Fraunhofer Institut
Experimentelles
Software Engineering

Inspection and Testing

Towards combining both approaches

Author:
María Victoria Cengarle

Funded by the German BMBF
under grant VFG0004A ("QUASAR")

IESE-Report No. 024.02/E
Version 1.0
April 10, 2002

A publication by Fraunhofer IESE

Fraunhofer IESE is an institute of the Fraunhofer Gesellschaft.
The institute transfers innovative software development techniques, methods and tools into industrial practice, assists companies in building software competencies customized to their needs, and helps them to establish a competetive market position.

Fraunhofer IESE is directed by
Prof. Dr. Dieter Rombach
Sauerwiesen 6
D-67661 Kaiserslautern

Executive Summary

Typical quality assurance activities are inspection and testing. This report presents both of them, explores similarities and differences, and proposes a number of ways to combine them.

Keywords: quality assurance, defect, defect detection, inspection, testing

Table of Contents

1 Preamble

Nowadays it is no longer necessary to emphasize the importance and quasi omnipresence of software systems. Everyday life is unimaginable without some kind of computational-based device. We encounter not only what can be called information systems; CD players, watches, telecommunications, cars, and train control systems, to name a few, hide also another form of systems. The failure of any of these or those systems can have a variety of consequences. Independently of its consequences, a failure is unacceptable and the fault within the system that caused or can cause the failure has to be removed. The task of developing software systems, at the same time, has become complex enough to require the definition of precise guidelines for the construction and for the review of those systems as well as the participation of many people in their development. In other words, the development of software systems has to become an engineering.

Quality assurance is the activity that takes care of defects of a software. Quality assurance consists of two steps, defect detection and defect correction. Detection of defects of a software system, unlike the constructive tasks of its analysis, design and implementation, is an analytical activity. There are two main approaches to defect detection, inspection and testing. Inspection, in the literature also called review, may take place at any time, whereas testing requires the subject of interest to be executable (i.e., it has to be a prototype, a specification that can be animated, or the software system itself). Defect correction, in the literature also termed revision, is undertaken with the purpose of correcting a construction that deviates from the intention.

Ordinarily, inspection and testing are considered an "either-or" choice, meaning that one or the other approach is used but not both. Inspection and testing, however, should be considered complementary activities. Furthermore, a synergy effect could be generated by the combined application of both activities. In the present document this thesis is worked out. After a brief introduction to the terminology, the testing activity, which seems the older one, is presented. Afterwards the inspection activity is introduced. Finally the relation between both activities is explored. The Appendix A contains a small glossary of testing nomenclature, the Appendix B two examples of perspective-based reading.

In this article we abstract away from particular kinds of features or characteristics that have to be quality assured. It is well-known that different reading techniques and different testing approaches focus on or are more suitable for different kinds of requirements. This fact cannot be further disregarded when analyzing the instances of the proposed combinations of testing and inspection.

This work is part of the QUASAR project. It constitutes the first step of the work package "integration of testing and inspection". The aim of this work package is described in [PSBK+01]. This report sketches the fundamentals of inspection and testing and presents first findings on their similarities, differences and synergies.

Acknowledgements. The author is indebted to Barbara Paech, Hans-Gerhard Groß, Oliver Laitenberger, and Maud Schlich for reading and/or discussing previous versions of this work, in particular to Oliver, Gerd and Maud who kindly delivered the input for the appendices. Many thanks also to Nikolas Mayer for his technical support.

2 Introduction

In this section the subject is introduced as well as the terminology employed throughout the rest of the article.

Whenever a software system is conceived, an enlarged reality is envisioned, a reality augmented with a hypothetical *posit* or software artifact to be. This vision is then formalized and, ideally, successive formalization steps conduct the draft conception to a more or less formal design. A design is nothing else than a *theory*. The reification of the design is finally a posit or software artifact. Any of these posits delivers evidence or observed behavior, if it is hypothetical by producing instances of its hypothetical behavior, if it is a real software artifact by executing it. Besides posit and theory, and when the relationship between them is analyzed, a third actor is the property or *hypothesis* we want to confirm, which is derived from the theory. Confirmation in this setting means to contrast the actual/hypothetical behavior of the posit with its theoretical behavior as assumed by the hypothesis, where the "rest" of the theory[1] is taken for granted.

2.1 Validation and verification

If we are dealing with a hypothetical posit, confirmation undertakes the *validation* of the theory. That is, the question "are we constructing the correct thing?" is addressed. If we are dealing with an actual posit, the *verification* of the theory (which is assumed to correctly capture the desired behavior of the software artifact to be) is undertaken. That is, the question "are we constructing this thing correctly?" is addressed. If we arrive to a negative answer, in case of validation we have to revise the theory, in case of verification the realization.

Scientific theories try to explain natural phenomena. Whenever a theory has taken form in a human brain, the subsequent activity is to confirm or refute the correctness or accuracy of the theory. The activity of confirmation is also called testing of scientific theories; in the framework of testing of scientific theories, there is no (hypothetical) posit and thus it makes no sense to speak of validation, i.e., the theory just tries to explain facts and does not intend enlarge reality. The computer science community, however, can take advantage of centuries

1 Usually, the hypothesis is an axiom of the theory presentation, and the theory induced by the presentation without this hypothesis is called the *background theory*. In fact, the background theory must not entail the hypothesis.

of experience in testing scientific theories,[1] since the very same methods apply for defect detection (be it validation or verification).

2.2 Kinds of requirements

In the software engineering community, the word hypothesis has a slightly different meaning as the one above: it is also a property that can be subject to confirmation, yet not about a single system but about a process or modus operandi. When speaking of a software system, the denomination *requirement* is preferred. There are two major categories of requirements: functional and non-functional ones. Roughly speaking, functional requirements state what the system has to do, whereas non-functional ones prescribe properties of the system. Non-functional requirements are also called quality requirements.

A finer grained classification of requirements is given by the ISO/IEC 9126-1991 standard, which provides a framework for the evaluation of software quality. Six product quality characteristics are defined, namely functionality, reliability, usability, efficiency, maintainability, and portability. Within the scope of this article, the last five characteristics are referred to as non-functional requirements.

Functional requirements, in the simplest case, describe the input/output relation of a program; they constitute the functionality of the system. Functional requirements can be divided into safety and liveness properties; informally, safety means that "something bad will *not* happen" whereas liveness means that "something good *must* happen"; see [Kin94]. Use cases are an example of a concrete representation of functional requirements.

Non-functional properties of the system can be for instance robustness, security, performance, and user friendliness. They can include ease of maintenance, size and cost. A popular classification of some special kind of properties distinguishes between safety and security properties, meaning that the system will not harm the environment and that the environment will not harm the system, respectively; see [BKLW95].

1 Scientific theories can only be black-box tested (see Appendix A), since we cannot access the source code of the Universe!

3 Testing

The most popular approach to test a scientific theory is the hypothetico-deductive method. Developed by Sir Isaac Newton during the late 17th century (but named at a later date by philosophers of science), the hypothetico-deductive method assumes that properly formed theories arise as generalizations from observable data that they are intended to explain. Theories, through inference, predict further effects that can then be verified or disproved by empirical evidence derived from other experiments.[1]

The hypothetico-deductive method apparently has inherent flaws: [Gly80a] shows that it leads to a circular reasoning. There were attempts to save it (see for instance [Mer79]), also alternative approaches were proposed to test of scientific theories (see [Gly80b,Gly83]). The implication of these statements for the activity of testing software systems is unexplored up to now, neither improved versions of the hypothetico-deductive method nor alternative methods have been thoroughly investigated. These matters, however, are outside the scope of this article.

3.1 Overview

Within the context of software testing, the program (i.e., the actual posit) delivers evidence. The only difference between natural phenomena and software artifacts resides in the fact that, if the observation obtained through experimentation contradicts the prediction, then not the theory but the artifact producing evidence (i.e., the program) has to be revised.

In the computer science realm, real software artifacts (i.e., not artifacts to be) are tested. In the specific jargon, the theory is the *specification* or the *design*, the prediction is called *test case*, a discrepancy found between predicted behavior and observation is called *error* (or *failure*, if the software does not meet its performance requirements), and the program segment that led to the error is called *fault*.[2]

1 "hypothetico-deductive method", Encyclopædia Britannica Online. <http://members.eb.com/bol/topic?eu=42772&sctn=1>

2 In the literature there are many denominations for closely related concepts. The present work uses the definitions of the IEEE-Std-610.12-1990; see Appendix A.

The activity of software testing, once identified the unit under test (see Section 3.2), consists of at least the following activities:

1 test case generation (i.e., hypothesis selection and experiment design),

2 test case run (i.e., experiment conduction and observation), and

3 error/failure detection (i.e., comparison and decision).

In case of error, the program has to be worked out. Thereby, further activities to be undertaken (and that might be considered as part of the testing task) are:

4 fault isolation and

5 fault correction.

