CLEI ELECTRONIC JOURNAL, VOLUME 15, NUMBER 1, PAPER 3, APRIL 2012

An Experimental Study to Evaluate the Impact of the Programming Paradigm in the Testing Activity

Simone do Rocio Senger de Souza
Universidade de São Paulo, Instituto de Ciências Matemáticas e de Computação
P.O. 668, São Carlos (SP), Brazil, 13560-970
srocio@icmc.usp.br

and

Marllos Paiva Prado
Universidade Federal de Goiás, Instituto de Informática
P.O. 131, Goiânia (GO), Brazil, 74001-970
marllosprado@gmail.com

and

Ellen Francine Barbosa, José Carlos Maldonado
Universidade de São Paulo, Instituto de Ciências Matemáticas e de Computação
P.O. 668, São Carlos (SP), Brazil, 13560-970
{*francine,jcmaldon*}*@icmc.usp.br*

Abstract

Several techniques and criteria are available to help conducting testing activity. The choice for one of them depends on different aspects, such as the time restrictions, effectiveness of the testing criteria or the features of the program under test. In this context, the programming paradigm might influence in the testing activity cost. This paper presents the results of an experimental study to characterize and evaluate the cost and strength of structural and functional testing criteria, comparing object-oriented and procedural programming paradigms. A set of 32 programs from the data structure domain was considered in this study. The main goals in the execution of this research were: i) to obtain initial results about the investigated questions; ii) to generate artifacts which can be used as basis to define and conduct further experimental studies; iii) to support training and teaching of software testing activity.

Keywords: Functional and Structural Testing, Programming Paradigms, Experimental Study.

1 Introduction

One of the main goals of the testing activity is to reveal the defects introduced in the program during its development. As a consequence, testing contributes to reduce the maintenance costs and to improve the software quality as well. In this scenario, researchers have worked on the investigation and proposition of testing techniques and criteria to be applied into different programming paradigms, such as procedural and object-oriented (OO) ones.

In short, programming paradigms are responsible for defining the way the software has to be structured, being considered as one of the foundations to the development of computer systems. The adoption of a particular paradigm implies on differences from the way to understand and represent the software structures [1] to the process and technologies used [2]. Currently, the procedural and OO paradigms stand out as the two most used programming paradigms in the software development.

The particularities and differences in how to deal with the issues imposed by each paradigm provide evidences of the possible existence of different sources of errors during the software development [3]. Testing techniques and criteria have been proposed in order to reduce the number of defects in the software through the definition of test cases with a high probability of identifying errors. Functional and structural techniques (and their related criteria) have been widely adopted both in industry and in academy. Besides that, a comparison of testing techniques and criteria can be performed by using some testing properties such as cost, effectiveness and strength.

Some defects can also be related to the programming paradigm adopted. In this perspective, it is important to assess the impact of a specific paradigm on the testing activity. However, most of the experimental studies to evaluate and compare testing criteria are performed with programs built on the same paradigm, not considering the impact of the programming paradigm on the results obtained.

In general, conducting a controlled experiment, or even a *quasi*-experiment, is an expensive activity, with several requirements to be satisfied. A major problem faced by researchers when conducting a new experimental study in software testing is the lack of infrastructure, i.e., materials, tools and results properly prepared to be used as the basis of the new trial. This is not a new concern and efforts in this direction can be found in [4]. In Do et al.' work, the authors made a survey of several papers with experimental results on testing techniques, reporting the difficulties in conducting them. To address such difficulties, the authors proposed an infrastructure consisting of programs, test cases, defects and scripts already established and that would be ready to perform and replicate the controlled experiments.

Besides the need of conducting new experiments on testing techniques and criteria, many researchers have suggested the introduction of testing concepts in conjunction with programming foundations in introductory Computer Science courses [5, 6, 7]. Since testing requires the student to know the behavior of their programs, such activity could be explored to help them understand the abstract concepts of programming and develop the expected skills [6]. Furthermore, since testing forces the integration and the application of theories and skills of software analysis, project and implementation, students who start testing earlier could become better testers and developers as well [5, 8].

Motivated by this scenario, in this paper we summarize the results of an experimental study to characterize and evaluate the cost and strength of structural and functional testing criteria, comparing both procedural and OO programming paradigms. A set of 32 programs from the data structure domain was considered in our experimental study. The main goals in the execution of this research were: i) To obtain initial results about the investigated questions; ii) To generate artifacts which can be used as basis to define and conduct further experimental studies; iii) To support training and teaching of software testing activity.

The remainder of this paper is organized as follows. In sections 2 and 3 the definition and planning of the experimental study are briefly described. In Section 4, the main elements for conducting the experiment are presented. In Section 5 the collected data are analised and the achieved results are discussed. Finally, our conclusions and further work are presented in Section 6.

2 Experiment Definition

The definition of the experimental study was based on the template *Goal Question Metric* [9]. Next, we summarize the main points defined:

- **Object of Study:** The objects of study are the functional and structural testing criteria, applied to procedural and OO paradigms.
- **Purpose:** The purpose is to characterize and evaluate testing criteria, particularly with respect to the differences between the programming paradigms.
- **Quality Focus:** The quality focus is the cost and strength of testing criteria, to be evaluated considering the procedural and OO paradigms.
- **Perspective:** The perspective of the experimental study is considered from the researcher's point of view.
- **Context:** The study is performed by the researcher who defined the experiment, considering a set of programs in the data structure domain. It is conducted as a multi-object variation study, i.e., a participant working on a set of objects.

3 Experiment Planning

In the experiment planning we set out the hypotheses and variables of the study. A plan is created, being used as a guide for the conduction and analysis of the investigated subject. Next we provide an overview of the main steps of the planning activity.

3.1 Context Selection

As we defined before, the main goal of our experiment is to investigate if there are differences in cost and strength of testing criteria due to the paradigms in which the programs under test are written. A set of C and Java programs were taken from [10] and [11], respectively. The programs were developed with academic purposes, aiming at exemplifying, for each paradigm, the same solutions in data structure.

