TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY
Julkaisu 847 • Publication 847

Jani Metsä

Aspect-Oriented Approach to Testing – Experiences from a Smartphone Product Family

Tampere 2009

Tampereen teknillinen yliopisto. Julkaisu 847
Tampere University of Technology. Publication 847

Jani Metsä

Aspect-Oriented Approach to Testing – Experiences from a Smartphone Product Family

Thesis for the degree of Doctor of Technology to be presented with due permission for public examination and criticism in Tietotalo Building, Auditorium TB109, at Tampere University of Technology, on the 4th of December 2009, at 12 noon.

Tampereen teknillinen yliopisto - Tampere University of Technology
Tampere 2009

ISBN 978-952-15-2255-0 (printed)
ISBN 978-952-15-2309-0 (PDF)
ISSN 1459-2045

Abstract

The size of software running in smartphones has increased significantly during recent years. As a result the testing of such systems is inherently more difficult and expensive. Meanwhile, the quality expectations are more demanding and development life-cycles shorter. Since testing is commonly considered one of the most resource consuming activities of a modern software project, device manufacturers need to find solutions to more effectively manage the testing effort and related investments in order to succeed in a fiercely competitive environment.

This thesis presents an approach to capturing system-wide, cross-cutting concerns to be modularized as manageable units for testing. Considering testing as a system-wide concern allows separating the testing concern from other concerns, thus modularizing testability issues into manageable units. In this thesis aspect-orientation is used in formulating such testability issues and in implementing testware.

In the approach presented here the origin of test case definitions is revised from code and specifications to requirements and expectations. Expected behaviors and system characteristics are formulated as testing objectives and further developed as test aspects. These aspects provide representations of system-wide testing concerns, such as performance, and implement testing in a non-invasive manner, allowing system-wide testing issues to be addressed already at the unit testing level. Furthermore, visualizing the testing scenarios as sequence diagrams can be used to automatically generate the related testware in system testing, as well as in unit testing, without the need for understanding the original code.

Based on the experiments the aspect-oriented approach proposes a technique to implement testware without actually touching the code. The system under test and the related testware are easier to separate, as the original implementation remains oblivious to testing in most cases. An aspect-oriented approach allows system-wide evaluation of testing concerns, but formulating the correct aspects can be difficult. Furthermore, adopting the technique on an industrial scale requires changes to processes and additional tools.

Preface

This thesis was made possible by several people. First of all, I want to thank Professor Tommi Mikkonen for his guidance, support, and valuable observations during and throughout the process. I would also like to thank Adjunct Professor Mika Katara for his invaluable help in composing the research and related publications. I am also thankful to Shahar Maoz for his help in completing the final parts of the research and Professor Kai Koskimies for review comments. I am grateful for Professor Markku Sakkinen and Professor João Araújo for their valuable comments as preliminary examiners. Furthermore, I would like to thank all the reviewers of the publications for their useful comments.

I would like to express my gratitude to the Nokia Corporation for providing the opportunity to carry out the research in an actual work environment and for offering the environment and tools necessary for the research. Also my colleagues deserve acknowledgment for the pleasant working atmosphere and their valuable contribution in the pragmatic research work. Especially Jouni for his words of wisdom and Sami for his assistance in conducting the measurements.

Further, I wish to thank the Nokia Foundation for granting sponsorship to carry out the writing of this thesis and Tampere University of Technology for providing the opportunity and funding to complete it. I am also grateful for funding from the Academy of Finland (grant number 121012).

I would like to express my deepest gratitude to my family for their everlasting support and encouragement. This thesis would not have been possible without the foundation created by my parents, Erkki and Liisa, and my siblings, Anne and Harri, spurring me on further. I am extremely grateful to my wife, Hanna, for standing up for me during the hard times and for her never-ending understanding. Finally, the existence and sincere happiness of my son Atte gave me the courage and strength to complete this thesis. Thank you!

Contents

List of Included Publications

[I] Pesonen, J., Assessing Production Testing Software Adaptability to a Product-line. In *Proceedings of the 11th Nordic Workshop on Programming and Software Development Tools and Techniques* (NWPER 2004), Turku Centre for Computer Science, Turku, Finland, August 2004. TUCS General Publication Number 34. Turku Centre for Computer Science, 2004.

[II] Pesonen, J., Katara, M. and Mikkonen, T. Production-Testing of Embedded Systems with Aspects. In *Proceedings of the Haifa Verification Conference*, IBM Haifa Labs, Haifa, Israel, November 2005. Number 3875 in Lecture Notes in Computer Science. Springer, 2005.

[III] Pesonen, J., Extending Software Integration Testing Using Aspects in Symbian OS. In *Proceedings of Testing: Academic & Industrial Conference - Practice And Research Techniques* (TAIC PART 2006), Windsor, United Kingdom, August 2006. IEEE Computer Society, 2006.

[IV] Metsä, J., Katara, M. and Mikkonen, T., Testing Non-Functional Requirements with Aspects: An Industrial Case Study. In *Proceedings of the Seventh International Conference on Quality Software* (QSIC 2007), Portland, Oregon, USA, October 2007. IEEE Computer Society, 2007.

[V] Metsä, J., Katara, M., and Mikkonen, T., Comparing Aspects with Conventional Techniques for Increasing Testability. In *Proceedings of the First International Conference on Software Testing Verification and Validation* (ICST 2008), Lillehammer, Norway, April 2008. IEEE Computer Society, 2008.

[VI] Maoz, S., Metsä, J., Katara, M., Model-Based Test Specification and Execution Using Live Sequence Charts and the S2A Compiler: an Industrial Experience. Technical Report 4, Tampere University of Technology, Department of Software Systems, 2009. Appearing also as a short paper: Model-Based Testing using LSCs and S2A. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MODELS 2009)*, Denver, Colorado, USA, October, 2009. Number 5795 in Lecture Notes in Computer Science, Springer, 2009.

The permission of the copyright holders of the original publications to reprint them in this thesis is hereby acknowledged.

In the course of publishing the papers the candidate's last name has changed from Pesonen to Metsä.

1 Introduction

During recent years the sizes of software systems running in mobile devices have increased tremendously – particularly so when considering smartphones that have evolved from plain old telephones to complex multimedia computers. While the techniques used to develop such software systems evolve, the produced systems often become more complex and testing them becomes inherently more difficult. Traditional approaches to verify correctness have become insufficient and today's research aims at high-level verification against expectations. Raising the level of testing towards the level of system complexity requires research targeted at high-level testing objectives and the techniques to achieve them. Aspect-oriented programming [1] is a programming paradigm that allows modularizing system-wide issues into manageable units and proposes a technique for such a testing approach.

1.1 Motivation

Testing complex software systems is difficult and expensive, to the extent that testing is considered as one of the most resource-consuming activities in the development of any modern software systems [2]. A typical software system today is composed of a number of subsystems, exercising a significant amount of resources, and interacting with a number of other systems. Testing such systems requires both investment and effort, making it an important factor in the overall development. Since the quality expectations regarding software in consumer appliances, such as smartphones and embedded software systems, in general are increasingly demanding, proper testing methodologies are important for quality assurance.

As such systems are implemented as a composition of a number of building blocks, the overall impact on the system behavior is increasingly difficult to anticipate. Although individual elements realize concerns – abstract concepts about program behavior – as system building blocks, composing the system results in a collection of the thoughts and decisions of a number of developers. Hence, while developed independently of each other, merging components

into complete systems tends to produce surprising behaviors. While a system is more than the sum of its components, it is also more than the sum of the developers' thoughts. Therefore new behaviors, which nobody may have thought of, commonly emerge during integration, and as a result, the system behavior cannot be anticipated in full. This introduces new kinds of issues to be covered in system testing, and due to their system-wide nature, calls for methods to modularize them into manageable units.

Capturing such cross-cutting concerns under evaluation at lower testing levels is complicated when using established testing techniques due to the lack of single components to bind the test implementation to. Although commercially available tools provide efficient support for the traditional testing approach, such tools are mostly application specific and aim at supporting a certain testing level only. Acceptance and system testing are typically the only testing levels addressing system-wide testing. Furthermore, not all existing tools allow the implementation of system-wide test cases due to limitations in the granularity of the expressible tests. Moreover, although functional testing has sound tool support, special tools are often needed for non-functional testing.

Performance measurements, profiling, and reliability testing are all examples of non-functional testing activities typically involving not only tailored tools but also a varying amount of instrumentation and refactoring of the original system. The lack of tools that allow cross-cutting, system-wide issues to be addressed independently of the testing level limits the testing possibilities and leads to situations where non-functional testing is performed in a more ad-hoc fashion. Testing cross-cutting issues is thus not possible in a non-invasive manner, that is, without touching the original implementation. Hence, regardless of the testing approach, specific software is required to accomplish these objectives.

Testware – specific software developed to conduct testing related tasks – requires a development effort of its own. For obvious reasons, the maintenance of testware is laborious for complex systems. Enabling proper testing and explicitly addressing system testability requires a refactoring of the system and introducing a code specific for testing. Due to the lack of techniques for separation of concerns, this tends to lead to code tangling [3] – implementations of concerns intermixed together – of the implementations of the System Under Test (SUT) and the testware. Although commercial testing tools offer methods to cover basic testing needs, enabling testability in the system design allows for more efficient testing.

1.2 Thesis questions

In this thesis the statement is that raising the level of abstraction of test case descriptions from the code-level to the requirements level and enabling a descriptive enough, high-level language or syntax to describe the test cases allows defining system-wide testing concerns for all testing levels. These concerns can already be defined based on the requirements and provide a simple but effective method of capturing the testability concerns under consideration already in the architecture design phase. Furthermore, a high level language allows formulating test cases that exercise such system-wide issues and simplifies the implementation and execution of such test cases already at the stage of unit testing. Using aspect-oriented programming, the related testware can be implemented in a non-invasive and application-independent manner, thus enabling high reusability.

To undertake the aforementioned issues, this thesis presents an approach for addressing system-wide issues and provides a technique for capturing them for testing. Furthermore, common guidelines on selecting the system characteristics potential for such treatment are introduced. In the proposed approach, formulating testing concerns as modules should allow separating the testing concerns from the underlying system, thus making them easier to manage. A set of thesis questions is set for elaboration. These are divided into overall testability of the system, maintenance, test coverage, and quality of testing, and are presented in the following:

1. *How could aspect-oriented programming be utilized in production verification of a product line of smartphones?*

   The quality verification of smartphone manufacturing, the production verification[1], requires software that can be varied to a number of different products with a number of different characteristics. Nevertheless, this software needs to offer common functionalities for all the product variants, thus adapting them to the manufacturing environment. Does aspect-orientation provide techniques that are useful in developing such software?

2. *What are the benefits of using an aspect-oriented approach for testing over conventional techniques? Furthermore, is there a systematic approach for identifying the testing concerns and formulating them as aspects?*

[1] In this context term "production verification" refers to the quality assurance of mobile device manufacturing, the process of verifying that the devices are assembled correctly. In some of the included publications, also the term "production testing" is used in this denotation.

Software testing has been traditionally divided into different phases and conducted in a number of different ways. Also the testing objectives vary from application to another, and it is not self-evident when aspect-orientation is applicable. Should conventional techniques, although proven efficient, be replaced with aspects or are there any specific types of testing that benefit most from aspect-oriented treatment?

3. *What kind of methods and techniques are required by an aspect-oriented approach to testing?*

   Moreover, since the approach is somewhat different from conventional approaches, does it propose new kinds of issues to be tested? In case such issues emerge, any systematic approach to practice the methodology should be outlined to form a basic set of guidelines and practices.

4. *How could aspects be used to increase software testability?*

   Initially, testing aims at verifying that any software system satisfies all the expectations on the behavior set for it. Furthermore, testing typically involves defining the quality-related characteristics that the system should have. However, in order to achieve proper satisfaction of these concerns, the system design should support testing, at least in the case of complex systems. Does aspect-orientation provide a method for enhancing the testability of such systems and thus better support the overall testing objective?

Ultimately, the thesis question is therefore "does modularizing concerns as testable elements both increase the overall testability of the system and also make the testing more efficient?" Furthermore, we consider "are there any benefits from using aspects in testing object-oriented systems?".

1.3 Contributions of the thesis

This thesis presents an approach for capturing cross-cutting testing concerns in software systems and modularizing them as testing objectives. This approach was evaluated in a number of consecutive case studies conducted while working for Nokia during the years 2004 – 2009 and using a small-scale industrial software system of commercial value. Hence, the approach of this research is constructive augmented with pragmatic experiments. All experiments, including the necessary measurements and tool instrumentation, were

performed on real target prototypes of Nokia handheld terminals, i.e., real smartphones.

In this thesis, I show how to use aspect-oriented programming to formulate testing objectives based on non-functional requirements. Furthermore, the required changes to processes and tool chains are explained in addition to examples of implementing traditional testware using aspects. It is shown how aspects are used to implement testware in a non-invasive manner. The presented technique is evaluated in an industrial application, which shows the technique is usable in testing smartphones.

Specifically, the key contributions of the thesis are the following:

- An approach is presented to using aspects for non-functional testing above the unit testing level, starting already with requirements analysis.
- The approach refines common development processes to address test development using aspects. Required changes to the process are described when defining test aspects based on the requirements.
- A set of basic guidelines is formulated on defining reusable testing aspects and identifying cross-cutting concerns in the early development phases to be formulated as testing aspects.
- The applicability of Aspect-Oriented Programming (AOP) for different levels of testing is evaluated, providing guidelines on whether to utilize aspects or traditional techniques.

The candidate's contributions in the included publications are presented in the following.

1.4 Introduction to the included publications

The contributions of the research in the included publications is divided as follows. First, publication [I] introduces the context and analyzes implementation related issues. The publication presents an assessment of the initial system's applicability as a product family and discusses design and implementation issues related to such systems. It acts as the starting point for the thesis, and the thesis question for approach to adapting AOP in this context is defined based on the findings of this assessment.

Issues related to implementing production verification software systems are further discussed in publication [II]. A rough partitioning between implementations, which can be either object- or aspect-oriented, is presented as result of the analysis. This publication provides evaluation of using an

aspect-oriented approach for developing production verification software of smartphones. In the publication, AOP has been applied to the initial real-life system in order to evaluate the possibilities of utilizing it in the testing of such systems. This sets a research goal for the subsequent publications.

The initial study on applying the approach in the production verification of smartphones is followed by a test harness implementation for the underlying system, discussed in publication [III]. The publication introduces a feasibility study of utilizing the approach as a test harness for sophistically capturing software testing concerns. This evaluation continues the pragmatic evaluation of the approach in implementing integration testing for the system. The obtained results are compared to the results gained without the proposed approach. This publication sets a baseline for the approach in the context of testing smartphones on an industrial scale and refines the research problem. As a result a list of issues related to the setting are pointed out and practical research on the applicability of the approach is conducted.

The approach is further evaluated in publication [IV] by expanding the scope and widening the approach from the implementation level towards studies on higher-levels of abstraction and working on earlier stages of the software development life cycle. The publication studies the identification of non-functional requirements to be modularized as the testing concerns using aspect-oriented techniques and provides an initial comparison to conventional techniques. Furthermore, to study the impacts on the higher-level method, a requirements management study on mapping the requirements to testing objectives and further test cases is conducted. This includes a comparison between existing test cases and the ones derived using the approach. This analysis is followed by a qualitative evaluation using subjective analysis on the resulting data as to whether this increases the overall testability of the system.

A pragmatic comparison to traditional techniques is performed in the form of a case study in publication [V]. In this study, the proposed approach is compared to conventional techniques, macros and interfaces, in the scope of increasing the testability in the context of the original system. This involves an implementation of the testing technique using conventional techniques and the approach presented in this thesis. The comparison is based on the results of running the implemented tests on the target system and comparing the results to the ones gained with conventional techniques. This is based on the resulting data in terms of number of test cases, number of found errors, and subjective analysis on the easiness of implementing the test cases. These results were partially gathered by interviewing the test case developers and personnel executing the tests.

Finally, publication [VI] presents an approach to creating a tool for vi-

sualizing the test scenarios and automatically generating testware based on the diagrams. In this study Live Sequence Charts are used for modeling the behavior and defining the objectives for testing. This model is used to generate aspect code that is to be woven into the system for testing purposes, thus allowing testware to be implemented without seeing the original code.

The candidate is the sole author in publications [I] and [III]. In these publications Tommi Mikkonen had an advisory role and provided comments that lead to improvements. Publications [II], [IV], and [V] are a joint effort with co-authors Mika Katara and Tommi Mikkonen. In these publications the candidate was the main author. All the case studies were conducted by the candidate in addition to the context descriptions. The main contribution of the candidate in publication [IV] is the practical research work in the case study based on the hypothesis. Furthermore, the problem statement of deriving test objectives from requirements is the candidate's contribution in this publication. In publication [V] the writing was shared with the co-authors and the candidate evaluated the techniques and presented the aspect-oriented approach to the context. Publication [VI] is a joint effort with co-authors Shahar Maoz and Mika Katara. In this publication the candidate shared the writing with co-authors and contributed in altering the existing compiler to be able to generate proper aspect code and conducted the pragmatic experiments including modeling.

1.5 Organization of the thesis

The content of the introductory part of this thesis is organized as follows. Chapter 2 discusses software testing in general and introduces the problem areas related to the approach of this thesis. The basic theory of aspect-orientation is presented in Chapter 3. Chapter 4 describes the approach of this thesis in more detail defining the approach of using aspect-oriented programming in achieving a non-invasive technique to capture cross-cutting issues for testing. A definition of the approach is followed by an introduction to the case studies in Chapter 5, discussing the contributions of the publications in detail. Chapter 6 discusses related work. Finally, Chapter 7 concludes the thesis with the evaluation of the thesis questions.

2 Software Testing

This chapter discusses software testing in general and the testing of complex smartphone systems in particular. Towards the end of the chapter the problems related to conventional testing techniques are described. The chapter is mainly based on Craig and Jaskiel [4] unless otherwise denoted.

