ESCOLA POLITÉCNICA
DE PERNAMBUCO

Programa de Pós-Graduação em Engenharia da Computação

Testware Support: Addressing Test Elements and Supporting a Benchmarking Framework in Aspect-Oriented Software Assessment

Dissertação de Mestrado

Engenharia da Computação

Liana Soares de Oliveira e Silva
Orientador: Prof. Dr. Sérgio Castelo Branco Soares

Recife, 22 de Junho de 2009

UNIVERSIDADE
DE PERNAMBUCO

ESCOLA POLITÉCNICA
DE PERNAMBUCO

Testware Support: Addressing Test Elements and Supporting a Benchmarking Framework in Aspect-Oriented Software Assessment

Dissertação de Mestrado

Engenharia da Computação

Esta Dissertação é apresentada como requisito parcial para obtenção do título de Mestre em Engenharia da Computação pela Escola Politécnica de Pernambuco – Universidade de Pernambuco.

Liana Soares de Oliveira e Silva
Orientador: Prof. Dr. Sérgio Castelo Branco Soares

Recife, 22 de Junho de 2009

ESCOLA
POLITÉCNICA DE
PERNAMBUCO

Liana Soares de Oliveira e Silva

Testware Support: Addressing Test Elements and Supporting a Benchmarking Framework in Aspect-Oriented Software Assessment

Acknowledgements

First of all, I would like to thank God for giving me strength and sanity to get this far. To my mother Miriam who has always believed in me and has never measured efforts to support me in the pursuit of my goals. To my father Frederico who is no longer among us but has taught me so much, I did not join the Navy but I know I made him proud. To my sister Daniela and my niece Daphne who cheer up my days and keep me standing. To my dear grandfather Francisco of whom many gifts I have inherited. To my dear grandmother Luisa who has showed me that God is always above all things.

To my advisor Sérgio for having always been supportive along these two years, and since college classes. Has found the balance between work demanding and being friendly, has been a fundamental piece to assemble this conquer.

Thanks to my friends and family spread out over Natal, Recife, Rio and São Paulo.

Thanks to my MSc colleagues from "O que vale é a gréia". I am sure the tracks in the forest, the events attendance and the nerdy talks have somehow helped me along the way. To Georgina for being so available to help on administrative issues at the University.

Thanks to the BSc students who have helped me on the experiments, data collections and facing issues. To the researchers from Lancaster University, USP and UFPE. To SPG research group for the discussions and support.

Thanks to the professors from the Department of Computing and Systems at University of Pernambuco, and the University for funding my research.

Abstract

The process of developing or evolving a software system involves many activities and different principles, techniques, methods and tools have been used to help building a reliable product. New project development ideas and programming principles have been suggested to support the software development process. Aspect-Oriented Programming has been proposed as a new technique to develop software achieving better modularity through separation of crosscutting concerns, that otherwise would be tangled with other concerns and spread along the modules. Also, the reliance on services and information requires software to function correctly over many years ahead. This scenario stands out the need of having qualified software. Software testing plays an essential role to uncover and correct as many of the potential errors as possible according to the software project's strategy and policies. Without an adequate technique to assure quality and reliability, the software system may decrease over its lifetime. Experiments and studies in software engineering have taken place to provide sufficient evidence regarding suitability, limits, qualities, costs and associated risks of the subject under observation within this context. A Benchmarking Framework (BF) has been proposed in this regard defining criteria and appropriated guidelines to assess the representativeness of empirical studies and Aspect-Oriented (AO) benchmark candidate applications characteristics. Therefore, this dissertation presents a Testware Support with test core elements addressing test elements to purport the capabilities of a software project against testing issues to conduct the project into the right test direction. Such initiative extends and helps supporting the BF, providing the minimum testware information required within a software project or case study in order to provide effective test measures and support more reasonable decisions regarding its maintainability and evolution. The effectiveness of the testware support is assessed through its appliance in two different studies with AO applications and the evaluation of the BF extension considering the insertion of the testware support.

Keywords: Software Testing, Testware, Software Maintenance, Aspect-Oriented Software Development, Benchmarking, Testbed.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The evolution of mankind has been a subject of social and anthropological studies and theories dealing with social and cultural evolution, ever since the evolutionism paradigm concept was made in Biology. The idea of Charles Darwin [138] that the living species are capable to transform themselves throughout time and the most adapted to their environment are naturally prioritized and more able to survive longer has also fit as a concept to early social sciences. Many sociologists have created social theories with different perspectives and some of them [139, 140, 141, 142] defend technological progress as the main factor heading the development of human civilization. Gerhard Lenski [1] declares that the more information and knowledge a given society has, the more advanced it is.

Lenski's approach became more relevant when technological advances followed and increased through time representing massive progress on different areas, such as in economic system, agriculture, civil engineering, architecture, communication and telecommunication systems. In the ancient world, technology meant the invention of means that would evolve a civilization era, such as fire, bow, pottery, domestication of animals, agriculture, metalworking, the alphabet, the writing and the like. Thus, the researchers were able to establish a link between social progress and technological progress. They argued that different environments and technology required different adaptations, and that as a resource base or technology changed, so too would a culture. They point that the determinant factors in the development of a given culture are mostly related to technology, but noted that there are secondary factors. Lenski has focused on information, its amount and use. Such declaration was of paramount importance towards the growing importance of information technologies and technology progress over the decades. The globalization in the 1970s and the upcoming years have been studied [143, 144] towards the trends that would determine the development of computers and technological progress. Thus, the importance of software in the modern world could no longer be underestimated.

Technological improvements have exponentially grown in such a way that information systems came to a point where daily activities and business decisions depend on their efficiency. For example, in the modern world, a failure in a transaction software from a bank, a long time of response in a device management connection or even just a software engineering experiment which has not been properly tested and analyzed can cause serious consequences, such as financial damages, loss of customer trust, waste of time and energy and mistaken pipelined paths, among other disadvantages.

The process of developing a software system involves many activities and different principles, techniques, methods and tools have been suggested to help building a reliable product. In the middle of these advances, new project development ideas and programming principles have been suggested to support the software development process [39]. As a result, a more comprehensive and flexible software code became available. With the evolution of these principles and the programming techniques gradual upgrades, Object-Oriented programming (OOP) has arisen and became the dominant programming paradigm in the 90s [40]. Despite OOP has caused a great revolution on the way how to create software, it still had some issues, such as not being able to separate all concerns, especially the ones regarding functional interests. Hence, not even a decade later, many developers found what seemed to be the resolution of OOP problems: Aspect-Oriented Programming (AOP) [41] has been proposed as a new technique to develop software achieving better separation of crosscutting concerns, instead of being spread into the code.

The reliance on services and information requires software to function correctly over a long time, so that means that possible errors that could occur on the software product are not meant to be there. This scenario stands out the need of having qualified software, which means the bugs and associated risks are to be reduced. Software quality entails more than just the elimination of failures. According to ISO/IEC-Standard 9126-1 [2], some factors belong to software quality: efficiency, reliability, maintainability, portability, functionality, usability, security, and interoperability.

Among the existing techniques of software verification and validation in order to increase quality, software testing plays an essential role to uncover and correct as many of the potential errors as possible before its delivery to final use. Software testing can provide evidences to measure quality based on requirements and warn the software engineer of potential errors. The test engineer can determine if the observed system behavior conforms to its specifications and aiming at finding defects, testing can bring to light the lack in quality, which may reveal itself in defects. Besides testing, there are other quality assurance alternatives such as formal verification, inspection, defect prevention and fault tolerance. All these activities need to be managed during quality engineering process in order to keep pace with quality assurance strategies early defined in the product development planning.

Different levels can be distinguished when software is tested throughout its life cycle [5]. Thus, the focus and test objectives may change and different types of testing can be more relevant than others. Generally, four main types of testing can be identified: functional testing, non-functional testing, structural testing and testing related to changes. These techniques must be seen as complementary one of the other. It is appropriate to consider all the existing different types of test techniques in a project, since testing a system’s functionality or component only at a specific level may not be enough to meet overall test objectives [6]. Based on those types of testing, different approaches and test criteria have been developed to provide a systematic evaluation of test activity.

Because exhaustive testing is not possible and because there are so many different established criteria, a crucial point that has to be faced when dealing with software development and software testing is the choice of a test strategy to be followed in project or study. The test strategy is part of the test plan and it defines priorities depending on the risks involved, it specifies test techniques and test exit criteria.

The cost of a failure on a high-tech system increases accordingly to its importance and when a critical application or study fails, serious damages can be derived. Considered one of the most expensive tasks in a software development process [3], software testing can reach up until 50 percent of a project's total cost. Still, the information obtained with testing is of paramount importance to other tasks of the software development process, such as debugging, maintenance and reliability estimation [4].

Empirical and experimental studies regarding software testing and software testing techniques are of great importance for they allow the establishment of strategies of low cost and high efficacy [7]. Empirical studies, such as to verify the effectiveness of structure-based criteria [8] and evolutionary testing for structural test data generation [9], have been developed throughout the years evaluating testing criteria, analyzing their characteristics and performing comparisons among each other in order to provide reliable examples and models to be taken for future studies.

Regarding the context of this dissertation, testing also plays an important role on software maintenance. More than 50 percent of a software system's life cycle costs are spent on maintenance [10]. As the system is used after it has been released, it is modified either to correct errors or to improve the original system. After each modification, the system must be retested in order to verify if the modifications have impacted overall system's functionalities. This task is known as regression testing [23, 24, 25, 85, 86, 199], in which the goal is to minimize the cost of system revalidation. However, changes at any level may lead to all-levels updates and this can lead to an enormous task force.

Generally, one of the drawbacks towards the progress of software engineering is associated with the first steps to be taken into a software maintainability empirical study [13], which include selecting, developing and adapting representative systems that can be used as benchmark for future studies. In spite of software engineering being part of a science that aims to fulfill the need of smoothing complex activities, software systems development tasks are still strongly dependable on human creativity and participation; and this may reduce the accuracy of software engineering studies. Thus, scientific experiments are required to testify and increase the level of reliability of new methods, techniques, processes and tools suggested by a theoretical software engineering study.

Literature regarding software maintenance contains few pieces when comparing to software development activities [11]. Still, it is possible to say that most of the issues associated with software maintenance are related to the way the software has been planned and developed. Based on latest studies [12], software maintainers can use information from test analysis to reveal valuable maintenance information. The testability of a system, i.e., "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", according to IEEE Standard Glossary definition, has the power of reducing testing difficulties, and thereby reducing testing costs and time, and, thus, increasing effectiveness on software maintenance. The ideas and importance regarding the attention to testability in the beginning of the software development process have been studied by different authors [14, 15, 16, 17, 18], but what can certainly be outlined from all of them is that the higher the testability level in a software system, the easier it will be to test it and, hence, the lower the testing effort [19].

Test models [20], test procedures [21] and studies [22] regarding programming languages particularities that need to be taken into account when a software system is under maintenance have helped testers and software maintainers to understand the way to handle software evolution and systematically improve their jobs on providing a better strategy on this task. Different regression test techniques have also been suggested [23, 24, 25, 88, 89] for the test engineers to find the best strategy that would fit their project, considering test environment and test automation. Nevertheless, regression testing is not always well seen and followed in a project or is inadequately performed: either the testing of new features or the revalidation of old ones, or both, is sacrificed [24, 26]. In a survey of 118 software development organizations, only 12 percent of these organizations were found to have mechanisms for assuring some level of adequacy in their regression testing [27].

Without an adequate technique to assure quality and reliability to software development or maintenance, the system will decrease over its lifetime. Software systems require continuous change and enhancement to satisfy new and changed user needs and preferences, business requirements and other reasons.

Empirical studies in software engineering aim to provide sufficient evidence regarding suitability, limits, qualities, costs and associated risks of the subject under observation. However, software development's organizations face the heterogeneity of study reporting, which hinders the integration of information into a common body of knowledge and, hence, incomprehensive decision support when selecting a software engineering technology, method, technique or tool [28]. It is possible to cite some reasons for this lack of solidity in software engineering empirical studies:

- There is no pattern of where to find required information: it is difficult to find relevant basic information for the study because the same type of information is located in different sections of different study reports;
- The difficulty to handle and control every artifact involved in the study: there may be variables, such as different technologies, methodologies, process, personnel, partial results, environment, that may turn the experimental job into an ad-hoc confused procedure;
- The portability of the results: what has been derived from a study may not always fit in another study, if a minor variable is altered. Empirical evidences and conclusions cannot always be generalized and applied to a different project;
- The general costs to allocate people, tools and energy: during the process of software development, experiments may be required but may be declined due to the high probability of costs that may be required from different fields.

However, not to drive attention (and money) to empirical study in the beginning of software development or in software maintenance can lead to even higher costs later on when the software is already being finally used and not studied as it should. With this principle in mind, different researchers [28, 29, 30, 31, 32, 33] have reported the need of a standardized system to conduct empirical studies that could improve the way new technologies are emerged, especially from academia to industry. The transference of a technology from academic field to industry usually takes an average of 18 years to be effectively disseminated [34]. Thus, the need of results and scientific evidences become an obstacle when organizations' and institutions' decision makers are required to evaluate the risks and benefits of technologies.

Several guidelines [34, 35, 36, 37, 38, 115, 116] have been proposed aiming to reduce the gap of scientific substantiation address and assessment. However, the particularities of the area have increased the level of difficulty in finding the best path to follow.

AOP, for instance, is a relatively new approach and there is little evidence about the advantages and disadvantages regarding its maintainability [42]. Testing and analyzing Aspect-Oriented (AO) software must deal with new problems introduced by AO language characteristics and the ones inherited from OOP as well. Encapsulation, inheritance, polymorphism, aspects; they all provide benefits to a project and to coding, but they also offer new challenges to testing and maintenance [43].

1.1 Motivation

One of the creators of AOP, Gregor Kiczales, has suggested that "for a programming paradigm to reach a broad acceptance, it needs to be sufficient expressive, efficient, intuitive, compatible and provide a good supportive tool" [44]. In spite of still being considered a new concept, AOP is already surfacing these characteristics. Several studies [45, 46, 47, 82, 100] have been conducted to show that AOP can indeed work and, furthermore, it is even being adopted on the implementation of applications from different domains like web systems [48, 99], operational systems [47], software product lines [50, 51, 96] and middleware [49]. However, there is a lack of contributions to the community when advantages and disadvantages regarding AOP maintainability are under discussion [51, 52].

Benchmarks have been used in software engineering to compare different techniques dealing with software evolution. Few examples [53, 54] have been proposed listing the characteristic attributes of software systems, representative cases and attributes to classify the merits of the techniques and tools themselves. Still to create such benchmark model within the context of OA software maintenance required dealing with empirical studies general difficulties and overcoming AOP related challenges. Hence, such benchmarking may not be considered an easy task. On top of that, a Benchmarking Framework (BF) [42] has been proposed to assist on preparing and standardizing empirical evaluations regarding AO software maintenance techniques. It defines criteria, standards and appropriated guidelines to assess the AO software maintenance characteristics, across benchmark applications.

The development and maintenance of software systems involve complex relations among a high number of artifacts. Different versions and configurations evolve in such a way that the update of requirements can mean adaptations in associated testware[1]. While it is clear that testing intends to detect failure, test activities are strongly related to software evolution, for when a bug is corrected, a new functionality may become available. When testing different software features and components in a larger perspective, the number of new available functionalities that could have been bugged and, thus, properly detected and fixed, can understandably be increased and significant. Besides, a piece of software that evolves without any endorsement that it indeed functions as expected and there is a minimum quality assured with it, is mostly vulnerable to address inappropriate maintenance or evolution scenarios concerning inadequate software elements, and not even knowing it.

The strong interconnection between the test process and the development and maintenance processes means that appropriate adjustments must be made to achieve optimum collaboration [55]. Different studies have aimed to provide a characterization schema for software testing techniques [56, 57], tools [58] and/or framework [59, 60], where test engineers could define data models or be assisted when selecting a test

[1] **Testware**: Artifacts produced during the test process required to plan, design and execute tests, such as documentation, scripts, inputs, expected results, set-up and clear-up procedures, files, databases, environment, and any additional software or utilities used in testing [6].

technique. However, the studies were not general enough to be used as a whole element within a software evolutionary benchmarking; let alone adequate to fit and support an AO software maintenance benchmarking framework, as the one [42] mentioned above.

With proper testware in a software maintenance benchmarking, the quality improvement engineers in a software project define the related necessary information about how the test data can be useful during a software maintenance or evolution process in order to downsize the dimensions of studies to a more specific scenario to be handled, such as AO software evolution scenario. Thus, this dissertation has developed a testware support to extend such BF structure to consider the test context in it.

As it is explained in details in Chapter 5, this work has been motivated from a case study that has taken place within the context of the BF and was provided with poor test strategy and no test planning. As it was a complex case study, it happened not to be able to provide the expected successful outcome in the end of it, and the reasons mostly lie upon the lack of testing process flow knowledge. Based on such experience, the context and what the testing literature offers nowadays, this dissertation has proposed to create the testware support with test core elements to be addressed in a case study or software project, so that testing flows smoothly and in the right direction.

1.2 Objectives

The subject explored in this dissertation is related to the definitions of the AO software maintenance Benchmarking Framework [42], which is already a consistent contribution in which the context this work is inserted and makes use of. The main purpose of it is to define a testware support with testing core element to be introduced to the existing BF.

The testware support provides what it takes from the testware context to allow a higher confidence and positive effect on the scientific assessment of AO software maintenance and evolution. While the BF defines criteria and guidelines to evaluate maintainability characteristics of AO software, the core testing element embraces the minimum test information to enable a proper strategy, according to the software project particularities. Such information comprises the first basic concerns to consider, testing strategy, test environment, test tools, and the test people. To support maintenance and / or evolution decisions, questionnaires are presented to purport and surface the existing test information there is in the study and what can be done with it.

The issues observed in the case study that motivated the start of this work are described in details along Chapter 3, 4 and 5 and they served as lessons learned of what needed to have been addressed so that it could have been a successful study.

The testing core element aims to help design, verify and validate a software system under maintenance or through evolution using appropriate testing process flow to address adequate approaches to assess aspects for the software to evolve. With the introduction of the testing core element into the BF, researchers and professionals will have more resources to spread their visions and scope to reach a more critical and safe evaluation when selecting an application or effectively performing its evolution or maintenance than if they did not consider testware in the context.

This study will also help upcoming studies that rely on the BF for evaluation of scenarios to represent a proper AO software maintenance. It differs from a software process because it addresses the core test elements directly, initially assessing the first concerns in testing through a first set of questions in a questionnaire, as it shows ahead, and, after, assessing and addressing specific test elements, and not only the process. The test process can be handled along together with the useful information provided by the tesware support study assessment, according to the best approach that fits the project so that it flows properly.

This study aims reducing the likelihood of wrongly addressing maintenance assertions and decisions for a system to flow on its process. It also aims to improve the accuracy of planning and evolving future projects. This work is relevant and important for the reasons here introduced and because it is the first initiative to address test elements within the context of this kind of framework.

This initiative intends to reduce the lack of scientific results in AO studies, for its purpose is to smooth over the evaluation process and enable to spread their replications involving AO software maintenance. Hence, it will also become easier to evaluate the pros and cons when adopting AO techniques and understand the benefits of different AO testing approaches in different circumstances.

1.3 Contributions

The contributions of this dissertation can be summarized as:

- Definition of testware support:
  - Definition of test elements (first testing ideas, strategy, environment, tools, test factors, risks, test people) to assess test characteristics and test process flow in a software project or case study;
- Use of the testware support to guide researchers interested in testing perspectives:
  - Guide selection of applications and their maintenance scenarios, based on their testware and testing needs;
  - Guide evaluation of what test direction in a process flow to follow, based on test process, environment, tools, personnel, test factors and risks;
  - Guide test characteristics assessment in a software project within different contexts in its life cycle (development, maintenance, evolution);
  - Possibility of criteria extension considering further test elements to be addressed to comply with software project or case study goals.
- The extension of the AO software maintainability benchmarking framework:
  - More detailed assessment of characteristics in AO software;

  - More confidence in spreading the scope and considering information from another process that flows along software development or maintenance;
  - More controlled and assisted decision making from researchers and practitioners using the BF.
- Evaluation of using testware support:
  - Use of the testware support to address testing elements in software development or test projects, software maintenance projects, case study, experiments;
  - Use of testware support to surface existing test elements in a software project or case study;
  - Use of testware support to assess the state of existing test elements and detect lack of test direction in a project.
- Evaluation of using the extension of the BF considering testware support:
  - To plan more detailed new experiments;
  - To identify applications and benchmarking scenarios.

1.4 Organization of Dissertation

This dissertation is organized in further 6 chapters, in which the context and evaluation are described in details to provide more consistencies to the contributions. Chapter 2 describes the state of art, where it introduces the main concepts of OO and AO paradigms, software testing and a brief description of software maintenance, its importance and problems. In Chapter 3, background information is introduced, such as a discussion regarding the empirical studies on software maintenance and evolution, the existing structures to support such works and the case study inserted in this context that has motivated this work, for it was provided with a poor test strategy and testware, which led to unsuccessful results. In Chapter 4, the testware support is presented addressing the test core elements to be considered in a case study, such as the one which motivated this dissertation. The testware support provides such test elements, and its benefits and advantages are discussed. The proposed extension of the BF now considering test issues with the insertion of the testware support in it is presented in Chapter 5. Chapter 6 presents the evaluations of both testware support analyzed alone and its insertion within the BF with two different applications in different case studies. The conclusions of this work are presented in Chapter 7. Finally, the References are found next and the Appendixes afterwards.

Chapter 2

State of Art Review

This chapter presents some of the concepts and perspectives related to software development, testing, maintenance and evolution, as well as it characterizes the scenario for the associated perspectives regarding AO software. Most attention is paid to testing views, concepts and challenges, as it is the main focus of this dissertation. Initially, in Section 2.1 and Section 2.2, there is a brief description of the main characteristics and concepts of Object-Oriented and Aspect-Oriented software development. Next, in Section 2.3, some of the main directions in software testing are described. The definitions of testing during software development life cycle, test levels, types of testing, Test Maturity Model concepts and main test techniques are emphasized because of the context of this dissertation. In Section 2.4, the main criteria defined for OO and AO software testing are described. The importance of software maintenance, drawbacks and testing process flow within maintenance are discussed in Section 2.5. Following this line, software evolution and testing process flow when evolving software are described in Section 2.6. Finally, Section 2.7 presents final considerations about this chapter.

2.1 Object-Oriented Software Development

The programming language Smalltalk can be seen as the result of the full concept of Object-Oriented Programming (OOP) developed in the 80s [61]. Born as the solution for the search of quality software improvement that would keep up with hardware increasing complexity back then, OOP addressed common problems focusing on data rather than processes. Reuse, modularization, simplification and maintenance costs reduction were some of the benefits of using OOP.

The OO programming language must provide support for some key language concepts, such as objects, classes and subclasses, inheritance and inclusion polymorphism, which are briefly described as follows. In OOP, data and procedures are part of a single basic element, the **object**. Objects provide a natural way to model real and cyber-world entities, such as files, databases, web pages and user interface components. A family of objects with similar variable components and methods (functions) is known as a **class**. An extension of a class with additional components or methods is known as a **subclass**, which can extend itself to its own subclasses, building a hierarchy of classes. The subclass usually inherits all methods of its superclass. This **inheritance** concept helps programmers on reuse, which impacts his productivity. The concept of **inclusion polymorphism** represents the possibility of an object to have different forms, i.e., an object of a subclass may be treated as an object of its superclass.

This concept enabled adding new methods to existing classes without the need of recompiling the application by **dynamic binding**. Also objects can send and receive messages. A **message** is an object's request to execute one of its methods. It can change the state of an object or send messages to other objects.

Because of the enormous progress from procedure-oriented and imperative programming that dominated the industry before OOP arose, OOP became the master programming paradigm in 1990s. OOP enabled a higher abstraction level than other methodologies, allowing a better representation of the elements that would compose a problem's solution and this would be good for reuse. However, the advantages would also be a challenge for a developer to face and be loyal to. The implementation of concerns (also known as some functionality, requirement or interest in a program) should follow the methodology's rules and design patterns principles. The goal was to design a program so that functions could be optimized independently on other functions, making it easier to understand, design and maintain complex systems; but some concerns defy these forms of implementation and crosscut multiple abstractions in a program, making a class handle more than its own concern.

The incapability of OOP to modularize systemic concerns causes the decentralization of the code and its implications [62]:

- Code replication: a same treatment may be necessary in different classes that do not belong to the same inheritance tree;
- Maintenance hindrance: every change on existing concerns implies a complete scan throughout the code to locate a possible implementation modification;
- Less code reuse: a change in a specific orthogonal concern in a generic class may hinder the generic method use by another class;
- Understanding: the code becomes less comprehensible with systemic concerns implemented (scattering code and tangled code).

These problems break the essential principles of OOP, impoverish software design quality and restrict the tasks of projecting, implementing and maintaining software artifacts. Thus, it is possible to observe that although OOP may simplify some issues from older programming languages and provide benefits to software systems projects, it may also generate new issues and concerns to software quality.

2.2 Aspect-Oriented Software Development

Aspect-Oriented Programming (AOP) [44] has been proposed as a way to improve the separation of concerns in software development aiming to support better reuse and software evolution.

Concerns are relevant characteristics of an application, which can be divided into aspects, and thus, representing the requirements of the application. Crosscutting concerns are transversal aspects of an application (concerns) which affect (crosscut) other concerns, i.e., that can interfere in a system implementation. These concerns usually could not be cleanly separated from the whole implementation, so they remained scattered and tangled along with the code; AOP supports these crosscutting concerns modularization through the abstractions, which enables their separation and composition when building a software system. AOP enables a higher abstraction level in software development, enabling a well defined separation of the system components. And because the components can be well separated and organized, they can be better reused and maintained and their readability becomes friendlier.

Using AOP, it is possible to organize a software implementation based on functional and non-functional requirements. From functional requirements, a set of components can be expressed in a contemporaneous programming language, such as Java [63]. From non-functional requirements, a set of aspects (crosscutting concerns) can be derived, which are related to the properties that affect the system's behavior. With this technique, the non-functional requirements can be easily manipulated without causing major impact on general code (functional requirements), since they are not tangled and scattered throughout system's units. Thus, AOP enables software development using such aspects, which implies on isolation, composition and reuse of aspect implementation code [64].

AspectJ [65] is the most widely used AO programming language, created as of Java [63]. It uses Java-like syntax and has included IDE[2] integrations for displaying crosscutting structure since its initial public release in 2001. The main language constructors are described as follows. The concept of **aspect** is introduced as a new abstraction that encapsulates a crosscutting functionality. Similar to a class, an aspect contains methods and attributes and it can introduce methods, attributes, interface implementation declaration and class extension declaration using a construction known as **inter-type declaration**. Introduced members may be visible to every class and aspect

[2] Integrated Development Environment.

(public inter-type declaration) or be only internal to the aspect (private inter-type declaration). **Join points** are essential to the composition process between aspects and classes. They represent a well defined location in a program execution, such as a method call or an attribute access, where an aspect may alter the behavior of the base code. **Pointcuts** are the execution points in an application at which crosscutting concern need to be applied. An aspect may specify **advices** to define some code that should be executed when a pointcut is reached. They can be executed **before**, **after** or instead (**around**) of the pointcut.

The improved software modularization with the separation of code of similar concerns that affects different parts of the system, provided by AO paradigm, is an essential factor to ease the development, maintenance and evolution of software systems, because it reduces dependencies and system modules coupling [42]. Since AOP is a relatively new technique, but there are already many implemented programming languages of it [65, 100], further specific features and advantages may vary according to the adopted language.

2.3 Software Testing

Many books and papers have already been published and presented regarding software testing. The first edition of the Myers' book [3] from 1979 introduced the software testing principles that are still applicable even today. Despite it had become both easier and more difficult than ever, according to Myers – more difficult due to the vast technology progress and complexity and easier because of the sophistication evolvement making the testing tasks smoother and more reusable – software testing is still a process followed in software development projects to make sure an application does what it has been designed to do. In order to reach the specified software behavior, the existing bugs in it should be found and fixed, and a better software quality should be, therefore, achieved.

As mentioned in Chapter 1, testing activities are important to support quality assurance. Its basic idea involves the execution of software and the observation of its behavior or outcome. If a failure is observed, the execution record is analyzed to locate and fix the fault(s) that caused the failure. Otherwise, confidence is obtained only by when the software under testing is more likely to fulfill its designated functions [68].

Although it is possible to see that executing tests is important, it is also necessary to have a plan of actions and reports on the outcome of the testing activities. The idea of a fundamental test process for all levels of testing has developed over the years. Whatever the level of testing in a project, the same type of main activities was always observed, although there may be a different amount of formality at different levels. Still, it is possible to state the following division of major activities within the fundamental test process [5, 6, 68]:

- Test planning and preparation;
- Test analysis and design;
- Test implementation and execution;
- Test exit criteria evaluation and reporting;
- Test closure activities.

Although logically sequential, these activities may overlap or take place concurrently.

It is important to differentiate the existing nomenclature in software testing literature, as it will be used many times on this dissertation:

- **Defect** is a bug, a fault, a flaw in a component or software system that can cause the component or software system to fail to perform its required function. If a defect is found during execution, it may cause a failure of the component or software system.
- **Error** is a mistake, a human action that produces an incorrect result.
- **Failure** is a deviation of the component or software system from its expected delivery, service or result [5, 6].

The definitions of the test levels and how testing is inserted in a software project is described in the following sections.

2.3.1 Testing within Software Life Cycle

To understand the relation between software development and software test activities, it is important to know that the test levels (as it has been mentioned in Section 2.3) are related to the software development life cycle. There are many software development life cycle models [69, 70, 71] that have been created in order to achieve different aims and goals in a software project. Since software testing is not a stand-alone task – it is inserted in software development life cycle – the chosen model in a project directly impacts software testing activities. The existing models specify the various stages of the process and they influence the determination of test techniques to use.

In general V-model [70], testing is handled as an activity that is performed throughout development, being as important as such. Starting test design early based on business requirements and development artifacts enables test engineers to find defects on specifications and documentations. As it can be observed in Figure 2.1 [5], the two branches of the letter “V” symbolizes development and testing processes, showing that the software system is gradually created as the program elements are inserted at each level. In practice, the V-model may have more, less or different levels depending on the project. At each test level, the development outcomes must be checked whether they fulfill original requirements (**validation**), whereas the outcome of a particular development phase is verified whether satisfies the imposed conditions at the start of that phase (**verification**).

The V-model divides the development into constructive activities in which, at each level, new testing characteristics are added. On the left branch, in the initial phase of defining and gathering requirements, which is made by the customer or user, the system needs are specified and, therefore, the features and purpose of the software system can be defined. Then, in functional system design, the requirements are mapped onto functions of the new system being developed and the requirements documentation is used to create functional system test cases. Next, the implementation of the system is designed: interfaces are defined and the system is decomposed into manageable smaller components, enabling independent development but still considering interaction with each other. During components specification, the behavior and inner structure within a component are defined for each subsystem. Finally, each specified component (module, unit, class) is implemented in a program language during programming phase.

