Component-Based Software: An Overview of Testing

Auri Marcelo Rizzo Vincenzi[1], José Carlos Maldonado[1], Márcio Eduardo Delamaro[2], Edmundo Sérgio Spoto[2], and W. Eric Wong[3]

[1] Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo São Carlos, São Paulo, Brazil
{auri,jcmaldon}@icmc.usp.br
[2] Faculdade de Informática, Fundao Eurípedes Soares da Rocha Marília, São Paulo, Brazil
{delamaro,dino}@fundanet.br
[3] Department of Computer Science, University of Texas at Dallas Richardson, Texas, USA
ewong@utdallas.edu

Abstract. Component-based development makes heavy use of Object Oriented features which have motivated a major re-evaluation of software testing strategies. This chapter introduces the basic concepts of software testing focusing on the state-of-the-art and on the state-of-the-practice of this relevant area in the context of component-based software development.

1 Introduction

Software testing is a crucial activity in the software development process. Its main goal is to reveal the existence of faults in the product under testing: a program unit or a component. It is known as one of the most expensive activities in software development that can take up to 50% of the total cost in software development projects [1]. Besides its main objective—to reveal faults—the data collected during the testing phases are also important for debugging, maintenance, and reliability assessment.

Before software testing starts it is necessary to identify the characteristics of the input and output data required to make the component run (for instance, type, format and valid domain of that data), as well as how the components behave, communicate, interact and coordinate with each other. The Testing Plan is a document in a Test Design Specification that defines the inputs and expected outputs and related information like expected execution time for each run, format of the output and testing environment. Any anomaly found during testing should be reported and documented in a Test Incident Report.

In order to perform software testing in a systematic way and on a sound theoretical basis, testing techniques and criteria have been investigated. Testing techniques are classified as functional, structural, fault based, and state based, depending on the source used to determine the required elements to be exercised (the test requirements). Testing techniques and criteria are the mechanisms available to assess testing quality.

A. Cechich et al. (Eds.): Component-Based Software Quality, LNCS 2693, pp. 99–127, 2003.

Concerning the functional (black box) technique, testing requirements are obtained from the software specification. The structural technique uses implementation features to obtain such requirements, and the fault-based technique uses information about common errors that can occur during the development. In the state-based technique the requirements are obtained from a state-based specification like a Finite State Machine [2] or a Statechart [3]. These testing techniques are complementary and the question to be answered is not "Which one to use?" but "How to use them in a coordinated way, taking advantage of each one?"

According to Freedman [4], the research in software testing has been concentrated on two main problems:

- Test effectiveness: What is the best selection of test data?
- Test adequacy: How do we know that sufficient testing was performed?

Each of the above mentioned techniques has several criteria that define specific requirements that should be satisfied by the test data. In this way, requirements determined by a testing criterion can be used either for test data evaluation or test data generation. Since exhaustive testing, i.e., executing the product under test with its entire input domain, is not possible in general, test effectiveness is related with to task of creating the smallest test set for which the output indicates the largest set of faults. Testing adequacy is related to the determination of the effectiveness of a test criterion [4].

The object oriented (OO) paradigm has been extensively used, in particular because of its potential to promote component reuse. In general, component development uses much of the OO terminology. It is possible, though, to have components implemented according to the procedural paradigm, for instance, as COBOL procedures or C libraries, although, currently, most parts of software components are developed by using OO methodologies and languages.

As defined in [5], a 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. Reusable components exist in several different types. In general, a software component can be as simple as an individual class or module, or as sophisticated as a JavaBeans, Enterprise JavaBeans [6] or COM objects [7]. It can be observed that software components inherit many characteristics of the OO paradigm, but the notion of components transcends the notion of objects. Reuse in OO in general means reuse of class library, in a specific programming language. Components can be reused without knowledge of the programming language or environment in which they have been developed [8, 9].

Encapsulation, inheritance, polymorphism, and dynamic binding bring benefits to software design and coding but, on the other hand, introduce new challenges to software testing and maintenance [10]. Software testing should incorporate the intrinsic characteristics of OO and component-based development. Considering specifically the component-based approach, two points of view related to testing can be observed: the provider's and the user's point of view.

Component providers test their components without knowledge about the context of their applications. Component users in their turn test their applications possibly without access to the component source code or data about the testing performed on them by the providers.

In the last decade, several tools and criteria have been developed for program testing [11, 12]. In the 90's researchers also started to concentrate on investigating approaches to test OO programs and components [13, 14]. With the adoption of component technology, component users observed that the quality of their component-based systems was dependent on the quality of the components they used and on the effectiveness of the testing activity performed by the component providers.

Software components may range from in-house components to Commercial Off-The-Shelf (COTS) components. The former allow clients to have complete access and control, including the development process. For the latter, on the other hand, knowledge about the development or testing processes may not be accessible. In this chapter we consider components implemented by using the OO approach. This implies that, from the provider point of view, the problems related to component testing are similar to those for OO program testing, in addition to the intrinsic problems related to the component technology. Therefore, since the component provider has access to the component source code, functional, structural, fault-based and/or state-based testing technique can be used. From the client point of view, when the source code of the component is not available, only functional and/or state-based testing techniques can be applied.

As highlighted by Weyuker [13], it is necessary to develop new methods to test and maintain components to make them reliable and reusable in a large range of software projects, products and software environments. Harrold [1] restates that the test of component-based systems, the development of effective testing processes and the demonstration of the effectiveness of testing criteria constitute the main directions to the software testing area.

In this chapter we present an overview on testing of OO software and components. In this section the basic concepts and terminology related to component testing were presented. In Section 2 the testing phases are discussed, comparing procedural and OO/component approaches. In Section 3 a classification of testing techniques is presented, as well as the definition of some testing criteria representative of each technique. Section 4 discusses more questions related to the testing of OO programs and components. In Section 5 some testing criteria specifically for OO/component testing are described. Also in Section 5 the problems to component testing automation are discussed and some existent testing tools are described. Section 6 presents an example about how to use one of the testing criteria described earlier. In Section 7 the conclusions are presented.

2 Testing Phases

As the development process is divided into several phases, allowing the system engineer to implement its solution step by step, the testing activity is also divided into distinct phases. The tester can concentrate on different aspects of

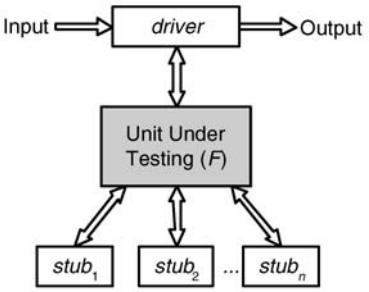

Fig. 1. Required environment for unit testing

the software and use different testing criteria in each one [15]. In the context of procedural software the testing activity can be divided into three incremental phases: unit, integration and system testing [16]. Variations in this pattern are identified for OO and component-based software, as discussed later.

Unit testing focuses on each unit, to ensure that the algorithmic aspects of each of them are correctly implemented. The goal is to identify faults related to logic and implementation in each unit. In this phase, structural testing is widely used, requiring the execution of specific elements of the control structure in each unit. Mutation testing is also an alternative to unit testing. In this phase it is common to need to develop drivers and stubs (Fig. 1). Considering F the unit to be tested, the driver is responsible for coordinating the testing of F. It gathers the data provided by the tester, passes them to F in the form of arguments, collects the results produced by F and shows them to the tester. A stub is a unit that replaces, during unit testing, another unit used (called) by F. Usually, a stub is a unit that simulates the behavior of the used unit with a minimum of computation effort or data manipulation. The development of drivers and stubs may represent a high overhead to unit testing.

After each unit has been tested, the integration phase begins and in consequence, the integration testing. But why shouldn't a program that was built from tested units that individually work as specified function adequately? The answer is that unit testing presents limitations and cannot ensure that each unit works in every single possible situation. For example, a unit may suffer from the adverse influence of another unit. Sub-functions when combined may produce unexpected results and global data structures may present problems.

After being integrated, the software works as a whole and should be submitted to system testing. The goal is to ensure that the software and the other elements that are part of the system (hardware and database, for instance) are adequately combined and the function and performance are obtained. Functional testing has been most used in this phase [16].

