KUOPION YLIOPISTON JULKAISUJA H. INFORMAATIOTEKNOLOGIA JA KAUPPATIETEET 15
KUOPIO UNIVERSITY PUBLICATIONS H. BUSINESS AND INFORMATION TECHNOLOGY 15

TANJA TOROI

# Testing Component-Based Systems

## Towards Conformance Testing and Better Interoperability

Doctoral dissertation

To be presented by permission of the Faculty of Business and Information Technology of the University of Kuopio for public examination in Auditorium L22, Snellmania building, University of Kuopio, on Friday 17th April 2009, at 12 noon

Department of Computer Science
University of Kuopio

KUOPION YLIOPISTO

KUOPIO 2009

**Distributor:** Kuopio University Library
P.O. Box 1627
FI-70211 KUOPIO
FINLAND
Tel. +358 40 355 3430
Fax +358 17 163 410
www.uku.fi/kirjasto/julkaisutoiminta/julkmyyn.shtml

**Series Editors:** Professor Markku Nihtilä, D.Sc.
Department of Mathematics and Statistics

Professor Hannu Tanninen, D.Sc. (Econ)
Department of Business and Management

**Author's address:** Department of Computer Science
University of Kuopio
P.O. Box 1627
FI-70211 KUOPIO
FINLAND
Tel. +358 40 355 2374
Fax +358 17 162 595
E-mail: Tanja.Toroi@uku.fi

**Supervisors:** Professor Anne Eerola, Ph.D.
Department of Computer Science
University of Kuopio

Professor Martti Penttonen, Ph.D.
Department of Computer Science
University of Kuopio

**Reviewers:** Professor Cem Kaner, Ph.D., J.D.
Department of Computer Sciences
Florida Institute of Technology, USA

Professor Markku Sakkinen, Ph.D.
Department of Computer Science and Information Systems
University of Jyväskylä

**Opponent:** Adjunct Professor Ville Leppänen, Ph.D.
Department of Information Technology
University of Turku

ISBN 978-951-781-994-7
ISBN 978-951-27-0113-1 (PDF)
ISSN 1459-7586

Kopijyvä
Kuopio 2009
Finland

Toroi, Tanja. Testing Component-Based Systems - Towards Conformance Testing and Better Interoperability. Kuopio University Publications H. Business and Information Technology 15. 2009. 59 p.
ISBN 978-951-781-994-7
ISBN 978-951-27-0113-1 (PDF)
ISSN 1459-7586

**ABSTRACT**

Interoperability between applications is extremely important within organizations and in networks of organizations. In addition, critical systems require high quality and reliability. Nowadays applications do not interoperate properly and their quality is often poor. There are frequent reports in the media about software systems crashing and damages occurring due to software errors. One reason for this is that there are many software testing methods and techniques but they are often non-practical and difficult to use. In addition, in networks of organizations it is often impossible to test unforeseeable side effects of the systems. Interoperability can be improved by standards and conformance testing. However, standards are often inadequate for software testing purposes.

The aim of the study was to improve existing testing methods and their practicality especially from the integrator viewpoint. The objective was to improve interoperability between applications, and familiarize software companies and their customers with conformance testing. This thesis examines component-based system testing from the integrator viewpoint. In component-based system development, components of different granularities must be tested. We give examples of UML-based test cases for components of different granularities. To ease the integrator's work, a conformance testing model was developed. The model was evaluated in software companies and their customer organizations. We noticed that more testable specifications are needed in order to make testing as automatic as possible, so testability requirements for the interface specifications are proposed. In addition, we make recommendations for the organizations about how to improve their testing processes. Recommendations are also made for improving the quality of applications, the interoperability between applications, and conformance testing activities.

The main contributions of the thesis are: 1) a systematic component-based system testing model for the integrator; 2) a rapid conformance testing model to test applications against interface specifications; 3) a list of testability improvements for the interface specifications; and 4) recommendations for the organizations about how they can improve the quality of the applications, and compliance to interoperability standards. These contributions can be used by software companies and integrators to improve their testing processes and the quality of the applications, and by software customers and authorities to contribute to software quality and interoperability.

Universal Decimal Classification: 004.05, 004.415.5, 004.415.53, 004.422

Inspect Thesaurus: software quality; software reliability; conformance testing; program testing; integrated software; software standards; information systems; software process improvement; software engineering; medical information systems

## Acknowledgements

This thesis is the result of research carried out in the Department of Computer Science at the University of Kuopio. The preparation of the thesis was financially supported by the Department of Computer Science at the University of Kuopio, and the Finnish Funding Agency for Technology and Innovation TEKES together with software companies and healthcare organizations who have participated in the PlugIT, OpenTE, and SerAPI projects. In addition, the work was supported by the Finnish Concordia Fund. I would like to thank all these organizations for the financial support.

I would like to thank my supervisors, Professor Anne Eerola and Professor Martti Penttonen for their guidance and support during the preparation of this thesis. Without their support this work would not have come into the world.

I would like to thank the reviewers, Professor Cem Kaner and Professor Markku Sakkinen for their valuable comments on the manuscript. I am grateful to Vivian Paganuzzi for editing the language.

I am grateful to the whole staff of the Department of Computer Science at the University of Kuopio for their support during this work. Special thanks belong to Marika and Irmeli (nowadays at the IT Centre) for the valuable discussions I had with them. Additionally, I would like to thank my co-authors, Anne, Marko and Juha, for their contribution to this thesis.

I also thank my friend Maarit for her encouragement and support, and for our discussions.

My warmest thanks belong to my mother Terttu, and my father Pentti who believed in me and encouraged me through the project. Finally, my dearest thanks I owe to my husband Mika, and my children Valtteri and Veera for they love and patience.

Kuopio, March 2009

_____

Tanja Toroi

## List of the original publications

This thesis is based on the following articles, which are referred to in the text by the Roman numerals I-V:

I Toroi T, Jäntti M, Mykkänen J, Eerola A. Testing component-based systems - the integrator viewpoint. In: *Proceedings of the IRIS 27, Information Systems Research Seminar in Scandinavia*, Falkenberg, Sweden, August 14-17, 2004.

II Toroi T, Mykkänen J, Eerola A. Conformance Testing of Open Interfaces in Healthcare Applications - Case Context Management. In: Konstantas D., Bourriéres J.-P., Léonard M., Boudjlida N., eds. *Interoperability of Enterprise Software and Applications*, p. 433-444. Springer-Verlag, 2006.

III Toroi T, Eerola A. Requirements for the testable specifications and test case derivation in conformance testing. In: Dasso A, Funes A, eds. *Verification, Validation and Testing in Software Engineering*, p. 118-135. Hershey: Idea Group Publishing, 2006.

IV Toroi T, Eerola A, Mykkänen J. Conformance Testing of Interoperability in Health Information Systems in Finland. In: Kuhn K, Warren J, Leong T-Y, eds. *Medinfo 2007*, Brisbane, Australia, August 20-24, 2007, p. 127-131. Amsterdam: IOS Press, 2007.

V Toroi T. Improving software test processes. In: Fujita H, Mejri M, eds. *New Trends in Software Methodologies, Tools and Techniques. Proceedings of the fifth SoMeT_06*, Québec, Canada, October 25-27, 2006, p. 462-473. Amsterdam: IOS Press, 2006. Vol. 147.

## Author's Contribution

For all the articles, the author of the thesis was the corresponding author and she wrote most of each article.

In Paper I, the method of level by level testing of software components from the lowest to the highest granularity is presented. The testing process is analogous at each level. Paper I emphasizes the role of the component integrator in component-based software testing and gives a method for the integrator to test component-based systems. In addition, examples of test cases of different granularities are given. The author of the thesis designed and wrote the paper. Examples were elaborated jointly with the co-authors.

In Paper II, a conformance testing model of open interfaces is presented. The model was developed and evaluated with the co-operation of software companies and healthcare districts. The author of the thesis outlined the paper and researched the background theory in the literature. The elaboration and evaluation of the testing model and conformance testing were performed jointly in the PlugIT project.

Paper III proposes new testability requirements for the interface specifications used in conformance testing. In addition, test case derivation from different kinds of specifications is considered. The author of the thesis researched the background theory of the testability of the systems and conformance testing. In addition, she outlined the way test cases are generated from different specifications. New testability requirements were elaborated jointly with the co-author.

In Paper IV, findings of a survey of conformance testing are presented. The idea of a survey study came from the OpenTE research group. The author of the thesis organized the survey and analyzed the results. The two other authors commented on the paper and contributed the healthcare viewpoint to it.

In Paper V, test process improvement is considered based on the models in the theory and on the results of the survey. The idea of the paper originated with the author of the thesis and the results were elaborated by her.

# Contents

# 1 Introduction

In the healthcare domain in Finland in the early years of this decade software systems were mainly monolithic, the complexity of the systems was increasing, time-to-market and efficiency were emphasized, and more test resources were needed (time, money, testers, tools). However, schedules were often shortened and there was even less time than before to plan extensive test cases and perform proper tests. All these features caused many problems especially in software testing and maintenance. Hence, monolithic systems were unpopular and systems modularity was promoted. Modularity supports composability and reusability (Meyer, 1988, 11-26). This means that different software parts can be changed without changing the rest of the system, and they can be reused in new environments. When systems are composed of different software parts, several new suppliers can offer their own software parts and customers can select the most suitable ones.

Increased modularity enabled different testing roles to arise, such as integrators, developers, and customers. It was usual that customers had business with several suppliers. However, different suppliers supplied systems which were not necessarily meant to be integrated with other systems. This caused integration and interoperability problems. Furthermore, there was uncertainty about how integrated systems had to be tested and who was responsible for it. Often, suppliers supposed that their customers would perform more testing, while customers supposed that the systems had been properly integrated and tested by the suppliers. These problems still exist.

At that time, software components with open interfaces started to became common in healthcare applications. As the use of open interfaces increased, the need to check that interfaces had been implemented according to the interface specification also increased. Thus, interface testing and conformance testing activities were requested. Additionally, evidence or brand was required to prove conformity to interface specifications. However, interface specifications were not so exact and unambiguous that conformity could be assured, so more accurate specifications were needed. All these challenges in the software industry led to the need for more efficient software testing methods and practices, and conformance testing activities had to be introduced in the healthcare domain in Finland.

The experiences of the end users reflect the quality of the software systems. We regularly read in the media about software systems crashing or software errors causing huge damages. For example, in 1996 the maiden flight of the European Ariane 5 launcher crashed about 40 seconds after takeoff (Jézéquel & Meyer, 1997) as a result of a software error caused by a floating-point conversion error. Another example relates to a software error found in the SAS check-in system in Helsinki-Vantaa airport in August 2006, which delayed dozens of flights and affecting about two thousand passengers. A third example relates to a defective emergency system which almost caused a death in Kuopio, Finland, in 2007. A pleurodynia patient

called the emergency number and asked for an ambulance, but the ambulance never received the request because the German emergency system that was in use then did not recognize Nordic characters. The system caused the request for an ambulance to disappear from the screen but no error message was displayed.

The quality of software systems can also be inferred from comments made by customers. The comments reveal a "seller's market" situation. For example, one healthcare software customer said: "From time to time there is a "take it or leave it" situation and testing does not help much. We know there are errors but we have no other choice than to buy the product". Another customer reported that "It is good that the systems work at least this way". The user experiences and customers' comments result from the fact that exhaustive testing of even a small and simple software code is impossible. Consequently, the end users' usage profiles and workflows, and the context of the resulting system, must be identified carefully when developing and testing software systems.

Several software testing techniques and methods have been developed (see Section 3.3). Specification-based testing techniques (Offutt & Abdurazik, 1999), such as finite-state machines, define test cases based on the system's specification. These techniques require that specifications are accurate and have been updated. Unfortunately, in most cases the specifications are not explicit. Code-based testing techniques (Hutchins et al. 1994), such as all-definitions, define test cases based on the source code. The problem with these techniques is that they do not verify code based on the users' requirements. The code can be correct but the system does not do what it is required to do. In addition, the persons involved, such as an integrator, do not all have the source code available, so they cannot use code-based techniques. Fault-based testing techniques, such as fault seeding and error guessing, try to demonstrate that pre-specified faults are not found in a program (Morell, 1990). Unfortunately, several of above techniques are so theoretical that testers cannot use them (Bertolino, 2004), which is why they have not been used very much in industry.

In software testing theory, single code segments or state machines are often studied, and the whole software system or dependencies between the systems are not tested. In theoretical studies, the complexity and largeness of the systems in the real world are not studied much. In this situation, it is not enough to just develop a new testing method: other influencing aspects must also be considered, such as attitudes towards testing, interface specifications, dependencies between components, responsibilities of authorities, and the whole testing process as a part of the whole software development process and software lifecycle.

There are many critical systems on which human lives depend in the healthcare domain. Thus, software quality and reliability are even more important in this domain than in the non-critical domains. The projects we have been working in concentrate on health information system research and development. A health information system is defined as one that is used in healthcare activities or organizations, with the aim of supporting high-quality and efficient patient care (Haux, 2006). The special characteristics in healthcare are high quality requirements, the safety criticalness of the systems, and non-deterministic and non-predictable processes, which often exceed organization borders. There are similar problems and a need for improvement in the critical systems in other domains, as well. The methods presented in this thesis can also be used in other domains, although not tested in the study.