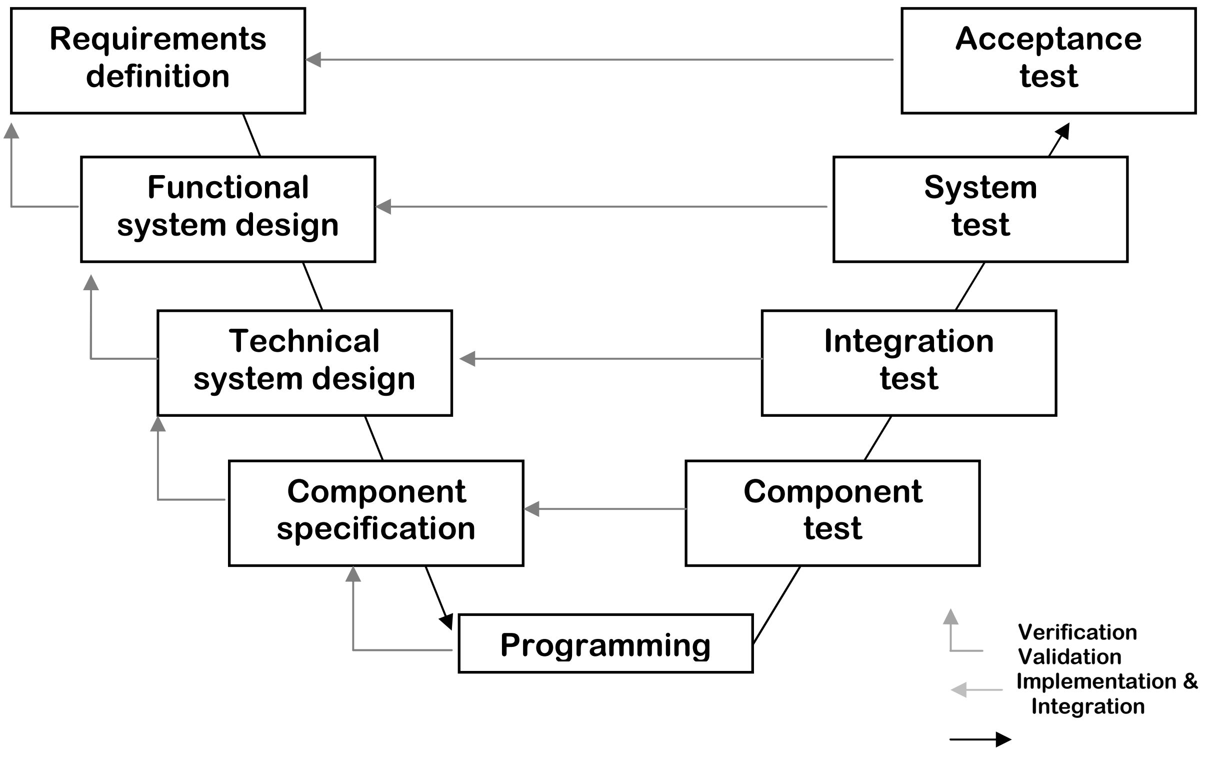


Figure 2.1 General V-model

Having levels of software development helps detailing it and finding mistakes' root cause. It also helps defining test levels along the process; thus, the right branch of the V-model represents correspondent test levels to every development level previously explained:

- **Component test**: the software unit is tested for the first time. Modules, units, programs, classes, objects are independently tested to make sure external attributes do not influence if a defect is detected. Typically, this level of testing considers the code behavior being tested with support of developers and development support, such as a unit test framework or debugging tool;

- **Integration test**: at this level, software components are supposed to have already been individually tested and eventual defects are supposed to have already been corrected, so that the components can be tested in groups. The goal of integration testing is that eventual faults derived from integrated components interaction and interfaces can be exposed, in a manner that such conflicts could not occur during component testing, but could only be manifested by integrating the system parts and detected by such further interoperability tests;

- **System test**: checks whether the integrated system meets specified requirements. It concerns the system's behavior from the customer and future user's point of view, investigating both functional and non-functional requirements of the system. Typical non-functional tests would include performance and reliability. Failures from wrong, incomplete or inconsistent implementation of requirements are the ones that should be detected by system testing, and requirements that are not documented should be identified;

- **Acceptance test**: most often focused on a validation type of testing that aim to establish confidence in the system; for instance, by verifying its usability. The user or customer are usually responsible for this testing and their main focus should not be to find defects in the system, but to determine whether the software system fits its purpose, before being deployed.

After the software system has experienced the constructive phases and test levels throughout its development, it should be ready to be deployed, whether in customer field or for other purposes, and used for a period of time. For non-experienced software engineers and testers, it can be easy to imagine that all the work has been accomplished by then. However, what happens in the real world, and experienced software engineers and testers are well aware of, is that, afterwards, changes in requirements are common requests to update the system to keep up with technology evolution for instance, such as adding new functionalities, correcting or improving existing ones, improving data base capacity, inserting new mechanisms and techniques of programming to enable a better, faster and more user-friendly system and so on. Thus, software life cycle maintenance is a forward-focused activity used to prolong the productive lifespan of a software system [72]. Such incremental modifications of software systems are often referred to collectively as software evolution.

Maintenance Testing within Software Life Cycle

In Chapter 1, the testability concept of a system has been briefly introduced and it is possible to understand that the higher the testability level in a software system is, the easier it is to test it. This subsection introduces the concept of testing that is handled during this life cycle phase, which is called **maintenance testing**, and it usually consists of two parts [6]:

1. Testing the change;
2. Regression testing to show that the rest of the system has not been affected by the maintenance work.

A deep care is necessary when analyzing what parts of the code may be unintentionally affected by the maintenance. The regression testing is usually planned based on risk and impact analysis.

From the point of view of testing, there are two types of modifications than are part of maintenance: the ones in which testing may be planned, and the ones ad-hoc corrective modifications which cannot be planned [6]. The following types of planned modification may be identified:

- Perfective: to adapt software to what the user wants and needs;
- Adaptive: to adapt software to environmental changes;
- Corrective: deferrable correction of defects.

Overall, the corrective planned modifications represent over 90% of all maintenance work on software systems [73], i.e., the standard structured test existing approach is almost fully applicable to planned modifications only, while ad-hoc corrective modifications need to immediately be found a solution, and the risk analysis and the choice of best test approach may be compromised.

Moreover, regression testing is crucial for large organizations with critical software systems. Therefore, a software system being inadequately regression tested in a software project without a proper technique applied during its maintenance, its quality and reliability will languish over its lifetime [24, 26].

Testing in Incremental Development

Incremental development is usually related to pieces of development that are gradually delivered. The initial version of software contains basic functionalities and the requirements are grown over time to upcoming versions. Throughout its development, software receives increments with new features and each version contains more and improved functionalities than the previous ones. Some models [74, 75, 76] following this principle have been created, and testing must be adapted to be continuous, having reusable test cases that can be reused for every increment.

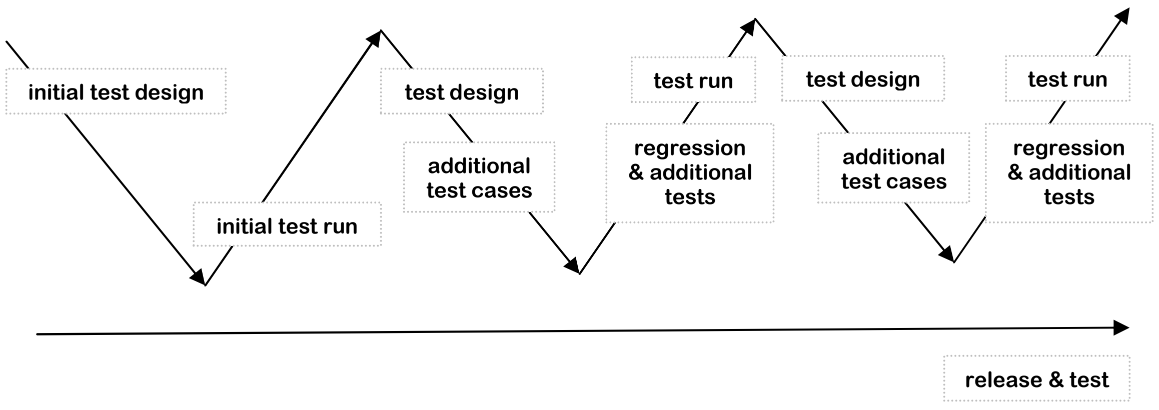


Figure 2.2 Testing in incremental development

The practical way for testing to follow such model is to run several V-models one after the other, in which every phase reuses the existing testware and adds what is needed for what is new. Figure 2.2, adapted from Spillner et. al [5], pictures how the testing is handled in incremental development.

2.3.2 Types of Testing

As stated on Section 2.3.1, there are different levels of testing. The objective of testing in each level changes distinguishing different types of testing that are classified in software testing literature as follows:

- Functional testing (or black box testing);
- Non-functional testing (or white box testing);
- Experience-based testing.

Functional Testing

Also known as black box testing because the test object is seen as a black box, functional testing is mostly used for high levels of testing, where the inner structure of the test object is not considered. Instead, it concerns the compliance of the system or component with specified functional requirements. Its primary goal is to assess whether the software does what is supposed to do. It can be performed at different levels but, for different levels, there are different requirements and objectives.

Functional testing approach can be used as a technique to elaborate test cases. Whether formal or informal, the requirements are the basis from which the test cases will be designed to exercise the software system in order to find defects, not considering its implementation structure. Test design is supposed to make a reasonable selection of all possible test cases, thus, there are some specification-based techniques that assess the test object in different ways, the two most important are briefly described below:

- Equivalence partitioning: test only one condition from each partition. The equivalence partition (or equivalence class) contains a division of a test conditions set, in a way that they are sufficiently representative to be tested as one, avoiding unnecessary test cases;

- Boundary value analysis: test the boundaries between partitions, addressing test efficiency, for the number of need test cases is reduced in order to obtain a certain level of confidence in the software under test;

Non-Functional Testing

Known as white box testing because the test object is seen a white box, since the software code is known and used for test design. Non-functional testing is also known as structural testing, because the structure of the test object (component hierarchy, flow control, data flow) is considered. It can be applied to lower levels of testing, i.e. component and integration test.

The generic idea behind this code-based testing is that every part of the code of the test object is to be executed at least once. The usual primary goal of such technique is to reach a previously defined coverage percentage of code statements while testing; thus, there are some basic white box test criteria, such as:

- Statement coverage: test a predefined number of statements of the test object's code, which can be better understood with the use of a control flow graph, where control elements, connections and sequences can be represented;
- Branch coverage: test every decision (`true` or `false`) that can determine the next move of the program; in other words, all possible transitions from a decision node are to be tested. It also can be better understood with the use of a control flow graph, in which, for this technique, the focus is on edges in the graph;
- Multiple-condition coverage: test of decisions based on several conditions. It must be considered which input data lead to which result of the condition or condition part and which parts of the software system will be executed after the decision;
- Path coverage: test all different paths through the test object, combining nodes and edges from a control flow graph. There may be a huge number of existing paths in a software system; hence, it may not be a practical approach to require execution of all paths from the code.

Experience-based Testing

Besides the methodical criteria described above, there are non-systematic test techniques that can be used as complementary to uncover still remaining faults in the software system. This type of testing is based on a person's knowledge, experience, imagination, creativity and testing skills, so that she can act as a "bug hunter". The most common techniques are:

- Error guessing: with the experience of a tester in previous similar test objects from previous studies, she is able to know where the defects are most likely to lurk;
- Exploratory testing: when there is no test planning. Based on intuition and experience of the tester, this approach is mostly adopted when the specifications are poor and time is limited, however it can also be used as complementary to systematic testing.

2.3.3 Test Maturity Model

A software process can be defined as a set of activities, methods, practices and transformations that people use to develop and maintain software and the associated products [108]. The capability of a software process describes the range of expected results that can be achieved by following a software process, and the software process capability of an organization provides one means of predicting the most likely outcomes to be expected from the next software project the organization undertakes. Based on this idea, software process maturity can be defined as the extent to which a specific process is explicitly defined, managed, measured, controlled and effective. It implies a potential growth in capability and indicates both the richness of an organization's process and the consistency with which is applied in its projects.

It is easy to cite some of the benefits for an organization to reach maturity in its projects: long-term success of a project increase, reduction of software development risks, manageability, etc. As testing is applied in its broadest sense to encompass all software quality-related activities, a Test Maturity Model (TMM) [109] has been created to improve the testing process thorough application of the TMM maturity criteria, with the goal of having a highly positive impact on software quality, software engineering productivity, and cycle time reduction efforts.

The development of the initial version of the TMM was guided by the work done on the Software Capability Maturity Model (CMM), a process improvement model that has received widespread support from the software industry [110]. Figure 2.3 [109] illustrates the five levels of such model that are self explained. The characteristics of each level are described in terms of organizational goals and testing capability. Such model works as a reference to help organizations assess and improve their testing processes, for it is based on a set of principles in which software engineers practitioners can assess and evaluate their software testing processes.

A TMM Assessment Model (TMM-AM) can help organizations assess and improve their testing processes by:

- determining its level of testing maturity;
- identifying its test process strengths and weaknesses;
- developing action plans for test process improvement;
- identifying mature testing subprocesses that are candidates for reuse.

For such, the TMM-AM is composed by a set of three components: a questionnaire, the assessment procedure and team training and selection criteria. A set of inputs and outputs is also prescribed for the TMM-AM that guide an assessment team in carrying out a testing process self-assessment. Whereas some models for test process improvement focus only on high-level testing or address only one aspect of structured testing, e.g. test organization, the TMM addresses static and dynamic testing. With respect to dynamic testing both low-level and high-level testing are within the TMM scope. Studying the model more in detail one will learn that the model addresses all four cornerstones for structured testing (life cycle, techniques, infrastructure and organization).

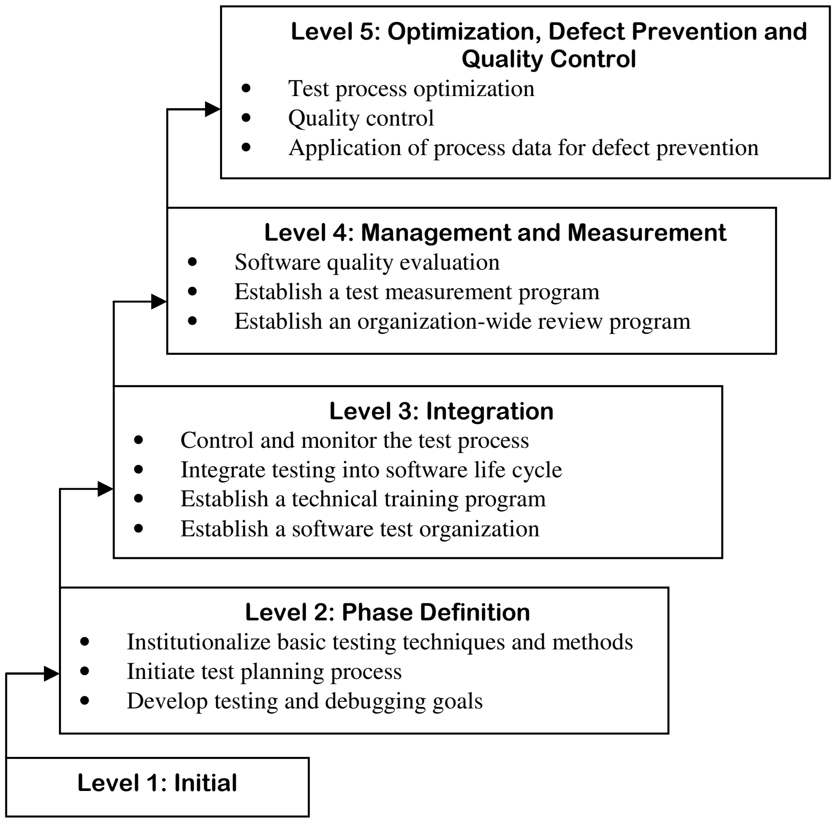

Level 5: Optimization, Defect Prevention and Quality Control
• Test process optimization
• Quality control
• Application of process data for defect prevention
Level 4: Management and Measurement
• Software quality evaluation
• Establish a test measurement program
• Establish an organization-wide review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

Figure 2.3 The 5-level structure of the TMM

2.4 Object-Oriented and Aspect-Oriented Software Testing

Despite AO and OO features mentioned in Sections 2.1 and 2.2 represented great advances for software development activities, they also stress out new technical challenges to software testing. Testing and analyzing OO and AO software systems must deal with the new challenges introduced by the paradigm's characteristics

Encapsulation may create obstacles that limit the visibility of the implementation state but it may also prevent defects from global variables stored data access and few lines methods may turn control flow defects less likely to occur – the many ways a system can be composed due to different context that can derive from distinct hierarchy levels and run-time defined system behavior may lead to occurrence of errors, once an incorrect association of the language resources is taken place. According to Binder [66], some essential features of OOP languages pose new fault hazards:

- Dynamic binding and complex inheritance structures create many opportunities for faults due to unanticipated bindings or misinterpretation of correct usage;
- Interface programming errors are a leading cause of faults in procedural languages. OOP typically have many small components and therefore more interfaces. Interfaces errors are more likely;
- Objects preserve state, but state control (the acceptable sequence of events) is typically distributed over an entire program. State control errors are likely.

Towards OO software testing specifically, there are some researches in the community combining testing strategies and the paradigm's particularities. For instance, Lima and Travassos [106] have presented a new strategy for integration testing of Object-Oriented software systems by creating a set of heuristics and a process for its use allowing establishing an integration priority order for the classes to be tested. Harrold and Rothermel have considered a class as the smallest unit of an OO code [107] and proposed the test of a class dataflow that considers the interactions between the public methods when being called in different sequences. They have also considered the dataflow test on classes' integration.

Based on the previous studies mentioned above, it is possible to conclude that conventional testing techniques can be adapted to test Object-Oriented software systems, since OO code matters a lot and an usual data-flow approach may not fit properly.

As previously mentioned, AspectJ [65] has been created to address some of the OOP issues and it is currently the most used AOP language to develop software. This Java extension adds new constructors, as presented in Section 2.2, to allow the modularization of crosscutting concerns implementation. However, with AOP, the aspectized code affects the implementation of multiple classes and methods; the software development process is changed. Classes and methods are still developed as before, but instead of embedding the crosscutting code into method bodies, separated aspects are defined containing this kind of code. Later, the aspects are woven into the classes that represent the core concerns of the software system. Once complete, the woven targets should be the behavior union between core and crosscutting concerns. On top of that, AOP raises new issues, such as [67]:

- Aspects do not have independent identity or existence. They depend upon the context of some other class for their identity and execution context;
- Aspect implementations can be tightly coupled to their woven context. Aspects depend on the internal representation and implementation of classes into which they are woven. Changes to these classes will likely propagate to the aspects;
- Control and data dependencies are not readily apparent from the source code of aspects or classes. Due to the nature of the weaving process, the developer of classes or aspects knows neither the resulting control flow nor data flow structure of the resulting woven artifact. Thus, relating failures to the corresponding faults may be difficult;
- Emergent behavior. The root cause of a fault may lie in the implementation of a class or an aspect, or it may be a side effect of a particular weave order of multiple classes.

These challenges should not and cannot be addressed by traditional unit or integration test techniques applied in OO software systems, for they are not feasible to aspects. Zhao was the very first researcher to propose a structural testing approach for AOP. He has suggested [101] that the basic testing unit is an aspect, in Aspect-Oriented programming. On his approach, he suggests to test the aspects together with the methods which the behavior might be affected by advices and to test the classes together with the advices that might affect their behavior. The research conducted by Xu et al. [102] deals with combining state models (class and aspect) and flow graphs (method and advice) as an aspect scope coverage model for producing test suites. Essentially, the result is a hybrid testing model which is a combination of a responsibility-based testing model and an implementation-based testing model. The approach consists of merging the class state model and the aspect state model into an aspect scope state model (ASSM). The ASSM allows tracing the behavior of aspect-oriented programs by identifying sequence results of the states transitions of the AOP.

On some other researched works, it is discussed the problematic of testing AO programs. A defect in an aspect-oriented program may not be found in components, nor in aspects, but on the weaving process, for example. Alexander and others [67] propose a fault model for AO programs. The model explores the types of defects that may happen due to the paradigm's particularities and group them according to its nature. They suggest that criteria and testing strategies for AOP should be developed in terms of the fault model. A discussion about the proposed fault model [103] states that the only two fault types that seem to be not adequately tested by means of extensions of traditional techniques are types 2 and 5, as detailed below. They demand AOP exclusive testing techniques.

- Type 2 – Incorrect aspect precedence: When the same code portion is affected by more than one aspect, depending on the order in which aspects are woven to the base code, differences can occur. When no composition precedence is defined, all the possible compositions are potential instances to be considered.

- Type 5 - Incorrect focus of control flow: Pointcut designators that contain conditions on the execution stack define a join point set that cannot be evaluated statically. Therefore, errors can be hidden in them which are difficult to expose, in that they require very specific execution conditions to hold.

Some works [90] suggest techniques for incremental testing of programs that use aspects. On this approach, the base classes should first be implemented and tested for, afterwards, have the aspects added and tested one by one, so that integration testing tasks can be performed. Zhou [104] proposes an approach for unit testing, integration testing and system testing for AOP. Another incremental testing approach [103] suggests that the base classes should be tested initially not considering the aspects, intending to reveal errors that are not related to aspects. Smoothly, the aspects are added to software testing. One of the issues pointed out is the need of creating stubs and drivers to simulate the aspect's behavior.

Another thing that may be issued from incremental AOP testing is the creation of cycles that would cause a dependency inside the code between aspects and classes. Elsewhere [105], different types of dependencies are studied and two alternatives for sorting classes on AOPs are applied.

2.5 Software Maintenance

Canning [77] has described software maintenance as an "iceberg", where there are many further problems and potential costs hidden under the surface than what comes up during software development. More than 50 percent of the costs from the life cycle of a software system are spent on maintenance [10]. Even adopting the best development criteria, a software system is created concerning deployment details constraints. When a system is deployed and has been running for quite some time, eventual changes, such as platform changes, operational system changes and other technology update changes, may require an update from the system so it is able to attend user's needs. Such update range from simple code edit to meaningful improvements and new requirements adjustments.

It is not reasonable to presume that testing software throughout development will uncover every existing bug. When using the system, different bugs that have not been detected on testing activities may be manifested, whether because a combination of features in use was necessary to arouse such bug or because testing activities were not vast enough or due to many other possible reasons. The diagnosis and correction of one or more bugs on this phase is known as **corrective maintenance**.

When a new hardware generation is launched, new operational systems or updates from old ones are on the market and gadgets and other system's elements are constantly being modified, the **adaptive maintenance** is responsible to modify the software system so it has the adequate interface to the environment.

When the system is being used, the users usually have recommendations to the developers for some kind of improvement on it. New functionalities, changes on existing ones and expansions are the general requests that the **perfective maintenance** is determined to attend. This activity is responsible for the most part of all applied effort in software maintenance [11]. These three terms defining the three types and activities on software maintenance described above have been determined in 1976 [78] when it was already possible to visualize such trend.

Finally, when the software system is modified to improve its reliability or its future maintenance, or to offer a better framework for future expansions and updates, the **preventive maintenance** takes place. Such term, on the other hand, is commonly used on hardware maintenance, although software and hardware development and maintenance processes strongly differ from one another.

2.5.1 Problems in Software Maintenance

Most of the issues regarding software maintenance can be connected to software planning and development. Based on Pressman's list [11], below are the main reasons of the problems associated to software maintenance:

- Lack of documentation of software requirements

  Sometimes the developers are part of the software project planning and responsible to capture the system's requirements; they start developing right away without specifying well with the project managers and stakeholders[3] the main functionalities of it. When the system is ready, it is deployed and the developers are usually the focal point for when the users need help. This may be familiar to small projects, but in projects of any size, when the documentation is not adequate or sufficient and / or the developers who initially built the system are no longer reachable, the system will certainly be hard to evolve, since it will be difficult for the current developers to understand the exact requirements just on looking at the code or at the system functioning;

- Undefined software process

  If a software system is developed in a project without an accurate software development process, it will be hard to track actions and milestones that would help software maintenance activities identifying the best actions and techniques to follow on its evolution.

- Many different versions

  When a software system is developed and evolved in different branches, it is difficult or impossible to track all the changes applied to such application. This is very common within academic works, when there are not many suitable available applications to an experiment, and the existing ones are used by so many different institutions and organizations that sometimes the people working on it do not even known about the update releases from other studies.

- Most of the software systems are not designed to evolve

  It is not common, except when it is pre-determined, to find software developers coding systems concerning future possible changes on each class being created. The developers need to make extra effort to visualize future

[3] **Stakeholders**: Group who shares interests, as in an enterprise, which affects and can be affected by an organization's actions.

changes and to separate eventual dependencies, for instance, so this is not usually the case.

- Motivation

  It is not usually a fun task having to modify or adapt a software system. Psychologically, the individuals are mostly satisfied when creating things, but modifying them tells them that they have done their jobs perfectly, since they need a modification. It is rather hard for an individual to admit and hear the critic that the application he developed is not good enough and requires improvement; even if it is just an adaptive maintenance due to new technology arising that could not have been predicted by the time the system was being developed. For those reasons, it is not easy to deal with motivation skills from developers having to work on a software system maintenance.

After these problems' considerations, it can be understood how hard it may be to guarantee a well planned and integrate system when it has to be modified. Those problems cited above are general to any kind of software system. When effectively maintaining one, low-level aspects, such as particularities of the code and programming language used to code, need to be taken into consideration.

2.5.2 Maintenance Testing

The testing activities for maintenance basically resumes on regression testing [23, 24], which aims to provide confidence that the changes were made correctly and other portions of the software are not affected by them. Researchers on this subject performed studies analyzing the best regression technique selection [24, 25, 85] and the cost-effective metric from them [86], among other matters. Regression testing is important but expensive. Regression test selection technique could reduce the cost of regression testing by selecting a subset of an existing test suite to use in retesting a modified program. Such strategy is very suitable to software projects, since not every existing test case is required to be run. The prioritization of test selection usually helps on software projects cope with deadlines and costs. A test suite is a set of several test cases for a component or system under test, where the post condition of one test case is often used as the precondition for the next one

2.6 Software Evolution

The approach of Lenski stating that the development of human civilization is dictated by technological progresses reflects the importance of keeping software up to date. The identification and observation of relevant software behaviors has been established as the Laws of Evolution [92], firstly set in the early seventies, in which rules to software system evolution planning and management have enabled a gradual understanding of software process over the years. The eight laws of software evolution are listed in Table 2.1 below.

A full analysis of the the meaning and implications of this classification requires more discussion than cannot be provided here, but it is possible to observe that continuing changes and growth are necessary to keep the system's functionalities up to date to user's requirements over its lifetime. The system usually implements new features on its evolution and this usually implies increasing complexity, which makes it a strong proposition in this scene. The feedback from the results of the behavior under software system execution, observed by stakeholders, client and users reflects experience that changes perception, understanding, desires and ambition towards system evolution; hence it plays an important role on consistently concluding the refined version of the eight laws of software evolution.

Table 2.1 Current Statements of the Laws of Evolution

#	Law	Description
1	Continuing Change	A system must be continually adapted, else they become progressively less satisfactory in use
2	Increasing Complexity	As a system is evolved, its complexity increases unless work is done to maintain or reduce it
3	Self Regulation	Global system evolution processes are self regulating
4	Conservation of Organizational	Unless feedback mechanisms are appropriately adjusted, average effective global activity rate in an

	Stability	evolving system tends to remain constant over product lifetime
5	Conservation of Familiarity	In general, the incremental growth and long term growth of systems tend to decline
6	Continuing Growth	The functional capability of systems must be continually increased to maintain user satisfaction over the system lifetime
7	Declining Quality	Unless rigorously adapted to take into account for changes in the operational environment, the quality of a system will appear to be declining
8	Feedback System	Evolution processes are multi-level, multi-loop, multi-agent feedback systems

Evolution in large software systems can lead to serious challenges when dealing with increasing complexity. Empirical studies [28, 29, 32] are a useful mechanism for highlighting areas that need maintenance attention [78, 83], providing information to be taken into account when evolving systems [30, 31, 35, 36, 37, 38, 47] and creating frameworks [42, 53, 54] with models that may forecast effects and special conditions for specific types of system evolution.

2.7 Final Considerations

This chapter described the state of art on what regards to the context of this dissertation's subject. It has discussed the main directions regarding software testing, the importance of testing software throughout software development and maintenance, the testing levels, phases and types, TMM and it has stressed the associated knowledge for testing AO software.

The main testing studies and criteria related to AO software testing have been characterized, as a result from OO software evolution and its concepts, also presented in this chapter. It has been possible to verify that there are many issues when maintaining or evolving a software system and the main reasons can be related to software planning and development. Following this line, there is not much research considering testing activities during software maintenance or evolution that is not related to regression testing or regression test selection criteria. Testing may be, therefore, limited to regression testing and regression test selection techniques during software maintenance or evolution, which does not consider test tasks in a broader view, when evolution takes place in large software projects, arising higher possible test changes and challenges. The existing research is valid for specific scenarios, not always possible to become general enough to adapt to higher leveled views or fit different processes. There is a lack of contributions when the testware elements need update during a software system evolution or maintenance.

The TMM concept is also introduced and presented as a model to assess testing process and determine its maturity in a software project in an organization. Still, such model has proven to be of a great validity for evaluating testing process, but apart from or not considering the maintenance scenario and evolution details of a software system.

Therefore, the revision here presented motivates the development of this dissertation that aims to provide testware support, with test core elements to be taken into consideration and applied when maintaining or evolving an AO software system, as well as to define the extension of an existing Benchmarking Framework for AO software maintenance to consider such testware, as introduced in Chapter 1 and further detailed in this dissertation.

Chapter 3

Background

This chapter presents essential information regarding the context in which this dissertation is inserted. A general view on the empirical studies on software maintenance, introducing concepts of frameworks, benchmarks and testbeds are described as well as the perspective for Aspect-Oriented (AO) software in the software engineering community. Such information is fundamental to introduce the testbed (Section 3.2) and the benchmarking framework (Section 3.3) to which the case study introduced in the next section (Section 3.4) is associated. Such case study was the starting point of the work of this dissertation to identify testing issues and needs and, therefore, provide the solution for such problems (Chapter 4).

3.1 Empirical Studies on Software Maintenance

Section 2.5 has introduced the basic concepts of software maintenance, presenting the types of possible maintenance changes; it has discussed its importance and drawbacks, as well as it has introduced the testing process flow within such context. It has been possible to comprehend that such activity is long and expensive [11] and such issues can be explained by a bad, incomplete software project planning, which lead to bad projected software, hence, difficult to maintain and evolve. Thus, it becomes essential to stimulate the development and creation of new techniques, methods, processes, technologies and paradigms in order to ease and reduce costs of maintenance tasks and, therefore, enable to identify the most adequate approaches to comply with the changing scenarios.

To improve software evolution it is necessary to identify what are the characteristics of the software system, environment and personnel that may affect such process. This need has motivated several empirical studies to search for the perfect answers. Although many issues have been detected as harmful to software maintenance (Section 2.5.1), there are still other factors from other perspectives related to the invested effort in maintenance tasks. Hence, it is important to identify them to enable better and more effective software maintenance. In some studies [111, 112], these factors include software system's characteristics, such as structure, size, age, input and output data, type of application and programming language. Large software systems tend to more maintenance effort when compared to smaller and simpler systems. This happens because the larger the software system is, more time is necessary to understand it, and also because of the number of diverse functionalities that complex applications usually posses [113].

With the objective of extending the discussion regarding the factors that may cause problems in maintenance, other researches [114, 115] have investigated the relation between types of changes and generated effects. The characteristics repeat each other many times, generating cause and consequence patterns that can be associated to the factors previously mentioned. Both studies mentioned on this paragraph indicate the need of following a process and specific models during software maintenance phase, which help controlling environmental variables. However, such studies are not general enough when generating empirical evidence, nor when analyzing methods and techniques to best fit software maintenance.

There is still further research [116] that has evaluated a broader view of software maintenance, which contribute to software engineering empirical data. However, such investigation during software maintenance remains challenging; the existence of studies analyzing software systems maintenance that have characteristics which can be categorized to be used in various experiments, especially using new approaches like AO, is very limited.

Hence, the experience from related areas is important to define guides, models and methodologies for software experiment evaluations, in order to accelerate the generation of empirical evidence within software maintenance research. In the next sections, it is introduced the use of frameworks, benchmarks and testbeds focused on software maintenance activities.

