Effectiveness and Cost of Verification Techniques Preliminary Conclusions on Five Techniques

Diego Vallespir
Instituto de Computación
Facultad de Ingeniería, Universidad de la República
Montevideo, Uruguay
dvallesp@fing.edu.uy

Juliana Herbert
Herbert Consulting
Porto Alegre, RS, Brazil
juliana@herbertconsulting.com

***Abstract*—A group of 17 students applied 5 unit verification techniques in a simple Java program as training for a formal experiment. The verification techniques applied are desktop inspection, equivalence partitioning and boundary-value analysis, decision table, linearly independent path, and multiple condition coverage. The first one is a static technique, while the others are dynamic. JUnit test cases are generated when dynamic techniques are applied. Both the defects and the execution time are registered. Execution time is considered as a cost measure for the techniques. Preliminary results yield three relevant conclusions. As a first conclusion, performance defects are not easily found. Secondly, unit verification is rather costly and the percentage of defects it detects is low. Finally desktop inspection detects a greater variety of defects than the other techniques.**

***Keywords*-Unit testing; Testing; Software engineering; Empirical software engineering;**

I. Introduction

It is normal to use a hammer to hammer a nail into a wall. There are different types of hammers but it is easy to choose one and even more, a lot of different hammers do the same job. It is normal to use a software verification technique to verify a software unit. It is not known which one to choose.

To know which verification technique to choose for unit testing we must know several things, for example, the cost, the effectiveness and the efficiency of each technique. Even more, these things can vary depending on the person who applies it, the programming language and the application type (information system, robotics, etc.). Some advances have been made but we have a long way to go.

Here we define the cost of the technique as the time that takes its execution, the effectiveness as the percentage of defect found by the technique and the efficiency as the time that takes to find a defect.

In [1] the authors examine different experiments on software testing: [2], [3], [4], [5], [6], [7], [8], [9], [10], finding that:

- It seems that some types of faults are not well suited to some testing techniques.
- The results vary greatly from one study to another.
- When the tester is experienced
  - Functional testing is more effective than coverage all program statements, although the functional approach takes longer.
  - Functional testing is as effective as conditional testing and consumes less time.
- In some experiments data-flow testing and mutation testing are equally effective.
- In some other experiments mutation testing performed better than data-flow testing in terms of effectiveness.
- In all the experiments mutation testing was more expensive than data-flow testing.
- Changes of programming languages and/or environments can produce different results in replications of experiments that are rather old.
- Most programs used in the experiments suffer from at least one of these two problems:
  - They are small and simple.
  - The defects are seeded by the researches.

One of their observations is that researchers should publish more information not only about the number of faults the techique can remove but also about the types.

We are in the execution phase of an experiment. It uses 4 programs that are built specially for the experiment. We use 2 taxonomies to classify defects, IBM Orthogonal Defect Classification [11] and Beizer's Taxonomy [12], therefore we are able to discuss the results by defect type.

Before an experiment starts, the testers (the subjects who will execute the verification techniques) need previous preparation. In our experiment this includes a course on every technique to be applied, a course on the scripts to use during execution, a course on IBM's and Beizer's Taxonomies and a training execution of the techniques in a simple program. This execution serves to adjust the scripts, to assure that every tester understands the techniques and to have some ideas of what it can be expected from each technique. In this paper we present the training phase of the experiment and the associated results. The results do not have statistical validity. They are just observations during the tester's training and before the execution of the real experiment.

The most important result is that verification is really expensive and can find a poor quantity of the defects. As it was said before, this is the result of a training phase in an experiment with undergraduate students, thus more data is needed. However, the results can be still considered from a software development point of view: quality has to be built during the construction phase and not during the testing phase. This is in some way related to PSP and other Humphrey ideas [13], [14] and to Pair programming ideas as well [15].

The article is organized as follows. Section II presents the techniques used in the training, the scripts and the taxonomies. Section III presents the Java program that is verified by the testers. The defects that the program contains are listed in section IV. The results obtained are presented in section V and the conclusions in section VI.