## 1.1 Research questions

The justification for this study is that software systems have to operate properly and they must fit in the workflow of the customers. The thesis is meant for software developers and integrators, their customers, and authorities. When this study started, at the beginning of the millennium, component-based software development was quite a new trend in Finland. Thus, a starting point for the study was software testing in the component-based software development. When systems are composed of components, different roles can be identified in software testing, such as software developer, component integrator, and end user. The component industry made the integrator role necessary but few studies of software testing examine the integrator viewpoint in testing (e.g. Zheng et al. 2005). Therefore, there is a need for more knowledge about how software testing performed by the integrator differs from testing on the developer's side, and what kinds of methods the integrator needs when testing.

One important issue in component-based software is the interoperability of the components. In many organizations, huge amounts of information systems have to interoperate. Application integration, better interoperability, and avoidance of extra tailoring can be accomplished by common specifications and standards, open interfaces, and conformance testing. If all the interfaces were implemented according to the standards, the integration of systems into other systems could be performed almost always in the same way, reducing the need for tailoring and extra work. However, in many cases the standards are quite broad and open to various interpretations. Thus, standards and the other specifications have to be improved and conformance testing is needed to assure that implementations comply exactly with the standards, and that all the obligatory features have been implemented. Specification testability has been promoted by W3C and OASIS, for example (Rosenthal & Skall, 2002; W3C, 2005). Conformance testing has been studied and performed extensively in the telecommunication domain, for example (ITU-T, 1996), but much less so in healthcare, and it is particularly scarce in Finland (see Section 3.6). Therefore, in the course of this study the emphasis moved towards interoperability of applications and conformance testing.

Comparisons in software testing between small and large software companies have rarely been made. Moreover, the study performed in the PlugIT project (see Section 4.1), revealed that it was very important that customers perform more software testing. Therefore, we studied testing methods in organizations of different kinds and sizes, and the extent to which customers can influence conformity of the applications they acquire. So far, the customers' perspective has not been studied much. However, it is extremely valuable to take the customer’s viewpoint into account when improving software testing methods. Customers need different kinds of testing methods than developers because of the availability of testing materials and it is not the customers' task to discover the secrets of software testing trends.

To address these challenges, we formulated the following research questions:

1. *How can the software test processes and test process improvement models be improved?*
2. *How does the granularity of software components influence software testing in component-based software development?*

3. *How is software and conformance testing performed in the healthcare domain in Finland, and how can conformance testing and compliance to interoperability standards be improved?*
4. *How can interface specifications be improved to help in testing the conformity to specifications?*

This thesis identifies the features influencing software testing in component-based software development, with the aim of improving testing methods and their practicality, especially from the integrator point of view. Another aim is to familiarize software companies and their customers with conformance testing, and to provide them and the authorities with guidelines for improving conformance testing activities and the interoperability between applications. The ultimate goal is to improve the quality of applications through improved testing methods, and help testers in their work in the organizations of the software developers and their customers.

## 1.2 Research approach

Several research approaches can be used in software engineering research. Approaches which study a part of reality can be categorized into conceptual-analytical and empirical studies (Järvinen & Järvinen, 1995, 9). *Conceptual-analytical studies* analyze constructs, examine existing theories, models and frameworks, and make logical deductions. *Empirical studies* examine the present and the past states, or are constructive studies. *Studies examining the present or past states* include theory-testing and theory-creating studies. *Theory-testing studies* test whether a certain theory, model, or framework, which has been selected after a competition, describe a certain part of reality (Järvinen, 2004, 36-65). *Theory-creating studies* investigate what kind of theory, model, or framework best describes or explains a part of a reality (Järvinen, 2004, 66-97). Empirical studies in software engineering include several approaches, such as observational studies, formal experiments, case studies, and surveys (Kitchenham, et al. 2002). Case studies and survey research are used in this thesis. *Survey research* is used to collect information to describe, compare and explain knowledge, attributes and behavior (Pfleeger & Kitchenham, 2001). Surveys can be either supervised (e.g. interviews), where one researcher is assigned to each respondent, or unsupervised, such as mailed questionnaires. Our survey was unsupervised with emailed questionnaires. A c*ase study* is an empirical inquiry where a phenomenon is investigated and evaluated in a real-life context, especially when the boundaries between the phenomenon and the context are not clearly evident (Yin, 2003). A case study can involve single or multiple cases. In *constructive research* the aim is to develop new knowledge, methods, and tools based on the existing information. New solutions to existing problems are constructed and then evaluated. Constructive research can be performed with an action research emphasis (Järvinen, 2004, 98). *Action research* simultaneously aims at solving an immediate problem situation and expands scientific knowledge (Baskerville, 1999). It has two stages: a diagnostic stage, which involves a collaborative analysis of the situation, and a therapeutic stage, in which changes are introduced and effects analyzed. The action research process is cyclical and has five phases: diagnosing, action planning, action taking, evaluating, and specifying learning.

We used both conceptual-analytical and empirical approaches during this study. At the beginning, literary research was performed to find out what kinds of testing methods exist, what are their characteristics, and where they have been applied. Thereafter, empirical research and constructive research approaches were used in Papers I-III. In Paper I, a method to test components of different granularities from the component integrator viewpoint was introduced. In Paper II, conformance testing model was constructed and evaluated with several software companies and healthcare districts. In Paper III, requirements for more testable interface specifications were proposed. These were constructed and evaluated in workshops together with the software companies participating in the OpenTE research project (see Section 4.1). In Paper IV, survey research was used to find out the current state of the art in conformance testing of interface specifications. Based on the survey results recommendations were made for healthcare organizations, software companies, and authorities to improve conformance testing activities and interoperability between applications. In Paper V, constructive and action research approaches were used to improve software test processes in a software company that was co-operating with us.

The main research results in this thesis are software testing methods and recommendations. The results were evaluated in collaboration with software companies participating in the research projects (see Section 4.1). At the beginning and end of our study we performed a survey to clarify out the state of the art in software and conformance testing in Finland.

## 1.3 Structure of the thesis

This thesis consists of five research Papers and the summary, which has 6 chapters. Chapter 2 outlines the environment where component-based software systems are developed. Chapter 3 presents the theoretical background for software and conformance testing. Chapter 4 presents the practical experiments and conclusions made during this study. Chapter 5 introduces and summarizes the research Papers and their relationships. Finally, Chapter 6 summarizes the study and presents some future work.

# 2 General concepts in component-based software development

In the following sections we define the concepts used in this thesis. First, we describe software quality factors. Second, modularity and its sub-concepts are defined. Third, the concepts used in object-oriented approach are examined. Fourth, component-based software engineering and concepts related to it are described. Finally, we define concepts related to standardization and interoperability.

## 2.1 Software quality

We mean by a *software developer* a software organization that develops software for the use of end users. An *integrator* acquires software parts from the developers and also develops his/her own components. The integrator integrates components into a system and tests it as a whole before delivering it to a customer. A *software customer* buys software from developers or integrators and carries out acceptance tests. An *end user* (e.g. a physician) uses the system (e.g. electronic patient record) in his/her work.

The aim of software engineering is to find techniques to build quality software (Meyer, 1988, 3). Software quality is quite a broad and vague concept, and objective evaluation of software quality is not always possible. Therefore, the ISO 9126 standard has been developed for the evaluation of software. The standard has four parts, which address the following subjects: quality model, external metrics, internal metrics, and quality in use metrics. Part one, ISO 9126-1 (ISO/IEC, 2001), is based on work done by McCall et al. (1977), Boehm et al. (1978), and others who have defined a set of software quality factors. The ISO 9126-1 standard identifies six key quality factors (Pressman, 2005, 432-433): functionality, reliability, usability, efficiency, maintainability, and portability. *Functionality* measures the degree to which the software satisfies the needs and functional specifications. Functionality can be described by the following sub-attributes: suitability, accuracy, interoperability, compliance, and security. *Reliability* refers to the capability of the system to function under defined conditions for periods of time. Its sub-attributes are maturity, fault tolerance, and recovery. *Usability* refers to the ease of use of the system. Usability can be divided into understandability, learnability, and operability. *Efficiency* can be described by system resources (e.g. the amount of disk space, memory, and time) used. Its sub-attributes are time behavior, and resource behavior. *Maintainability* indicates how easy it is to identify and fix a fault in a system, and manage changes. Maintainability includes the following sub-attributes: analyzability, changeability, stability, and testability. *Portability* describes how easy it is to transport software from

one environment to another. Its sub-attributes are adaptability, installability, conformance, and replaceability. Some of these factors can be directly measured and tested (e.g. functionality), while others can only be measured indirectly (e.g. usability). Software testing is extensively discussed in Chapter 3.

## 2.2 Modularity

In the early days of programming, programs were constructed by creating a main program to control a number of subroutines. To reduce programming efforts, programmers in a project team reused subroutines during a project's implementation (Clements, 1995). *Software reuse* takes place when one or more software elements are applied from one system to another so that development and maintenance cost and effort can be reduced (Biggerstaff & Perlis, 1989). Reusable software elements include architectural structures, requirements, design, code, documentation, test cases, and so on. A prerequisite for software reuse is that reusable elements have been split into small parts, such as modules. *Modularity* means that software is not monolithic but divided into pieces (Meyer, 1988, 11-26). Modularity can be evaluated by the following five criteria: decomposability, composability, understandability, continuity, and protection. *Decomposability* indicates that a system can be divided into several subsystems: it reduces complexity. *Composability* means that software elements can be freely combined with each other to produce new systems: it promotes reusability. *Understandability* means that modules must be separately understandable: it increases maintainability. *Continuity* means that a small change to a system specification should affect only individual modules. *Protection* means that the effects of an abnormal condition occurring at run-time in a module will affect only that module. Parnas has presented criteria for decomposing systems into modules (Parnas, 1972): system must be divided into a number of modules with well-defined interfaces, modules must be designed so that changes to one module affect other modules as few as possible, each module must be comprehensible and simple enough, and it must be possible to develop each module independently.

## 2.3 Object-oriented software engineering

In the 1980s the object-oriented approach became popular. The entities in that approach were called objects and classes. An *object* can be defined as an entity which has a state (information) and which offers a number of operations (behavior) to examine or affect that state (Jacobson et al. 1992, 44-49). A *class* can be defined as a set of objects that share a common structure and a common behavior (Booch, 1991). In object-oriented design, the main goal is not to ask what the system does but to find out what objects are needed to get work done (Meyer, 1988, 50). Objects model reality, thus there is only a small semantic gap between reality and the system model (Jacobson et al. 1992, 42-43). This increases the understandability of the system. However, the object-oriented approach did not address the modularity problem very well. Applications were developed in an object-oriented way but the end user still received a monolithic application (Herzum & Sims, 2000, 12-18).

The object-oriented approach has different characteristics when compared with procedural programs, such as inheritance, polymorphism, message passing, state-

based behavior, encapsulation, and information hiding (Chen et al. 1998). Furthermore, the execution order of the methods is not necessarily predefined, and the structure of the object-oriented programs is different from that of procedural programs. Understanding a single line of code may require tracing a chain of method invocations through several object classes, and up and down the object hierarchy, to find where the work is really done (Wilde and Huitt, 1992). This complicates both testing and software maintenance. The dependencies in procedural and object-oriented software also vary (Wilde & Huitt, 1992). Procedural software has the following dependencies: data dependencies between two variables, calling dependencies between two modules, functional dependencies between a module and the variables it computes, and definitional dependencies between a variable and its type. In object-oriented software, besides the above, the following dependencies also need to be considered: class to class, class to method, class to message, class to variable, method to variable, method to message, and method to method (Wilde & Huitt, 1992).

## 2.4 Component-based software engineering

Software complexity has dramatically increased since the mid-1980s (Gao et al. 2003, 5). Nowadays, new systems are seldom developed from scratch, so software developers need cost-effective methods to construct complicated software systems. The software business is moving increasingly towards component-based software development (Brown & Wallnau, 1996). *Component-based software engineering* (CBSE) shifts the emphasis from programming software to composing software systems (Clements, 1995). In CBSE, complex systems are constructed by assembling them from software components according to a software architecture. In CBSE it is impossible to build a coherent system of interworking components without a comprehensive architectural model (Klingler, 2000). The s*oftware architecture* of a system can consist of several structures of the system, and is composed of software elements, the externally visible properties of those elements, and the relationships between them (Bass et al. 2003, 21). Some of the most common architectural structures of a system are decomposition, layered, process, deployment, and client-server.

The advantages of a component-based approach are the possibility to master development and deployment complexity, modularity, decreased time to market, the quality and reusability of software and its components, the composed services of components, and the scalability and adaptability of software systems (Herzum & Sims, 2000, 21-23). Furthermore, software suppliers can specialize in their strategic competitive edge and buy other properties as ready-made COTS (commercial-off-the-shelf) components. The greatest challenges in component-based software development are that suitable ready-made components cannot be easily found, and if they are found, they are not necessarily of good quality (Vitharana et al. 2003). In addition, components have different dependencies between them, source code is seldom available, and the target context of the components is not known. Maxville et al. have described a process to help an integrator to select the right components from a huge amount of third party components (Maxville et al. 2003). The method has the following phases: problem definition and requirements for the component (ideal component specification), short-list creation of candidate components, test case generation based on the ideal component specification, test adaptation, test execution,

evaluation of results, candidates' ranking based on the results and other suitability and context information, and reporting on the results. It is a manual process but their aim is to automate it as much as possible. The other method for component selection is called *a trusted third party*, which requires different roles to support component-based software development and each of these roles are responsible for the quality of the component (Stafford & Wallnau, 2001). The basic roles are component technology specifier, component technology implementer, reasoning framework developer, component implementer, and system developer. Paper I considers software testing in the CBSE approach.