3.1.1 Frameworks, Benchmarks and Testbeds

The term **framework** introduced in Chapter 1 in the context of this dissertation can be defined as a set of rules and attributes that are responsible to assist the execution and evaluation of process, techniques or methods of a specific field. It is not easy to define a generic framework to be used within software maintenance that simultaneously attend different specific domains, therefore researches [59, 60, 117] have been conducted in order to build it in a more limited scope, restricting its use to determined domains, but with an increased level of details. The frameworks, however, have the deficiency of not paying much attention to the technologies and techniques used during software maintenance process.

Thus, with the aim of covering such lack and analyze maintenance in a broader view, some studies [42, 53, 54, 58] have been conducted with the use of **benchmarks** to evolve and maintain software systems. Differently from frameworks that only provide criteria but do not allow the comparison between representative cases, benchmarks have the goal to enable the comparison among different techniques used in software maintenance. Researches [42, 53] have been conducted to establish a trustworthy method to evaluate such techniques, since the experiment environment is adequately configured by standards and the applications and studies are selected representatively and according to criteria. A specific benchmark study [42] has been used as background source for the development of this dissertation, as mentioned in Section 3.1.1 and is described in details in Section 3.3.

Finally, some other researches [97, 105, 118] have evaluated systematically the maintainability and performance of different methodologies in a specific matter. The difference here is that they regard a **testbed** and an evaluation process with specific metrics in order to evaluate the maintainability and performance of techniques. A testbed can be understood as a platform for experimentation in large development projects [145]. Testbeds allow for rigorous, transparent and replicable testing of scientific theories, computational tools, and other new technologies. In spite of restricting the studies to a specific domain, such research can serve as a base to further general maintenance studies and experiments. The testbed in which the context of this dissertation is inserted is introduced in Section 3.2.

The research regarding frameworks, benchmarks and testbeds definitions are innovative and provide a valid approach to advance software engineering on what regards to controlled empirical studies, which enable the production of significant scientific results. Such research should be exhaustively tested, extended and evaluated before being broadcasted to vast used in the community. The process of evaluating and validation is important to identify existing limitations and possible extensions, since the techniques, methods and technologies in this field are constantly evolving. Therefore, it is nearly impossible to define a structure, set of attributes or process that are static and complete. Technology evolves quickly and it demands concepts to be defined in a way that improvement and extensions to it are possible and allowed, such as considering a new domain in it or just a simple removal of bugs.

3.1.2 Aspect-Oriented Software

This subsection presents an introduction of relevant information regarding Aspect-Oriented (AO) software maintenance, evolution and testing, which is the context in which this dissertation is inserted.

Maintenance and Evolution

Aspect-Oriented Programming (AOP) modularizes crosscutting concerns that otherwise would be implemented spread over other modules and tangled with other requirements implemented in the software system. Thus, AOP improves software modularity with separation of the code implementing similar concerns, as explained in Section 2.2. Such characteristic is very important when developing, maintaining and evolving software systems, for it reduces the dependency and coupling of software modules with the introduction of a new implementation unit: the aspect. The aspect changes a software system behavior, since it encapsulates the code that is spread throughout software modules and it becomes the unique transversal implementation unit in the system. Such change may occur in well defined execution points (*join points*), as explained in Section 2.2, and some other points (*pointcuts*) may be affected by their execution as well.

AOP can be used in a software project as an approach to support and ease software evolution through the representation of new features development using aspects as patches in separated units composing the original software for evolving it to the next generation, in a planned evolution process. Using aspects within incremental modifications to a system's base classes increases the state space of the software model, after the evolution, it provides higher code readability and visibility which makes the code friendlier and easy to manipulate.

AOP is relatively a new software development paradigm and most of the studies [45, 46, 47, 49, 52] taken on this regard mostly evaluate advantages and disadvantages of whether using it or not. There is not much evidence [51, 52] of studies analyzing its maintainability, nor testing directions, values and issues. Some studies [51, 79] have presented case studies reporting several limitations on modularizing features when using AspectJ, such as the increase of coupling between aspects and classes due to the strong dependency of *pointcuts* on implementation details of the base code. This stands out the importance of performing studies evaluating crosscutting concerns implementation so that it is possible and easier to understand how they may behave when they become vulnerable to changes and updates associated to software maintenance and evolution. Aware of such information, the developers handling AO software maintenance or evolution have more tools to deal with necessary code changes and, thus, carry out a more reliable and reasonable approach for this task.

On top of that, some studies [46, 48, 51, 52] have discussed how crosscutting concerns could affect future implementation by comparing modularity characteristics and software stability; and have also analyzed the impacts of changing scenarios towards software evolution. Despite having few contributions on this regard, it is possible to cite some of the difficulties when dealing with AO software maintenance or evolution:

- Knowing which parts of the code are affected by the aspects;
- The need of understanding other modules to understand the behavior of one module;
- Identifying the behavior that may be affected by the aspect;
- Identifying parts of the programs affected by a defect that needs correction.

Some researchers [80, 81] have tried to solve some of these problems creating models to define modules and aspects' dependencies. However, there is little evidence of studies [51, 52] and guides [37, 42, 47] that help developers identifying general elements to consider when evolving or maintaining AO software. Some studies [31, 32, 33, 34] sometimes are handled and applied so specifically in a special context or case study that cannot be reused to other scenarios.

As briefly mentioned on Chapter 1, a Benchmarking Framework [42] has been proposed to assist on the evaluation of AO software techniques, within the context of software maintenance. Such guide is of fundamental background information towards the contributions of this dissertation. Therefore, it is presented and discussed in more details in Section 3.3.

Software Testing

Although few but still helpful and useful empirical studies and analysis on regression testing, the contributions are limited when the scope is focused to AO software maintenance.

Crosscutting concerns degrade software quality of an OO software system. They negatively impact internal quality metrics, such as program size, coupling and separation of concerns. Such proposition served as a starting point to many researchers [46, 49, 51, 52, 80, 81, 82, 83] study different aspects of the paradigm that would provide more confidence towards quality in AO software maintenance and AO software evolution. Identifying the crosscutting concerns in order to manage them [80, 81] throughout software evolution has served as a motivation to studies focusing to keeping up the modularity [46, 82] or relative to design patters [49, 83]. Identifying control and data dependencies [84] could also help on determining when the semantics of one statement can affect the behavior of another statement and this would contribute to anticipating effects on the program's behavior.

Although the above mentioned studies have helped improving quality in AO software development and maintenance and may have eased the costs and issues, generally speaking, after the modifications for the evolution, the system still needed to face testing activities. The issues of crosscutting concerns affecting code quality has inspired researchers [83, 87] on studying the connections and relations between crosscutting concerns and defects in a software system. The researchers have studied the impact code quality is vulnerable to suffer. Some of them [52, 82, 87] focused on developing new metrics or adapting existing ones for quantifying crosscutting and assessing the impact of modularization crosscutting concerns using AOP techniques, improving separation of concerns. The theory [87] that a crosscutting concern is harmful for maintenance is acceptable for multiple (and sometimes unrelated) locations in the code have to be found and update simultaneously. Such complexity is still even higher when the concerns are required to untangle the code to attend update or maintenance requirements.

The existing approaches [106, 107] for OO software regression testing unfortunately are not effective for aspectized code. Specific testing techniques for AO software have been derived and developed [88, 89, 90, 91, 101, 102, 103, 104], in which structure-based methodology is unanimously the followed approach to develop regression test selection strategies. Xu [91] and Zhao et. Al [89] both have proposed a control flow graph (CFG) to model the control flow of AO programs. Xu [91] defines a new test selection criterion and implements the technique in a regression test selection for AO programs framework. In the technique proposed by Zhao et. Al [89], it is not included any complex situations, such as multiple advices or dynamic advices and there is no evaluation of this model. Xu and Xu [90] have presented an incremental approach to testing whether or not AO programs. With regard to aspect-orientation, they analyze the impact of aspects on the state transitions of base class objects and generate tests for base class and AO programs based on their state models. According to the authors, model-based testing is appealing because of several benefits:

- The modeling activity helps clarifying the requirements and enhance communication between developers and testers;
- Design models, if available, can be reused for testing purposes;
- Model-based testing process can also be automated;
- Model-based testing can improve error detection capability and reduce testing cost by automatically generating and executing many test cases.

Modeling is a broad concept that can be involved different perspectives of software development, such as design, specification, code generation, testing and reverse engineering. Taking aspects as incremental modifications to their base classes, the authors [90] identify how to reuse the concrete base class tests for testing AO programs according to AO state models. The incremental approach can be seen as similar to regression testing.

Xu and Rountev [88] propose a regression test selection technique based on AspectJ language features also building a CFG, which represents the interactions among multiple advices and captures the semantic intricacies of aspect-related interactions in order to be able to compare the algorithm for test selection.

In every approach here mentioned the aim is to reduce the cost of regression testing by selecting the criterion that best fits the object under test and software project’s constraints. The regression test technique is the most adequate way to validate the system after modifications have been applied. However, when the system needs change for upgrades, regression test approach may not be the only requested test direction to head up. Also regression testing may not be good for radically modified software, coverage-based testing, GUI testing, or complex simulation of software or hardware [119]. Specialized tools and support regarding necessary test elements (testware) may be required for such cases to address the existing variables and, therefore, the most adequate test criteria to fit them, so that the testing activities can provide the expected confidence that testing is intended to provide: that a software system has been safely and properly tested, in order to increase software system quality. This is the context in which this dissertation is inserted; it addresses a testware support (Chapter 4) provided with test core elements to be considered within an existing benchmarking framework, the BF [42], described in more details in Section 3.3.

3.2 The AO Software Development Testbed

In order to promote a smooth adoption of AO software development (AOSD) techniques, an AOSD testbed [105] has been idealized to provide end-to-end systematic comparison between such techniques, enabling the proponents of AO and non-AO techniques to compare their approach in a consistent manner. A testbed is a platform for experimentation, an environment to allocate and address experiments and tests, in which different aspects and elements are considered to compose such structure to become useful for its purposes. Therefore, the AOSD testbed here presented is not yet concluded. Research is currently taking place to address further items, problems and / or elements to provide higher effectiveness and consistence in its use and purposes.

The testbed is composed of:

i. A benchmarking application;
ii. An initial set of metrics suite to assess certain internal and external software attributes; and
iii. A repository of artifacts derived from AOSD approaches that are assessed based on the application of (i) and (ii).

The purpose of the testbed is to help answer questions regarding the effectiveness of AOSD throughout the development life cycle by:

i. Providing a set of core applications from different domains in which a variety of software engineering approaches can be applied;
ii. Defining metrics suites to facilitate end-to-end software life cycle assessment under different quality perspectives; and
iii. Providing a set of artifacts that have been created from applying a variety of AO and non-AO approaches to the applications provided by (i).

From providing such artifacts and case studies, the testbed has the goal to provide proponents of software engineering approaches with the ability to easily compare their approach to others. Such study is essential to stimulate the application of AO approaches in order to enable results to reach a wider audience.

Such testbed consists of four core elements (Figure 3.1 [105]):

i. Applications that contain a variety of crosscutting concerns;
ii. AO and non-AO approaches that are applied to a common application to generate artifacts;
iii. A suite of metrics associated with a variety of internal and external software attributes; and
iv. A set of metric results that have been gathered from applying the metrics suite mentioned in (iii) to the artifacts produced.

Each of these elements is extendable to include new instances of applications, approaches and metric suites. As it has been described on its definition and study [97, 105], for such testbed to be successful it requires contributions from software engineering community. Hence, in order to expand the testbed into a valuable resource, the benchmarking framework – mentioned many times throughout Chapter 1 and previous sections from this chapter – complements this initiative, as it is explained in details in the next section.

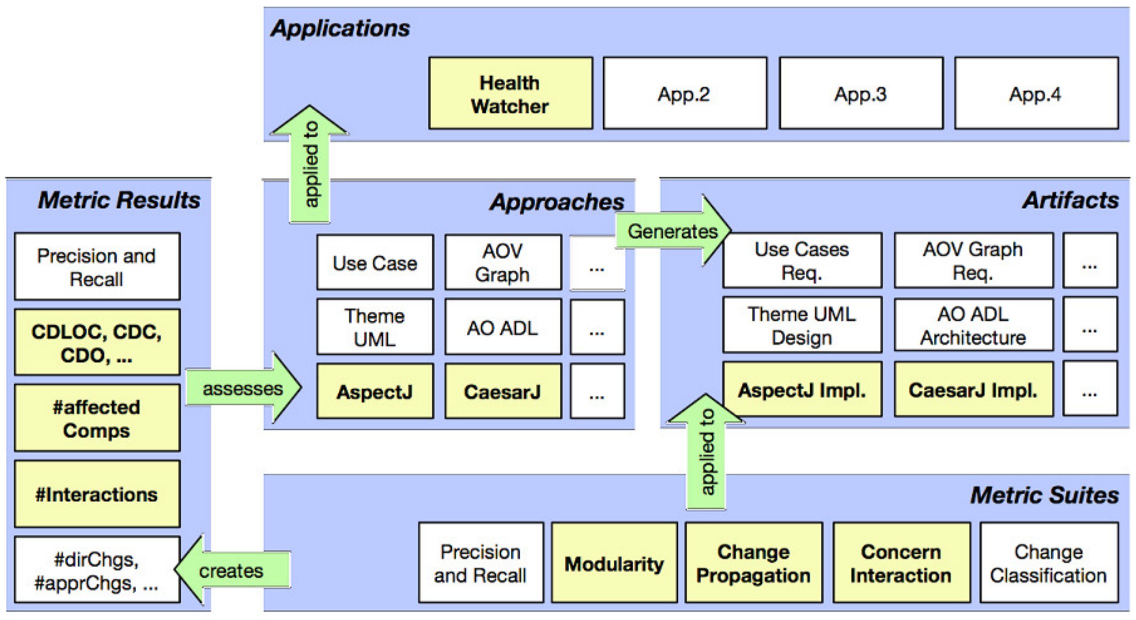


Figure 3.1 The various elements that compose the testbed

3.3 The Benchmarking Framework

This section introduces the Benchmarking Framework (BF) [42], which supports maintainability assessment of AOSD techniques by a definition of an idealized scheme for benchmark applications to assess maintainability attributes of AO techniques. The framework guides researchers and practitioners in selecting or adapting applications and their releases that best fit for specific experimental maintainability goals. The BF can also be used to support the design replication and evaluation of empirical studies. The users or framework stakeholders have been classified into two categories: the designer of empirical studies on AO software maintainability, and the benchmark designer. The goals and interests of each category and further details on this matter are described in Chapter 5 (Section 5.1).

The BF is the result of a research [42] extending the proposed AOSD Testbed described in Section 3.2 that aimed to provide a solid structure to fill the gap in scientific evidence associated to AO software maintenance in order to enable reduction of difficulties and challenges when generating such evidence. The BF provides the appropriate structure for applications to be evaluated and compared with regard to maintainability attributes of the applied techniques. Therefore, as mentioned, it is able to guide researchers and practitioners when selecting, elaborating or adapting applications in their process and in their respective maintenance scenarios, helping them become more adequate to attend specific goals of specific experiments or software system project on what regards to AO software maintenance. Besides, its systematic use when evaluating and comparing applications and change scenarios allows it to be used as a tool to verify benchmarks examples. Hence, it eases the identification of applications and change scenarios that are representative enough to be considered benchmarks of specific domains, which can be placed in a repository and used in future case studies and experiments. Thus, the main benefits of the BF can be summarized in:

- Assisting the evaluation of software development techniques;
- Supporting the elaboration, replication and evaluation of empirical studies;
- Easing the identification of applications and change scenarios to consider as benchmarks;
- Possible extension of the criteria to comply with different context of studies and applications.

The framework is composed by three main components:

i. The process;
ii. The product;
iii. The maintenance scenarios.

Further details on each of these components are explained in Chapter 5 (Section 5.1).

Figure 3.2 [42] illustrates the composition of the BF. The BF defines a process that consists in different stakeholder categories and some input/output transformations. For instance, a set of experimental requirements can serve as an input to generate the first version of an experiment plan. The output can also work as framework feedback in order to improve it. Besides the output generated by different kinds of stakeholders, there is a possible output that is common to all kinds: the cookbook, in which the actions experienced by the users, related to the use of evaluated techniques and methods, and the use of the framework itself are described. The cookbook is consisted by: (i) lessons learned and (ii) best practices. Such practice is important to gather and disseminate technical knowledge, issues and experiences from software engineers that can be used and learned from by other users. The BF also defines a list of criteria that classify the characteristics related to the product, i.e., applications or software systems. The criteria can be used in different approaches that go according to the goals of users, stakeholders and/or software project. The criteria include general attributes, common properties to any applications, such as system identification, packaging, life cycle documentation and development techniques; and AO attributes, the ones specific to the main characteristics related to the applications developed using AO techniques, such as classification of crosscutting concerns, composition of concerns, scope and AO language constructors. The BF also embraces the criteria that describe the main characteristics regarding change scenarios, which can help, for example, software engineers identify and map the impacts generated in an application during the implementation of different scenarios. Such criteria include the scenario description and the identification of the type of change, goals, levels, etc.

Nevertheless, there is no support on what regards to software testing within the BF, nor the testbed (Section 3.2). As the BF is a piece within the testbed universe, it has been considered as the base to insert the testware support (Chapter 4), which is the subject of this dissertation. The proposed insertion and the BF extension is presented in Chapter 5 and the evaluation of the testware support within the BF is presented in Chapter 6.

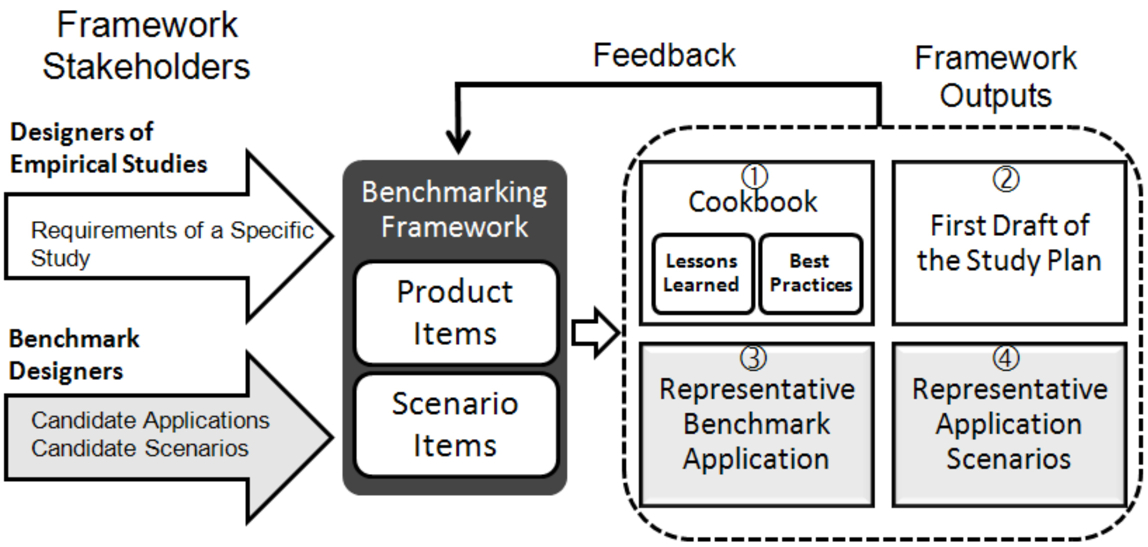


Figure 3.2 The inputs and outputs of the benchmarking framework process

3.4 The Case Study

This section introduces the case study that served as the starting point of this dissertation. The case study aimed to be an exploratory study on the error-proneness of the AOP mechanisms. It involved four OO releases of two medium-sized systems from different domains, each with an AO counterpart. The goal was to obtain high-test coverage and reveal results not yet found regarding the collected AOP faults. Such experiment was part of the research related to testbed here presented (Section 3.2); the two applications involved are already part of the testbed and the results to be obtained were supposed to provide the testbed with more artifacts and useful information regarding AOP related faults to be used in future studies. That was the first experiment within the testbed context considering testing elements, therefore, there were many different test factors to address. As this section explains, the test factors became risks and, therefore, issues, which motivated the creation of the testware support (Chapter 4) addressing testing process flow to address when considering testing as part of a software experiment or project. The case study involved different variables that are presented in this section. However, the details of the issues faced throughout this case study are described in Chapter 4, in which the different test elements can be identified and the respective associated issues in the case study are, then, detailed.

3.4.1 Goals

As briefly mentioned above, the goal of such case study was to identify error-prone AOP mechanisms when they are applied to evolving programs. The analysis if the nature of specific concerns exerts some influence on fault location embraces a wide range of heterogeneous concerns implemented as aspects. In addition, each AOP mechanism naturally introduces different forms of dependencies between the base classes and the aspectual modules; therefore, the goal was to evaluate how the degree of base-aspect coupling could help developers identify error-prone modules in evolving software systems.

3.4.2 Target Systems

Two medium-sized applications were the target systems in this exploratory study. The first one, iBATIS [120] is a Java-based open source framework for data mapping. It was developed in 2002 and over 60 releases are available at SourceForge.net[4] and Apache.org[5] repositories. The second is a Java Mobile application – MobileMedia – that manipulates music, photo and video for mobile devices. As different research groups have handled the applications in order to test them, this dissertation only regards the manipulation with MobileMedia testing.

MobileMedia (MM) is a J2ME (Java To Platform Micro Edition) application that has been developed based on a previous software product line (SPL) called MobilePhoto [96], which is a photo album (image viewer) appropriate for cellular phones and personal digital organizers (PDAs). The requirements for this application

[4] http://sourceforge.net/project/showfiles.php?group_id=61326

[5] http://archive.apache.org/dist/ibatis/binaries/ibatis.java/

included several different features. MobileMedia has been developed by including new mandatory, optional and alternative features, as it is illustrated by the application feature model in Figure 3.3.

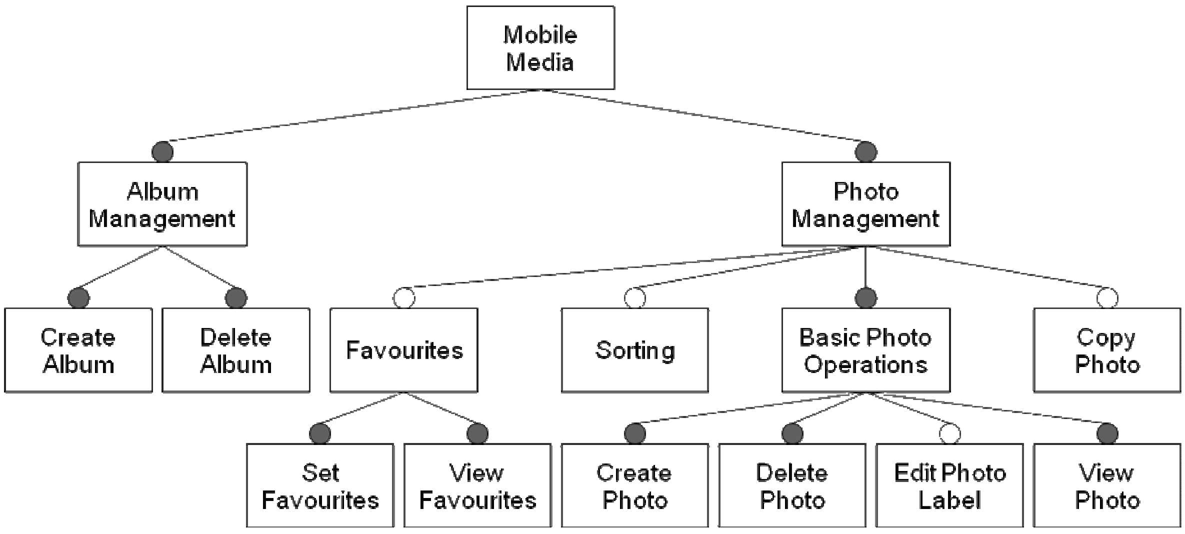


Figure 3.3 MobileMedia feature model

Software product lines (SPLs) [123] represent an important technology to support software variations. Assets reuse and cycle time reduction are some of the benefits SPLs can provide to a project. The products from a product line are very similar but differ from each other [124]. The baseline contains mandatory feature(s) and every evolution or update of it introducing optional/alternative features results in a new product. The evolution of product lines imposes changes from diverse nature, such as the transformation of mandatory features into optional or alternative ones and vice-versa [51]. These changes imply new test definitions to check whether or not they impacted other existing features or general software expected behavior.

The core features of MobileMedia are: create/delete media (music, photo, video), label media and view/play media. The alternative features relate to types of media supported: music, photo and/or video. The optional features are: count and sort media, copy media, edit photo label and set favorite media. For the case study, it has been used four Object-Oriented releases of MobileMedia SPL, and in each release there was an incorporated changed scenario by the introduction of a new feature. Table 3.1 summarizes the main changes for each product.

Table 3.1 Summary of changes in MobileMedia SPL

Release	Description
01	MobilePhoto core with mandatory features.
02	Mandatory features plus Sorting and Edit Photo Label feature implementation.
03	Mandatory features, Sorting and Edit Photo Label feature plus Set and View Favorite feature implementation.
04	Mandatory features, Sorting and Edit Photo Label features, Set and View Favorite feature plus Copy Photo feature implementation.

As MobileMedia is an application for academic studies, the releases used for this experiment can be found at http://mobilemedia.wiki.sourceforge.net/. The number details regarding Total Lines of Code (TLOC), Total Lines of Bytecode (TLOB), total number of methods and classes from the tested releases are illustrated on Table 3.2 and can be observed as they reflect how large the application is. The complexity and inclusion analysis of the SPL and criteria have not been considered for this scope.

Table 3.2 MobileMedia SPL OO Releases' details

	TLOC	TLOB	Classes	Methods
Release 01	1159	3058	24	122
Release 02	1316	3527	25	139
Release 03	1364	3728	25	142
Release 04	1559	4180	30	159

There was not much information regarding the AO releases of MM, nor how they were implemented, aside the information that the optional and alternative features and the exception handling code were what have been aspectized from the OO releases. Further details from the implications of such lack of information are described in Chapter 4 along with the definitions of the test core elements from the testware support, and in Section 3.4.4, where the tool support and the issues with AO releases are described.

3.4.3 Testing Strategy

Since the goals were all related to code-based metrics – such as fault location, relation between fault and aspects, etc. – the strategy adopted was structure-based testing, focusing on code coverage. However, such decision was not supported by a test process, plan or previous studies, as this was the first one regarding testing within the context of the AOSD testbed (Section 3.2). There was no testware support at all, besides the experience of the software engineers with JUnit testing [121] and the little documentation of the applications to be tested.

To achieve the determined goals, coverage-based testing was then determined to be applied, which relied on the claim that one cannot trust a piece of code if it still contains elements that have never been executed during testing [122]; structural-based testing has been discussed in Section 2.3.2. It was also taken into consideration that there is a larger currently available tooling support for code-based testing, once the software engineers involved were more experience with software development rather than software testing skilled. No other test techniques were discussed, nor was it ever considered. Therefore, the testing strategy resumed to write unit tests that would execute the pieces of code in order to cover it.

The only major decision regarding testing that was truly analyzed in this case study was the test exit criteria. To decide which coverage should be considered enough to stop testing, the most experienced software engineers researched about other experimental studies and had considered studies that used similar applications to the target systems in this case study based on the size, i.e., the executable lines of code. However, the degree of coverage should be one of the several criteria for deciding when testing is complete [55]. The target systems were from different domains and contained very different numbers of lines of code and classes, as, for example, can be observed from Table 3.3. For instance, the first OO release of iBATIS had 10,270 lines of code while MobileMedia first release had only 1,159. With such a large difference expressed in numbers, the same test exit criteria should not be appropriate to fit both applications. The purpose of exit criteria is to prevent a task from being considered completed when there are still outstanding parts of the task, which have not been finished [6]. Still, to consider structure-based testing for two so different software systems had also to consider how the concerns were implemented in AO releases and to apply a different strategy accordingly and that has not been done. The strategy to fulfill AO releases testing for both applications were established as the same from OO releases testing as explained in this section, not considering the details and risks of testing AO releases.

Table 3.3 Numbers from OO Releases – iBATIS and MobileMedia

	iBATIS				**MobileMedia**			
OO Release	**01**	**02**	**03**	**04**	**01**	**02**	**03**	**04**
Lines of Code	10,270	10,210	10,316	11,269	1,159	1,316	1,364	1,159
Classes	216	215	216	229	24	25	25	30

As it is further explained in the next section, MobileMedia had to follow a different test strategy than iBATIS, since JUnit framework did not support the micro edition (ME) extension for Java in which the application had been developed. The required documentation of the application had to be collected in order to prepare test cases with steps to execute in a mobile simulator. A test case (TC) includes not only input data, but also any condition and relevant procedure for the execution, and a way to determine whether the program has failed or passed the test [131]. A TC specification is a requirement that must be satisfied by a TC. Traditionally, software testing validates that specification was implemented as specified, but previous discussions on testing scope expand that definition to whether user needs are met as well [10].

The definition of the TCs for this case study has been set based on analysis of existing documentation about MobileMedia, features' behavior and talks to developers. All that was put on a template containing: description, pre-condition, steps and expected results, as shows model in Table 3.4 an example of Test Case description for the Favorites feature.

It has been stated an each expected result for every step to be followed on the test case. This would help finding where a defect would be coming from; for example, this test case may be blocked when step 2 cannot be executed. Hence, the test is not even allowed to push further and check whether it fails or not, since step 3 is not even reached. A test case would only fail when all the steps were executed and the results were not the same as the ones expressed on the expected results. When a test case needed to be stopped in the middle of its execution, i.e., step 2 blocks the rest of the execution, then it was not a failure; it was blocked until it was possible to go further.

Table 3.4 Test case example

Description	Set a photo as favorite.
Pre-conditions	1. Application must be launched; 2.There must a photo available and stored on phone's images folder.
Steps	1. The user creates a new album and inserts photo in it with correct label and path; 2. Set an item as favorite; 3. View Favorites.
Expected Results	1. Items must be inserted properly; 2. Item should be set flag Favorite as true; 3. The item just added as Favorite should be shown.

This case study has considered the testing of four releases of Object-Oriented MobileMedia software product line (R1, R2, R3 and R4). Due to optimization of test execution, the test cases have been executed in a manner combining one test case to the other, in a way that test cases from related features have been executed at once, (not restarting the emulator, not restarting the application). For example, a test case to add a photo and a test case to delete a photo were ran as one together. Thus, the number of TCs executed for each release of the SPL has been optimized and reduced, compared to if they were ran separately, as shown in Table 3.5.

Table 3.5 Number of test cases executed on MobileMedia SPL

R1	**R2**	**R3**	**R4**
04	07	11	17

3.4.4 Tooling Support

As the case study was performed within academic field by MSc and PhD researchers, open source tools were the target tools to be considered for such. In case of iBATIS application, the common JUnit framework fit the desired actions to write unit tests, however for MM, the same framework did not support the micro edition (ME) extension for Java in which the application had been developed. Thus, a different tool needed to be used to execute the code from MobileMedia releases. Plus, when the unit tests had been written, a coverage tool support would also be necessary in order to evaluate which parts of the code the tests were or were not covering and finally assess the code coverage percentage.

As this dissertation is focused on MobileMedia testing, this subsection describes the strategy taken regarding the tool used for this application's testing. There are not many tools available for testing J2ME application, especially when it comes to open source tools. Nevertheless, there was a tool that seemed to support the desired testing tasks. JaBUTi (Java Bytecode Understanding and Testing) [127] is a tool developed in academic field that provides different structure-based testing criteria to analyze coverage and other static metrics for further verifications on Java applications. The tool considers the analysis of exception dependent and exception independent point of view for each criteria, so that makes a total of eight different techniques derived from structure-based criteria are implemented by JaBUTi and can be applied to software testing; in which four are data flow and four are control flow criteria: all-nodes-exception-independent, all-nodes-exception-dependent, all-edges-exception-independent, all-edges-exception-dependent, all-uses-exception-independent, all-uses-exception-dependent, all-potential-uses-exception-dependent and all-potential-uses-exception-independent. The difference of JaBUTi from other testing tools is that it performs the static analysis directly on Java bytecode, not on Java source code. This allows the application of supported criteria to non-conventional software, such as mobile agents [128] and software components [129].

