Effectiveness of Five Verification Techniques

Diego Vallespir, Cecilia Apa, Stephanie De León, Rosana Robaina
Instituto de Computación, Facultad de Ingeniería
Universidad de la República,
Montevideo, Uruguay

Juliana Herbert
Herbert Consulting
Porto Alegre, RS, Brazil

***Abstract*—Here we present a formal experiment in order to know the effectiveness of 5 different unit testing techniques. The techniques are: desktop inspection, equivalence partitioning and boundary-value analysis, decision table, linearly independent path, and multiple condition coverage. The proposed design is a one factor with multiple levels one. This is the first formal experiment we know about that uses decision table, linearly independent path and multiple condition coverage techniques. The experiment is executed by 14 testers that apply the techniques in 4 different programs developed especially for this experiment. The statistical results show that decision table and equivalence partitioning techniques are more effective than linearly independent path technique.**

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

I. Introduction

It is quite common to use a software testing technique to verify a software unit, but choosing one can be very intricate.

In order to do so in a simpler way, we must know several things beforehand, 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.

Many formal experiments to analyze the behavior of some unit testing techniques were conducted. The first we know about is from 1978 [1]. Despite we have several years of empiric research on the matter, we do not have definite results yet.

In [2] the authors examine different experiments on software testing: [3], [4], [5], [1], [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 of all program statements, although the functional approach takes longer.
  - Functional testing is as effective as conditional testing and takes 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 the following two problems:
  - They are small and unusually simple.
  - The defects are seeded by the researches instead of looking for naturally occurring ones.

The authors belive that researchers should publish more information not only about the number of faults the technique can remove but also about the types.

We agree with Moreno in the sense that the programs used in these experiments are unreal, so we decided to make an experiment with more real programs attacking both mentioned problems.

Before an experiment begins, the testers (the subjects who execute the testing techniques) need previous training. In our experiment this includes a course on every technique to be applied, a course on the scripts to be used during execution, and a training execution of the techniques in a simple program. The results of this training execution are presented in [11].

Our experiment uses 4 programs that are especially built for the experiment. The programs are of different types. We propose a balanced design in which 14 testers test the programs. Every tester except one tests 3 of the 4 programs, each program with a different technique. Every program is tested twice with each technique. Having 5 testing techniques results on each program being tested 10 times by 10 different testers. In order to compensate for the effect of the learning on testing, the design changes the order in which the techniques are applied by the different testers.

The techniques used in the experiment are: desktop inspection, equivalence partitioning and boundary-value analysis, decision table, linearly independent path, and multiple condition coverage. This is the first formal experiment we know about that uses decision table, linearly independent path and multiple condition coverage techniques. The statistical results show that decision table and equivalence partitioning techniques are more effective than linearly independent path technique.

The article is organized as follows. Section II briefly describes the techniques used in the experiment. In section III the taxonomies used to classify the defects are presented. The testers follow specific scripts that provide them with them

with guidance which are presented in section IV. Section V focuses on the Java programs that are verified by the testers. The experiment design is presented in section VI. In section VII the experiment execution is presented. The results obtained are presented in section VIII and the conclusions in section IX.

II. The Techniques

We use the same terminology for the verification techniques as Swebok [12]. The techniques can be divided into different types: static, tester intuition or experience, specification based, code based, fault based and usage based. At the same time code-based is divided into control flow and data flow based criteria.

In our experiment we used 5 testing techniques: desk-top 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 techniques types as well as control-flow ones are covered. Usually equivalence partitioning and boundary-value analysis are considered as two separate techniques. Given they are generally used together, they are considered as one technique in the experiment.

The inspection technique consists of examining the code to find defects. A check-list is used in order to formalize the inspection. The tester considers the items in the check-list one by one and checks the code to find a defect associated with the current item. The check-list used in our experiment is presented in the Appendix.

