Nr.: FIN-05-2008

Evaluation Approaches in Software Testing

Ayaz Farooq, Reiner R. Dumke

Arbeitsgruppe Softwaretechnik

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Technical Report

Impressum (§ 10 MDStV):

Herausgeber:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Der Dekan

Verantwortlich für diese Ausgabe:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Reiner Dumke
Postfach 4120
39016 Magdeburg
E-Mail: dumke@ivs.cs.uni-magdeburg.de

http://www.cs.uni-magdeburg.de/Preprints.html

Auflage: 81

Redaktionsschluss: Juli 2008

Herstellung: Dezernat Allgemeine Angelegenheiten,
Sachgebiet Reproduktion

Bezug: Universitätsbibliothek/Hochschulschriften- und
Tauschstelle

Otto-von-Guericke-University of Magdeburg

Faculty of Computer Science
Institute for Distributed Systems
Software Engineering Group

Evaluation Approaches in Software Testing

Authors:

Ayaz Farooq
Reiner R. Dumke

University of Magdeburg
Faculty of Computer Science
P.O. Box 4120, 39016 Magdeburg
Germany

Farooq, Ayaz
Dumke, Reiner R.
Evaluation Approaches in Software Testing
Technical Report
Otto-von-Guericke-University of Magdeburg
2008.

Contents

1 Introduction

There have been many reports of catastrophic effects of software failures. Peter Neumann's regular column *Risks to the Public* in ACM's Software Engineering Notes magazine lists several accounts of everyday incidents arising primarily due to software failures. The consequences of software failures may vary between mild and severe depending upon the kind of system involved [iee, 1990].

Software testing is used as a primary quality assurance technique to establish our confidence over successful execution of software. A detailed report [Tassey, 2002] analyzes economic impacts of insufficient software testing. The report summarizes the effects on software industry due to inadequate test technology as,

- increased failures due to poor quality
- increased software development costs
- increased time to market due to inefficient testing
- increased market transaction costs

When we acknowledge the criticality of software testing we must pay special attention to manage this activity. While we are attempting to manage testing, we often come across probably the two most common questions. First, when testing should be stopped and software be released? While there may be many structured approaches for the purpose based on reliability, defects, or economic value [Sassenburg, 2005], a practitioner's response most probably would be 'when there is no more time or money left to invest!'. Second, how effectively and efficiently testing is being (or has been) performed? It is a kind of continuous in-process and post-process evaluation of testing to track, monitor and control these activities. This spans determining efficiency and effectiveness of techniques used, process and activities carried out, and testing tools applied. But other finer criteria such as predictability and reliability could also be interesting to investigate. Defect detection rate is commonly used to evaluate testing artifacts, but we will need other measures too, for evaluating such numerous criteria. In this regard, a lot of evaluation techniques and criteria have been developed.

This report intends to summarize available evaluation approaches in the area of software testing. Available functional and quality criteria against which we can benchmark our various testing artifacts will be surveyed. Strengths and weaknesses of existing techniques will be analyzed and possibility of future work will be explored and suggested.

1.1 Evaluation Defined

When it comes to software engineering in general and software process in particular, the terms *evaluation* and *assessment* are interchangeably used in literature and practice. We however differentiate between them and follow the viewpoint of Kenet and Baker [Kenett and Baker, 1999] which seems quite logical specially in view of available process evaluation approaches. The nature of the software evaluation, according to him, may be qualitative ("*assessment*") or quantitative ("*measurement*"). "Measurement encompasses quantitative evaluations that usually use metrics and measures which can be used to directly determine attainment of numerical quality goals. On the other hand, any evaluative undertaking that requires reasoning or subjective judgment to reach a conclusion as to whether the software meets requirements is considered to be an assessment. It includes analysis, audits, surveys, and both document and project reviews" [Kenett and Baker, 1999]. Figure 1.1 visualizes this relationship.

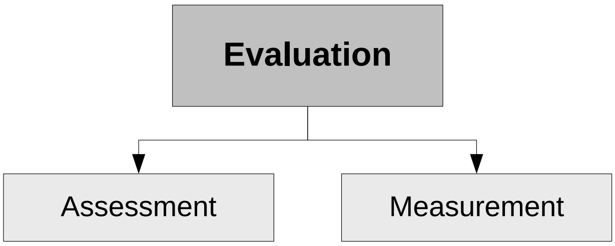


Figure 1.1: Relationships among evaluation, assessment, and measurement

This text will follow this distinction between qualitative and quantitative evaluations while studying and analyzing evaluative works in the discussed areas.

1.2 Evaluation in Software Engineering

A very promising classification of software engineering (SE) research problems has been given by Lázaro and Marcos [Lázaro and Marcos, 2005]. They distinguish SE research into *engineering problems* (concerned with the formulation of new artifacts) and *scientific problems* involving analysis of existing artifacts. One of the criticisms to software engineering research is that it ignores evaluation [Zelkowitz and Wallace, 1997]. This opinion is further strengthened by a survey conducted by Glass et al. [Glass et al., 2004] in which it was found that 79% of approaches in the field of general computer science and 55% of approaches in software engineering were formulative in nature while only about 14% approaches were evaluative works. Perhaps still today many research efforts follow the research model that Glass [Glass, 1994] once described as *advocacy research* consisting of steps, "conceive an idea, analyze the idea, advocate the idea" ignoring the comparative evaluation among the proposed and existing approaches.

Evaluation is an important tool of software quality assurance. A typical software quality program involves i) establishment, implementation, and control of requirements, ii) establishment and control of methodology and procedures, and iii) software quality

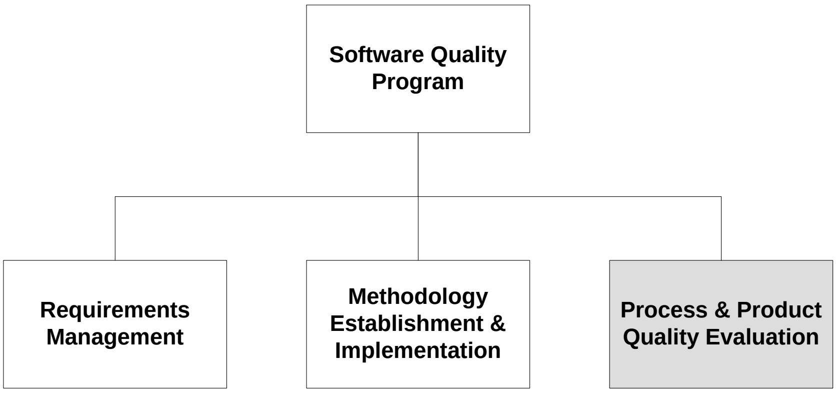

Software Quality Program
Requirements Management
Methodology Establishment & Implementation
Process & Product Quality Evaluation

Figure 1.2: Software Quality Elements [Kenett and Baker, 1999]

evaluation [Kenett and Baker, 1999, p. 4]. Figure 1.2 summarizes this observation. The software quality evaluation component is aimed at evaluating products (both in-process and at completion), activities and processes (for optimization and compliance with standards), and methodologies (for appropriateness and technical adequacies).

In addition to the conventional subjective evaluation methods such as interviews, surveys, and inspections, software measurement is a tool for objective evaluation in software engineering. Kan [Kan, 2002] has analyzed the role of measurement in a variety of perspectives of software quality engineering. It is not very recent at all that the application of software measurement as an evaluation technique has been advocated by researchers and realized by practitioners. Software measurement is part of almost all key areas within IEEE's Software Engineering Body of Knowledge [Abran et al., 2004]. It has itself now become a well established research area with the availability of dedicated measurement frameworks and processes [Zuse, 1998][Dumke, 2005] [Dumke et al., 2005][Dumke and Ebert, 2007] [Ebert et al., 2004][iso, 2007]. With the application of software measurement we are better able to perform a cost benefit analysis of software tools, methods, and processes.

But despite all these advancements and envisaged benefits, software measurement does not seem to have fully penetrated into industrial practices. It still seems to reside in the minds and works of researchers while industry and practitioners, who are overwhelmed by the pursuit of immediate business goals constrained by time and cost limits, tend to pay less attention to it than it deserves. As far as the use of software measurement for quality evaluation is concerned, Höfer and Tichy [Höfer and Tichy, 2007] have observed that its application has been as yet limited since most software metrics are still being used mainly for cost estimation.

1.3 Evaluation in Software Testing

Software testing is a complex and critical task among software development activities. Figure 1.3 presents a visualization of different elements that are involved with and support the task of software testing. Testing methods and techniques, tools, standards, measurements, and empirical knowledge etc. are the main elements of interest in the software testing domain.

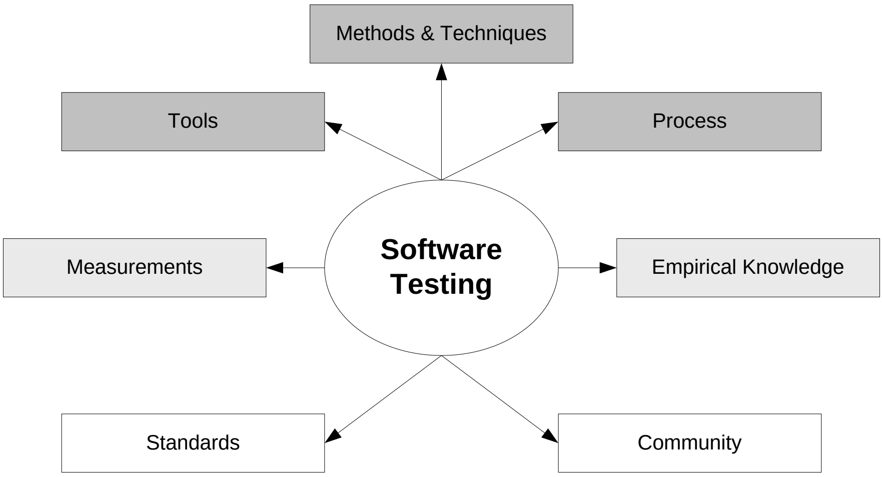


Figure 1.3: Software Testing Elements of Interest

The area of software testing research is almost as old as the software engineering itself. It has largely been driven by quest for quality software. Historically speaking, an overwhelming portion of software testing research has focused on test case design, static and dynamic testing techniques, problem-centered testing approaches such as for object-oriented design or for embedded systems software, testing tools, and designing effective testing processes. A few articles [Harrold, 2000][Taipale et al., 2005][Bertolino, 2007] have discussed about past and future research trends in software testing. It has been observed that the research on fundamental testing issues such as testing methods, tools, and processes has somewhat matured (however, the same is not true for emerging technologies such as for example service-oriented architectures etc.). Our focus is now more on advanced and finer problems such as establishing empirical baseline on testing knowledge, test process improvement, standardization, demonstrating effectiveness of testing methods, tools, and processes, and on test automation. Table 1.1 summarizes lists of active research issues in software testing mentioned in latest literature on testing research.

One of these open and rather neglected issues is evaluation of various testing artifacts. The role of measurement in software testing has been exemplified by Munson [Munson, 2003] with various examples. He maintains that evaluating the test activities will give great insight into the adequacy of the test process and the expected time to produce a software product that can meet certain quality standards. But the first question is which testing artifacts can be and should be evaluated? A study of the list of topics over software testing given in IEEE's Software Engineering Body of Knowledge [Abran et al., 2004, p. 5-2] and in an initial work on Testing Body of Knowledge [Harkonen, 2004, p. 26] can give us an answer. The topics contained therein consist mainly of test levels, test techniques, test measures, test process, and test tools. Therefore, *test techniques* are one element of evaluation, we need to know how much effective is our technique in terms of effort and defect finding capability. *Test tools* are another target of measurement. We need to assess and analyze our tools themselves for their efficiency. *Test process* is perhaps the most substantial element to evaluate since evaluation itself is the first step in improving the test process. By evaluating test process

Table 1.1: Research Issues in Software Testing

Reference	**Issues Highlighted**
[Harrold, 2000]	Testing component-based systems *Test effectiveness* *Creating effective testing processes* Testing evolving software
[Abran et al., 2004, p. 5-3]	Test selection *Test effectiveness* Test oracles Testing for defect identification Testability Theoretical and practical limitations of testing
[Taipale et al., 2005]	Testing automation Standardization *Test process improvement* Formal methods Testing techniques
[Bertolino, 2007]	*Test process improvement* *Test effectiveness* Compositional testing *Empirical body of evidence* Model-based testing Test oracles Domain specific test approaches

we try to find out how much effective and efficient is it in terms of money, time, effort, and defect identification and removal.

1.4 Structure of the Report

Starting with a short overview of status of evaluation in software engineering and software testing in the current chapter, the report dedicates three chapters to analyze evaluation works relative to each of the three core elements of evaluation in software testing, i.e. *process*, *techniques*, and *tools*. Chapter 2 reviews test process in different paradigm contexts, summarizes existing test process descriptions, and analyzes strengths/weaknesses and capabilities/limitations of current test process evaluation models and methods. Chapter 3 and 4 present similar works related to testing techniques and tools, respectively. A summary of findings and future research directions in this context are discussed in chapter 5.

2 Test Processes: Basics & Maturities

With fast growing size of software systems, numerous complexity issues and wealth of professional practices, software development is no longer a programmer oriented activity. Process based software engineering methodology has evolved out of this chaos as a systematic approach that can handle issues related to development methodology & infrastructure, organization, and management of software development activities. Software processes has become a key research area in the field of software engineering today.

Being critical to the quality of the developed product, testing activities occupy major portion of the software development process and involve heavy expenses, development effort, and time. Owing to their important role, testing related activities and issues are generally seen as a separate software testing process. Similar to the two levels of studying software engineering processes as mentioned in IEEE SWEBOK [Abran et al., 2004, p. 9-1], the test process can also be studied at two levels. The first level refers to technical and managerial activities that are carried out to verify and validate development artifacts throughout the software development lifecycle. The second is the meta-level which involves the definition, implementation, assessment, measurement, management, change, and improvement of the test process itself. This chapter mainly concerns with this meta-level description of the test process which applies to all kinds of testing methods and domains.

2.1 Test Process Fundamentals

Different kinds of meta-level descriptions of test process exist. It is usually described as generic process phases or as a series of various levels of testing. It is commonly studied as an organization of testing techniques [Everett et al., 2007], as a quality assurance approach [Tian, 2005][Lewis, 2004], or a means to managing different kinds of testing activities [Pol et al., 2002]. A well established test process can bring about many benefits to all stakeholders. According to Perry [Perry, 2006] these advantages include,

- *Testing is consistent*: Following test process matures the practices. Successful practices can be re-implemented for other projects which reduces variability of activities and increases our confidence.

- *Testing can be taught*: In a heroic testing where no process exists, testing is mainly an art confined to a master tester. Breaking testing into processes makes it understandable and teachable.

- *Test processes can be improved*: By using processes we can identify ineffective areas and activities. Such deficiencies can be removed to make testing cost-effective and improve product quality.

- *Test processes become manageable*: When a process is in place, it can be managed. If it is not, then things are being done in an ad-hoc manner where there can be no management.

A generic very high level structure of test process activities has been given by Tian [Tian, 2005, p. 68]. He divides test process into three main groups of test activities which are,

- *Test planning and preparation*, which set the goals for testing, select and overall testing strategy, and prepare specific test cases and the general test procedures.

- *Test execution* and related activities, which also include related observation and measurement of product behavior

- *Analysis and follow-up*, which include result checking and analysis to determine if a failure has been observed, and if so, follow-up activities are initiated and monitored to ensure removal of the underlying causes, or faults, that led to the observed failures in the first place.

Figure 2.1 summarizes these common test process activities.

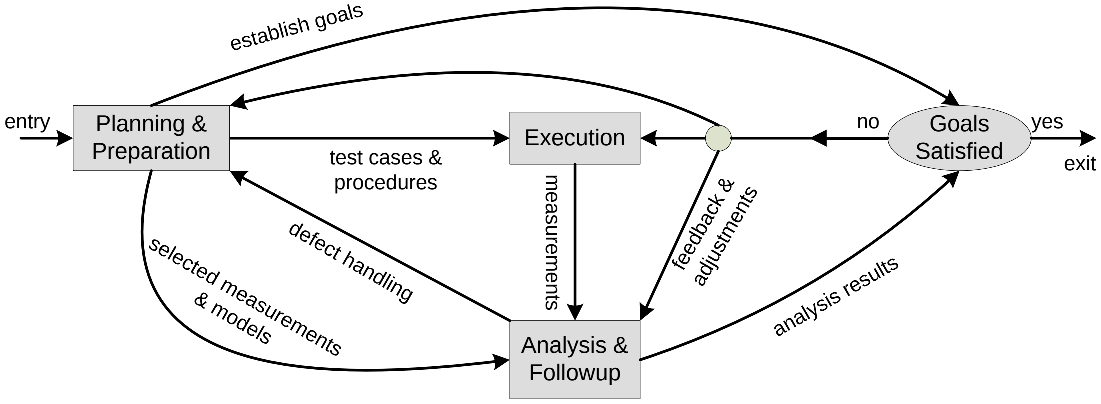


Figure 2.1: Generic Structure of Testing Process [Tian, 2005]

Scope of Testing in Software Process: Testing is mainly a support activity of the development process. It serves as an evaluation technique for the software development artifacts as well as a tool for quality assurance.

- Guide to the Software Engineering Body of Knowledge (SWEBOK) [Abran et al., 2004, p. 11-1] lists testing related processes inside software quality knowledge area. It describes software quality management processes as comprising software quality assurance, verification, validation, reviews, and audits.

- Jeff Tian in [Tian, 2005, p. 27] describes verification, validation and testing as part of quality assurance.
- IEEE/EIA 12207 standard [iee, 1998c] organizes software life cycle processes into three categories, namely primary life cycle processes, supporting processes, and organizational life cycle processes. Quality assurance, verification, validation, joint reviews, and audit are listed inside supporting life cycle processes, while quality assurance process may in turn make use of results of other supporting processes such as verification, validation, joint reviews, and audit.

Figure 2.2 gives a visual representation of context relationship among software quality engineering, software quality assurance and software testing discussed above.

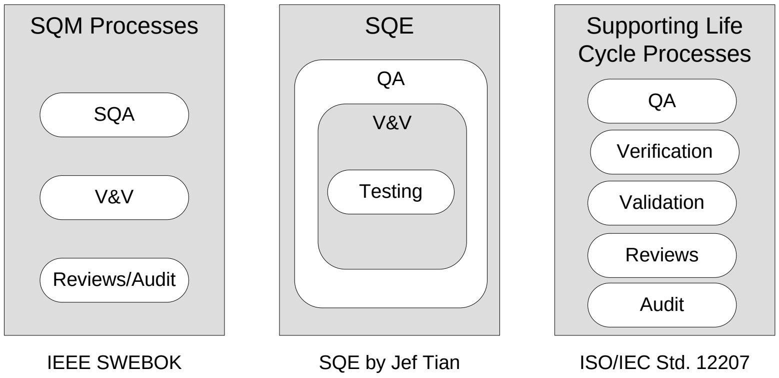


Figure 2.2: Some Context Descriptions of Software Testing

2.1.1 Test Process Contexts

The field of software engineering possesses a number of dimensions. On one axis is the development methodology. Here we refer to methodology as the software development life cycle followed, whether it is based on traditional waterfall or an iterative approach. The second axis refers to software engineering technologies which have evolved in the form of assorted programming paradigms and software architectures. We write our programs using structured programming, object-oriented or aspect-oriented programming approaches or others and design our software systems using distributed, component-based or service-oriented architectures etc. On the third side we have the kind of application system to which our software will be serving. Examples are information systems, embedded systems, or communication systems etc. Figure 2.3 visualizes these dimensions. Each of these SE dimensions involve peculiarities which pose special requirements on software testing. Although a meta-level generic testing process may fit any of these contexts, these three dimensions will warrant some corresponding testing considerations at lower levels of test process abstractions.

For example, testing activities follow a different path in a waterfall kind of development life cycle in comparison to iterative approaches. Testing may pose different requirements in case of component-based systems and in service-oriented architectures (SOA). For component-based systems unit testing, integration testing and performance

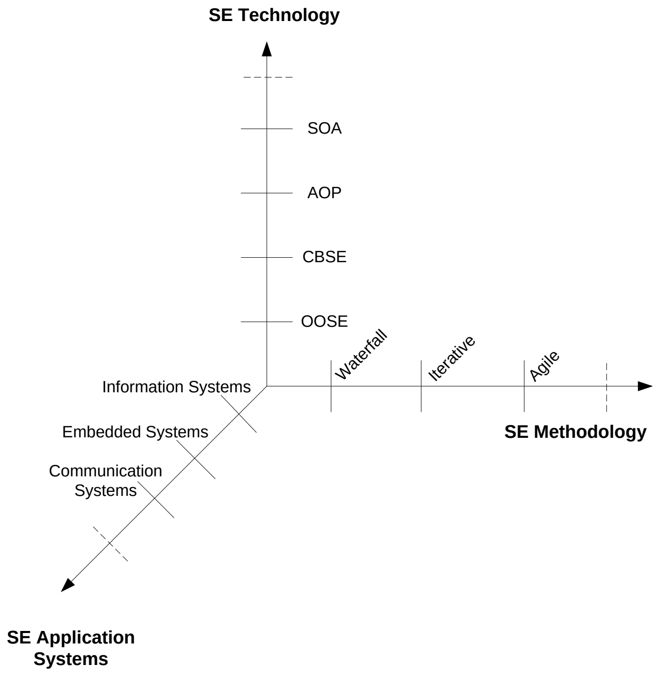


Figure 2.3: Software Engineering Dimensions

testing are the main concerns. On the other hand, SOA poses different quality concerns [O'Brien et al., 2007] and new testing challenges [Zhu, 2006]. Testing techniques and approaches for communication systems, embedded systems and business information systems will also certainly differ. Alongside generic test processes, some custom test processes also exist that take care of some of these domain specific requirements and constraints.