The main task executed by JaBUTi tool comprehends: class files (bytecode) instrumentation and coverage data collection during execution trace. However, to test a

J2ME application using a device or an emulator is not a trivial task due to storage space, memory and processing constraints. Thus, JaBUTi uses a client-server solution for this. The testing server runs in a desktop machine and is responsible to receive tracing information of client programs execution under an emulator or mobile device. This is possible due to a commands combination via DOS prompt. More details on how this is implemented can be obtained on tool's user's manual [127].

To enter a test case for this case study, one simply executed the midlet using instrumented classes by the tool. The interface of the application was, then, opened and ready to receive test data input, as observed on Figure 3.4. By the end of each test case execution, the execution trace was sent to the testing server and analyzed by JaBUTi. A new test case is considered every time the application is restarted, that is, closed and opened again. The execution of the application was possible due to the emulator from WTK (Sun Java Wireless Toolkit) [130].

The functionalities implemented by JaBUTi provide very useful information for empirical studies, which allow enabling different requirements and test case combinations, making feasible the use of incremental test strategies. In this case study, there were test cases created based on functional requirements that served as start point to evaluate coverage of MobileMedia Software Product Line testing. It was possible to evaluate critical parts of the code that needed to be covered more urgently in order to increase coverage. This is very valuable information in a test project, when time and resource restrictions are severe and what is more critical usually is required to be tested first. The tool also displays different colors representing weights that are associated to test requirements, indicating which test requirement, if covered, would increase coverage as much as possible. According to the considered criterion, the tester can work on testing code that has higher weights and consistently increase coverage. The stronger the color is, the heavier its weight represent, which corresponds to an important requirement to be covered. For better visualization of it, Figure 3.5 shows a print screen from JaBUTi tool showing a part of the bytecode for ImageUtil class from MobileMedia Object-Oriented Release 1. It shows the difference of the colors (weights) that are applied to the code, indicating how critical a specific part of the code may be for testing coverage purposes. Testing critical parts of the code, based on the indications provided by the tool, increases the coverage faster. The tool, however, does not consider complexity metrics or how critical a determined part of the code can be to the project, outside code perspective. The test engineer, based on his/her experience, may decide to cover other test requirements that have been set as highly complex and do not have high weight, though. The test engineer may use the information provided by the tool to help improving the testing activity and, sometimes, to find source to increase code coverage of a set of test cases in a faster way. For example, the test engineer is able to notice that a test case that exercises the part of the code marked with red will increase the coverage, regarding the test criteria set on the tool. On the other hand, test cases that cover the code marked with white or blue will not increase the code coverage very much. Therefore, based on the information provided by JaBUTi, it was possible to see the difference of coverage results regarding the applied criteria.

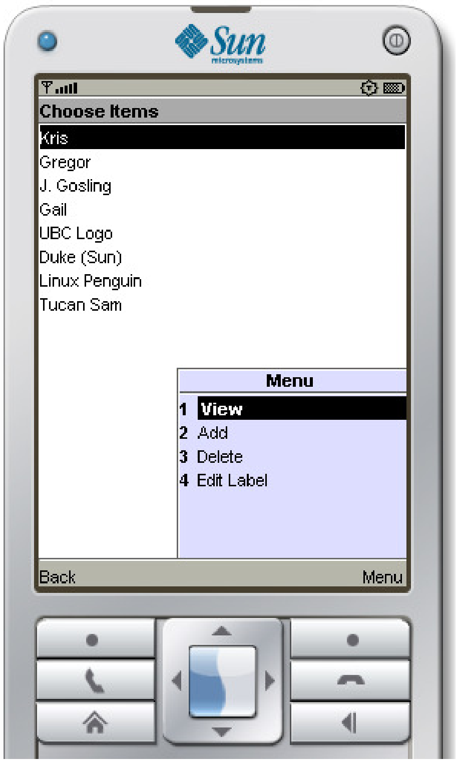
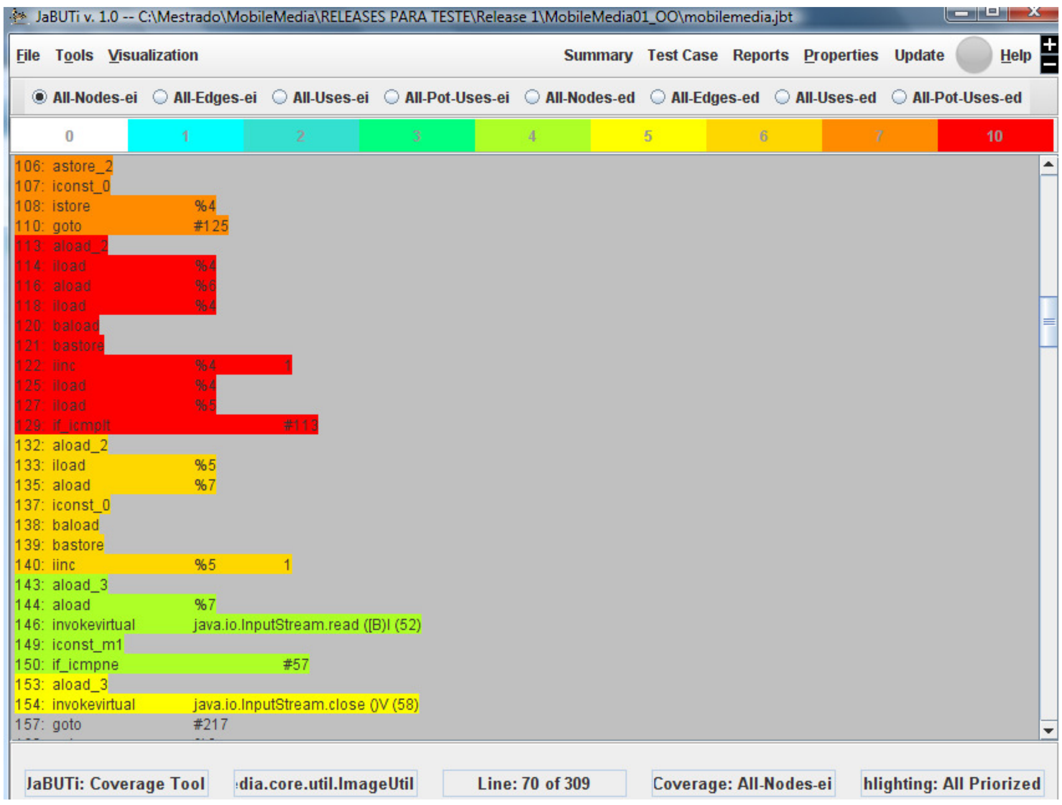


Figure 3.4 Interface of MobileMedia Release 2

Although JaBUTi supported all the functionalities introduced by the different OO versions of the product line, and seemed to fit the case study's needs for MobileMedia testing, it did not support the AO releases of MM. The criteria developed by the tool did not support the increased complexity combined from AspectJ language particularities and J2ME. JaBUTi developers have been contacted to try to solve such issue, which has been identified as a setup issue; however even using an AspectJ compiler, as it has been suggested, and properly setting libraries and necessary setup, the tool kept showing different kinds of errors. Due to the bad (or lack of) test planning that should have identified that such tool was / was not ready to support all the constraints the case study implicated, the testing activities were blocked. The tool developers and the case study managers were contacted to help finding a solution for such bottleneck. The days went by while the developers tried to find time in their planned activities to solve an issue from an unplanned activity. Therefore, by the end of the case study, the AO releases from MobileMedia were indeed not tested. Further details from such bottleneck and the implications of it are all described in Chapter 4 along with the definitions of the test core elements from the testware support.


JaBUTi v. 1.0 -- C:\Mestrado\MobileMedia\RELEASES PARA TESTE\Release 1\MobileMedia01_OO\mobilemedia.jbt
File
Tools
Visualization
Summary
Test Case
Reports
Properties
Update
Help
[✓ All-Nodes-ei] All-Edges-ei All-Uses-ei All-Pot-Uses-ei All-Nodes-ed All-Edges-ed All-Uses-ed All-Pot-Uses-ed
0 1 2 3 4 5 6 7 10

```
106:  astore_2
107:  iconst_0
108:  istore              %4
110:  goto                #125
113:  aload_2
114:  iload               %4
116:  aload               %6
118:  iload               %4
120:  baload
121:  bastore
122:  iinc                %4         1
125:  iload               %4
127:  iload               %5
129:  if_icmplt                     #113
132:  aload_2
133:  iload               %5
135:  aload               %7
137:  iconst_0
138:  baload
139:  bastore
140:  iinc                %5         1
143:  aload_3
144:  aload               %7
146:  invokevirtual       java.io.InputStream.read ([B)I (52)
149:  iconst_m1
150:  if_icmpne                     #57
153:  aload_3
154:  invokevirtual       java.io.InputStream.close ()V (58)
157:  goto                #217
```

JaBUTi: Coverage Tool | :dia.core.util.ImageUtil | Line: 70 of 309 | Coverage: All-Nodes-ei | hlighting: All Priorized

Figure 3.5 JaBUTi screen showing different colors for different weights associated to test requirements for bytecode of MobileMedia SPL Release 1

3.4.5 Bug Reporting

MobileMedia case study has taken place as a joint study along with iBATIS case study, being handled by different test people (Subsection 3.4.6). A spreadsheet template for reporting bugs found have been created and suggested to be used in both studies (Appendix B).

Although the template had important fields to be filled in case a bug was found, it has been designed mostly to specify code based testing bugs, i.e., bugs from the unit testing or component level. For example, if a robustness bug had been observed in a MobileMedia release test through the emulator (interface testing), it is possible that the tester could be unable to identify the root cause in the code for such robustness issue. And as MobileMedia testing activities were determined to be handled by the use of the emulator, even with the support of JaBUTi, it was common for the tester to be unable to locate the bugs' root cause. And since there were no developers available to support testers, the root cause analysis has, therefore, been left aside. While for iBATIS, the bug reporting template seemed to fit very properly, for the only required testing in that case was unit testing and there were developers supporting testers with bugs' root cause analysis.

3.4.6 Test People

MobileMedia case study was handled in academic field between two universities at two different sites. The two different sites were very distant, i.e., different time zone and thousands of kilometers from one to the other. Nevertheless, the communication was handled over the web and through e-mails and the roles have not been assigned formally. But it was possible to identify that there was one researcher managing the activities and the testers executing test tasks, while other performed support tasks, so the roles could have been formally assigned as:

- ✓ Test Manager – A (Site 1)

- ✓ iBATIS Tester / Test Manager – B (Site 1)
- ✓ Wild card role – C (Site 1)
- ✓ MobileMedia Tester – D (Site 2)

The names of the ones involved have been preserved, that is why letters are then here addressed in the case study roles.

Even still not addressing the roles and responsibilities specifically here just yet (it will be detailed in Section 4.6), it is possible to see that the responsible person, the test manager to administrate planning, process and issues was located at the same site as the tester from the other application from the case study, as well as the wild card role. The wild card role is non-specific role, undefined role of a person who was not a developer, not a tester, but has used and handled MobileMedia application within other contexts, so he was pointed out as a resource to contact in case of need, but he was not directly assigned to the case study. This role has been helpful throughout the study, but, as expected, he was not available every needed time, nor could support with every request of help. The MobileMedia tester was located at the other (far) site and did not participate on meetings or discussions regarding the planning of the activities of the case study. Even though it was poor, they have run some meetings to decide what to do. D has never joined any of these meetings and just received the duties without space to question. Such situation did not help when the activities started to take place and there was no role to report bugs to. Therefore, the bugs have been reported to the wild card role, who informed the test managers of them, but could not and did not have the time to look into them in order to fix them.

3.5 Final Considerations

This Chapter has as main purpose to introduce the case study that served as a base to create the testing problems that motivated the development of this dissertation. The case study was part of an experiment that had as one of the tasks to test a J2ME software product line (Java an AspectJ releases). The objective was evaluating and comparing AOP related faults and assess the error-proneness of AOP mechanisms. Therefore, the chapter has firstly introduced the context of the case study, providing information on the empirical studies regarding software maintenance, and within such context, it introduces the state of art of AO software maintenance, evolution and testing that is important to understand in which context the case study is applied.

The case study was inserted within a much broader context involving an AOSD testbed (Section 3.2), which is why details of how the testbed is composed are given in this chapter. Furthermore, a piece of the testbed composition, the benchmarking framework, is also detailed in Section 3.3, so one can better understand how the case study became a cause of a problem that the testware support in this dissertation suggests to solve. Details on how the case study has been set and developed have been explained in Section 3.4 and in its subsections; but, still, further explanations of the issues are also described in Chapter 4, in which the test elements composing the testware are cited and examples of the issues experienced in this case study are described along to give clear examples of the need of the testware support elements and provide confidence of the effectiveness of the testware support, within the context here described.

To conclude this chapter it is important to yet mention that, in a general way, testing tasks become even more critical when software varies more than usual. Considering complexity increase for software testing from the case study perspective described in Section 3.4, J2ME application shall be taken a closer look, since it has memory restrictions and capability particularities. The wide variety of Java technology-enabled devices implies in each device running a different implementation of CLDC (Connected Limited Device Configuration) and MIDP (Mobile Information Device Profile) [125]. Many developers who work in Java for micro edition products have been initially trained to develop desktop software. This stresses the little existing experience regarding mobile software, hence, mobile software testing [126].

Chapter 4

Testware Support

Lehman [92] affirmed that a software system must evolve in order to become more satisfactory. Evolving a software system is not only about adapting the source code. Software is multidimensional and the process behind the development and maintenance of it involves different relations among other artifacts, such as [93]: specifications, constraints, documentation, tests.

Some authors and studies defend the idea that regression testing [12, 20, 114, 119] and software testability analysis [14, 19] are enough to address the test directions to look up to when evolving or maintaining a software system. Others have aimed to provide a characterization schema for software testing techniques [56, 57], tools [58] and/or framework [59, 60, 136], where test engineers could define data models or be assisted when selecting a test technique. However, these studies are not general enough to be used as a whole test element within a software evolution benchmarking, nor are adequate to specifically support an AO software maintenance benchmarking framework, as the one described in Section 3.3. In Chapter 1, it has been mentioned that the idea of benchmarking has been used to support decisions in which software engineering techniques to use when dealing with its evolution. The proposed Benchmarking Framework (BF) presented in Section 3.3 intends to assist AO software engineering empirical studies by defining appropriate guidelines to assess the AO software characteristics.

In order to support maintenance decisions, a satisfactory testware support available eases software evolution by offering a safety net against unwanted change. Therefore, this chapter presents an important contribution of this dissertation: it introduces the testware support with test core elements to be taken into consideration and applied when maintaining or evolving a software system, within the context of AO software maintenance presented in Chapter 3. The test core elements address topics in test process flow in each subsection where they are listed as questions and/or items to purport the capabilities of a software project against testing process flow to conduct the project into the right test process flow. Such elements help supporting aspect-oriented software maintenance, providing the minimum testware information required within a software project or case study in order to make more reasonable decisions regarding its maintainability and evolution.

The testware support can be defined as follows:

Set of test elements and attributes that are able to identify specific test characteristics of different applications or software test projects, so that it is feasible to evaluate them under software maintainers' perspective and, therefore, address the most adequate test process flow to fit.

This dissertation defends the idea that the existence of the testware support in the BF is able to express an adequate test structure for the applications to be evaluated and compared with regard to the test attributes related to the used software development techniques, under the maintenance and evolution point of view. The extension of the BF considering the testware support and the evaluation of such insertion are presented in the next chapters.

The testware support provides:

- Quality assessment of the current state of the software system;
- Identification of test factors and/or risks in a software project;
- Decision making support on what regards to testing within the context of software maintenance or evolution process.

Thus, the testware elements will be able to guide researchers and professionals interested in testing perspectives when selecting, preparing or adapting a software system and its maintenance scenarios making them more adequate to reach specific objectives and determined goals of experiments or case studies related to AO software maintainability. The testware support can be used to identify test factors and guide the testing activities planning according to the identified test attributes.

The definition of the criteria and elements that compose the testware support is not an easy task, for they have to be general and specific at the same time. They should be general enough to identify test characteristics that are mostly common to be considered in every software project or case study addressing testing, within the context of AO software maintenance; and they should also be specific to comply with the test elements that may/may not be feasible in each different software project being evaluated. Yet another factor that makes this definition not an easy task is the quick appearance of new techniques, which hinders the criteria list to be statically defined; therefore, it should be dynamic to adapt imposed needed changes. The list of test criteria, known as the testware support provided with testware elements, has been structured to enable its extension in further studies in which it can be addressed and applied.

Provided with an appropriate testware support, a software project or case study is more likely to:

- Identify and address potential risks, so that they do not become a bottleneck;
- Prioritize tasks according to its importance and associated risk;
- Plan the necessary resource in advance, so that the tests are executed smoothly;
- Address the required test tasks and allocate enough resources to them;
- Define the minimum test criteria required for its goals and objectives.

The whole testware element is composed by different parts: the first topics presented in Section 4.1, in which the most general attributes are addressed in questions and tables; a test strategy described in Section 4.2 introducing the main topics to consider when planning a test strategy; Section 4.3 presents the test environment element addressing topics on this subject; relevant information regarding test tools to be taken into consideration in a software project is described in Section 4.4; the test people composing a test team and the roles' definitions are described in Section 4.5; Section 4.6 introduces Aspect-Oriented topics to be addressed within such context and, finally, in Section 4.7, the final considerations about the chapter are presented. Every section represents an element of the testware support. Figure 4.1 illustrates the testware support and its elements and represents a general view of how this structure is composed and organized, so that it is easier to understand the process of the test elements address; the figure also shows the benefits as a result of gathering and making use of the elements that can be achieved in a software project or case study.

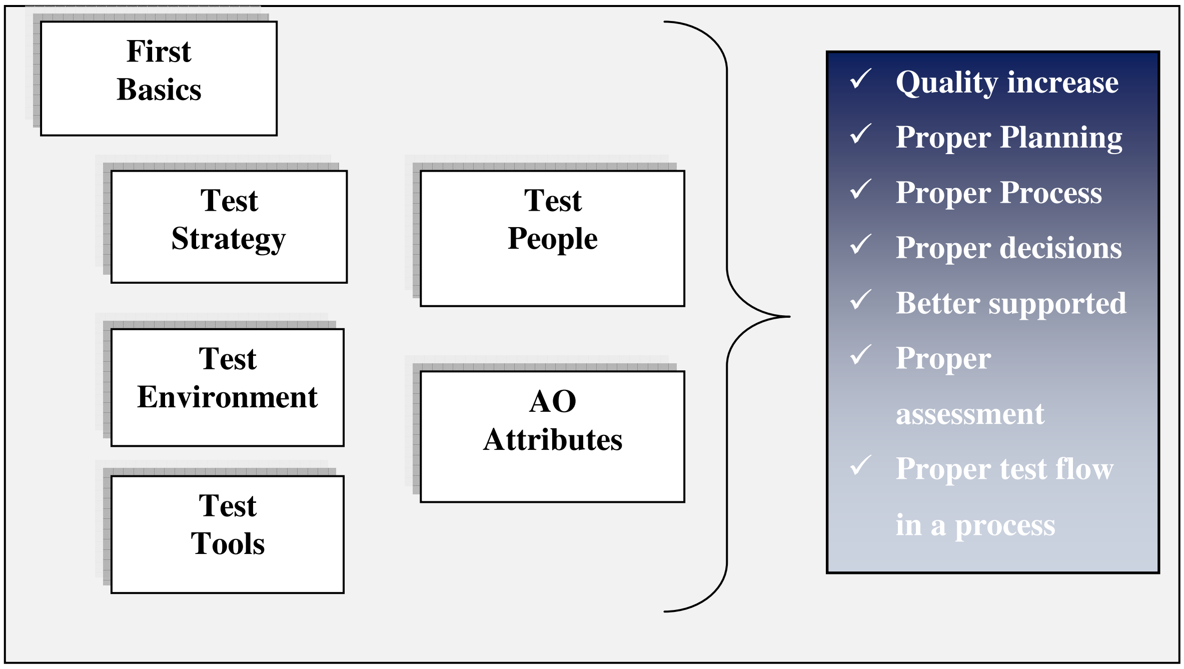


Figure 4.1 Testware Support

The elements are shown in boxes and the reason why they are not connected is because they do not compose a process, but they may support a process flow in which they are not dependable one on another. A case study can make use of some of the elements, but not all, according to its goals. However, the reason why the "First Basics" box is on top and not in line together with the others on purpose is because the first basics are quite indispensable to projects concerned in addressing or assessing test attributes, for they represent the very first test issues to filter and surface the basics of testing that may not be identified and / or identify the test needs. The other three elements below it are usually the most common topics to address in a software test project, so they behave smoothly at the same state one from the other inside the testware support, with no definitive priorities among them, but it is important to stress that they are equally relevant. The other two from the next column are test elements that usually come in second plan, despite addressing fundamental topics in software testing. Together, all these elements in a common and proper use have the power to provide successful outcome to a software project or case study as it is described in the box on the right.

4.1 First Basics

There are infinite possible ways to be used to test a piece of software. As mentioned earlier on this dissertation, testing activities are important to support software quality assurance, but when there are no test process flow definition in a software project, there are no defined strategies to systematically evaluate a target system. A definition of quality can change accordingly to user and project goals, so unless it has been previously set, the test engineers would be unable to reach it without knowing it.

Before the software system is being developed or before its maintenance or evolution, it needs to be defined the purpose of testing. Since there are different forms to test an application, it is required to know what is important to test, what is relevant, what may cause issues to the project if it fails, what may bring impact to the system. When these questions are answered properly, then it is possible to define the required test system – testware, test environment, and test process – to a specific project and determine test efforts.

This section lists an assessment questionnaire with questions to first indicate the capabilities of a software project against testing topics to conduct the project into the right test process flow. The questions are based in previous contributions [2, 3, 4, 5, 6, 7, 10, 55, 68, 70, 95, 108, 124, 131, 132, 146, 147] that have determined the specifications and concepts for test activities and test elements, have recommended requirements and specifications for identifying and resolving software problems in its life cycle and have had a well acceptance in the community to address test topics. The questions are categorized in different modules, as illustrated in Figure 4.2, and have the goal to address general test items to strict test topics in different scenes from the software project, such as: initial basics, general topics, items in test planning and test process.

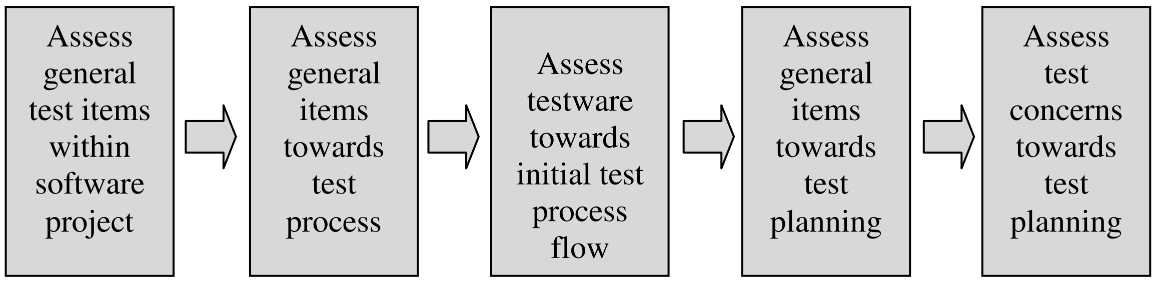

Assess general test items within software project
Assess general items towards test process
Assess testware towards initial test process flow
Assess general items towards test planning
Assess test concerns towards test planning

Figure 4.2 **Modules of testing first basics assessment questionnaire**

The questions are gathered in groups of five; for each one, a 'yes' or 'no' response can be applied. For a successful outcome of the testware support, the meanings of the answers have been defined as follows:

- Yes – There has been a smooth understanding of the item and the answer complies with definition and skill support, and any necessary measure, such as training, skills, experience or self-study supporting the test item in a positive sense.

- No – There have not been definitions, understanding or skill support to address such item, nor its concept or definition within a software project, case study or organization; or the definitions are incomplete in a negative sense that such item cannot be formally verified.

On Table 4.1, the questions address the test items or topics to consider when preparing for a software testing project or case study involving tests. In many questions, test people, i.e., the people composing the test team, such as tester and test manager are mentioned. The definitions of the test people and roles are detailed in Section 4.6.

Table 4.1 Test items to address in software project

1	Are the objectives and requirements defined?
2	Are the requirements testable?
3	Are there time and resources available allotted for development and testing?
4	Has the test process been defined?
5	Are the testers familiar with the development methodologies and the required testware to test them?

These five questions from Table 4.1 consider the first items to look ahead when starting a software project and testing is a demand. To identify the testing competencies into more details, different items need to be assessed as state the questions on Table 4.2.

Knowing the answer to these five questions enables the software project managers, stakeholders and testers to know how far the testing project can be on what regards to software testing initial planning, and how much further it has to go in order to have a satisfactory answer. For instance, if the answer to all these five questions is “no”, then there are many actions to take place and this project certainly will need a lot more time to reach a satisfactory testing process than one that has answered “yes” to at least half of them.

Table 4.2 Items to address initial test topics

1	Is there a policy regarding software testing?
2	Is there a software testing strategy?
3	Are there trained resources to allocate on software testing?
4	Are there available tools to support a testing strategy?
5	Has the test object been under a testing process before?

If the organization in which the software project is inserted already has a defined policy regarding software testing and has a testing strategy, then it is important to known whether such existing strategy embraces the necessary items to satisfy the use of a testing process. Hence, the questions on Table 4.3 have been created as general considerations to help identify and address the items of the software test process.

It is no use for an organization to have a test policy and for a test project to have a test strategy when there is no one following it or it does not provide enough information for a desired outcome because it is outdated or it does not fit the circumstances or many other possible reasons. The test process should be able to clearly identify what are the steps to take in order to make effective testing since the very beginning of the software project. From the definitions to plan test activities and available testware for test case execution and bug and results reporting: all regarding testing tasks must be stated within a feasible test process. When a project is able to positively answer the questions from Table 4.3, it means that there is a safe test procedure taking place; it may be time to evaluate details of them. On the other hand, if the testing process fails to provide knowledge of whether it is enough to surface satisfactory test results, then it needs to go through a whole new evaluation and restructuring process.

Table 4.3 General items to address test process

1	Do testers follow the test process to plan the testing phase, prepare test environment, design and execute test cases and report test results?
2	Does the test process cover all the necessary activities to perform effective testing?
3	Is the test process sufficient to adapt to different test strategies?
4	Are there risks defined and prioritized and potential dependencies able to be identified?
5	Are the project and test roles defined as well as schedule, resources and budget?

The first things to look at, in a software project, in order to know if the basic testware and test process flow are available are defined on Tables 4.1, 4.2 and 4.3. After answering these fifteen questions, a project manager is able to know what actions to take next. The following actions, of course, depend on the needs of the project and depend on how developed and suitable its test strategy is at the moment, therefore adaptations may be necessary. Still, it is possible to drive attention to the items addressed on Table 4.4 towards test planning, regardless the type of software under test.

Table 4.4 expresses the questions regarding general topics when test planning, i.e., more in a test management level rather than in a (lower) tester level. The answers from Table 4.4 will be of paramount importance to proceed to further details when planning software testing in a project. If, for instance, the test activities are scheduled out of synchronization with the development schedule, it may be difficult for testers to contact developers in case a bug needs deep investigation and repair. The test objectives are fundamental to know what is the purpose of testing a piece of software, what are the goals aimed to be achieved. To identify the constraints and risks that may impact the activities at some point of the project is also important for they have the power of putting the schedule behind or block test activities.

Table 4.4 General items to address test planning

1	Have the risks associated with the software under test been identified?
2	Have the test objectives been defined?
3	Have the test activities been scheduled according to development schedule?
4	Have the test constraints been identified?
5	Have the test metrics been defined?

In the case study presented in Section 3.4, for example, if the questionnaire presented above had been applied, many test issues that occurred in the MobileMedia testing activities could have been avoided, such as the planning in accordance with development schedule, for example. If the questions from Table 4.4 had been addressed during test planning, the risk of the tool not support every release from the software product line and the AO releases could have been detected and mitigated accordingly so that it would not become a bottleneck and block further testing in the project.

More detailed items on what regards testing topics when test planning can be found on what has been defined on Table 4.5.

Table 4.5 Test topics to address test planning

1	Have the entry/exit criteria been defined?
2	Have the test techniques to be used been defined?
3	Are the testers familiar with the test techniques to be used?
4	Has the number of necessary test cycles been defined? Has the test schedule considered all cycles?
5	Has the bug report and analysis system been defined?

Table 4.5 assesses capabilities of the test activities that need to be satisfied in order to proceed with a satisfactory test planning. The entry and exit criteria define when to start and stop testing. Such decisions rely on software project goals and test objectives. If the chosen techniques to be used in a software project are related to structure based techniques, then code coverage decisions are usually the best metric used to define when testing has reached a specific percentage that indicates sufficient coverage of branches, statements, etc.

In the case study presented in Section 3.4, the same test exit criterion was defined for both applications under test. Such decision turned out to be unsuccessful since the applications were different from each other in many aspects (such as size, domain) and the test techniques applied could not proceed. The test techniques needed to have been evaluated considering the variables and the risks that could affect the tests' performance, as it indeed happened, not only because the technique was not suitable for both applications under analysis but because it was not evaluated the constraints in such project nor if the test process covered a plan that had (not) identified such constraints.

Yet the test planning does not succeed if the testers are not familiar with the techniques to be used. Testers must be willing and prepared to familiarize themselves with the respective application domain and to acquire the necessary knowledge [55], however time may be a constraint and there might not be enough time to train testers. Therefore, an experienced tester plays a very important role, for he/she is responsible for the creation and maintenance of test specifications, must know appropriate test methods and executes the test cases according to the test schedule and test specification.

Yet not even the most expert would succeed in a project where many test cycles are necessary and the test schedule is insufficient for them.

In the case study from Section 3.4, MobileMedia was a software product line that was already developed when it was introduced to the case study. On the other hand, iBATIS had some releases but in the case study, some new extensions were developed as the tests had been performed, plus iBATIS team was all located at one site, while MobileMedia team had the tester located at a different far site from where the case study was being managed and there were no developers working at the same time as the tester, so that was a constraint that has not been considered either and has affected the performance of the activities, as it has been explained in Section 3.4 and will be mentioned in the next sections.

Bugs found must be reported to developers over a report system, whether a database with forms or a controlled system of spreadsheets and their versions. This system should be able to register bug's location, behavior, steps for reproduction and the most of available information so that the developers can reproduce it and quickly identify its root cause and provide fix. Such factor (addressed in Question number 5 from Table 4.5) has also not been well addressed in the case study, i.e., there was bug report template (Appendix B), but the bugs found in MobileMedia were reported to the project managers at a different site, but there was no developer taking care of such bugs, because there was no developer working for MobileMedia at that moment, as described in Subsection 3.4.6. Hence, there was no one responsible to receive, address and fix the bugs and work along with the tester on it.

When the test items addressed on Table 4.5 are understood and properly addressed in a software project, it means that the project is ready to proceed to start its testing tasks, such as test case design, execution, report and analysis. As of this point, a test manager (test people is discussed in Section 4.6) may be required to handle further details on the test strategy, as they get more complex, such as test environment, test tools, test case design. Test management is responsible for the administration of the test process, the test infrastructure, and testware [55]. Regular control is necessary to verify if planning and project progress are in line. This may result in the need for updates and adjustments to plans to keep the test process under control.

In the case study, as there had been no test planning, the manager did not do follow such responsibilities and was unable to help or solve the test issues that came along the way for he was not experienced in test process or projects, so the decisions and actions to take relied on the tester herself.