Fig. 2 illustrates the three phases as mentioned above, as well as the elements used in each of them, either for procedural or OO programs. According to the IEEE 610.12-1990 (R2002) [17] standard, a unit is a software component that cannot be divided. In this way, considering that testing is a dynamic activity, a

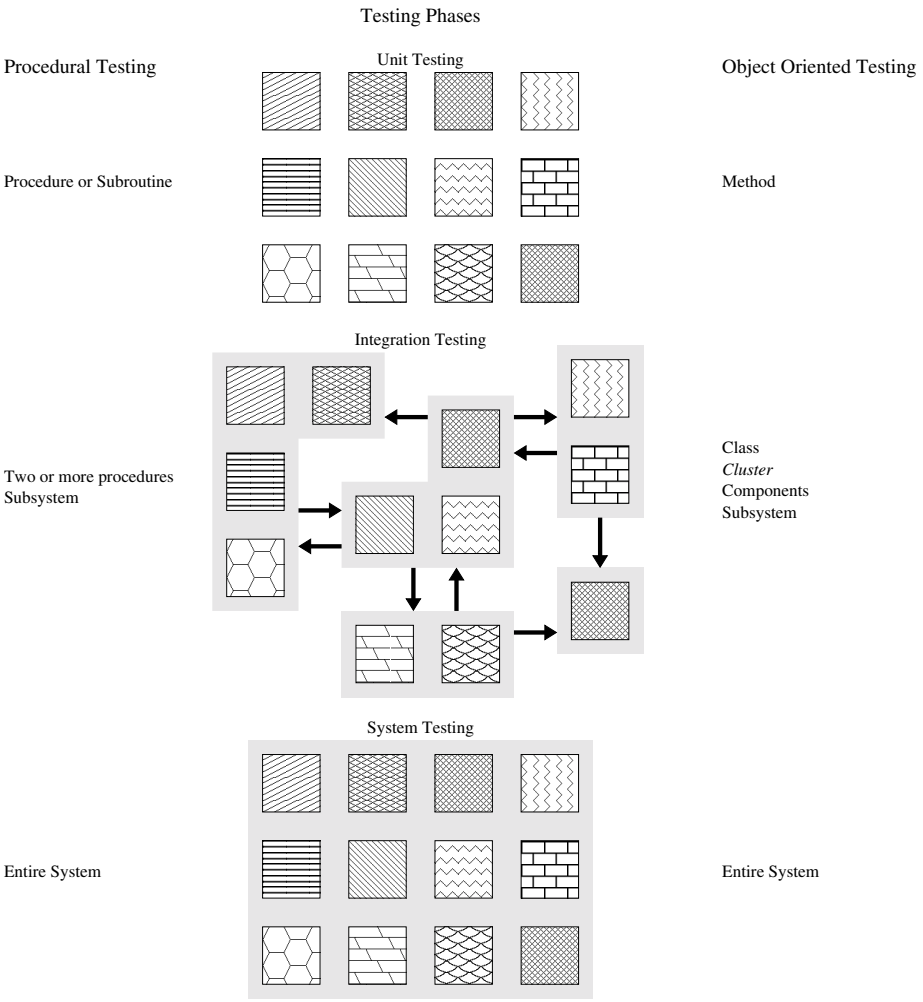

Fig. 2. Relationship between unit, integration and system testing: procedural and OO programs

unit in procedural programs is a subroutine or procedure. In OO programs, the smallest part to be tested is a method. The class to which the method belongs is seen as the driver of that method because without the class it is not possible to execute the method. In the procedural paradigm, unit testing is also called intra-procedural, and in the OO it is called intra-method [18]. By definition, a class gathers a set of attributes and methods. In this way, taking in consideration a single class it is possible to think about integration testing. Methods of the same class may interact to implement certain functionality—what characterizes the kind of integration that should be tested. This is inter-method [18] testing. In the procedural paradigm this phase is also called inter-procedural.

Table 1. Relationship between procedural and OO testing phases

Smallest Unit: Method		
Phase	**Procedural Testing**	**Object Oriented Testing**
Unit	Intra-procedural	Intra-method
Integration	Inter-procedural	Inter-method, Intra-class and Inter-class
System	Entire System	Entire System
Smallest Unit: Class		
Phase	**Procedural Testing**	**Object Oriented Testing**
Unit	Intra-procedural	Intra-method, Inter-method and Intra-class
Integration	Inter-procedural	Inter-class
System	Entire System	Entire System

Harrold and Rothermel [18] define two other types of OO testing: intra-class and inter-class. The former is used to test public method interactions through different sequences of calls to such methods. The goal is to identify possible sequences that lead the object to an invalid state. According to the authors, since the user may invoke the public method in many different orders, the intra-class testing gives one the confidence that different sequences of calls act correctly. In the inter-class testing the same concept is applied to public methods but not only those in a single class, i.e., the test requires calls among methods in different classes. After integration testing, system testing may commence. Since system testing is generally based on functional criteria, there is no fundamental difference in this phase from procedural and OO software.

A few variations regarding the testing phases for OO software are identified in the literature. Some authors understand that the smallest unit of an OO program is a class [19, 20, 10, 14]. In this way, a unit test would be composed of intra-method, inter-method and intra-class testing, and integration testing would be the same as inter-class.

Table 1 summarizes the types of testing that may be applied in each phase, either for procedural or OO software, taking either a method or a class as the smallest unit.

3 Techniques and Criteria for Software Testing

According to Howden [21], testing may be classified as specification-based or program-based testing. Such classification suggests that functional and state-based techniques are specification-based, and structural and fault-based are program-based.

In the specification-based testing the objective is to reveal faults related to the external functionality, to the communication interfaces between modules, to the required constraints (pre- and post-conditions) and to the program behavior. The problem is that often the existing specification is not formal, which makes harder the creation of test sets able to systematically exercise the component [19]. However, the specification-based criteria can be used in any context (procedural or OO) and in any testing phase without any fundamental adaptation.

Program-based testing, on the other hand, requires code handling and selection of test sets that exercise specific pieces of the code, not its specification [19]. The goal is to identify faults in the internal structure and in the component behavior. The disadvantage is that this approach may be dependent on several factors like target programming language and the need to have access to the source code. In the case of COTS, for instance, this is not always possible.

3.1 Functional Testing

Functional or black box testing (specification-based) has this name because the software is handled as a box from which the content is not known. Only the external side is visible. In this way the tester uses basically the specification to obtain the testing requirements or the test data, without any concern about the implementation [11]. A high-quality specification that matches the client's requirements is fundamental to support the application of functional criteria. Examples of such criteria are [16]: 1) Equivalency Partition, 2) Boundary Value, 3) Cause-Effect Graph, and 4) Category-Partition Method [22].

Statistical software testing can also be considered as functional testing since it is also based on the product specification. Example of such criteria are presented in [23, 24, 25, 26]. The idea behind these criteria is to exercise a program with inputs that are randomly generated according to a given distribution over the input domain, the key of its effectiveness being the derivation of a distribution that is appropriate to enhance the program failure probability. Basically, as defined in [23], the statistical test sets are defined by two parameters: (1) the test profile, or input distribution, from which the inputs are randomly drawn; and (2) the test size, or equivalently the number of inputs (i.e. of program executions) that are generated.

The use of the functional technique may make it difficult to quantify the testing activity. That is because it is not possible to ensure that certain essential parts of the product's implementation have been exercised. Another problem is that (non formal) specifications may be incomplete and so will be the test set created based on them.

On the other hand, since functional criteria are based solely in the specification, they can be used to test procedural, OO programs and software components as well [27, 28, 22, 29, 30, 10, 31].

3.2 Structural Testing

The structural testing technique, also known as white box (in opposition to "black box") fits in the class of program-based testing, since it takes into consideration implementation aspects to determine the testing requirements. Most of the criteria of the structural technique use a Control Flow Graph—CFG (also called Program Graph) to represent the program under test. A program P represented by a CFG has a correspondence between the nodes of the graph and blocks of code and between the edges of the graph and possible control-flow transfer between two blocks of code. From the CFG it is possible to select elements to be exercised during testing, characterizing the structural testing.

Structural testing faces several constraints and disadvantages as the need to determine unfeasible testing requirements such as unfeasible paths and associations [21, 32, 33, 34]. These constraints pose serious problems to testing automation. Nevertheless, this technique is seen as complementary to functional testing [16] and information obtained with its application is also relevant to maintenance, debugging and software reliability estimation [35, 36, 16, 37, 38, 1, 39].

The first structural criteria were based exclusively on control-flow structures. The most known are all-nodes, all-edges, and all-paths [40]. In the middle 70's appeared the dataflow-based criteria [41], that require interactions amongst variable definitions and variable uses to be exercised [41, 42, 43, 44]. The reasoning behind such an approach is the indication that even for small programs, control-flow-based testing is not effective for revealing even trivial faults. The use of data-flow criteria provides a hierarchy of criteria from all-edges to all-paths, trying to make the testing a more rigorous activity. Amongst the best known data-flow criteria are those introduced by Rapps and Weyuker in the middle 80's, for instance, all-defs, all-uses, all-du-paths and all-p-uses [34].

At the beginning of the 90's Maldonado [45] presented a family of criteria named Potential-Uses and the corresponding feasible criteria, obtained with the elimination of unfeasible associations. These criteria are based on associations between a variable definition and the possible points in the program where it can be used, not necessarily requiring the actual use of it.

Several extensions of data-flow criteria can be found in the literature, either for integration testing of procedural programs [46, 47, 48] or for unit and integration of OO programs [46].

3.3 Fault-Based Testing

The fault-based technique uses information about faults frequently found in software developments and also about specific types of faults that one may want to uncover [49]. Two criteria that typically concentrate on faults are Error Seeding and Mutation Testing.