One of the earliest definitions of a software component is that it is a cohesive module, i.e. it denotes a single abstraction, and is loosely coupled (Booch, 1987, 7). A more well-known and accepted definition is that of Szyperski (2002, 41): "A *software component* is a unit of composition with contractually specified interfaces and context dependencies only. A software component can be deployed independently and is subject to composition by third parties." Thus, the components interoperate with each other through interfaces. An *interface* defines the access points of the component and allows other components to use services provided by the component (Szyperski, 2002, 42-43). Components can have different types of interfaces (Sametinger, 1997, 71-76): a programming interface, a data interface, a user interface, and/or an interface to the infrastructure. Testability requirements for the interface specifications are proposed in Paper III. To use the interfaces, contracts are needed. A *contract* states what the client needs to do to use the interface and what the provider has to implement to meet the services promised by the interface (Szyperski, 2002). Contracts can be divided into four levels: basic or syntactic, behavioral, synchronization, and quality-of-service (Beugnard et al. 1999). *Basic contracts* specify the operations that a component performs, the input and output parameters each component requires, and the possible exceptions that might be raised during operations. A *behavioral contract* specifies the behavior of operations more precisely than a basic contract. It sets out preconditions, post conditions, and class invariants. A *precondition* has to be met before the execution of an operation. A *post condition* has to be met just after the completion of an operation. A *class invariant* will always keep its truth value throughout a specific sequence of operations: it constrains objects of a class and the state stored in an object. A *synchronization contract* specifies the dependencies between the services provided by a component. A *quality-of-service contract* specifies the quality features the service will respect, such as maximum response delay, average response, and the quality of the result. Contracts are discussed in Paper I.

Software components can be of different granularity levels (Herzum & Sims, 2000, 36-46). The most fine-grained software component, used in this thesis, is called a *lowest level component*. It is normally based on a distributed component technology (e.g., CORBA, J2EE, .NET) and has a well-defined build-time and run-time interface. The component can be addressed over the network at run-time. Furthermore, it may have dependency relationships to other components. A medium-grained software component is called a *business component* (BC): it "consists of all the artifacts necessary to represent, implement, and deploy a given business concept or business process as an autonomous, reusable element of a larger distributed information system". The BC conforms to a layered architecture style including user interface, business logic and resource level. The coarsest-grained software component is called a *component-based system* (CBS): this is a component whose parts are business components that constitute a viable system. Business components of the CBS can be

classified into functional layers such as, process, entity, and utility. The granularity of components is further discussed in Paper I.

An *implementation* is a realization of a technical specification (e.g. interface) or algorithm. Implementation can be realized by objects, components or services, for example. OASIS has defined a *service* as "a mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description" (OASIS, 2006).

## 2.5 Standardization and interoperability

The use of open interface specifications and information-exchange standards has increased. *Open interface* is a published interface through which applications can communicate. Features in the open interface are public but only the owner of the interface can change them. The open interfaces are usually based on standards or other specifications. *Standardization* is a process of developing common methods and approaches. It helps to increase interoperability and security of the applications, protect consumers and environment, and facilitate national and international trade. Furthermore, standardization helps to reuse pre-built, standardized software components in other contexts than those in which they were initially implemented. Examples of standards in software engineering are IT Service Management ISO/IEC 20000, software quality standards ISO/IEC 25000 series, and UML (Unified Modeling Language) ISO/IEC 19501.

In healthcare, new applications, measuring equipments, tools, and sensors, which collect data from the patients, are constantly developed. The collected data must be easily transferred to the electronic patient record. In order to manage this, open standard-based interfaces have to be implemented. ISO (International Organization for Standardization), IEC (International Electrotechnical Committee), CEN (The European Committee for Standardization), and HL7 (Health Level 7) co-ordinate standardization work of healthcare information systems. Examples of standardization work of healthcare informatics are Health Card ISO/TR 21549, Electronic health record ISO/IEC 20514, and Public Key Infrastructure (PKI) ISO/TS 17090. (SFS, 2008)

Standardization work is performed in working groups. ISO Health Informatics working groups concentrate on data structures, data interchange, semantic content, security, health cards, pharmacy and medicines business, devices, business requirements for Electronic Health Records, and global health information standardization. CEN Health informatics working groups study information models, terminology and knowledge representation, security, safety and quality, and technology for interoperability. (SFS, 2008)

Standardization increases interoperability. Interoperability means that a system is capable of executing services to other systems and utilizing services from other systems. *Interoperability* is defined by ISO/IEC as follows: the "capability to communicate, execute programs, or transfer data among various functional units in a manner that requires the user to have little or no knowledge of the unique characteristics of those units" (ISO/IEC, 2003).

Besides standardization, conformity assessment against standards is needed. *Conformance* is defined by ISO/IEC as the fulfillment of a product, process or service of specified requirements (ISO/IEC, 1996). It is usually performed by testing to see whether an implementation meets the requirements of a standard (Gray et al. 2000). Software testing can be performed against different criteria, such as performance, behavior, functions and interoperability.

Conformance testing differs in one fundamental way from other testing: the requirements or criteria for conformance must be specified in publicly available standard or standard-like specification. In this thesis, we use the term *specification* to mean standards and other standard-like specifications, not internal specifications during software development. The criteria for conformance are usually specified in a conformance clause section of a standard. A *conformance clause* is defined as a section of the standard that states all the requirements or criteria that must be satisfied to claim conformance. The conformance clause defines at a high-level, what is required of implementers of the specification. It refers to other parts of the specification or other specifications. It provides communication between the specification's creators, implementers, and users as to what is required. If the criteria for conformance are not specified, there can be no conformance testing. (W3C, 2005)

Interoperability testing and conformance testing are clarified in Section 3.6 and more further in Papers II, III, and IV.

# 3 Theoretical Background for Software Testing

## 3.1 What is software testing?

Software testing has been increasingly covered in books since 1972 (Gelperin & Hetzel, 1988). The meaning of the term has changed from debugging-oriented to evaluation and prevention-oriented testing. Myers defined *software testing* as a process of executing a program with the intent of finding errors (1979, 5). He claims that the testing process is destructive in that it tries to rip a program apart. Beizer has presented five phases how attitudes have progressed in software testing (Beizer, 1990, 4-6):

Phase 0 thinking - There is no difference between testing and debugging. In this phase there can be no effective testing, no quality assurance, and no quality. Phase 0 thinking is the greatest cultural barrier to good testing and quality software.

Phase 1 thinking - The software works. The purpose of testing is to show that the software works. Phase 1 represents progress because testing and debugging are distinguished. However, the objective is unachievable. You can not prove that software works in every case.

Phase 2 thinking - The software doesn't work. The purpose of testing is to show that the software does not work. Compared to purpose of phase 1 this purpose leads to strong and revealing tests. However, the trouble with phase 2 thinking is that you do not know when to stop testing.

Phase 3 thinking - Test for risk reduction. The purpose of testing is not to prove anything but to accept the principles of statistical quality control. The more you test, the more confidence you have in the product. You can release when that confidence is high enough.

Phase 4 thinking - A state of mind. In this phase testing is not an act but it is a mental discipline that results in low-risk software. In this phase testability is the goal because it reduces the labor of testing and testable code has fewer bugs than code that is hard to test.

Kaner broadened the definition of software testing so that "software testing is an empirical technical investigation conducted to provide stakeholders with information about the quality of the product or service under test" (Kaner, 2004). Kaner has also broadened the purposes of testing, such as to prevent damages, maximize the bug count, block premature product releases, help managers make ship/no-ship decisions, minimize technical support, assess conformance to specification, and assess quality. A

*test case* is usually defined as a set of test inputs, execution conditions, and expected results developed for particular objective (Kruchten, 2003, 198-199). Similar test cases, such as regression tests or certain specific functionality tests, can be grouped together into a *test suite*. We define a test case of different granularities in component-based systems in Paper I. In practice, it is not possible to test systems exhaustively, so test cases must be diversified and they must be designed accurately. Moreover, testers must know when test coverage is adequate. *Coverage* measures the extent to which a certain criterion is satisfied. Different test coverage criteria, such as statement coverage, branch coverage, path coverage, mutation adequacy, and interface-based criteria, have been presented by Zhu et al. (1997).

Test phases can be divided into module (unit) testing, integration testing, system testing, and acceptance testing. *Module testing* is a process of testing the individual subprograms, subroutines or procedures in a program on the level of source code. Module testing manages system complexity and efficiency, since attention is first focused on smaller parts of the program. Modules can be built and tested simultaneously and thereafter integrated into bigger units. Thus, module testing eases the task of debugging because if errors exist, they are known to be in a particular module.

In *integration testing*, modules tested in the previous phase are integrated into bigger modules and then integration tested. The focus is on the co-operation of the modules and their interfaces. Integration can be organized as a "*big bang*" by combining all the modules together or to test modules *incrementally* either from bottom-up or top-down. In incremental integration, drivers and stubs are needed. A *driver* corresponds to a module which calls the particular module under test. A *stub* corresponds to a module that is called by the module under test.

Binder has presented nine integration test patterns (Binder, 1999, 627-714): big bang, bottom-up, top-down, collaboration, backbone, layer, client/server, distributed services, and high-frequency integration. In *big bang integration* all components are tested at the same time. Big bang integration is not recommended unless just a few new components are added to a stable system. *Bottom-up integration* is the most widely used technique. In bottom-up integration components are added to the system under test in uses-dependency order so that components having the fewest dependencies are added first. It works well for small to medium systems. In *top-down integration* components are added to the system under test in control hierarchy order so that the top-level control object is added first. Top-down integration works for almost any architecture. In *collaboration integration* sets of components, which include in a particular collaboration, are added to the system under test. Collaboration integration works for almost any architecture, too. *Backbone integration* combines elements of top-down integration, bottom-up integration, and big bang integration. It aims to identify the components that support application control, the backbone, and application subsystems. The sequence of testing is based on this analysis. Backbone integration is especially suited to embedded applications. *Layer integration* verifies interoperability in a system that can be modeled as a hierarchy. It applies to layered architectures and combines elements of top-down and bottom-up integrations. In layer integration interfaces between adjacent layers are tested. *Client/server integration* tests interaction among clients and servers. Clients and servers are tested first with stubs in isolation. Then pairs of clients are tested with the actual server or servers. *Distributed services integration* is needed when the system under test includes many

components that run concurrently with no single locus of control. Distributed services integration is appropriate for decentralized networks. *High-frequency integration* is useful when a stable baseline is available and new code is being frequently added, such as in iterative and incremental development. High-frequency integration can be applied at almost any architecture.

At the *system testing* level, the purpose is to compare the system to its original objectives. System test cases are designed by analyzing the objectives and then formulated by analyzing the user documentation. System testing covers different forms of testing, such as performance testing, volume testing, stress testing, security testing, and recovery testing. *Acceptance testing* tests whether the product meets customers' and users' needs. Acceptance testing can be conducted by the end users of the system in a real environment (beta testing) or at the developer's site (alpha testing) (Myers, 1979).

Besides the above-mentioned testing phases, s*ystem integration testing* has to be performed when systems are integrated into the customer's environment. In the real world, organizations have several different applications which have to interoperate. System integration testing is needed to check the performance and interoperability of all the applications. For example, in hospital, electronic patient records and different clinical applications store customers' demographic information separately. When information (e.g. address) changes it should be changed into every application. This is not cost-effective and it is hard to maintain. The clinical applications must retrieve customers' information from one application through the open interfaces. If the applications use common standards and implement open interfaces their maintainability improves and application integration needs less local adaptation.

Kaner has reappraised some of the earlier views concerning software testing (Kaner, 2004). The most salient ones are: document manual tests in full procedural detain, specify the expected results for a test case, and design most tests early in development. The first claim is that manual tests should be fully documented so that they can be used by less experienced testers. Kaner calls this an industry worst practice because it takes a long time to document and maintain the written procedures, junior testers often miss suspicious behavior when they follow the test script like robots, and there is no evidence that indicates that scripted practice is a good training tool. According to Kaner, the problem with the second claim, that the expected results should be defined for every test case, is that one expected result is not enough because there can be different kinds of expected results, such as a program or system state, monitored outputs, and impacts on connected devices. In addition, expected results do not always exist, for example when testing memory leaks, or when the results cannot be calculated by hand. The problems with the third claim, that most or all tests should be designed early in development, are the following. Firstly, there are an infinite number of possible test cases. At the beginning of the project, testers' knowledge is at its minimum. The better the testers understand the product and its risk, the more wisely those few test cases can be selected. Secondly, at the start of the project the testers cannot know what kind of mistakes the programmers will make when programming. However, they will learn from experience which types of mistakes are common in the project, with these languages and tools and these programmers. This is valuable knowledge when designing test cases. Thirdly, almost always the program changes during its development. If the test cases are designed at the start of the project, extra work has to be done, which of course is a waste of time and money.

Defect management should be connected very closely to the lifecycle of testing. The *defect management process* includes defect prevention, deliverable base lining, defect discovery, defect resolution, process improvement, and management reporting (Quality Assurance Institute, 1995). If defect management is combined with testing, it becomes clear which test case found a certain error and which one did not even if it should have (an erroneous test case). However, establishing an organization-wide defect management process is a complicated task (Jäntti et al. 2006). Bugzilla (http://www.bugzilla.org/) and Testlog (http://www.testlog.com/) can be used together, for example, so that a test case and the errors found by it can be recorded.

The following sections consider software development processes and how software testing is included in those processes, software testing techniques, software testing in component-based development, regression testing, and conformance testing.