Thus, the context of our experiment can be characterized as an offline study, performed by a graduate student, addressing a real problem – identification and comparison of cost and strength of testing criteria – in a specific context.

3.2 Formulation of Hypotheses

Our experimental study aims at defining characteristics about an unknown subject, looking for evidences that can be used in the definition of future hypotheses. Therefore, the proposed hypotheses are not intended to be definitive, but were established in a pragmatic way in order to facilitate the conduction of our experiment.

We assumed that the cost of the testing criteria on a set of programs can be measured in terms of *the size of the test set* and *the number of elements required*, and these are mainly influenced by: (i) the programming paradigm in which the programs are implemented; (ii) the testing tools adopted; (iii) the type of the testing criteria used; (iv) the skill(s) of tester(s) to adequately test the programs; and (v) the size and complexity of the programs to be tested.

Setting itens (iii) and (iv) as being the same for two sets of programs A and B, respectively implemented in procedural and objected-oriented paradigms, and considering that the programming paradigms represent different ways to understand and structure the same solution, we believe that there is a cost difference between the sets A and B, influenced by the differences in the paradigms.

Similarly to the costs, we assumed that the strength of each functional and structural testing criterion among two paradigms can be measured in terms of the *adequacy of the adequate test set to a given criterion in one paradigm on the same criterion in the opposite paradigm and vice-versa.* Considering that the programming paradigms represent different ways to understand and structure the same solution, we believe that there is a strength difference between the sets A and B (the same as the previous hypothesis), influenced by differences in the paradigms. The formal hypotheses are listed in Table 1.

First Hypothesis	**Second Hypothesis**
Null Hypothesis: There is no difference of cost among the testing criteria, due to the paradigm, among the sets A (procedural) and B (OO). $H0 : Cost(A) = Cost(B)$	Null Hypothesis: There is no difference of strength among the testing criteria, due to the paradigm, among the sets A (procedural) and B (OO). $H0 : Strength(A) = Strength(B)$
Alternative Hypothesis: There is a difference of cost among the testing criteria, due to the paradigm, among the sets A (procedural) and B (OO). $H1 : Cost(A) \neq Cost(B)$	Alternative Hypothesis: There is a difference of strength among the testing criteria, due to the paradigm, among the sets A (procedural) and B (OO). $H1 : Strength(A) \neq Strength(B)$

Table 1: Hypotheses Formalized – Cost and Strength

3.3 Selection of Variables

Based on the context and on the hypotheses established, indepedent and dependent variables for the experiment were defined.

3.3.1 Independent Variables

Independent variable is any variable that can be manipulated or controlled in the process of experimentation. The main independent variables of our study are: i) the paradigm in which the programs are implemented; ii) the testing tools adopted; iii) the testing criteria used; iv) the size and complexity of the programs under test; and v) the tester's skills for applying the testing criteria.

Despite the existence of all these independent variables, the only one considered of interest for our research is the paradigm in which the programs are implemented; indeed, it is the only factor of interest to the experiment. This factor has two treatments: a programming paradigm for OO development and a programming paradigm for procedural development. The other variables in the experiment were set to not interfere in the results obtained.

3.3.2 Dependent Variables

The dependent variables are those in which the result of manipulation of the independent variables can be observed. In our study, the dependent variables defined to describe the costs of the criteria are: i) the number of required elements/program; ii) the number of test cases/program; iii) the number of infeasible required elements/program; iv) the number of required elements/criterion; v) the number of test cases/criterion; and vi) the number of infeasible required elements/criterion; The measures by program correspond to those collected to **each** program, during the application of a given criterion, considering a particular programming paradigm. The measures by criterion correspond to those collected to **all** programs, during the application of a criterion in a particular paradigm.

The dependent variables to describe the strength of the testing criteria are: i) coverage/program in the opposite paradigm; and ii) coverage/criterion in the opposite paradigm.

In addition to the dependent variables mentioned above, some other metrics are defined and collected in our study. The goal is to keep some properties that can be relevant and that are correlated to the behavior of the dependent variables, describing details of the scenario of functional and structural testing with respect to the differences in the programming paradigm. These metrics can also be reused in the context of further experimental studies, both for deriving new hypotheses from the information they provide, as well as to comparing test data from measures which are similar to the programs that compose this study.

Actually, a total of twelve implementation metrics are defined: total units (procedures/methods and constructors), total LOC (lines of code without comments), total decision commands; total recursive commands, total units without parameters, total units with primitive parameters only, total unit with abstract parameters (value or reference), total units with mixed parameters (primitive or abstract), total units without return, total units with return of the primitive type, total units with return of the abstract type (value or reference), and cyclomatic complexity (McCabe).

3.4 Experiment Design

The experiment design has a direct impact on how to analyze the results. A suitable design also serves to minimize the influence of adverse factors on the results. As established earlier, our study addresses only one factor of interest, i.e., the programming paradigm of the programs under test – procedural or OO.

Regarding the other independent variables fixed, we applied some principles of design, aiming at reducing the likelihood of experimental errors within the context defined:

- Nesting of testing criteria: Different testing criteria imply different ways of determining cost and strength in each paradigm. To address this issue, we grouped the testing criteria into three levels (functional, structural control flow and structural data flow) within each paradigm, according to the principle of *nesting*. At the functional level we have *Equivalence Partitioning* and *Boundary Value Analysis* testing criteria. At the structural control flow level we have *All Nodes* and *All Edges* criteria. Finally, at the structural data flow level we have *All Uses* and *All Potential Uses* criteria. The *nesting* principle allows each level to be evaluated separately within each treatment, enabling an effective comparison of similar conditions (levels) between the two paradigms. "Paradigm" is considered the nesting factor while "Testing criterion" is called the nested factor. We chose the *nesting design* instead of the *crossed design* since it allows the comparison of paradigms with respect to the same type of criterion, without implications on the inverse analysis (characteristic of *crossed design*).