2.1 Conventional approach to testing

Software testing is an empirical method of checking that the produced system fulfills the quality and functionality expectations of the stakeholders. This involves typically executing the program and performing a number of predefined experiments in order to find deviating behavior between the expected and experienced ones. In essence, testing is about producing inputs, assessing the outputs, and observing the related behavior in respect to expected system behavior. Comparing the behavior to submitted inputs is used to validate whether the system behaves correctly, and if no extra behaviors are experienced, the system is considered to pass the related test cases. Basic elements for testing are the test cases, control and data, System Under Test (SUT), expected output, observed output, comparing expected and observed output, test results, and test report, as illustrated in Figure 1.

Prior to testing, a certain setup is required in order to set the SUT state corresponding to testing objectives [5]. *Control and data* is input that is fed

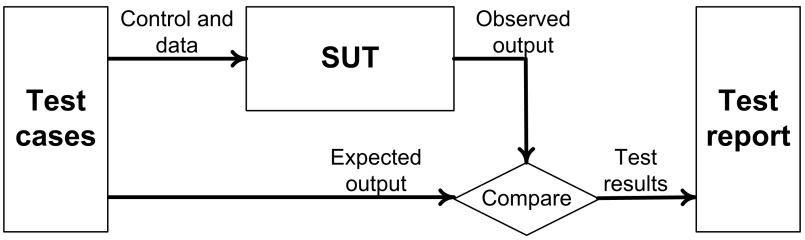

Figure 1: Basic testing setup.

to the SUT to produce *expected output*. According to the original definition of what the SUT is designed to achieve, the expected output reflects the stakeholders' expectations of the SUT against the given input. In contrast to expected output *observed output* is the actual output of the SUT while executing according to the input. Based on the implementation and the type of the result in question, comparing them could be easy or extremely difficult. Observed and expected output are finally compared based on variables or conditions defined in *test cases*, which are also used to define the control and data required to achieve the necessary outputs. Hence, a test case is a collection of control and data required to achieve the conditions where a comparison between expected and observed outputs can be conducted. Finally, a *test report* collects and documents the test results.

Testing is a target-oriented activity. The goal is both to verify that the system behaves as it is intended to, and also to try to achieve situations where it does not. Any testing activity has a test objective, that the testing aims at accomplishing. These objectives can be either functional or non-functional. Moreover, setting the goal of breaking the system makes testing destructive in nature.

Traditionally, software testing has been divided into white-box and black-box testing, depending on the required level of understanding of the program structure, as illustrated in Figure 2. While white-box testing relies on understanding of the structure and implementation details, the code, black-box testing operates at the level of interfaces – User Interface (UI) or Application Programming Interface (API), for instance – thus overlooking the code behind the interfaces. In the latter case, test cases are created based on specifications instead of the system structure. If the system structure is used in test case definitions together with the specifications, the testing approach is called grey-box testing, as an intermediate form of white-box and black-box techniques.

When concentrating on the system behavior instead of the code structure, testing is considered functional testing [6]. In other words, in functional testing the focus is on verifying that the system functions as specified. With

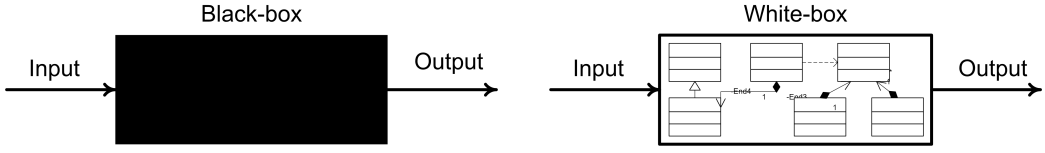

Figure 2: Black-box and white-box testing principle.

software systems the functional testing is the most experienced area of testing starting from module and integration testing and ending in acceptance testing [4, pages 98–144]. There are a number of established testing techniques to choose from, and sound support for functional testing exists.

According to the level of knowledge of the program structure when defining the functional test cases, functional testing can be performed in either a white-box, black-box, or grey-box manner. Furthermore, quality expectations, schedules, and available resources affect the testing strategy. Since functional testing concentrates on the behavior of the system, it binds functional requirements to the actual functionality of the system. However, not all system characteristics can be considered functional or testable and verifiable using functional testing methodologies. These properties include security, robustness, reliability, and performance, and they are examples of non-functional properties.

Non-functional testing strives to evaluate non-functional system characteristics. In general, non-functional testing is more difficult than functional testing due to the more imprecise nature of the mapping between the testing objectives and the input-output relationship. Implementing a test case that exercises the non-functional characteristics using a pre-defined set of inputs and expected outputs is more difficult to formulate than a corresponding functional test verifying system functionality. For instance, non-functional issues, such as security, are typically scattered into a number of places throughout the system instead of being implemented as a single module [7].

2.2 Testing process

Software testing, especially with large and complex systems, is often regarded as an effort of its own, distinct from software development, and is conducted as a separate project. As such, modern software testing is performed according to an established testing process, which includes different testing phases, testing levels, and testing steps.

Testing phases

The software testing process includes testing-related phases such as planning, analysis, design, implementation, execution, and maintenance [4, 8]. First, test planning selects relevant testing strategies and analysis sets testing objectives. The testing strategy defines how testing is to be performed and the testing objectives define what is to be accomplished by testing. Test design specifies the tests to be developed and implemented as test procedures and test cases at the implementation phase. In test execution the test cases are

run and updated in the maintenance phase according to changes in the SUT implementation, specifications, or test objectives, for instance.

Similar to the software development process, a number of documents are associated with different testing phases. At the planning and analysis phase a test plan defines what is to be done and with what resources, for instance. Roughly, the test documentation consists at least of a test plan, test specifications, test cases, and a test report [9]. Based on the test plan a test specification defines in the design phase the testing objectives, test conditions, and pass/fail criteria. A test case documents the test setup, data, and expected outputs for test implementation. Finally, in the test execution phase a test report documents the incidents and observed behavior. A test case is thus a basic instance for test execution and a test report records the execution. The contents of these documents vary, depending on the associated testing level.

Testing levels

Software development processes, the traditional V-model [10] and the spiral model [11], as in iterative development [12] for instance, involve testing as part of the overall development process. The V-model, illustrated in Figure 3, is an example of a traditional view of dependencies between development and verification levels, where each development level has a corresponding testing level. In iterative development each iteration refines system functionality, thus involving iterative testing, too. The iterative testing process is illustrated in Figure 4, where testing is considered as a separate development phase in the overall development process. Although considered a separate development phase, testing is typically performed on different levels also in

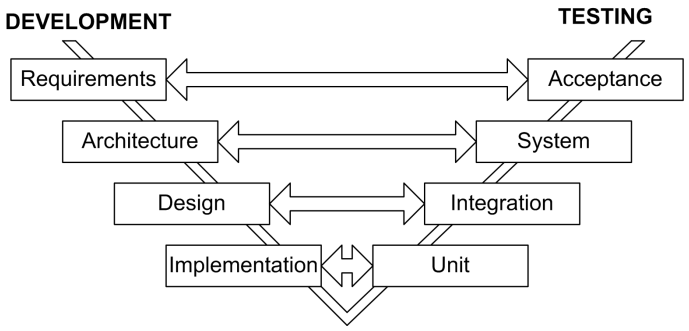

Figure 3: Traditional view of development and verification levels in software software development, the V-model as adapted from [10].

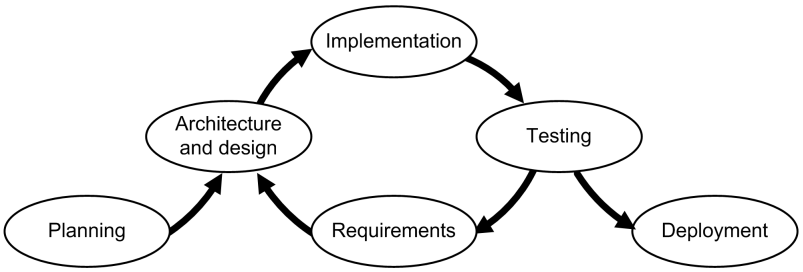

Figure 4: Iterative software development as adapted from [12].

the iterative process, as described in the V-model.

Testing can be divided into levels, depending on the testing objectives and the development level the testing is targeted to. These testing levels are divided based on the level of abstraction. Unit and component testing target the development in the lowest abstraction levels, whereas system and acceptance testing target system level issues, thus operating on higher levels of abstraction. Initial expectations for the system behavior are set against the requirements, to which the system behavior is typically compared in the acceptance testing phase. Hence, the system behavior is evaluated against the requirements prior to deploying the software, while unit and integration testing aim at verifying the quality of the outcome from the corresponding development phase.

Testing steps

Undertaking testing requires a number of associated testing steps to be performed, including preparations, execution, and restorations [9]. Based on the test specification, a certain amount of preparations are required to set the SUT into proper state for testing, often including instrumenting the code with related testware. This typically involves a re-compilation and a specific build of the software, dedicated for testing purposes. Hence, the SUT must be prepared prior to being able to execute the tests in a controlled manner.

After preparations the test execution takes place by exercising test cases according to the test specification. During test execution the SUT receives data defined by the test cases and depending on the testing level – data can be considerably different whether it is acceptance testing or unit testing, for instance – from the test execution and the produced outputs are recorded in the test report.

Depending on the testing strategy, the state of the SUT should be restored after test execution in order to continue with testing, either in the

case of multiple test cases run on the same occasion, or in order to return to normal operation of the system. Furthermore, testware is often unnecessary in the production code and the system should be restored to its normal condition prior to deployment. Identified and reported incidents as a result of test execution provide information on the further development needs in comparison to expected outcomes. In unit testing this is feedback for the developer about the correctness of the unit behavior against the implementation, and in acceptance testing about the criteria set for the product by stakeholders, for instance. Furthermore, test execution provides feedback for test development, as the test cases or the testware might require further development as well. This process is illustrated in Figure 5.

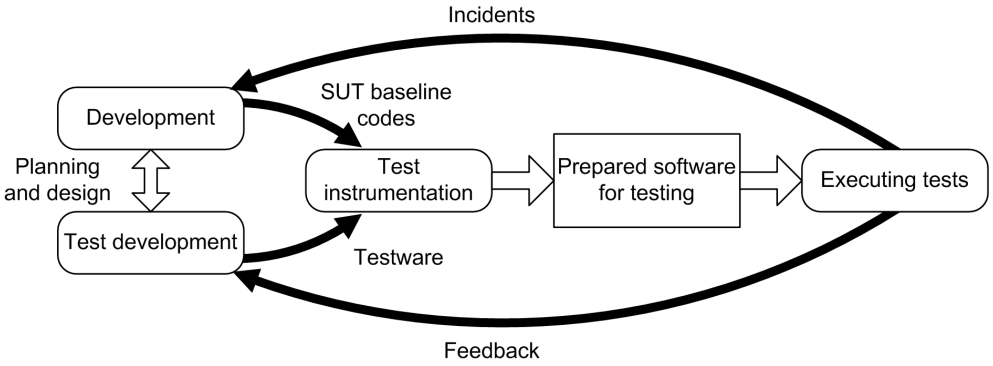

Figure 5: Testing typically requires preparing the SUT for testing by instrumenting the system with proper testware. Derived from [9].

2.3 Increasing testability

Testability of a system depends on various issues. In general testability is defined as follows:

> "Testability. (1) The degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met. (2) The degree to which a requirement is stated in terms that permit establishment of test criteria and performance of tests to determine whether those criteria have been met" [6].

According to the definition, the common testability needs are independent of the context or testing targets. These are the ability to set up test cases, satisfying pre-conditions, the ability to access the internal state of the

system, and the ability to decide the test results [13]. Executing test cases requires satisfying the preconditions that are set for the test case execution. This usually requires executing predefined events, deploying data values, and setting system internal states, for instance. Furthermore, a method of governing which parts of the test code are excluded from the final products should be established.

Proper testing is often dependent on the testability of the system, that is, system properties that enable the implementing of the testware. Typically this means extra code, and with complex and large systems implementing and maintaining proper testware is a major issue. Complex applications are commonly built by composing open-source, commercial off-the-shelf (COTS), legacy, and custom-made components creating joint behaviors.

A software architecture describes the relationships between the components constituting systems and subsystems facilitating different configurations and maintenance operations. It is commonly understood that the system design and architecture have a significant effect on the required development effort [14, 15, 16] and explicit software architecture design has clear benefits. Furthermore, software product lines having a common core design and implementation are enabled by a solid architecture describing the commonalities and variation points. That said, testability should be addressed already in the architecture definition phase.

Macros and interfaces are the most common and widely acknowledged design-level techniques for increasing testability [17]. Furthermore, the programming level assertions provide a simple and generic method for evaluating expressions and terminating the program in case the assertion fails. Embedding assertions in code allows the evaluating of the program correctness and monitoring the system state, while macros and interfaces are methods of extending the implementation for testing purposes.

Macros enable the inclusion and exclusion of test code in selected places in the system structure. On the implementation level this provides a compilation-time variation technique for separating testware and production code. As an example, in C++ macros can be defined using the pre-compiler directive `#define` and included or excluded in code using the directives `#ifdef` and `#ifndef`, respectively. Based on the directives and macro definitions the pre-compiler either includes (if defined) or excludes (if not defined) the code segment enclosed by the macro. Macros are commonly used amongst developers, especially during the pragmatic coding effort, but require an understanding of the actual code structure. Hence, macros represent a code-level approach to introduce testing viewpoints, whereas interfaces hide the component internal structure and provide an easy access to the component functionality. Interfaces allow a clear separation between the SUT and the

testware. However, using interfaces requires implementing the related logic behind test interfaces.

A common technique to implement testing functionality behind interfaces is to use stubs [18] or mock objects [19, 20]. These simplified components are used to replace the actual behaviour behind the interface for testing purposes. This allows the SUT to behave normally, although the actual surrounding components are not exercised, thus isolating the SUT from the surroundings for testing. Typically such stubs are dummy presentations of the original interface and do not include more than the dummy interface. It is argued [21] that mock objects add test logic to stub objects, thus making them a more advanced technique for testing. However, from the architecture and testability perspective the stub and mock objects share a number of commonalities.

All these techniques operate on the existing SUT and offer improved testability of the system. However, these approaches do not consider testability as a system design issue and are typically applied only at the test case design phase after the SUT has been completed. The issue of testability as a design artefact is addressed by Test Driven Development (TDD) [22, 23], which is a design methodology from the testing perspective. Although TDD is a design methodology for developers, it is still a testing methodology for testers from the testing point of view. In TDD an automated unit test is written prior to writing the actual code and based on the test runs the software is reworked to satisfy the tests. Hence, the software evolves based on the test iterations and related code improvements. TDD promotes simple design composed of smaller units as the developers target passing the tests, thus avoiding complex structures. Thus, TDD is not a technique for implementing testing but more of a design methodology. Writing test code for TDD utilizes the same techniques for coding the testware as other design methodologies.

2.4 Problems related to testing

Successful system architecture and overall design are considered the most important factors in building successful and functional software systems. While constraints and guidelines for the design are set by the architecture, the system architecture is determined on basis of the requirements. Hence, the requirements, often set by stakeholders, define the targets for the system architecture and expected system design. Considering testing as an activity that targets unspecified behavior of the system and shows that the system fulfills requirements, deriving test cases directly from the requirements requires an approach to formulate test cases to present these expectations. Ultimately, testing is about fulfilling exactly these objectives.

Limitations in expressiveness

When using conventional techniques it is difficult to formulate all the necessary test cases and implement the related testware that covers the needed testing concerns. It is difficult to anticipate the resulting implementation already in the requirements phase, when neither the system design nor the architecture is defined. Such test cases would result in written descriptions of pre-conditions, actions, and expected outcomes, perhaps in relation to system components, subsystems, or similar. Due to the strictly technically oriented nature of conventional techniques, the semantics of the test case implementations are not expressive enough to cover such concerns. It is argued, though, that the earlier the testing concerns are taken into strong consideration, the better are the results gained in both testing and in system design [14].

Furthermore, while the conventional approaches have proven efficient in capturing the functional testing issues under evaluation, non-functional and system-wide issues are difficult, if not impossible, to test using conventional methods. Conventional techniques suffer from scattering implementations of concerns to various components and code tangling as single components implement multiple concerns [24]. Consider, for example, memory allocations and de-allocations. Memory operations are scattered throughout the code, and no single interface can be harnessed for testing that no memory leaks or similar problems exist. Furthermore, a tracing support would require implementing a related code snippet into all the places throughout the code to invoke the tracing functionality when necessary. A common issue in both of these examples is code tangling and scattering: the required testware must be written amongst the original code, and test code implementation is scattered throughout the system, thus breaking the modularity of the system.

Invasive techniques

With the tangling and scattering problems, it is evident that the test code is intermixed with the original code, making it harder to separate the test code from the original implementation. In resource-aware systems, such as embedded systems, the excessive code for testware must be minimized, and thus proper methods to separate the testware from the original system are required. This is particularly problematic with macros. Test code separation after a couple of iterations is difficult: the number of macro definitions and related code snippets has become large and produces a complex mixture of testware and SUT code that is no longer manageable. The code segments belonging to the original implementation and the ones related to testing

are tightly bound together. Hence, it is difficult to create test code that is both reusable and maintainable. Furthermore, managing such code segments, possibly scattered throughout the whole system and behind a large number of different pre-compiler directives, calls for designing a systematic method.

Some of the above problems, for instance problems related to separating the testware from the original code and testware maintenance, are solved when using interfaces. The separation of testware and the original implementation is explicit, and thus code tangling is no more as big an issue. Furthermore, using interfaces allows the reusing of the test code, as long as the test functionality as such is reusable. Variation of such test interfaces is more straightforward and provides a technique to adapt the testware to different systems. However, providing the necessary interfaces for testing can be complicated. It is possible that the system does not encourage the introducing of new interfaces or the existing interfaces are too simple or complex to modify for testing purposes. Nevertheless, from the software architecture point of view the approach of interfaces better promotes modularisation and reuse.