The EP and DT techniques are specification-based ones. Those techniques divide the entrance domain of a program into classes based on the specified behavior of the program, unit or system under test. The techniques are different but the main concept is similar: to divide the entrance into "meta-test cases" and then choose one test case for each meta-test case. In order to develop the test cases the tester only uses the specification of the program. In other words, the tester does not use the code of the program during test development. This is the reason why this kind of techniques is also called black-box testing techniques. The tester executes functions of the "black-box" and compares the expected result of the test case with the result obtained in the test case execution.

The LIP and MCC techniques are control flow based techniques. The LIP technique divides the control flow of the program into linearly independent paths. The tester analyzes the code to find those paths. After this step, the tester develops a set of test data, which ensures the execution of the linearly independent paths previously found. Once the tester gets the set of test data, he reads the specification to add the expected result to each test data, obtaining with this method the test cases.

The MCC technique criteria determines that every condition combinations of each decisions of the code must be executed with the set of test cases. Using only the source code, the tester first developed the set of test data that covers the criteria. After that, the proceeding is the same as in LIP technique, which is to add the expected results to the set of test data using the specification of the program.

These techniques are defined in several basic software engineering books and basic software testing books, a complete explanation of them is beyond the scope of this paper.

We could not find literature describing experiments in which DT, LIP and MCC techniques are applied, neither could Juristo [13]. Given so, our experiment and the corresponding empiric results, are the first ones that involve these techniques.

III. Defect Taxonomies

Since our objective is to find the effectiveness of the techniques according to defect types, a defect taxonomy is necessary. Various defect taxonomies are presented in the literature. The IBM Orthogonal Defect Classification (ODC) is the most used [14]. Other taxonomy of interest is Beizer's Defect Taxonomy [15].

ODC allows the defects to be classified in many orthogonal views: defect removal activities, triggers, impact, target, defect type, qualifier, age and source. In this experiment only the defect type and the qualifier are considered. 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. Therefore, every defect must be classified in both views; for example, a defect could be classified as "timing/serial incorrect".

Beizer's taxonomy is hierarchical meaning that each category is divided into sub-categories and so on. For example, the category number 3 is "Structural bugs" and is divided into 3.1 "Control flow and sequencing" and 3.2 "Processing". Category 3.1 is divided into other several sub-categories. This taxonomy presents a lot of different defect types, 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.

We are developing and executing experiments at the same time that we are conducting a research on defect taxonomy. We are not convinced that Beizer's taxonomy or ODC are the best taxonomies for unit defects. Some initial results are presented in [16].

IV. Scripts

The testers follow scripts that provide them with guidance, which allows them to execute the technique and correctly collect and record the data required for the experiment. There are 3 scripts, one for each type of technique used: static, specification-based and control-based. The static script consists of three phases: preparation, execution and finish. The specification-based and control based scripts have four phases: preparation, design, execution and finish.

After the **preparation phase** the tester is ready to start the verification job. In this phase the tester must prepare the testing environment and all the necessary material in order to start the testing. This phase consists of the following steps:

- Download the files related to the program to be verified: specification, design, javadoc and the source code.

- Record the starting date and hour of the work.
- Read and understand the functional specification of the program.
- Prepare the environment for the testing (only applies for the dynamic techniques).

In the **design phase** (only for dynamic techniques) the tester develops test cases that achieve the testing technique criteria. This phase is based on the following steps:

- Design of test cases that satisfy the technique.
- Codify the test cases in JUnit.
- Record the total time spent in designing and codifying the test cases.
- If some defects are found during design, the tester must register them. The detection time in this case is zero.

During **execution**, the test cases are executed (or the inspection is executed) and the tester searches for the defects of those cases that fail. According to this phase, the steps are the following:

- Execution of the test cases or execution of the inspection.
- Record the defects that are found during the testing. In the case of inspection technique the time to find the defect is always zero.

In the last phase, the finish, the tester ends the job and records the finishing date and hour. In every phase the tester has to register the time elapsed during the activities and every defect found.