Error seeding introduces in the program under test a known number of artificial faults before it is tested. After the test, from the total number of faults found and the rate between natural / artificial faults found, it is possible to estimate the number of remaining natural faults. The problems with this approach are 1) artificial faults may hide natural faults; 2) in order to obtain a statically reliable result it is necessary to use programs that can have 10,000 faults or more; and 3) it is based on the assumption that faults are uniformly distributed in the program, which in general is not the case—real programs present long pieces of simple code with few faults and small pieces with high complexity and high concentration of faults [50].

Mutation testing appeared in the 70's at Yale University and the Georgia Institute of Technology. It was strongly influenced by a classical method for digital circuit testing known as "single fault test model" [51]. One of the first papers describing mutation testing was published in 1978 [49]. This criterion uses a set of products slightly different from the product P under test, called

mutants, to evaluate the adequacy of a test set T. The goal is to find a set of test cases able to reveal the differences between P and its mutants, making them to behave differently [49]. When a mutant has a behavior diverse from P it is said to be "dead"; otherwise it is a "live" mutant. A live mutant must be analyzed to check whether it is equivalent to P or it can be killed by a new test case, promoting in this way the improvement of T.

Mutants are created based on mutant operators, which are rules that define the (syntactic) changes to be done in P to create the mutants. It is known that one of the problems with mutation testing is related to the high cost to execute a large number of mutants. Besides, there is also the problem of deciding mutant equivalence, which in the general case is undecidable. Extensions of this criterion have also been proposed for integration testing as well as for program specifications. Delamaro *et al.* [52] defined the Interface Mutation criterion that applies the mutation concept to the integration testing phase. With that criterion a new set of mutant operators that model integration errors was proposed.

In the context of test specifications, mutation can be used to test Petri Nets [53, 54], Statecharts [55, 56], Finite state machines [57, 58] and Estelle [59, 60].

Recently, researchers have also been investigating the use of mutation testing for the OO paradigm. Kim *et al.* [61] proposed the use of a technique called Hazard and Operability Studies (HAZOP) to determine a set of mutant operators for Java. In general, this approach does not significantly differ from the traditional mutation with respect to mutant operators' creation, but introduces a more rigorous and disciplined way to do it. The technique identifies in the target language grammar those point candidates for mutation and then the mutant operators are created based on some predefined "guide words." A more extensive set of mutant operators for Java (inter-class mutant operators), that includes the ones proposed in [61], is defined in [62].

Researchers have also explored mutation testing in the context of distributed components communicating through CORBA [63, 64]. Yet, Delamaro *et al.* [65] define a set of mutant operators to deal with concurrency aspects of Java programs.

3.4 State-Based Testing

State-based testing uses a state-based representation of the unit or component under test. Based on this model, criteria to generate test sequences are used to ensure its correct behavior. One of these criteria, based on Finite State Machines (FSM), is the criterion W [2]. Other similar criteria can be found in the literature, as DS [66], UIO [67] and WP [68]. As mentioned before, mutation testing has also been used in test case generation for FSM [57, 58].

Criteria based on FSM are also widely used in OO context to represent the behavioral aspect of objects [27, 28, 29, 30, 10, 31, 14] and in the context of software components [69] since they only require a state-based representation to be applied. As can be observed, there are a large number of criteria available to evaluate a test set for a given program against a given specification. One

important point to be highlighted is that the testing criteria and techniques are complementary and a tester may use one or more of these criteria to assess the quality of a test set for a program and enhance the test set, if it is the case, by constructing additional test cases needed to fulfil the testing requirements.

4 Issues Related to Component Testing

Component testing and component-based system testing face a series of particular issues, as will be discussed in this section. According to Harrold *et al.* [70], it is possible to analyze the problem from two points of view: the component user and the component provider.

4.1 Perspectives of Component-Based Testing

User's Perspective

Component users are those that develop systems by integrating third-part components. To help them, there are several initiatives to adapt traditional analysis and testing techniques to be used in the component development. However, there are issues that make this task difficult. First, the code of the components is not always accessible. Techniques and criteria based on the program implementation as data-flow-based criteria and mutation testing need the source code to determine their testing requirements. When the component source is not available to the user, such criteria cannot be used or at least an alternative setting between the parts is needed. Second, in component-based systems, even if the code is available, the components and the component-based system may have been developed in different languages and a tool to analyze/test the entire system may fail to analyze the components.

Third, a software component frequently offers more functionality than the client application needs. In this way, without the identification of the piece of the functionality that is actually required by the user, a testing tool will provide useless reports. For example, structural criteria evaluate how much a test set covers the required elements, but in a component-based system the unused part of the components should be excluded from this evaluation, otherwise the coverage assessed by the tool would be low even if the test set was a good one for the used portion of the code [71].

Developer's Perspective

The component provider implements and tests the component independently of what kind of application will use it. Unlike the user, the provider has access to the source code. Thus, testing the component is the same as traditional unit/integration testing. However, traditional criteria like control-flow-based criteria may not be enough to test the components due to its inefficiency to reveal faults [72]. Correcting a fault in a component after it is released has a high cost, many times higher than if the fault had been found during integration testing

in a non-component based system because the component will probably be used in many applications.

The provider has to have mechanisms to solve two problems: first, the provider must effectively test the components as independent software units. Doing so, the provider increases the user's confidence in the component quality and reduces the testing cost for the user. Rosenblum [71] describes an approach for component unit testing that depends on the application context and so is more relevant to the user than to the provider. The second approach, proposed by Harrold *et al.* [70], separates the analysis and testing of the user application from the analysis and testing of the components.

A fundamental aspect for component or application testing is their testability. According to Freedman [4], the component's testability usually has two aspects: observability and controllability. The latter shows how easy it is to control the component in relation to its input, operation, output and behavior. The former indicates how easy it is to observe the program behavior according to its operational behavior and its output as function of its input.

As mentioned before, the features of OO programs bring a series of obstacles to the testing activity. In the next section the impact of such features in the testability of OO programs and components will be discussed. For further information the reader may refer to [19, 14, 73].

4.2 The Impact of OO on Software Testability

Encapsulation

Encapsulation means a control access mechanism that determines the visibility of methods and attributes in a class. With the access control, undesirable dependencies between client and a server class are avoided, making visible to the client only the class interface, and hiding implementation details. Encapsulation aids in the information hiding and in the design of a modular structure.

Although encapsulation does not directly contribute to the introduction of faults, it may be an obstacle to the testing activity, reducing the controllability and observability. Program-based testing requires a complete report about the concrete and abstract state of an object, as well as the possibility to change that state easily [14]. OO languages make it harder to get or set the state of an object. In the case of C++, for instance, friend functions have been developed to solve this problem. However, in the case of languages that do not provide such mechanism, other solutions must be adopted. Harrold [1] says that the solution would be the implementation of methods `get` and `set` for every attribute in a class. Another alternative would be the use of reflection (although, as highlighted by Rosa and Martins [74] not every language allows the reflection of private methods) or metadata[1] [75, 76].

[1] Metadata are used by the component provider to include additional information about the component without revealing the source code or other sensitive details of the component. In general, as defined in [75], metadata are *data about components* that can be retrieved or calculated by *metamethods*.

Inheritance

Inheritance is essential to OO development and component-based development. It permits the reuse by sharing features present in classes previously defined. However, Binder [14] highlights the fact that inheritance weakens encapsulation and may create a problem similar to the use of global variables in procedural programs. When implementing a class that uses inheritance it is important to know the details about the ancestor classes. Without that, it may happen that the class seems to work but it actually violates implicit constraints of the ancestor classes. Large class chains make it more difficult to understand the program, increase the chance for faults, and reduce testability.

Offutt and Irvine [22] comment that inheritance may lead to a false conclusion that aspects already tested in the super-classes do not need to be retested in the subclasses. Perry and Kaiser [19] state that even if a method is inherited from a super-class, without any modification, it has to be retested in the subclass context.

Harrold *et al.* [77] use the results from Perry and Kaiser [19] and develop an incremental testing strategy based on the class hierarchy. They propose to identify which inherited methods have to be tested with new test cases and which can be retested by using the same test cases used to test the super-class. With this strategy the testing effort can be reduced since many test cases can be reused in the subclasses. In addition, the way inheritance is implemented changes from one language to another and this can also have some influence on the testing strategy.

Multiple Inheritance

Multiple inheritance allows a class to receive characteristics from two or more super-classes that in their turn may have common features (attributes or methods with the same name, for instance). Perry and Kaiser [19] state that although multiple inheritance leads to small syntactic changes in programs, it can lead to high semantic changes, which can make the testing activity for OO programs even more difficult.

Polymorphism

Polymorphism is the ability to refer to different types of objects using the same name or variable. Static polymorphism makes such association at compilation time. For example, generic classes (C++ templates for instance) allow static polymorphism. Dynamic polymorphism allows different types of associations at execution time. Polymorphic methods use dynamic binding to determine at execution time which method should answer to a given message, based on the type of the object and on the arguments sent with the message.