Implementing stubs or mock objects, special testing interfaces, or simple test code behind pre-compiler directives are thus invasive techniques that alter the original implementation in favour of testing. Such a test-related alteration is the selection of stub interface instead of the original interface, for instance. This is cumbersome if the original implementation does not enable such behaviour, for example in the case of COTS or old legacy code if the documentation of the code is insufficient to allow future developers to follow the thinking. Furthermore, although there have been attempts to withdraw these roles, test developers are not, and should not be, developers. Thus, they should not be required to understand the details of the code structure in order to be capable of developing good test cases. Since conventional techniques are invasive in nature, i.e. affect the original implementation, more advanced techniques are required in order to manage the testing of components that cannot be modified. Such a test system both supports the testing by keeping the original code intact and protects the code from testware-related alterations that could, in the worst case, affect not only the test results but also the original functionality.

Overall testability issues

The testability as a concern is typically not included in the original set of system design concerns. The system fulfills the stakeholders' expectations and no customer generally explicitly requires testability, although a positive outcome from the testing process would be desirable. In mobile settings in

general the conventional techniques to promote testability concentrate on the implementation level and neglect the testing in the architectural or design respect. This is partly a result of the available techniques that dictate the manifestation of the testing concerns into implementation level elements, thus limiting the possibilities to concentrate testing on any higher levels of abstraction.

When using conventional techniques, increasing the system testability and implementing test cases that capture testing concerns under evaluation requires a considerable amount of understanding of the resulting implementation and therefore proper insight into the system design and behaviors. Typically, such information is not available, since testing is considered to be the last and necessary, but undesirable, part of a typical software project and potentially can be outsourced. Hence, the lack of required insight into the system is evident, as well as the lack of testability artifacts in the design, especially when testing is performed by external teams as a separate project.

These issues are highlighted in Test Driven Development (TDD), where the testing is considered a more important design issue than in traditional development. However, since the target is at the unit-testing level, the method is inconvenient when system-wide issues are to be considered. Furthermore, it can be argued that TDD is difficult when considering functional testing that requires complete functionality. Although testing is better addressed in TDD than in traditional software development, the problems related to conventional implementation techniques are not solved by changing the design methodology.

2.5 Example

As a simple example of software testing, consider a simple pedometer application. The pedometer hardware is a physical sensor that senses acceleration and is used to measure steps during a walk or run, for instance. A related application calculates the traveled distance based on the measured step count and given step length. The application uses services from a device driver, which further controls and receives data from the hardware sensor.

The C++ code snippet illustrated in Listing 1 presents an example implementation of the simple pedometer controller.[2] For black-box testing there are three classes: `PedometerCtrl`, `HWPedometer`, and `Client`. The pedometer controller class `PedometerCtrl` controls the actual hardware device driver, `HWPedometer`, which calls the `AddStep` callback to add a step to

[2]The example code includes intentional problems related to bad coding style and potential errors.

```
class PedometerCtrl : public Callback {
public:
  static PedometerCtrl* NewCtrl( int length );
  void Start();             // Start measurements
  void Stop();              // Stop measurements
  double Distance();        // Calculate traveled distance
  virtual void AddStep(); // Callback for device driver
private:
  PedometerCtrl( int length );
  ~PedometerCtrl();
  int _length;        // Single step length in centimeters
  double _distance;  // Travelled distance
  PedometerDrv* _hw; // Pointer to device driver
  };

PedometerCtrl* PedometerCtrl::NewCtrl( int length ){
  PedometerCtrl* p = new PedometerCtrl( length );
  return p;
  }

PedometerCtrl::PedometerCtrl( int length ):
  _length(length),_distance(0),_hw(NULL) {}

PedometerCtrl::~PedometerCtrl() {}

void PedometerCtrl::Start() {
  _distance = 0;
  if (_hw == NULL) _hw = new PedometerDrv();
  if (_hw == NULL) Abort("Pedometer allocation failed");
  _hw->initialize( this );
  }

void PedometerCtrl::Stop() {
  delete _hw;
  _hw = NULL;
}

double PedometerCtrl::Distance() { return _distance; }

void PedometerCtrl::AddStep() { _distance += _length; }

class Client {
  public:
    Client();
    ~Client();
    double Measure( int step_length, double time );
};

Client::Client() {}
Client::~Client() {}

double Client::Measure( int step_length, double time ) {
  PedometerCtrl* p = PedometerCtrl::NewCtrl( step_length );
  p->Start();
  Timer* timer = Timer::New( time, this );
  timer->Start();
  WaitForTimer( timer );
  p->Stop();
  return p->Distance();
  }
```

Listing 1: Example code for a simple pedometer application.

the measurement on each interrupt caused by the motion sensor. `Client` is a simple class representing a client code for distance measuring application. It provides a simple function for measuring distance for a given period of time. A class diagram of the example application is illustrated in Figure 6.

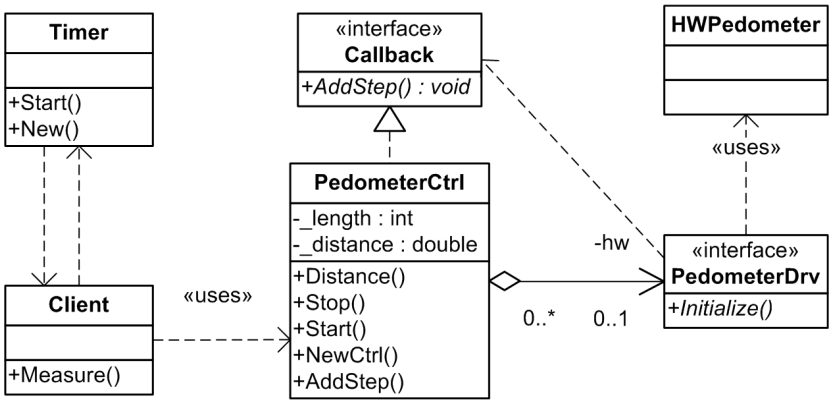

Figure 6: Class diagram of the example code.

If one concentrates only on the interfaces, a couple of programming errors, obvious when reading the code, are not noticed. For example:

According to the code, it is possible to create multiple instances of the `PedometerCtrl`, since the factory function `PedometerCtrl::NewCtrl` does not limit the number of instances. This is a potential problem if the pedometer controller is intended to implement the singleton design pattern [25], which we assume is the case in this example. Furthermore, there can be other users for the controller class in addition to the measuring function described here, as there are no limitations in that either. However, we can exploit this property in our test code and the singleton issue can be easily tested by simple test code, as presented in Listing 2.

```
1 // Test singleton pattern: Try to create two instances.
2 // If pointers match the implementation is correct.
3 PedometerCtrl* p1 = PedometerCtrl::NewCtrl( 50 );
4 PedometerCtrl* p2 = PedometerCtrl::NewCtrl( 100 );
5 if ( p1 == p2 ) // If pointers match, the singleton works
6   test_result = TEST_PASSED;
7 else
8   test_result = TEST_FAILED;
```

Listing 2: Example code for a simple singleton check.

Further considering the simple example, the following issues arise:

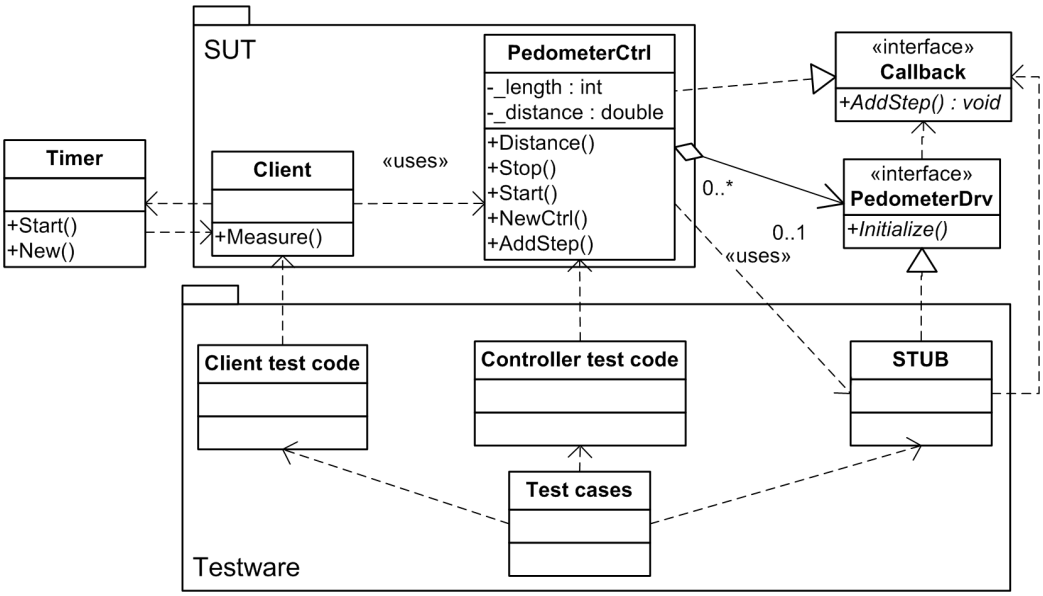

Figure 7: Test setup for the example code.

- Memory leaks are possible due to the lack of garbage collection. Based on studying the code, it is apparent that the code leaks memory under certain conditions.
- There is no protection against problems arising from out of memory situations. On lines 17, 28, 53, and 55 the memory allocation might fail due to either a bug in the code or because of any other reason.
- The code assumes that the timer is able to complete after the given time. However, there is no guarantee that the timer is correct and testing for a situation where the timer does not complete requires a test stub to be implemented.
- Testing the interface towards the hardware device driver also requires a stub implementation. A common problem related to hardware devices is possible jamming of the client code. In this example this could cause the device driver initialization call on line 30 to fail to complete.
- The callback function `AddStep`, used to increase the distance, may also be called by other objects, thus causing the measurement data to be incorrect. This suggests a difficult case to test.

As a result of this analysis the following test setup is to be implemented. In the diagram in Figure 7 we illustrate the basic setup of test stubs, the SUT

and the testware executing the test cases. Using the static analysis tools, code inspections, and basic structural testing, all the aforementioned issues can be resolved and a satisfactory test coverage achieved. However, success in finding all the aforementioned issues requires a considerable amount of software developer skills and insight into the related programming issues. Furthermore, testing more complex and larger systems is extremely laborious and expensive because of the amount of required human work.

2.6 Summary

Although sound support for testing is available, and tools and techniques provide means to implement testing, there are a considerable number of problems related to immature software. With the increasing complexity of the systems to be tested, more advanced techniques are required. The emergence of new types of issues requires new techniques to capture them for testing. Furthermore, separating the testability concern from the others presents an efficient method for modularizing testability issues and to achieve better satisfaction of the requirements. The problems of separating the testware from the original design as well as requirements for non-invasive methods for implementation calls for a new technique for capturing all the scattered, system-wide testability issues for testing. A non-invasive technique with enough expressiveness allows a solution by modularising testing concerns as reusable and manageable units that can be used throughout the development process, starting already at the requirements phase.

3 Aspect-Orientation

Aspect-orientation is a method to increase modularity by introducing improved facilities for the separation of concerns. In the following the principles of aspect-orientation are discussed, followed by a brief description of aspect-oriented languages and related implementation considerations. Unless otherwise indicated, the discussion is based mainly on Filman et al. [1, pages 1–35].

3.1 Fundamentals

Aspect-orientation is a modularization paradigm that seeks to express concerns separately and their automatic composition into working systems. A basic approach is to argue that AOP languages allow "making quantified programmatic assertions over programs that lack local notation indicating the invocations of these assertions", and, hence, the basic properties necessary for AOP are *obliviousness* and *quantification* [26].

Obliviousness

Obliviousness implies that the original program is unaware of the aspects and the aspect code cannot be identified based on examining the original code. The concerns can be separated at higher-level specifications instead of low-level implementations during the system creation process. The earliest computer programing languages were *unitary* and *local*: each statement had effect on exactly one place in the program and the effect was located closely to statements around it. Since then programming languages have evolved away from local and unitary languages and allow the making of quantified statements that have the ability to affect a number of places in the program. For instance, object-oriented programming introduces inheritance and other related mechanisms that make the execution non-local. The ability of oblivious quantification distinguishes the AOP from conventional programming languages and an essential feature of an AOP language is thus explicit support to modularize cross-cutting concerns as single units.

Quantification

For a system to be AOP, the demands for locality and unitarity demands must be broken. The organization of the program and the realization of the concerns should be able to be formed in the most appropriate way for coding and maintenance. The AOP statements thus define behavior based on actions in a set of programs. This leads to three major choices: the conditions we can define, how actions interact, and how the program is arranged to incorporate the actions. The AOP system allows us to quantify over the static structure and its dynamic behavior.

Black-box AOP systems – such as Composition-Filters [27] – quantify over the components' public interfaces, for example by wrapping components with aspect behavior. *Clear-box* AOP systems – such as AspectJ [28] – quantify over internal structure of components. This can be implemented as a static quantification with pattern matching on the program structure or decorating subprogram calls with aspects, for instance. Both techniques have their advantages and disadvantages. Clear-box techniques require insight into the source or object code, but allow access to the program details and can easily implement aspects associated with the caller side environment of the subprogram. However, black-box techniques are easier to implement and can be used with components whose source code is not available.

Implementing clear-box techniques typically requires implementing a major part of a compiler that is at least able to create a parsed version of the underlying software. Black-box techniques quantify over the program's interfaces that are likely to produce reusable and maintainable aspects. In addition to static quantification, in dynamic quantification aspect behavior is tied to something happening at run-time, such as raising an exception, context switch, and the call stack exceeding a certain depth, for instance. The choice of program properties depends on the programming language and although missing explicitly from the language, an aspect language may still allow quantification over them. In this thesis we limit the discussion to the modular AOP languages AspectJ [28] and AspectC++ [29] and related characteristics of AOP languages. These languages will be addressed in Section 3.3.

Terminology

In order to further discuss AOP, we must define certain fundamental terminology. A *concern* refers to anything any engineering process is taking care of, and it could be high-level requirements, low-level implementation issues, measurable properties, aesthetic, or things involving systematic behav-

ior. *Cross-cutting concerns* are concerns whose implementation is scattered throughout the implementation of the rest of the system. Implementing such cross-cutting concerns results in *scattering*, where instead of promoting modularity, the code is scattered to multiple modules. *Code tangling* is a result of single components or methods implementing multiple concerns, where the code for implementing the concerns gets intermixed. *Aspect*, on the other hand, is a modular unit implementing a concern, and *join points* are definitions used to describe places where and when to attach this additional code. Furthermore, an *advice* determines the desired behavior at the join points. Figure 8 illustrates the concept of modularization of scattered implementations using AOP.

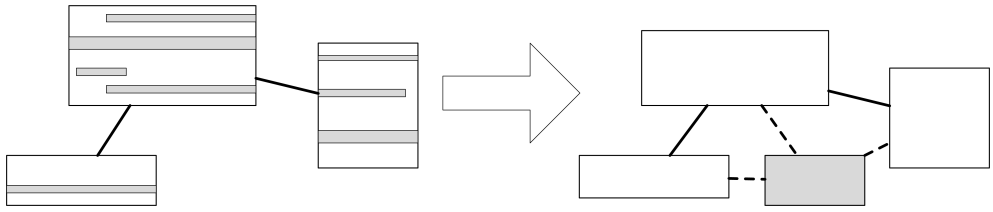

Figure 8: An object-oriented design with scattered implementations transformed to a module using AOP. Grey areas represent the code modularized as an aspect.

The programmer of the original code is *oblivious* to the advice code, which is contrary to the conventional method of modularization using subprograms. To describe one or multiple places within a program to invoke an aspect AOP uses a *pointcut designator* (also known as *pointcut*), which defines a collection of join points. Pointcuts are thus a way of describing all the places something must happen using a single statement and an advice implements this additional behavior. The method of bringing separately created software together, thus forming a combined piece of software is called *composition*, which typically involves checking that the elements fit together. *Weaving* is the process of composing working systems with aspects included. In practice, the AOP languages define several mechanisms for weaving, including statically compiling the advice to the code, for instance.

3.2 Aspect-oriented software development

Adopting an aspect-oriented approach to software development, or software testing, requires employing a number of different methods and techniques. In other words, in order to utilize the separation of concerns conventional design, specification, and implementation methods must be equipped with related tools.

Aspects in design

Due to the size and complexity of recent software systems, their development involves a considerable amount of describing the architecture and design, which are defined using computer-aided software engineering (CASE) tools and modeling languages, for instance. An example of such language is Unified Modeling Language (UML) [30], which is a common notation technique for designing object-oriented software systems. In order to successfully model aspect-oriented concepts, join points and aspects, the corresponding elements must be included in the UML language. Extending UML for AOP purposes has been widely studied, including studies in activity diagrams [31], profiles [32, 33, 34], stereotypes [35, 36], and state-transition diagrams [37], for instance.

A key issue in modeling AOP is specifying concerns and the related aspect invocations together. Static diagrams, for instance a class diagram, are used to present the relationships between the elements and related aspect pointcuts. In this structural description the problems arise from the distribution of concerns into a number of elements, thus making it difficult to illustrate aspect relationships without aspect-oriented extensions to the language. Comprehensive techniques have been proposed to solve these problems, although no single technique can be preferred over others. These methods typically exclude the actual weaving process, and the dynamic characteristics are modeled in behavioral models.