2.1.2 Research over Test Process

Three main issues concerning test process research are: definition or modeling, evaluation, and improvement.

The *definition of the test process* refers to the definition of the processes as models, plus any optional automated support available for modeling and for executing the models during the software process (derived from [Acuña et al., 2001]). This may be in the form of a description of part/whole of test process using a suitable process modeling language. Examples include model-based testing approaches. Another way to define a test process is to give an activity based description of the process aimed at activity management. Examples include well known testing standards and other generic and domain-specific test process descriptions.

Test process evaluation is a systematic procedure to investigate the existence, adequacy, and performance of an implemented process system against a model, standard, or benchmark (derived from [Wang and King, 2000, p. 42]). It is the investigation of the current state of the process with a view of finding necessary improvement areas. Process evaluation is typically performed prior to any process improvement initiative. Test process evaluation and improvement is motivated by a concern for cutting on testing costs and improving product quality.

Test process improvement is a systematic procedure to improve the performance of an existing process system by changing the current process or updating new processes in order to correct or avoid problems identified in the old process system by means of a process assessment (derived from [Wang and King, 2000, p. 42]). In parallel with the concern for software process improvement, test process improvement also continues to be a major research direction within software testing. It has been ranked by Taiple [Taipale et al., 2005] as one of the top three important issues in software testing research.

In most cases a solution may address more than one of the above mentioned three issues at the same time. For instance process evaluation and improvement are mutually connected issues of software test process. Any software process improvement initiative needs first an evaluation of the current level of performance of the process. Any process evaluation exercise should eventually follow an identification of and suggestions over most important process improvement areas. Therefore, test process evaluation and improvement will be reviewed in the same section in this text.

2.2 Test Process Definition & Modeling

Existing test process modeling approaches include some *empirical and descriptive* and *formal and descriptive* process models. According to Wang and King [Wang and King, 2000, p. 40] an empirical process model defines an organized and benchmarked software process and best practices, a descriptive model describes "what to do" according to a certain software process system, while a formal model describes the structure and methodology with an algorithmic approach.

2.2.1 Generic Test Process Descriptions

An activity-based description of the software test process has been given by Perry [Perry, 2006, Ch. 6]. He divides the test process intro seven steps. The process has been designed to be used by both developers and an independent test team. Since the details of the process activities are very generic in nature, the process must be customized by organization before its actual use.

Figure 2.4 gives an overview of the proposed process. It follows the V concept of development/testing. The seven steps as given in [Perry, 2006, p. 157] are being summarized below.

1. **Organizing for testing:** This is a kind of preparation step which is aimed at defining scope of the testing activities and responsibilities of whoever will be involved in testing process. Furthermore, the development plan must be analyzed for completeness and correctness which is the basis for the next step of test plan development.

2. **Developing the test plan:** After the preliminary steps, a test plan must be developed that precisely describes testing objectives. A test plan will mention exactly

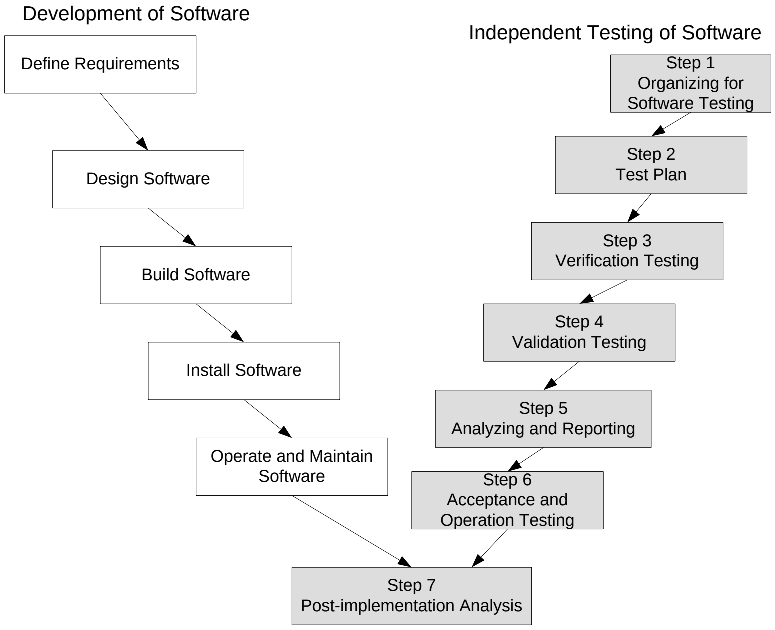

Development of Software
Define Requirements
Design Software
Build Software
Install Software
Operate and Maintain Software
Independent Testing of Software
Step 1 Organizing for Software Testing
Step 2 Test Plan
Step 3 Verification Testing
Step 4 Validation Testing
Step 5 Analyzing and Reporting
Step 6 Acceptance and Operation Testing
Step 7 Post-implementation Analysis

Figure 2.4: V-Diagram for Seven Step Test Process [Perry, 2006]

how and what kinds of testing activities will be performed. Possible risks should also be identified at this step.

3. **Verification testing:** The purpose of this step is verify activities and products of each of the design and development process to ensure that software is being constructed correctly. This will enable an early detection of defects before development is complete.

4. **Validation testing:** Dynamic testing of the code using the pre-established methods and tools should be performed now. This step should ensure that the software fulfill the stated requirements.

5. **Analyzing and reporting test results:** Test results should be analyzed to compare the developed product with the intended development goals. Results should be reported with the defect reports etc.

6. **Acceptance and operational testing:** A final step is the testing of the software by the actual users. Upon completion of the acceptance testing, the software must once again be testing in the production environment to observe and conflicts or other faults.

7. **Post-implementation analysis:** This step is a kind of post-mortem analysis of the whole testing process. Efficiency and effectiveness of the testing process must be analyzed. This will help us identify lessons learned, and future improvement areas for the test activities.

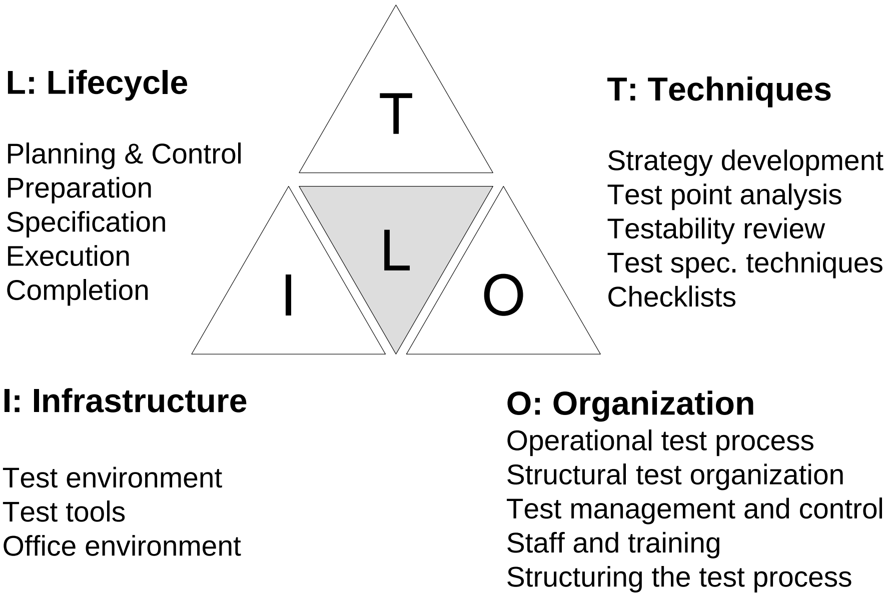


Figure 2.5: Test Management Approach-TMap

2.2.1.1 Test Management Approach-TMap

The Test Management Approach (TMap) has been developed by a Dutch firm Sogeti. A detailed description of the approach can be found in [Pol et al., 2002]. The TMap approach primarily focuses on structured testing and provides answers to the *what, when, how, where*, and *who* questions of software testing [van Veenendaal and Pol, 1997]. Figure 2.5 gives an overview of TMap. It is founded on four cornerstones;

L a development process related life cycle model for the testing activities

O solid organizational embedding

I the right resources and infrastructure

T usable techniques for the various testing activities

Relating to each of these four aspects, TMap provides guidelines on objectives, tasks, responsibilities, deliverables and related issues. For example, the life cycle model (L) contains a sequence of testing activities which operate in parallel to the software development life cycle phases.

2.2.1.2 Drabick's Formal Testing Process

Drabick [Drabick, 2003] presents a task-oriented process model for formal testing intended for use on medium-to-large software-intensive programs. The model provides a concise framework of testing tasks to assist test engineers. The author of the approach assumes the model to be helpful in a number of ways, for example,

- Manage defects

- Create efficient test plans
- Provide work breakdown structure for the test engineering function
- Provide a basis for documenting testing processes

The test process model is composed of a collection of Input-Process-Output (IPO) diagrams. Each IPO diagram lists inputs, process names, and relevant outputs. Figure 2.6 gives structure of the level 0 model for the formal testing. The description is very primitive in nature at this level. This level of detail is not much meaningful and is meant to present only a top-level picture of the test process.

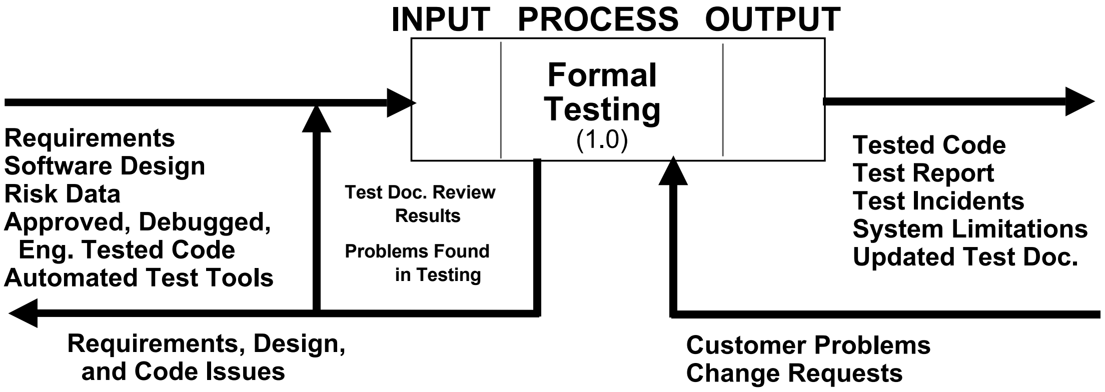


Figure 2.6: Drabick's Formal Software Test Process-Level 0 IPO Diagram [Drabick, 2003]

Figure 2.7 expands the level 0 description of the model into several sub-processes which are listed below. The proposed model further drills down to level 2 and 3 for each of these processes (which are not given here for the sake of brevity).

1. Extract test information from program plans
2. Create test plan
3. Create test design, test cases, test software, and test procedures
4. Perform formal test
5. Update test documentation

Although the process model contains several useful details of testing activities, yet it speaks nothing about the evaluation of the process itself. It provides no mechanism of evaluating how good the process has been performed or any other form of assessing effectiveness or efficiency of the activities performed.

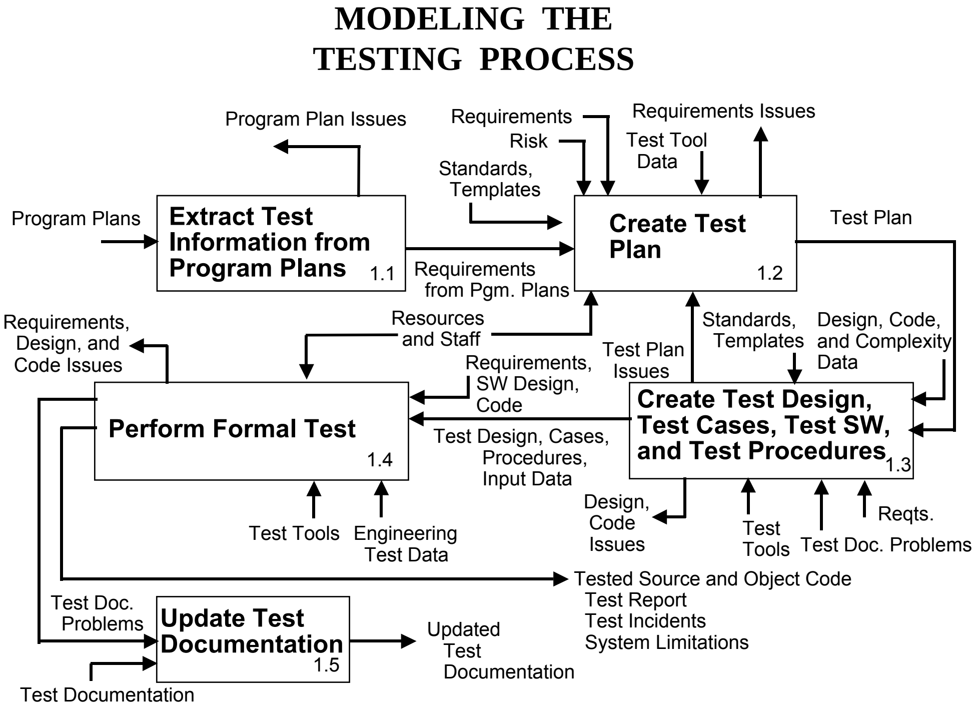


Figure 2.7: Drabick's Formal Software Test Process-Level 1 IPO Diagram [Drabick, 2003]

2.2.1.3 Test Driven Development

Agile software development is a conceptual framework for software development that promotes development iterations, open collaboration, and adaptability. Agile methods are development processes that follow philosophies of Agile manifesto and principles. Some examples of these methods include Extreme Programming (XP), Adaptive Software Development (ASD), Scrum, and Feature Driven Development (FDD) etc. Agility, change, planning, communication, and learning are common characteristics of these methods .

Extreme Programming (XP) is a well known and probably the most debated of the Agile methods. Two of the twelve practices of XP include *Test First* and *Refactoring*. The test first principle requires that automated unit tests be written before writing a single line of code to which they are going to be related. Test Driven Development (TDD) [Beck, 2002] has evolved from this test first principle. Although TDD is an integral part of XP but it can also be used in other development methods.

TDD is not a not a testing technique nor a testing method or a process, it is only a style of development. Under this approach software evolves through short iterations. Each iteration involves initially writing test cases that cover desired improvement or new functionality. Necessary code is then implemented to pass these tests and the software is finally refactored to accommodate changes. Test-driven development cycle consists

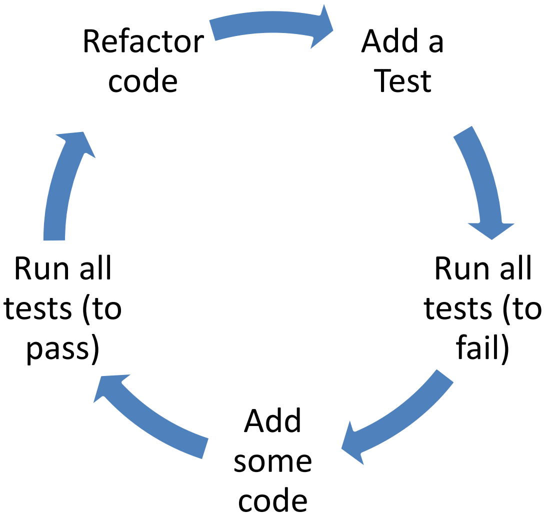


Figure 2.8: Test-driven Development Cycle

of following sequence of steps; [Beck, 2002]

- **Quickly add a test:** A simple test is written as the first step which covers some aspect of functionality of code.
- **Run all tests and see the new one fail:** Running the test cases in absence of required code should essentially fail. This validates that the test harness is working correctly and that the new test does not mistakenly pass without requiring any new code.
- **Make a little change:** The next step is to implement some code that is just enough to pass the existing tests. This is meant to incrementally add functionality to developed code.
- **Run all tests and see them all succeed:** If all tests now pass, the programmer can be confident that the code meets all the tested requirements.
- **Refactor to remove duplication:** Refactoring is the process of making changes to existing, working code without changing its external behavior. This step removes cleans up the code and any duplication that was introduced getting the test to pass.
- **Repeat:** This test-code-refactor cycle is repeated which leads to an evolution of the whole program where the program-units are developed gradually.

Figure 2.8 summarizes the TDD cycle. As in other conventional development and testing practices, testing under TDD is not done in a linear fashion. The continuous evolution and feedback that is obtained from running tests makes this method circular. Since its inception, a number of techniques and tools have been developed that support TDD style [Astels, 2003].

Improved quality, testability, and extensibility and other benefits are believed to be associated with TDD style of development. Some empirical works exist that have attempted to validate some of these claimed benefits [Canfora et al., 2006][Siniaalto, 2006]. However certain TDD is limited in certain aspects too. First, it concentrates on automated unit tests to build clean code. It is a fact that not all tests can be automated, for example user interface testing. Second in database applications and those involving different network configurations full functional tests are a necessity. Test-first approaches for these kinds of applications are still missing. TDD's lack of proper functional specifications and other documentations also limit this style to small projects. There are some social factors such developer's attitude and management support will certainly be a hurdle in adoption of this evolutionary approach.

2.2.1.4 Independent Verification & Validation

Zero defect software is a highly sought goal for some particular kinds of safety critical and complex large applications. Sometimes managerial commitments, financial constraints and developer's or tester's bias may cause adverse affects on testing and software quality compromises. According to IEEE independent verification and validation (IV&V) refers to the verification and validation performed by an organization that is technically, managerially, and financially independent of the development organization. But whether IV&V differs from V&V in more than just the independence of its practitioners is still open to debate [Arthur et al., 1999].

IV&V activities have been found to help detect faults earlier in the software development life cycle, reduce the time to remove those faults, and produce a more robust product [Arthur et al., 1999]. The advantages of an independent V&V process are many. In particular, the independence in V&V [Arthur and Nance, 1996],

- provides an objective assessment of the product during its creation,
- adds a new analytical perspective not present in the development environment,
- brings its own set of tools and techniques to bear on ensuring development accuracy and validity,
- introduces "intermediate" users of the system who serve as "beta testers" before the product goes to market, and
- significantly enhances testing and the discovery of design flaws and coding errors.

Several software companies offer IV&V services. NASA's IV&V Facility is a well-known IV&V service provider for NASA's critical projects and missions. Analysis of IV&V approaches for different domains such as simulation and modeling and object-oriented software applications has been performed.

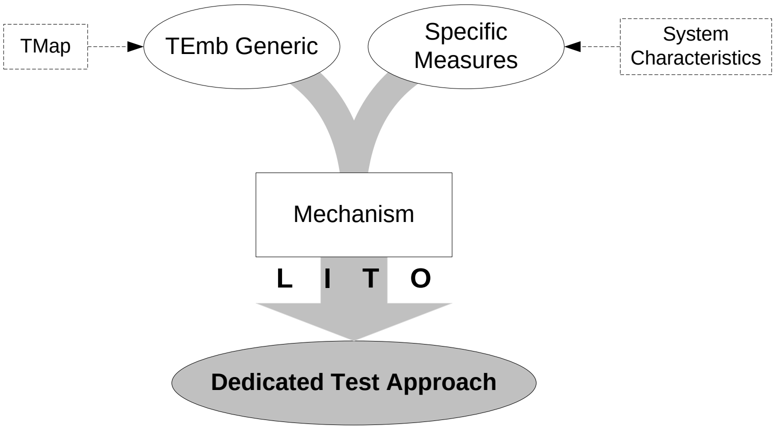


Figure 2.9: TEmb:Test Process for Embedded Systems [Broekman and Notenboom, 2003]

2.2.2 Domain Specific Test Processes

A very wide variety of software applications are being developed today, for example those for distributed systems, communication systems, and embedded systems etc. Type of the application domain naturally affects scope and range of software testing involved. Certain techniques and levels of testing may no longer be applicable, and new approaches to testing may be required. Testing activities and process will also be affected. The next two sections will review testing process for embedded systems and service-oriented applications as well-known examples which require specialized testing requirements.

2.2.2.1 Test Process for Embedded Software

Many different types of embedded systems exist today such as mobile phones, electrical home appliances, railway signal systems, hearing aids and other health care systems, missile guidance systems, satellites, and space shuttles. Zero defect software is needed for such systems since a failure can cause human lives or extremely huge financial losses. Within this context, testing of embedded software becomes very complex and poses much more challenges and requirements on testing than that of other common software applications.