II. Techniques, Taxonomies and Scripts

We use the same terminology for the verification techniques as Swebok [16]. The techniques can be divided in different types: static, tester intuition or experience, specification based, code based, fault based and usage based. At the same time code-based is divided in control flow and data flow based criteria.

In our experiment we choose 5 testing techniques: desktop inspection, equivalence partitioning and boundary-value analysis (EP), decision table (DT), linearly independent path (LIP), and multiple condition coverage (MCC). Using these techniques the static, specification-based and control-flow based techniques types are covered. Swebok considers equivalence partitioning and boundary-value analysis as two separate techniques. Given they are generally used together, the testers apply them as one.

We could not find literature describing experiments in which DT, LIP and MCC techniques are applied, neither could Juristo [17]. So, the experiment we are leading may be the first one that applies these techniques.

We want to know the effectiveness of the techniques according to defect types, so a defect taxonomy is necessary. Various defect taxonomies are presented in the literature. The IBM Orthogonal Defect Classification (ODC) is the most used [11]. Other taxonomy of interest is Beizer's Defect Taxonomy [12].

ODC allows the defects to be classified in many orthogonal views: defect removal activities, triggers, impact, target, defect type, qualifier, age and source. In the experiment we only take into account the defect type and the qualifier. The defect type can be one of the following: assign/init, checking, algorithm/method, function/class/object, timing/serial, interface/O.O. messages and relationship. The qualifier can be: missing, incorrect or extraneous. So, every defect must be classified in both views, for example, a defect could be classified as "timing/serial incorrect".

Beizer's taxonomy is hierarchical in the sense that each category is divided in sub-categories and so on. For example, the category number 3 is "Structural bugs" and is divided in 3.1 "Control flow and sequencing" and 3.2 "Processing". Category 3.1 is divided in several sub-categories more. This taxonomy presents a lot of different types of defects, so it may be interesting to use it. By doing this, our knowledge about the effectiveness of the techniques by defect type will be highly improved.

The testers follow scripts that provide them with guidance, thus they are able to execute the technique and register the data required in the experiment correctly. There are 3 scripts, one for each type of technique used: static, specification-based and control-based. These always consist of the same phases: preparation, design, execution and finish.

In the preparation phase the tester is already able to start the verification job. In the design one the tester develops the test cases that achieve the verification technique criteria. During execution the test cases are executed and the tester searches for the defects of every case that fails. In the last phase, the finish, the tester closes the job. In every phase the tester has to register the time elapsed during the activities and every defect found. These phases are slightly different for inspection technique.

III. The Program

The program that the testers used in the training is an uncomplicated and very simple Java program. It is uncomplicated because its function is to order an array of integers and eliminate every duplicated element. It is simple because it only consists of two classes, one of these has 18 uncommented LOCs and the other has 19. This program is used in the training and not in the experiment.

A. Specification and Source Code

Figure 1 shows the collaboration diagram of the two classes of the program. Each class has a only one public method with its specification.

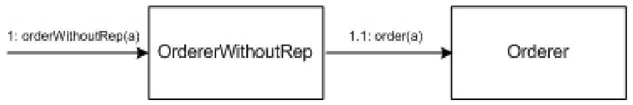


Figure 1. UML Collaboration Diagram of the Program

The following presents both the signature and the specification of the ***order*** method of the ***Orderer*** class.

---

public static void **order**(int[] a)

> This method returns the a array ordered from the lowest to the greatest.
> In the case that the array is null or empty it remains unchanged.

For example: a = [1, 3, 5, 3, 3]. After the method is executed the a array changes to [1, 3, 3, 3, 5]

Parameters
a - array of integer to be ordered

The following is the signature and the specification of the ***orderWithoutRep*** method of the ***OrdererWithoutRep*** class.

---

public static int **OrdererWithoutRep**(int[] a)

This method returns the a array orderer from the lowest to the greatest and without repeated integers from the position 0 to the position "a.length - quantity of repeated integers - 1".
The values in the a array from position "a.length - quantity of repeated integers - 1" to position "a.length - 1" are unknown.
In the case that the array is null or empty it remains unchanged and the method returns the value 0.