## 3.2 Software testing in the development process

There are several software development process models, where software testing is a part of a total software development process. The *waterfall model* was one of the first software process models, where a number of independent steps are carried out in sequence one after the other (Royce, 1970). The steps are system requirements, software requirements, analysis, program design, coding, testing, and operations and maintenance. Each of the stages produces a product, which is the input into the next stage. There is no input from the final stage to the other stages. The problem with this model is that testing comes at the end of the whole development process and customer feedback is received too late.

The *spiral model* consists of iterative cycles in which the same phases (e.g. requirements, design, prototyping, testing) are repeated at every cycle (Boehm, 1988). The spiral model emphasizes especially risk analysis at every cycle. Software testing has been especially noticed in the V-model (Forsberg et al. 2005, 109-116). In the *V-model*, corresponding test plans are designed at every construction phase, i.e. acceptance and system test plans in the requirement specification phase, integration test plans in the architecture design phase, and module test plans in the module design phase. In the testing phases, testing is carried out in reverse order to software construction, and it verifies the software with respects to its test plan. The problem is that testing is not performed until code has been generated, and in practice the testing and development phases do not follow each other straightforwardly.

Nowadays software development processes are increasingly performed *iteratively* and *incrementally*, so that a product is developed iteratively, enhancing and adding new functionality with each release (Pfleeger, 2001, 56-57). The Rational Unified Process (RUP) is one example of an iterative and incremental software development process (Kruchten, 2003; Jacobson et al. 1999). RUP supports six best practices for software development: develop software iteratively; manage requirements; use component-based architectures; model software visually; verify software quality continuously; and control changes to software. It uses the Unified Modeling Language (UML) for modeling (Booch et al. 1998). In addition, there are models in which software testing has been integrated into the software development process, such as TSP (Test design Stages Processed) (Tsai et al. 1997). In the TSP model, iterative test design stages are incorporated at each phase of the software development lifecycle.

When a development phase is completed, testing of the phase should also be completed at that time.

## 3.3 Software testing techniques

Testers need different testing techniques at different test phases. Software testing techniques can be divided into specification-, code-, fault-, and usage-based techniques (Juristo et al. 2006). *Specification-based testing techniques* (or black-box testing techniques) generate test cases from the specifications, and study whether the software meets the requirements. They consider the program to be a black box into which test inputs are entered, but do not tell anything about the path a certain input traverses inside a program. Examples of specification-based techniques are equivalence partitioning, boundary value analysis, and testing of finite-state machines.

- *Equivalence partitioning* means that the input domain of a program is partitioned into classes such that a test case of a certain value of each class is equivalent to a test case of any other value in that class. Test cases are selected so that at least one test case is selected from each class (Myers, 1979, 44-50).
- In *boundary-value analysis*, test cases are selected from the boundaries of each equivalence class (Myers, 1979, 50-55).
- A *finite-state machine* is used to model the behavior of a state-based system (Wagner et al. 2006, 63-64). It is composed of a finite number of states, transitions between those states, and performed actions. Test cases are developed on the basis of different transitions between states.

The advantages of specification-based testing techniques are that the tester does not need a source code for testing, and also the customer can use some of the techniques (e.g. boundary value analysis). The problems with specifications-based techniques are outdated and skeletal specifications, and uncovered code segments, i.e. if the techniques are not properly applied, test cases may not cover the whole functionality of the program.

*Code-based testing techniques* (or white-box testing techniques) study the source code and describe the code coverage: for example, whether all the statements/branches of the program are executed at least once. They do not tell whether the program is doing what the requirement specification says it is supposed to do. Code-based testing uses either control-flow criteria or data-flow criteria for test case generation. *Control-flow-based testing* techniques select test cases on the basis of the program's control flow. Examples of control-flow-based testing techniques are sentence coverage, branch coverage, condition coverage, and path coverage. *Data-flow-based testing techniques* explore the events related to the status of data objects (variables) during the program's execution. The essential events are the *assignments* of value and the *uses* of value, i.e. where the variables are defined and where they are used. Examples of data-flow testing techniques are all-definitions, all-c-uses, all-p-uses, and all-du-paths (c means computation, p predicate, and du definition-use pair). However, these techniques are quite theoretical and complex to use in practice. Furthermore, the customer and the integrator cannot usually use any of the code-based testing techniques because the source code is not necessarily available and even if it were there would be an enormous amount of code lines to go through.

*Fault-based testing techniques* can be divided into error guessing, fault seeding, and mutation testing (Juristo et al. 2006). In *error guessing*, the generation of test cases is guided by the tester's knowledge of what are the typical faults. This method requires that a tester has a lot of experience of different kinds of faults and domains. In *fault seeding*, errors are artificially introduced into a program to evaluate the quality of the test suite. The total number of errors can be assessed from the errors found. Fault seeding is quite risky because the seeded errors may stay in the program. In addition, one cannot actually say anything about existing errors on the basis of the seeded errors. We can find the same kind of errors as the seeded ones, but how can we know whether there are novel errors as well? In *mutation testing*, the faults are seeded into a program by creating many versions of the program, each of which contains one fault (Offutt et al. 1996). Program variants (mutants) are generated by mutation operators. *Mutation operators* represent the faults a program will likely contain. The goal is to find such test cases that cause each faulty program to fail. These test cases are the most useful. However, mutation testing is computationally expensive, which is why different versions of mutation testing have been developed, such as weak mutation and selective mutation. Furthermore, there are tools for different programming languages (e.g. for Java and Ada) to help mutation testing. The problem with mutation testing is that it is an indirect testing method. Testing can become ineffective if the mutants are not generated extensively or testers do not have knowledge of all the possible errors.

In *usage-based testing* (statistical testing), test cases are derived according to their probability of occurring in actual operation, expectations of use, and criticality of different functions. Usage-based testing can also be considered to be a black box testing method (Kouchakdjian & Fietkiewicz, 2000). Operations that the user is more likely to use during the execution or that are more critical than the others are tested more. Usage-based testing can be cost effective if a tester knows the users' most probable operations. Nonetheless, the other operations have to tested, too. If the probability of the errors is known, the operations that have a bigger probability of occurring should be tested more. However, if the users' workflow changes, it may happen that the new workflow has not been tested as properly as the old workflow and errors remain. Moreover, the workflows of different users (e.g. novices and experts) have to be tested equally.

It can be concluded that in order to test software properly, various software testing techniques are needed; code-based or specification-based testing alone is not enough. The existing testing methods do not necessarily fit in practice. The more critical and complex the software, the more diversified must be the testing techniques used.

## 3.4 Testing component-based systems

A component-based system is not monolithic: it contains components of different granularities (e.g. lowest level components, business components, and component-based systems), which are integrated with other components and into legacy systems with interfaces. In such situations, testing and documentation are even more important than in conventional software projects with monolithic applications.

When moving from legacy systems to component-based systems, interfaces and interface testing are needed. The components do not have to know each other's

implementation, only the content of the interfaces, i.e. syntax, semantics, and instructions for using the interface. The components can be implemented by the integrator or they can be acquired ready-made and then integrated into the one's own systems. If components are re-used in a new context, regression testing (see Section 3.5) is always needed (Weyuker, 1998). However, it is often forgotten that testing activities depend on who is performing testing, and thus on the availability and quality of the specifications and documentations. A tester may be a component developer, a component integrator or a customer (end user). The component developer usually has all the needed testing materials, so for the developer the component is a white box, and code-based and specifications-based testing techniques can be used. The integrator and customer seldom have a source code available. Thus, the component is a black-box with interfaces, and test cases must be developed based on the other available documents, such as design and interface specifications, user manuals, and contracts. Integration has two parallel parts: on the one hand, user interface components are integrated and tested, and on the other hand business logic and resource components are integrated and tested. Furthermore, the components' external interfaces have to be tested with operation calls, i.e. how the component behaves if it is called outside the component. The different actors and viewpoints in component-based testing are described in more detail in Paper I.

Dependencies between components can be controlled with the help of a dependency graph, which shows the dependencies between the components of the same granularity level, and assures that the whole functionality of the component has been covered by test cases. Without the dependency graph, some critical paths may remain non-executed. More details about dependency graphs and an example of dependency graph creation can be found in (Toroi et al. 2002).

## 3.5 Regression testing

*Regression testing* means retesting a new version of a system after code changes. According to earlier studies, at least 10% of software defects are due to code changes (Collefello & Buck, 1987), so regression testing is an important part of software maintenance. However, it is expensive, which is why only the modified parts and those influenced by modifications directly or indirectly have to be retested, and test cases should be selected as automatically as possible (Rothermel et al. 1997). This raises the regression testing selection problem, i.e. how to select the right test cases from the existing test suite. Trade-offs have to be made between the time required to select and run regression test cases, and the fault detections ability of the test cases. The following questions have to be answered (Graves et al. 2001): how do different regression testing techniques differ in terms of their ability to reduce regression testing costs and their ability to detect faults? What trade-offs exist between test suite size reduction and fault detection ability? When is one technique more cost-effective than another?

There exist several regression test selection techniques. *Code-based regression test selection techniques* can be applied to regression testing at the unit level. The technique is not suitable when large and complex components are tested, because it is difficult to manage all the information obtained from the code, testers must read and understand the code, and it is time-consuming. In addition, code-based regression

techniques are language-dependent, so more than one code-based regression technique may be needed for regression testing (Chen et al. 2002).

S*pecification-based regression test selection techniques*, in which test cases are designed based on information obtained from program specifications, can be applied to large and complex system regression testing (Chen et al. 2002). However, these techniques also have problems, such as they may not be as exact as code-based techniques, the specifications have to be up-to-date and accurate, in order to get good test cases, all the complex dependencies between components are not necessarily known, and all the modifications are not necessarily properly written down in the specifications.

The other regression test selection techniques are *retest all*, which selects all the available test cases for retesting, *random techniques*, which randomly selects n% of the test cases from the test suite (e.g. random50 selects 50% of the test cases), and *risk-based techniques*, which try to select test cases so that risks are minimized (Chen et al. 2002; Graves et al. 2001).

The regression testing process in an industrial environment usually has the following steps (adapted from Onoma et al. 1998).

*Software artifact modification*: Usually, the source code is changed but the specifications or design can also be changed.

*Test case selection and execution*: Test cases must be selected from a test suite to run regression testing. Different test selection techniques can be used here, or sometimes all the test cases are reused. After test case selection, the test cases are run. Test execution is worth automating since the number of test cases is often large.

*Failure identification by examining test results*: Test results must be examined to see if the modified software behaves as expected. If the result is not as expected, the code has to be examined to see if it has an error or if the test case is erroneous. Furthermore, it is necessary to identify precisely which components, versions, and modifications caused the failure (*fault identification)*.

*Fault mitigation*: Once the components that caused the failure are identified, the fault must be removed. After mitigation, new regression testing is needed to check that the code has not been adversely affected by the changes, and caused any side effects. However, if the fault is not serious it can be decided not to correct until next version.

*Test suite maintenance*: In the maintenance step, outdated and duplicated test cases are removed from the test suite.

The object-oriented paradigm introduces challenges in software regression testing. The complex relationships between the object classes make it difficult to identify the affected parts and the ripple effects when changes are made in object-oriented class libraries or programs. Kung et al. (1994) have described a method for automatically identifying the affected classes. The method is based on a reverse engineering approach.

Onoma et al. (1998) claim that “regression testing is probably the most commonly used software testing technique”. However, we discovered that very little regression testing is performed in Finland (see Paper V): as many as 28% of the developers in our study stated that they did not regression test applications. Moreover, 24% of the

IT customers did not acceptance test a new version of the application they acquired. Thus, we recommended that the software organizations and their customer organizations should increase their regression testing activities, and have a plan and strategies (e.g. a test case selection strategy and an exit strategy for regression testing) for regression testing (in Paper V). Furthermore, regression testing theory and practice have to become closer, so that researchers see what is the problem with regression testing (why applications are not regression tested) and organizations get directed training in testing techniques.

## 3.6 Interoperability and conformance testing

In this section we define interoperability and conformance testing, and explain why they are needed. Thereafter we describe what has been studied in conformance testing in the healthcare domain and consider the state of the art in Finland.

### 3.6.1 General

Nowadays, organizations need to integrate their applications and processes into the network of organizations. Too often, new systems are integrated into existing ones by tailoring them separately by point-to-point integration. This is expensive and inefficient in the long run. There is a need to agree about common standards and open interfaces. If the systems have open standard-based interfaces, their interoperability improves, introduction and integration become easier, and less local adaptation work is needed. *Interoperability testing* is defined as the assessment of a product to determine if it behaves as expected while interoperating with another product (Kindrick et al. 1996). It should be borne in mind that interoperability testing only assures that a previously tested set of systems are interoperating: it does not guarantee that other systems interoperate with these systems. To get a higher level confidence that the systems will successfully interoperate with other non-tested systems, conformance testing is needed.

Conformance testing is necessary in accomplishing effective and rigorous interoperability testing (Moseley et al. 2004). *Conformance testing* is a way to verify the implementation of the standards to determine whether or not there are any deviations from the standard (Rosenthal & Skall, 2002). It determines which areas of the standard are implemented correctly, thus promoting portability and interoperability. Conformance testing is always performed against publicly available standard, such as ISO/IEC 20514 (Electronic health record). However, it can also be performed against the standard-like official specifications, such as HL7 recommendations.