The first step, test case generation, consists in devising experiments. Functional requirements are tested by feeding the system with input and comparing expected and obtained output. The difficulty of the generation of test cases for functional requirements, and given that the theory (in our case, the specification) consists of generalizations, resides in the fact that it is usually impossible to cover all possible cases with a finite set of test cases; see [Don98].

The generation of test cases for non-functional requirements is also difficult. First, it might be required that the system performs within certain bounds, for instance referring to time responses ("the system takes a decision within 5 milliseconds") or resource allocation ("the network is accessed at most once per time unit"). Such kind of requirements depend on the context in which the system is inserted. These requirements are verified by the so-called workload testing, which in turn is subdivided into stability testing or test within expected workload levels over extended periods of time, and stress testing or test at peak workload levels; see [Tha99].

Second, non-functional requirements that also depend on the context are those that have to do with protocols as well as with the version of the underlying platform, compiler, library, and/or other software components. These kinds of requirements are corroborated by conformance testing and deployment testing; see e.g. [Tan98,Ros97].

Third, non-functional requirements might state which kind of behavior is expected in an infinite run, as for example fairness and deadlock/starvation prevention/avoidance (for instance, "if a process asks for a resource infinitely often, then it gets the resource infinitely often"). These properties are usually not tested but proved. A treatment of them can be found in the literature concerning operating systems; see e.g. [SGG01].

Techniques for the generation of test cases are sketched in Section 3.3.

The advantage of a computer scientist is that the second step of the testing activity, test case run, can be repeated at will, whereas for instance an astronomer, after revising his/her theory, normally has to devise a new experiment in order to test the theory again. Sometimes, however, the task of restoring the complete constellation in order to repeat an experiment might be non-trivial even if testing software. Test case run is further detailed in Section 3.4, after having presented the concept of unit under test.

The third step, error or failure detection, seems at first sight trivial: an error is detected if predicted and observed behavior differ, a failure is detected if the response is not obtained within the performance requirements of the system specification. The comparison, however, between expected and actual behavior can be far from trivial. The problem is to define a decision procedure for interpreting the results of tests according to a specification. Whether or not such a decision procedure can be defined is known as the *oracle problem*; see [Mac00]. In Section 3.5 we revisit this issue.

An important matter is to decide when to stop testing and how to evaluate the test results. The ultimate goal of testing, namely to prove correctness of a piece of code, cannot be reached: at most one can say that no error could be found. Test end criteria are substitutes for the real success criteria. Once the test end criterion has been reached, the test results have to be evaluated. Test metrics help not only in stating how many errors might still remain to be uncovered but also in measuring quantity and quality of the tests themselves; see [SW02].

In case of error, the fourth step, fault isolation, is more an art than a technique. The designer guesses which module contains the fault, the programmer which procedure. The module respectively the procedure has to be carefully read, and once the fragment was found that gives rise to the error, it has to be found out if the fault propagates. In case of failure, the system as a whole might need to be considered.

Once the fault was isolated, its removal respectively correction may be associated with more or less effort, depending on the presence or absence of fault propagation. The effort might range from correcting an index in an iteration (e.g., replace *1* by *0* in *for i=1 to n*) to the re-engineering of a whole subsystem in the worst case. These days a considerable amount of effort is put in order for this last case not to arise; at least with respect to functional requirements (i.e., in case of error), the trend is that quality assurance measures intervene in early stages of software development in such a way that major misunderstandings can early enough be avoided.

If we are dealing with a failure, the situation is even worse. Regarding performance requirements, it might be necessary to find a compromise between those

requirements and available (or affordable) technologies. Regarding safety issues, it might be unavoidable to consider weakening those requirements. As [Sto96] puts it:

> *It is tempting to say that human life is beyond price and that no expenditure is too great if it might save a single life. [However,] it is well known that we could make vehicles safer by reducing their speed. [...] Would we be prepared to accept increased journey times and greater congestion on our roads in order to save lives?*

3.2 Unit under test

The unit under test (UUT) is the subject of interest. When performing unit testing, the UUT is the smallest software unit. When performing deployment integration testing, the UUT is a whole (sub)system and/or its environment; see Appendix A. The decision of what is the UUT is necessary for the generation of suitable test cases.[1]

When speaking of testing, one usually thinks of software testing consisting in executing a program with particular input stimuli and observing the output generated by the program. There are, however, other scenarios where it might make sense to speak of testing. Techniques exist that allow the "animation" of specifications (written in particular formal specification languages); see [DLMT96]. Some CASE tools support validation by way of simulation (also called model-level debugging), for instrance SoftModeler [Sof02], ARTiSAN Real-time Studio [Art01], and Rhapsody [I-L]. In this way, testing can be performed at an earlier stage of system development and not necessarily only when the system has been completely implemented. That is, design can also be tested and one of the drawbacks of testing, namely that failures are detected at too late a point in time, is weakened. In this case, the UUT is the specification or design. This technique is directly connected to design inspection, see below.

In the literature it is sometimes called requirements testing what should be more properly called requirements-based testing. Requirements-based testing refers to the technique of test case generation driven by some requirements document. Requirements as such cannot be tested since they are expressed in an informal language (natural language), i.e., it is not possible to animate them automatically. The activity that resembles testing and is applied to requirements is given by the succession of meetings between analyst/designer and customer.

1 In this work, the concept of UUT also comprises the ones of system under test (abbreviated to SUT) and of application under test (abbreviated to AUT) that are sometimes separately considered in the literature.

The fine tuning of requirements that takes place in these meetings is not as systematic as the testing activity. This is explained by the fact that the quality assurance activities *compare* the so-far delivered product with the idealized artifact. At the requirements phase there is still no hypothetical posit, i.e., no definition of the artifact to be; see the definition of validation in Section 2. In other words, requirements testing would be better denominated inspection of requirements documents.

3.3 Test case generation

A test case consists of input values for the UUT, an expected output, and additional hypotheses or context information like for instance network load or database size (see Section 3.4) as well as any further information necessary for running the test case. The expected output refers to an *observable* behavior. The tester might have different observation tools at disposal; for instance, a database entry might be (but not necessarily is) observable. It is absolutely necessary that the test case also provides instructions on how to observe the actual output of the UUT after being fed with the input as described by the test case, and the observation mechanism must be a means for observing output at the same level of abstraction as the output predicted by the test case. Furthermore, from the very beginning it must be clear how to compare the expected output with the output actually obtained; see Section 3.5.

The definition of test cases is an utmost difficult task. It is dealt with by dividing it in *what* to test and, depending on what, *how* to test. These subdivisions give rise to different approaches. So for instance, and following the above mentioned classification, what to test can mean functional as opposed to non-functional requirements.

Functional requirements can be tested following the white-box or the black-box approach, meaning whether or not the test cases are generated using the source code. White-box testing is usually employed only for verification purposes. Within black-box testing of functional requirements we have two kinds of coverage, input domain coverage and specification coverage. On the one hand, to cover the input domain means to generate a test case for every possible input value of a domain that usually is infinite. On the other hand, to cover the specification means to generate test cases for every description of the specification, which (formal or not) usually consists of axiom schemata which abstract from infinitely many axioms, or of behavior descriptions which abstract from infinitely many execution sequences. This means, we have two dimensions of infinitude.

Uniformity and regularity hypotheses help reduce or subdivide the input domain; see [Gau95,GJ99]. Also methods that analyze equivalence classes and

determine boundary values are used in order to deal with infinite input; see e.g. [Bei90].

In order to cover a specification and depending on the kind of specification, use-case-based testing and state-based testing can be used; see [RG99,TR93]. Techniques for constraint reduction are also useful for the generation of test cases starting from a specification; see [CF00].

We also find a notion of coverage within white-box testing of functional requirements. Test cases can be generated that, according to a given criterion, cover the source code. The criterion establishes which parts of the code are to be covered and in which way. By coverage it is meant that there exists at least one test case such that, if the UUT is fed with the input as required by the test case, then the command or line of code to be covered is executed, and moreover it is executed in the way imposed by the criterion. Different kinds of coverage include [Mar95,Mye79]:

1 branch coverage, meaning that as many test cases as branches are generated, such that every branch of the program is traversed at least once;

2 multicondition coverage, similar to but stronger than branch coverage, meaning that for every branching condition the combination true/false of all participant atomic formulas is checked;

3 relational coverage, a refinement of the previous one, meaning that each binary number comparison, e.g. $a < b$, is covered three times using trichotomy, i.e. the cases $a < b$, $a = b$ and $a > b$ are generated;

4 loop coverage, meaning that every iteration is reached with such input stimuli that it is executed zero, one and many times.

Non-functional requirements are by far more difficult to test. They may have to do with different aspects of safety/security, for instance "no message is lost" or "unauthorized access is always denied". They may also have to do with time constraints, for instance "the page is loaded within 3 seconds". Time constraints are subdivided into soft and hard, the latter meaning that by no means they are to be violated. Furthermore, non-functional requirements might regulate the interplay of processes in a concurrent, distributed environment, for instance starvation prevention when priorities are given.