Aspects are also considered as part of architecture engineering and related engineering and architecture design methods have been proposed, for instance Aspect-Oriented Requirements Engineering (AORE) [38] and Aspectual Software Architecture Analysis Method (ASAAM) [39]. These approaches target applying aspect orientation in the early stages of software development. However, it is explicitly recognized that these produce not only design level concerns but also cross-cutting concerns on the requirements level. In addition to these architectural aspects, an approach to defining *early aspects* [40] is presented to capture such cross-cutting concerns on the requirement specification level.

Programming aspects

On the programming level *pointcut* provides the quantification mechanism as a method of describing an activity to be performed in multiple places in a program. A pointcut designator describes a set of join points, thus allowing the programmer to designate all join points in a program without explicitly referring to each of the join points. A join point model defines the frames

for the AOP language to define the places in program structure or execution flow where the aspect code can be attached. In AspectJ and AspectC++ the elements of the join point model are method or construction call or execution, for instance. AspectC++ supports name and code pointcuts. Name pointcuts refer to statically known program entities, whereas code pointcuts refer to the control flow of the program. Furthermore, pointcut designators for name pointcuts are given as match expressions, and as an example, describing any call join point inside a program can be formulated as:

```
pointcut tracer() = call("% %::%(...)");
```

where the pointcut definition matches all methods in all classes, since % is interpreted by the aspect compiler as a wildcard, and the character sequence `...` matches any number of parameters. Code pointcuts are defined with the help of aspect language functions and name pointcuts. For instance, a pointcut function `within(pointcut)` can be used to limit the aforementioned pointcut to methods of a certain class:

```
pointcut tracer() = call("% %::%(...)")&&
                    within("myClass");
```

An advice is a set of program statements that are executed at the join point based on the matched pointcut. The most common types of advice are *before*, *after*, and *around* advice, thus executing the advice code before, after or around the join point. For instance, an advice for printing out the method signature before executing any of the methods of `myClass` would be defined as:

```
advice tracer():  before() {printf(tjp->signature());}
```

An aspect is a unit of modularity and is declared using the keyword `aspect`. Aspects have similar characteristics to classes in object-oriented programming and have attributes, methods and support inheritance. Furthermore, aspects have their own state and behavior. In AspectC++ aspects have the same basic structure as C++ classes and have exactly the same structure. In addition to class characteristics, aspects can contain pointcuts, advices, and inter-type declarations. Hence, a simple tracing aspect could be written as:

```
aspect TracingAspect {
  pointcut tracer()=call("% %::%(...)")&&within("myClass");
  advice tracer():before() { printf(tjp->signature()); }
};
```

From the advices the `around` advice is the most powerful one, allowing controlling of the join point invocation inside the advice code. The before and after advices execute the join point implicitly, whereas the join point execution must be explicitly called inside the around advice. However, the possibility to define execution both before and after the join point using single advice makes the around advice useful for testing purposes. The around advice invocation can be illustrated as a sequence diagram, where the `before` advice is executed before and `after` advice after the join point execution, as illustrated in Figure 9. Corresponding aspect code is written as follows:

```
aspect TracingAspect {
 pointcut tracer()=call("% %::methodX(...)");
 advice tracer() :  around() {
  // Here the before part printing the method signature
  printf(tjp->signature());
  tjp->proceed(); // execute original method
  // Here the after part
 };
};
```

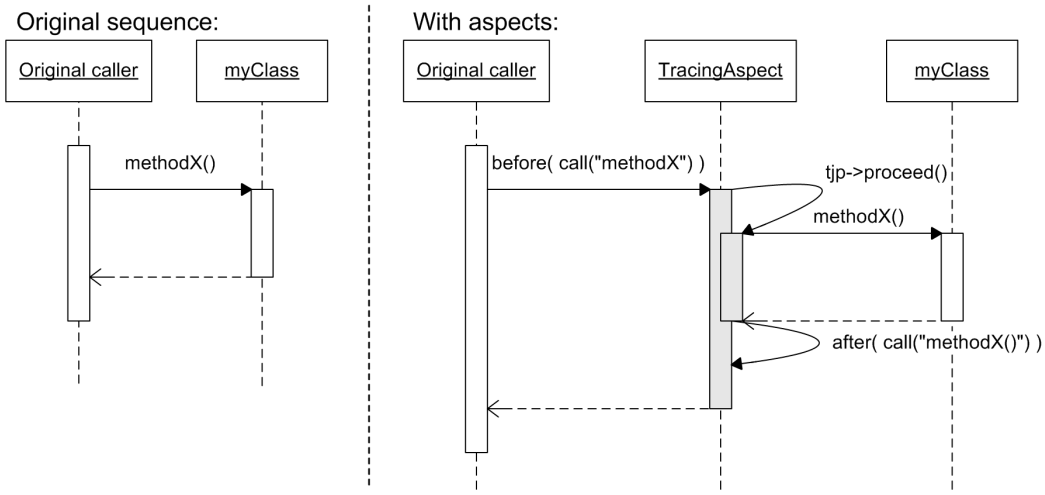

Figure 9: Advice invocation as a sequence diagram. The grey activation denotes the around advice activation.

Aspect instantiation

Aspect weaving is a mechanism of coordinating aspect composition with the other modules of the system. Aspects are woven into the system using a

specialized compiler, called aspect weaver. For instance, AspectJ is woven at bytecode level. Bytecode transformation adds aspects into existing code by modifying compiled Java class files. The transformation is performed by first traversing the methods to clarify the method visits, followed by selecting the necessary join points according to the transformation, and finally generating the resulting code including the sequences of new code. Static transformation relies on the symbolic information of the class files and dynamic transformation at class load time when the class enters the Java Virtual Machine (JVM) by altering the operation of the class loader in Java. In other words, dynamic transformation operates by transforming classes at the time they are loaded.

This means that in AspectJ bytecode of the aspects and the code the aspects should affect must be available for the weaving to succeed, and allows operating outside the source code. AspectC++ aspects are woven on the source code level, and the associated AOP compilers are source-to-source compilers. In weaving the compiler replaces the original function calls with calls to the aspect's advice code and executes the original code in context of the woven aspect code, for instance.

3.3 AspectJ and AspectC++

The AspectJ language, an extension of Java language, is arguably the most commonly acknowledged programming level technique to implement aspect-orientation. Due to the popularity of the Java programming language, AspectJ has become the de-facto standard in aspect-oriented programming. While Java is a popular language in programming, the performance requirements of running the Java virtual machine make it a non-desirable language for implementing resource critical systems, such as embedded systems often written in C or C++, for instance.

The C++ support for aspect-oriented programming is not as popular as it is for Java. However, a couple of approaches exist: Flexible Object Generator (FOG) [41], OpenC++ [42], AspectC [43], and AspectC++. Both FOG and OpenC++ are meta-programming approaches to support a superset of the language to allow a meta-program to manipulate the base-level C++ code, but they do not provide a sufficient join point model nor AOP language extensions to specifically support AOP. OpenC++ is targeted for plain C and no plans exist to support C++, and although based on AspectJ, OpenC++ does not support object-oriented characteristics due to the non-object-oriented nature of the C language. Hence, the choice for AOP programming in C++ is AspectC++, since it provides object-oriented features as well as a proper join point model and AOP-specific extensions to the C++. It should be noted here that FeatureC++ [44], a feature-oriented

Table 1: Summary of AOP language characteristics.

Language	Join point model	AOP extensions	O-O characteristics
FOG	insufficient	no	yes
OpenC++	insufficient	no	no
AspectC	yes	no	no
AspectC++	yes	yes	yes
FeatureC++	yes	from AspectC++	yes

```
1 public aspect TestAspect extends GenericTestAspect {
2
3   pointcut callBackCalled(PedometerCtrl pedometer):
4   execution(void PedometerCtrl.AddStep(..)) && target(pedometer);
5   after(PedometerCtrl pedometer): callBackCalled(pedometer)
6   {
7     // here perform whatever necessary
8   }
9 }
```

Listing 3: AspectJ implementation of the example aspect.

language extension to C++, also supports AOP, but uses AspectC++ to implement the related AOP extensions. Differences between the AOP languages discussed here are summarized in Table 1.

Semantically, AspectJ and AspectC++ are very close to each other, with differences only due to the base languages. Both AspectJ and AspectC++ extend the language by adding aspect-orientation constructs. Hence, most of the properties of the original language are supported and the writing aspect code is syntactically no different than writing Java or C++, respectively. For example, inheritance is supported by both aspect languages. However, the aspect code cannot be compiled using native language compilers but requires specific compilers to be used instead.

As an example, consider an aspect derived from aspect `GenericTestAspect` with a pointcut defined to the execution of `AddStep` callback function for the pedometer controller introduced in Listing 1. The resulting aspect code for Java is presented in Listing 3, and for C++ in Listing 4.

The following case studies have been implemented using AspectC++ due to the limitations of the system used in the experiments that is written in C++ and utilized in the context of a smartphone.

```
1 aspect TestAspect : public GenericTestAspect {
2
3   pointcut callBackCalled() = execution(
4   "void PedometerCtrl::AddStep(...)") && within(PedometerCtrl);
5   advice callBackCalled() : after() {
6     // here perform whatever necessary
7   };
8 };
```

Listing 4: AspectC++ implementation of the example aspect.

3.4 Summary

Aspect-oriented programming provides the separation of concerns into manageable units. The key properties of AOP are obliviousness and quantification, which separate AOP languages from conventional languages. A variety of approaches to support AOP in CASE tools and techniques have been researched and the number of choices makes it difficult to nominate the most suitable one. Due to the popularity of Java, Java-based approaches have gained the most support, but the implementations vary a lot according to the context and goals. As a programming level technique, AspectC++ provides AOP support for the embedded system domain, where other AOP languages fail to meet the performance criterion.

4 Applying AOP in Testing Object-Oriented Systems

In this chapter AOP is discussed in the context of testing. First, testing-related characteristics suitable for aspect-oriented treatment are discussed, followed by a discussion on the required changes to software development and testing processes. Towards the end of the chapter an aspect-oriented approach for implementing the testware in the scope of this thesis is defined.

4.1 Separation of concerns for testing

Considering the concept of separation of concerns from the testing point of view, testing can be separated as specific concerns. These *testing concerns* represent conceptual parts of the system related to the testing of the system. Analogous to functional characteristics implementing non-functional concerns, testing system characteristics implement testing concerns. However, any other concern implies a related testing concern that should be covered by testing. For instance, *robustness* represents a non-functional concern that cannot be directly connected to single elements of code but is a system-wide issue, though still a part of the system and its characteristics. This presents a system-wide testing concern to verify system robustness. Testing is thus considered an equal design concern.

We propose an approach where the test objectives are derived from requirements and formulated on a relatively generic level. Furthermore, test cases covering the test objectives are formulated as test aspects. These aspects can be further specialized according to the system design and implementation. This relationship between concerns and test aspects is illustrated in Figure 10.

Although test objectives can be high-level descriptions and map of the concerns, the test implementation is not fully design independent. By implementing the required functionality, the system design also manifests both further testability concerns and emerging issues. Figure 11 illustrates the de-

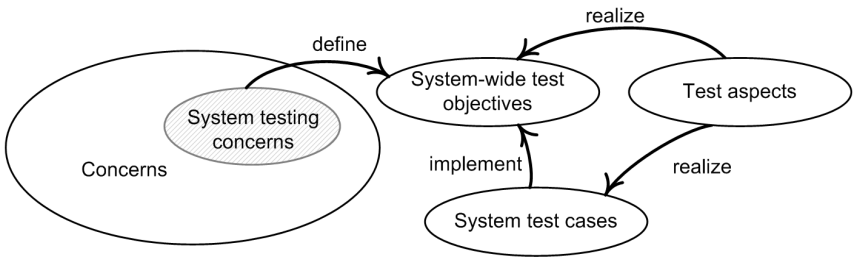

Figure 10: System testing concerns are defined by test objectives and further realized as test aspects.

pendencies between test aspects, SUT, and implementation level test aspects. Composing the system from components introduces testability concerns related to system integrity, robustness, and reliability, for instance. Furthermore, composition and integration result in new behaviors that might not have been anticipated while developing the individual components. These suggest good candidates to be formulated as testing concerns. For instance:

> "Verify that whenever component X and component Y exist in the same system and are executed concurrently, component X has privileged access to resource Z and there should be no simultaneous access from both components at any time of execution".

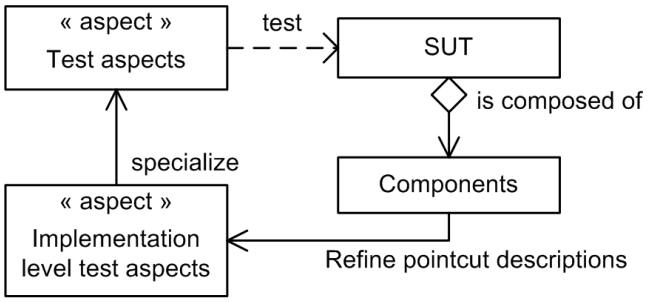

Figure 11: Baseline test aspects are defined to test concerns of the SUT and further specialized as implementation level aspects according to the SUT structure.

Testability concerns can be divided roughly into two categories: non-functional and functional concerns. From non-functional characteristics security, robustness, and reliability suggest concerns that benefit from being modularised as testable units. For example, a testing concern could be:

> "Verify that the system fulfills compliance requirements of standard X."

As a functional concern the approach is more permissive: the functional testing concern can be anything from basic integration testing to regression testing and system testing. Such a functional concern could be defined as:

> "Verify that with input Y the system output is Z."

A very simple example of a functional testing concern is the integration testing objective, which targets verifying that the system functionality is implemented properly.

Defining generic test aspects based on the requirements and test objectives could be difficult without any guidelines. For this purpose we suggest defining a generic baseline for test aspects including guidelines on how test objectives can be translated into aspects. These baseline aspects are refined and varied by implementation level aspects to match the design and code. Such guidelines should include characteristics that are presented by certain cross-cutting concerns and their translation to aspects. These include generic pointcut descriptions, system characteristics and functionalities typically representing such concerns, simplified examples, and patterns for joinpoint and pointcut matching. The process illustrated in Figure 12 describes the proposal related to processing requirements to develop concerns associated with testing further to testability and testing concerns to be considered in testware and system development.

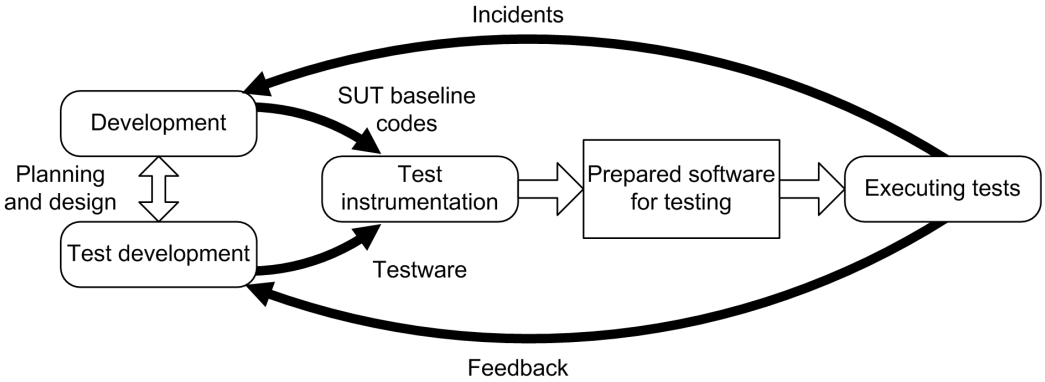

Figure 12: Requirements are processed to derive testability and testing concerns to be addressed respectively in system development and testing.

Example

When considering the example application implementing a simple pedometer controller for measuring traveled distance, presented in Section 2.5, the following requirements could be set for such a system:

- `REQ1`: Using the hardware, the system must measure traveled distance based on the input from the pedometer hardware and length of each step.
- `REQ2`: It must be possible to adjust the length of each step using metric units of measurement.
- `REQ3`: The distance is measured for a given period of time, presented as a number of seconds.

Based on these requirements, a number of testing objectives can be derived. Firstly, the tester wants to guarantee that the system is able to measure the distance by reacting to the interrupts from the hardware. Secondly, the system is able to handle a variety of different lengths and distances. Thirdly, the system is able to measure the distance for a number of different periods of time. As an example of test objectives, the following functional objectives could be defined:

- `TO1`: Verify that the system is able to measure the distance for a given time period using the interrupts from pedometer hardware.
- `TO2`: Incorrect inputs in time and length are treated properly and will not cause unwanted behaviors.
- `TO3`: A certain number of steps results in different distances according to step length and time.
- `TO4`: At any time of execution, no other clients are allowed for the pedometer controller than the pedometer controller application.

Hence, the testing objectives verify the system functionality against the given requirements, for instance `TO1` verifies `REQ1`, `TO2` verifies `REQ2`, and `TO3` verifies `REQ3`, respectively. Stakeholders could also set non-functional requirements for the system, such as, for instance, reliability:

> "The system is able to capture all steps and increase the distance accordingly."

and security:

```
aspect securityAspect {
  pointcut guard() = execution("% PedometerCtrl::%(...)") &&
                     !cflow(execution("% Client::%(...)"));
  advice guard() : before() {
      // Illegal access to PedometerCtrl
      printf("Illegal access to PedometerCtrl from %s",
          tjp->that()->signature());
      abort();
  };
};
```

Listing 5: Implementation level aspect definition for the security aspect example guarding access to the pedometer. In the advice namespace context *tjp* is a pointer to the joinpoint.

> "No other applications can control the pedometer than the pedometer client."

Although the example system is very simple and does not present truly crosscutting issues since it is implemented as a single unit, these concerns are typically implemented by a number of functional concerns, thus scattering the implementation. A generic aspect description for the security concern, described as:

> "Catch any attempts to call pedometer methods by any other class than the client and report an error."

can be formulated based on `TO4`. This aspect can be generalized as a guarding aspect by simply replacing the *pedometer* and *client* with the desired instances and specialized using the following implementation level aspect described in Listing 5.

