“A Component Testing Approach Supported by a CASE Tool”

by

Fernando Raposo da Camara Silva

M.Sc. Dissertation

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE (PE), August 2008
FERNANDO RAPOSO DA CAMARA SILVA

“A Component Testing Approach Supported by a CASE Tool”

Este trabalho foi apresentado à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

A master dissertation presented to the Federal University of Pernambuco in partial fulfillment of the requirements for the degree of M.Sc. in Computer Science.

ADVISOR: SÍLVIO ROMERO DE LEMOS MEIRA
CO-ADVISOR: EDUARDO SANTANA DE ALMEIDA

RECIFE, AUGUST 2008
Silva, Fernando Raposo da Camara
A component testing approach supported by a CASE tool / Fernando Raposo da Camara Silva. – Recife: O Autor, 2008.
98 folhas : il., fig., tab.


Inclui bibliografia e apêndices.


004 CDD (22.ed.) MEI2008-094
This work is dedicated to my parents Marcos and Marcia, my brother Caius and my dearest friends.
In the text of this dissertation the term test is used many times. And nothing can be more appropriate to define my last years as one of the hardest tests of my life!

In the long road to conclude this work, I would like to thank a lot of people and entities that made their best to support me in this journey.

I would like to thank the RiSE group to give me the support and the space to discuss the aspects that were useful to finish my work. In special, the guys whom I had the honor of working together: Bart, Fred, Vinicius, Alexandre Álvaro, Alexandre Martins, Rodrigo, Leandro, Lica, Yguaratã, Kellyton, Paulo, Ivan, Heberth, Flávio, Aline, Jarley and José Jorge (sorry if I forgot someone).

I would like to thank the people from SERPRO for giving me the time necessary to develop my master activities. In special, I am thankful to Helena Cristina Bastos, Suedy Leuthier and Janete Lipski for recognizing the importance of my work for SERPRO.

There are two persons that I have special thanks. The first one is my advisor, Silvio Meira with his visionary thoughts about life and the future. I am proud to be his student. The second person is my co-advisor, Eduardo Almeida, for his patience in giving me the directions for my work. Eduardo is the right person at the right moment all the time. Under his shoulders the RiSE group is bigger than ever was.

I would like to thank my parents, Marcos and Marcia. I am so lucky to have you with me. You are the best example of good and honest people to anyone. I swear I will try to be as good as you are.

I have few, but very good friends. So, this work is dedicated to Caius, André, Heros, Aline and Felipe. I will always remember our Sunday afternoons acting like real nerds playing RPG. Those moments made history! Also, this work is dedicated to my sweet little girl from Portugal. Nilza, only we know that
the saddest fado is the one sung via Skype... But I know that we will be together soon!

Finally, I would like to thank God for giving me the peace and health to support the pressure of writing this work. In the end, You never abandoned me!

Fernando Raposo da Camara Silva
Recife (PE)
August 2008
"Algún día, en cualquier parte, has de encontrarte contigo mismo. Y solo de ti depende, que sea la más amarga de tus horas, o tu momento mejor"

M. de Combi

[“One day, somewhere, you are going to meet yourself. And it depends only on you, that this time, would be your bitterest time, or the best time, you ever had”]
Abstract

The reuse of assets instead of building them from scratch, a process generally known as software reuse, can be considered a good option of competitive advantage for software development companies. In this context, component-based development has been one of the most used techniques applied to promote software reuse, establishing a promising component market. However, the main consolidate methods to develop components are more aware in demonstrating component development as a feasible approach and some important activities, for example, related to quality, and more specifically, to testing, are sometimes neglected. Component producers have difficulties to design their products to work properly on different contexts of use, as well as to prepare their products to be validated by third party. In the same way, component consumers have limitations to test if external components, these ones addressing specific needs of their systems under development, work as specified, and if they are able to work integrated with their systems. This dissertation presents an approach to support component testing defining activities to be performed by component producers to prepare their products to be tested by third party, as well as activities to be performed by component consumers to understand and test external components aiming their integration to systems under development. In addition, a tool was developed to aid both component producers and consumers to accomplish the activities defined by the workflows. Finally, an experimental study was conducted to evaluate the proposed approach and the tool that covers its activities.

Keywords: Software Reuse, Software Quality, Component-based development, Component testing.
A reutilização de artefatos, ao invés de construí-los do zero, processo usualmente conhecido como reuso de software, mostra-se uma boa opção de vantagem competitiva para as empresas de desenvolvimento de software. Neste contexto, o desenvolvimento baseado em componentes é uma das técnicas mais aplicadas para promover o reuso de software estabelecendo um promissor mercado de componentes. Entretanto, os métodos mais consolidados de desenvolvimento de componentes estão mais focados em demonstrá-lo como uma abordagem viável, e algumas atividades importantes, por exemplo, atividades relacionadas com qualidade, e mais especificamente, relacionadas a teste, são muitas vezes negligenciadas. Produtores de componentes têm dificuldades para preparar seus produtos para funcionar corretamente em diferentes contextos, como também para serem validados pelos consumidores que pretendem adotá-los. Similarmente, os consumidores de componentes possuem limitações para poder testar se componentes externos funcionam como especificados, e se são capazes de serem integrados aos seus sistemas. Assim, esta dissertação apresenta uma abordagem para apoiar o teste de componentes definindo atividades para guiar produtores de componentes no preparo de seus produtos para serem testados por terceiros, como também atividades para os consumidores de componentes para compreender e testar componentes externos objetivando a sua integração com seus sistemas em desenvolvimento. Por fim, é apresentada uma ferramenta para realizar as atividades definidas para produtores e consumidores. Um experimento foi conduzido avaliando a abordagem proposta e sua ferramenta relacionada.

**Palavras Chave:** Reuso de Software, Qualidade de Software, Desenvolvimento baseado em componentes, Testes em componentes.
# Table of Contents

ACKNOWLEDGEMENTS........................................................................................................ IV
ABSTRACT ...................................................................................................................... VII
RESUMO ....................................................................................................................... VIII
LIST OF FIGURES ....................................................................................................... XI
ACRONYMS.................................................................................................................. XIII

1. INTRODUCTION ........................................................................................................ 1
   1.1. MOTIVATION ......................................................................................................... 2
   1.2. GOAL OF THE DISSERTATION .............................................................................. 3
   1.3. OVERVIEW OF THE PROPOSED SOLUTION .......................................................... 3
      1.3.1. Context ........................................................................................................... 5
   1.4. OUT OF SCOPE .................................................................................................... 6
   1.5. STATEMENT OF THE CONTRIBUTIONS ............................................................... 7
   1.6. ORGANIZATION OF THE DISSERTATION ........................................................... 8

2. COMPONENT QUALITY AND TESTING: AN OVERVIEW ....................................... 9
   2.1. SOFTWARE QUALITY ASSURANCE .................................................................... 10
   2.2. COMPONENT QUALITY ....................................................................................... 10
   2.3. SOFTWARE COMPONENT TESTING .................................................................... 13
   2.4. SUMMARY .......................................................................................................... 15

3. SOFTWARE COMPONENT TESTING: A SURVEY ................................................. 16
   3.1. APPROACHES AIMING AT THE CAUSES OF LACK OF INFORMATION ................. 17
   3.2. APPROACHES AIMING AT THE EFFECTS OF LACK OF INFORMATION .............. 20
   3.3. COMPONENT TESTING TOOLS ........................................................................... 23
   3.4. SUMMARY OF THE STUDY ................................................................................ 27
   3.5. SUMMARY .......................................................................................................... 30

4. A COMPONENT TESTING APPROACH SUPPORTED BY A CASE TOOL 32
   4.1. INTRODUCTION ..................................................................................................... 33
   4.2. OVERVIEW OF THE PROPOSED APPROACH ....................................................... 33
   4.3. OBJECTIVES ......................................................................................................... 35
   4.4. APPROACH REPRESENTATION ............................................................................ 36
   4.5. COMPONENT PRODUCER WORKFLOW ............................................................... 37
      4.5.1. Collect Information ....................................................................................... 38
List of Figures

FIGURE 1.1 – RISE FRAMEWORK FOR SOFTWARE REUSE.......................... 6
FIGURE 3.1 – COMPONENT TEST BENCH (BUNDELL ET AL., 2000) ...... 24
FIGURE 3.2 – SOFTWARE COMPONENT TESTING TIMELINE:
APPROACHES AND TOOLS.................................................................27
FIGURE 4.1 – WORKFLOW OF ACTIVITIES AT COMPONENT PRODUCER
SIDE 38
FIGURE 4.2 – WORKFLOW OF ACTIVITIES AT COMPONENT CONSUMER
SIDE 44
FIGURE 4.3 – A SYSTEM UNDER DEVELOPMENT INTERACTING WITH
TWO DIFFERENT CANDIDATE COMPONENTS................................. 45
FIGURE 4.4 – INTERACTIONS BETWEEN FUNCTIONALITIES OF A
CANDIDATE COMPONENT AND TARGET SYSTEM FUNCTIONALITIES . 46
FIGURE 4.5 – EXAMPLE OF TEST PROCESS FROM BS – 7925-2 .......... 47
FIGURE 4.6 – PUG INTEGRATED TO ECLIPSE ENVIRONMENT .......... 50
FIGURE 4.7 – CTP INTEGRATED TO ECLIPSE ENVIRONMENT .......... 51
FIGURE 4.8 – ECLIPSE EXTENSIBLE ARCHITECTURE ......................... 55
FIGURE 4.9 – ARCHITECTURE OF THE PLUG-INS .......................... 57
FIGURE 4.10 – PUG FUNCTIONALITIES ............................................. 58
FIGURE 4.11 – USAGE CAPTURE ALGORITHM PSEUDO-CODE .......... 59
FIGURE 4.12 – COMPONENT OSDREADWRITE MAIN INTERFACE ....... 60
FIGURE 4.13 – PREPARATION OF A COMPONENT TO BE TESTED BY
THIRD PARTY..................................................................................... 61
FIGURE 4.14 – REPRESENTATION OF USAGE MANIFEST DTD .......... 62
FIGURE 4.15 – CTP FUNCTIONALITIES .............................................. 63
FIGURE 4.16 – CTP AND ITS FUNCTIONALITIES ................................. 64
FIGURE 4.17 – INTEGRATION WITH DEVELOPMENT ENVIRONMENT
AND IDENTIFICATION OF INTERACTIONS ...................................... 65
FIGURE 4.18 – CTP PATTERN VERIFIER ............................................. 66
FIGURE 4.19 – CTP CODE ASSIST ...................................................... 67
FIGURE 5.1 – DIFFICULTIES AT COMPONENT PRODUCER SIDE ...... 85
FIGURE 5.2 – DIFFICULTIES AT COMPONENT CONSUMER SIDE .... 87
# List of Tables

<table>
<thead>
<tr>
<th>Table Number</th>
<th>Table Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE 1.1</td>
<td>STRATEGIES OF COMPONENT TESTING APPROACHES MAPPED TO ACTIVITIES</td>
<td>4</td>
</tr>
<tr>
<td>TABLE 2.1</td>
<td>MAIN SOFTWARE STANDARDS FOR EVALUATING PRODUCTS</td>
<td>11</td>
</tr>
<tr>
<td>TABLE 2.2</td>
<td>COMPONENT QUALITY MODEL (ALVARO ET AL., 2006)</td>
<td>12</td>
</tr>
<tr>
<td>TABLE 3.1</td>
<td>APPROACHES PROVIDING TEST INFORMATION TO COMPONENT CONSUMERS</td>
<td>19</td>
</tr>
<tr>
<td>TABLE 3.2</td>
<td>APPROACHES AIMING THE EFFECTS OF LACK OF INFORMATION</td>
<td>22</td>
</tr>
<tr>
<td>TABLE 3.3</td>
<td>CHARACTERISTICS OF COMPONENT TESTING TOOLS</td>
<td>26</td>
</tr>
<tr>
<td>TABLE 4.1</td>
<td>INFORMATION TO SUPPORT FURTHER TESTING DERIVED FROM DIFFERENT TYPES OF ASSET</td>
<td>42</td>
</tr>
<tr>
<td>TABLE 5.1</td>
<td>ASSIGNATION OF ROLES TO SUBJECTS</td>
<td>74</td>
</tr>
<tr>
<td>TABLE 5.2</td>
<td>DISTRIBUTION OF COMPONENTS TO PHASES AND SUBJECTS</td>
<td>81</td>
</tr>
<tr>
<td>TABLE 5.3</td>
<td>SUMMARY OF THE COMPONENTS</td>
<td>82</td>
</tr>
<tr>
<td>TABLE 5.4</td>
<td>SUBJECT'S PROFILE IN THE EXPERIMENTAL STUDY</td>
<td>83</td>
</tr>
<tr>
<td>TABLE 5.5</td>
<td>RESULTS FOR RELATIVE TEST COVERAGE</td>
<td>84</td>
</tr>
<tr>
<td>TABLE 5.6</td>
<td>RESULTS FOR EFFORT VARIATION (IN MEN-HOURS)</td>
<td>88</td>
</tr>
<tr>
<td>TABLE 5.7</td>
<td>HYPOTHESES ANALYSIS</td>
<td>92</td>
</tr>
</tbody>
</table>
## Acronyms

<table>
<thead>
<tr>
<th>Terms</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT</td>
<td>Built-In-Testing</td>
</tr>
<tr>
<td>CBSD</td>
<td>Component-Based Software Development</td>
</tr>
<tr>
<td>COTS</td>
<td>Commercial Off-the-Shelf</td>
</tr>
<tr>
<td>CQM</td>
<td>Component Quality Model</td>
</tr>
<tr>
<td>CTP</td>
<td>Component Test Perspective</td>
</tr>
<tr>
<td>GQM</td>
<td>Goal Question Metric</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>ISO</td>
<td>International Standard Organization</td>
</tr>
<tr>
<td>OCL</td>
<td>Object Constraint Language</td>
</tr>
<tr>
<td>OSC</td>
<td>Open Source Component</td>
</tr>
<tr>
<td>PUG</td>
<td>Program Usage Generator</td>
</tr>
<tr>
<td>QIP</td>
<td>Quality Improvement Paradigm</td>
</tr>
</tbody>
</table>
Software reuse is considered a “generic” denomination, encompassing a variety of techniques aimed at getting the most from design and implementation work. The top objective is to avoid reinvention, redesign and reimplementation when building a new product, by capitalizing on previous done work that can be immediately deployed in new contexts. Therefore, better products can be delivered in shorter times, maintenance costs are reduced because an improvement to one piece of design work will enhance all the projects in which it is used, and quality should improve because reused software have been well tested (D'Souza et al., 1998).

Software componentization is one possible approach to promote software reuse. Thus, Component-Based Development (CBD) appears not only as an action to standardize the construction of components with the premise of software reuse, as well as an answer to the claim that CBD allows the reduction of cost and time to market, while increasing software quality.

Even so, according to Wu et al. (Wu et al., 2001) much work has been devoted to develop infrastructure for the construction of component-based software, striving to achieve quality objectives such as interoperability, extensibility, implementation transparency. However, techniques for testing component-based systems have not been well developed.

The instantiation of this problem is the main subject of this dissertation, which will discuss it in detail, beginning with the problem formulation, passing through the state-of-the-art of existing solutions, ending with the current proposal and its validation. This chapter describes the focus of this work and starts by presenting its motivation and a clear definition of the problem.
1.1. Motivation

According to Szyperski (Szyperski, 1998) the definition of software component is: “...a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties”. Apperly (Apperly, 2001) states that in companies adopting CBD, there is a produce-manage-consume process where producers are focused on producing and publishing to market components to be reused; and consumers are concentrated on finding and reusing components to reduce their development cost. However, the evaluation of components available in the market, which is often short on standards and established vendors, can be difficult because current offerings might not meet the organization’s specific needs and also due to inadequate vendor support (Ulkuniemi et al., 2004). In addition, current CBD approaches do not address well activities related to component quality. Later factors can be the source of the common belief that component markets are not reliable preventing the emergence of a mature software component market (Trass et al., 2000).

Thus, one of the main barriers to increase the extent of component markets and the extensive use of CBD is related to quality assurance. Among the techniques used to assure the components quality, component testing can be considered one of the most effective (Weyuker, 1998). According to Bertolino et al. (Bertolino et al., 2003), the conduction of deployment testing by the component customers would help to improve the “component trust”, since component developers cannot cover all possible behavior a component can assume in the system in which it will be later assembled.

Furthermore, there are many approaches that address component testing (more details about component test approaches will be discussed in Chapter 3). However, inadequacies in existing techniques are not related to characteristics associated to traditional software development, they occur due to characteristics that are inherent to software components. (Rehman et al., 2007).

When considering component testing nowadays, some issues arise such as: (i) the lack of information exchanged between component producers and consumers is a crucial problem that limits the capacity of component consumers
to properly test candidate components (Beydeda et al., 2003b); (ii) the frequent source code unavailability due to organizational or legal restrictions limits, for instance, the execution of white-box tests; (iii) how to deal with the inherent diversity of components and their impact on the development of component testing standards; (iv) there are several tools to support component testing such as (Bundell et al., 2000) and (Teixeira et al., 2007), however they are not integrated with development environment, limiting, for instance, the conduction of integration tests; and, finally, (v) how to construct test suites containing generic test cases to validate components with similar behavior but with different usage assumptions (Bertolino et al., 2003).

1.2. Goal of the Dissertation

According to the issues gathered from the discussion of the previous section, the work described in this dissertation focuses in achieving the following goal:

This work defines, based on the-state-of-the-art in the area, an approach to support component testing aiming to aid component producers to prepare their components to be tested by third party, as well as to aid component consumers to understand and test external components checking if they correctly fulfill their needs. Moreover, a tool to be used by both component stakeholders to perform the activities of the proposed approach was specified, designed and implemented.

1.3. Overview of the Proposed Solution

In order to achieve the goals stated in the previous section, we performed a study on component testing approaches analyzing their flaws and future trends. In addition, we analyzed component testing tools discussing their strong and weak points.

In general, the results of our analysis associate methodologies or strategies adopted by current component testing approaches to macro activities that can be performed by component producers or by component consumers (Table 1.1). Those identified activities form the basis of two proposed
workflows\textsuperscript{1}. One workflow details activities to be performed by component producers to prepare their components to be understood and tested by someone that was not involved with the development of the component, but is considering adopting it. The other workflow encompasses activities to be performed by component consumers to understand candidate components, check if candidate components work as designed and analyze if they work integrated to the system under development fulfilling acceptance criteria.

### Table 1.1 - Strategies of Component Testing Approaches Mapped to Activities

<table>
<thead>
<tr>
<th>Stakeholder</th>
<th>Activity</th>
<th>Strategy</th>
<th>Related Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Component Producers</strong></td>
<td><strong>Information Analysis</strong></td>
<td>- Test case creation</td>
<td>(Harrold et al., 1999)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Flow-control data capture</td>
<td>(Liu et al., 1999)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Invariants definition</td>
<td>(Wang et al., 1999)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Regular Expressions</td>
<td>(Orso et al., 2000)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Textual Descriptions</td>
<td>(Gao et al., 2002)</td>
</tr>
<tr>
<td></td>
<td><strong>Information Publishing</strong></td>
<td>- Pack information in standard format</td>
<td>(Bundell et al., 2000)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Attach information to component or to repository</td>
<td>(Edwards, 2001)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(Teixeira et al., 2007)</td>
</tr>
<tr>
<td><strong>Component Consumers</strong></td>
<td><strong>Understand Component Usage</strong></td>
<td>- Usage assumptions</td>
<td>(Liu et al., 1999)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Method call order</td>
<td>(Orso et al., 2000)</td>
</tr>
<tr>
<td></td>
<td><strong>Elaborate Integration Test Cases</strong></td>
<td>- Elaborate test cases according to acceptance criteria</td>
<td>(Bertolino et al., 2003)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(Beydeda et al., 2003b)</td>
</tr>
<tr>
<td></td>
<td><strong>Test Case execution and Evaluation</strong></td>
<td>- Test case execution</td>
<td>(Wang et al., 1999)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Analysis of results</td>
<td>(Bundell et al., 2000)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(Teixeira et al., 2007)</td>
</tr>
</tbody>
</table>

In addition, in order to automate the activities of the defined workflows, a tool integrated to the development environment was implemented. It applies a combination of techniques (automatic, computer-assisted and manual) aiming to reduce the programming overheads on both stakeholder sides. Taking

---

\textsuperscript{1} In Chapter 4 all these workflows are discussed in details.
advantage of Eclipse\textsuperscript{2} extensible architecture to the creation of tools integrated to
development environment, the tool is composed of two modules implemented as Eclipse plug-ins. One module, called **PUG (Program Usage Generator)** is at component producer side aiding component producers to collect data, analyze and pack relevant information to support further testing of the component. On the other side, with component consumers, a module called **CTP (Component Test Perspective)**, helps consumers to understand a candidate component, to construct test cases and identify interactions between the system under development and the component to validate if the candidate component fulfills its necessities.