Many different kinds of techniques and tools have been developed to answer specific testing concerns of embedded softwares. Instead of discussing individual techniques we review here a testing method which covers a wider perspective of embedded software in comparison to specific techniques or tools. The method is called TEmb. TEmb provides a mechanism for assembling a suitably dedicated test approach from the generic elements applicable to any test project and a set of specific measures relevant to the observed system characteristics of the embedded system [Broekman and Notenboom, 2003, Ch. 2]. This method actually adapts the concepts of TMap [Pol et al., 2002] approach to the embedded software domain. Figure 2.9 gives an overview of the TEmb method.

The generic elements of the method involve descriptions of *lifecycle*, *techniques*, *infrastructure*, and *organization* issues. The second part of the method involves applying *measures* specific to the system context based on the analysis of *risks* and *system characteristics*. Example of these specific measures include specific test design techniques, system modeling, dedicated test tools and lifecycle etc [Broekman and Notenboom, 2003, p. 18].

2.2.3 Formal Approaches

Wang and King [Wang and King, 2000, p. 40] define a formal process model as a model that describes the structure and methodology of a software process system with an algorithmic approach or by an abstractive process description language. Formal approaches to software process have been variably applied. Dumke et al. [Dumke et al., 2006a] mention few of such approaches. The same concept has been used in the domain of testing process. The next two sections explain these approaches.

2.2.3.1 Model based Testing

A major portion of software testing costs is associated with test case related activities. Test case generation consumes resources such as for their planning, design, and execution. Manual design and execution of test cases is a tedious task. Therefore, automation of test case generation and execution could be an interesting mechanism to reduce the cost and effort of testing. Automatic *execution* of tests is offered by many automated test tools. Model based testing (MBT) [Utting and Legeard, 2006] takes a step forward to automate the *design* process of test cases.

MBT involves creating an abstract model of the system under test which is mostly based on functional requirements. Then a test tool automatically generates test cases from this model of the system. A direct benefit is that overall test design time is reduced and a variety of test cases can be generated from the same model simply by changing test selection criteria. MBT is supposed to offer many benefits such as shorter schedules, lower cost and effort, better quality, early exposure of ambiguities in specification and design; capability to automatically generate many non-repetitive and useful tests, test harness to automatically run generated tests, and convenient updating of test suites for changed requirements [El-Far and Whittaker, 2001]. Utting and Legeard [Utting and Legeard, 2006, p. 27] divide MBT into following five steps,

- **Model:** The very first step is to create an abstract model which describes behavior of the system under test (SUT). This model is abstract in the sense in that it mostly covers key aspects of the SUT. Some design language or a test specification language must be used to create this model. Unified Modeling Language (UML), TTCN-3 [1], or Test Modeling Language (TML) [Foos et al., 2008] can be used for the purpose. Hartman et al. [Hartman et al., 2007] provide a survey of test modeling languages which relevant to this step.

- **Generate:** The next step is to generate abstract tests from the model. An automated test case generator tool can be exploited at this step. To reduce the almost infinitely possible test cases, a test selection criteria must be used. In addition

[1] http://www.ttcn-3.org/

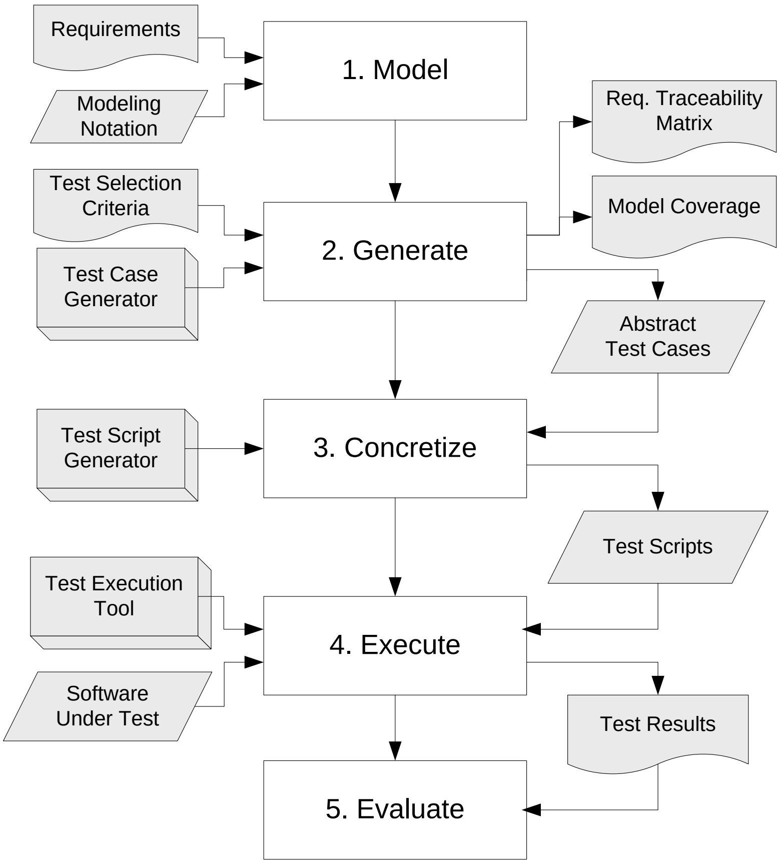


Figure 2.10: Model-based Testing Process

to a set of abstract test cases, this step sometimes also produces a requirements traceability matrix and a model coverage report.

- **Concretize:** The abstract test cases from the previous step cannot be executed directly on the SUT. They must be transformed into executable concrete form which is done under this step. A test script generator tool may be used for the purpose.
- **Execute:** This step executes the concrete steps over the system under test (SUT) with the help of a test execution tool. The step produces the final test results. With *online testing*, the above three steps are merges and tests are executed as they are produced. In case of the *offline testing*, the above three steps will be performed as described.
- **Analyze:** The final step is to analyze the test results. Actual and expected outputs are compared and failure reports are analyzed. The step also involves deciding whether to modify the model, generate more test cases, or stop testing.

Figure 2.10 gives a detailed description of the MBT process with necessary inputs and outputs of each step.

Hundreds of MBT approaches have be developed to date. However, they are not aimed at covering all testing aspects. For example MBT techniques mainly aimed at

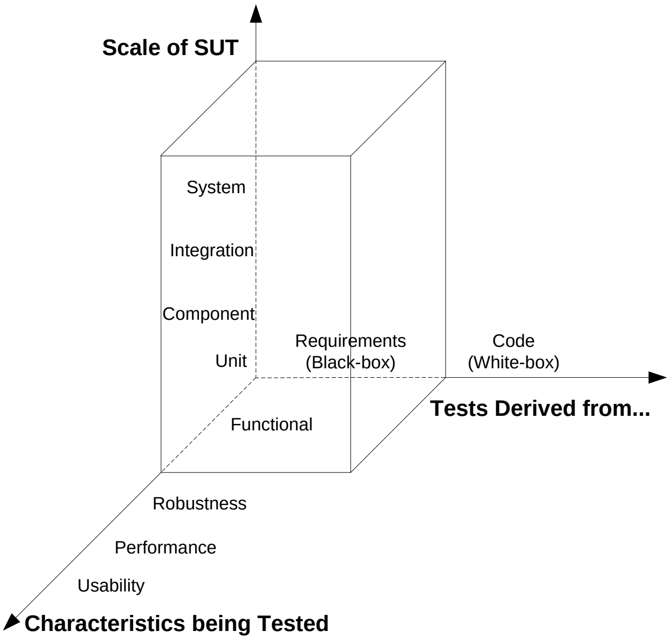


Figure 2.11: Scope of Model-based Testing [Utting and Legeard, 2006]

functional testing since test cases are derived from functional specification of the system. Only in very few cases have the MBT approaches been used for testing some non-functional characteristics. Furthermore, MBT is a kind of black-box approach since the system model has been derived from the behavioral descriptions. However, MBT can be applied at any testing level (although it has mostly been applied for system level tests). Figure 2.11 summarizes the scope of MBT with reference to different testing aspects.

A comprehensive characterization of these techniques has been given by Neto et al. [Neto et al., 2007]. MBT techniques differ by behavioral model, test generation algorithm, test levels, software domain, or level of automation etc. Choice of a particular MBT approach out of the many can influence efficiency of the overall test process.

2.2.3.2 Cangussu's Formal Models

A mathematical model of a software process attempts to describe its behavior and provides a feedback mechanism which guides the managers in adjusting model parameters to achieve desired quality objectives. The generic procedure to select, adopt and apply these kinds of models as quoted by Apel [Apel, 2005] is outlined below.

1. Postulate general class of models
2. Identify model to be tentatively entertained
3. Estimate model parameters
4. Perform diagnose checking (model validation)
5. Use model for prediction or control

Several mathematical models of software test process have been developed by Cangussu et. al[Cangussu, 2002][Cangussu, 2003][Cangussu et al., 2003a] These mathematical models attempt to predict some aspect of the software test process (with special focus on system test phase) such as effort, schedule slippage, failure intensity or effect of learning etc. Most of these approaches followed a feedback control mechanism as outlined in the figure 2.12.

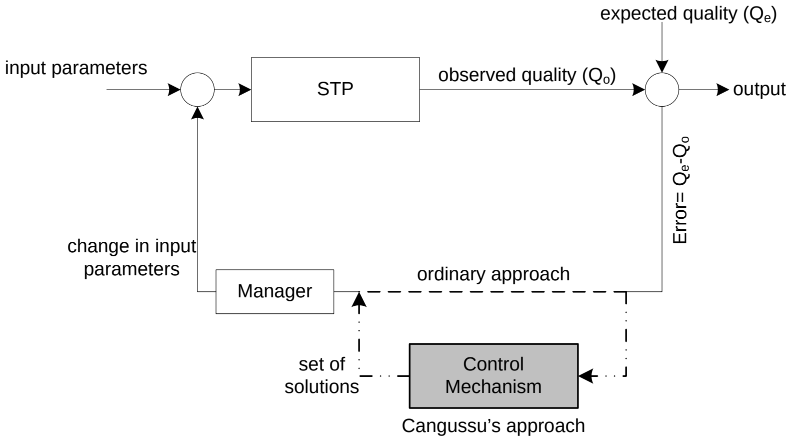


Figure 2.12: Cangussu's Approach of STP Models [Cangussu, 2002]

Now we briefly describe each of Cangussu's approaches one by one.

- **State Variable Model** [Cangussu et al., 2000]
  This model uses the theory of state variables to capture the dynamic behavior of the software test process by focusing on time and effort required to debug the software. It then applies feedback control for adjusting the variables such as work force and quality of the test process to improve the test process performance, and meeting the deadlines. This model has been validated with data from two large industrial projects.

- **A State Model** [Cangussu et al., 2001a]
  This model attempts to predict completion time and cost to perform software test process. The model provides an automated method for parameter identification. The closed-loop feedback mechanism consisting of determination (based on adjustment of different parameters) of minimum decay rate needed to meet management objectives guides the managers to correct deviations in the software test process.

- **Feedback Control Model** [Cangussu et al., 2001b]
  Feedback control model is quite similar to formal and state models discussed above. It differs only in control variables which in this case are product reliability and failure intensity. These variables are calculated at specific checkpoints within the software test process and result is fed back to the controller to adjust model parameters to meet desired process objectives.

- **A Formal Model** [Cangussu et al., 2002]
  Current formal model of the software test process is based on the theory of process control. Estimations of the number of remaining errors and schedule slippage are performed at specific checkpoints inside a feedback control structure which helps meet the schedule and quality requirements.

- **Stochastic Control Model** [Cangussu, 2003]
  The stochastic control model is a variation of state variable model and formal model of the software test process discussed above. This model is designed to account for foreseen and unforeseen disturbances and noise in the date collection process. The model has been verified with some simulation results while still needs validation with actual project data.

- **A Quantitative Learning Model** [Abu et al., 2005]
  This model is also derived from the formal model of the software test process described above. This approach investigates the effect of learning behavior and experience to improve the software test process. Prediction process is improved by adjusting different model parameters such as initial knowledge and learning rate. The model has been validated with two large industrial case studies.

Some general aspects of concern about such mathematical models are:

- *Model Validation*: Usually these kinds of models are validated through simulation runs, analytical approaches, or empirical investigations and industrial case studies. The models outlined above have been validated through simulation and same two case studies applied to each of these model evaluations. We still need more empirical studies on these models to highlight any new aspects of model behavior and effect of different model parameters.

- *Prediction Quality*: One of the criticisms of software engineering research is that it ignores evaluation [Zelkowitz and Wallace, 1997]. An evaluation of above mentioned mathematical models involves assessment of their prediction quality. Apel [Apel, 2005] mentions some criteria to evaluate prediction quality of such mathematical models.

  - *Prediction Accuracy* answers the question how accurate is the prediction.
  - *Prediction Distance* determines how far in future does the prediction lie.

  The models mentioned above need to be evaluated in the light of these criteria. The only related evaluation reported by authors in this regard is a sensitivity analysis [Cangussu et al., 2003b] of the state variable model discussed above. This analysis attempts to quantify effects of parameter variations on the behavior of the model such as its performance.

- *Practical Application/Industrial Acceptance*: The mathematical complexity involved in construction and application of such models may be difficult to be handled by process managers who usually do not have enough background in such areas. In this case, a tool encapsulating mathematical procedures may simplify adoption of these models in industry.

2.2.4 Test Process Standardization

This section will present a summary of the works of international standards bodies in the area of software testing. These standards define requirements, methods, processes and practices relevant to the testing area covered by them. Most such standards partially focus some element of the testing process such as some particular level or type of testing with the exception of [iee, 1998a] and [iee, 1998c] which consider a broader range of testing activities at the level of the whole process. Following standards exist in this context;

- **IEEE Standard on Unit Testing** [iee, 1987]: Aimed at providing a standard approach to unit testing, this standard defines inputs, tasks, and outputs to each of the eight activities defined as part of the unit testing process.
- **IEEE Standard on Software Reviews** [iee, 1997]: This standard contains detailed procedures for the five types of reviews. For each review type, it defines input/output, entry/exit criteria, and procedures.
- **IEEE Standard for Software Verification and Validation** [iee, 1998a]: This standard covers a broader perspective of all V&V activities with reference to each of the software life cycle processes as defined in [iee, 1998c]. It defines all kinds of V&V activities alongside details of inputs, outputs, and tasks.
- **British Computer Society Standard for Software Component Testing** [bcs, 2001]: It concerns with test case design and test measurement techniques, and procedures for testing software components. The standard also addresses evaluation of these techniques.
- **ISO/IEC Standard for Information Technology-Software life cycle processes** [iee, 1998c]: Although this standard mainly covers complete life cycle process for software development, it also refers to verification, validation, review, and audit process as *supporting life cycle processes* and defines activities for each of these processes.

2.3 Test Process Evaluation & Improvement

Evaluation theory [Ares et al., 1998] (figure 2.13) defines six primary elements of any process evaluation approach. These elements are target, criteria, reference standard, assessment techniques, synthesis techniques, and evaluation process. The relationships among these elements are mentioned in figure 2.14. Existing software process and the test process evaluation approaches can be framed inside this structure for comparison and purpose of identifying missing elements.

Evaluation and improvement of software test process bears many similarities with and borrows common concepts from that of the software process. A huge number of assessment and measurement techniques for generic software processes have been developed over the years. Few examples involving explicit process measurement are [Dumke et al., 2004][Schmietendorf and Dumke, 2005]. Therefore, prior to

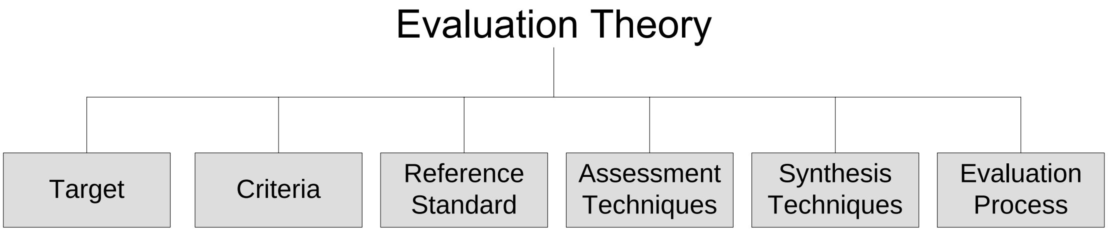


Figure 2.13: Components of Evaluation Theory

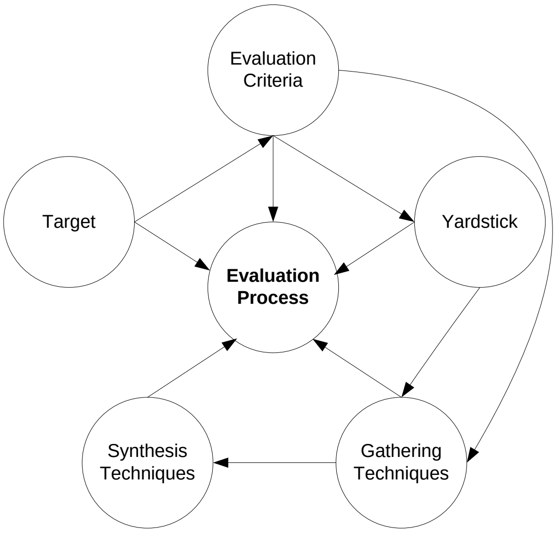


Figure 2.14: Components of Software Process Evaluation and Interrelationships

discussing individual test process evaluation, we should present a broad picture of these available approaches in comparison to existing software process quality evaluation and improvement models. Surveys of current software process quality models have been given in [Tate, 2003] [Komi-Sirviö, 2004, Ch. 3] while some future research directions in test process evaluation and improvement have been discussed by Farooq and Dumke [Farooq and Dumke, 2007b]. Table 2.1 compares existing test process evaluation approaches in comparison with those for generic software processes.

2.3.1 Qualitative Approaches

Most test process evaluation approaches have been qualitative in nature. The first well known model of this kind is Testing Maturity Model (TMM) which was introduced in 1996. It was followed by Test Process Improvement (TPI) Model and Test Improvement Model (TIM) both in 1997. Two later approaches were Metrics-based Verification & Validation Maturity Model ($MB - V^2M^2$) and Test Process Assessment Model (TPAM). The latest development in this direction is the Test Maturity Model Integrated (TMM*i*). Figure 2.15 summarizes time-line of these test process evaluation models.

TIM [Ericson et al., 1998], and $MB - V^2M^2$ [Jacobs and Trienekens, 2002] appear to have vanished from literature probably due to their insignificance or incompleteness. These two models along with TPAM [Chernak, 2004] will be ignored here from fur-

Table 2.1: Software Process vs. Test Process Research

Model Type	Software Process	Test Process
Management	Deming's Cycle QIP IDEAL Model ISO 15504 Part 7	TMap
Best Practices	CMMI Bootstrap SPICE ISO 9000-3	TMM TPI TMM*i* IEEE Std. V&V IEEE Std. Unit Testing
Measurement	SPC GQM PSP	Cangussu's Mathematical Models
Product Quality	ISO/IEC 25000 IEEE Std. 1061	-
Knowledge Management	Experience Factory (EF)	-

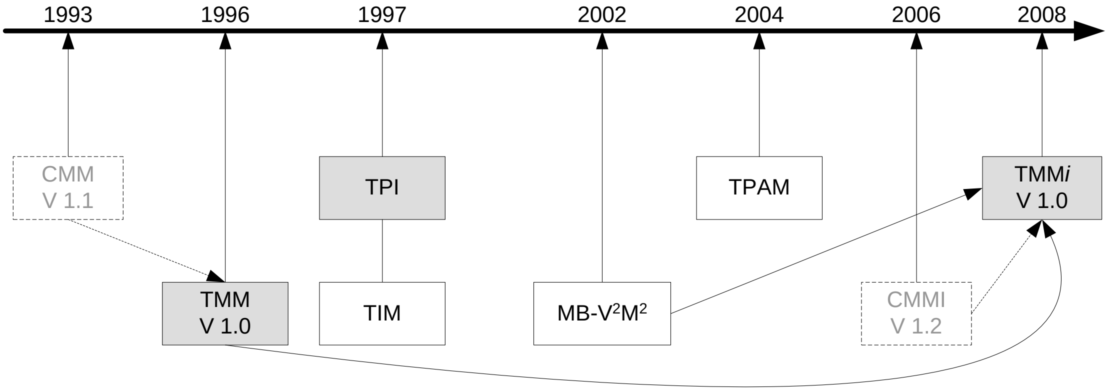


Figure 2.15: History of Test Process Assessment Models & Dependencies

Table 2.2: Comparison of Test Process Assessment Models

Model	**Dependency**	**Approach**	**Scope**
TMM Testing Maturity Model	CMM	Implicit	General
TPI Test Process Improvement	TMap	Implicit	Structured Testing
TMM*i* Test Maturity Model Integrated	CMMI	Implicit	General

ther discussion. Below we present an overview of three *living* test process assessment frameworks.

2.3.1.1 Testing Maturity Model (TMM)

Testing Maturity Model (TMM) was developed by Ilene Burnstein [Burnstein, 2003] to assist and guide organizations focusing on test process assessment and improvement. Since release of its first Version 1.0 in 1996 no further release has appeared. The principal inputs to TMM were Capability Maturity Model (CMM) V 1.1, Gerlperin and Hetzel's Evolutionary Testing Model [Gelperin and Hetzel, 1988], survey of industrial testing practices by Durant [Durant, 1993] and Beizer's Progressive Phases of a Tester's Mental Model [Beizer, 1990]. It is perhaps the most comprehensive test process assessment and improvement model to date.