- Order of tests: The principle of *randomness* is used to define the order in which the programs should be tested and, for each program, the order of the paradigm to be considered first. The order of tests is nested at each level of the criterion, since the testing strategy defined for the generation of test sets is incremental and, therefore, there is a relationship of interdependence with respect to the test set and the criteria.

For the other independent variables fixed we did not apply any principle of design.

The variable "testing tool" was defined with a fixed and distinct value for each one of the paradigms. In the procedural paradigm, the value was determined by the pair CUTE and Poke-Tool, considering the aplication of functional and structural criteria, respectively [12, 13]. In the OO paradigm, JUnit and JaBUTi play, in the same order, the tools to apply the testing criteria [14, 15]. Although the tools applied in both paradigms are not the same, there is a great similarity in how they operate and in the assertive used. Moreover, both structural testing tools support testing criteria of interest for our study.

The specification of the programs used in the procedural set is the same as the programs used in the OO set. Also, both sets are balanced (same amount of programs for both paradigms).

The test cases were defined for each program and in each level of criteria in an incremental way, i.e., the test set generated at the functional level was reused in the construction of the test set for control flow testing and so on. Actually, it was necessary to organize a testing strategy for enabling such conduction in a systematic way, following the experimental design established.

The testing strategy defined was divided into four sections. The first consisted of generating an adequate test set to the functional criteria. The test set was derived from the specification; therefore, it should be adequate for functional criteria considering both the procedural and the OO versions of the program. This set was the basis for defining the test sets in the next section and the number of required elements in this phase was determined in terms of the number of equivalence classes and boundary values generated.

In the next section we evaluated the coverage of the test set for the second level of testing criteria, i.e. the control flow ones. The coverage achieved in the two paradigms was registered in a testing form of each implementation. Next, new test cases were included into the test set in order to make it adequate for the structural criteria at this level; the infeasible elements were also determined.

The adequate test set obtained in the second section was used as the basis to the structural data flow testing in the third section. Likewise, the coverage achieved in the two paradigms was registered in the testing form of each implementation and new test cases were added in order to make it adequate for the structural criteria at this level; the infeasible elements were also determined.

Finally, the adequate test sets generated (second and third sections) were converted into the language of the opposite paradigm for checking the strength of both types of structural testing.

Figure 1 summarizes the testing strategy described.

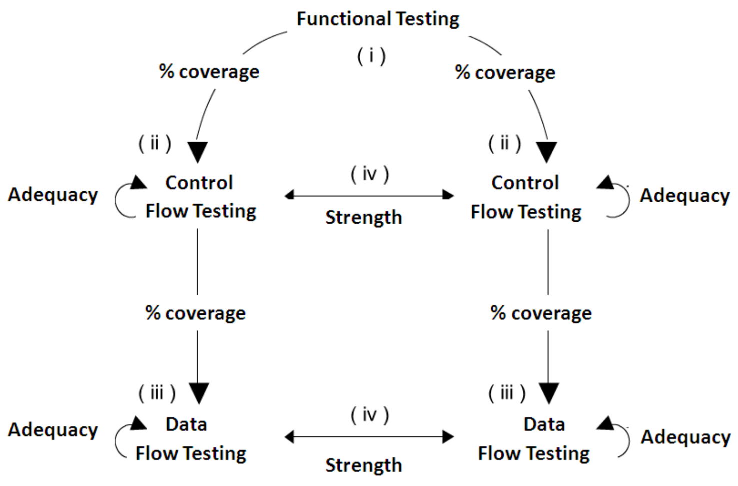

Figure 1: Testing Strategy

3.5 Experiment Instrumentation

The experiment instrumentation was performed in several steps in order to satisfy the three categories of artifacts required for conducting the experiment: experiment objects, guidelines and measurement tools.

The first category consists of the benchmark programs to which were applied the treatments of the experiment. The programs that constitute the benchmark were obtained from [10] (C programs, representatives of the procedural paradigm) and [11] (Java programs, representatives of the OO paradigm). However, designed with academic purposes, these programs required some adjustments to be used in our study. One reason was the mismatch and/or lack of specifications in an appropriate format for testing and independent for each program. The other reason was the possible incompatibility between the implementations of the two paradigms, in terms of functionalities: some operations implemented in the OO language simply did not exist or were different from those implemented in the procedural language, although they refer to the same specification. Because of this, it was necessary to previously prepare these programs according to the pre-established procedures in the experiment plan. In addition, the specifications were rewritten in a pre-defined standard. The format for the specifications was derived and adapted from the Basili's lab package [16].

Most of the specifications were based on the original text of [10] and [11]. The texts were modified or adapted, according to each case, in order to strictly meet the operating needs of the study (as the decoupling from the description of a specification with some sample source code and the removal of dependencies between texts of different specifications). This action was taken in order to interfere as little as possible on the content of the original text and, at the same time, to meet the needs imposed by the principle of randomness, as previously discussed.

The application of treatments in the experiment required other documents to assist in the execution of tests and specifications. Thus, an implementation document was designed in order to provide the tester with the necessary information for constructing the functional test cases, without the need of reading the programs source code (such as input parameters and types of return expected by each operation).

Additionally to the specification and implementation documents, two forms were defined: (i) a form to assist the definition of equivalence classes for the functional testing; and (ii) a testing form used to register the test cases for each criterion evaluated and the test results obtained.

4 Experiment Operation

The experiment operation includes the preparation of artifacts and tools, the execution of activities defined in the experiment plan and the validation of the data collected during the execution.

4.1 Preparation

The preparation for the experiment took place from the plan definition. After setting up the documents and forms for the tests execution, we had to adapt and configure the testing tools in a robust way so that they could be used throughout the experiment.