As illustrated in Figure 2.1, testing can and should occur throughout the phases of a project. The tables presented in this section address the test basics that should be considered in a project scope so that further details can be identified. Table 4.6 has gathered examples of test activities to be performed during a project's phases.

Table 4.6 Examples of test activities during the phases of a software project

Requirements phase activities • Determine test strategy • Determine adequacy of requirements • Generate functional test conditions
Design phase activities • Determine consistency of design with requirements • Determine adequacy of design • Generate structural and functional test conditions
Program phase activities • Determine consistency with design • Determine adequacy of implementation • Generate structural and functional test conditions for programs/units
Test phase activities • Determine adequacy of the test plan • Test application system
Operations phase activities • Place tested system into production
Maintenance phase activities • Modify and retest

This dissertation addresses further test topics, as explained in the next sections: test strategy, test environment, tools, test factors and risks, test people and the AO attributes.

4.2 Test Strategy

Since exhaustive testing is impossible [55], priorities must be set. Depending on the risks involved, different test techniques and test exit criteria must be specified when establishing a test strategy. Prioritizing tests leads the most critical software components to be tested first, when there are time or resource constraints, which is very common in software test projects.

The test strategy defines the test design techniques to be used. The test basis needs to be checked to see if all required documents are detailed and accurate enough to be able to derive the test techniques in agreement with the test strategy. In some cases, this check is done during test analysis only, but if it is the case that the documentation is already available before starting such phase, it can be done while developing the test strategy. In any case, the specification of the test object determines its expected behavior, and the tester (or test designer) uses it to derive the prerequisites and requirements of the test cases.

In a global view, the test strategy drives the definition and elaboration of the test plan addressing concerns such as the estimation of the test effort, organization and coordination of the different test levels. The quality attributes of the software under test and the sequence of activities that need to be performed are also all responsibilities associated to the test strategy. It should also be able to correlate each different software feature to be tested with methods for adequate testing to that specific feature.

In our case study, there was no priority set regarding which features should have been tested first in MobileMedia, nor it has been detected and identified that the AO releases from it needed to be prioritized. The test planning was informal and poor, without many important factors that should have been discussed before even starting test activities. There was no control of the test activities from the management side. The test manager was located in a different site and was much more focused on iBATIS tests, rather than in MobileMedia. MobileMedia tester needed to plan his/her own test tasks as the project was already happening along with no sequence definition of test execution, aside of the software product line statements and the features introduced in each release. The schedule was prepared week by week as the week as coming to an end and the next one coming needed to be planned, so it was an ad hoc schedule. Since it was a project within academia, the people involved were not 100% focused on this case study, this was also not taken into consideration and it has affected the schedule when, for example, there was a week coming and the tester had to be away for a conference or something else. When something like that occurred, the test tasks were simply on hold, regardless the impacts and dependencies from them.

Based on the experience from the case study and based on software testing literature, it is possible to define a Test Plan addressing the necessary aspects to be taken into consideration in more details. Such Test Plan is part of the test strategy, addressed as an element from the testware support. The Test Plan is described in Table 4.7

Table 4.7 Standard Test Plan

1. General Information 1.1 **Summary** – Summarize the functions of the software and the tests to be performed. 1.2 **Environment and Pretest Background** – Summarize the history of the project. Identify the user organization where the testing will be performed. Describe any prior testing and note results that may affect this testing. 1.3 **Test Objectives** – State the objectives to be accomplished by testing. 1.4 **Expected Behavior** – State the expected behavior for this kind of software. 1.5 **References** – List applicable references, such as previously published documents on the project, documentation concerning related projects.
2. Plan 2.1 **Software Description** – Provide a chart and briefly describe the inputs, outputs and functions of the software being tested as a frame of reference for test descriptions. 2.2 **Test Team** – State who is on the test team and their test assignment(s). 2.3 **Milestones** – List the locations, milestones events and dates for testing.

2.4 **Budgets** – List the funds allocated to test by task and checkpoint.

2.5 Testing

2.5.1 **Schedule** – Show the detailed schedule of dates and events for the testing at this location. Such events may include familiarization, training, data, as well as the volume and frequency of the input. Resources allocated for tests should be shown.

2.5.2 **Requirements** – State the resource requirement, including: equipment, software, personnel.

2.5.3 **Testing Materials** – List the materials needed for the test, such as system documentation, software to be tested, test inputs, test documentation, test tools.

2.5.4 **Test Training** – Describe the plan for providing training in the use of the software being tested. Specify the types of training, personnel to be trained, and the training staff.

3. Specifications

3.1 **Business Functions** – List the business functional requirement established by earlier documentation.

3.2 **Structural Functions** – List the detailed structured functions to be exercised during the overall test.

3.3 **Test/Function Relationships** – List the tests to be performed on the software and relate them to the functions in items 3.2.

3.4 **Test Progression** – Describe the manner in which progression is made from one test to another so that the entire test cycle is completed.

4. Methods and Constraints

4.1 **Methodology** – Describe the general method or strategy of the testing.

4.2 **Test Tools** – Specify the type of test tools to be used.

4.3 **Extent** – Indicate the extent of the testing, such as total or partial.

4.4 **Data Recording** – Discuss the method to be used for recording the test results and other information about the testing.

4.5 **Constraints** – Indicate anticipated limitations on the test due to test conditions, such as interfaces, equipment, personnel, data-bases.

5. Evaluation

5.1 **Criteria** – Describe the rules to be used to evaluate test results, such as range of data values used, combinations of input types used, maximum number of allowable interrupts or halts.

5.2 **Data Reduction** – Describe the techniques to be used for manipulating the test data into a form suitable for evaluation, such as manual or automated methods, to allow comparison of the results that should be produced to those that are produced.

The standard Test Plan presented in Table 4.7 can and should be extended and adapted to specific concerns from software projects or be generalized. In the case of the case study presented in Section 3.4, such planning would have enabled the early identification of constraints and would have forced the identification and analysis of important test factors such as methodology and tools. In case of AO specifications, item 3.2 should identify the structural functions and item 3.3 should be able to relate them with testing.

4.3 Test Environment

Project managers are responsible to help the test manager in creating an environment in which testing a piece of software is effective and efficient. The management controls all the attributes of the environment and approves the tools to use. Tooling support is further discussed in Section 4.4. This section addresses the importance of having an efficient test environment based on the experience learned from the case study presented in 3.4 and based on what can be learned from software testing literature.

Sometimes during testing in different levels (component, integration system), a test driver[6] is required when the testing deals with low level test objects that need developers' support, because to write test drivers, programming skills and knowledge of the component under test is necessary. This is why mostly the developers themselves do the component testing [5]. Also the same test drivers may be used in different testing levels. However a unit test, for instance, requires different configuration of the test environment than a production acceptance test.

Other times it is necessary to install software or hardware components in order to test the test objects and it can very complex to install and configure the test

[6] Test driver: a software component or test tool that replaces a program that takes care of the control and/or the calling of a component or system [5].

environment for a tester to handle. In such instances, there are two alternatives [94]: either having the necessary level of system administration support available, or having the expertise available on call from the project team, information systems, technical support, or another appropriate group.

Every test environment has different needs, depending on the software under test. A suitable test environment is required for testing a test object. Every program that should be present on the available hardware in order to run the software under test needs setup and it all represents a critical success factor for the software project.

In the case study presented in Section 3.4, which is where it was able to learn from, the environment became a bottleneck. First of all, there had been no definitions of it and the test team was spread out, i.e., as explained in Section 3.4, the team had been divided according to the assignment: the ones supposed to work with iBATIS and the ones supposed to work with MobileMedia. But the testing in MobileMedia was assigned to a person in a different site from the most of the rest of the team, besides the application had already been developed, the developers were not available to work with the testing team along a suitable test process. Still the test manager (from the different site) did not support the MobileMedia tester on test definitions and the necessary environment was not previously identified or settled. Second of all, the tool support (which is further discussed in Section 4.4) resulted in a serious bottleneck due to incompatibilities with the application, especially with the AO releases. It was not identified in advance that the application embraced two different paradigms to be considered when test planning – especially because there has been no planning, but there has been no discussion about it at all. When it was time to test AO releases, then it was detected that the test environment set for testing OO releases was insufficient. To stop a test project at this point and start looking for another alternative breaks all the rules of the project regarding people assignments, schedules, deadlines, deliveries, etc. Finally, if the test environment had been previously looked upon, analyzed and discussed, the chances of happening such a bottleneck would decrease and if it still did happen, it would be properly addressed and the risks (Section 4.5) would have been identified. Therefore, this dissertation provides Table 4.8 gathering the most important requirements with which a test environment must comply to guarantee reliable test execution.

Table 4.8 Requirements for environment

1	Manageable	A manageable environment is required to test the test object under the same conditions every time.
2	Flexible	A test environment must be easy to adapt. This may conflict with the previous requirement, but deciding which one takes precedence depends on the aim of the test and the phase of the test process.
3	Continuous	The test environment must be able to keep its availability even if there are disturbing situations in it.
4	Safe	The testing activities may deal with privileged data or private tools. Therefore, the test environment must be safe enough to protect its testware.
5	Centralized	The test environment should be, as much as possible, centralized in one test team and in a single physical space, preferably not far from developers.

The first and second requirements stated in Table 4.8 may conflict, but, as mentioned, to decide which one takes place will depend on test goals. For instance, adjustments may be necessary when analyzing defects or implementing a new version of the software system; hence, if this is done in a test environment of one project in which the impacts have been considered, flexibility should be the requirement to adopt. On the other hand, if this happens on a shared environment, manageability is preferred.

The consequences of a failure of a test environment should be limited in a software test project. The continuous requirement addresses the importance of making regular backups so that they can be restored, if required. The environment should be able to support different releases of the software under test.

In the case study from Section 3.4, the environment required different tools and specific configurations. Such needs should have been addressed as requirements in test environment configuration. To help identify what kind of test environment is the best fit in a software project, it needs to be identified first the test objectives (as previously discussed), the context and the type of the software system under test. In case of the case study, the context was software maintenance, so it should have been possible to analyze previous test activities applied in the software systems, but unfortunately that was not the case.

4.4 Test Tools

A tool can be defined as "anything that serves as a means to get something done." [10]. It is important to recognize that it must be determined first what that something is before acquiring a tool. There are many supporting tools in use for software testing. It is possible to distinguish between different tool classes depending on their intended use, such as, for instance, tools for test management and control, tools for test specification, tools for static, dynamic and nonfunctional testing [5].

Testware often involves one or more test tools. There is a vast list of kinds of tools and intended use but not all available tools are applied in a single project. The test manager should know available tool types in order to be able to decide if and when to use a tool efficiently in a project. It is important that tools are integrated into software tester's work processes. The use of tools should always be mandatory.

In this section a generic categorization of the tools used by testers is presented, but it does not discuss specific vendor tools. There are too many operating platforms and too many vendor tools to effectively identify and describe the availability of tools in this dissertation. Therefore, Table 4.9 presents a list with kinds of tools that should cover a wide range of test activities. Some techniques are manual, some are automated; some perform static tests, other dynamic; some evaluate the system structure, and others, the system function.

Table 4.9 List of test tools

1	Boundary value analysis	A method of dividing the software system into pieces (segments) so that testing can occur within the boundaries of those segments.
2	Capture /	A technique that enables the capture of data and results of testing,

	playback	and then play it back for future tests.
3	Cause-effect graphing	Attempts to show the effect of each test event processed. The purpose is to categorize tests by the effect that will occur as a result of testing.
4	Checklist	A series of probing questions designed to review a predetermined area or function.
5	Code comparison	Identifies differences between two versions of the same program.
6	Compiler-based analysis	Utilizes the diagnostics produced by a compiler or diagnostic routines added to a compiler to identify program defects during the compilation of the program.
7	Confirmation / examination	Verifies the correctness of many aspects of the system by contacting third parties, such as users, or examining a document to verify that it exists.
8	Control flow analysis	Requires the development of a graphic representation of a program to analyze the branch logic within the program to identify logic problems.
9	Correctness proof	Involves developing a set of statements or hypotheses that define the correctness of processing. These hypotheses are then tested to determine whether the application system performs processing in accordance with these statements.
10	Data dictionary	The documentation tool for recording data elements and the attributes of the data elements that can produce test data to validate the system's data edits.
11	Data flow analysis	A method of ensuring that the data used by the program has been properly defined, and that the defined data is properly used.
12	Defect management	Captures, administrates and evaluates incident reports.
13	Design-based functional testing	Recognizes that functions within a software system are necessary to support the requirements.
14	Design reviews	Ensures compliance to the design methodology of reviews conducted during the software development process.
15	Disaster test	A procedure that predetermines a disaster as a basis for testing the recovery process.
16	Error guessing	Uses the experience or judgment of people to predict what the most probable errors will be and then test to ensure that the software system can handle those conditions.
17	Flowchart	Graphically represents the software system and/or software flow in order to evaluate the completeness of the requirements, design, or program specifications.
18	Inspections	A highly structured step-by-step review of the deliverables produced by each phase of the software development life cycle in order to identify potential defects.
19	Instrumentatio n	The use of monitors and/or counters to determine the frequency with which predetermined events occur.
20	Mapping	A process that analyzes which parts of a software system are

		exercised during the test and how frequently each statement or routine in a piece of software is executed.
21	Modeling	A method of simulating the functioning of the software system and/or its environment to determine if the design specifications will achieve system objectives.
22	Peer review	A review process that uses peers to review that aspect of the software development life cycle with which they are most familiar.
23	Ratios / relationships	Quantitative analysis that enables testers to draw conclusions about some aspect of the software to validate the reasonableness of the software.
24	Risk matrix	Test the adequacy of controls through the identification of risks and the controls implemented in each part of the software system to reduce those risks to a level acceptable to the user.
25	Snapshot	A method of printing the status of computer memory at predetermined points during processing. Computer memory can be printed when specific instructions are executed or when data with specific attributes are processed.
26	Symbolic execution	Permits the testing of programs without test data. The symbolic execution of a program results in an expression that can be used to evaluate the completeness of the programming logic.
27	System logs	Uses information collected during the operation of a computer system to analyze how well the system performed.
28	Test data	System transactions that are created for the purpose of testing the software system.
29	Test data generator	Software systems that can be used to automatically generate test data for test purposes. Frequently, these generators require only parameters of the data element values in order to generate large amount of test transactions.
30	Test scripts	A sequential series of actions that a user of an automated system would enter to validate the correctness of software processing.
31	Tracing	A representation of the paths followed by computer programs as they process data or the paths followed in a database to locate one or more pieces of data used to produce a logical record for processing.
32	Use cases	Test transactions that focus on how users will use the software in an operational environment.
33	Walkthroughs	A process that asks the programmer or analyst to explain the software system to a test team, typically by using a simulation of the execution of the software system. The objective of the walkthrough is to provide a basis for questioning by the test team to identify defects.

Many of these tools have not been widely used [10] due to high cost of their use and required specialized use. Many represent the state of the art and are in areas where research is continuing. As better tools are developed for testing during the requirements and design phases of software testing, an increase in automatic analysis is possible. In addition, more sophisticated analysis tools are being applied to the code during construction. More complete control and automation of the actual execution of tests, both in assistance in generating the test cases and in the management of the testing process and result, are also taking place.

An integral part of this process is the selection of the appropriate testing tool. Table 4.10 lists the steps involved in selecting the proper testing tool.

Table 4.10 Steps to consider when selecting a test tool

1	Match the tool to its use
2	Select a tool appropriate to its life cycle phase
3	Match the tool to the tester's skill level
4	Select an affordable tool

The better a tool is suited to accomplish its task, the more efficient the test process will be. The wrong tool not only decreases the efficiency of testing, but it may not permit testers to achieve their objectives. The objective for using a tool should be integrated into the process in which the tool is to be incorporated.

A test tool introduces an automated instrument that provides support to one or more test activities. The introduction of test tools can have various advantages. Table 4.11 presents some of the advantages described in previous research [96] which do not apply to the entire list of tool types presented in Table 4.9.

In one of the advantages, the 'higher testing quality', the human factor is mentioned. It is certain that a test task that can be automated and run faster with the use of a tool will certainly decrease the risk of errors that may be inserted by humans. However, it is important to point out that test automation using a test tool is once programmed by a human and if an error is inserted then, the following activities will be impacted. The point here is the automation of the tasks that can be repetitive and the tool can increase the quality of these tasks by doing it much faster while the tester can do something else, increasing the quality of the tasks and the productivity of the testers.

Table 4.11 Advantages of using test tools

Increase of productivity	By using a test tool for routine test work, the tester has much more time for other tasks. In particular, with tools for automated test execution, a sizeable quantity of tests can be executed 'unmonitored', which means that much more thorough testing can be done in different fields and the test environment can be used more efficiently.
Higher testing quality	The use of test tools to support a structured test approach is an emphatic step towards a higher testing quality and quality software. The reason is the consistent execution of an activity that is supported by the tool. A tool imposes a standard work method, eliminating the human factor.
Work satisfaction	The execution of routine tasks can be boring. When repetitive tasks can be automated, it increases the work enjoyment of the test team in addition to increasing reliability.
Extension of test options	Some tests cannot be simulated fully when done manually. One example is the execution of stress tests. The deployment of test tools is virtually indispensible here.

The test tools can trace defects that are very difficult to detect manually. In the case study, it was practically impossible to go through the tests without a tool. Since the application was already full developed and there would be no support from development side, unit testing was discarded since the tester was not familiar with the application and since it was complex and large, it would take a very long time for the familiarization process. Plus, to unit test such application in such conditions by a single tester does not sound like a successful test task. Therefore, the tests with the use of an emulator provided by the use of JaBUTi tool were possible to get started. They did not finish, but they did indeed start. One of the reasons of not finishing and not being able to go further, was when the project needed to start testing the AO releases of MobileMedia. The tool did not support AspectJ implementations, although the tool developers affirmed it was just a matter of configuration, they themselves were also not able to make such configuration, therefore the testing could not proceed due to the tool support. In what regards to AOP, the number of available tools is strongly reduced, especially when it comes to open source tools. Thus, the scope became very limited. Such limitation should have been identified during test planning when analyzing tooling support for such application and its particularities.

The activities in the test process supported by a test tool and how this will be set up depend on the tool policy pursued in the organization. In the case study, as it was an academic study, there was no budget to buy licensed tools available on the market that could help with our issues; hence it was necessary to pursue an open source tool that would handle the different variables managed in the application, such as support to J2ME, AO software and the configuration to set up JaBUTi with the test server and other requirements as described in Section 3.4.

Typically, software testing must be achieved within a budget or time span. The choice for the test tool to be used in a software project relies on different reasons, but some general questions should not be left aside. Based on this, Table 4.12 presents questions addressing aspects to consider when selecting the best tool according to the software project's conditions and needs. Such questioning would have helped in the case study from Section 3.4 identify points of concerns that should have been detected in advance before becoming an issue later on.

With the steps listed in Table 4.10 and the questions addressed in Table 4.12, the test manager should be able to clarify issues that could not be visible at that point of the project, if the questioning would not have been made. Therefore, he/she is able to be more prepared if the questions are addressed and answered properly according to his/her software project conditions and needs to face what could become a difficult issue to solve or a bottleneck later on when the test activities are in a tight schedule.

In the case study, if it has been early detected that the tool needed adjustments to handle the AO MobileMedia releases, a lot of time would certainly have been saved, for when the issue had been observed, many researchers and tool supporters needed to be contacted to help solving the problem and providing the necessary tool updates, but as previously explained, that was not possible to finish with a successful outcome.

Table 4.12 Items addressing concerns when selecting test tools

1	Are test tools selected in a logical manner?
2	Can testers use test tools only after they have received adequate training in how to use them?
3	Is the tool usage specified in the test plan? Is there a tool manual?

4	Has a process for obtaining assistance in using test tools been established, and does it provide testers with the needed instructional information?
5	Have the dependencies to use the tool been identified and mitigated in order not to become a bottleneck?

Test tools, by nature, cannot solve process-related problems and they do not work by themselves. Thus, it is important to address the tool expertise levels of the ones involved to use them. Based on the definitions of previous study [55] and previous discussions about it [5, 6, 10, 95, 132], the range of tool expertise used in this dissertation in this matter is explained below:

- High: The tool master. He has cutting-edge expertise regarding the tool usage and is able to provide support.
- Medium: The tool integrator. His method is methodical and deliberate. He can plan, organize, and document the tool's introduction into more details.
- Low: The tool beginner. He is convinced of the tool and is able to recognize potential problems and positive aspects of the tool's operational use.

4.5 Test People

To build software tester competency it is necessary to gather different aspects and characteristics that represent in the end test skills. In most environments, the more experience a tester has, the less a degree matters; in others, the degree is critical regardless of experience [132]. The "perfect" member of a test team has excellent knowledge and comprehensive experience in several domains, but only very rarely people like that are found. In the context of this dissertation, regardless the degree of the people involved in testing activities, it is important that the test team member possess excellent organizational skills and extraordinary attention to detail, among other things. Below are some important character traits for a tester [55]:

- The ability to familiarize himself quickly with complex domains and applications;
- The ability to detect defects;
- The ability to cope with and voice criticism adequately;
- The ability to distinguish essentials from nonessentials and the courage to leave out what is less important;
- Discipline, exactitude, patience, perseverance, frustration tolerance, determination;
- The ability to work in a team and ability to communicate.

Although the ability to work in a team and to communicate are listed last, both qualities are highly important. Because testing is a teamwork, only those who are able to work in a team and communicate with colleagues and customers alike will have lasting success in their work as testers [55]. Based on what the author from the previous sentence has stated, the case study from Section 3.4 had low chances to be successful then, for testing did not mean teamwork, and the testers available were not experienced enough to be assigned to the responsibility of testing such complex software product line, especially without the support of developers or a test process. The team roles from the project have not been assigned and the "test manager" was not well experienced to exercise such role. Table 4.14 lists the roles and responsibilities that should be exercised within a software project involving test tasks.

Table 4.13 Test Team Roles

Test manager	The test manager leads the test team. He is responsible for the creation of the test schedule and its technical and on-time implementation. He reports test status and test results to the project manager or leader. The test manager is experienced in test planning, test control, and test process. He has knowledge and practical experience in general methods of software testing.
Test designer	The test designer is responsible for the creation and maintenance of test specifications. His role involves identifying the appropriate test methods and the definition of a suitable test environment. He supports the test manager in the creation of the test plan and test schedule. He requires know-how in the areas of software testing, test specification techniques, and general software engineering. He must be able to familiarize

	himself quickly with complex application domains, requirements documents, functional specifications and system prototypes. Furthermore, he must be able to comprehend the function and expected behavior of the system under test. Using this information, he derives appropriate test cases and documents them in such a way that they are fully traceable.
Test administrator	The test administrator is responsible for the installation, operation and maintenance of the test environment. Among other responsibilities, this involves installing and setting up the system software (operating systems, database systems, application server), installing and configuring the test object, installing and setting up the test tools, and creating, managing and restoring system configurations.
Tester	The tester is responsible for the execution of the tests and the documentation of the test results. He executes the test cases according to the test schedule and test specification. If he notices a deviation from the expected system behavior, he writes an incident report. The necessary qualifications for this job are test fundamentals, IT basics, ability to operate applied test tools, and a basic understanding of the test objects.
Experts	Their duty is to support the “core” roles mentioned above in technically sophisticated matters or in problem solving.

Ideally, a specially trained member of the team exercises each of these roles. If in smaller teams several roles must be combined into one, the following combinations are best suited: test manager/test designer, test designer/tester, tester/test administrator [55].

The term “tester” is also used as a generic term for all of the roles listed in Table 4.14, as it has been mentioned in the context of this dissertation.

In the case study presented in Section 3.4, no test team roles have been identified. The terms used in this dissertation regarding “test manager” and “tester” are related to the experienced researcher responsible for manage the case study on what regards to goals and decisions, in spite of the lack of effectiveness regarding software testing, and the tester is related to the person who was responsible to perform designer and administrator tasks. Thus, it is easy no notice that the case study was not well structured with the proper test roles to reach a successful test result.

The test roles need to be well identified in order to address the responsibilities according to the knowledge of the ones involved. Table 4.15 provides the basics of software testing principles and tasks to be matched as competencies from the test team. Such practice may help identify how capable and competent the test team is and, therefore, provide the expected results from activities it is able to perform.

Table 4.14 Software testing principles and tasks against competency

		Fully competent	Partially competent	Not competent
1	**Testing techniques** Understanding of the various approaches used in testing and the methods for designing and conducting tests.			
2	**Levels of testing** Identifying testing levels.			
3	**Testing different types of software** The changes in the approach to testing when testing different development approaches.			
4	**Vocabulary** The technical terms to describe various test techniques, tools, principles, concepts and activities.			
5	**Test process** An overview of the processes that testers use to perform a specific test activity, policies, standards, procedures, tools.			
6	**Test planning** Assessing requirements, design, execution, reports, risks, test methods, environment, schedule, objectives, criteria, test scope, test team.			

4.6 Aspect-Oriented Attributes

The previous sections have identified the concerns with regard to software testing to build the testware support addressing different test items that together help defining what is relevant and may cause issues to a software maintenance project. The testware support has also received inputs and contributions from the case study presented in Section 3.4 and its lessons learned explained throughout every aspect from the previous sections. The case study serves as a basis and starting point to verify and compare the current situation of the existing testware and test direction (or lack of it) in and the desired assessment that should have taken place before the case study start, as discussed in Chapter 6.

This section lists attributes that are characterized to be specific and directly related to the main characteristics of software systems developed using AO techniques. These attributes have been addressed in the definitions of the BF [42] to assess specific AO characteristics related to code concerns. It has been added to the structure of the testware support, in case the testware support is used separately from the BF and the use of such attributes is necessary. As it is presented in Chapter 5, the testware support proposes to be an extension to the BF to consider test direction in such context, and the evaluation of it is presented in Chapter 6, but Chapter 6 also shows the evaluation of the single use of the testware support. Thus, it also addresses the AO attributes as Table 4.16 presents the characteristics derived from the definitions of the AO attributes defined in the BF [42] that need to be taken a closer look when the test object in a software test project regards an Aspect-Oriented software system.

As it has been explained in the definitions of these attributes [42], the Crosscutting Concern (CC) Classification attribute provides a classification of CC types according to different dimensions based on a representative subset of the aforementioned existing classifications. The first dimension of the BF classification categorizes the concern as being Functional or Non-functional. A functional concern relates to business functionality, whereas a non-functional concern relates to the quality of the services provided by the system (e.g. security, reliability, distribution, etc.). Both functional and non-functional concerns can further be classified as either being Homogeneous or Heterogeneous. A homogeneous concern extends program at multiple join points by adding the same code at each join point. A heterogeneous concern extends multiple join points but with different pieces of code at each join point. The final dimension of the classification identifies if a concern affects a single component or multiple components, by either being an Intra-Component or Inter- Component concern. In this case, the meaning of the term "component" depends on how one examines the system. It might pertain to a class, an aspect, a distributed object, a package or even an architecture-level component.

Table 4.16 AO attributes to address in software test project

Classification of crosscutting concerns	Functional / Non-functional
	Homogeneous / Heterogeneous
	Intra-component / Inter-component
Interaction and composition of crosscutting concerns	Invocation based
	Tangled code in component level
	Tangled code in operational level
	Overlapping
Crosscutting concerns scope	
AO languages constructions	

Concerns in a system may be composed in a variety of ways. These compositions cause interactions between concerns that may affect system maintainability. Therefore, it has been important to understand and classify these different types of composition. The Concern Composition attribute specifies the ways in which concerns can be combined, according to a classification employed in some well-known empirical studies [49, 148]. The simplest form of composition is Invocation-Based Composition. This arises if two concerns, C1 and C2, have no classes or aspects in common, i.e., they only communicate via method calls. Component-Level Interlacing occurs if multiple concerns have one or more components (classes or aspects) but no operation (i.e. method, advice, etc.) in common. As a result, the concerns are interlaced and tangled at the component level only. In contrast, Operation-Level Interlacing occurs if multiple concerns have one or more operations in common. In this case, concerns are interlaced at the operation level. Finally, Overlapping identifies points where multiple concerns share one or more statements, operations or components. In contrast to interlacing interactions, which have disjoint common parts, overlapping concerns share elements (i.e. an entire component operation or statement contributes to multiple concerns).

The Aspect Scope attribute identifies the stage at which a particular concern emerges in an application. For example, the exception handling concern has been observed to emerge at a number of different development stages (e.g. requirements, architecture, detailed design and implementation). Finally, it is important to identify the various language features used to implement a particular concern. The AO Language Constructs attribute identifies the language elements used to implement a crosscutting concern, such as intertype declarations, different kinds of pointcuts and advice, bindings [100], etc. This information may be crucial when deciding if an application should be considered or not because an application that leverages a large number of language constructs is likely to be a benchmark. Furthermore, these language constructs can also be classified into static (e.g. intertype declarations) or dynamic (e.e. cflow pointcut designator), according to the time when they are handled by the compiler (compile-time, load-time, run-time). This is useful, for instance, to evaluate the influence that certain language constructs have on specific application characteristics, such as product line variabilities.

The other general attributes addressed in the BF [42] (Subsection 5.1.2) are also important to consider when the software project or case study is inserted in the context of software maintenance and assessment of such characteristics, but the testware support can also be of help and use outside such context. It can be used, for instance, in general software development as well.

The criteria presented in Table 4.16 are discussed and evaluated into more details on the definition of the BF [42] and have been based on previous works and researches that had a good level of acceptance in software engineering, as previously explained. Every attribute represents the respective work that guided its definition. Thus, it is possible to summarize here that the classification of crosscutting concerns into different aspects is a much discussed subject in previous studies [42]. In order to exercise different resources from the AO languages it is necessary for the software system to have a broad and diverse set of crosscutting concerns and to analyze them under different perspectives. Thus, the classification is achieved in an orthogonal form, i.e., a single concern may be classified in each one of the categories defined on Table 4.16.

Chapter 5 presents the extension of such benchmarking framework addressing the AO software system's concerns to consider the insertion of the testware support. More details of the BF are there presented as well. The evaluation of such insertion is then presented in Chapter 6.

4.7 Final Considerations

The previous sections have presented the elements that compose the testware support, which is the main contribution of this dissertation. A satisfactory testware support eases software evolution by offering a safety net against unwanted change. As the test elements composing the testware are cited, examples of the issues experienced in the case study introduced in Section 3.4 are described along to give clear examples of the need of the testware support elements and provide confidence of the effectiveness of the testware support, within the context here described. The test core elements address test issues to assess the capabilities of a software project towards a proper testing direction to be conducted in it. Such elements have the aim to support software maintenance, providing the minimum testware information required within a software project or case study in order to make more reasonable decisions regarding its maintainability and evolution.

The testware support does not intend to substitute a test maturity analysis, supported by the TMM, explained in Section 2.3.3. It is important to understand and consider the difference between them in the context of this dissertation. The test maturity analysis verifies the test process in an organization and evaluates it in order to provide the required information so that the test process can evolve to a more mature model, provided with more test knowledge and efficient test process. The testware support, on the other hand, has the goal to analyze the testware elements in a software test project – including its process, but not evaluating its maturity – and provide the necessary testware information, i.e., test elements to be addressed in such project, as described in this chapter. Both initiatives can and should be used as complementary of each other in order to reach even more effective test knowledge and test procedures in a software project.

Chapter 5

Benchmarking Framework Extension

Chapter 3 has introduced the background of this dissertation's context. Section 3.3 has presented the Benchmarking Framework (BF) [42] provided with criteria and attributes to evaluate AO software maintainability. There is no driven test issues within the BF though. Therefore, this dissertation has been developed starting from the issues that a case study (Section 3.4) originated and motivated the research in order to build a testware support that would fit and help situations such as this case study and other studies or software project to address test issues within the context here presented. The context involves a testbed, as presented in Section 3.2, which has the goal of facilitating proponents of AO and non-AO approaches to compare and contrast their approaches with others in a more effective fashion. As part of the testbed composition, the BF is introduced to support AOSD techniques assessment. In fact, the performed studies had given more attention to maintainability assessment, but the BF can also be used in a wider sense.