TMM derives most of its concepts, terminology, and model structure from CMM. This model consists of a set of maturity levels, a set of maturity goals and sub-goals and associated activities, tasks and responsibilities (ATRs), and an assessment model. The model description follows a staged architecture for process improvement models. Relationships between its model elements have been summarized in figure 2.16.

TMM contains five maturity levels which define evolutionary path to test process improvement. The contents of each level are described in terms of testing capability organizational goals, and roles/responsibilities for the key players in the testing process, the managers, developers/testers, and users/clients. Level 1 contains no goals and therefore every organization is at least at level 1 of test process maturity. The maturity goals at each level of the TMM are shown in figure 2.17.

A comparison of TMM with other test process improvement has been performed by Swinkels [Swinkels, 2000]. He concludes that TMM and other test process improvement models of its era appear to complement each other. Another detailed criticism of TMM can be found in [Farooq et al., 2007b] which suggests some improvements to model structure, an update to its assessment model, and an expansion of its process areas.

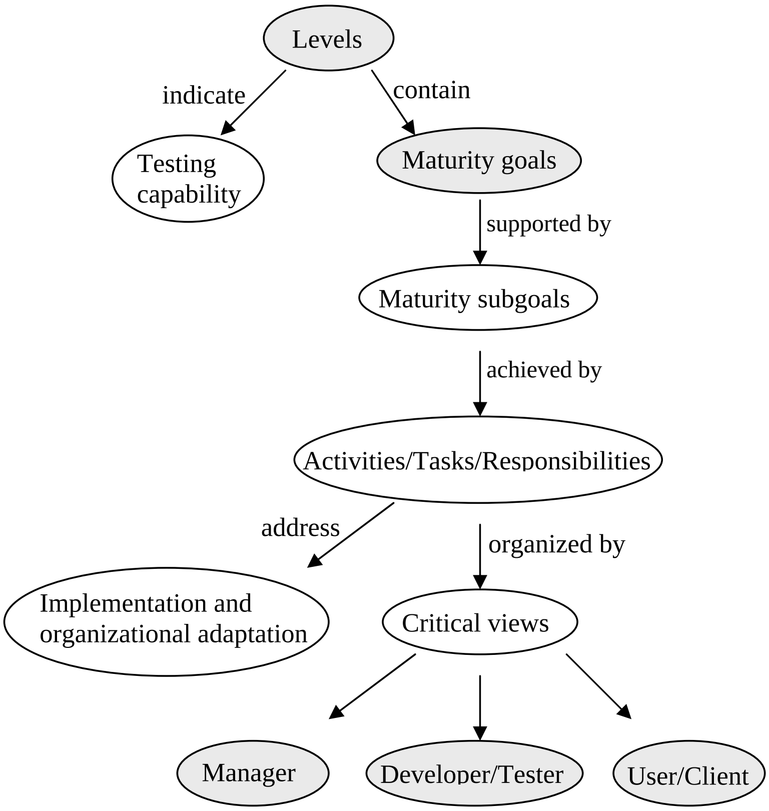


Figure 2.16: Structure of Testing Maturity Model [Burnstein, 2003]

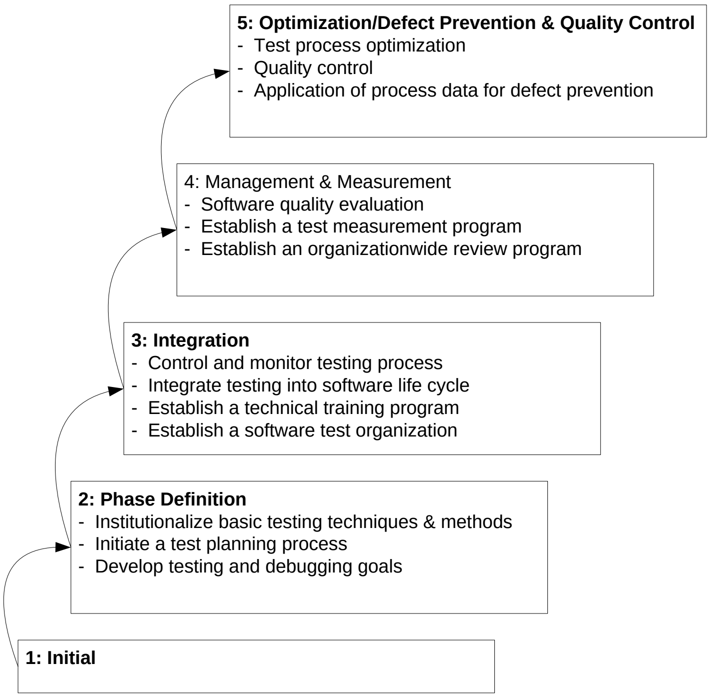

5: Optimization/Defect Prevention & Quality Control
- Test process optimization
- Quality control
- Application of process data for defect prevention
4: Management & Measurement
- Software quality evaluation
- Establish a test measurement program
- Establish an organizationwide review program
3: Integration
- Control and monitor testing process
- Integrate testing into software life cycle
- Establish a technical training program
- Establish a software test organization
2: Phase Definition
- Institutionalize basic testing techniques & methods
- Initiate a test planning process
- Develop testing and debugging goals
1: Initial

Figure 2.17: TMM Maturity Levels

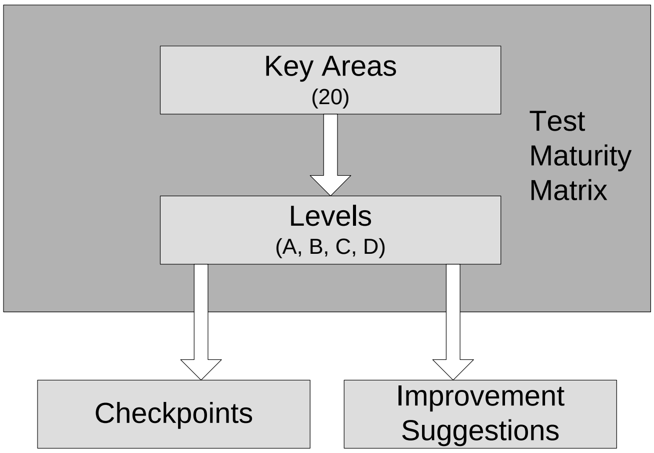


Figure 2.18: Structure of Test Process Improvement (TPI) Model

2.3.1.2 Testing Process Improvement Model (TPI)

Test Process Improvement (TPI) [2][Koomen and Pol, 1999] model is an industrial initiative to provide test process improvement guidelines based on the knowledge and experiences of a large number of professional testers. The first release of this model appeared in 1997. The model has been designed in the context of structured high level testing. It is strongly linked with the Test Management Approach (TMap) [Pol et al., 2002] test methodology.

The model elements include several key areas, each with different levels of maturity. A maturity matrix describes the levels of all key areas. Several checkpoints have been defined corresponding to each maturity level; questions that need to be answered positively in order to classify for that level. Improvement suggestions, which help to reach a desired level, are also part of the model.

The 20 key areas within TPI are organized by means of the four cornerstones of structured testing as defined by TMap: life cycle, organization, infrastructure and techniques. Level of achievement relevant to these key areas is defined through maturity levels. There can be three to four maturity levels for each key area. Each level consists of certain requirements (defined in terms of checkpoints) for the key area. Relationships among TPI model elements are summarized in figure 2.18.

Two world-wide surveys on adoption of TPI by software industry have been reported in [Koomen, 2002][Koomen, 2004]. These surveys reported positive improvements and better control of the testing process by the organizations applying the TPI model. Critical review and comparisons of TPI with other test process improvement models can be found in [Swinkels, 2000][Goslin et al., 2008, p. 70].

2.3.1.3 Test Maturity Model Integrated (TMM*i*)

TMM*i* is being developed by a non-profit organization called TMM*i* Foundation. This framework is intended to complement Capability Maturity Model Integration (CMMI) with a special focus on testing activities and test process improvement in

[2] http://www.sogeti.nl/Home/Expertise/Testen/TPI.jsp

both the systems engineering and software engineering discipline. An initial version 1.0 [Goslin et al., 2008] of this framework has been released in February 2008. The current version follows staged representation and provides information only up to maturity level 2 out of the five proposed levels. The assessment framework itself is not part of TMM*i* and has not been released yet.

TMM*i* borrows its main principles and structure from Capability Maturity Model Integration (CMMI), Gelperin and Hetzel's Evolution of Testing Model [Gelperin and Hetzel, 1988], Beizer's testing model [Beizer, 1990], IEEE Standard for Software Test Documentation [iee, 1998b], and ISTQB' Standard Glossary of terms used in Software Testing [ist, 2006]. Similar to CMMI, this framework defines three types of components.

- **Required**: These components describe what an organization must achieve to satisfy a process area. Specific and generic goals make up required component of TMM*i*.
- **Expected**: These components describe what an organization will typically implement to achieve a required component. Expected components include both specific and generic practices.
- **Informative**: These components provide details that help organizations get started in thinking about how to approach the required and expected components. Sub-practices, typical work products, notes, examples, and references are all informative model components.

The TMM*i* model required, expected, and informative components can be summarized to illustrate their relationship as in figure 2.19. To each maturity level several process areas are associated which in turn involve several generic and specific goals and generic and specific practices. Informative components such as typical work products, notes, and examples describe other components.

TMM*i* defines five maturity levels. A maturity level within this framework indicates the quality of organizational test process. To reach a particular maturity level, an organization must satisfy all of the appropriate goals (both specific and generic) of the process areas at the specific level and also those at earlier maturity levels. All organizations possess a minimum of TMM*i* level 1, since this level does not contain any goals that must be satisfied. Figure 2.20 summarizes the maturity levels of this framework.

Test Maturity Model Integrated is no doubt a long awaited enhancement to its predecessor Testing Maturity Model. Below we present some critical observations of TMM*i*.

- The model description is yet incomplete since the currently available document only provides information up to maturity level 2.
- The assessment framework for TMM*i* is also not part of current release and is not yet publicly available.
- The current release of TMM*i* provides only a *staged* model representation. This same limitation was also observed for TMM [Farooq et al., 2007b]. A *continuous* representation on the other hand lets an organization to select a process area

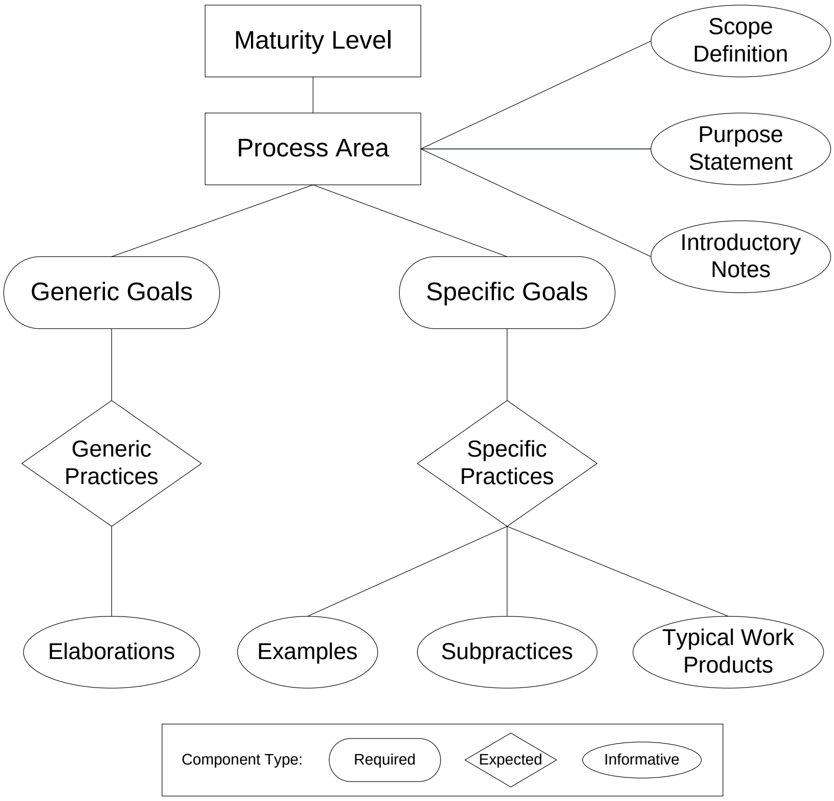


Figure 2.19: Structure of Test Maturity Model Integrated (TMM*i*)

(or group of process areas) and improve processes related to it. While staged and continuous representations have respective pros and cons, the availability of both representations provides maximum flexibility to organizations to address their particular needs at various steps in their improvement programs.

- TMM*i* is designed to be a complementary model to CMMI. The model description [Goslin et al., 2008, p. 6] states that "in many cases a given TMM*i* level needs specific support from process areas at its corresponding CMMI level or from lower CMMI levels. Process areas and practices that are elaborated within the CMMI are mostly not repeated within TMM*i*; they are only referenced". Now there are organizations which offer independent software testing services. Such or other organizations may solely want to concentrate on improvement of their testing process only. Strong coupling and references between TMM*i* and CMMI may limit independent adoption of this framework without implementing a CMMI process improvement model.

After reading through the above mentioned model descriptions, the reader might be interested in a more systematic and deeper analysis and comparison among these models. Two comparison frameworks applicable in this context are worth mentioning here. First is a generic taxonomy [Halvorsen and Conradi, 2001] to compare software process improvement (SPI) frameworks which can also be applied to compare test process improvement models. The second is a specialized evaluation frame-

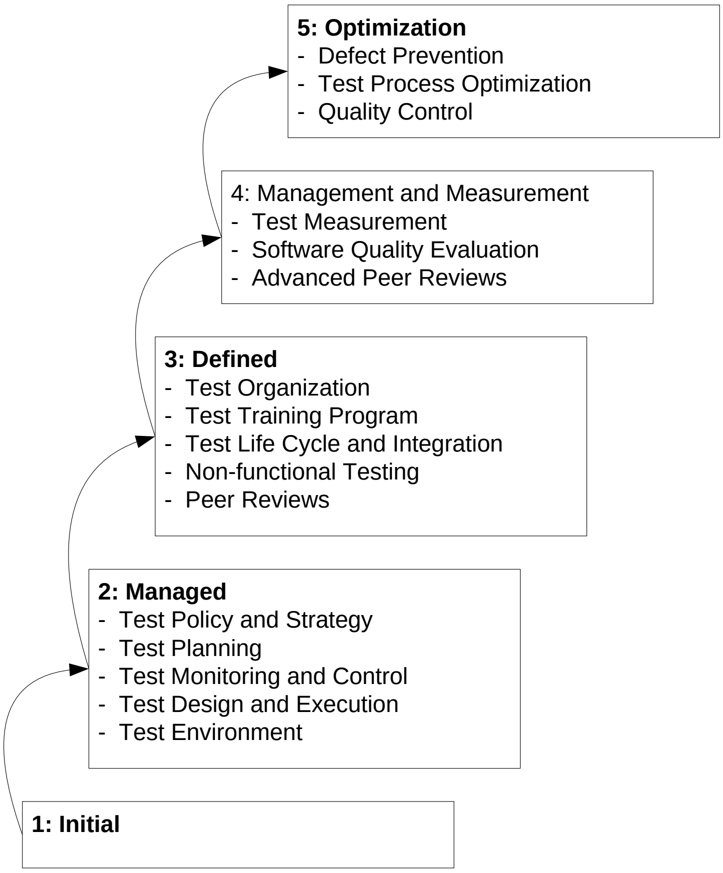


Figure 2.20: TMM*i* Maturity Levels

work [Farooq and Dumke, 2007a] for comparing test process improvement approaches. The later also compares characteristics of some of the above mentioned test maturity models.

2.3.2 Quantitative Approaches

Quantitative approaches to process management work by evaluating one or more of its attributes through measurement. The measurement information so obtained reflects some key characteristics of measured process such as size, involved effort, efficiency, and maintainability etc. The objectivity of the information provides possibility of precise and unbiased evaluation as compared to that obtained through assessments. Although several measurement tools and frameworks exist for the generic software process and can possibly be tailored to test process with minor or major changes, but very few have been developed solely for the test process. Measurement techniques for software test process exist broadly in the form of metrics for the test process. The next section analyzes available metrics in this area.

Table 2.3: Existing Test Metrics Resources

Reference	Test Aspect Covered
[Hutcheson, 2003, Ch. 5]	Some fundamental test metrics
[Rajan, 2006][Harris, 2006] [Whalen et al., 2006][Verma et al., 2005] [Sneed, 2005][Peng and Wallace, 1994]	test cases, coverage, failure
[Burnstein, 2003, p. 266][Chen et al., 2004]	testing status, tester productivity test effectiveness
[Nagappan et al., 2005][Kan et al., 2001] [Chaar et al., 1993][Kan, 2002, Ch. 10]	in-process metrics
[Liggesmeyer, 1995] [Suwannasart and Srichaivattana, 1999]	test complexity
[Burnstein, 2003, p. 372]	test process metrics
[Pusala, 2006][Sneed, 2007] [Abran et al., 2004, p. 5-7][Perry, 2006, Ch. 13]	miscellaneous

2.3.2.1 Test Process Metrics

Like other knowledge areas within software engineering, testing related measures are very helpful to managers to understand, track, control, and improve the testing process. For example, metrics of testing costs, test effectiveness, tester productivity, testability, test cases, coverage, defects and faults and other similar aspects can give us very valuable insight about many different aspects of software testing. Realizing necessity of such measurements, a number of test process metrics have been proposed and reported in literature. However, with few exceptions, test metrics definitions found in literature do not explicitly state if a metric is related to test process or some other aspect of software testing. The table 2.3 provides a non-comprehensive list of test metrics definitions.

Nonetheless, we can distinguish several of these metrics which are meaningful at the process level only, for example few maturity level metrics and process progress and effectivity metrics. Availability of so many metrics may sometimes confuse practitioners rather than help them. A well organized list of these metrics may help a test manager better understand metrics available at hand and to select them according to particular situations and needs. Feeling this need, Farooq et al. [Farooq et al., 2008] presented a classification of test metrics considering various test contexts. The authors also reviewed available test related metrics and existing test metrics classifications. Figure 3.1 shows Farooq's [Farooq et al., 2008] classification of test metrics. Another related approach to classify software process metrics was presented by Dumke et al. [Dumke et al., 2006b].

An examination of literature on test related metrics has revealed that research in this context is as yet immature. Each of set existing test metrics have been defined only in a confined context, serving the need of some particular analysis problem of a given testing aspect. We still lack widely known common set of test metrics. Moreover, existing test metrics remain poorly validated both from theoretical and empirical point of view.

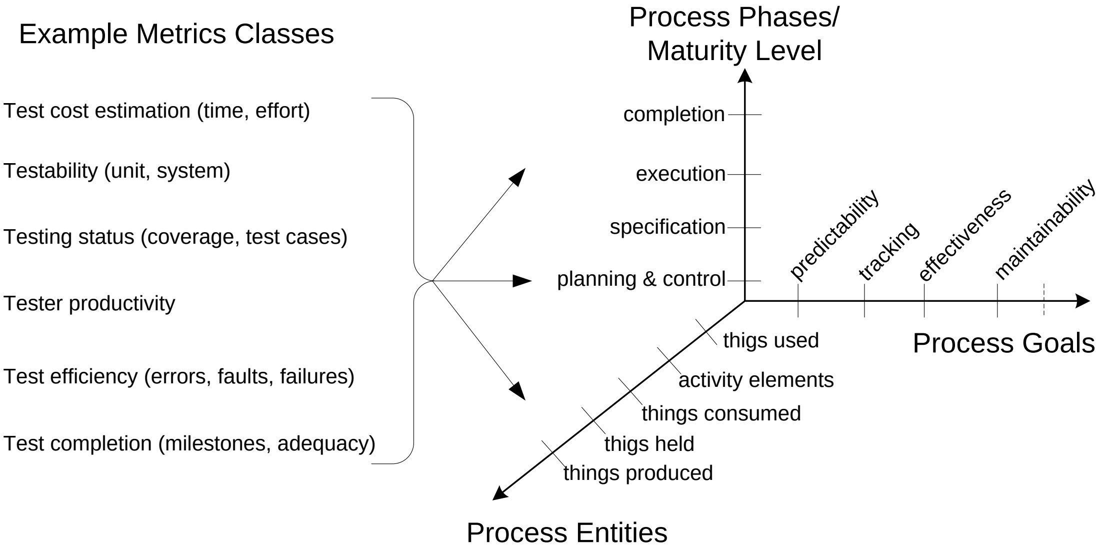


Figure 2.21: Classification of test process metrics [Farooq et al., 2007a]

3 Test Techniques: Fundamentals & Efficiencies

There are many reasons why the evaluation of testing techniques should be carried out. Issue of technique selection is one reason. We need to assess fault finding capability of candidate testing techniques. This kind of information is useful before we have implemented a given technique, but the same information is also useful (as a post mortem analysis) when we are finished with testing. This post-implementation assessment and analysis is needed for subsequent improvement of the technique to increase its effectiveness. This chapter surveys testing techniques, empirical knowledge about them, and existing ways for assessing them from different quality perspectives.