For example: a = [5, 4, 5, 6, 6, 5]. The quantity of repeated integers is equal to 3. Number 5 is repeated twice and number 6 is repeated one.
After this method is executed the a array from position 0 to position 2 must be: [4, 5, 6]. And the values in the a array from position 3 to 5 are unknown (do not matter).
The position 2 is calculated as $6 - 3 - 1$. This is length - quantity of repeated integers - 1.
The method returns the value 3 (quantity of repeated integers).

Parameters
a array of integers to be ordered
Returns
the quantity of repeated elements

The following is the source code of the **Orderer** class:

```
1  public class Orderer {
2
3      public static void order (int[] a){
4          for(int i=a.length-1; i>0; i--){
5              int swapped = 0;
6              int find = 0;
7              for (int j=0; j<i; j++){
8                  if (a[j] > a[j+1]){
9                      int aux = a[j];
10                     a[j+1] = a[j];
11                     a[j] = aux;
12                     swapped=1;
13                 }
14             }
15             if (swapped == 0) {
16                 return;
17             }
18         }
19     }
20 }
```

The following is the source code of the **OrdererWithoutRep** class:

```
1  public class OrdererWithoutRep {
2
3      public static int orderWithoutRep(int[] a){
4          int countElim = 0;
5          Orderer.order(a);
6          for(int i=0; i<a.length-1; i++){
7              if (a[i] == a[i+1]) {
8                  move(a, i+1);
9                  countElim++;
10             }
11         }
12         return countElim;
13     }
14
15     private static void move(int[] a, int i){
16         for(int j=i; j<a.length-1; j++){
17             a[j]=a[j+1];
18         }
19     }
20 }
```

IV. Defects

This section presents the defects in the code that are relevant to our analysis, other defects exist but are of much less importance.

The defects are classified as *Possible Failure* (PF) or *Not Failure* (NF). The PF defects are those which may produce a failure during the execution of the program. The NF defects never produce a failure during execution but may cause other problems; for example, performance problems or problems during the software maintenance phase.

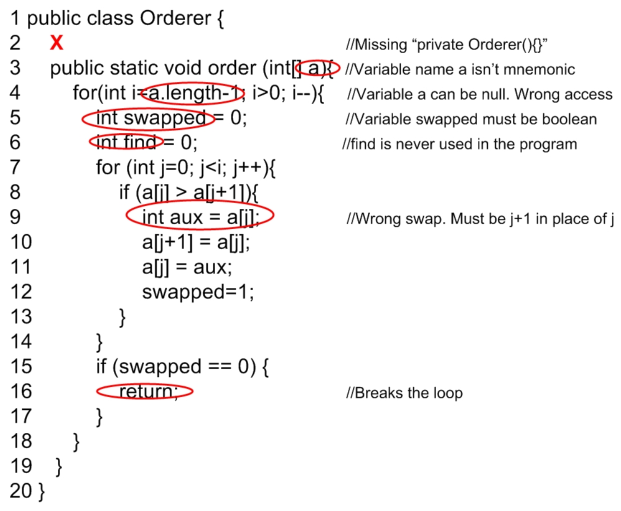
The class ***Orderer*** has 7 defects to consider. They are named with uppercase letters from letter *A* to letter *G*. From the class *OrdererWithoutRep* 6 defects are analyzed, which are named with lowercase letters from letter *a* to letter *f*.

A. Orderer's Defects

Here we present the defects of the ***Orderer*** class. Figure 2 shows the defects in the code with a ellipse around them. The following presents the description of each defect.

Defect A - PF
The ***order*** method starts with a `for` sentence in line 4: This sentence access to `a` array through `length`. If the array is null during execution a failure is produced and the program finishes abruptly. The defect is that `a` is not checked for `null` prior to access it.