As one of the purposes of the experiment was to enable the definition of further experimental studies in testing, the testing tools used were installed and configured so that they could be easily reused in new contexts without the need for rework and with some guarantee of a stable operation. We set up a virtual machine on which the testing tools could be combined and configured to be used. Among the benefits noticed, we point out: (i) portability, not only of the tools and of the operating system, but of a entire system configuration; (ii) possibility of replication and backup generation in a systematic and relatively fast way; (iii) feasibility of evolving tools configurations without compromising a configuration already stable; (iv) possibility of interruption and continuation of a task on different dates, maintaining the same state of the operating system and of the programs used; and (v) persistence of the results internally generated by the copy in use.

According to the experiment planning, the test order was randomly assigned. So, as we had 32 specifications, a distinct value in the range of 1 to 32 was assigned to each program in the benchmark. This number was also used to identify each program during the analysis of the data collected and to name the root directory for each program in the virtual machine. Table 2 shows the identifier of each program, associated with its name and a brief description of its functionality.

4.2 Execution

Execution was the phase that required more time in conducting the experiment. Particularly, the functional testing was responsible for most of the time spent, since its entire process of application is manual and only the verification of assertives is supported by tools. Among the tasks required for carrying out the functional testing we highlight: reading and correct understanding of the specification, definition of the equivalence classes and the limits specified for each operation, writing the test cases (defining inputs, outputs and expected results), implementation of test cases according to the languages under study, verification of the assertives and collection of the measures generated.

The conduction of structural testing consumed less time than the functional testing. At this stage, the initial test sets were settled. Among the tasks required, we can point out: measuring the number of elements generated by the required criteria, coverage analysis of the criteria by the functional-adequate test set, adequacy of the test set to the criteria and identification of infeasible elements. The structural data flow testing, in particular, demanded more time than the control flow because the number of required elements generated were, on average, higher than the first and the identification of the causes of non-coverage were less direct.

The strength analysis was the last testing stage performed. It consumed more time than the structural control flow testing and less time than the structural data flow testing. The time spent, however, was significant in relation to the traditional verification of strength. Indeed, because of the changes of language and paradigm, it was necessary to rewrite the test sets of each language in the language of the opposite paradigm.

Identifier	**Name**	**Description**
1	Max	Program to get the maximum value of a set.
2	MaxMin1	Program to obtain the maximum and minimum values of a set, in a non-efficient way.
3	MaxMin2	Program to obtain the maximum and minimum values of a set, in a more efficient way than *MaxMin1*.
4	MaxMin3	Program to obtain the maximum and minimum values of a set, in a more efficient way than *MaxMin2*.
5	Sort1	Program to sort an array of integers.
6	FibRec	Program to calculate the Fibonacci sequence (recursive).
7	FibIte	Program to calculate the Fibonacci sequence (iterative).
8	MaxMinRec	Program to obtain the maximum and minimum values of a set *MaxMin* recursively.
9	Mergesort	Sorting program by the mergesort method.
10	MultMatrixCost	Program to obtain the minimun cost for multiplying n matrices.
11	ListArray	Program to manipulate a list data structure, implemented through arrangements.
12	ListAutoRef	Program to manipulate a list data structure, implemented through auto-reference structures.
13	StackArray	Program to manipulate a stack data structure, implemented through arrangements.
14	StackAutoRef	Program to manipulate a stack data structure, implemented through auto-reference structures.
15	QueueArray	Program to manipulate a queue data structure, implemented through arrangements.
16	QueueAutoRef	Program to manipulate a queue data structure, implemented through auto-reference structure.
17	Sort2	Sorting program by the selection, insertion, shellsort, quicksort and heapsort methods.
18	HeapSort	Program to manipulate priority queues, implemented through arrangements.
19	PartialSorting	Sorting program to get the first k elements of an ordered set of size n.
20	BinarySearch	Program that implements the methods of sequential and binary search for retrieval of information.
21	BinaryTree	Program that implements binary tree and its operations.
22	Hashing1	Program that implements the hashing method, using linked lists for solving collisions.
23	Hashing2	Program that implements the hashing method, using open addressing for solving collisions.
24	GraphMatAdj	Program to manipulate a graph data structure, implemented through adjacency matrices.
25	GraphListAdj1	Program to manipulate a graph data structure, implemented through adjacency lists using auto-reference structures.
26	GraphListAdj2	Program to manipulate a graph data structure, implemented through adjacency lists using arrangements.
27	DepthFirstSearch	Program that implements a depth-first search in a graph.
28	BreadthFirstSearch	Program that implements a breadth-first search in a graph.
29	Graph	Program to obtain strongly connected components of a graph.
30	PrimAlg	Program that implements the Prim's algorithm for discovery the minimum spanning tree in a weighted graph.
31	ExactMatch	Program that implements the exact-match algorithm to search in strings.
32	AproximateMatch	Program that implements the approximate-matching algorithm to search in strings.

Table 2: Set of Programs Used in the Experiment

4.3 Data Validation

A program can contain many procedures/methods and the testing form is intended for general measures, i.e., it considers all the procedures/methods together. In order to avoid misunderstandings during the collection of measures we used drafts to record the individual measures by procedure/method. After all the operation data of a program had been collected in this draft, they were grouped and counted for the same general context. This decision not only served to modularize the task of collecting but also to facilitate the verification of measurements collected during the review of a testing phase.

5 Data Analysis

The data analysis conducted consisted of presenting all the measures for each type of metric considered in the experiment and, together with the measures of central tendency, obtain information that could be individually analyzed, characterizing the context of programs and test sets involved.

Tables 3 and 4 summarize the results obtained, considering the implementation metrics defined in the experiment plan. Taking into account these results we observed that:

- The medium number of conditional statements (*if, while, for*) is equivalent in both paradigms, with $\bar{x} = 8,65$ in the procedural paradigm and $\bar{x} = 8,81$ in the OO paradigm. The standard deviation ($SD_{proc} = 7,12$ and $SD_{OO} = 7,541$) demonstrates an equivalence in the data variation in these paradigms.
- The cyclomatic complexity was the same in 53.13% of the programs and the medium complexity is approximately the same, $\bar{x} = 13,2$ and $SD = 10$ to procedural paradigm and $\bar{x} = 13,8$ and $SD = 10,2$ to OO paradigm.