Safety and security properties are broke down into single requirements that usually are functional requirements, for instance "the user is authenticated". These are then treated as any functional requirement, i.e., for them test cases are generated as usual, e.g. using a use-case-based approach.

In order to assess if a system performs within its timing requirements usually static analysis is used, i.e., the program text is analyzed or inspected, but no test case generation takes place. Timing requirements cannot be tested using a white-box approach, since it makes no sense to take the program text into account if the property being tested has to do with time responses. By default, then, timing requirements are tested using a black-box approach.[1] The technique usually employed is just random testing. Another testing approach suited for timing requirements is evolutionary testing; see Appendix A.

3.4 Test case run

A test case run consists in feeding the UUT with the input as defined by the test case, and in recording the output obtained. Afterwards, the output has to be compared with the output predicted by the test case; see Section 3.5. That is, a test case run is divided into two steps, namely execution and record.

The first step requires moreover to insert the UUT in the context of interest as given by the hypotheses of the test case. For instance, it might be necessary to (possibly artificially) generate workload or a number of database entries when testing performance or time responses.

The second step requires the tester to observe the output generated by the UUT. Different observation mechanisms can be put at disposal. Database entries, for example, might be directly observable or not.

3.5 Error/failure detection

After running a test case, the observed output is compared with the expected one. As sketched in the overview of Section 3.1, to compare and to decide if the run satisfies the test case is not trivial. As a simple example suppose the UUT is the routine that inserts data in a database, among the values to be inserted a date. Assume moreover that the database entries are observable. The UUT might be considered erroneous if dates are stored in a format different from the one we are used to. However, if the routine that reads the stored dates restores them to our usual format, then the UUT should not be rejected. For instance, a noteworthy case is the practice of storing years by omitting the century. This led to the well-known Y2K problem; that program "feature" was not considered an error until the year 2000 approached.

1 This might be the reason why black-box testing and functional testing are sometimes used as synonymous.

In general, the task of comparing expected and observed output, and thus of detecting error or failure, is not trivial. It is referred to as the *oracle problem*; see [Mac00]. This is, on the one hand, because expected output not necessarily is completely specified (i.e., it could be expressed using (meta)variables), and on the other because not necessarily equality but another kind of equivalence suffices for the actual output to be acceptable, where the exact notion of equivalence not necessarily is stated precisely enough.

Unfortunately there is no definitive answer to the question what testing strategy is the best one. It is very difficult to compare two or more given techniques. These can focus on different aspects of the software system under consideration, and thus detect different kinds of defects. Even if they are theoretically designed to uncover any kind of defect, they may prove more adequate for the detection of a particular class of defects and therefore more appropriate for systems showing certain characteristics. In general, effectiveness relates to several factors: testing technique, software type, fault type, tester experience, and any interactions among these factors; see [BS87]. The interpretation of figures arising from an experiment, makes us refer back to the oracle problem, is an utmost difficult task.

4 Inspection

Software inspection is a technique proposed in [Fag76] for early detection (and removal) of defects of software systems. It was introduced in the seventies in reaction to the fact that software testing, as it was performed at those times and to a great extend is still performed nowadays, required the execution of programs for the detection of failures, which in turn means that testing took place after the program had been specified, designed and implemented. That is, the activity of testing occurred simply too late. This meant that the cost of a defect was too high and, in order to reduce it, the idea of inspection was to detect defects earlier. However, an inspection requires an up-front investment in quality. The effort is about 5% to 15% of the total software development effort.

4.1 Overview

The inspection activity is usually organized in four phases:

1 inspection planning,

2 defect detection,

3 defect collection, and

4 defect handling.

An organizer plans the whole inspection activity. A goal and a strategy must be set and training might need to be included. Moreover, the inspection organizer has to determine the so-called unit of inspection, see Section 4.2. Further duties of the inspection organizer is to draw a schedule and to assign roles to experts.

Defect detection is also called reading. This is the most important step of the whole inspection activity. Inspectors have to read their assigned documents searching for defects. These might be classified as violating correctness (or consistency), testability, maintainability, and any of the quality requirements imposed on the software system being developed. Reading needs to be systematic and the inspector has to have the necessary skills to examine and understand any given unit of inspection (of a particular type if using logical entities as unit of inspection).

The defect collection activity simply consists in filtering defect candidates found in the defect detection step by deciding which of them are true defects and

moreover to be handled. The author of the document respectively logical entity collects the defects possibly accompanied by the reader and perhaps also a moderator.

Under defect handling one classically understands defect correction, i.e., the removal of the discovered defect. However, considerable difficulties arise if the defect in question propagates. Although the consequent defects ideally are also detected, and thus there should be no need to care about them, when handling the originating defect the possibility of propagation is to be taken into account. This means, its handling might mean to shift its correction to the next iteration of the software's life cycle.

Possible roles that participate in this process are the organizer, the inspector(s), the author(s), and moderator of the defect collection meeting.

4.2 Unit of inspection

As above mentioned, a very early decision (if not the first) to be taken when undertaking the task of inspecting, is what is the unit of inspection, i.e., what is the information subject of a single inspection. Initially a software document was defined as unit of inspection, and inspection strategies were oriented to document types, like for instance a requirements document. A drawback of defining the unit of inspection to be documents or document types is its lack of scalability.

Meanwhile, a more significant drawback stems from the advent of the object-oriented paradigm and the Unified Modeling Language (UML). Relevant pieces of information related to each other are spread across various (parts of) documents. In this context, it makes more sense to inspect logical entities like for instance a class or a component.

4.3 Inspection concerns

The activity of inspection analyzes the quality of the unit of inspection of interest in two ways, namely regarding its internal consistency and its external consistency. The second aspect takes into account the context in which the document is inserted, in particular in comparison with the documents or units of inspection that previously occur in the system development chain. The two aspects are the following:

1 document type guidelines and other standards, and

2 consistency with previous document(s).

The first concern is not addressed by testing[1]. The second one is addressed by testing only in what respects to dynamic aspects. When considering source code, it is the main regard of testing; see Section 5.

4.4 Reading techniques

The defect collection activity is carried out by reading the different units of inspection. Existing reading techniques are

- ad-hoc reading,
- checklist/questionnaire-based reading
- reading by stepwise abstraction and active design review
- defect-based reading,
- reading based on function point analysis, and
- perspective-based reading; see [Lai00].

The last three usually use a scenario-based strategy for defect detection. A scenario is an algorithmic guideline that tells inspectors what to check and how to detect defects. A scenario thus alleviates the handicap an unexperienced reader may have.

When choosing a reading technique its characteristics have to be taken into account, e.g., in which context can it be applied, its usability, repeatability, adaptability, level of coverage, focus, as well as how far it was validated.

Ad-hoc reading provides no explicit advice for inspectors as to how to proceed, or what specifically to look for, during the reading activity. Checklists offer a better support mainly in form of yes-or-no-questions. Checklist questions interpret specified rules within a project or an organization; see [GG93]. Checklists provide advice about *what* to look for an inspection but not *how* to identify the necessary information and how to check. Structured reading instructions and scenarios were designed to fill this gap.

Reading by stepwise abstraction for code documents (or any document allowing abstraction) and active design review for design documents are reading techniques that provide the inspector with more structured and precise instructions. Stepwise abstraction requires to read a sequence of statements in the code and to abstract the functions these statements compute. This procedure is repeated until the outermost function of the inspected code document has been

1 When performing e.g. white-box testing, the internal consistency might also be critically considered. In this case, however, it is not pure testing what is taking place but testing combined with inspection.

abstracted. This function is then compared to the one originally specified; see [Dye92].

Active design review assigns responsibilities to inspectors and requires of each of them to take an active role in inspecting design documents. For instance, an inspector has to make assertions about the inspected document rather than simply pointing out defects; see [PW87].

A scenario requires an active involvement of the inspector. It helps the inspector focus on a particular kind of defect. It does so by constraining the attention of the inspector with custom guidance, stating what and how to check. In this way the inspection team, whose participants concentrate each on a different type of defects, becomes more effective; see [Bas97]. Three different scenario approaches and hence three different scenario-based reading techniques have been developed: defect-based reading, function point analysis, and perspective-based reading.

Defect-based reading is a customized reading technique for requirements documents. For each defect class there is a scenario, in form of a set of questions, that helps the respective inspector; see [PVB95].

Reading based on function point analysis is also a technique for requirements documents. Function point analysis defines a software system in terms of its inputs, files, enquiries, and outputs. The function point scenarios are defined around these items and let the inspector focus on one such item; see [CJ96].