Defect B - PF
The ***order*** method makes a swap between variables of the array, this occurs from line 9 to 11 in the code. The swap is wrong because the value that contains `a[j+1]` is not



```
1 public class Orderer {
2   X                                    //Missing "private Orderer(){}"
3   public static void order (int[] a){  //Variable name a isn't mnemonic
4     for(int i=a.length-1; i>0; i--){   //Variable a can be null. Wrong access
5       int swapped = 0;                 //Variable swapped must be boolean
6       int find = 0;                    //find is never used in the program
7       for (int j=0; j<i; j++){
8         if (a[j] > a[j+1]){
9           int aux = a[j];              //Wrong swap. Must be j+1 in place of j
10          a[j+1] = a[j];
11          a[j] = aux;
12          swapped=1;
13        }
14      }
15      if (swapped == 0) {
16        return;                        //Breaks the loop
17      }
18    }
19  }
20 }
```

Figure 2. Defects of the Orderer Class

preserved. A failure due to this defect is shown in Figure 3. This figure presents the state of the array `a` after the execution of the inner `for`.

```
a = [1, 3, 5, 3, 3]      Expected Result = [1, 3, 3, 3, 5]

i = 4, j = 0, a[0] <= a[1]  =>  a = [1, 3, 5, 3, 3]
i = 4, j = 1, a[1] <= a[2]  =>  a = [1, 3, 5, 3, 3]
i = 4, j = 2, a[2]  >  a[3]  =>  a = [1, 3, 5, 5, 3] //Failure due to B
i = 4, j = 3, a[3]  >  a[4]  =>  a = [1, 3, 5, 5, 5] //Failure due to B
i = 3
The array is in order, so there won't be more swaps.
Line 16 will return from the method.

Obtained result = [1, 3, 5, 5, 5]
```

Figure 3. An Execution of ***order*** Method Showing Defect B Occurrence

Defect C - NF

The class has only one method and it is a static one. It is not reasonable to construct an object of this class. If the Java compiler does not found a constructor it automatically creates a parameterless public method by default, allowing the creation of objects of this class. A private constructor method is needed in order to avoid this.

Defect D - NF

The variable `swapped` must be boolean but it is defined as an int. The variable is defined in line 6.

Defect E - NF

The variable name for the array `a` is not mnemonic. Replacing the name affects several lines of code.

Defect F - NF

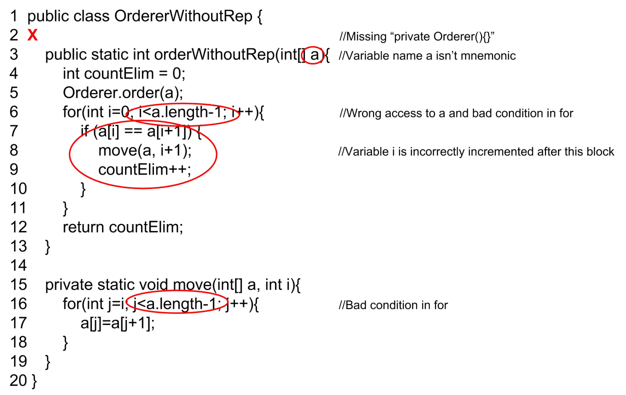
The method has two loops, the outer one starts at line 4. Line 15 checks the value of `swapped` and in the case that it is cero (no changes has been made in the inner loop) the method returns. This can be avoided by adding the condition to the outer `for`. This defect is particular because it could not be considered a defect on its own.

Defect G - NF

In line 6 the variable `find` is defined and it is never used in the program.

B. OrdererWithoutRep Defects

Here we present the defects in the ***OrdererWithoutRep*** class. Figure 4 shows the defects in the code with a ellipse around them. The following is the description of each defect.



```
1  public class OrdererWithoutRep {
2  X                                                //Missing "private Orderer(){}"
3    public static int orderWithoutRep(int[] a){    //Variable name a isn't mnemonic
4      int countElim = 0;
5      Orderer.order(a);
6      for(int i=0; i<a.length-1; i++){             //Wrong access to a and bad condition in for
7        if (a[i] == a[i+1]) {
8          move(a, i+1);                            //Variable i is incorrectly incremented after this block
9          countElim++;
10       }
11     }
12     return countElim;
13   }
14
15   private static void move(int[] a, int i){
16     for(int j=i; j<a.length-1; j++){             //Bad condition in for
17       a[j]=a[j+1];
18     }
19   }
20 }
```

Figure 4. Defects of the OrdererWithoutRep Class

Defect a - PF

The defect is similar to defect A.