- The LOC of the procedural programs is, in average, 60.60% greater than the LOC of the OO programs.
- Considering OO programs, the "total of units with parameters of the primitive type" is, in average, 10 times greater than for procedural programs.
- Considering procedural programs, the "total of units with parameters of the abstract type" is 2 times greater than for OO programs.

The obtained information allows verifying some features of the experiment programs. For instance, the results about conditional statements and cyclomatic complexity demonstrate that the logical structure of the programs in different paradigms is very similar. In relation to input parameters, it can be observed that the arguments of the primitive type are more used in OO programs than in procedural programs and, by contrast, the procedural paradigm uses more arguments of the abstract data type. These results provide evidence that the OO paradigm might help the simplest interactions among entities (or object method) than the procedural paradigm, probably due to the cohesion property between the operations of each entity.

The results related to the return statements demonstrate a well-known behavior. In procedural paradigm, for example, it is natural to use pointer against the parameters with the objective to return values between program functions. In OO programs, the reference normally is created inside the method (creation of an object) and returned in the final of the operation.

Static information of the C Programs												
programs	units	LOC	conditional	recursion	without param.	primitive param.	abstract param.	misc param.	without return	primitive return	abstract return	MCabe
Max	1	12	2	0	0	0	1	0	0	1	0	3
MaxMin1	1	14	3	0	0	0	0	1	1	0	0	4
MaxMin2	1	14	3	0	0	0	0	1	1	0	0	4
MaxMin3	1	27	11	0	0	0	0	1	1	0	0	9
Sort1	1	17	3	0	0	0	1	0	1	0	0	4
FibRec	1	6	2	1	0	1	0	0	0	1	0	2
FibIte	1	9	1	0	0	1	0	0	0	1	0	2
MaxMinRec	1	20	5	2	0	1	0	0	1	0	0	5
MergeSort	2	23	4	2	0	0	0	2	2	0	0	6
MultMatrixCost	1	25	6	0	0	1	0	0	0	1	0	7
ListArray	5	44	5	0	0	0	5	0	4	1	0	10
ListAutoRef	4	42	1	0	0	0	4	0	3	1	0	5
StackArray	5	36	4	0	0	0	5	0	3	2	0	7
StackAutoRef	5	49	1	0	0	0	5	0	3	2	0	6
QueueArray	5	39	5	0	0	0	5	0	4	1	0	8
QueueAutoRef	5	50	2	0	0	0	5	0	4	1	0	7
Sort2	9	107	24	2	0	0	9	0	9	0	0	31
HeapSort	7	76	10	0	0	0	6	0	5	0	1	17
PartialSorting	9	117	27	3	0	0	0	9	9	0	0	35
BinarySearch	4	52	10	0	0	0	4	0	2	0	2	11
BinaryTree	7	94	20	9	0	0	6	1	7	0	0	25
Hashing1	12	118	17	0	0	0	12	0	9	1	2	28
Hashing2	8	98	17	0	0	0	7	1	6	2	0	28
GraphMatAdj	9	116	19	0	0	0	9	0	6	2	1	28
GraphListAdj1	2	76	6	1	0	0	0	1	2	0	0	8
GraphListAdj2	9	123	14	0	0	0	8	1	6	2	1	23
DepthFirstSearch	2	76	6	1	0	0	0	1	2	0	0	8
BreadthFirstSearch	2	199	8	0	0	0	1	1	2	0	0	23
Graph	3	130	10	1	0	0	2	1	2	0	1	13
PrimAlg	2	199	8	0	0	0	1	1	2	0	0	23
ExactMatch	4	68	17	0	0	0	0	4	4	0	0	24
AproximateMatch	1	38	6	0	0	0	0	1	1	0	0	7
average	**4,063**	**66,1**	**8,656**	**0,688**	**0**	**0,13**	**3**	**0,84**	**3,187**	**0,594**	**0,25**	**13,2**
std dev	**3,182**	**51,8**	**7,124**	**1,712**	**0**	**0,34**	**3,445**	**1,71**	**2,6933**	**0,756**	**0,568**	**10**

Table 3: Results related to Implementation Metrics for Procedural Programs

Static information of the Java Programs												
programs	units	LOC	conditional	recursion	without param.	primitive param.	abstract param.	misc param.	without return	primitive return	abstract return	MCabe
Max	1	8	2	0	0	0	0	1	0	0	1	3
MaxMin1	1	13	3	0	0	0	1	0	0	1	0	4
MaxMin2	1	13	3	0	0	0	0	1	0	1	0	4
MaxMin3	1	25	11	0	0	1	0	0	0	1	0	9
Sort1	1	14	3	0	0	0	0	1	1	0	0	4
FibRec	1	7	2	2	0	1	0	0	0	1	0	2
FibIte	1	11	1	0	0	1	0	0	0	1	0	2
MaxMinRec	1	20	8	2	0	0	1	0	0	1	0	5
MergeSort	5	22	6	2	0	0	0	2	2	0	0	8
MultMatrixCost	1	31	6	0	0	1	0	0	1	0	0	7
ListArray	5	28	7	0	3	0	2	0	3	1	1	13
ListAutoRef	4	19	1	0	3	0	1	0	2	1	1	5
StackArray	5	23	3	0	4	0	1	0	2	1	2	7
StackAutoRef	5	32	1	0	4	0	1	0	1	2	2	6
QueueArray	5	29	3	0	4	0	1	0	3	1	1	8
QueueAutoRef	5	38	2	0	4	0	1	0	2	1	2	7
Sort2	10	86	21	2	1	1	8	0	9	0	1	32
HeapSort	8	54	10	0	4	2	1	1	5	0	3	19
PartialSorting	9	88	28	3	1	1	0	7	8	0	1	35
BinarySearch	4	30	8	0	0	1	3	0	2	2	0	11
BinaryTree	11	74	30	11	2	0	9	0	5	0	6	29
Hashing1	10	64	12	4	3	4	1	2	3	4	3	19
Hashing2	11	72	17	0	2	5	2	2	5	4	2	29
GraphMatAdj	9	74	14	0	3	6	0	0	2	2	5	27
GraphListAdj1	15	73	6	0	6	8	1	0	2	6	7	24
GraphListAdj2	13	82	17	0	6	7	0	0	4	5	4	27
DepthFirstSearch	3	36	6	1	1	1	1	0	1	1	1	9
BreadthFirstSearch	6	51	9	1	1	4	1	0	4	2	0	15
Graph	5	51	9	1	3	1	1	0	2	1	2	14
PrimAlg	6	51	9	1	1	4	1	0	4	2	0	27
ExactMatch	4	47	18	0	0	4	0	0	4	0	0	25
AproximateMatch	1	26	6	0	0	1	0	0	1	0	0	7
average	**5,25**	**40,4**	**8,813**	**0,938**	**1,75**	**1,69**	**1,188**	**0,53**	**2,438**	**1,313**	**1,406**	**13,8**
std dev	**3,992**	**24,9**	**7,541**	**2,109**	**1,901**	**2,28**	**2,055**	**1,34**	**2,257**	**1,512**	**1,864**	**10,2**