As suggested on the definitions of the BF [42], such framework is the starting point to other studies making use of such structure or its criteria validation. Besides, further and more ambitious works can and should be extended from such framework, continuing its potential benefits. Therefore, this chapter introduces an extension derived from the creation of the testware support (Chapter 4) to be inserted in such structure in Section 5.2, but before, in Section 5.1, more details about the BF.

5.1 Original Definitions of the Benchmarking Framework

As it has been explained in Section 3.3, the Benchmarking Framework has been proposed in order to guide researchers and practitioners in selecting or adapting applications and their releases that best fit specific experimental maintainability goals. It can also be used to support the design, replication and evaluation of empirical studies. The effectiveness of the framework has been evaluated [42] from two different perspectives: (i) as a guide to determine whether an application is an appropriate benchmark; and (ii) as an aid to designers of studies on AO software maintainability.

The set of characteristics and attributes present in the BF have been chosen based on the existing experience in conducting AO software maintenance studies from the BF creator and related researchers. They have identified a recurring set of relevant attributes that were useful to assess the maintainability of AO techniques. As it has been explained in Section 3.3, the BF is structured according to two major components: product attributes and maintenance scenarios, which address complementary assessment issues relevant to AO software maintenance. And the usage process (Figure 3.2) presents the framework components and its workflow.

5.1.1 Process

The framework has a group of potential stakeholders and encompasses a process that transforms inputs into outputs; the latter may also serve as feedback to improve the set of framework attributes. Figure 3.2 shows a schematic representation of the framework workflow. The users or framework stakeholders have been classified into two categories: the designer of empirical studies on AO software maintainability, and the benchmark designer. The first group is interested in conducting a maintainability study involving one or more AO techniques. In this case, the input is the set of experimental requirements and the output is the initial configuration of the experiment. In contrast, the second group includes people who change or add new artifacts to the benchmark application. This group needs to analyze applications and maintenance scenarios to determine if they are suitable to conduct a variety of studies. The output of this process is one or more applications and change scenarios that are appropriate to benchmark AO techniques.

5.1.2 Attributes of AO Software Products

The several characteristics identified in the BF definitions study [42] regarding the AO software product (target system) are here presented. The attributes guide the various decisions on the design or assessment of candidate benchmark application for maintainability studies. It has been considered a wide range of possible characteristics that a candidate application might exhibit, such as its domain, the development techniques that were employed in its construction, and the types of the crosscutting concerns (CC) that appear in its implementation. It is important to stress that it is impossible to define a set of characteristics that applies to every possible empirical study or application domain. Therefore, the proposed list of application characteristics can and should be constantly evolved and extended to more closely meet the stakeholder's goals, according to the BF author. Nevertheless, the attributes described in this section were found to be relevant in many different empirical studies [42, 49, 51, 148] with different goals and targeting different systems.

In general, it has been stated that the stakeholders do not need to consider all the framework attributes. Instead, they should focus on a representative subset that covers their experimental objectives. To ease this identification process and improve the organization of the different framework attributes, the Product Attributes have been divided into two groups: General Attributes, which encompass common application characteristics, and AO Attributes (Section 4.7), which consist of specific characteristics of AO applications. Figure 5.1 [149] provides an overview of the product attributes discussed in the rest of this section.

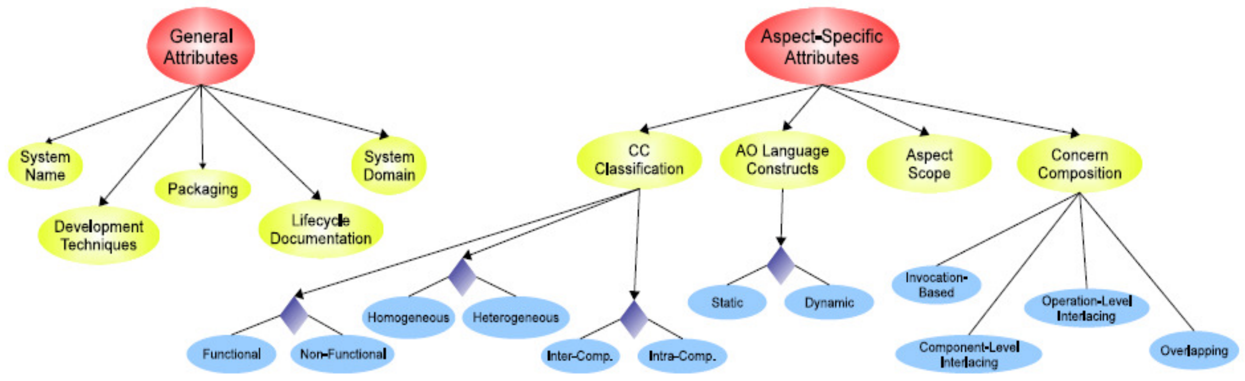


Figure 5.1 Schematic overview of the BF's product attributes

➢ **General Attributes**. A variety of general application attributes must be considered by framework users, such as **System Name, System Domain** and **Packaging**. The domain of a system is often an important aspect to consider, so as to ensure that the system exhibits some expected properties (e.g. embedded systems are often of a resource-constrained nature). Related to the System Domain, the Packaging attribute specifies the technologies and programming languages used in an application, in addition to development frameworks and target platforms. When considering an AO application for use in an empirical study, it is important to consider the development artifacts (e.g. requirements specification, architecture documentation, and the design diagrams) available to ensure the goals of the study can be realized. The software **Life-Cycle Documentation** attribute lists all the documentation artifacts available for assessment. In addition to listing the documentation, the **Development Techniques** that have been used to create the system (e.g. design patterns, application toolkits, etc.) should also be described. Different techniques can influence the quality of the final product and it is therefore important to take this into account when selecting an application. More details about the framework attributes are available in [42].

- **Aspect-Specific Attributes**. The definitions of these attributes are introduced in the definition of the BF [42], and they have also been presented in Section 4.7, when the testware support introduces the AO Attributes as an element to consider within its structure.

5.1.3 Maintenance Scenarios

The framework for empirical studies in software maintenance [42] has also included a catalogue of scenarios that are representative of real software changes. This is necessary to ensure the framework can evaluate a variety of recurring maintenance issues. These scenarios aim to support decisions of both benchmark designers and empirical study designers. An overview of the maintenance scenarios attributes of the BF is illustrated in Figure 5.2 [149] and the attributes are discussed as follows.

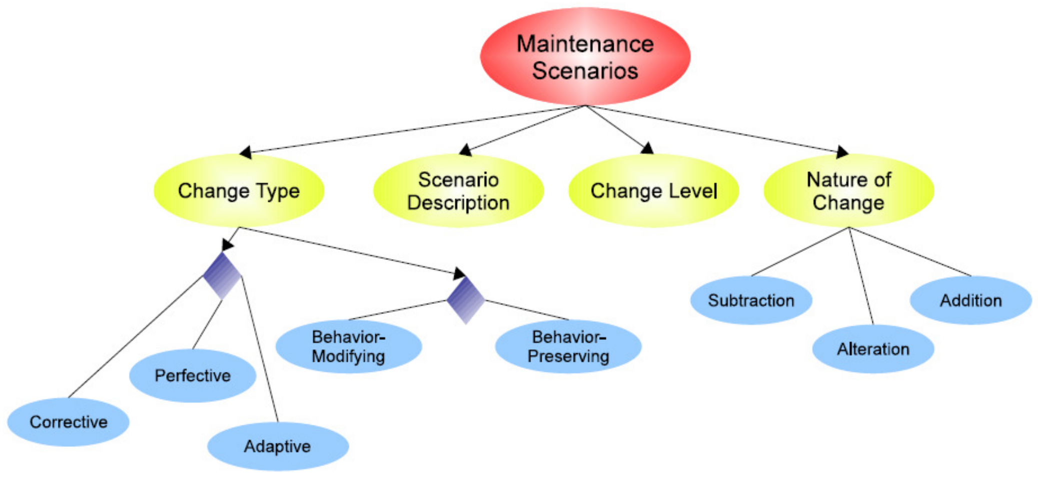


Figure 5.2 Schematic overview of the BF's maintenance scenarios attributes

The **Scenario Description** attribute provides the name and description of the scenario. It is also necessary to specify the **Change Type**. This involves classifying the change according to the effect it has on the base application. It has been considered that changes can be of one amongst three types: corrective, adaptive, or perfective [42]. Moreover, they can be behavior-modifying or behavior-preserving.

The **Nature of the Change** has also been considered, as it indicates how a given change modified a development artifact. A change can involve the addition (e.g. introduction of a new method), subtraction (deletion of an attribute), or alteration (to modify an existing element) of functionality. Finally, it has been stated that it is necessary to actually document the changes that are made. This involves specifying the **Changes at the Requirements Level, Changes at the Analysis** and **Design Level**, and Changes to the Implementation attributes. The changes performed in the requirements can influence artifacts at later development stages. Almost all changes made to the analysis and design artifacts are critical due to these artifacts being the core of software development activities. Analysis and design changes are likely to affect the rest of the application and can, consequently, impact software modularity.

5.2 The Test Attributes

Rather than a single component or element to be inserted into the BF process, as illustrated in Figure 3.2, the testware support is orthogonal. Nevertheless, to be inserted as such, the use of it can be driven by test attributes to be addressed, as detailed in test elements (Chapter 4). The resumed definitions of the attributes listed below include examples not necessarily regarding the same case study or application. Hence, the criteria presented are more representative.

As explained in Chapter 4, the definition of the criteria and elements that compose the testware support is not an easy task, since they have to be general and specific at the same time. They should be general enough to identify test characteristics that are mostly common to be considered in every software project or case study concerned with testing, and they should also be specific to comply with the test concerns that may/may not be feasible in each different software project being evaluated. Still the quick appearance of new techniques is another factor that makes this definition not an easy one. Hence, the listed criteria are not statically defined; they should be dynamic to adapt imposed needed changes.

The criteria have been defined based in previous studies [2, 3, 4, 5, 6, 7, 10, 55, 68, 70, 95, 108, 124, 131, 132, 146, 147] that have been able to identify and solve issues from software development and maintenance, addressing and discussing relevant test topics and directions into software projects. Test literature has determined the test basics, specifications and concepts for test activities and test elements to address in a software test project. Research has recommended requirements and specifications for identifying and resolving software problems in its life cycle. The studies here referenced that guided the definitions of the test elements in Chapter 4, and therefore the test attributes to consider into the BF, have had a well acceptance in the community to address test directions.

5.2.1 First Basics

As it has been discussed in Section 4.1, there are infinite possible ways to test a piece of software. The first basics regarding testing within a software project or case study are the first things that are essential to identify the existing test capabilities in it and they enable the detection of missing test concerns that should exist, according to the project goals. Therefore, there are important attributes that can surface such information, as described below.

- **Test policy** – identifies whether the software project is provided with a test policy with relevant test information objectively (YES/NO). The existence of a test policy eases the test activities, since the organization has previously defined test directions and has some experience with it, which can be useful for current project. Example:
  - ➢ *Test policy: YES.*
- **Test objective** – states the purpose of testing so that it is clear to every party involved. Example:
  - ➢ *Test objective: evaluate adequacy of functional requirements to software system behavior.*

- **Test strategy** – informs the test strategy to be followed in the project so that it is clear throughout the whole project development. Example:
  - ➢ *Test strategy: test the most critical features of the software system first.*
- **Test process** - identifies whether the software project is provided with a test process with relevant test information objectively (YES/NO). The test process should be able to guide the test activities. Example:
  - ➢ *Test process: NO.*
- **Test history** - identifies whether the application under test has been tested before and there is a test history available with relevant information that can be applied to the current test objective. An objective statement (YES/NO) is encouraged. Example:
  - ➢ *Test history: YES.*
- **Test criteria** – identifies the test criteria that have been selected as the ones to address. Example:
  - ➢ *Test criteria: structure-based criteria and code coverage.*
- **Functional and structural test conditions** – determines the major functional and structural test conditions that will guide testing activities. Such attributes have been derived from business and structural functions from item 3 (Specifications) in the Standard Test Plan (Table 4.7). These attributes can be used in a lower level test condition analysis to improve the accuracy of details to be concerned when test planning and addressing testing techniques. There are no list of possible answers to these attributes in Chapter 4, for they depend on the software under evaluation, so the examples here shown should not be understood as a limitation. Example:
  - ➢ *Functional test condition: interface testing, input domain testing.*

- ➢ *Structural test condition: data flow testing.*

- **Test environment** – describes the constraints and minimum requirements of environment to adequately functioning, which can be associated to the project's success. Example:

  - ➢ *Test environment: Two client-server machines configuration with WTK 2.5.2 installed.*

- **Test tools** – identifies the test tools necessary to perform required test activities properly. Example:

  - ➢ *Test tools: Test Link version 1.7.4.*

- **Test people** – identifies whether the software project is provided with necessary test personnel to perform the required test activities. An objective statement (YES/NO) is encouraged. Example:

  - ➢ *Test people: YES.*

5.2.2 Test Strategy

It is nearly impossible to thorough test a software system, for there are infinite combinations that can be applied towards test execution. Thus, relevant attributes need to be taken a closer look and assessed to build an adequate test strategy covering essential information to the ones involved.

- **Test plan** - identifies whether the software project is provided with a test plan addressing the necessary aspects to be taken into consideration. An objective statement (YES/NO) is encouraged. Example:

  - ➢ *Test plan: NO.*

- **Milestones** – Defines important and strategic locations and events throughout testing activities. The milestones will help identify when one test activity is completed or ready to begin. Example:
    - *Milestones: Full bug reporting at the end of testing of each release from software product line; new feature delivery every other week.*
- **Test schedule** – determines the test schedule for test activities assigning every test person to specific tasks. Example:
    - *Test schedule: Feature 1 – Tester A and B – From August 10th to September 1st.*
- **Test requirements** – states the resources available to assess testing activities against schedule and test objectives. Whatever requirement not stated in such attribute is not available within the software project. Example:
    - *Test requirements: specifications / feature model / personnel.*
- **Test techniques** – lists the test techniques to be applied in the testing activities. Example:
    - *Test techniques: equivalence partitioning and exploratory testing.*

5.2.3 Test Environment

Every software project has different needs for test environment, and it depends on the software under test. A suitable test environment is required and fundamental to smoothly test a test object. Every program that should be installed on the available hardware in order to run the software under test needs previous setup; the software systems and other tools necessary to continuous run the software under test: all is part of the test environment and represents a critical success factor for the software project.

The restrictions and constraints of testing environment are to be early identified so that it does not become a bottleneck in a software project or case study. The attributes are detailed below.

- **Requirements** – identifies the requirements a test environment must comply with to provide reliable test activities. From the definitions in Table 4.8, the requirements can be identified in a range of five different conditions. Example
    - ➢ *Requirements: manageable, safe and centralized.*
- **Dependencies** – describes the dependencies associated to the test environment, so that they can be addressed and not become a configuration issue or bottleneck. Example:
    - ➢ *Dependencies: the test environment relies on JDK1.5.0_09 and Apache server running and port 80 open.*

5.2.4 Test Tools

The fact that testing is a time-consuming and costly activity is no news in the community. Tools have the advantage to increase testers' productivity, when combined to a structured test approach. In a properly controlled process, tools can certainly add a lot of value to a software project or case study. With the increase of software complexity, tooling support is somewhat indispensable to testing activities. Therefore, it is important to address the proper concerns in order to choose and use the right ones.

The attributes related to test tools have the goal to identify the aspects that can contribute to a successful tool performance in a software project or case study, as described below.

- **Type of tool** – identifies the type of the tool according to Table 4.9. Example:
    - ➢ *Type of tool: control flow analysis and peer review.*

- **Tool name** – informs the name of the tool to be used. Example:
  - ➢ *Tool name: Selenium.*
- **Tool version** – informs the version of the tool to be used. Example:
  - ➢ *Tool version: 3.2 (beta).*
- **Tool support** – identifies whether there is tool support available, in case of need. Whether by developers support or manual, it is important to identify it so that it can be reached if necessary. Example:
  - ➢ *Tool support: user's manual and developers support at developers' site at Computing Department of University of Pernambuco.*
- **Tool expertise** – identifies the level of expertise of the test people involved in knowledge on the tool to be used. The tool expertise range is defined in Section 4.4, in which: high, medium, low or even none can be the answers, in a general way. Example:
  - ➢ *Tool expertise: medium.*
- **Tool dependencies** – states the dependencies that the tool is trustworthy to run. Such dependencies need to be addressed and prepared along with test environment dependencies and setup. Example:
  - ➢ *Tool dependencies: the tool only works with JBoss server.*

5.2.5 Test People

Section 4.5 has presented the general roles existing in a test team from a software test project. The attributes below shows what needs to be addressed in this regard within the testware support to be inserted in the BF.

- **Test roles** – identifies the test roles within a test team in a software project or case study. The roles are defined in Table 4.13. Each role needs to be addresses to one or more persons. Example:

  - ➢ *Test roles: Test manager – John Smith / Test designer – Mary Stuart and Joe Jacob / Test administrator – Joe Jacob / Tester – Joseph Ryan, Joan Marcs and John White.*

- **Training** – identifies whether is necessary or not to provide training to the test team. Example:

  - ➢ *Training: YES.*

- **Competency** – Identifies the competency of the test team according to the items introduced in Table 4.14. Example:

  - ➢ *Competency: Testing techniques – fully competent / Levels of testing – fully competent / Testing different types of testing – partially competent / Vocabulary – fully competent / Test process – fully competent / Test planning – fully competent*

5.3 Final Considerations

This chapter has presented the extension of the Benchmarking Framework [42] to consider the elements of the testware support defined here as attributes to be addressed in such structure to assess test directions in a software project or case study.

Table 5.1 presents the summarized attributes discussed in this chapter as general test attributes to be considered as test elements within the testware support – as introduced in Chapter 4 – to be inserted in the BF.

Such attributes associated with the general and AO related attributes (Section 4.6) presented and explained in the BF [42] compose the BF and testware support insertion, providing a more confident and thorough analysis of the software and the current software project or case study capabilities. As explained on the definitions of the BF [42], the main purposes of it are that the researcher or the practitioner is able to identify a representative application to his study or help plan and create experiments, within the context of software maintenance. In fact, the performed studies had given more attention to maintainability assessment, but the BF can also be used in a wider sense.

Table 5.1 **Test attributes**

First Basics	Test policy
	Test objective
	Test strategy
	Test process
	Test history
	Test criteria
	Functional and structural test conditions
	Test environment
	Test tools
	Test people
Test strategy	Test plan
	Milestones
	Test schedule
	Test requirements
	Test techniques
Test environment	Requirements
	Dependencies
Test tools	Type of tool
	Tool name
	Tool version
	Tool support
	Tool expertise
	Tool dependencies
Test people	Test roles
	Training
	Competency

Thus, provided with more information now regarding tests, the researcher or practitioner using the BF will be able to:

- Increase the scope of the study considering the testing process;
- With the increase of the scope, will be provided with the necessary testware support to address the test elements;
- Assess the most adequate test criteria and test elements accordingly;
- Have a broader view of the context.

The use of elements from the testware support can and should also be associated with the maintenance scenario attributes [42], instead of only addressing regression testing to them, as it could commonly be done. Also the elements from the testware support can and should also be used in different assessment, other than maintainability, for the BF can and should be used also in a wider sense.

Such extension here presented is the first study contribution extending the BF and the testware support is the first initiative regarding testing within the context of the testbed (Section 3.2) and the BF (Section 3.3). This initiative enriches the content of the BF for it proposes testware support to consider tests within a software project or case study. It also provides means to turn broader the view of researchers and practitioners who is interested in selecting, adapting or evolving/maintaining their software applications or planning or creating an experiment. Hence, it supports decision making and enables faster and more confident empirical evaluations and assessments in software engineering and software test engineering.

Chapter 6

Evaluation

The evaluation of test criteria is of great importance to software engineering for it allows the comparison of their features and benefits between each other and the analysis of examples and models to which they would best fit. Hence, it allows the establishment of the best test strategies according to the project, considering costs and efficacy [133]. Based on and motivated by such purposes, this chapter presents the evaluation of the proposed elements of the testware support and discusses the obtained results. The evaluation is divided into two different views: (i) the testware support evaluation and (ii) the Benchmarking Framework extension evaluation.

6.1 Testware Support Evaluation

The testware support evaluation has the goal to assess whether the proposed benefits can be achieved when the criteria defined are applied, i.e., the test elements are addressed as they should and whether this can be advantageous to a software project or case study. Throughout such evaluation, it is possible for a software project or case study to surface the existing test elements and detect gaps in process that need to be fulfilled, which may influence on decision making within the context of where it is inserted.

The testware support evaluation was performed by applying the concept associated to the test elements on two different applications. One of the applications, as mentioned in Section 3.4.2, was the starting point of the testware support study. It served as a basis in an empirical study to evaluate error proneness in AOP mechanisms, in which testing was required to be performed but there were no test directions or process to address test issues that should have been previously identified in such a case study in order to reach the expected results. After the creation of the testware support with test elements to support and address testing issues and directions in a software development process study, such as maintenance or evolution study, the same application has been used to evaluate its fit to the proposed testware support. The other application was used as a target to evaluation and analysis purposes only and was chosen based on recent empirical studies.

MobileMedia (MM) [96] and HealthWatcher (HW) [99] are the two different applications used on this evaluation. The evaluation of both applications has been performed by a Test Manager, who is usually the role responsible for this kind of analysis in a software project. The evaluation of the testware support is able to extend its focus on the testing perspective to a much broader context (as explained in Chapter 3) to help determining if the applications may be used in empirical studies to provide patterns on the comparison and evaluation (benchmarking) of AO techniques and to help identifying representative software systems. A testware support provided with test core elements to address testing issues has the goal to provide a safer evaluation of an application within such context, since it is considering now a whole new software process that should run closely to development, maintenance and evolution.

The applications used on this evaluation have been used in previous experiments [51, 96, 96, 98, 99], hence they had already been extensively analyzed to be used in different contexts. Still, two different research groups (SPG[7] and AOSE[8]) are focused on several studies to evaluate different aspects of such applications, such as their maintainability. Tables 6.1 and 6.2 show both applications in numbers of code elements, so it can be observed the size of each system here used.

[7] http://www.cin.ufpe.br/spg
[8] http://www.comp.lancs.ac.uk/computing/aod

The two applications are also composed by diverse characteristics, which make them able to represent AO software systems evolution. Both applications have been developed in Java and AspectJ languages and there are different versions available of them, in which each one of them exercises different kinds of changes that range from one version to another. The applications are from different domains and have been originally developed by different research groups. HW is a web information system, based in classical n-tier architecture model [134], while MM is a software product line, based in MVC architecture model [135], to manipulate data running in mobile devices. Besides the applications share the concept of maintainability and reuse through non-functional requirements control, they have different concepts of portability, performance, privacy, usability and others [96, 98, 99]. The applications have been projected and developed without any access to the information and goals proposed in this dissertation.

The criteria defined in the testware support through the existing test elements in it were used and observed with the goal of evaluating if they were indeed effective in the analysis and adaptation of the applications and their change scenarios, based on the objectives of the planned case study or software project. Thus, the evaluation was conducted based on the elements defined in Chapter 4. The objective in the evaluation study with both applications was to include the address of test issues to expand the scope when assessing applications' characteristics within software maintenance context, in order to provide more confidence and decision making support in such study.

Table 6.1 **Details of OO releases of MM and HW**

Release	Classes	Methods	Interfaces	Interface methods
MMOO_01	*24*	*122*	-	-
MMOO_02	*25*	*139*	-	-
MMOO_03	*25*	*142*	-	-
MMOO_04	*30*	*159*	-	-
HWOO_01	*88*	*370*	*11*	*73*
HWOO_02	*92*	*375*	*12*	*73*
HWOO_03	*104*	*510*	*12*	*73*
HWOO_04	*106*	*520*	*14*	*21*

Table 6.2 Details of AO releases of MM and HW

Release	Aspects	Pointcuts	Advices
MMAO-01	*4*	*22*	*20*
MMAO-02	*4*	*22*	*25*
MMAO-03	*7*	*35*	*33*
MMAO-04	*10*	*36*	*34*
HWAO-01	*11*	*2*	*16*
HWAO-02	*13*	*3*	*18*
HWAO-03	*17*	*3*	*33*
HWAO-04	*19*	*4*	*37*

6.1.1 MobileMedia

The evaluation of the testware support against MobileMedia application is here presented with the test elements assessed through the definitions from Chapter 4 and the data from the application and case study in which it was inserted, as previously explained. To ease identification of items and tables assessing test elements and test directions, the case study regarding the use of MobileMedia application is here called as "MobileMedia case study".

First Basics

The first basics are the first points of interest in what regards to testing that need to be addressed in order to identify existing test capabilities through test elements assessed and the initial test needs that have to be taken a closer look. Thus, Tables 6.3 and 6.4 have been applied to question the first concerns against MobileMedia case study capabilities.

Table 6.3 Evaluating test items to address in MobileMedia case study

1	Are the objectives and requirements defined? *No.*
2	Are the requirements testable? *No.*
3	Are there time and resources available allotted for development and testing? *No.*
4	Has the test process been defined? *No.*
5	Are the testers familiar with the development methodologies and the required testware to test them? *No.*

Table 6.4 Evaluating items to address initial test directions

1	Is there a policy regarding software testing? *No.*
2	Is there a software testing strategy? *Yes.*
3	Are there trained resources to allocate on software testing? *Yes.*
4	Are there available tools to support a testing strategy? *No.*
5	Has the test object been under a testing process before? *No.*

As it is possible to observe, the answers to the questions from the tables above regarding the evaluation of test items and initial test direction were not positive within the MobileMedia case study. Thus, such study would need to start defining the basics of testing in order to be able to positively assess the following items to address. The basics of testing would, then, be to define the questioned items, such as to define the test objective, requirements, resources, test process and so on. The next tables to continue addressing test items and test directions (Table 4.3, 4.3 and 4.5) will not be applied, since to go further with the evaluation, it is necessary to address the previous items first. The elements from the testware support assist on the definitions of the test strategy, as explained next.

Test strategy

Here, a desirable test plan for MobileMedia case study is shown in Table 6.5 as a proper plan to address the necessary aspects to consider in more details. The test people from the test team have had their names changes to preserve the identity of the ones involved.

Table 6.5 MobileMedia case study Test Plan

1. General Information

1.1 **Summary** – *The application is a software product line containing four different AO and OO releases, in which each release contains more complex features than its previous. For details on the features, please see the feature model. The tests to be performed can be code based testing and interface testing.*

1.2 **Environment and Pretest Background** – *Such application has never gone under testing before, it is a complex and large application. The organization in which the study is being developed is academic.*

1.3 **Test Objectives** – *Analyze code covered by testing.*

1.4 **Expected Behavior** – *The application should manipulate media files properly by adding, removing and editing files. For specific behavior at each feature level, please see the feature model available.*

1.5 **References** – *Feature model, features' description file.*

2. Plan

2.1 **Software Description** – *As this application is a software product line, a specific chart would be necessary for each release regarding the specific features' inputs from that release. For example*:

MobileMedia OO Release 2	
Sorting	→ *Try different orders*
Edit Photo Label	→ *Try different labels*
Others	→ *Reach exceptions and test exception handling.* → *Compilation without optional features.* → *Access main variables and test initialization*

2.2 **Test Team** – *Test manager – A*
Test designer – B
Tester – C, D and E

2.3 **Milestones** – *Full bug reporting at the end of testing of each release*
New feature delivery every other week as of July 20th until

December 20th, holidays excluded.

2.4 **Budgets** – *None.*

2.5 Testing

2.5.1 **Schedule**

	May	*Jun*	*Jul*	*Aug*	*Sep*	*Oct*	*Nov*	*Dec*
Plan	▨	▨	▨					
Training		▨	▨					
Tests			▨	▨	▨	▨	▨	
Analysis					▨	▨	▨	▨
Evaluation						▨	▨	▨

2.5.2 **Requirements** – *A desktop computer to each test person involved. Testers must have test tools installed and required software to run the test tools running as well. Testers must have access to features' specification, requirements documents and feature model documents.*

2.5.3 **Testing Materials** – *Four AO and OO versions of MobileMedia software product line, JaBUTi tool, code coverage tool, test documentation (test cases).*

2.5.4 **Test Training** – *Two experienced users of the application under test will provide training on how to use the application and how it has been incrementally developed release by release. Plus, the testers should also receive training on how to manipulate the test tools as defined in test schedule.*

3. Specifications

3.1 **Business Functions** – *Interface testing as per test case description and steps; input domain testing.*

3.2 **Structural Functions** – *Data flow testing. In Aspect-Oriented releases, first test base code and then aspectized code and observe whether a bug is observed when introducing the aspects in the base code.*

3.3 **Test/Function Relationships** – *For functional testing, a table is here attached* (Appendix A) *to address the functional test cases as per functional descriptions. For structural testing, unit test is applied along the code.*

4. Methods and Constraints

4.1 **Methodology** – *To follow functional and structure-based testing.*

4.2 **Test Tools** – *Capture/playback, code comparison, data flow analysis, defect management, walkthroughs.*

4.3 **Extent** – *At least 80% of the code need to be tested by the end of the testing activities.*

4.4 **Data Recording** – *A bug tracking system will be used to record bugs found and relevant data* (Appendix B).

4.5 **Constraints** – *The test server from JaBUTi tool has a limitation of 3MB of data* *transfer. There is only one JaBUTi developer available to solve eventual issues that may come up with the tool.*

5. Evaluation

5.1 **Criteria** – *The analysis is based on code coverage analysis and AOP related faults assessment.*

5.2 **Data Reduction** – *The functional tests performed throughout the application's interface s can and should make use of a automating tool, i.e., capture/replay tool that records the steps of the tests executed in one release that can be applied to the following releases, so that the tester does not have to do it manually all over again. Also code comparison tools can be useful to identify and compare the differences between two versions of the application, i.e., the piece of code introduced by the addition of an optional or alternative feature.*

Test environment

The test environment element needs to be addressed when test planning to consider relevant information regarding the environment where the testing should occur. Considering the requirements defined in Table 4.8 and what has been discussed in Section 4.3, in MobileMedia case study, the environment has different needs because it had to handle different releases in a product line, developed in different programming languages. Already considering tooling usage, but not too much, (test tool is discussed next), the environment needed to conform the tool support for both AspectJ and Java releases and the features implemented in each release. Besides, the test server required connection between JaBUTi tool and the mobile emulator, thus, the environment needed support from wireless toolkit application and client-server configuration management. Besides other tooling support – in case of the automating tool and others – the test environment required exclusive machines to the testers so that he/she could save, use and reuse information from previous releases tests. Still, a test administrator would be the right role to address the requirements for a manageable, continuous, safe and centralized test environment in this case study.

Test tools

Continuing the discussing regarding test environment, tooling support represent a great part of the configuration needs for a proper test environment. First, the concerns addressed in Table 4.12 are associated to MobileMedia case study's conditions when selecting test tools and Table 6.6 represents such evaluation.

Table 6.6 Items addressing issues when selecting test tools

1	Are test tools selected in a logical manner? *No.*
2	Can testers use test tools only after they have received adequate training in how to use them? *Yes.*
3	Is the tool usage specified in the test plan? Is there a tool manual? *Yes.*
4	Has a process for obtaining assistance in using test tools been established, and does it provide testers with the needed instructional information? *Yes.*
5	Have the dependencies to use the tool been identified and mitigated in order not to become a bottleneck? *Yes.*

In the ideal case, the case study would identify that JaBUTi tool does not attend the requirements to test AO releases as it does with OO releases. Hence, the tool developers are previously contacted in order to plan and deliver such an update on time. Also, the tool dependencies on the test server and the maximum of 3MB of data transfer between it and the emulator to gather test trace execution. Such dependency needs to be identified, so that long test executions that result in more than 3MB data transfer does not become an issue.