**1.3.1. Context**

This work is part of a broader reuse initiative promoted by the Reuse in Software Engineering research group (RiSE\textsuperscript{3}) (Almeida et al., 2004). According to (Almeida et al., 2007), “RiSE’s goal is to develop a robust framework for software reuse in order to enable the adoption of a reuse program. The proposed framework has two layers, as shown in Figure 1.1. The first layer (on the left side) is formed by best practices related to software reuse. Non-technical aspects, such as education, training, incentives, program to introduce reuse, and organizational management are considered. This layer constitutes a fundamental step before the introduction of the framework in organizations. The second layer (on the right side), is formed by important technical aspects related to software reuse, such as processes, environment, and tools.”

---

\textsuperscript{2} Eclipse - An Open Development Platform: http://www.eclipse.org/

\textsuperscript{3} RiSE – Reuse in Software Engineering group – http://www.rise.com.br/research
As can be seen in Figure 1.1, the RiSE project addresses other reuse aspects not included in the scope of this dissertation, such as software reuse processes (Almeida, 2007) and component repository management (Burégio, 2006). Besides, there are tools proposed by the project, including domain analysis tools (Lisboa et al., 2007), reverse engineering (Brito et al., 2007) and component search engines (Garcia et al., 2006), (Mascena, 2006), (Vanderlei et al., 2007).

These efforts are coordinated and will be integrated in a full-fledged enterprise scale reuse solution. The role of the Component Testing Approach in the RiSE project is to be part of the activities that compose the **Component Certification Process**. The main idea of a certification process is to have a group independent from development team to evaluate the quality of software components in an efficient way.

### 1.4. Out of Scope

As more third-party components are available in the commercial market, more software developers started to use CBD approaches to develop their products. They have encountered new issues and challenges in testing of software components and component-based programs (Gao, 1999).

Components and tests appear in many other contexts together, however, due to scope limitations, there are other possibilities and work directions that were discarded in this dissertation. Thus, the following issues are not directly addressed by this work:
• **Methods for component-based development:** There are several methods that address the development of software components such as (D’Souza et al., 1998), (Atkinson et al., 2000) and (Chessman et al., 2001). Tests can support the development of reliable reusable components that may be plugged into different contexts. However, how to adapt current methods to develop components inserting testing to the quality assurance phase of component development processes is considered out of scope. The approach presented in this dissertation is intended to be independent of development method;

• **Software Product Lines:** There are also components that are the output of Software Product Lines (SPL). Briefly, a SPL is a set of software-intensive systems sharing a common, managed set of features (Clements et al., 2001). In this context, there are processes to manage the production of similar assets with their own activities related to component testing. Rise group has research branch dedicated exclusively to software product lines. However, this is not discussed in this dissertation; and

• **Component Search:** Our approach to test components can be applied to support the selection of the most reliable component from a set of candidate components that fulfill a specific necessity of an organization. However, how components were discovered in an open-source or commercial repository is considered out of scope. As cited previously, RiSE group has a research branch that focuses on search engines and due to scope limitation this subject is not discussed in this dissertation.

### 1.5. Statement of the Contributions

Basically, the main contributions of this work are:

• The definition of an approach to support component testing providing guidelines for producers to prepare a component to be tested by third party; and guidelines for component consumers to understand and test external components before deciding upon their integration to a system under development;
• Two Eclipse plug-ins, integrated to development environment to support component stakeholders. One plug-in called **PUG (Program Usage Generator)** to support component producers to collect information about their components; and another one called **CTP (Component Test Perspective)** to support component consumers to understand external components and elaborate test cases; and

• The definition, planning, analysis of an experimental study in order to evaluate the proposed approach.

### 1.6. Organization of the Dissertation

In order to find some insights for the questions to be investigated, this dissertation investigates the current state-of-the-art on software component testing, in an attempt to analyze this area and with the objective of looking for research directions to follow. Besides this chapter, this dissertation is organized as follows.

Chapter 2 presents a brief overview of the area of component quality and how it is related to component testing. The main concepts of these topics are considered.

Chapter 3 presents the survey of the state-of-the-art of the software component testing area.

Chapter 4 describes the proposed approach, showing in detail the guidelines to be performed by component stakeholders in a form of workflows of activities.

Chapter 5 presents an experimental study related to the proposed approach analyzing possible benefits and/or issues of its applicability.

Chapter 6 summarizes the main contributions of this work, presents the related works, academic contributions, the concluding remarks and directions for future work.
According to a recent Gartner Group research\textsuperscript{4}, “The lack of testing and Quality Assurance standards, as well as a lack of consistency, often lead to business disruption, which can be costly”. In the same way, (Beizer, 1990) stressed that testing is considered as one of the most costly activities in a system under development, sometimes exceeding fifty per cent of the total development costs.

The complexity and costs to test can be even increased in component-based development. While components can be used in different contexts, component producers can find difficulties to check all possible contexts which their product will be inserted. The widespread reuse of a software component with poor quality can cause problems to organizations. In addition, the improper reuse of software components of good quality may also be disastrous (Jezequel et al., 1997). Thus, testing and quality assurance are therefore critical for both software components and component-based systems.

In this Chapter a brief analysis related to testing is conducted in two aspects. One aspect is the relation of testing and quality assurance in software engineering projects in general; and the other is the similarities and differences between traditional COTS testing and testing in component-based development to support quality assurance.

\textsuperscript{4} Gartner Research, August 17, 2007, ID Number G00150394, page 4.
2.1. Software Quality Assurance

In general, quality can be considered an abstract or an intangible feature— it can be discussed, felt and judged, but cannot be precisely weighed or measured (Kan, 2002). This view is in vivid contrast to the professional view held in the discipline of quality engineering that quality can, and should, be operationally defined, measured, monitored, managed, and improved.

According to (Kan, 2002), the narrowest sense of product quality is commonly recognized as lack of "bugs" in the product. It is also the most basic meaning of conformance to requirements, because if the software contains too many functional defects, the basic requirement of providing the desired function is not met. Techniques such as software testing can be applied to check this level of conformance to requirements, or in other words, the lack of “bugs”, reducing the supposed subjectiveness of software quality.

However, software testing alone does not assure quality. Quality comes from the people who build the product (Kaner et al., 2001). Thus, software testing is just one of the many tools used to ascertain the quality of software.

Furthermore, to successfully perform testing, we must know what is to be tested, how to perform the tests, and what the system is expected to accomplish. In a traditional system development process, this implies that the specifications from the analysis and design activities are used to define test cases. A problem in obtaining the necessary test cases is that the analysis results must be translated into concrete tests during the implementation activity, both on the module and on the system level.

Next Sections discuss the challenges of assuring quality in the context of software components and how can test support quality assurance of component based software systems.

2.2. Component Quality

Component-based development is being used in a wide variety of application areas. Thus, the correct operations of the components are often critical for business success and, in some cases, human safety. In this sense,
nowadays, the reuse community is active and focused on solving this issue. Thus, the assessment and evaluation of software components has become a compulsory and crucial part of any CBSD lifecycle.

Literature provides several approaches that address improving the quality of software components such as component certification (Councill, 1999) and (Alvaro et al., 2007), design-by-contract (Atkinson et al., 2002) and testing (Bertolino et al., 2003) among others. However, given the inherent heterogeneity of components there is no consensus about how to evaluate them. Frequently, software component evaluation can occur through models or standards that measure its quality. These models describe and organize the component quality characteristics that will be considered during the evaluation, and provide guidelines to be followed by its users. In order to provide a general vision, Table 2.1 shows current standards for evaluating software products.

The SQuaRE project (ISO/IEC 25000, 2005), which revises standard ISO/IEC 12119, was created specifically to make two standards (ISO/IEC 9126 and ISO/IEC 14598) converge. It presents a generic software quality model that can be applied to any software product by tailoring it to a specific purpose. However, the main issue of SQuaRE, and other existing international standards is that they provide very general quality models and guidelines, and are very difficult to apply to specific domains such as Component-Based Development (Alvaro et al., 2007).

### Table 2.1 – Main Software Standards for Evaluating Products

<table>
<thead>
<tr>
<th>Standard</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>(ISO/IEC 9127, 1988)</td>
<td>Describes User Documentation and Cover Information Supplied With Software Packages</td>
</tr>
<tr>
<td>(ISO/IEC 9126, 1991)</td>
<td>Software Products Quality Characteristics</td>
</tr>
<tr>
<td>(ISO/IEC 12119, 1994)</td>
<td>Quality Requirements and Testing for Software Packages</td>
</tr>
<tr>
<td>(ISO/IEC 14598, 1998)</td>
<td>Guides to evaluate software products</td>
</tr>
</tbody>
</table>

Considering the existent gap in component evaluation, Alvaro et al. (Alvaro et al., 2006), proposed a Component Quality Model. The CQM, presented in Table 2.2, has the purpose of determining which quality characteristics should be considered, defining the essential CBD characteristics,
and which sub-characteristics are necessary. Such a quality model serves as a basis for determining if a component has a number of quality attributes that fulfills stakeholder needs.

Analyzing the CQM model of Table 2.2, it can be noted that a variety of techniques can be applied to evaluate some of the defined quality sub-characteristics. For instance, code inspection can use source code or other documentation to measure some aspects of component understandability. In the same way, a benchmark analysis can be conducted to measure time to market. Testing is another valuable technique that can be used to measure component quality. However, often, only the final users of components can effectively test how the components will react with the system they are being integrated in. If the quality assurance of in-house developed software is a demanding task, doing it with software developed elsewhere, often without having access to its source code and detailed documentation, presents an even greater concern (Goulão et al., 2002).

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>Sub-Characteristics (Runtime)</th>
<th>Sub-Characteristics (Life cycle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functionality</td>
<td>- Accuracy</td>
<td>- Suitability</td>
</tr>
<tr>
<td></td>
<td>- Security</td>
<td>- Interoperability</td>
</tr>
<tr>
<td>Reliability</td>
<td>- Fault Tolerance</td>
<td>- Compliance</td>
</tr>
<tr>
<td></td>
<td>- Recoverability</td>
<td>- Self-contained</td>
</tr>
<tr>
<td>Usability</td>
<td>- Configurability</td>
<td>- Maturity</td>
</tr>
<tr>
<td>Efficiency</td>
<td>- Time Behavior</td>
<td>- Understandability</td>
</tr>
<tr>
<td></td>
<td>- Resource Behavior</td>
<td>- Learnability</td>
</tr>
<tr>
<td>Maintainability</td>
<td>- Stability</td>
<td>- Operability</td>
</tr>
<tr>
<td>Portability</td>
<td>- Deployability</td>
<td>- Changeability</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Replaceability</td>
</tr>
<tr>
<td>Marketability</td>
<td></td>
<td>- Adaptability</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Reusability</td>
</tr>
</tbody>
</table>

- Development Time
- Cost
- Time to Market
- Affordability
Councill (Councill, 1999) considered software components as a commodity provided by component producers that must be delivered to component consumers with high level of quality. Thus, in component quality, the main issue is **Trust** and the main question is – “How can software producers and consumers assure trustworthiness?”. Next section is dedicated to discuss the main concepts involving software components testing, which is an attempt to achieve this kind of trust.

### 2.3. Software Component Testing

Szyperski (Szyperski, 2003), emphasizing the necessity of component testing stated: “In a sense, a correct component is 100% reliable and a component that has the slightest defect is actually incorrect and thus 100% unreliable.” According to (Weyuker, 1998), testing is sometimes advocated as the major quality assurance mechanism for components. In addition, if the component to be used is not adequately described, testing may be the only practical approach, although very resource consuming.

This means that component testing can play an essential role for a proper maturation of component market, since even if there are components ready for reuse, and repositories to store those components are available, no client would neither buy nor even search for components with low quality standards.

Thus, to support organizations to test components, the British Standard BS 7925-2 - *Standard for Software Component Testing* (BS 7925-2, 2001) can be considered to provide guidelines to the process of component testing. The standard defines a component as a minimal program for which a separate specification is available. It has the objective to enable the measurement and comparison of testing performed on software components. However, the standard can be considered limited and needs to be improved in order to capture all the concerns involved in developing software components (Gao et al., 2003). For instance, it lacks guidelines to test components that will be distributed to be integrated by third-party.

The limitations of BS 7925-2 demonstrate that testing components is not an easy task. Thus, traditional software testing technique should have its applicability to components analyzed, because according to (Gao et al., 2003),
there are a number of misunderstandings, and myths about component testing that are summarized below:

**Myth 1:** A software component can be tested once and reused everywhere. The problems of reusing a component in a new context may cause disasters. This issue is best represented by the Ariane rocket incident (Jezequel et al., 1997). Thus, testing a component must have two major purposes. The first is to validate the selected component to see if its functions and behaviors meet the given specifications in a new context environment. The second one is to make sure that it is reused properly in the specific context;

**Myth 2:** If component producers have adequately tested a component, then it can be used by a third party in a project without validation. Unfortunately, component producers cannot test a component exhaustively before releasing it to consumers because they cannot predict all possible contexts of use of their product. What producers can do is to define and perform an adequate test set based on a predefined test model and criteria. This test set only can be executed in a limited number of context environments to exercise a few usage patterns. To achieve adequate testing for component reuse, therefore, a user must define an adequate test model and test set for the component in a new context to focus on its reuse patterns;

**Myth 3:** Only component producers need a rigorous component test process. Component consumers must establish an internal component validation process to assure the quality of components before they are integrated into a system. A dedicated validation process is needed for reused components to focus on reuse functions and behaviors in a new context;

**Myth 4:** Component testing in CBD is the same as the unit testing of software modules. Unlike traditional unit testing, component producers should pay attention to the new features of modern components, such as reusability, interoperability, built-in packaging, customization, and deployment mechanisms. On the other side, consumers have different objectives and focuses, for instance check integration of candidate components and the system under development.
Thus, it can be noted that testing components involves two types of stakeholders. One type is component producers that should evaluate their components to satisfy a possibly diverse spectrum of unknown users regarding function, performance, interface, application environment and operating environments. In addition, producers should provide enough resources to support third-party testing.

Another type of stakeholder is component consumers that should evaluate, sometimes based on limited access to information, if an external component works properly in conjunction to their system under development fulfilling its necessities.

2.4. Summary

This chapter presented the main concepts about software quality and, in the context of this dissertation, component quality assurance supported by component testing. Component testing can be an error-prone process due to its complexity, lack of standards and guidelines.

Next Chapter presents a survey of the state-of-the-art and practice in component testing, discussing current approaches, tools to test components and strong and weak points, in order to define a base to the component testing approach defined in this work.
Software Component Testing: A Survey

The field of software component testing is vast; since it refers to all activities that are related to test in the scope of a component-based development project. This comprises tests and testing activities during component engineering, carried out by the producer of a component, as well as all testing activities performed by the consumer of the component during component integration and deployment. According to Szyperski (Szyperski, 1998), “Testing of software components (and component based software systems) is possibly the single most demanding aspect of component technology”. In this regard, and in order to understand plausible answers to insights for the questions discussed in Chapter 1, this Chapter presents a survey of the state-of-the-art in software component testing.

The area of component quality started to blossom in the end of the nineties and beginning of this century when methods to develop components were presented (D'Souza et al., 1998), (Atkinson et al., 2000) and (Chessman et al., 2001). Thus, COTS software testing and component based software engineering converged to a common sub-area of the former: Software Component Testing. However, the literature contains several work related to software components quality achievement (Alvaro et al., 2005) such as model-driven architecture, component contracts, component certification, among others. Since the spotlight of this dissertation is on component testing, this survey does not cover these work, which are related to other specific aspects of component quality.

One possible classification of component testing approaches, as suggested by (Beydeda et al., 2003b), is to classify approaches in two categories related to how they take into account the lack of information between
component producers and consumers. The lack of information is reported by (Bertolino, 2007) as the crucial problem related to component testing, because while component interfaces are described according to specific component models, they do not provide enough information for functional testing. In addition, this Chapter presents an analysis of component testing tools in section 3.3.

### 3.1. Approaches Aiming at the Causes of Lack of Information

Some approaches try to minimize the dependence of component consumers on the information provided by component producers. In this way, the general strategy behind it is to aggregate valuable information to the component in order to facilitate test activities at component consumer side.

In 1999, Liu et al. (Liu et al., 1999) as a complement of component retrospectors effort (Liu et al., 1998), proposed an algorithm to capture information related to the usage of the component inside source code snippets. They emphasized that component usage is another piece of important information frequently absent to component consumers when developing tests to validate a component. The key idea is that component producers design a component according to certain usage assumptions. Information like correct usage of the component is critical to a successful integration with the rest of the system; however, expected component usage is never explicitly specified. Frequently, usage information may be expressed in the form of source code comments, component interface specification or spread throughout documentation such as user manual or JAVADOC⁶ for instance.

Two interesting points are associated to this approach; one is that usage specification can aid component consumers to elaborate their own tests with their own acceptance criteria since usage specification is not a static test case elaborated by the producer of the component. Thus, usage information is more flexible than simply attaching test cases related to the producer scope. The second point is that by providing usage specification, testers may have tool support to automatically verify the usage of application software that uses that

---

⁶ Javadoc Tool Homepage - http://java.sun.com/j2se/javadoc/ (last visit: June 1, 2008)
component, in other words, they can check if the tests they have elaborated do not violate component usage.

In the same year, Harrold et al. (Harrold et al., 1999) proposed an approach for analyzing and testing component-based systems. This work suggests that a component producer must have tools to capture component summary information such as program dependencies among its statements, information about exceptions handling that can help consumers to elaborate test cases and data-flow information and to measure test-suite coverage.

In the same way, in 2000, Orso et al. (Orso et al., 2000) suggested another category of summary information to support component testing called component metadata. It can be defined as information added within the component to increase consumer analysis capability and thus facilitate testing. Augmenting the component with methods which provide functionality to query the types of the metadata available and retrieve a specific type of metadata can be a possible implementation.

The approach is not related to component testing exclusively as it defines an infrastructure that lets the component producer add to the component different types of data that are needed in a given context (i.e., deployment, testing and debugging) or for a given task. Basic properties of the metadata approach are: (i) it is originated from the component producer; (ii) it is packaged in a standard way with the component; and (iii) it is processed by tools. One issue of this approach that must be considered is the effort taken by component producers to gather and package all the information that compounds the metadata.

In 2001, Edwards (Edwards, 2001) proposed the Reflective Wrapper approach that analyzes the best way to package specification and verification information of formally specified components for distribution to component consumers. The approach suggests to component producers to take the benefits of reflection mechanisms that many OO languages have to provide information about its own internal structure, behavior or implementation and uses it to provide information to testers at consumer’s side. The information provided may vary from pre- and post-conditions for each method from exported interface to object abstract invariant.
One interesting observation of this approach is to comply with Szyperski’s (Szyperski, 1998) definition of software component: “A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only”. By providing introspective access to component’s imported interface producers are providing for component testers access to contracts between the component and other lower-level components on which it relies on.