Table 4: Results related to Implementation Metrics for OO Programs

Table 5 summarizes the results about the application cost of the testing criteria, considering the number of required elements (ReqElem), the infeasible elements (Inf.) and the number of test cases adequate (T_{Func}, T_{CF} and T_{DF}).

For the initial test set, the number of required elements required and the number of test cases generated in the procedural paradigm is equal to the OO, since they are derived in terms of the same specifications to both implementations. The data collected provide relevant information about the complexity of the programs under investigation and the test set generated to them.

From Table 5, we can highlight some interesting points. Firstly, the occurrence of the higher values to both "the number of required elements generated" and "the number of generated test cases" (program 25) is not related to the higher occurrence of "cyclomatic complexity" or "conditional statements" (program 19) in any of the two paradigms. Widening the scope for the four higher occurences of "the number of required elements generated" and "the number of generated test cases" (programs 31, 19, 18 and 26), we notice that only program 19 is simultaneously associated with the four higher occurences of "cyclomatic complexity". This information, while not unexpected, is relevant since it provides evidences that, in practice, the increased costs of functional criteria is not necessarily associated to the programs with higher structural complexity. This point highlights the importance of the complementary use of functional and structural techniques.

Another remark of the functional testing with respect to the test set considered is that the average of "the number of test cases that passed" ($\bar{x} = 10,12$ and $SD = 7,38$) is very close to the average of "the number of generated test cases" ($\bar{x} = 10,59$ and $SD = 7,34$). Such information demonstrates that the

	Functional	Control Flow				Data Flow			
	Procedural/OO	Procedural		OO		Procedural		OO	
Programs	ReqElem/T_{Func}	ReqElem/Inf.	T_{CF}	ReqElem/Inf.	T_{CF}	ReqElem/Inf.	T_{DF}	ReqElem/Inf.	T_{DF}
Max	3/3	10/0	3	16/0	4	36/2	3	66/3	4
MaxMin1	5/5	13/0	5	21/0	6	54/2	5	121/3	5
MaxMin2	5/5	14/0	5	21/0	6	47/4	5	121/4	6
MaxMin3	5/5	35/1	6	48/3	7	171/46	6	422/119	8
Sort1	4/3	13/0	3	23/0	4	73/8	3	73/8	3
FibRec	3/3	6/0	3	11/0	4	6/0	3	37/0	4
FibIte	2/3	7/0	3	11/0	4	29/0	3	37/0	4
MaxMinRec	7/7	18/0	7	28/0	8	45/0	7	173/0	8
Mergesort	8/7	15/0	7	37/0	8	64/6	7	304/32	8
MultMatrixCost	7/5	13/0	5	37/0	6	73/8	5	304/37	6
ListArray	11/14	17/0	14	21/0	14	25/0	14	54/0	14
ListAutoRef	7/7	11/0	7	17/0	7	16/0	7	31/0	7
StackArray	13/17	19/0	17	19/0	17	17/0	17	36/0	17
StackAutoRef	10/10	14/0	10	15/0	10	11/0	10	20/0	10
QueueArray	8/14	16/0	14	18/0	14	16/0	14	42/0	14
QueueAutoRef	10/10	17/0	10	19/0	10	23/0	10	46/0	10
Sort2	20/15	71/4	15	103/0	16	329/17	19	676/38	21
HeapSort	23/21	55/0	23	72/0	21	140/11	27	194/11	27
PartialSorting	25/25	113/2	25	130/0	21	560/59	25	979/76	22
BinarySearch	9/12	33/2	13	40/0	13	92/0	15	198/3	16
BinaryTree	16/14	48/0	15	63/0	15	79/0	15	280/0	15
Hashing1	13/11	45/0	12	54/0	11	89/4	14	240/6	12
Hashing2	13/11	58/0	12	98/0	14	184/10	14	559/25	17
GraphMatAdj	17/19	81/0	19	96/0	20	262/44	19	450/37	20
GraphListAdj1	19/18	87/1	21	75/0	19	166/21	32	233/21	21
GraphListAdj2	19/19	66/0	21	85/0	22	182/21	25	516/25	25
DepthFirstSearch	1/1	25/0	1	42/0	1	113/2	2	253/2	2
BreadthFirstSearch	5/5	30/0	5	61/0	9	140/6	5	384/22	9
Graph	5/4	42/2	4	34/0	4	204/19	5	150/3	5
PrimAlg	4/4	16/0	4	49/0	3	232/15	4	399/22	31
ExactMatch	28/32	75/0	32	131/0	33	464/7	40	984/3	42
AproximateMatch	9/10	25/0	10	44/0	11	224/2	15	427/5	16

Table 5: Application Cost of the Functional and Structural Testing Criteria

sample set of programs considered have almost no errors, at least in terms of the functional testing. This is an expected result since the programs used were taken from materials for teaching.