In conformance testing, software implementations are black boxes; only the interfaces and their relationship to the specifications are examined. In other words, when an interface receives a request, a test is carried out to see whether the interface can handle the request and respond to it correctly. Conformance testing is always performed against the specification, and testing is bound in scope by the specification. All the features mentioned in the specification have to be implemented according to it, and are tested (e.g. all the required interfaces exist, all their operations and parameters have been implemented, and they are of the right type). However, nothing else is tested. Conformance testing ensures the presence of the specified characteristics. The internal structure of the system is not accessible and thus not the focus of testing.

Therefore, conformance testing does not guarantee 100% interoperability, but it increases substantially our confidence in the system. In addition, conformance clauses (or specifications) typically only cover some aspects of interoperability. Implementation-specific features, such as infrastructure and techniques, are not specified in the specification, which influence interoperability. On the other hand, error and exception conditions cannot always be forced by interoperability testing (Moseley, et al 2004). It can be concluded that interoperability and conformance testing cannot be substituted for each other and both testing activities are needed. Customers benefit from conformance testing when they make request for proposals because applications become more interoperable, and comparison of the proposals becomes easier.

Today, standardization and conformance testing of open interfaces are emphasized at all levels, nationally and internationally. However, development is still in its infancy. Conformance testing and integration efforts also provide feedback on standardization (Chronaki & Chiarugi, 2006). If standards do not adequately address the problems in practice, they can be further amended with standardization organizations. The following Papers consider conformance testing: Paper II describes a conformance testing model of open interfaces, Paper III proposes a conformance testing environment, and Paper IV presents the results of a survey of conformance testing.

### 3.6.2 Conformance testing in the healthcare domain

Conformance testing has been performed extensively in the domains where specifications are based on formal languages or protocols, such as in the telecommunication domain (ITU-T, 1996) but the practice is not so well-established in other domains. In the healthcare domain, problems with conformance testing and the interoperability of healthcare applications have been noticed, for example, in the USA (CCHIT), Britain (NHS), and Denmark (MedCom), and in various organizations such as HL7 (Health Level 7) and IHE (Integrating the Healthcare Enterprise). The mission of the Certification Commission for Healthcare Information Technology (CCHIT) is to create an efficient, credible and sustainable product certification program (CCHIT, 2007). Under the CCHIT certification program, many Ambulatory EHR systems have already been certified against functionality, interoperability, and security criteria.

The mission of the national strategic program for IT in the National Health Service (NHS) is to deliver a robust infrastructure, including authentication, security, and confidentiality, to enable electronic booking of appointments and electronic transfer of prescriptions, and to deliver integrated care records services (NHS, 2002). One important feature of the NHS is the shift to more corporate and national approaches. This means that there is a national approach to procurement and implementation, and services need to conform to national standards and must interoperate with emerging national services.

MedCom was a long-term project (from 1995 to 2007) whose mission was to contribute to the development, testing, dissemination, and quality assurance of electronic communication and information in the healthcare sector (MedCom, 2005). Its aim was to enhance test tools, and effort was put into self-service in testing.

Health Level 7 (HL7) Conformance SIG (Special Interest Group) improves the interoperability and certification processes. HL7 provides a mechanism to specify

conformance for HL7 Version 2.X and HL7 Version 3 messages, and provides a framework for facilitating interoperability using the HL7 standard. Besides standards, HL7 support the development of national HL7 specifications.

Integrating the Healthcare Enterprise (ACC/HIMSS/RSNA, 2005) promotes the use of standards by developing integration profiles. *Integration profiles* are at a more accurate level than standards, so they help to implement the product according to standards. The IHE acts worldwide, and several nations have national IHE initiatives. At the moment, Finland does not belong to the IHE but HL7 Finland has established IHE Special Interest Group (SIG) in the summer of 2008. HL7 Finland IHE SIG disseminates information of the IHE integration profiles and IHE activities in Finland. At the moment, the most important activity in IHE SIG is to find out what integration profiles software companies and customers are interested in.

In Finland, conformance testing is considered to be very important and in need of improvement, but there is a reluctance to use external interface testing services (see Paper IV). Conformance testing activities in the healthcare domain are quite low at the moment, which is one reason why there are problems in the interoperability between different applications. However, the selection of the Social Insurance Institution as a national actor for health IT and the increased coordination by the Ministry of Social Affairs and Health may alleviate the confusion related to standards and conformance testing in Finland. Furthermore, to be able to test conformity more thoroughly and more reliably, more accurate and varied specifications are also needed. Each specification should focus on only certain aspects, such as interface, technical, contextual, or concepts (see Paper III).

The question that usually arises is, when must conformity to standards be proved officially, and when is the supplier's label of conformity enough (Rada, 1996)? In our case study (see Paper II) we examined the conformance testing process of context management interface specification, and tried to improve the interoperability between different applications. The examination revealed that more automated testing processes and more accurate and diverse test cases are needed. Moreover, it became clear that regulations by the competent authorities and demand by the market for certified interfaces are needed to make conformance testing common.

# 4 Experiments in practice

There is a gap between software testing theory and practice (Bertolino, 2004). In software testing theory, short pieces of software code or state machines are often studied, but interoperability between large applications and entire software systems are neglected. Furthermore, not enough attention is paid to the scope and complexity of the applications in the real world.

In this chapter we first introduce the research projects in which the results of the thesis have been developed and evaluated. Second, the practical experiments carried out with the conclusions are presented. We define test cases based on the UML test model and analyze the model, present the state of the art in software and conformance testing in Finland, and describe how test process improvement models can be improved. In all these cases, the viewpoint is complex systems operating in the network of organizations.

## 4.1 Research projects

The results presented in this thesis were obtained in three national research and development projects in Finland: the PlugIT (Plug IT: Healthcare Applications Integration) project, the OpenTE (Open Testing Environment) project, and the SerAPI (Service-oriented Architecture and Web Services in Healthcare Application Production and Integration) project. All the projects were funded by the Finnish Funding Agency for Technology and Innovation, Tekes. In addition, PlugIT was funded by 15 software companies and eight healthcare organizations, OpenTE was funded by seven software companies, two healthcare districts, and the Ministry of Social Affairs and Health, and SerAPI was funded by 14 software companies and four healthcare organizations.

PlugIT (2001 - 2004) concentrated on healthcare application integration. The objective was to contribute to better healthcare services through more interoperable clusters of software applications. The project aimed at three kinds of results: 1) practical solutions in the form of application program interface (API) specifications; 2) methods for further work beyond the project; and 3) a long-term centre of expertise for healthcare software companies and their customers. In the PlugIT project, a survey focusing on software engineering in healthcare software companies and healthcare organizations was conducted. The results of the survey imply that it is very important that not only the developers but also the customers are responsible for testing. In addition, there was a desire to increase the amount of conformance testing activities.

Individual applications and regional information systems need testing environments for their interfaces and integration. In the OpenTE project (2004 - 2006), conformance and interoperability testing methodologies were studied, the architecture of testing environments for web service-based implementations was specified, and reference implementations were developed. A testing environment gives feedback for software developers and developers of specifications, and increases the probability that products are implemented correctly.

In the SerAPI project (2004 - 2007), the integration approach developed in the PlugIT project was specialized with a more focused approach towards service-oriented architecture and Web Service technologies. In the project, open interfaces and software services were defined, and methods, reference implementations and modeling examples were developed. The project was also involved in national and international standardization work.

## 4.2 UML test model

To experiment with a UML test model (Jacobson et al. 1999, 297), we arranged a case study to test a healthcare application with UML diagrams (Paper I and Jäntti & Toroi, 2004). Our finding from the experiment is that using a test model together with equivalence partitioning reveals several serious defects in the system. However, if the documentation is poor, as it was in our case, establishing a UML test model for the legacy system is a big challenge and requires a good experience of UML modeling and enough domain knowledge. Quite similar results have been reported in a user survey and industry case study (Lange et al. 2006). However, Lange et al. claim that the large community of UML users is the evidence of the usefulness of UML.

We found the following advantages in using the UML test model. 1) Use cases (Schneider & Winters, 2006) provide a way to divide the system under test into smaller functional units, and test cases can be organized by use cases. Therefore, software testing becomes more systematic with a test model. 2) Well-organized test case documents increase the quality of the software product, and a project customer is able to see whether the system meets the requirements and how it has been tested. 3) Use case scenarios include descriptions of exceptional and alternative flows, which are often sources of defects. 4) Visual modeling helps testers to understand the structure and behavior of the system in a shorter time than without models. 5) States of the concepts and test cases can be easily identified from the UML state diagram and the state transition table, and transition coverage can be used to measure test coverage. 6) Activity diagrams show the different action flows that a tester must go through in testing.

Besides the advantages, some problems were found. 1) Unfortunately, often no system documentation or related UML diagrams are available. Software testers do not have time to draw UML diagrams in a testing phase. Even if they did, the diagrams would not measure the realization of the requirements. 2) Even if documentation exists, UML diagrams are often too abstract and simple for testing purposes, or they have not been maintained when the requirements changed. In many cases, the textual description of diagrams is a better source of supporting information of testing. 3) A test model should focus on behavioral diagrams, such as use cases, activity diagrams, and state diagrams because the dynamic (behavioral) defects are discovered in testing.

4) Testers often consider that establishing a UML test model for a legacy system is less exciting than establishing a test model for new software; building and testing something new sounds more attractive.

## 4.3 State of the art in software and conformance testing

While the PlugIT project was running, interoperability problems were recognized but conformance testing was almost totally missing in the healthcare domain in Finland. Therefore, we focused our study more on conformance testing, and a model for conformance testing of open interfaces was developed. A clinical context management interface specification was used here as an example (see Paper II).

It seems that only a few studies are based on software testing *practices* (Torkar & Mankefors, 2003; Ng et al. 2004; Groves et al. 2000; Runeson, 2006), or conformance testing in the healthcare domain (Chronaki et al. 2006; Chronaki et al. 2005). There are no studies in which both parties, i.e. software companies and their customers, are involved. We conducted a survey of software and conformance testing practices of healthcare software companies and of their customers in Finland in 2006 (see Papers IV and V). The aim of the survey was twofold: to study the state of conformance testing of the interface specifications and the use of standards, and to find out how software testing and test process improvement is performed in practice.

It was surprising that there were significant differences in conformance testing activities in different organizations. In some organizations the system and its specifications conformed to required standards, while in others it was only tested whether two particular applications could be integrated. However, standards were widely used in healthcare applications. In addition, increasing the use of standards and official specifications was considered to be very important. Interestingly, there was a big difference between small and large software companies in test case documentation. In small companies (fewer than 50 employees), the amount of documented test cases is not even a half of the amount of documented test cases in large companies (more than 250 employees). Furthermore, software customers had invested more in testing and used more rigorous methods than did small companies. The most interesting result was that component regression testing was performed rather rarely before the re-use of the component. The results of the conformance testing part of the survey are discussed in Paper IV, and the practice in software testing is discussed in Paper V.

## 4.4 Test process improvement

There are several software process improvement models which can be applied from the software testing viewpoint, such as TPI (Test Process Improvement Model), TIM (Test Improvement Model), and TMM (Testing Maturity Model) (Koomen & Pol, 1999; Ericson et al. 1997; Burnstein et al. 1996; Swinkels, 2000). The problem with most of the models is that they are too cursory and theoretical, so they cannot be used properly in practice. Consequently, the models are not widely used in software test process improvement (see Paper V). We performed test process improvement with the TPI model in a small software company in the telecommunication domain. The TPI model examines the test process from 20 different key areas in two to four levels.

Examples of the key areas are Test strategy, Estimating and planning, Metrics, Test tools, Reporting, and Test process management. In our case study, more flexibility was needed in the Moment of involvement, Metrics, and Communication key areas.

*The moment of involvement* key area requires that testing is started as early as possible in the software development lifecycle. In our case the object of testing consisted of several separate parts, such as subsystems, software and hardware platforms, software versions, language versions, and constraints by the domain of the product. If one part changes, all the ready-made test plans may have to be discarded. Therefore, it is not always best to start testing (planning test cases) as early as possible if the system under test consists of several frequently changing parts. However, this influences the maturity score of many TPI key areas.

In the *Metrics* key area, the resources used and activities performed are measured in hours. In our case, a customer did not pay for elapsed time on software development but for the quality of the product. Thus, it was not considered necessary to record in hours the time spent in performing activities. The company used other metrics to find out how much the quality costs (e.g. the number of errors found and the number of changes implemented).

The *Communication* key area measures the internal communication of the test team and communication within the organization about the quality of the test processes. In our case, the communication key area did not reach a good maturity level because the information *from* the other teams that was passed *to* the test team (e.g. changes in the implementation or delivery date) caused a lowering of the maturity score of the testing process. Even if internal communication within and outside the test team worked, a good maturity level cannot be reached if other teams/units do not share the information which influences the test team as well. Thus, the point is not the immaturity of the test process, but that of the general software development process.

Test process improvement is further discussed in Paper V.

# 5 Summary of the Papers

This chapter summarizes and reviews the original Papers, illustrates their relationships and presents their contributions. The Papers and their relationships are shown in Figure 1. The study started by examining the software testing of component-based systems (Paper I). Paper I shows how component-based systems can be tested, from components of the lowest granularity to those of the highest. Component-based system testing can be performed from different viewpoints. We examined testing from the integrator viewpoint because component-based development highlighted problems in application integration and testing. In addition, software testing has been studied extensively from the developer and end user viewpoints but rarely from the integrator viewpoint. Integration and testing can be made easier with open interfaces and common standards. Besides standards, conformance testing practices are needed. We developed a conformance testing model, applied it to conformance testing of context management interface, and evaluated it in Paper II. However, conformance testing cannot be performed efficiently if interface specifications are not defined clearly and unambiguously. We found that more testable specifications are needed, and we propose new requirements for open interface specifications in Paper III. The results of the survey of conformance testing showed that the interoperability of applications and conformance testing must be improved. Therefore, we give recommendations on how to improve them (Paper IV). However, improving conformance testing activities is not enough if the whole software testing process is performed badly. We study and analyze test process improvement models in Paper V.