V. The Programs

We use 4 programs in the experiment, each of which is developed especially for this experiment. These programs differ from the ones found in the experiment literature in two aspects. First, the defects in the programs are not injected by the researchers. Second, the programs are more real and more complex. Given we are considering unit testing instead of system testing, the programs are real enough for our experiment.

These are different types of programs and all are codified in Java. We classify them as

- Accountancy with data base (Accountancy).
- Mathematic.
- Text processor (Parser).
- Document creation from data in a data base (Doc DB).

The accountancy program is a small salary liquidation program. The program has the following functionalities:

- Add, modify and delete an employee. The employees have a position in the organization and hours worked per week.
- Add, modify and delete positions in the organization. The positions have a base salary.
- Increase the salary (in different forms) of a position.
- Calculate the salary liquidation.

The database used for this program is HSQLDB[1].

[1] http://hsqldb.org/

TABLE I
Measures of the programs

Program	LOCs	Meth. LOCs	#Cl.	#Met.	#Def.
accountancy	1979	1497	14	153	107
Mathematic	468	375	13	29	50
Parser	828	634	10	64	272
Doc DB	566	362	10	61	32

The mathematic program receives two arrays of real numbers of the same size: $x_1 \dots x_n$ and $y_1 \dots y_n$, and a real number x_k. The program calculates the following items:

1) The mathematical correlation between the arrays. The correlation determines the relationship between two ordered sets of numbers.
2) The significance of the correlation.
3) The parameters of the linear regression, β_0 and β_1, for the pairs of numbers of the form $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$. The linear regression is a way of approximating a straight line to a set of points. The equation of the straight line is: $y = \beta_0 + \beta_1 x$.
4) The result y_k of using x_k with the straight line equation: $y_k = \beta_0 + \beta_1 x_k$.
5) The prediction interval of 70% for the value x_k.

The parser program is a small text processor. The parser recognizes a small set of Pascal language. The program receives a file with a program written in Pascal. It parses the code and produces a XML file that presents the structures that are recognized in the code.

The doc db program generates a multiple choice exam from data in a data base. The questions in the exam are chosen at random. The document is a Latex one that contains the questions, the possible answers and the correct one. The data base used for this program is HSQLDB.

Table I shows some measures for the four programs. The columns are the locs without comments, executable locs without comments (method locs), the number of classes, the number of methods and the number of defects in the program.

We developed a framework to compare formal experiments and we used it to compare four experiments [17]. Unfortunately, some experiments are not described enough in the articles in which are presented. However, we can compare locs and defects. In [6], Basili and Selby (BS) used four programs. The smallest of 145 locs and the biggest of 365 locs. The programs have 34 defects in total. In [18], Macdonald and Miller (MM) used two programs. One of 143 locs and the other of 147. Each program has 12 defects. In [19], Juristo and Vegas (JV) used four programs, each of them with 200 locs and 9 defects.

All these experiments have defects injected by the researchers. In BS some defects are injected while some are not. In JV all the defects are injected. In MM it is not clear how many of the defects are injected.

Our programs are considerably bigger and more complex. Actually, we do not inject defects on them, so the defects are those introduced during the development. Therefore, our

programs are more real.

As an example, Figure 1 presents the design of the Mathematic program. The name of the classes and methods are in Spanish.

Given we do not inject the defects we must define a way to find them. Some individuals from our research group performed testing on the programs and recorded the defects found. During the experiment 14 students tested the programs with different techniques. We consider that the union of the sets of the defects found are an excellent estimation of the defects of the programs.

VI. Experiment Design

The objective of the experiment is to study 5 testing techniques in order to evaluate their effectiveness and cost when used in unit testing. The techniques are: desktop inspection, equivalence partitioning and boundary-value analysis, decision table, linearly independent path, and multiple condition coverage. The effectiveness is defined as the percentage of defects found by a technique. The cost is the time that takes to execute a technique.

The type of our experiment design is **one factor** with multiple **levels**. The **factor** is testing technique and the **levels** are the 5 different techniques. There are 14 **subjects** (the testers) and 4 **experimental units** (the programs). The **independent (response) variables** are: the defects found and the time elapsed during the technique execution.