Let's now summarize the main results observed about the application cost of the control flow criteria. In short, we noticed that the All-Edges criterion showed an increase of approximately 100% in "the number of required elements", from the procedural to the OO paradigm, both in the lower as well as in the higher and in the medium occurences. Therefore, to be a threat of validity of the results, this metric should not be used alone for comparing programming paradigms.

The All-Nodes criterion, in turn, did not present major discrepancies in the number of required elements, since the measures of central relative tendency (mean and median) showed very close for both paradigms. The unexpected similarity in this case may be due to the advantage of measures of LOC noticed in the procedural set, which would be "compensating" the optimization gains produced by the testing tool.

Regarding the "infeasible elements", due to the low incidence of them on the criteria analysed, the results obtained did not allow to achieved significant conclusions. However, because they are related to the number of required elements, the results of this metric can also be influenced by the optimization algorithms of the testing tool in C and the same decision of "the number of required elements" must be considered.

With respect to "the number of generated test cases", we cannot detect major differences in cost between the paradigms in relation to the control flow criteria, since the measures of central tendency showed to be very close in both cases. This result is consistent with the observations of "cyclomatic complexity" and "the number of conditional statements", which also showed to be close in both paradigms.

Finally, let's consider the main results observed about the application cost of the data flow criteria. Again, we noticed significant discrepancies among the values for "the number of required elements" and "infeasible elements" with respect to the procedural and OO paradigms. As we pointed our for the control flow criteria, these results may be influenced by the optimizations applied by the structural testing tool in the procedural paradigm, which favors the reduction of these measures.

Regarding "the number of generated test cases", we cannot detect major differences in cost between the paradigms in relation to the data flow criteria, since the measures of central tendency showed to be very close in both cases.

Table 6 summarizes the results about the strength of the testing criteria, considering the procedural and OO paradigms. In the left columns, the coverage refers to the execution of the procedural programs with OO-adequate test sets; in the right columns, the coverage refers to the execution of the OO programs with procedural-adequate test sets. From the table we highlight:

- For the control flow criteria, the OO-adequate test sets presented a relatively higher coverage on the procedural programs ($\bar{x} = 96\%$ and $SD = 0,09$) than the procedural-adequate test sets on the OO programs ($\bar{x} = 93\%$ and $SD = 0,09$).
- For the data flow criteria, the opposite occurred. The procedural-adequate test sets presented a relatively higher coverage on the OO programs ($\bar{x} = 93\%$ and $SD = 0,08$) than the OO-adequate test sets on the procedural programs ($\bar{x} = 89\%$ and $SD = 0,12$).
- The lowest coverage of a testing criterion on a program occurred in the procedural paradigm (All-Pot-Uses criterion, 33%). Such coverage was 50% lower than in the OO paradigm.
- All criteria had occurrences with coverage equals to 100% in both paradigms.

Based on the results obtained, we can notice that the control flow criteria presented, on average, a slightly higher strength in the OO paradigm, while the data flow criteria presented, on average, a slightly higher strength in the procedural paradigm.

6 Conclusions and Further Work

In this paper we described an experimental study to characterize and evaluate the cost and strength of structural and functional testing criteria with respect to different programming paradigms, more specifically the procedural and the object-oriented paradigms. A set of 32 programs from the data structure domain was considered. In short, the results obtained from our experiment did not provide evidences for the existence of differences in cost and strength between procedural and OO paradigms. More details about the definition, planning and conduction of the experiment as well as a detailed analysis of the data collected are available at [17].

Even though we are in the right direction, further experimental studies should be performed, considering a broader context of programs, languages, and participants. Such studies can provide more evidence in favor of accepting the null hypothesis, or reveal other areas where the data provide sufficient evidence to refute them. Further studies to verify the hypotheses of this work considering other testing techniques and criteria are also appropriate.

As a final remark, we highlight that our work also contributes to the current need of Software Engineering in establishing and performing more systematic and rigorous experimental studies, increasing the validity of the conclusions achieved and enabling replication/expansion of such studies in different settings. In this perspective, the framework of artifacts generated, the practical experience gained in the execution of each step of the experimental process (definition, planning, operation and analysis) and the results obtained at the end of the investigation, can be useful to the definition and conduction of further experimental studies in the testing area. Moreover, we intend that both the generated material as well as the testing results obtained can be useful to support teaching and training of software testing.

Acknowledgments

The authors would like to thank the Brazilian funding agencies (FAPESP, CAPES, CNPq) for their support.

