point-scoring session and does not detect faults, no matter how well the session leader tries to run it. It has been suggested that the manager who is responsible for the document being reviewed should be a member of the walkthrough team. If this manager also is responsible for the annual evaluations of the members of the walkthrough team (and particularly of the presenter), the fault detection capabilities of the team will be compromised, because the primary motive of the presenter will be to minimize the number of faults that show up. To prevent this conflict of interests, the person responsible for a given workflow should not also be directly responsible for evaluating any member of the walkthrough team for that workflow.

## 6.2.3 Inspections

Inspections were first proposed by Fagan [1976] for testing designs and code. An **inspection** goes far beyond a walkthrough and has five formal steps.

1. An **overview** of the document to be inspected (requirements, specification, design, code, or plan) is given by one of the individuals responsible for producing that document. At the end of the overview session, the document is distributed to the participants.
2. In the **preparation**, the participants try to understand the document in detail. Lists of fault types found in recent inspections, with the fault types ranked by frequency, are excellent aids. These lists help team members concentrate on the areas where the most faults have occurred.
3. To begin the inspection, one participant walks through the document with the inspection team, ensuring that every item is covered and that every branch is taken at least once. Then fault finding commences. As with walkthroughs, the purpose is to find

and document the faults, not to correct them. Within one day the leader of the inspection team (the **moderator**) must produce a written report of the inspection to ensure meticulous follow-through.

4. In the **rework**, the individual responsible for the document resolves all faults and problems noted in the written report.
5. In the **follow-up**, the moderator must ensure that every issue raised has been resolved satisfactorily, by either fixing the document or clarifying items incorrectly flagged as faults. All fixes must be checked to ensure that no new faults have been introduced [Fagan, 1986]. If more than 5 percent of the material inspected has been reworked, then the team must reconvene for a 100 percent reinspection.

The inspection should be conducted by a team of four. For example, in the case of a design inspection, the team consists of a moderator, designer, implementer, and tester. The moderator is both manager and leader of the inspection team. There must be a representative of the team responsible for the current workflow as well as a representative of the team responsible for the next workflow. The designer is a member of the team that produced the design, whereas the implementer is responsible, either individually or as part of a team, for translating the design into code. Fagan suggests that the tester be any programmer responsible for setting up test cases; it is, of course, preferable that the tester be a member of the SQA group. The IEEE standard recommends a team of between three and six participants [IEEE 1028, 1997]. Special roles are played by the moderator, the **reader** who leads the team through the design, and the **recorder** responsible for producing a written report of the detected faults.

An essential component of an inspection is the checklist of potential faults. For example, the checklist for a design inspection should include items such as these: Is each item of the specification document adequately and correctly addressed? For each interface, do the actual and formal arguments correspond? Have error-handling mechanisms been adequately identified? Is the design compatible with the hardware resources or does it require more hardware than actually is available? Is the design compatible with the software resources; for example, does the operating system stipulated in the analysis artifacts have the functionality required by the design?

An important component of the inspection procedure is the record of fault statistics. Faults must be recorded by severity (major or minor; an example of a major fault is one that causes premature termination or damages a database) and fault type. In the case of a design inspection, typical fault types include interface faults and logic faults. This information can be used in a number of useful ways:

- The number of faults in a given product can be compared with averages of faults detected at the same stage of development in comparable products, giving management an early warning that something is amiss and allowing timely corrective action to be taken.
- If inspecting two or three code artifacts results in the discovery of a disproportionate number of faults of a particular type, management can begin checking other code artifacts for faults of that type, and take corrective action if necessary.
- If the inspection of a particular code artifact reveals far more faults than were found in any other code artifact in the product, there is usually a strong case for redesigning that artifact from scratch and implementing the new design.

- Information regarding the number and types of faults detected at an inspection of a design artifact aids the team performing the code inspection of the implementation of that artifact at a later stage.

The first experiment of Fagan [1976] was performed on a systems product. One hundred person-hours were devoted to inspections, at a rate of two 2-hour inspections per day by a four-person team. Of all the faults found during the development of the product, 67 percent were located by inspections before unit testing was started. Furthermore, during the first 7 months after the product was installed, 38 percent fewer faults were detected in the inspected product than in a comparable product reviewed using informal walkthroughs.

Fagan [1976] conducted another experiment on an application product and found that 82 percent of all detected faults were discovered during design and code inspections. A useful side effect of the inspections was that programmer productivity rose because less time had to be spent on unit testing. Using an automated estimating model, Fagan determined that, as a result of the inspection process, the savings on programmer resources were 25 percent despite the time that had to be devoted to the inspections. In a different experiment Jones [1978] found that over 70 percent of detected faults could be detected by conducting design and code inspections.