Table II shows the design of the experiment. For each of the four programs the table presents the testers who test these programs and the techniques used. Represented by numbers from 1 to 3, is the order in which a tester tests these programs; 1 is the first program tested by the tester and 3 corresponds to the last one. For example, tester number one tests the accountable program with inspection technique first, then tests the mathematic program with the MCC technique and the last program he tests is the data base program with the LIP technique.

The design has the following characteristics:

- It is a balanced design.
- Each technique is used 8 times.
- Each program is tested 10 times.
- Each program is tested 2 times for each technique.
- Each technique is applied 8 times.
- Every tester except one tests 3 different programs with 3 different techniques.
- Only one technique is used in a program by a tester. Therefore, the testers never test the same program twice, which avoids the learning of the defects on the program.
- In order to compensate for the effect of learning on testing, the design changes the order in which the techniques are applied by different testers.
- The assignment of the set of techniques and programs previously defined in the design to the testers is completely at random.

The subjects are students in the 4th and 5th year of the Computer Engineering career thus we consider them to have an equal experience in testing.

VII. Experiment Execution

The execution consists of three phases: courses phase, training phase and test phase. The courses phase and the training phase prepare the testers to execute the techniques and the scripts correctly. In the test phase the design of the experiment is executed.

The testers participate in 7 different courses during the courses phase. Every course takes around 2 hours of class. The courses present the techniques to be used, the scripts and the tool to record the data.

The training phase is a small experiment on its own [11]. In this phase the testers test a small program and record the data in the same way they will do in the test phase. This serves to assure they are executing the techniques correctly and that they are recording the data as expected.

The last phase is the execution of the design. The testers get the program one at a time. When a tester finishes the execution of a technique in a program the researchers gives him another program to test.

During execution some testers abandon the experiment. This clearly impacts on the design and some properties described are not longer valid. For example, various properties of the design, mainly regarding the balance, do not hold. However, we can do a statistical analysis of the data.

VIII. Analysis of the Results

Due to the execution problems that arose, we have different number of samples for each technique. Table III shows every unitary experiment executed in the experiment. Each row in the table shows the defects found and the effectiveness of a tester testing a program with a technique. The total defects of each program was presented in table I. Remember that we define effectiveness as the percentage of defects founds divided by the total defects.

Table IV presents the effectiveness grouped by technique. This data is used for the descriptive statistics and for the hypothesis testing.

Table V shows the observations quantity, average and standard deviation of the effectiveness of each technique. It seems like DT and Insp technique are more effective than the rest of the techniques while LIP technique is the less effective. The standard deviation could be considered as high so it is probably that we need more observations. This can be obtained by replication of the experiment.

Due to the few observations we have, we can not make an analysis about the effectiveness by defect type. Therefore, it is only presented de total effectiveness of each technique. The null hypothesis (H_0) is that every technique has the same effectiveness. The alternative hypothesis (H_1) is that at least exists a technique with different effectiveness from the others.

Due to the few observations it is not very reliable to apply a parametric test. So, we decide to use a non parametric test: Mann-Whitney test. We compare all the couples of techniques.