Perspective-based reading supports an inspector by putting a questionnaire at disposal, that requires the inspector to read from the perspective of a particular stakeholder. The inspector reads the document(s), performs activities, and answers the questions. If logical entities and their documentation are treated separately, there are two possibilities: perspectives can be defined with respect to logical entities or with respect to the documentation of logical entities; there are reasons that speak in favor of following the first alternative. The advantage of this reading technique is that the inspector gets guidance on the reading activity. Disadvantageous is that the use of the reading scenarios requires up-front training, and that the reading activity may require more preparation effort on the inspector's behalf; see [BGL+96,Lai00]. In Appendix B examples of perspective-based questionnaires and instructions are given.

Recently, reading techniques have been studied in the context of experiments. [BGL+96,Lai00] compared perspective-based reading and the NASA SEL reading of requirement documents; [Lai00] compared checklist-based reading and perspective-based reading of design documents and of code documents.

4.5 Functional vs. non-functional requirements

Inspection is suited for the detection of violations of both functional as well as non-functional requirements, as long as these are concerned with static aspects of the system. Dynamic aspects, such as for instance performance, cannot be quality assured simply by inspecting. These can only be addressed by testing, i.e., when the point in time is reached where testing can be carried out.

5 Similarities and differences

Both testing and inspection are quality assurance activities that can be used to detect violations of the functional as well as of the non-functional requirements imposed on the software artifact (or software artifact to be). Inspection and testing are both suited for the quality assurance of functional requirements. With respect to the non-functional ones, it seems that they can differently be employed, depending on the class of requirement. For instance, and considering the taxonomy of the ISO/IEC 9126-1991 standard, inspection might prove more useful than testing for maintainability requirements, whereas testing might prove more useful than inspection for efficiency requirements. This hypothesis can be more precisely stated as follows.

Let us term static those properties of the software system that can be checked without having to run the software. In turn we call dynamic those properties that can only be checked by running the software. On the one hand, inspection makes sense as long as it focuses on requirements that refer to static properties of the system. On the other hand, testing is suited for dynamic properties as well as for external consistency of static properties, meaning consistency with respect to previous document(s) and in particular with the system specification;[1] see Table 1.

	static properties	dynamic properties
external consistency	inspection / testing	testing
internal consistency	inspection	testing

Table 1: Quality assurance concerns

We have already mentioned the difference that testing can only be applied to executable UUTs, whereas any (part of a) document can become unit of inspection, thus inspection can be applied earlier than test. Another difference is given by the specific concerns of both activities as given in Table 1. An additional difference is that inspection further aims at work process improvement and might also be employed to prevent defects; see [GG93].

The additional concerns of inspection enable the development of systems with increased maintainability. In the intersection of concerns, the inspection technique may derive benefit from the experiences in testing and vice versa.

1 Unless the term testing is used when performing not pure testing but testing combined with inspection, e.g. when devising test cases in the framework of white-box testing.

Undoubtedly, both techniques are complementary and by no means can really compete with one another.

A point of contact of these two techniques is given by the concepts of UUT and of unit of inspection. They seem to be one and the same concept whenever the following two conditions hold. First, the subject of interest must belong to the intersection of concerns, namely external consistency of static properties; see Table 1. Second, if interplay of a module or system has to be quality assured (for instance at deployment or integration phase), then the reading technique must allow abstraction in terms of architectural design, akin to stepwise abstraction; cf. the ATAM method in e.g. [KKC00]. This means, and assuming that individual components have passed the quality criteria, the interaction among them can not only be tested but also inspected if the inspection technique includes instructions on how to read protocol definition and architectural design. Only in this case a UUT being the architecture can be also considered the unit of inspection of such an inspection technique.

Before test case generation, i.e. when determining what is the UUT, as well as when generating test cases, a by-product is the inspection of the specification. The rough hypothesis reads: test planning *is* specification inspection.

The error/failure detection step of the testing activity is directly related to the defect collection step of the inspection activity. The first one is addressed to as the oracle problem. A synergy effect could be attained by comparing both steps.

Furthermore, there are three pairs of steps of the above described activities that can be put in relation. Reading could be useful when selecting test data for a white-box testing approach and also for test case generation in general. Fault isolation resembles reading. Defect correction resembles fault correction, especially when software code is being inspected. These three pairs of steps are considered in more detail in the sections to follow. Furthermore, an interlocking of both activities, inspection and testing, is considered in the fourth subsection.

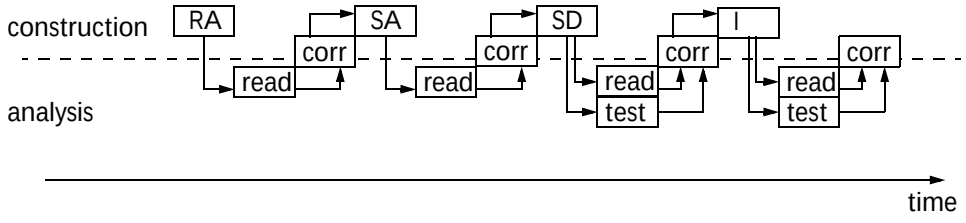

Figure 1: Software systems' lifecycle

Figure 1 shows the lifecycle of software systems (abstracting away from further possible characteristics like iterative/incremental, waterfall, etc.). There, each box represents an activity. These activities are RA, which stands for requirements

analysis, SA for system analysis, SD for system design, I for implementation, corr for correction, read for inspection, and test for testing. The arrows indicate the output of which step (at least) is input for which other step. The dotted line separates activities that can be considered creative from those that are analytical.[1] We have inserted a testing step between design and implementation since in some cases the design can be animated as already commented in Section 3.2.

In the following we use the picture in Figure 1 without arrows to display where the proposed improvements allow for more information flow between the single steps. This added information flow will likewise be depicted in form of arrows but dashed instead of solid. The difference in notation is because the information output at one end not precisely is input (in terms of functional dependency) of the activity at the arrow head. This additional information, sometimes a by-product, could (and should) be taken into account.

5.1 Test case generation and reading

The techniques used for the generation of test cases and those used for reading could be used in combination, in such a way that they profit from one another. On the one hand, the perspective-based reading technique when applied to requirements or analysis documents and from the perspective of a tester is designed only for the detection of defects within the document. While reading, however, the inspector devises test scenarios (see Appendix B.1) and thus could already take care of the definition or even generation of test cases as well. In this way, a drawback of the testing activity, namely that it takes place too late, can be overcome to some degree. Specification tuning and test case generation would be integrated in a single activity: as above mentioned, planning specification-based testing is at the same time specification inspection.

Figure 2 shows the information flow that can be gained in this way: the test scenarios devised while reading requirements (and/or analysis) documents from the perspective of a tester are taken into account while designing test experiments for design or implementation.

On the other hand, in order to perform white-box testing the program source is necessarily examined. Interesting is to consider using the reading technique by stepwise abstraction, which could be useful for the design of a coverage strategy. Moreover, stepwise abstraction of the functions present in the code together with function coverage (i.e., execution of every single function of the

1 This separation is fuzzy. While e.g. the design of test experiments is a creative task, its ultimate purpose is to assess the correctness of the tested software, i.e. testing should be considered an analytical activity.

system, see Appendix A) could be useful for testing performance, which usually cannot be assessed by inspection.

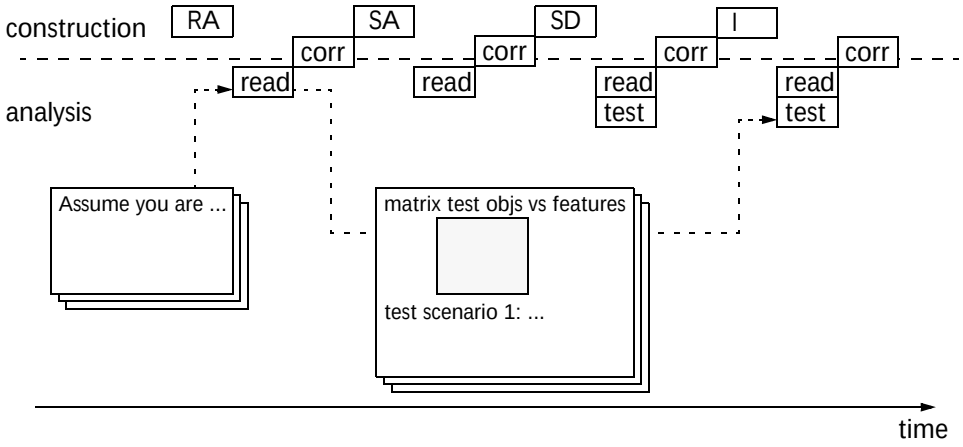

Figure 2: Test case generation and reading combined

Likewise, test documents, as any document, are error prone. Test documents can also be inspected, and it is surely beneficial to do so early. That means, test plans should be early done and inspected. It is argued that it is more effective to define test before developing the life cycle deliverable; see [GG93].