A summary of component test approaches characteristics related to approaches that aim at the causes of lack of information is presented in Table 3.1.

**Table 3.1 - Approaches Providing Test Information to Component Consumers**

<table>
<thead>
<tr>
<th>Approach</th>
<th>Target User</th>
<th>Information Collection</th>
<th>Source code Dependence</th>
<th>Architecture Dependence</th>
<th>Information Provided</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retrospectors</td>
<td>Producer</td>
<td>Algorithmically</td>
<td>No</td>
<td>Low</td>
<td>Usage Information</td>
</tr>
<tr>
<td>(Liu et al., 1999)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Metadata</td>
<td>Producer</td>
<td>Automatic /</td>
<td>No</td>
<td>High</td>
<td>Data-flow Coverage</td>
</tr>
<tr>
<td>(Harrold et al., 1999)</td>
<td></td>
<td>Semi-automatic</td>
<td></td>
<td></td>
<td>Exception Handling</td>
</tr>
<tr>
<td>(Orso et al., 2000)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Dependence among</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>statements</td>
</tr>
<tr>
<td>Reflective Wrapper</td>
<td>Producer</td>
<td>Reflection</td>
<td>No</td>
<td>High</td>
<td>Pre- and post</td>
</tr>
<tr>
<td>(Edwards, 2001)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>conditions</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Context Dependencies</td>
</tr>
</tbody>
</table>

Based on the characterization of component test approaches from Table 3.1, some conclusions can be considered:

- A similar aspect of the presented approaches is that information about how component functionalities work is preserved. The information captured from a component and provided by its producer can be used by component consumers regardless of the presence of the source code. This possible source code absence is noted by (Weyuker, 1998) as one of the biggest barriers in component testing;
• In general, the approaches do not consider component consumer side. They are focused on packing a variety of data but do not cover how the information can be presented, analyzed and properly used by consumers; and

• Some approaches are too connected to architecture designs or formal specification of the component, for instance (Edwards, 2001). This may reduce the use of the approach to less bureaucratic component developers or agile method adopters. Moreover, this condition inhibits the use of the approach, for instance, on legacy components because to comply with the approach a component would have to be re-engineered.

In general, the approaches try to reduce the programming overhead of component producers to capture and pack valuable information to support tests on component consumer side by using algorithms or other techniques to capture execution trace, method call dependencies among other information.

3.2. Approaches Aiming at the Effects of Lack of Information

As it will be discussed, approaches aiming at the effects of lack of information to support testing at component consumer side, in spite of providing more information within the component, try to increase component testability by adding executable test cases that are built in the component together with the normal functions; or try to equip the component with a specific testable architecture that allows component consumers to easily execute test cases.

In 1999, Wang et al. (Wang et al., 1999) presented the Built-in Test Approach (BIT) that is based on the construction of test cases inside component source code as additional functionalities. A component can operate in two modes: normal mode and maintenance mode. In maintenance mode, component consumers can invoke dedicated methods that provide access to test cases constructed to test the component whereas in normal mode testing features access are hidden.
Another example of BIT approach, called component+, was presented in 2002 by Atkinson et al. (Atkinson et al., 2002). It is related to contract testing in model-driven component-based development. The tests provided within the component can check if the environment that a component is to be plugged by consumers does not deviate from that which it was developed to expect.

In the same year, Gao et al. (Gao et al., 2002) proposed the testable beans approach. A testable bean has two parts: one containing the functionalities that can be reused and another supporting component testing. To be considered a testable bean, a component must provide: (i) Test interfaces: to allow testers, test scripts, or test tools to interact with a testable bean to control test operations, including test setup, test execution, and test result validation; (ii) Built-in test code: to run tests, compare results with expected results and invoke basic test functions; (iii) Tracking operation interfaces: to enable component testers to monitor component trace and test results; and (iv) Built-in tracking code: which is program code that interacts with the component-tracking interface to monitor component behaviors.

This approach makes components increase in complexity since there is no decoupling of test-dedicated parts and functional parts. Also, an analysis of the feasibility of this approach is important in areas like mobile or portable applications that posses memory constraints requirements. The entire testable bean infrastructure makes component size to be increased in memory and disk space.

In 2003, Beydeda et al. (Beydeda et al., 2003a) presented the STECC (Self-Testing COTS Components) method that enhances components with functionality specific to testing tools. Component source code is parsed to capture relevant parts that are packed within the component. This information in conjunction with component consumer tester input is used by algorithms to generate test cases related to the context of the consumer. Although STECC method was recently discontinued, this is one of the few approaches that focused on supporting consumers to elaborate its own test cases.
A summary of component test approaches characteristics followed by some conclusions related to approaches that aiming at the effects of lack of information is presented in Table 3.2.

**Table 3.2 - Approaches Aiming the Effects of Lack of Information**

<table>
<thead>
<tr>
<th>Approach</th>
<th>Target User</th>
<th>Information Provided</th>
<th>Acceptance Criteria</th>
<th>Source code Dependence</th>
<th>Architecture Dependence</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT                 (Wang et al., 1999)</td>
<td>Producer</td>
<td>Static Test Cases</td>
<td>Producer</td>
<td>No</td>
<td>High</td>
</tr>
<tr>
<td>Contract            (Atkinson et al., 2002)</td>
<td>Producer</td>
<td>Contract Test Cases</td>
<td>N/A</td>
<td>No</td>
<td>High</td>
</tr>
<tr>
<td>Testable Bean       (Gao et al., 2002)</td>
<td>Producer</td>
<td>Monitoring Environment/Test Interfaces</td>
<td>Consumer</td>
<td>No</td>
<td>High</td>
</tr>
<tr>
<td>STECC               (Beydeda et al., 2003b)</td>
<td>Producer/Consumer</td>
<td>Test Cases generation</td>
<td>Consumer</td>
<td>No</td>
<td>Low</td>
</tr>
</tbody>
</table>

- In summary, all the approaches increase the programming overhead. However, component producers cannot have its effort to develop a reusable component too increased because activities to produce information to support tests were added. This will inhibit the development of reusable software;

- The increased architectural complexity to support consumer testing has maintainability issues. If the producer adds modifications to the component, the self-testing infrastructure would have to be checked and adapted to reflect the changes;

- Some BIT approaches (Wang et al., 1999) and (Atkinson et al., 2002) are founded in static data, with a pre-defined set of test cases. This means component consumers use adequacy criterion anticipated by the component producers. As a consequence, consumers may have difficulties to check if the candidate component really fulfills its specific needs;
- Provision of static and automatic test cases (Wang et al., 1999) contribute little to increase consumers understanding of the component they intent to integrate. According to Kaner et al. (Kaner et al., 2001), “the advantage of automatic testing comes from using a computer to do things a person cannot do”, which means avoiding repetitive test operation over and over again or comparing thousands of outputs. BIT approaches apply automated test cases to maintain knowledge gap of the consumer about the component when they should work to reduce it; and

- BIT approaches, although (Beydeda et al., 2003a) is an exception, lack focus on consumers. This means they are more an additional feature of reusable components than a tool to effectively interact with component consumers and support test activities like component reasoning, test case elaboration and analysis of results.

### 3.3. Component Testing Tools

Component testing approaches go ranging from guidelines of architectural decisions to improve component testability to workflows of activities to collect and pack information to support third-party testing. A solution frequently adopted by organizations to increase the productivity of its employees is to use tools to accomplish critical or exhaustive tasks. Thus, in this section, tools to support component test activities at both sides (producer and consumer sides) are surveyed.

In 2000, Bundell et al. (Bundell et al., 2000) presented a component verification tool to help component consumers to integrate components into their systems called Component Test Bench (CTB). Producers write generic test specifications in XML and pack them together with a pattern verifier application inside the component. Consumers will be enabled to manipulate those generic tests to fit their own needs and run/check test cases in pattern verifier. In Figure 3.1, the basic strategy of the approach is depicted, which is a general XML based test case that can be adapted by consumers to validate their environment.
Shortcomings of CTB are: (i) test pattern verifier is a stand-alone application independent of development environment. This may reduce the possibility for consumers to elaborate tests linking the candidate component and other resources like libraries and other components; in addition, (ii) component consumer is supposed to understand component behavior completely from other source of information because the XML specification is strictly related to data structured packing, not its semantic meaning.

In 2003, Bertolino et al. (Bertolino et al., 2003) proposed a framework for component deployment testing (CDT). The key idea of the framework is the complete decoupling between what concerns deriving and documenting the test specifications and what concerns the execution of the tests over the implementation of the component. Consumers can early specify test cases to cover their needs independently to any possible component implementation. Then, after some candidates are selected, for instance, in a repository, CDT uses reflection to retrieve information (mainly method signature) from the candidate component and link these method calls with the test cases previously developed. Test execution also uses JUnit\(^7\) to run the test cases.

The main benefit of this approach is also a serious trade-off because to overcome the differences among the methods of different components, a file to

---

\(^7\) [http://www.junit.org](http://www.junit.org) (last visit: June 1, 2008).
map each test to each component methods called “XMLAdapter” must be constructed manually. This effort can be costly to component consumers.

Since 2005, IBM Rational Application Developer⁸ (IRAD) provides support for software producers to test their component-based systems under development. IRAD is fully integrated with WebSphere® development environment and uses JUnit to execute test cases. Its main focus is on providing analytical support to testing activities of systems composed of components or to components that will be delivered as products. Among its features are: (i) architectural metrics to measure the complexity of relationships such as method calls, inheritance, or variable usage; (ii) component complexity metrics like nesting level to measure complexity of the control flow of the source code; and (iii) test coverage metrics to indicate the percentage of lines of code covered and the number of tests directly using a method of a class.

IRAD main limitation is its applicability only to component producers. This condition is mainly because the tool was designed to test components under construction not to help consumers to integrate external components into their systems. Solutions to overcome lack of information like requirements, component usage or source code are not covered by the tool.

In 2007, Teixeira et al. (Teixeira et al., 2007) presented a tool called FATEsc – in english (Tool to Support Component Structural Testing)⁹. This tool uses structural coverage measures produced by component producer as an aid to the integration of the component in a user application. Test cases elaborated by the producer are processed by another tool called JaBUTi (Vicenzi et al., 2005) that provides coverage analysis information. Next, this information and test case descriptions are packed within the component to component consumers. Consumers can use the information to check if their tests cover all the internal flows of the candidate component comparing their coverage analysis results to those provided within the component.

Followed by some conclusions, a summary of the main characteristics related to tools that support component test activities at producer and/or consumer side is presented in Table 3.3.

---

⁸ http://www.ibm.com/developerworks/rational/library/05/kelly-stoker/ (last visit: June 1, 2008).
⁹ In Portuguese: “Ferramenta de Apoio ao Teste Estrutural de Componentes”
Component testing is only part of a workflow of activities that aim at constructing a system with the benefits of software reuse (Lim, 1994). Tools like CTB and FATESc are stand-alone application (CDT can be considered more a library rather than an application). However, today many development environments like Eclipse\textsuperscript{10} and .NET\textsuperscript{11} can concentrate system modeling, programming interface and testing. A better integration with development environments would improve productivity on both producer and consumer sides;

- FATESc tool gives an extra chance to improve the description of the component by providing means to component producer add informal description about test cases as metadata. This is an important feature because any additional information to increase the consumer understanding level of candidate components is valid. The less dependant of other source of information, the better tests are; and

- Among the tools surveyed only IRAD is commercial, yet focused solely on producers. Maybe because the area is not

---

\textsuperscript{10}http://www.eclipse.org/ (last visit: June 1, 2008).
\textsuperscript{11}http://www.microsoft.com/net/ (last visit: June 1, 2008).
sufficiently mature or because a tool to test components alone is not enough to overcome all the challenges of component based-development (Crnkovic, 2003). Probably, a tool integrated with development environment to act at both sides, facilitating component producers to prepare their components to be tested elsewhere, and supporting consumers on the elaboration of test cases and to understand how the component behaves can be more effective.

3.4. Summary of the Study

Based on the approaches and tools surveyed, the timeline of the research on the software component testing is presented in Figure 3.2. There is a clear concentration of work in the end of nineties and beginning of years two thousand.

Figure 3.2 – Software component testing timeline: approaches and tools

In addition, based on the analysis and characterization of the approaches and tools, a set of seven necessities (Nec), that can be considered the foundations to achieve effective component testing, were defined.

**Nec 1. Test whether source code is absent or not**
Component producers may have legal barriers to provide the source code of their commercial components. But this necessity is more related to the current state of CBSD. Components are heterogeneous by many aspects. Some of them are generated from software product lines inside an organization, or they can be produced according to different development processes such as (D’Souza et al., 1998), (Atkinson et al., 2000) and (Chessman et al., 2001). Also, there is a huge number of open-source components, some of them constructed in an ad-hoc manner. As a result, consumers cannot rely on the source code to test a candidate component. Moreover, the effort to understand source code of a candidate component that in the end may the discarded can be too costly to be considered a reasonable approach.

**Nec 2. Integrate component testing and development environment**

Testing cannot be a disjoint activity of the overall component-based development process. It should be integrated to it. Nowadays, there are extensible development environments such as Eclipse® or NetBeans®\(^\text{12}\) that provide complete tooling infrastructure for building IDEs as internal modules of themselves. This can contribute to incorporate component testing as part of the activities workflow.

Achieving productivity gains and the benefits of software reuse is possible by supporting producers to elaborate effective and sufficient documentation to ease further testing; and by providing means for component consumers understand and test a candidate component before its integration.

**Nec 3. Data collection**

Tools to create/execute test cases such as JUnit or testNG\(^\text{13}\) are well-known, but tools to help producers to collect test data to be provided to consumers are less common. For instance, an algorithm to capture component usage information and generate a regular expression to be provided to consumers was proposed by (Liu et al., 1999), in a similar way, FATESc tool (Teixeira et al., 2007) uses another tool (JABUTi) to provide coverage analysis.

---

\(^\text{12}\) [http://www.netbeans.org/](http://www.netbeans.org/) (last visit: June 1, 2008).

\(^\text{13}\) [http://testng.org/doc/](http://testng.org/doc/) (last visit: June 1, 2008).
of test cases and provide it within the component. This reduces the effort of component producers to elaborate documentation dedicated to testing.

**Nec 4. Decouple component testing from architecture design**

There are some approaches that rely on easily testable architectures to overcome component testing difficulties such as (Wang et al., 1999), (Atkinson et al., 2002) and (Gao et al., 2002). They are founded on software testability that is considered by Voas et al. (Voas et al., 1995) one of the 3 pieces of the reliability puzzle. But exclusively architectural solutions have some shortcomings like its increased complexity (and programming overhead associated), memory usage issues, maintainability issues and, unless older components from repositories were re-engineered to adapt their architectures to provide testability, they have applicability limitations. Also, among agile methods adopters, that according to Highsmith et al. (Highsmith et al., 2001) consider the simplest solution usually the best solution; complex architecture may find some resistance.

By providing detailed information to support consumer testing activities component producers can overcome the limitations of architectural solutions.

**Nec 5. Component misunderstanding problem minimized**

According to a recent research on open-source component (OSC) integration by Merilinna et al. (Merilinna et al., 2006), “...there are methods and techniques to evaluate and select components, represent and predict architectural mismatches, express interface semantics and handle platform-related aspects, and some means for controlling issues caused by the evolution of the OSCs. However, these methods and techniques are seldom applied in practice, probably due to small non-distributed development teams, relatively small scale of the software systems developed and agile methods applied in software development. **Primary problems in practice were issues concerning vertical integration and the lack or impreciseness of documentation.**” This indicates that even if source code cannot be provided as well as requirements specification to support testing, the focus of component producers should be on providing useful information to consumers in a way
consumers can understand how their component behaves, how to use it and how to adapt it to their needs.

Nec 6. Information provided in a standard format

There are some approaches, such as (Orso et al., 2000), that do not specify any format to the data provided to consumers. Data provided in natural language suffer from problems associated to requirements specification. Basically, they can be ambiguous, incomplete and out dated. Also, according to Krueger (Krueger, 1992), \textit{“these descriptions can often be as difficult to understand as source code”}. Nowadays, XML\textsuperscript{14} can be considered the main standard format to share of structured data across different information systems. Packing data in a standard format makes it easily processed by tools. Also, given the hierarchical self-descriptive structure of XML it can be readable by humans too as an additional source of information. Probably, a combination of natural-language and formatted information related to testing may improve consumers’ understanding level.

Nec 7. Support consumer acceptance criteria

Some BIT approaches such as (Wang et al., 1999) and (Atkinson et al., 2002), are founded in static data, with a pre-defined set of test cases. Consumers may change the input of the pre-defined test cases and check output data but this is not enough to validate a component. Consumers have their own needs and expectations related to a candidate component. This means they must be able to create their own tests to validate components under their own criteria and context of use, not the one provided by the producer.

3.5. Summary

This Chapter presented a survey related to the state-of-the art in the software component testing research describing eight approaches to test components and four tools.

Through this survey, it can be noticed that software component testing has many issues and further research is needed to support component producers to prepare a component to be tested by third party, and also to help

\textsuperscript{14} http://www.w3.org/XML/ (last visit: June 1, 2008).
component consumers to understand how a candidate component works in order to test it and decide about its integration to the system under development. In this context, next chapter presents the component testing approach proposed by this work.
Based on the study of component quality and testing discussed in Chapter 2, and the survey presented in Chapter 3 that analyzed current approaches to test components identifying issues and necessities to achieve effective component testing, this chapter presents an approach to support component testing in a form of workflows to be performed by component stakeholders. The workflows encompass activities at component producer side, to support producers to prepare a component with test-related information to someone that was not involved with the development of the component, but is considering to adopt it; and activities at component consumer side, to support consumers to understand and test candidate components checking if they correctly fulfill their needs.

In addition, to support both component producers and consumers to accomplish the activities of the proposed workflows, it was specified, designed, implemented and evaluated a CASE tool integrated to the development environment to be used by producers and consumers to perform the activities of the workflows.

In this context, this chapter presents an overview of the proposed approach to support component testing, as well as the tool covering its activities.
4.1. Introduction

Literature contains several work related to CBD methods and approaches, however, the main consolidated CBD methods (D'Souza et al., 1998), (Atkinson et al., 2000) and (Chessman et al., 2001) are more aware in demonstrating component development as a feasible approach, while others important activities, for instance related to quality and, more specifically, testing, are sometimes neglected.

According to Councill (Councill, 1999), it is mandatory to provide components with a minimum level of quality to promote software reuse and take the benefits provided by them. However, components are heterogeneous by nature. There are open-source components, COTS components, components presented in different programming languages, besides components presented with or without source code. In addition, the level and quality of information varies from component to component.

Testing components does not mean simply to execute tests and correct defects. While components can be used in different contexts, sometimes the context that the component producer has used to validate its component is different from the component consumer’s one (Gao et al., 2003). This means that component testing is less trivial than traditional testing in which the acceptance criteria of the customer are clearly translated into acceptance criteria of functional and non-functional requirements.

Thus, the objective of our approach is to propose guidelines to be followed by component stakeholders to conduct testing activities focused on component quality in the context of CBD projects. In addition, tool support is provided to facilitate component producers and component consumers to accomplish the proposed approach.

4.2. Overview of the Proposed Approach

The foundations to achieve effective component testing presented in Chapter 3 act as a motivation to the construction of our approach. It can be noted that current approaches to test components such as: (Wang et al., 1999), (Harrold et al., 1999), (Liu et al., 1999), (Orso et al., 2000), (Bundell et al., 2000), (Edwards, 2001), (Gao et al., 2002), (Beydeda et al., 2003b) and
(Teixeira et al., 2007) share common limitations, which are to assume that detailed documentation about the component is frequently available to consumers and to address a specific activity of component testing such as test-case creation or integration testing.