Hence, the test aspects formulated based on test objectives should be generic in nature, not binding to concrete code elements. The mapping on implementation level is performed at implementation level aspects specializing the generic baseline aspect to the particular component. This allows the creating of generic, reusable, test aspects covering system-wide testing concerns and implementing them at the unit testing phase.

4.2 Impact on the testing process

Adopting aspect-oriented methodology requires changes in the development and testing processes. Depending on the chosen approach, the changes are either technical or process related. If AOP is used in addition to conventional techniques merely as an implementation technique for the testware, the

changes are related to tool chain and implementation techniques. However, if the AOP methodology is applied to the whole testing process, including the analysis and test case definitions, the impact is more extensive. These viewpoints are discussed in more detail in the following.

Test aspect development process

Due to the nature of aspect orientation, the aspect creation process can be based on the test objective descriptions and the property definitions the aspects are supposed to capture. Hence, the definitions can be written using natural language instead of programming languages, thus achieving a more user friendly approach. This is a vital element for declaring aspects for capturing the non-functional concerns under testing. In comparison to the traditional V-model, this means that the development phase and related test development are no longer tightly bound to each other. AOP proposes an iterative model for test development, where generic test objectives are fine-tuned during the development process. The test case descriptions developed based on the requirements are further focused when the system structure advances. Figure 13 illustrates the proposed test aspect development process in relation to phases of the software development process in the V-model. The test aspects are developed in iterations that refine the aspect details according to the corresponding development phases and provide test implementation

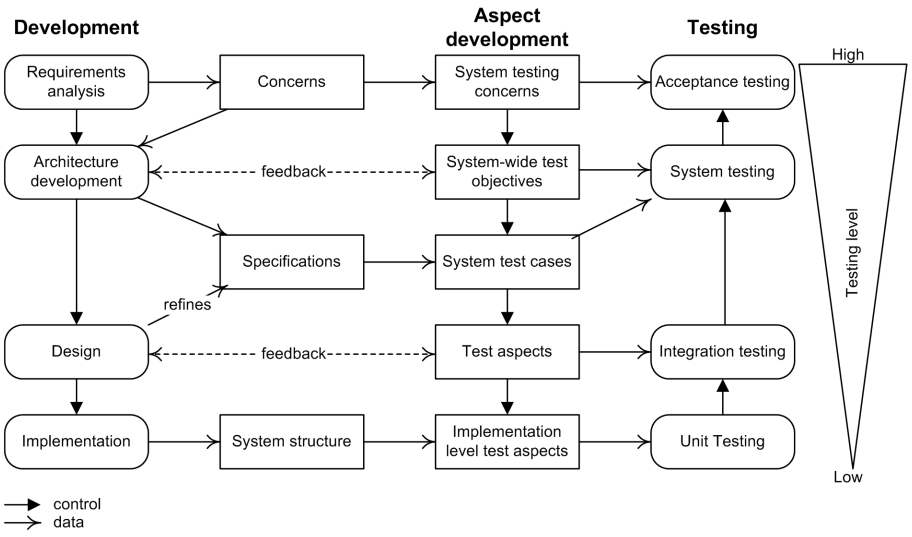

Figure 13: An example of a modified development process for developing test aspects based on the V-model.

to related testing levels.

A pragmatic approach to test implementation using aspects requires an implementation level process to follow. Since programming-level aspects are, due to the aspect language definitions, bound to code elements (function, attribute, type, or variable, for instance), aspect creation can be based on either the code itself or the corresponding model. A class diagram or sequence diagram are examples of a simple starting point that can be used to derive test aspects and the related join points. The original UML model, a class diagram for instance, is used to select and define join points and to create the aspect code. The original code is instrumented with aspects using an aspect weaver that produced the instrumented code and information about the created aspects. This information can be used to update the model according to the created changes and highlight the relationships between aspect and original code. An example of such an aspect creation process is illustrated in Figure 14.

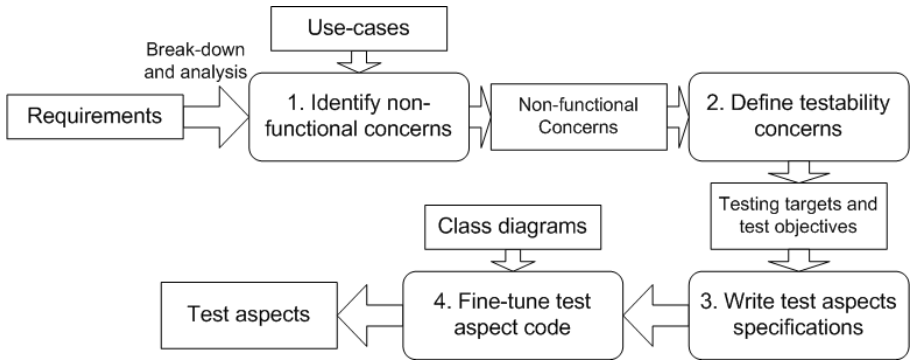

Figure 14: An example process for creating test aspects based on requirements.

Separating SUT and testware in AOP

AspectC++ offers source-to-source compilation for injecting a C++ based SUT with test aspects. In practice, this requires establishing a tool chain that instruments the original source code with aspect code, translates the source code into object code, and finally creates the executable code. Considering the aspect code similarly to baseline code simplifies the configuration management, as the tools remain the same.

In contrast to using test interfaces, there is no need to instrument the original source code with directives separating the implementations for final code from the testware. This can be automatically built into the tool chain,

which is based on information as to whether manipulating the final code or code for testing either includes or excludes the steps necessary for weaving the aspect code into the system. Although this seems to just move the problem from code into the tool chain, it considerably simplifies the separating of SUT and testware. Managing the inclusion of testware on the code level is far more complicated in practice, especially with large systems, than on the tool level, since the problem changes from a diverged problem of scattered implementations across the system into a converged problem, the choice of proper files.

4.3 Testing non-functional properties

Common to all the non-functional requirements is their impact on the whole system instead of single units and, thus, their scattered nature. Since aspect-orientation proposes an approach to modularize such concerns, it promises a solution for capturing the non-functional properties for testing. However, this requires determining the non-functional concerns for testing and defining the related testing targets in order to be able to formulate the desired testing concerns. In the following, we describe a method for deriving testing aspects from the requirements and thus establishing a non-functional testing framework using aspect-orientation.

Formulating non-functional testing aspects

Since requirements set the basic design guidelines and expectations for the system characteristics, we consider them the starting point for defining the non-functional testing objectives. In addition to non-functional requirements, also the functional requirements present expectations for the system functionality and often implicitly set non-functional requirements that the system should satisfy. Hence, the desired system characteristics should also be analyzed for such implicit testing targets.

We realize that at the requirements definition time there is little, if any, knowledge of the resulting design and implementation and it is difficult to formulate exact test cases based on such abstract definitions. However, the language used to define testing objectives at this phase can be on a high level of abstraction, written in natural language, thus having far richer semantics and expressiveness. This allows writing the test objectives already when there is no insight into the resulting technical details.

After defining the testing objectives and mapping the related requirements to them, the objectives are categorized based on the non-functional characteristics they present. This allows the resolving of both conflicting and

missing objectives. It is commonly acknowledged that due to the nature of requirements documentation, a tangling of requirements cannot be avoided. The granularity of requirements, especially with industrial-scale systems, is too coarse to avoid expressing them simply enough. While test objective revision back to requirements does not solve this problem, it helps in identifying such issues.

Finally, after formulating test objectives and categorizing them into groups of non-functional characteristics, the corresponding test aspects can be formulated. The notation of aspect-oriented languages allows describing the aspect pointcuts in a rather natural manner, thus making it straightforward to translate the test objectives into corresponding test aspects. A test objective:

> "Track procedure executions and create execution time profiles."

for a non-functional *profiling* concern can be formulated as *performance profiler* aspect using an understandable expression, for instance:

> "Capture function executions and create an execution profile over the SUT during the testing process."

Without actually knowing the resulting implementation, the test aspect can be outlined using this definition, as in the code snippet presented in Listing 6.

```
1  aspect profilerAspect {
2    pointcut profile() = execution("% %::%(...)");
3    advice profile() : before() {
4        // Write function signature into log with time stamp
5        record_start(tjp->signature(), current_time);
6    };
7    advice profile() : after() {
8        // Write function signature into log with time stamp
9        record_stop(tjp->signature(), current_time);
10       // and update profiling data
11       update();
12   };
13 };
```

Listing 6: Simple profiler aspect.

Comparing conventional techniques to aspects

The proposed approach includes a systematic method for defining system-wide testing objectives for validating correctness, and resulting testware,

based on the initial requirements and desired system characteristics and utilizing them also in the unit and integration testing. In comparison to the existing ad-hoc methods, this represents a significant enhancement on the experienced testing effectiveness. Furthermore, this approach enables using the technique in systems that are difficult to test using conventional techniques.

In non-functional testing, aspects provide richer semantics to express the testing objectives as test modules. These test aspects are easier to develop in the early phases of system development, when considering the traditional incremental or the V-model, as they are not as tightly bound to the design or structure of the SUT. Furthermore, the test aspects do not require altering the original SUT in order to instrument the system with the necessary means to perform the testing activities. This should be considered a major advantage over conventional techniques, making it an attractive possibility.

4.4 Using aspects in implementing testware

AOP provides a means for presenting testing related testware implementation in a non-invasive fashion. This includes altering and extending the original code, capturing function calls and code execution, and parameter value manipulation, for instance. The following section describes how aspects can be used to implement common functional testing related items.

Altering the original code to embed test control

An AspectC++ property *slice* allows extending of the original class with additional implementation, thus allowing the adding of internal data, and methods, for test execution purposes. Hence, the slice is a form of refactoring using *program slicing* [45]. Aspect code in Listing 7 extends the SUT code by introducing a private data member for the original pedometer controller class to count the number of steps. In this example the first advice code adds corresponding functionality for initializing the counter into the original class constructor, and the second advice increases the counter after executing the callback function. Listing 8 is an example of test code for realization of the singleton pattern using the around property, which, before allowing the original code to be executed (on line 14), checks for preconditions for the singleton.

Although the slice extends the class itself, the extension is separated from the original code, which remains intact. In object-oriented languages a similar effect is achieved using inheritance: the derived test class extends the original functionality. However, this requires altering the original code to utilize the interfaces implemented by the test classes instead of the original ones.

```
aspect TestwareExtensions {

  // Extend original class declaration
  pointcut extendClass() = "PedometerCtrl";
  advice extendClass() : slice class {
    private:
      int numberOfSteps;
  };

  // Initialize counter in constructor
  advice construction(PedometerCtrl()) : after() {
    tjp->target()->numberOfSteps = 0;
  };

  // Increase step counter after callback
  pointcut increaseStep() =
      execution("void PedometerCtrl::AddStep(...)");
  advice increaseStep() : after() {
    tjp->target()->numberOfSteps++;
  };
};
```

Listing 7: Extending class using slice.

```
aspect SingletonMonitor {

  // Default constructor for this aspect
  // initializes also the instance counter
  SingletonMonitor() {
   numberOfInstances = 0;
  }

  pointcut instanceCreation() =
    execution("% PedometerCtrl::NewCtrl(...)");
  advice instanceCreation() : around() {
    if (numberOfInstances == 0)
    {
      tjp->proceed(); // proceed with the original function
      numberOfInstances++;
    } else {
      printf("Singleton pattern violation!");
      abort();
    }
  };
};
```

Listing 8: Monitoring advice for the singleton pattern.

Hence, conventional techniques do not allow full obliviousness, although in C++ namespaces can be used to perform the task to some extent.

In this example the declaration extension is statically woven to the original code at the class declaration, whereas the constructor code alteration is woven after class constructor execution. For testing purposes this allows writing test code that is dependent not only on the static structure of the code, but also on the dynamic properties. Hence, the test cases and related control can be varied at run-time while executing the tests, thus allowing the test cases to be tailored according to the intended behavior.

A typical test case in integration testing requires a variety of input data to be fed into the SUT related to the function parameters. In other words, a set of input data is used to exercise the interface with a selected input data set to verify the functionality in the case of different parameters. Aspect languages provide syntax for defining joint points to function call and execution pointcuts, which can be used to implement such integration testing functionality. Furthermore, the function parameters and return values can be manipulated. This allows writing test aspects that are woven to SUT interfaces and exercising them using a predefined set of data. Referring to the previous example, aspect code in Listing 9 defines test code for the `Client` class and exercises the measuring function with a number of different parameter values.

```
1  // Simple test code for exercising the function with different
2  // parameters in integration testing.
3
4  aspect SimpleTest {
5    int numberOfTestCases = 6;
6    int lengthCaseValue[numberOfTestCases] = {0,1,2,50,100000,-1};
7    double timeCaseValue[numberOfTestCases] = {0,1,2,60,3600,-1};
8    pointcut testCode() = call("% Client::Measure(...)");
9    advice testCode() : around() {
10     int* lengthArg = reinterpret_cast<int*>(tjp->arg(0));
11     double* timeArg = reinterpret_cast<double*>(tjp->arg(1));
12     for (int a=0;a<numberOfTestCases;a++) {
13       for (int b=0;b<numberOfTestCases;b++) {
14         *lengthArg = lengthCaseValue[a]; // Change parameters
15         *timeArg = timeCaseValue[b];
16         tjp->proceed(); // Proceed with function
17       }
18     }
19   };
20 };
```

Listing 9: Test code for the client class.

Since aspect-oriented code is not directly dependent on the original code, an interesting opportunity to write generic test code to be reused in testing a variety of different code segments arises. As an example of such generic test code, consider an example given in the aspect code snippet presented in Listing 10. The advice is woven around any function call in the SUT and exercises all of them with the predefined parameter values related to the parameter type according to the test plans. This demonstrates the possibilities enabled by the AOP languages that allow generalizing the underlying SUT code and concentrating on the concerns related to testing.

Integration testing concerns with AOP

Integration testing concerns are related to the connection between the components, that is the interfaces, and how the SUT exercises them. Exercising the code with a variety of different data allows controlling the test execution in order to achieve better test coverage. Furthermore, in addition to setting the internal state, other testing concerns include the controlled execution of tests, verifying the results, and enabling code variation to separate testware from final code.

In Section 2.1 we saw that the callback function in the example code can be accessed outside the SUT, thus questioning the system robustness. This requests tools for observation: test case developers are interested in monitoring the access to the callback function and want to capture situations when any code outside the SUT, to be precise any instances of any class other than `PedometerDrv`, calls the callback function. This testing concern can be realized using advice code presented in Listing 11 that captures any illegal calls to the callback function and reveals the object responsible for it. The `cflow` function defines a runtime scope for the join point and effectively causes the advice code to be executed only if the method `AddStep` is being executed outside of the scope of the class `PedometerCtrl`.

Corresponding testing pointcuts defining specific situations can be used to capture known issues in the SUT. An error once fixed and reappearing in the system after a number of iterations is an example of this kind of issue presenting a situation that is to be captured in testing. Although conventional techniques support regression testing, that is testing for software regression, implementing testware for observing the situation leading to the problem could be extremely difficult. The ability of aspects to access system internal data can prove beneficial when the original root cause for the error was not fixed, but due to certain circumstances appeared to be fixed. Using the code presented in Listing 11, a test case, e.g.:

> "Whenever the callback function is called outside class `PedometerDrv`,

```
aspect ModifiedTestAspect {

  pointcut functionCall() = ("% %::%(...)");
  advice functionCall : around() {
    int count = tjp->args(); // Number of arguments
    int n = 0;
    vector<void*> args;
    vector<string> argtypes;
    // Resolve parameter pointers and types
    while (n < count ) {
      void* arg = tjp->arg( n );
      string tname = tjp->argtype( n );
      args.push_back(arg); // store pointer to value
      // Call function cleanTypename, which returns a
      // single string containing the name of the argument type
      // cleaned from extra characters and the value.
      tname = cleanTypename(tname);
      argtypes.push_back(tname);
      n++:
      }
    // Test case data is available via class TestCases
    int caseCounter = 0;
    while (caseCounter < TestCases.GetNumberOfCases()) {
      // Use different input values depending on the test case.
      TestCases.SetParameters(caseCounter,args,argtypes);

      // Proceed with the original function to actually
      // run the test. It is here assumed that the original
      // method can be called multiple times in a sequence.
      tjp->proceed();

      // Write test results into log.
      TestCases.LogResults(caseCounter);
      TestCases.LogWrite( tjp->signature() );
      for (int idx == 0; idx<count; idx++)
      {
        TestCases.LogWrite( argtypes[idx] );
      }
      caseCounter++;
    }
    // Testing done, clear vectors.
    args.clear();
    argtypes.clear();
  };
};
```

Listing 10: A generic test code implemented as aspect code.

```
pointcut callback() = execution("void PedometerCtrl::AddStep()")
  && !cflow(execution("% PedometerDrv::%(...)"));
advice callback() : before() {
  const char* sig = tjp->signature();
  const char* callerSig = tjp->that()->signature();
  error_log_unauthorized_access(sig, callerSig);
  // here the callback is not finally executed as not allowed.
};
```

Listing 11: Advice code for monitoring access to the callback function.

> raise an exception and print out the name of the instance calling the function."

may be utilized in future to monitor whether the situation reappears, for instance.

Using aspect-orientation, the testware implementation can be based on the system requirements and the desired functionality. This implies basing the test development on the assumed behavior and writing the test code using descriptions that capture such behavior under evaluation. To some extent this can be achieved without seeing the original code. Consider the example code controlling the pedometer hardware, for instance. Based on the definition, the desired system functionalities are obviously:

> "The controller should be able to control the pedometer hardware via device driver and receive changes in the distance data via calls to the callback function."

Based on this information and the interfaces, the following tests can be derived and formulated as aspects:

- Monitoring code for the singleton pattern, jamming situations, and memory leaks.
- A driver stub defined as a stub aspect with join point in the driver calls for testing driver jamming situations and testing timer related functionality.
- Pedometer controller test harness defined using the controller interface definition, as well as a test harness for the client application code.
- Reliability and robustness tests for multiple accesses to interfaces and concurrent use.