5.2 Fault isolation and reading

Both activities consist in examining a document. In case of fault isolation, it is known that the behavior of the system is erroneous. When reading, on the contrary, it is possible that no defect is found and moreover that no defect exists. Furthermore, reading techniques are concerned with quality aspects that not necessarily are related with faults, for instance the question of a well commented program source. Nevertheless, reading techniques could be helpful for looking for a particular fault. In fact, usually testing is concerned with error/failure detection but no technique or advise is put at disposal when a test fails.

We believe that reading techniques can be helpful once a system has failed to pass a test. Possibly the lessons learned from years of inspecting and of debugging (be this deductive, inductive or ad-hoc) can be smoothly put in relation or even combined. This hypothesis should be empirically supported.

Figure 3 shows the additional information flow that would be gained by combining the activities of fault isolation and reading. The results of the testing activity are used for searching the program fragment that lead to a failure. In this context, searching is performed by reading. (Both activities would then

occur not simultaneously but one after the other, namely reading after testing.) The same applies to the activities of reading and testing design, in the case the specification be animable.

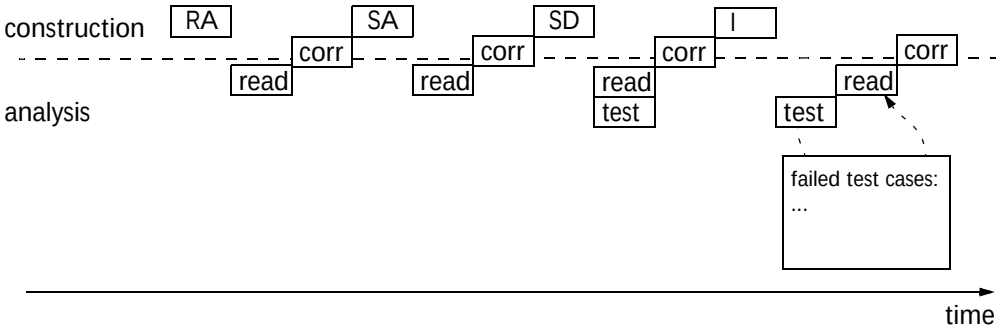

Figure 3: Fault isolation and reading combined

5.3 Fault correction and defect correction

The activity of correction of an error or fault discovered by testing and afterwards isolated in the source code is, in fact, the very same activity of correction of a defect in the source code that was discovered by inspecting. In the second activity other kinds of defects are also corrected, such as those that have to do with aspects that are not taken care of by testing. However, it might be interesting to investigate whether or not the fact that a defect was detected by either technique influences the activity of correcting that defect.

In order to graphic this, the landscape is probably clearer if we have two correction steps before and after the implementation phase, one for the correction of defects detected by testing and one for those detected by reading.[1] In Figure 4 we depict this additional information flow as two additional arrows, one arrow from the testing activity of documents that can be tested (normally just programs but in some cases also design documents) to the correction step of defects unveiled by reading, the other arrow from the reading activity to the correction step of defects discovered by testing.

1 This, however, could potentially confront us with possible document inconsistencies arising from parallel and independently tackled corrections.

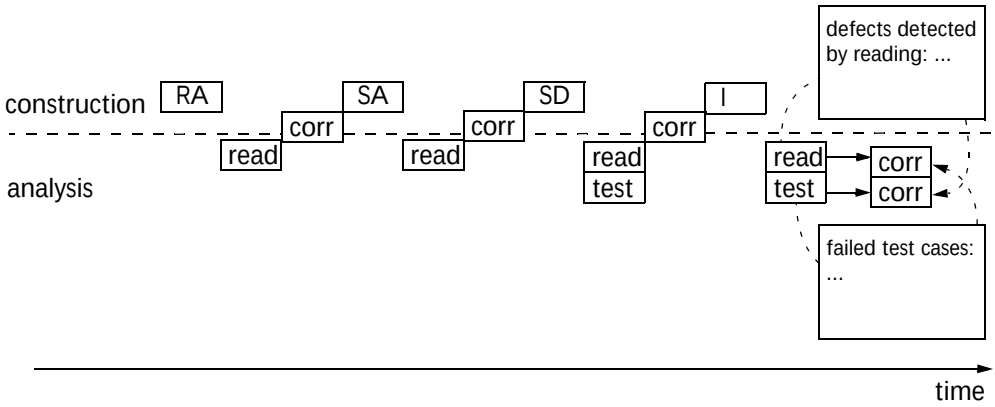

Figure 4: Fault correction and defect correction combined

5.4 Inspection and testing interleaved

Inspection and testing can be complementary employed in more than one way. For instance, once a unit has been inspected, classes of errors are identified which serve as a basis for inspections to follow. These classes could also be used to generate test cases. The question here is if it makes sense to focus on recurrent faults discovered by inspection or, on the contrary, to focus on orthogonal faults in this way relying on successful defect correction. This standpoint is referred to as methodical integration of inspection and testing in [PSBK+01]: the results of an inspection are useful for the creation of a test specification.

Another kind of integration of inspection and testing is the so-called empirical one (see [PSBK+01]), in which the interaction of both activities is bidirectional. On the one hand, the analysis of fault classes identified by inspection allow a targeted test planning. On the other hand, the analysis of the test results make possible an improvement of the inspection method (and of the test cases for the next life cycle iteration as well).

The proposed interaction is pictured in Figure 5 still as an arrow from one activity to the other, although the proposal does not consist, as the former ones do, in an additional information flow from one step to another one which is not usually present in conventional system development process (see Figure 1). The above mentioned interleaving rather suggests to melt both steps in a single one. (The fact that they might occur simulatneously supports the idea.) Or, more ambitiously, to improve each one of the involved activities on the basis of the

experiences gained after performing the other activity (in the next iteration of the software's life cycle.

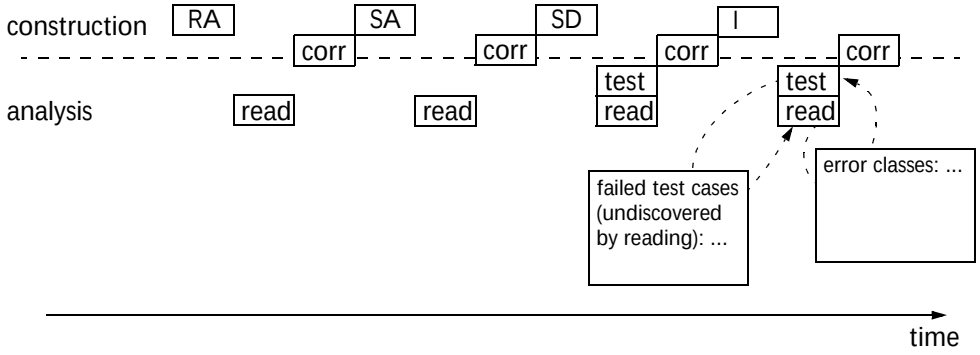

Figure 5: Inspection and testing interleaved

The hypothesis of [PSBK+01] with respect to both methodical and empirical integration of inspection and testing techniques is twofold. First, more defects are detected using an integrated approach than using any of the two techniques in isolation. Second, the integrated approach is more cost-effective, i.e. more defects are detected per time unit, than inspection or testing carried out in isolation. These assumptions should be empirically supported, too.

6 Conclusions and outlook

After consulting different libraries and asking some experts we come to the conclusion that unfortunately none or at most very little experience is reported (and this hints at little experience done) on combining both approaches, testing and inspection, although it seems that doing so can only be of benefit. The experiment presented in [Mye78] compares black-box testing, white-box testing and inspection, performed by experienced programmers, of a text formatting program written in PL/I; it draws interesting conclusions:

1 *None of the methods, when used alone, is very good, since they detected only about a third of the errors of this small and simple program. An implication here is that the walkthrough/inspection technique should be viewed as a supplement to, rather than a replacement for, traditional testing methods.*

2 *Inspection has a higher labor cost. On the macroscopic level, however, such differences should prove to be insignificant.*

The second sentence, however, can probably be weakened if one takes into account that at those days inspection was a relatively new technique, i.e. people were less experienced with it. Furthermore, [Mye78] compares testing and inspection of a program; other experiments like for instance the one given account in [FKLR00], reports on impressive savings reached by employing inspections in earlier stages of development compared to the use of testing techniques only.

Surely as future work further controlled experiments should take place, in order to compare conventional processes with a combined one.

7 References

[Art01] ARTiSAN Software Tools. ARTiSAN Real-time Studio. http://www.artisansw.com/products/professional_overview.asp, 2001.

[Bas97] Victor Basili. Evolving and packaging reading technologies. *Journal of Systems and Software*, 38(1):3–12, July 1997.

[BDG+98] David Banks, William Dashiell, Leonard Gallagher, Charles Hagwood, Raghu Kacker, and Lynne Rosenthal. Software testing by statistical methods. Technical report NISTIR 61 29, National Institute of Standards and Technology, March 1998.

[Bei90] Boris Beizer. *Software testing techniques*. Van Nostrand Reinhold, New York, 1990.