Before diving into a review and analysis of testing techniques lets first try to understand some overlapping terms in this context such as testing, verification, validation, static, and dynamic techniques. Following the traditional definition of V&V and testing given by IEEE [iee, 1990], *testing* refers to those techniques which involve execution of software code. However, a contemporary resource of testing related glossary provided by International Software Testing Qualification Board [ist, 2006] defines *static testing* and *dynamic testing* uniquely which closely overlap with commonly known definitions of *verification* and *validation* respectively. We will follow this later definitions of testing.

Software testing literature contains rich source of several books and articles explaining various kinds of testing techniques. Some well known resources include Beizer [Beizer, 1990], Perry [Perry, 2006, Ch. 17], Liggesmeyer [Liggesmeyer, 2002], Tian [Tian, 2005, Ch. 8-11], Pezze and Young [Pezzè and Young, 2007]. It seems appropriate here to first draw a wide picture of available techniques by presenting their classification. Some classes of testing techniques have been given in [Abran et al., 2004][Harkonen, 2004, p. 26][Juristo et al., 2004a][Tian, 2005]. However we prefer classification of testing techniques given by Liggesmeyer [Liggesmeyer, 2002, p. 34] which seems to be quite comprehensive in covering available techniques. The testing techniques reviewed in this chapter have been organized based on this classification. Figure 3.1 is a modified version of his classification.

3.1 Static techniques

Static testing techniques are usually applied at the initial steps in software testing. These are verification techniques which do not employ actual execution of code/program. These techniques attempt to ensure that organizational standards and guidelines for coding and design are being followed. Formal verification, inspection, reviews, and mea-

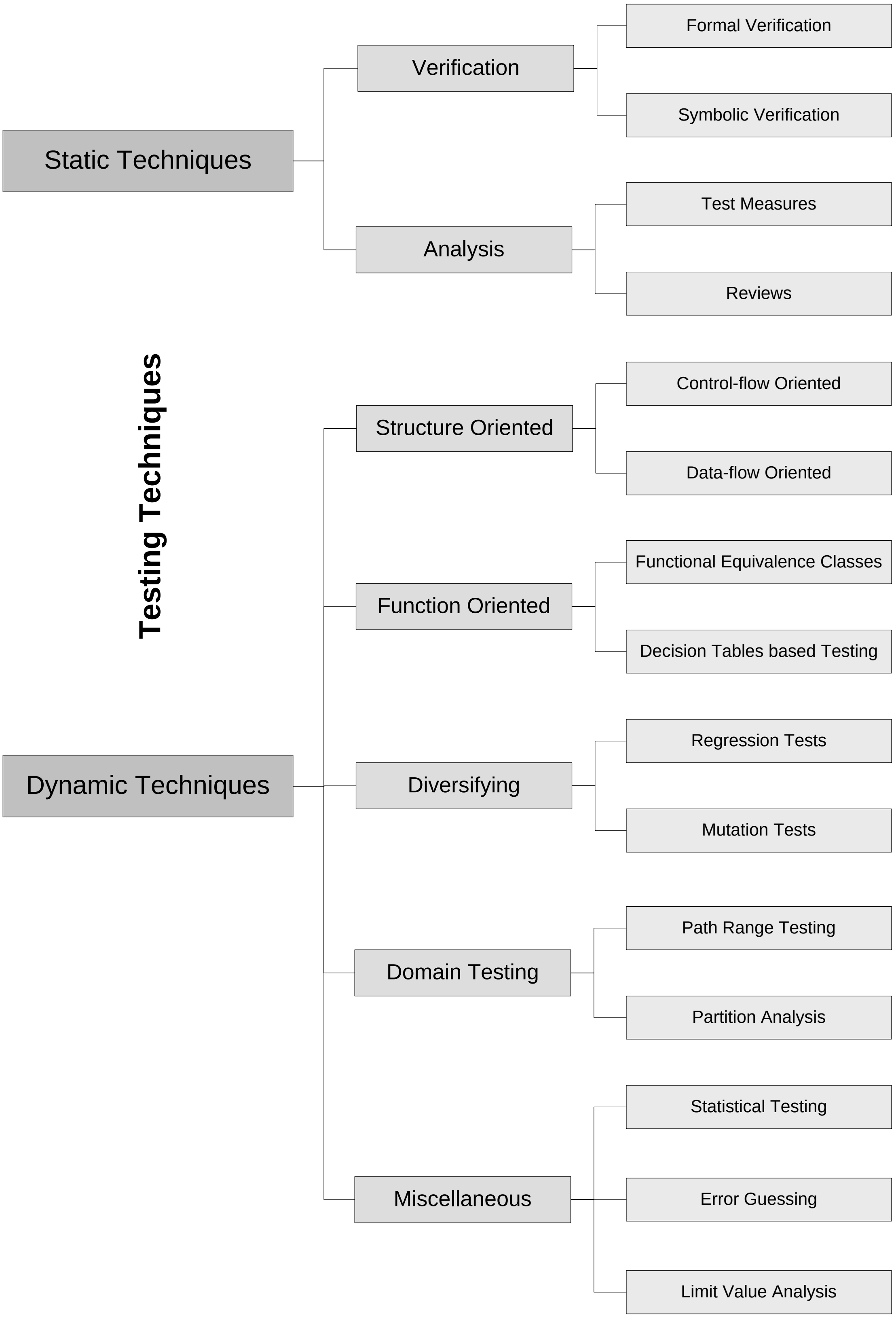


Figure 3.1: Liggesmeyer's classification of testing techniques

surement are main types of static techniques. Table 3.1 presents an abridged summary of static testing techniques.

Table 3.1: Summary of Static Testing Techniques

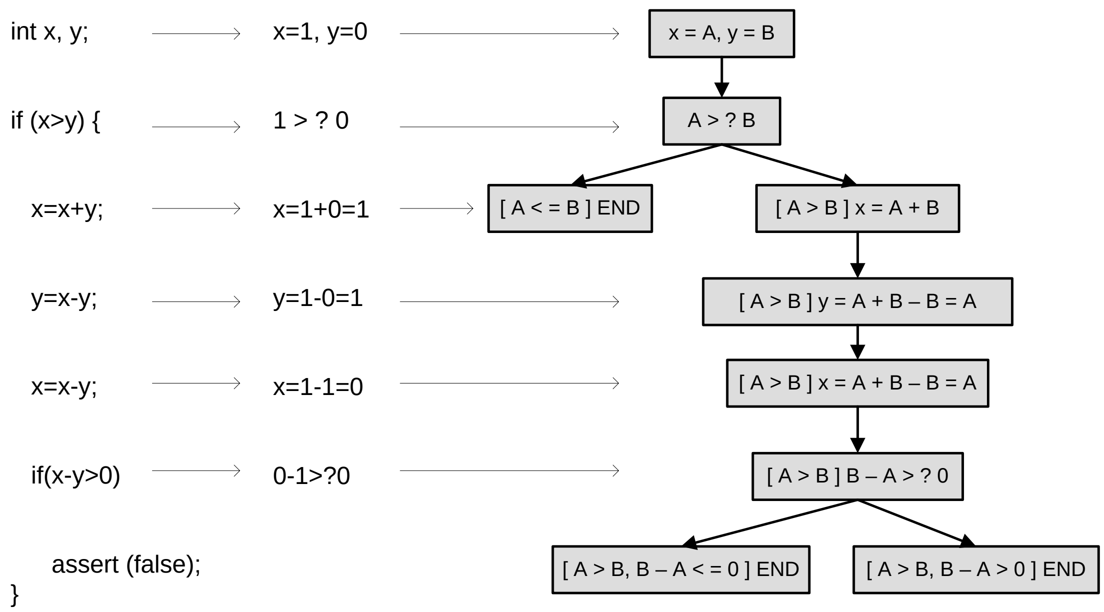
Category	Technique	Description
Verification	Formal Verification	Analyzes correctness of software systems based on their formal specification.
	Symbolic Verification	Program is executed by replacing symbolic values in place of original program variables to provide general characterization of program behavior.
Analysis	Measurement	Provides quantitative view of various attributes of testing artifacts.
	Review	A work product is examined for defects by individuals other than the producer.
	Inspection	Disciplined engineering practice for detecting and correcting defects in software artifacts
	Walk-through	The producer describes the product and asks for comments from the participants.
	Audit	An independent examination of work products to assess compliance with specifications, standards, or other criteria.
	Slicing	Technique for simplifying programs by focusing on selected aspects of semantics for debugging.

3.1.1 Verifying

Formal specifications is a way to precisely describe customer requirements, environmental constraints, and design intentions to reduce the chances of common specification errors. Verifying techniques check the conformance of software design or code to such formal specifications of the software under test. Verifying techniques are mainly focused on investigating functional requirements and aspects such as completeness, clarity, and consistency. Only a few of some well known techniques of this type will be discussed below. Verification techniques for several kinds of software programs are given in [Francez, 1992].

3.1.1.1 Formal verification

Formal verification is the use of mathematical techniques to ensure that a design conforms to some precisely expressed notion of functional correctness. Software testing alone cannot prove that a system does not have a certain defect, neither can it prove that it does have a certain property. The process of formal verification can prove that a system does not have a certain defect or does have



```
int x, y;
if (x>y) {
  x=x+y;
  y=x-y;
  x=x-y;
  if(x-y>0)
    assert (false);
}
```

Figure 3.2: An Example of Symbolic Execution

a certain property. Formal verification offers rich toolbox of mathematical techniques such as *temporal-logic model checking*, *constraint solving* and *theorem proving* [Lüttgen, 2006]. Clarke [Clarke and Wing, 1996] mentions two well established approaches to verification: *model checking* and *theorem proving*. Two general approaches to model checking are *temporal model checking* in which specifications are expressed in a temporal logic and systems are modeled as finite state transition systems while in second approach the specification is given as an automaton then the system, also modeled as an automaton, is compared to the specification to determine whether or not its behavior conforms to that of the specification [Clarke and Wing, 1996]. One of the most important advances in verification has been in decision procedures, algorithms which can decide automatically whether a formula containing Boolean expressions, linear arithmetic, enumerated types, etc. is satisfiable [Heitmeyer, 2005].

3.1.1.2 Symbolic testing

Symbolic testing [King, 1976][Pezzè and Young, 2007, Ch. 19] or symbolic execution is a program analysis technique in which a program is executed by replacing symbolic values in place of original program variables. This kind of testing is usually applied to selected execution paths as against formal program verification. Symbolic execution gives us a general characterization of program behavior which can help us in designing smarter unit tests [Tillmann and Schulte, 2006] or in generating path-oriented test data [Zhang et al., 2004]. Figure 3.2 gives an example of symbolic execution. Although this technique was developed more than three decades before, it has only recently become practical with hardware improvements and automatic reasoning algorithms.

3.1.2 Analyzing

Analyzing techniques attempt to find errors in software without executing it. However these techniques are not just limited to checking software entities but also involve reviewing designs and relevant documents. The main premise behind these techniques is that an earlier detection of bugs in software is less expensive than finding and fixing them at later development stages. These techniques analyze requirements, specifications, designs, algorithms, code, and documents. Examples of these techniques are;

- test measurements
- inspections
- reviews
- walk-throughs
- audits

3.1.2.1 Test measures

Measurement is a static analysis technique which can give us valuable information even before actually executing dynamic tests. Size, effort, complexity, and coverage like information can readily be obtained with the help of numerous test metrics. A detailed review of test related metrics has already appeared in this text in chapter 1 (under the section 'test process metrics').

3.1.2.2 Software reviews, inspections and walk-throughs

Software reviews [Hollocker, 1990] as defined by IEEE [iee, 1990] is a process or meeting during which a work product, or set of work products, is presented to project personnel, managers, users, customers, or other interested parties for comment or approval. IEEE standard [iee, 1997] which defines requirements for software reviews describes five types of reviews as *management reviews, technical reviews, inspections, walk-throughs*, and *audits*. Slightly different opinions over review types have been maintained by Ebenau et al. [Ebenau and Strauss, 1994] and Galin [Galin, 2004].

Reviews are usually performed for code, design, formal qualification, requirements, and test readiness etc. Since it is virtually impossible to perform full software testing, reviews are used as an essential quality control technique. A review increases the quality of the software product, reduces rework and ambiguous efforts, reduces testing and defines test parameters, and is a repeatable and predictable process [Lewis, 2004].

3.1.2.3 Fagan Inspections

Fagan inspection refers to a structured process of trying to find defects in development documents such as programming code, specifications, designs and others during various phases of the software development process. In a typical Fagan inspection the inspection process consists of the operations shown in figure 3.3.

Surveys, state-of-the-art studies, and future research directions within software reviews and inspections have been given by Aurum et al. [Aurum et al., 2002], Laitenberger [Laitenberger, 2002], and Ciolkowski [Ciolkowski et al., 2002]. Another very recent industrial practice survey of software reviews was performed by

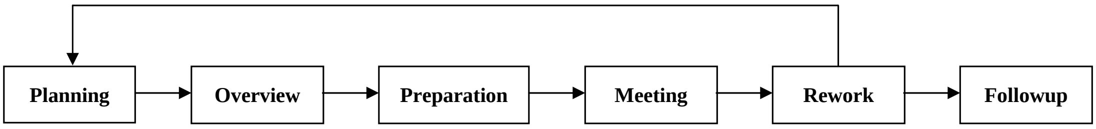


Figure 3.3: Fagan inspection basic model [Fagan, 1986]

Ciolkowski [Ciolkowski et al., 2003]. The authors concluded that "companies conduct reviews regularly but often unsystematically and full potential of reviews for defect reduction and quality control is often not exploited adequately".

A recent case study to judge effectiveness of software development technical reviews (SDTR) [Sauer et al., 2000] has concluded that the most important factor in determining the effectiveness of SDTRs is the level of expertise of the individual reviewers. Additionally, this study highlights three ways of improving performance: selection of reviewers who are expert at defect detection; training to improve individuals' expertise; and establishing group size at the limit of performance. Another study [Laitenberger et al., 1999] reported similar results rates preparation effort as the most important factor influencing defect detection capability of reviews.

3.2 Evaluation of Static Techniques

3.2.1 Evaluation criteria & methods

As observed in literature, the evaluation criteria for static testing techniques has largely been their ability for detecting defects, costs incurred, or expended time and effort. Lamsweerde [van Lamsweerde, 2000] mentions few qualitative criterion for evaluating specification techniques, namely constructibility, manageability, evolvability, usability, and communicability. Some of these attributes are applicable to other static techniques as well. For determining return on investment (ROI) for software inspection process, Rico [Rico, 2004] specifies several methodologies to determine benefit, benefit/cost ratio, return on investment percentage, and net present value of software inspections. Wu et al. [Wu et al., 2005] incorporate number of remaining faults in a Bayesian network model of the inspection process to measure its effectiveness. Another example of similar a model-based approach in this direction is [Freimut and Vollei, 2005]. An empirical technique for comparing inspection and testing has been worked out by Andersson et al [Andersson et al., 2003]. One of the very few approaches for evaluating formal verification techniques is [Wang et al., 1998] which involves an experiment for measuring effectiveness of design validation techniques based on automatic design error injection and simulation.

3.2.2 Evaluation results

Summarizing studies conducted by various researchers to evaluate the effectiveness of inspections as compared to testing , Eickelmann et al. [Eickelmann et al., 2002] mention that inspections are two times more effective than tests to identify errors, cause four times less effort than tests and are 7.4 times more productive than tests. However a recent case study [Chatzigeorgiou and Antoniadis, 2003] has identified that project planning methodologies, as currently applied in software project management, do not account for the inherent difficulties in planning software inspections and their related activities and as a result, inspection meetings accumulate at specific periods towards the project deadlines, possibly causing spikes in the project effort, overtime costs, quality degradation and difficulties in meeting milestones.

Finally, our analysis of literature on software reviews and inspections has revealed that current research in this area is now not focusing much on developing new inspection or review techniques. Rather, the modern (and some past) research effort is now being devoted mainly to studying factors that influence success and efficiency of reviews and inspections and to evaluating (relative) effectiveness of these techniques in comparison to other testing and related techniques.

3.3 Dynamic techniques

Dynamic testing techniques involve tests which employ system operation or code execution. Two broad categories of such dynamic methods exist, structural-based and functional-based. Dynamic techniques that exploit the internal structure of the code are known as structural, white-box, glass-box or coverage based tests. In contrast, those that do not involve the internal structure of the code are known as functional, black-box, behavioral or requirement-based tests. We will discuss these kinds of testing in the coming sections. Table 3.2 presents a very short summary of dynamic testing techniques.

3.3.1 Structure oriented

Types of testing techniques under this category exploit structural information about the software to derive test cases as well as determining coverage and adequacy of these test cases. In this context, data element and control element are two main elements in any computation or information processing task that are grouped through some implemented algorithms. Structural testing techniques [Pezzè and Young, 2007, Ch. 12] are mainly based on this control-flow and data-flow information about our code design.

3.3.1.1 Control-flow oriented

Control-flow testing focuses on the complete paths and the decisions as well as interactions along these execution paths. Control flow elements that may be examined are statements, branches, conditions, and paths. These elements are also generally considered for coverage criteria. For most computation intensive applications, which cover most of the traditional software systems, mere state and link coverage would not be enough because of the interconnected dynamic decisions along execution paths [Tian, 2005]. Therefore, control-flow testing is generally a necessary step among the variety of testing techniques for such systems.

Table 3.2: Summary of Dynamic Testing Techniques

Category	Technique	Description
Structured oriented	Data-flow oriented	Select test cases based on program path to explore sequences of events related to the data state.
	Control-flow oriented	Select test cases using information on complete paths and the decisions as well as interactions along these execution paths
Function oriented	Functional equivalence classes	Input domain of the software under test is partitioned into classes to generate one test case for each class.
	Decision tables	Select test cases exploiting information on complex logical relationships between input data.
	Cause-and-effect graphs	Causes and effects in specifications are drawn to derive test cases.
	Syntax testing	Test cases are based on format specification obtained from component inputs.
Diversifying	Regression testing	Selective retesting of a system to verify that modifications have not caused unintended effects.
	Mutation testing	Works by modifying certain statements in source code and checking if test code is able to find the errors.
	Back-to-back testing	For software subject to parallel implementation, it executes tests on similar implementations and compares the results.
Domain Testing	Partition analysis	Compares a procedure's implementation to its specification to verify consistency between the two and to derive test data.
Miscellaneous	Statistical testing	It selects test cases based on usage model of the software under test.
	Error guessing	Generate test cases based on tester's knowledge, experience, and intuition of possible bugs in the software under test.

3.3.1.2 Data-flow oriented

Data-flow testing [Pezzè and Young, 2007, Ch. 13][Beizer, 1990, Ch. 5] is based on principle of selecting paths through the program's control flow in order to explore sequences of events related to the status of data objects, for example, pick enough paths to assure that every data object has been initialized prior to use or that all defined objects have been used for something. It attempts to test correct handling of data dependencies during program execution. Program execution typically follows a sequential execution model, so we can view the data dependencies as embedded in the data flow, where the data flow is the mechanism that data are carried along during program execution [Tian, 2005]. Data flow test adequacy criteria improve over pure control flow criteria by selecting paths based on how one syntactic element can affect the computation of another.

3.3.2 Function oriented

IEEE [iee, 1990] defines function oriented testing or black-box testing as:

- Testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions.
- Testing conducted to evaluate the compliance of a system or component with specified functional requirements.

This type of testing does not exploit any knowledge about inner structure of the software. It can be applied towards testing of modules, member functions, object clusters, subsystems or complete software systems. The only system knowledge used in this approach comes from requirement documents, specifications, domain knowledge or defect analysis data. This approach is specifically useful for identifying requirements or specifications defects. Several kinds of functional test approaches are in practice such as

- decision tables
- functional equivalence classes
- domain testing
- transaction-flow based testing
- array and table testing
- limit testing
- boundary value testing
- database integrity testing
- cause-effect analysis
- orthogonal array testing
- exception testing
- random testing

Out of these, only a few commonly used techniques will be discussed in the coming sections.

3.3.2.1 Functional equivalence classes

This technique is used for minimizing the test cases that need to be performed in order to adequately test a given system. It produces a partitioning of the input domain of the software under test. The finite number of equivalence classes that are produced allow the tester to select a given member of an equivalence class as a representative of that class and the system is expected to act the same way for all tests of that equivalence class. A more formal description of equivalence classes has been given by Beizer [Beizer, 1995]. While Burnstein [Burnstein, 2003] regards derivation of input or output equivalence classes mainly a heuristic process, Meyers [Myers, 2004] suggests some more specific conditions as guidelines for selecting input equivalence classes.

3.3.2.2 Cause-and-effect graphing analysis

Equivalence class partitioning does not allow combining conditions. Cause-and-effect graphs can be used to combine conditions and derive an effective set of test cases that may disclose inconsistencies in a specification. Based on some empirical studies, Paradkar [Paradkar, 1994] relates some experiences of using cause-effect graphs for software specification and test generation. He found it very useful in reducing the cardinality of the required test suite and in identifying the ambiguities and missing parts in the specification. Nursimulu and Probert [Nursimulu and Probert, 1995] and Adler and Gray [Adler and Gray, 1983] pointed out ambiguities and some known drawbacks to cause-effect graphing analysis. Tai and Paradkar [Tai et al., 1993] developed a fault-based approach to test generation for cause-effect graphs, called BOR (Boolean operator) testing, which is based on the detection of boolean operator faults.