Defect b - PF

After calling the method `move` the index `i` is incremented in one, this produces a failure if there are more than 2 equal integers in the array because some of them are not considered. An execution showing this defect, defect d and their associated failures are shown in Figure 5. The interrogation mark in the expected result means that the value in this position does not matter. This defect can be removed in different ways. An easy but not very good one is to decrement `i` after calling the `move` method. A better solution is to add a loop until a different integer appears.

```
a = [1, 3, 3, 3, 5]
Expected Result = [1, 3, 5,  ?, ?]    Return 2

i = 0, a[0] !=  a[1]  =>  a = [1, 3, 3, 3, 5],  countElim = 0
i = 1, a[1] == a[2]  =>  a = [1, 3, 3, 5, 5], countElim = 1  //defect b execution
i = 2, a[2] !=  a[3]  =>  a = [1, 3, 3, 5, 5], countElim = 1  //failure due to b
i = 3, a[3] == a[4]  =>  a = [1, 3, 3, 5, 5], countElim = 2  //failure due to d
i = 4 => the method finish

Defect d's failure makes the method returns the right countElim value

Obtained result = [1, 3, 3, 5, 5]      Return 2
```

Figure 5. An Execution Showing Defects b and d Occurrences

Defect c - NF

The defect is similar to defect C.

Defect d - PF

When equal elements are found the `move` method is executed, and it leaves repeated elements at the end of the array as a side effect. These last elements are of no importance and the specification is clear about this. However, these repeated elements are considered as equal elements causing an incorrect result in the method ***orderWithoutRep***. This defect can be removed by changing the line 6 of the method from:

```
for(int i=0; i<a.length-1; i++){
```

to

```
for(int i=0; i<a.length-1-countElim; i++){
```

Figure 5 shows this defect and defect b causing failures during execution. The failure due to defect d fixes the counter of equal elements. Another example showing a failure in both results (the `a` array and the counter) is shown in Figure 6.

```
a = [1, 3, 3, 3, 3, 5]
Expected Result = [1, 3, 5,  ?, ?, ?]    Return 2

i = 0, a[0] !=  a[1]  =>  a = [1, 3, 3, 3, 3, 5],  countElim = 0
i = 1, a[1] == a[2]  =>  a = [1, 3, 3, 3, 5, 5],  countElim = 1  //failure due to b
i = 2, a[2] == a[3]  =>  a = [1, 3, 3, 5, 5, 5],  countElim = 2  //failure due to b
i = 3, a[3] == a[4]  =>  a = [1, 3, 3, 5, 5, 5],  countElim = 3  //failure due to d
i = 4, a[4] == a[5]  =>  a = [1, 3, 3, 5, 5, 5],  countElim = 4  //failure due to d
i = 5 => the method finish

Obtained result = [1, 3, 3, 5, 5, 5]       Return 4
```

Figure 6. Another Execution Showing Defects b and d Occurrences

Defect e - NF

The defect is similar to defect E.

Defect f - NF

The `move` method has a `for` in line 16 that traverses the array from an initial position (the one that is passed to the method) to the final position. It is not necessary to traverse the array until the final position because at the end there are elements that should not be considered. This is a defect that affects the performance but not the results. To optimize the method the line mentioned can be sustituted with the next one:

```
for(int j=i; j<a.length-1-countElim; j++){
```

The variable `countElim` must be passed to the method.

Table I presents the quantity of defects discriminated by class, type of defect (PF, NF) and the totals.

Table I
QUANTITY OF DEFECTS BY CLASS, TYPE AND TOTAL

Class/Defect Type	PF	NF	Total
Orderer	2	5	7
OrdererWithoutRep	3	3	6
Total	5	8	13

V. RESULTS

A group of 17 undergraduate students participated of the testing experience. These were students in the fourth year of the Computer Engineering career at the Facultad de Ingeniera of the Universidad de la Repblica. Every student applied only one testing technique in the program. We divided the group as follows: 3 students applied desktop inspections, 4 students applied MCC, 3 students applied LIP, 4 students applied EP and 3 students applied DT.