[BG99] Imram Bashir and Amrit L. Goel. *Testing Object-Oriented Software: Life-Cycle Solutions*. Springer, New York, 1999.

[BGL+96] Victor Basili, Scott Green, Oliver Laitenberger, Filippo Lanubile, Forrest Shull, Sivert Sørumgård, and Marvin Zelkowitz. The empirical investigation of perspective-based reading. *Journal of Empirical Software Engineering*, 2(1):133–164, 1996.

[BKLW95] Mario Barbacci, Mark Klein, Thomas Longstaff, and Charles Weinstock. Quality attributes. Technical report CMU/SEI-95-TR-021, Software Engineering Institute, Carnegie Mellon University, 1995.

[BS87] Victor Basili and Richard Selby. Comparing the effectiveness of software testing strategies. *IEEE Transaction on Software Engineering*, SE-12(12), December 1987.

[CF00] Angelo Ciarlini and Thom Frühwirth. Automatic derivation of meaningful experiments for hybrid systems. In *ACM SIGSIM Conference on AI, Simulation and Planning (AIS'2000, proceedings), Tucson, Arizona, USA*, March 2000.

[CJ96] Benjamin Cheng and Ross Jeffery. Comparing inspection strategies for software requirements specifications. In *1996 Australian Software Engineering Conference (proceedings)*, pages 203–211, 1996.

[DLMT96] Marita Duecker, Georg Lehrenfeld, Wolfgang Mueller, and Chris-

toph Tahedl. Specification and analysis of concurrent systems in a complete visual environment. In *European Simulation Multiconference (ESM'96, proceedings)*, 1996.

[Don98] Michael R. Donat. *A Discipline of Specification-Based Test Derivation*. PhD thesis, University of British Columbia, 1998.

[Dye92] Michael Dyer. Verification-based inspection. In *26th Annual Hawaii International Conference on System Sciences (proceedings)*, pages 418–427, 1992.

[Fag76] Michael E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

[FKLR00] Bernd Freimut, Brigitte Klein, Oliver Laitenberger, and Günther Ruhe. Measurable software quality improvement through innovative software inspections technologies at Allianz Life Assurance. Electronic Publication IESE-Report 014.00/E, Fraunhofer Institute for Experimental Software Engineering, 2000.

[Gau95] Marie-Claude Gaudel. Testing can be formal, too. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *Theory and Practice of Software Development, International Joint Conference CAAP/FASE (TAPSOFT'95, proceedings), Aarhus, Denmark*, volume 915 of *Lecture Notes in Computer Science*, pages 82–96. Springer-Verlag, May 1995.

[GG93] Tom Gilb and Dorothy Graham. *Software Inspection*. Addison-Wesley, 1993.

[GJ98] Marie-Claude Gaudel and Perry R. James. Testing data types and processes, an unifying theory. In *3rd ERCIM International Workshop on Formal Methods for Industrial Critical Systems (FMICS'98)*. CWI, May 1998.

[GJ99] Marie-Claude Gaudel and Perry R. James. Testing algebraic data types and processes: a unifying theory. *Formal Aspects of Computing*, 10(5-6):436–451, 1999. Long version of [GJ98].

[Gly80a] Clark Glymour. Hypothetico-deductivism is hopeless. *Philosophy of science*, 47:322–325, 1980.

[Gly80b] Clark Glymour. *Theory and Evidence*. Princeton Univ. Press, New Jersey, 1980.

[I-L] I-Logix. Rhapsody. http://www.ilogix.com/products/rhapsody/

index.cfs.

[Gly83] Clark Glymour. On testing and evidence. In John Earman, editor, *Testing scientific theories*, volume X of *Minnesota Studies in the Philosophy of Science*, chapter 1. Univ. of Minnesota Press, 1983.

[Kin94] Ekkart Kindler. Safety and liveness properties: A survey. *EATCS-Bulletin*, 53:268–272, June 1994.

[KKC00] Rick Kazman, Mark Klein, and Paul Clement. ATAM: Method for architecture evaluation. Technical Report CMU/SEI-2000-TR-004, Software Engineering Institute, Carnegie Melon University, 2000.

[Lai00] Oliver Laitenberger. *Cost-effective detection of software defects through perspective-based inspections*. PhD thesis, Universität Kaiserslautern, 2000.

[Mac00] Patricia Machado. *Testing from Structured Algebraic Specifications: The Oracle Problem*. PhD thesis, University of Edinburgh, 2000.

[Mar95] Brian Marick. *The Craft of Software Testing: Subsystem testing*. Prentice Hall, Englewood Cliffs, New Jersey, 1995.

[Mer79] G. H. Merrill. Confirmation and prediction. *Philosophy of science*, 46:98–117, 1979.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997.

[Mye78] Gleford Myers. A controlled experiment in program testing and code walkthroughs/inspections. *Communications of ACM*, 21(9):760–768, September 1978.

[Mye79] Glenford Myers. *The Art of Software Testing*. John Wiley and Sons, 1979.

[PSBK+01] Barbara Paech, Thomas Santen, Ulrike Becker-Kornstaedt, Bernd Freimut, Erik Kamsties, Antje von Knethen, Oliver Laitenberger, Maud Schlich, Dirk Seifert, and Carsten Sühl. Ziele, Hypothesen und Validierungsmöglichkeiten für das Projekt QUASAR. Technical Report IESE-Report 049.01/D, Fraunhofer Institute for Experimental Software Engineering, 2001. (In German.)

[PVB95] Adam Porter, Lawrence Votta, and Victor Basili. Comparing detection methods for software requirements inspections: a replicated experiment. *IEEE Transactions on Software Engineeering*,

21(6):563–575, June 1995.

[PW87] David Parnas and David Weiss. Active design reviews: principles and practice. *Journal of Systems and Software*, 7:259–265, 1987.

[RG99] Johannes Ryser and Martin Glinz. A practical approach to validating and testing software systems using scenarios. In *Third International Software Quality Week Europe (QWE'99, proceedings)*, November 1999.

[Ros97] David Rosenblum. Automated monitoring of component integrity in distributed object systems. In *Advanced Topics Workshop of the 3rd USENIX Conference on Object-Oriented Technologies and Systems*, June 1997.

[SGG01] Abraham Silberschatz, Baer Peter Galvin, and Greg Gagne. *Operating System Concepts*. John Wiley & Sons Inc, 6th edition, 2001.

[Sof02] Softera Ltd. SoftModeler. http://www.softera.com/products.htm, 2002.

[Sto96] Neil Storey. *Safety-Critical Computer Systems*. Addison-Wesley, Harlow, England, 1996.

[SW02] Harry M. Sneed and Mario Winter. *Testen objektorientierter Software*. Carl Hanser Verlag, 2002.

[Tan98] Qiang Ming Tan. *On Conformance Testing of Systems Communicating by Rendezvous*. PhD thesis, Laboratoire de téléinformatique, University of Montreal, 1998.

[Tha99] Peter Thanisch. Workload characterisation and generalisation. Technical Report ICSA-46-99, Department of Computer Science, University of Edinburgh, May 1999.

[TR93] Chris Turner and Dave Robson. State-based testing and inheritance. TR 1/93, Computer Science Division, University of Durham, 1993.

[Voa00] Jeffrey Voas. Software fault injection. *IEEE Spectrum*, 2000.

A Testing nomenclature

The definitions herein were kindly put at disposal by Hans-Gerhard Groß (and here classified and slightly edited).

A.1 Error, fault, failure, defect

According to IEEE-Std-610.12-1990:

1 *Error* is the difference between computed, observed, or measured value or condition and the true, specified or theoretically correct value or condition: (a) an incorrect step, process, or data definition, (b) an incorrect result, (c) a human action that produces an incorrect result.[1]

2 *Fault* is a defect in a hardware device or component (short circuit or broken wire), or an incorrect step, process, or data definition in a computer program. (This definition is primarily used in fault tolerance discipline. In common usage, the terms error and bug are used to express this meaning.)

3 *Failure* is the inability of a system or component to perform its required functions within the specified performance requirements.

Errors, faults and failures are *defects*. A defect is defined as a product anomaly, according to the IEEE standard 100-1992.

Alternative definitions see the term error as being used during the development of software, and the terms fault or failure as being used after the software has been released to a customer. Every fault in the system is either caused by decay or by an error during the development of the software. Decay is usually limited to hardware. Software as such cannot decay, therefore it cannot fail randomly.[2] Every fault in the software is associated to an error which was introduced during the development.

1 Some authors call error what the programmer, designer, etc., does that leads to a failure. That is, for these authors, *error* is what is stated under 1(c), *fault* is what is stated under 2, and *failure* is what is stated under 1(a), 1(b), and 3.

2 It could nevertheless be said that software can indeed decay namely through a virus.

A.2 Different kinds of testing

The terminology used by the testing community includes a large number of names, and here we list many of them. Before doing so, however, we want to classify the different terms and group them under a common denominator.