3.3.2.3 Syntax testing

Syntax testing [Beizer, 1995][Liggesmeyer, 2002], also called grammar-based testing, is a testing technique for testing applications where the input data can be described formally. Some example domains where syntax testing is applicable are GUI applications, XML/HTML applications, command-driven software, scripting languages, database query languages and compilers. According to Beizer [Beizer, 1995], syntax testing begins with defining the syntax using a formal meta-language such as Backus-Naur form (BNF) which is used to express context-free grammars and is a formal way to describe formal languages is the most popular. Once the BNF has been specified, generating a set of tests that covers the syntax graph is a straightforward matter.

The main advantage with syntax testing is that it can be automated, easily making this process easier, reliable and faster. Tools exist that support syntax testing. Tal et al. [Tal et al., 2004] performed a syntax-based vulnerability testing of frame-based network protocols. Marquis et al. [Marquis et al., 2005] explain a language called SCL (structure and context-sensitive) that can describe the syntax and the semantic constraints of a given protocol, and constraints that pertain to the testing of network application security. Their method reduces the manual effort needed when testing implementations of new (and old) protocols.

3.3.3 Diversifying

The diversifying test techniques pursue quite different goals. Diversifying test techniques do not serve in contrast to the structure-oriented or function-oriented test techniques. A goal of the diversifying test techniques is to sometimes avoid the often hardly possible evaluation of the correctness of the test results against the specification. Different types of diversifying techniques are back-to-back test, mutation test, and regression tests [Liggesmeyer, 2002]. Only regression testing, being probably the most widely researched technique in this category, will be discussed next.

3.3.3.1 Regression tests

Regression testing is defined by IEEE [iee, 1990] as selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements. Regression tests may apply at any level of testing such as unit tests etc to confirm no undesired changes have occurred during functional improvements or repairs.

The main issues in regression testing include;

- Removal of redundant and obsolete test cases
- Test case selection to reduce cost and time of retesting

The new version of software involve structural or other changes to modules which renders some of the previous test cases non-executable. Redundant test cases are those that are still executable but are irrelevant with rest to testing criteria. Re-executing all test cases other than obsolete and redundant affects regression testing complexity, effort and cost. We must select a suitable subset of these test cases. A number of techniques exist which attempt to reduce the test suite in this case. Some of these approaches are;

- Test case prioritization
- Test case selection
  - Code based
  - Specification based
  - Control-flow based
  - Data-flow based
  - Random sampling

Several regression testing techniques exist for specific problem situation. Muccini et al. [Muccini et al., 2005a] [Muccini et al., 2005b] [Muccini et al., 2006] explore how regression testing can be systematically applied at the software architecture level in order to reduce the cost of retesting modified systems, and also to assess the regression testability of the evolved system. Few other recently developed regression testing techniques include a scenario-based functional regression testing [Paul, 2001], regression testing for web-applications based on slicing [Xu et al., 2003], agile regression testing using record & playback [Meszaros, 2003], and regression testing technique for component-based software systems by enhancing change information [Mao and Lu, 2005].

Regression test selection and prioritization: Rothermel et al. [Rothermel et al., 2001] analyze few techniques for test case prioritization based on test case's code coverage and ability to reveal faults. Their analysis shows that each of the prioritization techniques studied improved the rate of fault detection of test suites, and this improvement occurred even with the least expensive of those techniques. Harry Sneed [Sneed, 2004] considers a problem which arises in the maintenance of large systems when the links between the specification based test cases and the code components they test are lost. It is no longer possible to perform selective regression testing because it is not known which test cases to run when a particular component is corrected or altered. To solve this problem, he proposes applying static and dynamic analysis of test cases. Other techniques include a new regression test selection technique for Java programs that is claimed to be safe, precise, and yet scales to large systems presented by Orso et al. [Orso et al., 2004], a regression test selection method for AspectJ program by Zhao et al. [Zhao et al., 2006], a regression test selection method for aspect-oriented programs by Guoqing Xu [Xu, 2006], a regression testing selection technique when source code is not available by Jiang Zheng [Zheng, 2005], and regression test selection method for COTS-based applications by Zheng et al. [Zhao et al., 2006].

Analysis of regression test techniques: Several other research works have performed cost-benefit or effectiveness analysis of regression test selection techniques. These include Rothermel and Harrold [Rothermel and Harrold, 1994], Harrold and Jones [Harrold et al., 2001], Graves and Harrold [Graves et al., 2001], Bible and Rothermel [Bible et al., 2001], Malishevsky and Rothermel [Malishevsky et al., 2002], Gittens and Lutfiyya [Gittens et al., 2002], and Rothermel and Elbaum [Rothermel et al., 2004]. These studies reveal that very few safety-based regression test selection techniques exist as compared to coverage-based techniques. Although the safety-based techniques were most effective in detecting faults, yet such techniques could not considerably reduce the test suite. The minimization techniques produced smallest and least effective set suites while safe and data-flow techniques had nearly equivalent behavior in terms of cost effectiveness.

3.3.4 Domain Testing

Selection of appropriate test data from input domain maximizing fault detection capability and minimizing costs is one major problem in black-box test design approach. Domain testing [Liggesmeyer, 2002, p. 190] attempts to partition the input domain and to select best representatives from these partitions to achieve these goals. Path analysis, partition testing, and random testing are usually used to short-list test data in domain testing.

Much research effort has been devoted to comparative analysis of these different domain testing approaches and varying opinions have been held by researchers. According to Gutjahr [Gutjahr, 1999], "in comparison between random testing and partition testing, deterministic assumptions on the failure rates systematically favor random testing, and that this effect is especially strong, if a partition consists of few large and many small sub-domains". He maintains that partition testing is better at detecting faults than random testing. In a later work, Ntafos [Ntafos, 2001] concluded that although partition testing generally performs better than random testing, the result can be reversed with a little addition in number of test cases.

3.4 Evaluation of Dynamic Techniques

A rich body of research work is available concerning evaluation of dynamic testing techniques as compared to static techniques. This research work has mainly been triggered by a need to select an appropriate technique among the many competing ones or due to an interest in validating usefulness or effectiveness of a given technique. For a given testing problem, there may exist several techniques of the same kind which differ by the underlying mechanism. Several regression testing techniques are available, they belong to same family, yet they follow a different way to solve the problem at hand. Contrary to this are techniques which solve the same testing problem, but exploit totally different set of information for the purpose. For example, the aim of control-flow and data-flow techniques is to generate tests but both of them derive these test cases quite differently. Following this distinction, Juristo et al. [Juristo et al., 2004a] identify two classes of evaluation studies on dynamic techniques as inter-family, and intra-family,

- Intra-family studies
  - Studies on data-flow testing techniques
  - Studies on mutation testing techniques
  - Studies on regression testing techniques
- Inter-family studies
  - Comparisons between control-flow, data-flow and random techniques.
  - Comparisons between functional and structural control-flow techniques.
  - Comparisons between mutation and data-flow techniques.
  - Comparisons between regression and improvement techniques.

We have already discussed *inter*-family analyses of individual techniques in respective sections. This section deals with wider range of *intra*-family studies over the state of research covering all dynamic testing techniques.

3.4.1 Evaluation criteria & methods

Three directions of research have been found related to evaluation of dynamic techniques,

1. Actual evaluations and comparisons of testing techniques based either on analytical or empirical methods

2. Evaluation frameworks or methodologies for comparing and/or selecting testing techniques

3. Surveys of empirical studies on testing techniques which have summarized available work and have highlighted future trends

During the past few decades, a large number of theoretical and empirical evaluations of numerous testing techniques have been executed. Morasca and Capizzano [Morasca and Serra-Capizzano, 2004] presented an analytical technique that is based on the comparison of the expected values of the number of failures caused by the applications of testing techniques, based on the total ordering among the failure rates of input sub-domains. They have also reviewed other approaches that compare techniques using expected number of failures caused or the probability of causing at least one failure.

The second stream of research in evaluation of dynamic technique is developing framework or guidelines for comparing and thus selecting an appropriate testing technique for a given problem domain. Some such frameworks are [Hierons, 2004][Misra, 2005][Eldh et al., 2006]. The most commonly considered attributes of test techniques are their efficiency, effectiveness, and applicability in detecting errors in programs. However, the major problems with these comparison frameworks are that they treat all types of faults and the underlying programs on which these techniques are to be evaluated as equal which can affect validity of such comparison results.

3.4.2 Evaluation results

Juristo [Juristo et al., 2002][Juristo et al., 2004a][Juristo et al., 2004b] performed very comprehensive analysis of several years of empirical work over testing techniques. She has highlighted following issues with current studies namely,

- Informality of the results analysis (many studies are based solely on qualitative graph analysis)
- Limited usefulness of the response variables examined in practice, as is the case of the probability of detecting at least one fault
- Non-representativeness of the programs chosen, either because of size or the number of faults introduced
- Non-representativeness of the faults introduced in the programs

An analysis of the maturity of empirical studies of various testing techniques has been given in [Juristo et al., 2004b]. Figure 3.4 has been adapted from the summary given therein. Additionally, Briand and Labiche [Briand and Labiche, 2004] discussed issues facing empirical studies of testing techniques. Criteria to quantify fault-detection ability of a technique is one such issue, while threats to validity arising out of the experimental setting (be it academic or industrial) is another. They suggest using (common) benchmark systems for such empirical experiments and standardizing the evaluation procedures.

Characteristic	Data-flow Testing	Mutation testing	Control-flow vs. Data-flow	Mutation vs. Data-flow	Functional vs. Control-flow
Experimental design rigour	▨	▨	▨	▨	■
Data analysis rigour	▨	▨	▨	▨	■
Findings beyond mere analysis	▨	▨	▨	▨	■
Use of programs/faults representative of reality	**N/A**	▨	▨	▨	▨
Response variables of interest to practitioners	■	▨	▨	□	■
Real technique application environment is taken into account	□	□	□	□	■
There are no topics remaining to be looked at or confirmed	□	□	□	□	▨
Experiment chaining	□	□	□	□	■
Methodological advancement in experimentation sequence	▨	□	▨	▨	□

■ Empirical study fully meets the characteristic ▨ Empirical study partially meets the characteristic □ Empirical study does not meet the characteristic

Figure 3.4: Study Maturity by Families

4 Capabilities of Test Tools

With the growth in size, maturity of practices, and increased workload, software organizations begin to feel a need for automating (some of the) testing procedures. A test tool is defined to be an automated resource that offers support to one or more test activities, such as planning and control, specification, constructing initial test files, execution of tests, and analysis [Pol et al., 2002, p. 429]. Supporting the testing process with tools can possibly increase the efficiency of test activities, reduce the effort required for executing routine test activities, improve the quality of software and the test process, and provide certain economic benefits. In summary, test tools automate manual testing activities thereby enabling efficient management of the testing activities and processes. But the level of test automation depends upon many factors such as type of application under development, testing process, and type of development and target environment. One hundred percent automatic testing has been regarded as a dream of modern testing research by Bertolino [Bertolino, 2007].

Evaluation of testing tools is important for many reasons. Due to an overwhelming number of testing tools available in the market, the decision to select the best tool remains elusive. We need subjective and objective evidence about candidate tools before we can arrive at a final choice for one of them. Only systematic guidelines and precise criteria to compare and evaluate tools is the befitting solution to this problem. This kind of quality evaluation, when at hand, establishes our confidence in the capability of the tool in solving our testing issues. This chapter deals with existing research work which has focused on developing procedures and criteria for testing tools evaluation.

4.1 Fundamentals

The first thought that should concern us while considering a tool implementation is determining whether it is really inevitable that a tool should be used. If the answer is positive, we must then look around for resources where we can find some appropriate tools. With the advancement of research and technology we expect to come across lot of tools of different kinds. At this stage we will be interested to organize this list in some fashion which could facilitate us in grasping an overview of available tools. These fundamental topics about test tools will be discussed shortly in the coming sections.

4.1.1 Is a Test Tool Inevitable?

Despite the long list of possible benefits expected of test tools, it is not wise to instantly start using a tool in all kinds of testing problems. The decision to use a test tool warrants careful cost-benefit analysis. Some testing tools may be very expensive in terms of money and effort involved and an organization may even be doing well without application of a sophisticated tool. Different sets of circumstances exist which may encourage or discourage adopting a testing tool. Ramler and Wolfmaier [Ramler and Wolfmaier, 2006] analyze trade-off between automated and manual testing and present a cost model based on opportunity cost to help decide when tests should be automated. Some situations that motivate organizations in automating testing tasks include,

- Test practices are mature
- Large size of the software
- Large number of tests required
- Time crunch

Some circumstances where using a testing tools may not be a wise choice include [Lewis, 2004, p. 321],

- Lack of a testing process
- Education and training of testers
- Technical difficulties with tool
- Organizational issues
- Ad hoc testing
- Cost
- Time crunch
- Organizational culture

4.1.2 Tool Resources

Once an organization decides that it will use a test tool, it comes across an immense variety of tools. These range from those supporting test planning to test design, generating and executing test cases, tracking defects, test documentation, logging and reporting software errors, and performance and load testing. Hundreds of testing tools of different capabilities are available both commercially and as open source software. A comprehensive or even partial survey of these tools is out of the scope of the current work. Instead, below we mention few resources which list some well known test tools.

- http://www.aptest.com/resources.html
- http://www.opensourcetesting.org/
- http://www.testingfaqs.org/
- SourceForge.net Project Repository
- Tool listing by Lewis [Lewis, 2004, p. 313-320]
- Tool listing by Farrell-Vinay [Keyes, 2003, p. 457-465]

Table 4.1: Classifications of Testing Tools

Classification Perspective	Resource
Functionality	http://www.opensourcetesting.org/, http://www.testingfaqs.org/, Lewis [Lewis, 2004, Ch. 29], Perry [Perry, 2006, Ch. 4], Farrell-Vinay [Keyes, 2003, appndx. D]
Testing Technique	Liggesmeyer [Liggesmeyer, 2002, Ch. 11], Perry [Perry, 2006, Ch. 4]
Testing/Process Level	Lewis [Lewis, 2004, Ch. 29], Pol [Pol et al., 2002, Ch. 27]
Process Maturity Level	Burnstein [Burnstein, 2003, Ch. 14]

Out of this vast variety of candidate test tools, selecting an appropriate tool to satisfy an organization's goals and constraints is undoubtedly a great challenge. To cope with such issues and to manage and understand these numerous test tools, few classifications and evaluation and selection criteria have been proposed. In the next sections we discuss these two issues.

4.1.3 Testing Tool Classifications

Test tool classifications help us understand state of research in tool development and the role of tools themselves in supporting test process activities. As for the classification of the testing tools, variety and very large number of test tools makes is difficult if not impossible at all to derive a single appropriate categorization of test tools. Sommerville [Sommerville, 2007, p. 86] presents three perspectives on classifying CASE (computer-aided software engineering) tools as functional, process, and integration perspective. In a similar way, many classification aspects of test tools are also possible in this context. Table 4.1 gives an overview of these aspects along with the literature resources which have based their classification of tools on these perspectives.

The classification given by Liggesmeyer [Liggesmeyer, 2002, Ch. 11] seems quite comprehensive in covering possible types of test tools. His technique is based on involved testing technique supported by a tool. Figure 4.1 shows his categorization of test tools.

4.2 Evaluation of Testing Tools

Evaluation of a testing tool is aimed at determining its functionality and quality. This evaluation may be meaningful at three different stages in development and test process. First and perhaps the most important stage is when we feel that we need a tool to automate our testing tasks, we have several candidate tools available at hand, and we want to make a judicious choice of selecting and implementing the most relevant tool

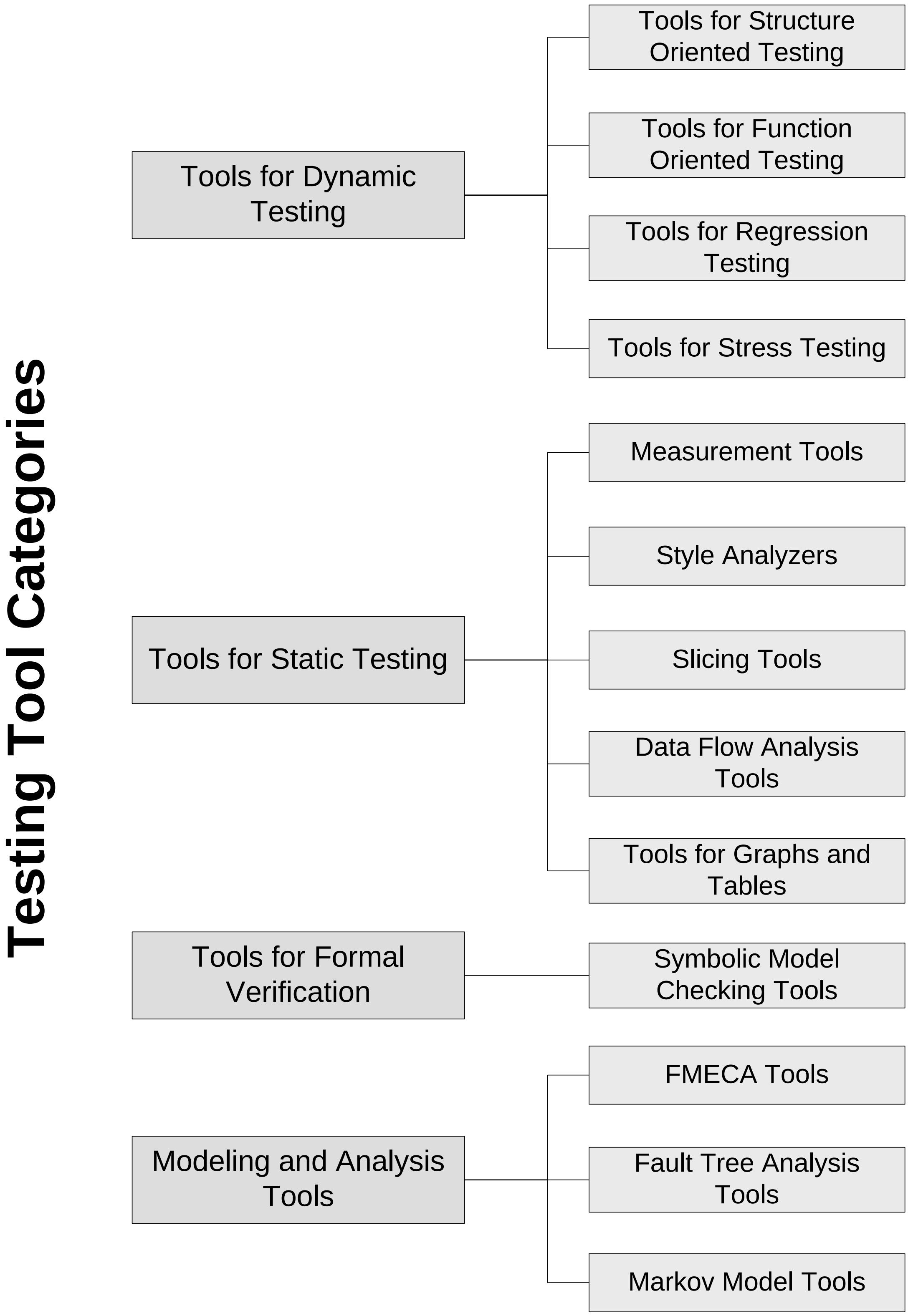

Testing Tool Categories
Tools for Dynamic Testing
Tools for Structure Oriented Testing
Tools for Function Oriented Testing
Tools for Regression Testing
Tools for Stress Testing
Tools for Static Testing
Measurement Tools
Style Analyzers
Slicing Tools
Data Flow Analysis Tools
Tools for Graphs and Tables
Tools for Formal Verification
Symbolic Model Checking Tools
Modeling and Analysis Tools
FMECA Tools
Fault Tree Analysis Tools
Markov Model Tools

Figure 4.1: Liggesmeyer's classification of testing tools

matching our needs and constraints. This is the pre-implementation stage. Second is the in-process stage. It is when we are in the middle of our test process and we want to track and control progress of our testing tasks. At this state it would be interesting to see number of test cases run in comparison to time, number of faults detected etc. A quite similar and third level of evaluation will be helpful when we are finished with a project and we want to assess what we have spent for a tool and what have we gained. If a tool is found to do well according to our cost benefit analysis, it will likely be re-implemented for next projects or otherwise. This third point of tool evaluation is a kind of post-implementation evaluation.

4.2.1 Pre-Implementation Analysis/ Tool Selection

Most test tool evaluation approaches belong to the type of pre-implementation analysis which involves assessing a tool based on certain criteria. The assessment results are used by a subsequent tool selection process. IEEE Standard 1209 [iee, 1992] distinguishes between evaluation and selection as, "evaluation is a process of measurement, while selection is a process of applying thresholds and weights to evaluation results and arriving at decisions".

A short discussion of some well known such evaluation techniques is given below.