For instance, (Harrold et al., 1999) presented an approach focused on the activity of producers collecting summary information to support consumers to analyze and test external components. Another approach, (Bundell et al., 2000), packs test information in a standard XML format and provides tool support to help component consumers to integrate components into their systems. In addition, solutions such as (Bertolino et al., 2003), are mainly focused on consumers to reuse the same generic test cases in different candidate components.

Thus, it can be noticed that current component testing approaches do not integrate well component producers to component consumers since they address only punctual aspects. However, the integration of both stakeholders is important because it is the same bridge that connects, in traditional software development, the system architects to software programmers, requirements staff to testers, and so on. Unfortunately, this bridge is missing in CBD because component producers and consumers can be complete independent teams with or without any communication channel.

An analysis was conducted, in order to define guidelines to be performed by component producers to prepare a component to be tested by third-party, as well as guidelines to be performed by component consumers to understand and validate the behavior of external components against their acceptance criteria. Our analysis associates current component testing approaches to groups of test-related strategies. Then, for each group of strategies, a general activity was derived representing the main purpose of the group.

Table 1.1 from Chapter 1 presents how strategies related to testing adopted by the studied approaches were mapped into general activities that can be performed by component stakeholders. Stakeholder column defines which stakeholder is responsible for performing one or more Activities (stakeholders can be component producers or consumers). Strategy column links component test approaches and a (macro) activity that represents the main focus or strategy
adopted by the approaches. *Related Approach* column associates component test approaches and the group of one or more strategies performed by the approaches.

The activities presented in Table 1.1 are the starting points to define guidelines to effectively support component producers and consumers when testing components. Guided by the literature related to quality and testing, current approaches to test components, international standards and experience testing components we propose two workflows of activities. One workflow is to be conducted at component producer side encompassing activities to prepare a component to be tested by someone that was not involved with the development of the component (third-party testing). Another workflow is to be conducted at component consumer side. It presents activities to consumers to understand candidate components, to check if candidate components work as designed and to analyze if they work integrated to the system under development fulfilling acceptance criteria.

In the next sections the objectives of workflows and their elements are presented in detail.

**4.3. Objectives**

According to Table 1.1, current component testing approaches provide solutions to support component testing addressing specific issues. In addition, the problem of the lack of information between component producers and consumers, as noted by (Beydeda et al., 2003b) and (Bertolino, 2007), indicates that current approaches do not integrate well component producers and component consumers. In general, the workflows were defined under the need to answer the following questions:

**Q1.** How can component producers prepare a component to be tested by someone that was not involved with the development of the component?

**Q2.** How can component consumers understand candidate components, check if candidate components work as designed and analyze if they work integrated to their systems fulfilling their acceptance criteria?
Based on the expected answers for these questions, the main objective of our approach can be stated as follows:

“In order to increase quality of component based systems, our approach is defined presenting guidelines to aid stakeholders to conduct component testing activities”

4.4. Approach Representation

Kruchten (Kruchten, 2000) defines a workflow as “a sequence of activities that produces a result of observable value”. In order to be better understood and used, the proposed workflows need to describe a language that aids users to use it effectively. A language is also useful to repeat the workflows for many different component development projects and offer a reference for it. In this dissertation the Structured Analysis (SA) language is used (Ross, 1977).

SA’s thesis is that the human mind can accommodate any amount of complexity as long as it is presented in easy-to grasp chunks that are structured together to make the whole. This can contribute to stakeholders understand and repeat workflows many times.

The main benefits of using SA according to (Ross, 1977) are:

- It incorporates any other language; its scope is universal and unrestricted;
- It is concerned only with the orderly and well-structured decomposition of the subject matter;
- The decomposed units are sized to suit the modes of thinking and understanding of the intended audience;
- Those units of understanding are expressed in a way that rigorously and precisely represents their interrelation;
- This structured decomposition may be carried out to any required degree of depth, breadth, and scope while still maintaining all of the above properties; and
SA increases both the quantity and quality of understanding that can be effectively and precisely communicated well beyond the limitations inherently imposed by the imbedded natural or formal language used to address the chosen subject matter.

Next sections will describe in detail the workflows at component producer side and at component consumer side.

4.5. Component Producer Workflow

The workflow of activities to be performed at component producer side encompasses 4 activities to prepare a component to be tested by someone that was not involved with the development of the component. Its main goal is to provide enough information about the component to enable component consumers to understand how a candidate component works, its limitations and provided functionalities, usage assumptions and how to integrate it to another system. In summary, how to avoid the problems of the lack of information and to comply with the second reuse truism, which is one of the key aspects of software reuse (Krueger, 1992): “For a software reuse technique to be effective, it must be easier to reuse the artifacts than it is to develop the software from scratch”.

Figure 4.1 presents the SA diagram of the workflow at component producer side with its input elements, outputs, controls and actors involved. The sequence of the activities is not necessarily a waterfall because some activities, such as Collect Information and Restrict Information, can be done interleaved, while others can be skipped. Also, as people may fill many different positions in a software development organization, the actors executing the activities were named Component Producer to demonstrate that the responsible for a given activity is someone occupying that position at that moment. The responsible can be a tester, a developer, a requirements analyst among others. Next, each activity will be described in detail.
Figure 4.1 – Workflow of Activities at Component Producer Side

4.5.1. Collect Information

The first activity corresponds to an information search phase. Although it can noted that in Table 1.1 there are no activities related to information collection, this activity is important to be performed because before the producers prepare a component to be further tested, they should be aware of what kind of documentation was produced while the component was under development.

This kind of activity can be transparent in traditional in-house software development where the team to test software and the programmers, in general, can interact with each other more frequently. Furthermore, documentation from other development phases should be in a repository to be accessed by everyone involved in the project.

The main goal of this activity is to make component producers collect all documentation available that contain information to improve the understanding level of someone that was not involved with the development of the component.

The types of documentation available may vary depending on the development process that produced the component. On one hand, processes close to Agile methods such as Extreme Programming\textsuperscript{15} should produce less descriptive documentation because its main output is the source code, but on the other hand, processes such as UML Components (Chessman et al., 2001)

\textsuperscript{15} Extreme programming homepage: http://www.extremeprogramming.org/ (last visit: June 1, 2008)
produce detailed descriptions including class diagrams and interface specification in OCL (Warmer et al., 1998).

### 4.5.2. Restrict Information

The second activity of the workflow at component producer side – Restrict Information - is also not present in Table 1.1. For open source components this activity can be considered to be skipped if there is no confidential information. However, restriction of information is necessary due to component market needs.

Even so, apart from open source components, there is a growing component market which deals with the same copyright and intellectual property regulations of ordinary commodities. ComponentSource\(^{16}\) is just one example of mature component markets that retails a variety of components. However, not all the components sold by ComponentSource provide requirements sheet, use cases diagram, unit test cases or source code. Normally what is provided is a limited evaluation version, and after purchase, user manuals and sometimes customer assistance.

Thus, this activity guides producers possessing diverse types of information gathered in the previous activity to check their copyright or internal regulations. The objective is to mark information that must not be disclosed to consumers for any legal/strategic reason.

Although this activity can classify proprietary algorithms, business rules or certain process assets as restricted to consumers, it does not mean that these kind of information should be ignored by other activities of the workflow, on the contrary, source code or use cases can be used to support producers in the construction of other documents that can be provided to facilitate component testing at component consumer side.

\(^{16}\) Component Source has available over 10,000 SKU’s from more than 230 publishers. Website: http://www.componentsource.com/ (last visit: June 1, 2008)
4.5.3. Analyze Information

The third activity – *Analyze Information* – is the one that demands most of the effort at component producer side. It consists of producers using information gathered from previous steps to generate information dedicated to make testing at component consumer side a viable activity.

As surveyed in Chapter 3, a wide number of techniques can be applied to perform this activity. It can be considered the use of automated approaches to extract relevant information (e.g., from source code), or the use of tools to aid producers to generate test-related documentation. For instance, Component Retrospectors (Liu et al., 1999) is an approach to capture usage information from source code snippets. Another tool, presented by Teixeira et al., called FATESc (Teixeira et al., 2007) is focused specifically on collecting information to structural analysis of the source code. On one hand, the benefit of using tools to generate information for further testing is that it can reduce the effort of component producers when performing this activity. On the other hand, a trade-off of is that they lack diversity of information generated since they are commonly focused on a specific strategy to provide specific information.

Alternatively, producers can consolidate all information available in a textual document dedicated to support testers at component consumer side. A major concern is related to what kind of information such a document should provide. Related to what kind of information should be specified in components, according to (Crnkovic et al., 2002): “*Specifications should also be heterogeneous, since the diversity of properties that might be of interest is unlikely to be suitably captured by a single notation*”. Moreover, by analyzing International Standard ISO/IEC 12119 (ISO/IEC 12119, 1994), later updated by ISO/IEC 25051 (ISO/IEC 25051, 2006) both related to requirements for quality of Commercial Off-The-Shelf software products and instructions for testing; and IEEE Standard 829 for Software Test Documentation, we suggest that a document to support third-party to test external components should be composed (at least) of:

- **Functional Descriptions.** According to international standards related to quality requirements, a general description of each
functionality should be provided. The description should be consistent and without ambiguities that normally can exist in natural-language descriptions;

- **Usage Descriptions.** It should be provided instructions about how to correctly execute a given functionality. How to invoke it, its usage assumptions and restrictions; and

- **Data Descriptions.** Related to the functionalities described, it should be provided input values that can modify the state of a given functionality, its expected results, boundary values, wrong input values and any resource dependence that may exist.

Although ISO 12119 advocates the provision of test cases from producers to consumers, our approach suggests usage descriptions instead. This is because the international standard is focused on product evaluation whereas consumers in CBSD are focused not only on evaluating the provided functionalities but also on evaluating the integration of the candidate component and the system under development. Thus, providing static test cases according to producer’s context can be less effective than information about dependencies among interface method calls and usage assumptions. This demonstrates that the abstraction level provided to consumers should be at functionality level in spite of test case level.

In this context, Table 4.1 presents some assets that are products of component-based development process from where information to support further testing can be produced.
### Table 4.1 – Information to Support Further Testing Derived from Different Types of Asset

<table>
<thead>
<tr>
<th>Asset</th>
<th>Information Produced</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements Sheet</td>
<td>- Textual Description</td>
<td>ISO/IEC 12119</td>
</tr>
<tr>
<td></td>
<td>- System Functionalities</td>
<td>(ISO/IEC 12119, 1994)</td>
</tr>
<tr>
<td></td>
<td>- User manual</td>
<td></td>
</tr>
<tr>
<td>Source code</td>
<td>- Comments</td>
<td>JAVADOC®</td>
</tr>
<tr>
<td></td>
<td>- Usage Assumptions</td>
<td></td>
</tr>
<tr>
<td>Component Specification</td>
<td>- Invariants</td>
<td>OCL</td>
</tr>
<tr>
<td></td>
<td>- Usage assumptions</td>
<td>(Warmer et al., 1998)</td>
</tr>
<tr>
<td></td>
<td>- Interface Description</td>
<td>UML Components</td>
</tr>
<tr>
<td></td>
<td>- Use Cases Diagrams</td>
<td>(Chessman et al., 2001)</td>
</tr>
<tr>
<td>Test Cases</td>
<td>- Input Data</td>
<td>BS 7925-2</td>
</tr>
<tr>
<td></td>
<td>- Boundary Values</td>
<td>(BS 7925-2, 2001)</td>
</tr>
<tr>
<td></td>
<td>- Exceptional Conditions</td>
<td>IEEE Std 829</td>
</tr>
<tr>
<td></td>
<td>- Expected Results</td>
<td>(IEEE Std 829, 1998)</td>
</tr>
<tr>
<td>Testable Architecture</td>
<td>- Execution Trace</td>
<td>BIT</td>
</tr>
<tr>
<td></td>
<td>- Dedicated Test Interfaces</td>
<td>(Wang et al., 1999)</td>
</tr>
<tr>
<td></td>
<td>- Built-In test cases</td>
<td>Testable Beans</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(Gao et al., 2002)</td>
</tr>
</tbody>
</table>

### 4.5.4. Publish Information

The fourth step at component producer side is to Publish Information. This activity consists on packing information generated in previous steps in a standard format, if it is the case, and publishing it to consumers.

The decision to pack the information dedicated to testing elaborated in a standard format file is sometimes related to the use of tools at component consumer side. Bundell et al. (Bundell et al., 2000) generate information in XML files which permits test specifications to be read and interpreted on a wide variety of platforms. Similarly, (Teixeira et al., 2007) generate structured analysis information in XML files to make them easily portable to the other part of the tool at component consumer side.

The act of publishing information to consumers means that the information related to the component to aid consumers in order to conduct testing activities is made available. For instance, ComponentSource publishes information to support consumers to use components in its website. Other approaches related to architectural solutions such as (Gao et al., 2002) and
(Beydeda et al., 2003b) also attach information to component object code, i.e., inside executable files or JAR files.

Next Section describes the workflow of activities at component consumer side in detail.

4.6. Component Consumer Workflow

Assuming that the workflow at component producer side was conducted, the workflow of activities to be performed at component consumer side encompasses 4 activities to support consumers to test quality characteristics of external components. By testing candidate components, consumers can validate if the component they intend to plug their system works as designed and if the candidate component fulfills their acceptance criteria. Thus, the test results analysis can support business decisions, such as recommending or rejecting integration of candidate components to the system under development.

Figure 4.2 shows the SA diagram of the workflow at component consumer side with its input elements, outputs, controls and actors involved. It is important to highlight that the sequence of the activities is not necessarily a waterfall model. For instance, Understand Component and Elaborate Test Cases can receive as input test results from the last activity of the workflow. Also, as people may fill many different positions in a software development organization, the actors executing the activities were named Component Consumer to demonstrate that the responsible for a given activity is someone occupying that position at that moment. Next, each activity of Figure 4.2 is presented.
4.6.1. Identify Reuse Points

The first activity at component consumer side corresponds to identify reuse points at the system under development. A reuse point is considered the point where the system under development ends and where candidate components can be plugged in. Although this activity is not present at Table 1.1, it is important to clearly delimit the boundaries of the system.

The place in a system under development where consumers should plug in external components can already be clearly specified in a form of formal specified interfaces. It can also be where the mock objects were plugged to simulate the behavior of external components while the system where being tested. However, given the current diversity of CBD methods, these reuse points may be missing or not well defined yet. They can vary from functional requirements described in natural language to use cases that were elaborated but not realized. These issues can impact on the quality of future test cases created, specifically those focused on testing the integration of the component and the target system. Figure 4.3 depicts a system under development associated to candidate components, where reuse points, adapters and component interfaces are identified.
One example of this activity can be, for a hypothetical on-line store system, after customers complete registration in company’s website, an e-mail is sent to them with password and other information to confirm the provided data. The reuse point identified will be the point to invoke candidate e-mail components to send e-mails after user registration.

4.6.2. Map Reuse Points to Functionalities Provided

The second activity – Map Reuse Points to Functionalities Provided – can be considered a preparation to the elaboration of test cases. After system entries where external components can be plugged in are identified; an association among the candidate component’s functionalities and the reuse point that they probably address should be conducted.

This activity is important because generally not all the functionalities of a candidate component are used by consumers, but only a subset from the total of functionalities provided by the component. The mapping performed by this activity identifies three types of information:

- What is needed to be tested;
- What is not needed to be tested; and
- Where interactions occur
Figure 4.4 shows the types of information discovered when analyzing the reuse points of the system against the functionalities provided by a candidate component. Functionalities in area $A - (A \cap B)$, presented in light grey, are not directly accessed by the system under development and may not need to be tested. However, functionalities presented in area $B - (A \cap B)$, in dark grey, do not interact with the candidate component and can be tested by consumers according to traditional software testing processes. Finally, functionalities in area $(A \cap B)$, the dashed area, represent interactions of the system under development and the candidate component. It demands integration tests to check if the system and the candidate component can work properly together.

![Diagram](https://via.placeholder.com/150)

**Figure 4.4 – Interactions between Functionalities of a Candidate Component and Target System Functionalities**

The organization developing the system may have the internal policy to test all the functionalities of external components (even those functionalities not reused), however, the *second reuse truism* (Krueger, 1992) stimulates the identification of what is not required to be tested to reduce the effort of component consumers. For instance, in an e-mail component, if consumers just want a component to send simple text messages, only `sendSimpleMessage` functionality should be tested whereas a functionality to send e-mails with attachments, for instance, `sendMessageWithAttachment` should be ignored.

We suggest the consumers to consider the construction of an interaction matrix (similar to requirements traceability matrix) to mark where functionalities of the candidate component and the system under development interact.
According to (Bertolino et al., 2003), another advantage of identifying where interactions occur is that it may indicate the need of using adapters or some glue-code when elaborating future integration test cases.

### 4.6.3. Understand Component and Elaborate Test Cases

The third activity of the workflow at component consumer side – *Understand Component and Elaborate Test Cases* – uses all the information provided by component producers within the component.

In order to elaborate effective test cases, consumers should understand how candidate components work. They must be aware of usage assumptions of candidate components, correct order of method calls and inputs that do not violate interface specifications. A common problem related to misunderstanding, according to (Liu et al., 1999), is component misuse problem. It happens when a component is used in a way different from what the component producer expects. For instance, if a hypothetic component $\chi$ has methods $a()$ and $b()$ in its main interface, and the invocation of method $b()$ reads a variable that must be set by method $a()$ beforehand, this is a case of dependency between $a()$ and $b()$. If the consumer is unaware of that dependency, the lack of clarity may lead to component failure. Eventually, component consumer will reject component $\chi$. The information provided from component producer’s workflow is fundamental to avoid component misuse and the lack of information between component producers and consumers.

Adaptations to traditional test process can be applied to support tests in component based systems. Figure 4.5 depicts a representation of a test process from British Standard 7925-2 – Standard for Software Component Testing (BS 7925-2, 2001). To comply with specific demands of component based systems, adaptations to phases of the test process (BS 7925-2, 2001) related to this activity are described next.

---

**Figure 4.5 – Example of Test Process from BS – 7925-2**
• **Component Test Planning.** To support planning, consumers can analyze their acceptance criteria and functionalities descriptions provided within the component. In order to decide a test strategy, we suggest using the strategy selection criteria from (Gao et al., 2003). If source code is provided and white-box testing is considered an option, consumers can use techniques such as: flow graph-notation (Larsen et al., 1996), path-testing (Beydeda et al., 2001) or data-flow testing (Harrold et al., 1994). However, if source code is absent, consumers can use black-box techniques such as random testing, partition testing or boundary value testing;

• **Component Test Specification.** Based on the interactions between the system and the candidate component identified in the second activity of this workflow, consumers should construct test cases to validate each functionality they intend to reuse according to their acceptance criteria. A variety of tools can be used to automate component test specification/execution. Testing frameworks such as JUnit or testNG can also be applied integrated to the development environment.

### 4.6.4. Execute Test Cases and Evaluate Results

The last activity of the workflow at component consumer side – *Execute Test Cases and Evaluate Results* – executes the elaborated test cases and analyzes their results. To comply with specific demands of component based systems, adaptations to phases of the test process (presented in Figure 4.5) related to this activity are described next:

• **Component Test Execution and Recording.** In general, there are no significant changes compared to traditional test process. The test execution can be performed manually or with tool support. Test records should be preserved for further analysis; and

• **Checking for Component Test Completion.** Based on the analysis of the test record, consumers can decide to make
modifications on existing test cases to correct wrong test case design or elaborate additional test cases. The results can indicate that the candidate component fulfills the quality standards of consumers and can be considered to be integrated to the system, or on the contrary, can be recommended to be discarded.

In order to cover the activities of the proposed workflows supporting component producers and consumers to conduct test-related activities, a CASE tool was developed. Next section details this tool.

4.7. Case Tool to Support Component Testing

Based on the survey on Chapter 3, that identified the main approaches with techniques for testing components with its strong and weak points, as well as the proposed workflows of activities to be performed by component stakeholders, we specified, designed, implemented and performed an evaluation of a tool to support component testing. The tool is focused on: (i) reducing the lack of information between component producers and consumers, supporting component producers to prepare a component with test-related information; and (ii) supporting component consumers to understand and test candidate components checking if they correctly fulfill their needs.