Testing approaches can be classified according to:

- *what* to test
- *how* to test
- *when* to test

Among the second ones we distinguish those speaking of the testing procedure itself and those establishing how to select test data. The entire set of test cases generated for a particular test is called the *test suite*. Within the following sections the keywords are alphabetically ordered.

A.2.1 What to test

Acceptance testing is performed by the customer in order to verify whether the software complies with his/her expectations. Acceptance testing usually follows a black-box approach. Acceptance testing is divided into two categories, *alpha testing* performed at vendor's site on an non-released system, and *beta testing* performed at customer's site on a released system.[1]

Contract testing verifies that interacting components abide by their contracts; see [Mey97].

Conformance testing is the process of verifying that an implementation performs in accordance with a particular standard, specification, or environment. Conformance testing is exclusively concerned with the external behavior of an implementation, i.e., it follows a black-box approach. Service and functional behavior are tested in order to find logical errors and prerequisites for interoperability. Conformance testing is not intended to be exhaustive, and a successfully passed test suite does not imply a 100% guarantee. But it does ensure, with a reasonable degree of confidence, that the implementation is consistent with its specification, and it does increase the probability that implementations will inter-operate.

1 Sometimes under the term acceptance testing is also understood the test (automatically) performed in order to decide whether or not a process is allowed to enter in a context of other processes. This kind of testing is outside the scope of this article.

Design testing is not really testing but an inspection of the design documentation.

Functional testing concentrates on the input/output relation as given by the specification. The test case generation can be done by just taking the specification into account (see black-box testing) or else by considering also how the function was implemented (see white-box testing). Sometimes, however, functional testing and black-box testing are considered synonymous.

Performance testing is concerned with the testing of non-functional features. This verifies whether a software complies with its performance requirements. This may be a required number of accesses at a time or a required minimum or maximum response time. The second case is particularly important for scheduling analysis for real-time systems. There are automatic tools for the detection of performance bottle-necks.

Reliability testing verifies whether software works correctly when expected situations arise.

Requirements testing (or testing of the requirements) is not testing but an inspection of the defined requirements with respect to the user's needs. Requirements cannot be executed in a test. The software testing aspect only comes in through the way in which the inspection of requirements may be performed. Each requirement can be formally defined through a number of perspective test-cases. These will be used to test the implementation of that requirement. Test cases are a much more formal and distinct notation for requirements. They can therefore help to discover possible ambiguities in the specification; see [BG99]. Requirements testing is sometimes wrongly used for requirements-based testing.

Safety testing concentrates of non-functional features. Safety testing verifies whether the safety features of a safety critical system comply with the safety specification; see [Sto96].

Subsystem testing is use-case-based (functional) testing of system (system testing) parts. The use cases are part of the specification.

Syntax testing verifies whether internal and external inputs conform to the unit under test. This is typically done through the compiler, especially when strongly typed languages are used.

During the *system test*, the quality of an entire software product is assessed. This is corresponds to integration testing. System testing verifies whether the total combination of all components inter-operate properly.

Temporal testing verifies the timing behavior of a system, i.e., it is concerned with non-functional features. In particular, temporal testing verifies whether a real-time system complies with its timing specification. This relates to dynamic execution time analysis as opposed to static execution time analysis (static analysis) and may be based on optimization techniques (see optimization-based testing and evolutionary testing).

Unit testing verifies that the smallest software unit is performing according to its specification and design documents. In traditional procedural-oriented development this is a single function. In the object-oriented development paradigm, the smallest testable software unit is a class/object or a component.

Usage-based testing concentrates on testing system sections which are likely to be used more extensively than other sections. Usage-based testing is one aspect of statistical testing.

A.2.2 How to test: testing procedure

Adequacy testing measures whether a test set is adequate for a test criterion (mutation testing, for instance, is one way to verify test set adequacy) and provides a decision criteria on when to stop testing (i.e., when a software is adequately tested).

Automatic testing (or *test automation*) is concerned with automatic test-case generation and/or test-case execution through suitable automatic test programs.

Black-box testing is done with no knowledge of the internal structure of the system. It can be used to test the functional requirements (i.e., the input/output relation) of a system as well as its non-functional requirements (e.g., performance, reliability, etc.).

Built-in testing stipulates that components be equipped with testing procedures (or methods). This means, the testing software is designed and built together with and into the functional software. In doing so, the testing methods might violate the principle of encapsulation.

Dataflow testing concentrates on the dataflow relations in a program as a test case selection criterion. It is a family of test strategies based on selecting paths through the program's control flow in order to explore sequences of events related to the status of data objects; see [Bei90].

Definition-use path (or du-path) of a variable is a path in the implementation of a procedure (or method) that starts at the variable definition/initialization and ends at the point where the variable is first assigned. Along its du-path, a vari-

able passes unaffected. *du-path testing* is a particular case of dataflow testing that aims at covering all possible du-paths in a program.

Fault-injection testing reminds playing "what if" scenarios, which is done by code mutation (see mutation testing) and data mutation or perturbation. Fault-injection testing verifies whether the software safely behaves when unexpected situations arise; see [Voa00].

Glass-box testing is a synonym of white-box testing.

LCSAJ (linear code sequence and jump) testing is particular case of white-box testing. It uses a path coverage criterion that considers only sub-paths which can easily be represented in the program source code, without requiring a flow graph. An LCSAJ is a set of source code lines executed in sequence. This "linear" sequence can contain decisions as long as the control flow actually continues from one line to the next at run-time. Sub-paths are constructed by concatenating LCSAJs.

Mutation testing is a way to validate the adequacy of a test suite for a test object (testing the test data). Mutation testing applies the test suite to a set of programs, the so-called mutants, which are different from the original program. Each mutation corresponds to a fault in the original program. The test suite is applied to the mutants and its ability to detect the artificially introduced faults is verified. This is used to give a prediction on the testability of the test object and the adequacy of the test cases to uncover faults. One distinguishes between *strong mutation testing*, meaning that a test case causes a different output value at the end of the program, and *weak mutation testing*, meaning that a test case causes a different value at the point of mutation.

Program-based testing is a synonym of white-box testing.

Regression testing executes an amended software package with existing test cases in order to verify that through the change of the software no new errors have been introduced.

State-transition testing concentrates on the verification of the correct transition from one state to another according to external stimuli. The system is regarded as a state machine with an input as stimulus; see [Bei90].

Statistical testing concentrates on the testing (number of executed tests) on areas of the program according to the following criteria (see [BDG+98]):

- complexity of components: the most complex areas of a system are exercised more thoroughly;
- safety-criticality of components: the most critical areas of a system are exercised more thoroughly;

- usage of components: the areas of a system which are used the most (according to some usage profile) are exercised more thorouhgly.

Structural testing is a synonym of white-box testing.

Transaction-flow testing is a technique concerned with transactions. Transactions (scenarios) are coherent tasks which may be subdivided into multiple functions. In order to assess whether a transaction is carried out correctly, suitable test cases must be found which cover a single transaction. Identifying the transaction flows must be performed prior to testing if the possible scenarios are not already specified. Transaction flow testing becomes increasingly important in component-based testing. Here, components consist of multiple units, and several of these are involved in carrying out a single transaction.

White-box testing is used to test the functional requirements of a system using its internal structure.

A.2.3 How to test: test data selection techniques

Boundary value analysis is a test data selection technique which concentrates on boundary values (or values at domain boundaries; see domain testing). These typically are minimum and maximum values, values just inside/outside domain boundaries, typical values, and error values.

Branch-condition (combination) testing is a test data selection technique for white-box testing that concentrates on the outcome of the branch conditions of a program.

The input domain of a function can be subdivided according to the expected behavior of the function. *Domain testing* verifies whether the test object performs in each subdomain according to its specification. If that subdivision is performed taking into account only the function specification, then domain testing follows a black-box approach (see black-box testing); if the particular implementation is taken into account, then a white-box approach is followed (see white-box testing). In the first case, i.e., if only the specification is used to divide the domain in subdomains (which in fact might be reflected in the decision structure of the final implementation), then this testing technique is also called *equivalence partitioning*. In any case, the testing should always consider boundary values of these subdomains; see [Mar95].

Evolutionary testing (also called *optimization-based testing*) is an automatic test-case generation and testing technique based upon the application of evolutionary algorithms such as genetic algorithms, evolution strategies, or genetic programming. These algorithms are based on optimization techniques which perform on populations of individuals. These individuals represent possible

solutions to the problem; in case of testing, they are test-cases. Evolutionary algorithms recombine and mutate the strings and the resulting new strings are selected according to a cost function to form a new population. This process of generating new solutions out of existing ones is repeated, so that over time the population is likely to consist of fitter individual strings, which in case of testing represent better test-cases. Evolutionary algorithms can be used in combination with structural testing, with temporal testing, and with safety testing.

Functional requirements of programs can be represented as decision tables (Karnaugh-Veitch Charts). *Logic-based testing* is test-case generation according to decision tables.