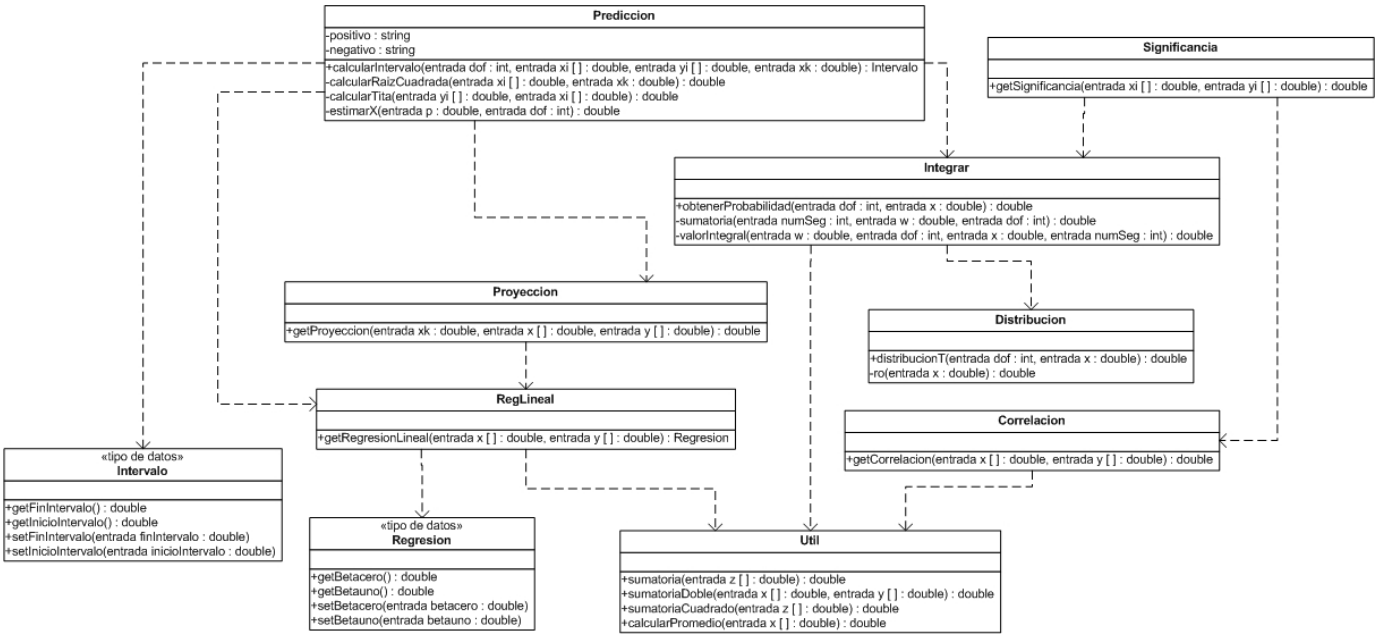

Fig. 1. Design of the Mathematic Program

TABLE II
THE EXPERIMENT DESIGN

	Contabilidad					Matemtico					MO-Latex					Parser				
	Ins	CCCM	TLI	PEyAVL	TD	Ins	CCCM	TLI	PEyAVL	TD	Ins	CCCM	TLI	PEyAVL	TD	Ins	CCCM	TLI	PEyAVL	TD
tester 1	1						2						3							
tester 2					2	3													1	
tester 3				3								1						2		
tester 4										1	2						3			
tester 5			1						2						3					
tester 6		2						3								1				
tester 7	3													1						2
tester 8							1						2						3	
tester 9					1	2						3								
tester 10				2						3								1		
tester 11			3								1						2			
tester 12									1						2	3				
tester 13		1						2						3						
tester 14																				1

This gives 10 different null and alternative hypothesis. For example, one is the EP and LIP hypothesis:

$$H_{0_{EP-LIP}} : \mu_{EP} = \mu_{LIP} \quad (1)$$

$$H_{1_{EP-LIP}} : \mu_{EP} \neq \mu_{LIP} \quad (2)$$

The results of the Mann-Whitney test are presented in table VI. The columns presents: the techniques to compare, the statistical U, the the quantities of observation for both techniques that are being compared and the probability associated.

This results shows that we can only reject two null hypotheses:

$$H_{0_{LIP-DT}} : \mu_{LIP} = \mu_{DT}.$$

$$H_{0_{LIP-EP}} : \mu_{LIP} = \mu_{EP}.$$

The rejection of the LIP-DT hypothesis is with an α of 1,1% and the rejection of the LIP-EP is with an α of 9%. We can conclude that for our programs it seems that using EP technique is more effective than using LIP technique and that using DT technique is also more effective than using LIP technique. This has to be validated with replications of this experiment or by executing other different experiments.