The results of the testing experience are presented in Table II. The rows present the defects that the participants detected, the total number of defects detected by the participants and the time in minutes that the application of the technique took them. For inspection technique the defect found by a participant is marked with an "X". For the other techniques appears the time in minutes that takes to find the defect after a test case fails. The time (last column) means different things depending on the technique. For inspection technique it is the time that takes executing the technique, while for the dynamics techniques it is the time that takes designing and programming the test cases. Also there are rows that show subtotals by technique and the last row presents the sum total.

The results associated to the LIP technique are strange in the sense that the 3 participants that use this technique discover only one and the same defect. From a theoretical point of view of the technique we know that this result is not "correct". Different things can cause this situation, for example, the LIP technique was not understood by the participants. This and other causes are beyond the scope of this paper. Due to the described situation the LIP technique is not considered for the analysis of the results.

It is not possible to have strong or statistically valid conclusions with this training. Our intention is to make a preliminary analysis of the results. After the formal experiment finishes we can validate (or not) the preliminary results of this training phase.

In the next subsections we briefly discuss the cost, the effectiveness, and the efficiency of every technique. The last subsection presents a discussion on the IBM and Beizer taxonomies.

A. The Cost

We consider the time that the execution of each technique takes as a measure of its cost for the program. The cost of the inspection technique is the time that takes the inspection. The cost of the dynamic techniques is the time that takes designing the test cases plus the time in detecting defects after a failure. We consider that the time that takes executing the Junit test cases is cero.

The cost of the techniques varies greatly from one tester to another and from technique to technique. These variations could have different explanations. We need to gather more data to make interesting and valid conclusions about the variations. We are not sure if this variations depend on the wrong or right application of the technique or are due to natural differences in humans beings. However, we can still make a general analysis of the cost of each technique.

Table II
DEFECTS DETECTED AND TIME EXPENDED

Tec.			Def.	A	B	C	D	E	F	G	a	b	c	d	e	f	TP	t
Static		Inspection		X	X						X						3	80
				X					X		X						3	300
						X		X	X				X	X	X		6	240
			Tot	2	1	1	0	1	2	0	2	0	1	1	1	0	12	
Static	White box	MCC		0.08	0.5					0.17		60					4	300
					6									10			2	90
					5												1	20
					5									10			2	90
			Tot	1	4	0	0	0	0	1	0	1	0	2	0	0	9	
		LIP			5												1	330
					X												1	260
					10												1	150
			Tot	0	3	0	0	0	0	0	0	0	0	0	0	0	3	
	Black box	EP		1	5						1	15		10			5	102
					0						0						2	360
				5	12									15			3	210
				1	1						1			5			4	120
			Tot	3	4	0	0	0	0	0	3	1	0	3	0	0	14	
		DT		15	7						0			0			4	440
				1	3						1						3	150
				X	X						X	X		X			5	350
			Tot	3	3	0	0	0	0	0	3	1	0	2	0	0	12	
			ST	9	15	1	0	1	2	1	8	3	1	8	1	0	50	

The costs of inspection techniques for the 3 testers that applied the techniques are: 80, 240 and 300 minutes. The average cost of the inspection is 207 minutes (3.5 hours approximately). Considering that the program has a total of 37 locs we can conclude that doing desktop inspection in Java is really expensive; an average of 5.5 minutes/loc.

The costs of the MCC have the same variability. The minimum time elapsed is 20 minutes and the maximum is 300 minutes. The average cost of MCC is 125 minutes (2 hours approximately). Here we have to consider the conditions that appear in the code as a measure of the complexity to generate the test cases. Class *Orderer* has two nested `for` with an `if` in the inner one and an `if` in the other. Each decision has only one condition in these sentences. Class *OrdererWithoutRep* has two nested loops too; we are considering the `for` in the private method `move`. In terms of nesting and decisions these classes can be considered from low to normal complexity. Again, testing the two classes in 2 hours seems to be rather expensive.