In other words, all the issues raised in Section 2.5 can be covered without altering the original code by simply writing a test code to be woven into the system whenever testing the system.

Comparing conventional techniques to aspects

The discussion in the previous sections has presented issues related to implementing conventional testware using aspects. In comparison to conventional techniques, the most significant differences are in the expressiveness of the techniques and in the obliviousness.

While macros are easy and fast to write and the developers are proficient in using them, the technique still suffers from possibly exhaustive additional complexity of the code segments. With interfaces the problem is not as evident, but managing the test interfaces together with the original code requires proper test interface development and limits the possible test code implementations due to lack of access to the component code behind the interfaces.

COTS source codes and legacy code as part of the SUT implementation are problematic from the testing perspective when using macros and interfaces because of the lack of insight into the component implementation. It is difficult, if not impossible, to create testware based on possibly minimal information on the code structure. However, programming-level aspects cannot fully solve these problems either due to the limitations of the language. Since the pointcuts can be matched on certain code elements only, the possibilities of designing test cases are limited to these same elements, too.

Conventional techniques, specifically macros and interfaces, and the possible different realizations of these, are limited to the code structure and the related design, whereas aspects constitute semantics of a higher level in abstraction. It is possible to formulate testability concerns based on higher level artifacts, such as requirements, desired functionalities, objectives, and specifications, for instance. This is especially useful in the early phases of the system development, when there is not enough information to implement the testing using conventional techniques, but enough to formulate the related testing objectives and hence, the test aspects.

With current aspect-oriented languages the aspect code is woven into the target system and thus from the testing perspective the SUT remains oblivious to the aspect-oriented testware. This provides immediate separation between the SUT and testware. A stub or mock object implementation can be bound to a generalized interface function or based on the class declarations on the actual interfaces. This allows defining the stub framework without the need for white-box analysis, but still achieving similar results, as the test

cases can be formulated using system characteristics.

4.5 Summary

The expressiveness of AOP offers powerful means to capture non-functional issues for testing already in the very beginning of system development. The ability to formulate testware, independent of the resulting structures of the implementation, and not interfering with the final production version, offers a method of generalizing the testing assets. From the maintenance perspective this offers the possibility of reusing and updating testware more efficiently.

5 Evaluation

In this chapter aspects and their usage in testing is evaluated based on the case studies. First the target system and the related context is described in detail and an overview of the situation before the case studies is discussed. Discussion of the test implementation with aspects and the results from the case study follows.

5.1 Evaluation approach

Based on the results from the case studies we evaluate the aspect-oriented approach to testing software systems, how AOP can be used in implementing testing, what different tools and practices are needed when using aspects in testing compared to conventional techniques, and in what respect AOP changes the approach to testing. The evaluation is qualitative rather than quantitative, and it is performed according to a number of case studies, which have been conducted using a real-life industrial application.

In the evaluation the target is to define the basic guidelines on using aspects for testing and define the areas of testing where the AOP is applicable, rather than collecting empirical data, for instance on the impact on the test coverage. The industrial application used for the experiments evolved during the experiments and was not possible to isolate for the purposes of test coverage measurements. Hence, we believe that preparing the SUT for AOP testing influences the testing approach already in such a significant manner that collecting such test coverage data would have been erroneous and inappropriate for this research.

5.2 Context and target system

To study aspect-orientation and its use in testing, the technique was applied to an industrial software system used as part of quality verification in smartphone manufacturing. This software is used to verify that the manufactured devices are functional and assembled correctly and is commonly

called *production-testing* software. However, in order to distinguish this *testing* from *software testing*, production-testing software is hereafter referred to as *production-verification* software.

The context of the case studies consisted of a number of Nokia smartphones running Symbian operating system [46, 47], belonging to the product family [48] of S60 [49] devices. The development tool chain involved target hardware architecture specific compilers, RealView Compiler Tools (RVCT) for ARM processors starting from Symbian release 9.1 [50] and a variant of GCC for release 8.1 [51] and earlier. Furthermore, this software was used in device manufacturing and operated based on the inputs of the production system, test instruments, and the resulting outputs were evaluated by the manufacturing devices. For practical and business reasons the experiments were conducted on a copy of SUT product-line instead of the production version of the SUT. Hence, aspects were not used in the production version of the SUT and are thus not included in the device after the device is delivered from the factory.

A simplified illustration of the production-verification is presented in Figure 15, where the production-verification software is marked as Software Under Test (SUT). The SUT receives control from the *tester* and, using terminal software, controls the terminal Hardware Under Test (HUT). Based on the resulting data from the SUT and test instruments, the tester decides whether the verification is passed or not. Hence, the tester exercises the terminal HW to produce outputs using certain input control and data.

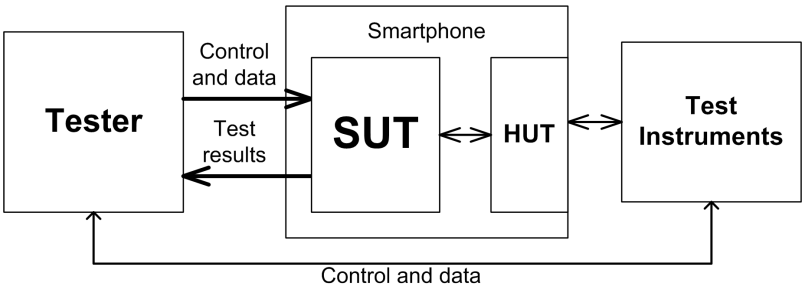

Figure 15: Basic setup in smartphone manufacturing verification. In this context the tester was a test application running on a factory PC, including test cases and test reporting.

The SUT architecture follows principles of a software product-line [48, pages 5-15] and represents a relatively small-sized industrial system of close to 200kLOC of C++ source code, resulting in 100kB of target binary. The SUT is composed of three subsystems: a client interface, a server, and proce-

dures. The SUT offers services according to received messages. These service requests are processed by the server, which invokes the necessary procedures based on them. The procedures implement different verification-related functionality and interact with the terminal software, device drivers and other system servers to control the related hardware. A simplified diagram of the original SUT design is illustrated in Figure 16.

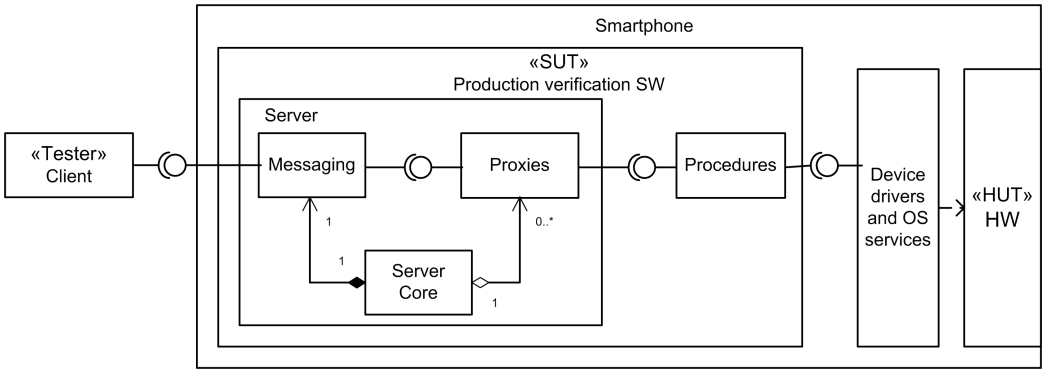

Figure 16: Diagram illustrating the design of the verification SW.

Due to the role of the SUT in manufacturing process, two types of testing can be performed: either verification of the manufacturing process or software testing of the SUT. While starting the research for this thesis the testing of the SUT was performed by exercising the client interface with a predefined set of input data and observing whether the SUT accomplished the expectations for the desired behavior. Testing thus targeted in verifying the SUT is able to respond to service requests and perform related verification functionalities using the terminal hardware. The SUT testing effort was divided into three phases:

- Design and code inspections [52]. While this technique was first introduced more than thirty years ago for detecting defects in software, it is still a valuable tool [53].
- Running static analysis tools on the code. With static analysis tools we refer to popular tools like PC-Lint [54], for instance.
- Exercising the system tests. System testing consisted of a combination of black-box unit testing and integration testing, with emphasis placed on the latter.

As a simplified example of such a test case, consider a camera test: the tester requests the SUT to take a photo using terminal camera hardware. If

it succeeds in taking the photo, the test is considered to be passed. Obviously, such testing is superficial: the SUT is only tested for successful cases. However, it should be noted that the SUT is a small part of the overall verification software used in actual device manufacturing and is only responsible for a limited subset of verification activities. Considering the SUT objectives and the role in the overall verification process, the described strategy is here considered sufficient.

5.3 Implementing hardware testing using aspects

Based on assessment of the system (presented in publication [I]), the system to a large extent fulfills the criteria of meeting a product-line approach. By partially redesigning the system, the principles of a product-line were exercised in the system development by concentrating the variation into the smallest possible units and promoting large-scale reuse. This suggested a fruitful characteristic for aspect-oriented treatment, since the varying parts in the design were the procedures implementing the verification functionalities. Hence, the first approach was to evaluate the use of aspect-orientation in implementing such functionalities using aspects. This approach is evaluated in the following and discussed in detail in publication [II].

As already discussed the SUT was designed for verifying smartphone manufacturing, and therefore, concerns related to hardware variants should be covered by the design of related software components. In order to manage the variability in the product line, the components were composed using a layered approach, where basic and generic test layers operate on top of more hardware and operating system specific layers. These generic test components further utilize specialized components to cover the particular hardware and driver versions. However, the implementation included a heavy overhead due to code repetition and extra code embodied in all the test components. This produced tangled implementations, especially with the test components that had been used for a longer time and had faced a number of iterations due to hardware evolution.

The problem the aspects were used to solve was two-fold: first, to provide an easy method to expand existing procedures to support one-shot deviations from the baseline; and second, to allow the injecting of common system functionalities into existing procedures. In other words, the proposed solution was to instrument either product level specifications or common general-purpose functionalities into the common assets using aspects. The solution was restricted to the use of aspects to testware implementations that were supposed to be either left out from the final products or dedicated to only one specific product.

Based on the experiments, the most significant observations were on the pragmatic level. Problems with the tool chain and the AspectC++ compiler, poor parser performance (AspectC++ uses PUMA [55] to parse source code) and inability to manipulate the original code without modifications caused difficult problems for industrial adoption. This was a major drawback, especially when the promises of obliviousness were not fulfilled due to the need to add workarounds into the original source code for the parser and aspect weaver in order to succeed with the compilation process. Therefore, the original source had to be parametrized according to whether it is the final C++ compiler parsing the code or AspectC++ compiler.

After the principal technical difficulties were solved, the produced aspect code resulted in a significant size overhead. Although the performance overhead is minimal (aspect code has the execution performance of native C++ code), this is a drawback in resource critical systems, such as embedded systems, that have strict memory footprint restrictions. However, careful selection of the assets to implement as aspects increases the system's adaptability with just moderate effort. The observed size overhead was due to the excessive nature of the injected functionality (injecting a tracing aspect, for instance) and is mostly avoided when incorporating one-time specializations. Such issues seldom address cross-cutting characteristics, though. Another possibility is to use AOP to simulate hardware when testing the software of embedded systems, as presented in [56].

5.4 From hardware to software testing

Based on these experiments, the applicability of aspect-orientation in this context was questioned, since the existing framework offered sound support for adapting new functionalities, and the gained benefits seemed to remain small. The ability to inject new behaviors into the existing system was still attractive, and based on the experiments we concluded that instead of the approach of implementing verification-related procedure extensions to the system, it could be better to utilize the technique in testing the system itself. The refined testing context, now concentrating on software testing is illustrated in Figure 17. This approach is discussed in the following.

Using aspect-orientation to implement integration testing

This section is based on the included publication [III], which was inspired by the experiences gained during the previous publication [II]. The results from previous experiments indicated that in an embedded setting the aspects demand an environment with no strict memory footprint restrictions or con-

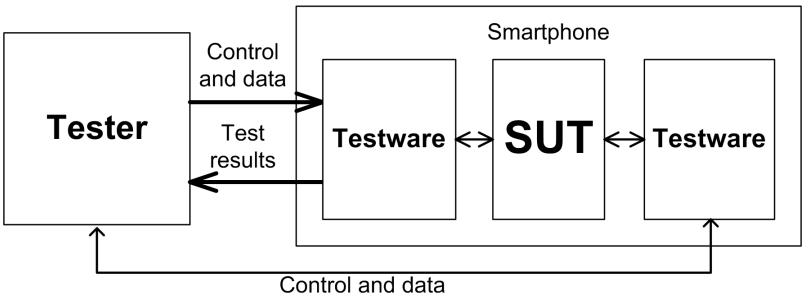

Figure 17: Refined setup where testware is added to control the SUT.

centrating on simple concerns. Based on these observations and the original testing strategy, it was concluded that aspect orientation could be useful in implementing a testing concern, integration testing, into the system, thus utilizing the non-invasive nature of AOP. In contrast to the previous publications [I] and [II], the focus is from this point onwards on software testing, not in manufacturing verification. The experiment is further discussed in publication [III].

In this case the aspects were proposed to be used in checking invariants, capturing and replaying inputs, and generating stubs. Furthermore, a *test harness* (a unit testing technique to isolate the SUT for testing) was to be implemented using an aspect-oriented approach. For these purposes, test stubs were formulated to isolate the SUT from the surrounding components, additional test aspects were formulated to implement system-wide testing concerns, and a core implementation for the test control was presented. In terms of the implementation, the test harness was composed of smaller test harnesses for each of the procedures, aspects for stub implementations, and aspects dedicated for generic test cases.

Commercially available testing tools (EUnit [57], LDRA tool suite [58], for instance) offer sound support for unit and integration testing and creating stubs. In other words, the rationale for writing stubs using aspects is questionable, as the proper tools make it more or less automatic. However, these tools are unable to capture intrinsic functionalities for testing. Although creating stubs using aspects is possible, and relatively straightforward, the benefits of doing so are minimal if tool support is already established in the context. Based on the these experiments, the true potential lies in regression testing, as formulating aspects for preventing errors from reappearing, and measuring system resources is simply and effectively achieved using aspects. Injecting such test aspects into the system prior to running the test cases clearly improves the effectiveness of the testing by providing more insight

into the system in otherwise black-box testing.

Hence, as a conclusion from these experiments the aspects should be used to complement the integration, or unit, testing of such systems. If no tool support is available for the system, or the tools fail to generate the required testware, aspects propose a method to implement such tools in a non-invasive manner. The greatest opportunity is the ability to capture parts of an existing system and to isolate them for testing. In Java environments a similar approach to inject unit testing on-the-run is a widely used application of AOP. However, in an embedded setting this requires significant improvements on the tools and techniques used to implement aspect-orientation.

It was also concluded that generating the testing aspects from higher-level descriptions, specifications or requirements, for instance, could better demonstrate the potential of aspects. This issue is further studied in the following.

Deriving test aspects from requirements

Based on the earlier discussion of functional testing, in our next case study, explained in detail in publication [IV], we focused on non-functional testing in order to study the effectiveness of aspect-oriented techniques in formulating aspects based on requirements. In this case we evaluated the possibility to derive non-functional testing assets from the requirements and initial expected system characteristics, and to formulate corresponding test aspects. Based on our earlier experiences of aspects, we concluded that non-functional concerns such as performance, reliability, profiling, monitoring, and robustness, for instance, have the potential to be covered using aspects.

The evaluation began with close to 150 requirements, but in the end we found six very generic system characteristics and 16 basic requirements based on them. These requirements were evaluated for cross-cutting properties and tangling presentations. Furthermore, the system characteristics revealed implicit non-functional characteristics, which were included in the list of requirements. With only 16 requirements we identified already four cases where the requirements themselves tangled. Such requirements tangling is a common problem due to the granularity of requirement descriptions, which are too ambiguous and imprecise to avoid mixing requirements. However, we considered this unavoidable and resolving the issue is beyond the scope of the case study.

As a result of the requirements analysis, we formulated seven test objectives, "*Measure time consumed on serving requests.*", for instance. These objectives were categorized according to the non-functional concern they cover: performance, profiling, robustness, reliability, or coverage. Using the cate-

gorization, the objectives were formulated as test aspects, resulting in the following five test aspects:

- *Memory Aspect* for supervising memory operations.
- *Performance Profiler* for profiling SUT execution.
- *Robustness Aspect* for generating jams on the services the SUT relies on.
- *Reliability Aspect* for collecting SUT state information.
- *Coverage Aspect* for monitoring SUT execution during test execution.

While other aspects cover single test objective, the *Performance Profiler* serves the interests of three test objectives. We believe this ingenious combination of three objectives into one aspect was a result of manually manipulating the objectives, and could not be the result if derived automatically or without the knowledge of the SUT characteristics that we had. However, the semantics of aspect descriptions allows the formulating of such generic test aspects to cover a number of objectives presented by a single concern. Based on our experiences, deriving test aspects from requirements benefits from the AOP characteristics related to managing cross-cutting issues.

5.5 Tool support

The success and practical applicability of any new technique depends on tool support. For this reason, a testing tool for embedded systems, such as the SUT in the case studies, was also studied.

Scripting tool

The very first experiments were conducted using hand-written aspect code that was woven into the target system without any specific tool support. This is very cumbersome and requires a considerable amount of time and experience from the test engineer, and thus a better solution is required prior to industrial adoption.

During the experiments the following methods were used:

- Reading the source code and interface documentation to identify call and execution join points (interfaces present all the possible targets for function calls) and writing advice code for pointcuts based on the experience from the designers.