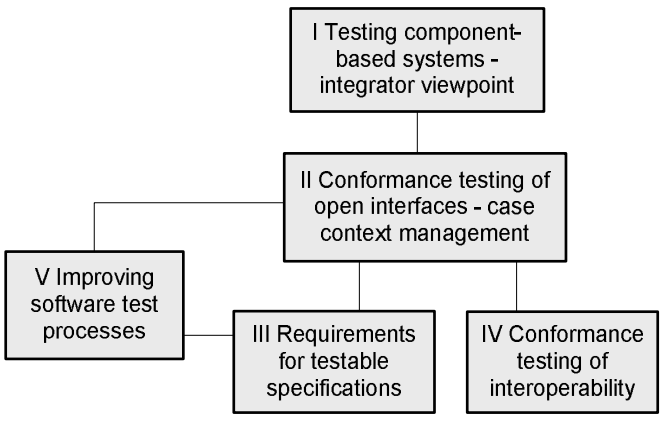


Figure 1: The Papers and their relationships to each other

## 5.1 Testing component-based systems - the integrator viewpoint

Different actors (roles) perform software testing at different phases of the software development process, i.e. software developers, component integrators, and customers (or end users). Test methods must be selected on the basis of the testing roles. The selection is influenced by whether the source code is available (developer), whether the application is integrated into existing system (integrator) or whether acceptance testing is based on workflow (customer). Paper I outlines how component-based systems are tested step by step, especially form the integrator viewpoint, and gives examples of test cases of different granularities based on UML diagrams.

We utilize and modify the business component approach introduced by Herzum and Sims (2000), and describe components of different granularities, i.e. a lowest level component, a business component, and a business component system. The granularity hierarchy means that a component-based system is a composition of business components, which in turn are compositions of the lowest level components. In the testing approach, white box testing and black box testing occur alternately at each level, utilizing test cases of different granularities and dependency graphs.

Figure 2 shows components of different granularities from the integrator viewpoint. The lowest level components (2a) are those that the integrator acquires or builds. If the integrator acquires components, they must have been tested by the component developer. The developer performs white-box testing and creates a test report. The test report should be enclosed with the component when the integrator acquires it. After the acquisition, the integrator acceptance tests the component as a black box. For the acquired components, only the interfaces are available, not the inner implementation of the component. The integrator has to evaluate the quality of the components with the help of the available test documents and user manuals through black box testing methods. The functionality and high quality requirements of the component influence the level and thoroughness of the evaluation. If the integrator develops his/her own components then the testing technique is the traditional white-box testing. Business components (2b) are those which the integrator usually builds from the lowest level components. The business component can be seen as a white box which contains several black boxes (components). The integrator integrates the lowest level components and performs integration testing, where the relationships between components are tested. Thereafter, the external interfaces of the business component are tested. A component-based system (2c) is the assembled application that the integrator deploys to the customer. The integrator integrates self-made and acquired business components into a business component system and tests their relationships, then the external interfaces are tested. The integrator has to use various testing methods at all different component levels. This incurs challenges to testing. The customer who acquires a component-based system performs acceptance testing for it. This is performed using black-box testing methods.

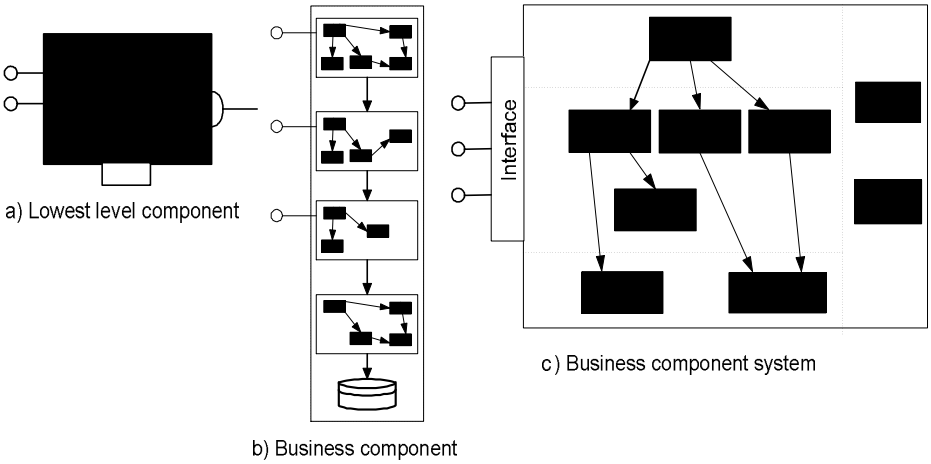


Figure 2: Components of different granularities from the integrator viewpoint

Since the components are of different granularities, the test cases must also be at different granularity levels. We give examples of test cases of different granularities based on our case studies with UML diagrams. The test case is a set of test input and expected result pairs, and execution conditions at every granularity level. Execution conditions contain environmental needs, preconditions, post conditions, and invariants. An ordered sequence of test input and expected result pairs is:

- an *action flow* between co-operating users and business components at the component-based system level.
- an *operation flow* between one user and the lowest level components at the business component level.
- a *method flow* between object classes inside the component at the lowest component level.

Nowadays, information systems are seldom made from scratch: they are usually assembled piece by piece into an existing system. Then the testing process differs from the one, where the module testing phase precedes integration testing, followed by the system testing phase. Thus, it is valuable to consider the integrator viewpoint in software testing and give him/her tools to test systems in practice in the software industry.

## 5.2 Conformance testing of open interfaces in healthcare applications

Application integration is often a combination of problems, where each organization has its own set of issues which must be dealt with (Mykkänen et al. 2004). In this study the research problem was, how can we ease the workload of the integrator and reduce the extra work without sacrificing software functionality and quality? In order to integrate different software applications without too much extra local adaptation work, open standard-based interfaces and conformance testing are needed. We propose a conformance testing model, developed and evaluated with the co-operation of software companies and healthcare districts. The model consists of four phases: an

initial phase, testing performed by the developer, testing performed by the testing lab, and certificate (brand) issuing. In the initial phase, a software customer asks developers to append the certificate or brand of interoperability to the request for proposals. Developers make their proposals and the customer selects and approves the best one. The developer develops an application and applies for the brand for the application. The testing lab performs conformance testing and issues the brand if the application meets the requirements.

The following challenges were faced during the conformance testing process. 1) Software product versions are introduced in rapid cycles, so the brand has to be renewed often and re-evaluation has to be performed automatically using, for example, web-based testing services. However, software companies are not very keen on public conformance testing, maybe because of the fear of revealing the testing results. Thus, regulation by the authorities or demand by the market for certified interfaces are the only effective ways to make certification and interface "branding" common. 2) Integration specifications must contain enough information about the requirements for the solutions, in addition to mere interface signatures. Integration specifications and standards should be developed to express clearly which options are implementation-specific or optional. In addition, specifications should provide guidance on how implementation-specific features should be documented and used. 3) Customers need advice when gathering and setting their requirements. Interface specifications do not currently contain all the information needed, including basic requirements, conformance levels, and different types of parameters which must be conformed to. 4) Some parameters are static for a given environment, but several are specific to the applications used or to the implementation or infrastructure of the server. A standard way to identify and classify this sort of parameters for test case definitions is needed.

During the conformance testing process the most important finding was that interface specifications have to be improved so that they are more accurate, more understandable, and more testable before conformance testing can be performed efficiently.

## 5.3 Requirements for testable specifications and test case derivation

In this study we propose new testability requirements for interface specifications and evaluate test case derivation from different specifications. In addition, one research problem was how to develop a reliable conformance testing environment for healthcare applications. The environment should be applicable in the software industry and healthcare organizations.

The specifications have to be defined so that testability and conformance are taken into consideration early in the specification lifecycle. W3C and OASIS have given guidance on how specifications should be written (Rosenthal & Skall, 2002; W3C, 2005). We propose the following additional aspects which the specification must address. *Architecture description* describes in which context the specification is being used and all the dependencies on the other specifications. Specifications have to be divided into *levels* so that functional, conceptual and technical details are distinguished from each other. All the issues concerning software *interoperability*

have to be mentioned in the specification. Also, the quality requirements (e.g. continuous operating time, execution time, robustness, usability, response time) must be stated. In the specification we must describe how different versions of implementation, specification, and test suites affect conformance, that is, in which situations *re-evaluation* of conformity is needed. *Key words* are used to specify requirements in the specification. Requirements can be mandatory (MUST), optional (MAY), or recommended (SHOULD) (Rosenthal & Skall, 2002). However, the key word SHOULD must be avoided because it is confusing and causes interoperability problems.

There are several formats of interface specifications, such as formal languages, semiformal languages and natural languages. The more formal the format, the more automatically test cases can be derived. However, the most formal specifications are not the best formats in practice because both developers and customers consider them too complex to use and understand. Semiformal languages are the most suitable for the healthcare domain because their notation is more understandable and they are easier to use and interpret than formal languages. In addition, they are more precise than natural languages.

## 5.4 Conformance testing of interoperability in health information systems

Conformance testing has been studied extensively in the telecommunication domain (ITU-T, 1996) but the practice is not well-established in other domains. Only a few studies on conformance testing in the healthcare domain (e.g. Chronaki & Chiarugi, 2006; Chronaki, et al. 2005) have been reported. No empirical studies on the level of conformance testing in software companies and in their customers' organizations in healthcare have been reported. Therefore, we conducted a survey to find out how much and how often conformance testing is performed in relation to the interface specifications in the healthcare domain in Finland. We also wanted to find out how customers perform testing in their organizations, and how they can influence the conformance of the software they acquire. Based on the survey results, we make the following recommendations for healthcare organizations, software companies, and authorities to improve conformance testing and compliance to interoperability standards.

*Perform interoperability conformance testing more rigorously.* Various conformance testing activities must be performed and they must be more diversified and disciplined than previously.

*Utilize open interfaces and use interface testing services*. The developers can use interface testing services before the interface release, while the customers can use them during the software acquisition and introduction. Proper interface testing services promote cost-effective interoperability testing. A well-established testing service allows everyone to select the needed counterparts of the application interfaces and their correct versions, and interoperability testing can be performed against them.

*Provide proper skills and knowledge*. Developers must be provided with knowledge of conformance testing through open seminars and training. In addition, customers must be provided with knowledge and skills to demand standard-based interfaces and certified software products, and the knowledge to test them.

*Recommendations must be enforced by the authorities.* Official recommendations and laws are needed to promote interoperability and the quality of the applications. Therefore, recommendations made by the authorities have to be introduced and enforced more actively. In addition, it is important to support and supervise the implementation of the recommendations, and facilitate it with tools.

*Organize autonomous and unbiased interface testing services.* Conformance testing must be unbiased and as automatic as possible. To accomplish this, studies and projects are needed in which interface testing services are organized objectively and independently from software companies.

*Reuse testing experiences and information*. Testing experiences and information should be transferred between similar customer organizations. One feasible idea is a mentor activity, in which the introduction of the applications has been distributed between different hospital districts and the experiences from one district are utilized in another.

*Improve the testability of specifications and standards*. Utilizing standards successfully and testing against standards requires accurate guidelines and constraints to generic standards, such as HL7 specifications. One way is to complement specifications with IHE (Integrating the Healthcare Enterprise) integration profiles.

Unfortunately, we had quite a small number of software companies in our survey. The questionnaire was sent to all the hospital districts (customers) in Finland and all the software companies (developers) belonging to a Health and Welfare IT R&D cluster in Finland. We obtained responses from 15 hospital districts (75% of all districts) and 14 software companies (52% of the number of companies in the cluster). However, it can be concluded that there are many problems to be solved before the applications in the healthcare domain interoperate appropriately and conform to standards. It heavily seems that laws and regulations are required in order to motivate the software providers to improve software testing and the interoperability.

## 5.5 Improving software test processes

The whole software testing process must be considered when interoperability is to be improved. Test process improvement was ranked as the most important issue in software testing research (Taipale, 2007). In this study, we introduce and analyze three of the best known test process improvement models (TMM (Burnstein et al. 1996), TIM (Ericson et al. 1997) and TPI (Koomen & Pol, 1999)). We investigated whether the models are dependent on domains, how well they are suited to practice, and whether there is any need to improve them. The TPI model is more practical than the other two. It appears that the models are applicable to different domains, such as telecommunication and healthcare. However, we noticed that there are some difficulties to apply the models in the small organizations. Quite similar results with large and small organizations have been reported in Dybå (2003).

Based on the results of the survey presented in Section 5.4, we found that test process improvement is considered to be very important. However, the maturity of the test processes is still surprisingly low. The most troublesome issues hindering software testing are lack of resources, especially time and knowledge, lack of documentation, such as proper requirement specifications and test case specifications, proper testing methods (test culture), insufficient test environments, attitude problems

(testing is undervalued), and rush to deliver incomplete applications to the customers. We make the following recommendations for organizations which want to overcome these barriers in software testing. 1) Test case documentation has to be increased and the organizations have to take a more positive attitude towards test case documentation. 2) More regression testing must be performed. The organizations have to plan regression testing and have a regression test strategy. 3) The level of training in testing must be improved. Training organizations have to study what kind of competence software organizations need, and plan their courses utilizing this information. 4) The clarity and unambiguity of the specifications have to be improved, and testers must participate in this phase. When developing specifications, people from different domains have to be involved. Furthermore, test case development simultaneously with the development of the specifications is extremely important. 5) Information and guidance about test process improvement models must be shared. We have good experiences of test improvement in cooperation with engineers, researchers, and customers.