Subsequent studies have produced equally impressive results. In a 6000-line business data-processing application, 93 percent of all detected faults were found during inspections [Fagan, 1986]. As reported in [Ackerman, Buchwald, and Lewski, 1989], the use of inspections rather than testing during the development of an operating system decreased the cost of detecting a fault by 85 percent; in a switching system product, the decrease was 90 percent [Fowler, 1986]. At the Jet Propulsion Laboratory (JPL), on average, each 2-hour inspection exposed 4 major faults and 14 minor faults [Bush, 1990]. Translated into dollar terms, this meant a saving of approximately $25,000 *per inspection*. Another JPL study [Kelly, Sherif, and Hops, 1992] showed that the number of faults detected decreased exponentially by classical phase. In other words, with the aid of inspections, faults can be detected early in the software process. The importance of this early detection is reflected in Figure 1.6.

One advantage that code inspections have over running test cases (execution-based testing) is that the testers need not deal with failures. It frequently happens that, when a product under test is executed, it fails. The fault that caused the failure must now be located and fixed before execution-based testing can continue. In contrast, a fault found in the code during non-execution-based testing is logged and the review continues.

A risk of the inspection process is that, like the walkthrough, it might be used for performance appraisal. The danger is particularly acute in the case of inspections because of the detailed fault information available. Fagan dismisses this fear by stating that, over a period of 3 years, he knew of no IBM manager who used such information against a programmer, or as he put it, no manager tried to "kill the goose that lays the golden eggs" [Fagan, 1976]. However, if inspections are not conducted properly, they may not be as wildly successful as they have been at IBM. Unless top management is aware of the potential problem, misuse of inspection information is a distinct possibility.

### 6.2.4 Comparison of Inspections and Walkthroughs

Superficially, the difference between an inspection and a walkthrough is that the inspection team uses a checklist of queries to aid it in finding the faults. But the difference goes deeper than that. A walkthrough is a two-step process: preparation followed by team analysis of the document.

An inspection is a five-step process: overview, preparation, inspection, rework, and follow-up; and the procedure to be followed in each step is formalized. Examples of such formalization are the methodical categorization of faults and the use of that information in the inspection of the documents of the succeeding workflows as well as in inspections of future products.

The inspection process takes much longer than a walkthrough. Is inspection worth the additional time and effort? The data of Section 6.2.3 clearly indicate that inspections are a powerful, cost-effective tool to detect faults.

### 6.2.5 Strengths and Weaknesses of Reviews

There are two major strengths of a review (walkthrough or inspection). First, a review is an effective way to detect a fault; second, faults are detected early in the software process, that is, before they become expensive to fix. For example, design faults are detected before implementation commences, and coding faults are found before the artifact is integrated into the product.

However, the effectiveness of a review can be reduced if the software process is inadequate.

- First, large-scale software is extremely hard to review unless it consists of smaller, largely independent components. A strength of the object-oriented paradigm is that, if correctly carried out, the resulting product consists of largely independent pieces.
- Second, a design review team sometimes has to refer to the analysis artifacts; a code review team often needs access to the design documents. Unless the documentation of the previous workflows is complete, updated to reflect the current version of the project, and available online, the effectiveness of review teams is severely hampered.

### 6.2.6 Metrics for Inspections

To determine the effectiveness of inspections, a number of different metrics can be used. The first is the **inspection rate**. When specifications and designs are inspected, the number of pages inspected per hour can be measured; for code inspections, an appropriate metric is lines of code inspected per hour. A second metric is the **fault density**, measured in faults per page inspected or faults per 1000 lines of code (KLOC) inspected. This metric can be subdivided into major faults per unit of material and minor faults per unit of material. Another useful metric is the **fault detection rate**, that is, the number of major and minor faults detected per hour. A fourth metric is the **fault detection efficiency**, that is, the number of major and minor faults detected per person-hour.

Although the purpose of these metrics is to measure the effectiveness of the inspection process, the results instead may reflect deficiencies of the development team. For example, if the fault detection rate suddenly rises from 20 faults per thousand lines of code to 30, this does not necessarily mean that the inspection team has suddenly become 50 percent more efficient. Another explanation could be that the quality of code has decreased and there simply are more faults to be detected.

Having discussed non-execution-based testing, we now move on to execution-based testing.

## 6.3 Execution-Based Testing

It has been claimed that testing is a demonstration that faults (“bugs”) are not present. Even though some organizations spend up to 50 percent of their software budget on testing, delivered “tested” software is notoriously unreliable.