Polymorphism can be used to produce elegant and extensible code but a few drawbacks can be realized in its use. For example, a method x in a super-class needs to be tested. This method is overwritten in a subclass. The correction of

x in the subclass cannot be assessed because the pre- and post-conditions in the subclass may not be the same as in the super-class [14].

Each possible binding in a polymorphic message is a unique computation. The fact that several polymorphic bindings work correctly together does not ensure that all will work correctly. A polymorphic object with dynamic bindings may easily result in sending improper messages to the wrong class, and it can be difficult to have all possible binding combinations.

Dynamic Binding

Dynamic binding allows a message to be sent to a server class that implements that message. Since server classes are frequently developed and reviewed without further concern about the client code, some methods that usually work correctly in a client class may lead to unexpected results. A client class may require a method that is no longer part of a server class, incorrectly use the methods available, or call the methods with wrong arguments.

Besides these problems, Binder [14] reports errors related to the state of the objects and sequences of messages. The packaging of methods in a class is fundamental to OO; as a consequence, messages have to be executed in some sequence, leading to the question: "Which message sequences are valid?"

Objects are created at execution time, taking memory space. Each new configuration this memory space assumes is a new state of such object. Thus, besides the behavior encapsulated by an object it also encapsulates states.

Analyzing how the execution of a method can change the state of an object, four possibilities are observed [10]:

1. It can leave the object in the same state;
2. It can take the object to a new, valid state;
3. It can take the object to an undetermined state; or
4. It can change the object to an inappropriate state;

Possibilities 3 and 4 characterize erroneous states. Possibility 1 characterizes an error if the method is supposed to behave as in possibility 2 and vice-versa.

4.3 Other Issues in Component Testing

Specifically for component testing, one additional issue that has to be considered is the kind of information that has to be included/delivered with or within the component to help, as much as possible, the testing on the user's side. As mentioned by Orso *et al.* [75], the drawbacks of component-based software technologies arise because of the lack of information about externally provided components.

Although existing component standards, including DCOM [7] and Enterprise JavaBeans [6], already provide additional information about a component by the use of metadata that are packaged with the component, it is necessary to define a standard such that, independently of the component provider, the component user can always consider that a given set of information will be available.

Moreover, such a standard makes easier the job of component testing tool developers by providing a common interface to access component features requested to perform the testing activity on a single component or on a component-based system.

Gao [78] defined a maturity model to evaluate the maturity level of a given process in an organization. The maturity model is composed by four levels focusing on the standardization of the component testing, testing criteria, management procedures and measurement activities. Although published almost five years ago, this reference is still up-to-date because not much has been done in this direction. Changes will only occur when the component user demands high-quality or certified components [79].

To achieve either a high maturity level or a certified component, the component provider has to use an incremental testing strategy that combines testing criteria of different testing techniques such as functional, structural and behavioral aspects of the component being tested. The next section presents a summary of the most relevant criteria developed for testing OO programs and software components considering both sides: component providers and component users.

5 Component-Based Testing Criteria

Testing techniques and criteria have been investigated with the aim of establishing a systematic and rigorous way to select sub-domains of the input domain able to reveal the presence of possible faults, respecting the time and cost constraints determined in a software project.

When discussing testing criteria for software components we need to consider the two perspectives of component development, because, in general, structural testing can only be performed by the component provider since such criteria require the availability of the source code. When the source code of the component is not available, the component user has to use functional and/or state-based testing criteria to perform component/system testing. Below we present different testing criteria that can be used by the component provider, component user or both.

According to Binder [80], the biggest challenge of OO testing is to design test sets to exercise combinations of sequences of messages and state interactions that give confidence that the software works properly. In some cases, test sets based on sequences of messages or states are enough. However, Binder warns that the state-based testing is not able to reveal all kinds of faults requiring also the use of program-based criteria [81, 14]. Methods in a class make use of the same instance variables and should cooperate with the correct behavior of the class, considering all the possible activation sequences. The visibility of the instance variables for all the methods in the class creates a fault hazard similar to the use of global variables in the procedural languages. Since the methods in a superclass are not explicit when a subclass is coded, this may lead to an inconsistent use of the instance variables. In order to reveal this kind of fault it is necessary

Table 2. Criteria for testing components and OO programs

Technique	Criteria	Phase†		
		U	I	S
Functional	Category-Partition Method [22]	•	•	•
Structural	Data-flow [18]	•	•	
Structural	FREE [81]	•	•	
Structural	Modal Testing [80]	•	•	
Fault-Based	Class Mutation [61, 82]	•	•	
Fault-Based	Mutation on Distributed Programs(CORBA Interface) [63]		•	
Fault-Bases	Mutation on Concurrent Programs (Java) [65]	•	•	
State-Based	FREE [81]	•	•	•
State-Based	Modal Testing [80]	•	•	•

† Testing Phases: Unit (U), Integration (I) and System (S).

to use control-flow and data-flow criteria that ensure the inter-method (or intra-class) coverage. Table 2 shows some well known criteria to test OO programs identified in the literature, and the respective phases they are applied in.

As can be observed, functional criteria, including statistical software testing [23, 24, 25], can be applied indiscriminately to procedural, OO programs and components [11] since the testing requirements are obtained from the product specification. A case study investigating the *Category-Partition Method* in detecting faults on OO programs is described in [22]. The method provides guidelines to identify, from the product specification, different categories related to the input domain and the specific functionality that should be in the domain of these categories. Statistical software testing, in which inputs are randomly sampled based on a probability distribution representing expected field use, is also based on the specification and can also be used for testing software components [23, 24, 25]. Therefore, as mentioned in Section 3, the major problem with the functional testing technique is that, since it is based on the specification, its criteria can not assure that essential parts of the implementation have been covered.

In the case of structural testing, important work has been done by Harrold and Rothermel [18] who extended data-flow testing to class testing. The authors comment that data-flow criteria, designed for procedural programs, can be applied to OO programs both for a single method test and for interacting methods in the same class [34, 46]. They do not consider tough data-flow interactions when the users of a class make sequences of calls in an arbitrary order. To solve this problem they present an approach to test different types of interactions among methods in the same class. To test the methods that are accessible outside the class a new representation named class control flow graph was developed. From this representation new inter-method and intra-class associations can be derived. An example that illustrates this criterion is presented in the next section.

Considering the fault-based criteria, a point to be highlighted is the flexibility to extend the concepts of mutation testing to different executable entities. The criterion, initially developed for unit testing of procedural programs, has been

extended to OO programs [63, 61, 62], to FSM specifications [57, 58], Petri Nets [53, 54], Statecharts [55, 56] and Estelle [59, 60].

Specifically, for OO programs mutation testing has been used to exercise aspects concerning concurrency, communications and testing of Java programs at unit and integration level: 1) Kim *et al.* [61] used a technique named HAZOP (Hazard and Operability Studies) to define a set of mutant operators for Java programs; 2) Ghosh *et al.* [63] defined a set of mutant operators aiming at testing the communications interfaces between distributed (CORBA) components; and 3) Delamaro *et al.* [65] defined mutant operators specific for concurrent Java programs. Since structural and fault-based criteria require the component source code, alternative criteria, that do not require the source code, have been proposed. Such criteria are based on metadata [76] and metacontent [75], reflection [83], built-in testing [84, 83], polymorphism [85] and state-based testing [69].

Computational reflection enables a program to access its internal structure and behavior and this is beneficial for testing by automating the execution of tests through the creation of instances of classes and the execution of different sequences of methods. In component testing, computational reflection has been used to load a class into a testing tool extracting some of the methods and invoking them with the appropriate parameters. Therefore, reflection can be used to build test cases by extracting the needed characteristics of classes [76]. Rosa and Martins [86] propose the use of reflexive architecture to validate OO applications.

Another solution, like a wrapper, is proposed by Soundarajan and Tyler [85] considering the use of polymorphism. Given the component's formal specification containing the set of pre- and post-conditions that have to be satisfied for each method invocation, polymorphic methods are created such that, before the invocation of the real method (the one implemented by the component) the polymorphic version checks if the pre-conditions are satisfied, collects method invocation information (parameters values, for example), invokes the real method, collects output information (return value, for example) and checks if the posconditions are satisfied. The disadvantage of this approach is that it requires formal specification and does not guarantee the code coverage. Moreover, to be useful, requires the implementation of an automatic wrapper generator since, once the specification changes (any pre- or post-condition), the set of polymorphic class/methods has to be regenerated/reevaluated.

Metadata are also used, in existing component models, to provide generic usage information about a component (e.g. the name of its class, the name of its methods) as well as information for testing. By using metadata, static and dynamic aspects of the component can be accessed by the component user to perform different tasks. The problem with this approach is that there is no consensus about which information should be provided and how it should be provided. Moreover, it demands more work by the component provider.

Other strategies propose an integrated approach for component-based testing generation. They combine black- and white-box information from a formal/semiformal component's specification. The idea is to construct a graphical represen-

tation of the component using control-flow and data-flow information gathered from the specification and, from this graphical representation, control-flow- and data-flow-based criteria can be applied. The problem is that since the structural testing criteria are obtained from the component's specification, satisfying such criteria does not assure component code coverage but only component specification coverage. Moreover, they require that the component has a formal/semi-formal specification to be applied [87, 69].