The automating tool to record test executions through the simulator interface also needs to be identified and provided with necessary information in order to use it. The use of it can and should reduce and optimize the executing time of manual tests – common tests of the base code, present in every release, no matter the added feature.

Test People

The test team is an essential subject within the case study to address people to appropriate test roles. The roles can be associated with testing competencies, as defined in Table 4.14, to identify how competent the members of the test team are when evaluated in testing principles, as shows Table 6.7 in the case for MobileMedia case study, in a general way. Such evaluation helps identifying and addressing the appropriate tasks to appropriate personnel, which is fundamental to a smooth flow in a case study like this.

Table 6.7 Evaluating software testing principles and tasks against competency in MobileMedia case study

		Fully competent	Partially competent	Not competent
1	**Testing techniques** Understanding of the various approaches used in testing and the methods for designing and conducting tests.	*Test designer*	*Test manager / Testers*	
2	**Levels of testing** Identifying testing levels.	*Test designer*	*Test manager / Testers*	
3	**Testing different types of software** The changes in the approach to testing when testing different development approaches.		*Test manager / Test designer / Testers*	
4	**Vocabulary** The technical terms to describe various test techniques, tools, principles, concepts and activities.		*Test designer / Testers*	*Test manager*
5	**Test process** An overview of the processes that testers use to perform a specific test activity, policies, standards, procedures, tools.	*Test designer*	*Test manager / Testers*	
6	**Test planning** Assessing requirements, design, execution, reports, risks, test methods, environment, schedule, objectives, criteria, test scope, test team.	*Test designer*	*Test manager*	*Testers*

Aspect-Oriented attributes

The criteria defined in the BF [42] list important AO attributes to be addressed. Table 6.8 shows the application of such in MobileMedia case study, presenting the existence of such attributes or not.

Table 6.8 Addressing AO attributes in MobileMedia

Classification of crosscutting concerns	**Functional**	*Yes*
	Non-functional	*Yes*
	Homogeneous	*Yes*
	Heterogeneous	*Yes*
	Intra-component	*Yes*
	Inter-component	*Yes*
Interaction and composition of crosscutting concerns	**Invocation based**	*Yes*
	Tangled code in component level	*Yes*
	Tangled code in operational level	*Yes*
	Overlapping	*Yes*

6.1.2 HealthWatcher

This subsection presents the evaluation of the testware support against HealthWatcher application with the test elements assessed through the definitions from Chapter 4. As mentioned, this application is a web information system with four different releases (implemented in OO and AO languages), in which each release contains applied changes to the previous one, as can be observed in Table 6.9 a summary of the changes described at a more developer's level. This application was used in this dissertation as a target to evaluation and analysis purposes only and was chosen based on recent empirical studies. The data here presented within the test elements being addressed flow along towards reaching the goal of identifying and assessing the test elements in the software system to provide the best test direction to fit software system's purposes. Hence, as of the context already here presented, the evaluation introduced next is inserted within the software maintainability evaluation benchmarking framework. To ease identification of items and tables assessing test elements and test directions, the study regarding the use of HealthWathcer application is here called as "HealthWatcher study".

Table 6.9 HealthWatcher releases' scenarios of changes

Release	**Description**
01	Factor out multiple Servlets to improve extensibility.
02	Ensure the complaint state cannot be updated once closed.
03	Encapsulate update operations to improve maintainability.
04	Improve the encapsulation of the distribution concern.

First Basics

Table 6.10 Evaluating test items to address in HealthWatcher study

1	Are the objectives and requirements defined? *Yes.*
2	Are the requirements testable? *Yes.*
3	Are there time and resources available allotted for development and testing? *Yes.*
4	Has the test process been defined? *No.*
5	Are the testers familiar with the development methodologies and the required testware to test them? *No.*

Table 6.11 Evaluating items to address initial test directions

1	Is there a policy regarding software testing? *No.*
2	Is there a software testing strategy? *Yes.*
3	Are there trained resources to allocate on software testing? *Yes.*
4	Are there available tools to support a testing strategy? *Yes.*
5	Has the test object been under a testing process before? *No.*

It is possible to observe that although HealthWatcher is provided with requirements and test strategy, there is no test process defined for it to follow. Hence, the items to assess test process from Table 4.3 are not applicable, since the answers to all five questions would be, therefore, negative. The next tables presented in the first concerns of the testware support concerns test planning, which is also still not applicable to HealthWatcher case at this point. Thus, the responsible manager would have to define into more details the existing strategy and assemble the test process and test planning. The tesware support assists such activities as follows.

Test strategy

Here, a desirable test plan for HealthWatcher study is shown in Table 6.12 as a proper plan to address the necessary aspects to consider in more details. The names of the members of the test team have been changed to preserve the identity of the ones involved.

Table 6.12 HealthWatcher study Test Plan

1. General Information

1.1 **Summary** – *The application is a web information system containing four different AO and OO releases, in which each release is the result of applying a number of heterogeneous types of changes to the previous one. The tests to be performed can be code based testing and testing through the browser.*

1.2 **Environment and Pretest Background** – *Such application has gone under unit testing before, but only at one release for a case study. The history of such testing is unclear. The organization in which the study is being developed is academic.*

1.3 **Test Objectives** – *Analyze code covered by testing and if behavior meet the specifications.*

1.4 **Expected Behavior** – *The purpose of the system is to collect and control the complaints and notifications, also providing important information to the people about the Health System. Allow exchange of information with the SSVS system (Sanitary Surveillance System). This exchange will firstly be only to query sanitary licenses and on another time – when SSVS have deployed the Complaint Control module – it will be given the automatic entry of the Sanitary Surveillance complaint nature.*

1.5 **References** – *HealthWatcher Use Cases document.*

2. Plan

2.1 **Software Description** – *The frame of reference for the functions to be tested is based on use cases' definitions.*

HW System Overview	
Query information	***Query Health Guide*** *The citizen might query:* • *Which health units take care of a specific specialty.* • *Which are the specialties of a health unit.* ***Query Diverse Information*** *The citizen might query:* ➢ *Information about the complaint made by the citizen:* ✓ *Complaint specification.* ✓ *Situation (OPENED, SUSPENDED, or*

		CLOSED). ✓ *Technical analysis.* ✓ *Analysis date.* ✓ *Employee that made the analysis.* ➢ *Information about diseases:* ✓ *Description.* ✓ *Symptoms.* ✓ *Duration.* ➢ *Inputs and pre-conditions:* ✓ *The data to be queried must be registered on the system* ➢ *Outputs and post-conditions:* ✓ *The query result to the citizen*	
	Specify complaint	***Animal Complaint – DVA*** • *Animals apprehension.* • *Control of vectors (rodents, scorpions, bats, etc.)* • *Diseases related to mosquitos (dengue, filariose).* • *Animals maltreatment.* ***Food Complaint - DVISA*** • *Cases where it is suspicious the ingestion of infected food.* ***Diverse Complaint - DVISA*** • *Cases related to several reasons, which are not mentioned above (restaurants with hygiene problems, leaking sewerage, suspicious water transporting trucks, etc.).* *The three kinds of complaints have the following information in common:* ✓ *Complaint data: description (mandatory) and observations (optional);* ✓ *Complainer data: name, street, complement, district, city, state/province, zip code, telephone number and e-mail. All these information are optional;* ✓ *Complaint state (mandatory), which might be:*	

	OPENED, SUSPENDED or CLOSED. In the event of a registration, its state must be OPENED; ✓ *The system must register the complaint registration date.* ***Inputs and pre-conditions****:* • *None* ***Outputs and post-conditions****:* • *The complaint saved on the system*
Update complaint	***Inputs and pre-conditions****:* • *The complaint must be registered and have the OPENED state.* • *Logged employee.* ***Outputs and post-conditions****:* • *Complaint updated and with state CLOSED.*

2.2 **Test Team** – *Test manager – A1*
Test administrator – B1
Tester – C1, C2 and C3

2.3 **Milestones** – *The milestones are the end of each main flow test activities of each functionality. At every milestone, test reports are delivered to assess test activities.*

2.4 **Budgets** – *None.*

2.5 Testing

2.5.1 **Schedule**

	Feb	*Mar*	*Apr*	*May*	*Jun*	*Jul*
Plan	▨	▨				
Training		▨				
Tests			▨	▨	▨	
Analysis					▨	▨
Evaluation					▨	▨

2.5.2 **Requirements** – *A desktop computer to each test person involved. Testers must have test tools installed and required software to run the test tools running as well. Testers must have access to use cases document specification to understand the specified behavior in order to test design.*

2.5.3 **Testing Materials** – *Four AO and OO versions of HealthWatcher system, code coverage tool, test documentation (test cases).*

2.5.4 **Test Training** – *Testers will be provided with training on unit testing and code coverage tools usage.*

3. Specifications

3.1 **Business Functions** – *Interface testing through browser as per test case description and steps; input/output testing.*
3.2 **Structural Functions** – *Unit testing. In Aspect-Oriented releases, first test base code and then aspectized code and observe whether a bug is observed when introducing the aspects in the base code.*
3.3 **Test/Function Relationships** – *For functional testing, a table is here attached* (Appendix C) *to example the functional test cases as per functional descriptions. For structural testing, unit test is applied along the code.*

4. Methods and Constraints
4.1 **Methodology** – *To follow functional and structure-based testing.*
4.2 **Test Tools** – *Code comparison, defect management.*
4.3 **Extent** – *At least 80% of the code need to be tested by the end of the testing activities.*
4.4 **Data Recording** – *A bug tracking system will be used to record bugs found and relevant data* (Appendix B).
4.5 **Constraints** – *The system should have an easy to use GUI, because it might be used by any person who has access to the internet. The system should have an on-line HELP to be consulted by any person that uses it.*

5. Evaluation
5.1 **Criteria** – *The analysis is based on code coverage analysis and AOP related faults assessment.*
5.2 **Data Reduction** – *Not Applicable.*

Test environment

As explained when addressing MM test environment element, it needs to be addressed when test planning to consider relevant information regarding the environment where the testing should occur. The system should be available 24 hours a day, 7 days a week. Because of the nature of not being a critical system, the system might stay off until any fault is fixed. Therefore, the system should use a security protocol to send data over the internet. To have access to the complaint registration features, the use must be allowed by the access control sub-system. Plus, the hardware and software to be used for the system to work has been previously defined by developers as follows:

- ✓ Software: one license for the Microsoft Windows for the workstation
- ✓ Hardware: One computer with: Pentium III processor, 256 MB of RAM memory, net card 3Com 10/100. This equipment shall be used by the attendant as a workstation.

After learning such conditions and based on the requirements presented in Table 4.8, the test environment for HW should, then, be manageable, safe and centralized, as the most important characteristics.

Test tools

Tooling support represents great importance in testing activities to make them become faster and more efficient, at most times. In case of HW study, the only tool support to address in testing activities is the code coverage tool. Thus, the concerns addressed in Table 4.12 are assessed to HW study's conditions when selecting test tools and Table 6.13 represents such evaluation.

Table 6.13 Items addressing issues when selecting test tools

1	Are test tools selected in a logical manner? *Yes.*
2	Can testers use test tools only after they have received adequate training in how to use them? *Yes.*
3	Is the tool usage specified in the test plan? Is there a tool manual? *Yes.*
4	Has a process for obtaining assistance in using test tools been established, and does it provide testers with the needed instructional information? *No.*
5	Have the dependencies to use the tool been identified and mitigated in order not to become a bottleneck? *Yes.*

In the HW study case, the only tool support is the code coverage tool to assess how much of the code has been covered by the unit tests. The tests through browser are executed manually. Therefore, in the test plan, it specifies the tool to be used. As it is an academic study and there is no budget available, the tool used is an open source tool free of cost. However, it has not been identified whether the tool supports AspectJ code to assess such code coverage, nor if the unit tests will need a different framework, rather than JUnit.

Test People

The test roles attributed to test people with appropriate test activities have to be associated against the individuals' competencies so that essential test tasks are identified and assigned to the most appropriate person. The roles and people assignment and evaluation for HW study are shown in Table 6.14, as per definitions in Section 4.6.

Table 6.14 Evaluating software testing principles and tasks against competency in HealthWatcher study

		Fully competent	Partially competent	Not competent
1	**Testing techniques** Understanding of the various approaches used in testing and the methods for designing and conducting tests.		*Test manager / Testers*	*Test admin.*
2	**Levels of testing** Identifying testing levels.		*Test manager / Testers*	*Test admin.*
3	**Testing different types of software** The changes in the approach to testing when testing different development approaches.		*Test manager / Test designer / Testers*	*Test admin.*
4	**Vocabulary** The technical terms to describe various test techniques, tools, principles, concepts and activities.		*Testers*	*Test manager / Test admin.*
5	**Test process** An overview of the processes that testers use to perform a specific test activity, policies, standards, procedures, tools.		*Test manager / Testers*	*Test admin.*
6	**Test planning** Assessing requirements, design, execution, reports, risks, test methods, environment, schedule, objectives, criteria, test scope, test team.		*Test manager*	*Testers / Test admin.*

Aspect-Oriented attributes

Important AO attributes are listed in the criteria defined in the BF [42]. Such attributes are the ones to be addressed in order to assess AO related required information in the study. Table 6.15 shows the application of such attributes in HealthWatcher study.

Table 6.15 Addressing AO attributes in HealthWatcher

Classification of crosscutting concerns	**Functional**	*No*
	Non-functional	*Yes*
	Homogeneous	*Yes*
	Heterogeneous	*Yes*
	Intra-component	*Yes*
	Inter-component	*Yes*
Interaction and composition of crosscutting concerns	**Invocation based**	*Yes*
	Tangled code in component level	*Yes*
	Tangled code in operational level	*Yes*
	Overlapping	*Yes*

6.1.3 Discussion over Testware Support Evaluation

Subsections 6.1.1 and 6.1.2 have presented the use and application of the proposed testware support against two different software systems. Such application provides more consistency on the discussion of the evaluation. First, it is important to remember that the testware support provides the necessary elements and test issues to address in order to lay out a proper test direction in a software project. The elements provided in Chapter 4 have been used with the two studies of the two applications to surface the existing test information and identify yet necessary achievable test topics. These goals have the aim to enlarge software project visions and scope and supplying it with more confidence towards decision making.

The existing test information identified in both studies (Subsection 6.1.1. and 6.1.2) enables test managers, designers and testers becoming aware of what are the test elements the project currently has, what can be done with them and how to do it according to the needs and to what they represent. The test people may also become aware of, according to what test information they have, what is the information and elements they still need in order to achieve the test objective in the software project.

Despite this discussion and the benefits may seem too general at this point, the connection of the elements is related to each project specifically. It has been possible to surface and identify useful information in the two studies though. In case of MobileMedia case study, the testware support has surfaced important information (and lack of it) that had not been identified before (case study Section 3.4). Such information could have helped on the issues that occurred throughout the case study, or even better, could have avoided bottlenecks, wrong decisions making and unsuccessful results. It has been possible to observe through the testware support evaluation that the case study had no test process nor policy defined, and it had been executed without assistance and planning. Such lack of coordination could hardly lead to successful results. If the testware support had been used to identify such issues, the case study could have taken a different test direction instead in time.

The evaluation with the other study regarding another application (HealthWatcher) allowed the observation that the testware support has also provided the identification of relevant test data and information that can be applied into the right test topic (environment, tool, planning) and clarify the test activities, milestones, process in a smooth and organized way. The identified characteristics support building the right test process provided with the right assessed test strategy, test environment and test tools.

Thus, the outcome of such evaluations indicates that the use of the testware support can be beneficial in software projects that is concerned with test directions and properly addressing the test elements. Such benefit can be even more important when the software project concerns critical levels of software life cycle, like its maintenance or evolution phase. When a piece of software has never been under test before and it needs to evolve, some minimum quality and reliability that the software system indeed does what it is supposed to do needs to be assessed and/or (re)assured. Without any prior test activity or history, such confidence is hardly achievable. Or even worse, with prior test activity or history, but inappropriate ones or not suitable to the new scenarios, the software system may be evolved or maintained through an inadequate path, which can lead to serious damaged and unsuccessful outcome in its life cycle. The test strategy and test criteria to be applied in a piece of software need to be evaluated each time it goes under changes. The existing testware of a software project may not be suitable to the new scenario. In other words, the software evolution can be inadequate in what regards to test issues, or be provided with the inadequate testware, and this may be harmful to the software in the future. Therefore, testware support comes to play a very important role to assess its testware and provide means to reach a proper test direction in a process flow, so that the software system can evolve, provided with proper information that has been evaluated.

The use of the testware support is fundamental to avoid misunderstandings during software assessment, since it documents the test elements. It is essential to help standardizing the terms used in analysis, replication and evaluation of studies that concern this subject and rely on initiatives, like this testware support, to progress on consistent and reliable information.

The testware support may guide researchers and practitioners to analyze software test projects and software test case studies to analyze test characteristics inherent and / or assessed from software systems towards many different purposes.

The next section evaluates the BF extension presented in Chapter 5 with the addition of the testware support, and discusses its benefits.

6.2 Benchmarking Framework Extension Evaluation

As explained in the introduction of the BF [42], the definition of the criteria that compose the BF is not an easy task, for they have to be general and specific at the same time. They should be: (i) general enough to identify different characteristics of the same application and, (ii) specific to comply with applications of different domains that use AO techniques. Yet another factor that makes this definition not an easy task is the quick appearance of new AO techniques, which hinders the criteria list to be statically defined; therefore, it should be dynamic to adapt imposed needed changes.

The list of test criteria, as known as the testware support addressing the test elements have been created and inserted to the BF (Chapter 5) so that testing, in a general way, is inserted within the context presented of the BF regarding the empirical studies on software maintenance. The insertion of the testware support in the BF has the goal to increase the scope of the software assessments and consider test issues, providing means to achieve more confident and reliable scenes and paths to evolve and maintain software, as it has been explained previously in this chapter.

This section presents the evaluation of the test attributes to consider within the BF extension. Such evaluation has followed the same line adopted in the evaluation of the BF in its original study [42]. Besides the goals described above regarding the expected achievements with the BF extension, this evaluation aims to provide means to assess whether the use of the BF extension in an experiment, software project or case study can indeed be beneficial, i.e., its use is advantageous than what it would be if it had not make use of such structure. The test attributes addressed in the BF extension aim to surface and provide further relevant information to a project that is important and can help on the outcome in the manners discussed in the previous chapters, but such need is unknown until it appears.

The evaluation has been applied into the two studies explained in the previous section, provided with two applications (MobileMedia and HealthWatcher). As the attributes exhibited in the extension are a summarized way to present the analysis assessed through the thorough use of the testware support, the evaluation here presents Tables 6.16 and 6.17 assessing the test elements discussed in Section 6.1, as per definitions from Table 5.1, in Chapter 5.

The test attributes introduced in Chapter 5 as the extension to the Benchmarking Framework structure to now consider test issues in software maintenance and evaluation studies and research have been assessed for two different applications above. The evaluation assesses the important test elements that surface test information, or the lack of it, providing the necessary knowledge for a software project or case study to pursue and follow the right path in what regards to software testing. Considering the testing process and elements in such BF broadens the view of researchers or practitioners that are assessing software systems to evolve. It also provides means to assure a higher level of quality and confidence in the related studies. This only confirms the benefits of the testware support discussed in Section 6.1 and expands such benefits towards the BF extension.

Together with the general software attributes and Aspect-Oriented attributes presented in the definitions of the BF [42], the test attributes, here assessed, contribute to the evaluation of the representativeness of applications, because it enlarges the scope to a different process that may impact on other related criteria. Thus, the extension enables the evaluation of applications in a more in-depth and embracing fashion of what may impact the case study or software project, increasing its confidence. Besides, the attributes also contribute to guiding, selecting and analyzing software systems to experimental studies, for it enables the consideration of further characteristics inherent to the project, that were unknown, without the use of the testware support.

Table 6.16 MobileMedia case study test attributes

First Basics	**Test policy**	*No*
	Test objective	*Analyze code covered by testing*
	Test strategy	*Code based testing and interface testing with full bug reporting at the end of testing each release*
	Test process	*No*
	Test history	*No*
	Test criteria	*Functional and structural based testing (data flow testing). Analysis based on code coverage and AOP related faults.*
	Functional and structural test conditions	*Interface testing as per test case description and steps; input domain testing. Data flow testing. In AO releases, first test base code and then aspectized code and observe whether a bug is observed when introducing the aspects in the base code.*
	Test environment	*No definitions*
	Test tools	*JaBUTi tool, code corage tool, automating tool*
	Test people	*Yes*

Test strategy	**Test plan**	*No*
	Milestones	*Full bug reporting at the end of testing of each release. New feature delivery every other week as of July 20th until November 20th*
	Test schedule	*See schedule table in item 2.5.1 in Table 6.5*
	Test requirements	*Feature model*
	Test techniques	*Functional and structure-based testing*
Test environment	**Requirements**	*Manageable, continuous, safe and centralized*
	Dependencies	*Tools, wireless toolkit, client-server configuration management, test server*
Test tools	**Type of tool**	*Capture/playback, code comparison, data flow analysis, defect management, walkthroughs*
	Tool name	*JaBUTi, Automate5, Cobertura*
	Tool version	*1, 5, any – respectively*
	Tool support	*User's manual*
	Tool expertise	*Low*
	Tool dependencies	*No definitions*
Test people	**Test roles**	*Test manager – A* *Test designer – B* *Tester – C, D and E*
	Training	*Yes*
	Competency	*See table 6.5*

Table 6.17 HealthWatcher case study test attributes

First Basics	**Test policy**	*No*
	Test objective	*Analyze code covered by testing and if behavior meet the specifications*
	Test strategy	*Code based testing and interface testing through browser*
	Test process	*No*
	Test history	*Yes*
	Test criteria	*Functional and structural based testing. Analysis is based on code coverage analysis and AOP related faults assessment*
	Functional and structural test conditions	*Interface testing through browser as per test case description and steps; input/output testing. Unit testing. In AO releases, first test base code and then aspectized code and observe whether a bug is observed when introducing the aspects in the base code*
	Test environment	*Software: one license for the Microsoft Windows for the workstation.* *Hardware: One computer with: Pentium III processor, 256 MB of RAM memory, net card*

		3Com 10/100. This equipment shall be used by the attendant as a workstation.
	Test tools	*Code comparison tool*
	Test people	*Yes*
Test strategy	**Test plan**	*No*
	Milestones	*The milestones are the end of each main flow test activities of each functionality. At every milestone, test reports are delivered to assess test activities*
	Test schedule	*See schedule table in item 2.5.1 in Table 6.12*
	Test requirements	*Use cases*
	Test techniques	*Functional and structure-based testing*
Test environment	**Requirements**	*Manageable, safe and centralized*
	Dependencies	*Other information services*
Test tools	**Type of tool**	*Code comparison, defect management*
	Tool name	*Cobertura, spreadsheets*
	Tool version	*any*
	Tool support	*User's manual*
	Tool expertise	*Medium*
	Tool dependencies	*No definitions*
Test people	**Test roles**	*Test manager – A1* *Test administrator – B1* *Tester – C1, C2 and C3*
	Training	*Yes*
	Competency	*See table 6.12*

6.3 Final Considerations

This chapter has presented the evaluation of the proposed testware support and the evaluation of the Benchmarking Framework [42] extension with the insertion of the testware support in it. The evaluation relied on two different applications, HealthWatcher and MobileMedia, which, because of their differences and particularities, have contributed to increase the representativeness of such assessment and provided more consistences to the results. It has been a positive and beneficial outcome to make use of the testware support, presented in this dissertation. The use of it has surfaced important and relevant test information from test elements that need to be considered in order to provide and pursue a proper test direction in a software project or case study. The benefits become even more important when looking at the perspective that to evolve a piece of software, the quality and confidence are essential aspects to be considered in order to reach and develop proper evolution or maintenance scenarios. Without test support, such confidence and reliability is hardly achievable, for a software system's test conditions or assessments are unknown. This throws a software to a very vulnerable situation and this is unlikely accepted in the software engineering community.

Thus, the testware support elements introduced in this dissertation contributes to the improvement of studies in this regard planning and elaboration, analysis and replication, in the context in which the BF has proposed to assist. Further details on beneficial implications of the testware support and its insertion in the BF are presented in the next chapter, the conclusions.

Chapter 7

Conclusions

As it has been introduced in Chapter 1, software testing plays an essential role to uncover and correct as many of the potential errors as possible in a piece of software. However, when the test issues worked in a project or case study are not provided with the proper way in techniques, tools and test environment to make them right, test objectives become hard to achieve. If the testware in a software project is not addressed adequately, it may be difficult critical for this system to evolve or maintain, since its quality will be provided with inappropriate elements and incorrect or incomplete information and assessment. Motivated by a case study that has taken place without a proper test direction to follow in a process flow, this work presented in this dissertation has developed a structure to serve as a guide to address test issues through test attributes in such situations, as known as the testware support. The context in which the case study has taken place regards software evolution and maintenance case studies, and focus on Aspect-Oriented (AO) software systems. The environment in which the case study was applied has been introduced in Chapter 3 and it concerned a Testbed [105], a systematic structure to provide end-to-end comparison between techniques, approaches, metrics, empirical studies, artifacts and applications, essential framework to stimulate the research, contributions and use of its benefits within software engineering community. A specific part of the Testbed, the Benchmarking Framework (BF), has been developed [42] to support the maintainability assessment of AO software development (AOSD) techniques through a definition of an idealized scheme for benchmarking applications to evaluate AO attributes within the maintenance context. The framework guides researchers and practitioners in selecting or adapting applications and their maintenance scenarios that best fit specific experimental goals as well as it supports the design replication and evaluation of empirical studies, as it has been previously explained. Such framework, however, did not address any test issues within this context. A study handling a software system, assessing its characteristics and disregarding any information related to testing activities or test attributes, quality and confidence could easily be addressing an improper maintenance scenario, once such scenario has not been tested or evaluated.

When a maintenance scenario is taken into consideration to evolve or maintain a piece of software, it is important to assess whether it is indeed specified as it should. If there is a bug in it, it may not be known and may imply further complications and damages in software life span. Based on such proposition and motivated by the case study executed without much of testing knowledge, this dissertation presents the testware support addressing test core elements to be taken into consideration in case studies and software projects and its insertion within the BF. The BF extension now considering testware support addresses test attributes, as presented in Chapter 5. Chapter 6 has introduced the results of this work evaluation within two different perspectives: the testware support evaluation itself and the BF extension considering the test attributes as a support to consider test activities and learn from what they can provide along the software project. The benefits have been observed in both evaluations for the elements have surfaced relevant and important information that can make the difference. The avoided problems can also avoid further damages, provide confidence and enlarge the scope when assessing AO or non-AO software attributes in a software maintenance or evolution case study.

The next section discusses the relevance of such initiative and related work, Section 7.2 presents the final considerations about this chapter and dissertation and, finally, Section 7.3 discusses the future work following the line of the testware support and the context here presented.

7.1 Related Work

When discussing software testing in the context of software evolution or maintenance, it is mostly related to regression testing and regression test selection technique [23, 24, 25, 27, 85, 86, 88, 89, 91]. Once testing the whole software is practically not feasible and testing is an expensive task, test cases prioritization and best techniques matched with project goals are then performed. However, further interests have taken place in this regard and some studies [14, 15, 16, 17, 18, 103] have researched about a software system testability, i.e., 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. Such research is important to reduce testing cost and time and increase software maintenance rate of success, once testing effort can be reduced and optimized with testability levels. Still, some studies [7, 8, 9, 12, 20, 21, 22, 56, 57, 60, 70, 73, 90, 104, 106, 128, 129, 133, 136] have taken place to experiment test techniques and prove their suitability to specific contexts and requirements, address a specific model to a specific kind of software, application or case study, others [84] have focused on identifying dependencies that could decrease quality and hinder proper test activities, while other authors [58, 59, 60, 136] have even built testing framework with specific tools for specific programs. However when focusing on empirical studies context, there are many works [29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 94, 111, 113] that presents relevant aspects to be taken into consideration when evaluating methods, presenting empirical evidence, discussing existing ways to purport software methodologies or providing new techniques into the community but does not consider the testing world in it. Some works in software evolution [42, 115, 116, 117, 118] even address attributes and discuss relevant perspectives in this matter, but very few works [9] extend the subject with regard to testing in the evolutionary context. Thus, because no authors have provided a testware support that would be nearly applicable to the context here inserted, none is discussed into more details or compared. Some are described in the state of art (Chapter 2) but their comparison to the subject from this dissertation is not pertinent, since the work here presented, besides being focused in software maintenance and AOP, its use is appropriate to a wider sense. And in a wider sense, since this work has gathered existing information in the testing world that would best fit its purposes, i.e., there are no new testing concepts, its comparison to the existing testware is also not applicable.

Benchmarking as a way to assess software characteristics is still a recent subject within the community with not so many works [42, 54, 58] available. Yet, testing is not present in this research. There is no testware support along the software evolution in any of these works, which can lead to the issues revealed and discussed in this dissertation.

The lack of planning necessary resources to develop and / or maintain or evolve software systems can be pointed out as one of the serious reasons from unsuccessful results, software crisis and lack of directions to assist towards a proper path [133]. A proper path, in this context, is the one to lead to the expected software project's results, confidence and successful outcome. The test activities planning should be part of system's global planning, whether in development, maintenance or evolution, in any phase of the software life cycle, the lack of an adequate test direction can lead to serious issues to be faced by researchers and practitioners as it has been discussed in this dissertation.

7.2 Final Considerations

This dissertation has discussed the importance and shortage of testware support in the field of software engineering, more specifically in AOSD. The testware support provides test elements to be considered with regard to testing in a software project, so that the researcher or practitioner has the means to expand its scope, vision and capabilities towards decision making. The expansion comes from the context of the Benchmarking Framework [42] here discussed. The insertion of the testware support in such context can only provide benefits and contribute to a more confident and further acquainted and explored scenario, rather maintenance scenario, evolution or application assessment. However, to manipulate the test elements, some expertise is required and test resource is necessary so that its proposed benefits can indeed be achieved, as assessed and demonstrated in Chapter 6.

The results shown in Chapter 6 are of extreme importance for software engineering field for this is the first initiative to extend the BF structure and the first contribution regarding testing within the Testbed [105]. The benefits of the BF can be summed up as facilitating the execution of empirical studies and accelerating the collaborative progress of the software engineering field, in a general way. Looking closer at it, it constitutes a significant progress towards the creation of a comprehensive methodology for designing and assessing AO software benchmarks. It provides a systematic approach to the design of AOSD maintainability studies and it guides the selection, design or adaption of representative evolving applications and releases to be used in such studies and their replications. Finally, it helps the community to accelerate the improvement of software engineering body of knowledge and better support for judgment of industrial decision makers. Extending such advantages, the testware support provides means to surface existing test information and testware in a software project and identifies the lack of other elements in the testware that need to be addressed in order to build an adequate test strategy, followed by a test process, test planning and so on. Such testware support helps the researchers to address and follow through a proper test direction in a case study that regards testing. Considering testing elements within the BF structure, as mentioned, enables the improvement of confidence and quality level of the information available to researchers and practitioners that are responsible for decision making and software characteristics assessment.

Hence, despite being great advance, in the perspective of software testing, there is still a lot to improve and progress. The elements handled in the testware support are the basics of software testing, as it has been previously stated in this dissertation. The basic testware in order to assess the minimum information in order to achieve a proper test direction with a proper test process and test plan is here presented and discussed. However, there are still many important subjects that are not here discussed which can and should be further investigated, as it is discussed in the next and last section.

7.3 Future Work

As it has been introduced in Section 7.2, the testware support provides the "basics" to address basic software test elements to reach an adequate test direction to follow in a test process, with appropriate own test process, planning, environment and other decisions. However, there are still many other aspects that can and should be considered in future studies.