	OO-adequate → Procedural				Procedural-adequate → OO			
Programs	all-nodes	all-edges	all-uses	all-Pot-uses	all-nodes	all-edges	all-uses	all-Pot-uses
Max	100%	100%	95,23%	93,33%	75%	87,50%	100%	97,56%
MaxMin1	100%	100%	96,67%	95,00%	80%	90,90%	100%	98,72%
MaxMin2	100%	100%	93,33%	90%	80%	90,90%	100%	98,72%
MaxMin3	95,24%	92,31%	75,83%	72,50%	85,71%	88,88%	63%	66%
Sort1	100%	100%	93,93%	85%	81,81%	91,66%	93,94%	92,30%
FibRec	100%	100%	100%	100%	66,66%	80%	100%	96,15%
FibIte	100%	100%	85,71%	83,33%	66,66%	80%	100%	96,15%
MaxMinRec	100%	100%	100%	100%	84,62%	93,33%	100%	97,52%
Mergesort	100%	100%	95%	91,66%	88,23%	95%	92,20%	88,11%
MultMatrixCost	100%	100%	93,93%	85%	88,23%	95%	92,21%	86,50%
ListArray	100%	100%	100%	100%	100%	100%	100%	100%
ListAutoRef	100%	100%	100%	100%	100%	100%	100%	100%
StackArray	100%	100%	100%	100%	100%	100%	100%	100%
StackAutoRef	100%	100%	100%	100%	100%	100%	100%	100%
QueueArray	100%	100%	100%	100%	100%	100%	100%	100%
QueueAutoRef	100%	100%	100%	100%	100%	100%	100%	100%
Sort2	87,50%	72,82%	93,29%	89,44%	95,74%	98,22%	93,10%	92,39%
HeapSort	91,89%	83,33%	87,65%	76,27%	97,33%	65,78%	95,55%	95,50%
PartialSorting	100%	100%	86,22%	81,66%	96,61%	98,59%	90,84%	91,79%
BinarySearch	95,45%	90,90%	93,87%	81,40%	100%	100%	95%	88,16%
BinaryTree	100%	100%	100%	100%	89,79%	86,54%	83,12%	91,11%
Hashing1	96,96%	93,75%	88,88%	85,71%	96,77%	93,55%	94,06%	92,50%
Hashing2	100%	100%	94,59%	91,43%	78,33%	72,60%	68,89%	70,25%
GraphMatAdj	100%	100%	88,97%	76,98%	100%	100%	93%	86%
GraphListAdj1	96,11%	55%	46,06%	33,33%	97,36%	100%	94,44%	88,54%
GraphListAdj2	100%	100%	88,17%	87,64%	95,74%	95,83%	96,27%	91,80%
DepthFirstSearch	100%	100%	94,12%	95,16%	100%	100%	97,50%	97,11%
BreadthFirstSearch	100%	100%	100%	100%	72,41%	78,12%	75,67%	84,61%
Graph	96,55%	92,30%	87,77%	90,35%	100%	100%	96%	96%
PrimAlg	100%	87,50%	94,37%	93,17%	95,65%	100%	95,55%	92,88%
ExactMatch	100%	100%	92,51%	93,86%	96,61%	98,66%	97,07%	97,30%
AproximateMatch	100%	100%	90%	90,97%	90%	95,83%	98,63%	99,43%
average	99%	96%	92%	89%	91%	93%	94%	93%
std dev	0,03	0,10	0,10	0,13	0,10	0,09	0,09	0,08

Table 6: Strength of the Functional and Structural Testing Criteria

References

[1] T. Takashi, H. K. E. Liesenberg, and D. T. Xavier, *Programação Orientada a Objetos – Uma visão Integrada do Paradigma de Objetos*, 1st ed. São Paulo - SP: VII Escola de Computação, 1990.

[2] R. S. Pressman, *Engenharia de Software*, 5th ed. McGraw-Hill, 2002.

[3] A. M. R. Vincenzi, A. L. S. Domingues, M. E. Delamaro, and J. C. Maldonado, *Introdução ao Teste de Software*, 1st ed. Campus / Elsevier, 2007, ch. Teste Orientado a Objetos e de Componentes, pp. 119,174.

[4] H. D. S. E. G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering: An International Journal*, vol. 10, no. 4, 2005.

[5] E. F. Barbosa, R. LeBlanc, M. Guzdial, and J. C. Maldonado, "Introducing testing practices into objects and design course," in *16th Conference on Software Engineering Education and Training (CSEET 2003)*, Madrid, Spain, Mar. 2003, pp. 179–286.

[6] S. H. Edwards, "Improving student performance by evaluating how well students test their own programs," *Journal on Educational Resources in Computing*, vol. 3, no. 3, p. 24p., 2003.

[7] E. Lahtinen, K. Ala-Mutka, and H. Järvinen, "A study of the difficulties of novice programmers," in *ITiCSE '05: Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education.* New York, NY, USA: ACM Press, 2005, pp. 14–18.

[8] E. L. Jones, "An experimental approach to incorporating software testing into the computer science curriculum," in *31st ASEE/IEEE Frontiers in Education Conference*, Reno, Nevada, 2001, pp. 7–11.

[9] L. C. Briand, M. C. Differding, and H. D. Rombach, "Practical guidelines for measurement-based process improvement," *Software Process Improvement and Practice*, vol. 2, no. 4, pp. 253,280, 1996.

[10] N. Ziviani, *Projeto de Algoritmos com Implementações em Pascal e C*, 2nd ed. Thomson, 2005.

[11] ——, *Projeto de Algoritmos com Implementações em Java e C++.* Thomson, 2005.

[12] P. Sommerlad and E. Graf, "Cute: C++ unit testing easier," in *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion.* New York, NY, USA: ACM, 2007, pp. 783–784.

[13] M. L. Chaim, "Poke-Tool — uma ferramenta para suporte ao teste estrutural de programas baseados em análise de fluxo de dados," Master's thesis, DCA/FEEC/UNICAMP, Campinas, SP, Apr. 1991.

[14] D. Riehle, "Junit 3.8 documented using collaborations," *SIGSOFT Softw. Eng. Notes*, vol. 33, no. 2, pp. 1–28, 2008.

[15] A. M. R. Vincenzi, W. E. Wong, M. E. Delamaro, and J. C. Maldonado, "JaBUTi: A coverage analysis tool for Java programs," in *XVII SBES – Simpósio Brasileiro de Engenharia de Software*, Manaus, AM, Brasil, Oct. 2003, pp. 79–84.

[16] V. Basili, S. Green, O. Laitenberger, F. Lanubile, F. Shull, S. Sørumgård, and M. Zelkowitz, "Lab package for the empirical investigation of perspective-based reading," *World Wide Web*, 2008. [Online]. Available: http://www.cs.umd.edu/projects/SoftEng/ESEG/manual/pbr_package/manual.html

[17] M. P. Prado, "Um estudo de caracterização e avaliação de critérios de teste estruturais entre os paradigmas procedimental e OO," Dissertação de Mestrado, ICMC/USP, São Carlos, SP, 2009.