The concept of auto-testable components has also been explored. The idea is to provide components with built-in testing capabilities that can be enabled and disabled, depending on when the component is working in the normal operation mode or in the maintenance mode [84, 73, 76]. Edwards [76] discusses how different kinds of information can be embedded into the component. He propose a reflexive metadata wrapper that can be used to pack more than the component itself. It also includes the component specification (formal/semi-formal), its documentation, verification history, violation checking services, self-test services, etc. Such an approach, although very useful, requires more work for the component provider since she/he has to collect and include all of this information inside the component. Moreover, which information should be provided and how it should be provided is not yet standardized, making difficult the development of a testing tool that uses such information during testing.

In general, from the user point of view, when no source code is available, only functional or state-based testing criteria can be used. This implies that, except when the component has built-in test or metacontent capabilities, no information about the coverage of that component with respect to its implementation can be obtained.

Java components, such as applets or JavaBeans, this situation can be overcome by carrying out the testing directly on the Java class file (.class) instead of on the Java source file (.java). The so called "class file" is a portable binary representation that contains class related data such as class name, its superclass name, information about the variables and constants and the bytecode instructions for each method.

Bytecode instructions resemble an assembly-like language, but a class file retains very high-level information about a program such that it is possible to identify control-flow- and data-flow-data dependency on Java bytecode [88]. Working at the bytecode level, both component provider and component user can use the same underlined representation to perform the testing activity. Moreover, the user can evaluate how much of the bytecode of a given component has been covered by a given test set, i.e, he/she can evaluate the quality of his functional test set on covering structural elements of the component [89, 90].

Besides the criteria described above, other examples can be found in the literature, among them, the work of McGregor [91], Rosenblum [71] and Harrold *et al.* [70]. Bhor [92] also carried out a survey that includes other testing techniques for component-based software. Independently of the testing criterion, it is essential that testing tools support its application. In the next section the aspects related to testing automation are discussed.

5.1 Issues on Component Testing Automation

Software component testing, as testing in general, requires several types of supporting tools: test case generators, drivers, stubs and an environment for component testing (component test bed). Taking the developer's point of view, most testing tools for test case generation can be used to support component testing. The main problems in component testing automation are also related with the lack of information embedded into the component by the component providers. The lack of a standard is a drawback since the development of a generic testing environment, in the sense that it would be able to deal with components from different providers and implemented in different languages, would be more difficult. Nevertheless, there are some tools in the literature that automate as much as possible the testing of a component-based system.

A component verification tool, named Component Test Bench (CTB), is described in [93]. The tool provides a generic pattern that allows the component provider to specify the test set used to test a given component. The test cases are stored in standard XML files. The component user can reuse the same test set to retest the component to determine whether the component performs according to its specification. The tool does not yet provide testing criteria to help the test set generation and evaluation. It only does the conformance testing on a given component, i.e., evaluate if the component behaves as specified.

A similar approach is used by JTest [94], a tool that automatically run tests on Java programs. The main difference is that JTest uses the JUnit framework[2] [95] to store the test cases instead of XML file as in CTB.

Glass JAR Toolkit (GJTK) is a coverage testing tool that operates at the bytecode level and does not require the source code to apply a white-box testing criterion (statement or decision) on bytecodes. It can be used for testing both compiled .jar and .class files [96].

We are also working on the implementation of a Java bytecode understanding and testing tool, named JaBUTi [3]. The idea is to provide a complete environment that allows both component provider and component user to carry out white-box testing on Java bytecode. Currently, the tool supports the application of three structural testing criteria (all-nodes, all-edges and all-uses) that can be used in an incremental testing strategy for component-based testing. The idea is 1) to evaluate the coverage of a given functional test set and then, 2) based on the coverage information with respect to the structural testing criteria, to identify which area of the component requires additional test cases to be covered, improving the quality of the test set [90].

[2] JUnit is an open source Java testing framework used to write and run repeatable tests. The basic idea is to implement some specific classes to store the information about test case input and expected output. After each test case execution, the expected output is compared with the current output. Any discrepancy can be reported.

[3] Interested readers can contact us by e-mail to obtain additional information about JaBUTi.

6 JaBUTi: Example of Component Testing Criterion Usage

Most part of the tools developed for component-based testing focus on the functional aspects of the component (black box testing) and do not guarantee statement coverage of the component due to the unavailability of the corresponding source code. JaBUTi supports coverage analysis to test Java programs and Java components even if the Java source code is not available what enables the application of white-box testing. It is designed to support test set adequacy evaluation and to provide guidance in test case selection by using advanced control-flow-dependence analysis.

In [18] a set of testing criteria is defined based on a graphical representation of a class called Class Control Flow Graph (CCFG). Basically, after each CFG is constructed for each method, these CFGs are interconnected representing the method calls inside the class, resulting in a CCFG. Based on the CCFG, the authors consider three testing levels:

Intra-method: each method is tested alone. This level is equivalent to the unit testing for procedural programs;

Inter-method: addresses public methods interacting with other methods in the same class. This level is equivalent to the integration testing for procedural programs; and

Intra-class: tests the interactions between public methods when they are called in different sequences. Since the users of the class can call the methods in an undetermined order, intra-class testing aims at building the confidence that those invocation sequences do not put the object in an inconsistent state. The authors warn that only a subset of the possible sequences can be tested, since the complete set is infinity.

Based on these three levels, Harrold and Rotheermel [18] establish def-use pairs that allow the evaluation of data-flow relations in OO programs: Intra-method pairs, Inter-method pairs and Intra-class pair. Currently, JaBUTi supports the application of two control-flow-based criteria (all-nodes and all-edges) and one data-flow-based criterion (all-nodes) at intra-method level. We intent to implement additional testing criteria to also cover the inter-method level.

In this section we will illustrate the application of the all-uses criterion on a Java component considering the Java source code presented in Fig. 3. The example, extracted from [75], illustrates a component, `Dispenser`, and an application, `VendingMachine`, that uses the `Dispenser` component. Although not required by JaBUTi, we are showing the Java source code in Fig. 3 to facilitate the understanding of the example.

Fig. 4 illustrates the main screen of JaBUTi showing part of the bytecode instruction set of the method `Dispenser.available(int sel)`. Besides the bytecode, the tool also displays the component's source code (when available) as well as the CFG for each method as illustrated in Figures 5 and 6, respectively. Observe that in the CFG, by placing the mouse over a given CFG node, specific information about the node is displayed such as: the node number, the first and

```
01 package vending;
02
03
04 public class VendingMachine {
05
06    final private int COIN = 25;
07    final private int VALUE = 50;
08    private int totValue;
09    private int currValue;
10    private Dispenser d;
11
12    public VendingMachine() {
13       totValue = 0;
14       currValue = 0;
15       d = new Dispenser();
16    }
17
18    public void insertCoin() {
19       currValue += COIN;
20       System.out.println("Current value = " + currValue);
21    }
22
23    public void returnCoin() {
24       if (currValue == 0)
25          System.err.println("no coins to return");
26       else {
27          System.out.println("Take your coins");
28          currValue = 0;
29       }
30    }
31
32    public void vendItem(int selection) {
33       int expense;
34
35       expense = d.dispense(currValue, selection);
36       totValue += expense;
37       currValue -= expense;
38       System.out.println("Current value = " + currValue);
39    }
40 } // class VendingMachine
```

```
01 package vending;
02
03
04 public class Dispenser {
05    final private int MAXSEL = 20;
06    final private int VAL = 50;
07    private int[] availSelectionVals = {2, 3, 13};
08
09    public int dispense(int credit, int sel) {
10       int val = 0;
11
12       if (credit == 0)
13          System.err.println("No coins inserted");
14       else if (sel > MAXSEL)
15          System.err.println("Wrong selection " + sel);
16       else if (!available(sel))
17          System.err.println("Selection " + sel + " unavailable");
18       else {
19          val = VAL;
20          if (credit < val) {
21             System.err.println("Enter " + (val - credit) + " coins");
22             val = 0;
23          } else
24             System.err.println("Take selection");
25       }
26       return val;
27    }
28
29    private boolean available(int sel) {
30       for (int i = 0; i < availSelectionVals.length; i++)
31          if (availSelectionVals[i] == sel) return true;
32       return false;
33    }
34 } // class Dispenser
```

Fig. 3. Example of a Java component (`Dispenser`) and one application (`VendingMachine`) [75]