Off-nominal testing is a test data selection according to the operational profile of an application. This is equivalent to statistical testing.

Optimization-based testing is an automatic test case generation methodology based on the application of search/optimization techniques. Among these techniques we find *random testing* and *evolutionary testing*.

Program coverage criteria is an important test case selection criterion. Not every possible input combination of a program can be tested because of combinatorial explosion according to the number of input parameters. A reasonably sized but adequate subset of all possible tests must be found. The philosophy behind coverage is that an error in the program must have been executed (covered by the execution) before it can lead to a failure of the system and consequently become apparent. Therefore, a test set must be found which maximizes the coverage of a program. Coverage criteria are classified according to which code artifact they consider:

- *Full path coverage*: Execute all possible paths through the program's control flow graph. This is the strongest coverage criterion and usually impossible to achieve since the number of possible paths is infinite. Full path coverage includes multiple iterations through loops.
- *Path coverage*: Execute all possible entry-exit paths through the program's control flow-graph. This is a weaker criterion than full path coverage and covers only complete paths through the program. A complete path starts at the start-node of a control flow-graph and ends at its stop-node.
- *Statement coverage, node coverage*: Execute all nodes in the program's control flow-graph. This is equivalent to all statements in a program, since a statement corresponds to a node in the flow-graph. This is the weakest coverage criterion, and is denoted by C1.
- *Boundary value coverage*: Execute all boundary values of a program's equivalence classes.
- *Branch coverage, link coverage*: Execute all links in the program's control flow-graph. Links are the alternatives which start at a decision-node in a program's control flow-graph. Branch coverage strictly includes node coverage

in structured programming. This is a stronger coverage criterion than node coverage, and it is denoted by C2.
- *Branch condition coverage*: Execute all branch condition outcomes in all decisions.
- *Predicate coverage*: Execute all individual predicate expressions in all decision-nodes in a program's control flow-graph. A predicate is a logical function which evaluates to true or false. If a decision-node is consisting of a single individual predicate, then predicate coverage corresponds to link coverage. Compound predicates are consisting of a number of logically combined individual predicates. Full predicate coverage evaluates all of these single predicates.
- *Function coverage*: Execute all functions which are contained in a system.
- *Transaction coverage*: Execute all functions which belong to a distinct transaction in the software. This criterion is required for transaction-flow testing.

Random testing is a purely random test data selection mechanism; it is an optimization-based testing technique.

Requirements-based testing (testing according to the requirements) is the generation of test scenarios for a test which is driven by some requirements document. In fact, any testing focuses on the specification of the requirements for a system since the specification defines what a system must do (and also how fast, etc.) and a test verifies whether the final system actually does it.

Specification-based testing is another name for requirements-based testing.

Test effectiveness ratio (TER) is a group of metrics which express how effective a test set is according to some predefined criterion. For example the TER for statement coverage is the number of exercised source code statements divided by the total number of executable source code statements.

A.2.4 When to test

Component-based testing is concerned with testing the interactions (usually client-server relationship) between components when they are assembled and integrated to form a new system; see also integration testing. Component-based testing does not violate the principle of encapsulation.

Deployment testing is performed when a system is deployed into a new environment. Thereby it is verified that the interactions between the platform and the deployed system behave as expected. The integration of an existing external or remote component into an existing system can also be considered deployment; in this case, the system into which the component is deployed is the platform.

Integration testing is performed when several independent software components are combined into one product. This assesses the correctness and dependability of the smaller units in a new environment, in a similar way as deployment testing. Integration testing is typically performed incrementally. This means that tests are executed when an individual component is introduced. Integration testing can be regarded as a form or a subset of system testing.

B Examples of perspective-based reading

In the following we give detailed instructions for reading from the perspective of a tester and from the perspective of a designer. The first example was kindly put at disposal by Maud Schlich (here slightly edited as well as translated), the second one by Oliver Laitenberger.

B.1 PBR-scenario for tester

Assume you are the system tester. Your duty is to plan the testing activities as early as possible. In this setting it is especially important that both the described requirements be able of validation and you be able of estimate necessary resources for testing. The essential value of the analysis document is, for you, that the requirements be able to be validated.

Search in analysis document the use case diagram and the corresponding use cases.

1 Create the matrix of test objects vs. features.

   a Arrange the test objects or (sub)systems hierarchically, where the first test object is the whole system. Each system is assigned the corresponding observed (or monitored) and controlled variables (or items), and is divided in arbitrarily many subsystems.

   b Build a table whose columns are headed by the features and quality standards to be achieved by the system and whose rows are titled by the test objects of (a).

   c In each cell of this table note whether or not this concrete combination is to be tested. If yes, indicate by which test procedure; if not, succinctly justify.

2 Sketch the man/machine interface and write down possible actions of the user. Compare these with the use cases and compile exemplary test cases for all actions, valid or not.

   a Which observed and controlled variables are unnecessary or missing?, why?

   b Which features are unclear?

c For which cells in the matrix of test objects vs. features are still test procedures missing?

d Which test cases cannot be completely specified?

e Which events should happen due to invalid actions?, where would you amend with exceptions?

f Which quality standards do not suit your man/machine interface?

B.2 PBR-scenario for designer

Assume you are inspecting the design diagrams from the perspective of a designer. The main concern of a designer is to ensure the correctness and completeness of the design diagrams with respect to the analysis diagrams.

Correctness and completeness can be described as follows. Correctness means that a diagram or a set of diagrams is judged to be equivalent to some reference standard that is assumed to be an infallible source of truth. In the case of design diagrams, the analysis diagrams can be considered as such.

Completeness means that a diagram or a set of diagrams includes all its required elements. It is judged by determining if the entities in the (set of) diagrams describe the aspects of knowledge being depicted in sufficient detail for the part of the system under development.

The development products of relevance are the system class diagram, the operation schemata, the collaboration diagrams, and the design class diagrams. Follow the instructions below and answer the questions carefully.

Read the introductory part of the case study document and the use cases. Continue with the following instructions:

1 Locate the design class diagrams in the case study document. The design class diagrams needs to be compared to the system class diagram in the following manner:

- For each system class diagram ensure that each construct in the system class diagram is realized in the design class diagram. “Realizing” refers to
  – Classes: Go to a class in the system class diagram and check that the class is also part of the design class diagram. Check the correctness of the class names, attribute names, and method names between the class in the system class diagram and the design class diagram.
  – Attributes: Check the correctness of the number and types of attributes between the classes in the system class diagram and design class diagram.

  - Methods: Check the correspondence of methods between the system class diagram and the design class diagram.
  - Associations: Check the correctness of the binary associations between classes in the system class diagram and the design class diagram. Compare the cardinality and the arity of the associations.
  - Constraints: Check the correctness of the constraints between classes in the system class diagram and the design class diagram.
  - Abstract Concepts: Check that abstract concepts such as association classes and ternary associations in the system class diagram have been properly mapped to constructs in the design class diagram.
    Be aware that the design class diagrams are a refinement of the system class diagrams and may therefore contain more classes than the system class diagrams!
- Mark the classes in the system class diagram with a pen *after* you have performed the checks.
- Look at the system class diagrams and check that each class has been marked in the previous effort.

2 Locate the collaboration diagrams in the case study document. Each collaboration diagram needs to be compared to the corresponding operation schemata in the following manner:

- Check that the collaboration diagram has a message that corresponds to the system operation.
- Find the entities in the operation schema preceded by the keyword supplied. These represent parameters of the operation. Find the message in the collaboration diagram corresponding to the system operation and check that the parameters and their types match. Mark the checked messages with a pen.
- For each entity in the "reads" and "change" clause not preceded by the keyword supplied, locate one or more corresponding messages in the collaboration diagram and check that it has the necessary parameters or return values to carry the required information.
- For each message in the "sends" clause check that there is a corresponding message in the collaboration diagram with corresponding parameters.
- Check that the messages in the collaboration diagram as a whole achieve the effects defined in the "result" clause of the operation schema.

While completing the instructions answer the following questions:

1 Is there anything you realized in the analysis document that is not reflected in the design document?

2 Is an appropriate message sent to the corresponding object in the collaboration diagram for every attribute, object or link that is changed in the operation schema?

3 Are the initial conditions for starting up a system operation clear and correct?

4 For every message that is defined in the "sends" clause of the operation schema, is there a corresponding message sent in the collaboration diagram?

5 Is the sequence of messages in the collaboration diagram correct?

6 For every attribute, object, or link that is changed in the operation schema, is an appropriate message sent to the corresponding object in the collaboration diagram?

7 Are there any discrepancies between the functionality defined in the "results" clause of the operation schemata and the one described in the collaboration diagram?

Appendix B:Examples of perspective-based reading

Document Information

Title:	Inspection and Testing Towards combining both approaches
Date:	April 10, 2002
Report:	IESE-024.02/E
Status:	Final
Distribution:	Public