- **IEEE Standard 1209, Recommended Practice for the Evaluation and Selection of CASE Tools** [iee, 1992]: This standard comprises three main sections; *evaluation process, selection process*, and *criteria*. The evaluation process provides guidelines on determining functionality and quality of CASE tools. The selection process chapter contains guidelines on identifying and prioritizing selection criteria and using it in conjunction with evaluation process to make a decision about a tool. The third section of the standard is criteria which is actually used by evaluation and selection process. It presents a framework of tool's quality attributes based on ISO 9126-1 standard.

- **Lewis' Methodology to Evaluate Automated Testing Tools** [Lewis, 2004, Ch. 30]: Lewis provides step-by-step guidelines for identifying tool objectives, conducting selection activities, and procuring, implementing, and analyzing the tool.

- **Task Oriented Evaluation of Illes et al.** [Illes et al., 2005]:They have defined functional and quality criteria for tools. Quality criteria has been specified using set of several quality attributes and sub-attributes influenced from ISO 9126-1 standard. The functional criteria are based on a task oriented view the test process and tools required for each test process phase are described. Their approach attempts to avoid laboratory test by forming the criteria which can be analyzed based on tool vendor's provided instructions.

- **Miscellaneous**: Some un-structured guidelines in this regard have been presented by Fewster and Graham [Fewster and Graham, 1999, Ch. 10], Spillner [Spillner et al., 2007, Ch. 12] et al. and Perry [Perry, 2006, Ch. 4].

The authors have discussed various implications involved with selecting, evaluating, and implementing test tools. Perry suggests considering development life cycle, tester's skill level, and cost comparisons for tools. Another similar example is Schulmeyer and Mackenzie's test tool reference guide [Schulmeyer and MacKenzie, 2000, p. 65].

4.2.2 In-Process & Post-Implementation Analysis

No specific method for an in-process evaluation of test tools exists. However, a quantitative criteria presented by Michael et al. [Michael et al., 2002] can be used both during the test process and also as a post-implementation analysis. They proposed several metrics for the purpose which are named below.

- Tool Management
- Human Interface Design
- Maturity & Customer Base
- Maximum Number of Parameters
- Test Case Generation
- Estimated Return on Investment
- Maximum Number of Classes
- User Control
- Ease of Use
- Tool Support
- Response Time
- Reliability
- Features Support

4.2.3 Summary

Many different subjective and objective criteria have been suggested in tool evaluation techniques. For the evaluations at all three different stages mentioned above, here we provide a combined list of various evaluation criteria which could contribute to a tool's evaluation or affect its selection.

- Quality attributes
  - Reliability
  - Usability
  - Efficiency
  - Functionality
  - Maintainability
  - Portability
- Vendor qualifications
  - Profile

  - Support
  - Licensing

- Cost
  - Purchasing & installation
  - Training

- Organizational constraints

- Environmental constraints
  - Lifecycle compatibility
  - Hardware compatibility
  - Software compatibility

In contrast to many evaluation works over testing techniques, our search into existing literature resources over evaluation of test tools returned very few results. It seems that the development of new testing tools has been given far more attention than analysis, measurement, and comparison among existing tools. Summarizing the above discussion we observe that systematic test tool selection and evaluation involves several steps, these include;

1. Principal decision to use a tool
2. Understanding concerned testing tools
3. Identification of tool requirements
4. Pre-evaluation
5. Selection
6. Post-evaluation

5 Summary & Future Work

The report has attempted to review status of evaluation in the field of software testing. Three core elements of the software testing have been identified as process, techniques, and tools. These elements have been variably exposed to evaluation works right from the beginning of testing research. This summary recounts purpose or motivation of evaluation, the technique used for the purpose (assessment, or direct measurement), the level of the evaluation (whether it provided an overall picture or only a partial reflection of the attribute of evaluation), and the evaluation type (relative, or an isolated analysis). The synopsis of the scientific disquisition contained in this report follows next. Table 5.1 further presents an abridged version of this summary.

The evaluation of testing tools has been motivated by a need for selection of an appropriate tool. Several descriptive guidelines exist for this purpose which discuss how to compare different testing tools. Measurement of some quality attributes of tools based on metrics has also been suggested in some works.

The evaluation of testing techniques has also been induced by questions of technique selection and effectiveness or efficiency determination. In most cases a comparative analysis enabled by empirical analysis has provided the answer. Additionally, partial measurements of some quality attribute have also been performed in isolation.

Finally, the quest for test process improvement led to its evaluation from different perspectives. Majority of approaches have targeted maturity or capability assessments spanning the whole range of testing activities and thus building a complete picture of process quality. Nevertheless, few other works concentrated on fragmentary measurement of quality attributes by exploiting sets of process metrics for the purpose.

5.1 Future Work

- **Explicit and Lightweight Measurement of Test Process:** Existing test process evaluation and improvement models are implicit in nature and either resource and cost intensive or are capable of only partial process measurements. Lightweight and explicit process measurement enabled by comprehensive test metrics can provide remedy to current deficiencies in this context.

- **Test Process for SOA-based Systems:** Numerous software application domains warrant their own testing challenges. Testing of service-oriented applications is different than testing of ordinary programs. Keeping in mind the relevant business processes and specialized testing strategies, test process for SOA-based systems needs to be reformed.

- **Test Process for Small and Medium Enterprises:** Rigorous process improvement and assessment models such as CMMI and SPICE require exhaustive procedures to ensure a well managed process. While this may work well for large organizations involved in development of large software projects, the same may be difficult to implement in small IT organizations. Although small software companies face similar quality requirements yet they have limited resources. A customized, simplified, and less resource-intensive test process improvement model needs to be considered keeping in mind the constraints of the small software organizations.

Table 5.1: Summary of Evaluation in Software Testing

	Purpose	Technique Used	Evaluation Level	Evaluation Type
Tools	Selection, Quality Determination	Descriptive Guidelines, Measurement	Partial Quality Attributes	Comparative, Solitary
Techniques	Selection, Quality Evaluation	Empirical Analysis, Measurement	Partial Quality Attributes	Comparative, Solitary
Process	Process Improvement	Maturity Assessment, Measurement	Complete, Partial Quality Attributes	Solitary

List of Tables

List of Figures

Bibliography

[iee, 1987] (1987). ANSI/IEEE Std 1008-1987:IEEE standard for software unit testing.

[iee, 1990] (1990). IEEE Std 610.12-1990:IEEE standard glossary of software engineering terminology.

[iee, 1992] (1992). IEEE Std 1209-1992:IEEE recommended practice for evaluation and selection of CASE tools.

[iee, 1997] (1997). IEEE std 1028-1997:IEEE standard for software reviews.

[iee, 1998a] (1998a). IEEE Std 1012-1998:IEEE standard for software verification and validation.

[iee, 1998b] (1998b). IEEE Std 829-1998 IEEE standard for software test documentation.

[iee, 1998c] (1998c). IEEE/EIA 12207.0-1996 standard for information technology-software life cycle processes.

[bcs, 2001] (2001). BCS SIGiST standard for software component testing.

[ist, 2006] (2006). ISTQB standard glossary of terms used in software testing.

[iso, 2007] (2007). ISO/IEC 15939:2007: Systems and software engineering – measurement process.

[Abran et al., 2004] Abran, A., Bourque, P., Dupuis, R., and Moore, J. W., editors (2004). *Guide to the Software Engineering Body of Knowledge - SWEBOK*. IEEE Press, Piscataway, NJ, USA.

[Abu et al., 2005] Abu, G., Cangussu, J. W., and Turi, J. (2005). A quantitative learning model for software test process. In *HICSS '05: Proceedings of the 38th Annual Hawaii International Conference on System Sciences - Track 3*, page 78.2, Washington, DC, USA. IEEE Computer Society.

[Acuña et al., 2001] Acuña, S. T., Antonio, A. D., Ferré, X., López, M., , and Maté, L. (2001). The software process: Modelling, evaluation and improvement. *Handbook of Software Engineering and Knowledge Engineering*, pages 193–237.

[Adler and Gray, 1983] Adler, M. and Gray, M. A. (1983). A formalization of Myers cause-effect graphs for unit testing. *SIGSOFT Softw. Eng. Notes*, 8(5):24–32.

[Andersson et al., 2003] Andersson, C., Thelin, T., Runeson, P., and Dzamashvili, N. (2003). An experimental evaluation of inspection and testing for detection of design faults. In *ISESE '03: Proceedings of the 2003 International Symposium on Empirical Software Engineering*, page 174, Washington, DC, USA. IEEE Computer Society.

[Apel, 2005] Apel, S. (2005). Software reliability growth prediction-state of the art. Technical report, IESE-Report No. 034.05/E Fraunhofer Institute of Experimental Software Engineering.

[Ares et al., 1998] Ares, J., Dieste, O., Garcia, R., Löpez, M., and Rodriguez, S. (1998). Formalising the software evaluation process. In *SCCC '98: Proceedings of the XVIII International Conference of the Chilean Computer Science Society*, page 15, Washington, DC, USA. IEEE Computer Society.

[Arthur et al., 1999] Arthur, J. D., Groner, M. K., Hayhurst, K. J., and Holloway, C. M. (1999). Evaluating the effectiveness of independent verification and validation. *Computer*, 32(10):79–83.

[Arthur and Nance, 1996] Arthur, J. D. and Nance, R. E. (1996). Independent verification and validation: a missing link in simulation methodology? In *WSC '96: Proceedings of the 28th conference on Winter simulation*, pages 230–236, Washington, DC, USA. IEEE Computer Society.

[Astels, 2003] Astels, D. (2003). *Test Driven development: A Practical Guide*. Prentice Hall Professional Technical Reference.

[Aurum et al., 2002] Aurum, A., Petersson, H., and Wohlin, C. (2002). State-of-the-art: software inspections after 25 years. *Softw. Test., Verif. Reliab.*, 12(3):133–154.

[Beck, 2002] Beck, K. (2002). *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Beizer, 1990] Beizer, B. (1990). *Software Testing Techniques*. Van Nostrand Reinhold, New York, USA.

[Beizer, 1995] Beizer, B. (1995). *Black-box testing: techniques for functional testing of software and systems*. John Wiley & Sons, Inc., New York, NY, USA.

[Bertolino, 2007] Bertolino, A. (2007). Software testing research: Achievements, challenges, dreams. In *FOSE '07: 2007 Future of Software Engineering*, pages 85–103, Washington, DC, USA. IEEE Computer Society.

[Bible et al., 2001] Bible, J., Rothermel, G., and Rosenblum, D. S. (2001). A comparative study of coarse- and fine-grained safe regression test-selection techniques. *ACM Trans. Softw. Eng. Methodol.*, 10(2):149–183.

[Briand and Labiche, 2004] Briand, L. and Labiche, Y. (2004). Empirical studies of software testing techniques: challenges, practical strategies, and future research. *SIGSOFT Softw. Eng. Notes*, 29(5):1–3.

[Broekman and Notenboom, 2003] Broekman, B. and Notenboom, E. (2003). *Testing Embedded Software*. Addison-Wesley, Great Britain.

[Burnstein, 2003] Burnstein, I. (2003). *Practical Software Testing: A Process-oriented Approach*. Springer Inc., New York, NY, USA.

[Canfora et al., 2006] Canfora, G., Cimitile, A., Garcia, F., Piattini, M., and Visaggio, C. A. (2006). Evaluating advantages of test driven development: a controlled experiment with professionals. In *ISESE '06: Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 364–371, New York, NY, USA. ACM.

[Cangussu, 2002] Cangussu, J. W. (2002). *A Mathematical Foundation for Software Process Control*. PhD thesis, Purdue University, West Lafayette, IN, USA.

[Cangussu, 2003] Cangussu, J. W. (2003). A stochastic control model of the software test process. In *ProSim'03: Proceedings of the Workshop on Software Process Simulation Modeling*.

[Cangussu et al., 2000] Cangussu, J. W., DeCarlo, R., and Mathur, A. (2000). A state variable model for the software test process. In *Proceedings of 13th International Conference on Software & Systems Engineering and their Applications, Paris-France*.

[Cangussu et al., 2001a] Cangussu, J. W., DeCarlo, R., and Mathur, A. P. (2001a). A state model for the software test process with automated parameter identification. *2001 IEEE International Conference on Systems, Man, and Cybernetics*, 2:706–711.

[Cangussu et al., 2002] Cangussu, J. W., DeCarlo, R. A., and Mathur, A. P. (2002). A formal model of the software test process. *IEEE Trans. Softw. Eng.*, 28(8):782–796.

[Cangussu et al., 2003a] Cangussu, J. W., DeCarlo, R. A., and Mathur, A. P. (2003a). Monitoring the software test process using statistical process control: a logarithmic approach. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 158–167, New York, NY, USA. ACM Press.

[Cangussu et al., 2003b] Cangussu, J. W., DeCarlo, R. A., and Mathur, A. P. (2003b). Using sensitivity analysis to validate a state variable model of the software test process. *IEEE Trans. Softw. Eng.*, 29(5):430–443.

[Cangussu et al., 2001b] Cangussu, J. W., Mathur, A. P., and DeCarlo, R. A. (2001b). Feedback control of the software test process through measurements of software reliability. In *ISSRE '01: Proceedings of the 12th International Symposium on Software Reliability Engineering*, page 232, Washington, DC, USA. IEEE Computer Society.

[Chaar et al., 1993] Chaar, J. K., Halliday, M. J., Bhandari, I. S., and Chillarege, R. (1993). In-process evaluation for software inspection and test. *IEEE Trans. Softw. Eng.*, 19(11):1055–1070.

[Chatzigeorgiou and Antoniadis, 2003] Chatzigeorgiou, A. and Antoniadis, G. (2003). Efficient management of inspections in software development projects. *Information & Software Technology*, 45(10):671–680.

[Chen et al., 2004] Chen, Y., Probert, R. L., and Robeson, K. (2004). Effective test metrics for test strategy evolution. In *CASCON '04: Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, pages 111–123. IBM Press.

[Chernak, 2004] Chernak, Y. (2004). Introducing TPAM: Test process assessment model. *Crosstalk-The Journal of Defense Software Engineering*.

[Ciolkowski et al., 2003] Ciolkowski, M., Laitenberger, O., and Biffl, S. (2003). Software reviews: The state of the practice. *IEEE Software*, 20(06):46–51.

[Ciolkowski et al., 2002] Ciolkowski, M., Laitenberger, O., Rombach, D., Shull, F., and Perry, D. (2002). Software inspections, reviews & walkthroughs. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 641–642, New York, NY, USA. ACM.

[Clarke and Wing, 1996] Clarke, E. M. and Wing, J. M. (1996). Formal methods: state of the art and future directions. *ACM Comput. Surv.*, 28(4):626–643.

[Drabick, 2003] Drabick, R. D. (2003). *Best Practices for the Formal Software Testing Process: A Menu of Testing Tasks*. Dorset House.

[Dumke, 2005] Dumke, R. R. (2005). Software measurement frameworks. In *Proceedings of the 3rd World Congress on Software Quality*, pages 72–82, Erlangen, Germany. International Software Quality Institute GmbH.

[Dumke et al., 2006a] Dumke, R. R., Braungarten, R., Blazey, M., Hegewald, H., Reitz, D., and Richter, K. (2006a). Software process measurement and control - a measurement-based point of view of software processes. Technical report, Dept. of Computer Science, University of Magdeburg.

[Dumke et al., 2006b] Dumke, R. R., Braungarten, R., Blazey, M., Hegewald, H., Reitz, D., and Richter, K. (2006b). Structuring software process metrics. In *IWSM/MetriKon 2006: Proceedings of the 16th International Workshop on Software Metrics and DASMA Software Metrik Kongress*, pages 483–497, Aachen, Germany. Shaker Verlag.

[Dumke et al., 2004] Dumke, R. R., Côté, I., and Andruschak, O. (2004). Statistical process control (SPC) - a metric-based point of view of software processes achieving the CMMI level four. Technical report, Dept. of Computer Science, University of Magdeburg.

[Dumke and Ebert, 2007] Dumke, R. R. and Ebert, C. (2007). *Software Measurement: Establish Extract Evaluate Execute*. Springer Verlag, Berlin, Germany.

[Dumke et al., 2005] Dumke, R. R., Schmietendorf, A., and Zuse, H. (2005). Formal descriptions of software measurement and evaluation-a short overview and evaluation. Technical report, Dept. of Computer Science, University of Magdeburg.

[Durant, 1993] Durant, J. (1993). Software testing practices survey report. Technical report, Software Practices Research Center.

[Ebenau and Strauss, 1994] Ebenau, R. G. and Strauss, S. H. (1994). *Software inspection process*. McGraw-Hill, Inc., New York, NY, USA.

[Ebert et al., 2004] Ebert, C., Dumke, R., Bundschuh, M., and Schmietendorf, A. (2004). *Best Practices in Software Measurement*. Springer Verlag.

[Eickelmann et al., 2002] Eickelmann, N. S., Ruffolo, F., Baik, J., and Anant, A. (2002). An empirical study of modifying the Fagan inspection process and the resulting main effects and interaction effects among defects found, effort required, rate of preparation and inspection, number of team members and product. In *SEW'02: Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop*, page 58, Washington, DC, USA. IEEE Computer Society.

[El-Far and Whittaker, 2001] El-Far, I. K. and Whittaker, J. A. (2001). Model-based software testing. *Encyclopedia on Software Engineering*.

[Eldh et al., 2006] Eldh, S., Hansson, H., Punnekkat, S., Pettersson, A., and Sundmark, D. (2006). A framework for comparing efficiency, effectiveness and applicability of software testing techniques. In *TAIC-PART '06: Proceedings of the Testing: Academic & Industrial Conference on Practice And Research Techniques*, pages 159–170, Washington, DC, USA. IEEE Computer Society.

[Ericson et al., 1998] Ericson, T., Subotic, A., and Ursing, S. (1998). TIM a test improvement model. *J. Softw. Test., Verif. Reliab.*, 7(4):229–246.

[Everett et al., 2007] Everett, G. D., Raymond, and Jr., M. (2007). *Software Testing: Testing Across the Entire Software Development Life Cycle*. Wiley InterScience, Hobokon, NJ, USA.

[Fagan, 1986] Fagan, M. E. (1986). Advances in software inspections. *IEEE Trans. Softw. Eng.*, 12(7):744–751.

[Farooq and Dumke, 2007a] Farooq, A. and Dumke, R. R. (2007a). Developing and applying a consolidated evaluation framework to analyze test process improvement approaches. In *IWSM-MENSURA 2007: Proceedings of the International Conference on Software Process and Product Measurement*, volume 4895 of *Lecture Notes in Computer Science*, pages 114–128. Springer.

[Farooq and Dumke, 2007b] Farooq, A. and Dumke, R. R. (2007b). Research directions in verification & validation process improvement. *SIGSOFT Softw. Eng. Notes*, 32(4):3.

[Farooq et al., 2007a] Farooq, A., Dumke, R. R., Hegewald, H., and Wille, C. (2007a). Structuring test process metrics. In *MetriKon 2007: Proceedings of the DASMA Software Metrik Kongress*, pages 95–102, Aachen, Germany. Shaker Verlag.

[Farooq et al., 2008] Farooq, A., Dumke, R. R., Schmietendorf, A., and Hegewald, H. (2008). A classification scheme for test process metrics. In *SEETEST 2008: South East European Software Testing Conference*, Heidelberg, Germany. dpunkt.verlag.

[Farooq et al., 2007b] Farooq, A., Hegewald, H., and Dumke, R. R. (2007b). A critical analysis of the Testing Maturity Model. *Metrics News, Journal of GI-Interest Group on Software Metrics*, 12(1):35–40.

[Fewster and Graham, 1999] Fewster, M. and Graham, D. (1999). *Software test automation: effective use of test execution tools*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.

[Foos et al., 2008] Foos, R., Bunse, C., Höpfner, H., and Zimmermann, T. (2008). TML: an XML-based test modeling language. *SIGSOFT Softw. Eng. Notes*, 33(2):1–6.

[Francez, 1992] Francez, N. (1992). *Program Verification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Freimut and Vollei, 2005] Freimut, B. and Vollei, F. (2005). Determining inspection cost-effectiveness by combining project data and expert opinion. *IEEE Trans. Softw. Eng.*, 31(12):1074–1092.

[Galin, 2004] Galin, D. (2004). *Software Quality Assurance: From Theory to Implementation*. Addison-Wesley, Harlow, England.

[Gelperin and Hetzel, 1988] Gelperin, D. and Hetzel, B. (1988). The growth of software testing. *Communications of the Association of Computing Machinery*, 31(6):687–695.

[Gittens et al., 2002] Gittens, M., Lutfiyya, H., Bauer, M., Godwin, D., Kim, Y. W., and Gupta, P. (2002). An empirical evaluation of system and regression testing. In *CASCON '02: Proceedings of the 2002 conference of the Centre for Advanced Studies on Collaborative research*, page 3. IBM Press.

[Glass, 1994] Glass, R. L. (1994). The software-research crisis. *IEEE Software*, 11(6):42–47.

[Glass et al., 2004] Glass, R. L., Ramesh, V., and Vessey, I. (2004). An analysis of research in computing disciplines. *Commun. ACM*, 47(6):89–94.