Fig. 4. JaBUTi display: Hot-spot on bytecode

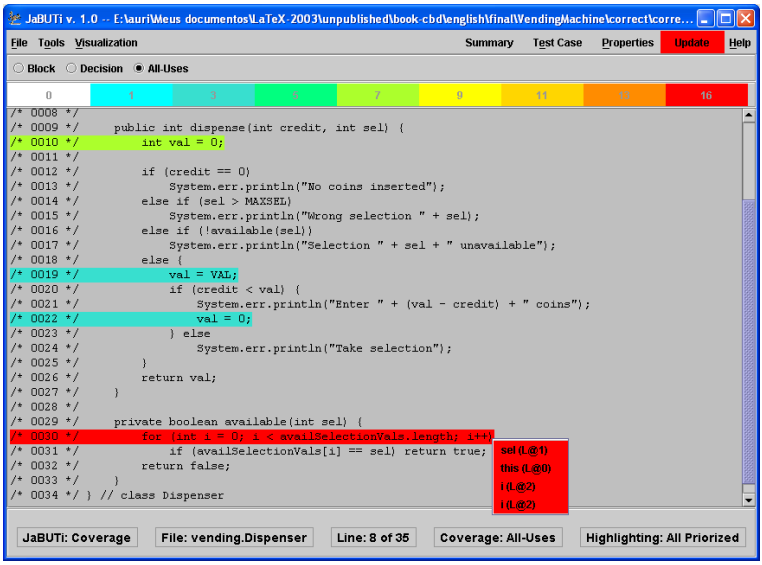

Fig. 5. JaBUTi display: Hot-spot on source code

the last bytecode instruction that compose the node, the set of defined and used variables, etc. In any representation, the tool shows which part has the highest weight [97] and should be covered first to increase the coverage with respect to a given criterion, as much as possible. Observe that, if the tester considers that another part of the component should be covered first, he/she can prioritize the coverage of that specific part and later use the recomputed hints provided by the tool to increase the coverage with respect to the other parts of the component.

For example, considering the all-uses criterion, the tool identifies the complete set of testing requirements (def-use pairs) and shows, using different background colors (weights), the set of instructions that contains a variable definition. By right clicking on one of those instructions, a pop-up menu shows the names of the defined variables at the corresponding bytecode instruction. As can be observed in Fig. 4, the method `Dispenser.available(int sel)` is the one that contains the definition with the highest weight. The same information can also be obtained from the source code or the CFG, as illustrated in Figures 5 and 6, respectively.

By selecting a given definition, the def-use pairs associated with it are shown and the graphical interface is updated. For example, considering Fig. 4, by selecting the definition of the local variable `L@2`[4] in the bytecode, the set of def-use pairs with respect to this variable is shown as illustrated in Fig. 7.

Observe that there are four def-use pairs with respect to `L@2` defined at CFG node 0: <L@2, 0,5>, <L@2, 0,17>, <L@2, 0,(5, 15)> and <L@2, 0,(5, 17)>. The def-use pair with the highest weight is <L@2, 0,5>. By covering this

[4] `L@2` refers to local variable number two. In case of Figure 4, `L@2` corresponds to local variable `i`.

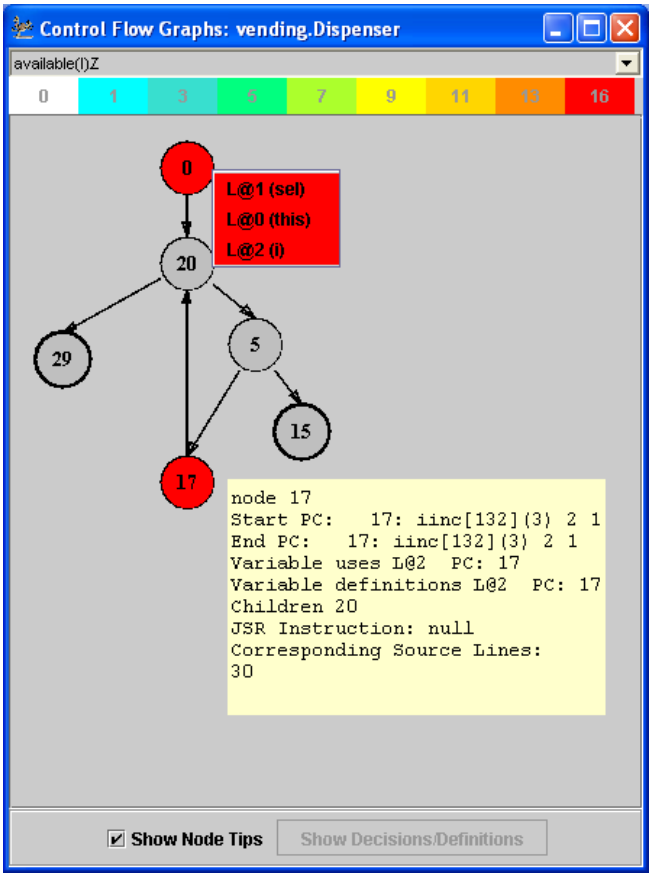

Fig. 6. JaBUTi display: Hot-spot on CFG

requirement, at least 16 other def-use pairs will be covered. For example, the test case "insertCoin, vendItem 13" that represents that a given user inserted one coin and requested the item 13 to be dispensed (a valid item) covers 38 out of the 46 def-use pairs in the two classes under testing (the Dispenser component and the VendingMachine application). By adding one more test case asking for an invalid item ("insertCoin, vendItem 15"), all the testing requirements above were covered.

The tool also generates coverage report with respect to each criterion, each class file, each method and each test case. Fig. 8 illustrates the summary with respect to each test case. The tester can enable and disable different combinations of test cases and the coverage is updated considering only the active test cases. By using JaBUTi, the component user without familiarity with Java bytecode can use the CFG to evaluate how well a given component has been tested. It is also possible to identify which areas of the component code require more test cases to increase the coverage and the confidence in the quality of the component.

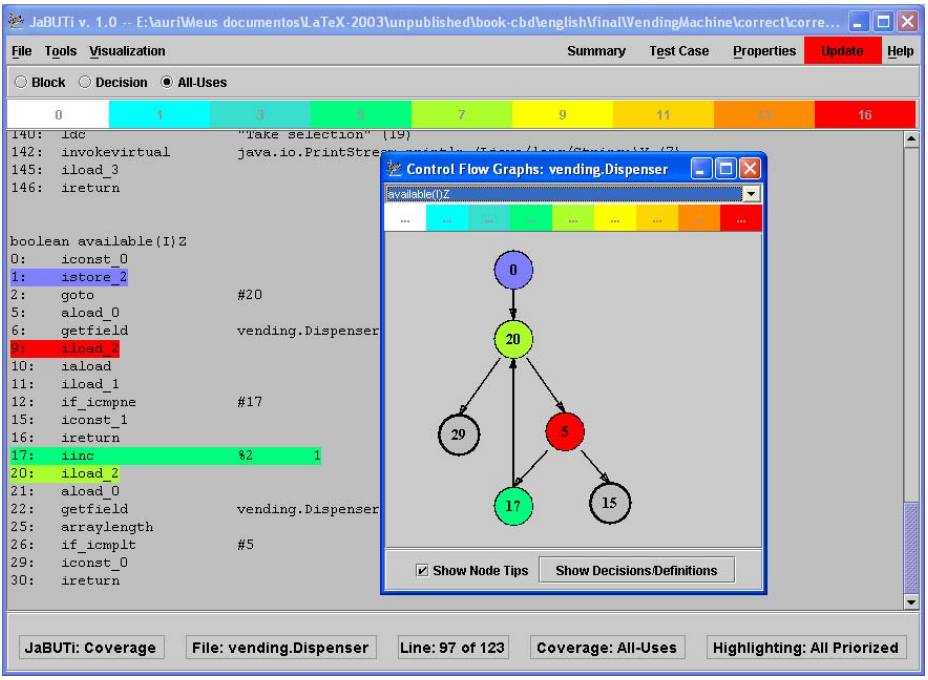

Fig. 7. JaBUTi display: Set of def-use pairs with respect to variable `i` defined at CFG node 0

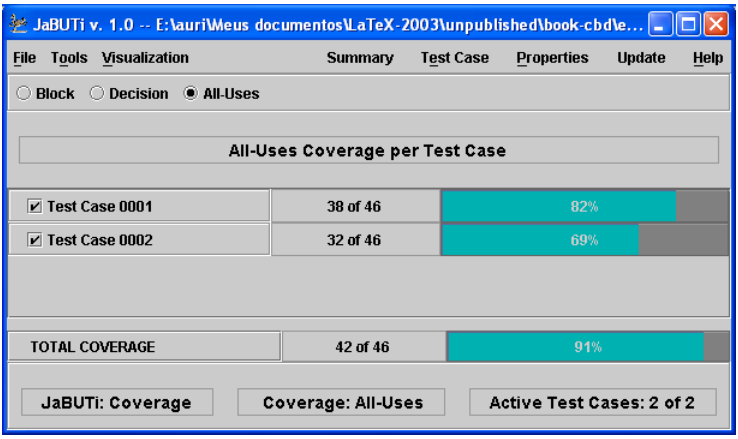

Fig. 8. Summary report by test case

7 Conclusions

This chapter highlighted the importance of the testing activity in the context of component-based system development. The component technology aims at

speeding up the development process by software reuse. However, in order to ensure system quality it is necessary both to consider the concern of the developer to provide mechanisms to promote the validation of their components in their client's applications and the concern of the clients to seek developers that offer components with the quality and reliability that they demand.