In this section, we present the requirements, architectural and implementation details and usage of the tool.

4.7.1. PUG & CTP By Example

This section presents the plug-ins from a user’s point of view at both component stakeholder sides. First, at component producer side, producers can elaborate a class containing methods exercising functionalities of the component. They can reuse, for instance, high-level JUnit tests with some adjustments. Separation of concerns is important when selecting high-level unit tests or creating code snippets. An interesting metric to component producers can be one unit test/code snippet per functionality to prevent overlapping of usage information and reduction of the quality of information gathered.

Figure 4.6 presents a class that contains methods with usage information (Figure 4.6, mark 1). A user to invoke PUG tool can select the option Manifest
*Generator* to activate input data functionality (Figure 4.6, mark 2), as presented in Figure 4.13. After test data is inserted, the user will be asked to select a destination folder to the Usage Manifest file (Figure 4.6, mark 3). The user can open the generated `usage-manifest.xml` file in a text editor before attaching it to the component.

![Figure 4.6 – PUG Integrated to Eclipse Environment](image)

At component consumer side, the first step is to open CTP and associate a JAVA project (which is the system under development) to a candidate component as presented in Figure 4.7 (marks 1 and 2). When a component contains a *Usage Manifest* file attached, CTP will automatically load the tree view Figure 4.7 (mark3). Then, the user will be presented to the functionalities of the candidate component available to be reused.

Once the functionalities of a candidate component are presented in a tree view, users can interact with CTP accessing the features described in Section 4.7.6 when constructing, for instance, JUnit test cases to analyze interactions of the system under development and the candidate component loaded. Figure 4.7 presents the steps to load a Usage Manifest file in CTP with functionalities descriptions of a candidate component.
4.7.2. Requirements

In Chapter 3, we presented the foundations to achieve effective component testing and discussed the gaps of current component testing tools. From those issues and from the proposed workflows, a set of functional and non-functional requirements was defined. In addition, the experience of the RiSE Group and the author experience was considered in the definition of the requirements.

The identified Functional Requirements (FR) are:

**At component producer side.** (FR1) Pack information in a standard format, (FR2) Collect input data from functionalities provided by a candidate component, (FR3) Capture usage information from candidate component; and

**At component consumer side.** (FR4) Presentation of usage assumptions from candidate components, (FR5) Register reuse points and their interactions with candidate component, (FR6) Support test case construction.

In addition, four Non-Functional Requirements (NFR) were identified: (NFR1) Extensibility, (NFR2) Integration with development environment, (NFR3) Enable component consumer to test whether source code is available or not, (NFR4) Architecture independency. Next, we discuss these requirements more deeply.
FR1. **Pack information in a standard format:** According to (Krueger, 1992), data provided in natural language suffer from similar problems associated to requirements specification. To reduce the ambiguity level of information generated and to make it easily processed by tools this requirement is considered important. Similarly, other component testing tools such as (Bundell et al., 2000) and (Teixeira et al., 2007) also implement this requirement.

FR2. **Collect input data from functionalities provided by a candidate component:** In order to consumers understand a candidate component, input values that modify the state of a functionality they intend to reuse should be captured by test tools. Input data can support consumers when designing integration test cases. In addition, according to standard (IEEE Std 829, 1998), when documenting users should: “specify each input required to execute the test case. Some of the inputs will be specified by value (with tolerances where appropriate) while others, such as constant tables or transaction files, will be specified by name”. Thus, it should be possible to aggregate input data from functionalities in components;

FR3. **Capture usage information from candidate component:** The provision of static test cases executed by the producer suffers from context restriction. According to (Weyuker, 1998) and (Bertolino et al., 2003), “the tests established and run by the developer lose much of their power in the realm of the assembled system. In fact, the developer cannot account for all the possible deployment environments”. Thus, better than providing static test cases, a test tool should capture how a component works, its usage assumptions and restrictions to provide to consumers enough resources to understand a candidate component.

FR4. **Presentation of usage assumptions from candidate components:** Consumers should be presented to the functionalities available in a visual hierarchical manner to facilitate their understanding of the provided functionalities. **JAVADOC**® is one example of tool-generated presentation style where descriptions of classes that compound a component are presented hierarchically in static HTML. However, JAVADOC can be considered a general-purpose documentation because it was not specifically developed for testing; it
lacks, for instance, guidelines for inserting input values of methods and presentation of the order of method calls to accomplish a given functionality.

**FR5. Register reuse points and their interactions with candidate component:** According to the workflow at component consumer side, identifying where the system under development and the functionalities of the candidate component interact eases the design of test cases and reduces the test scope by excluding unused functionalities. A similar strategy is also considered by (Bertolino et al., 2003) to perform component deployment testing. Thus, more effective test cases can be created if interactions were identified;

**FR6. Support test case construction:** Since consumers cannot rely on other resources than those provided by producers, component test tools must support test case construction. The use of test pattern verifiers as in (Bundell et al., 2000) to analyze violations to correct usage of components can reduce the number of test cases wrongly designed.

**NFR1. Extensibility:** A component test tool must be flexible to be extensible to new types of data or techniques to test component based systems.

**NFR2. Integration with development environment (IDE):** As presented in Chapter 3, stand alone tools may inhibit the construction of integration tests. We consider important that a component test tool can be part of development environment not only to effectively support users to perform tests, but also to be part of CBSD processes.

**NFR3. Enable component consumer to test whether source code is available or not:** A component test tool can use the source code of the component to provide more information about it. However, it cannot exclusively rely on it. According to (Gao et al., 2003), “In practice, under budget and time limitations, white-box testing very often cannot be carried out as thoroughly as we wish”. Thus, a component test tool should support testing regardless of the presence of the source code.

**NFR4. Architecture independency:** (Wang et al., 1999), (Gao et al., 2002) and (Atkinson et al., 2002) suggest building a complete testable architecture. However, given the increased complexity, memory usage issues,
maintainability and applicability issues of architectural solutions, we consider that a component test tool should be architecture independent.

We do not believe that the identified requirements are the complete set of requirements for a component test tool. However, we believe that they are the basis for the development of an effective component test tool.

Based on these six functional and four non-functional requirements, the component test tool was defined. Next, we present the tool overview and its implementation details.

4.7.3. Tool Overview

Focused on reducing the lack of information between component stakeholders, to conform to the identified requirements and the proposed workflows, we propose a tool that apply a combination of techniques (automatic, computer-assisted and manual) aiming to reduce programming overheads on both stakeholder sides, to prepare components to be tested by third-party and help component consumers to create tests to validate if a candidate component fulfills its needs.

Our tool is composed of two modules implemented as Eclipse plug-ins. One module, called **PUG (Program Usage Generator)** is at component producer side aiding component producers to collect data, analyze and pack relevant information to support further testing of the component. On the other side, with component consumers, a module called **CTP (Component Test Perspective)** helps consumers to understand a candidate component, to construct test cases and identify interactions between the system under development and the component to validate if the candidate component fulfills its necessities.

Eclipse was chosen because it is an open-source generic framework integrated with development environment. It was conceived to support the construction of tools that can be used to develop applications and to manipulate all kinds of documents. At the core of Eclipse is a relatively small plug-in loader. All additional functionality in Eclipse is performed by plug-ins loaded when the framework is started or when called upon.
To be capable of receiving pluggable tools, Eclipse platform is structured around the concept of extension points, which are the access points where other tools can be plugged in and contribute functionality and/or their own new extension points. For instance, the *Java Development Tools* (JDT) integrated development environment is itself a tool consisting of plug-ins and designed for the development of Java software. When Eclipse is started the platform dynamically discovers registered plug-ins and starts them when they are needed.

Figure 4.8 presents the extensible architecture of Eclipse where PUG and CTP are built as plug-ins. Next sections present a detailed description of the functionalities provided by the modules at both component stakeholder sides.

**Figure 4.8 – Eclipse Extensible Architecture**

### 4.7.4. Tools Architecture

PUG and CTP are integrated as plug-ins to Eclipse. However, both have their own internal architecture separated in modules. Figure 4.9 shows the architecture of each plug-in.

PUG plug-in has five modules:

---

17 JDT Homepage: [http://www.eclipse.org/jdt/](http://www.eclipse.org/jdt/) (last visit: June 1, 2008)
• **GUI module** is responsible to accept user interactions and present forms to be filled with input groups;

• Module *Plug-in Controller* acts as a bridge among other business modules and the *GUI*;

• The *XML* module is responsible for the generation of the *usage-manifest.xml* file;

• *Code Parser* module processes code snippets and collect only the relevant parts of the code that represents usage assumptions such as correct sequence of method calls to accomplish a functionality; and

• *Input Group* module is responsible to hold the input data that users can add to a *Usage Manifest*.

**CTP plug-in** has six modules:

• **GUI module** is responsible to accept user interactions to the different features of CTP. It also presents forms to be filled by users and the tree view with the functionalities of a *Usage Manifest* file;

• The module *Plug-in Controller* acts as a bridge among other business modules and the *GUI*;

• Module *Pattern Verifier* is responsible to analyze if methods inside classes have similarities to functionalities described in the *Usage Manifest* file;

• The *Reuse Point* module handles the identified associations of the system under development to functionalities of candidate components;

• *XML* module is required to load usage-manifest.xml files; and

• Finally, *Code Assist* module adds behaviors to Eclipse content assist support to manipulate the code completion suggestions when users are programming.
The current version of the tool implementation, considering PUG and CTP together, contains 72 classes with 398 methods, divided into 26 packages containing approximately 5,200 line of code.

### 4.7.5. Providing Information to Component Consumer Tester

PUG (Producer Usage Generator) is an Eclipse plug-in implemented in JAVA to be used by component producers to prepare a component to be tested by component consumers. It accepts as inputs source code snippets, textual documentation and input groups related to functionalities of the component. Its strategy is to capture as much as possible information producers are able to provide to consumers combining an algorithm to capture usage information and functionality descriptions provided by producers. The information collected is consolidated in a standard format XML file that can be attached to the component before its publishing. Figure 4.10 presents a use case diagram of the basic functionalities of PUG. Next, each functionality is described in detail.
The strategy adopted by PUG is to capture usage information from source code snippets. It parses methods from provided code snippets and uses algorithms similar to program slicing approach, as described by (Harrold et al., 1999), and component retrospectors as presented by (Liu et al., 1999). It extracts textual descriptions from JAVADOC, dependencies among variables and correct order of method calls to accomplish a functionality of a component. It always skips unrepresentative usage commands like `System.out.println`, or local variable definition, for instance.

To reduce programming overhead, producers can reuse, as code snippets, existent high-level component unit tests with little adjustments. High-level unit tests are those conceived while producers are developing the component to validate general functionalities from component’s main interface, not the ones to test `getXXX()` and `setXXX()` methods of a single class. A component that has a unit test method to validate a class in the bottom of component’s class hierarchy can probably be discarded.

The algorithm implemented within PUG to capture usage information. In general lines, the idea of the algorithm is to capture relevant statements from a selected part of a program and save the sequence of method calls that represents the usage of a functionality.

Figure 4.11 presents the general idea of the algorithm that can be defined as follows:
Let \( M \) be the set of all methods from a test suite \( TS \) of a component \( C \).

Let \( \text{UsagePolicy}(m, C) \) be the set that represents the Usage Manifest from a component, and \( m \in M \)

\[
\begin{align*}
\text{NEW usage-manifest} &= \text{NULL}; \\
\text{for each method } m \text{ of } M \text{ and } m \text{ of } \text{UsagePolicy}(m, C) \text{ do:} \\
\text{NEW usage-policy} &= \text{NULL}; \\
\text{if } (m \text{ has JAVADOC}) \text{ then} \\
\text{usage-policy.add(JAVADOC);} \\
\text{end if;} \\
\text{for each statement } s \text{ of } m \text{ do:} \\
\text{Visit } (s, \text{usage-policy);} \\
\text{end for;} \\
\text{usage-manifest.add( usage-policy);} \\
\text{end for;} \\
\text{Visit } (s, \text{usage-policy}) \{ \\
\text{• if } s \text{ is a MethodInvocation // like x.foo() } \\
\quad \text{o if MethodInvocation(s) of type Expression.Identifier(Expression)} \\
\quad \text{where qualified name is not from java.* or javax.* hierarchy like System.out.println("test") or a JUnit Assertion then} \\
\quad \text{usage-policy.add(s);} \\
\quad \text{o if composite MethodInvocation(MethodInvocation(s)) like System.out.println(x.foo()); The algorithm must be aware of it because x.foo() should be captured whereas System.out.println should not. So, System.out.println must be skipped and Visit(x.foo() ,usage-policy) should be called recursively.} \\
\text{• if } s \text{ is a VariableDeclarationStatement //like int i; or: Xxx aa = new Xxx(); } \\
\quad \text{o if VariableDeclarationStatement(s) of type Identifier=Expression like int i; or File file = new File() and Foo f = Foo();} \\
\quad \quad \text{• if Expression(s) of type StringLiteral like int i; then} \\
\quad \quad \text{Skip it;} \\
\quad \text{o if Expression(s) of type MethodInvocation like Foo f = x.test() and qualified name is not from java.* or javax.* hierarchy then} \\
\quad \text{Visit(Expression(s) ,usage-policy);} \\
\text{• if Expression(s) of type ClassInstanceCreation where qualified name is not from java.* or javax.* hierarchy like File file = new File() , if a method argument is a parameter, add it too then} \\
\text{usage-policy.add(s);} \\
\text{• if } s \text{ a ForStatement or IfStatement} \\
\quad \text{o } \{\text{Question relevance to user}\} \\
\} 
\]

**Figure 4.11 – Usage Capture Algorithm Pseudo-Code**

Figure 4.13 (mark 1) presents how the implemented algorithm aids component producers to capture information to be provided to consumers. The method at Figure 4.13 (mark 1) specifies the write usage of \textit{ODSReadWrite} component (with its main interface presented in figure 4.12). Although more than one method describing a functionality can be in the same class, this snippet
has only one method inside a JAVA class. From the class, the algorithm captures
the JAVADOC as description of `writeFunc` functionality and the sequence of
method calls inside the functionality to successfully execute a write operation in
a cell. The `println` command of line 18 will be skipped by the algorithm as well
as any other non necessary information.

**Decorate Functionalities**

After some functionality usages were captured by the algorithm, PUG
enables producers to decorate the captured functionalities with additional
information in an input form. This information is data that affects the state of
the captured functionalities and can be used by consumers to understand the
behavior of the component and to create their own test cases. For instance, in an
e-mail component, for `sendSimpleMessage` functionality, an input address such
as “test@@address” is classified as “incorrect usage” that posses the message:
“INVALID E-MAIL ADDRESS” as expected result.

When the algorithm to capture usage information is concluded, a window
(at Figure 4.13, mark 2) appears to let producers decorate the functionality
captured with possible input groups and its expected results. It is possible to
associate more than one input group to a functionality, and type, value and
description to each input.

The aggregation of this information is valuable to consumers when they
are elaborating tests to validate basic functionalities of the candidate component
and tests with their own acceptance criteria to check integration of the
candidate component with the system under development.

![Figure 4.12 – Component OSDReadWrite Main Interface](image-url)
Chapter 4 – A Component Testing Approach Supported by a Case Tool

After usage information and additional information were collected, the data is consolidated in a XML file called *Usage Manifest* that PUG generates in an area specified by the producer. Other approaches, such as (Bundell et al., 2000) and (Teixeira et al., 2007), use XML to format the data provided to consumers. The generated file, called *usage-manifest.xml* can be attached by producers to the component before publishing it to market or open-source repositories.

Figure 4.12 presents a model of the information that compounds the DTD file of the *Usage Manifest*. Each *Usage Manifest* tag is a sequence of one or more *Usage Policy* tags. If usage information is discovered in a method parsed by the algorithm a *Usage Policy* tag is created. Each *Usage Policy* tag contains attributes for functionality name, functionality description (captured from JAVADOC) and inner tags: *Step Policy* to store a sequence of statements to correctly accomplish a functionality, and *Test Information* to store possible inputs for a given functionality and its expected results.
Next Section describes CTP and its main features to be used by component consumers.

4.7.6. Towards Supporting Component Consumer Tester

In order to support component consumers to understand and test a candidate component, and assuming they have received a candidate component with a *Usage Manifest* attached, an Eclipse plug-in was implemented. The plug-in, called CTP (Component Test Perspective), combines visual and analytical support to guide testers at component consumer side to validate a candidate component covering the activities defined by the workflow of Section 4.6.

CTP has features to:

- register reuse points of the system under development;
- a pattern verifier to check if a test case under construction contains the commands to correctly accomplish a functionality;
- a visual representation of the functionalities provided in the *Usage Manifest*;
- functionalities to improve component understanding level; and
- code assist support to facilitate when testers are creating integration and basic test cases to validate a candidate component.

Figure 4.15 presents a use case diagram of the basic functionalities of CTP. Next, each functionality is described in detail.
Load Candidate Component/Load Usage Manifest

Once CTP is opened component consumers can load a usage-manifest.xml file, or a *.jar file (representing a component) containing a usage-manifest.xml file in it. Alternatively, the opened Usage Manifest can be linked to a JAVA project under development in Eclipse to facilitate integration testing. This project can be a component based system where candidate components are plugged in. In Figure 4.16 (mark 1), CTP is loaded with a usage manifest that specifies write functionality of a component. Functionalities from a Usage Manifest file are presented in a tree view with the name of the functionality as root node and commands to accomplish a functionality (presented in order) as child nodes.

Component Test Information

To support consumers to understand external components, CTP provides different types of information about the component. In Figure 4.14 (mark 2), a popup window accesses other features of CTP once a node is selected. Testers can access information about a functionality by selecting: Component Information - to access different types of information about the component; Test Template - to be presented to a test template of the functionality selected; and Test Data - to access the input data, if the producers have decorated the Usage Manifest with additional data.
For instance, *Component Information* feature (in Figure 4.13, mark 3) presents a description of the functionality related to the node selected. Among different types of information a regular expression generated from the functionality is also available. The use of regular expressions generated from usage information to support third-party testing is also suggested at (Liu et al., 1999) and its presence is also important to the pattern verifier.

**Figure 4.16 – CTP and its Functionalities**

In addition, by using reflection mechanisms provided by JAVA language, component main interface is presented to testers.

**Associate Reuse Points to Functionalities**

To support the analysis that testers should conduct to understand how to use a component for validating its basic functionalities; and also to create integration tests, CTP can link a candidate component to a JAVA project. Figure 4.17 presents how component consumers can link a candidate component to a JAVA project. Optionally, the candidate component can be added to the project *classpath* together with testing frameworks that can be used to implement test cases (currently *JUnit* and *testNG* are supported). In addition, as a preparation to create test cases, testers can register the mapping of identified reuse points of the system under development to functionalities of the candidate component to
requirements identifying reuse opportunities. Also, types of strategy to test a
functionality of candidate components, such as random testing, boundary value
or partition, can be registered.

Figure 4.17 – Integration with Development Environment and Identification of
Interactions

Pattern Verifier

CTP provides support to test cases construction. A pattern verifier can
check if the set of method calls at the tree view is present in a test case under
construction. Figure 4.18 depicts a part of a test case (testWrite) elaborated
by the consumer to check the write functionality of the component. When
comparing testWrite with writeFunc (the functionality from Usage
Manifest) two methods are in sequence, and four method calls, from the five
existent at the tree view, were found. The absent methods are presented in bold
at details table. Method calls to other functionalities not related to the pattern
are not considered by the verifier. This functionality is useful to support testers
when many method calls are required to be executed in some sequence due to
variable or context dependencies. Also, when creating integration tests the
pattern verifier helps testers to ensure the adherence of the test to the
functionality under test.
CTP pattern verifier uses regular expressions generated from usage information to analyze if all required methods to correctly accomplish a given functionality are present in the test case under development and if they are in the right order. For instance, for `writeFunc` functionality from Figure 4.15 (Selection), CTP generates the following regular expression considering any possible statement:

\[(w^*(getInstance)+w^*(setMode)+w^*(openBuffer)+w^*(pushData)+w^*(closeBuffer)+w^*)\].