[Goslin et al., 2008] Goslin, A., Olsen, K., O'Hara, F., Miller, M., Thompson, G., and Wells, B. (2008). *Test Maturity Model Integrated-TMM*i *(http://www.tmmifoundation.org/downloads/resources/TMMi%20Framework.pdf).* TMMi Foundation.

[Graves et al., 2001] Graves, T. L., Harrold, M. J., Kim, J.-M., Porter, A., and Rothermel, G. (2001). An empirical study of regression test selection techniques. *ACM Trans. Softw. Eng. Methodol.*, 10(2):184–208.

[Gutjahr, 1999] Gutjahr, W. J. (1999). Partition testing vs. random testing: The influence of uncertainty. *IEEE Trans. Softw. Eng.*, 25(5):661–674.

[Halvorsen and Conradi, 2001] Halvorsen, C. P. and Conradi, R. (2001). A taxonomy to compare SPI frameworks. In *EWSPT '01: Proceedings of the 8th European Workshop on Software Process Technology*, pages 217–235, London, UK. Springer-Verlag.

[Harkonen, 2004] Harkonen, J. (2004). Testing body of knowledge. Master's thesis, Faculty of Technology, University of Oulu, Oulu, Finland.

[Harris, 2006] Harris, I. G. (2006). A coverage metric for the validation of interacting processes. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 1019–1024, 3001 Leuven, Belgium, Belgium. European Design and Automation Association.

[Harrold, 2000] Harrold, M. J. (2000). Testing: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 61–72, New York, NY, USA. ACM Press.

[Harrold et al., 2001] Harrold, M. J., Jones, J. A., Li, T., Liang, D., Orso, A., Pennings, M., Sinha, S., Spoon, S. A., and Gujarathi, A. (2001). Regression test selection for java software. In *OOPSLA'01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 312–326, New York, NY, USA. ACM Press.

[Hartman et al., 2007] Hartman, A., Katara, M., and Olvovsky, S. (2007). Choosing a test modeling language: A survey. In *HVC 06: Second International Haifa Verification Conference*, volume 4383 of *Lecture Notes in Computer Science*, pages 204–218. Springer.

[Heitmeyer, 2005] Heitmeyer, C. (2005). A panacea or academic poppycock: Formal methods revisited. *HASE '05: Proceedings of the Ninth IEEE International Symposium on High-Assurance Systems Engineering*, 0:3–7.

[Höfer and Tichy, 2007] Höfer, A. and Tichy, W. F. (2007). Status of empirical research in software engineering. In *Empirical Software Engineering Issues*, volume 4336/2007, pages 10–19. Springer.

[Hierons, 2004] Hierons, R. M. (2004). A flexible environment to evaluate state-based test techniques. *SIGSOFT Softw. Eng. Notes*, 29(5):1–3.

[Hollocker, 1990] Hollocker, C. P. (1990). *Software reviews and audits handbook.* John Wiley & Sons, Inc., New York, NY, USA.

[Hutcheson, 2003] Hutcheson, M. L. (2003). *Software Testing Fundamentals: Methods and Metrics*. John Wiley & Sons, Inc., New York, NY, USA.

[Illes et al., 2005] Illes, T., Herrmann, A., Paech, B., and Rückert, J. (2005). Criteria for software testing tool evaluation-a task oriented view. In *Proceedings of the 3rd World Congress of Software Quality*.

[Jacobs and Trienekens, 2002] Jacobs, J. C. and Trienekens, J. J. M. (2002). Towards a metrics based verification and validation maturity model. In *STEP '02: Proceedings of the 10th International Workshop on Software Technology and Engineering Practice*, page 123, Washington, DC, USA. IEEE Computer Society.

[Juristo et al., 2002] Juristo, N., Moreno, A. M., and Vegas, S. (2002). A survey on testing technique empirical studies: How limited is our knowledge. In *ISESE '02: Proceedings of the 2002 International Symposium on Empirical Software Engineering*, page 161, Washington, DC, USA. IEEE Computer Society.

[Juristo et al., 2004a] Juristo, N., Moreno, A. M., and Vegas, S. (2004a). Reviewing 25 years of testing technique experiments. *Empirical Softw. Engg.*, 9(1-2):7–44.

[Juristo et al., 2004b] Juristo, N., Moreno, A. M., and Vegas, S. (2004b). Towards building a solid empirical body of knowledge in testing techniques. *SIGSOFT Softw. Eng. Notes*, 29(5):1–4.

[Kan, 2002] Kan, S. H. (2002). *Metrics and Models in Software Quality Engineering*. Addison-Wesley Pub. Company, Inc.

[Kan et al., 2001] Kan, S. H., Parrish, J., and Manlove, D. (2001). In-process metrics for software testing. *IBM Systems Journal*, 40(1):220–241.

[Kenett and Baker, 1999] Kenett, R. S. and Baker, E. R., editors (1999). *Software Process Quality Management and Control*. Marcel Dekker Inc., New York, NY, USA.

[Keyes, 2003] Keyes, J. (2003). *Manage Engineering Handbook*. Auerbach Publications, Boston, MA, USA.

[King, 1976] King, J. C. (1976). Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394.

[Komi-Sirviö, 2004] Komi-Sirviö, S. (2004). *Development and Evaluation of Software Process Improvement Methods*. PhD thesis, Faculty of Science, University of Oulu, Oulu, Finland.

[Koomen, 2002] Koomen, T. (2002). Worldwide survey on Test Process Improvement. Technical report, Sogeti.

[Koomen, 2004] Koomen, T. (2004). Worldwide survey on Test Process Improvement. Technical report, Sogeti.

[Koomen and Pol, 1999] Koomen, T. and Pol, M. (1999). *Test Process Improvement: a Practical Step-by-Step Guide to Structured Testing*. Addison-Wesley, New York, NY, USA.

[Laitenberger, 2002] Laitenberger, O. (2002). A survey of software inspection technologies. *Handbook on Software Eng. and Knowledge Eng.*, 2:517–555.

[Laitenberger et al., 1999] Laitenberger, O., Leszak, M., Stoll, D., and Emam, K. E. (1999). Quantitative modeling of software reviews in an industrial setting. In *METRICS '99: Proceedings of the 6th International Symposium on Software Metrics*, page 312, Washington, DC, USA. IEEE Computer Society.

[Lewis, 2004] Lewis, W. E. (2004). *Software Testing and Continuous Quality Improvement, Second Edition*. Auerbach Publications, Boca Raton, FL, USA.

[Liggesmeyer, 1995] Liggesmeyer, P. (1995). A set of complexity metrics for guiding the software test process. *Software Quality Journal*, 4:257–273.

[Liggesmeyer, 2002] Liggesmeyer, P. (2002). *Software-Qualität. Testen, Analysieren und Verifizieren von Software*. Spektrum Akademischer Verlag, Berlin, Germany.

[Lüttgen, 2006] Lüttgen, G. (2006). Formal verification & its role in testing. Technical Report YCS-2006-400, Department of Computer Science, University of York, England.

[Lázaro and Marcos, 2005] Lázaro, M. and Marcos, E. (2005). Research in software engineering: Paradigms and methods. In *CAiSE Workshops Vol. 2, Proceedings of the 17th International Conference, CAiSE 2005, Porto, Portugal*, pages 517–522. FEUP Edições, Porto.

[Malishevsky et al., 2002] Malishevsky, A., Rothermel, G., and Elbaum, S. (2002). Modeling the cost-benefits tradeoffs for regression testing techniques. In *ICSM '02: Proceedings of the International Conference on Software Maintenance*, page 204, Washington, DC, USA. IEEE Computer Society.

[Mao and Lu, 2005] Mao, C. and Lu, Y. (2005). Regression testing for component-based software systems by enhancing change information. *APSEC'05: Proceedings of the 12th Asia-Pacific Software Engineering Conference*, 0:611–618.

[Marquis et al., 2005] Marquis, S., Dean, T. R., and Knight, S. (2005). Scl: a language for security testing of network applications. In *CASCON '05: Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research*, pages 155–164. IBM Press.

[Meszaros, 2003] Meszaros, G. (2003). Agile regression testing using record & playback. In *OOPSLA'03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 353–360, New York, NY, USA. ACM Press.

[Michael et al., 2002] Michael, J. B., Bossuyt, B. J., and Snyder, B. B. (2002). Metrics for measuring the effectiveness of software-testing tools. In *ISSRE '02: Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE'02)*, page 117, Washington, DC, USA. IEEE Computer Society.

[Misra, 2005] Misra, S. (2005). An empirical framework for choosing an effective testing technique for software test process management. *Journal of Information Technology Management*, 16(4):19–25.

[Morasca and Serra-Capizzano, 2004] Morasca, S. and Serra-Capizzano, S. (2004). On the analytical comparison of testing techniques. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 154–164, New York, NY, USA. ACM.

[Muccini et al., 2005a] Muccini, H., Dias, M. S., and Richardson, D. J. (2005a). Reasoning about software architecture-based regression testing through a case study. *COMPSAC'05: Proceedings of the 29th Annual International Computer Software and Applications Conference*, 02:189–195.

[Muccini et al., 2005b] Muccini, H., Dias, M. S., and Richardson, D. J. (2005b). Towards software architecture-based regression testing. In *WADS'05: Proceedings of the 2005 workshop on Architecting dependable systems*, pages 1–7, New York, NY, USA. ACM Press.

[Muccini et al., 2006] Muccini, H., Dias, M. S., and Richardson, D. J. (2006). Software architecture-based regression testing. *Journal of Systems and Software*, 79:1379–1396.

[Munson, 2003] Munson, J. C. (2003). *Software Engineering Measurement*. CRC Press, Inc., Boca Raton, FL, USA.

[Myers, 2004] Myers, G. J. (2004). *Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA.

[Nagappan et al., 2005] Nagappan, N., Williams, L., Vouk, M., and Osborne, J. (2005). Early estimation of software quality using in-process testing metrics: a controlled case study. In *3-WoSQ: Proceedings of the third workshop on Software quality*, pages 1–7, New York, NY, USA. ACM Press.

[Neto et al., 2007] Neto, A. C. D., Subramanyan, R., Vieira, M., and Travassos, G. H. (2007). Characterization of model-based software testing approaches. Technical report, PESC/COPPE/UFRJ, Siemens Corporate Research.

[Ntafos, 2001] Ntafos, S. C. (2001). On comparisons of random, partition, and proportional partition testing. *IEEE Trans. Softw. Eng.*, 27(10):949–960.

[Nursimulu and Probert, 1995] Nursimulu, K. and Probert, R. L. (1995). Cause-effect graphing analysis and validation of requirements. In *CASCON '95: Proceedings of the 1995 conference of the Centre for Advanced Studies on Collaborative research*, page 46. IBM Press.

[O'Brien et al., 2007] O'Brien, L., Merson, P., and Bass, L. (2007). Quality attributes for service-oriented architectures. In *SDSOA '07: Proceedings of the International Workshop on Systems Development in SOA Environments*, page 3, Washington, DC, USA. IEEE Computer Society.

[Orso et al., 2004] Orso, A., Shi, N., and Harrold, M. J. (2004). Scaling regression testing to large software systems. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 241–251, New York, NY, USA. ACM Press.

[Paradkar, 1994] Paradkar, A. (1994). On the experience of using cause-effect graphs for software specification and test generation. In *CASCON '94: Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research*, page 51. IBM Press.

[Paul, 2001] Paul, R. (2001). End-to-end integration testing. In *APAQS '01: Proceedings of the Second Asia-Pacific Conference on Quality Software*, page 211, Washington, DC, USA. IEEE Computer Society.

[Peng and Wallace, 1994] Peng, W. W. and Wallace, D. R. (1994). *Software Error Analysis*. Silicon Press, Summit, NJ, USA.

[Perry, 2006] Perry, W. E. (2006). *Effective methods for software testing*. Wiley Publishing Inc., Indianapolis, IN, USA, third edition.

[Pezzè and Young, 2007] Pezzè, M. and Young, M. (2007). *Software Testing and Analysis: Process, Principles, and Techniques*. John Wiley & Sons, Inc, Hobokon, NJ, USA.

[Pol et al., 2002] Pol, M., Teunissen, R., and van Veenendaal, E. (2002). *Software Testing-A Guide to the TMap Approach*. Addison-Wesley, New York, NY, USA.

[Pusala, 2006] Pusala, R. (2006). Operational excellence through efficient software testing metrics. InfoSys White Paper, (http://www.infosys.com/IT-services/independent-validation-services/white-papers/operational-excellence.pdf).

[Rajan, 2006] Rajan, A. (2006). Coverage metrics to measure adequacy of black-box test suites. In *ASE '06: Proceedings of the 21st IEEE International Conference on Automated Software Engineering*, pages 335–338, Washington, DC, USA. IEEE Computer Society.

[Ramler and Wolfmaier, 2006] Ramler, R. and Wolfmaier, K. (2006). Economic perspectives in test automation: balancing automated and manual testing with opportunity cost. In *AST '06: Proceedings of the 2006 international workshop on Automation of software test*, pages 85–91, New York, NY, USA. ACM Press.

[Rico, 2004] Rico, D. F. (2004). *ROI of Software Process Improvement: Metrics for Project Managers and Software Engineers*. J. Ross Publishing, Inc.

[Rothermel et al., 2004] Rothermel, G., Elbaum, S., Malishevsky, A. G., Kallakuri, P., and Qiu, X. (2004). On test suite composition and cost-effective regression testing. *ACM Trans. Softw. Eng. Methodol.*, 13(3):277–331.

[Rothermel and Harrold, 1994] Rothermel, G. and Harrold, M. J. (1994). A framework for evaluating regression test selection techniques. In *ICSE'94: Proceedings of the 16th international conference on Software engineering*, pages 201–210, Los Alamitos, CA, USA. IEEE Computer Society Press.

[Rothermel et al., 2001] Rothermel, G., Untch, R. H., Chu, C., and Harrold, M. J. (2001). Prioritizing test cases for regression testing. *IEEE Trans. Softw. Eng.*, 27(10):929–948.

[Sassenburg, 2005] Sassenburg, H. (2005). *Design of a Methodology to Support Software Release Decisions: Do the Numbers Really Matter?* PhD thesis, University of Groningen, Netherlands.

[Sauer et al., 2000] Sauer, C., Jeffery, D. R., Land, L., and Yetton, P. (2000). The effectiveness of software development technical reviews: A behaviorally motivated program of research. *IEEE Trans. Softw. Eng.*, 26(1):1–14.

[Schmietendorf and Dumke, 2005] Schmietendorf, A. and Dumke, R. (2005). Complex evaluation of an industrial software development project. In *IWSM 2005: Proceedings of the 15th International Workshop on Software Measurement*, pages 267–280, Aachen, Germany. Shaker Verlag.

[Schulmeyer and MacKenzie, 2000] Schulmeyer, G. G. and MacKenzie, G. R. (2000). *Verification and Validation of Modern Software Systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[Siniaalto, 2006] Siniaalto, M. (2006). Test driven development: Empirical body of evidence. Technical report, ITEA, Information Technology for European Advancement.

[Sneed, 2004] Sneed, H. M. (2004). Reverse engineering of test cases for selective regression testing. *CSMR'04: Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering*, 00:69.

[Sneed, 2005] Sneed, H. M. (2005). Measuring the effectiveness of software testing: converting software testing from an art to a science. In *Proceedings of MetriKon 2005: DASMA Software Metrik Kongress*, pages 145–170, Aachen, Germany. Shaker Verlag.

[Sneed, 2007] Sneed, H. M. (2007). Test metrics. *Metrics News, Journal of GI-Interest Group on Software Metrics*, 12(1):41–51.

[Sommerville, 2007] Sommerville, I. (2007). *Software Engineering*. Pearson Education Limited, Harlow, England, 8th edition.

[Spillner et al., 2007] Spillner, A., Rossner, T., Winter, M., and Linz, T. (2007). *Software Testing Practice: Test Management*. Rocky Nook Inc., Santa Barbara, CA, USA.

[Suwannasart and Srichaivattana, 1999] Suwannasart, T. and Srichaivattana, P. (1999). A set of measurements to improve software testing process. In *NCSEC'99: Proceedings of the 3rd National Computer Science and Engineering Conference*.

[Swinkels, 2000] Swinkels, R. (2000). A comparison of TMM and other test process improvement models. Technical report, Frits Philips Institute, Technische Universiteit Eindhoven, Netherlands, (http://is.tm.tue.nl/research/v2m2/wp1/12-4-1-FPdef.pdf),.

[Tai et al., 1993] Tai, K.-C., Paradkar, A., Su, H.-K., and Vouk, M. A. (1993). Fault-based test generation for cause-effect graphs. In *CASCON '93: Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research*, pages 495–504. IBM Press.

[Taipale et al., 2005] Taipale, O., Smolander, K., and Kälviäinen, H. (2005). Finding and ranking research directions for software testing. In *EuroSPI'2005: 12th European Conference on Software Process Improvement*, pages 39–48. Springer.

[Tal et al., 2004] Tal, O., Knight, S., and Dean, T. (2004). Syntax-based vulnerability testing of frame-based network protocols. In *Proceedings of the Second Annual Conference on Privacy, Security and Trust*, pages 155–160.

[Tassey, 2002] Tassey, G. (2002). The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards & Technology.

[Tate, 2003] Tate, J. (2003). Software process quality models: A comparative evaluation. Master's thesis, Department of Computer Science, University of Durham, Durham, UK.

[Tian, 2005] Tian, J. (2005). *Software Quality Engineering: Testing, Quality Assurance, and Quantifiable Improvement*. Wiley-IEEE Computer Society Pres, Los Alamitos, CA, U.S.A.

[Tillmann and Schulte, 2006] Tillmann, N. and Schulte, W. (2006). Unit tests reloaded: Parameterized unit testing with symbolic execution. *IEEE Software*, 23(04):38–47.

[Utting and Legeard, 2006] Utting, M. and Legeard, B. (2006). *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[van Lamsweerde, 2000] van Lamsweerde, A. (2000). Formal specification: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 147–159, New York, NY, USA. ACM Press.

[van Veenendaal and Pol, 1997] van Veenendaal, E. and Pol, M. (1997). A test management approach for structured testing. *Achieving Software Product Quality*.

[Verma et al., 2005] Verma, S., Ramineni, K., and Harris, I. G. (2005). An efficient control-oriented coverage metric. In *ASP-DAC '05: Proceedings of the 2005 conference on Asia South Pacific design automation*, pages 317–322, New York, NY, USA. ACM Press.

[Wang et al., 1998] Wang, L.-C., Abadir, M. S., and Zeng, J. (1998). On measuring the effectiveness of various design validation approaches for powerpc microprocessor embedded arrays. *ACM Trans. Des. Autom. Electron. Syst.*, 3(4):524–532.

[Wang and King, 2000] Wang, Y. and King, G. (2000). *Software engineering processes: principles and applications*. CRC Press, Inc., Boca Raton, FL, USA.

[Whalen et al., 2006] Whalen, M. W., Rajan, A., Heimdahl, M. P., and Miller, S. P. (2006). Coverage metrics for requirements-based testing. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 25–36, New York, NY, USA. ACM Press.

[Wu et al., 2005] Wu, Y. P., Hu, Q. P., Ng, S. H., and Xie, M. (2005). Bayesian networks modeling for software inspection effectiveness. In *PRDC '05: Proceedings of the 11th Pacific Rim International Symposium on Dependable Computing*, pages 65–74, Washington, DC, USA. IEEE Computer Society.

[Xu, 2006] Xu, G. (2006). A regression tests selection technique for aspect-oriented programs. In *WTAOP'06: Proceedings of the 2nd workshop on Testing aspect-oriented programs*, pages 15–20, New York, NY, USA. ACM Press.

[Xu et al., 2003] Xu, L., Xu, B., Chen, Z., Jiang, J., and Chen, H. (2003). Regression testing for web applications based on slicing. *COMPSAC'03: Proceedings of the 27th Annual International Conference on Computer Software and Applications*, 0:652.

[Zelkowitz and Wallace, 1997] Zelkowitz, M. and Wallace, D. (1997). Experimental validation in software engineering. *Information and Software Technology*, 39(1):735–743.

[Zhang et al., 2004] Zhang, J., Xu, C., and Wang, X. (2004). Path-oriented test data generation using symbolic execution and constraint solving techniques. In *SEFM '04: Proceedings of the Software Engineering and Formal Methods, Second International Conference*, pages 242–250, Washington, DC, USA. IEEE Computer Society.

[Zhao et al., 2006] Zhao, J., Xie, T., and Li, N. (2006). Towards regression test selection for AspectJ programs. In *WTAOP'06: Proceedings of the 2nd workshop on Testing aspect-oriented programs*, pages 21–26, New York, NY, USA. ACM Press.

[Zheng, 2005] Zheng, J. (2005). In regression testing selection when source code is not available. In *ASE'05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 752–755, New York, NY, USA. ACM Press.

[Zhu, 2006] Zhu, H. (2006). A framework for service-oriented testing of web services. In *COMPSAC '06: Proceedings of the 30th Annual International Computer Software and Applications Conference*, pages 145–150, Washington, DC, USA. IEEE Computer Society.

[Zuse, 1998] Zuse, H. (1998). *A Framework of Software Measurement*. Walter de Gruyter & Co., Berlin, Germany.