In general, the client spends a large amount of time understanding how to use and evaluate the component's quality, making harder their reuse. In this way, for effective reuse it is essential to develop tools to support component testing and to encourage developers to make available, as much as possible, information about the components, reducing the cost and effort required to test a component-based system.

A large research field is open in this area, in particular in the development of techniques, criteria, and tools for component testing. In addition, standards for component communication, for documentation and for the development of built-in test mechanisms are also required. The idea of maturity models for component testing also represents an interesting research line, contributing to component certification, what is fundamental for component systems quality assurance.

The real benefits of component reuse will be achieved to the extent that components fulfill the needs of their clients in terms of quality and reliability and can be easily identified, tracked and tested in the context of the client's applications.

Acknowledgments

The authors would like to thank the Brazilian Funding Agencies – FAPESP, CAPES and CNPq – for their partial support to this research. The authors would also like to thank the anonymous referees for their valuable comments.

References

1. Harrold, M.J.: Testing: A roadmap. In: 22th International Conference on Software Engineering. (2000)
2. Chow, T.S.: Testing software design modelled by finite-state machines. IEEE Transactions on Software Engineering **4** (1978) 178–187
3. Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming **8** (1987) 231–274
4. Freedman, R.S.: Testability of software components. IEEE Transactions on Software Engineering **17** (1991) 553–564
5. Szyperski, C.: Component Software Beyond Object-Oriented Programming. Addison-Wesley (1998)
6. Matena, V., Stearns, B.: Applying Enterprise JavaBeans: Component-Based Development for the J2EE Platform. 2 edn. Addison-Wesley (2001)
7. Microsoft: COM: Delivering on the promises of component technology. web page (2002) Available on-line at: `http://www.microsoft.com/com/` [01-20-2003].
8. Gimenes, I.M.S., Barroca, L., Huzita, E.H.M., Carnielo, A.: The process of component development by examples. In: VIII Regional Scholl of Informatics, Porto Alegre, RS, Brazil (2000) 1–32 (in Portuguese).

9. Werner, C.M., Braga, R.M.: Component-based development. Brazilian Symposium on Software Engineering'2000 (2000) (in Portuguese).
10. McDaniel, R., McGregor, J.D.: Testing polymorphic interactions between classes. Technical Report TR-94-103, Clemson University (1994)
11. Beizer, B.: Software Testing Techniques. 2nd edn. Van Nostrand Reinhold Company, New York (1990)
12. Zhu, H., Hall, P., May, J.: Software unit test coverage and adequacy. ACM Computing Surveys **29** (1997) 366–427
13. Weyuker, E.J.: Testing component-based software: A cautionary tale. IEEE Software **15** (1998) 54–59
14. Binder, R.V.: Testing Object-Oriented Systems: Models, Patterns, and Tools. Volume 1. Addison Wesley Longman, Inc. (1999)
15. Linnenkugel, U., Müllerburg, M.: Test data selection criteria for (software) integration testing. In: First International Conference on Systems Integration, Morristown, NJ (1990) 709–717
16. Pressman, R.S.: Software Engineering – A Practitioner's Approach. 5 edn. McGraw-Hill (2000)
17. IEEE: IEEE standard glossary of software engineering terminology. Standard 610.12-1990 (R2002), IEEE Computer Society Press (2002)
18. Harrold, M.J., Rothermel, G.: Performing data flow testing on classes. In: Second ACM SIGSOFT Symposium on Foundations of Software Engineering, New York, ACM Press (1994) 154–163
19. Perry, D.E., Kaiser, G.E.: Adequate testing and object-oriented programming. Journal on Object-Oriented Programming **2** (1990) 13–19
20. Arnold, T.R., Fuson, W.A.: In a perfect world. Communications of the ACM **37** (1994) 78–86
21. Howden, W.E.: Software Engineering and Technology: Functional Program Testing and Analysis. McGrall-Hill Book Co, New York (1987)
22. Offutt, A.J., Irvine, A.: Testing object-oriented software using the category-partition method. In: 17th International Conference on Technology of Object-Oriented Languages and Systems, Santa Barbara, CA (1995) 293–304
23. Thevenod-Fosse, P., Waeselynck, H.: STATEMATE applied to statistical software testing. In: International Symposium on Software Testing and Analysis, Cambridge, MA, ACM Press (1993) 99–109
24. Whittaker, J.A., Thomason, M.: A markov chain model for statistical software testing. IEEE Transactions on Software Engineering **20** (1994) 812–824
25. Whittaker, J.A.: Stochastic software testing. Annals of Software Engineering **4** (1997) 115–131
26. Banks, D., Dashiell, W., Gallagher, L., Hagwood, C., Kacker, R., Rosenthal, L.: Software testing by statistical methods. Technical Report NISTIR 6129, NIST – National Institute of Standards and Technology (1998)
27. Hoffman, D., Strooper, P.: A case study in class testing. In: CASCON 93, IBM Toronto Laboratory (1993) 472–482
28. Turner, C.D., Robson, D.J.: The state-based testing of object-oriented programs. In: Conference on Software Maintenance, Montreal Quebec, Canada, IEEE Computer Society Press (1993) 302–310
29. Kung, D., J.Gao, Hsia, P., Toyoshima, Y., Chen, C.: On regression testing of object-oriented programs. The Journal of Systems and Software **32** (1996) 21–40
30. McGregor, J.D., Korson, T.D.: Integrated object-oriented testing and development process. Communications of the ACM **37** (1994) 59–77

31. Hoffman, D., Strooper, P.: Classbrench: A framework for automated class testing. Software Practice and Experience **27** (1997) 573–597
32. Frankl, F.G.: The Use of Data Flow Information for the Selection and Evaluation of Software Test Data. PhD thesis, Universidade de New York, New York, NY (1987)
33. Ntafos, S.C.: A comparison of some structural testing strategies. IEEE Transactions on Software Engineering **14** (1988) 868–873
34. Rapps, S., Weyuker, E.J.: Selecting software test data using data flow information. IEEE Transactions on Software Engineering **11** (1985) 367–375
35. Ostrand, T.J., Weyuker, E.J.: Using data flow analysis for regression testing. In: Sixth Annual Pacific Northwest Software Quality Conference, Portland – Oregon (1988)
36. Hartmann, J., Robson, D.J.: Techniques for selective revalidation. IEEE Software **7** (1990) 31–36
37. Veevers, A., Marshall, A.: A relationship between software coverage metrics and reliability. Software Testing, Verification and Reliability **4** (1994) 3–8
38. Varadan, G.S.: Trends in reliability and test strategies. IEEE Software **12** (1995) 10
39. Chaim, M.L.: Program Debugging Based on Structural Testing Information. Doctoral dissertation, Scholl of Computer Science and Electrical Engineering, UNICAMP, Campinas, SP, Brazil (2001) (in Portuguese).
40. Myers, G.J.: The Art of Software Testing. Wiley, New York (1979)
41. Herman, P.M.: A data flow analysis approach to program testing. Australian Computer Journal **8** (1976) 92–96
42. Laski, J.W., Korel, B.: A data flow oriented program testing strategy. IEEE Transactions on Software Engineering **9** (1983) 347–354
43. Rapps, S., Weyuker, E.J.: Data flow analysis techniques for program test data selection. In: 6th International Conference on Software Engineering, Tokio, Japan (1982) 272–278
44. Ntafos, S.C.: On required element testing. IEEE Transactions on Software Engineering **10** (1984) 795–803
45. Maldonado, J.C.: Potential-Uses Criteria: A Contribution to the Structural Testing of Software. Doctoral dissertation, DCA/FEE/UNICAMP, Campinas, SP, Brazil (1991) (in Portuguese).
46. Harrold, M.J., Soffa, M.L.: Interprocedural data flow testing. In: Third Testing, Analysis, and Verification Symposium. (1989) 158–167
47. Harrold, M.J., Soffa, M.L.: Selecting and using data for integration test. IEEE Software **8** (1991) 58–65
48. Vilela, P.R.S.: Integration Potencial-Uses Criteria: Definition and Analysis. Doctoral dissertation, DCA/FEEC/UNICAMP, Campinas, SP, Brazil (1998) (in Portuguese).
49. DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Hints on test data selection: Help for the practicing programmer. IEEE Computer **11** (1978) 34–43
50. Budd, T.A.: Computer Program Testing. In: Mutation Analysis: Ideas, Example, Problems and Prospects. North-Holand Publishing Company (1981)
51. Friedman, A.D.: Logical Design of Digital Systems. Computer Science Press (1975)
52. Delamaro, M.E., Maldonado, J.C., Mathur, A.P.: Interface mutation: An approach for integration testing. IEEE Transactions on Software Engineering **27** (2001) 228–247