The average costs of the two specification based techniques are: 198 minutes (3.3 hours) for EP and 313 minutes (5.2 hours approximately) for DT. Considering that the functionality of the problem is really simple (ordering an array) we can conclude that testing a program with this techniques is expensive too.

After a test case fails the tester searches for the defect that produces it. The time varies a lot from tester to tester in finding the same defect. We need further data to make conclusions about the time that is needed to find different types of defects in unit testing.

B. The Effectiveness

We measure the effectiveness of a technique as the defects it founds. Table III shows the effectiveness of each technique for each defect and the total effectiveness for each defect.

As it was mentioned before this is just the training phase of the experiment so we do not analyze the results in a statistical way. Nevertheless, we present some results that can be refuted by experiments later.

PF defects are more easily to find than NF defects. The PF defects are A, B, a, b and d. Defects A, B, a and d have more than 50% detection effectiveness, defect b has a 21% detection effectiveness. The other defects have less detection effectiveness.

Inspections detect a greater variety of defects than the other techniques. Among the 3 testers applying the inspection 9 out of the 13 defects are found. MCC and EP discover 5 defects with 4 testers and DT discovers 4 different defects with 3 testers.

Dynamic techniques have problems in founding NF defects. Using EP and DT the participants did not discover any of the NP defects. With MCC only one NP defect is found: G. The reason for this is that dynamic techniques are based in the execution of the program. So defects that do not produce a failure are never sought directly. However, when a test case fails the tester reviews the code to find the defect that produce the failure, in this review the tester can find other defects, including a NF defect.

Table III
EFFECTIVENESS BY TECHNIQUE IN PERCENTAGE

Def. / Tec.	A	B	C	D	E	F	G	a	b	c	d	e	f
Insp.	67	33	33	0	33	67	0	67	0	33	33	33	0
MCC	25	100	0	0	0	0	25	0	25	0	50	0	0
EP	75	100	0	0	0	0	0	75	25	0	75	0	0
DT	100	100	0	0	0	0	0	100	33	0	67	0	0
All	64	86	7	0	7	14	7	57	21	7	57	7	0

Performance defects are not easily found. Defect f is the only performance defect of the program and it was not detected during the experience.

Effectiveness is really low. The effectiveness of each technique can be calculated as the number of defects found divided by the number of total defects. This simple experience presents the following effectiveness: inspections 31% (12/37), MCC 17% (9/52), EP 27% (14/52) and DT 31% (12/39). Considering the effectiveness of the "team", we have that 14 people verifying a really small program (37 locs) only found 11 defects on a total of 13.

C. The Efficiency

We calculate the efficiency as: Defects Found / Cost. We use the average of defects found by technique. The efficiency of each technique is: 1.16 defects/hour for desktop inspections, 1.08 defects/hour for MCC, 1.06 defects/hour for EP and 0.77 defects/hour for DT. We can conclude that it takes too much time to find a defect, for every technique the cost is about an hour or more.

D. The Defect's Classification

The testers had to classify each defect in IBM and Beizer taxonomies. Tables IV and V present the IBM and Beizer classifications obtained in the experience of each defect. In this case is considered the LIP technique. In both tables are presented only those defects that have been found more than once.

Table IV
CLASSIFICATION WITH IBM

Type	Qualifier	Defect					
		A	B	F	a	b	d
Checking	Missing	8			8		
Checking	Incorrect						6
Assing/Init.	Incorrect	1	15				
Assing/Init.	Missing						1
Algorithm/Method	Incorrect			2			1
#Found		9	15	2	8	3	8
Different Clasif.		2	1	1	1	1	3

It is clear that IBM classification works better for the testers than Beizer classification. At the time of classifying with IBM, the testers normally classified the same defect under the same type. On the other hand, when it comes to Beizer classification it is completely the opposite. Thus,

Table V
CLASSIFICATION WITH BEIZER

Defect Type	Defect					
	A	B	F	a	b	d
2.1.1		2			1	1
2.4.1	1			1		
2.4.3				1		1
2.6.	1					
3.1.1					1	
3.1.2	1					
3.1.4	1		2	1		2
3.2.1	1			1	1	3
3.2.2	1	1		1		
3.2.3		6				
4.1.3		5				
4.2.2	2			2		1
4.2.3		1				
4.2.4	1			1		
#Found	9	15	2	8	3	8
Different Clasif.	8	5	2	7	3	5

Beizer could be a good taxonomy for presenting the effectiveness by defect type but it seems rather complicated to testers or developers. IBM seems to be easier to use.