Usually, the test improvement models do not cover all the software testing aspects that are important from the testing process improvement viewpoint in component-based software development. Therefore, we suggest new aspects and aspects that must be considered more deeply when improving test processes: interoperability testing, conformance testing, connections between test processes and service management (e.g. Information Technology Infrastructure Library, ITIL), regression testing, specifications, and the architecture of the applications.

## 5.6 Summary of the results

Here we consider how the research results answer the research questions presented in Section 1.1. The first research question was: How can the software test processes and test process improvement models be improved? Software quality has been studied extensively and there are many software testing methods and techniques which can be used in quality assurance in the software test process. However, the methods are often theoretical and difficult to apply to practice in the software industry and in customer organizations. That is one reason why software quality is often poor. A brand-new testing method is not needed but we need to examine existing methods and techniques, adopt suitable ones and adapt them to our own process. We gave organizations several recommendations to improve their testing methods. One recommendation is that testers must get training in testing, especially in UML-based test cases. In addition, those responsible for education should tailor their training programs to make them suitable for hands-on testers. In order to improve software quality, all the stakeholders (e.g. developers, customer, managers) must be committed to the testing process. Furthermore, a test team must introduce a test process improvement model and check its own test process against it regularly. After the evaluation, plans must be generated specifying the level to which the team will aspire. One important issue in increasing software quality is regression testing: more is needed, and test teams must have a strategy for regression testing.

In general, the test process improvement models do not cover software testing aspects that are important in the component-based development. Therefore, we suggested aspects that should be considered more deeply when improving test

processes, such as interoperability testing, conformance testing, testing of open interfaces, certification, and the authorities' requirements and their test cases.

The second research question relates to the granularity of software components and how it influences software testing in component-based software development. The proliferation of software components of different granularities enabled the integrator role to arise. Integrators need different kinds of testing methods than developers, because source code is not necessarily available to them. Therefore, testing methods are usually based on black-box testing. We developed a component-based system testing model with which the integrator can test systems piece by piece. The developer tests the finest-grained components, while the integrator usually tests the medium-grained (business component) and coarsest-grained components (component-based system). The finest-grained components are tested on the basis of a method flow between object classes inside a component. The medium-grained components are tested on the basis of an operation flow between the lowest level components belonging to the business component. The coarsest-grained component and the relationships between its inner business components are tested on the basis of an action flow which can be derived from an activity diagram, a use case diagram or a sequence diagram. There are several dependencies between components. The dependencies can be tested with a dependency graph. The granularity of the software components improves modularity, facilitates software testing, and assists software quality management.

The third research question relates to software and conformance testing in the healthcare domain in Finland. We studied how software and conformance testing is performed at the moment and how conformance testing and compliance to interoperability standards can be improved. Fortunately, we found that software testing is considered the most important technique in quality assurance. However, the level of software and conformance testing vary very much in different organizations. In some software companies, software testing is really diversified, while in others applications are not tested properly. It is usual that too little time is scheduled for testing both within software companies and in the customer organizations but nothing is being done to improve the situation. Furthermore, small and large software companies document their test cases differently. In small software companies test cases are not documented so often. However, defects are documented quite well and they are compared with the defect statistics of the previous versions of the applications. With the help of defect statistics, software testing can be focused on the most error-prone areas in the next software versions.

It appears that there are various problems to be solved before application conformity to standards and interoperability can be tested properly. The term *conformance testing* is not understood by all stakeholders. Some developers muddled the term up with traditional software testing. We developed a rapid conformance testing model to ease the conformance testing process. Based on the model, specifications for an open testing service, which acts as a counterpart in interface testing, were developed. However, we found that requirements issued by the authorities, laws, and sanctions are needed to get testing services into use and to ensure interoperable systems. Therefore, we made several recommendations for organizations to improve conformance testing and compliance to interoperability standards. Conformance testing can be performed against any kind of specifications, such as interoperability standards, process descriptions, or generic requirements for

the structure of the system. Thus, application interoperability requires various conformance testing activities. Conformance testing requires standard-based applications, customer demand, and investments in testing improvement. In addition, customers must be instructed to demand standard-based applications in their requests for proposals.

The fourth research question relates to interface specifications and especially, how specifications can be improved to help testing of conformity? The form of the specifications ranges from formal specifications to natural language statements. We examined their suitability in the healthcare domain and compared the ability to generate test cases of them. The more formal the specification, the more testable it is. On the other hand, when formality increases, understandability decreases. Test case generation based on natural language specifications is awkward, even impossible. However, the specifications used in conformance testing in the healthcare domain are often natural language specifications because customers and end users understand them better and can use them in requests for proposals and negotiations for an agreement. Thus, the development of the specifications has to be invested in, too, and testers must participate in this phase. We proposed several important issues that the specifications must address. The testability of the specifications has to be taken into consideration early in the specification's life cycle. Architecture description, interoperability factors, the conformity re-evaluation strategy, and special key words must be added to the specifications to assist in the testing process.

# 6 Conclusions and future work

The aim of the study was to improve existing testing methods and their practicality especially from the integrator point of view, to improve the interoperability between applications, and to familiarize software companies and their customers with conformance testing. We developed a component-based system testing model, with which the integrator can integrate and test systems piece by piece, and a conformance testing model, which demonstrates how applications can be tested against interface specifications. The conformance testing model was developed and evaluated in healthcare software companies and hospital districts. In addition, we made recommendations for the software companies and the relevant authorities about how to improve interface specifications, interoperability, and the quality of the applications.

## 6.1 Contributions of the thesis

The main contributions of the thesis can be considered through the research questions. The first research question was: How can the software test processes and test process improvement models be improved? We found that there are many theories of different software testing methods but still software quality is often poor. The existing methods are often theoretical and difficult to apply to practice. Practical solutions and recommendations are needed in the software industry and in customer organizations. We gave organizations various recommendations to improve their software processes and software quality, including recommendations for test case documentation and for regression testing. Furthermore, software testing has to be considered as a part of continuous software process improvement.

The second research question relates to a granularity of software components and how it influences software testing in component-based software development. We noticed that granularity helps software testing because it promotes modularity. The granularity of software components led to different roles being created. We highlight the integrator and customer roles in software testing because their viewpoints have not been extensively studied, and we developed a systematic component-based testing model for the integrator. A component-based viewpoint also created a need for granular test cases. Test cases are formed from an input flow which may be at user action level, operation level, or method level. We give examples of input flows of different granularity levels.

The third research question concerns software and conformance testing. We developed a rapid conformance testing model with the co-operation of several

software companies and hospital districts. During the evaluation, the organizations tested their application conformity to HL7 clinical context management interface specification. On the basis of the model, specifications for an open interface testing service were developed. Thereafter, a survey to discover how software and conformance testing is performed at the moment in Finland was conducted. On the basis of the survey results we made several recommendations about how to improve conformance testing and compliance to interoperability standards. The recommendations are meant for organizations and the authorities who aim to improve conformance testing and the interoperability between applications. They concern open interface utilization, enforcement of official recommendations and laws, interface testing service organization, the testability of the specifications and standards, and the reuse of testing experiences, for example.

The fourth research question concerns interface specifications and how they can be improved to help testing of conformity. We found that the existing interface specifications in the healthcare domain are often insufficient and testing cannot be performed against them. We identified several issues that the specifications must address, such as the target architecture of the system, specification levels, interoperability features, and re-evaluation strategy. In addition, we examined test case derivation from the specification, especially in the healthcare context. Specifications of formal languages, semiformal languages and natural language were studied. We considered their suitability in the healthcare context, and compared how easy it is to generate test cases automatically from them.

Based on the results of this thesis, we make the following concluding remarks:

1. The integrator and customer roles must be considered in the software testing process. They both need different kinds of testing methods than developers. The existing testing methods are often so theoretical that they cannot be utilized in practice.
2. We found that most software companies do not devote resources to conformity assessment voluntarily. The quality of the software and interoperability improves only by increasing the number of requirements issued by the authorities. Common rules are needed and responsibilities must be clearly defined. In addition, customers must demand standard-based applications more often.
3. Interoperability conformance testing needs good quality specifications. Without proper specifications, conformity cannot be tested so that interoperability can be assured. At the moment, specifications in the healthcare domain are not suitable for interoperability testing purpose.
4. Continuous improvement of the test process is important. When test processes are improved, organizations have to pay attention to not only the actual test processes, but also to test case documentation, regression testing, and the level of training in testing.

There are some limitations to this study. Firstly, the number of software companies in our survey was quite small. This affects the generalizability of the results. It appears that test cases are documented in small software companies only half of that in large companies. However, no statistical conclusions can be made. Secondly, the results of the study were obtained in the healthcare domain, which has many critical

systems. Therefore, the results can be generalized to domains where software quality and reliability are prioritized.

## 6.2 Future work

This thesis investigates software testing using a component-based approach. In future, it is important to expand the method to cover service-oriented approach and its new requirements. We emphasized the integrator and customer viewpoints in testing in the software industry and in customer organizations, but the service provider viewpoint with problem management must also be studied. Additionally, more research is needed to discover how business knowledge influences testing. In future, how test process improvement models can be applied in small and large software companies, must be studied. In addition, it would be valuable to survey how the proposed recommendations have been utilized in practice and how well they are suited to different organizations in different domains. A lot of research relating to software and conformance testing has been done. However, practical viewpoints in theory are needed so that theory can be efficiently applied in practice.

# Bibliography


ACC/HIMSS/RSNA. (2005). IHE IT Infrastructure Technical Framework, Volume 1, Integration Profiles Revision 2.0, Retrieved June 9, 2008, http://www.ihe-j.org/file2/comments/card/ihe_iti_tf_2_0_vol1_FT_approved.pdf.

Baskerville, R. (1999). Investigating information systems with action research. *Communications of the Association for Information Systems*, 2(19).

Bass, L., Clements, P. and Kazman, R. (2003). *Software Architecture in Practice*. Addison-Wesley.

Beizer, B. (1990). *Software Testing Techniques*. Second Edition. Van Nostrand Reinhold.

Bertolino, A. The (Im)maturity Level of Software Testing. (2004). *ACM SIGSOFT Software Engineering Notes*, 29(5), 1-4.

Beugnard, A., Jèzèquel, J-M., Plouzeau, N. and Watkins, D. (1999). Making Components Contract Aware. *Computer*, 32(7), 38-45.

Biggerstaff, T. and Perlis, A. eds. (1989). *Software Reusability, Volume I, Concepts and Models*. ACM Press.

Binder, R. (1999). *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison Wesley.

Boehm, B. (1988). A Spiral Model of Software Development and Enhancement. *Computer*, 21(5), 61-72.

Boehm, B., Brown, J.R., Kaspar, H., Lipow, M., McLeod, G. and Merrit. M. (1978). *Characteristics of Software Quality*. North Holland.

Booch, G. (1987). *Software Components with Ada: Structures, Tools, and Subsystems*. Addison-Wesley.

Booch, G. (1991). *Object oriented design with applications.* The Benjamin/Cummings Publishing Company.

Booch, G., Rumbaugh, J. and Jacobson, I. (1998). *The Unified Modeling Language User Guide*. The Benjamin/Cummings Publishing Company, Inc.

Brown, A. and Wallnau, K. (1996). Engineering of Component-Based Systems. In *Proceedings of Second IEEE International Conference on Engineering of Complex Computer Systems*, 414-422.

Burnstein, I., Suwannasart, T. and Carlson, C.R. (1996). Developing a Testing Maturity Model: Parts I and II, Illinois Institute of Technology.

CCHIT. (2007). Physician's Guide to Certification for Ambulatory Electronic Health Records, Retrieved May 22, 2008, from http://www.cchit.org/files/CCHITPhysiciansGuide2007.pdf.

Chen, H.Y., Tse, T.H., Chan, F.T. and Chen, T.Y. (1998). In Black and White: An Integrated Approach to Class-Level Testing of Object-Oriented Programs. *ACM*

*Transactions on Software Engineering and Methodology*, 7(3), 250-295.

Chen, Y., Probert, R. and Sims, D.P. (2002). Specification-based regression test selection with risk analysis. In *Proceedings of the 2002 conference of the Centre for Advanced Studies on Collaborative research*.

Chronaki, C.E. and Chiarugi, F. (2006). OpenECG: Testing conformance to CEN/EN 1064 standard. *5th European Symposium on BioMedical Engineering*. Patras, Ellas.

Chronaki, C.E., Chiarugi, F., Sfakianakis, S. and Zywietz, C. (2005). A web service for conformance testing of ECG records to the SCP-ECG standard. *Computers in Cardiology*, 32, 961-964.

Clements, P.C. (1995). From Subroutines to Subsystems: Component-Based Software Development. *The American Programmer*, 8(11).

Collefello, J. S. and Buck, J. J. (1987). Software Quality Assurance for Maintenance. *IEEE Software*, 4(5), 46-51.

Dybå, T. (2003). Factors of Software Process Improvement Success in Small and Large Organizations: An Empirical Study in the Scandinavian Context. In *European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 148-157.