IX. CONCLUSIONS

We present a formal experiment to evaluate testing technique. One of the contributions is to have 4 programs that are specially designed for these experiments. These programs are bigger than other used in similar experiments and the defects

TABLE III
EFFECTIVENESS FOR EACH UNITARY EXPERIMENT

Program	Technique	# Def. Found	% Effectiveness
Accountancy	MCC	3	2.80
Accountancy	MCC	7	6.54
Accountancy	DT	44	41.12
Accountancy	DT	9	8.41
Accountancy	EP	21	19.63
Accountancy	Insp	5	4.67
Accountancy	Insp	7	6.54
Accountancy	LIP	8	7.48
Mathematic	MCC	5	10
Mathematic	MCC	3	6
Mathematic	DT	4	8
Mathematic	EP	12	24
Mathematic	EP	7	14
Mathematic	Insp	8	16
Mathematic	Insp	23	46
Mathematic	LIP	4	8
Mathematic	LIP	4	8
Parser	MCC	116	42.65
Parser	MCC	42	15.44
Parser	DT	77	28.31
Parser	DT	32	11.76
Parser	EP	41	15.07
Parser	EP	5	1.84
Parser	Insp	46	16.91
Parser	Insp	10	3.68
Parser	LIP	5	1.84
Doc DB	MCC	1	3.13
Doc DB	MCC	12	37.5
Doc DB	DT	6	18.75
Doc DB	DT	8	25
Doc DB	EP	4	12.5
Doc DB	EP	1	3.13
Doc DB	Insp	0	0
Doc DB	Insp	19	59.38
Doc DB	LIP	3	9.38
Doc DB	LIP	3	9.38

TABLE IV
EFFECTIVENESS GROUPED BY TECHNIQUE

MCC	Insp	EP	DT	LIP
3.13	0	12.5	18.75	9.38
37.5	59.38	3.13	25	9.38
2.80	6.54	19.63	41.12	7.48
6.54	4.67	14	8.41	8
10	16	24	8	8
6	46	15.07	28.31	1.84
42.65	16.91	1.84	11.76	-
15.44	-	-	3.68	-

TABLE V
AVERAGE AND STANDARD DEVIATION OF THE EFFECTIVENESS

	MCC	Insp	EP	DT	LIP
Observations Quantity	8	8	7	7	6
Average	15.51	19.15	12.88	20.19	7.35
Standard Deviation	15.75	21.81	8.09	12.16	2.81

are not injected by the researches. These characteristic are the ones that Juristo et al complain about.

Although the execution of the experiment differs from the initial balanced design, we consider that we obtain two interesting results. The first one is that this work presents the first formal experiment that uses DT, LIP and MCC testing techniques. The second one is that we can reject two null hypotheses. This means that it seems like DT and EP are more effective than LIP. Further experiments are needed to validate these results.

TABLE VI
MANN-WHITNEY TEST

	U Mann-Whitney	$(n_1;n_2)$	P(U>x)
DT vs. MCC	18.0	(7;8)	0.140
EP vs. MCC	27.5	(7;8)	0.522
Insp vs. EP	26.0	(8;7)	0.389
Insp vs. MCC	28.5	(8;8)	0.360
LIP vs. MCC	20.0	(6;8)	0.331
LIP vs: Insp	21.0	(6;8)	0.377
EP vs. DT	17.0	(7;7)	0.191
Insp vs. DT	20.0	(8;7)	0.198
LIP vs. DT	5.0	(6;7)	0.011
LIP vs. EP	10.5	(6;7)	0.090

As future work we pretend to design another experiment using different techniques. One is running at this moment with sentence coverage and all uses techniques. We also want to run other experiment with more testers so we can get more observations, with these observations maybe we can reject more effectiveness hypotheses, make some conclusions about the cost of the techniques and make hypotheses that evaluate effectiveness considering the different types of defects.