**Code Assist**

Another functionality of CTP is code assist support. Code assist is present in many development environments. It reduces the typing/analytical effort of programmers providing suggestions of probable next commands to be written. However, one issue of code assisters is the generality of the suggestions provided, usually, in lists with many items. CTP manipulates Eclipse’s code assist to provide better suggestions based on the functionality selected in the tree view.

Figure 4.19 (mark 1), demonstrates CTP’s suggestion to the next command after a `getInstance()` command, which is a `setMode(char mode)` command. In addition, the stamp “[CTP Context - writeFunc]” appears
in the end of the suggestion, indicating that this suggestion is related to the context of use of [writeFunc] functionality. The suggestions provided by CTP depend on the functionality selected in the tree view. In Figure 4.17 (mark 2), another functionality of the component (loadAsText) appears. However, since loadAsText functionality it is not part of [writeFunc] usage it appears without the stamp and in a lower level of the suggestions provided. If no functionality is selected in the tree view, Eclipse standard code assist is invoked providing suggestions, for instance, in alphabetical order.

![CTP code Assist](image)

**Figure 4.19 – CTP code Assist**

### 4.7.7. Requirements Compliance

PUG and CTP were defined to be compliant with the defined set of functional and non-functional requirements. Being developed as Eclipse plug-ins, PUG and CTP adhere to requirement (NFR2) related to integration to development environment.

At component producer side, PUG packs usage information collected by an algorithm in a XML file called *Usage Manifest*. This capability accomplishes (FR1) related to provision of *data in a standard format* as well as (NFR1) related to *extensibility* because new tags can be added to the XML file to incorporate new types of data. In addition, after the capture algorithm selects relevant statements to correctly execute a given functionality, producers can add test inputs that affect the state of a functionality captured. Thus, these capabilities accomplish (FR2) related to *input data collection* and (FR3) related to *usage capture*. 
At component consumer side, CTP processes the XML file received and *presents to the user* the functionalities available in the component in a hierarchical manner. This capability complies with requirement (FR4). Furthermore, consumers can *register the identified reuse points* of their system under development and interactions with the candidate component complying with (FR5). In addition, with respect to (FR6) and (NFR3), CTP has features to provide information about the component such as textual descriptions and test templates. These features in conjunction with Pattern Verifier and Code Assist can support consumers to understand candidate components and construct test cases. The plug-ins can be used to support component testing of any component implemented in JAVA regardless of development process, architecture or technique used to implement. Thus, this capability complies with (NFR4) that is related to *architecture independence*.

### 4.8. Summary

This Chapter presented the proposed approach to support component testing. We propose workflows of activities to be performed at component producer side and at component consumer side following guidelines to aid component stakeholders to conduct testing activities. Moreover, we presented a tool integrated to Eclipse development environment to cover the activities of the proposed workflows. Finally, we illustrated the usage of the tool, a detailed description of its features and how implementation accomplishes requirements.

Next Chapter presents an experimental study of the proposed approach and the use of the tool, in order to evaluate the effectiveness of the approach and the applicability of the tool.
When areas for improvement are identified in an organization through some form of assessment it is difficult to determine how these areas may be addressed to cope with the identified problems. Transferring new software engineering methodologies and tools into active practice may find some resistance because research and practice in software development have been divided organizationally and by cultural values impeding good communication (Basili et al., 1991).

In this sense, this chapter presents an experimental study to analyze the impact of the proposed approach to support component testing as well as the tool support that covers the activities of the proposed approach in the context of component testing. Before discussing the experimental study defined, it is necessary to introduce some definitions to clarify its elements.

5.1. Introduction

According to Wohlin et al. (Wohlin et al., 2000), when conducting an experiment, the goal is to study the outcome when are varied some of the input variables to a process. Those variables can be of two kinds: independent and dependent.

The variables that are objects of the study which see the effect of the changes in the independent variables are called dependent variables. Often there is only one dependent variable in an experiment. All variables in a process that are manipulated and controlled are called independent variables.

Experiments study the effect of changing one or more independent variables. Those variables are called factors. The other independent variables
are controlled at a fixed level during the experiment, or else it would not be possible to determine if the factor or another variable causes the effect. A treatment is one particular value of a factor.

The treatments are being applied to the combination of objects and subjects. An object can, for example, be a suite that will be constructed using different test techniques such as boundary value or partition tables. The people that apply the treatment are called subjects. At the end, an experiment consists of a set of tests where each test is a combination of treatment, subject and object.

5.2. The Experimental Study

According to Wohlin et al. (Wohlin et al., 2000), the experimental process can be divided into the following main activities:

- **Definition.** This step addresses the experiment definition in terms of its problem, objective and goals;

- **Planning.** In this step, the model presents three main concerns: the design of the experiment, the definition of the instrumentation and the identification of the possible threats;

- **Operation.** In this activity, there are two main sub-activities: the analysis and the interpretation. In this step, all experiment measurements are collected according to the planned activities; and

- **Presentation/Package.** These steps represent the activities for presenting and packaging the set of results generated after the analysis and interpretation activities.

The experimental plan presented follows the model proposed in (Wohlin et al., 2000). Additionally, the experiments defined in (Barros, 2001), (Vegas et al., 2005) and (Almeida, 2007) were used as inspiration. The definition and planning activities will be described in future tense, showing the logic sequence between the planning and operation.
5.2.1. The Definition

In order to define the experimental study, the GQM paradigm (Basili et al., 1994) was used. According to the paradigm, the main objective of this study is to:

5.2.1.1. Goal

G1: To analyze an approach to support component testing for the purpose of evaluating it with respect to efficiency and difficulties of its use from the point of view of researchers and testers in the context of component testing.

5.2.1.2. Questions

Q1: Does the approach to support component testing increase the coverage of test cases elaborated by consumers?

Q2: Do the subjects have difficulties to apply the approach?

Q3: Do the Eclipse plug-ins provide effort reduction to component producers to prepare a component to be further tested; and to component consumers understand and create test cases for external components?

5.2.1.3. Metrics

M1. Relative Test Coverage. According to (Konda, 2005), test coverage is defined as the extent to which testing covers the product’s complete functionality. Coverage could be with respect to requirements, functional topic list, business flows, use cases, among other assets. It can be calculated based on the number of items that were covered vs. the total number of items. Test coverage metric is commonly used as a criterion to stop testing. In the context of this experiment, it will compare the number of test cases elaborated by consumers to cover functionalities of an external component to a list of tests that serves as the basis for counting the total number of tests identified for a specific component and a system under development.

According to Hutcheson (Hutcheson, 2003), such a list is defined as the test inventory of a system. The value of this test coverage metric depends on the quality and completeness of the test inventory. Thus, with the most precise test inventory as possible, and tests elaborated by consumers with, and without the
proposed approach, it will be possible to analyze the test coverage relative to a specific test inventory with the following indicator – *Relative Test Coverage* (RTC):

\[ \text{RTC} = \frac{\sum \text{Functionalities covered by Consumer}}{\sum \text{Functionalities from Test Inventory}} \times 100 \]

Additionally, it is important to establish two thresholds to RTC metric to act as checkpoints and help in monitoring the collected data. The minimum threshold level will indicate that the collected data being lower than this value represents a problem. In the same way, the maximum threshold level will indicate that the collected data being greater than this value also represents a problem. Thus, this experimental study will arbitrarily consider the RTC values lower than 50% and higher than 95% as suspicious results. In other words, the data from a subject covering less than half of the functionalities of a component and almost 100% will be considered suspicious.

**M2. Difficulty to understand the approach.** In order to identify possible weaknesses and misunderstanding in the approach, and to define improvements, it is necessary to identify and analyze the difficulties found by users using the approach. Thus, metrics related to difficulty will measure this issue:

- **D_CP:** %Subjects that had difficulties to accomplish the activities of the workflow at component producer side and the difficulties distribution.

- **D_CC:** %Subjects that had difficulties to accomplish the activities of the workflow at component consumer side and the difficulties distribution.

It is important to emphasize that these metrics were never used before, and thus there are no well-known values for them. Thus, arbitrary values were chosen, based on practical experience and common sense. Values equal or higher than 30% for D_CP, and 30% for D_CC, are considered as an indicative that the approach, in this particular step, is too difficult and should be improved.

**M3. Effort Variation.** One of the issues identified in current component testing tools was their lack of integration to development environments and CBSD processes. Thus, in order to analyze possible benefits to component
producers and consumers when using PUG and CTP plug-ins to accomplish the activities of the proposed workflows the indicator, *Effort Variation* (EV) was defined to compare the effort of subjects when acting as component producers and consumers:

$$ EV = \frac{E_{wp}}{E_{wOp}} $$

*E_{wp}*: *Average Effort to subjects execute a workflow using a plug-in*; and

*E_{wOp}*: *Average Effort to subjects execute a workflow using a template*.

The effort variation metric has range \([0, \infty)\), where EV \(\geq 0\) indicates the maximum effort reduction when using a plug-in to execute one of the workflows, and EV \(\geq 1\) indicates that the effort to execute one of the workflows using a plug-in is equal, or even higher than not using it.

### 5.2.2. The Planning

After the definition of the experiment, the planning is started. The definition determines the foundations for the experiment, the reason for it, while the planning prepares for how the experiment is conducted.

**Context.** The objective of this study is to analyze the impact of applying the proposed approach to support component testing to improve a component-testing process. The study will be conducted following principles of the Quality Improvement Paradigm (QIP) (Basili, 1985), which is, in general lines, a cyclical process for planning organizational improvement on the basis of: understanding the current baseline\(^{19}\), setting quantifiable goals for improvement, choosing development processes to meet those goals, measuring whether the goals are met, and abstracting lessons learned about the conditions that led to success or failure. Thus, first the subjects will be oriented to act as component stakeholders according to their own personal knowledge in the area of component testing to collect referential data (this is what is considered a baseline in the context of this experiment), and then they will execute the proposed workflows at both component producer and consumer sides. Based on

---

\(^{19}\) According to Cambridge Dictionary - Baseline: “An imaginary line used as a starting point for making comparisons” at: http://dictionary.cambridge.org/define.asp?key=6131&dict=CALD
the defined metrics, an analysis will be conducted comparing the proposed approach to the baseline to identify possible benefits and/or trade-offs.

Table 5.1 presents the assignation of roles, defined randomly, to the subjects of the experiment. All the subjects will act, depending on the phase of the experiment, as a component producer focused on preparing a component to be tested by third party, or as a component consumer understanding the component behavior and elaborating test cases to validate an external component. The way the subjects interact with a component will also depend on the phase under consideration. For instance, in Phase 1 (execution of the baseline), all the subjects will perform the roles of producer and consumer according to their own knowledge (ad hoc) in the area. However, in Phase 2 only subjects ID1, 2, 3 and 4 will participate using exclusively textual templates (see Appendix B and C) to accomplish the proposed workflows instead of using the tool support. In phase 3, only subjects ID 5, 6, 7 and 8 will participate using PUG to execute the workflow at component producer side, and CTP to execute the workflow at component consumer side.

The design of this experimental study, according to standard design types presented by (Wohlin et al., 2000), is a case of an experiment of two-stage nested design with two factors (the baseline and the proposed component testing approach). In addition, it was used balancing and randomization to distribute the treatments (the roles) to subjects.

Table 5.1 – Assignation of roles to subjects

<table>
<thead>
<tr>
<th>ID/Role</th>
<th>Baseline</th>
<th>Proposed Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Producer</td>
<td>Consumer</td>
</tr>
<tr>
<td>ID1</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>ID2</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>ID3</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>ID4</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>ID5</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>ID6</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>ID7</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>ID8</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>
Subjects. To augment the variability of profiles participating to this experimental study, among the eight subjects, there are four M.Sc. students in Software Engineering from the Federal University of Pernambuco, Brazil. In addition, there are also four employees from a large software company located in Pernambuco, Brazil to participate as well. The subjects will perform, in different moments, the role of component producers preparing their component to be tested by third party, as well as the role of component consumers to test an external component before integration to a system under development.

Training. The training of the subjects will be conducted in meetings at the university for the M.Sc. students and at the company to the employees. The training will be divided in two steps. First, general concepts such as component-based development, software testing, and software reuse will be explained briefly in two meetings of two hours. To prevent the proposed approach to influence on the execution of the baseline, specific training about the proposed approach will be conducted in three meetings only after the baseline is executed. This training, in three meetings of two hours, will encompass detailed explanation about each activity of the workflows, their objectives as well as how to use PUG plug-in at component producer side, and CTP plug-in at component consumer side or the templates. In addition, in both steps, the context of the experiment, the components used, and the roles that the participants will perform will be explained and discussed with the subjects.

Instrumentation. All the subjects will receive a questionnaire (QT1) about his/her education and experience. In addition, the subjects will receive a second questionnaire (QT2) for the evaluation of the subjects’ satisfaction/difficulties using the proposed approach and the plug-ins. The questionnaires compose the Appendix A. To guarantee more precision to the data collected, specially data related to the effort/productivity, the subjects were oriented to use a time tracking program\textsuperscript{20} to register the time elapsed while performing the experiment.

Criteria. The quality focus of the study demands criteria that evaluate the benefits obtained by the use of the proposed approach and the plug-ins. The

\textsuperscript{20} The subjects have used a shareware version of \textit{AllNetic Time Tracker 2.1}, at: http://www.allnetic.com/download.html
benefits obtained will be evaluated quantitatively through an analysis of the relative test coverage obtained by the subjects from the baseline in comparison to what was obtained using the approach. The benefits of PUG and CTP support to aid the execution of the activities of the workflows will be analyzed by computing the effort variation metric, for instance, of the effort of the subjects to accomplish the workflows filling textual templates with information dedicated to testing, to the effort of the subjects using PUG; or by the effort variation of the subjects acting as consumers to create test cases with, or without, CTP code-assist.

**Null Hypothesis.** This is the hypothesis that the experimenter wants to reject with a high significance as possible. In this study, the null hypothesis determines that the introduction of the proposed approach to support component testing in component-based development projects will not represent improvement to component producers, or to component consumers.

\[
H_0_1: \mu_{RTC_{Baseline}} > \mu_{RTC_{Approach}}
\]

\[
H_0_2: \mu_{DCP} \geq 30\%
\]

\[
H_0_3: \mu_{DCC} \geq 30\%
\]

\[
H_0_4: \mu_{EV} \text{ at producer side } \geq 1
\]

\[
H_0_5: \mu_{EV} \text{ at consumer side } \geq 1
\]

**Alternative Hypothesis.** This is the hypothesis in favor of which the null hypothesis is rejected. In this study, the alternative hypothesis determines that the introduction of the proposed approach to support component testing in component-based development projects will represent an improvement or to component producers, or to component consumers.

\[
H_1_1: \mu_{RTC_{Baseline}} < \mu_{RTC_{Approach}}
\]

\[
H_1_2: \mu_{DCP} < 30\%
\]

\[
H_1_3: \mu_{DCC} < 30\%
\]

\[
H_1_4: \mu_{EV} \text{ at producer side } < 1
\]

\[
H_1_5: \mu_{EV} \text{ at consumer side } < 1
\]
**Independent Variables.** In this study, all variables in a process that are manipulated and controlled are called independent variables. The independent variables are the components and systems used, the experience of the subjects, and the technology.

**Dependent Variables.** The dependent variables are the variables that are objects of the study which are necessary to see the effect of the changes in the independent variables. The quality of the proposed approach will be measured by the relative test coverage that consumers are able to formulate using the approach. The benefits of PUG and CTP will be measured by an analysis of the effort variation when producers or consumers are executing the approach with, and without the plug-ins.

**Qualitative Analysis.** The qualitative analysis aims to evaluate the difficulty of the application of the proposed approach and the quality of PUG and CTP. This analysis will be conducted through a questionnaire (Appendix A). This questionnaire is very important because it will allow evaluating the difficulties that the subjects have with the approach and when using the plug-ins to execute the approach. It will support an analysis of the training and the provided material to the subjects.

**Randomization.** This technique can be used in the selection of the subjects. Ideally, the subjects must be selected randomly from a set of candidates. Randomization will be applied to assign the subjects to phases 2 or phase 3.

**Blocking.** Blocking is used to systematically eliminate the undesired effect in the comparison among the treatments. In this study, it was not identified the necessity of dividing the subjects into blocks, since the study will evaluate just one factor, which is the use of the approach.

**Balancing.** In some experiments, balancing is desirable because it both simplifies and strengthens the statistical analysis of the data. Balancing was used since the subjects will only have training related to the proposed approach after the execution of the baseline. In addition, subjects cannot be assigned to execute both phases 2 and 3 because the knowledge acquired in one phase can affect how they will execute the other.
**Internal Validity.** The internal validity of the study is defined as the capacity of a new study to repeat the behavior of the current study, with the same subjects and objects with which it was executed (Wohlin et al., 2000). The internal validity of the study is dependent of the number and expertise of the subjects. This study is supposed to have at least eight subjects with similar testing and reuse background to guarantee a good internal validity.

**External Validity.** This external validity aims to measure the capability of the study to be affected by generalization. It implies in considering the capability to repeat the same study in other research groups (Wohlin et al., 2000). Due to the time constraints, it was not possible to apply the study in other research groups. Nevertheless, the external validity of the study is considered sufficient, since it aims to analyze the impact of the proposed approach to support component testing compared to a baseline. Thus, additional studies can be planned with the same profiles of subjects.

**Construct Validity.** This validation aims to measure the relation between the theories that are to be proved and the instruments and subjects of the study (Wohlin et al., 2000). The study will use a common problem of any software developer willing to reuse a component instead of building it from scratch, which is to understand and validate the provided functionalities before integrating the candidate component to the system under development.

**Conclusion Validity.** This validation determines the capability of the study to generate conclusions (Wohlin et al., 2000). The analysis and interpretation of the results of this experiment will be described using descriptive statistic.

**Validity Threats.** The following validity threats related to this experimental study were identified:

- **Lack of Historical Data:** Due to lack of data related to testing from real component producers and consumers, the experiment was designed to analyze possible benefits of the proposed approach compared to a baseline. Such analysis, without any data from real practice for calibration may not express the real gain (or loss) of the proposed approach if applied to a real CBD project;
• **Boredom:** Subjects may find the experiment boring and, therefore, their performance may be below normal. A prize will be offered to the subjects in the end of the experiment as an attempt to overcome this threat. Furthermore, the subjects who have performed the experiment are volunteers, which mean that they should have at least some interest in the subject;

• **Lack of Specific Skills:** None of the subjects work exclusively and specifically with software testing, thus they can find difficult to execute some of the required activities such as definition of test scenarios and elaboration of test cases. Testing is an area that depends mostly on the skills of testers. According to (Kaner et al., 2001), “testing is applied epistemology which is the study of how you know what you know”. Good testers must think technically, creatively, critically, and practically. To mitigate this threat, books and other references will be made available to the subjects;

• **Object Learning:** To prevent subjects from acquiring unconscious formalization (Vegas et al., 2005), which is a recurrent threat when one approach (in this case, the proposed one) is more formal than the other (the baseline, which can be considered *ad hoc*), subjects will be oriented to act as producers and consumers first with the baseline. However, by executing the baseline first, subjects may have their knowledge on the testing area improved. This threat may increase the quality of the test cases produced when executing the proposed approach reducing the reliability of the output data;

• **Test Inventory Reliance:** The *Test Inventory* will provide, for a given component, a set containing *all-possible-tests* that can be elaborated to the component and a system under development. Thus, the precision of the metric *Relative Test Coverage* will depend on the completeness of the test inventory. To overcome this threat, a *test expert* may conduct an inspection in the test inventories;

• **Limited Component Knowledge:** In this experimental study the subjects will be asked to perform the role of component producers
preparing their component to be tested by third party. However, they
did not participate to the construction of the components. As a result,
the information elaborated to be provided to consumers may vary in
quality. To mitigate this threat, the information provided to subjects
when acting as consumers will not be from other subjects.
Furthermore, the information dedicated to testing generated by
subjects when acting as producers will not be quantitatively analyzed,
only the effort to elaborate it in the baseline, in the proposed
approach using templates and using PUG;

- **Analytical data obtained from different methods:** The
analytical data obtained from the proposed approach will come from
different sources: from the execution of the proposed approach
through the plug-ins (PUG and CTP), and from execution of the
proposed approach from textual templates (Appendixes B and C).
Thus, for instance, if PUG or CTP contain bugs it may influence in the
results. In the same way, the results from the execution of the
approach through the templates may be boring to subjects reducing
the precision of the results. To mitigate this threat the quantitative
analysis will show the results from both forms of execution separated
instead of only presenting the average.