We conclude that it is better if the researchers classify the defects instead of the testers.

VI. CONCLUSIONS

We present a Java program with its defects and a training experience during an experiment. The training consists of 17 students applying 5 testing techniques, desktop inspection, equivalence partitioning and boundary-value analysis, decision table, linearly independent path, and multiple condition coverage.

We show the results on the effectiveness and cost of these techniques. When the formal experiment ends we will be able to analyze these results with more data.

ACKNOWLEDGMENT

The authors would like to thank Lorena Ibaceta for her helpful comments on this article.

REFERENCES

[1] A. Moreno, F. Shull, N. Juristo, and S. Vegas, "A look at 25 years of data," *IEEE Software*, vol. 26, no. 1, pp. 15–17, Jan.–Feb. 2009.

[2] P. G. Frankl and S. N. Weiss, "An experimental comparison of the effectiveness of branch testing and data flow testing," *IEEE Transactions on Software Engineering*, vol. 19, no. 8, pp. 774–787, Aug. 1993.

[3] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria," in *Proc. ICSE-16. th International Conference on Software Engineering*, 16–21 May 1994, pp. 191–200.

[4] P. G. Frankl and O. Iakounenko, "Further empirical studies of test effectiveness," *ACM SIGSOFT Software Engineering Notes*, vol. 23, no. 6, pp. 153–162, November 1998.

[5] G. Myers J., "A controlled experiment in program testing and code walkthroughs/inspections," *Communications of the ACM*, vol. 21, no. 9, pp. 760–768, September 1978.

[6] V. R. Basili and R. W. Selby, "Comparing the effectiveness of software testing strategies," *IEEE Transactions on Software Engineering*, vol. 13, no. 12, pp. 1278–1296, Dec. 1987.

[7] E. Kamsties and C. M. Lott, "An empirical evaluation of three defect-detection techniques," in *Proceedings of the Fifth European Software Engineering Conference*, 1995, pp. 362–383.

[8] M. Wood, M. Roper, A. Brooks, and J. Miller, "Comparing and combining software defect detection techniques: a replicated empirical study," *ACM SIGSOFT Software Engineering Notes*, vol. 22, no. 6, pp. 262–277, 1997.

[9] A. P. Mathur and W. E. Wong, "Fault detection effectiveness of mutation and data flow testing," *Software Quality Journal*, vol. 4, pp. 69–83, 1995.

[10] P. G. Frankl, S. N. Weiss, and C. Hu, "All-uses versus mutation testing: An experimental comparison of effectiveness," *The Journal of Systems and Software*, vol. 38, pp. 235–253, 1997.

[11] R. Chillarege, *Handbook of Software Reliability Engineering - Chapter 9*. Mcgraw-Hill, April 1996, ch. 9: Orthogonal Defect Classification.

[12] B. Beizer, *Software Testing Techniques*, 2nd ed. Van Nostrand Reinhold, June 1990.

[13] W. Humphrey, *PSP(sm): A Self-Improvement Process for Software Engineers*. Addison-Wesley Professional, March 2005.

[14] ——, *A Discipline for Software Engineering*. Addison-Wesley Professional, January 1995.

[15] K. Beck and C. Andres, *Extreme Programming Explained: Embrace Change*, 2nd ed. Addison-Wesley Professional, November 2004.

[16] IEEE/ACM, *Software Engineering Body of Knowledge: Iron Man Version*, May 2004.

[17] N. Juristo, A. Moreno, S. Vegas, and M. Solari, "In search of what we experimentally know about unit testing," *IEEE Software*, vol. 23, no. 6, pp. 72–80, November 2006.