53. Fabbri, S.C.P.F., Maldonado, J.C., Masiero, P.C., Delamaro, M.E.: Mutation analysis applied to validate specifications based on petri nets. In: FORTE'95 – 8th IFIP Conference on Formal Descriptions Techniques for Distribute Systems and Communication Protocols, Montreal, Canada (1995) 329–337
54. Simão, A.S., Maldonado, J.C.: Mutation based test sequence generation for Petri nets. In: III Workshop of Formal Methods, João Pessoa (2000)
55. Fabbri, S.C.P.F.: The Mutation Analysis in the Context of Reactive Systems: a Constribution on the Establishment of Validation and Testing Strategies. Doctoral dissertation, IFSC-USP, São Carlos, SP, Brazil (1996) (in Portuguese).
56. Sugeta, T.: Proteum-rs/st: A tool to support the validation of statecharts based on mutation analysis. Master's thesis, ICMC-USP, São Carlos, SP, Brazil (1999) (in Portuguese).
57. Fabbri, S.C.P.F., Maldonado, J.C., Masiero, P.C., Delamaro, M.E.: Mutation analysis based on finite state machines. In: XI Brazilian Symposium on Computer Networks, Campinas, SP, Brazil (1993) 407–425 (in Portuguese).
58. Fabbri, S.C.P.F., Maldonado, J.C., Masiero, P.C., Delamaro, M.E.: Mutation analysis testing for finite state machines. In: 5th International Symposium on Software Reliability Engineering (ISSRE'94), Monterey – CA (1994) 220–229
59. Probert, R.L., Guo, F.: Mutation testing of protocols: Principles and preliminary experimental results. In: IFIP TC6 – Third International Workshop on Protocol Test Systems, North-Holland (1991) 57–76
60. Souza, S.R.S., Maldonado, J.C., Fabbri, S.C.P.F., Lopes de Souza, W.: Mutation testing applied to estelle specifications. Software Quality Journal **8** (1999) 285–301
61. Kim, S., Clark, J.A., Mcdermid, J.A.: The rigorous generation of Java mutation operators using HAZOP. In: 12th International Conference on Software & Systems Engineering and their Applications (ICSSEA'99). (1999)
62. Ma, Y.S., Kwon, Y.R., Offutt, J.: Inter-class mutation operators for Java. In: 13th International Symposium on Software Reliability Engineering - ISSRE'2002, Annapolis, MD (2002).
63. Ghosh, S., Mathur, A.P.: Interface mutation. Software Testing, Verification and Reliability **11** (2001) 227–247 (Special Issue: Mutation 2000 - A Symposium on Mutation Testing. Issue Edited by W. Eric Wong).
64. Sridhanan, B., Mundkur, S., Mathur, A.P.: Non-intrusive testing, monitoring and control of distributed corba objects. In: TOOLS'33 – 33rd International Conference on Technology of Object-Oriented Languages, Mont-saint-Michel, France (2000) 195–206
65. Delamaro, M.E., Pezzè, M., Vincenzi, A.M.R., Maldonado, J.C.: Mutant operators for testing concurrent Java programs. In: Brazilian Symposium on Software Engineering'2001, Rio de Janeiro, RJ, Brazil (2001) 272–285
66. Gönenç, G.: A method for design of fault-detection experiments. IEEE Transactions on Computers **19** (1970) 551–558
67. Sabnani, K.K., Dahbura, A.: Protocol test generation procedure. Computer Networks and ISDN Systems **15** (1988) 285–297
68. Fujiwara, S., Bochmann, G.V., Khendek, F., Amalou, M., Ghedamsi, A.: Test selection based on finite state models. IEEE Transactions on Software Engineering **17** (1991)
69. Beydeda, S., Gruhn, V.: An integrated testing technique for component-based software. In: AICCSA ACS/IEEE International Conference on Computer Systems and Applications, Beirut, Libanon, IEEE Computer Society Press (2001) 328–334

70. Harrold, M.J., Liang, D., Sinha, S.: An approach to analyzing and testing component-based systems. In: First International ICSE Workshop on Testing Distributed Component-Based Systems, Los Angeles, CA (1999)
71. Rosenblum, D.S.: Adequate testing of component-based software. Technical Report UCI-ICS-97-34, University of California, Irvine, CA (1997)
72. Ural, H., Yang, B.: A structural test selection criterion. Information Processing Letters **28** (1988) 157–163
73. Martins, E., Toyota, C.M.: Construction of autotestable classes. In: VIII Symposion of Fault Tolerance Computation, Campinas, SP (1999) 196–209 (in Portuguese).
74. Rosa, A.C.A., Martins, E.: Using a reflexive architecture to validate object-oriented applications by fault injection. In: Workshop on Reflexive Programming in C++ and Java, Vancouver, Canada (1998) 76–80
75. Orso, A., Harrold, M.J., Rosenblum, D., Rothermel, G., Do, H., Soffa, M.L.: Using component metacontent to support the regression testing of component-based software. In: IEEE International Conference on Software Maintenance (ICSM'01), Florence, Italy, IEEE Computer Society Press (2001) 716–725
76. Edwards, S.H.: Toward reflective metadata wrappers for formally specified software components. In: OOPSLA '01 Workshop on Specification and Verification of Component-Based Systems, Tampa, Florida, ACM Press (2001)
77. Harrold, M.J., McGregor, J.D., Fitzpatrick, K.J.: Incremental testing of object-oriented class structures. In: 14th International Conference on Software Engineering, Los Alamitos, CA, IEEE Computer Society Press (1992) 68–80
78. Gao, J.: Tracking component-based software. Technical report, San Jose State University, (San Jose, CA)
79. M.Voas, J.: Certifying off-the shelf-components. IEEE Computer **31** (1998) 53–59
80. Binder, R.V.: Modal testing strategies for OO software. Computer **29** (1996) 97–99
81. Binder, R.V.: The free approach to testing object-oriented software: An overview. Pgina WWW (1996) Available on-line at: `http://www.rbsc.com/pages/FREE.html` [01-20-2003].
82. Kim, S., Clark, J.A., Mcdermid, J.A.: Class mutation: Mutation testing for object-oriented programs. In: FMES. (2000)
83. Edwards, S.H.: A framework for practical, automated black-box testing of component-based software. Software Testing, Verification and Reliability **11** (2001) 97–111
84. Wang, Y., King, G., Wickburg, H.: A method for built-in tests in component-based software maintenance. In: Third European Conference on Software Maintenance and Reengineering, Amsterdam, Netherlands, IEEE Computer Society Press (1999) 186–189
85. Soundarajan, N., Tyler, B.: Testing components. In: OOPSLA '01 Workshop on Specification and Verification of Component-Based Systems, Tampa, Florida, ACM Press (2001) 4–9
86. Rosa, A.C.A., Martins, E.: Using a reflexive architecture to validate object-oriented applications by fault injection. In: Workshop on Reflexive Programming in C++ and Java, Vancouver, Canada (1998) 76–80 (Available on-line at: `http://www.dc.unicamp.br/~eliane`).
87. Edwards, S.H.: Black-box testing using flowgraphs: An experimental assessment of effectiveness and automation potential. Software Testing, Verification and Reliability **10** (2000) 249–262

88. Zhao, J.: Dependence analysis of java bytecode. In: 24th IEEE Annual International Computer Software and Applications Conference (COMPSAC'2000), Taipei, Taiwan, IEEE Computer Society Press (2000) 486–491
89. Zhao, J., Xiang, L., Nishimi, K., Harada, T.: Understanding java bytecode using kafer. In: 20th IASTED International Conference on Applied Informatics (AI'2002), Innsbruck, Austria, ACTA Press (2002)
90. Vincenzi, A.M.R., Delamato, M.E., Maldonado, J.C., Wong, W.E., Simão, A.S.: Jabá – a Java bytecode analyzer. In: XVI Brazilian Symposium on Software Engineering, Gramado, RS, Brazil (2002) 414–419
91. McGregor, J.D.: Parallel architecture for component testing. Journal of Object-Oriented Programming **10** (1997) 10–14 Available on-line at: `http://www.cs.clemson.edu/~johnmc/columns.html` [01-06-2003].
92. Bhor, A.: Component testing strategies. Technical Report UCI-ICS-02-06, Dept. of Information and Computer Science – University of California, Irvine – California (2001) Available on-line at: `http://www.ics.uci.edu/~abhor/` [01-06-2003].
93. Bundell, G.A., Lee, G., Morris, J., Parker, K., Lam, P.: A software component verification tool. In: International Conference on Software Methods and Tools (SMT'2000), Wollongong, Australia (2000)
94. Corporation, P.: Using design by contract to automate java software and component testing. web page (2002) Avaliable on-line: `http://www.parasoft.com/` [01-20-2003].
95. Beck, K., Gamma, E.: JUnit cookbook. web page (2002) Avaliable on-line: `http://www.junit.org/` [01-20-2003].
96. Edge, T.: Glass JAR toolkit. web page (2002) Available on-line at: `http://www.testersedge.com/gjtk/` [01-04-2003].
97. H. Agrawal. Dominators, super blocks, and program coverage. In *Proceedings of the 21st Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL'94)*, Portland, OR (1994) 25–34.