### 5.2.2.1. Components used in the Experimental Study

In order to represent the main characteristic of software components,
which is their heterogeneity, the scope of this experimental study will focus on
open-source components. In addition, another reason for choosing open-source
components is that they are typically subject of *opportunistic reuse*.
Opportunistic reuse differs from *predictive reuse*, which is the case of software
product lines (Atkinson et al., 2000) where software artifacts are created only
when reuse is predicted in one or more products of a family.

Four open-source components with similar complexity levels will be used.
Two components will represent components that producers should prepare to
be further tested before publishing to repositories. Another two will represent
external components that consumers should understand and elaborate test cases before recommending their integration to the system under development, or even their rejection.

On one hand, when acting as a component producer the subjects will receive a component with its source code and documentation about the component such as user manuals and/or JAVADOCS. On the other hand, when acting as a component consumer the subjects will receive the component object code (such as a *.jar file), and additional information to test this component depending on which context they are executing, the baseline context or the proposed approach context. To represent the system where consumers should plug a candidate component the same component the subject has prepared to be tested when acting as producers will be used. In other words, as showed in Table 5.2, if ID 1 has documented component jexcelapi when acting as a producer in Phase 2, ID 1 will use, when acting as a consumer, component jexcelapi as the system under development, and will use itext as candidate component.

Table 5.2 shows the distribution of the components to the phases of the experiment. In addition, a brief description of each component will be conducted.

<table>
<thead>
<tr>
<th>ID/Role</th>
<th>Producer</th>
<th>Consumer</th>
<th>Baseline</th>
<th>Proposed Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Phase 1</td>
<td>Phase 2</td>
</tr>
<tr>
<td>ID1.</td>
<td>java_mp3</td>
<td>jfreechart</td>
<td>jexcelapi</td>
<td>itext</td>
</tr>
<tr>
<td>ID2.</td>
<td>java_mp3</td>
<td>jfreechart</td>
<td>jexcelapi</td>
<td>itext</td>
</tr>
<tr>
<td>ID3.</td>
<td>java_mp3</td>
<td>jfreechart</td>
<td>jexcelapi</td>
<td>itext</td>
</tr>
<tr>
<td>ID4.</td>
<td>java_mp3</td>
<td>jfreechart</td>
<td>jexcelapi</td>
<td>itext</td>
</tr>
<tr>
<td>ID5.</td>
<td>java_mp3</td>
<td>jfreechart</td>
<td></td>
<td>jexcelapi</td>
</tr>
<tr>
<td>ID6.</td>
<td>java_mp3</td>
<td>jfreechart</td>
<td></td>
<td>jexcelapi</td>
</tr>
<tr>
<td>ID7.</td>
<td>java_mp3</td>
<td>jfreechart</td>
<td></td>
<td>jexcelapi</td>
</tr>
<tr>
<td>ID8.</td>
<td>java_mp3</td>
<td>jfreechart</td>
<td></td>
<td>jexcelapi</td>
</tr>
</tbody>
</table>
java_mp3\textsuperscript{21}. java_mp3 is a Java component to read and modify ID3 and ID3v2 tags on MP3 files and to gather extended information about MP3 files.

jfreechart\textsuperscript{22}. jfreechart is a Java component for drawing statistical data. It supports bar charts, pie charts, line charts, time series charts, scatter plots, histograms, simple Gantt charts, Pareto charts, bubble plots, dials, and thermometers.

jexcelapi\textsuperscript{23}. jexcelapi is a Java component which provides the ability to read, write, and modify Microsoft Excel\textregistered spreadsheets.

itext\textsuperscript{24}. itext is a Java component that contains classes that generate documents in the Portable Document Format (PDF) and/or HTML.

To balance the effort of the subjects, the scope of the components was reduced. For component jfreechart subjects were recommend to focus attention only on two graphics (Bar and Pie). For component jexcelapi, subjects were recommended to focus attention on the capability to read Excel spreadsheets. Finally, for the itext component, the subjects were recommended to focus on writing in PDF files only. Table 5.3 shows some information related to the four components used. It is presented the number of classes, the number of lines of code and the complexity (McCabe, 1976) of each component.

<table>
<thead>
<tr>
<th>Components</th>
<th>Classes</th>
<th>Lines of Code</th>
<th>McCabe’s Cyclomatic Complexity (mean)</th>
</tr>
</thead>
<tbody>
<tr>
<td>java_mp3</td>
<td>33</td>
<td>3126</td>
<td>2.127</td>
</tr>
<tr>
<td>jexcelapi</td>
<td>465</td>
<td>40808</td>
<td>1.998</td>
</tr>
<tr>
<td>itext</td>
<td>571</td>
<td>61553</td>
<td>2.305</td>
</tr>
<tr>
<td>jfreechart</td>
<td>475</td>
<td>66907</td>
<td>2.325</td>
</tr>
</tbody>
</table>

5.2.3. The Operation

Experimental Environment. The experimental study was conducted during May-June 2008 entirely at the organization for its employees, and at Federal University of Pernambuco and/or at home for the M.Sc. students.

\textsuperscript{21} http://www.vdheide.de/java_mp3/ (last visit: June 1, 2008)
\textsuperscript{22} http://sourceforge.net/projects/jfreechart/ (last visit: June 1, 2008)
\textsuperscript{23} http://sourceforge.net/projects/jexcelapi/ (last visit: June 1, 2008)
\textsuperscript{24} http://sourceforge.net/projects/itext/ (last visit: June 1, 2008)
Training. The subjects were trained according to plan. First, they were introduced to general ideas before executing the baseline. Later, the subjects were introduced to the concepts of the proposed approach. Table 5.4 shows Subject’s profiles in the Experimental Study.

Costs. The employees of the software organization were oriented to use their self-study time at the organization to perform the experiment. The environment for execution for the M.Sc. students was university’s labs and subject’s houses.

<table>
<thead>
<tr>
<th>ID</th>
<th>Years since Graduation</th>
<th>Participation in Industrial Projects</th>
<th>Experience in Software Testing</th>
<th>Experience in Software Reuse</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>4-5 low complexity 1-2 medium complexity</td>
<td>academic - medium academic - low</td>
<td>commercial – medium commercial - low</td>
</tr>
<tr>
<td>2</td>
<td>1,5</td>
<td>2-3 low complexity 1-2 medium complexity</td>
<td>academic - low academic - low</td>
<td>commercial – low commercial - low</td>
</tr>
<tr>
<td>3</td>
<td>0,5</td>
<td>1-2 low complexity 1-2 medium complexity</td>
<td>academic - low academic - none</td>
<td>commercial – low commercial - low</td>
</tr>
<tr>
<td>4</td>
<td>1,5</td>
<td>1-2 low complexity 1-2 medium complexity</td>
<td>academic - medium academic - low</td>
<td>commercial – medium commercial - low</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>4-5 low complexity 1-2 high complexity</td>
<td>academic - low academic - high</td>
<td>commercial – low commercial - low</td>
</tr>
<tr>
<td>6</td>
<td>19</td>
<td>&gt;5 low complexity &gt;5 medium complexity &gt;10 high complexity</td>
<td>academic - low academic - medium</td>
<td>academic - none commercial - none</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>2-3 medium complexity 1-2 high complexity</td>
<td>academic - low academic - none</td>
<td>commercial – low commercial - low</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>4-5 low complexity 1-2 medium complexity</td>
<td>academic - medium academic - low</td>
<td>commercial – medium commercial - low</td>
</tr>
</tbody>
</table>

5.2.3.1. Analysis and Interpretation

Training Analysis. The training was performed as planned. However, it is important to notice that during the execution of the experiment some subjects had doubts related testing area (e.g. some of them were considering to construct test cases with JUnit, others had difficulties to identify test scenarios in the components). The doubts were overcome with the provision of proper reference.

Quantitative Analysis. The quantitative analysis was divided in three analyses: an analysis of the relative test coverage comparing results from the proposed approach and the baseline, an analysis of the difficulties to understand the proposed approach and a comparison of the effort variation
when using textual templates of the proposed approach to the plug-ins PUG and CTP.

**Relative Test Coverage Analysis.** Such as explained, this indicator will analyze the coverage of the test cases elaborated by a component consumer with, and without the proposed approach compared to a test inventory. A consolidation of the data collected from subjects is shown in Table 5.5 in a paired format. This means that the columns in dark grey represent the results from subjects (ID 1, 2, 3, 4), and the columns in light grey represent the results from subjects (ID 5, 6, 7, 8).

### Table 5.5 – Results for Relative Test Coverage

<table>
<thead>
<tr>
<th>Measure</th>
<th>Baseline</th>
<th>Proposed Approach Using Template</th>
<th>Baseline</th>
<th>Proposed Approach Using CTP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>69.75%</td>
<td>79.20%</td>
<td>70.50%</td>
<td>75.76%</td>
</tr>
<tr>
<td>Maximum</td>
<td>82.00%</td>
<td>84.71%</td>
<td>82.00%</td>
<td>85.00%</td>
</tr>
<tr>
<td>Minimum</td>
<td>55.00%</td>
<td>71.71%</td>
<td>60.00%</td>
<td>62.00%</td>
</tr>
<tr>
<td>Standard Deviation</td>
<td>11.59%</td>
<td>5.88%</td>
<td>9.00%</td>
<td>9.89%</td>
</tr>
<tr>
<td>Null Hypothesis</td>
<td>µRTCBaseline &gt; µRTCApproachTemplates</td>
<td>µRTCBaseline &gt; µRTCApproachTemplates</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

As can be seen in Table 5.5 the mean of the relative test coverage executing the approach through templates (79.20%) was higher then with CTP support (75.76%) by 3.44%. This may indicate that some subjects had difficulties to use CTP to support the accomplishment of the workflow or had difficulties to elaborate test cases due to limited software testing background. Another reason for such an indication is that the minimum value of column *Proposed Approach Using CTP* was 62%. This means that some of the subjects have only elaborated tests covering less than 70% of component functionalities, this level of coverage can probably be improved with more training. However, none of the maximum or minimum measures reached the defined thresholds and cannot be considered suspicious. The coverage results from the execution of the proposed approach through the template was the most homogeneous, with a standard deviation of 5.88%. This can indicate that the template of Appendix C were not ambiguous.

Nevertheless, both executions (with or without tool support) reject the null hypotheses: $\mu_{RTC_{Baseline}}$ (69.75%) < $\mu_{RTC_{ApproachTemplates}}$ (77.20%) and
\( \mu_{\text{RTCBaseline}} \ (70.50\%) < \mu_{\text{RTCApproach}} \ (75.76\%) \). This validates the alternative hypothesis \( H_1: \mu_{\text{RTCBaseline}} < \mu_{\text{RTCApproach}} \).

**Difficulty to Understand the Approach.** All the subjects have answered the questionnaire QT2 (Appendix A). To analyze their answers when accomplishing the proposed approach, it is required a separation of the difficulties reported related to the component producer side and related to the component consumer side.

**Difficulties at Component Producer Side.** Among the difficulties found at component producer side, four subjects, 50% from the total, had reported difficulties. One subject (ID 6), representing 12.5% from the total, had difficulties to collect information to prepare documentation dedicated to support further testing. Also, four subjects (ID 2, 3, 6, 8), representing 50% had difficulties to select functionalities of the component to be detailed to component consumers. These numbers **did not reject** the defined null hypothesis (\( \mu_{\text{DCP}} \geq 30\% \)). However, it is necessary to highlight that this value for the null hypothesis was defined without any previous data, since it was the first time that this aspect was analyzed, differently from the other metrics, which had pre-established values. Nevertheless, the next time that the experiment is performed this value can be refined based on this experience, resulting in a more calibrated metric. Figure 5.1 shows the histogram with the distribution density of the found difficulties.

![Figure 5.1 – Difficulties at Component Producer Side](image_url)
According to the Figure 5.1, one subject (ID 6) had difficulties in *Collect Information* activity. Although it was not provided, the subject suggested that the most relevant information to support the elaboration of documentation of a component to be further tested is the use cases document, however, only source code, user manual, JAVADOC were provided to subjects. Four subjects (ID 2, 3, 6, 8) had difficulties in *Analyze Information* activity. All the four subjects suggested that detailed guidelines about how to document functionalities of a component are required. For one subject (ID 8), test cases are the best type of information to be provided to consumers to understand and test a candidate component. In this sense, guidelines to construct test cases can be provided in future executions of this experimental study. Finally, four subjects (ID 1, 4, 5, 7) did not have reported problems at component producer side.

**Difficulties at Component Consumer Side.** Among the difficulties found at component consumer side, six subjects, representing 75% from the total had reported difficulties. Two subjects (ID 3, 6), representing 25% from the total reported difficulties when mapping reuse points and functionalities of a candidate component. Six subjects (ID 2, 3, 4, 6, 7, 8), representing 75% from the total had problems to elaborate test cases. These numbers did not reject the null hypothesis ($\mu_{Dcc} \geq 30\%$). Similar to the other metric related to difficulties to understand the approach, it is important to emphasize that the value for the null hypothesis was defined without any previous data, since it was the first time that this aspect was analyzed, differently from the other metrics, which had pre-established values. Figure 5.2 shows the histogram with the distribution density of the found difficulties.
According to the Figure 5.2, two subjects (ID 3, 6) had difficulties in Map Reuse Points to Functionalities Provided activity, their argument is related to difficulties to determine exactly where reuse points and functionalities of the candidate component interact. However, during the training, it was mentioned that sometimes the interactions (candidate component x system) are not explicit, and glue code can be needed. In this sense, this situation should be more stressed in future. Six subjects (ID 2, 3, 4, 6, 7, 8), had problems in Understand Component and Elaborate Test Cases activity. Their problems were related to test case construction. Three subjects (ID 2, 3, 6) reported difficulties to elaborate test cases. Their arguments were lack of software testing background. This issue is important to identify the necessity to homogenize more the expertise of the subjects with more training in the future. Tree subjects (ID 4, 7, 8) suggested that it was not necessary to specifically construct test cases in this activity. In general their argument was that test scenarios were enough to be defined to easily check functionalities/interactions of the candidate component and the system under development. One subject (ID 7), suggested using test scenarios and exploratory tests to validate the candidate component with less effort. Finally, subjects (ID 1, 5) did not have reported problems at component consumer side.

**Effort Variation.** This indicator has the objective to analyze the benefits of using the plug-ins PUG and CTP in comparison to executing the same activity without them (using templates, see appendixes B and C). A consolidation of the data collected from subjects is shown in Table 5.6 in a paired format. This
means that the columns in dark grey represent the effort results of subjects (ID 1, 2, 3, 4) compared to results of (ID 5, 6, 7, 8) executing the proposed approach at component producer side. The columns in light grey represent the effort results of subjects (ID 1, 2, 3, 4) compared to results of (ID 5, 6, 7, 8) executing the proposed approach at component consumer side.

### Table 5.6 – Results for Effort Variation (in men-hours)

<table>
<thead>
<tr>
<th>Measure</th>
<th>Producer Using Template</th>
<th>Producer Using PUG</th>
<th>Consumer Using Template</th>
<th>Consumer Using CTP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>1,465</td>
<td>1,26</td>
<td>1,70</td>
<td>1,50</td>
</tr>
<tr>
<td>Maximum</td>
<td>1,95</td>
<td>1,83</td>
<td>2,20</td>
<td>2,10</td>
</tr>
<tr>
<td>Minimum</td>
<td>0,58</td>
<td>0,90</td>
<td>1</td>
<td>0,73</td>
</tr>
<tr>
<td>Standard Deviation</td>
<td>0,61</td>
<td>0,42</td>
<td>0,56</td>
<td>0,58</td>
</tr>
<tr>
<td>Effort Variation</td>
<td>0,858</td>
<td></td>
<td>0,883</td>
<td></td>
</tr>
<tr>
<td>Null Hypothesis</td>
<td>µEV at producer side &gt;= 1</td>
<td></td>
<td>µEV at consumer side &gt;= 1</td>
<td></td>
</tr>
</tbody>
</table>

On one hand, an analysis of Table 5.6 shows that the effort of subjects using PUG to prepare a component to be tested by third party (~1,26 men/hour) is the most homogenous with the lowest standard deviation (~0,42). This can indicate that the subjects have understood the usage of PUG. On the other hand, the effort of subjects using textual templates at component producer side (see Appendix B) is the most disperse (~0,61). This can indicate that more training is needed. With respect to the use of CTP, the difference of the means of subjects using templates and CTP indicates effort reduction when using CTP. However, the difference of ~0,2 indicates that more training is required to increase the productivity of subjects when using CTP.

Nevertheless, both stakeholder sides (component producers and consumers) reject the null hypothesis: µEV at producer side (~0,86) < 1 and also µEV at consumer side (~0,88) < 1. These results validate the alternative hypotheses: µEV at producer side < 1 and µEV at consumer side < 1.

**Conclusion.** The analysis of this experimental study cannot be considered conclusive. However, it indicates that: (a) In the context of this experiment, the proposed approach to support component testing allows component consumers to generate test cases with more coverage; and (b) In the context of this experiment, the use of the eclipse plug-ins (PUG and CTP) to
perform the activities of the proposed approach \textbf{provide effort reduction} to users. Although some null hypotheses were not rejected, those hypotheses were related to difficulties of the subjects to use techniques, tools and concepts that were sometimes new to them. Those results are very important to recalibrate training in the future. Problems related to those issues are described next in the qualitative analysis.

\textbf{Qualitative Analysis.} After concluding the quantitative analysis for the experiment, the qualitative analysis was performed. This analysis is based on the answers defined for the QT2 presented in Appendix A.

\textbf{Objective of the proposed approach.} From the questionnaire, 25\% of the subjects considered that the objective of the proposed approach was partially understood. Their main difficult was lack of testing background. However, for 75\% of them considered that the approach was totally understood.

\textbf{Training.} From the questionnaire, 37,5\% of the subjects suggested that more training in testing could improve the results of the experiment. However, for 62,5\% of the subjects, the training, in general, was enough.

\textbf{Execution of activities at component producer side.} For 62,5\% of the subjects, the activities of the approach at component producer side increases the effort of component producers. However, for 37,5\% of the subjects the extra work is little or irrelevant. The additional effort was expected because the approach is based on additional tasks that producers should perform to enable third party to test their components.

\textbf{Execution of activities at component consumer side.} For 25\% of the subjects, the activities of the approach at component consumer side increase the effort of component consumers whereas for 75\% of the subjects, the effort is little or irrelevant. One subject mentioned that mapping the interactions between reuse points and functionalities of the component is very useful because it helps on the elaboration of test cases. In addition, one subject suggested using \textit{test bureau} strategy to execute the activities at component consumer side.

\textbf{Quality of the test cases generated using the approach.} For one subject, 12,5\% the quality of the test cases generated by consumers using the
The approach seems to be the same compared to the Baseline. However, for 87.5% of the subjects, the approach generates better tests at component consumer side.