This work is only the first initiative to address test concerns within the BF and Testbed world. It can and should serve as a starting point to develop and address further concerns, such as test metrics, test case design, test prioritization, risks and others.

Further empirical research considering tests can and should take place to provide more confident means to address testing in this context and show, by each study, the importance and relevance of concerning test elements in this context.

Further research should also take place to evaluate in different manners the proposed testware support and the BF extension. The BF is a very new and important contribution within empirical studies community and it has not yet been continually used and tested. Such use is important to assess its benefits and representativeness by different kinds of applications and studies. Following this line, the testware support can and should also be used, tested and extensively assessed, so that it helps supplying the lack of test directions in software development, maintenance and evolution empirical studies and projects.

References

[1] Lenski, G. Power and Privilege: A Theory of Stratification. McGraw-Hill, 1966.

[2] ISO/IEC 9126-1:2001 Standard. Software Engineering – Product quality – Part 1: Quality model, Quality characteristics and sub-characteristics.

[3] Myers, G. J. The Art of Software Testing. Second Edition. John Wiley & Sons Inc., 2004.

[4] Harrold, M. J. Testing: A roadmap. In *22th International Conference on Software Engineering – Future of Software Engineering Track*, 2000, p. 61-72.

[5] Spillner A., Linz, T. and Schaefer, H. Software Testing Foundations. Dpunkt-verlag, 2006.

[6] Graham, D., Van Veenendaal, E., Evans, I. and Black, R. Foundations of Software Testing, ISTQB Certification. Thomson, 2007.

[7] Maldonado, J. C., Vincenzi, A. M. R., Barbosa, E. F., Souza, S. R. S. and Delamaro, M. E. Aspectos teóricos e empíricos de teste de cobertura de software. Relatório Técnico 31, Instituto de Ciências Matemáticas e de Computação – ICMC-USP, 1998.

[8] Hutchins, M., Foster, H., Goradia, T. and Ostrand, T. Experiments on the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria. In *Proceedings of the 16th International Conference on Software Engineering. Number 16 in ICSE*, IEEE, 1994, p. 191-200.

[9] Harman, M. and McMin, P. A Theoretical & Empirical Analysis of Evolutionary Testing and Hill Climbing for Structural Test Data Generation. In *International Symposium on Software Testing and Analysis*, 1997.

[10] Perry, W. E. Effective Methods for Software Testing. Third Edition. Wiley, 2006.

[11] Pressman, R. S. Engenharia de Software. Translation of Software Engineering: A Practitioner's Approach. Third Edition. Makron Books, 1995.

[12] Agrawal, H. et al. Mining Systems Tests to Aid Software Maintenance. IEEE Computer Society Press. Volume 31, Issue 7, p. 64-73, 1998.

[13] Tichy, W. F. Should Computer Scientists Experiment More? IEEE Computer Society Press. Volume 31, Issue 5, p. 32-40, 1998.

[14] Gupta, S. C. and Sinha, M. K. Impact of Software Testability Considerations on Software Development Life Cycle. In *Proceedings of the First International Conference on Software Testing, Reliability and Quality Assurance*, pp.105–110, 1994.

[15] Jimenez, G., Taj, S. and Weaver, J. Design for Testability. In *Proceedings of the 9th Annual NCIIA Conference,* 2005.

[16] Jungmayr, S. Design for Testability. In *Proceedings of CONQUEST 2002*, pages 57–64, 2002.

[17] Pettichord, B. Design for Testability. In *Proceedings of Pacific Northwest Software Quality Conference (PNSQC)*, Anaheim, California, 2002.

[18] Eickelman, N. S. and Richardson, D. J. What Makes One Software Architecture More Testable Than Another? In *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, pp. 65–67, 1996.

[19] Kolb, R. and Muthig, D. 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. ACM, pp. 22-27, 2006.

[20] Kung, D., Gao, J., Hsia, P., Toyoshima, Y., Chen, C., Kim, Y., and Song, Y. Developing an object-oriented software testing and maintenance environment. *Commun. ACM* 38, 1995.

[21] Fayad, M.E. Object-oriented software engineering: Problems and perspectives. Ph.D dissertation, University of Minnesota, 1994.

[22] Binder, R. Object-oriented software testing. *Commun*. ACM 37, 1994.

[23] Agrawal, H., Horgan, J., Krauser, E. and London, S. Incremental regression testing. In *Proceedings of the Conference on Software Maintenance*. IEEE, pp. 348–357, 1993.

[24] Rothermel, G. and Harrold, M. J. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol.* Volume 6, Issue 2, pp. 173-210, 1997.

[25] Graves, T. L., Harrold, M. J., Kim, J., Porter, A. and Rothermel, G. An empirical study of regression test selection techniques. *ACM Trans. Softw. Eng. Methodol.* Volume 10, Issue 2, pp. 184-208, 2001.

[26] Beizer, B. Software Testing Techniques. Van Nostrand Reinhold, 1990.

[27] Martinig, F. Software testing: Poor consideration. Testing Tech. Newsletter, 1996.

[28] Jedlitschka, A., Ciolkowski, M. Reporting Experiments in Software Engineering. In *14th International Software Engineering Research Network Annual Meeting ISERN*. 2006.

[29] Sjoberg, D. I., Dyba, T., and Jorgensen, M. The Future of Empirical Methods in Software Engineering Research. In *2007 Future of Software Engineering. International Conference on Software Engineering*. IEEE Computer Society, pp. 358-378, 2007.

[30] Ciolkowski, M. and Münch, J. Accumulation and presentation of empirical evidence: problems and challenges. In *Proceedings of the 2005 Workshop on*

Realising Evidence-Based Software Engineering REBSE '05. ACM, pp. 1-3, 2005.

[31] Vegas, S., Juristo, N., Moreno, A., Solari, M., and Letelier, P. Analysis of the influence of communication between researchers on experiment replication. In *Proceedings of the 2006 ACM/IEEE international Symposium on Empirical Software Engineering* ISESE '06. ACM, pp. 28-37, 2006.

[32] Zannier, C., Melnik, G., and Maurer, F. On the success of empirical studies in the international conference on software engineering. In *Proceedings of the 28th international Conference on Software Engineering* ICSE '06. ACM, pp. 341-350, 2006.

[33] Jedlitschka, A. and Ciolkowski, M. Towards Evidence in Software Engineering. In *Proceedings of the 2004 international Symposium on Empirical Software Engineering*. IEEE Computer Society, pp. 261-270, 2004.

[34] Redwine, T. and Riddle, E. Software Technology Maturation. In *Proceedings of the 8th international Conference on Software Engineering*. IEEE Computer Society, pp. 189-200, 1985.

[35] Kitchenham, B., Al-Khilidar, H., Babar, M. A., Berry, M., Cox, K., Keung, J., Kurniawati, F., Staples, M., Zhang, H. and Zhu, L. Evaluating guidelines for reporting empirical software engineering studies. *Empirical Software Engineering*. Volume 13, Issue 1, pp. 97-121, 2008.

[36] Shull, F., Mendoncça, M. G., Basili, V., Carver, J., Maldonado, J. C., Fabbri, S., Travassos, G. H. and Ferreira, M. C. Knowledge-Sharing Issues in Experimental Software Engineering. *Empirical Software Engineering*. Volume 9, Issue 1-2, pp. 111-137, 2004.

[37] Kitchenham, A., Pfleeger, L., Pickard, M., Jones, W., Hoaglin, C., El Emam, K. and Rosenberg, J. Preliminary guidelines for empirical research in software engineering. IEEE Transactions on Software Engineering, Volume 28, N^{o}. 8, pp. 721 -734, 2002.

[38] Singer, J. Association (APA) Style Guidelines to Report Experimental Results. In *Proceedings of Workshop on Empirical Studies in Software Maintenance*. pp. 71-75, 1999.

[39] Lemos, O. Teste de programas orientados a aspectos: uma abordagem estrutural para AspectJ. MSc dissertation. ICMS-USP, 2005.

[40] Watt, David. Programming Language Design Concepts. Wiley, 2004.

[41] Gradecki, J. D.; Gradecki, J.; Lesiecki, N. Mastering AspectJ: Aspect-Oriented Programming in Java. Wiley, 2003.

[42] Moura, M. Um Benchmarking Framework para Avaliação da Manutenibilidade de Software Orientado a Aspectos. MSc dissertation. DSC-UPE, 2008.

[43] McDaniel, R. and McGregor, J. Testing polymorphic interactions between classes. Technical Report TR-94-103, Clemson University, 1994.

[44] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.J. and Irwin, J. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, 1997.

[45] Murphy, C., Walker, J., Baniassad, L., Robillard, P., Lai, A. and Kersten, A. 2001. Does aspect-oriented programming work? *Commun.* ACM Volume 44, Issue 10, pp. 75-77, 2001.

[46] Coady, Y., Kiczales, G., Feeley, M., Hutchinson, N. and Ong, S. Structuring operating system aspects: using AOP to improve OS structure modularity. *Commun*. ACM Volume 44, Issue 10, pp. 79-82, 2001.

[47] Coady, M. Y. Improving Evolvability of Operating Systems with Aspectc. Doctoral Thesis. UMI Order Number: AAINQ86004., The University of British Columbia (Canada).

[48] Papapetrou, O. and Papadopoulos, G. A. Aspect Oriented Programming for a component-based real life application: a case study. In *Proceedings of the 2004 ACM Symposium on Applied Computing*. SAC '04. ACM, pp. 1554-1558, 2004.

[49] Cacho, N. et al. Composing Design Patterns: A Scalability Study of Aspect-Oriented Programming. AOSD ’06, 2006.

[50] Krüger, H., Mathew, R. and Meisinger, M. From scenarios to aspects: exploring product lines. In *Proceedings of the Fourth international Workshop on Scenarios and State Machines: Models, Algorithms and Tools*. SCESM '05. ACM, pp. 1-6, 2005.

[51] Figueiredo, E. et al. Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability. ICSE ’08, 2008.

[52] Kulesza, U., Sant’Anna, C., Garcia, A., Coelho, R., von Staa, A. and Lucena, C. Quantifying the Effects of Aspect-Oriented Programming: A Maintenance Study. In *Proceedings of the 22nd IEEE international Conference on Software Maintenance* ICSM, 2006.

[53] Demeyer, S., Mens, T. and Wermelinger, M. Towards a Software Evolution Benchmark. In *Proceedings of the 4th international Workshop on Principles of Software Evolution* IWPSE ’01. ACM, pp. 174-177, 2001.

[54] Sim, S., Easterbrook, S. and Holt, R. Using Benchmarking to Advance Research: A Challenge to Software Engineering. In *Proceedings of the 25th international Conference on Software Engineering*. IEEE Computer Society, pp. 74-83, 2003.

[55] Spillner, A., Rossner, T., Winter, M. and Linz, T. Software Testing Practice: Test Management. Rockynook Computing, 2007.

[56] Do, H., Elbaum, S. and Rothermel, G. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Software Engineering*. Issue 10, Volume 4, pp. 405-435, 2005.

[57] Vegas, S. and Basili, V. A Characterisation Schema for Software Testing Techniques. *Empirical Software Engineering*. Volume 10, Issue 4, pp. 437-466, 2005.

[58] Eytani, Y., Havelund, K., Stoller, S. D. and Ur, S. Towards a framework and a benchmark for testing tools for multi-threaded programs: Research Articles. *Concurrency and Computation: Practice & Experience*. Volume 19, Issue 3, pp. 267-279, 2007.

[59] Xie, J., Ye, X., Li, B. and Xie, F. A Configurable Web Service Performance Testing Framework. In *Proceedings of the 2008 10th IEEE international Conference on High Performance Computing and Communications* - Volume 00. HPCC. IEEE Computer Society, pp. 312-319, 2008.

[60] Masemola, S. S. and De Villiers, M. R. Towards a framework for usability testing of interactive e-learning applications in cognitive domains, illustrated by a case study. In *Proceedings of the 2006 Annual Research Conference of the South African institute of Computer Scientists and information Technologists on IT Research in Developing Countries*. J. Bishop and D. Kourie, Eds. ACM International Conference Proceeding Series, vol. 204. South African Institute for Computer Scientists and Information Technologists, pp. 187-197, 2006.

[61] Sebesta, W. Robert. Concepts of Programming Languages. 5th Edition. Addison Wesley, 2001.

[62] Winck, V. Diogo and Junior, G. V. AspectJ: Programação Orientada a Aspectos com Java. Novatec, 2006.

[63] Gosling, J., Joy, B., Steele, G. and Bracha, G. The Java Language Specification. 2nd Edition. Addison Wesley, 2000.

[64] Soares, S. and Borba, P. AspectJ – Programação orientada a aspectos em Java. UFPE.

[65] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W. G. Getting started with AspectJ. Commun. ACM Volume 44, Issue 10, pp. 59-65, 2001.

[66] Binder, R. V. Testing Object-Oriented Systems: Models, Patterns and Tools. Version 1. Addison Wesley Longman, Inc., 1999.

[67] Alexander, R. T., Bieman, J. M. and Andrews, A. A. Towards the Systematic Testing of Aspect-Oriented Programs. Technical Report, Colorado State University, 2004.

[68] Tian, J. Software Quality Engineering – Testing, Quality Assurance and Quantifiable Improvement. IEEE Wiley-Interscience, 2005.

[69] Boehm, B. W. Software Engineering Economics. Prentice Hall, 1981.

[70] Boehm, B. W. Guidelines for Verifying and Validation Software Requirements and Design Specifications. In *Proceedings of Euro IFIP*, pp. 711-719, 1979.

[71] URL: http://www.v-modell-xt.de/

[72] Swanson, E. B. and Dans, E. System Life Expectancy and the Maintenance Effort: Exploring Their Equilibration. MIS Quarterly, volume 24, pp. 277-297, 2000.

[73] Pol, M. and van Veenendaal, E. Structured Testing of Information Systems. Kluwer Bedrijfsinformatie: Deventer, The Netherlands, 1998.

[74] Martin, J. Rapid Application Development. Macmillan, 1991.

[75] Gilb, T. Competitive Engineering: A Handbook for Systems & Software Engineering Management using Planguage. Butterworth-Heinemann, Elsevier, 2005.

[76] Beck, K. Extreme Programming. Addison-Wesley, 2000.

[77] Canning, R. The Maintenance 'Iceberg'. EDP Analyser, volume 10, nº 10, 1972.

[78] Swanson, E. B. The Dimensions of Maintenance. In *Proceedings of 2nd International Conference on Software Engineering*, IEEE, pp. 492-497, 1976.

[79] Kästner, C., Apel, S. and Batory, D. A Case Study Implementing Features using AspectJ. In Proceedings *of International Software Product Line Conference* (SPLC), 2007.

[80] Dósea, M., Costa Neto, A., Borba, P. and Soares, S. Specifying Design Rules in Aspect-Oriented Systems. In *Proceedings of I Latin American Workshop on Aspect-Oriented Software Development – LA-WASP'2007*, affiliated with SBES 2007, pp. 67-78, Brazil, 2007.

[81] Ribeiro, M., Dósea, M., Bonifácio, R., Costa Neto, A., Borba, P. and Soares, S. Analyzing Class and Crosscutting Modularity With Design Structures Matrixes. In *Proceedings of XXI Brazilian Symposium on Software Engineering – SBES'2007*, pp. 167-181, Brazil, 2007.

[82] Garcia, A., Sant'Anna, C., Figueiredo, E., Kulesza, U., Lucena, C. and von Staa, A. Modularizing Design Patterns with Aspects: A Quantitative Study. In *Proceedings of the 4th international Conference on Aspect-Oriented Software Development* AOSD '05. ACM, pp. 3-14, 2005.

[83] Baniassad, E. L., Murphy, G. C., Schwanninger, C. and Kircher, M. Managing Crosscutting Concerns during Software Evolution Tasks: An Inquisitive Study. In *Proceedings of the 1st international Conference on Aspect-Oriented Software Development* AOSD '02. ACM, pp. 120-126, 2002.

[84] Podgurski, A. and Clarke, L. The Implications of Program Dependences for Software Testing, Debugging. Technical Report. UMI Order Number: UM-CS-1989-043., University of Massachusetts, 1989.

[85] Engström, E., Skoglund, M. and Runeson, P. Empirical Evaluations of Regression Test Selection Techniques: A Systematic Review. In *Proceedings of the 2nd ACM-IEEE international Symposium on Empirical Software Engineering and Measurement* ESEM '08. ACM, pp. 22-31, 2008.

[86] Rothermel, G., Elbaum, S., Malishevsky, A. G., Kallakuri, P. and Qiu, X. On Test Suite Composition and Cost-effective Regression Testing. *ACM Trans. Software Engineering Methodologies*, Volume 13, Issue 3, pp. 277-331, 2004.

[87] Eaddy, M., Zimmermann, T., Sherwood, K. D., Garg, V., Murphy, G. C., Nagappan, N. and Aho, A. V. Do Crosscutting Concerns Cause Defects? *IEEE Transactions on Software Engineering,* Volume 34, Issue, pp. 497-515, 2008.

[88] Xu, G. and Rountev, A. 2007. Regression Test Selection for AspectJ Software. In *Proceedings of the 29th international Conference on Software Engineering*. International Conference on Software Engineering. IEEE Computer Society, pp. 65-74, 2007.

[89] Zhao, J., Xie, T. and Li, N. Towards Regression Test Selection for AspectJ Programs. In *Proceedings of the 2nd Workshop on Testing Aspect-Oriented Programs* WTAOP '06. ACM, pp. 21-26, 2006.

[90] Xu, D. and Xu, W. 2006. State-based Incremental Testing of Aspect-oriented Programs. In *Proceedings of the 5th international Conference on Aspect-Oriented Software Development* AOSD '06. ACM, pp. 180-189, 2006.

[91] G. Xu. A Regression Tests Selection Technique for Aspect-oriented Programs. In *Workshop on Testing Aspect-Oriented Programs*, pp. 15–20, 2006.

[92] Lehman M. M. and Belady L. A., Program Evolution – Processes of Software Change. Acad. Press, London, 1985.

[93] Mens, T. and Demeyer, S. Software Evolution. Springer, 2008.

[94] Black, R. Managing the Testing Process,: Practical Tools and Techniques for Managing Hardware and Software Testing, Second Edition. Wiley, 2002.

[95] Koomen, T., Aalst, L. van der, Broekman, B. and Vroon, M. Tmap Next for result-driven testing. Sogeti. UTN Publishers, 2006.

[96] Young, T. Using AspectJ to Build a Software Product Line for Mobile Devices. Master Dissertation, Computer Science, University of British Columbia, Canada, 2005.

[97] Greenwood, P. et al. On the Design of an End-to-End AOSD Testbed for Software Stability. Proceedings of International Workshop on Assessment of Aspect-Oriented Technologies (ASAT.07), AOSD Conference, Canada, 2007.

[98] Soares, S. An Aspect-Oriented Implementation Method. PhD thesis, UFPE, Brazil, 2004.

[99] Soares, S. et al. Implementing Distribution and Persistence Aspects with AspectJ. In *Proceedings of International Conference on Object Oriented Programming Systems, Languages and Applications OOPSLA*, pp. 174-190, 2002.

[100] Mezini, M. and Ostermann, K. Conquering Aspects with Caesar. In *Proceedings of Aspect-Oriented Software Development Conference AOSD*, pp. 90-99, 2003.

[101] Zhao, J. Data-Flow Based Unit Testing of Aspect-Oriented Programs. In *Proceedings of the 27th Annual IEEE International Computer Software and Applications Conference*. USA, 2003.

[102] Xu, W.; Xu, D.; Goel, V. and Nygard, K. Aspect flow graph for testing aspect-oriented programs. In *Proceedings of the 8th IASTED International Conference on Software Engineering and Applications*, 2004.

[103] Ceccato, M.; Tonella, P. and Ricca, F. Is AOP code easier or harder to test than OOP code? In *4th International Conference on Aspect-Oriented Software Development AOSD – Workshop on Testing Aspect Oriented Programs*. USA, 2005.

[104] Zhou, Y.; Richardson, D. and Ziv, H. Towards a practical approach to test aspect-oriented software. In *Proceedings of the 2004 Workshop on Testing Component-based Systems TECOS*. Net.ObjectiveDays, 2004.

[105] Greenwood, P. et al. On the Contributions of an End-to-End AOSD Testbed. In *11th Workshop on Early Aspects - Aspect-Oriented Requirements Engineering and Architecture Design ICSE*. USA, 2007

[106] Lima, G. M. P. S. and Travassos, G. H. Estratégia para Testes de Integração aplicada a Software Orientado a Objetos. UFRJ. Rio de Janeiro, Brazil.

[107] Harrold, M. J. and Rothermel, G. Performing Dataflow Testing on Classes. In *Proceedings of the ACM SIGSOFT ´94 Symposium on the Foundations of Software Engineering*. ACM, pp. 154-163, 1994.

[108] Huynh, D. Software Testing Maturity Model^{SM} (SW-TMM^{SM}). University of Maryland, 2002.

[109] Burnstein, I., T. Suwanassart, and C.R. Carlson. Developing a Testing Maturity Model. Crosstalk, Software Technology Support Center, Hill Air Force Base, Utah; Part I: August 1996, pp. 21-24; Part II: September 1996, pp. 19-26.

[110] Paulk, M., C. Weber, B. Curtis, and M. Chrissis, The Capability Maturity Model: Guideline for Improving the Software Process, Addison-Wesley, Reading, Mass., 1995.

[111] Banker, R. D., Datar, S. M. and Kemerer, C. F. Factors Affecting Software Maintenance Productivity: An Exploratory Study. In *Proceedings of the 8th International Conference on Information Systems ICIS*. Pp. 160-175, 1987.

[112] Martin, J. and McClure, C. L. Software Maintenance: The Problems and Its Solutions. Prentice Hall Professional Technical Reference, 1983.

[113] Kemerer, C. F. Software Complexity and Software Maintenance: A Survey of Empirical Research. Annals of Software Engineering (1), pp. 1-22, 1995.

[114] Banker, R. D., Datar, S. M., Kemerer, C. F. and Zweig, D. Software Errors and Software Maintenance Management. Inf. Technol. and Management 3, 1-2, PP. 25-41, 2002.

[115] Barry, E., Slaughter, S. and Kemerer, C. F. An Empirical Analysis of Software Evolution Profiles and Outcomes. In *Proceedings of the 20th International Conference on Information Systems.* Association for Information Systems, pp. 453-458, 1999.

[116] Kemerer, C. F. and Slaughter, S. An Empirical Approach to Studying Software Evolution. IEEE Transactions on Software Engineering. 25, 4, pp. 493-509, 1999.

[117] Kajko-Mattson, M., Lewis, G. A. and Smith, D. B. A Framework for Roles for Development, Evolution and Maintenance of SOA-Based Systems. In

Proceedings of the 29th International Conference on Software Engineering Workshops ICSEW. IEEE Computer Society, pp. 117, 2007.

[118] Goldschmidt, T., Reussner, R. and Winzen, J. A Case Study Evaluation of Maintainability and Performance of Persistence Techniques. In *Proceedings of the 30th International Conference on Software Engineering ICSE*. ACM, pp. 401-410, 2008.

[119] Rosenblum, D. S. Validation and Verification – Regression Testing. Department of Computer Science, University College London UCL, 2009.

[120] iBATIS Data Mapper – http://ibatis.apache.org/ (13/03/2009)

[121] JUnit Testing Framework – http://www.junit.org/ (13/03/2009)

[122] Zhu, H., Hall, P. A. V. and May, J. H. R. Software Unit Test Coverage and Adequacy. ACM Computing Surveys 29 (4), pp. 367-427, 1997.

[123] Pohl, K. et. al. Software Product Line Engineering: Foundations, Principles and Techniques. Springer, 2005.

[124] McGregor, J.D. Testing in a Software Product Line, School on Software Engineering–Testing. Recife, Brazil, 2007.

[125] Li, S. and j. Knudsen. Beginning J2ME, Apress, 2005.

[126] Report at the conference Quality Assurance: Management & Technologies. Ukraine, 2007.

[127] Vincenzi, A. M. et. al. JaBUTi – Java Bytecode Understanding and Testing-user's guide-version 1.0, 2004

[128] Delamaro, M. E. and Vincenzi, A. M. R. Structural Testing of Mobile Agents. In E. A. Nicolas Guelfi and G. Reggio, editors, III International Workshop on Scientific Engineering of Java Distributed Applications (FIDJI'2003), Lecture Notes on Computer Science. Springer, 2003.

[129] Vincenzi, A. M. R., Maldonado, J. C., Wong, W. E. and Delamaro, M. E. Coverage testing of Java programs and components. Journal of Science of Computer Programming, 56(1-2):211-230, Apr. 2005.

[130] Sun Java Wireless Toolkit for CLDC – http://java.sun.com/products/sjwtoolkit/ (13/03/2009)

[131] Pezzè, M. et al. Software Testing and Analysis: Process, Principles and Techniques. Bookman, 2008.

[132] McKay, J. Managing the Test People – A Guide to Practical Technical Management. Rocky Nook Computing, 2007.

[133] Maldonado, J.C.. Critérios potenciais usos: Uma contribuição teste estrutural de software. PhD Thesis, Unicamp, Brazil, 1991.

[134] Buschmann, F. et al. Pattern-Oriented Software Architecture: A System of Patterns. John Wiley & Sons, 1996.

[135] Gamma, E. et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[136] Spring, L. J., Harris, G. H., Forster, J. J., and Berghel, H. A Proposed Benchmark for Testing Implementations of Crossword Puzzle Algorithms. In *Proceedings of the 1992 ACM/SIGAPP Symposium on Applied Computing: Technological Challenges of the 1990's*. H. Berghel, E. Deaton, G. Hedrick, D. Roach, and R. Wainwright, Eds. SAC '92. ACM, pp. 99-101, 1992.

[137] Vincenzi, A. M. Orientação a Objeto: Definição, Implementação e Análise de Recursos de Teste e Validação. PhD Thesis, ICMC/USP, Brazil, 2004.

[138] Darwin, C. The Origin of Species. P. F. Collier & Son, volume 11, Harvard University, 1909.

[139] White, L. The Evolution of Culture: The Development of Civilization to the Fall of Rome. University of Indiana. McGraw Hill, 1959.

[140] Foley, R. and Lahr, M. M. On Stony Ground: Lithic Technology, Human Evolution and the Emergency of Culture. Evolutionary Anthropology 12, pp. 109-12, 2003.

[141] Morgan, L. H. Ancient Society: Researchers in the Lines of Human Progress from Savagery through Barbarism to Civilization. University of Virgina. H. Hold and Company, 1907.

[142] Sahlins, M. D., Service, E. R. and Harding, T. G. Evolution and Culture. University of Michigan Press, 1960.

[143] Toffler, A. Future Shock. University of Michigan. Random House, volume 644, 1970.

[144] Naisbitt, J. Megatrends: Ten New Directions Transforming Our Lives. Warner Books, 6th Edition, 1982.

[145] The Free Dictionary by Farlex Copyright 2009 – http://encyclopedia.thefreedictionary.com (22/06/2009)

[146] Stellman, A. and Greene, J. Applied Software Project Management. Addison Wesley, 2005.

[147] Sharma, D. Ten Essentials Elements to Guarantee Enhanced Software Quality. Article at Embedded.com – The Official Site of the Embedded Development Community, 2009 - http://www.embedded.com/design/218000002 (17/06/2009)

[148] Filho, F. C.et al. Exceptions and Aspects: the Devil is in the Details. In Proceedings of FSE, pp. 152-162, 2006.

[149] Moura, M., Garcia, A., Soares, S., Castor, F., Monteiro, M. and Greenwood, P. On Deriving Benchmarks for Aspect-Oriented Software Maintainability. Submitted in XXIV Brazilian Symposium on Database – XXIII Brazilian Symposium on Software Engineering, 2009.

ESCOLA POLITÉCNICA DE PERNAMBUCO

Appendix A

Here there are some examples of the test cases created to execute MobileMedia releases. The complete set of the test cases is available in a spreadsheet file at http://www.cin.ufpe.br/~scbs/liana/.

TC #	3	1	33	32
Feature	Add / Delete Photo Label Photo	Create Photo Album	Edit Label	Sorting
Description	Adding a photo with correct attributes.	New photo album creation	Edit an item's label	Sort photo's list by view
Pre-condition	1. Application must be launched; 2. There must be a photo available and stored on phone's images folder.	Application must be launched	1. Application must be launched; 2. There must be photos available and stored on phone's images folder.	1. Application must be launched; 2. There must be photos available and stored on phone's images folder.
Steps	1. The user selects an available album and select the option to add a photo; 2. The photo name and path should be inserted (path as '/images/name.png') and saved.	1. The user selects the option to create a new photo album; 2. The user types something and saves it.	1. The user creates a new album and inserts photos in it with correct label and path; 2. Edit label of items; 3. View the items that had label changed.	1. The user creates a new album and inserts photos in it with correct label and path; 2. View the items; 3. Select option to sort by view.
Expected Result	1. A new screen should be displayed with photo's name and path information to be inserted; 2. The new photo is stored in the selected album.	1. A new screen should be displayed with album's name information to be inserted. 2. The new album should be created and displayed on the albums list.	1. Items must be inserted properly; 2. Labels should be changed properly; 3. Viewing shows correct images of the files.	1. Items must be inserted properly; 2. Images should be shown properly; 3. The photo's list should now be organized according to view order.

Appendix B

The bug report created by iBATIS/MobileMedia joint case study test people.

iBATIS/MobileMedia Joint Case Study

Fault Description Sheet

1	Tester/Developer who has identified the fault					
2	Fault location					
3	Fault description					
4	Test case name					
5	Test case file					
5	Has this fault been previously reported (e.g. in a bug repository)?	Yes	☐	No	☐	
6	If you answered "yes" in question 5, please provide details (e.g. fault repository, fault identifier etc.)					
7	Is the fault related to	OO mechanism?	☐	AO mechanism?	☐	Both ? ☐
8	Please, provide more details (e.g. which mechanism, which way it is applied)					
9	Is the fault related to	CC-concern?	☐	Non-CC concern?	☐	Both ? ☐
10	Please, provide more details.					

Appendix C

The example of a test case created in the HealthWatcher study. The other test cases created can be found at http://www.cin.ufpe.br/~scbs/liana/.

Insert Complaint

1. Normal flow

   Inputs

   - The system must be on air.
   - The user chooses the *Insert a new complaint* option and describes the refered complaint (*Animal*, *Food*, or *Other*).
   - The user fills all information on the form and asks for the complaint to be inserted.

   Results

   - The complaint is inserted and its code is displayed so that the user takes a note. No exception shall be thrown.
   - In case the system is using the database, a query to it must be done in order to confirm that the data typed on the form were inserted (use the class `util.ManageTables`).

2. Alternative flow 1

   Inputs

   - The system must be on air.
   - The user chooses the *Insert a new complaint* option and chooses the kind of the refered complaint (*Animal*, *Food*, or *Other*).
   - The user fills only the the mandatory information (defined by the use case) on the form and asks for the complaint to be inserted.

Results

- The complaint is inserted and its code is displayed so that the user takes a note. No exception shall be thrown.
- In case the system is using the database, a query to it must be done in order to confirm that the data typed on the form were inserted (use the class `util.ManageTables`).

3. Alternative flow 2

Inputs

- The system must be on air.
- The user chooses the *Insert a new complaint* option and chooses the kind of the refered complaint (*Animal*, *Food*, or *Other*).
- The user shall provide no information on the form and asks for the complaint to be inserted.
- If requested to, the user fills the information, but only the information requested by the form.
- The latter step is repeated as many times as the system asks so that the information is filled.

Results

- A message is displayed asking him/her to fill the mandatory fields in blank, and the complaint is not inserted.
- The latter step is repeated until all mandatory information is provided, when the complaint is inserted and its code is displayed so that the user takes a note. No exception shall be thrown.
- In case the system is using the database, a query to it must be done in order to confirm that the data typed on the form were inserted (use the class `util.ManageTables`).