Ericson, T., Subotic, A. and Ursing, S. (1997). TIM - Test Improvement Model. *Software Testing, Verification and Reliability*, vol. 7, 229-246.

Forsberg, K., Mooz, H. and Cotterman, H. (2005). *Visualizing Project Management: Models and Frameworks for Mastering Complex Systems*. John Wiley & Sons.

Gao, J., Tsao, H-S. and Wu, Y. (2003). *Testing and Quality Assurance for Component-Based Software*. Artech House.

Gelperin, D. and Hetzel, B. (1988). The growth of software testing. *Communications of the ACM*, 31(6), 687-695.

Graves, T., Harrold, M.J., Kim J-M., Porter, A. and Rothermel, G. (2001). An Empirical study of Regression Test Selection Techniques. *ACM Transactions on Software Engineering and Methodology*, 10(2), 184-208.

Gray, M., Goldfine, A., Rosenthal, L. and Carnahan, L. (2000). Conformance Testing. In *XML General Articles and Papers: Surveys, Overviews, Presentations, Introductions, Announcements*. Retrieved April 25, 2008, from http://xml.coverpages.org/conform20000112.html.

Groves, L. and Nickson, R. (2000). A Survey of Software Development Practices in the New Zealand Software Industry. In *Proceedings of IEEE Australian Software Engineering Conference*, 189-

Haux, R. (2006). Health information systems - past, present, future. *International Journal of Medical Informatics*, 75(3-4), 268-281.

Herzum, P. and Sims, O. (2000). *Business Component Factory*. Wiley Computer Publishing.

Hutchins, M. Foster, H., Goradia, T. & Ostrand, T. (1994). Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th international conference on Software engineering*, IEEE, 191-200.

ISO/IEC. (1996). Guide 2: Standardization and Related Activities: General Vocabulary.

ISO/IEC JTC1 SC36. (2003). Information Technology for Learning, Education, and Training. Working draft.

ISO/IEC 9126-1. (2001). Software engineering - Product quality - Part 1: Quality model.

ITU-T Recommendation X.290. (1996). OSI Conformance Testing Methodology and

Framework for Protocol Recommendations for ITU-T Applications - General Concepts.

Jacobson, I., Booch, G. and Rumbaugh, J. (1999). *The Unified Software Development Process*. Addison-Wesley.

Jacobson, I., Christerson, M., Jonsson, P. and Övergaard, G. (1992). *Object-Oriented Software Engineering. A Use Case Driven Approach*. Addison-Wesley.

Jäntti, M. and Toroi, T. (2004). UML-based testing: a case study. In: Koskimies K, Kuzniarz L, Lilius J, Porres I, eds. *Proceedings of NWUML'2004. 2nd Nordic Workshop on the Unified Modeling Language*, 33-44. TUCS General Publication 35.

Jäntti, M., Toroi, T. and Eerola, A. (2006). Difficulties in Establishing a Defect Management Process: A Case Study. In Münch J. and Vierimaa M. eds. *Proceedings of PROFES 2006*, Springer-Verlag, Lecture Notes in Computer Science, 142–150.

Järvinen, P. (2004). *On research methods*. Opinpajan Kirja.

Järvinen, P. and Järvinen, A. (1995). *Tutkimustyön metodeista*. Opinpaja Oy.

Jézéquel, J-M.and Meyer, B. (1997). Design by Contract: The Lessons of Ariane. *IEEE Computer*, 30(1), 129-130.

Juristo, N., Moreno, A., Vegas, S. and Solari, M. (2006). In Search of What We Experimentally Know about Unit Testing. *IEEE Software*, Nov/Dec, 72-80.

Kaner, C. (2004). The Ongoing Revolution in Software Testing. In *Proceedings of Software Test & Performance Conference*.

Kindrick, J., Sauter, J. and Matthews, R. (1996). Improving Conformance and Interoperability Testing. *StandardView*, 4(1).

Kitchenham, B., Pfleeger, S.L., Hoaglin, D., Emam, K. and Rosenberg, J. (2002). Preliminary Guidelines for Empirical Research in Software Engineering. *IEEE Transactions on Software Engineering*, 28(8), 721-734.

Klingler, A. (2000). An open, Component-based Architecture for Healthcare Information Systems. In Hasman A. et al. eds. *Medical Infobahn for Europe*. IOS Press.

Koomen, T. and Pol, M. (1999). *Test Process Improvement, a practical step-by-step guide to structured testing*. Addison-Wesley.

Kouchakdjian, A. and Fietkiewicz. R. (2000). Improving a product with usage-based testing. *Information and Software Technology*, 42(12), 809-814.

Kruchten, P. (2003). *The Rational Unified Process: An Introduction*. Addison-Wesley.

Kung, D., Gao, J., Hsia, P., Wen, F., Toyoshima, Y. and Chen, C. (1994). Change Impact Identification in Object Oriented Software Maintenance. In *Proceedings of IEEE International Conference on Software Maintenance*, 202-211.

Lange, C.F.J., Chaudron, M.R.V. and Muskens, J. (2006). In Practice: UML Software Architecture and Design Description, *IEEE Software*, March/April, 40-46.

Maxville, V., Lam, C. and Armarego, J. (2003). Selecting Components: a Process for Context-Driven Evaluation. In *Proceedings of the Tenth Asia-Pacific Software Engineering Conference*, IEEE.

McCall, J., Richards, P. and Walters, G. (1977). *Factors in Software Quality. Volume I. Concepts and Definitions of Software Quality*. Technical report, AD-A049 014/4.

MedCom. (2005). MedCom IV - how it turned out. The Danish Healthcare Data Network / December 2005. Retrieved June 13, 2007, from http://www.medcom.dk/dwn404.

Meyer, B. (1988). *Object-oriented Software Construction*. Prentice Hall.

Morell, L.J. (1990). A Theory of Fault-Based Testing. *IEEE Transactions on Software Engineering*, 16(8), 844 - 857.

Moseley, S., Randall S. and Wiles, A. (2004). In Pursuit of Interoperability. *International Journal of IT Standards and Standardization Research*, 2(2), 34-48.

Myers, G. (1979). *The art of software testing*. John Wiley & Sons.

Mykkänen, J. et al. (2004). Integration Models in Health Information Systems: Experience from the PlugIT project. In Fieschi M, Coiera E, Li Y-C J, eds. *MEDINFO 2004. Proceedings of the 11th World Congress on Medical Informatics*, 1219-1222. Amsterdam: IOS Press.

Ng, S.P., Murname, T., Reed, K., Grant, D. and Chen, T.Y. (2004). A Preliminary Survey on Software Testing Practices in Australia. In *Proceedings of the 2004 Australian Software Engineering Conference*, 116-

NHS. (2002). Delivering 21st Century IT Support for the NHS, National Specification for Integrated Care Records Service, Consultation Draft. Retrieved May 7, 2008, from http://www.prorec.it/documenti/EPR_EHR/ICRS-specs_12d.pdf.

OASIS. (2006). Reference Model for Service Oriented Architecture 1.0, OASIS Standard, 12 October 2006.

Offutt, J. and Abdurazik, A. (1999). Generating Tests from UML Specifications. In *Second International Conference on Unified Modeling Language*.

Offutt, J., Lee, A., Rothermel, G., Untch, R.H. and Zapf, C. (1996). An Experimental Determination of Sufficient Mutant Operators. *ACM Transactions on Software Engineering Methodology*, 5(2), 99-118.

Onoma, A., Tsai, W-T., Poonawala, M. and Suganuma, H. (1998). Regression Testing in an Industrial Environment. *Communication of the ACM*, 41(5), 81-86.

Parnas, D.L. (1972). On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12), 1053-1058.

Pfleeger, S.L. (2001). *Software engineering theory and practice*. Prentice Hall.

Pfleeger, S.L. and Kitchenham, B. (2001). Principles of Survey Research. Part 1: Turning Lemons into Lemonade. *Software Engineering Notes*, 26(6), 16-18.

Pressman, R. (2005). *Software Engineering, A Practitioner's Approach*. McGraw-Hill.

QAI, Quality Assurance Institute. (1995). Establishing a software defect management process. Research Report number 8.

Rada, R. (1996). Who will test conformance? *Communications of the ACM*, 39(1), 19-22.

Rosenthal, L. and Skall, M. eds. (2002). Conformance Requirements for Specifications v1.0. Retrieved February 16, 2007, from http://www.oasis-open.org/committees/download.php/305/conformance_requirements-v1.pdf.

Rothermel, G. and Harrold, J.M. (1997). A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2), 173-210.

Royce, W. (1970). Managing the development of large software systems. In Proceedings of *IEEE WESCON*, 1-9.

Runeson, P. (2006). A Survey of Unit Testing Practices. *IEEE Software*, July/August, 22-29.

Sametinger, J. (1997). *Software Engineering with Reusable Components*. Springer-Verlag.

Schneider, G. and Winters, J.P. (2006). *Applying Use Cases, Second Edition. A*

*Practical Guide*. Addison-Wesley.

SFS, The Finnish Standards Association. (2008). Retrieved December 16, 2008, from http://www.sfs.fi/it/aihealueet/terveydenhuolto/kansainvalinen/.

Stafford, J. and Wallnau, K. (2001). Is Third Party Certification Necessary? In *Proceedings of 4th ICSE Workshop on Component-Based Software Engineering, Component Certification and System Prediction*, On-line proceedings: http://www.sei.cmu.edu/pacc/CBSE4-Proceedings.html.

Swinkels, R. (2000). A Comparison of TMM and Other Test Process Improvement Models. Frits Philips Institute. Technical Report. Retrieved February 22, 2007, from http://wwwbruegge.informatik.tu-muenchen.de/static/contribute/Lehrstuhl/documents/12-4-1-FPdef.pdf.

Szyperski, C. (2002). *Component Software - Beyond Object-Oriented Programming*. Second Edition, Addison-Wesley.

Taipale, O. (2007). *Observations on Software Testing Practice*. Dissertation. Lappeenranta University of Technology.

Torkar, R. and Mankefors, S. (2003). A Survey on Testing and Reuse. In *Proceedings of IEEE International Conference on Software - Science, Technology & Engineering*, 164-

Toroi, T., Eerola, A. and Mykkänen, J. (2002). Testing Business Component Systems. University of Kuopio, Department of Computer Science and Applied Mathematics, Report A-2002-1. 12 p.

Tsai, B-Y, Stobart, S., Parrington, N. and Thompson, B. (1997) Iterative design and testing within the software development life cycle. *Software Quality Journal*, 6, 295-309.

Vitharana, P., Zahedi, F. and Jain, H. (2003). Design, Retrieval, and Assembly in Component-based Software Development. *Communications of the ACM*, 46(11), 97-102.

W3C. (2005). QA Framework: Specification Guidelines. W3C Recommendation 17 August 2005. Retrieved March 22, 2007 from http://www.w3.org/TR/qaframe-spec/.

Wagner, F., Schmuki, R., Wagner T. and Wolstenholme, P. (2006). *Modeling Software with Finite State Machines: A Practical Approach*. CRC Press.

Weyuker, E. (1998) Testing Component-Based Software: A Cautionary Tale. *IEEE Software*, 15(5), 54-59.

Wilde, N. and Huitt, R. (1992). Maintenance Support for Object-Oriented Programs. *IEEE Transactions on Software Engineering*, 18(12), 1038-1044.

Yin, R. (2003). *Case Study Research: Design and Methods*. Third Edition, Applied Social Research Methods Series, vol. 5, Sage Publishing.

Zheng, J., Robinson, B., Williams, L. and Smiley, K. (2005). A Process for Identifying Changes When Source Code is Not Available. In *Proceedings of the second international workshop on models and processes for the evaluation of off-the-shelf components MPEC '05*.

Zhu, H., Hall, P. and May, J. (1997). Software Unit Test Coverage and Adequacy. *ACM Computing Surveys*, 29(4), 366-427.

## Kuopio University Publications H. Business and Information technology

**H 1. Pasanen, Mika.** In Search of Factors Affecting SME Performance: The Case of Eastern Finland.
2003. 338 p. Acad. Diss.

**H 2. Leinonen, Paula.** Automation of document structure transformations.
2004. 68 p. Acad. Diss.

**H 3. Kaikkonen, Virpi.** Essays on the entrepreneurial process in rural micro firms.
2005. 130 p. Acad. Diss.

**H 4. Honkanen, Risto.** Towards Optical Communication in Parallel Computing.
2006. 80 p. Acad. Diss.

**H 5. Laukkanen, Tommi.** Consumer Value Drivers in Electronic Banking.
2006. 115 p. Acad. Diss.

**H 6. Mykkänen, Juha.** Specification of reusable integration solutions in health information systems.
2006. 88 p. Acad. Diss.

**H 7. Huovinen, Jari.** Tapayrittäjyys – tilannetekijät toiminnan taustalla ja yrittäjäkokemuksen merkitys yritystoiminnassa.
2007. 277 p. Acad. Diss.

**H 8. Päivinen, Niina.** Scale-free Clustering: A Quest for the Hidden Knowledge.
2007. 57 p. Acad. Diss.

**H 9. Koponen, Timo.** Evaluation of maintenance processes in open source software projects through defect and version management systems.
2007. 92 p. Acad. Diss.

**H 10. Hassinen, Marko.** Studies in mobile security.
2007. 58 p. Acad. Diss.

**H 11. Jäntti, Marko.** Difficulties in managing software problems and defects.
2008. 61 p. Acad. Diss.

**H 12. Tihula, Sanna.** Management teams in managing succession: learning in the context of family-owned SMEs.
2008. 237 p. Acad. Diss.