- Designers together with test engineers list classes and their methods to find join points and define pointcuts according to specifications.
- Test case designers and test engineers use sequence diagrams to identify join points and pointcuts.
- Aspect code is formulated using natural language based on a list of requirements, which were eventually translated into corresponding aspect code by a designer who was an expert in AOP.

Based on the experiences gathered from the personnel involved in the studies a preliminary tool support was planned. The first approach was to maintain a class diagram related to the source codes of the SUT and to use a visualization of the SUT design model to formulate the aspects based on the model. Compared to the lists of classes and their methods, this suggested a tool that would make it easier for the test engineer to see the relationships and find the join points for the test aspects. In other words, instead of requiring the ability to actually read the code, the tool would offer an easily understandable view of the system composition.

However, the problem with such tools is scalability. While diagrams used in example applications are simple to understand and effectively examined, complex and large systems produce complex and large diagrams, which result in difficult models for test engineers to examine. Due to some simple desk exercises and their outcome, and the observed practical size and complexity problems, the visualization tool was never implemented. We concluded that in addition to providing the anticipated benefits (and drawbacks), the tool would be both a very large and complex project to implement and complicated to use. For these reasons, only a prototype of the tool was implemented, where a command line script could be used for instrumenting subsystems with given aspects.

The use of the tool was based on the following procedure: First, aspects are formulated using a natural language. Second, join points are identified from lists of methods extracted from the source codes. Third, final aspect code is formulated using the join points and the corresponding expectations expressed in the aspect definitions. Finally, the tool manipulates the original source codes and weaves the aspect code to sources using external tools, the aspect weaver, and after compilation results in executable target binary. The critical phases regarding this procedure are identifying the proper aspects (that is, writing the desired functionality as an aspect) and identifying the join points.

Based on the experiences from using the tool, we concluded that it is possible to formulate generic test objectives that can be reused in different

systems, and it is essential to understand what the test cases are supposed to accomplish in order to formulate the correct test aspects. Hence, we concluded that a full model of the system is not necessary, but only the key properties and components of the system are needed to create test cases to cover these interactions. The rationale is that it is sufficient for the test engineer to be able to draw a test sequence diagram of the test case that is to be created. We concluded that a tool for creating the resulting aspect code based on such diagram would increase the efficiency of our simple tool based on scripts.

Creating test aspects using LSC-to-aspect compiler

In publication [VI] the tool support was experimented with using modeling and visualization. In contrast to the previous tool approaches, this publication studied the use of a Live Sequence Chart (LSC) [59] to visualize testing scenarios and generated related testware based on the model. In this approach, the test designer uses a high-level sequence chart to model the testing scenario instead of implementing testware, which is automatically generated by the tool as test aspects. These test aspects are woven into the SUT for test execution. For these experiments we created test scenarios that modeled 10 different test cases, starting from very simple ones and representing typical test cases related to the SUT.

Based on the experiments, deriving the LSC is complicated if the SUT has no proper model available. The modeling tool requires identifying the classes to illustrate the interactions as well as the related methods. This calls for a model of the system and identifying the relevant elements. Furthermore, cross-cutting concerns do not manifest themselves as single classes or methods, which makes it difficult to model such issues using scenarios. Although the AOP properties for modularizing cross-cutting concerns cannot thus be fully utilized, the tool effectively uses the non-invasive nature of AOP. The ability to generate the testware automatically makes the SUT oblivious to the testware.

Based on our experiments, considering the testing objectives as test scenarios is very different from the conventional approach and requires skilled personnel. Prior to wide deployment this is a clear disadvantage of this approach. However, a skilled designer can effectively describe testing scenarios using the sequence charts and due to the compiler easily create testware for the SUT using the model only. Furthermore, examples of the most common test cases can be created as LSC templates to help designers to learn required modeling skills. Although the tool provides modeling elements for expressing environmental variables (the user or the system, for instance), the modeling

semantics somewhat limits the possible test cases that can be expressed using the LSCs. However, since loops and branching are supported in the scenarios, the tool can still be used in most of the cases based on our experiments with the original system.

5.6 Results from the case studies

This section combines the results from the case studies and highlights the key ones. The results are also discussed in detail in the included publications.

Effects on the testing efficiency and code coverage

With all the existing tools and techniques available, the efficiency of any proposed approach is an important question: What benefits does adopting yet another testing technique provide, and does the technique allow deeper analysis by means of better test cases? Furthermore, initially the testing targets verifying that any software system satisfies all the expectations on the behavior set for it. Testing also typically involves defining qualitative characteristics the system should represent. In comparison to conventional techniques to implement software testing, does the approach improve testing of the system?

Based on our experiments (described in detail in publications [III], [IV], and [V][3]) aspects are an effective and non-invasive technique in implementing testware. We were able to increase the code coverage using aspects, in comparison with the starting situation and conventional techniques. However, it could be argued that better code coverage was obtained due to reinvented test cases and thus as a result of rethinking the overall testing strategy. Nevertheless, the purpose of introducing aspect-orientation into the testing context aims at these very same results: rethinking the testing in general in order to formulate test cases that better cover the testing concerns, and thus result in better overall testing of the system.

We believe the increased code coverage was also achieved because of the ability to formulate test aspects that implemented new test cases and thus covered issues previously disregarded. Comparing the history of the SUT and the testing experiences, this was anticipated: testing of such industrial systems suffers from the pressures of project schedules and limitations on the available investments. Although the testing had known deficiencies, they were on an acceptable level. We believe this is a common approach in industrial systems. Whenever there is at least a seemingly satisfactory amount of

[3]Notice a typing error in the abstract of publication [V]: instead of the word *invasive*, it should read *non-invasive*.

testing, it is considered sufficient if the project schedules indicate so. While adopting AOP does not provide thorough testing of systems with small effort, it concentrates the effort into what we believe are the correct issues. The earlier the more important design concerns are taken into consideration, the better are the results that both testing and system design can achieve.

Test cases implemented using aspects do not present better test cases than those implemented using conventional techniques, but instead, the concerns to be captured using aspects are different. We argue that conventional techniques are better than aspects in general when implementing conventional testing, such as unit testing, for instance. This is due to the established tool support and experiences in exercising them. AOP is useful when better expressiveness is required and when the test objectives are not too dependent on the code structure. When AOP tool support matures, the technique is easier to adopt and provides powerful means to increase both the effectiveness of the testing, either system or unit testing, and the quality of such testing. However, in the context of embedded systems, AOP techniques suffer from both performance problems (the weavers are inefficient) and size constraints, which limit the applicability of such techniques, at least in industrial applications.

Effects on the testware maintenance

The system under test is anticipated to be oblivious to the testing. Therefore, the related testware requires no changes to the original implementation. In the proposed approach and in comparison to conventional mechanisms, the actual perceived level of obliviousness is the key separating factor. With conventional techniques, separating testware from production code is a major issue, and aspect-orientation allows a non-invasive technique. Although the original system is not always fully oblivious to aspect-oriented techniques, our experiences indicate a clear improvement in comparison to conventional techniques. The ability to manipulate existing behaviors without modifying the original source codes provides a powerful means to implement testware for legacy and out-sourced code.

The aspect-oriented testware and the possible binding between test concerns and the SUT relies on the granularity of the AOP languages. With AOP languages, such as AspectJ and AspectC++, the pointcuts are matched on certain predefined code elements: function, attribute, type, or variable, for instance, which limits their possible test implementations. This could be too coarse for certain testing tasks, especially when the test objectives are very abstract. Although test aspects could be formulated using the AOP language expressions, this binds the resulting aspect code to the original source

code. This could also be a limiting factor when considering the maintenance and reuse of such test code. Pointcut expressions can be limited to focusing on the specific testing objective, thus making them specific to certain SUT. Furthermore, any changes in the SUT design or implementation must be reflected by corresponding changes in the aspect code, in order to keep the testware updated.

Finally, AOP obliviousness is challenged by current AspectC++ tool support for C++. This is due to the fact that the associated C++ parser and aspect weaver are not always capable of manipulating the code properly, resulting in failures. Although problematic, these practical issues can be solved by further developing the parser and the weaver. Currently such problems can be avoided by altering the original source codes correspondingly. Inserting directives for pre-compilers to either exclude or include the relevant code segments allows the aspect weaver to bypass such problematic structures. However, this questions the aspect-oriented approach of keeping the original code intact.

Effects on the overall testability of the system

Effective and reliable testing is expected to be performed explicitly and obliviously. Implementing an approach that satisfies both these qualities would thus not affect the implementation or design. However, being able to modularize additional elements as testable units increases visibility to the system at all levels. Hence, does the approach actually increase the overall testability of the system?

Based on the experiments we argue that defining the testing aspects based on requirements allows the capturing of more generic and cross-cutting concerns for testing than using conventional techniques. Furthermore, these testing concerns permits testing concerns to be satisfied by the system architecture and design, thus promoting better testability. This is achieved because of better understanding of the system-level design issues already at the requirements analysis phase.

In the presented approach, testability is considered as a design concern and explicitly addressed throughout the development process. Based on our pragmatic experiences on implementing the testability concerns using aspects, we argue that AOP increases the overal testability of the system by enabling a non-invasive technique to inject the implementation of the testability concern into the system. This allows the test designers to design generic and system-wide implementations for the testware, thus enabling better reuse and variation of the testware. Hence, the system testability is enhanced by developing better testware and by allowing better control over cross-cutting

concerns.

Although aspects allow the development of non-invasive testware and formulating of test cases based on more generic objectives, we consider the system design must explicitly support testability. More liberal technical limitations in implementing the testware and better promotion of testability using aspects still require support from the system design. Nevertheless, based on our experiences, we believe using AOP in the test development makes it easier to adjust the testability needs once the system design has been completed.

5.7 Summary

In the case studies that we have conducted the use of aspect-oriented techniques was evaluated in a number of different uses. First, aspects were used to implement test procedures for production-testing. In this use aspects benefit from their non-invasive nature and were considered useful in implementing one-shot deviations from the mainstream and for presenting generic functionalities. Aspects were then used for implementing traditional unit and integration testing. In this use, aspects present an oblivious approach to implement the relevant testware. Furthermore, the aspect-oriented approach was compared to conventional techniques, and although there is no clear demand for replacing other techniques with aspects, it provides an additional means to enhance the overall testability of the system. This is due to the ability to modularize cross-cutting testability concerns. Finally, testability concerns can be formulated as testing aspects based on requirements, thus allowing test development to be started based on less information on the resulting implementation. This is achieved due to the AOP properties for capturing cross-cutting concerns and implementing testware in a non-invasive manner.

6 Related Research

Although aspect-oriented approaches have been widely studied, incorporating them in testing has been the subject of a relatively small number of researchers. This chapter discusses approaches closest to the approach presented in this thesis.

6.1 Testing cross-cutting concerns

Similar approaches to using system-wide conceptual features as the starting point for test development include utilizing concerns and scenarios. Souter et al. [60] propose testing using concerns to gain savings in test suite execution and increased coverage information. In this approach [60] recommend selecting tests associated with a particular maintenance task and instrumenting the regions of code relevant in respect of the concerns of interest. They use a concern visualization and analysis tool to represent the concern relationships to the source code. The resulting concern graph is used to select the code segments to instrument in relation to the concerns to be covered, priorization, and selection of tests. Another approach starting from expectations is presented by Gregoriades and Sutcliffe [61] and uses scenarios for verifying non-functional requirements. In their approach they use Bayesian Belief Nets as a reasoning mechanism for requirements validation. Using a modeling language to model the system and use case narrations to describe behaviors the approach is intended for complementing the model-checking tools. However, neither of these approaches presents an implementation level approach that could be utilized in implementing testing for the SUT.

Kiviluoma et al. [62] present an approach of automatically generating monitoring concerns based on behavioral profiles using UML. In their approach a CASE tool is used to specify behavioral rules at the level of architectural descriptions and the tool generates AspectJ aspects based on these models. The target is to provide automatic generation of monitoring related code for observing violations of certain design concerns, for instance whether the API is used in correct manner or not. Their approach resembles our

approach of generating test aspects based on LCSs. However, while our emphasis is on non-functional testing and testing throughout the testing levels, the approach presented by Kiviluoma et al. target system and acceptance testing with emphasis on monitoring of behaviors. The approach of generating test code automatically from a behavioral model is very close to our approach presented in publication [VI].

6.2 Testing using AOP languages

A more pragmatic approach to testing is presented by Feng et al. [63], who propose generic non-functional unit test cases to be written using the product line approach and using aspects to implement them. In this approach, the test cases are written using Extensible Markup Language XML and related pointcut, and join point information is gathered manually from specifications or using a tool from component source codes. The test case definitions contain reusable core sections and sections for variations, thus allowing the same test cases to be used in a variety of product line instantiations. In more detail, abstract test cases are used as a basis for representing test cases and concretized manually as test case aspects according to the target system details. Their approach is limited to the unit testing level and, compared to the approach presented in this thesis, does not allow testing for system-wide issues.

Another attempt to create more reusable test cases and automating the testware creation is presented by Benz [64]. He presents an approach to automatically instantiate test cases defined as UML state charts and generated as test aspects. In this approach an abstract test case is derived from a behavioral model and transformed into a test script during instantiation. In this approach the testing concerns are regarded as aspects that can be woven to different test cases and executed as part of the test script. The test aspects are definitions written in an aspect-oriented language, AspectT, which is a modeling language based on the Ecore [65]. Hence, these aspects are not programming level aspects, as in the case with AspectJ and AspectC++, and require a mechanism to execute the generated test scripts. This approach thus relies on an existing testing framework including the related testware, whereas our approach also generates the related testware.

The expressiveness of AOP languages has been addressed as a clear benefit in implementing testing by a number of authors. Examples of such applications are presented by Navarro et al. [66], who propose using aspect-based high-level program abstractions for debugging and testing of distributed middleware systems, and Coelho et al. [67], using aspects to monitor and control the execution of agents in multi-agent systems during testing. Both of these

approaches address the ability to describe system-wide testing concerns in a more declarative manner, which is in line with the benefits of the approach presented in this thesis. However, the approach described in this thesis raises the testing level and addresses testing concerns as system design issues, instead of just a technique to write more efficient test cases.

For testing purposes, aspects have also been used to extend conventional testing techniques. Cechich et al. [68] use aspects for packaging metadata to components for black-box testing purposes. In this approach, the aspect metadata documents the test cases related to the component and is used to resolve applicable test cases and test data. This approach aims at resolving the test case completeness by analysing the scope of test cases in contrast to the components' metadata. Furthermore, van Deursen et al. [69] have proposed generic aspect-oriented refactorings of the original system to address cross-cutting issues for testing. Their approach proposes refactoring the test suite as well as the original application using AspectJ.

6.3 Testing aspect-oriented software

In addition to using AOP for testing, also testing aspect-oriented software is a subject of research interest [70, 71, 72]. Using aspect-oriented techniques to implement the system hardens the verification task by introducing new issues to cover. The aspect interaction is not visible by investigating the original object-oriented implementation, making it more difficult to verify the correct behavior. Based on our experiences, this is a real problem closely related to obliviousness: It is possible to create test aspects that are never realized because of incorrectly formulated pointcuts or join points. In order to verify proper aspect instantiation, Massicotte et al. [73] propose collaboration diagrams for resolving the aspect integration to the original system. In their approach, the collaboration diagrams are incrementally compared against the specification, while applying aspects to the system.

Another approach targeted in testing aspects is presented by Xu et al. [74], which uses state models for test generation in testing aspect-oriented programs, where aspects are used to group separate classes into a larger aggregate. In this approach the system is presented using a state model describing the classes and state predicates. This information is used to present an aspectual state model of the system. The model is transformed to a transition tree that is used for test generation. First, abstract tests are generated based on the model, followed by concrete test specifications, and finally the integrated class is combined with base class state to transform advice models to form composite states. Hence, the integration aspects are tested using the base classes state model.

6.4 Industrial adoption

Although aspects have been mostly of academic interest, aspect-oriented programming has also been successfully adopted in industrial projects as reported by Hohenstein et al. [75], for instance. The main obstacles in large-scale adoption of AOP techniques in the industrial context reported in [75] are in line with the obstacles presented in this thesis: The required changes in processes and tool chains are not easily tolerated due to tight schedules and attitudes. Furthermore, not all developers are aware of all the possible applications of AOP and the technique is limited to patching and debugging only. The approach presented in this thesis promotes aspect-orientation as a part of the overall development process instead of a tool for quick fixes and workarounds. Hence we believe that the aspect-oriented paradigm can be considered an equally valid development approach to traditional ones.

7 Conclusions

This thesis introduced an aspect-oriented approach to testing. It was shown that aspects can be used in implementing testware for conventional testing for unit, integration, and system testing purposes. We also discussed how aspects can be used to present system-wide issues. It was further shown that descriptive languages allow richer semantics for formulating the related test objectives and basing them on more abstract elements than the program structure, for instance requirements. Using LSCs it is possible to draw testing scenarios that represent test objectives in model diagrams and to generate the corresponding aspect code from the model.

7.1 Research questions revisited

The research questions presented in Chapter 1 are revisited in the following.

1. *How could aspect-oriented programming be utilized in production verification of a product line of smartphones?*

   Aspect-oriented techniques provide a means for injecting testware into already existing systems, thus allowing the altering of their functionality. This can be used to introduce changes, patches, and testware into such software without the need for variants in the product-line. This allows a convenient means for introducing one-shot deviations that are too expensive to adapt in the main baseline.

2. *What are the benefits of using an aspect-oriented approach for testing over conventional techniques? Furthermore, is there a systematic approach for identifying the testing concerns and formulating them as aspects?*

   Conventional techniques are very context sensitive and developing generic testware is difficult due to the limitations in the implementation level. Furthermore, conventional techniques produce test-

ware tangling with the original implementation and scattered over the system. AOP allows the modularizing of these testing concerns into modules and the writing of more reusable and generic testware. Separating the SUT and testware is easier due to the non-invasive nature of the technique. However, although AOP presents powerful techniques of implementing testware, the current tool support for conventional testing is strong in unit testing and static analysis. Therefore, there is no need to replace old tools with AOP-based tools, except when testing for non-functional characteristics where aspects appears to be more expressive. Successfully adopting the true potential of AOP in testing requires a systematic approach for identifying the testing concerns. In our approach, such testing concerns are addressed starting from the initial design phases and considered as a design artifact throughout the development process. However, introduction of a pragmatic and systematic method for identifying the testing aspects is still being awaited.