**Usage of PUG and CTP.** Only half of the subjects have performed the activities at both sides using the plug-ins in Eclipse. Among them, only one (25%) had difficulties to use CTP. In general, all the subjects considered PUG easier and more objective to use than CTP. These results were expected because PUG has mainly two functionalities (capture information from code snippets and decorate functionalities with input groups). In contrast, CTP has more functionalities and combines different techniques to support consumers (automatic, computer-assisted and manual). One subject suggested that CTP should provide a matrix similar to a traceability matrix to mark interactions of reuse points and functionalities of the candidate component. In fact, the subjects who have used templates have received a matrix to fill (Appendix C). This matrix feature can be considered to be implemented to CTP in the future. In addition, one subject has mentioned that pattern verification was unnecessary because the tree view with usage assumptions of the functionalities of the candidate component was always visible at one side of Eclipse.

**Correlational Analysis.** Even with a small number of subjects, it can be valid to make some correlations based on the profile of the subjects and the results obtained. For instance, it was noted a tendency that subjects with more years since graduation (the subjects varied from 0.5 years until 19 years) wrote less feedback on questionnaire QT2 than those subjects that have concluded their course more recently. Another tendency related to “Difficulty to Understand the Approach” metric is that subjects who have more participation in industrial projects have reported less difficulties than subjects with less experience. This suggests that the proposed approach is easily understood by experienced users.

**5.3. Lessons Learned**

After concluding the experimental study, we identified some aspects that should be considered in order to repeat the experiment, since they were seen as limitations of the first execution.
Skills of the Subjects. Testing is a difficult area to perform experiments because the efficiency of testing is not related to the number of test cases created, but the number of errors discovered or the coverage of the test cases created. Thus, it is difficult to state that, if the experiment were re-executed only with high-skilled testers, the results of the baseline and the proposed approach would diverge, because high-skilled testers are so adapted to the pitfalls of testing that would unconsciously do more than is required by the proposed approach. However, if particular subjects are identified with low experience on software testing, more training only to those subjects should be conducted.

The use of a Baseline. Analyze and compare the proposed approach to a baseline was valid to have reference values. Thus, for future executions is important to obtain real data related to the effort of an organization to test candidate components to recalibrate, for instance, the metric of difficulties to understand the approach. However, the results of this experiment can be used in future experiments in the same context as reference values.

Boredom. Although the scope of this experiment was restricted, some subjects spent two hours or more performing activities at producer or at consumer sides. Those long hours may have bored some subjects. However, if the scope of the experiment were reduced it would be difficult to capture metrics of coverage, for instance. For future executions, an analysis should be conducted to balance the effort of the subjects and the data they provide.

Questionnaires. The questionnaires used in the experiment can be improved, QT2 for instance, presented to subjects objective questions to be filled out with “x” and space for further comments. This questionnaire was good to collect the general opinion of the subjects, however, only 43.75% of the spaces reserved for further comments were filled out by subjects. Thus, it can be concluded that questionnaires with objective questions and optional subjective questions associated reduces the detailed feedback that subjects could provide.

5.4. Chapter Summary

This chapter presented the definition, planning, operation, analysis, and interpretation of an experimental study to evaluate the proposed approach to
support component testing, as well as the tool support that covers its activities. The study, summarized in Table 5.7, analyzed, from the perspective of researches and testers, the possible impact in an organization considering adopting the proposed approach to prepare their components to be tested, and also to understand and test candidate components.

Table 5.7 – Hypotheses Analysis

<table>
<thead>
<tr>
<th>Null Hypothesis</th>
<th>Result</th>
<th>Rejection</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H01$ $\mu_{RTC_{Baseline}} &gt; \mu_{RTC_{Approach (Template)}}$</td>
<td>69.75% &gt; 77.20%</td>
<td>Yes</td>
</tr>
<tr>
<td>$H02$ $\mu_{RTC_{Baseline}} &gt; \mu_{RTC_{Approach (CTP)}}$</td>
<td>70.50% &gt; 75.76%</td>
<td>Yes</td>
</tr>
<tr>
<td>$H02$ $\mu_{DCP} \geq 30%$</td>
<td>50% &gt; 30%</td>
<td>No</td>
</tr>
<tr>
<td>$H03$ $\mu_{DCC} \geq 30%$</td>
<td>75% &gt; 30%</td>
<td>No</td>
</tr>
<tr>
<td>$H04$ $\mu_{EV_{producer side}} \geq 1$</td>
<td>-0.86 &gt; 1</td>
<td>Yes</td>
</tr>
<tr>
<td>$H05$ $\mu_{EV_{consumer side}} \geq 1$</td>
<td>-0.88 &gt; 1</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Even with the reduced number of subjects participating to the experiment, and the identified validity threats, based on the metrics analysis and the answers of the subjects, the results suggest that the proposed approach for component testing may be viable after some calibrations. In addition, the subjects evaluated the Eclipse plug-ins (PUG and CTP) as useful and suggested improvements. Nevertheless, it is important to state that this experiment was performed in a particular context. To analyze possible benefits and trade-offs of the proposed approach, more experiments should be conducted in the future.

The next chapter will present the conclusions of this work, its main contributions and directions for future works.
6 Conclusions

Currently, many organizations strive in their attempts to reuse components in Component-based Software Development (CBSD). However, practice demonstrates that reusing software components without a previous validation before they are deployed into the system under development can have catastrophic results (Jezequel et al., 1997). Thus, the assessment and evaluation of candidate components has become an essential activity to assure the quality of component-based systems.

In this sense, to support the evaluation of software components, guidelines to support component testing are necessary. In order to solve the identified problems of component quality assurance, this work presented a component testing approach with tools to cover its activities. The approach and tools was based on an extensive review of current literature related to component quality and component testing.

6.1. Research Contributions

The main contributions of this work can be summarized in the following aspects: i. an analysis of component quality area and the support to its assurance through component testing; ii. the realization of a survey on component testing; iii. the definition of an approach with guidelines to aid stakeholders to conduct component testing activities, as well as the construction of two Eclipse plug-ins to support component producers and component consumers to accomplish the defined guidelines; and iv. an experimental study which evaluated the effects of the defined guidelines and the plug-ins to support component testing when applied to the context of component-based development. Next, a detailed description of the contributions is presented.
• **Component Quality and Testing – An Overview.** The goal was to analyze the application of concepts of software quality assurance, and more specifically, software testing, to support quality assurance of software components. International standards and quality models were discussed as well as some myths about component testing.

• **A Survey on Component Testing.** The objective of the survey was: to identify issues and best practices of current approaches addressing component testing and also to analyze existent tools to support component testing. Through this study, eight approaches and four tools were analyzed offering conditions to define a set of seven necessities to achieve effective component testing.

• **A Component Testing Approach Supported by a CASE Tool.** After concluding the study, we defined guidelines to aid stakeholders to conduct component testing. The guidelines were consolidated as activities to be performed by component producers to prepare their components to be tested by third party, and as activities to be performed by component consumers to understand and test candidate components. In addition, to support the accomplishment of the defined activities at both sides, two tools, PUG (Program Usage Generator) and CTP (Component Test Perspective), were implemented as Eclipse plug-ins.

• **An Experimental Study.** In order to evaluate the viability of applying the proposed component testing approach, as well as its tool support, an experimental study was performed. The study analyzed the proposed approach quantitatively as well as qualitatively. It presented indications that, even with the reduced number of subjects for the experiment, the approach and its tool support are viable.

### 6.2. Related Work

In the literature, some related work could be identified during this research. Chapter 3 presented eight approaches and four tools related to component testing that share similarities with this work. However, there are key differences between this work and the others. Initially, this work defines
guidelines that encompass activities at both component producer and consumer sides, whereas other approaches are focused on specific issues of component testing. Thus, our approach can be considered an integrated approach, because it provides means to reduce the main issue of component testing, which is the lack of information between component producers and component consumers (Beydeda et al., 2003b).

The strategy used by PUG to capture usage information was based on the Usage Discovery Algorithm presented by (Liu et al., 1999). In addition, the process of capturing the correct sequence of method calls that represents a functionality of a component shares some similarities with the work of Rajeev et al. (Rajeev et al., 2005) to extract what is called dynamic interface of a JAVA class. That work is related to software verification and formal methods and uses finite state machines to represent the interface synthesis of a class. One difference is that our approach represents the information to be provided to consumers in XML format.

The concept of reducing the lack of information between component producers and component consumers is also present in the work of Colby et al., 2000 (Colby et al., 2000). That work purposes the use of certifying compilers to process source programs written in JAVA at component producer side generating proof-carrying code (PCC). PCC is conceived to make it possible to the consumer system to determine if a program will behave safely, prior to installing and executing it. A significant difference between that work and our approach is that Colby et al. is focused mainly on automatically generating a single type of information, which is the proof-carrying code, whereas our approach gives producers the chance to add different types of information such as the correct order of method calls, or input data.

Finally, ten main requirements were derived from the seven necessities to achieve effective component testing. The requirements formed a base to the construction of the Eclipse plug-ins (PUG and CTP), which have main advantages of being integrated to development environment aiding users to accomplish the defined guidelines.

Other tools to test components are standalone ones, with limited capacity to elaborate, for instance, integration tests.
6.3. Future Work

Due to time constraints imposed on M.Sc. dissertation, this work had its scope limited. However, some directions for future work can be proposed as an extension of the study of software component testing. Thus, the following issues should be investigated as future work:

**PUG and CTP to other Environments.** Currently, PUG and CTP are supported by Eclipse as plug-ins. However, there are other very popular development environments such as *NetBeans*\(^{25}\) or *IntelliJ*\(^{26}\). Thus, a possible extension of this work can be porting PUG and CTP to be integrated with those other development environments;

**Support to other Programming Languages.** Component market is very heterogeneous, for instance, *ComponentSource* provides components in *.NET, JAVA, Javascript, Flash* and *C++*. However, PUG and CTP are limited to work with JAVA code. Thus, a future work can be to adapt the plug-ins to accept other programming languages than JAVA;

**Test Case Generation.** The usage information captured by PUG at component producer side and provided to consumers can be used to support the generation of integration test cases if consumers provides its required context of use. Thus, test case generation can be another feature to be added to CTP;

**Experimental Studies.** This dissertation presented the definition, planning, operation, analysis, interpretation, presentation and packaging of an experimental study. However, new studies, especially with more subjects and in different contexts can be very useful to calibrate the plug-ins as well as the guidelines behind them; and

**Approach Improvements.** Based on the feedback of the subjects of the experiment some improvements can be performed in the approach. For instance, as suggested by a subject, a matrix similar to a requirements traceability matrix can be added to CTP.

---

\(^{25}\) [http://www.netbeans.org/](http://www.netbeans.org/) (last visit: June 1, 2008)

\(^{26}\) [http://www.jetbrains.com/idea/](http://www.jetbrains.com/idea/) (last visit: June 1, 2008)
6.4. Academic Contributions

The knowledge developed during this work resulted in the following publications:


6.5. Concluding Remarks

Software reuse is considered a key aspect for companies interested in improvements related to productivity, quality and cost reductions. Forty years have passed since McIlroy’s visionary work about mass-produced software components (McIlroy, 1968). Currently, software components has emerged as an economic necessity shortening implementation costs and reducing unpredictability associated with developing custom application. Thus, without support to component quality assurance the benefits provided by software reuse can be jeopardized.

In this context, this work presented a component testing approach with guidelines to support component producers to prepare their products to be tested by third party, as well as guidelines to support component consumers to understand and test if candidate components properly fulfill its needs.
Additionally, to accomplish the activities that form the proposed guidelines, tool support is provided by two Eclipse plug-ins (PUG and CTP). The approach and the plug-ins were evaluated quantitatively and qualitatively presenting results that suggest that the proposed approach to support component testing is viable.

This dissertation can be considered a relevant contribution to the area of software reuse and component testing. However, there is more research to be done and the work is not finished yet. In this sense, we discussed new directions related to new features to be added to the proposed approach and the tools covering its activities.
References


(Edwards, 2001) S. H. Edwards, A Framework for Practical, Automated Black-Box Testing of Component-Based Software,


(Larsen et al., 1996) L. Larsen and M. Harrold, Slicing Object-Oriented Software, 18th International Conference on Software Engineering (ICSE), March, Berlin, Germany, 1996, pp. 495-505.


Appendix A. Questionnaires used in the Experimental Study

QT1. Questionnaire to participants of the Experimental Study of Component Testing (to be filled BEFORE the experiment)

Your Name: ____________________________________________

Years since graduation: ___________

1. How many commercial software projects did you participate after graduation?

<table>
<thead>
<tr>
<th>Category</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low Complexity:</td>
<td></td>
</tr>
<tr>
<td>Medium Complexity:</td>
<td></td>
</tr>
<tr>
<td>High Complexity:</td>
<td></td>
</tr>
</tbody>
</table>

2. About your personal experience in software testing (mark x):

<table>
<thead>
<tr>
<th>Area</th>
<th>None</th>
<th>Low</th>
<th>Medium</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td>Academic</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Commercial</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3. About your personal experience in software reuse (mark x):

<table>
<thead>
<tr>
<th>Area</th>
<th>None</th>
<th>Low</th>
<th>Medium</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td>Academic</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Commercial</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
QT2. Questionnaire to participants of the Experimental Study of Component Testing (to be filled AFTER the experiment)

Your Name: _______________________________________

1. With respect to the objective of the proposed approach to support component testing (mark x and/or write):

<table>
<thead>
<tr>
<th>I did not understood its applicability in practice</th>
</tr>
</thead>
<tbody>
<tr>
<td>I understood partially its applicability in practice</td>
</tr>
<tr>
<td>I understood its applicability</td>
</tr>
</tbody>
</table>

Further Comments: _______________________________________

2. With respect to the training to perform the experiment (mark x and/or write):

<table>
<thead>
<tr>
<th>The training was insufficient</th>
</tr>
</thead>
<tbody>
<tr>
<td>The training was partially sufficient</td>
</tr>
<tr>
<td>The training was sufficient</td>
</tr>
</tbody>
</table>

Further Comments: _______________________________________
3. With respect to the proposed approach at component producer side (mark x and/or write):

<table>
<thead>
<tr>
<th>The effort of component producers is increased</th>
</tr>
</thead>
<tbody>
<tr>
<td>The effort of component producers is not so increased</td>
</tr>
<tr>
<td>The additional effort of component producers is irrelevant</td>
</tr>
</tbody>
</table>

Further Comments:


4. With respect to the proposed approach at component consumer side (mark x and/or write):

<table>
<thead>
<tr>
<th>The effort of component consumers is increased</th>
</tr>
</thead>
<tbody>
<tr>
<td>The effort of component consumers is not so increased</td>
</tr>
<tr>
<td>The additional effort of component consumers is irrelevant</td>
</tr>
</tbody>
</table>

Further Comments:


5. With respect to the quality/coverage of the test cases generated by component consumers using the proposed approach, compared to the baseline (mark x and/or write):

<table>
<thead>
<tr>
<th>The approach generates the same test cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>The approach generates better test cases</td>
</tr>
<tr>
<td>The approach generates test cases with less quality</td>
</tr>
</tbody>
</table>

Further Comments:

6. With respect to the Eclipse Plug-ins PUG and CTP (mark x and/or write):

<table>
<thead>
<tr>
<th>The plug-ins seems to contribute to the execution of the proposed approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>The plug-ins seems to make the execution of the proposed approach difficult</td>
</tr>
<tr>
<td>The plug-ins neither contribute nor make difficult the execution of the proposed approach</td>
</tr>
</tbody>
</table>

Further Comments:
7. Did you have any difficulties to perform the workflow of activities at component producer side?

__________________________________________________

__________________________________________________

__________________________________________________

8. Did you have any difficulties to perform the workflow of activities at component consumer side?

__________________________________________________

__________________________________________________

__________________________________________________

9. Describe Positive Points of the proposed approach to support component testing.

__________________________________________________

__________________________________________________

10. Describe Negative Points of the proposed approach to support component testing.

__________________________________________________
Appendix B. Textual Templates (Component Producer)

The first template is a document called TestDoc. Component producers executing the proposed approach using templates should fill this template with information to support third party to test their components. This document has similar types of information that PUG tool collects from source code.

This is the front page of the TestDoc document; it contains general information about the component that it describes.

TestDoc Version 0.8

Component Name:

General Description:

Main Class:

(end of first page)
All the functionalities described by *TestDoc* should have a page similar to this one. It describes **Functionality Name**, a textual description of the functionality and usage assumptions (**Usage Policy** area) such as correct order of method calls to accomplish the functionality being described. On the right side, there are **Input Groups** where variables that alter the state of the Usage Policy are described.

<table>
<thead>
<tr>
<th>Functionality Name:</th>
<th>Description:</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Usage Policy</th>
<th>Input Groups</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Variable</td>
<td>Type</td>
</tr>
<tr>
<td>Value/ Description</td>
<td>Expected Result</td>
</tr>
</tbody>
</table>

| 2            | 2            |
| Variable     | Type         |
| Value/ Description | Expected Result |

| 3            | 3            |
| Variable     | Type         |
| Value/ Description | Expected Result |
## Appendix C. Textual Templates (Component Consumer)

This section presents a document to be filled by component consumers to understand and test a candidate component. The document is called *Interaction Matrix* and has some features that CTP tool provide.

This is the front page of the Interaction Matrix document; it contains general information about the system under development and its associations with candidate components.

<table>
<thead>
<tr>
<th>Interaction Matrix</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Version 0.4</td>
<td></td>
</tr>
</tbody>
</table>

### System Name:

### General Description:

### Main Class:
This is the **Interaction Matrix** that maps associations of the system under development to candidate components. Candidate components are added at vertical column on the left side whereas the functionalities of the system are added at top horizontal. The mapping can be of tree forms: (a) If consumers identify an interaction of one functionality of their system and the candidate component, they mark an “I”; (b) If consumers identify that they need to check the basic behavior of a functionality of a component, they mark a “B”; and (c) if consumers identify an interaction and the necessity to check the basic behavior of a functionality, they mark “I/B”.

<table>
<thead>
<tr>
<th>Candidate Components</th>
<th>System Under Development</th>
<th>Package 1</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Component 1</td>
<td>func0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>func1</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>func2</td>
<td></td>
<td>I</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>func3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>func4</td>
<td></td>
<td>I/B</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>func5</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The last page of this template is a test case template where consumers, based on the TestDoc they have received, additional information from other sources such as JAVADOC or user manuals, and the identified interactions of *Interaction Matrix*, can construct test cases to validate the behavior of the candidate component. This page can be considered optional because consumers can alternatively construct test cases using JUnit test cases or use other type of tool such as Rational Test Manager\textsuperscript{27}.  

<table>
<thead>
<tr>
<th>Test</th>
<th>(number)</th>
<th>(comment)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pre-Condition</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expected Result</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\textsuperscript{27} http://www-306.ibm.com/software/awdtools/test/manager/ (last visit: June 1, 2008)
Dissertação de Mestrado apresentada por Fernando Raposo da Câmara Filho à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título “Paralelizando Programas Java Usando Leis de Transformação”, orientada pelo Prof. Silvio Romero de Lemos Meira e aprovada pela Banca Examinadora formada pelos professores:

Prof. Silvio Romero de Lemos Meira
Centro de Informática / UFPE

Prof. Uirá Kulesza
Deptº de Informática e Matemática Aplicada / UFRN

Prof. Marcelo Bezerra D’Amorim
Centro de Informática / UFPE

Visto e permitida a impressão.
Recife, 22 de agosto de 2008.

Prof. FRANCISCO DE ASSIS TENÓRIO DE CARVALHO
Coordenador da Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco.

EM TEMPO: Na primeira linha onde se lê Fernando Raposo da Câmara Filho, leia-se Fernando Raposo da Câmara Silva e na terceira linha onde se lê “Paralelizando Programas Java Usando Leis de Transformação”, leia-se “A Component Testing Approach Supported by a CASE Tool”