Acknowledgment
The authors would like to thank Lorena Ibaceta and Fernanda Grazioli for their helpful comments on this article. This work is partially supported by the Programa de Desarrollo de las Ciencias Básicas (PEDECIBA), Uruguay.

APPENDIX

1)

- Are descriptive variable and constant names used in accord with naming conventions?
- Are there variables or attributes with confusingly similar names?
- Is every variable and attribute correctly typed?
- Is every variable and attribute properly initialized?
- Could any non-local variables be made local?
- Are all for-loop control variables declared in the loop header?
- Are there literal constants that should be named constants?
- Are there variables or attributes that should be constants?
- Are there attributes that should be local variables?
- Do all attributes have appropriate access modifiers (private, protected, public)?
- Are there static attributes that should be non-static or vice-versa?

2)

- Are descriptive method names used in accord with naming conventions?

- Is every method parameter value checked before being used?
- For every method: Does it return the correct value at every method return point?
- Do all methods have appropriate access modifiers (private, protected, public)?
- Are there static methods that should be non-static or vice-versa?

3)

- Does each class have appropriate constructors and destructors?
- Do any subclasses have common members that should be in the superclass?
- Can the class inheritance hierarchy be simplified?

4)

- For every array reference: Is each subscript value within the defined bounds?
- For every object or array reference: Is the value certain to be non-null?

5)

- Are there any computations with mixed data types?
- Is overflow or underflow possible during a computation?
- For each expressions with more than one operator: Are the assumptions about order of evaluation and precedence correct?
- Are parentheses used to avoid ambiguity?

6)

- For every boolean test: Is the correct condition checked? Is each boolean expression correct?
- Are the comparison operators correct?
- Has each boolean expression been simplified by driving negations inward?
- Are there improper and unnoticed side-effects of a comparison?
- Has an "&" inadvertently been interchanged with a "&&" or a "|" for a "||"?

7)

- For each loop: Is the best choice of looping constructs used?
- Will all loops terminate?
- When there are multiple exits from a loop, is each exit necessary and handled properly?
- Does each switch statement have a default case?
- Are missing switch case break statements correct and marked with a comment?
- Do named break statements send control to the right place?
- Is the nesting of loops and branches too deep, and is it correct?
- Can any nested if statements be converted into a switch statement?
- Are null bodied control structures correct and marked with braces or comments?
- Are all exceptions handled appropriately?
- Does every method terminate?

8)

- Have all files been opened before use?
- Are the attributes of the input object consistent with the use of the file?
- Have all files been closed after use?
- Are there spelling or grammatical errors in any text printed or displayed?
- Are all I/O exceptions handled in a reasonable way?

9)

- Are the number, order, types, and values of parameters in every method call in agreement with the called method's declaration?
- Do the values in units agree (e.g., inches versus yards)?
- If an object or array is passed, does it get changed, and changed correctly by the called method?

REFERENCES

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

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[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] D. Vallespir and J. Herbert, "Effectiveness and cost of verification techniques: Preliminary conclusions on five techniques," in *Proceedings of the Mexican International Conference in Computer Science*, IEEE-Computer-Society, Ed., 2009.

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

[13] 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.

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

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

[16] D. Vallespir, F. Grazioli, and J. Herbert, "A framework to evaluate defect taxonomies," in *Proceedings of the XV Argentine Congress on Computer Science*, 2009.

[17] D. Vallespir, S. Moreno, C. Bogado, and J. Herbert, "Towards a framework to compare formal experiments that evaluate verification techniques," in *Proceedings of the Mexican International Conference in Computer Science*, 2009.
[18] F. Macdonald and J. Miller, "A comparison of tool-based and paper-based software inspection," *Empirical Software Engineering*, vol. 3, no. 3, pp. 233–253, 1998.
[19] N. Juristo and S. Vegas, "Functional testing, structural testing, and code reading: What fault type do they each detect?" *Empirical Methods and Studies in Software Engineering*, vol. 2765/2003, pp. 208–232, 2003.