3. *What kind of methods and techniques are required by an aspect-oriented approach to testing?*

   Since AOP targets cross-cutting concerns, the approach is at its strongest in capturing such issues for evaluation. Hence, the testing objectives are more important in test case planning than before. This calls for changes in the development process that addresses testability, starting at the very beginning of the development process. Requirements analysis should yield test objective descriptions that are further evaluated at the architecture, design, and finally, implementation phases. Furthermore, adopting AOP techniques calls for tool chain changes. Moreover, AOP can be utilized as an implementation level approach, thus targeting the same issues as conventional approaches. However, thinking of testability as an equally important concern to any other design concern allows the taking of testability into consideration in all development phases. This presents new issues that can be covered in testing using AOP.

4. *How could aspects be used to increase software testability?*

   The non-invasiveness of AOP techniques allows the instrumenting of the original code with implementation of the testability concern, which allows enhancements for the overall testability of the

system. However, using aspects does not directly lead to increased testability, but allows better means to address testability in the system development and implementation.

Although the set of questions and answers is not comprehensive, it shows that aspects can be used to enhance the testing of software systems. The experiments were performed in the domain of smartphones that includes dimensions from embedded systems and characteristics generic to software systems. Hence, the results can be generalized to be applicable in a variety of different problem domains.

7.5 Summary

This thesis has presented an approach to enhancing the testing of software systems of smartphones using the cross-cutting and oblivious characteristics of aspect-oriented programming. During the research the focus has progressed from implementation details towards the non-functional characteristics presented by requirements. The included publications discuss the using of aspects in a variety of different testing purposes from testing the hardware to acceptance testing. It has been shown that aspects can be used to implementing testing in a conventional manner, while it also offers a technique for a somewhat different approach.

Using descriptive languages for test implementation allows the writing of testware that better addresses system-wide testing concerns, regardless of the testing phase. Aspect-oriented programming allows the modularizing of such testability concerns into units that can be used to implement related testing. The testing objectives are thus better in addressing testing concerns that reflect stakeholder expectations against the quality of the system. Furthermore, aspects can be used to implement testware, and the SUT remains oblivious to this testware. This non-invasive nature makes aspects a powerful technique to implement generic testware more applicable to in-house systems as well as outsourced and legacy systems.

Using an aspect-oriented approach in testing requires changes in development processes and tools. Formulating proper testing concerns and objectives and implementing related testware based on requirements requires skill and experience. In order to gain insight on the relevant issues necessary for these definitions, a set of proper guidelines is required. A pragmatic and systematic guideline and fine-tuning of the aspect development process is important prior to industrial adoption.

References

[1] Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors. *Aspect-Oriented Software Development.* Addison-Wesley, Boston, 2005.

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

[3] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *11th European Conference Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, June 1997.

[4] Rick D. Craig and Stefan P. Jaskiel. *Systematic Software Testing.* Artech House, Norwood, MA, USA, 2002.

[5] R. G. de Vries. Towards formal test purposes. In G. J. Tretmans and H. Brinksma, editors, *Formal Approaches to Testing of Software 2001 (FATES'01), Aarhus, Denmark*, volume NS-01-4 of *BRICS Notes Series*, pages 61–76, Aarhus, Denmark, August 2001.

[6] IEEE. IEEE standard glossary of software engineering, IEEE-Std-610.12-1990, September 1990.

[7] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton Jr. N degrees of separation: Multi-dimensional separation of concerns. In D. Garlan, editor, *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pages 107–119, Los Angeles, CA, USA, May 1999. ACM Press.

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

[9] IEEE. IEEE standard for software test documentation, IEEE-Std-829-1998, December 1998.

[10] Paul Rook. Controlling software projects. *Software Engineering Journal*, 1(1):7–16, 1986.

[11] Barry Boehm. A spiral model of software development and enhancement. *SIGSOFT Software Engineering Notes*, 11(4):14–24, 1986.

[12] Bor-Yuan Tsai, Simon Stobart, Norman Parrington, and Barrie Thompson. Iterative design and testing within the software development life cycle. *Software Quality Control*, 6(4):295–310, 1997.

[13] Robert V. Binder. Design for testability in object-oriented systems. *Communications of the ACM*, 37(9):87–101, 1994.

[14] Nancy S. Eickelmann and Debra J. Richardson. What makes one software architecture more testable than another? In *Joint proceedings of the Second International Software Architecture Workshop (ISAW-2) and International Workshop on Multiple Perspectives in Software Development (Viewpoints'96)*, pages 65–67, 1996.

[15] Benoit Baudry, Yves Le Traon, Gerson Sunyé, and Jean-Marc Jézéquel. Measuring and improving design patterns testability. In *Proceedings of the 9th IEEE International Software Metrics Symposium (METRICS'03)*, Sydney, Australia, September 2003. IEEE Computer Society.

[16] Ronny Kolb and Dirk Muthig. Making testing product lines more efficient by improving the testability of product line architectures. In *Proceedings of the ISSTA 2006 Workshop on Role of Software Architecture for Testing and Analysis (ROSATEA'06)*, pages 22–27, New York, NY, USA, 2006. ACM.

[17] Bret Pettichord. Design for testability. Pacific Northwest Software Quality Conference (PNSQC'02), October 2002. Portland, Oregon, USA.

[18] Robert V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[19] Dave Thomas and Andy Hunt. Mock objects. *IEEE Software*, 19(3):22–24, 2002.

[20] Tim Mackinnon, Steve Freeman, and Philip Craig. Endo-testing: unit testing with mock objects. In *Extreme programming examined*, pages 287–301. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[21] Steve Freeman, Tim Mackinnon, Nat Pryce, and Joe Walnes. Mock roles, objects. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'04)*, pages 236–246, New York, NY, USA, 2004. ACM.

[22] Kent Beck. *Test Driven Development: By Example.* Addison-Wesley Longman, 2002.

[23] David Janzen and Hossein Saiedian. Test-driven development: Concepts, taxonomy, and future direction. *Computer*, 38(9):43–50, 2005.

[24] Siobhán Clarke, William Harrison, Harold Ossher, and Peri Tarr. Subject-oriented design: towards improved alignment of requirements, design, and code. *ACM SIGPLAN Notices*, 34(10):325–339, October 1999.

[25] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[26] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *[1]*, pages 21–35.

[27] Lodewijk Bergmans and Mehmet Aksit. Composing crosscutting concerns using composition filters. *Communications of the ACM*, 44(10):51–57, 2001.

[28] AspectJ. AspectJ WWW site. At URL `http://www.eclipse.org/aspectj/`.

[29] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: An aspect-oriented extension to C++. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, Sydney, Australia, February 2002.

[30] Object Management Group (OMG). Unified Modeling Language (UML), at `http://www.uml.org/`.

[31] Albunni N. and Petridis M. Using UML for modelling cross-cutting concerns in aspect oriented software engineering. In *3rd International Conference on Information and Communication Technologies: From Theory to Applications (ICTTA'08).*, pages 1–6, 2008.

[32] OMG document ad/99 04-07. White paper on the profile mechanism, 1999.

[33] Omar Aldawud, Tzilla Elrad, and Atef Bader. UML profile for aspect-oriented software development. In *3rd International Workshop on Aspect-Oriented Modeling with UML*, 2003.

[34] Mika Katara and Shmuel Katz. A concern architecture view for aspect-oriented software design. *Software and Systems Modeling*, 6(3):247–265, 2007.

[35] Dominik Stein, Stefan Hanenberg, and Rainer Unland. A UML-based aspect-oriented design notation for AspectJ. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD'02)*, pages 106–112, New York, NY, USA, 2002. ACM.

[36] Omar Mohammed Aldawud. *Aspect-oriented modeling with UML*. PhD thesis, Illinois Institute of Technology, Chicago, IL, USA, 2002.

[37] Shin Nakajima and Tetsuo Tamai. Aspect-oriented software design with a variant of UML/STD. In *Proceedings of the 2006 International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools (SCESM'06)*, pages 44–50, New York, NY, USA, 2006. ACM.

[38] Awais Rashid, Peter Sawyer, Ana M. D. Moreira, and João Araújo. Early aspects: A model for aspect-oriented requirements engineering. In *Proceedings of the 10th Anniversary IEEE Joint International Conference on Requirements Engineering (RE'02)*, pages 199–202, Washington, DC, USA, 2002. IEEE Computer Society.

[39] Bedir Tekinerdogan. ASAAM: Aspectual software architecture analysis method. *Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA'04)*, 2004.

[40] Awais Rashid, Ana M. D. Moreira, João Araújo, Paul Clements, and Bedir Tekinerdogan. Early aspects: Aspect-oriented requirements engineering and architecture design. At `http://www.early-aspects.net`.

[41] Edward D. Willink. *Meta-Compilation for C++*. PhD thesis, Computer Science Research Group, University of Surrey, January 2000.

[42] Shigeru Chiba. Program transformation with reflection and aspect-oriented programming. In Ralf Lämmel, João Saraiva, and Joost Visser, editors, *Generative and Transformational Techniques in Software Engineering (GTTSE'05)*, volume 4143 of *Lecture Notes in Computer Science*, pages 65–94. Springer, 2006.

[43] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. *SIGSOFT Software Engineering Notes*, 26(5):88–98, 2001.

[44] Sven Apel, Thomas Leich, Marko Rosenmüller, and Gunter Saake. FeatureC++: On the symbiosis of feature-oriented and aspect-oriented programming. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE'05)*, pages 125–140. Springer, 2005.

[45] Ran Ettinger and Mathieu Verbaere. Untangling: a slice extraction refactoring. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD'04)*, pages 93–101, New York, NY, USA, 2004. ACM.

[46] Symbian Ltd. Symbian Operating System WWW site. At URL `http://www.symbian.com/`.

[47] Jane Sales. *Symbian OS Internals*. John Wiley & Sons, 2005.

[48] Paul Clements and Linda Northrop. *Software Product Lines : Practices and Patterns*. Addison–Wesley, 2001.

[49] S60 WWW site. At URL http://www.s60.com/.

[50] Sander Siezen. Symbian OS version 9.1 Product description. At URL `http://www.symbian.com/`.

[51] Catherine Thorpe. Symbian OS version 8.1 Product description. At URL `http://www.symbian.com/`.

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

[53] Harwey Siy and Lawrence Votta. Does the modern code inspection have value? In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, pages 281–290, Washington, DC, USA, 2001. IEEE Computer Society.

[54] Gimpel Software. PC-Lint software. At `http://www.gimpel.com`.

[55] Matthias Urban. The PUMA user's manual. At URL `http://ivs.cs.uni-magdeburg.de/~puma`, 2000.

[56] Haeng Kon Kim. Aspect oriented testing frameworks for embedded software. In *Studies in Computational Intelligence*, volume 149, pages 75–88. Springer, september 2008.

[57] Digia. EUnit Professional. At `http://www.digia.com`.

[58] LDRA Software Technology. LDRA testbed software. At `http://www.ldra.co.uk`.

[59] Werner Damm and David Harel. LSCs: Breathing life into message sequence charts. In *Proceedings of the IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 293–312, Deventer, The Netherlands, The Netherlands, 1999. Kluwer, B.V.

[60] Amie L. Souter, David Shepherd, and Lori L. Pollock. Testing with respect to concerns. In *Proceedings of the International Conference on Software Maintenance (ICSM'03)*, pages 54–63, Washington, DC, USA, 2003. IEEE Computer Society.

[61] Andreas Gregoriades and Alistair Sutcliffe. Scenario-based assessment of nonfunctional requirements. *IEEE Transactions on Software Engineering*, 31(5):392–409, 2005.

[62] Kimmo Kiviluoma, Johannes Koskinen, and Tommi Mikkonen. Runtime monitoring of architecturally significant behaviors using behavioral profiles and aspects. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis (ISSTA'06)*, pages 181–190, New York, NY, USA, 2006. ACM.

[63] Yankui Feng, Xiaodong Liu, and Jon Kerridge. A product line based aspect-oriented generative unit testing approach to building quality components. In *Proceedings of the 31st Annual International Computer Software and Applications Conference - Vol. 2- (COMPSAC'07)*, pages 403–408, Washington, DC, USA, 2007. IEEE Computer Society.

[64] Sebastian Benz. AspectT: aspect-oriented test case instantiation. In *Proceedings of the 7th International Conference on Aspect-Oriented Software Development (AOSD'08)*, pages 1–12, New York, NY, USA, 2008. ACM.

[65] Frank Budinsky, Stephen A. Brodsky, and Ed Merks. *Eclipse Modeling Framework*. Pearson Education, 2003.

[66] Luis Daniel Benavides Navarro, Rémi Douence, and Mario Südholt. Debugging and testing middleware with aspect-based control-flow and causal patterns. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware (Middleware'08)*, pages 183–202, New York, NY, USA, 2008. Springer-Verlag.

[67] Roberta Coelho, Uirá Kulesza, Arndt von Staa, and Carlos Lucena. Unit testing in multi-agent systems using mock agents and aspects. In *Proceedings of the 2006 International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS'06)*, pages 83–90, New York, NY, USA, 2006. ACM.

[68] Alejandra Cechich and Macario Polo. Black-box evaluation of COTS components using aspects and metadata. In *Proceedings of the 4th International Conference on Product Focused Software Process Improvement (PROFES'02)*, pages 494–508, London, UK, 2002. Springer-Verlag.

[69] Arie Van Deursen, Marius Marin, and Leon Moonen. A systematic aspectoriented refactoring and testing strategy, and its application to JHhotDraw, SEN-R0507. Technical report, Centrum voor Wiskunde en Informatica, 2005.

[70] Reza Meimandi Parizi and Abdul Azim Ghani. A survey on aspect-oriented testing approaches. In *Proceedings of the The 2007 International Conference Computational Science and its Applications (ICCSA'07)*, pages 78–85, Washington, DC, USA, 2007. IEEE Computer Society.

[71] Syed Asad Ali Naqvi, Shaukat Ali, and M. Uzair Khan. An evaluation of aspect oriented testing techniques. In *Proceedings of the IEEE Symposium on Emerging Technologies*, pages 461–466, Islamabad, Pakistan, 2005. IEEE.

[72] Nikhil Kumar, Dinakar Sosale, Sadhana Nivedita Konuganti, and Ajay Rathi. Enabling the adoption of aspects – testing aspects: a risk

model, fault model and patterns. In *Proceedings of the 8th ACM International Conference on Aspect-Oriented Software Development (AOSD'09)*, pages 197–206, New York, NY, USA, 2009. ACM.

[73] Philippe Massicotte, Mourad Badri, and Linda Badri. Generating aspects-classes integration testing sequences: A collaboration diagram based strategy. In *Proceedings of the Third ACIS International Conference on Software Engineering Research, Management and Applications (SERA'05)*, pages 30–39, Washington, DC, USA, 2005. IEEE Computer Society.

[74] Dianxiang Xu and Weifeng Xu. State-based incremental testing of aspect-oriented programs. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD'06)*, pages 180–189, New York, NY, USA, 2006. ACM.

[75] Uwe D.C. Hohenstein and Michael C. Jäger. Using aspect-orientation in industrial projects: appreciated or damned? In *Proceedings of the 8th ACM International Conference on Aspect-Oriented Software Development (AOSD'09)*, pages 213–222, New York, NY, USA, 2009. ACM.

[I] Pesonen, J., Assessing Production Testing Software Adaptability to a Product-line. In *Proceedings of the 11th Nordic Workshop on Programming and Software Development Tools and Techniques* (NWPER 2004), Turku Centre for Computer Science, Turku, Finland, August 2004. TUCS General Publication Number 34. Turku Centre for Computer Science, 2004.

[II] Pesonen, J., Katara, M. and Mikkonen, T. Production-Testing of Embedded Systems with Aspects. In *Proceedings of the Haifa Verification Conference*, IBM Haifa Labs, Haifa, Israel, November 2005. Number 3875 in Lecture Notes in Computer Science. Springer, 2005.

Reprinted, with permission, from S. Ur, E. Bin, and Y. Wolfsthal (Eds.): Haifa Verification Conf. 2005, LNCS 3875, pp. 90–102, 2006. © Springer-Verlag Berlin Heidelberg 2006

[III] © 2006 IEEE. Reprinted, with permission, from Pesonen, J., Extending Software Integration Testing Using Aspects in Symbian OS. In *Proceedings of Testing: Academic & Industrial Conference - Practice And Research Techniques* (TAIC PART 2006), Windsor, United Kingdom, August 2006. IEEE Computer Society, 2006.

[IV] © 2006 IEEE. Reprinted, with permission, from Metsä, J., Katara, M. and Mikkonen, T., Testing Non-Functional Requirements with Aspects: An Industrial Case Study. In *Proceedings of the Seventh International Conference on Quality Software* (QSIC 2007), Portland, Oregon, USA, October 2007. IEEE Computer Society, 2007.

[V] © 2006 IEEE. Reprinted, with permission, from Metsä, J., Katara, M., and Mikkonen, T., Comparing Aspects with Conventional Techniques for Increasing Testability. In *Proceedings of the First International Conference on Software Testing Verification and Validation* (ICST 2008), Lillehammer, Norway, April 2008. IEEE Computer Society, 2008.

[VI] Maoz, S., Metsä, J., Katara, M., Model-Based Test Specification and Execution Using Live Sequence Charts and the S2A Compiler: an Industrial Experience. Technical Report 4, Tampere University of Technology, Department of Software Systems, 2009.